UNIT- V

Trees: Introduction, Terminology, Representation of Trees, Binary Trees, The Abstract Data Type, Properties of Binary Tress, Binary Tree Representations, Binary Tree Traversal and Tree Iterators, Introduction, Inorder Traversal Preorder Traversal, Postorder Traversal, Thread Binary Trees, Threads, Inorder Traversal of a Threaded Binary Tree, Inserting a Node into a Threaded Binary Tree, Heaps, Priority Queues, Definition of a Max Heap, Insertion into a Max Heap, Deletion from a Max Heap, Binary Search Trees, Definition, Searching a Binary Search Tree, Insertion into a Binary Search Tree, Deletion from a Binary Search Tree, Height of Binary Search Tree.

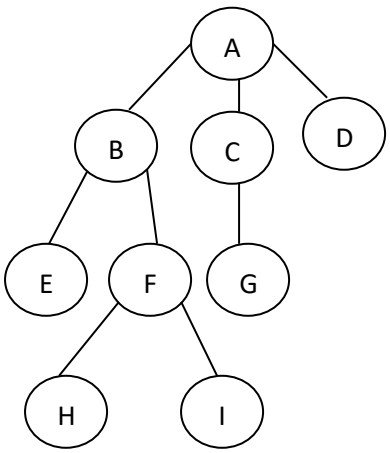
***

TREE

A **tree** is a finite set of one or more nodes such that:

- There is a specially designated node called the **root**,
- The remaining nodes are partitioned into n > 0 disjoint sets T_1 , T_2 , - - - T_n are called subtrees of the root.

Example:



BASIC TERMINOLOGY

Node: In tree structure each element is represented as a node. The concept of node is same as used in linked list. Node of a tree stores the actual data and links to the other node.

Edge: The connecting link between any two nodes is called as an edge. If a tree consists of 'N' number of nodes then the number of edges is N-1.

Path: The sequence of nodes and edges from one node to another node is called as a path between the two nodes.

Parent Node: Parent of a node is the immediate predecessor of a node.

Example: A is the parent of B, C and D
F is the parent of H and I etc.

Child Node: In a tree data structure, if the immediate predecessor of a node is the parent node then all immediate successor of a node are known as child nodes.

Example: B, C and D are the child nodes of A
H and I are the child nodes of F

Root Node: It is a specially designated node which has no parent node.

Example: A is the root node.

Leaf Node: The node which does not have any child is called a leaf node. Leaf nodes are also known as terminal nodes.

Example: E, H, I, G and D are leaf nodes.

Non-Leaf Node: The node which has children nodes is called a non-leaf node. Non-leaf nodes are also known as non-terminal nodes.

Example: A, B, C and F are non-leaf nodes.

Level: Level refers to the rank of the hierarchy. In general, root node is at level 0, its children are at level 1, their children are at level 2 and so on. Hence, if a node is at level 'K' then its children are at level 'K+1'.

Example:
Level 0 : A
Level 1 : B, C, D
Level 2 : E, F, G
Level 3 : H, I

Degree of a Node: The number of subtrees of a node is called its degree. Degree of a leaf node is 0.

Example:
Degree (A) : 3
Degree (C) : 1
Degree (I) : 0

Degree of a Tree: The degree of a tree is the maximum of the degree of nodes in the tree.

Example: Degree (Tree) : 3

Height of a Tree: Maximum number of nodes that is possible in a path starting from root node to a leaf node is called the height of a tree. Height of a tree is also known as **depth** of a tree.

Example: Height of the above tree : 4

Siblings: The nodes which have the same parent are called siblings.

Example: E and F are siblings.
B, C and D are siblings etc.

REPRESENTATION OF TREES

Trees can be represented in different ways such as:

A) List representation

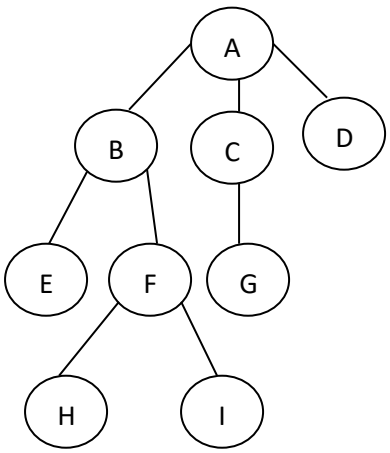
B) Left child – Right sibling representation

C) Degree – Two representation

A) List Representation

In list representation of a tree, the information in the root node comes first, followed by a list of subtrees of that node.

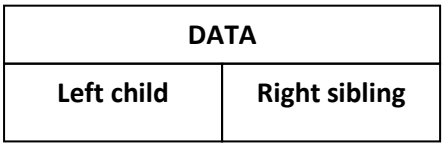
Example:



List representation: (A(B(E,F(H,I)),C(G),D)

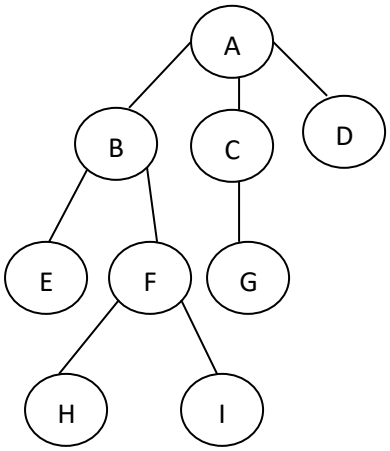
B) Left Child – Right Sibling Representation

In left child – right sibling representation, the node structure format can be shown as:

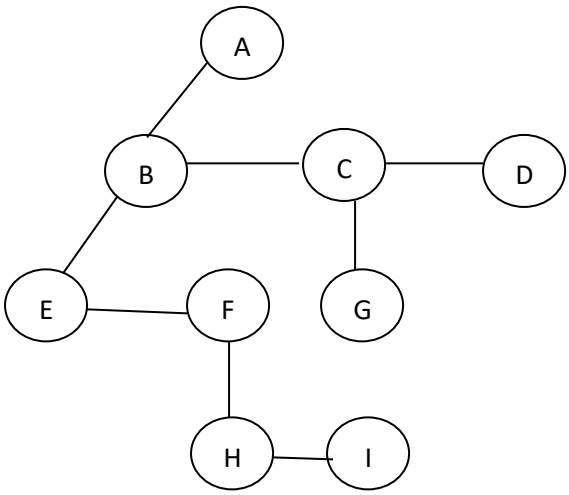


Here, every node has at most one leftmost child and at most one closest right sibling. The left child field of each node points to its leftmost child, and the right sibling field points to its closest right sibling. In this format, order of children in a tree is not important, any of the children of a node could be the leftmost child, and any of its siblings could be the closest right sibling.

Example:



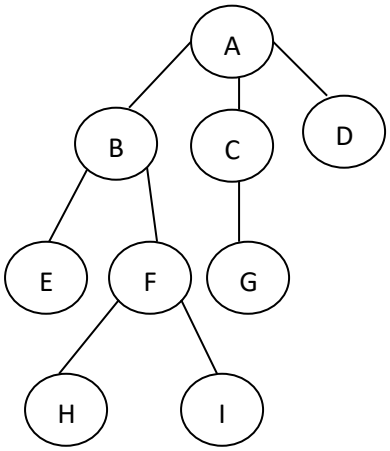
Left child – right sibling representation:



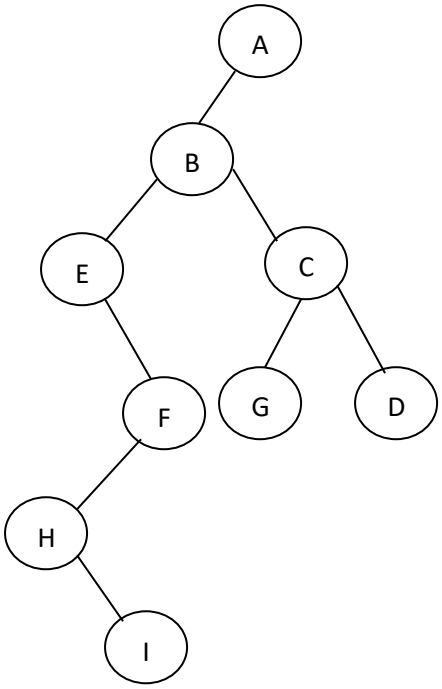
C) Degree – Two Representations

Degree two representation of a tree is obtained by rotating the right-sibling pointers in a left child – right sibling representation clockwise by 45 degrees. In this representation, every node has maximum degree as 2. So that it can refer as left and right children.

Example:



Degree – Two representation:

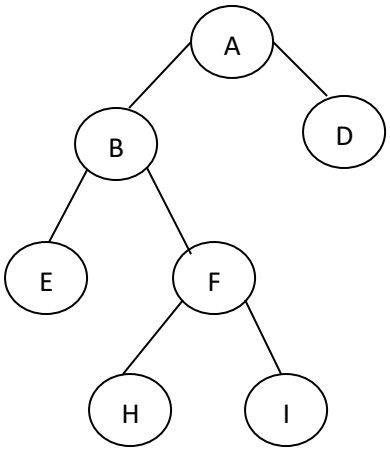


BINARY TREES

A binary tree T is a finite set of nodes, such that:

- T is empty (called empty binary tree), or
- T contains a specially designated node called the root of T, and remaining nodes of T form two disjoint binary trees T_1 and T_2 which are called left subtree and right subtree respectively.

Example:



Differences between tree and binary tree are:

- A tree can never be empty but binary tree may be empty.
- In binary tree, a node may have at most two children, whereas in a tree a node may have any number of children nodes.

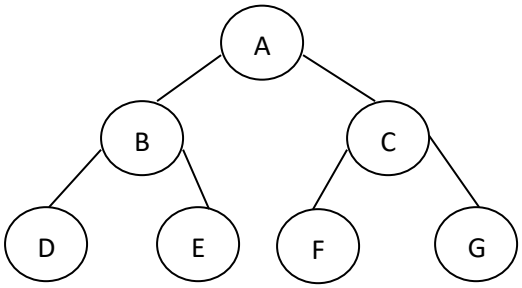
Note: Two special situations of binary tree are possible as:

a) Full binary tree
b) Complete binary tree

a) Full binary tree:

A binary tree is a full binary tree if it contains maximum possible number of nodes in all levels.

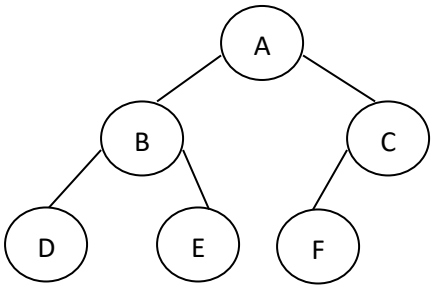
Example: Full binary tree of height 3.



b) Complete binary tree:

A binary tree is said to be a complete binary tree if all its levels except the last level have maximum number of possible nodes and all the nodes are appear as far left as possible.

Example: Complete binary tree of height 3.

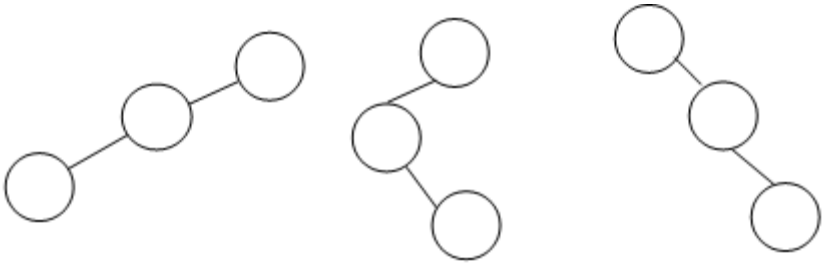


Properties of a Binary Tree

i. In any binary tree, maximum number of nodes on a level K is 2^K, where K≥0.

ii. Maximum number of nodes possible in a binary tree of height h is $2^h - 1$.

iii. Minimum number of nodes possible in a binary tree of height h is h.



These trees are called **skew binary** trees.

iv. For any non-empty binary tree, if n is the number of nodes and e is the number of edges, then n=e+1.

v. Height of a complete binary tree with n number of nodes is $\lceil \log_2(n+1) \rceil$.

REPRESENTATIONS OF A BINARY TREE

A binary tree can be represented in memory in two ways. Those are:

a) Sequential representation
b) Linked representation

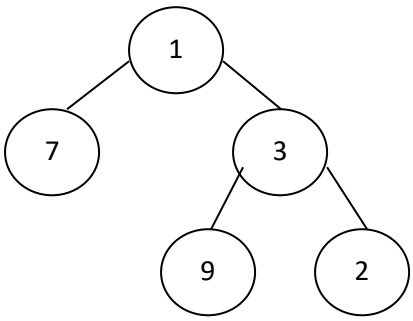
a) Sequential Representation

Sequential representation of a binary tree is static and uses array concept. In this representation, the nodes are stored level by level, starting from the zero level. Root node is stored in the first memory allocation.

Let X is an array used to store binary tree elements. Assume the root node is stored in index 1 of the array. Then the remaining nodes follow the properties as:

- The root node is stored in X[1] location.
- If a node N occupies X[K]th location, then its
  Left child is stored in **X[2*K]**th position and
  Right child is stored in **X[2*K+1]**th position.

Example: Consider the binary tree as:



For this, sequential representation is:

0	1	2	3	4	5	6	7
	12	72	34			99	21

Advantages:

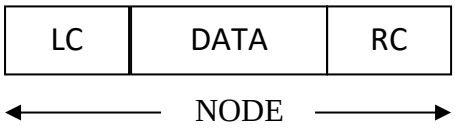
- Any node can be accessed from any other node by calculating its index value.
- Data stored easily without help of pointers.
- BASIC, FORTRAN programming languages doesn't support dynamic memory allocation concept. In such cases, array representation is an efficient way to store tree structures.

Disadvantages

- It allows only static representation. It is not possible to enhance the tree structure if the array size is limited.
- Other than full binary trees, majority of the array entries may be empty. Hence, the structure leads to space complexities problems.

b) Linked Representation

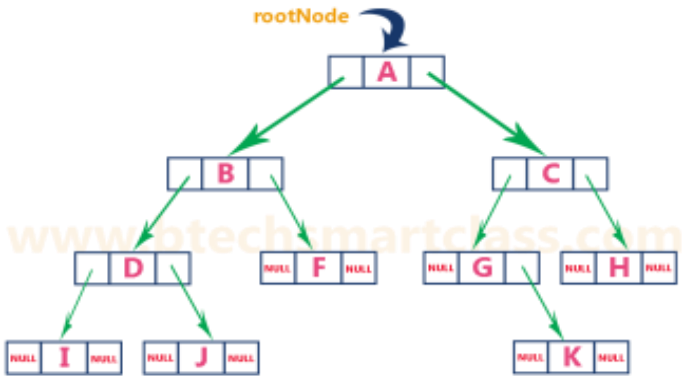
Linked representation of a binary tree is dynamic implementation and elements are stored in terms of nodes. Here, each node is divided into three parts as: LC, DATA and RC.



Where,

- ➔ DATA field contains information part of the element.
- ➔ RC is a link field that stores the address of the right child.
- ➔ LC is a link field that stores the address of the left child respectively.

Example:



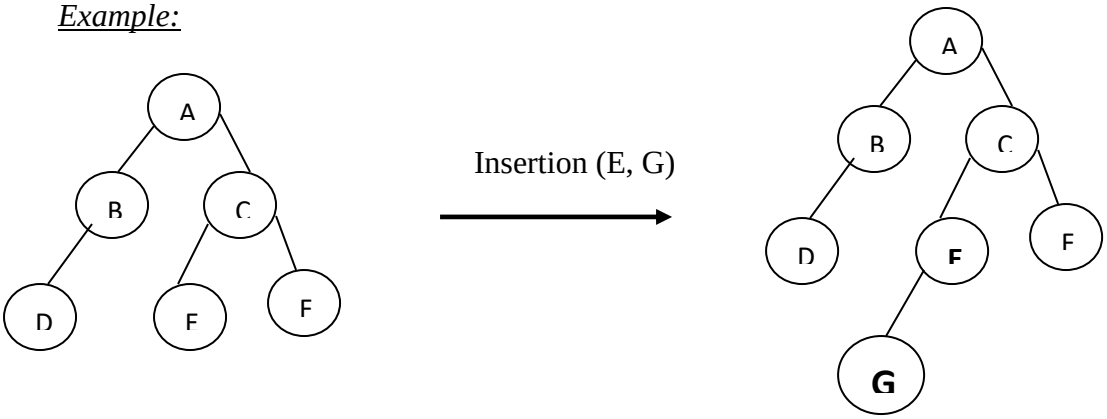
OPERATIONS ON BINARY TREE

Basic operations performed on a binary tree are: Insertion, Deletion, Traversal, Merging etc.

Insertion Operation

In a binary tree, a new node can be inserted at any position. For simplicity, assume the node is inserted as leaf node into the binary tree.

Example:

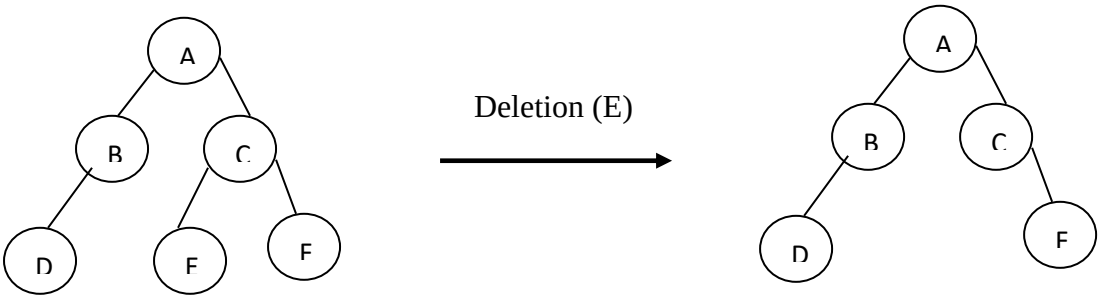


Insertion procedure is a two-step process. In this case, function is to be called by passing parent node and new node. Then search for the existence of parent node in the given binary tree and then insertion to be made. After placing new node establish a link in between parent node and child node as new node.

Deletion Operation

Deletion operation is used to delete any specified element from the binary tree. For simplicity, consider the way of deleting a leaf node from the binary tree.

Example:



In deletion process, first check whether the specified element is available in the binary tree or not. If it is available then deletion operation is possible.

Traversing Operation (Traversal Techniques)

Traversing operation refers to the process of visiting elements of the tree exactly once. Any tree can be traversed in three ways as:

a) Inorder traversal
b) Preorder traversal
c) Postorder traversal

a) Inorder Traversal

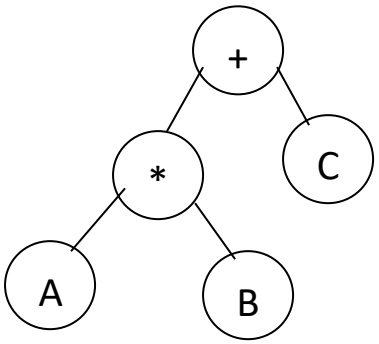
The inorder traversal of a binary tree can be stated as:

- Traverse the left subtree of the root node R in Inorder recursively.
- Visit the root node R.
- Traverse the right subtree of the root node R in Inorder recursively.

Algorithm Inorder (ROOT): This procedure is used to visit the binary tree in inorder recursively.

```
Step 1:   PTR ← ROOT
Step 2:   IF   PTR ≠ NULL   THEN
                Inorder (LC (PTR))
                Visit (DATA (PTR))
                Inorder (RC (PTR))
          ENDIF
Step 3:   STOP
```

Example:



Inorder Traversal Output:

A * B + C

b) Preorder Traversal

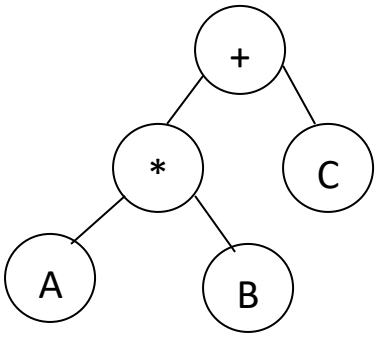
The preorder traversal of a binary tree can be stated as:

- Visit the root node R.
- Traverse the left subtree of the root node R in Preorder recursively.
- Traverse the right subtree of the root node R in Preorder recursively.

Algorithm Preorder (ROOT): This procedure is used to visit the binary tree in preorder recursively.

```
Step 1:   PTR ← ROOT
Step 2:   IF   PTR ≠ NULL   THEN
                Visit (DATA (PTR))
                Preorder (LC (PTR))
                Preorder (RC (PTR))
          ENDIF
Step 3:   STOP
```

Example:



Preorder Traversal Output:

+ * A B C

c) Postorder Traversal

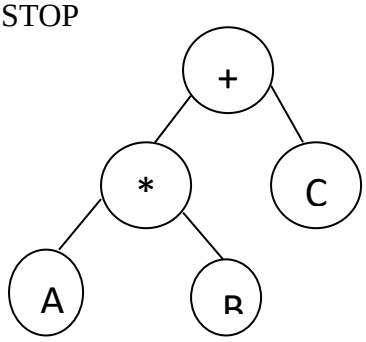
The postorder traversal of a binary tree can be stated as:

- Traverse the left subtree of the root node R in Postorder recursively.
- Traverse the right subtree of the root node R in Postorder recursively.
- Visit the root node R.

Algorithm Postorder (ROOT): This procedure is used to visit the binary tree in postorder recursively.

```
Step 1:   PTR ← ROOT
Step 2:   IF   PTR ≠ NULL   THEN
                Postorder (LC (PTR))
                Postorder (RC (PTR))
                Visit (DATA (PTR))
          ENDIF
Step 3:   STOP
```

Example:



Postorder Traversal Output:

A B * C +

Formation of a binary tree from its traversal techniques

A binary tree can be constructed with two traversal values in which one should be inorder traversal and other should be either preorder or postorder traversal. Basic principle for the formation can be stated as:

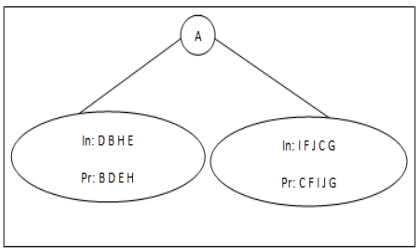
- If the preorder traversal is given, then the first node is the ROOT node. If the postorder traversal is given, then the last node is the ROOT node.
- Once the ROOT node is identified, all the nodes in the left subtree and right subtree of the ROOT node can be gathered.
- Same procedure is applied repeatedly on the left and right subtrees.

Example: Construct a binary tree with the following tree traversals.

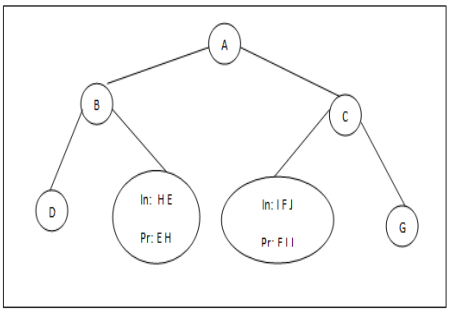
Inorder Traversal : D B H E A I F J C G

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

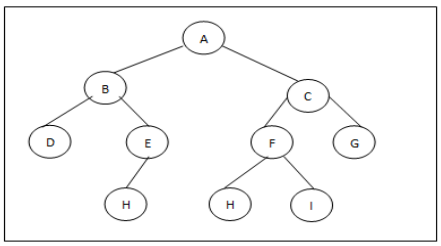
Step 1:



Step 2:



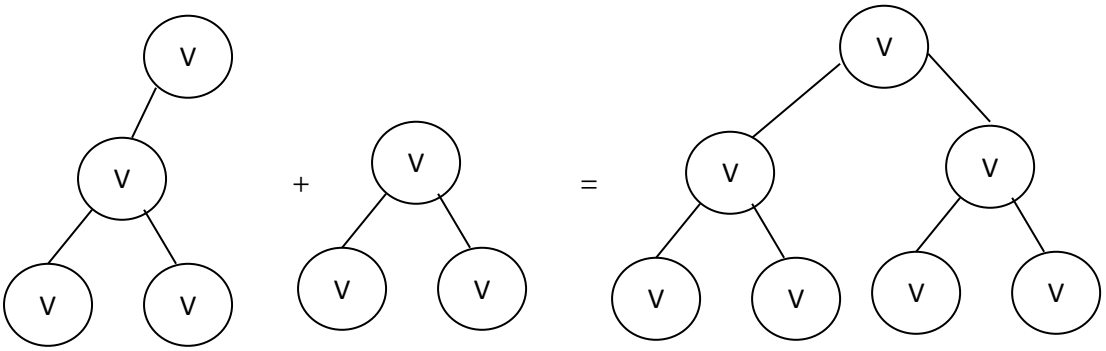
Step 3:



Merging Operation

Suppose T_1 and T_2 are two binary trees. Merging operation refers to the process of combining the entire tree T_2 (or T_1) as a sub tree of T_1 (or T_2). For this observe that in either (or both) tree there must be at least one null sub tree.

Example:

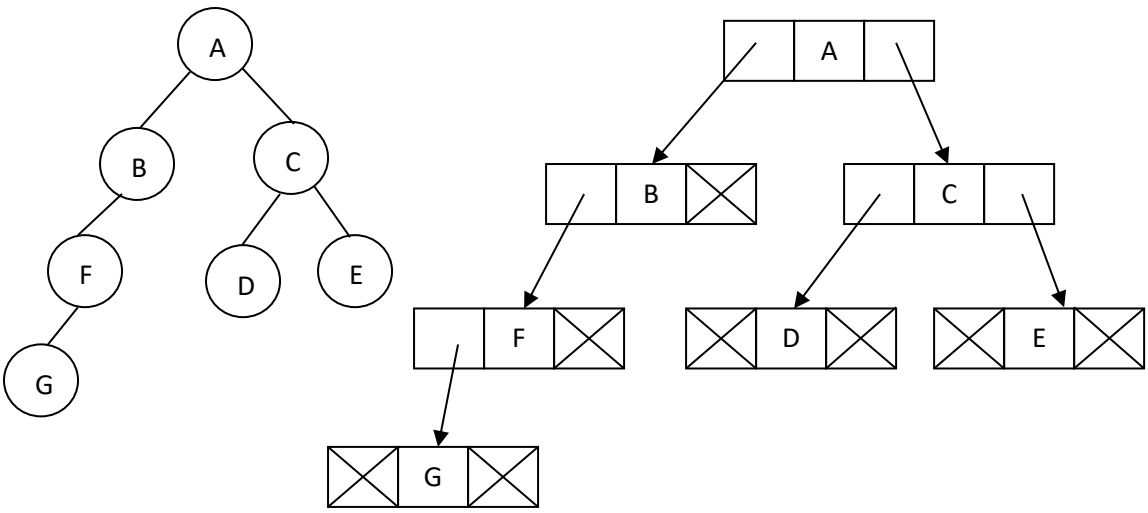


THREADED BINARY TREE

Threaded binary tree is a binary tree in which the null sub tree links are filled with thread links. Thread link is a pointer that points to other nodes of the binary tree. Here, nodes are pointed based on the rules as:

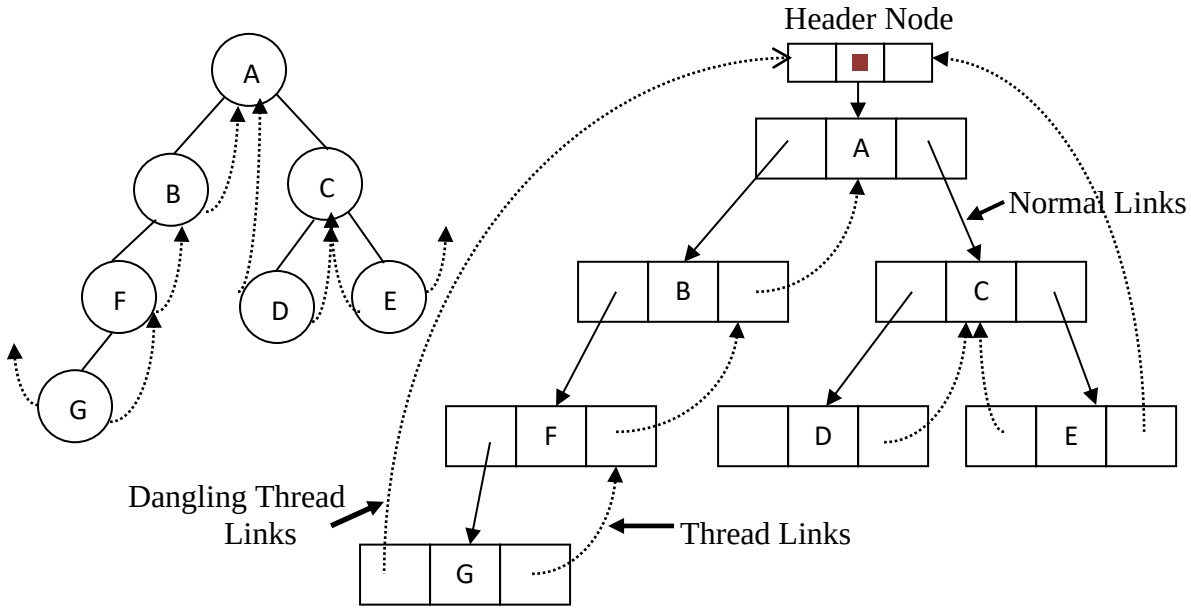
- ⇨ If RCHILD (NODE) is NULL then,
  The NULL pointer is replaced by a thread which points to the node that occurs after the node when the binary tree is traversed in inorder.
- ⇨ If LCHILD (NODE) is NULL then,
  The NULL pointer is replaced by a thread which points to the node immediately precedes node when the binary tree is traversed in inorder.

Example:



For this, threaded binary tree can be drawn as:

Inorder Traversal: G F B A D C E



PRIORITY QUEUES

A priority queue is a collection of zero or more elements. Each element has a priority. The basic operations on a priority queue are : insertion, deletion and display operations. These operations are performed based on the priority of the elements.

Priority queues can be classified into two types such as:

- Min priority queue
- Max priority queue

In **min priority queue,** minimum priority elements processed before the maximum priority elements.

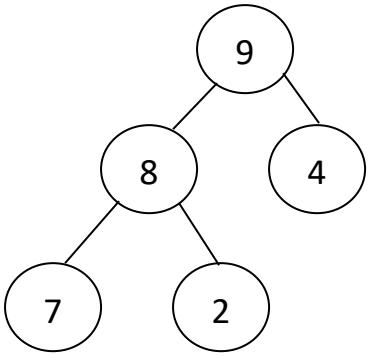
In **max priority queue,** maximum priority elements processed before the minimum priority elements.

HEAP TREE

Suppose H is a complete binary tree. Then it will be termed as a **heap tree / max heap** if it satisfies the property as:

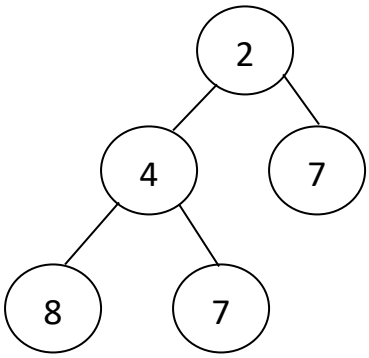
- ⇨ For each node N in H, the value at N is greater than or equal to the value of each of the children of N.

Example:



In addition to this a **min heap** is possible, where the value at N is less than or equal to the value of each of the children of N.

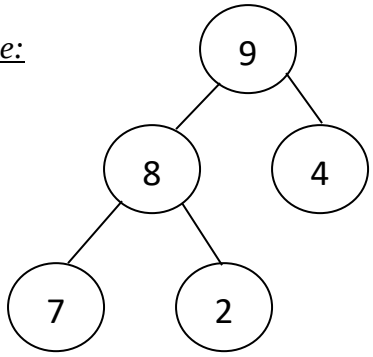
Example:



REPRESENTATION OF A HEAP TREE

Heap tree is a complete binary tree, it is better to represent with a single dimensional array. In this case, there is no wastage of memory space between two non-null entries.

Example:



Array Representation

1	2	3	4	5	6
95	84	48	76	23	

OPERATIONS ON HEAP TREE

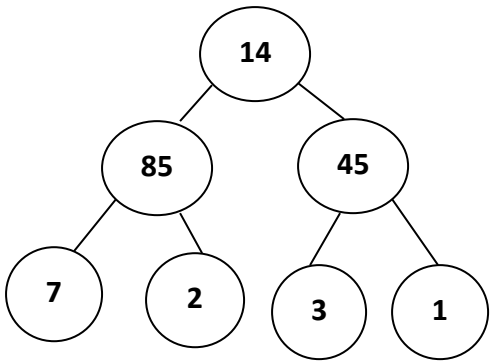
Basic operations on a heap tree are: insertion, deletion, merging etc.

Insertion into a heap tree: This operation is used to insert a new element into a heap tree. Let K is an array that stores n elements of a heap tree. Assume an element of information is given in the variable 'key' for insertion. Then insertion procedure works as:

- ⇨ First adjoin key value at the end of K so that still the tree is a complete binary tree, but not necessarily a heap.
- ⇨ Then raise the key value to its appropriate position so that finally it is a heap tree.

The basic principle is that first add the data element into the complete binary tree. Then compare it with the data in its parent node; if the value is greater than then the parent node then interchange these two values. This procedure will continue between every two nodes on the path from the newly inserted node to the root node till we get a parent whose value is greater than its child or we reached at the root node.

Example: Insert a value 124 into the existing heap tree



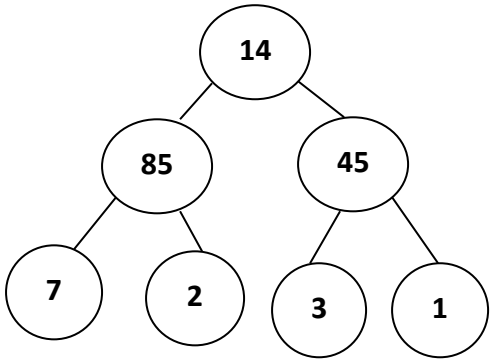
Algorithm InHeap (Key): Let K is an array that stores a heap tree with 'n' elements. This procedure is used to store a new element Key into the heap tree.

```
Step 1:   n ← n+1
Step 2:   K[n] ← Key
Step 3:   i ← n
          p ← i/2
Step 4:   Repeat WHILE  p > 0   AND  K[p] < K[i]
                Temp ← K[i]
                K[i] ← K[p]
                K[p] ← Temp
                i ← p
                p ← p/2
          End Repeat
Step 5:   Return
```

Deletion of a node from heap tree: This operation is used to delete an element from the existing heap tree. Deletion procedure works as:

- Assign the root node into a temporary variable as key.
- Replace the root node location by the last node in the heap tree. Then re-heap the tree as:
  - Let newly modified root node be the current node. Compare its value with its two children node values. Let X is the child whose value is the largest value. Then interchange the value of X with the value of the current node.
  - Make X as the current node.
  - Continue re-heap process, if the current node is not an empty node.

Example: Delete an element from the following heap tree



Algorithm Deletion (): This procedure removes root element of the heap tree and rearranges the elements into heap tree format.

```
Step 1:   IF   n = 0  THEN
                  WRITE   ' HEAP TREE EMPTY'
                  RETURN
          ENDIF
Step 2:   Temp ← K[1]
          K[1] ← K[n]
Step 3:   n ← n-1
          i ← 1
          Flag ← FALSE
Step 4:   Repeat WHILE   Flag = FALSE    AND    i < n
                  lchild ← 2 * i
                  rchild ← 2 * i + 1
                  IF   lchild ≤ n    THEN
                        X ← K[lchild]
```

```
                    ELSE
                        X ← -999
                    ENDIF
                    IF  rchild ≤ n   THEN
                        Y ← K[rchild]
                    ELSE
                        Y ← -999
                    ENDIF
                    IF  K[i] > X  AND  K[i] > Y  THEN
                        Flag ← TRUE
                    ELSEIF  X > Y  AND  K[i] < X  THEN
                        SWAP(K[i], K[lchild])
                        i ← lchild
                    ELSEIF  Y > X  AND  K[i] < Y  THEN
                        SWAP(K[i], K[rchild])
                        i ← rchild
                    ENDIF
                ENDREPEAT
Step 5:         RETURN
```

APPLICATION OF HEAP TREES

Two important applications of heap trees are: Heap sort and Priority queue implementations.

Heap Sort:

Heap sort is also known as **Tree** sort. The procedure of heap sort is as follows:

Step 1: Build a heap tree with the given set of data elements.
Step 2: a) Delete the root node from the heap and place it in last location.
b) Rebuild the heap with the remaining elements.
Step 3: Continue Step-2 until the heap tree is empty.

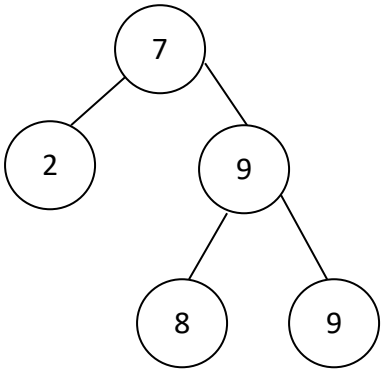
Example: Sort the elements **33, 14, 65, 2 and 99** using heap sort.

BINARY SEARCH TREE

A binary search tree is a binary tree. It may be empty. If it is not empty then it satisfies the following properties as:

- Each node has exactly one value and the values in the tree are distinct.
- The values in the left subtree are smaller than the value in the root node.
- The values in the right subtree are larger than the value in the root node.
- The left and right subtrees are also binary search trees.

Example:



The abstract data type of a binary search tree can be shown as:

AbstractDataType BSTree
{

Instances: A collection of elements such that all keys are distinct; values in the left subtree are smaller than the value in the root node; values in the right subtree are larger than the value in the root node.

Operations:

Insertion(K):	Inserts a new element K into the binary search tree
Deletion(K):	Removes an element from the binary search tree
Search(K):	Search for an element K in the existing binary search tree
Ascend():	Prints all keys of the binary search tree in sorted order

}

Operations on Binary search tree

Basic operations on a binary search tree are searching, insertion, deletion, merging, traversing etc.

i) Search Operation:

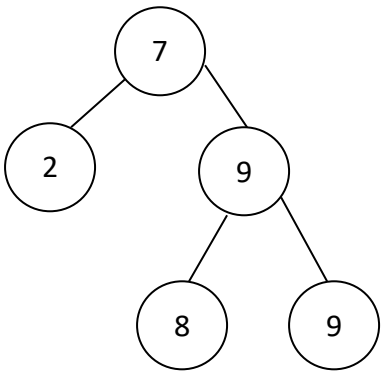
Suppose T is a binary search tree and is represented using linked structure. Assume search element is given in the variable Key. Then search procedure works as:

- Search procedure starts from the root node R.
- If Key value is less than the root node R, then proceed to its left child; If Key value is greater than the root node R, then proceed to its right child.
- The above process will be continued till the Key is found or NULL pointer occurred. If search procedure reached to NULL pointer, it refers to Key value not found.

Algorithm Search (Key): This procedure is used to search whether the element Key is exist in the binary search tree or not.

```
Step 1:   PTR ← ROOT
          Flag ← FALSE
Step 2:   REPEAT  WHILE  PTR ≠ NULL  AND  Flag = FALSE
                IF  Key < DATA(PTR)  THEN
                        PTR ← LCHILD(PTR)
                ELSEIF  Key > DATA(PTR)  THEN
                        PTR ← RCHILD(PTR)
                ELSE
                        Flag ← TRUE
                ENDIF
          ENDREPEAT
Step 3:   IF  Flag = TRUE  THEN
                WRITE  'KEY VALUE FOUND'
          ELSE
                WRITE  'KEY VALUE NOT FOUND'
          ENDIF
Step 4:   STOP
```

Example:



Search (92): Key Found

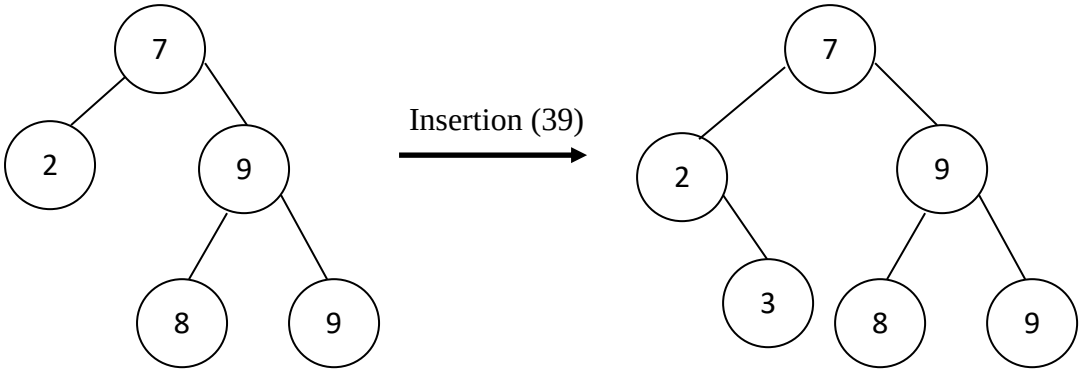
Search (77): Key Not Found

ii) Inserting an element into a binary search tree:

Suppose T is a binary search tree. Inserting element is given in the variable Key. Then insertion procedure works as:

- To insert a node with the given value, the tree T is to be searched from the ROOT node.
- If the same Key value is found at any stage, then print a message as 'Key already exists – Insertion not possible'; otherwise, a new node is inserted at the dead node where the search procedure terminates.

Example:



Algorithm Insertion (Key): This procedure is used to insert a new node with the data as Key into the binary search tree.

```
Step 1:   PTR ← ROOT
          Flag ← FALSE
Step 2:   Repeat  WHILE  PTR ≠ NULL  AND  Flag = FALSE
                IF  Key < DATA (PTR)  THEN
                        P1 ← PTR
                        PTR ← LCHILD(PTR)
                ELSEIF  Key > DATA (PTR)  THEN
                        P1 ← PTR
                        PTR ← RCHILD(PTR)
                ELSE
                        WRITE  'KEY ALREADY EXISTS'
                        EXIT
                ENDIF
          EndRepeat

Step 3:   IF  PTR = NULL  THEN
```

```
                        Allocate memory for a NEW node
                        DATA(NEW) ← Key
                        LCHILD(NEW) ← NULL
                        RCHILD(NEW) ← NULLL
                        IF   DATA (P1) < Key   THEN
                                RCHILD(P1) ← NEW
                        ELSE
                                LCHILD(P1) ← NEW
                        ENDIF
                ENDIF
Step 4:         STOP
```

iii) Deleing an element from a binary search tree:

Suppose T is a binary search tree and an element of information is given in the variable Key to delete from T.

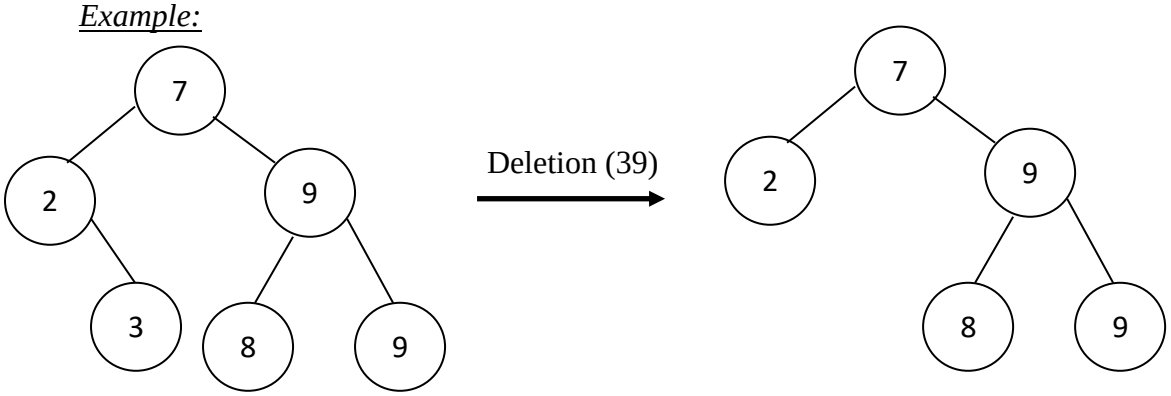
Consider N be the node which contains the information element as Key. Assume PARENT (N) denotes the parent node of N and SUCC (N) denotes the inorder successor of the node N. Then the deletion of the node N depends on any one of the three following cases based on its children nodes as:

Case 1: N has no children.
Case 2: N has only one child node.
Case 3: N has two children nodes.

Case 1: N has no children:

In this case, N is deleted from the tree T by simply setting the pointer of N in the parent node PARENT (N) by NULL pointer.

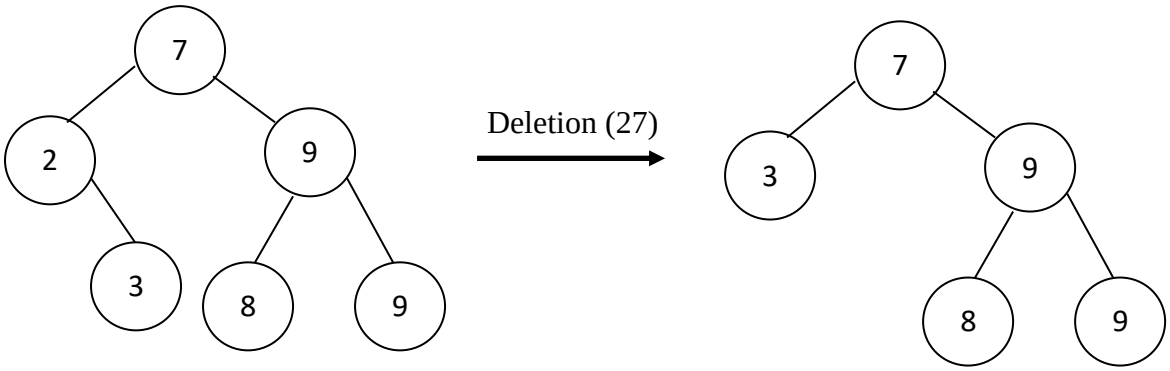
Example:



Case 2: N has only one child node:

In this case, N is deleted from the tree T by simply setting the pointer of N in the parent node PARENT (N) by the child pointer of the deleted Node N.

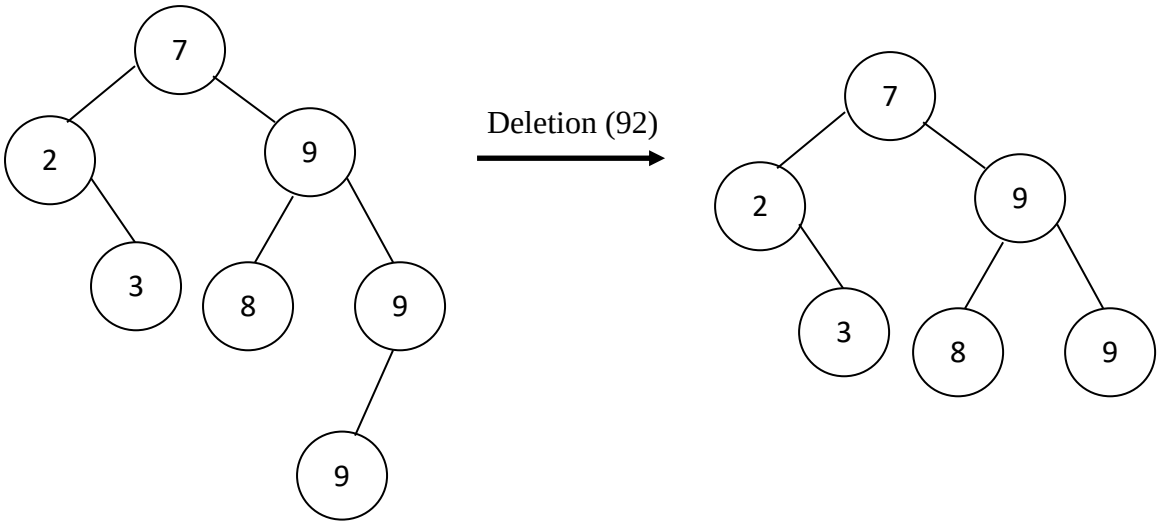
Example:



Case 3: N has two children:

In this case, N is deleted from the tree T by first deleting succ(N) from T (by using case 1 or case 2) and then replace the data content in node N by the data content of node succ(N).

Example: Inorder Traversal : 27 39 75 83 ~~92~~ 94 99



Algorithm deletion(Key): This function is used to delete a specified key element from the binary search tree.

```
Step 1:     PTR ← ROOT
            Flag ← FALSE
Step 2:     Repeat  WHILE  PTR ≠ NULL  AND  Flag = FALSE
                    IF  Key < DATA (PTR)  THEN
                            Parent ← PTR
                            PTR ← LCHILD(PTR)
                    ELSEIF  Key > DATA (PTR)  THEN
                            Parent ← PTR
                            PTR ← RCHILD(PTR)
                    ELSE
                            Flag ← TRUE
                    ENDIF
            EndRepeat
Step 3:     IF  Flag = FLASE  THEN
                    WRITE  ' KEY DOES NOT EXIST'
                    EXIT
            ENDIF
Step 4:     IF  LCHILD (PTR) = NULL  AND  RCHILD (PTR) = NULL  THEN
                    IF  LCHILD(Parent) = PTR  THEN
                            LCHILD(Parent) = NULL
                    ELSE
                            RCHILD(Parent) = NULL
                    ENDIF
            ELSEIF  LCHILD(PTR) ≠ NULL  AND  RCHILD(PTR) ≠ NULL  THEN
                    PTR1 ← SUCC(PTR)
                    K ← DATA(PTR1)
                    deletion(K)
                    DATA(PTR) ← K
            ELSE
                    IF  LCHILD(Parent) = PTR  THEN
                            IF  LCHILD(PTR) = NULL  THEN
                                LCHILD(Parent) ← RCHILD(PTR)
                            ELSE
                                LCHILD(Parent) ← LCHILD(PTR)
                            ENDIF
                    ELSEIF  RCHILD(Parent) = PTR  THEN
                            IF  LCHILD(PTR) = NULL  THEN
                                RCHILD(Parent) ← RCHILD(PTR)
                            ELSE
                                RCHILD(Parent) ← LCHILD(PTR)
                            ENDIF
                    ENDIF
            ENDIF
Step 5:     STOP
```

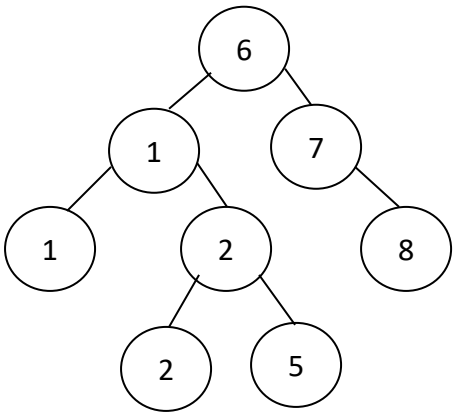
Algorithm SUCC(PTR): This procedure is used to locate inorder successor of PTR node.

```
Step 1:     P ← RCHILD(PTR)
Step 2:     IF  P ≠ NULL  THEN
                REPEAT  WHILE  LCHILD(P) ≠ NULL
                        P ← LCHILD(P)
                ENDREPEAT
            ENDIF
Step 3:     RETURN P
```

iv) Traversals on binary search tree:

A binary search tree can be traversed in three ways such as: Inorder traversal, Preorder traversal and Postorder traversal techniques.

Example:



Inorder Traveral : 15 19 25 28 57 65 74 88

Preorder Traversal : 65 19 15 28 25 57 74 88

Postorder Traversal : 15 25 57 28 19 88 74 65

Note:

- Inorder traversal on a binary search tree will give the sorted order of data in ascending order. To sort the given set of data, a binary search tree can be built and then inorder traversal can be applied. This method of sorting is known as **binary sort** and such binary search tree can be treated as **binary sorted tree.**

THE END