WOLLO UNIVERSITY
KOMBOLCHA INSTITUTES OF TECHNOLOGY
College of Informatics

Data Structures and Algorithms

Chapter 6

Tree Structures

Belachew N.
nbelay2112@gmail.com

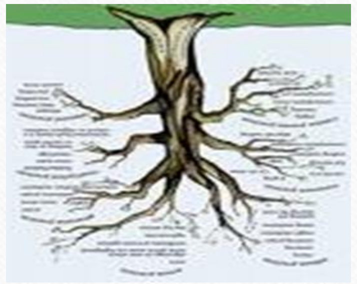
Outline

- ✓ Basic Concepts of Trees in Data Structure
- ✓ Tree Terminology
- ✓ Types of Tree in Data Structure
- ✓ Basic Operations on BST
  - Insertion
  - Searching
  - Deletion
  - Traversing
- ✓ Applications of Tree

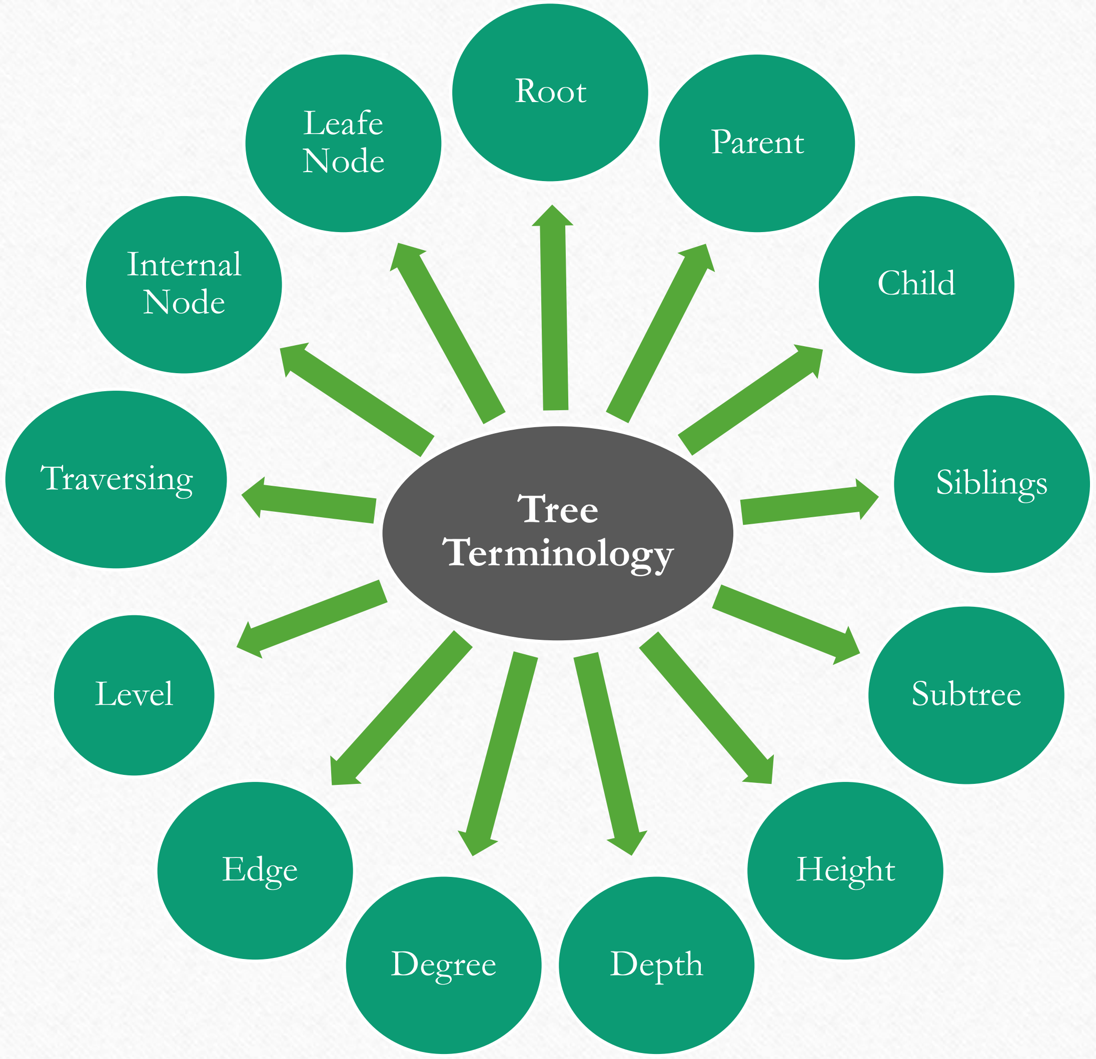
Trees in Data Structure

- ✓ Tree is a hierarchical data structure which stores the information naturally in the form of hierarchy style.
- ✓ Tree is one of the most powerful and advanced data structures.
- ✓ It is a non-linear data structure compared to arrays, linked lists, stack and queue.
- ✓ It represents the nodes connected by edges.
- ✓ Unlike a real tree, tree data structures are typically depicted upside down, with the root at the top and the leaves at the bottom.

Real tree



Tree in Data Structure

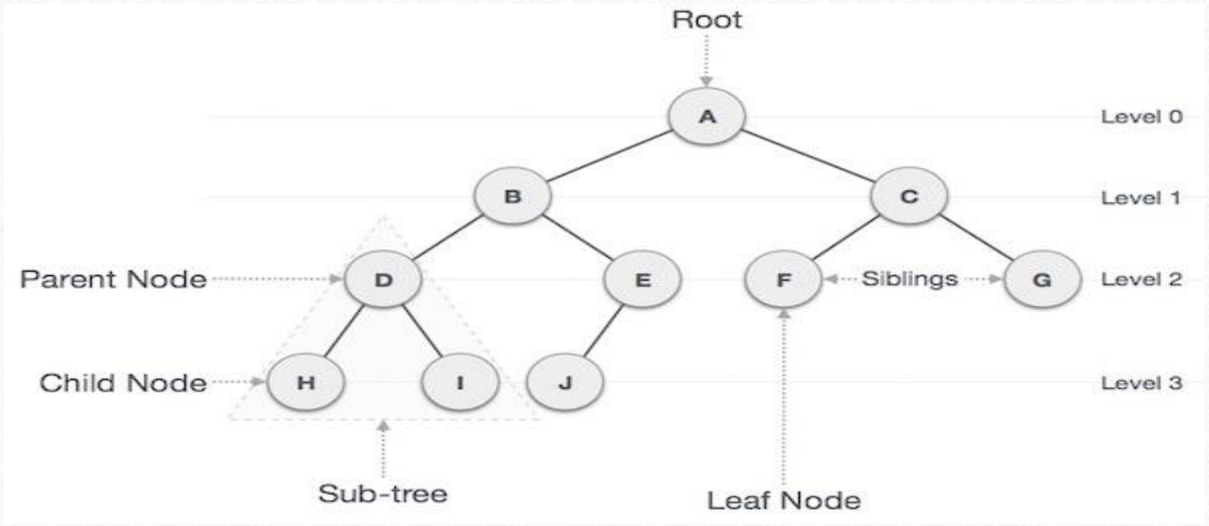

Tree Terminology
Root
Parent
Child
Siblings
Subtree
Height
Depth
Degree
Edge
Level
Traversing
Internal Node
Leafe Node

…cont'd

- ✓ **Root:** The node at the top of the tree is called root.
- ✓ **Parent Node:** Parent node is an immediate predecessor of a node.
- ✓ **Child Node:** All immediate successors of a node are its children.
- ✓ **Siblings:** Nodes with the same parent are called Siblings.
- ✓ **Path:** It is a number of successive edges from source node to destination node.
- ✓ **Subtree** − Subtree represents the descendants of a node.
- ✓ **Height of a node** represents the number of edges on the longest path between that node and a leaf.
- ✓ **Depth of Node** represents the number of edges from the tree's root node to the node.
- ✓ **Degree of Node** represents a number of children of a node.
- ✓ **Edge:** It is a connection between one node to another. It is a line between two nodes or a node and a leaf.
- ✓ **Levels** − Level of a node represents the generation of a node.
  - If the root node is at level 0, then its next child node is at level 1, its grandchild is at level 2, and so on.
- ✓ **Traversing** − Traversing means passing through nodes in a specific order.

…cont'd

- ✓ If node has no children, it is called **Leaves** or **External Nodes**.
- ✓ Nodes which are not leaves, are called **Internal Nodes**.
  - Internal nodes have at least one child.
- ✓ A tree can be empty with no nodes or a tree consists of one node called the Root.

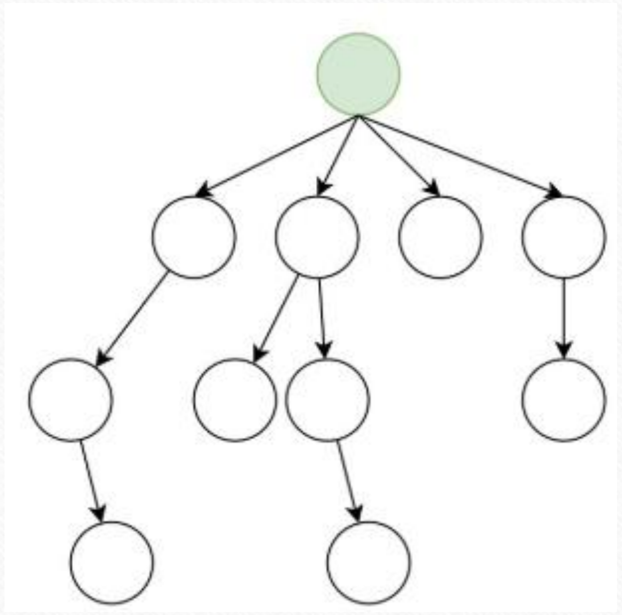


Types of Tree in Data Structure

- ✓General Tree
- ✓Forests
- ✓Binary Tree
  - Full binary tree
  - Balanced binary tree
  - Complete binary tree
  - Binary Search Tree
  - Expression Tree

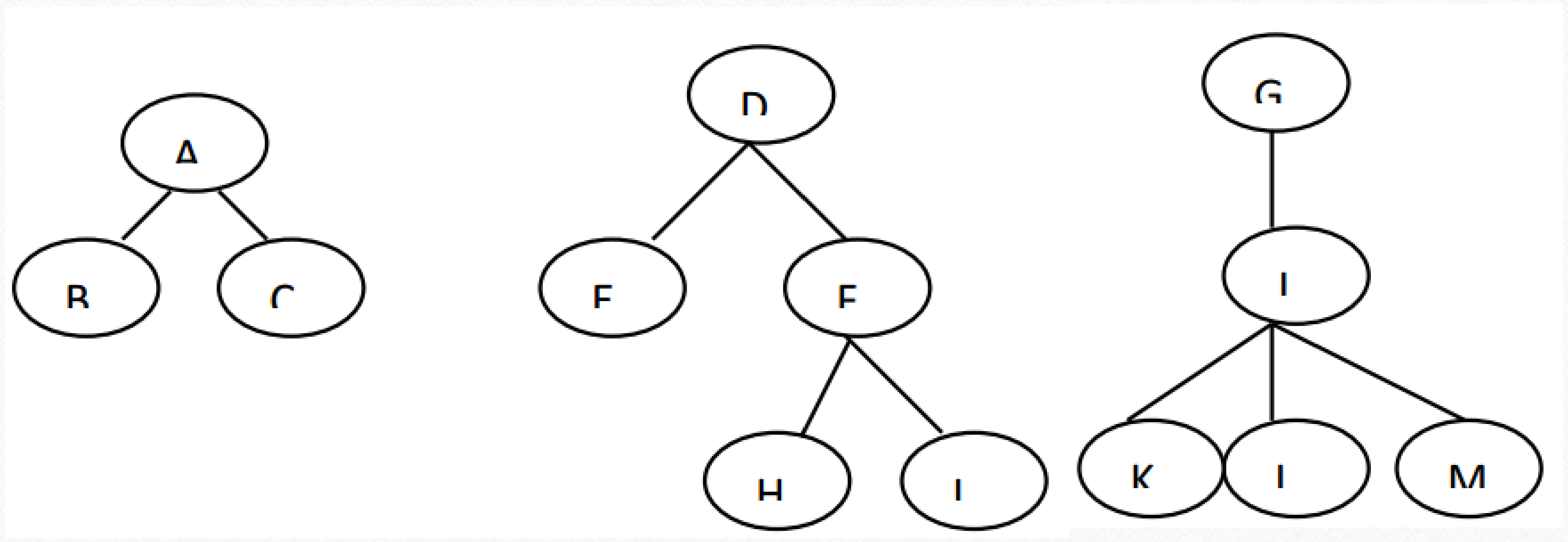
General tree

- ✓In the data structure, General tree is a tree in which each node can have either zero or many child nodes.
- ✓In general tree, there is no limitation on the degree of a node.



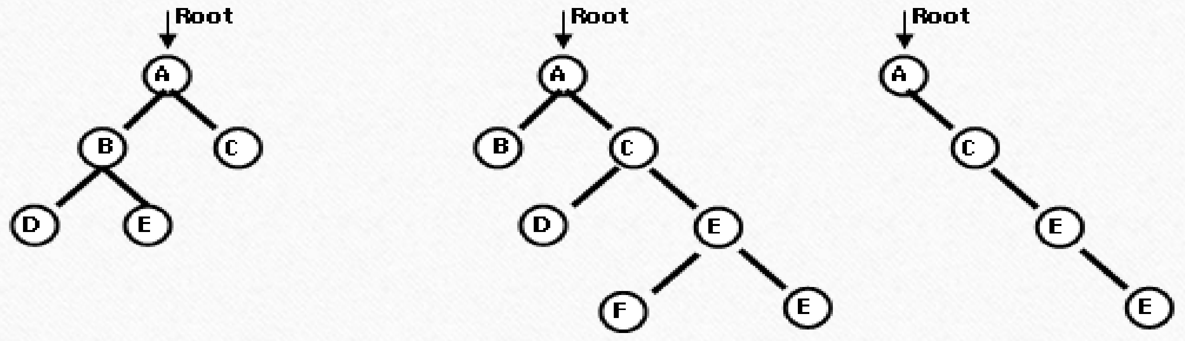
Forests

✓A set of trees is called **forest**;



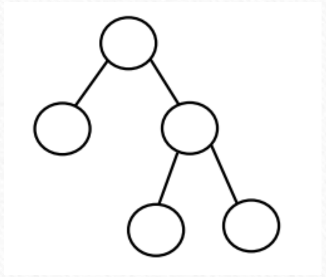
Binary tree

- ✓ A binary tree is the specialized version of the General tree.
- ✓ A binary tree is a tree in which each node can have at most two nodes.
- ✓ In a binary tree, there is a limitation on the degree of a node because the nodes in a binary tree can't have more than two child node(or degree two)
- ✓ The topmost node of a binary tree is called root node and there are mainly two subtrees one is left-subtree and another is right-subtree.

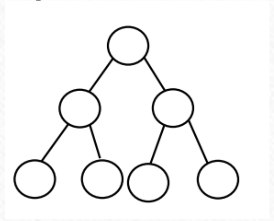


…cont'd

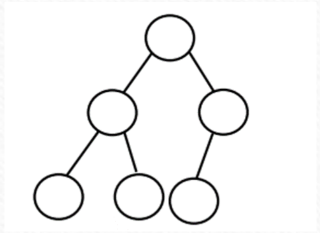
- ✓ **Full binary tree**
  - A binary tree where each node has either 0 or 2 children.



- ✓ **Balanced Binary Tree**
  - A binary tree where each node except the leaf nodes has left and right children and all the leaves are at the same level.

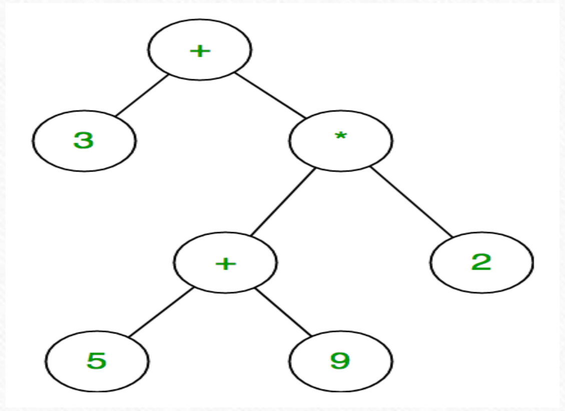


- **Complete Binary Tree**
  - A binary tree in which the length from the root to any leaf node is either h or h-1 where h is the height of the tree.
  - The deepest level should also be filled from left to right.



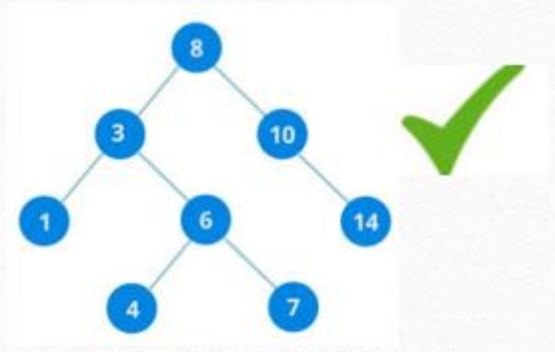
Expression Tree

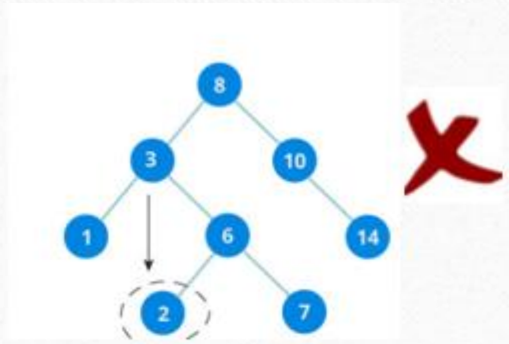
- ✓ Expression Tree is a special kind of binary tree with the following properties:
  - Each leaf is an operand.
  - The root and internal nodes are operators.
  - Subtrees are subexpressions with the root being an operator.
- ✓ For example expression tree for 3 + ((5+9)*2) would be:



Binary search tree (ordered binary tree)

- ✓ A Binary Search Tree (BST) is a tree in which all the nodes follow the below-mentioned properties
  - Every node has a key and no two elements have the same key.
  - The keys in the right subtree are larger than the keys in the root.
  - The keys in the left subtree are smaller than the keys in the root.
  - The left and the right subtrees are also binary search trees.





Binary Tree Properties

- ✓Minimum number of nodes in a binary tree of height H= H + 1
- ✓Maximum number of nodes in a binary tree of height H= $2^{H+1} – 1$
- ✓Maximum number of nodes at any level ‘L’ in a binary tree = 2^L
- ✓The maximum number of nodes $\underline{n}$ for any binary tree of depth $\underline{d}$ is:

$$n = 2^0 + 2^1 + 2^2 + \cdots + 2^{d-1} + 2^d = \sum_{k=0}^{d} 2^k = \underline{2^{d+1} - 1}$$

Data Structure of a Binary Tree

- ✓ The declaration of tree nodes is similar in structure to that for doubly linked lists, in that a node is a structure consisting of the key information plus two pointers (left and right) to other nodes.

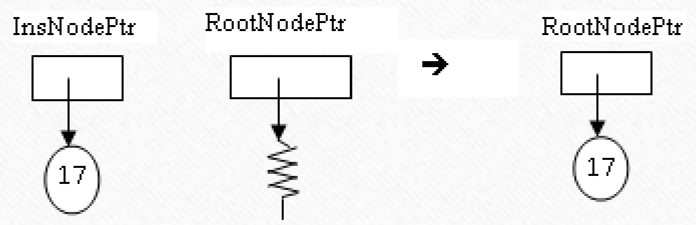
```
Struct datamodel{
        datafield declaration;
        datamodel  *left, *right;
}datamodel  *rootpointer=NULL;
```

Operations on BST

- ✓Insertion
- ✓Searching
- ✓Deletion
- ✓Traversing

Insertion

- ✓ Suppose there is a binary search tree whose root node is pointed by RootNodePtr and we want to insert a node (that stores 17) pointed by InsNodePtr.
- ✓ **Case 1:** There is no data in the tree (i.e. RootNodePtr is NULL)
  - The node pointed by InsNodePtr should be made the root node.

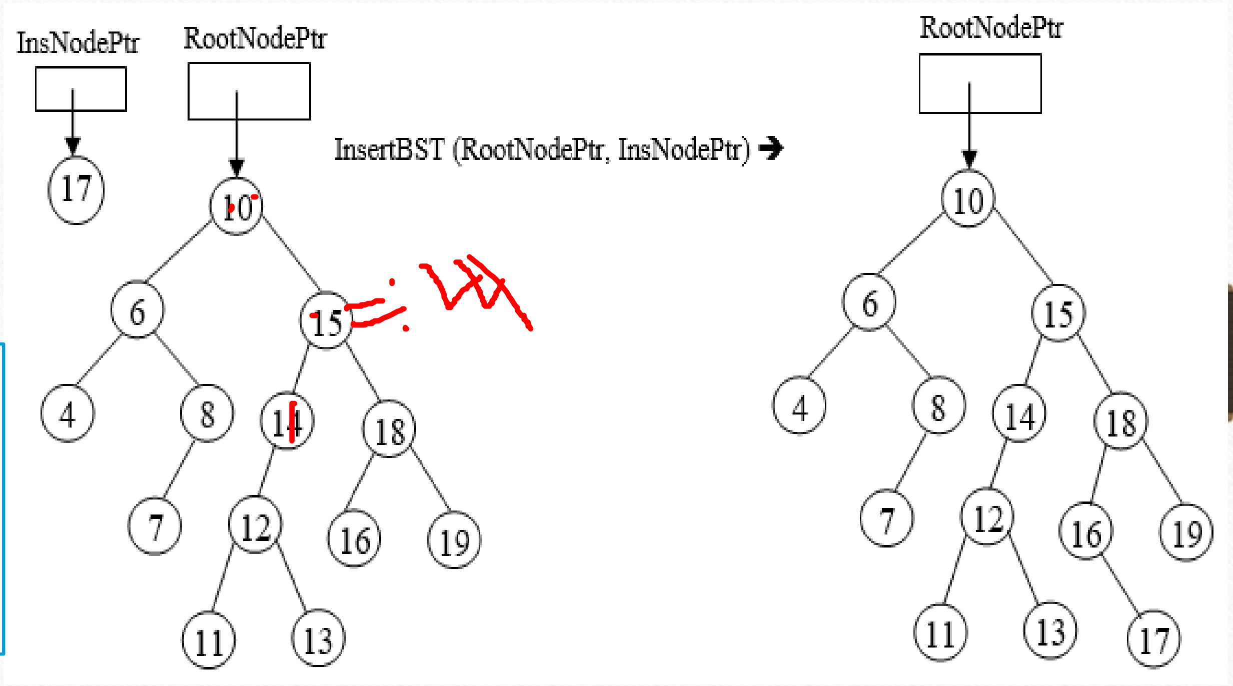


…cont'd

✓ **Case 2:** There is data

- Search the appropriate position.
- Insert the node in that position.

```
Function call:
if (RootNodePtr = = NULL)
    RootNodePtr=InsNodePtr;
else
    InsertBST (RootNodePtr, InsNodePtr);
```



…cont’d

```
void InsertBST(Node *RNP, Node *INP)
{
        //RNP=RootNodePtr and INP=InsNodePtr
        int Inserted=0;
        while(Inserted = =0)
        {
                if(RNP->Num > INP->Num)
                {
                        if(RNP->Left = = NULL)
                        {
                                RNP->Left = INP;
                                Inserted=1;
                        }
                        else
                                RNP = RNP->Left;
                }
                else
                {
                        if(RNP->Right = = NULL)
                        {
                                RNP->Right = INP;
                                Inserted=1;
                        }
                        else
                                RNP = RNP->Right;
                }
        }
}
```

…cont'd

A recursive version of the function can also be given as follows.

```
void InsertBST(Node *RNP, Node *INP)
{
        if(RNP->Num>INP->Num)
        {
            if(RNP->Left==NULL)
                RNP->Left = INP;
            else
                InsertBST(RNP->Left, INP);
        }
        else
        {
            if(RNP->Right==NULL)
                RNP->Right = INP;
            else
                InsertBST(RNP->Right, INP);
        }
}
```

Searching

- ✓ Searching a binary search tree not dissimilar to the process performed when inserting an item:
  - Compare the item that you are looking for with the root,
  - Then push the comparison down into the left or right subtree depending on the result of this comparison, until a match is found or a leaf is reached.
  - This takes at most as many comparisons as the height of the tree.
  - At worst, this will be the number of nodes in the tree minus one.

…cont’d

Function call:

ElementExists=searchBST(RNP, number);

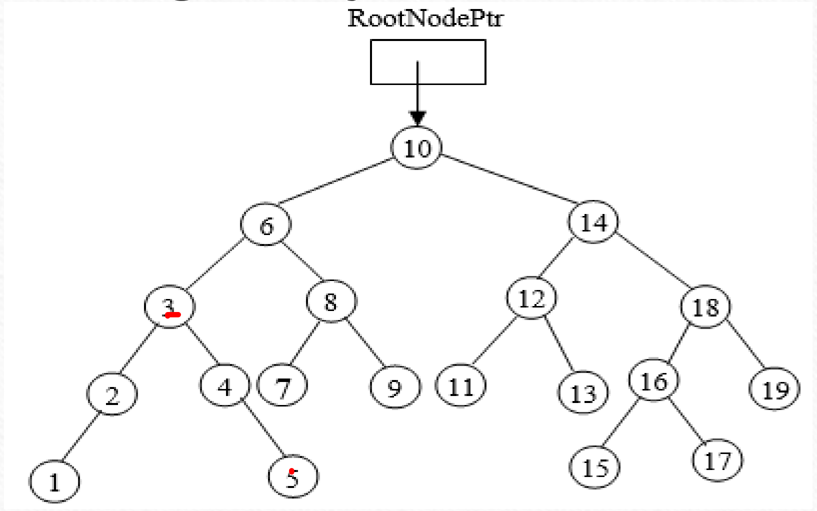
// ElementExists is a Boolean variable defined as: bool ElementExists = false;

The function (recursive implementation)

```
bool searchBST(Node *CurrNodeptr, int x)
   {
                if(currNodeptr==NULL)
                        return (false);
                else if (currNodeptr->num==x)
                        return (true);
                else if (currNodeptr->num>x)
                        return (searchBST(currNodeptr->left, x);
                else
                 return (searchBST(currNodeptr->right, x);
        }
```

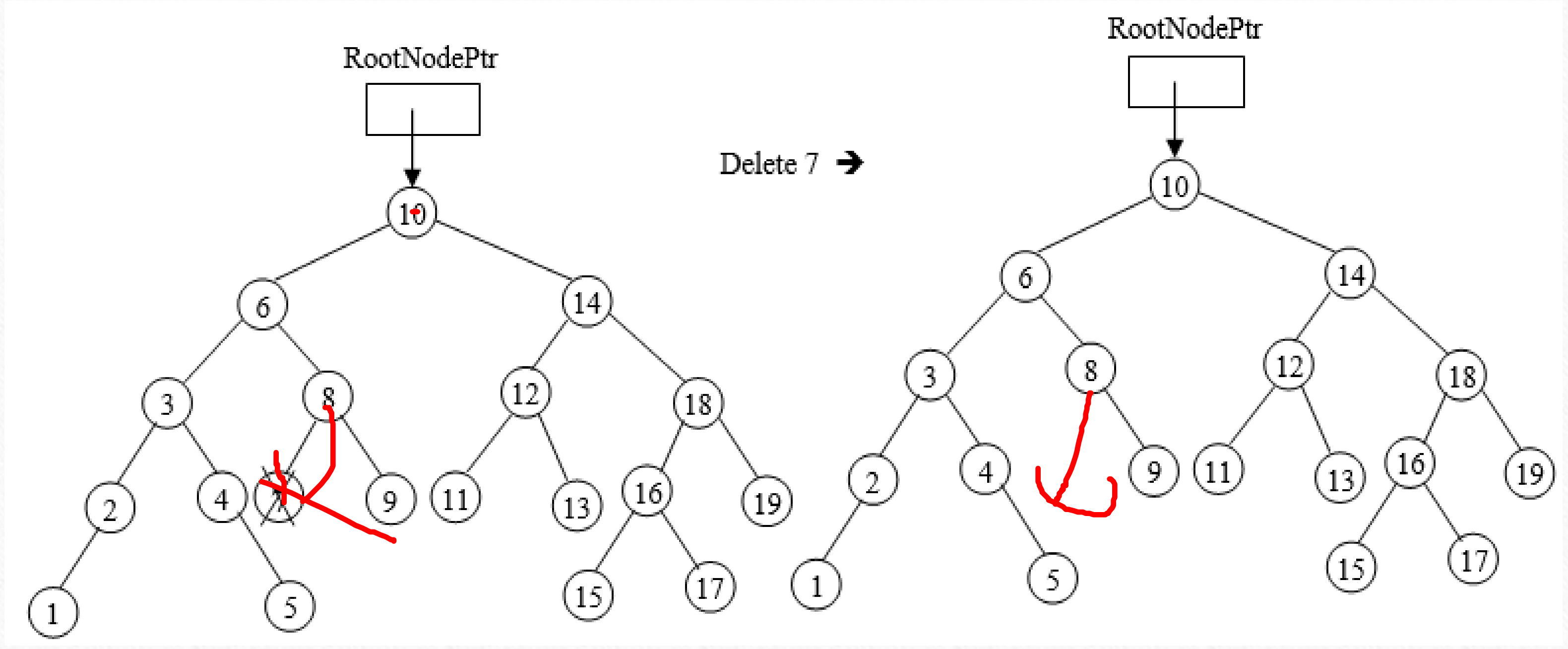
Deletion

- ✓Deletion of a node from a binary search tree can be more problematic.
- ✓The difficulty of the operation depends on the position of the node in the tree.
- ✓To delete a node from binary search tree, four cases should be considered.
- ✓Consider the following binary search tree.



…cont'd

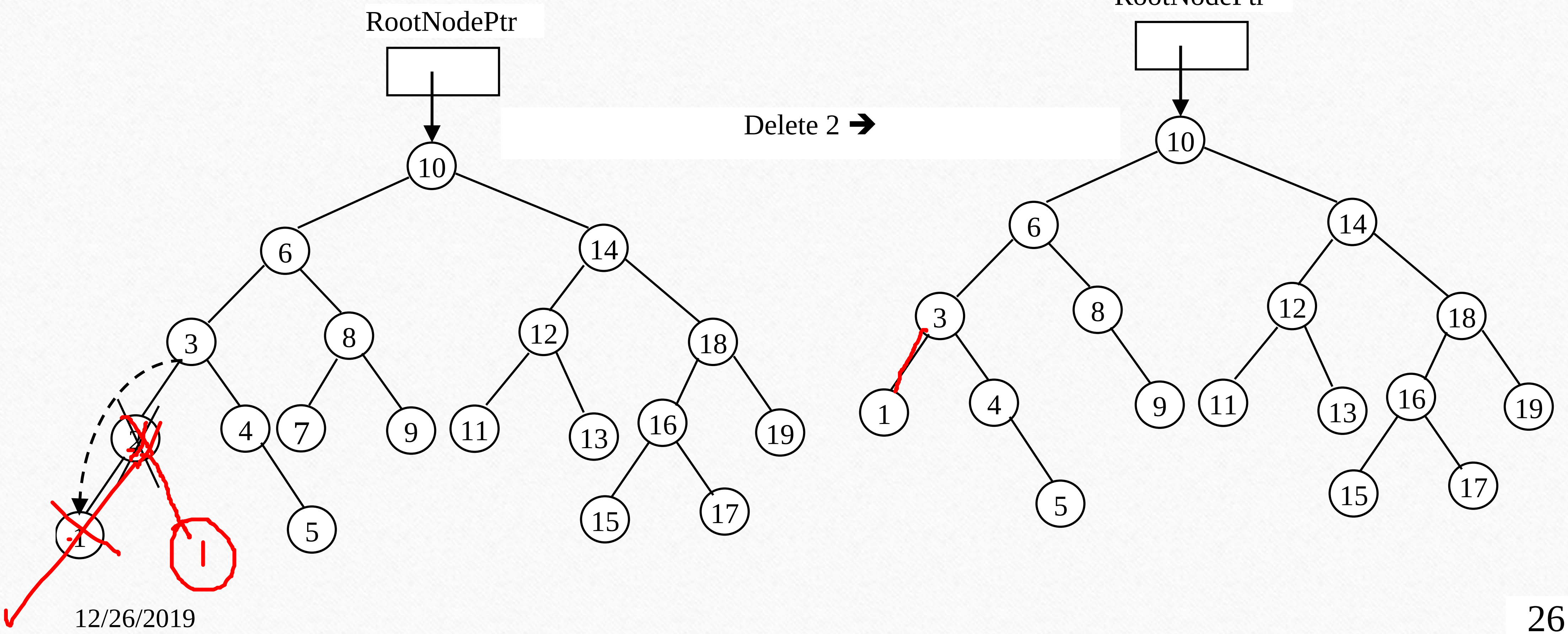
✓**Case 1**: Deleting a leaf node (a node having no child), e.g. 7



…cont'd

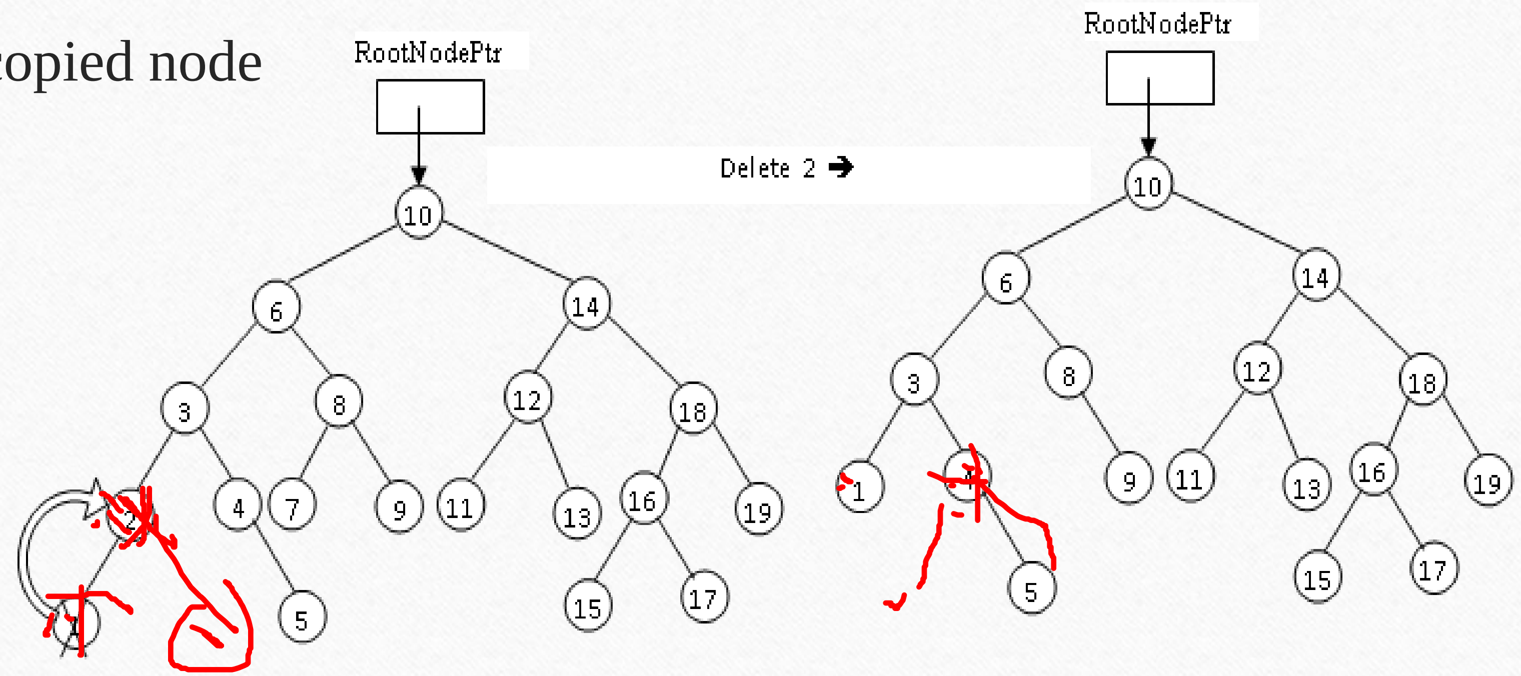
- ✓ **Case 2:** Deleting a node having only one child, e.g. 2
- ✓ **Approach 1**: Deletion by merging – one of the following is done
  - If the deleted node is the left child of its parent and the deleted node has only the left child, the left child of the deleted node is made the left child of the parent of the deleted node.
  - If the deleted node is the left child of its parent and the deleted node has only the right child, the right child of the deleted node is made the left child of the parent of the deleted node.
  - If the deleted node is the right child of its parent and the node to be deleted has only the left child, the left child of the deleted node is made the right child of the parent of the deleted node.
  - If the deleted node is the right child of its parent and the deleted node has only the right child, the right child of the deleted node is made the right child of the parent of the deleted node

…cont'd


RootNodePtr
Delete 2 ➔
RootNodePtr
10
6
14
3
8
12
18
2
4
7
9
11
13
16
19
1
5
15
17
1
10
6
14
3
8
12
18
1
4
9
11
13
16
19
5
15
17

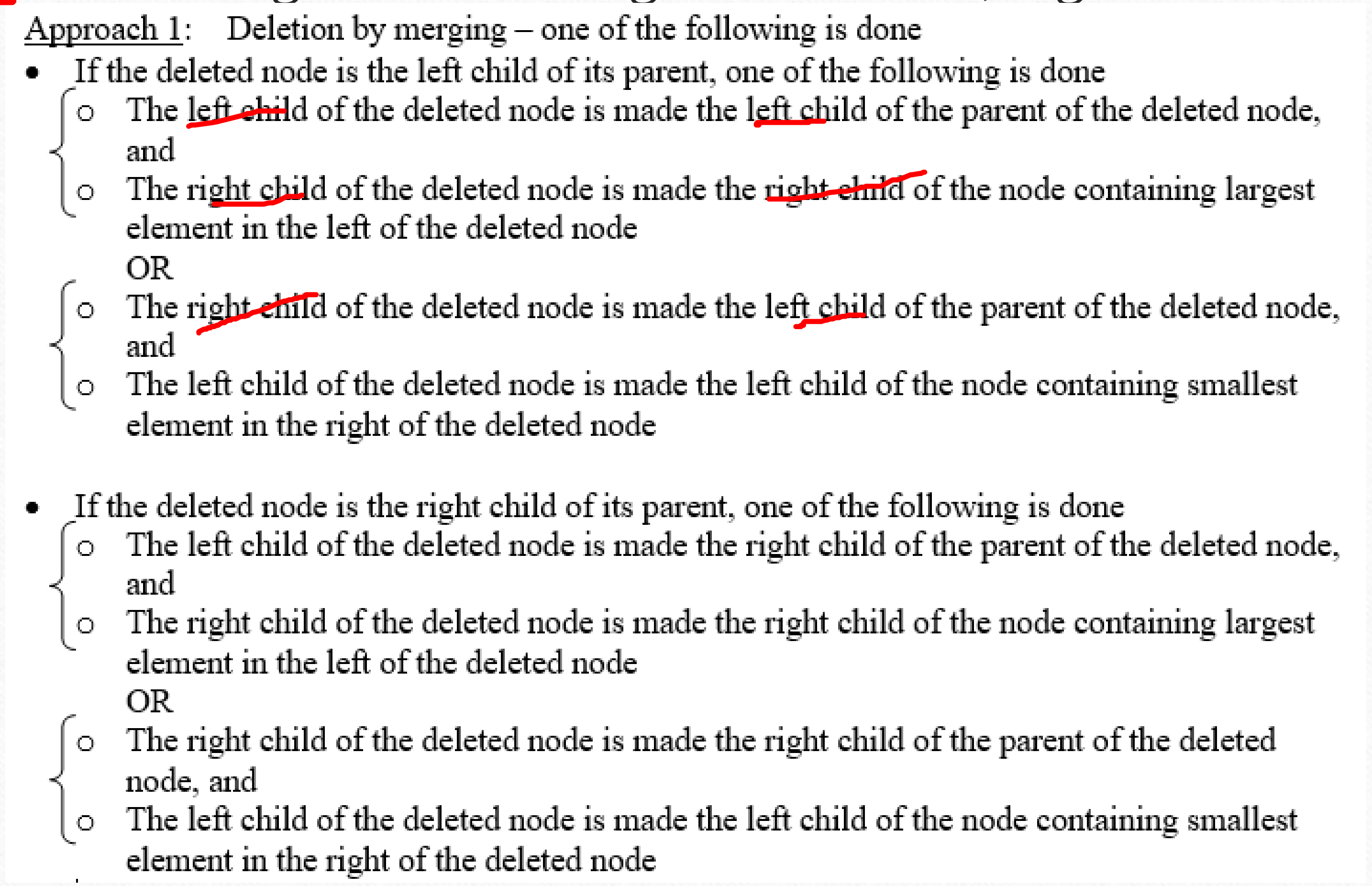
…cont'd

- ✓ **Approach 2**: Deletion by copying- the following is done
  - Copy the node containing the largest element in the left (or the smallest element in the right) to the node containing the element to be deleted
  - Delete the copied node



…cont'd

- ✓ Case 3: Deleting a node having two children, e.g. 6



Approach 1: Deletion by merging – one of the following is done

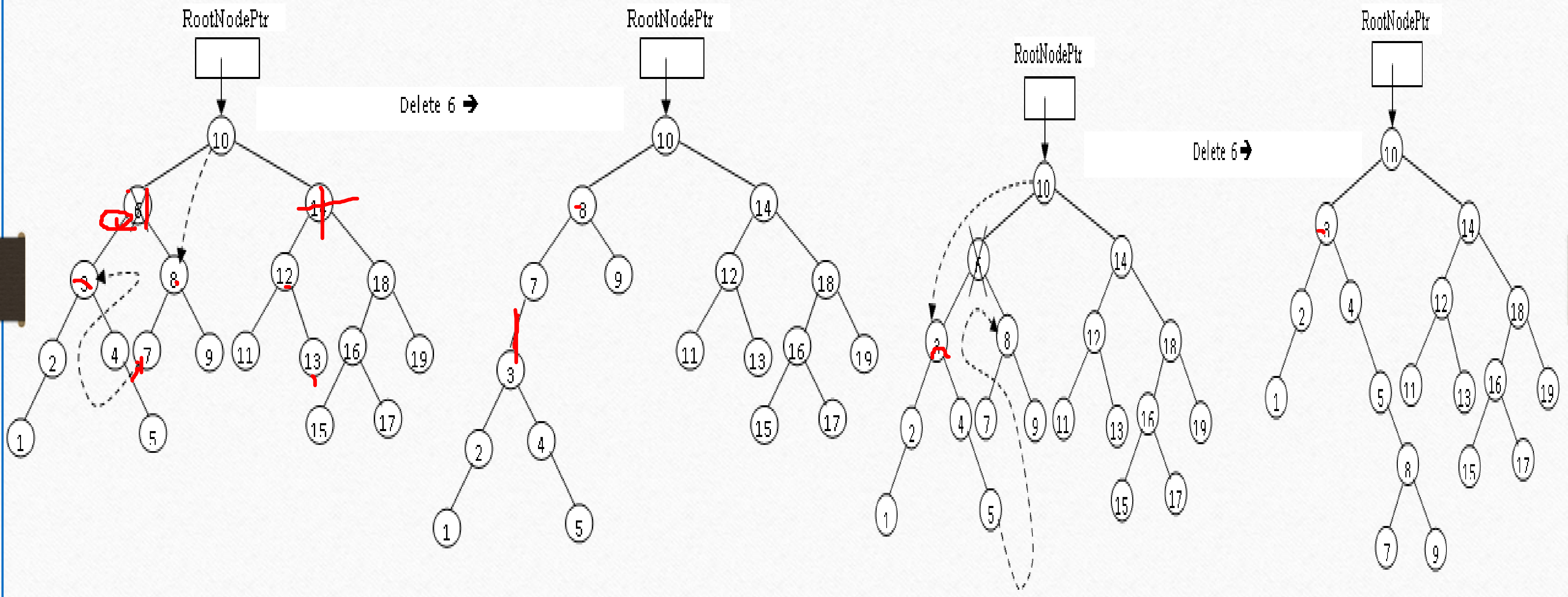
- If the deleted node is the left child of its parent, one of the following is done
  - The left child of the deleted node is made the left child of the parent of the deleted node, and
  - The right child of the deleted node is made the right child of the node containing largest element in the left of the deleted node

  OR
  - The right child of the deleted node is made the left child of the parent of the deleted node, and
  - The left child of the deleted node is made the left child of the node containing smallest element in the right of the deleted node

- If the deleted node is the right child of its parent, one of the following is done
  - The left child of the deleted node is made the right child of the parent of the deleted node, and
  - The right child of the deleted node is made the right child of the node containing largest element in the left of the deleted node

  OR
  - The right child of the deleted node is made the right child of the parent of the deleted node, and
  - The left child of the deleted node is made the left child of the node containing smallest element in the right of the deleted node

…cont'd

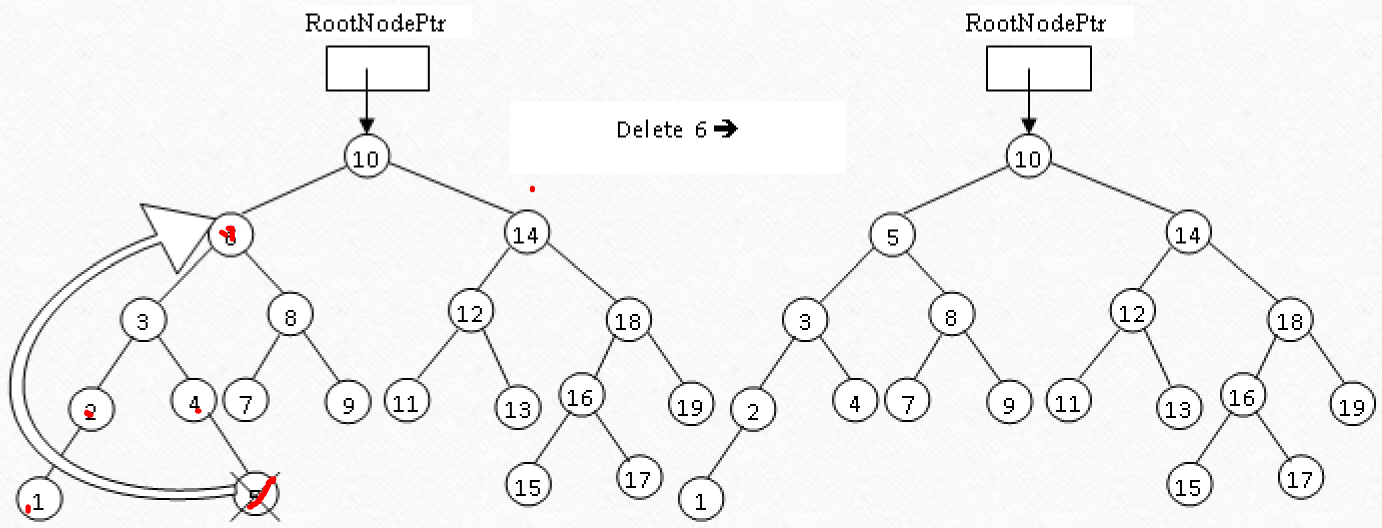


RootNodePtr
Delete 6 ➔
RootNodePtr
RootNodePtr
Delete 6 ➔
RootNodePtr

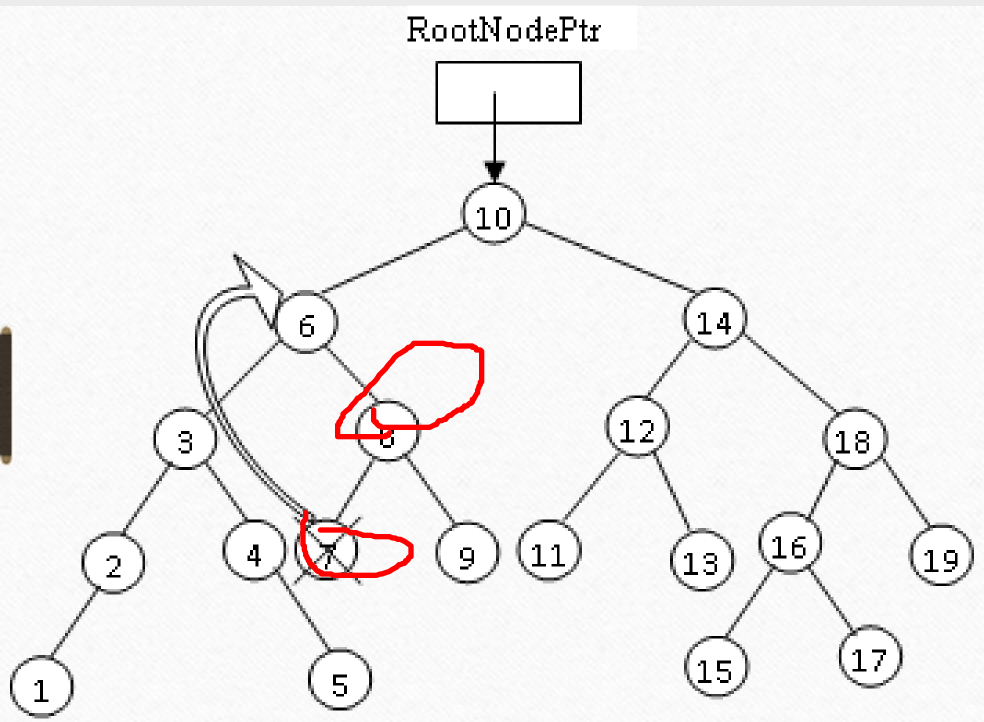
…cont’d

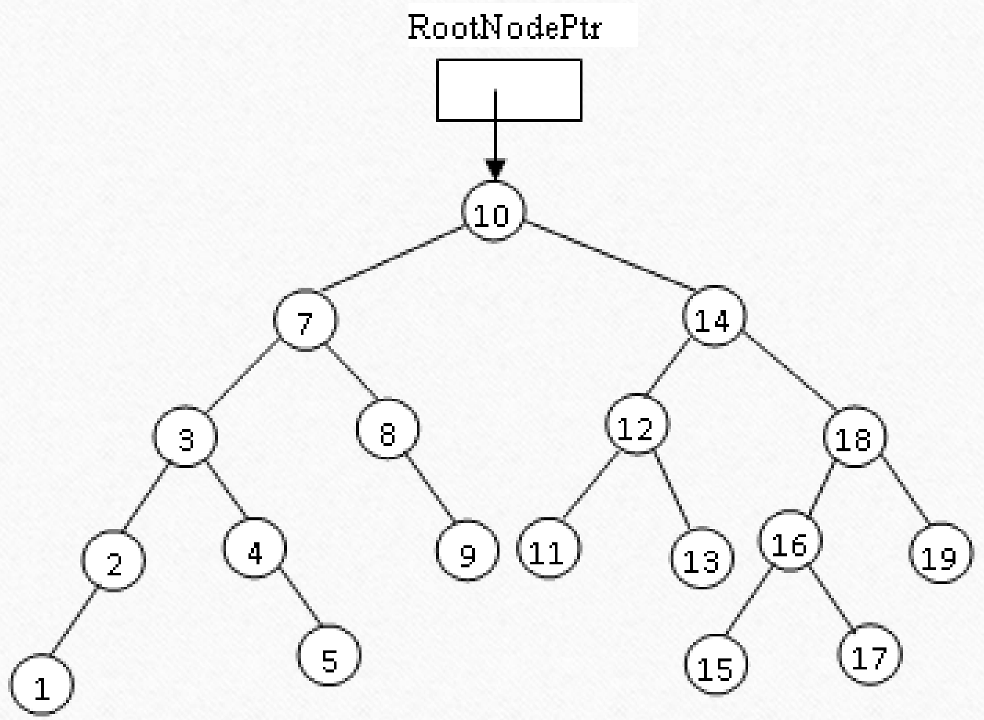
✓ Approach 2: Deletion by ~~copying~~- the following is done

- Copy the node containing the ~~largest~~ element in the left (or the smallest element in the right) to the node containing the element to be deleted
- Delete the copied node

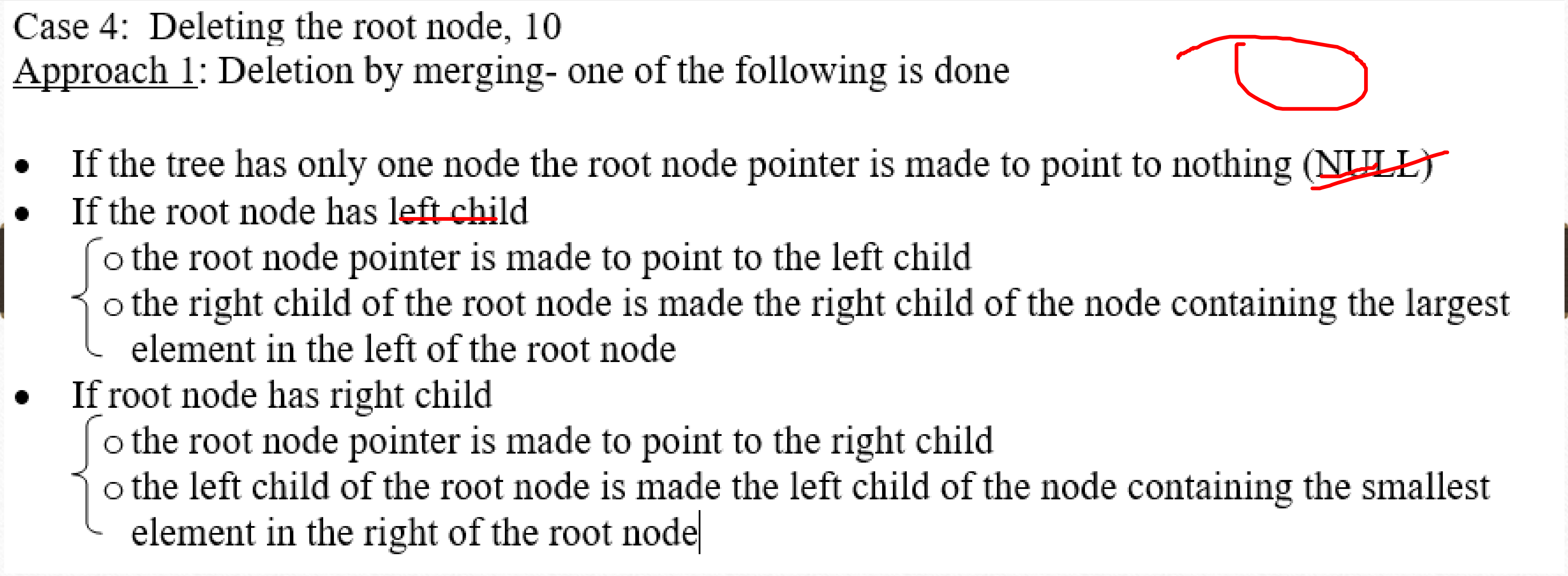


…cont’d


RootNodePtr
10
6
14
3
8
12
18
2
4
7
9
11
13
16
19
1
5
15
17


RootNodePtr
10
7
14
3
8
12
18
2
4
9
11
13
16
19
1
5
15
17

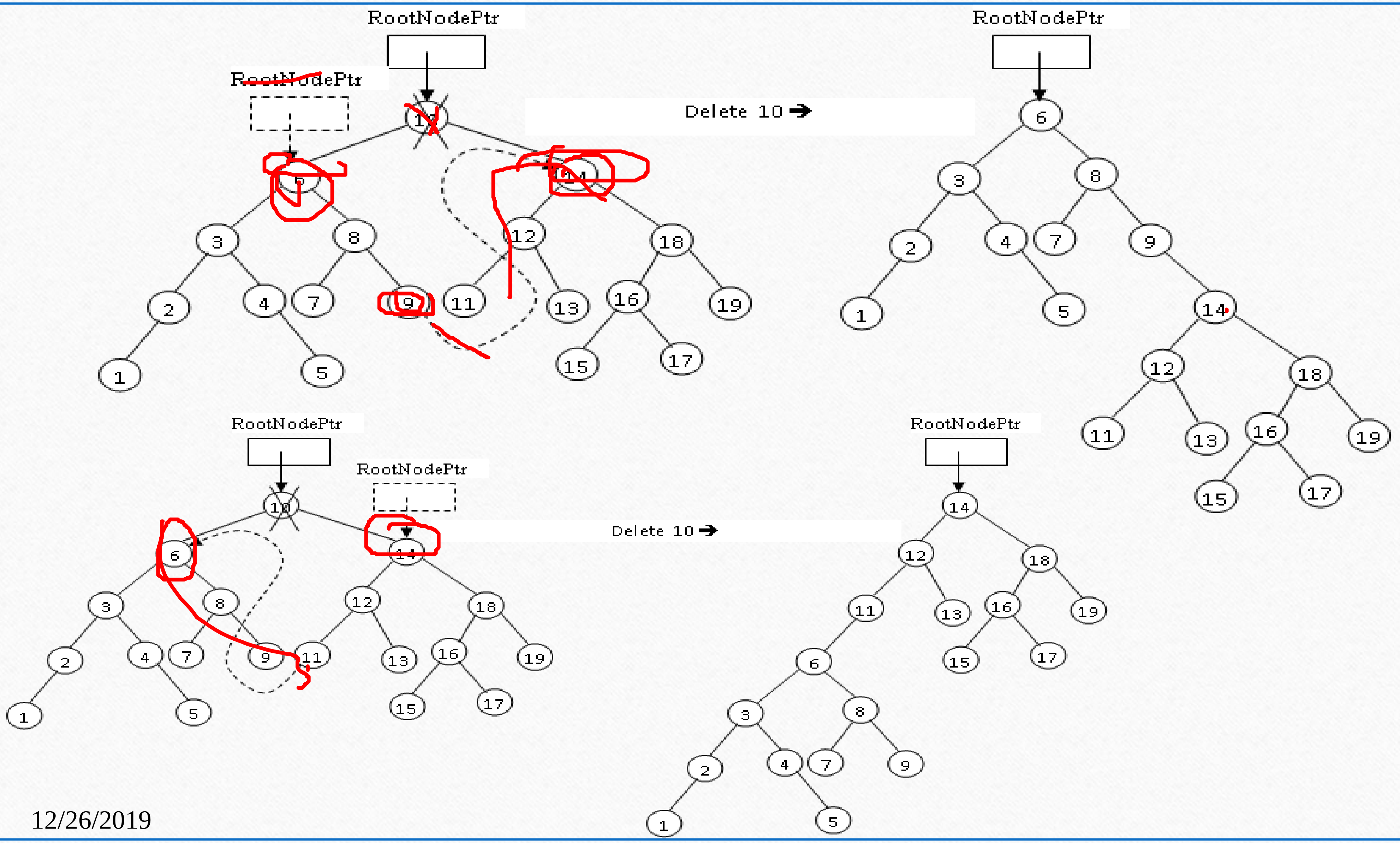
…cont'd



Case 4: Deleting the root node, 10

Approach 1: Deletion by merging- one of the following is done

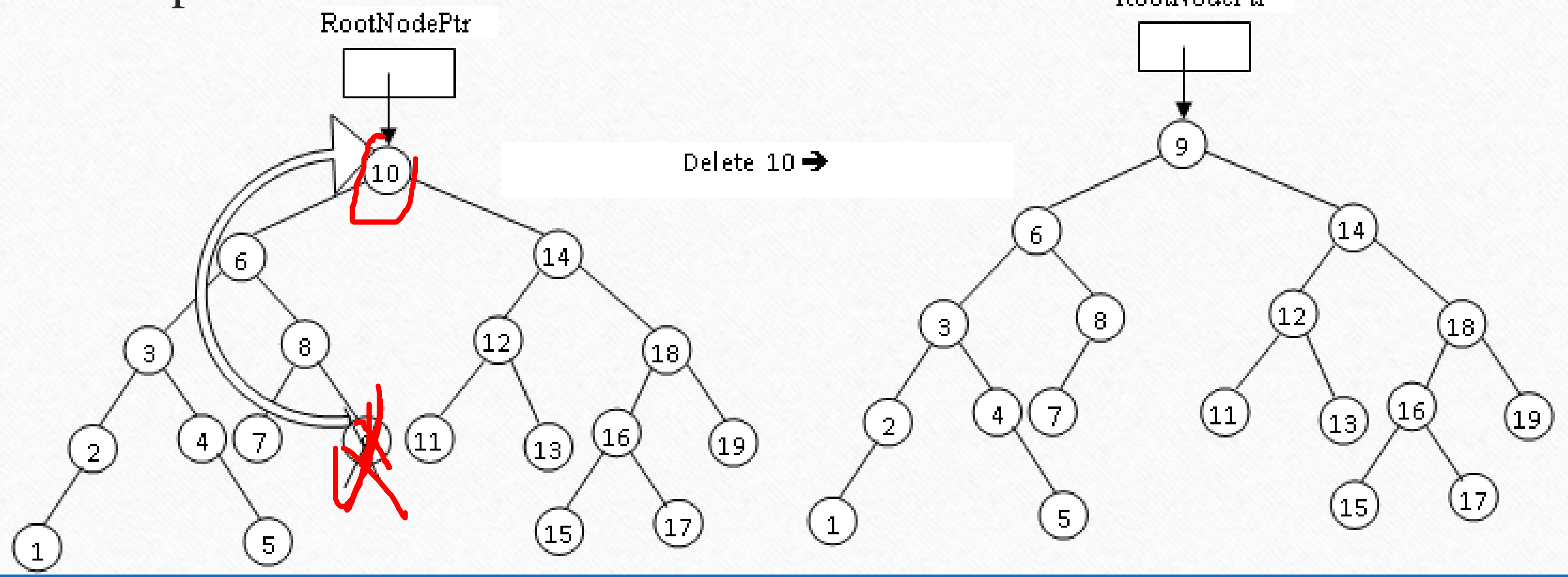
- If the tree has only one node the root node pointer is made to point to nothing (~~NULL~~)
- If the root node has ~~left child~~
  - the root node pointer is made to point to the left child
  - the right child of the root node is made the right child of the node containing the largest element in the left of the root node
- If root node has right child
  - the root node pointer is made to point to the right child
  - the left child of the root node is made the left child of the node containing the smallest element in the right of the root node

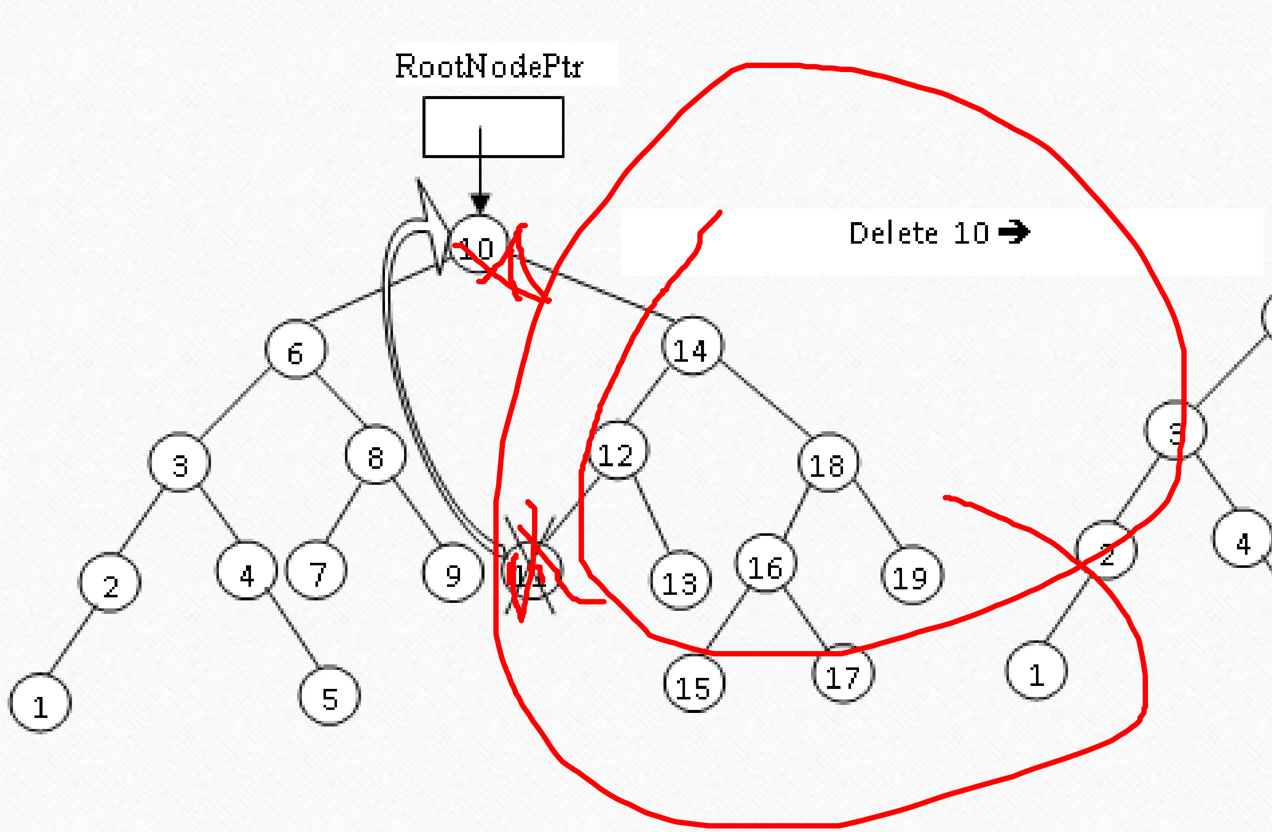

RootNodePtr
~~RootNodePtr~~
Delete 10 ➔
RootNodePtr
RootNodePtr
RootNodePtr
Delete 10 ➔
RootNodePtr

…cont'd

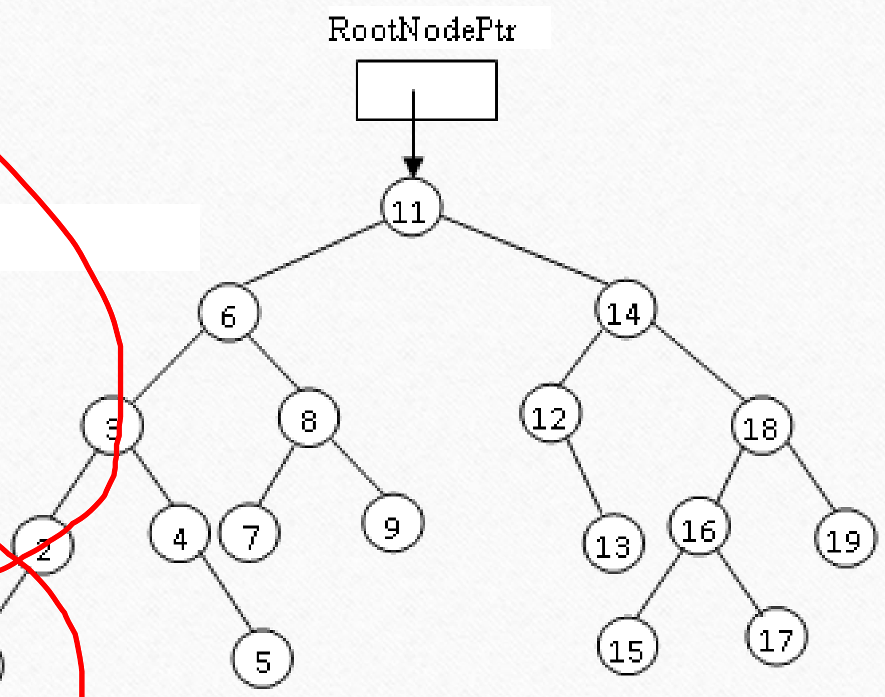
✓ Approach 2: Deletion by copying- the following is done

- Copy the node containing the largest element in the left (or the smallest element in the right) to the node containing the element to be deleted
- Delete the copied node




RootNodePtr
10
6
14
3
8
12
18
2
4
7
9
11
13
16
19
1
5
15
17

Delete 10 ➔


RootNodePtr
11
6
14
3
8
12
18
2
4
7
9
13
16
19
1
5
15
17

…cont'd

Function call:

```
if ((RootNodePtr->Left==NULL)&&( RootNodePtr->Right==NULL) && (RootNodePtr->Num==N))
{    // the node to be deleted is the root node having no child
RootNodePtr=NULL;
delete RootNodePtr;
}
else
DeleteBST(RootNodePtr, RootNodePtr, N);
```

Implementation: (Deletion by copying)

```
void DeleteBST(Node *RNP, Node *PDNP, int x)
{
        Node *DNP; // a pointer that points to the currently deleted node
        // PDNP is a pointer that points to the parent node of currently deleted node
        if(RNP==NULL)
                cout<<"Data not found\n";
        else if (RNP->Num>x)
                DeleteBST(RNP->Left, RNP, x);// delete the element in the left subtree
        else if(RNP->Num<x)
                DeleteBST(RNP->Right, RNP, x);// delete the element in the right subtree
        else
        {
                DNP=RNP;
```

…cont’d

```
if((DNP->Left==NULL) && (DNP->Right==NULL))
{
        if (PDNP->Left==DNP)
                PDNP->Left=NULL;
        else
                PDNP->Right=NULL;
        delete DNP;
}
else
{
  if(DNP->Left!=NULL) //find the maximum in the left
        {
                PDNP=DNP;
                DNP=DNP->Left;
                while(DNP->Right!=NULL)
                {
                        PDNP=DNP;
                        DNP=DNP->Right;
                }
                RNP->Num=DNP->Num;
                DeleteBST(DNP,PDNP,DNP->Num);
        }
```

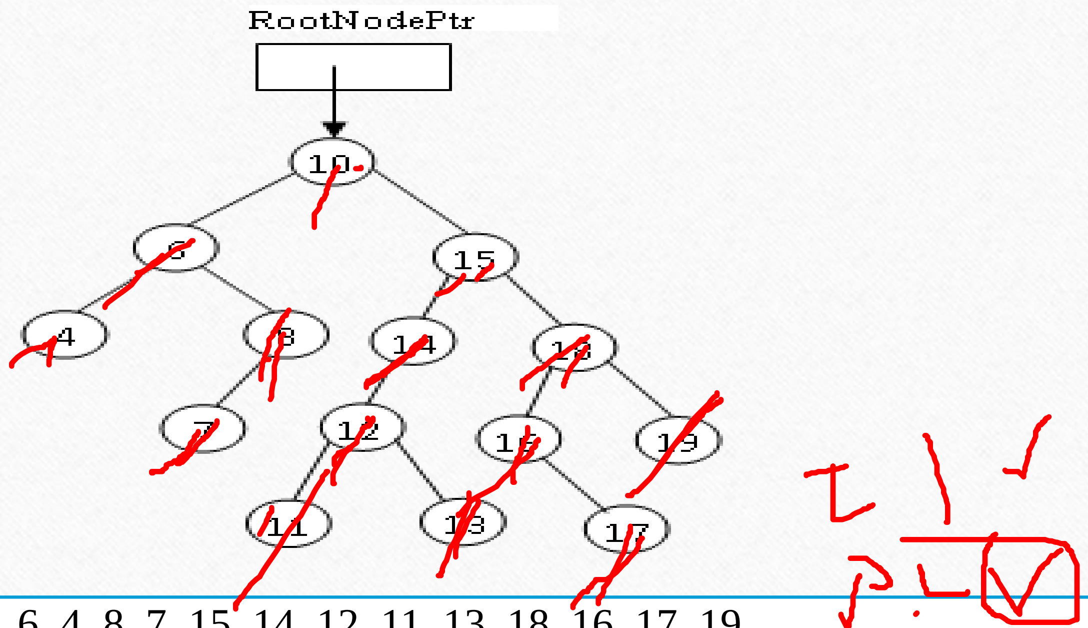
```
                        else //find the minimum in the right
                        {
                                PDNP=DNP;
                                DNP=DNP->Right;
                                while(DNP->Left!=NULL)
                                {
                                        PDNP=DNP;
                                        DNP=DNP->Left;
                                }
                                RNP->Num=DNP->Num;
                                DeleteBST(DNP,PDNP,DNP->Num);
                        }
                }
        }
}
```

Traversing

✓ There are mainly three types of binary tree traversals techniques.

1. **Preorder traversal** - traversing binary tree in the order of parent, left and right.
2. **Inorder traversal** - traversing binary tree in the order of left, parent and right.
3. **Postorder traversal** - traversing binary tree in the order of left, right and parent.

Example



- ✓ **Preorder traversal-** 10, 6, 4, 8, 7, 15, 14, 12, 11, 13, 18, 16, 17, 19
- ✓ **Inorder traversal-** 4, 6, 7, 8, 10, 11, 12, 13, 14, 15, 16, 17, 18, 19

  ==> Used to display nodes in ascending order.
- ✓ **Postorder traversal-** 4, 7, 8, 6, 11, 13, 12, 14, 17, 16, 19, 18, 15, 10

…cont'd

Function calls:

```
        Preorder(RootNodePtr);
        Inorder(RootNodePtr);
        Postorder(RootNodePtr);
```

```
void Preorder (Node *CurrNodePtr)
{
        if(CurrNodePtr ! = NULL)
        {
                cout<< CurrNodePtr->Num;
                   // or any operation on the node
                Preorder(CurrNodePtr->Left);
                Preorder(CurrNodePtr->Right);
        }
}
```

```
void Inorder (Node *CurrNodePtr)
{
        if(CurrNodePtr ! = NULL)
        {
                Inorder(CurrNodePtr->Left);
                cout<< CurrNodePtr->Num;
                        // or any operation on the node
                Inorder(CurrNodePtr->Right);
        }
}
```

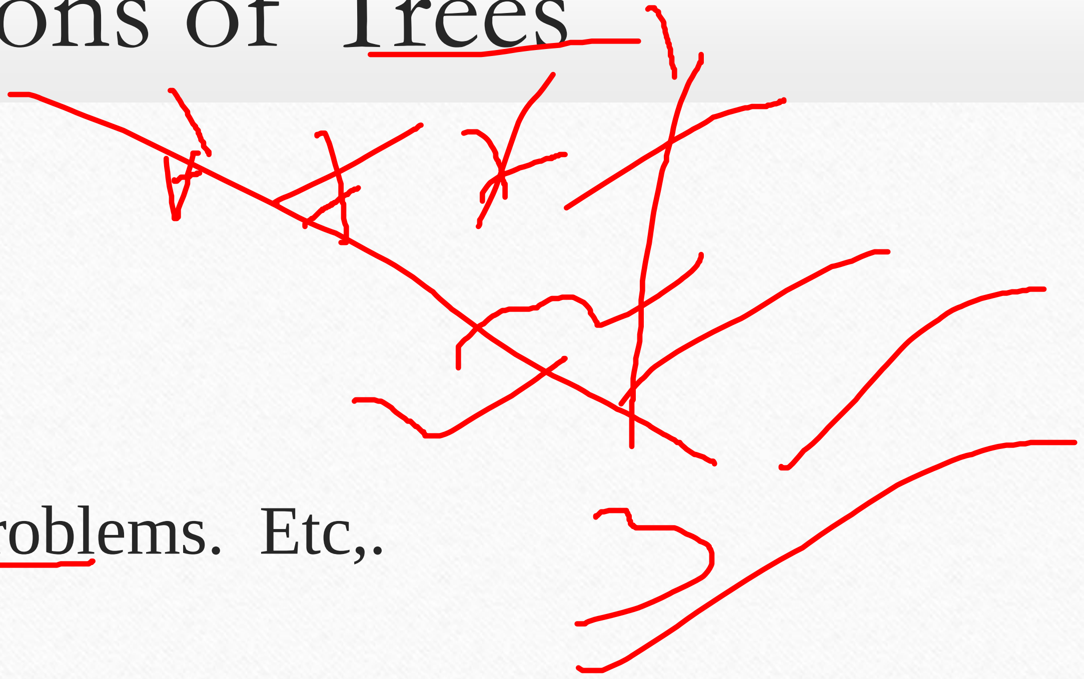
```
void Postorder (Node *CurrNodePtr)
{
        if(CurrNodePtr ! = NULL)
        {
                Postorder(CurrNodePtr->Left);
                Postorder(CurrNodePtr->Right);
                cout<< CurrNodePtr->Num;
                // or any operation on the node
        }
}
```

Application of binary tree traversal

- ✓ Store values on leaf nodes and operators on internal nodes
- ✓ Preorder traversal- used to generate mathematical expression in prefix notation
- ✓ Inorder traversal- used to generate mathematical expression in infix notation.
- ✓ Postorder traversal- used to generate mathematical expression in postfix notation

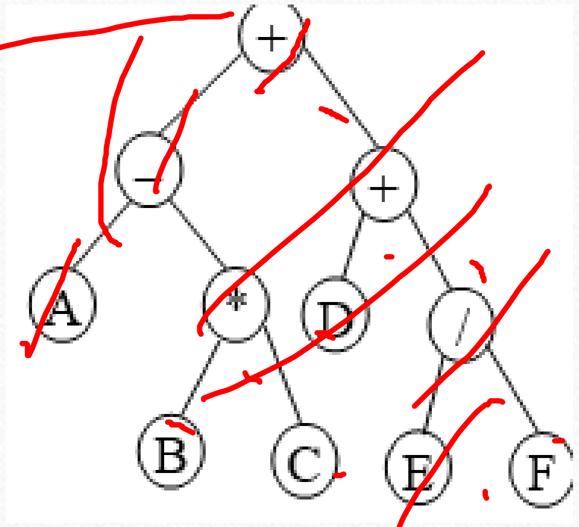
Applications of Trees

- ✓File system maintenance
- ✓Expression evaluation
- ✓Compiler design
- ✓To implement the shortest path problems. Etc,.



Example

- ✓Preorder traversal - + – A * B C + D / E F →Prefix notation
- ✓Inorder traversal - A – B * C + D + E / F → Infix notation
- ✓Postorder traversal - A B C * – D E F / + + → Postfix notation

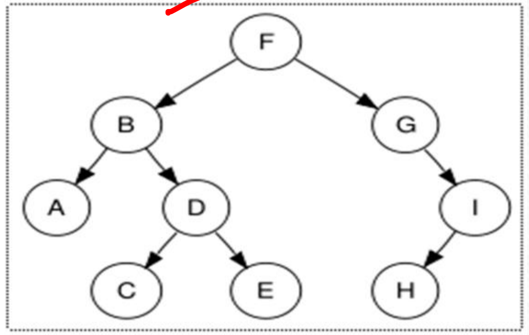


Review Questions

1. Explain trees in data structure?
2. What are the applications of tree?
3. Describe the difference between BST and binary tree.
4. Implement different operations of BST.
5. Create a binary search tree with the following values.
   - 10, 8, 15, 25, 22, 30
6. Create a binary search tree with the following values.
   - 14, 4, 15, 3, 9, 18, 16, 20, 7, 5, 17

Review Questions

7. Write the preorder, Postorder and Inorder traversal of the following tree.



End of Ch.6

Questions, Ambiguities, Doubts, … ???