

CHAPTER 7

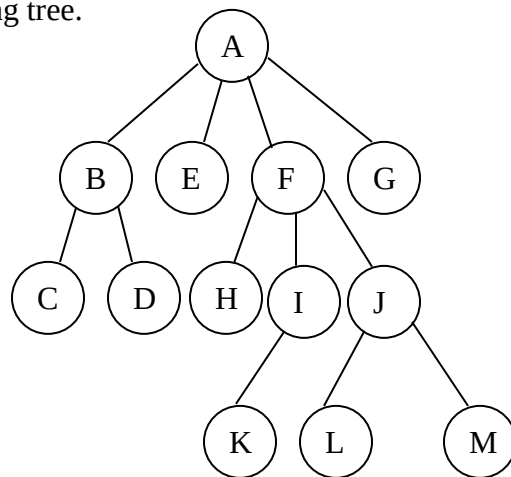
Trees

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.

6.1. Tree Terminologies

Consider the following tree.



Root: a node without 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, grand-grandparent, etc of a node.

Ancestors of K → A, F, I

Descendants of a node: children, grandchildren, grand-grandchildren etc of a node.

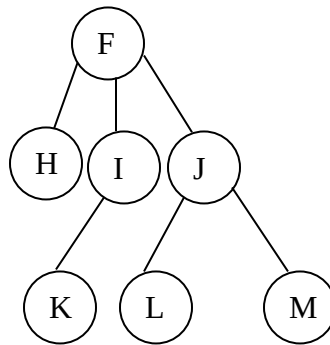
Descendants of F → H, I, J, K, L, M

Depth of a node: number of ancestors or length of the path from the root to the node.

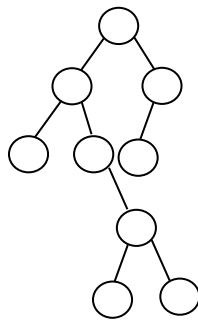
Depth of H → 2

Height of a tree: depth of the deepest node. → 3

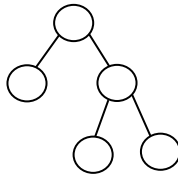
Subtree: a tree consisting of a node and its descendants.



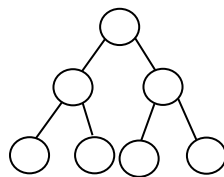
Binary tree: a tree in which each node has at most two children called left child and right child.



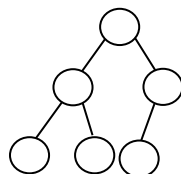
Full binary tree: a binary tree where each node has either 0 or 2 children.



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.

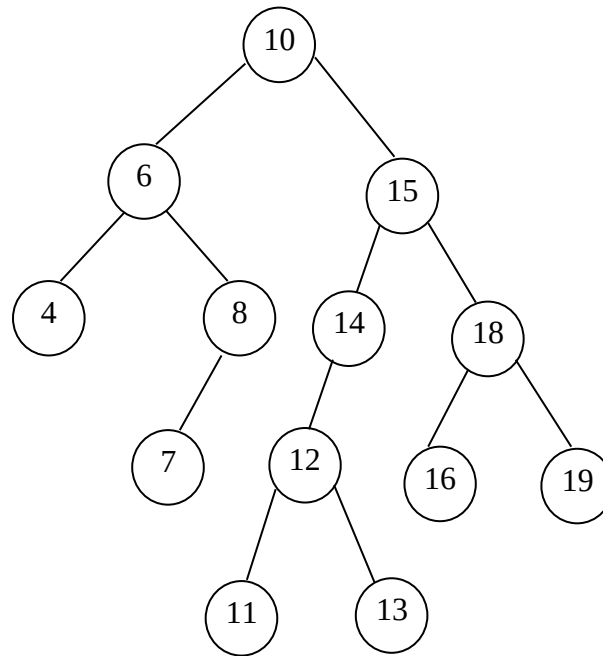


Complete binary tree: a binary tree in which the length from the root to any leaf node is either h or $h-1$ where h is the height of the tree. The deepest level should also be filled from left to right.



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.



6.2. Data Structure of a Binary Tree

```
struct DataModel
{
    Declaration of data fields
    DataModel * Left, *Right;
};
DataModel *RootDataModelPtr=NULL;
```

6.3. Operations on Binary Search Tree

Consider the following definition of binary search tree.

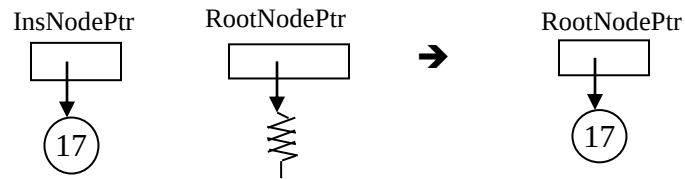
```
struct Node
{
    int Num;
    Node * Left, *Right;
};
```

Node *RootNodePtr=NULL;

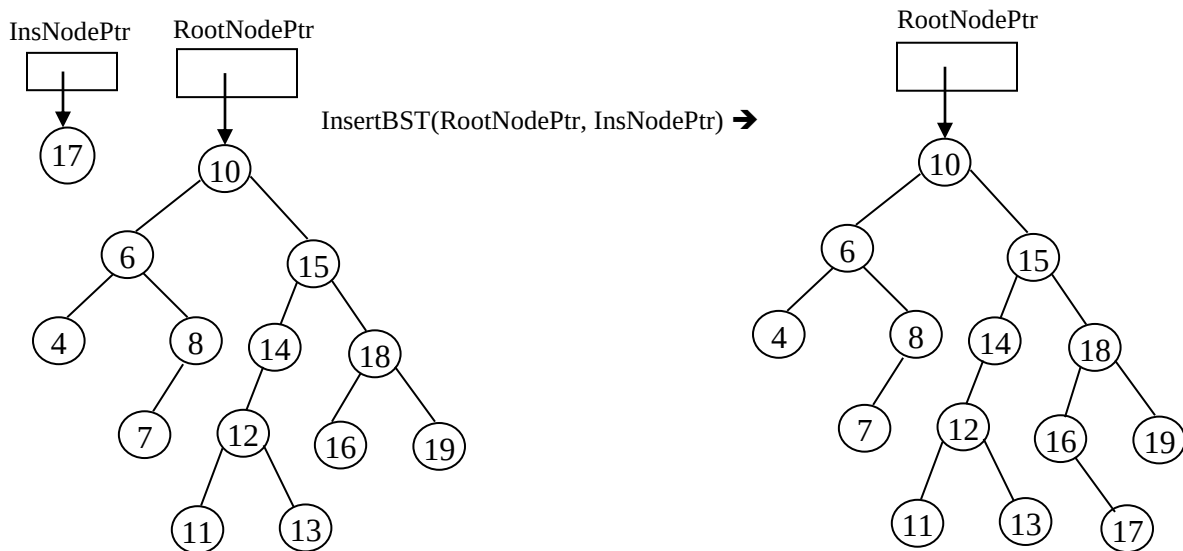
6.3.1. Insertion

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 (i.e. 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.



Function call:

```

if(RootNodePtr == NULL)
    RootNodePtr=InsNodePtr;
else
    InsertBST(RootNodePtr, InsNodePtr);

{
    //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;
    }
}
}

```

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);
    }
}
}

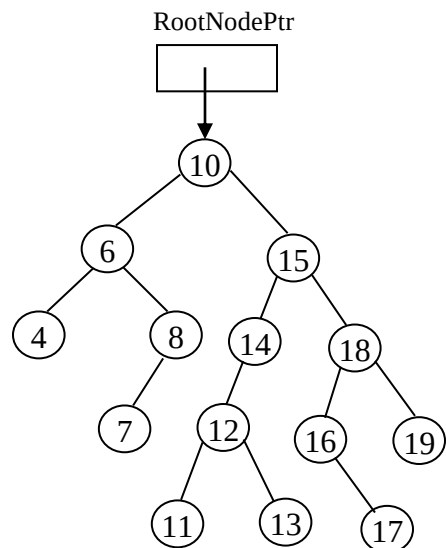
```

6.3.2. Traversing

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:



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

6.3.3. Application of binary tree traversal

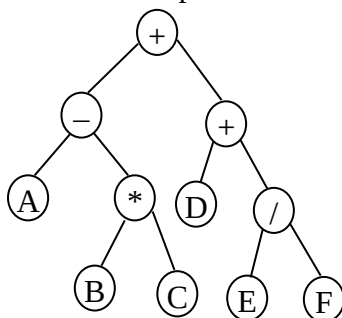
- Store values on leaf nodes and operators on internal nodes

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.

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

Function calls:

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

Implementation:

```
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
    }
}
```

6.3.4. 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.

Function call:

```
ElementExists = SearchBST (RootNodePtr, Number);
// ElementExists is a Boolean variable defined as: bool ElementExists = false;
```

Implementation:

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

When we search an element in a binary search tree, sometimes it may be necessary for the SearchBST function to return a pointer that points to the node containing the element searched. Accordingly, the function has to be modified as follows.

Function call:

```
SearchedNodePtr = SearchBST (RootNodePtr, Number);
// SearchedNodePtr is a pointer variable defined as: Node *SearchedNodePtr=NULL;
```

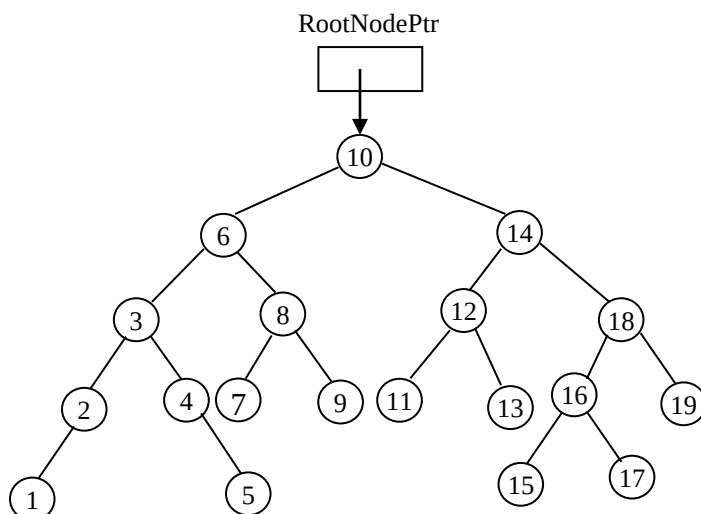
Implementation:

```
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));
}
```

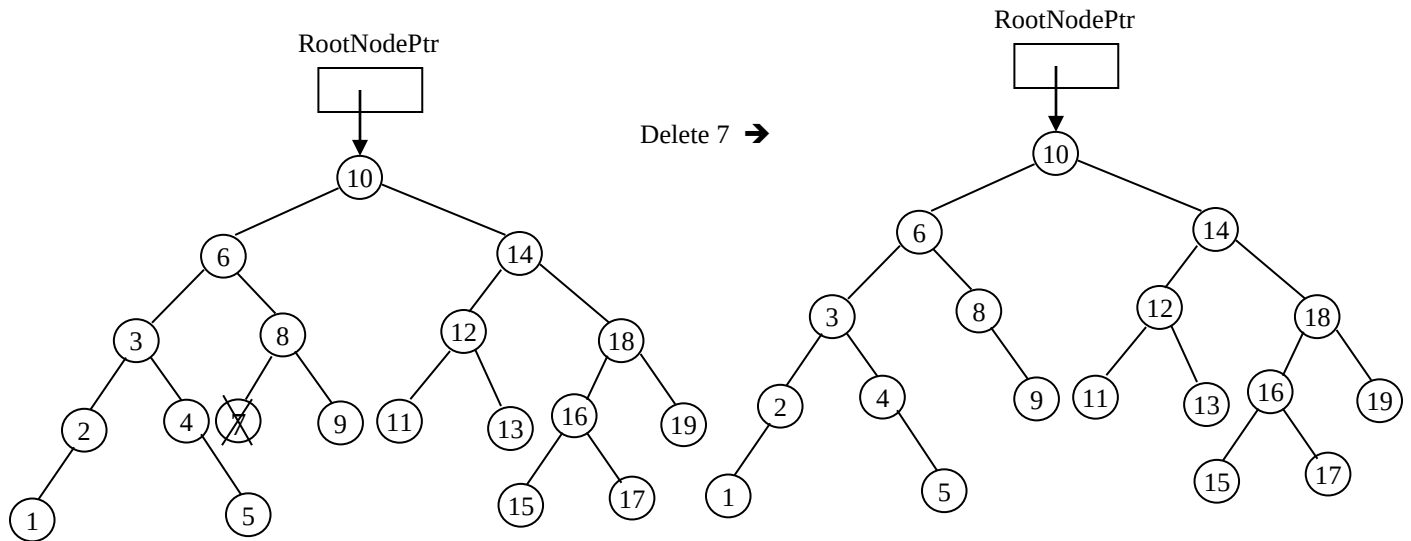
6.3.5. Deletion

To delete a node (whose Num value is N) from binary search tree (whose root node is pointed by RootNodePtr), 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.



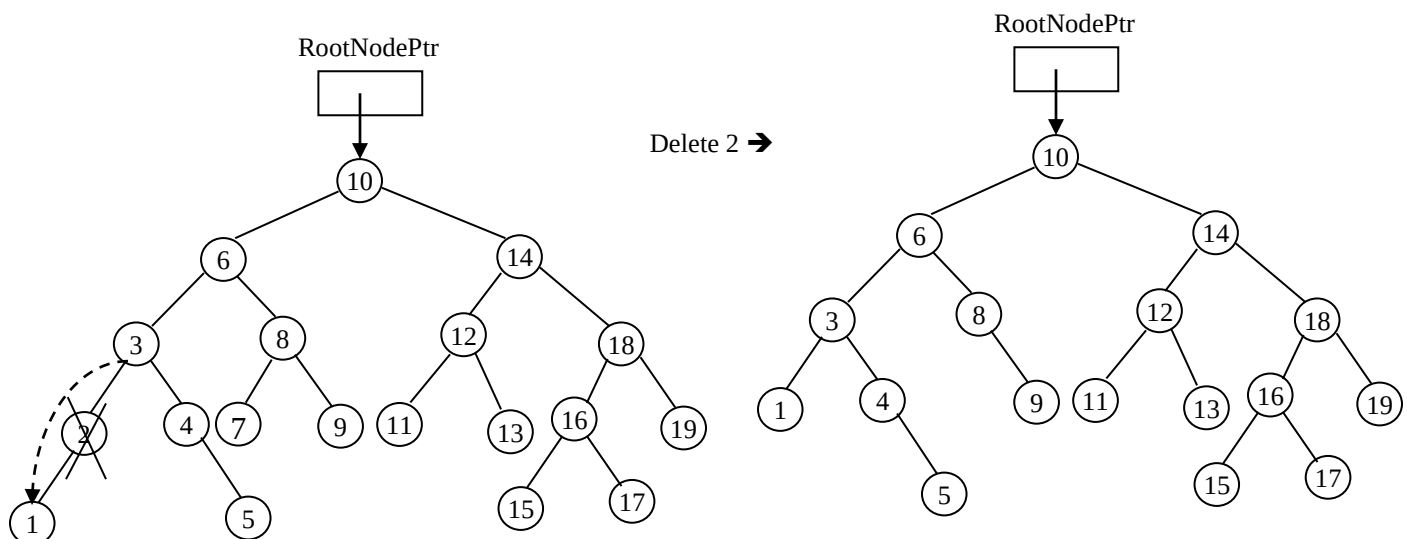
Case 1: Deleting a leaf node (a node having no child), e.g. 7



Case 2: Deleting a node having only one child, e.g. 2

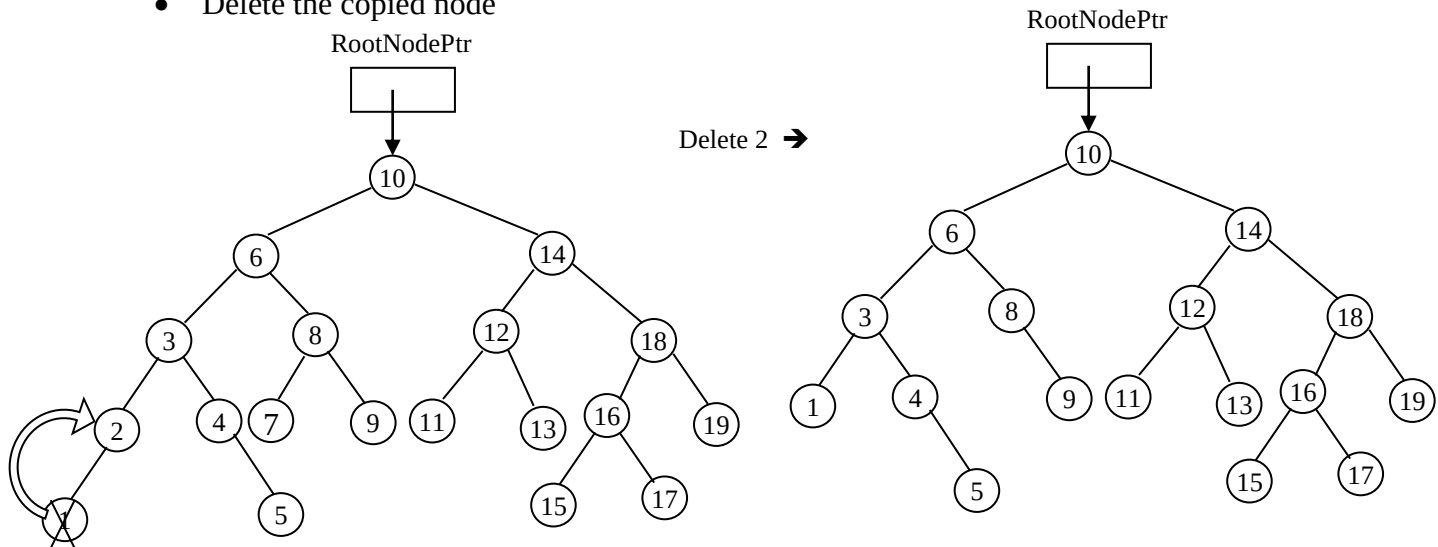
Approach 1: Deletion by merging – one of the following is done

- 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.



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

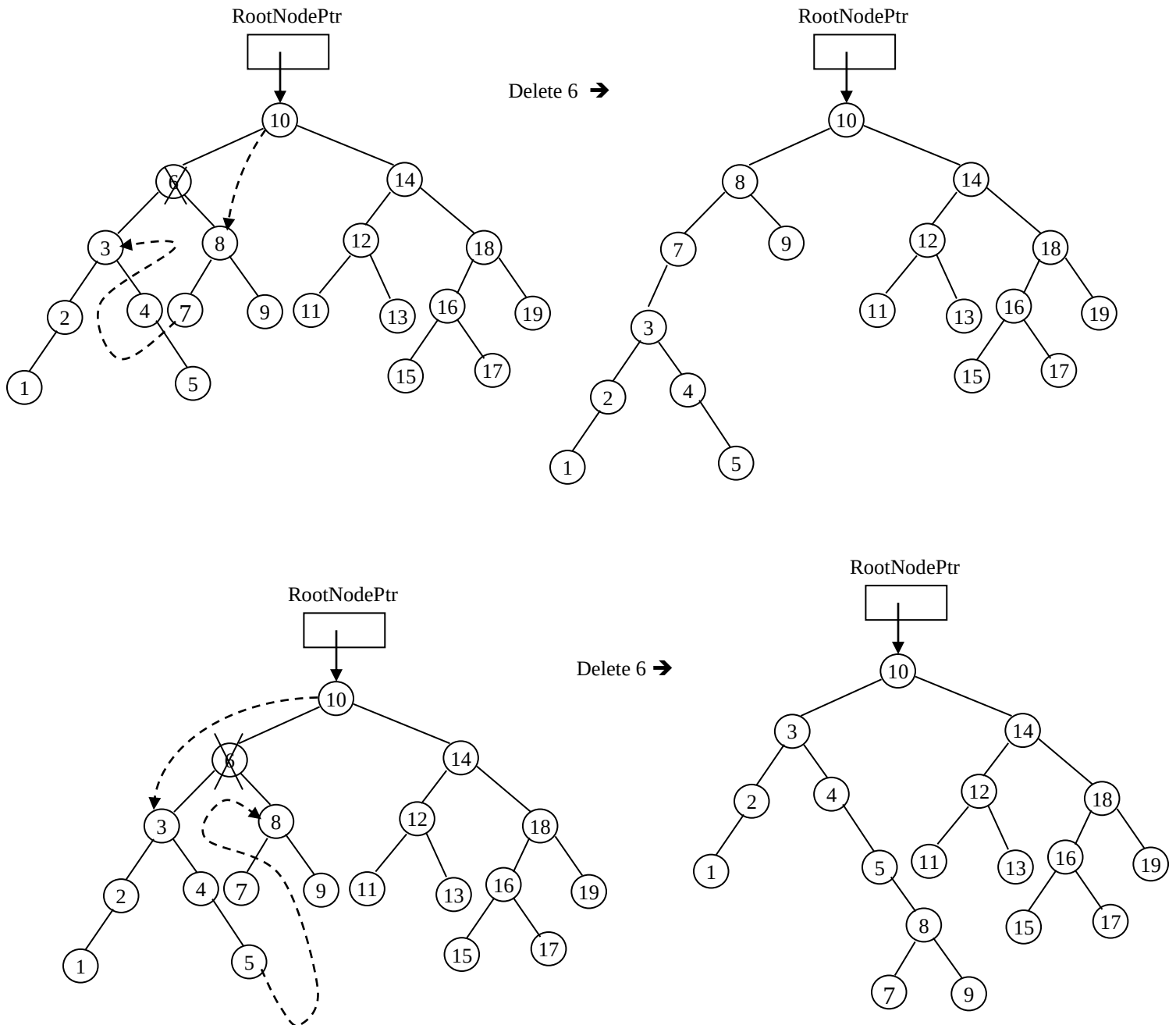


Case 3: Deleting a node having two children, e.g. 6

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

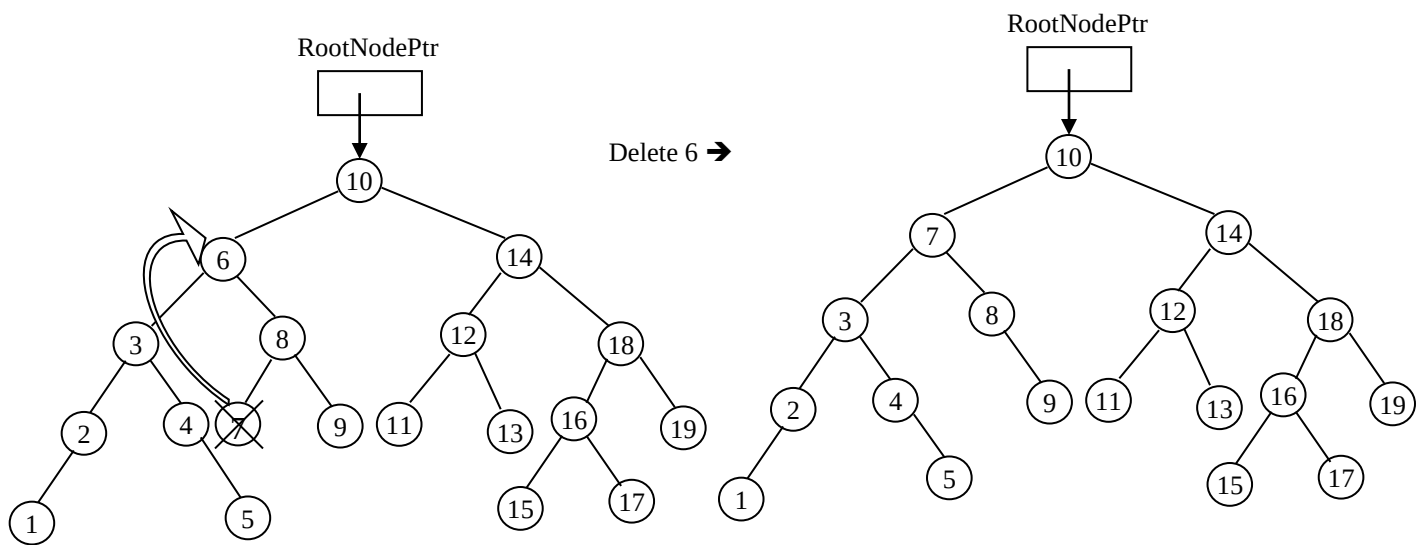
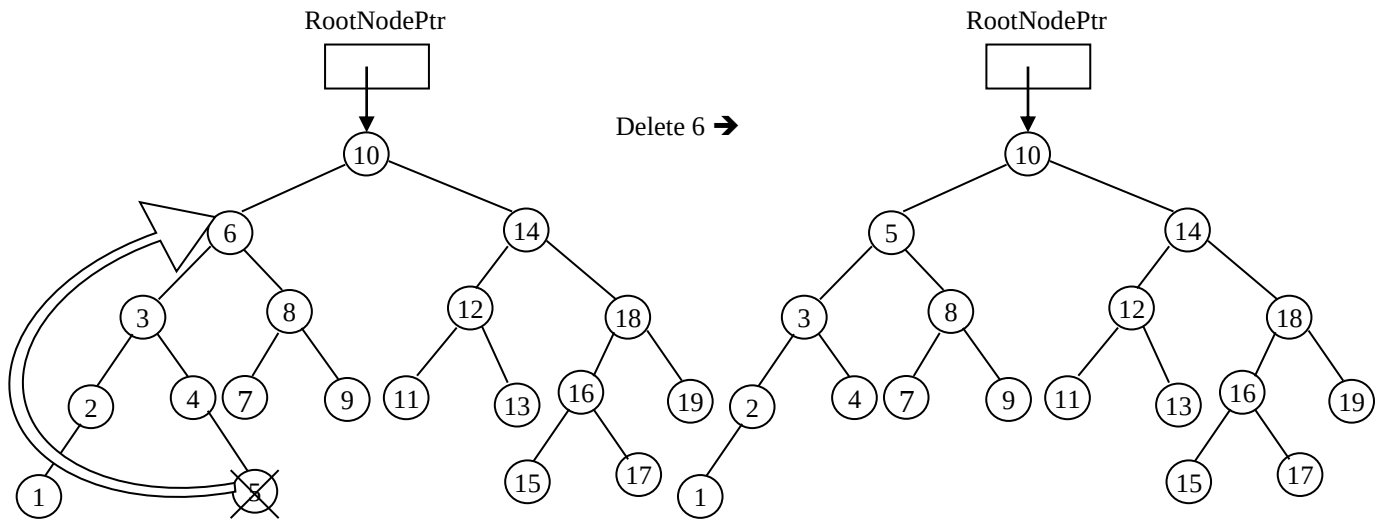
- If the deleted node is the left child of its parent, one of the following is done
 - The left child of the deleted node is made the left child of the parent of the deleted node, and
 - The right child of the deleted node is made the right child of the node containing largest element in the left of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



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

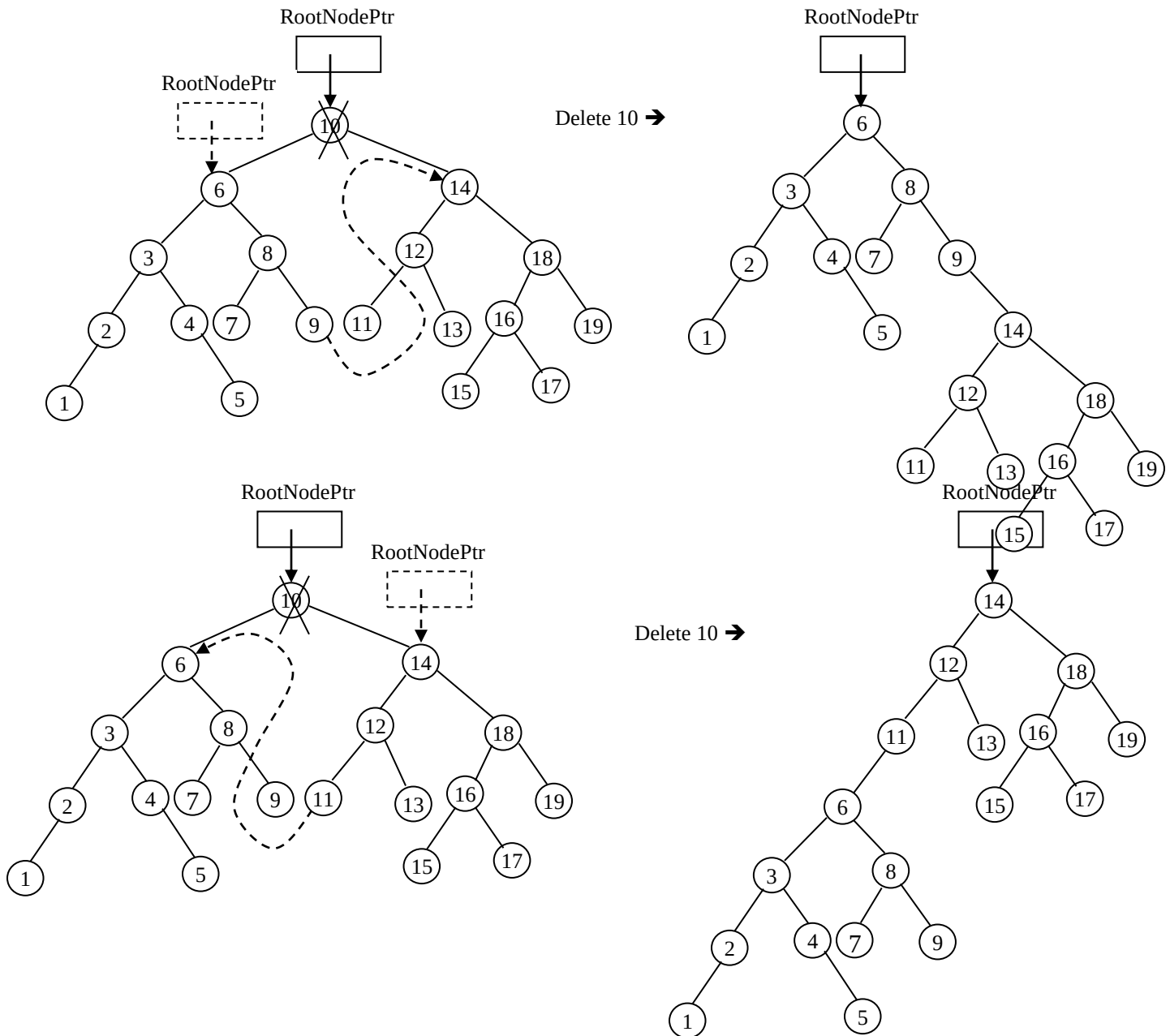


Case 4: Deleting the root node, 10

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

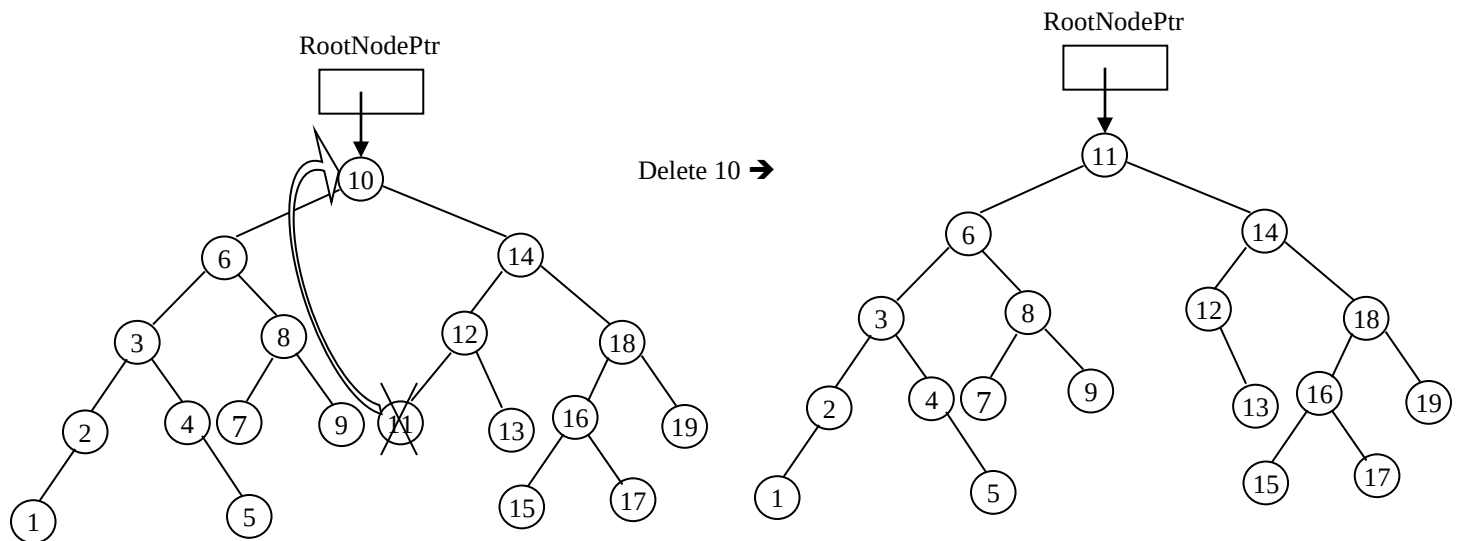
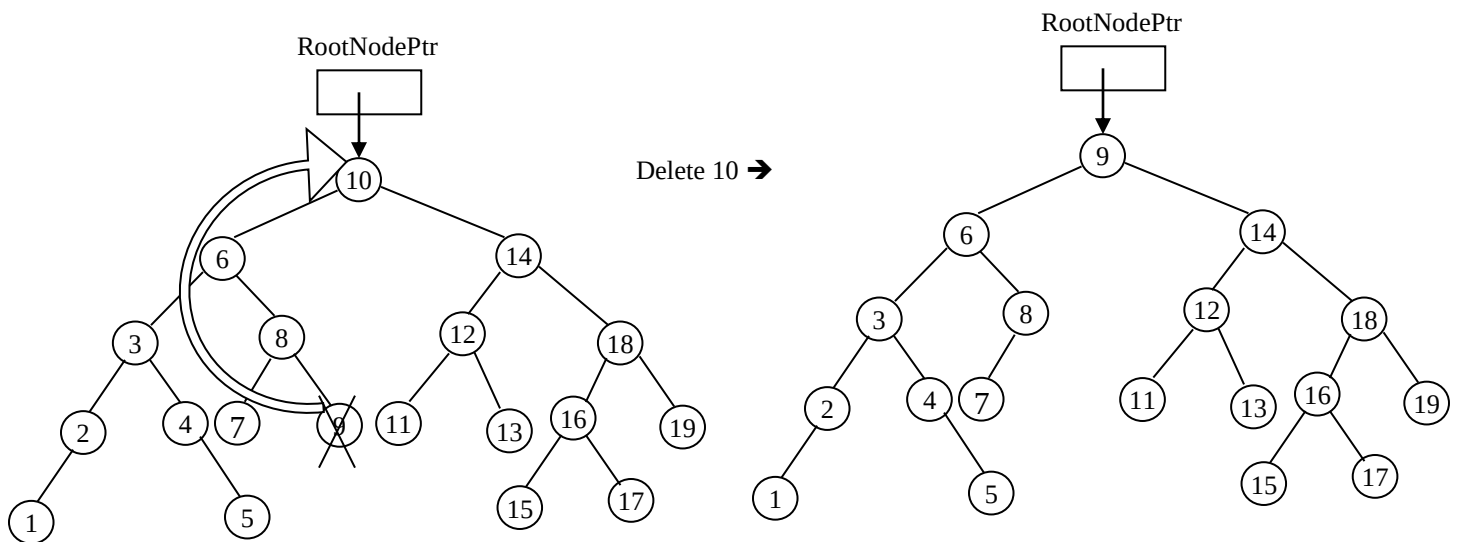
- If the tree has only one node the root node pointer is made to point to nothing (NULL)
- If the root node has left child
 - the root node pointer is made to point to the left child
 - the right child of the root node is made the right child of the node containing the largest element in the left of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



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



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;
        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);  
    }  
}
```