CHAPTER SIX
6. DATA STRUCTURES AND APPLICATIONS: TREES

6.1. Introduction

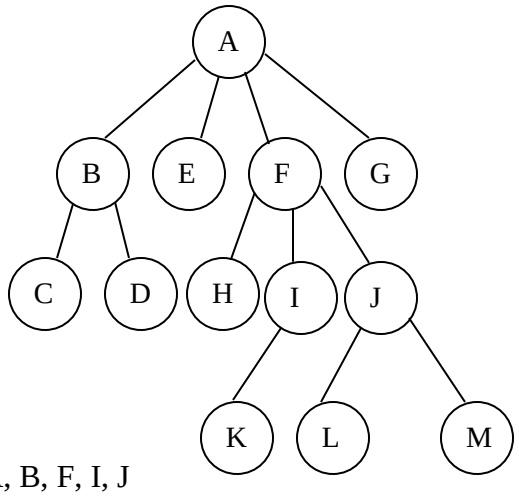
A tree is an abstract data type that stores elements hierarchically. With the exception of the top element, each element in a tree has a parent element and zero or more children elements. A tree is usually visualized by placing elements inside ovals, circles or rectangles, and by drawing the connections between parents and children with straight lines. We typically call the top element the root of the tree, but it is drawn as the highest element, with the other elements being connected below (just the opposite of a botanical tree).

Formally, we define tree T to be a set of nodes storing elements in a parent-child relationship with the following properties: (1) If T is non-empty, it has a special node, called the root of T, that has no parent. (2) Each node v of T different from the root has a unique parent node w; every node with parent w is a child of w.

Note that according to our definition, a tree can be empty, meaning that it doesn't have any nodes. This convention also allows us to define a tree recursively, such that a tree T is either empty or consists of a node r, called the root of T , and a (possibly empty) set of trees whose roots are the children of r. In a rooted tree, there is a unique path from the root to each node.

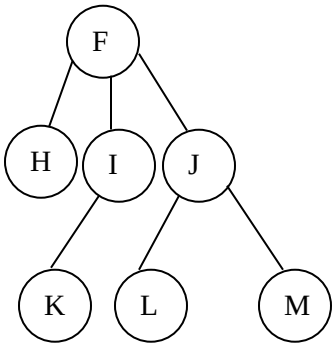
6.2. Tree Terminologies

Consider the following tree.

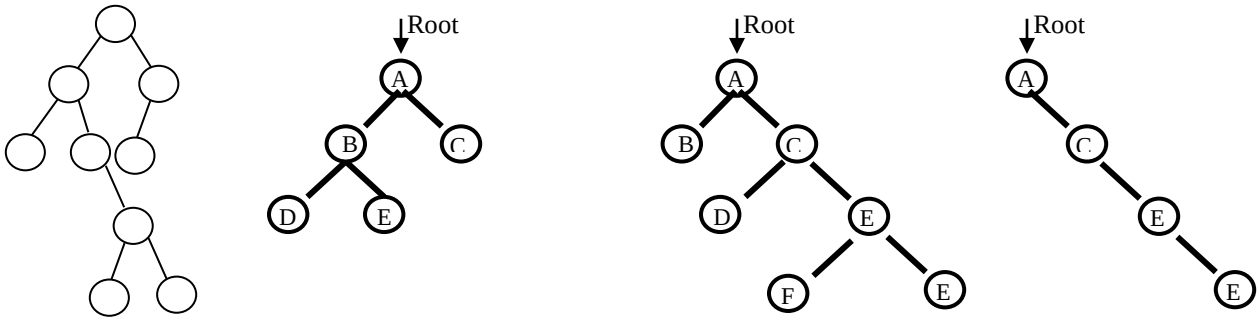


- **Root node**: a node without a parent. → A
- **Internal node**: a node with at least one child. →A, B, F, I, J
- **External (leaf) node**: a node without a child. → C, D, E, H, K, L, M, G
- **Ancestors of a node**: parent, grandparent, grand-grandparent, etc of a node.
  Ancestors of K → A, F, I
- **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
- **Siblings**: nodes with the same parent → e.g. H, I, J

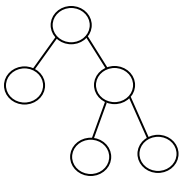
- ✓ **Subtree**: a tree consisting of a node and its descendants.



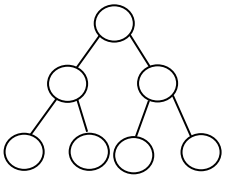
- ✓ **Binary tree**: a tree in which each node has at most two children called left child and right child.



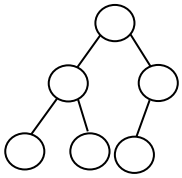
- ✓ **Full binary tree**: a binary tree where each node has either 0 or 2 children.



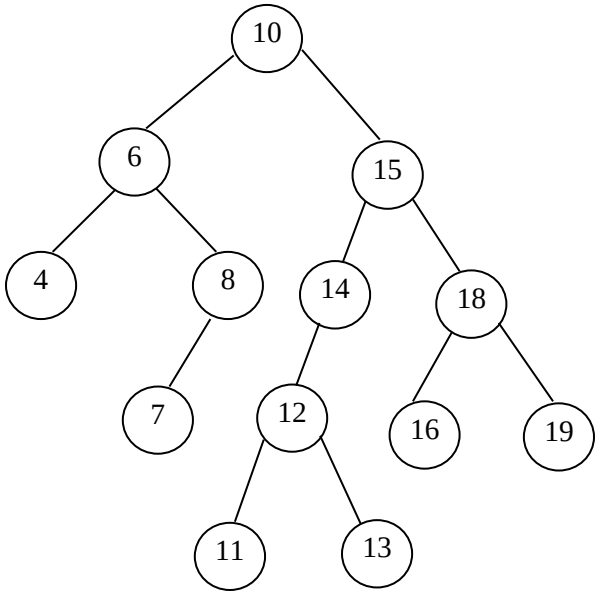
- ✓ **Balanced binary tree**: a binary tree where each node except the leaf nodes has left and right children and all the leaves are at the same level.



- ✓ **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 sub trees are also binary search trees.



6.3. Data Structure of a Binary Tree

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

6.4. Operations on Binary Search Tree

Consider the following definition of binary search tree.

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

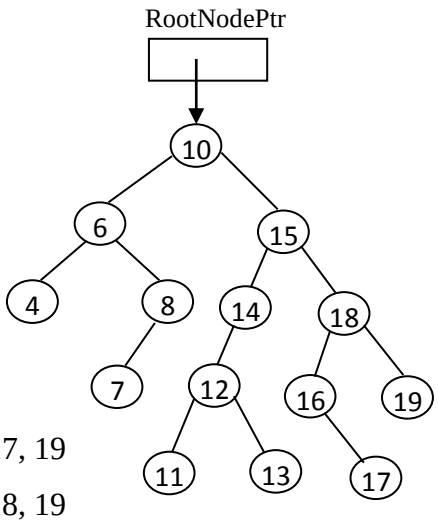
i. Traversing

Tree traversal is the process of visiting each node only one time. Traversal does not specify the order in which the nodes are visited. Hence, there are as many tree traversals as there are permutations of nodes; for a tree with n nodes, there are ***n!*** different traversals.

Binary search tree can be traversed in three ways.

a. Pre order traversal - traversing binary tree in the order of *parent*, *left* and *right*.
b. Inorder traversal - traversing binary tree in the order of *left*, *parent* and *right*.
c. 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.

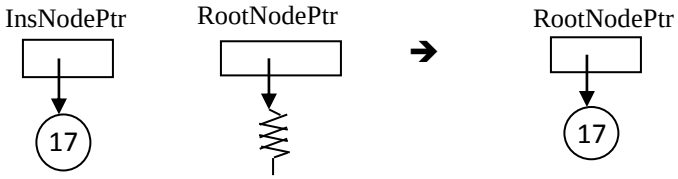
Postorder traversal - 4, 7, 8, 6, 11, 13, 12, 14, 17, 16, 19, 18, 15, 10

ii. 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.

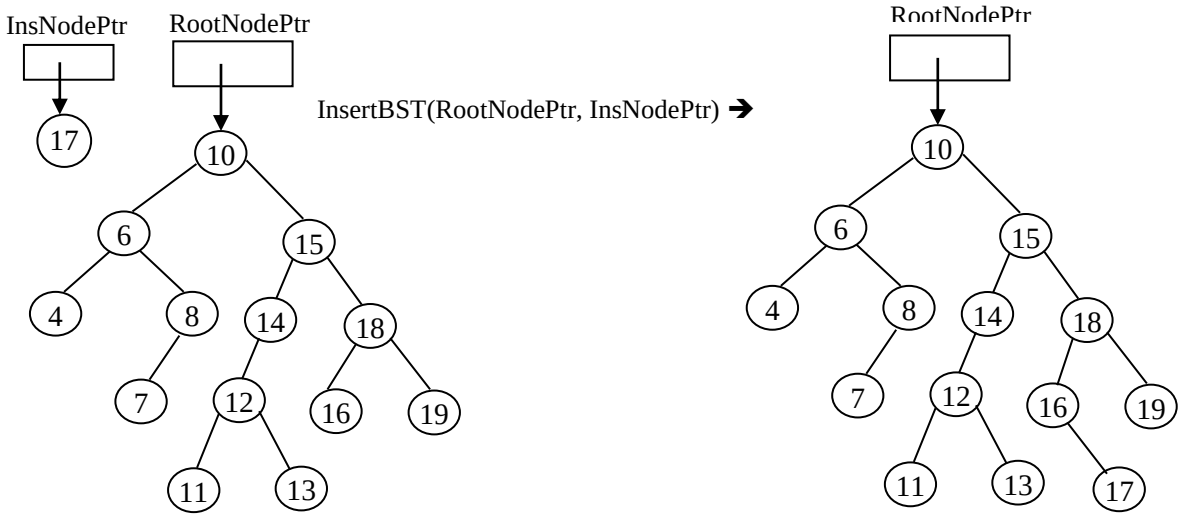
Case 1: - There is no data in the tree (i.e. RootNodePtr is NULL).

-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);
```

- **Implementation**

```
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;
             }
       }
  }
```

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);
     }
  }
```

iii. Searching

Searching is locating an element in a tree. For every node, compare the element to be located with the value stored in the node currently pointed at. If the element is less than the value, go to the left subtree and try again. If it is greater than that value, try the right subtree. If it is the same, obviously the search can be discontinued. The search is also aborted if there is no way to go, indicating that the element is not in the tree.

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:**

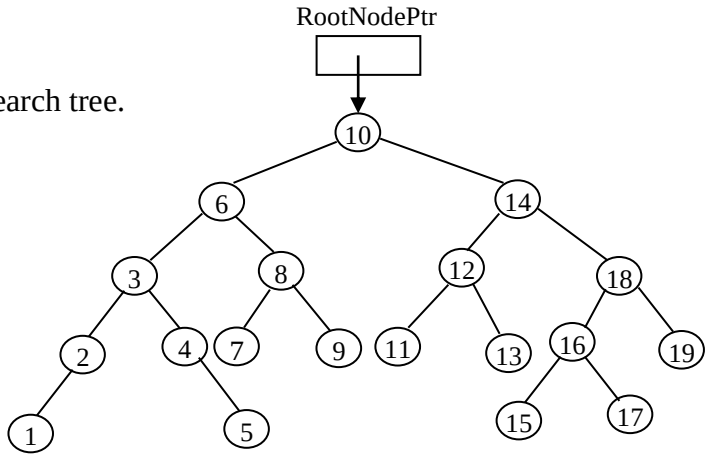
```
Node *SearchBST (Node *RNP, int x)
 {
   if((RNP = = NULL) || (RNP->Num = = x))
     return(RNP);
   else if(RNP->Num > x)
     return(SearchBST(RNP->Left, x));
   else
     return(SearchBST (RNP->Right, x));
 }
```

iv. Deletion

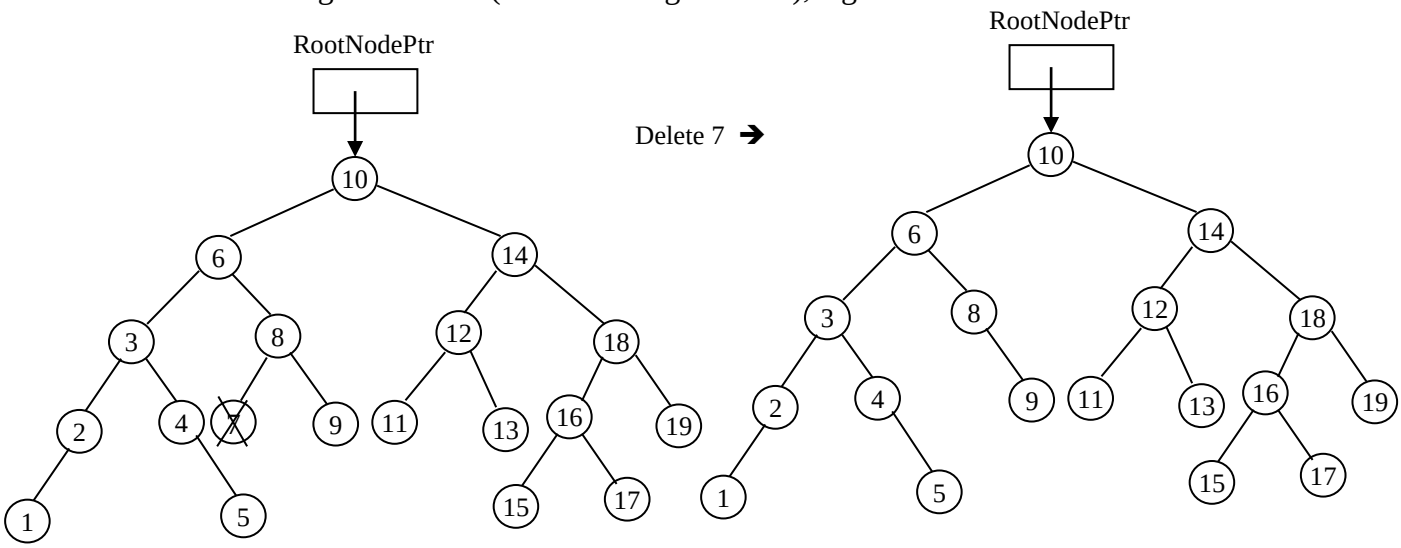
Deleting a node is another operation necessary to maintain a binary search tree. The level of complexity in performing the operation depends on the position of the node to be deleted in the tree. It is by far more difficult to delete a node having two subtrees than to delete a leaf; the complexity of the deletion algorithm is proportional to the number of children the node has.

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.



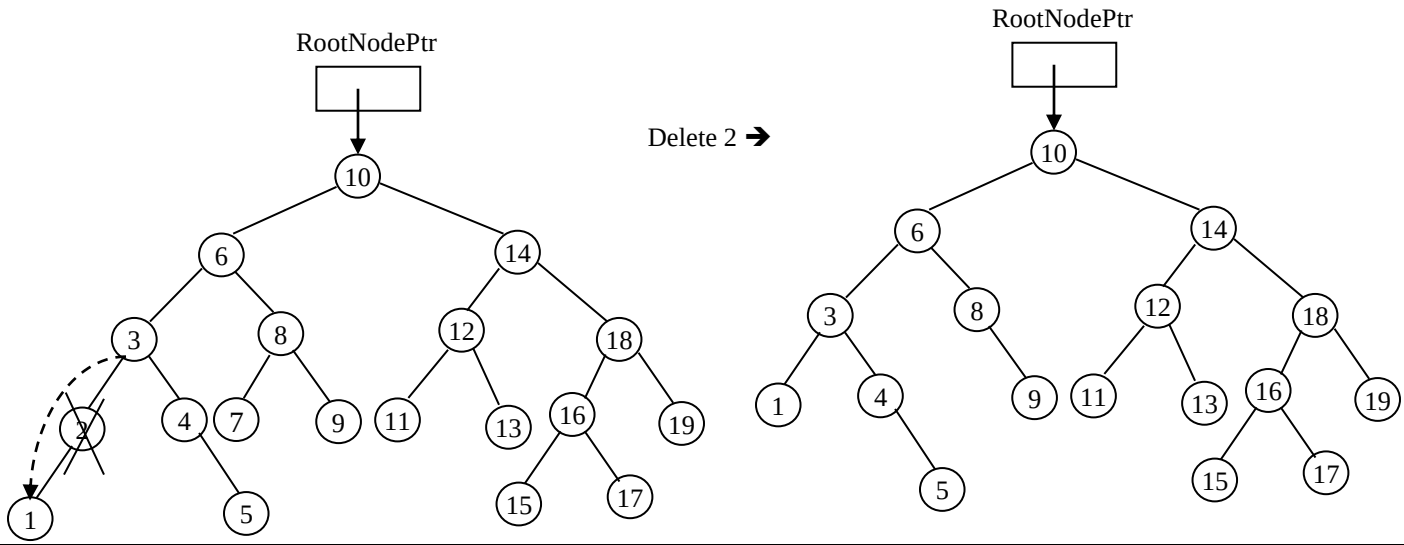
❖ **Case 1**: Deleting a leaf node (a node having no child), e.g. 7



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

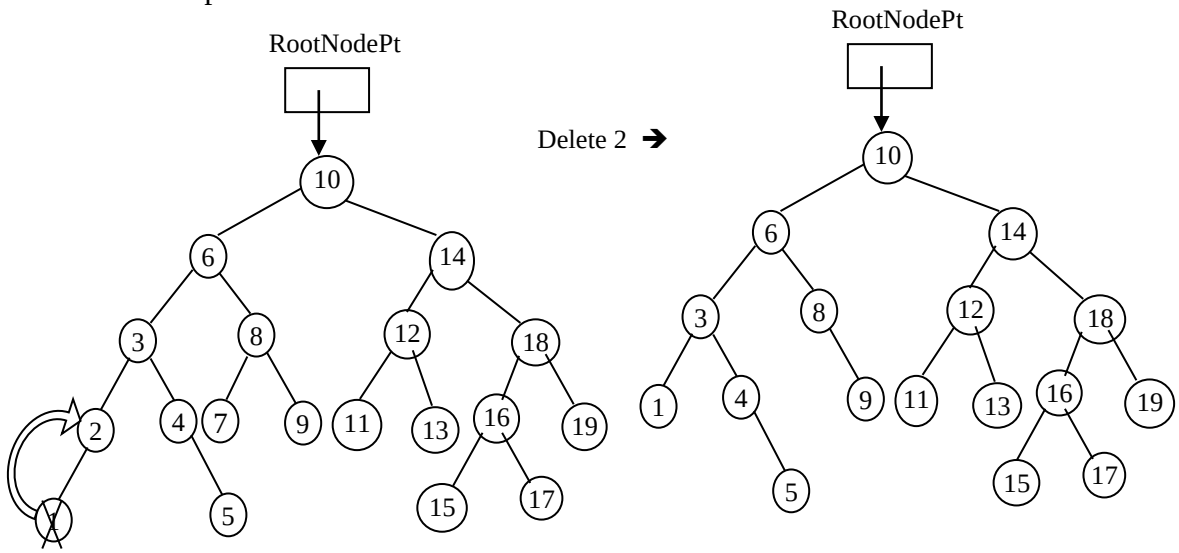
Approach 1: Deletion by merging – one of the following is done

- 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



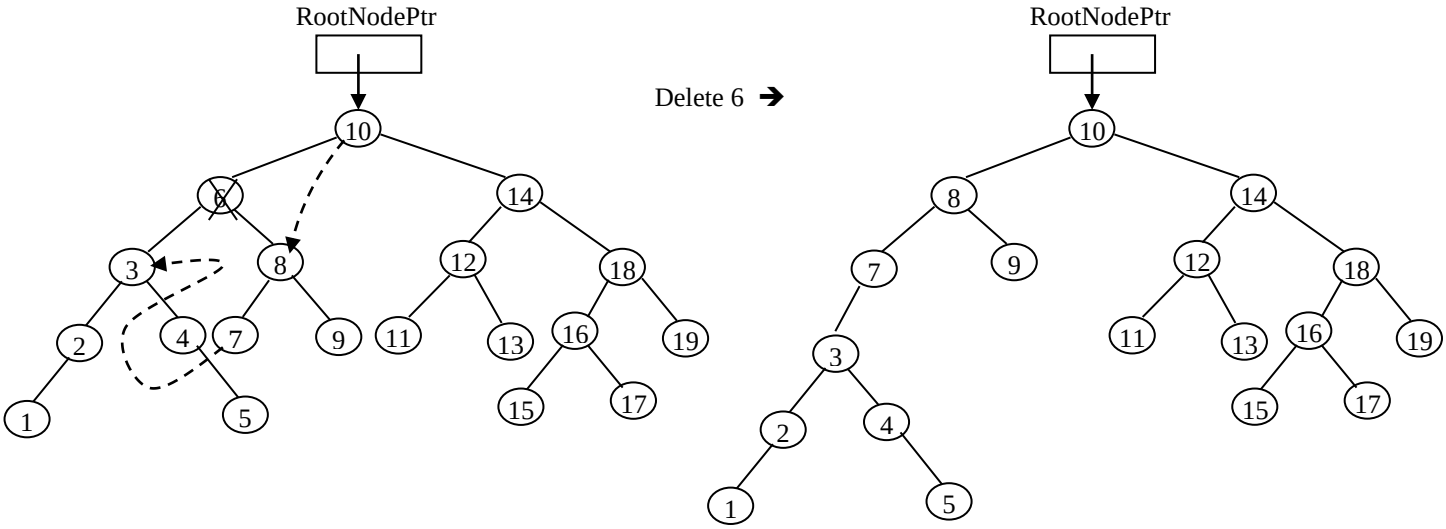
❖ **Case 3**: Deleting a node having two children, e.g. 6

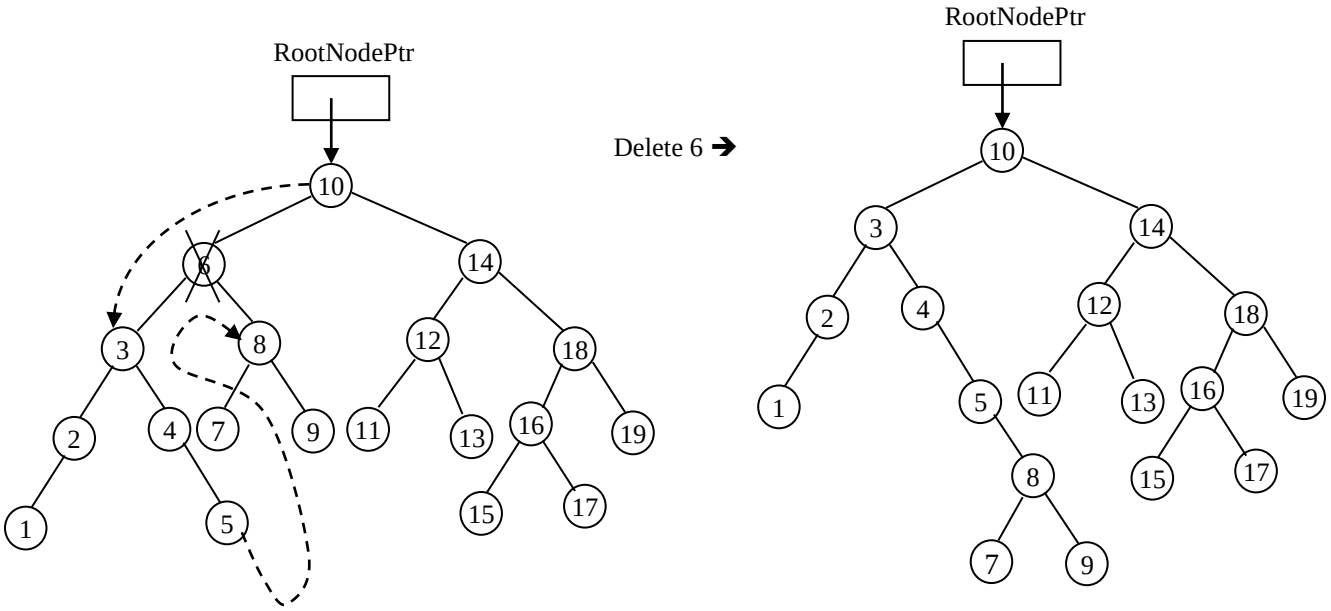
Approach 1: Deletion by merging – one of the following is done

- 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

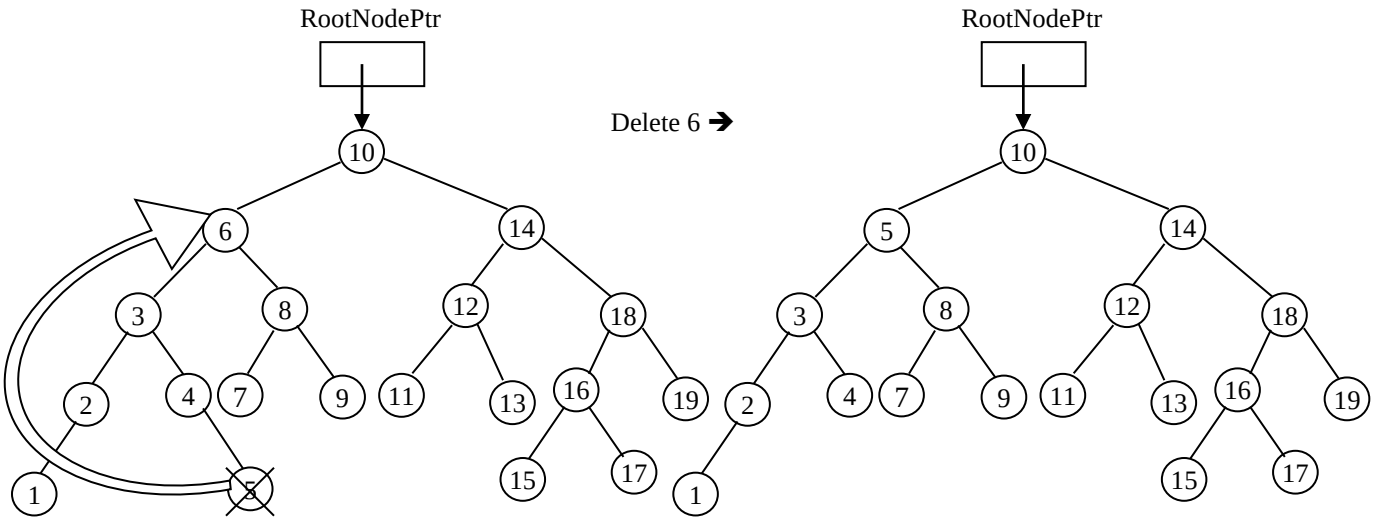
  OR
  - The right child of the deleted node is made the right child of the parent of the deleted node, and
  - The left child of the deleted node is made the left child of the node containing smallest element in the right of the deleted node.

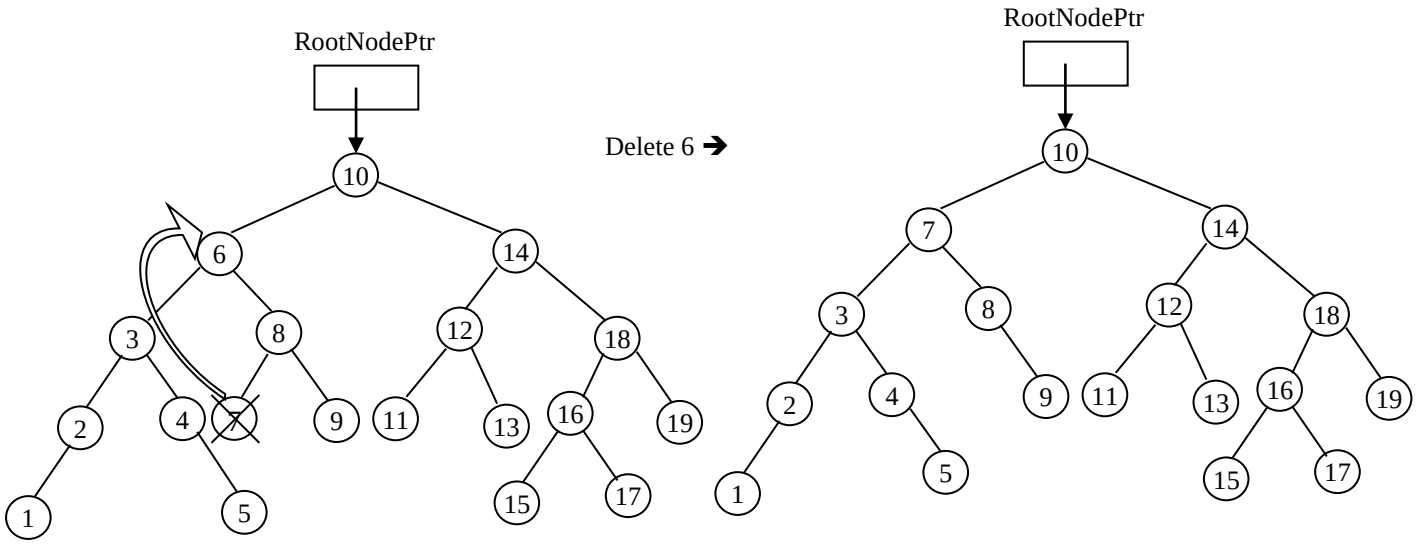




Approach 2: Deletion by copying- the following is done

- 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

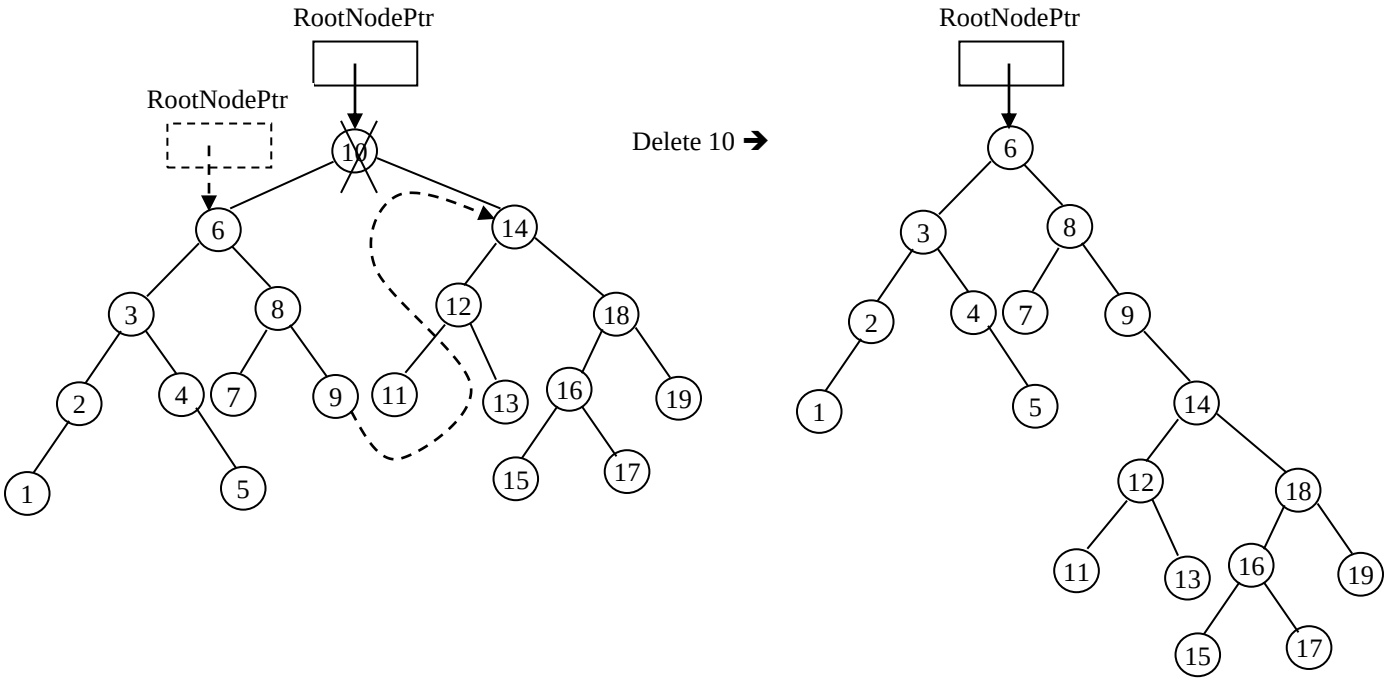


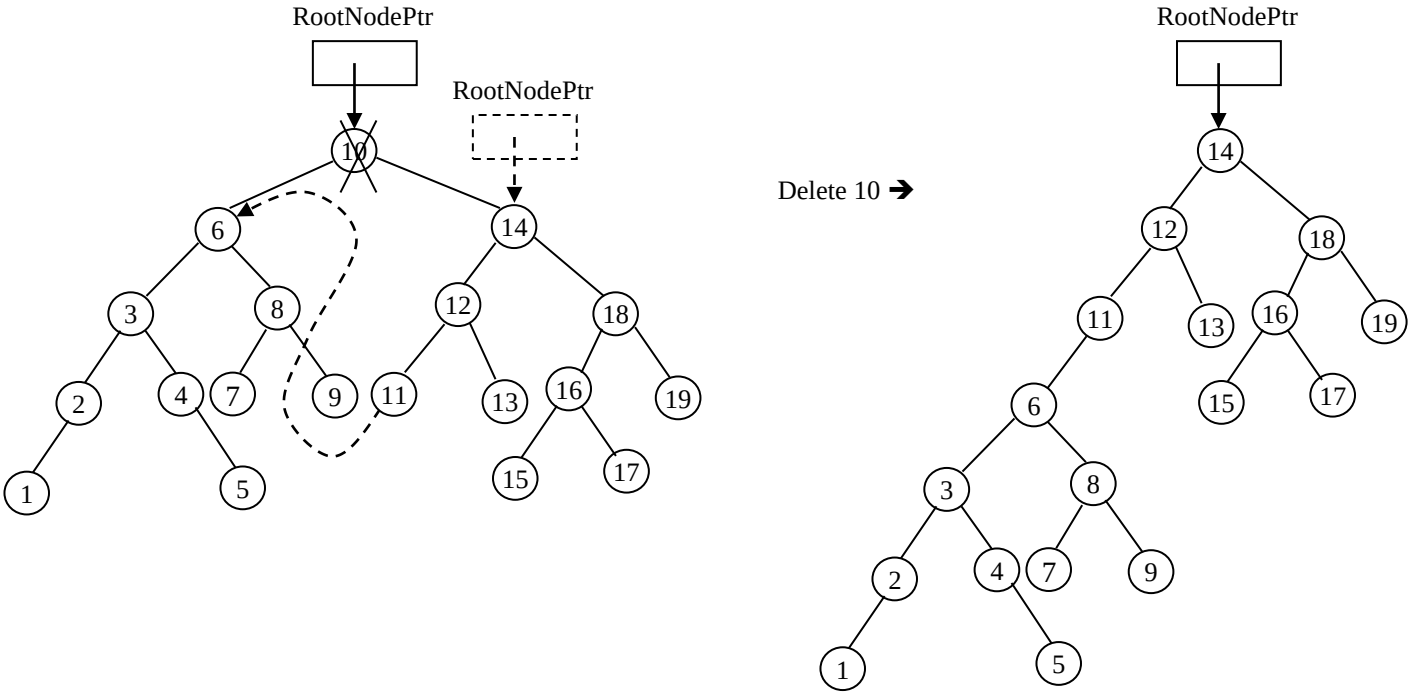


❖ **Case 4**: Deleting the root node, 10

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

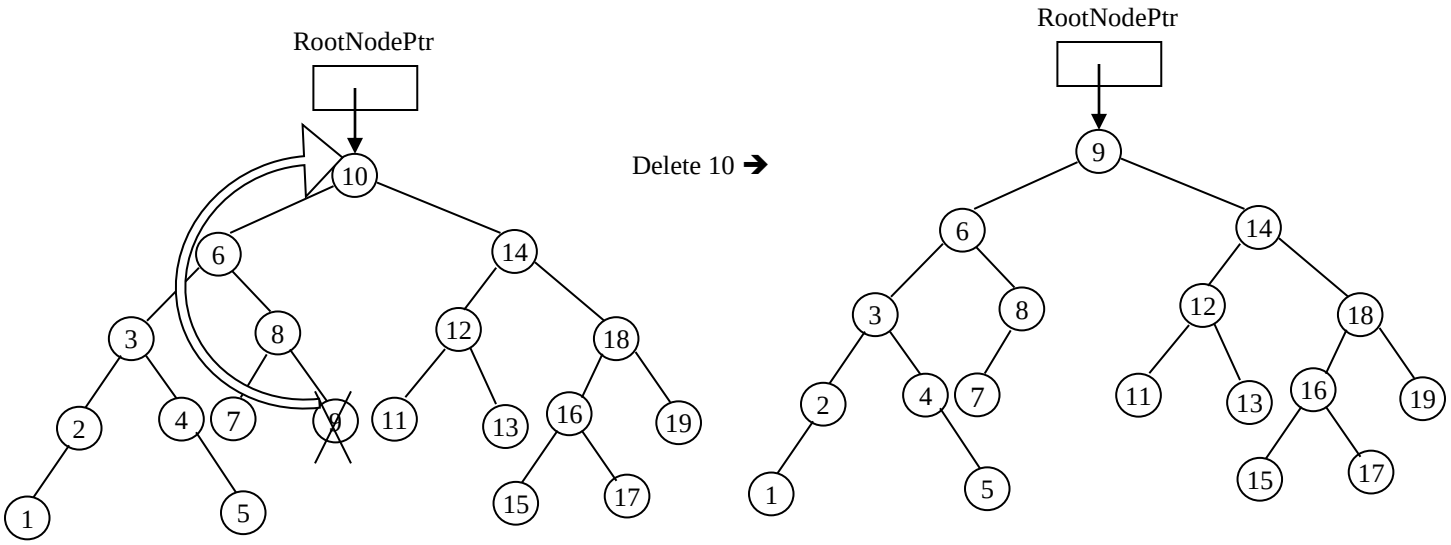
- If the tree has only one node the root node pointer is made to point to nothing (NULL)
- If the root node has left child
  - the root node pointer is made to point to the left child
  - the right child of the root node is made the right child of the node containing the largest element in the left of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

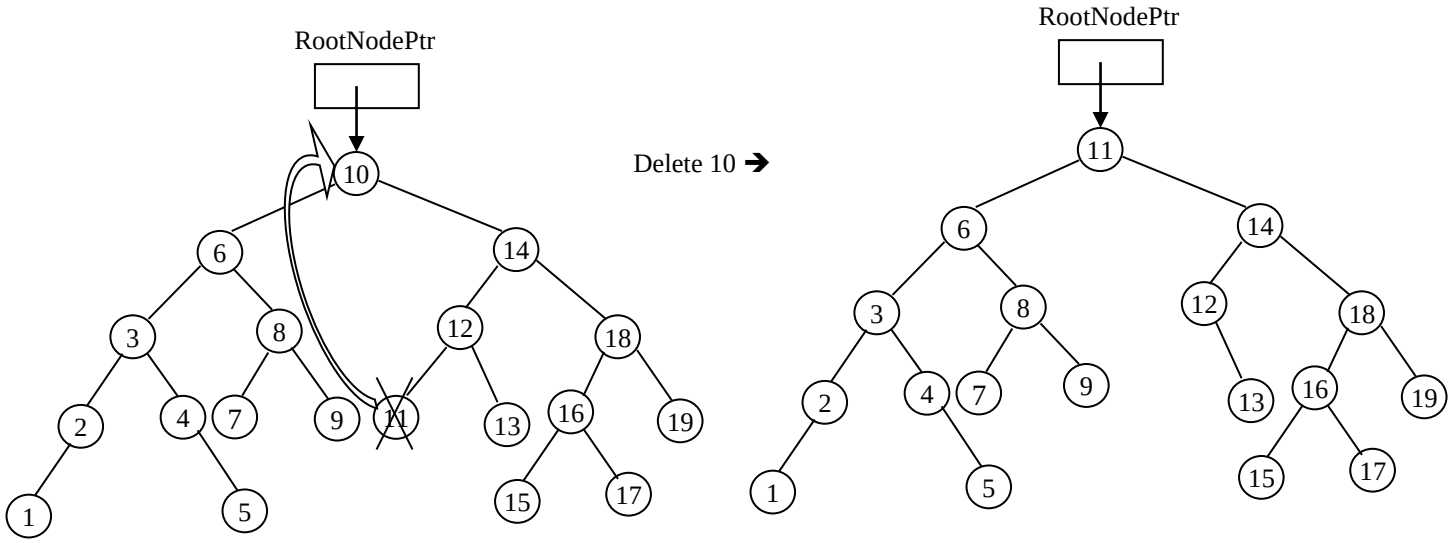




Approach 2: Deletion by copying- the following is done

- 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)

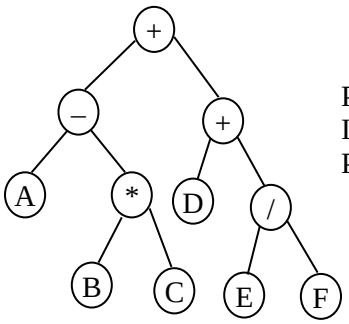
```
void DeleteBST(Node *RNP, Node *PDNP, int x)
{
   Node *DNP; // a pointer that points to the currently deleted node
   // PDNP is a pointer that points to the parent node of currently deleted node
   if(RNP = = NULL)
       cout<<"Data not found\n";
   else if (RNP->Num > x)
       DeleteBST(RNP->Left,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

6.5. Application of binary trees

I) For 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
    }
 }
```

II) For constructing an expression tree

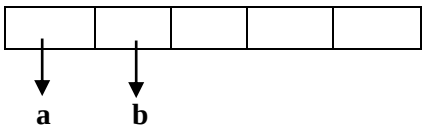
We can convert a postfix expression into an expression tree. Since we already have an algorithm to convert infix and prefix to postfix, we can generate expression trees from any types of input expression. This method strongly resembles the postfix evaluation algorithm.

Algorithm:

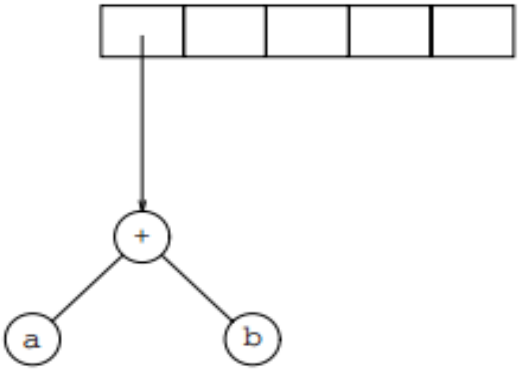
1. If the expression is not postfix, then convert the expression in to postfix.
2. Read the expression one symbol at a time.
3. If the symbol is an operand, we create a one node tree and push a pointer to it onto a stack.
4. If the symbol is an operator, we pop pointers to two trees T_1 and T_2 from the stack and form a new tree whose root is the operator and whose left and right children point to T2 and T2 respectively. A pointer to this new tree is then pushed onto the stack.

Ex. create an expression tree for a b + c d e + * *

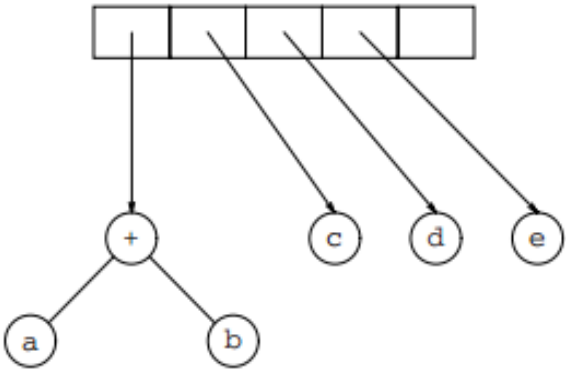
The first two symbols are operands, so we create one node trees and push pointers to them onto a stack.



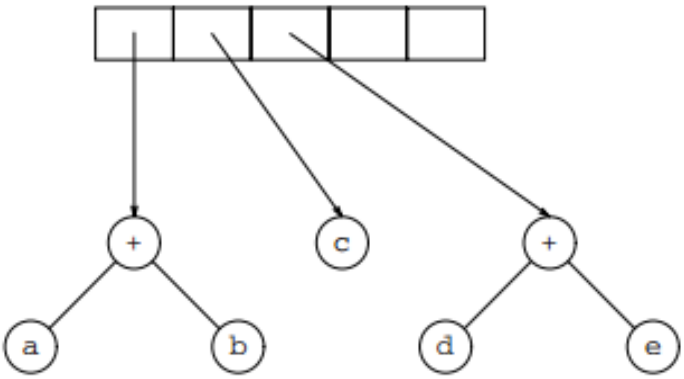
Next, a **+** is read, so two pointers to trees are popped, a new tree is formed, and a pointer to it is pushed onto the stack.



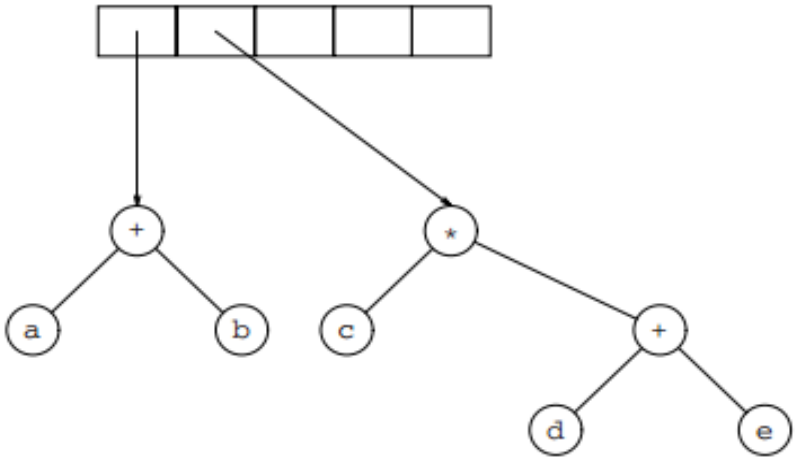
Next, **c**, **d**, and **e** are read, and for each a one-node tree is created and a pointer to the corresponding tree is pushed onto the stack.



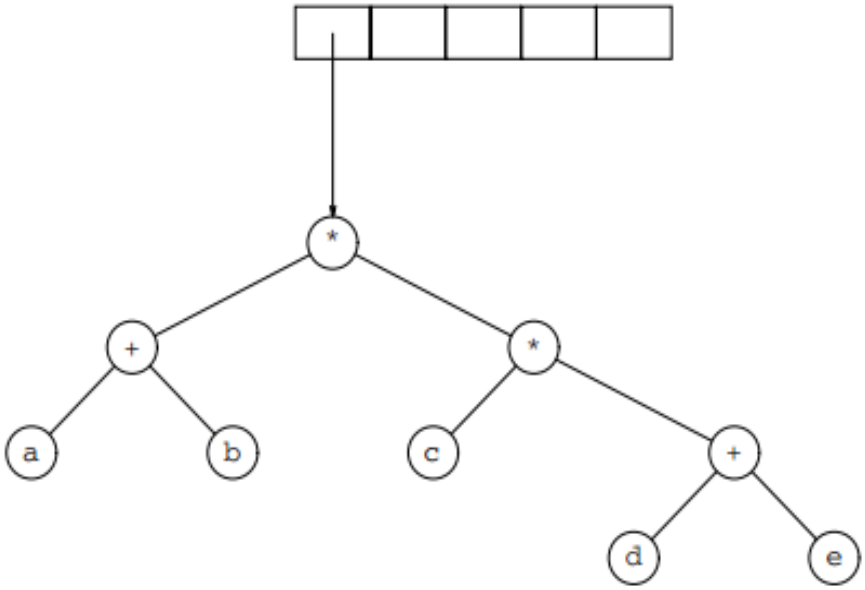
Now a **+** is read, so two trees are merged. Pop two pointers and create a new subtree by merging with **+** and push it on to the stack



Continuing, a ***** is read, so we pop two tree pointers and form a new tree with a ***** as root. Pop two pointers and create a new subtree by merging with * and push it on to the stack



Finally, the last symbol is read, two trees are merged, and a pointer to the final tree is left on the stack. Pop two pointers and create a new tree by merging with ***** and push it on to the stack. Thus we are having an expression tree.

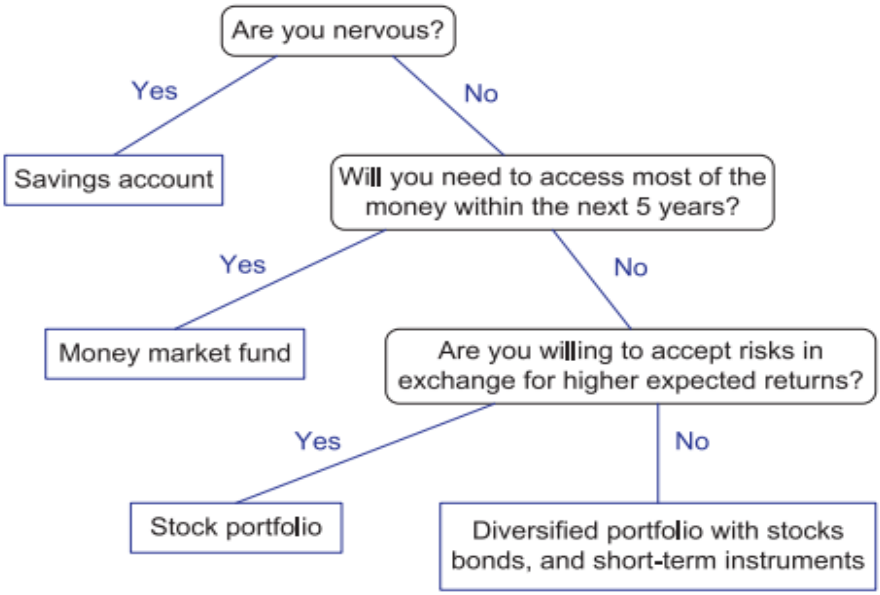


III) Some other applications of trees

An important class of binary trees arises in contexts where we wish to represent a number of different outcomes that can result from answering a series of yes-or-no questions. Each internal node is associated with a question. Starting at the root, we go to the left or right child of the current node, depending on whether the answer to the question is “Yes” or “No.” With each decision, we follow an edge from a parent to a child, eventually tracing a path in the tree from

the root to an external node. Such binary trees are known as decision trees, because each external node p in such a tree represents a decision of what to do if the questions associated with p's ancestors are answered in a way that leads to p.

The following figure illustrates a decision tree that provides recommendations to a prospective investor.

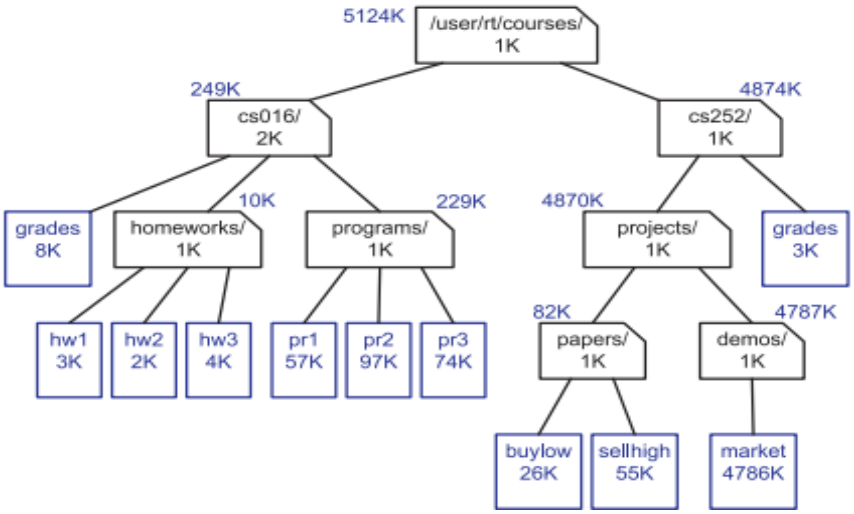


Another application of tree is for organizing file system hierarchically. In most operating systems, files are organized hierarchically into nested directories (also called folders), which are presented to the user in the form of a tree. More specifically, the internal nodes of the tree are associated with directories and the external nodes are associated with regular files. In the UNIX and Linux operating systems, the root of the tree is appropriately called the "root directory," and is represented by the symbol "/."

Consider a file-system tree *T*, where external nodes represent files and internal nodes represent directories. Suppose we want to compute the disk space used by a directory, which is recursively given by the sum of the following:

- ✓ The size of the directory itself
- ✓ The sizes of the files in the directory
- ✓ The space used by the children directories

The following tree represents a file system, showing the name and size of the associated file/directory inside each node, and the disk space used by the associated directory above each internal node.



Another application of tree is expressing organizational structures. Corporations or business entities often represent their internal organization in terms of a corporate hierarchy, similar to the following figure, with the president or CEO at the top and the rank-and-file employees at the bottom. Each connecting link represents a supervisory relationship. In general, each person has only one immediate boss, but each boss may have several underlings. Notice that the structure shows relative information about the individual employees: roughly, the closer a person is to the top of the tree, the higher the position in the chart—and on the corporate ladder. This particular mixed metaphor seems to make sense: a corporate hierarchy describes the corporation as a whole and is tree-structured, but "corporate ladder" views that same structure from the perspective of a single individual. In particular, it shows the linear path to the top. The chart even shows which persons are approximately equal in the organization; even though they have unrelated job titles and work in different divisions, they are placed at the same level of the tree.