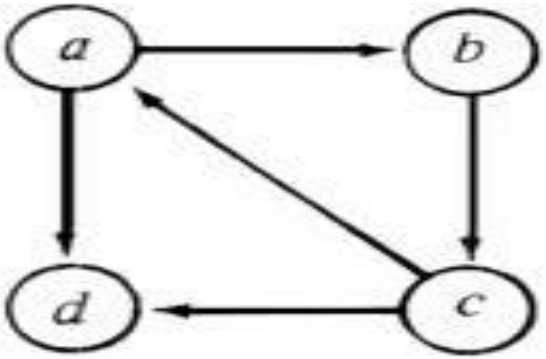
TREE

NON-LINEAR DATA STRUCTURE

Graphs



Trees

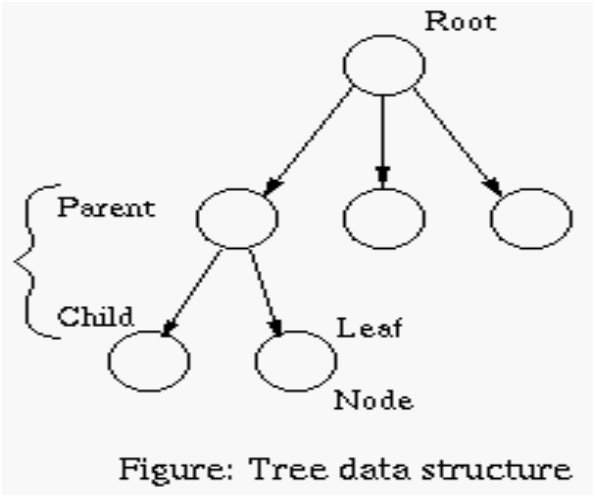


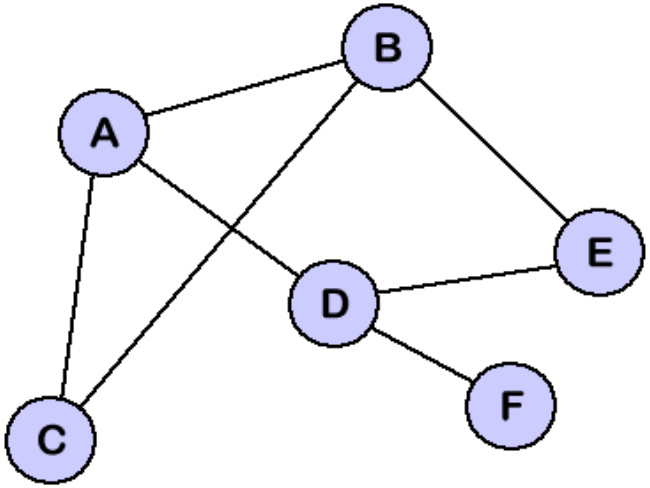
Figure: Tree data structure

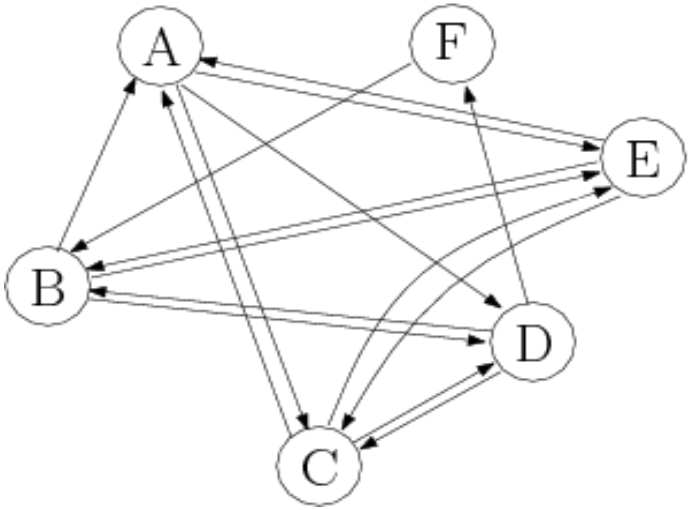
GRAPH

A graph G consists of a nonempty set V called the set of nodes, a set E called the set of edges, and a mapping from E to pairs of elements of V.

$$G : E \rightarrow V \times V$$

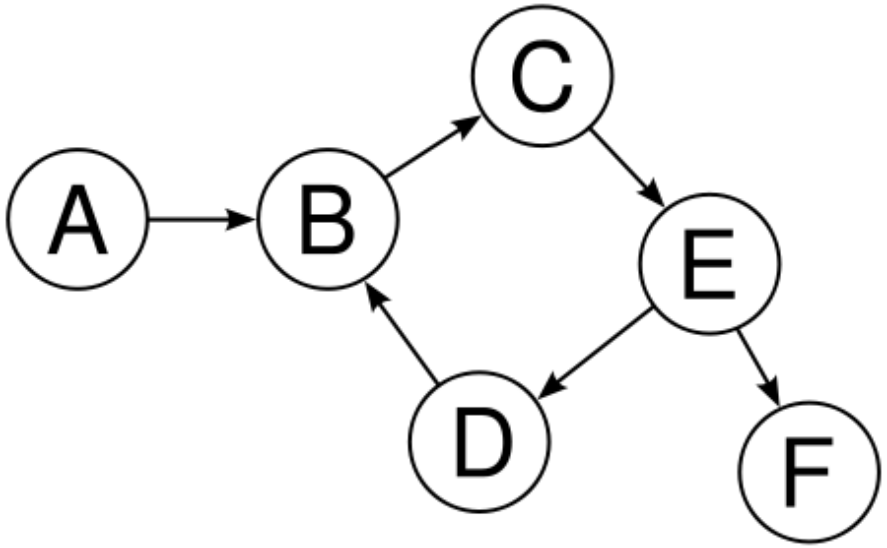
It is denoted by G = (V, E).





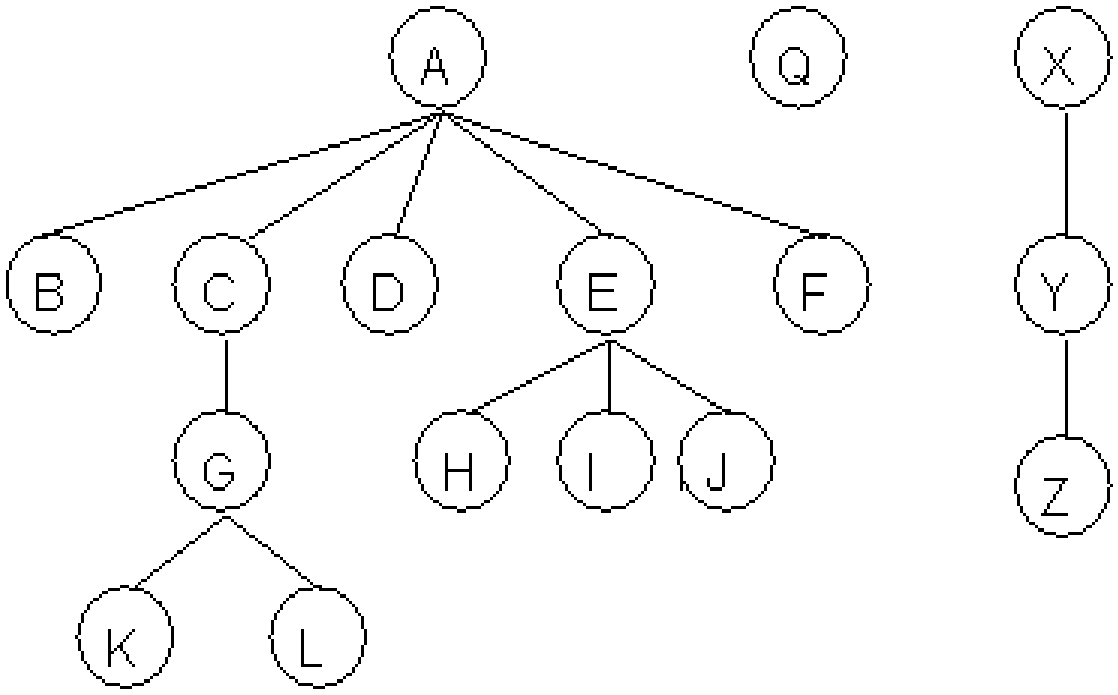
GRAPH

- **Edge:** Two nodes are connected by edge. It is the connection between two nodes.
- **Cycle:** A path in a graph that starts and ends at the same node is called a cycle, and the graph containing a cycle is called cyclic.
- **Degree of a Node:** The degree of a node is the no of children of that node.
- **In-degree:** Number of edges terminated at given node is called in-degree of node.
- **Out-degree:** Number of edges emerging from given node is called out-degree of node.



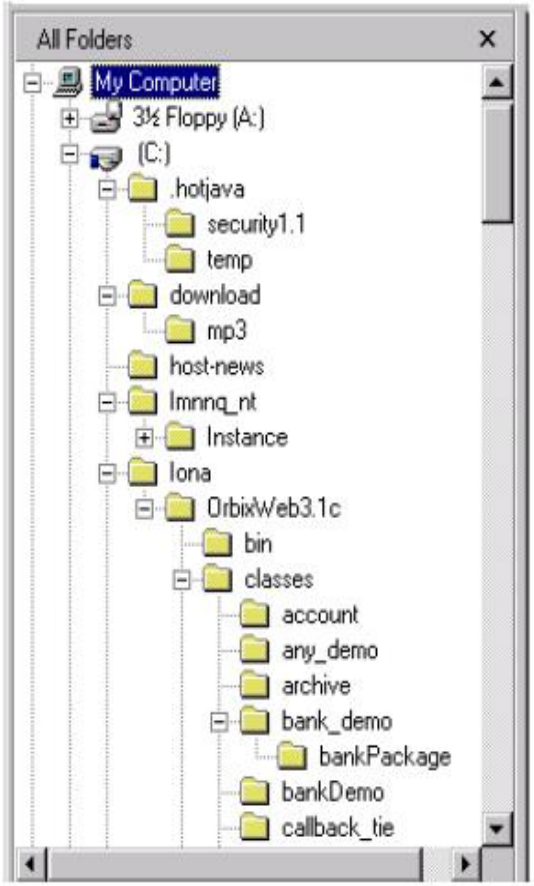
TREE

Definition: A directed tree is an acyclic graph which has one node called its root, with in-degree 0, while all other nodes have in-degree 1.



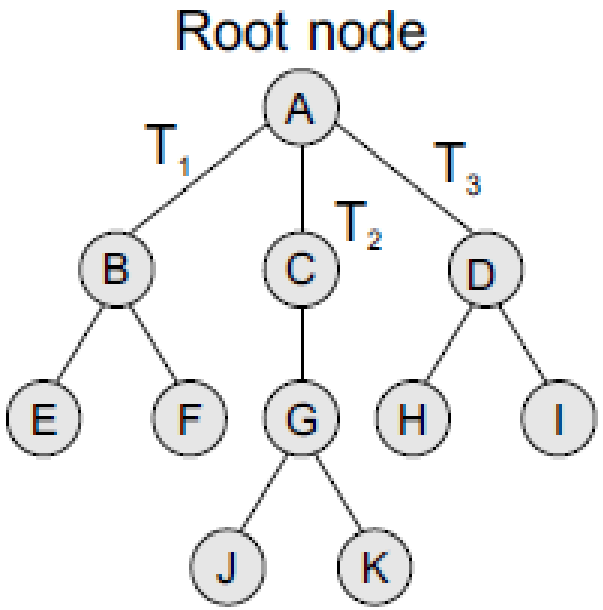
Where have you seen a tree structure before?

- Examples:
  1. Directory Tree
  2. Family Tree
  3. Company Organization Chart
  4. Table of Contents



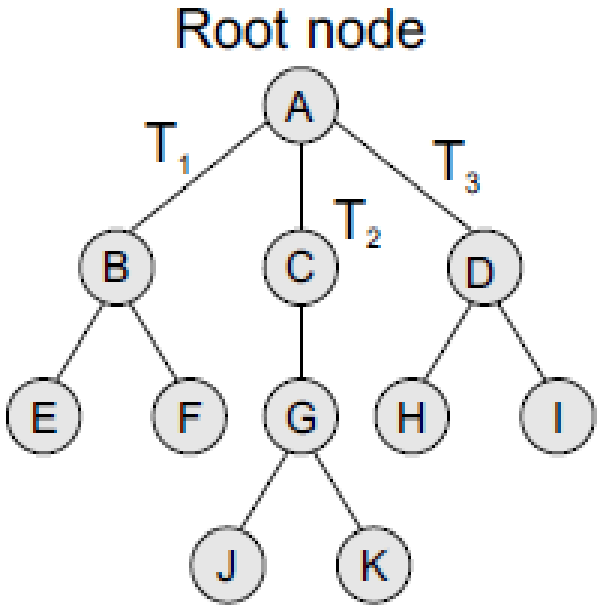
TREE TERMINOLOGIES

- **Root node:** The root node R is the topmost node in the tree. If R = NULL, then it means the tree is empty. Root node has in-degree 0.
- **Sub-tree:** : A node and all its descendent ignoring the node's parent, this is itself a tree. T1, T2, and T3 are called the sub-trees of R.
- **Path:** A sequence of consecutive edges is called a path.
- **Ancestor node:** An ancestor of a node is any predecessor node on the path from root to that node.
- **Descendant node:** A descendant node is any successor node on any path from the node to a leaf node.



TREE TERMINOLOGIES

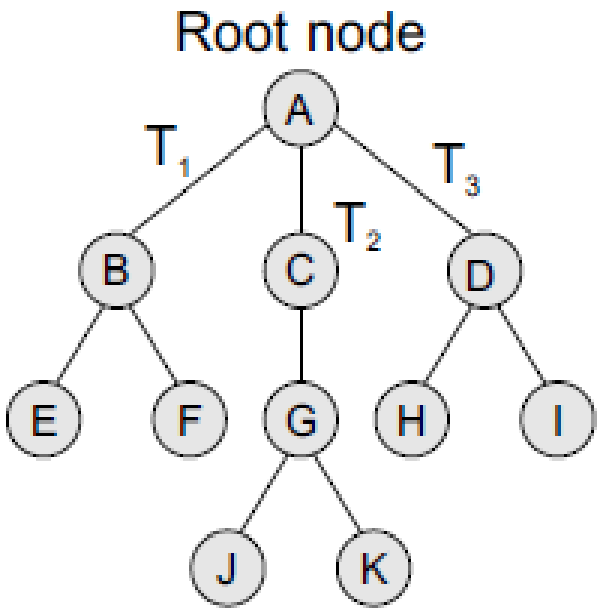
- **Parent**: Single node that directly precedes a node. All nodes have 1 parent except root (has 0)
- **Child**: One or more nodes that directly follow a node.
- **Level number:** Every node in the tree is assigned a level number in such a way that the root node is at level 0, children of the root node are at level number 1. Thus, every node is at one level higher than its parent. So, all child nodes have a level number given by parent's level number + 1.
- **Leaf node:** A node that has no children is called the leaf node or the terminal node. Leaf node has out degree 0.



TREE TERMINOLOGIES

- **Degree:** Degree of a node is equal to the number of children that a node has. The degree of a leaf node is zero.
- **Siblings**: All nodes that are at the same level and share the same parent are called siblings (brothers).
- **Depth:** The depth of a node N is given as the length of the path from the root R to the node N. The depth of the root node is zero.
- **Height:** It is the total number of nodes on the path from the root node to the deepest node in the tree.

  A tree with only a root node has a height of 1.
- **Forest:** A forest is a set of $n \geq 0$ disjoint trees.



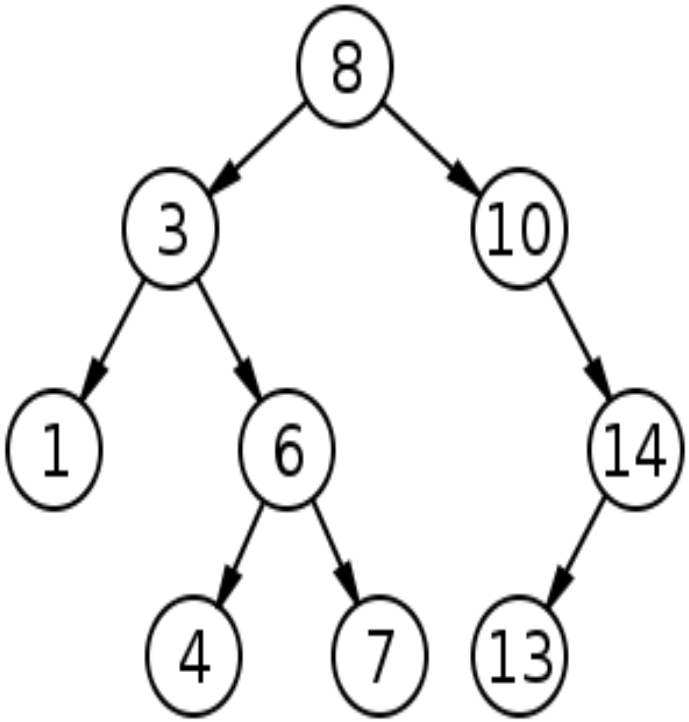
TREE

- A tree is recursively defined as a set of one or more nodes where one node is designated as the root of the tree and all the remaining nodes can be partitioned into non-empty sets each of which is a sub-tree of the root.

- Some Types of Trees are following:
  1. General trees.
  2. Forests.
  3. Binary trees.
  4. Binary search trees.
  5. B trees.
  6. B+ trees.
  7. AVL Tree.

BINARY TREE

Definition: The tree in which out-degree of every node is less than or equal to 2 (at most 2) is called a binary tree.



BINARY TREE

Recursive Definition:

A binary tree is a finite set of elements that is either empty or is partitioned into 3 disjoint subsets:

i. Root of tree

ii. Left sub-tree which is itself binary tree

iii. Right sub-tree which is itself binary tree

TYPES OF BINARY TREE

FULL BINARY TREE/STRICT BINARY TREE: A binary tree in which all nodes except leaf node have either 0 or 2 children is called a full **binary tree**.

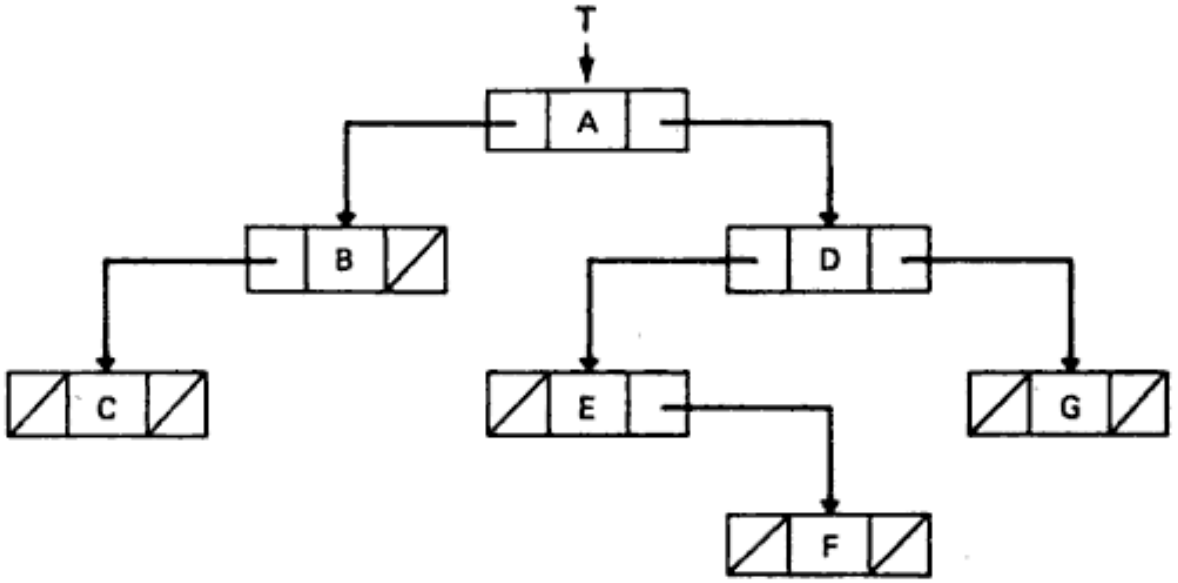
COMPLETE BINARY TREE: Every level except the last level is completely filled and all the nodes are left justified.

PERFECT BINARY TREE: Every node except the leaf nodes have two children and every level (last level too) is completely filled.

BINARY TREE REPRESENTATION

Typical node of a binary tree implementation:

```
struct node
{
        int info;
        struct node *left;
        struct node *right;
};
```



Linked representation of a binary tree.

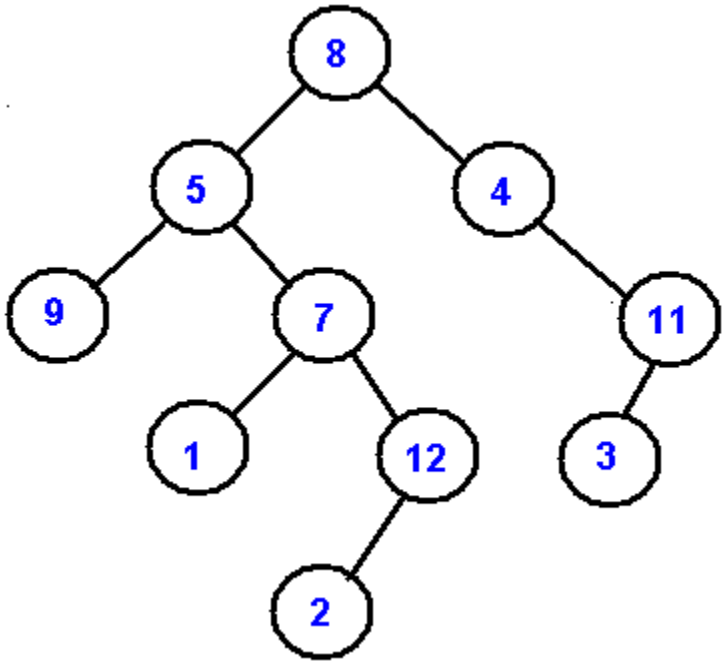
BINARY TREE TRAVERSALS

- The most common operation on a binary tree is the traversal of its nodes.
- To traverse a tree means to visit or pass through each node for some kind of processing, viz. printing, searching, updating, insertion, deletion, etc…
- Three types of traversals:
  1. Preorder Traversal – (VLR)
  2. Inorder Traversal – (LVR)
  3. Postorder Traversal – (LRV)

PREORDER TRAVERSAL – (VLR)

Steps:

1. Visit the root Vertex.
2. Traverse the Left sub-tree in preorder.
3. Traverse the Right sub-tree in preorder.

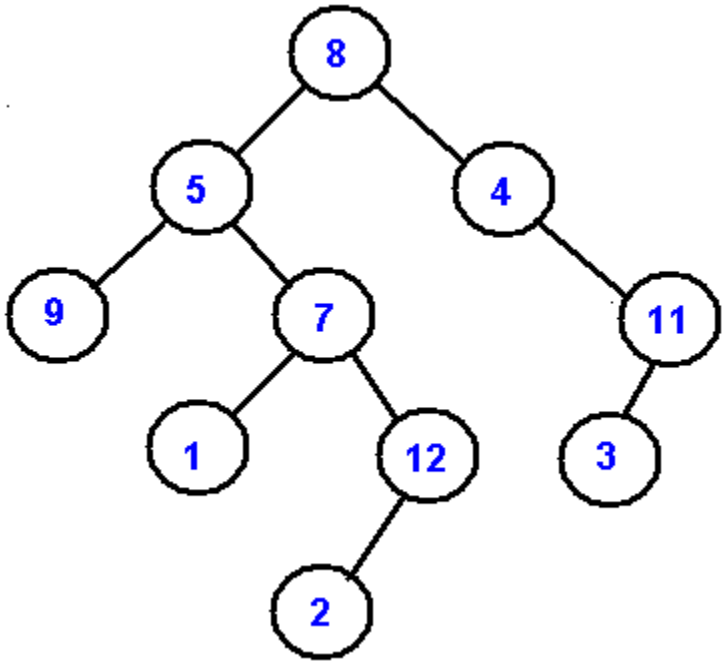


Preorder Traversal: 8, 5, 9, 7, 1, 12, 2, 4, 11, 3

INORDER TRAVERSAL – (LVR)

Steps:

1. Traverse the Left sub-tree in inorder.
2. Visit the root Vertex.
3. Traverse the Right sub-tree in inorder.

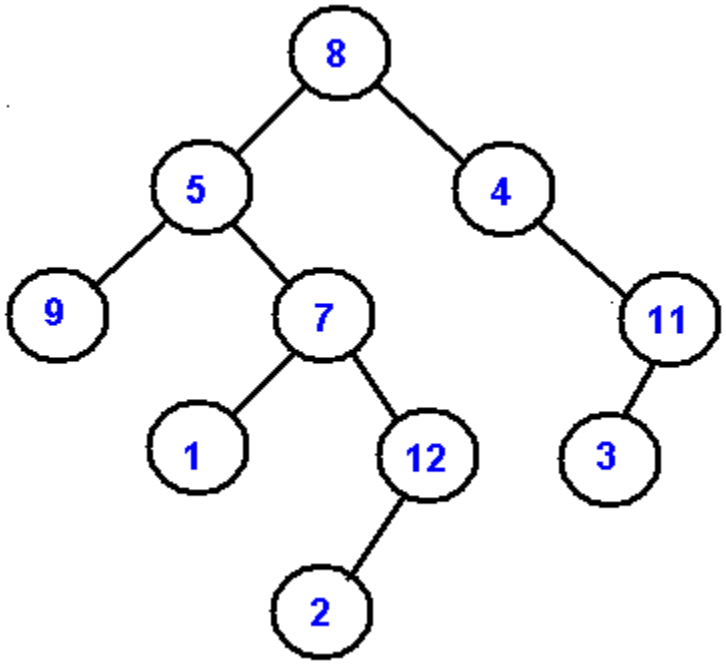


Inorder Traversal: 9, 5, 1, 7, 2, 12, 8, 4, 3, 11

POSTORDER TRAVERSAL – (LRV)

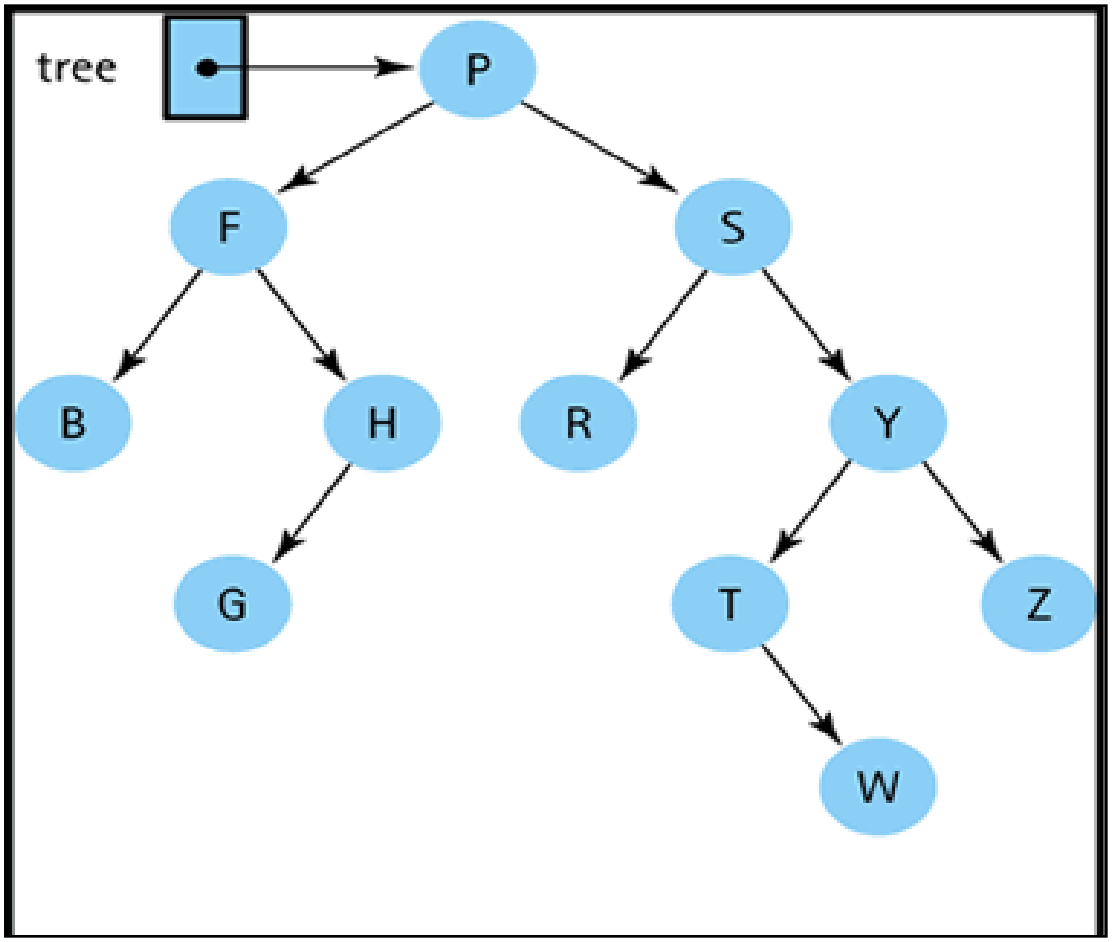
Steps:

1. Traverse the Left sub-tree in postorder.
2. Traverse the Right sub-tree in postorder.
3. Visit the root Vertex.



Postorder Traversal: 9, 1, 2, 12, 7, 5, 3, 11, 4, 8

TRAVERSAL EXAMPLE



PREORDER TRAVERSAL ALGORITHM

```
Step 1: Repeat Steps 2 to 4 while TREE != NULL
Step 2:              Write TREE -> DATA
Step 3:              PREORDER(TREE -> LEFT)
Step 4:              PREORDER(TREE -> RIGHT)
       [END OF LOOP]
Step 5: END
```

INORDER TRAVERSAL ALGORITHM

```
Step 1: Repeat Steps 2 to 4 while TREE != NULL
Step 2:              INORDER(TREE -> LEFT)
Step 3:              Write TREE -> DATA
Step 4:              INORDER(TREE -> RIGHT)
       [END OF LOOP]
Step 5: END
```

POSTORDER TRAVERSAL ALGORITHM

```
Step 1: Repeat Steps 2 to 4 while TREE != NULL
Step 2:              POSTORDER(TREE -> LEFT)
Step 3:              POSTORDER(TREE -> RIGHT)
Step 4:              Write TREE -> DATA
        [END OF LOOP]
Step 5: END
```

BINARY SEARCH TREE

Definition:

A BST is a binary tree, which is either empty or in which each node contains a key that satisfies the following conditions:

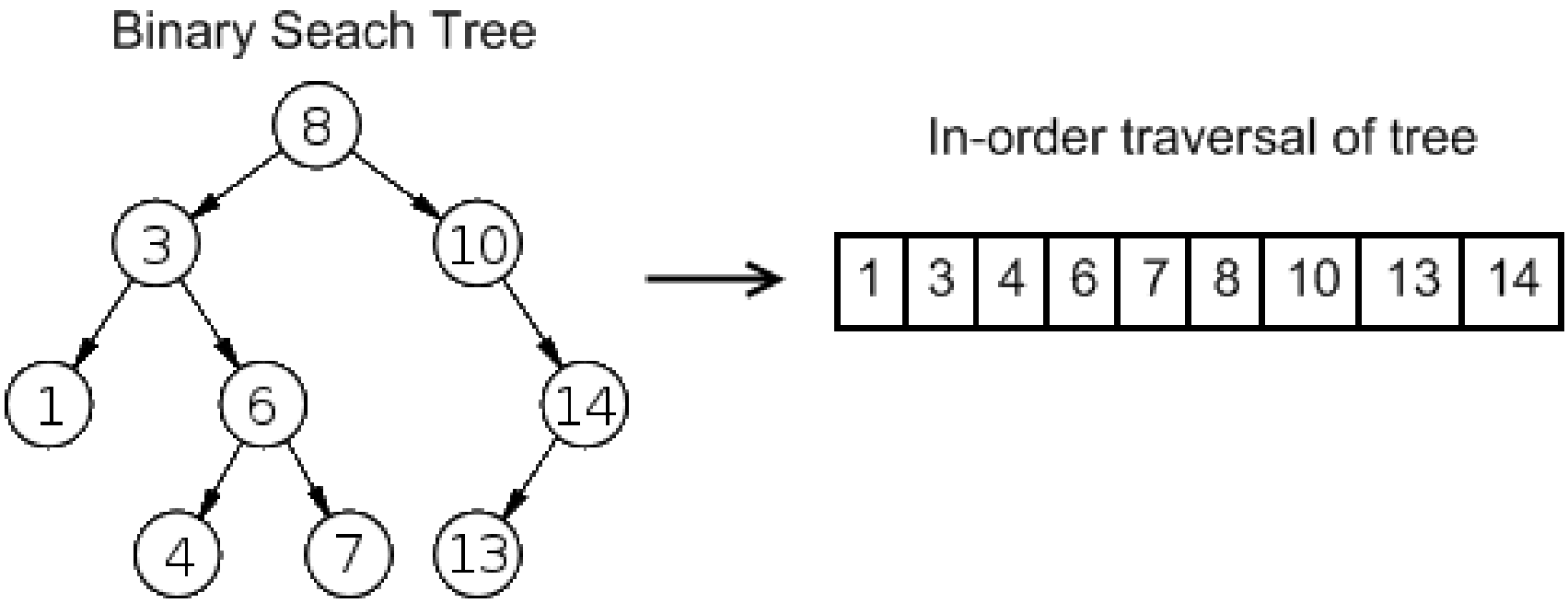
i. All keys are distinct.

ii. For every node X, the value of all the keys in its left sub-tree are smaller than X.

iii. For every node X, the value of all the keys in its right sub-tree are greater than X.

BINARY SEARCH TREE

BINARY SEARCH TREE



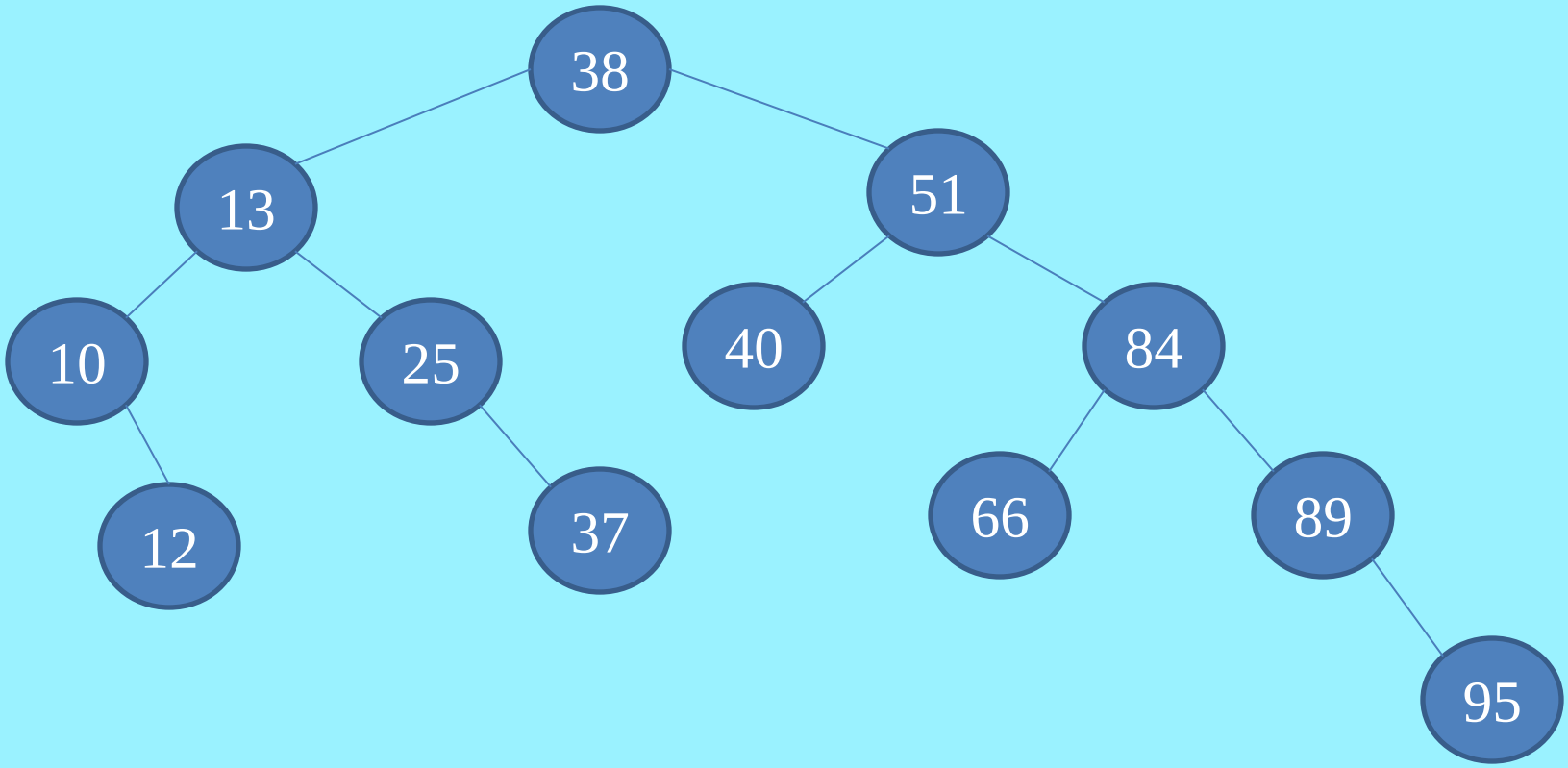
OPERATIONS ON BINARY SEARCH TREE

1. Insert
2. Delete
3. Search

INSERT OPERATION ON BINARY SEARCH TREE

Binary Search Tree Example

Tree resulting from the following insertions: 38, 13, 51, 10, 12, 40, 84, 25, 89, 37, 66, 95



Algorithm to insert a given value in a BST

```
Insert (TREE, VAL)

Step 1: IF TREE = NULL
            Allocate memory for TREE
            SET TREE -> DATA = VAL
            SET TREE -> LEFT = TREE -> RIGHT = NULL
        ELSE
            IF VAL < TREE -> DATA
                  Insert(TREE -> LEFT, VAL)
            ELSE
                  Insert(TREE -> RIGHT, VAL)
            [END OF IF]
        [END OF IF]
Step 2: END
```

DELETE OPERATION ON BINARY SEARCH TREE

- When the delete operation is performed on a BST, its properties should be maintained.

- There are 3 possibilities for node to be deleted:

  Case I: leaf node

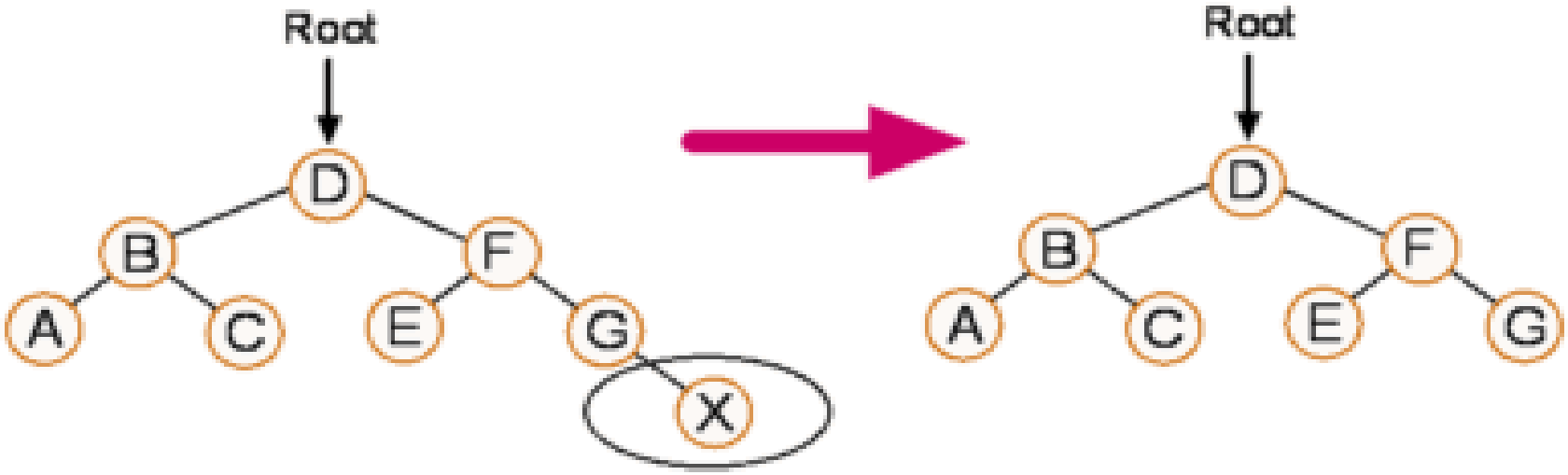
  Case II: node with one child

  Case III: node with two children

DELETE OPERATION ON BINARY SEARCH TREE

Case I: Leaf Node

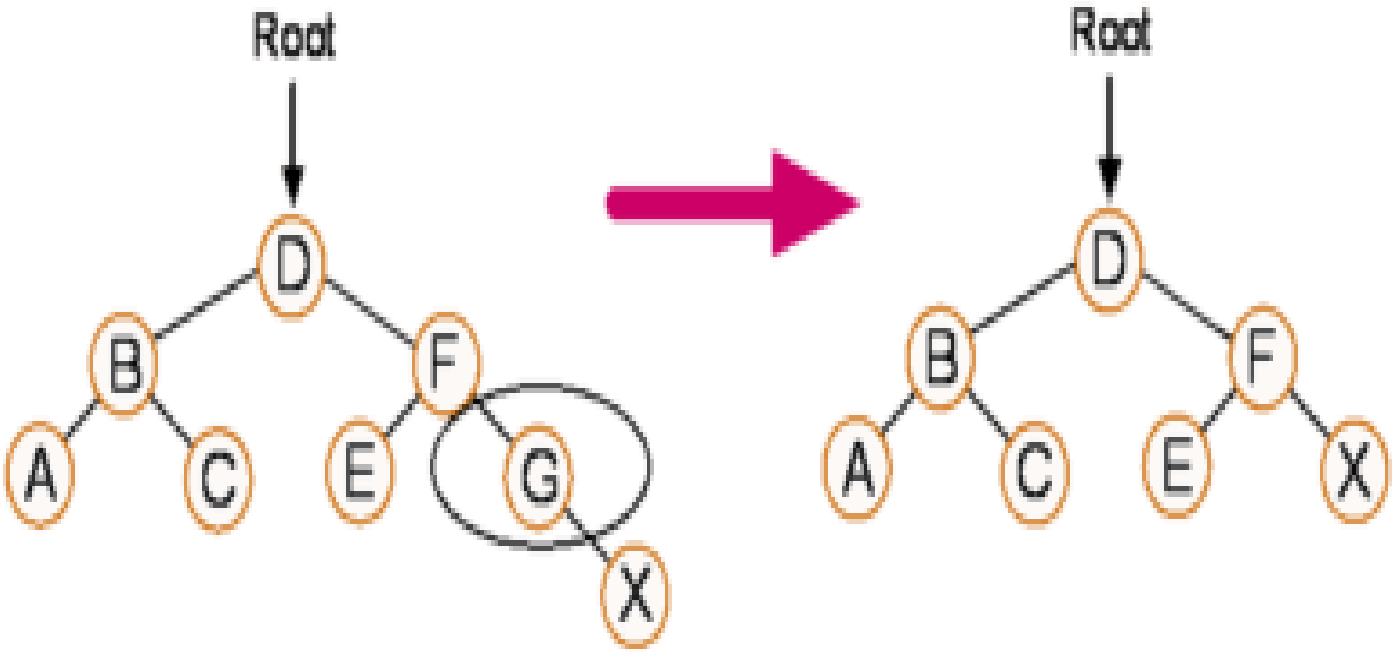
Simply set the parent node pointer of the node to be deleted to NULL.



DELETE OPERATION ON BINARY SEARCH TREE

Case II: node with one child

Set the parent node pointer of node to be deleted to point to its child.

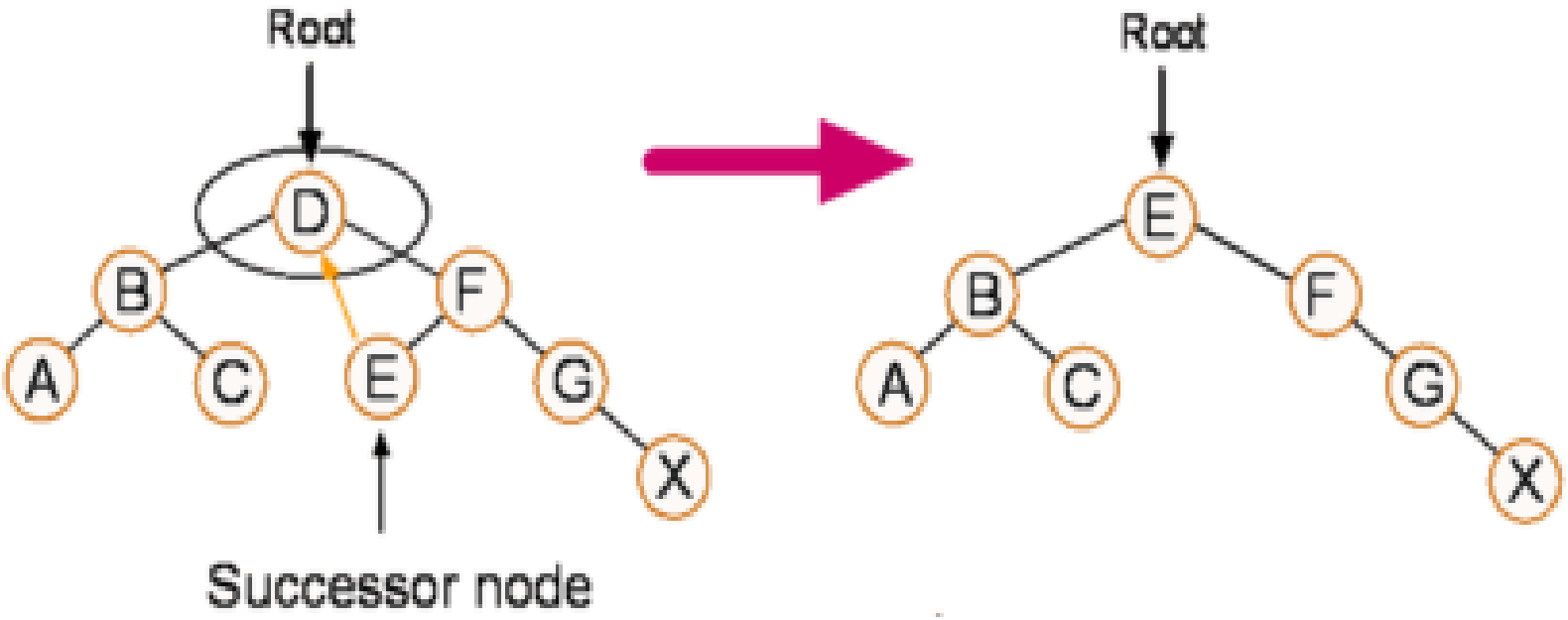


DELETE OPERATION ON BINARY SEARCH TREE

Case III: node with two children

Steps: (assume node ‘D’ is to be deleted)

1) Replace D with the smallest node ‘E’ in its right sub-trsee.
2) Replace ‘E’ with its right son, if any.



Algorithm to delete a node from a BST

```
Delete (TREE, VAL)

Step 1: IF TREE = NULL
          Write "VAL not found in the tree"
        ELSE IF VAL < TREE -> DATA
          Delete(TREE->LEFT, VAL)
        ELSE IF VAL > TREE -> DATA
          Delete(TREE -> RIGHT, VAL)
        ELSE IF TREE -> LEFT AND TREE -> RIGHT
          SET TEMP = findLargestNode(TREE -> LEFT)
          SET TREE -> DATA = TEMP -> DATA
          Delete(TREE -> LEFT, TEMP -> DATA)
        ELSE
          SET TEMP = TREE
          IF TREE -> LEFT = NULL AND TREE -> RIGHT = NULL
              SET TREE = NULL
          ELSE IF TREE -> LEFT != NULL
               SET TREE = TREE -> LEFT
          ELSE
               SET TREE = TREE -> RIGHT
          [END OF IF]
          FREE TEMP
        [END OF IF]
Step 2: END
```

SEARCH OPERATION ON BST

```
searchElement (TREE, VAL)

Step 1: IF TREE -> DATA = VAL OR TREE = NULL
          Return TREE
        ELSE
         IF VAL < TREE -> DATA
          Return searchElement(TREE -> LEFT, VAL)
         ELSE
          Return searchElement(TREE -> RIGHT, VAL)
         [END OF IF]
        [END OF IF]
Step 2: END
```

APPLICATION OF TREE

Some of the applications of a tree are:

- Binary Search Trees for Efficient Searching
- Dictionary
- Disk Storage
- Expression Tree