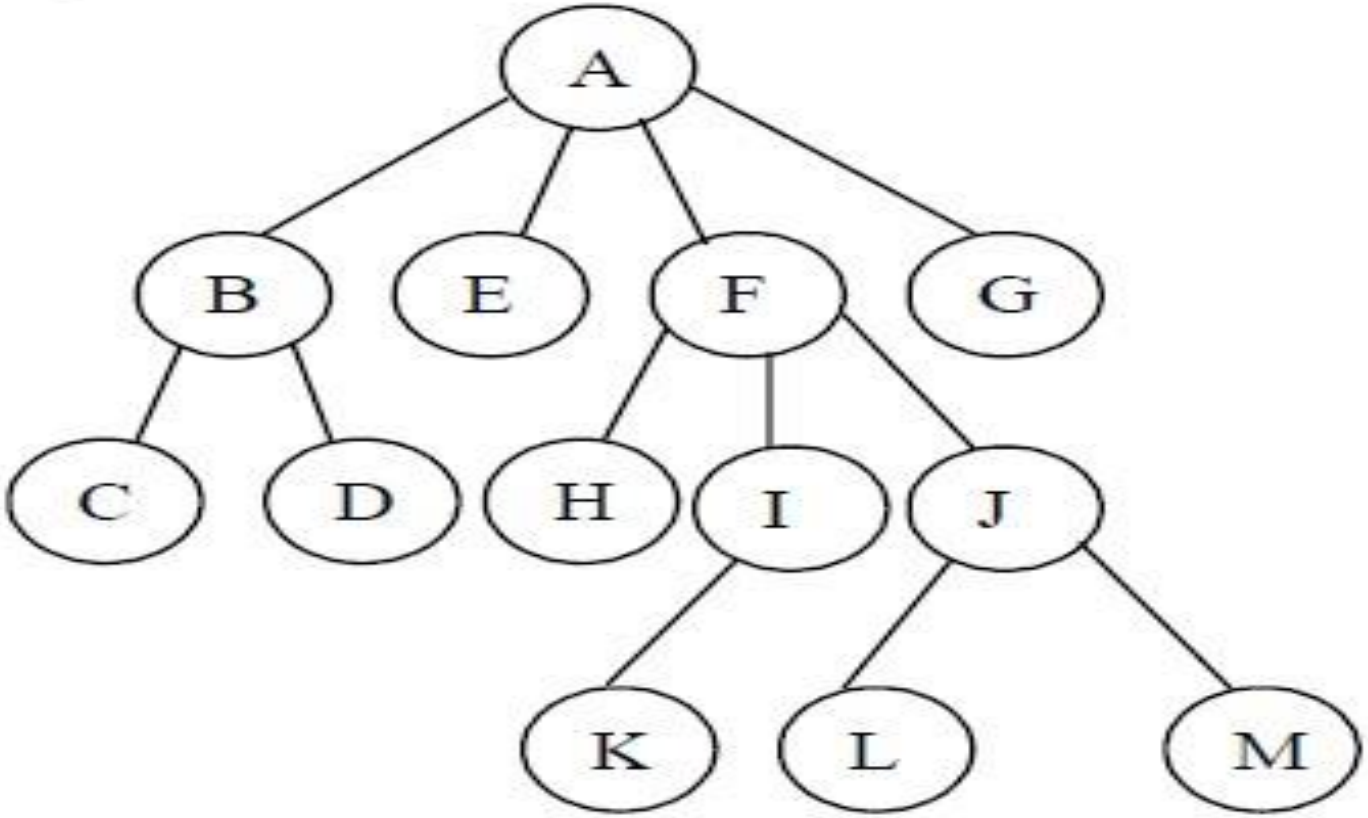
Chapter Four

Trees

- A **tree** is a set of **nodes** and **edges** that connect pairs of nodes. It is an abstract model of a hierarchical structure.
- **Rooted tree** has the following structure:
  - One node distinguished as root.
  - Every node C except the root is connected from exactly other node P. P is C's parent, and C is one of C's children.
  - There is a unique path from the root to the each node.
  - The number of edges in a path is the length of the path.

Tree Terminologies

- Consider the following tree.



Cont…

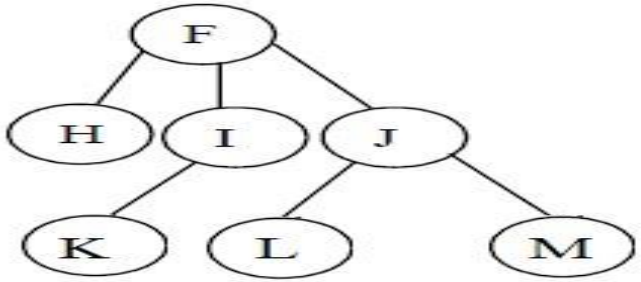
- Root: a node with out a parent. ►**A**
- Internal node: a node with at least one child.

  ►**A, B, F, I, J**
- External (leaf) node: a node without a child.

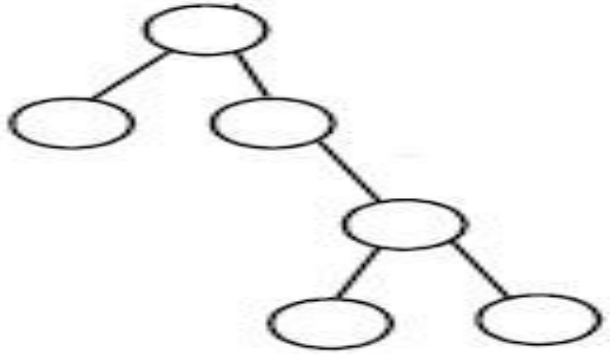
  ► **C, D, E, H, K, L, M, G**
- **Ancestors of a node**: parent, grandparent, grandgrandparent, etc of a node. Ancestors of **K-> A, F, I**
- Descendants of a node: children, grandchildren, grandgrandchildren etc of a node. Descendants of **F** -> **H, I, J, K, L, M**
- Depth of a node: number of ancestors or length of the path from the root to the node. Depth of **H** -> **2**
- **Height of a tree**: depth of the deepest node. ->**3**

Cont…

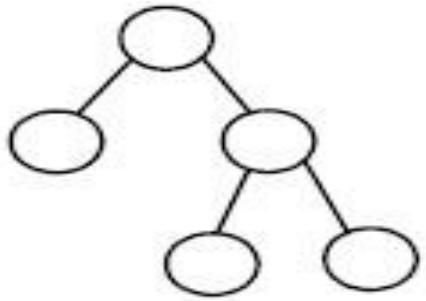
- Subtree: a tree consisting of a node and its descendants.



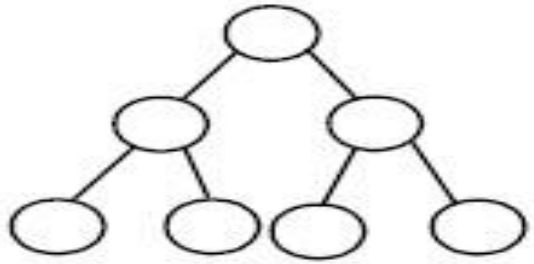
- Binary tree: a tree in which each node has at most two children called left child and right child.



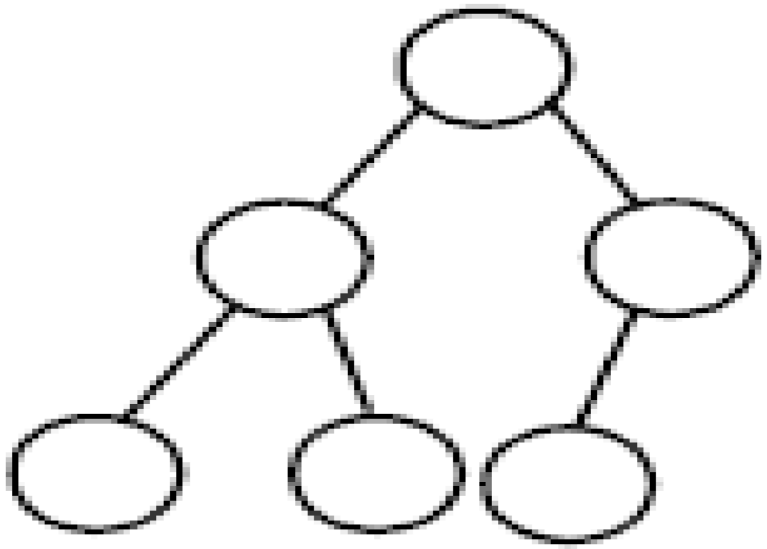
- Full binary tree: a binary tree where each node has either 0 or 2 children.



- **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.

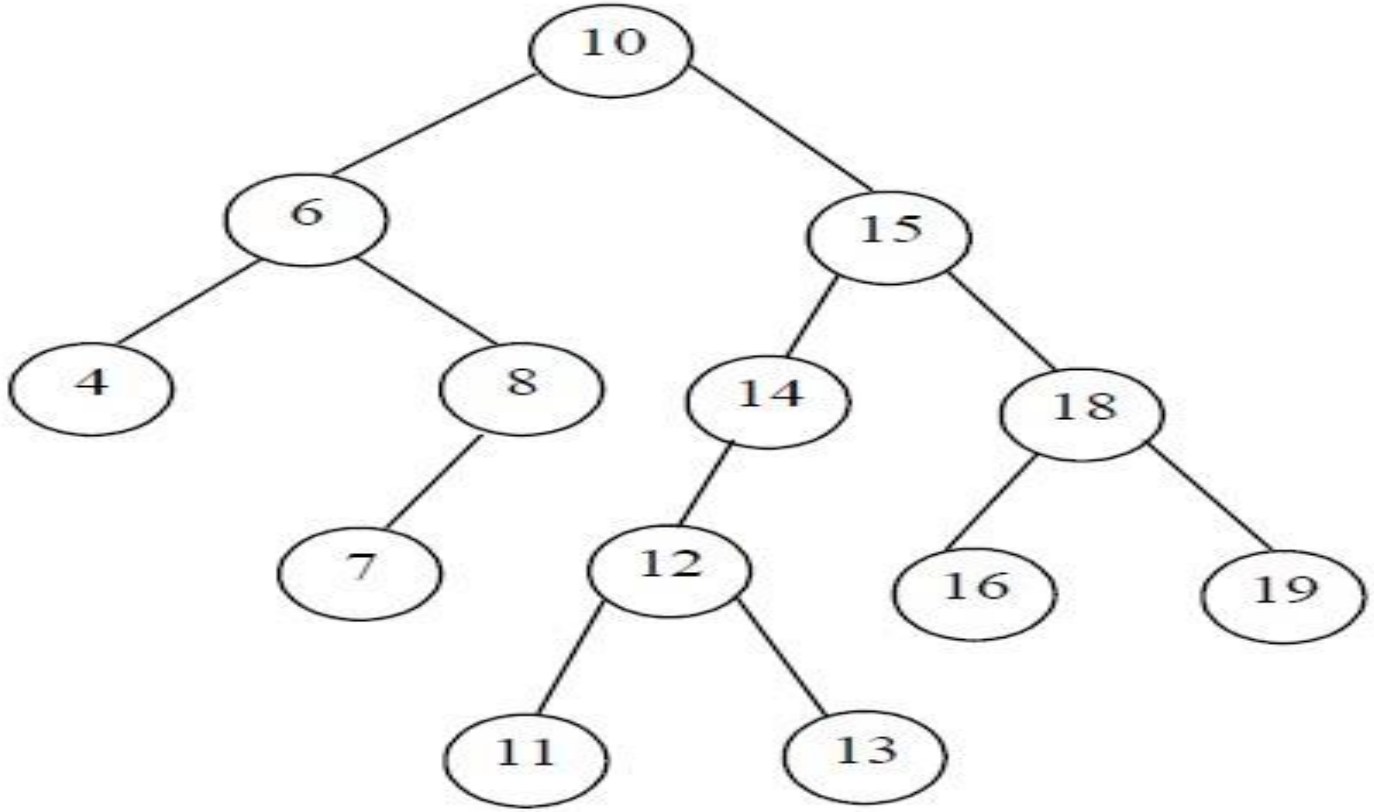


Binary search tree (ordered binary tree)

- a binary tree that may be empty, but if it is not empty it satisfies the following.
- 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.

Cont…

- Example of **Binary search tree (ordered binary tree)**



Operations on Binary Search Tree

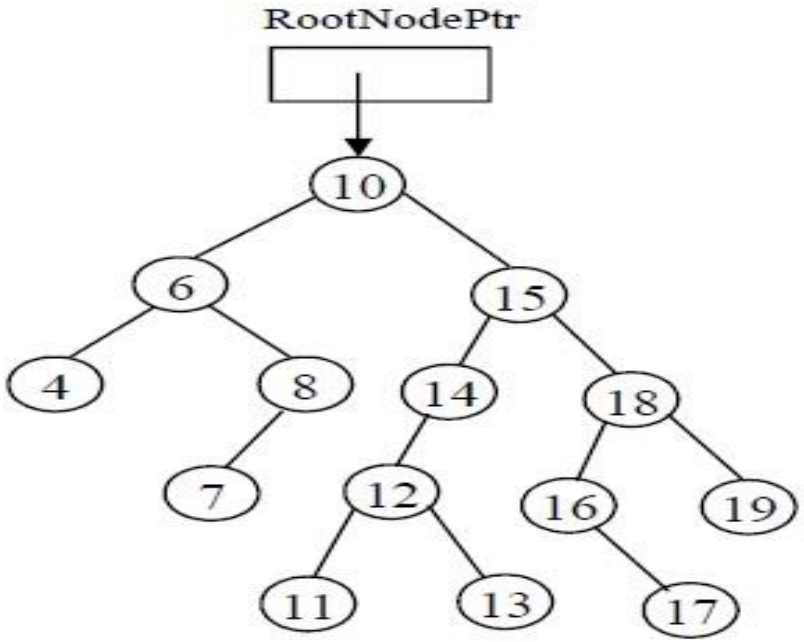
- Consider the following definition of binary search tree.

```
struct Node
  { int num;
 Node  * Left, *Right;
  } ;
 Node *RootNodePtr= NULL;
```

Traversing (breadth and depth first)

- **Binary search** tree can be traversed in three ways.
  - **Pre order traversal**: traversing binary tree in the order of ***parent, left*** *and* ***right***.
  - **Inorder traversal**: traversing binary tree in the order of ***left, parent*** *and* ***right***.
  - **Postorder traversal**: traversing binary tree in the order of ***left, right*** *and* ***parent***.

Example



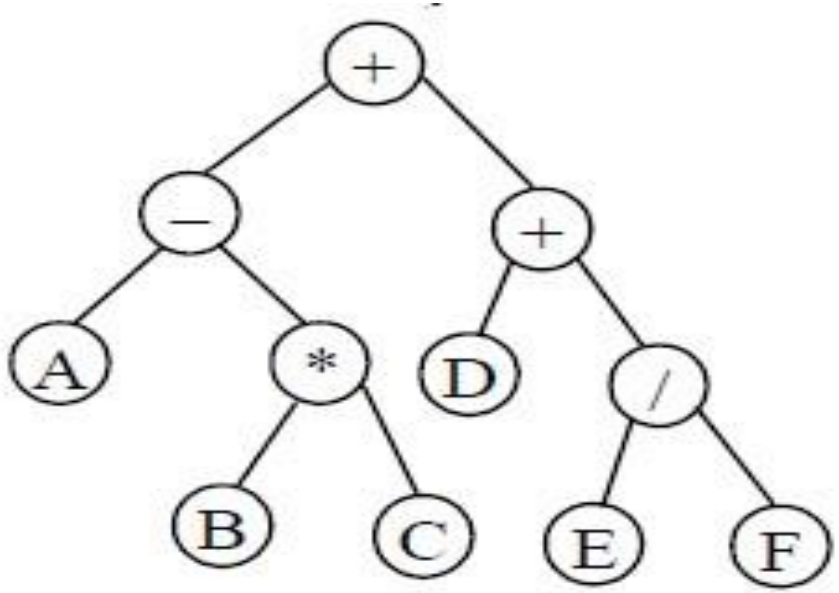
Cont…

- 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
- **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.

Cont…

- *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

Searching

- To search a node (whose num value is Number) in a binary search tree (whose root node is pointed by RootNodePtr), one of the three traversal methods can be used.

Implementation

```
bool SearchBST (Node *RNP, int x)
{ if (RNP = = NULL)
return (false);
Elseif (RNP-> num = = x )
 return(true);
Elseif (RNP->num>x)
return (SearchBST(RNP->Left, x));
else return(SearchBST(RNP-> Right, x)); }
```

OR

```
Node *SearchBST (Node *RNP, int x)
 { if((RNP = = NULL) || (RNP-> num = = x )
return(RNP);
else if(RNP-> num > x )
return(SearchBST(RNP-> Left, x));
else return(SearchBST (RNP-> Right, x)); }
```

Insertion

- When a node is inserted the definition of binary search tree should be preserved. 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 (RootNodePtr is NULL)

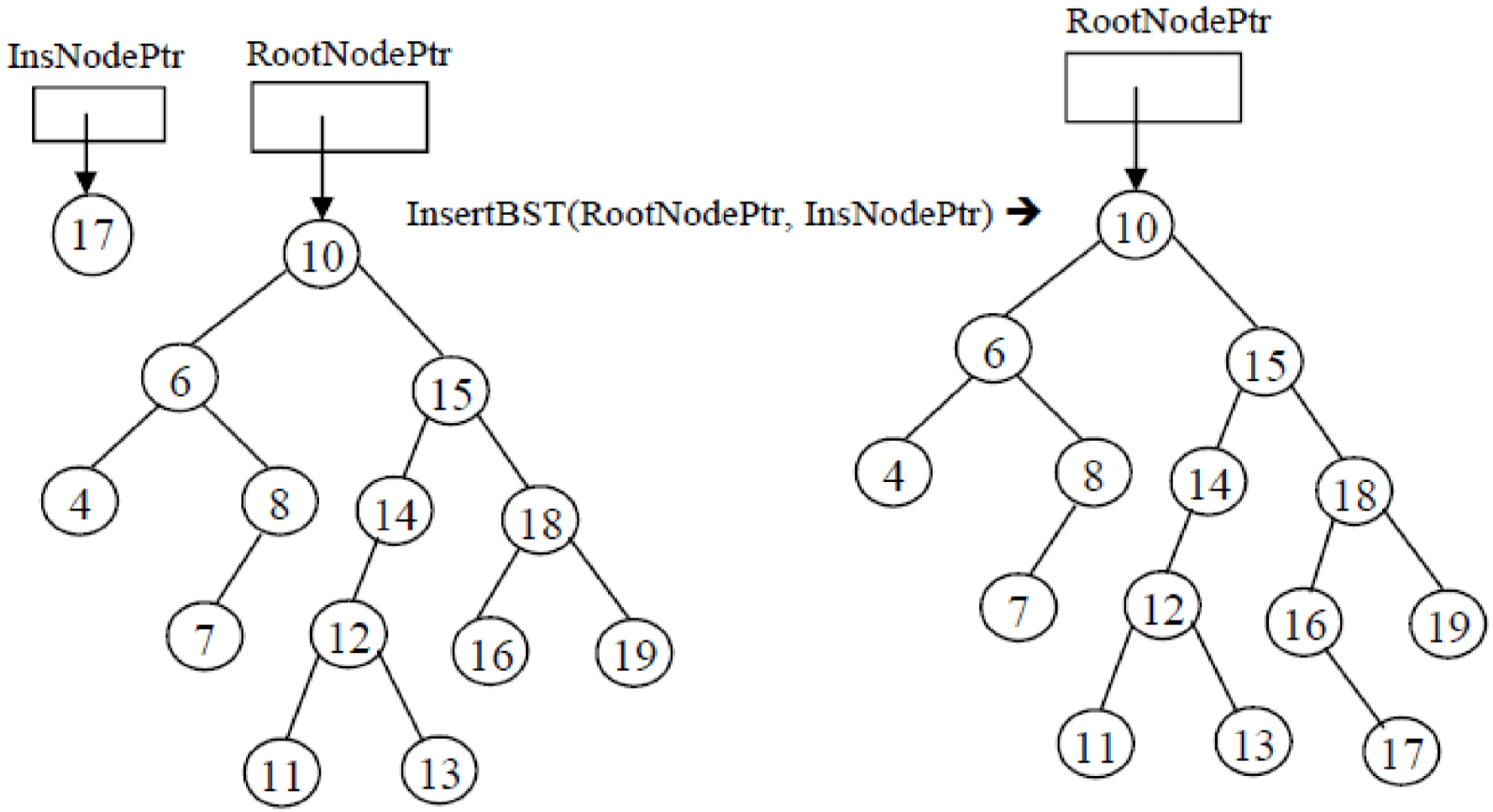
The node pointed by InsNodePtr should be made the root node.

- Case 2: There is data

Search the appropriate position.

Insert the node in that position.

Cont…


InsNodePtr
RootNodePtr
17
10
6
15
4
8
14
18
7
12
16
19
11
13
InsertBST(RootNodePtr, InsNodePtr)
RootNodePtr
10
6
15
4
8
14
18
7
12
16
19
11
13
17

Cont…

Function call:

if(RootNodePtr = = NULL)

RootNodePtr=InsNodePtr;

Else

InsertBST(RootNodePtr, InsNodePtr) ;

```
void InsertBST( Node *RNP, Node *INP)
{
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…

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

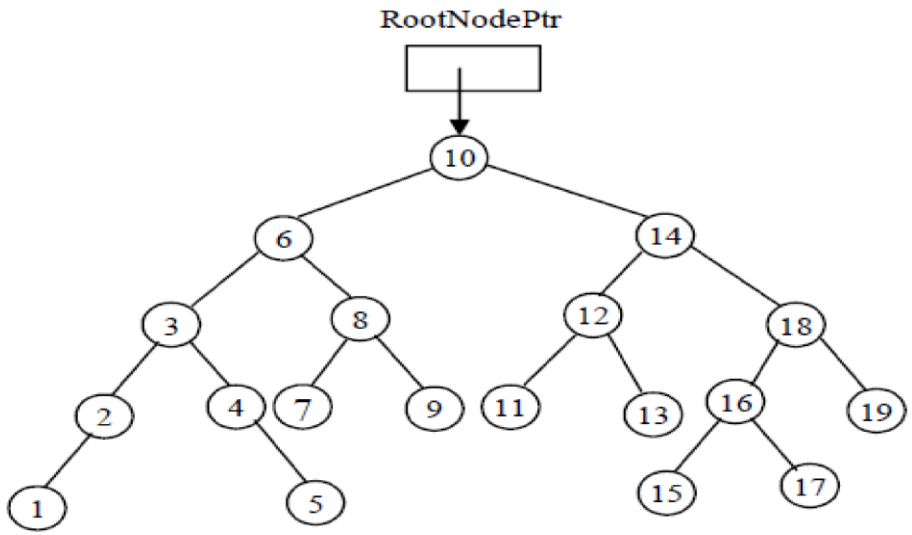
```
void InsertBST( Node *RNP, Node *INP)
{ if(RNP->num > INP-> num)
 { if(RNP-> Left==NULL )
RNP-> Left = INP; else
InsertBST(RNP-> Left, INP);
}
else {
if(RNP-> Right==NULL )
RNP-> Right = INP;
else
InsertBST(RNP-> Right, INP); }
}
```

Cont…

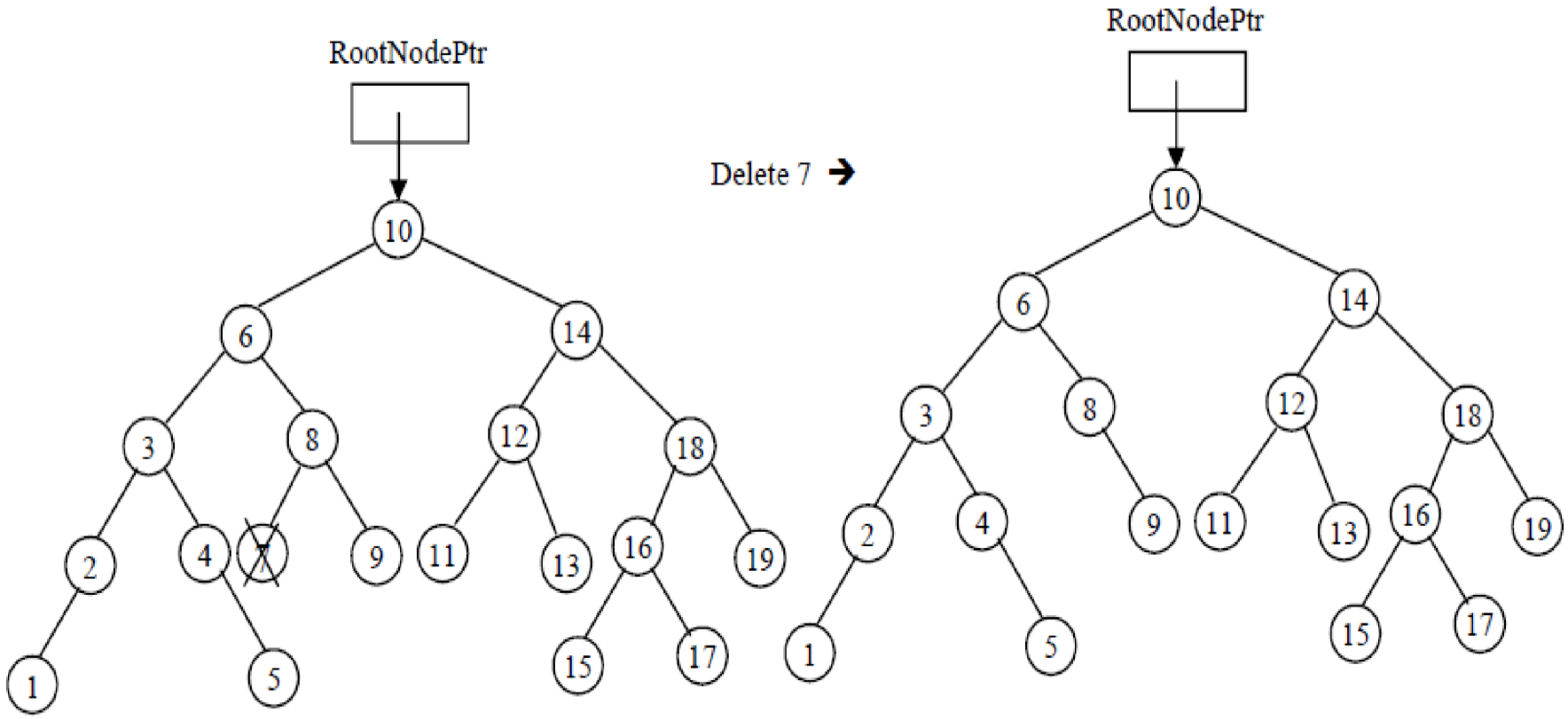
```
void InsertBST( Node *RNP, Node *INP)
{ if(RNP->num > INP-> num)
 { if(RNP-> Left==NULL )
RNP-> Left = INP;
 else
InsertBST(RNP-> Left, INP);
}
else {
if(RNP-> Right==NULL )
RNP-> Right = INP;
else
InsertBST(RNP-> Right, INP); }}
```

Deletion

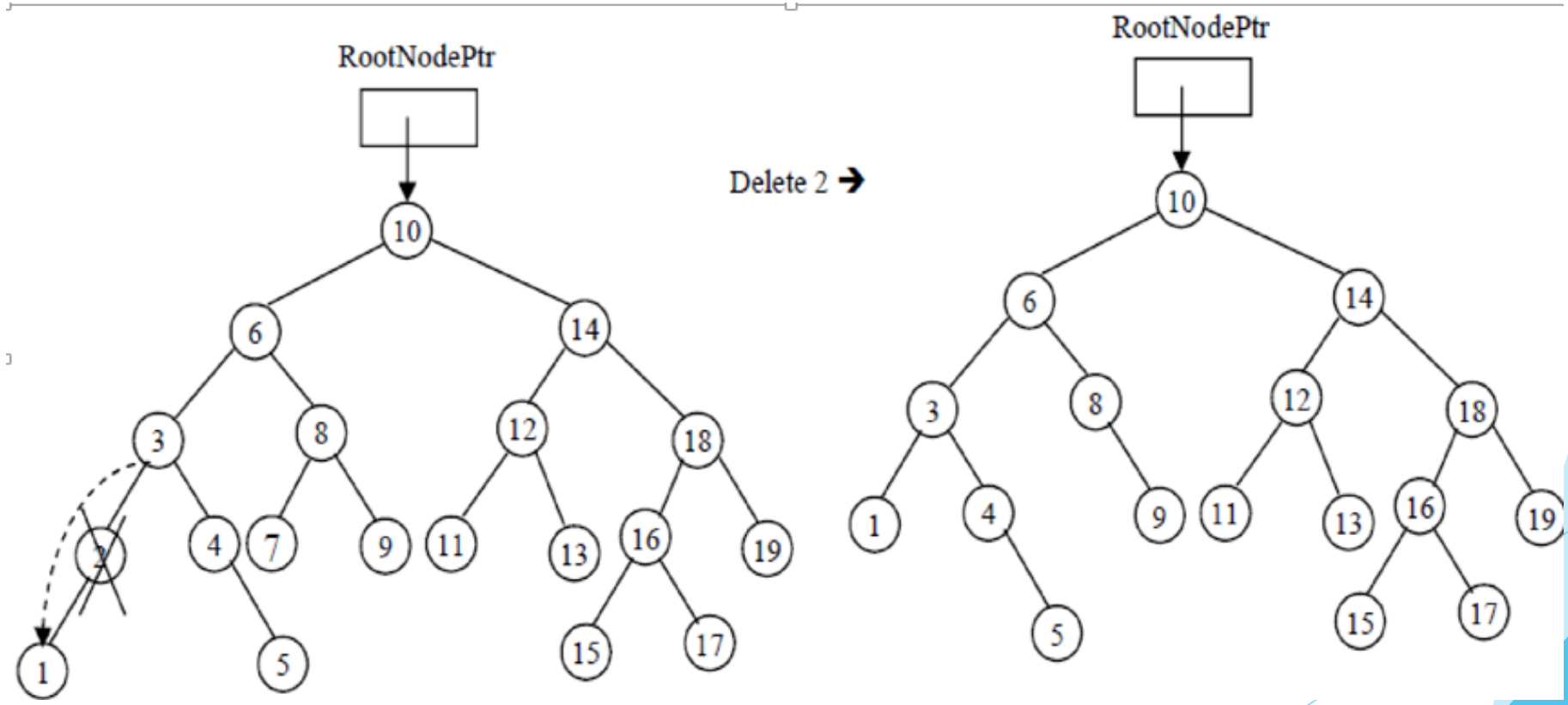
- To delete a node (whose num value is N) from binary search tree (whose root node is pointed by RNP)
- Four cases should be considered.
- When a node is deleted the definition of binary search tree should be preserved
- Consider the following binary search tree.



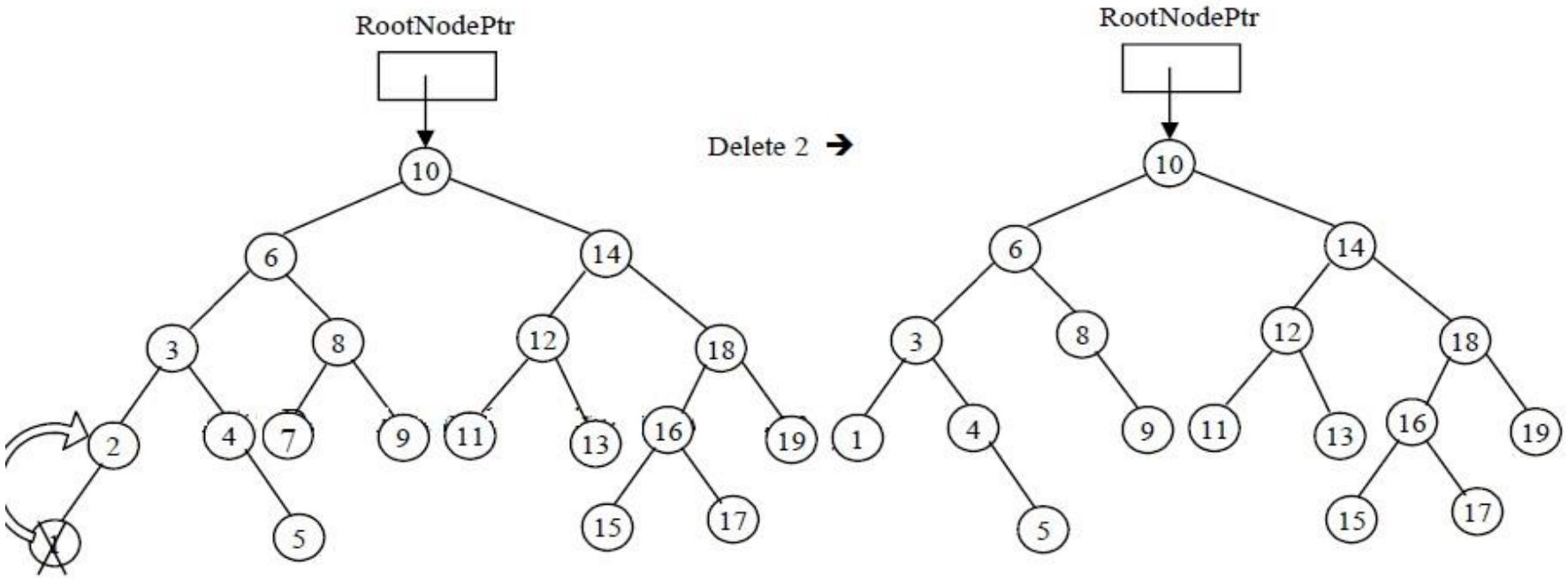
- Case 1: Deleting a leaf node (a node having no child), e.g. **7**



- Case 2: Deleting a node having only one child, e.g. **2**
- Approach 1: Deletion by merging



- Approach 2: Deletion by copying
- 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 and **delete the copied node**



- Case 3: Deleting a node having two children, e.g. **6**
- Approach 1: Deletion by merging

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

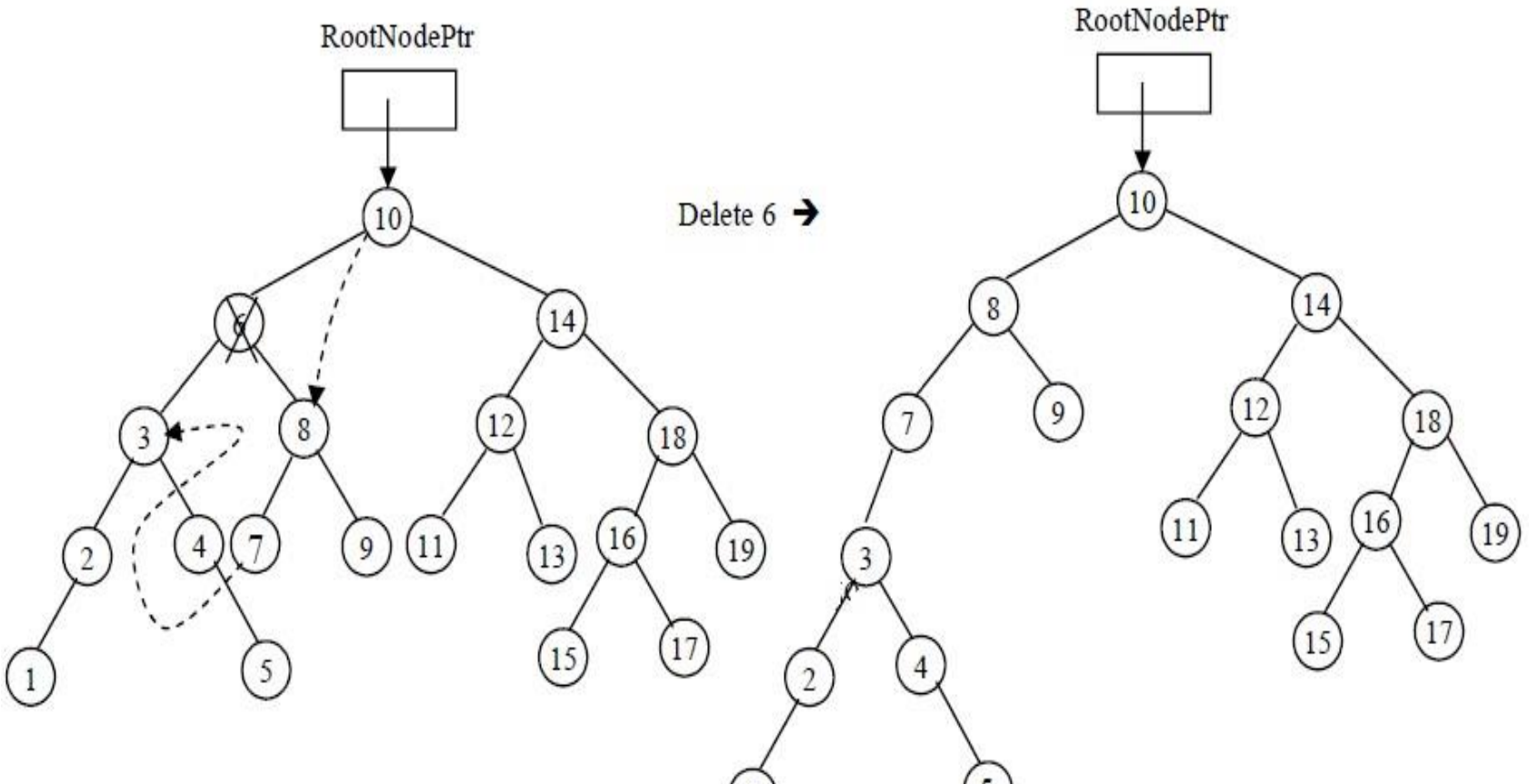
The left child of the deleted node is made the left child of the node containing smallest element in the right of the deleted node

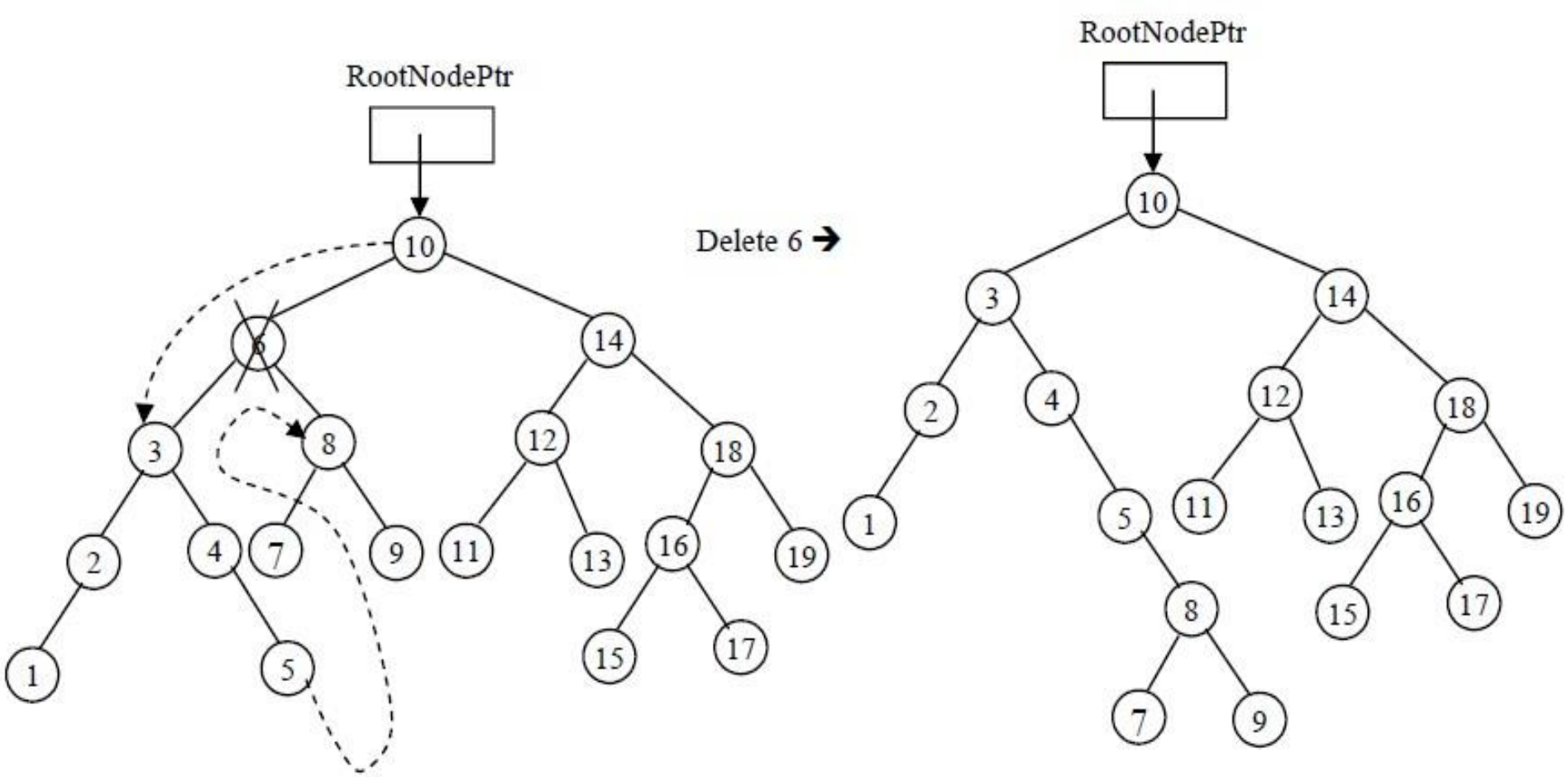
- If the deleted node is the right child of its parent, one of the following is done
- The left child of the deleted node is made the right child of the parent of the deleted node, and
- The right child of the deleted node is made the right child of the node containing largest element in the left of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

OR

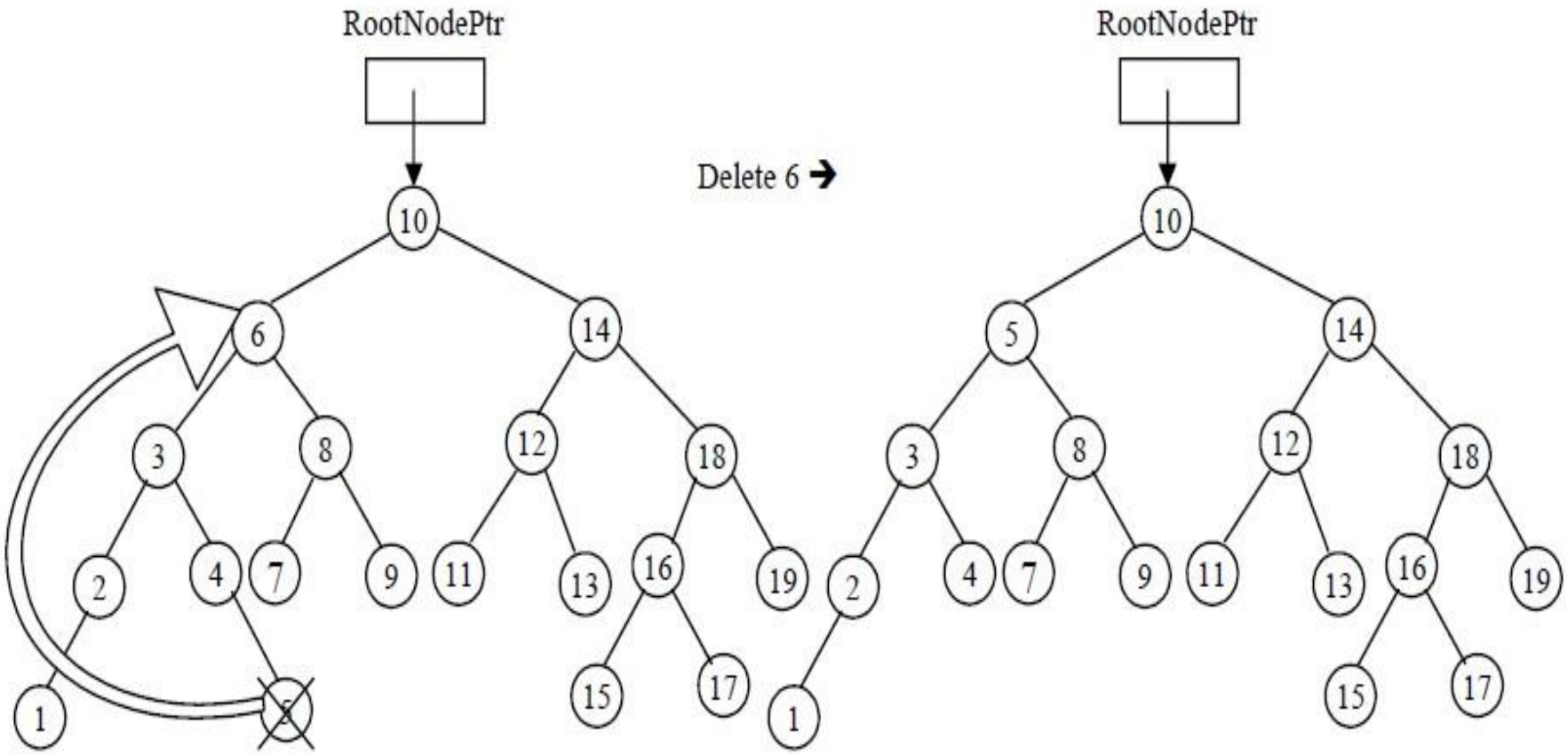
- The right child of the deleted node is made the right child of the parent of the deleted node, and

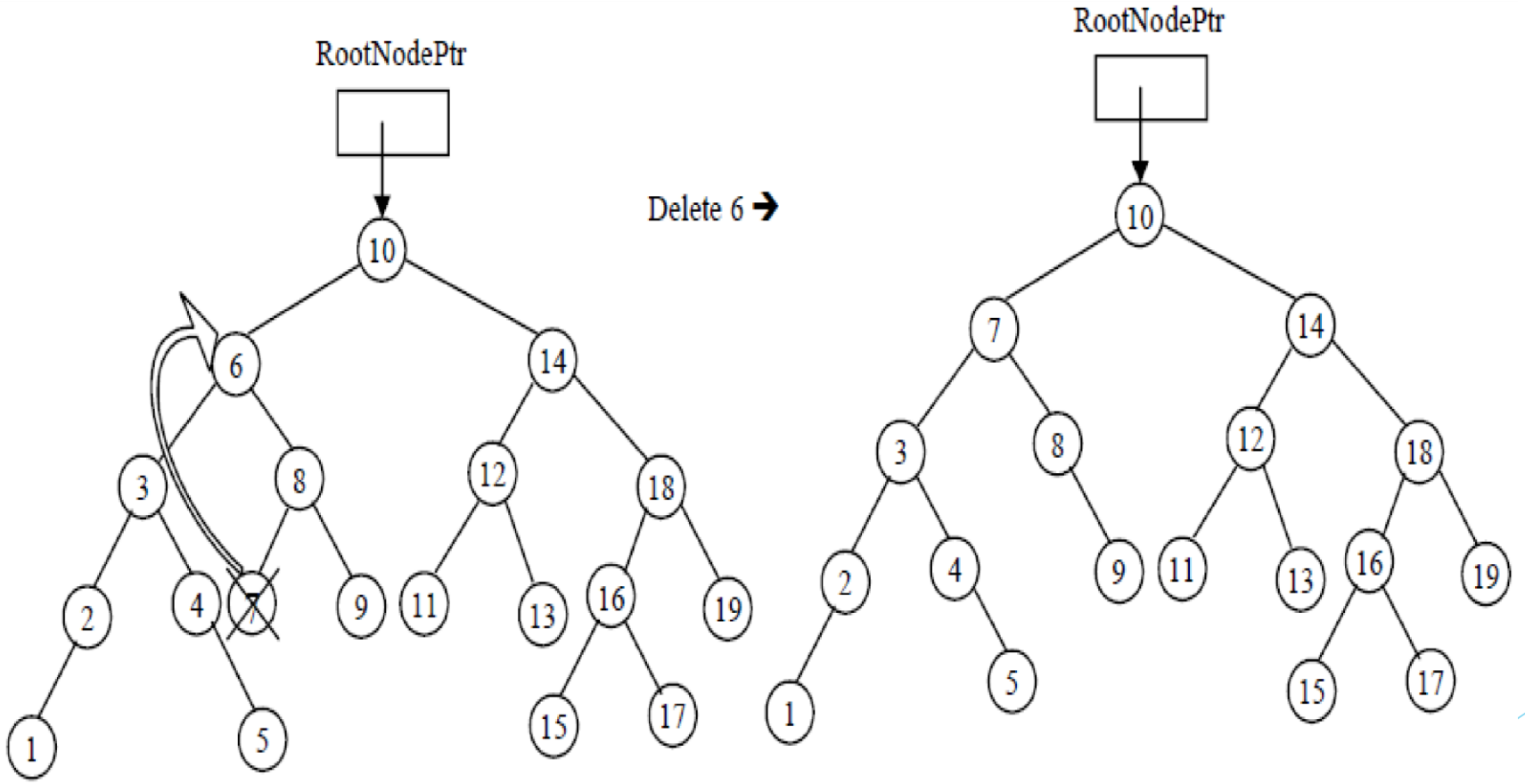
- The left child of the deleted node is made the left child of the node containing smallest element in the right of the deleted node



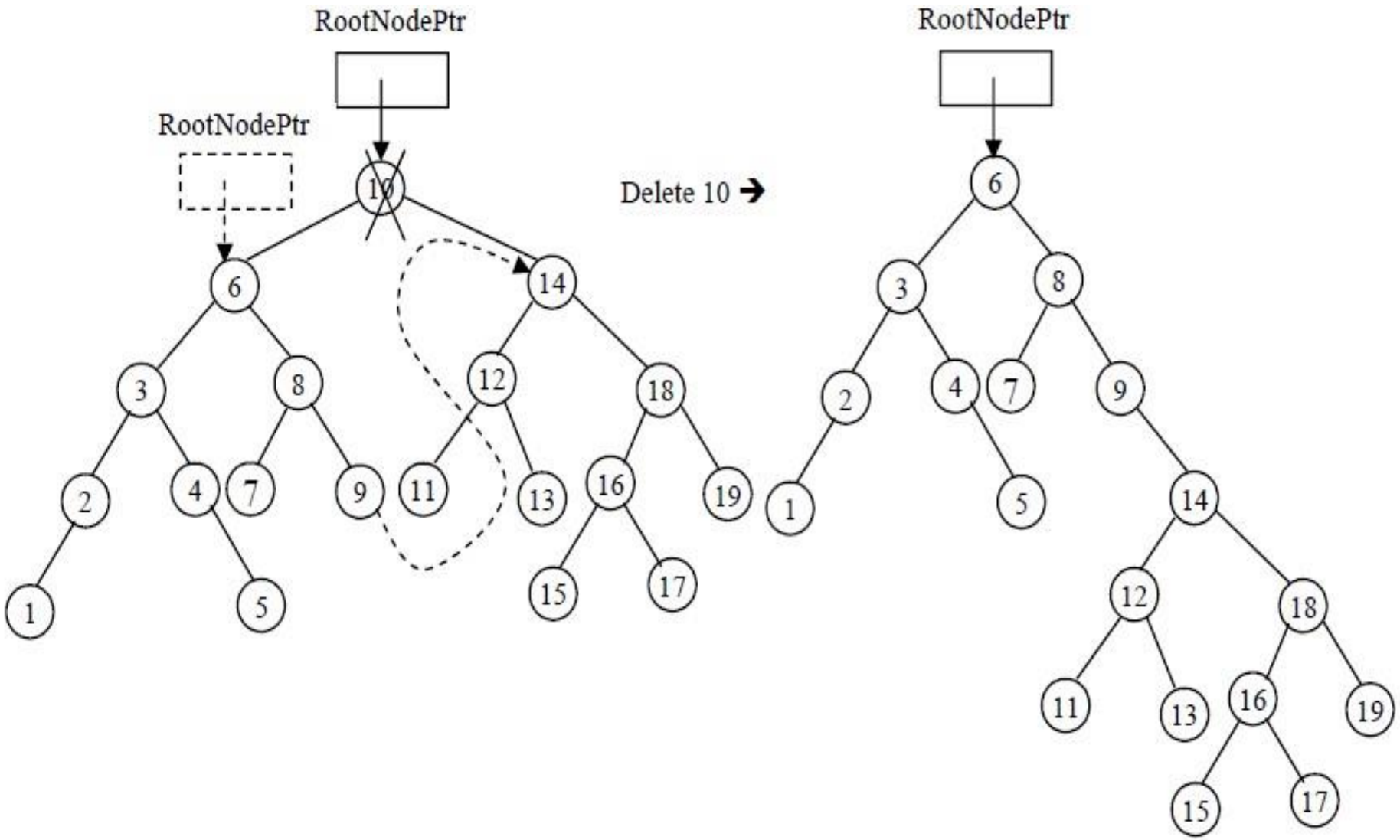

RootNodePtr
10
6
14
3
8
12
18
2
4
7
9
11
13
16
19
1
5
15
17
Delete 6 →
RootNodePtr
10
3
14
2
4
12
18
1
5
11
13
16
19
8
15
17
7
9

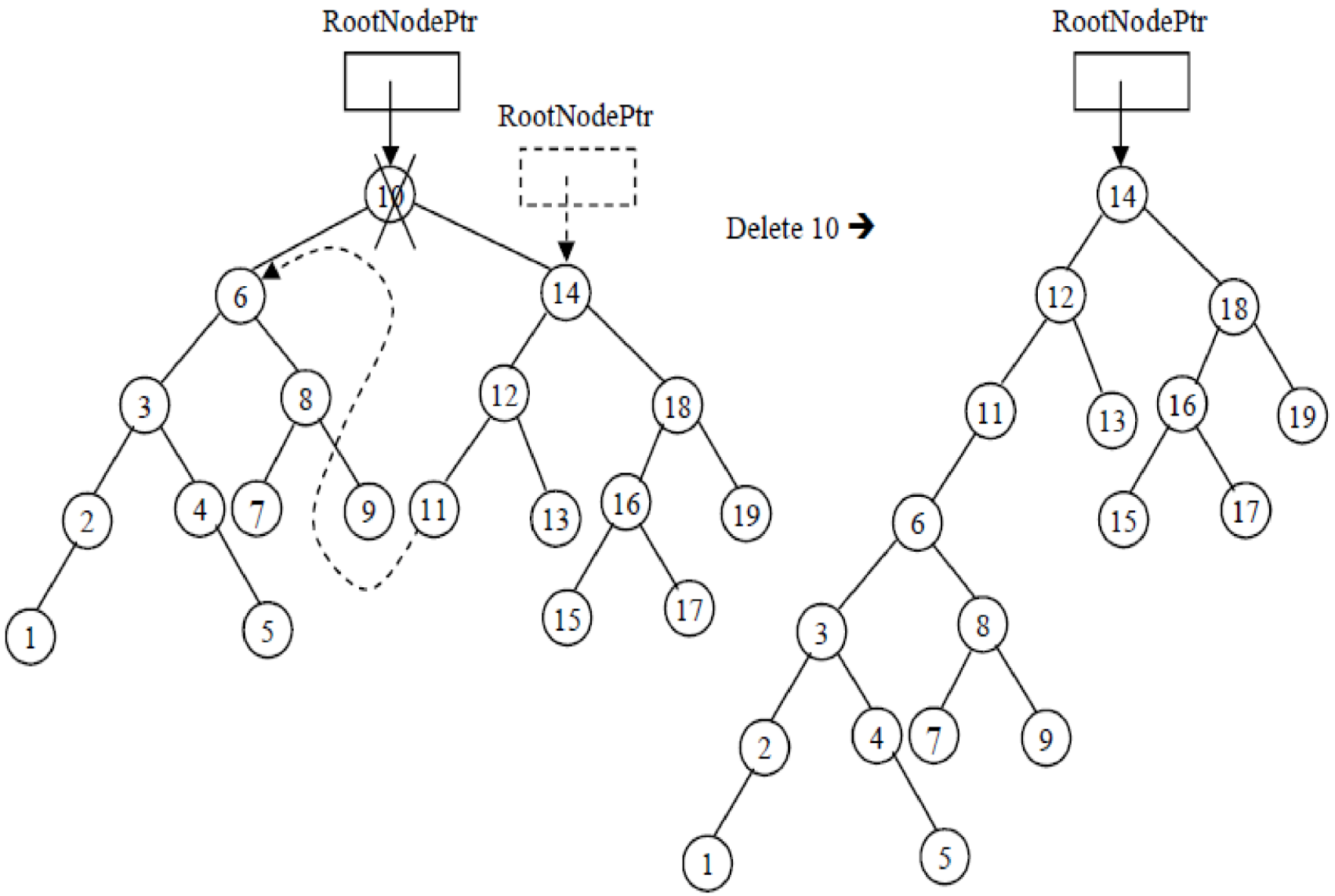
- Approach 2: Deletion by copying- the following is done
- Copy the node containing the largest element in the left (or the smallest element in the right) to the node containing the
- element to be deleted and **delete the copied node**


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


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

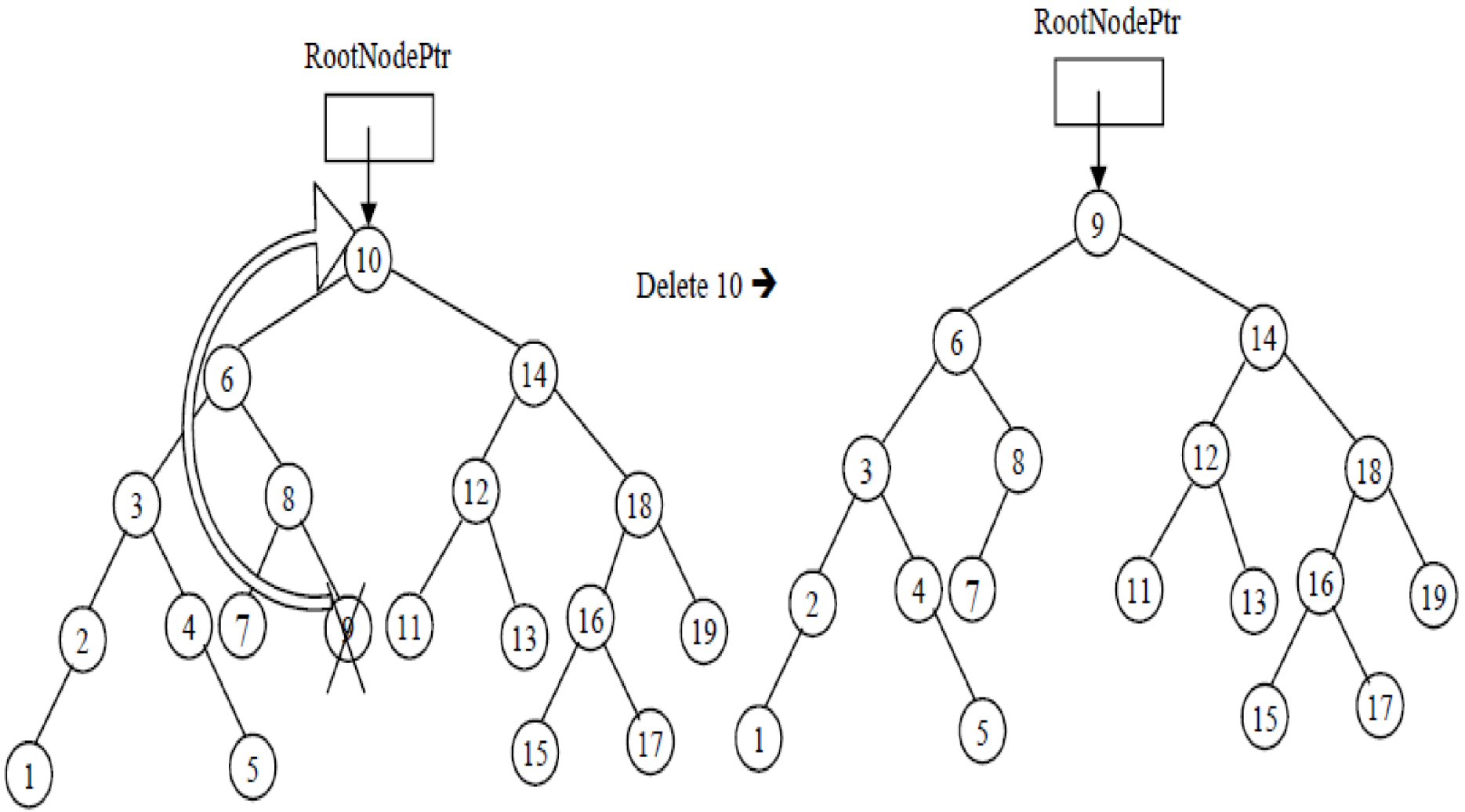
- Case 4: Deleting the root node, **10**
- Approach 1: Deletion by merging
- If the tree has only one node the root node pointer is made to point to nothing (NULL)
- If the root node has left child the root node pointer is made to point to the left child
- the right child of the root node is made the 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

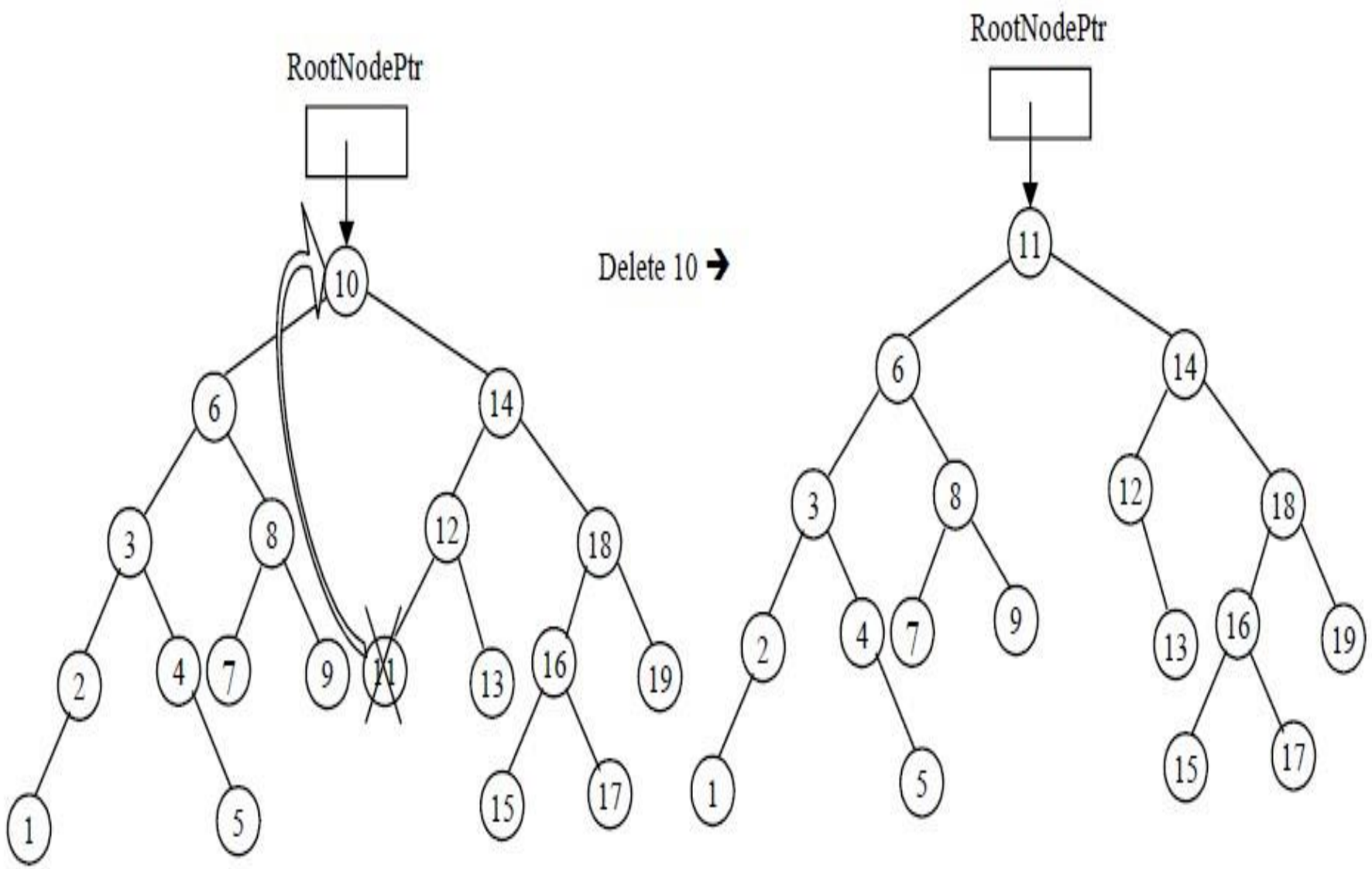

RootNodePtr
RootNodePtr
10
6
14
3
8
12
18
2
4
7
9
11
13
16
19
1
5
15
17
Delete 10 →
RootNodePtr
6
3
8
2
4
7
9
1
5
14
12
18
11
13
16
19
15
17


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

Approach 2: Deletion by copying-

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


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

```
void Delete(node* root, int data){
    if(root==NULL)
    return;
    else if(data<root->data)
    root->left=Delete(root->left);
    else if(data<root->data)
    root->right=Delete(root->right);
    else{
        //case 1:no child
        if(root->left==NULL && root->right==NULL){
            free(root);
        root=NULL;
        }

//case 2:one child
    else if(root->left==NULL){
        node* temp=root;
        root=root->right;
        free(temp);
    }
    else if(root->right==NULL){
        node* temp=root;
        root=root->left;
        free(temp);
    }
    else{
        //Case 3:two children
        node* temp=FindMin(root->right);
        root->data=temp->data;
        root->right=Delete(root->ridgt,temp->data);
    }
```