SCHOOL OF COMPUTER SCIENCE AND IT

Master of Computer Applications

Data Structures

Module 3

Non-Linear Data Structures-I

Module No.3
Non-Linear DS

Syllabus Contents - Nonlinear data structures-I

- Introduction to Tree Data Structure and Basic Terminology.
- Implementation of Tree ADT. Types of Trees.
- Binary Tree ADT, Enumeration of Binary Trees, Tree Traversal
- Expression Trees
- Applications of trees
- Binary Search Tree ADT.
- Construct BST from given preorder traversal,
- Binary Tree to Binary Search Tree Conversion.

Tree ADT

Why Trees required in DS?

- Trees are a natural **way to store data** that has a **hierarchical structure**
- Tree data structures allow for **quick searching and sorting**
- **Easy insertion and deletion**
- **No linear time of access** i.e O(N)

Contd..

- A tree is a collection of nodes
  - **The collection can be empty**
  - **(recursive definition) If not empty,**
  - a tree consists of a **distinguished node ‘r’ (the *root*),**
  - and **zero or more nonempty *subtrees* T_1, T_2, ..., T_k,** each of whose roots are connected by a directed *edge* from r

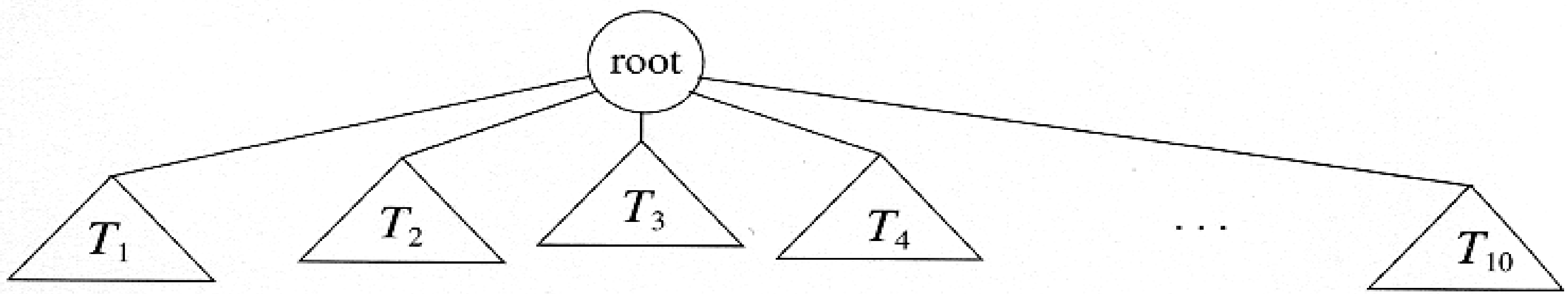
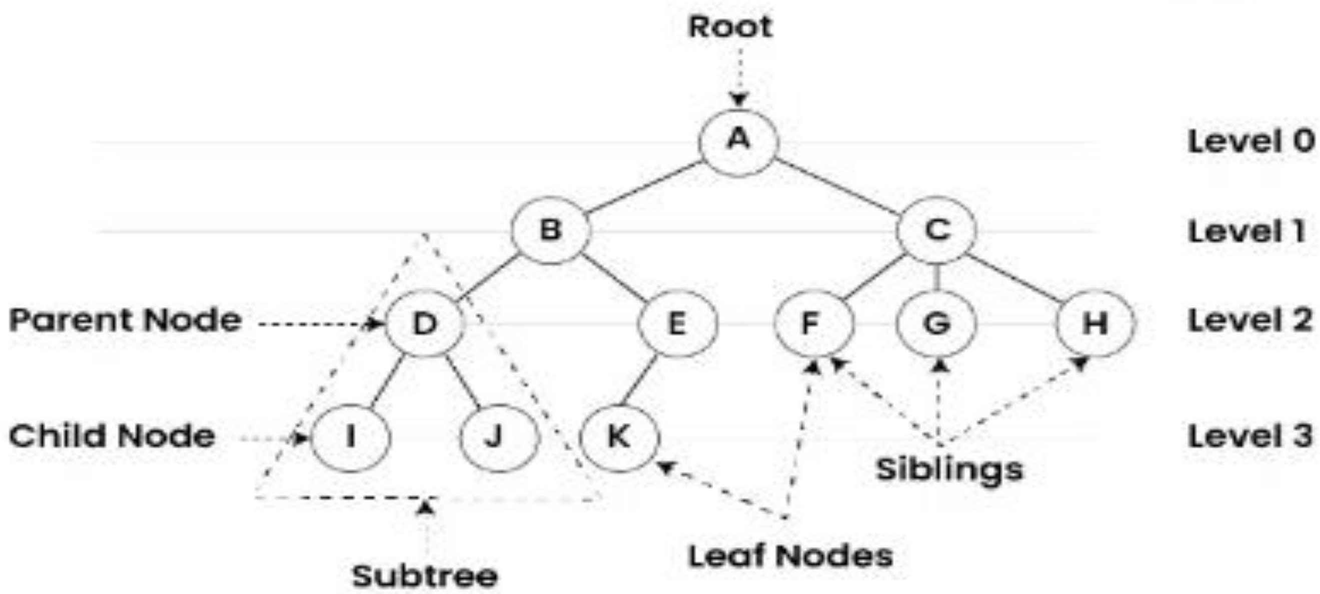


Figure 4.1 Generic tree

Terminologies



Root

- In a tree data structure, the root is the first node of the tree. And it is the initial node of the tree in data structures.
- In the tree data structure, there must be only one root node.

Terminologies

Parent

In the tree in data structures,

- node that **is the predecessor** of **any node** is known **as a parent node**

Child

The node, **a descendant of any node**, is known **as child nodes** in data structures.

In a tree, **any number of parent nodes** can **have any number of child nodes**.

Siblings

In trees in the data structure, **nodes that belong to the same parent** are called siblings.

Terminologies

Leaf

- **the node with no child**, is known as a leaf node.
- In trees, leaf nodes are also called **external nodes or terminal nodes.**

Internal nodes

- Trees in the data structure have at **least one child node known as internal nodes.**
- In trees, **nodes other than leaf nodes** are **internal nodes.**

Terminologies

Degree

- the **total number of children of a node** is called the **degree of the node**.
- The **highest degree of a node** among all the nodes in a tree is called the **Degree of Tree.**

Terminologies

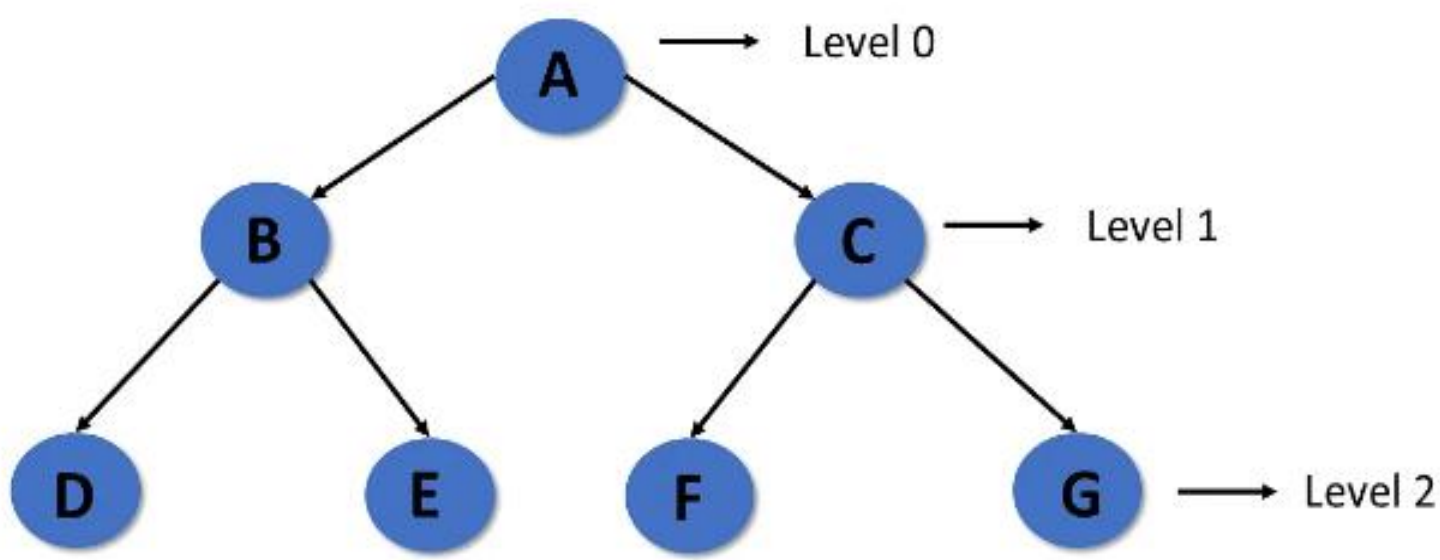
Level

In tree data structures,

- the **root node is said to be at level 0**, and

-the **root node's children are at level 1**,

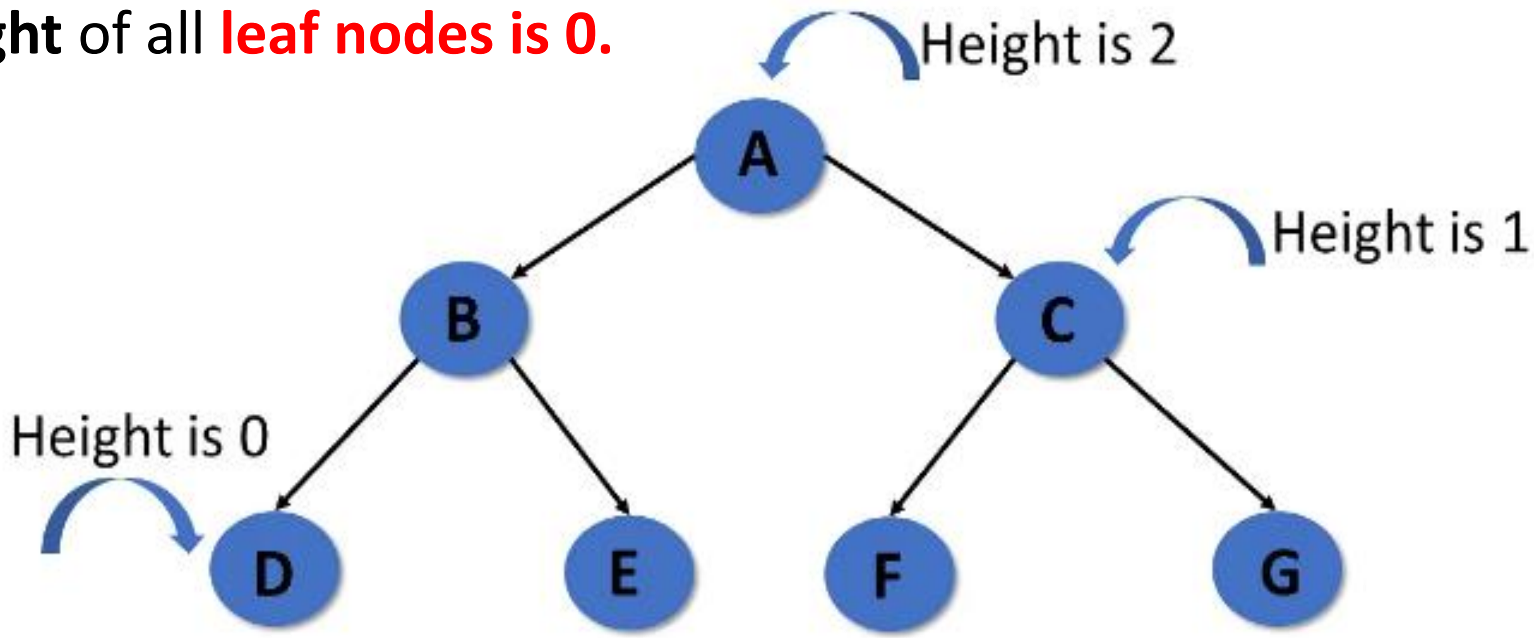
- and the children of that node at level 1 will be level 2, and so on.



Terminologies

Height

- the **no. of edges from the leaf node to the particular node in the longest path** is known as the height of that node.
- the **height of the root node** is called "**Height of Tree**".
- The **tree height** of all **leaf nodes is 0.**

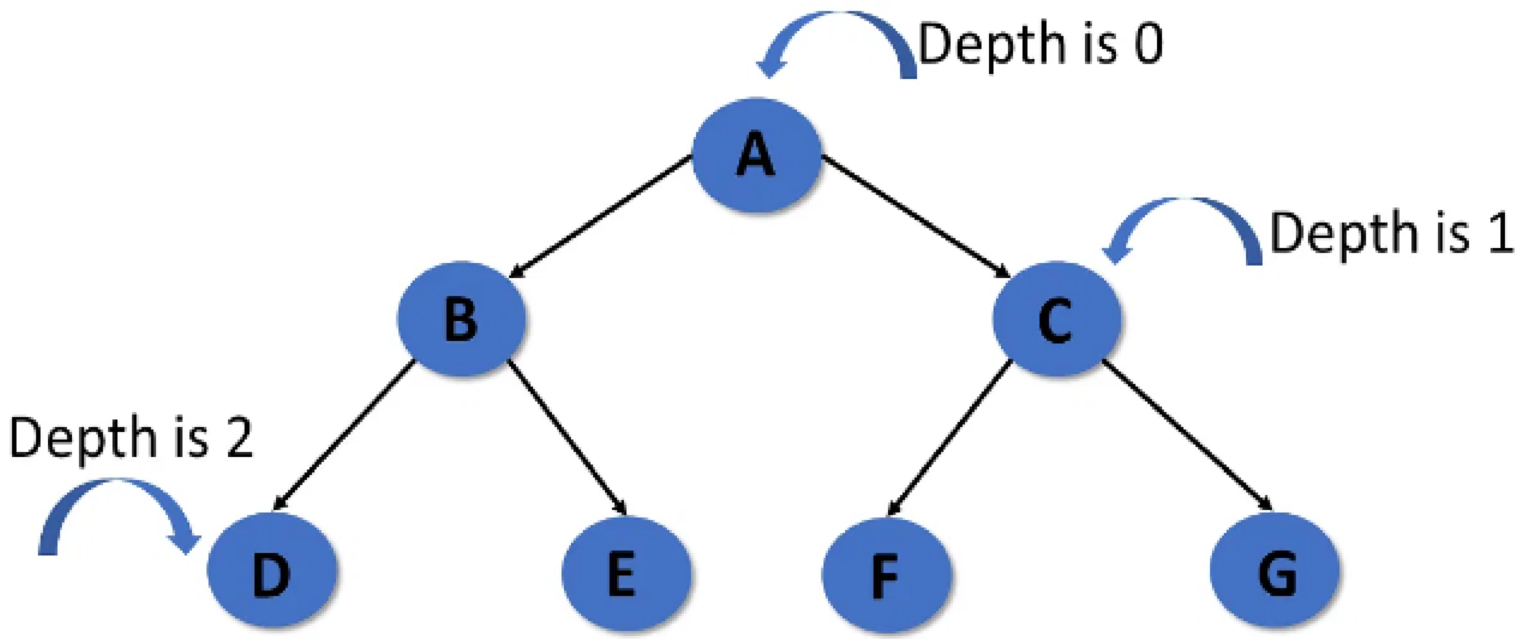


Terminologies

Depth

- the **total number of edges** from the **root node to the leaf node in the longest path** is known as **"Depth of Tree".**

In the tree data structures, the **depth of the root node is 0.**

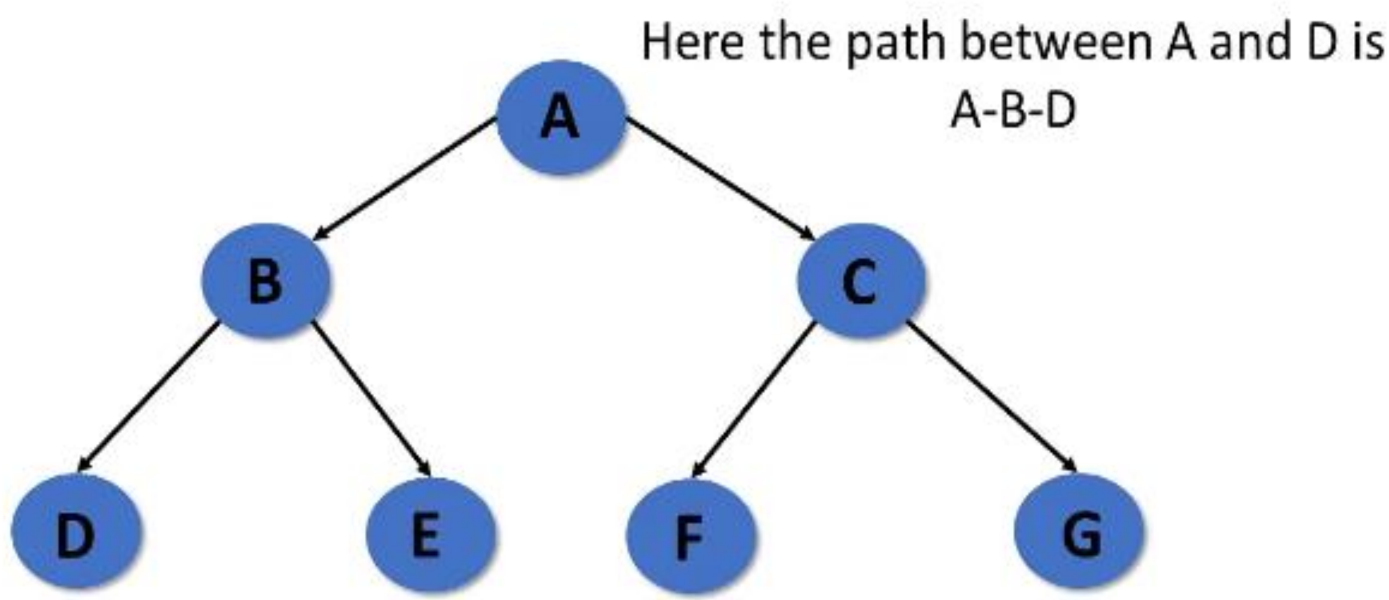


Terminologies

Path

- the **sequence of nodes and edges** from **one node to another node** is called the path between those two nodes.

The **length of a path** is **the total number of nodes** in a path



Representation – General Tree

- **each node** in a tree **can have an arbitrary number of children**, and that **number is not known in advance**
- the *general* tree can be implemented using a **first child/next sibling** method

```
struct tree_node
{
element_type element;
tree_ptr first_child;
tree_ptr next_sibling;
};
```

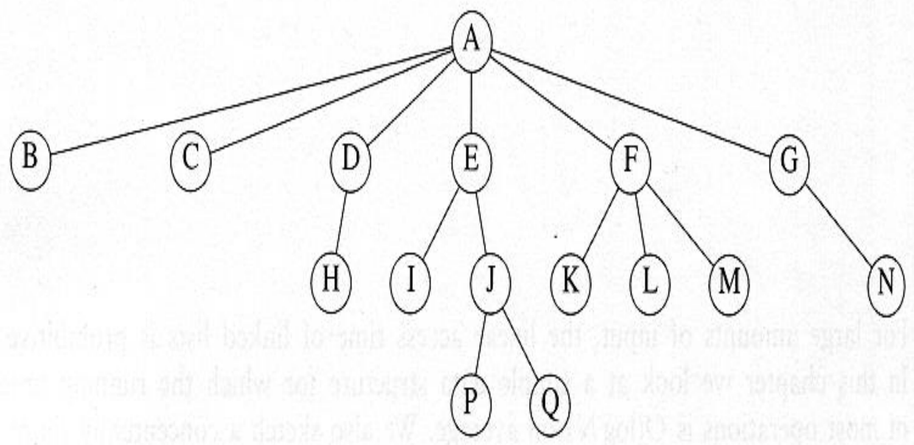
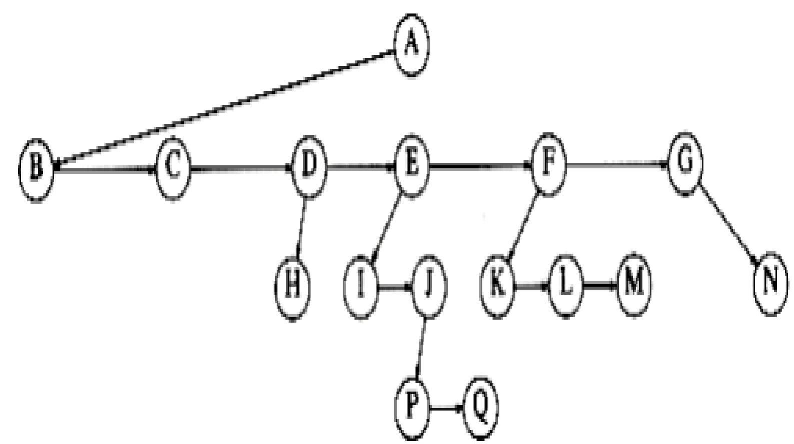


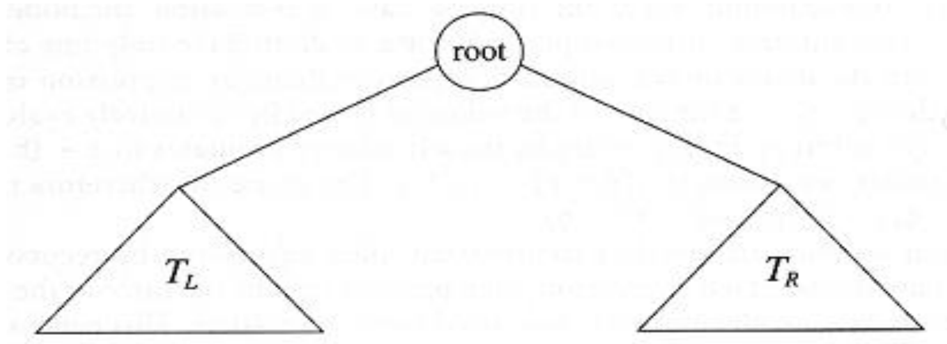
Figure 4.2 A tree



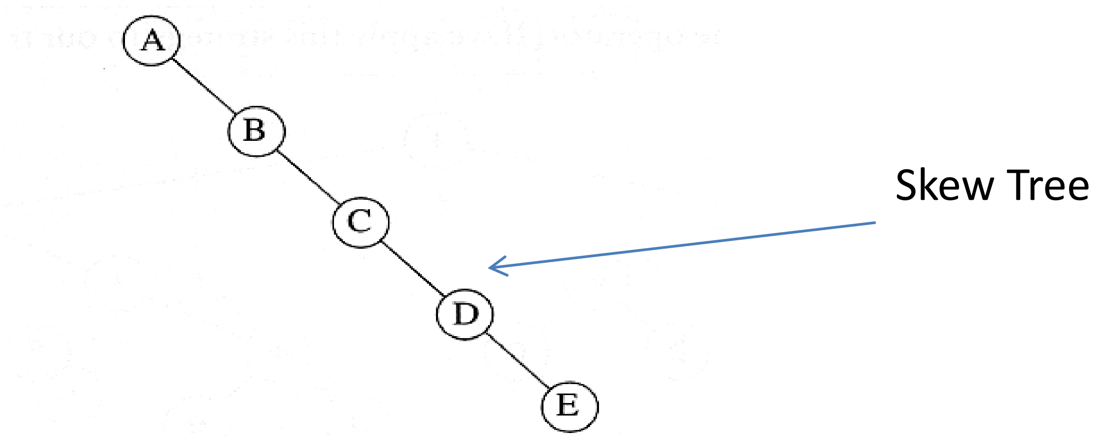
- Arrows that **point downward** are ***first_child*** pointers
- Arrows that **go left to right** are ***next_sibling*** pointers

Binary Trees

- A tree in which **no node can have more than two children**



- The depth of an "average" binary tree is considerably smaller than N, even though in the worst case, the depth can be as large as N – 1.



Expression Trees

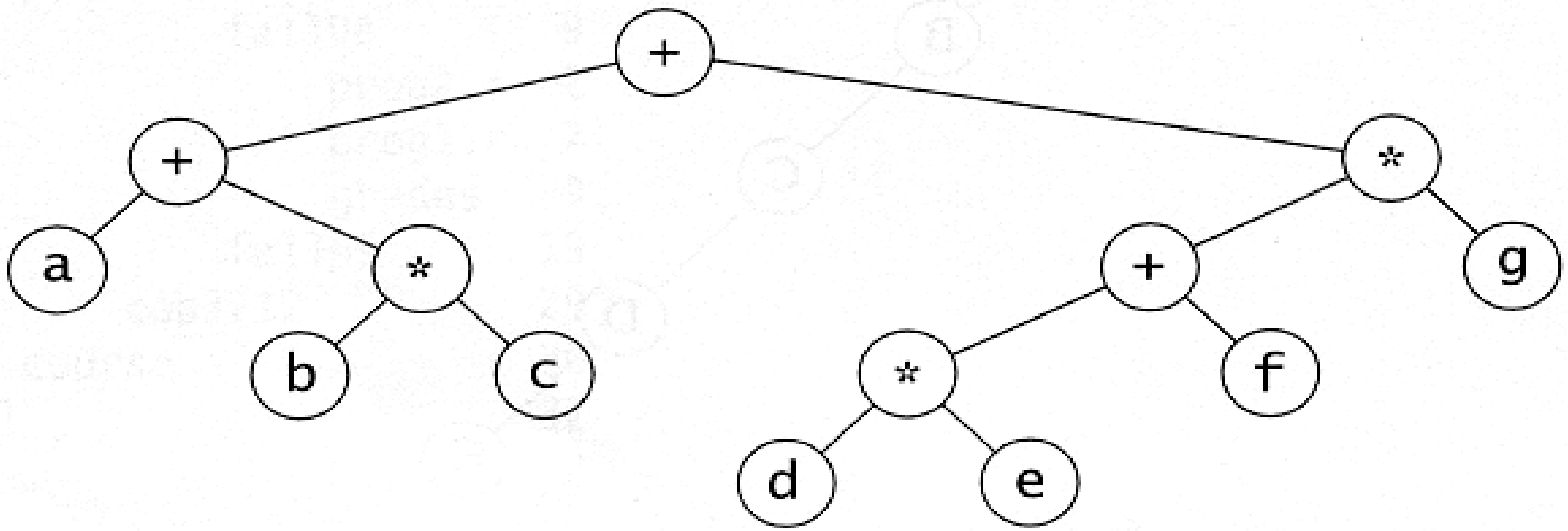


Figure 4.14 Expression tree for (a + b * c) + ((d * e + f) * g)

- The expression tree is a **binary tree** in which each **internal node corresponds to the operator** and **each leaf node corresponds to the operand**

Procedure - Expression Trees

Algorithm

1. Scan the expression from left to right.
2. If an operand is found. Then create a node for this operand and push it into the stack.
3. If an operator is found. Pop two nodes from the stack and make a node keeping the operator as the root node and the two items as the left and the right child. Push the newly generated node into the stack.
4. Keep repeating the above two steps until we reach the end of our postfix expression.
5. The node that is left behind in the stack represents the head of the expression tree.

Tree Traversals

- Used to print out the data in a tree in a certain order
- Pre-order traversal
  - **Print the data at the root**
  - **Recursively print out** all data in the **left subtree**
  - **Recursively print out** all data in the **right subtree**

Contd..

- Preorder traversal
  - Process Node data, visit Left Subtree recursively, visit Right Subtree recursively
  - **Produces prefix expression**

    ++a*bc*+*defg

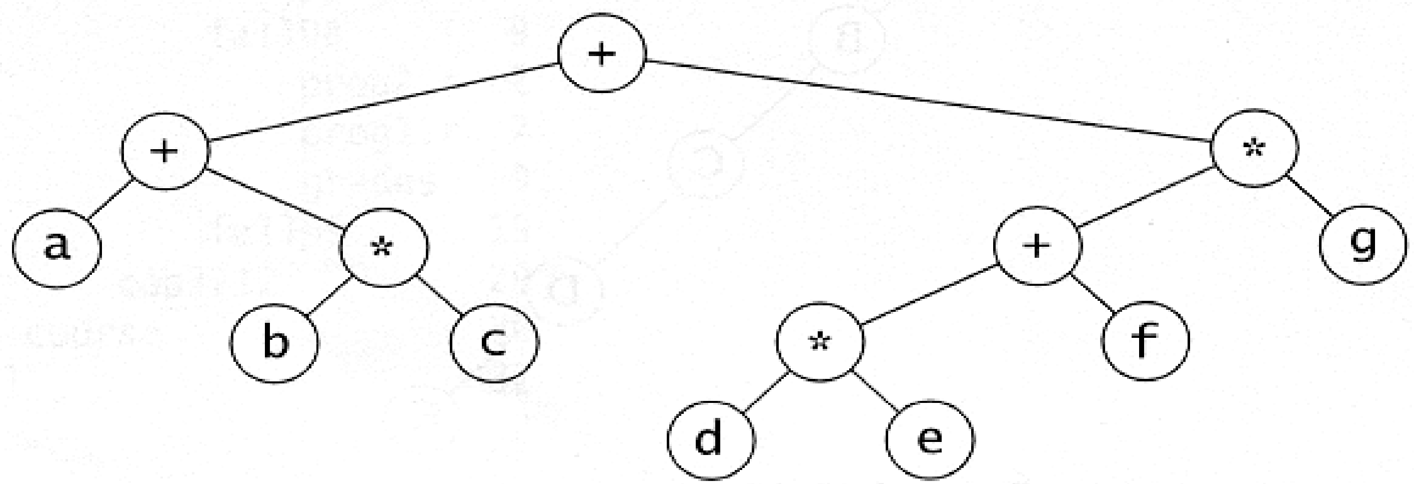


Figure 4.14 Expression tree for (a + b * c) + ((d * e + f) * g)

Contd..

- **Postorder traversal**
  - **Visit Left Subtree recursively, Visit Right Subtree Recursively, Node data**
  - postfix expression
    - abc*+de*f+g*+

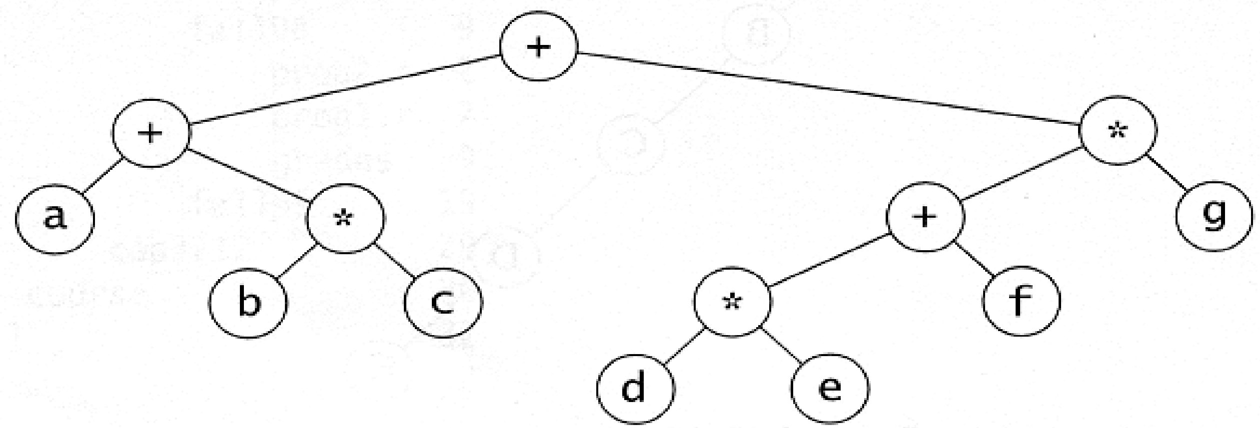


Figure 4.14 Expression tree for (a + b * c) + ((d * e + f) * g)

- **Inorder traversal**
  - **left, node, right**
  - **infix expression**
    - a+b*c+d*e+f*g

Contd..

Algorithm *Preorder*(x)
Input: x is the root of a subtree.
1. **if** $x \neq$ NULL
2. **then** output key(x);
3. *Preorder*(left(x));
4. *Preorder*(right(x));

Algorithm *Inorder*(x)
Input: x is the root of a subtree.
1. **if** $x \neq$ NULL
2. **then** *Inorder*(left(x));
3. output key(x);
4. *Inorder*(right(x));

Algorithm *Postorder*(x)
Input: x is the root of a subtree.
1. **if** $x \neq$ NULL
2. **then** *Postorder*(left(x));
3. *Postorder*(right(x));
4. output key(x);

Binary Tree - Representation

A **Binary tree** data structure is represented using two methods.

- **Array Representation**
- **Linked List Representation**

Array Representation

```
arr[] = {1, 2, 3, 4, 5, 6}
```

```
        1
       / \
      2   3
     / \ /
    4  5 6
```

- if the **parent node is at** index i in the array then **(array index starts at ‘0’)**
- the **left child of that node** is at **index (2*i + 1)** and
- the **right child** is at **index (2*i + 2)** in the array.

Binary Tree - Representation

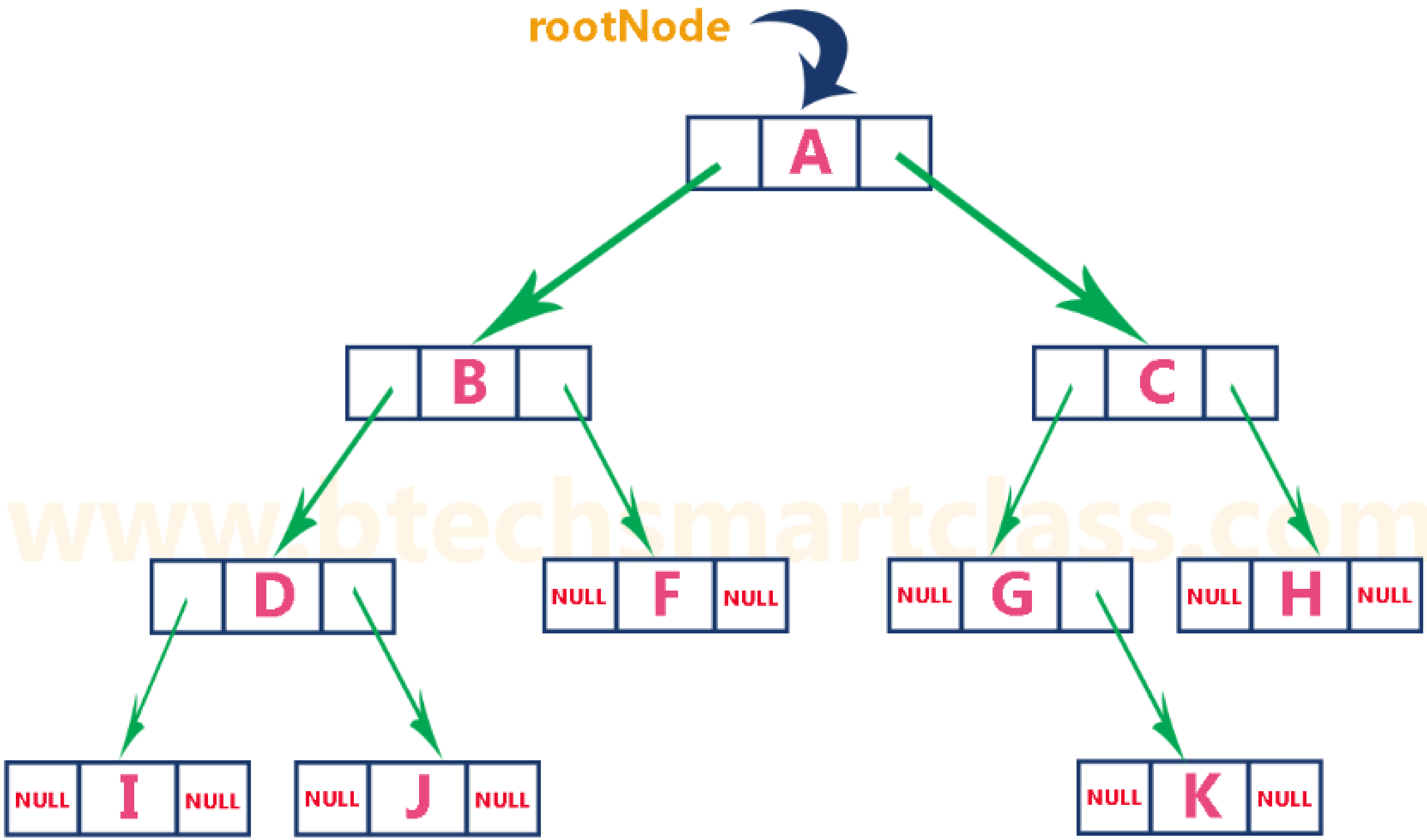
Linked List Representation

- use a **doubly linked list** to represent a binary tree
- every node consists of 3 fields.
- Field-1 for storing left child address,
- Field-2 storing actual data and
- Field-3 for storing right child address.

Node Structure:

Left Child Address	Data	Right Child Address

Binary Tree - Representation



Enumeration of Binary Tree

It means that the **no. of distinct binary trees created** from a **given no. of nodes or a binary tree.**

Unlabelled Binary Tree - Binary Tree is unlabelled if nodes are not assigned any label.

```
Below two are considered same unlabelled trees
    o                 o
  /   \             /   \
 o     o           o     o
```

Labelled Binary Tree - Binary Tree is labeled if every node is assigned a label

```
Below two are considered different labelled trees
    A                C
  /   \             /  \
 B     C           A    B
```

Enumeration of Binary Tree

- No. of Unlabelled Binary Trees = (2n)! / (n+1)! * n!
- When n = 3, No. of Unlabelled B.Trees is = 5

- No. of Labelled Trees = (Number of unlabelled trees) * n!

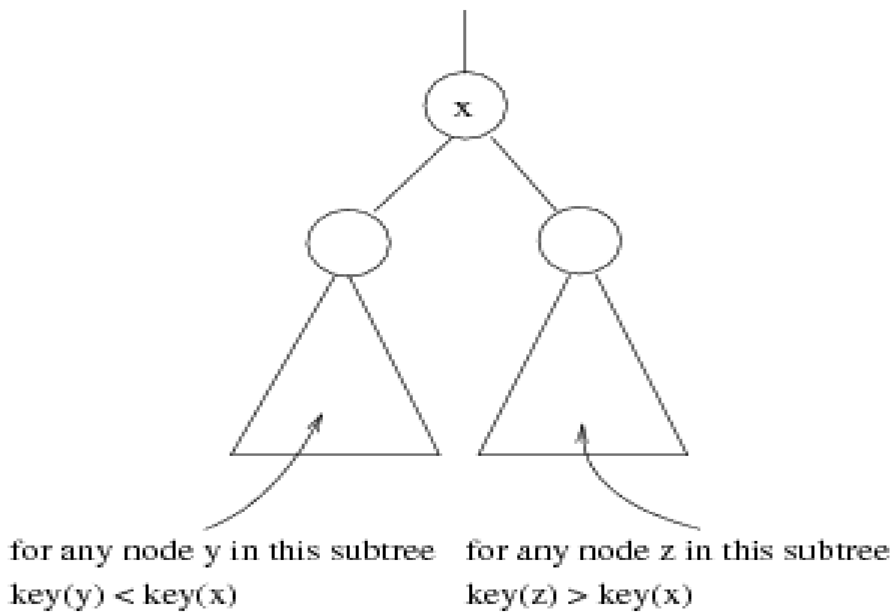
  = [(2n)! / (n+1)!n!] × n!
- For example for n = 3, there are 5 * 3! = 5*6 = 30 different labelled trees

Binary Search Trees

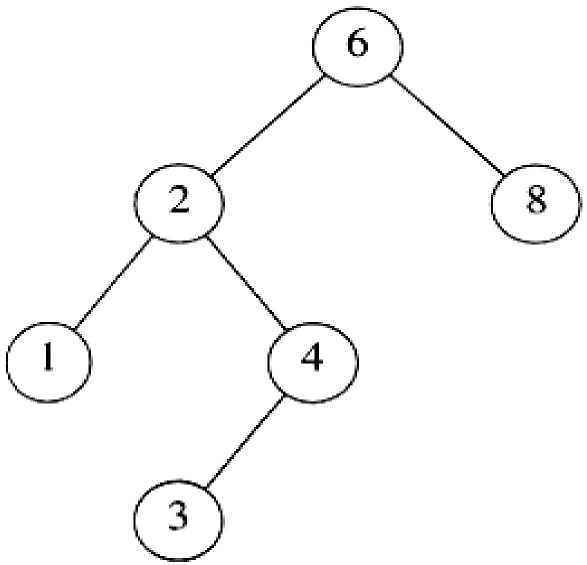
- **Stores keys in the nodes in a way so that searching, insertion and deletion** can be done efficiently.

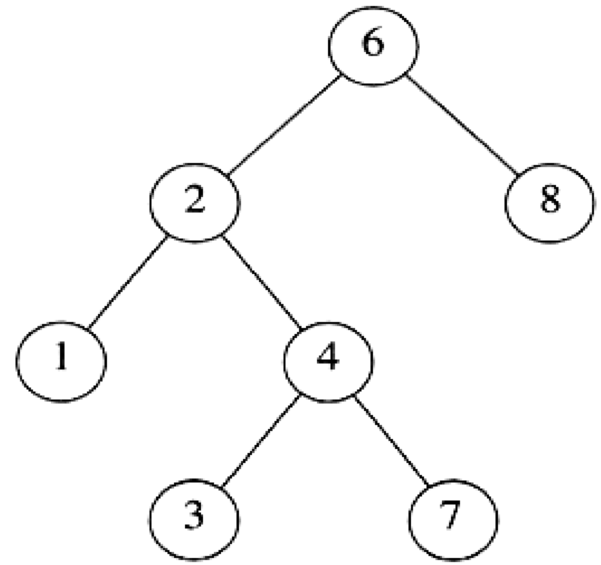
Binary search tree property

value(LST) < value (Root) < value (RST)



Binary Search Tree





After Insert(7)

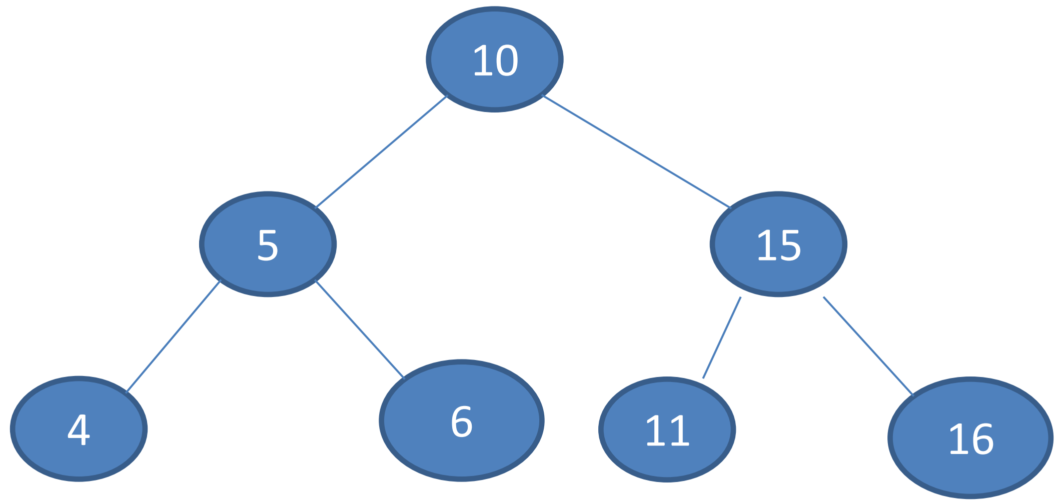
Binary Search Tree from Preorder Traversal

Procedure:

1. Take the **first element as the root** of the BST
2. **Identify the index (i)** of **immediate largest element** of **the root element**
3. **Take the elements** from **index i to n-1** for **'Right Sub Tree'**
4. The **elements between index of root to i** will be taken to **'Left Sub Tree'.**
5. **Recursively do the same for constructing BST**

Binary Search Tree from Preorder Traversal

Ex: 10, 5, 4, 6, 15, 11, 16



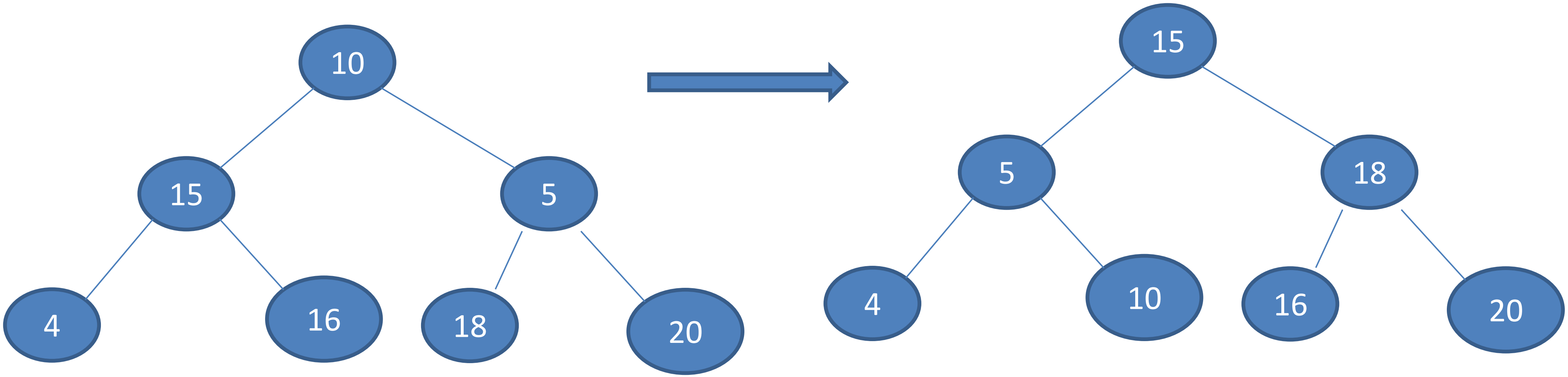
- The first element 10 will be root node, because in pre order traversal the root of the BT will be visited at first.
- The immediate largest element of root element 10 is 15, and its index 4.
- The elements in index 4 to 6 will be taken to RST.
- The elements in index 1 to 3 will be taken to LST.
- Repeat the same on the above portion of elements to find the roots of subtrees

Binary Search Tree from Binary Tree

Procedure:

1. Declare an array to have in-order traversal of Binary Tree
2. **Sort it using a Sorting Algorithm which consumes less time complexity**
3. Take the **in-order traversal** of the **Binary Tree** and **move the nodes to Binary Search Tree**

Binary Tree | **Binary Search Tree**



- Take the in-order traversal into an array: Temp[] = 4 15 16 10 18 5 20
- Sort the array: Temp[] = 4 5 10 15 16 18 20
- Take nodes using in-order traversal for constructing BST.