

# **Rajalakshmi Engineering College**

**Rajalakshmi Nagar, Thandalam, Chennai - 602 105**

## **Department of Computer Science and Engineering**



**CS19241 - Data Structures**

**Unit - III**

**Lecture Notes**

**(Regulations - 2019)**



**Prepared by:**

**B.BHUVANESWARAN**

**Assistant Professor (SG) / CSE / REC**

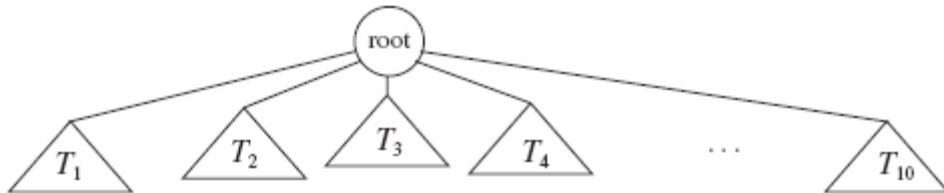
**[bhuvaneshwaran@rajalakshmi.edu.in](mailto:bhuvaneshwaran@rajalakshmi.edu.in)**



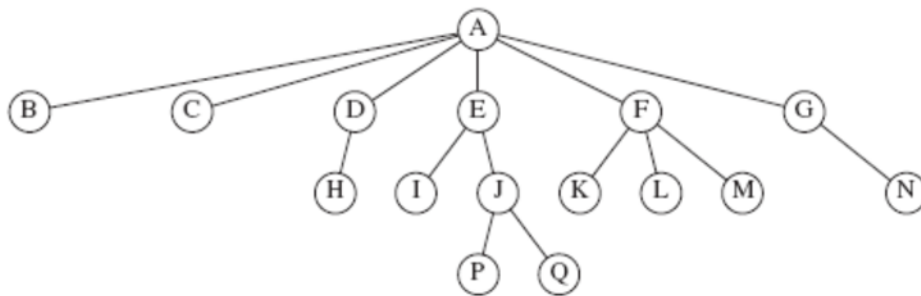
## 25.1 INTRODUCTION

A tree is a collection of nodes. The collection can be empty; otherwise, a tree consists of a distinguished node  $r$ , called the root, and zero or more nonempty (sub) trees  $T_1, T_2, \dots, T_k$ , each of whose roots are connected by a directed edge from  $r$ .

The root of each subtree is said to be a child of  $r$ , and  $r$  is the parent of each subtree root.



From the recursive definition, a tree is a collection of  $N$  nodes, one of which is the root, and  $N - 1$  edges. That there are  $N - 1$  edges follows from the fact that each edge connects some node to its parent, and every node except the root has one parent.



## 25.2 BASIC TERMINOLOGIES

### Root

The root of a tree is the node with no parents. There can be at most one root node in a tree.

### Node

Item of information.

### Leaves (or) Leaf Nodes

Nodes with no children are known as leaves or leaf nodes.

### Siblings

Nodes with the same parent are called siblings.

### Edge

An edge refers to the link from parent to child.

## Path

A path from node  $n_1$  to  $n_k$  is defined as a sequence of nodes  $n_1, n_2, \dots, n_k$  such that  $n_i$  is the parent of  $n_{i+1}$  for  $1 \leq i < k$ . In a tree there is exactly one path from the root to each node.

## Length

The length of the path is the number of edges on the path, namely,  $k - 1$ . There is a path of length zero from every node to itself. In a tree there is exactly one path from the root to each node.

## Degree

The number of subtrees of a node is called its degree. The degree of the tree is the maximum degree of any node in the tree.

## Level

The level of a node is defined by initially letting the root be at level one, if a node is at level  $L$  then its children are at level  $L + 1$ .

## Depth

For any node  $n_i$ , the depth of  $n_i$  is the length of the unique path from the root to  $n_i$ . Thus, the root is at depth 0. The depth of the tree is the maximum depth among all the nodes in the tree.

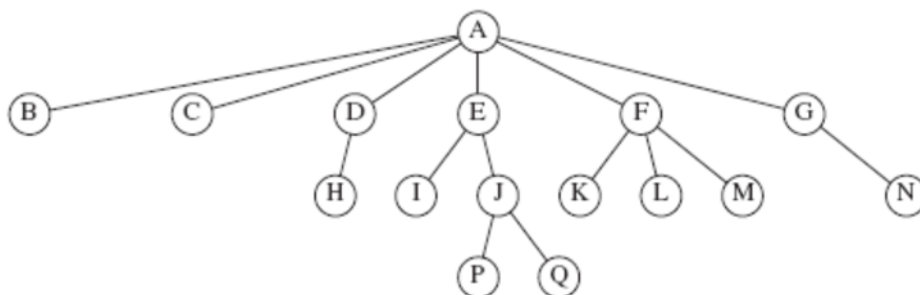
## Height

The height of  $n_i$  is the length of the longest path from  $n_i$  to a leaf. Thus all leaves are at height 0. The height of a tree is equal to the height of the root.

Note: For a given tree, depth and height returns the same value. But for individual nodes we may get different results.

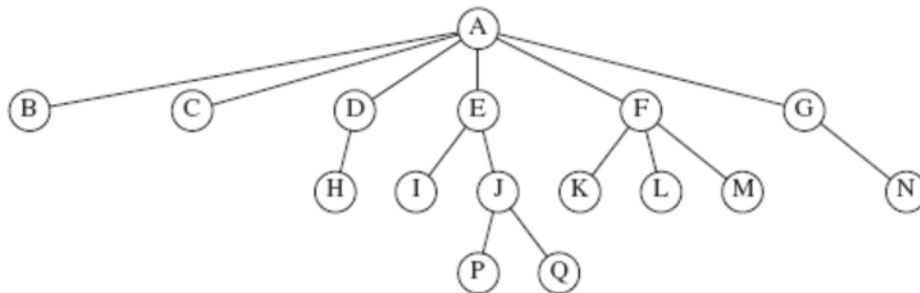
## 25.3 EXAMPLES

For the tree in following figure:



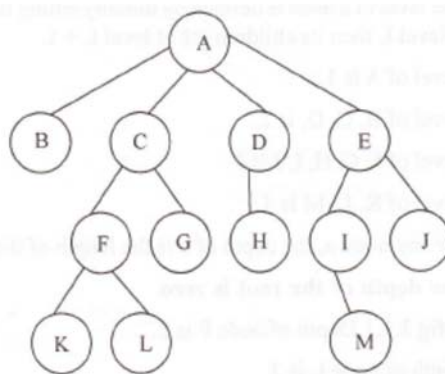
- The root is A.
- Node F has A as a parent and K, L, and M as children.
- Each node may have an arbitrary number of children, possibly zero.
- The leaves in the tree are B, C, H, I, P, Q, K, L, M, and N.
- K, L, and M are all siblings.
- Grandparent and grandchild relations can be defined in a similar manner.

For the tree in following figure:



- What is the depth and height E?
  - E is at depth 1 and height 2.
- What is the depth and height F?
  - F is at depth 1 and height 1.
- What is the height of the tree?
  - The height of the tree is 3.
- What is the depth of the tree?
  - The depth of a tree is equal to the depth of the deepest leaf; this is always equal to the height of the tree.

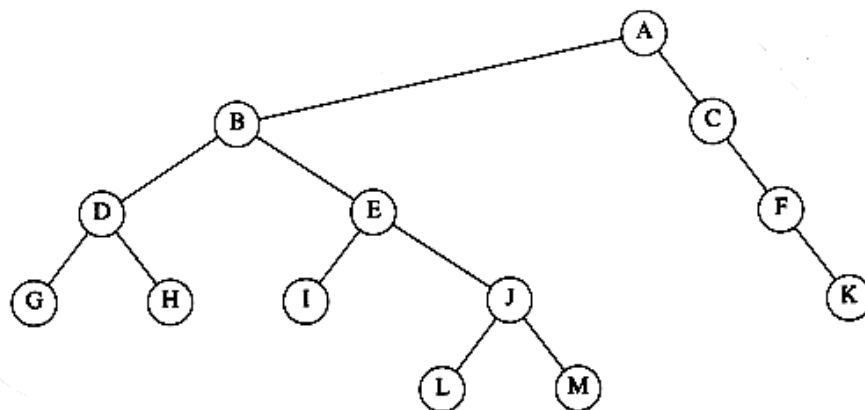
For the tree in following figure:



- What is the root?
  - The Root is A.
- What are leaves?
  - B, K, L, G, H, M, J are leaves.
- What are siblings?
  - B, C, D, E are siblings, F, G are siblings, I, J and K, L are siblings.
- What is the path from A to L?
  - Path from A to L is A, C, F, L.
- What is the length for the path A to L?
  - The length for the path A to L is 3.

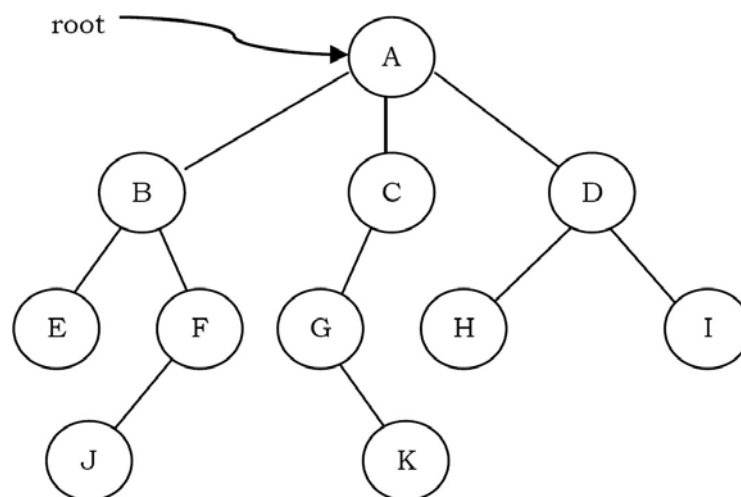
- What is the degree of A, C, D and H?
  - Degree of A is 4, C is 2, D is 1 and H is 0.
- What is the degree of the tree?
  - The degree of the tree is 4.
- What is the level of A?
  - Level of A is 1.
- What is the level of B, C, D?
  - Level of B, C, D, is 2.
- What is the level of F, G, H, I, J?
  - Level of F, G, H, I, J is 3.
- What is the level of K, L, M?
  - Level of K, L, M is 4.
- What is the depth of node F?
  - Depth of node F is 2.
- What is the depth of node L?
  - Depth of node L is 3.
- What is the height of node F?
  - Height of node F is 1.
- What is the height of L?
  - Height of L is 0.

For the tree in the following figure:



- Which node is the root?
  - A
- Which nodes are leaves?
  - G, H, I, L, M, K
- For node B in the tree of Figure:
  - Name the parent node. – A
  - List the children. - D and E
  - List the siblings - C
  - Compute the depth. – 1
  - Compute the height. – 3
- What is the depth of the tree in the Figure?
  - 4

For the tree in the following figure:



Root - A

Leaf node - E, J, K, H and I

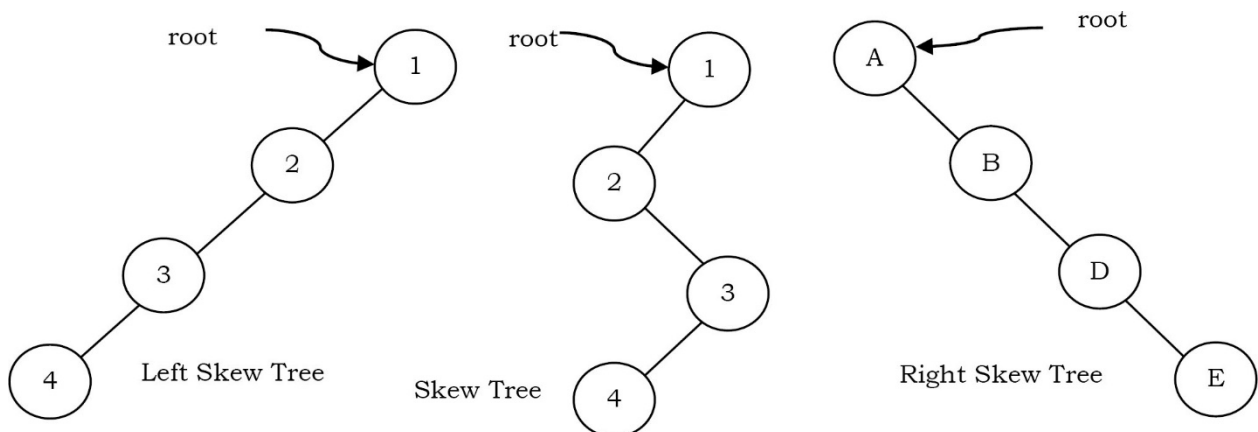
Siblings - B, C, D are siblings of A, and E, F are the siblings of B

Ancestor - A, C and G are the ancestors of K

Depth of G - 2, A - C - G

## 25.4 SKEW TREES

If every node in a tree has only one child (except leaf nodes) then we call such trees skew trees. If every node has only left child, then we call them left skew trees. Similarly, if every node has only right child then we call them right skew trees.



## REVIEW QUESTIONS

1. Define tree.
2. Define root of a tree.
3. Define child of a tree.
4. Define leaves of a tree.
5. Define siblings of a tree.
6. Define path of a tree.
7. Define length of a tree.
8. Define depth of a tree.
9. Define height of a tree.



## 26.1 BINARY TREES

A binary tree is a tree in which no node can have more than two children (each node has zero child, one child or two children). Maximum number of nodes at level  $i$  of a binary tree is  $2^{i-1}$ .

We can visualize a binary tree as consisting of a root and two disjoint binary trees, called the left ( $T_L$ ) and right ( $T_R$ ) subtrees of the root.

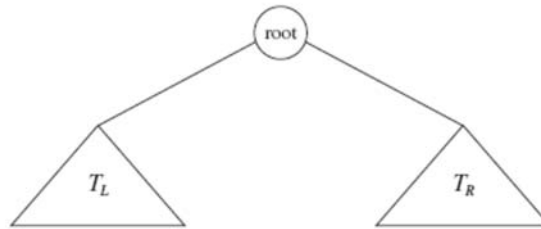


Fig. 26.1 Generic Binary Tree

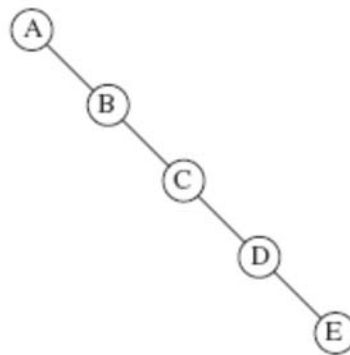


Fig. 26.2 Worst-case Binary Tree

## 26.2 BINARY TREE NODE DECLARATIONS

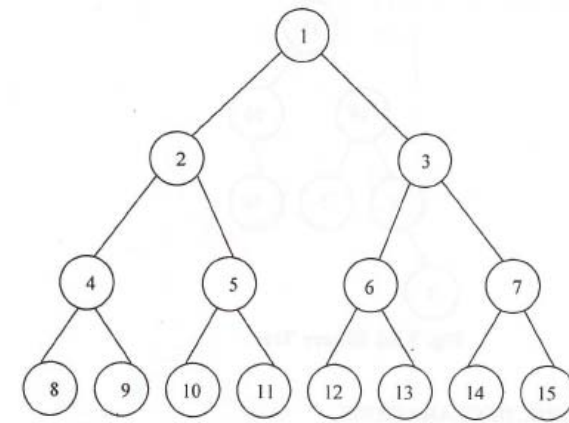
```
struct node
{
    struct node *left;
    int data;
    struct node *right;
};
```

## 26.3 COMPARISON BETWEEN GENERAL TREE AND BINARY TREE

General Tree	Binary Tree
General tree has any number of children.	A Binary tree has not more than two children.

## 26.4 FULL BINARY TREE (or) PERFECT BINARY TREE

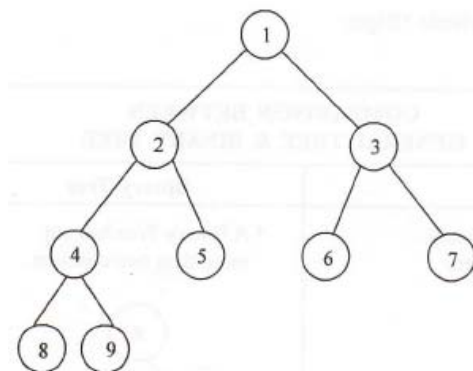
A full binary tree of height  $h$  has  $2^{h+1} - 1$  nodes.



Here height is 3. No. of nodes in full binary tree is:  $2^{3+1} - 1 = 15$  nodes.

## 26.5 COMPLETE BINARY TREE

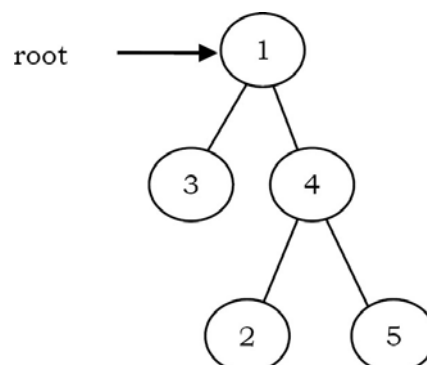
A complete binary tree of height  $h$  has between  $2^h$  and  $2^{h+1} - 1$  nodes. In the bottom level the elements should be filled from left to right.



A full binary tree can be a complete binary tree, but all complete binary tree is not a full binary tree.

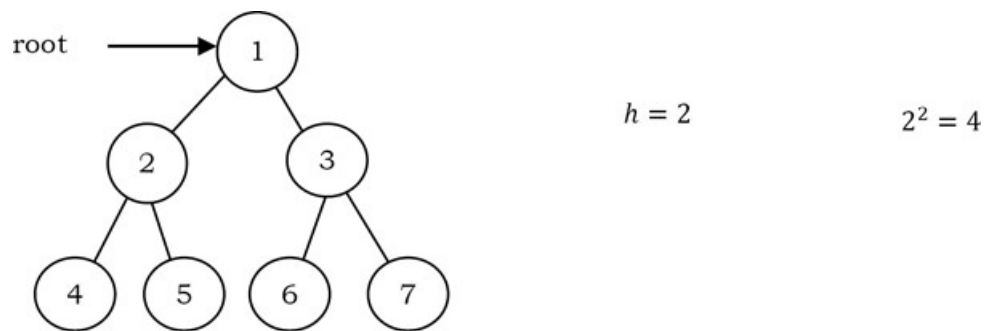
## 26.6 STRICT BINARY TREE

A binary tree is called strict binary tree if each node has exactly two children or no children.



## 26.7 PROPERTIES OF BINARY TREES

For the following properties, let us assume that the height of the tree is  $h$ . Also, assume that root node is at height zero.



From the diagram we can infer the following properties:

- The number of nodes  $n$  in a full binary tree is  $2^{h+1} - 1$ . Since, there are  $h$  levels we need to add all nodes at each level [ $2^0 + 2^1 + 2^2 + \dots + 2^h = 2^{h+1} - 1$ ].
- The number of nodes  $n$  in a complete binary tree is between  $2^h$  (minimum) and  $2^{h+1} - 1$  (maximum).
- The number of leaf nodes in a full binary tree is  $2^h$ .
- The number of NULL links (wasted pointers) in a complete binary tree of  $n$  nodes is  $n + 1$ .

## 26.8 OPERATIONS ON BINARY TREES

- Inserting an element into a tree
- Deleting an element from a tree
- Searching for an element
- Traversing the tree

## 26.9 APPLICATIONS OF BINARY TREES

Following are the some of the applications where binary trees play an important role:

- Expression trees are used in compilers.
- Huffman coding trees that are used in data compression algorithms.
- Binary Search Tree (BST), which supports search, insertion and deletion on a collection of items in  $O(\log n)$  (average).
- Priority Queue (PQ), which supports search and deletion of minimum (or maximum) on a collection of items in logarithmic time (in worst case).

## REVIEW QUESTIONS

1. What is a binary tree?
2. Write an algorithm to declare nodes of a binary tree structure.
3. Compare general tree and binary tree.
4. Show that the maximum number of nodes in a binary tree of height  $h$  is  $2^{h+1} - 1$ .
5. Define full binary tree.
6. Define complete binary tree.
7. Define strict binary tree.

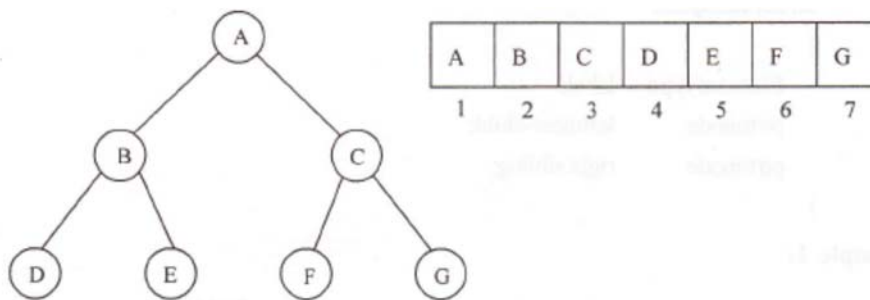
### 27.1 INTRODUCTION

There are two ways for representing binary tree:

- Linear Representation
- Linked Representation

### 27.2 LINEAR REPRESENTATION

The elements are represented using arrays. For any element in position  $i$ , the left child is in position  $2i$ , the right child is in position  $(2i + 1)$ , and the parent is in position  $(i/2)$ .

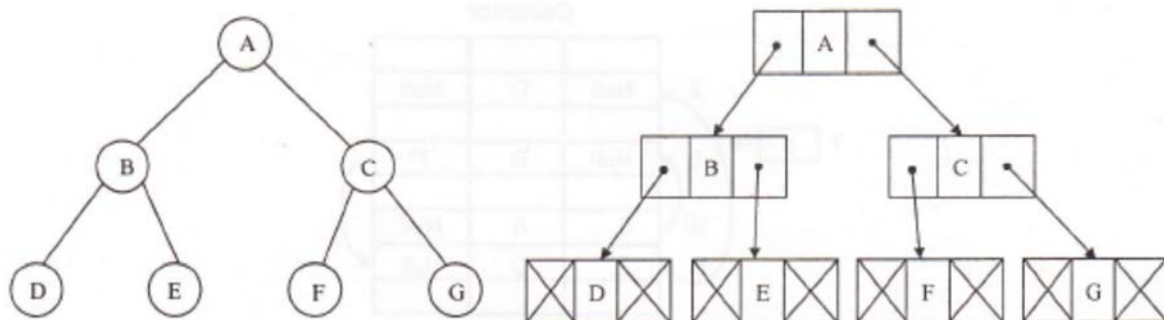


### 27.3 LINKED REPRESENTATION

The elements are represented using pointers. Each node in linked representation has three fields:

- Pointer to the left subtree
- Data field
- Pointer to the right subtree

In leaf nodes, both the pointer fields are assigned as NULL.

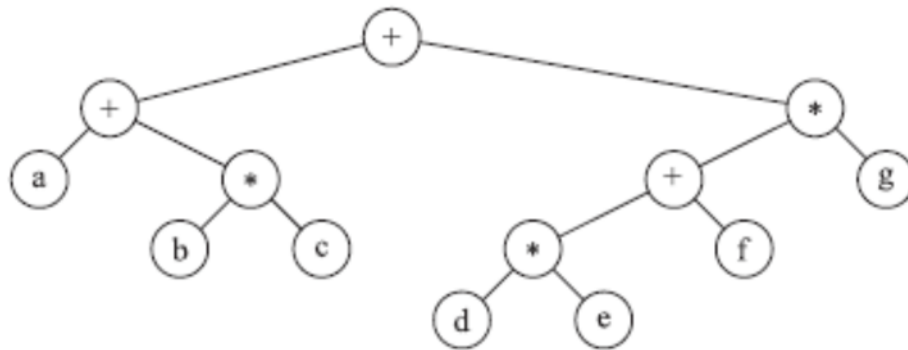


## REVIEW QUESTIONS

1. What are the different ways of representing binary tree?
2. How is binary tree represented using linear representation? Give an example. (or) How is binary tree represented using array? Give an example.
3. How is binary tree represented using linked representation? Give an example.

## 28.1 INTRODUCTION

The leaves of an expression tree are operands, such as constants or variable names, and the other nodes contain operators.



## 28.2 CONSTRUCTION AN EXPRESSION TREE

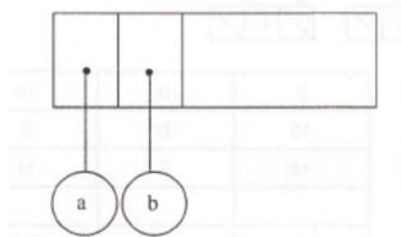
We now give an algorithm to convert a postfix expression into an expression tree. Since we already have an algorithm to convert infix to postfix, we can generate expression trees from the two common types of input. The method we describe strongly resembles the postfix evaluation algorithm.

- Read expression one symbol at a time.
- If the symbol is an operand
  - create a one-node tree and push a pointer to it onto a stack.
- If the symbol is an operator,
  - pop pointers to two trees  $T_1$  and  $T_2$  from the stack ( $T_1$  is popped first) and form a new tree whose root is the operator and whose left and right children point to  $T_2$  and  $T_1$ , respectively.
- A pointer to this new tree is then pushed onto the stack.

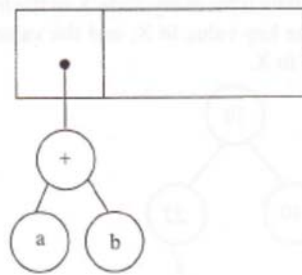
### 28.2.1 Example 1:

$a\ b\ +\ c\ *$

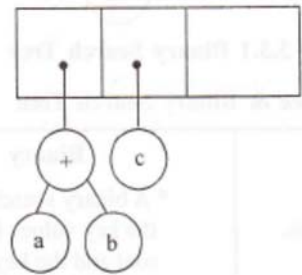
The first two symbols are operand, so create a one node tree and push the pointer on to the stack.



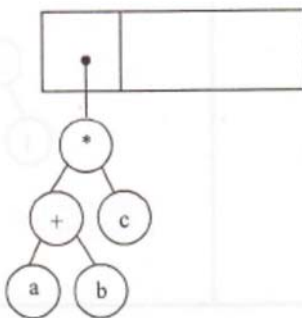
Next '+' symbol is read, so two pointers are popped, a new tree is formed and a pointer to this is pushed on to the stack.



Next the operand c is read, so a one node tree is created and the pointer to it is pushed onto the stack:



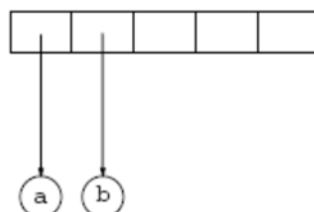
Now '\*' is read, so two trees are merged and the pointer to the final tree is pushed onto the stack.



### 28.2.2 Example 2:

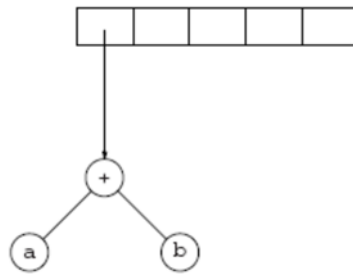
$a b + c d e + * *$

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

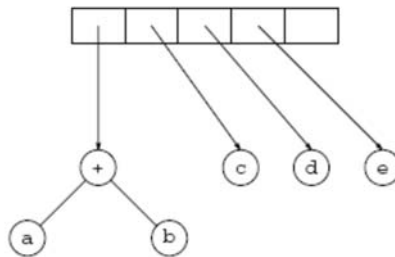


Next, a '+' is read, so two pointers to trees are popped, a new tree is formed, and a pointer to it is pushed onto the stack.

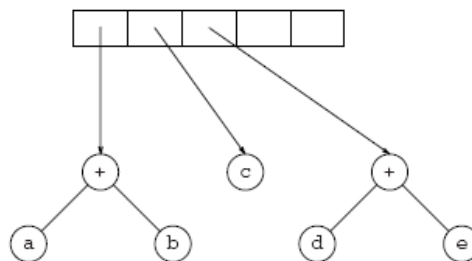




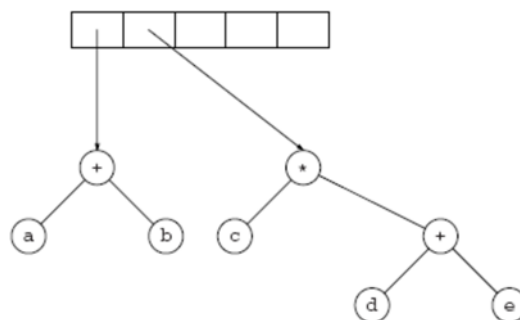
Next, c, d, and e are read, and for each a one-node tree is created and a pointer to the corresponding tree is pushed onto the stack.



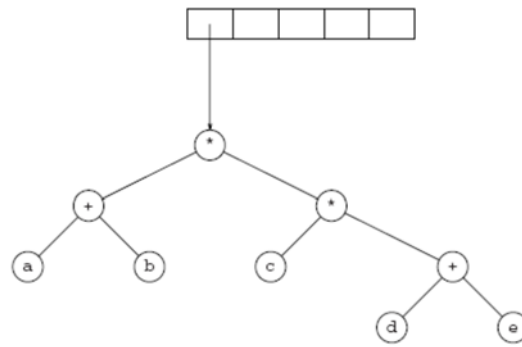
Now a '+' is read, so two trees are merged.



Continuing, a '\*' is read, so we pop two tree pointers and form a new tree with a '\*' as root.



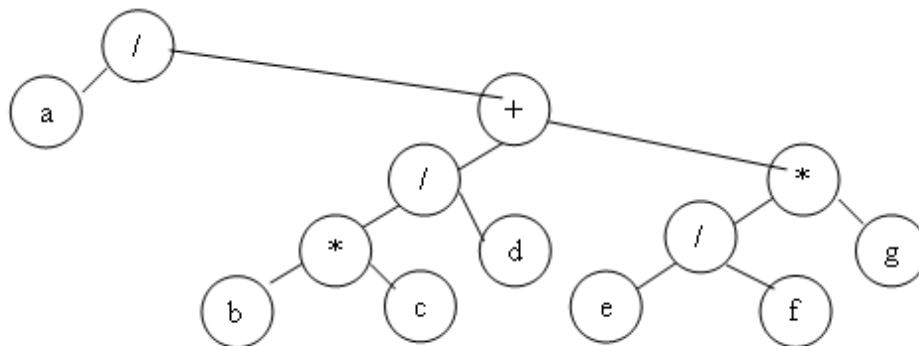
Finally, the last symbol is read, two trees are merged, and a pointer to the final tree is left on the stack.



### 28.2.3 Example 3:

Draw an expression tree for the given infix expression:

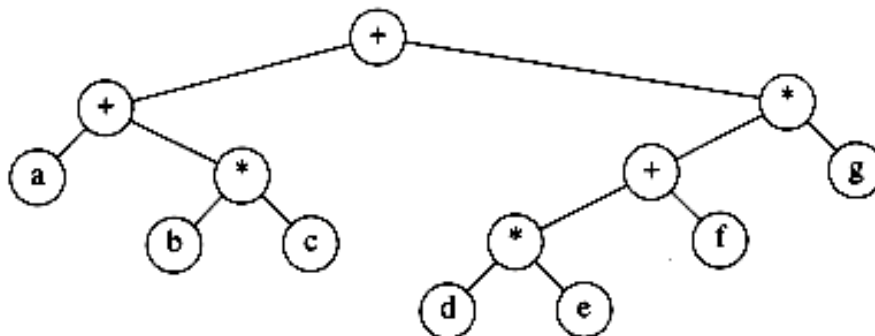
$$(a / (b * c / d + e / f * g))$$



### 28.2.4 Example 4:

Draw an expression tree for the given infix expression:

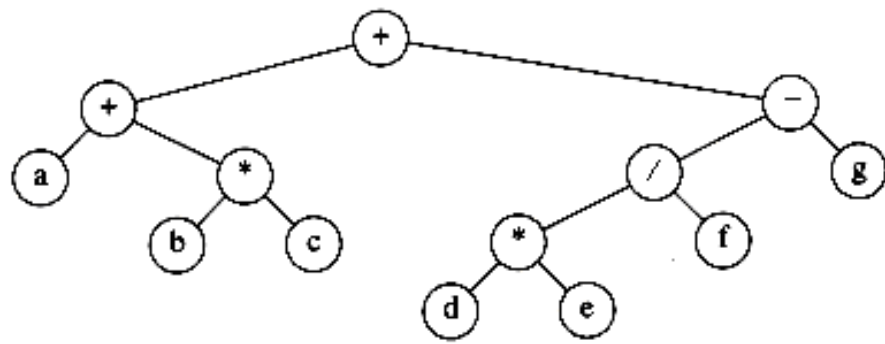
$$(a + b * c) + ((d * e + f) * g)$$



### 28.2.5 Example 5:

Draw an expression tree for the given infix expression:

$$(a + b * c) + (d * e / f - g)$$



### 29.1 INTRODUCTION

The process of visiting all nodes of a tree is called tree traversal. Each node is processed only once but it may be visited more than once.

Three types of tree traversal techniques, namely:

- Inorder traversal
- Preorder traversal
- Postorder traversal

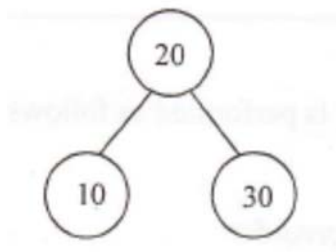
### 29.2 INORDER TRAVERSAL

The inorder traversal of a binary tree is performed as:

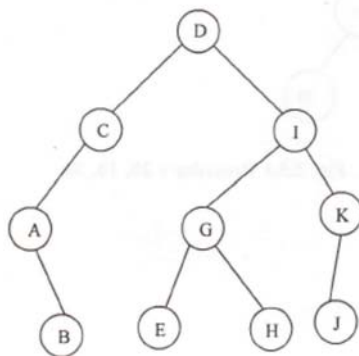
- Traverse the left subtree in inorder
- Visit the root
- Traverse the right subtree in inorder.

The inorder traversal of the binary tree for an arithmetic expression gives the expression in an infix form.

#### 29.2.1 Example



Inorder: 10 20 30



Inorder: A B C D E G H I J K

### 29.2.2 Recursive Routine for Inorder Traversal

```
void Inorder(Node *Tree)
{
    if (Tree != NULL)
    {
        Inorder(Tree->left);
        printf("%d\t", Tree->element);
        Inorder(Tree->right);
    }
}
```

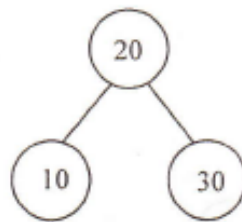
### 29.3 PREORDER TRAVERSAL

The preorder traversal of a binary tree is performed as:

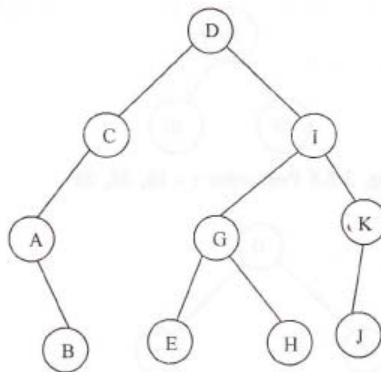
- Visit the root
- Traverse the left subtree in preorder
- Traverse the right subtree in preorder.

The preorder traversal of the binary tree for the given expression gives in prefix form.

#### 29.3.1 Example



Preorder : 20 10 30



Preorder : D C A B I G E H K J

### 29.3.2 Recursive Routine for Preorder Traversal

```
void Preorder(Node *Tree)
{
    if (Tree != NULL)
    {
        printf("%d\t", Tree->element);
        Preorder(Tree->left);
        Preorder(Tree->right);
    }
}
```

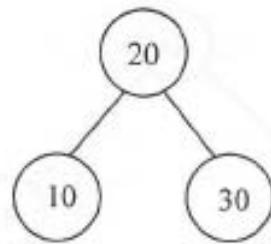
### 29.4 POSTORDER TRAVERSAL

The postorder traversal of a binary tree is performed by:

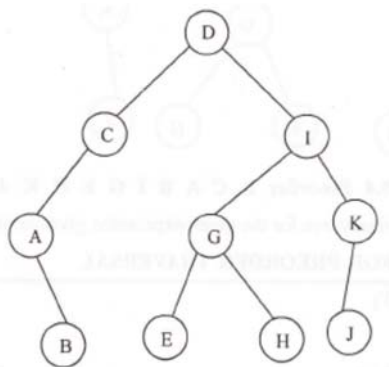
- Traverse the left subtree in postorder.
- Traverse the right subtree in postorder.
- Visit the root.

The postorder traversal of the binary tree for the given expression gives in postfix form.

#### 29.4.1 Example



Postorder : 10 30 20



Postorder : B A C E H G J K I D

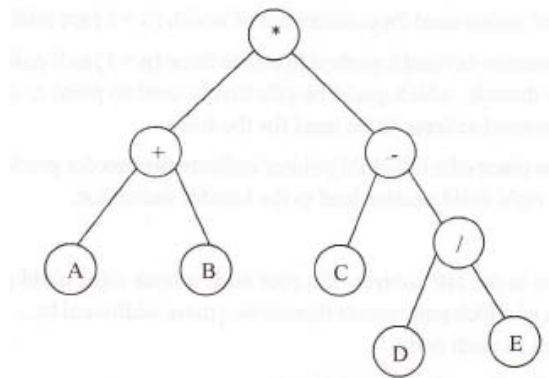
### 29.4.2 Recursive Routine for Postorder Traversal

```
void Postorder(Node *Tree)
{
    if (Tree != NULL)
    {
        Postorder(Tree->left);
        Postorder(Tree->right);
        printf("%d\t", Tree->element);
    }
}
```

## 29.5 EXAMPLES

### 29.5.1 Example-1

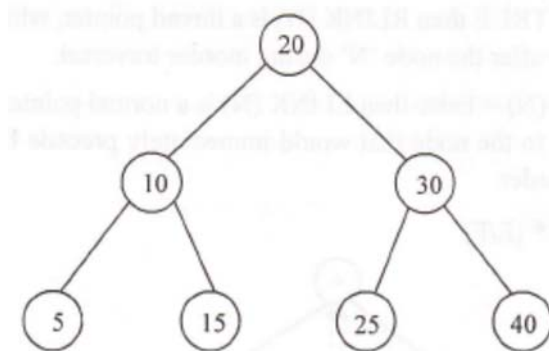
Traverse the given tree using inorder, preorder and postorder traversals.



- Inorder
  - $A + B * C - D / E$
- Preorder
  - $* + A B - C / D E$
- Postorder
  - $A B + C D E / - *$

### 29.5.2 Example-2

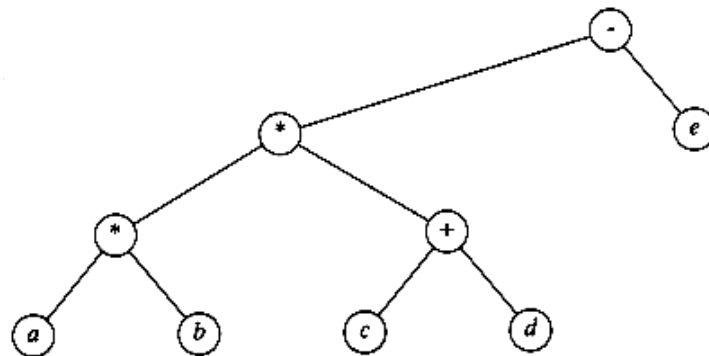
Traverse the given tree using inorder, preorder and postorder traversals.



- Inorder
  - 5 10 15 20 25 30 40
- Preorder
  - 20 10 5 15 30 25 40
- Postorder
  - 5 15 10 25 40 30 20

### 29.5.3 Example-3

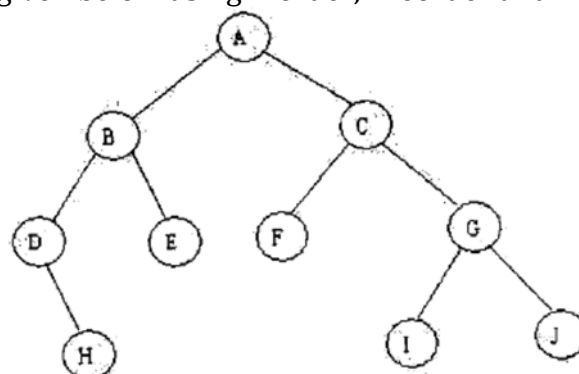
Give the prefix, infix, and postfix expressions corresponding to the tree in following Figure.



- Prefix
  - - \* \* a b + c d e
- Infix
  - a \* b \* c + d - e
- Postfix
  - a b \* c d + \* e -

### 29.5.4 Example-4

Traverse the tree given below using Inorder, Preorder and Postorder traversals.

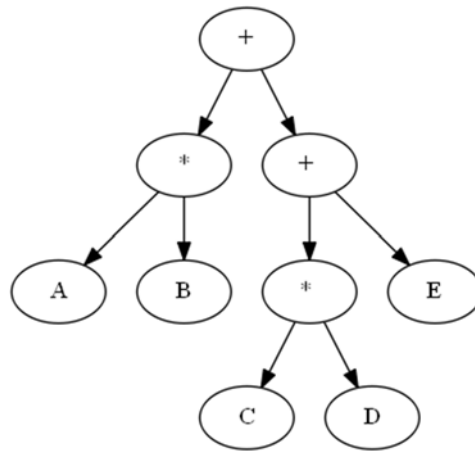


- Inorder
  - D H B E A F C I G J
- Preorder
  - A B D H E C F G I J
- Postorder
  - H D E B F I J G C A



### 29.5.5 Example-5

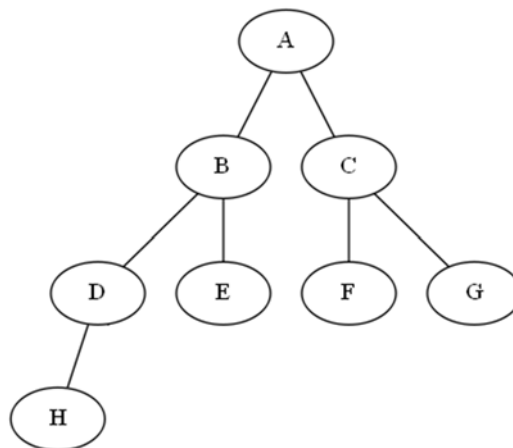
Traverse the tree given below using Inorder, Preorder and Postorder traversals.



- Inorder
  - $a * b + c * d + e$
- Preorder
  - $+ * a b + * c d e$
- Postorder
  - $a b * c d * e ++$

### 29.5.6 Example-6

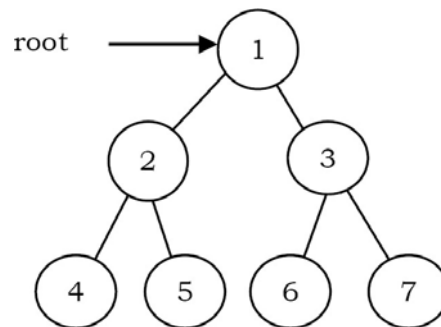
Traverse the tree given below using Inorder, Preorder and Postorder traversals.



- Inorder
  - H D B E A F C G
- Preorder
  - A B D H E C F G
- Postorder
  - H D E B F G C A

### 29.5.7 Example-7

Traverse the tree given below using Inorder, Preorder and Postorder traversals.



- Inorder
  - 4 2 5 1 6 3 7
- Preorder
  - 1 2 4 5 3 6 7
- Postorder
  - 4 5 2 6 7 3 1

### 29.6 PROGRAM

```
#include <stdio.h>
#include <stdlib.h>
struct node
{
    struct node *left;
    int element;
    struct node *right;
};

typedef struct node Node;

Node *Insert(Node *Tree, int e);
void Inorder(Node *Tree);
void Preorder(Node *Tree);
void Postorder(Node *Tree);

int main()
{
    Node *Tree = NULL;
    int n, i, e, ch;
    printf("Enter number of nodes in the tree : ");
    scanf("%d", &n);
    printf("Enter the elements :\n");
    for (i = 1; i <= n; i++)
    {
        scanf("%d", &e);
        Tree = Insert(Tree, e);
    }
}
```

```

do
{
    printf("1. Inorder \n2. Preorder \n3. Postorder \n4. Exit\n");
    printf("Enter your choice : ");
    scanf("%d", &ch);
    switch (ch)
    {
        case 1:
            Inorder(Tree);
            printf("\n");
            break;
        case 2:
            Preorder(Tree);
            printf("\n");
            break;
        case 3:
            Postorder(Tree);
            printf("\n");
            break;
    }
} while (ch <= 3);
return 0;
}

Node *Insert(Node *Tree, int e)
{
    Node *NewNode = malloc(sizeof(Node));
    if (Tree == NULL)
    {
        NewNode->element = e;
        NewNode->left = NULL;
        NewNode->right = NULL;
        Tree = NewNode;
    }
    else if (e < Tree->element)
    {
        Tree->left = Insert(Tree->left, e);
    }
    else if (e > Tree->element)
    {
        Tree->right = Insert(Tree->right, e);
    }
    return Tree;
}

```

```

void Inorder(Node *Tree)
{
    if (Tree != NULL)
    {
        Inorder(Tree->left);
        printf("%d\t", Tree->element);
        Inorder(Tree->right);
    }
}

void Preorder(Node *Tree)
{
    if (Tree != NULL)
    {
        printf("%d\t", Tree->element);
        Preorder(Tree->left);
        Preorder(Tree->right);
    }
}

void Postorder(Node *Tree)
{
    if (Tree != NULL)
    {
        Postorder(Tree->left);
        Postorder(Tree->right);
        printf("%d\t", Tree->element);
    }
}

```

**OUTPUT**

## **REVIEW QUESTIONS**

1. What is traversal? How traversal is performed in binary tree?
2. What is inorder traversal?
3. What is preorder traversal?
4. What is postorder traversal?

### 30.1 INTRODUCTION

The property that makes a binary tree into a binary search tree is that for every node, X, in the tree, the values of all the keys in its left subtree are smaller than the key value in X, and the values of all the keys in its right subtree are larger than the key value in X.

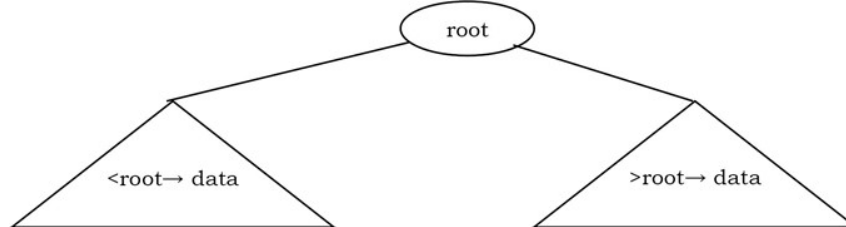
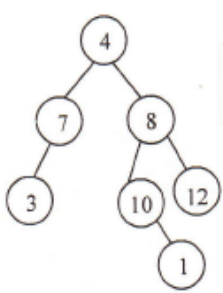
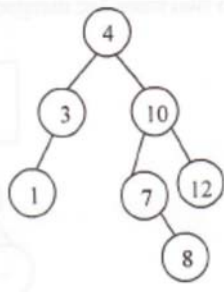


Fig. 30.1 Binary Search Tree



Fig. 30.2 Two binary trees (only the left tree is a search tree)

### 30.2 COMPARISON BETWEEN BINARY TREE AND BINARY SEARCH TREE

Binary Tree	Binary Search Tree
A tree is said to be a binary tree if it has atmost two children.	A binary search tree is a binary tree in which the key values in the left node is less than the root and the key values in the right node is greater than the root.
It does not have any order.	All the elements in the tree can be ordered in some consistent manner.
	

**Note:**

- Every binary search tree is a binary tree.
- All binary trees need not be a binary search tree.

### 30.3 DECLARATION ROUTINE FOR BINARY SEARCH TREE

```
struct node
{
    struct node *left;
    int element;
    struct node *right;
};
```

### 30.4 OPERATIONS ON BINARY SEARCH TREE

Following are the main operations that are supported by binary search trees:

- Find / Find Minimum / Find Maximum element in binary search trees
- Inserting an element in binary search trees
- Deleting an element from binary search trees

### 30.5 FINDING AN ELEMENT IN BINARY SEARCH TREES

Find operation is straightforward in a BST. Start with the root and keep moving left or right using the BST property. If the data we are searching is same as nodes data then we return current node. If the data we are searching is less than nodes data then search left subtree of current node; otherwise search right subtree of current node. If the data is not present, we end up in a NULL link

#### 30.5.1 Algorithm

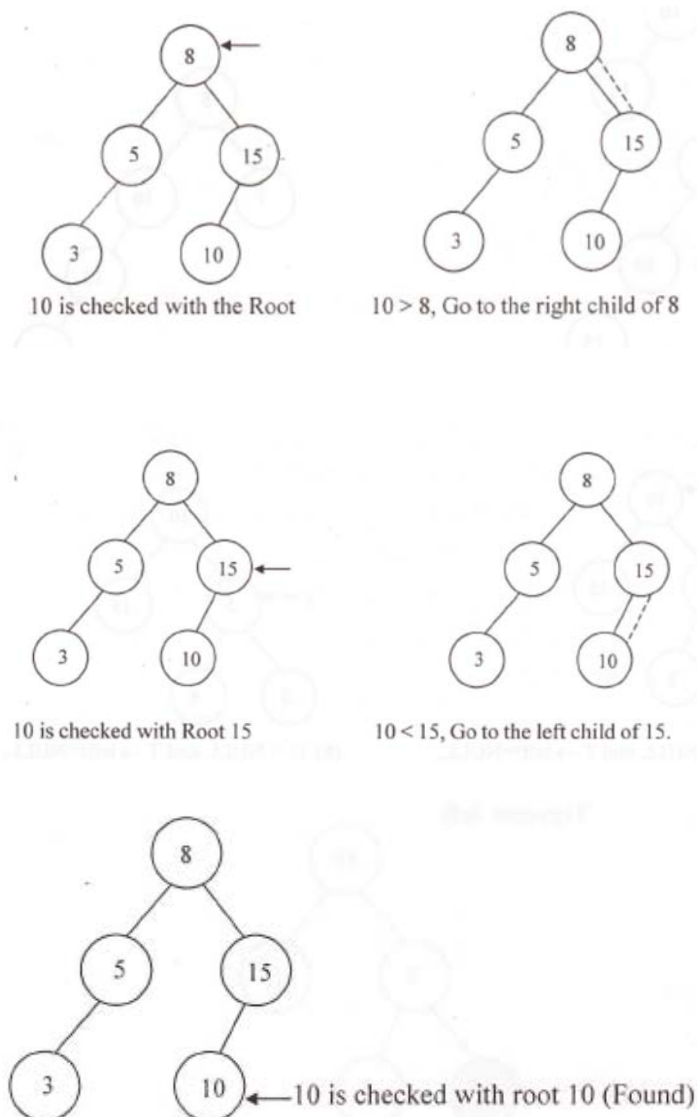
- Check whether the Tree is NULL, if so then return NULL.
- Otherwise check the value e with the Tree node value (i.e. Tree->element)
  - If e is less than Tree->element, traverse the left of Tree recursively.
  - If e is greater than Tree->element, traverse the right of Tree recursively.
  - If e is equal to Tree->element, return Tree.

#### 30.5.2 Routine

```
Node *Find(Node *Tree, int e)
{
    if (Tree == NULL)
        return NULL;
    else if (e < Tree->element)
        return Find(Tree->left, e);
    else if (e > Tree->element)
        return Find(Tree->right, e);
    else
        return Tree;
}
```

### 30.5.3 Example

To find an element 10 (consider,  $e = 10$ ).



### 30.6 FINDING MINIMUM ELEMENT IN BINARY SEARCH TREES

This operation returns the position of the smallest element in the tree. To perform findMin, start at the root and go left as long as there is a left child. The stopping point is the smallest element.

#### 30.6.1 Algorithm

- Check whether the Tree is NULL, if so then return NULL.
- Otherwise check the whether the Tree->left is NULL, if so then return Tree.
- Else traverse the left of Tree recursively.



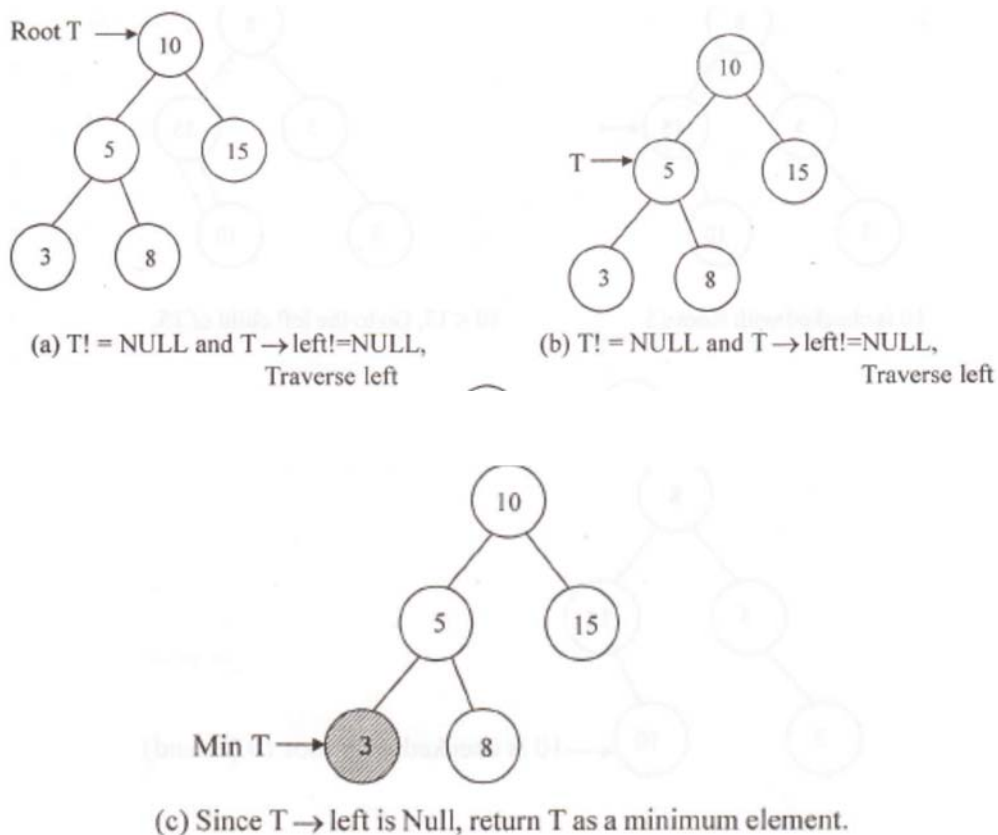
### 30.6.2 Routine (Recursive)

```
Node *FindMin(Node *Tree)
{
    if (Tree == NULL)
        return NULL;
    else if (Tree->left == NULL)
        return Tree;
    else
        return FindMin(Tree->left);
}
```

### 30.6.3 Routine (Non Recursive)

```
Node *FindMin(Node *Tree)
{
    if (Tree != NULL)
        while (Tree->left != NULL)
            Tree = Tree->left;
    return Tree;
}
```

### 30.6.4 Example



## 30.7 FINDING MAXIMUM ELEMENT IN BINARY SEARCH TREES

findMax routine return the position of largest elements in the tree. To perform a findMax, start at the root and go right as long as there is a right child. The stopping point is the largest element.

### 30.7.1 Algorithm

- Check whether the Tree is NULL, if so then return NULL.
- Otherwise, check the whether the Tree->right is NULL, if so then return Tree.
- Else traverse the right of Tree recursively.

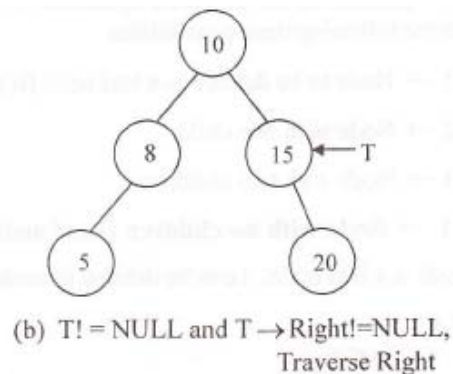
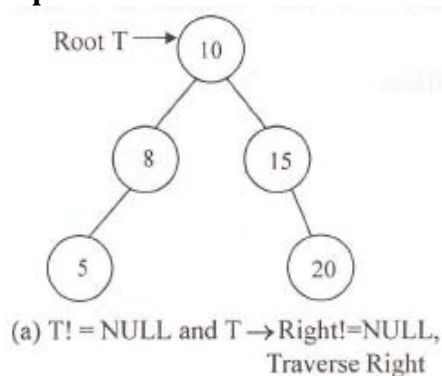
### 30.7.2 Routine (Recursive)

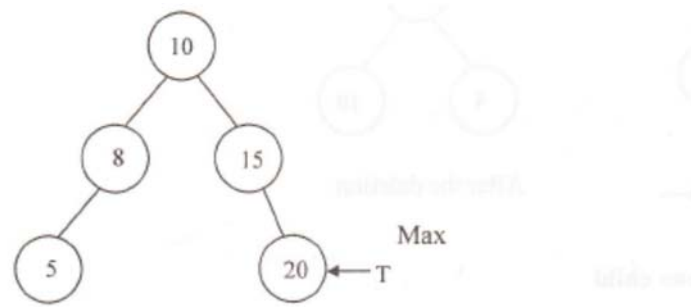
```
Node *FindMax(Node *Tree)
{
    if (Tree == NULL)
        return NULL;
    else if (Tree->right == NULL)
        return Tree;
    else
        return FindMax(Tree->right);
}
```

### 30.7.3 Routine (Non Recursive)

```
Node *FindMax(Node *Tree)
{
    if (Tree != NULL)
        while (Tree->right != NULL)
            Tree = Tree->right;
    return Tree;
}
```

### 30.7.4 Example





(c) Since  $T \rightarrow \text{Right}$  is NULL, return T as a Maximum element.

## 30.8 INSERTING AN ELEMENT INTO BINARY SEARCH TREE

To insert data into binary search tree, first we need to find the location for that element. We can find the location of insertion by following the same mechanism as that of find operation. While finding the location, if the data is already there then we can simply neglect and come out. Otherwise, insert data at the last location on the path traversed.

### 30.8.1 Algorithm

To insert the element  $e$  into the tree,

- Check with the Tree node.
- If  $e$  is less than the Tree:
  - Traverse the left subtree recursively until it reaches the  $\text{Tree} \rightarrow \text{left}$  equals to NULL. Then  $e$  is placed in  $\text{Tree} \rightarrow \text{left}$ .
- If  $e$  is greater than the root.
  - Traverse the right subtree recursively until it reaches the  $\text{Tree} \rightarrow \text{right}$  equals to NULL. Then  $e$  is placed in  $\text{Tree} \rightarrow \text{right}$ .

### 30.8.2 Routine

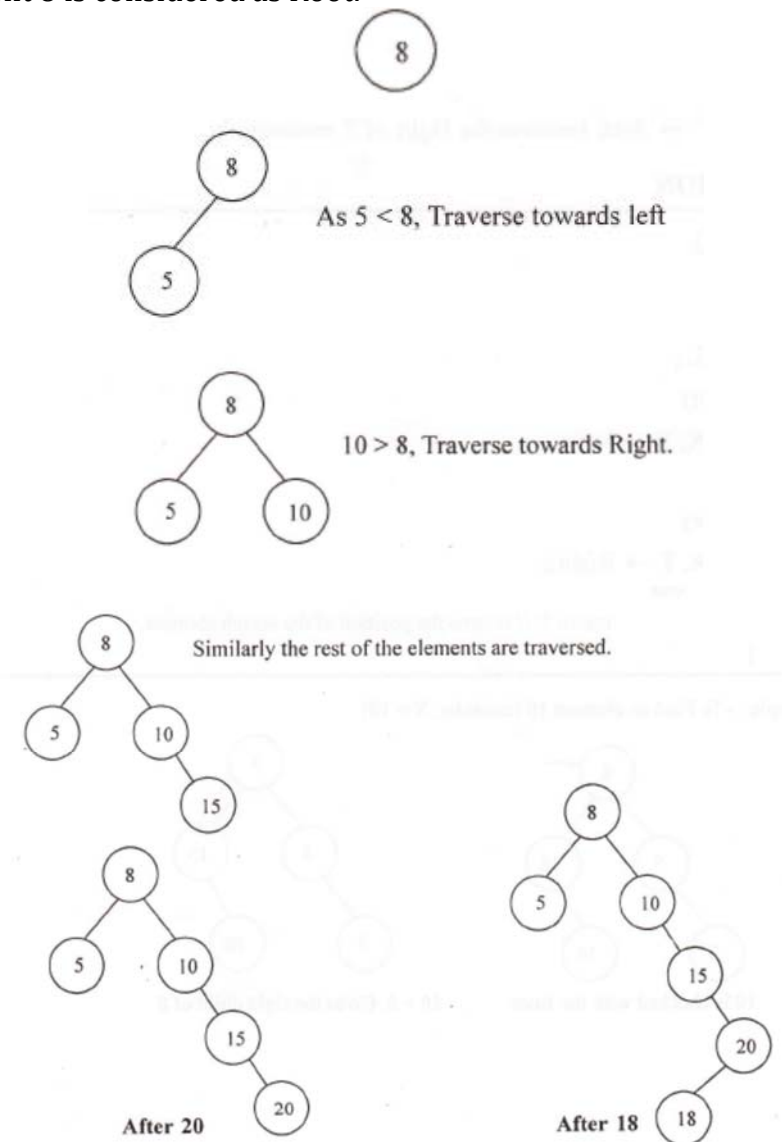
```

Node *Insert(Node *Tree, int e)
{
    Node *NewNode = malloc(sizeof(Node));
    if (Tree == NULL)
    {
        NewNode->element = e;
        NewNode->left = NULL;
        NewNode->right = NULL;
        Tree = NewNode;
    }
    else if (e < Tree->element)
        Tree->left = Insert(Tree->left, e);
    else if (e > Tree->element)
        Tree->right = Insert(Tree->right, e);
    return Tree;
}
  
```

### 30.8.3 Example

To insert 8, 5, 10, 15, 20, 18.

First element 8 is considered as Root.



### 30.8.4 Example

Binary search trees before and after inserting 5.



### 30.9 DELETING AN ELEMENT FROM BINARY SEARCH TREE

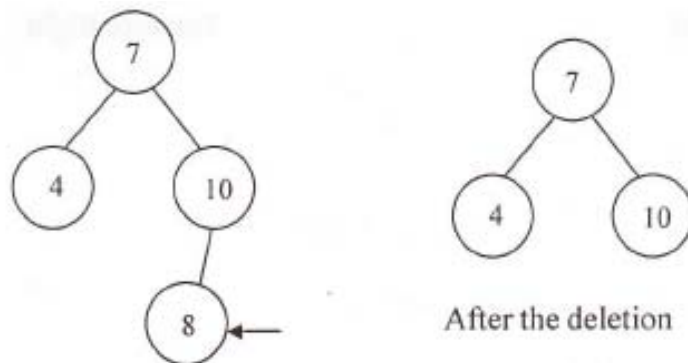
Deletion operation is the complex operation in the Binary search tree. To delete an element, consider the following three possibilities.

- Case 1:
  - Node to be deleted is a leaf node (ie) no children.
- Case 2:
  - Node with one child.
- Case 3:
  - Node with two children.

#### 30.9.1 Case 1: Node with No Children (Leaf Node)

If the node is a leaf node, it can be deleted immediately.

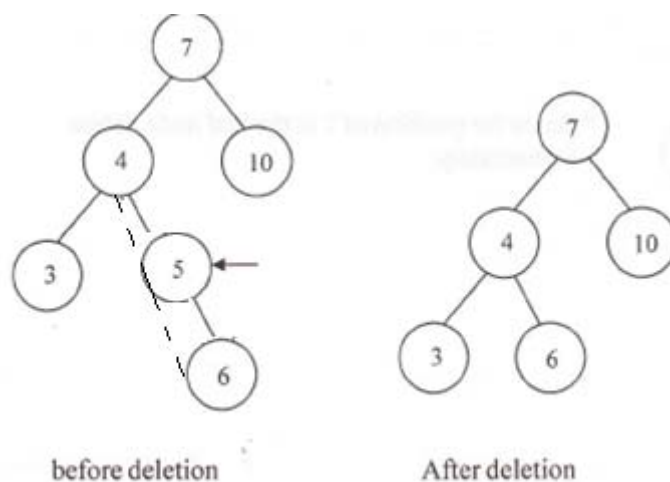
Delete: 8



#### 30.9.2 Case 2: Node with One Child

If the node has one child, it can be deleted by adjusting its parent pointer that points to its child node.

To delete 5, the pointer currently pointing the node 5 is now made to its child node 6.



Deletion of a node (4) with one child, before and after.

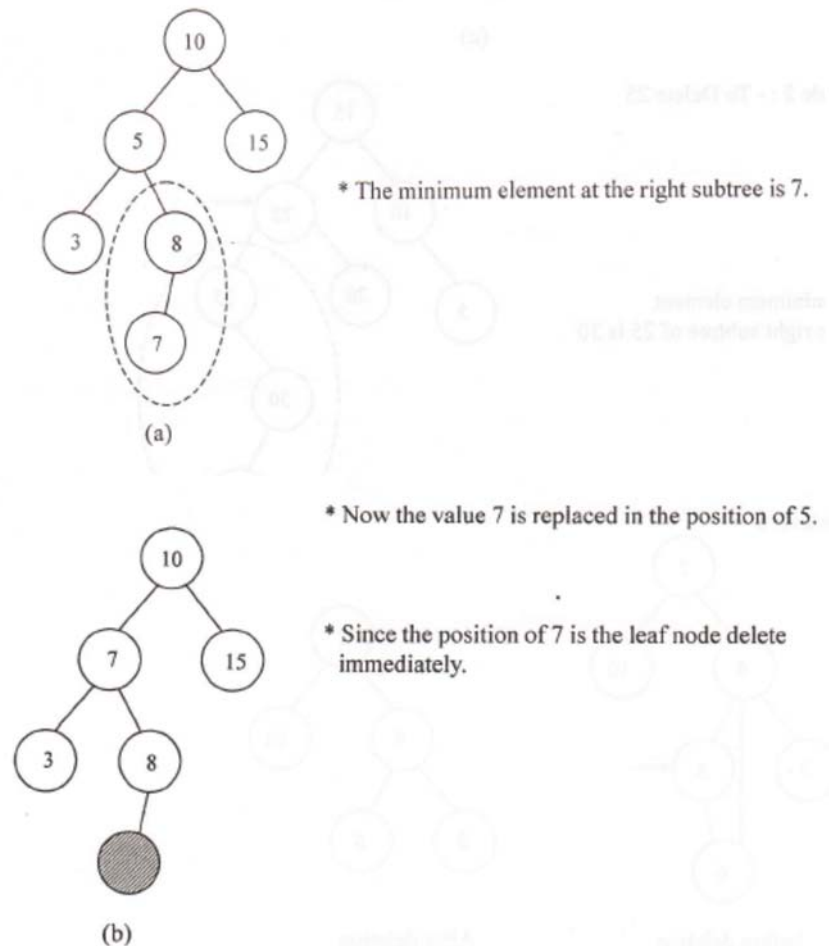


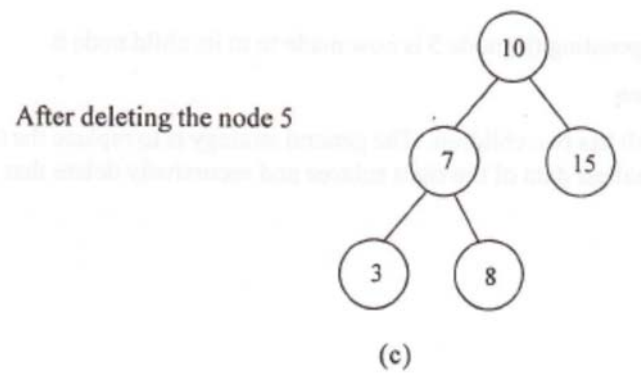
### 30.9.3 Case 3: Node with Two Children

It is difficult to delete a node which has two children. The general strategy is to replace the data of the node to be deleted with its smallest data of the right subtree and recursively delete that node.

### 30.9.4 Example

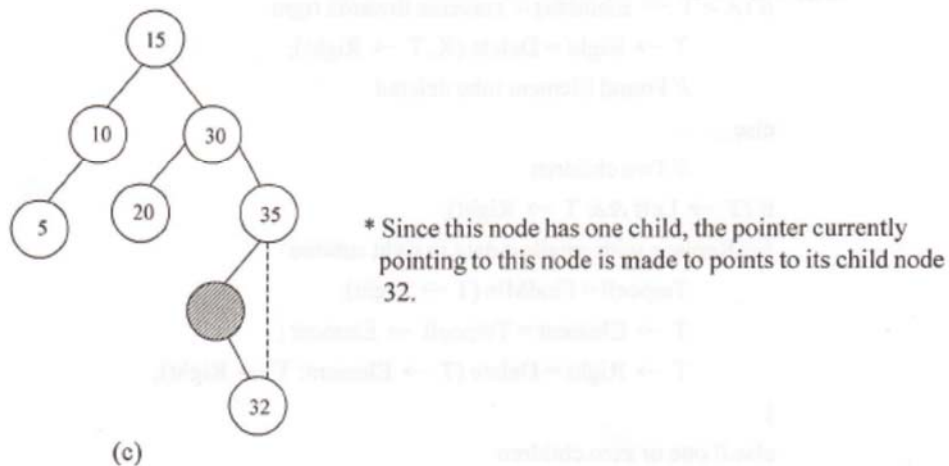
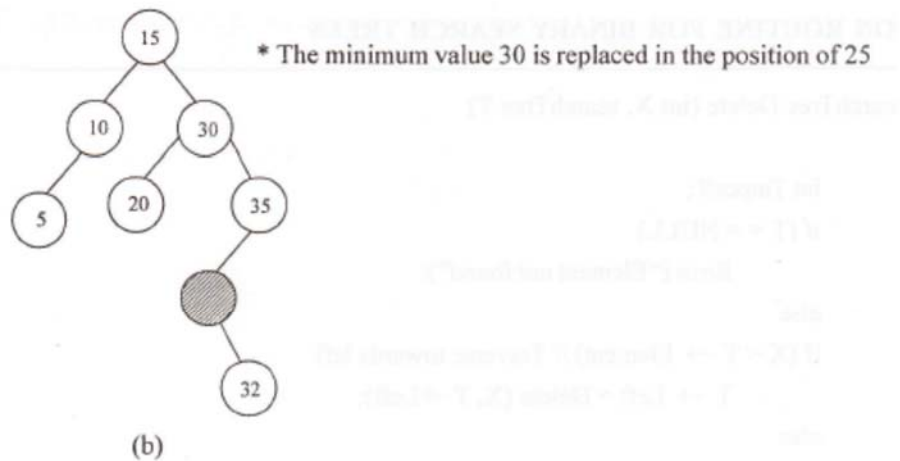
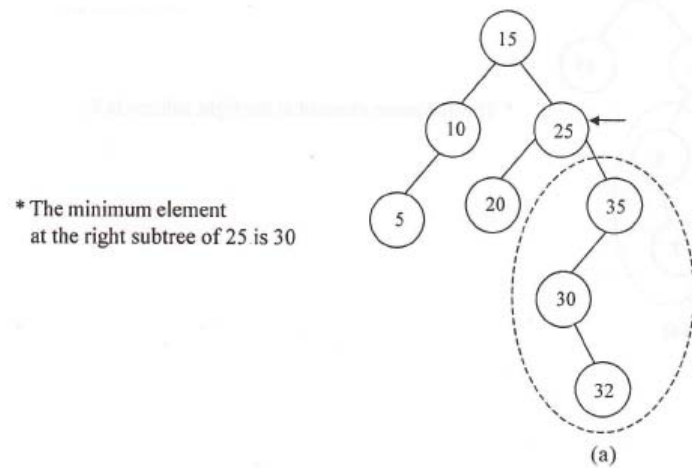
To Delete 5.

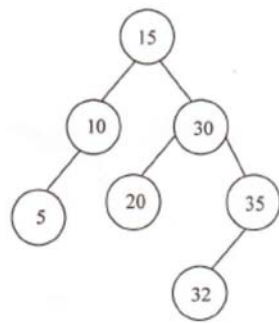




### 30.9.5 Example

To delete 25.



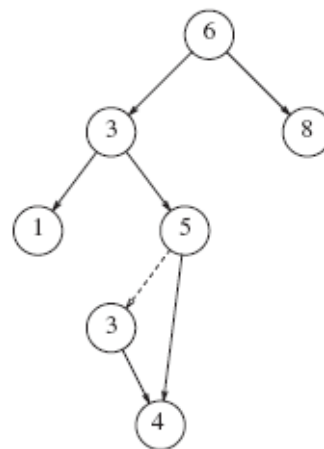
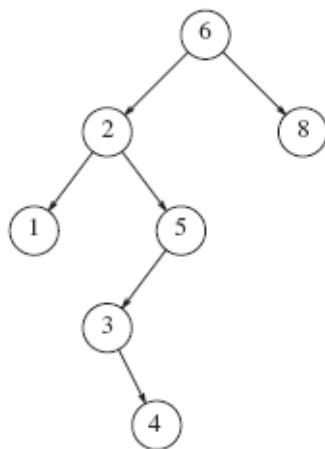


Binary Search Tree after deleting 25

(d)

### 30.9.6 Example

Deletion of a node (2) with two children, before and after.



### 30.10 ROUTINE

```
Node *Delete(Node *Tree, int e)
{
    Node *TempNode = malloc(sizeof(Node));
    if (e < Tree->element)
    {
        Tree->left = Delete(Tree->left, e);
    }
    else if (e > Tree->element)
    {
        Tree->right = Delete(Tree->right, e);
    }
    else if (Tree->left && Tree->right)
    {
        TempNode = FindMin(Tree->right);
        Tree->element = TempNode->element;
        Tree->right = Delete(Tree->right, Tree->element);
    }
}
```



```

        else
        {
            TempNode = Tree;
            if(Tree->left == NULL)
                Tree = Tree->right;
            else if (Tree->right == NULL)
                Tree = Tree->left;
            free(TempNode);
        }
        return Tree;
    }

Node *FindMin(Node *Tree)
{
    if (Tree != NULL)
    {
        if (Tree->left == NULL)
            return Tree;
        else
            FindMin(Tree->left);
    }
}

```

### 30.11 APPLICATIONS

The important application of binary search trees is their use in searching.

### 30.12 PROGRAM (BST-FIND OPERATIONS – METHOD – I)

```
#include <stdio.h>
#include <stdlib.h>
struct node
{
    struct node *left;
    int element;
    struct node *right;
};
typedef struct node Node;
Node *Insert(Node *Tree, int e);
void Find(Node *Tree, int e);
void FindMin(Node *Tree);
void FindMax(Node *Tree);

int main()
{
    Node *Tree = NULL;
    int n, i, e, ch;
    printf("Enter number of nodes in the tree : ");
    scanf("%d", &n);
    printf("Enter the elements :\n");
    for (i = 1; i <= n; i++)
    {
        scanf("%d", &e);
        Tree = Insert(Tree, e);
    }
    do
    {
        printf("1. Find \n2. Find Min \n3. Find Max \n4. Exit\n");
        printf("Enter your choice : ");
        scanf("%d", &ch);
        switch (ch)
        {
            case 1:
                printf("Enter the element to find : ");
                scanf("%d", &e);
                Find(Tree, e);
                printf("\n");
                break;
            case 2:
                FindMin(Tree);
                break;
            case 3:
                FindMax(Tree);
                break;
        }
    } while (ch <= 3);
    return 0;
}
```

```

Node *Insert(Node *Tree, int e)
{
    Node *NewNode = malloc(sizeof(Node));
    if (Tree == NULL)
    {
        NewNode->element = e;
        NewNode->left = NULL;
        NewNode->right = NULL;
        Tree = NewNode;
    }
    else if (e < Tree->element)
        Tree->left = Insert(Tree->left, e);
    else if (e > Tree->element)
        Tree->right = Insert(Tree->right, e);
    return Tree;
}

void Find(Node *Tree, int e)
{
    if (Tree == NULL)
        printf("Element is not found...!");
    else if (e < Tree->element)
        Find(Tree->left, e);
    else if (e > Tree->element)
        Find(Tree->right, e);
    else
        printf("Element is found...!");
}

void FindMin(Node *Tree)
{
    if (Tree == NULL)
        printf("Tree is empty...!");
    else if (Tree->left == NULL)
        printf("%d\n", Tree->element);
    else
        FindMin(Tree->left);
}

void FindMax(Node *Tree)
{
    if (Tree == NULL)
        printf("Tree is empty...!");
    else if (Tree->right == NULL)
        printf("%d\n", Tree->element);
    else
        FindMax(Tree->right);
}

```

## OUTPUT

Enter number of nodes in the tree : 6

Enter the elements :

6

2

8

1

4

3

1. Find

2. Find Min

3. Find Max

4. Exit

Enter your choice : 1

Enter the element to find : 2

Element is found...!

1. Find

2. Find Min

3. Find Max

4. Exit

Enter your choice : 1

Enter the element to find : 9

Element is not found...!

1. Find

2. Find Min

3. Find Max

4. Exit

Enter your choice : 2

1

1. Find

2. Find Min

3. Find Max

4. Exit

Enter your choice : 3

8

1. Find

2. Find Min

3. Find Max

4. Exit

Enter your choice : 4

### 30.13 PROGRAM (BST-FIND OPERATIONS – METHOD – II)

```
#include <stdio.h>
#include <stdlib.h>

struct node
{
    struct node *left;
    int element;
    struct node *right;
};
typedef struct node Node;

Node *Insert(Node *Tree, int e);
Node *Find(Node *Tree, int e);
Node *FindMin(Node *Tree);
Node *FindMax(Node *Tree);

int main()
{
    Node *Tree = NULL;
    Node *Result = NULL;
    int n, i, e, ch;
    printf("Enter number of nodes in the tree : ");
    scanf("%d", &n);
    printf("Enter the elements :\n");
    for (i = 1; i <= n; i++)
    {
        scanf("%d", &e);
        Tree = Insert(Tree, e);
    }
    do
    {
        printf("1. Find \n2. Find Min \n3. Find Max \n4. Exit\n");
        printf("Enter your choice : ");
        scanf("%d", &ch);
        switch (ch)
        {
            case 1:
                printf("Enter the element to find : ");
                scanf("%d", &e);
                Result = Find(Tree, e);
                if (Result == NULL)
                    printf("Element is not found...!");
                else
                    printf("Element is found...!");
                printf("\n");
                break;
        }
    }
}
```

```

        case 2:
            Result = FindMin(Tree);
            if (Result == NULL)
                printf("Tree is empty...!");
            else
                printf("%d\n", Result->element);
            break;
        case 3:
            Result = FindMax(Tree);
            if (Result == NULL)
                printf("Tree is empty...!");
            else
                printf("%d\n", Result->element);
            break;
    }
} while (ch <= 3);
return 0;
}

```

```

Node *Insert(Node *Tree, int e)
{
    Node *NewNode = malloc(sizeof(Node));
    if (Tree == NULL)
    {
        NewNode->element = e;
        NewNode->left = NULL;
        NewNode->right = NULL;
        Tree = NewNode;
    }
    else if (e < Tree->element)
        Tree->left = Insert(Tree->left, e);
    else if (e > Tree->element)
        Tree->right = Insert(Tree->right, e);
    return Tree;
}

```

```

Node *Find(Node *Tree, int e)
{
    if (Tree == NULL)
        return NULL;
    else if (e < Tree->element)
        return Find(Tree->left, e);
    else if (e > Tree->element)
        return Find(Tree->right, e);
    else
        return Tree;
}

```

```

Node *FindMin(Node *Tree)
{
    if (Tree == NULL)
        return NULL;
    else if (Tree->left == NULL)
        return Tree;
    else
        return FindMin(Tree->left);
}

Node *FindMax(Node *Tree)
{
    if (Tree == NULL)
        return NULL;
    else if (Tree->right == NULL)
        return Tree;
    else
        return FindMax(Tree->right);
}

```

## OUTPUT

Enter number of nodes in the tree : 6

Enter the elements :

6

2

8

1

4

3

1. Find

2. Find Min

3. Find Max

4. Exit

Enter your choice : 1

Enter the element to find : 2

Element is found...!

1. Find

2. Find Min

3. Find Max

4. Exit

Enter your choice : 1

Enter the element to find : 9

Element is not found...!

1. Find

2. Find Min

3. Find Max

4. Exit

Enter your choice : 2

1

```

1. Find
2. Find Min
3. Find Max
4. Exit
Enter your choice : 3
8
1. Find
2. Find Min
3. Find Max
4. Exit
Enter your choice : 4

```

### 30.14 PROGRAM (BST-DELETE OPERATIONS)

```

#include <stdio.h>
#include <stdlib.h>

struct node
{
    struct node *left;
    int element;
    struct node *right;
};
typedef struct node Node;

Node *Insert(Node *Tree, int e);
void Display(Node *Tree);
Node *Delete(Node *Tree, int e);
Node *FindMin(Node *Tree);

int main()
{
    Node *Tree = NULL;
    int n, i, e;
    printf("Enter number of nodes in the tree : ");
    scanf("%d", &n);
    printf("Enter the elements :\n");
    for (i = 1; i <= n; i++)
    {
        scanf("%d", &e);
        Tree = Insert(Tree, e);
    }
    printf("Tree elements in inorder :\n");
    Display(Tree);
    printf("\nEnter the element to delete : \n");
    scanf("%d", &e);
    Tree = Delete(Tree, e);
    printf("Tree elements in inorder after deletion :\n");
    Display(Tree);
    return 0;
}

```



```

Node *Insert(Node *Tree, int e)
{
    Node *NewNode = malloc(sizeof(Node));
    if (Tree == NULL)
    {
        NewNode->element = e;
        NewNode->left = NULL;
        NewNode->right = NULL;
        Tree = NewNode;
    }
    else if (e < Tree->element)
        Tree->left = Insert(Tree->left, e);
    else if (e > Tree->element)
        Tree->right = Insert(Tree->right, e);
    return Tree;
}

void Display(Node *Tree)
{
    if (Tree != NULL)
    {
        Display(Tree->left);
        printf("%d\t", Tree->element);
        Display(Tree->right);
    }
}

Node *Delete(Node *Tree, int e)
{
    Node *TempNode = malloc(sizeof(Node));
    if (e < Tree->element)
    {
        Tree->left = Delete(Tree->left, e);
    }
    else if (e > Tree->element)
    {
        Tree->right = Delete(Tree->right, e);
    }
    else if (Tree->left && Tree->right)
    {
        TempNode = FindMin(Tree->right);
        Tree->element = TempNode->element;
        Tree->right = Delete(Tree->right, Tree->element);
    }
    else
    {
        TempNode = Tree;
        if (Tree->left == NULL)
            Tree = Tree->right;
        else if (Tree->right == NULL)
            Tree = Tree->left;
    }
}

```

```

        free(TempNode);
    }
    return Tree;
}

Node *FindMin(Node *Tree)
{
    if (Tree != NULL)
    {
        if (Tree->left == NULL)
            return Tree;
        else
            FindMin(Tree->left);
    }
}

```

## OUTPUT

Enter number of nodes in the tree : 7

Enter the elements :

6

2

8

1

5

3

4

Tree elements in inorder :

1      2      3      4      5      6      8

Enter the element to delete :

2

Tree elements in inorder after deletion :

1      3      4      5      6      8

### **REVIEW QUESTIONS**

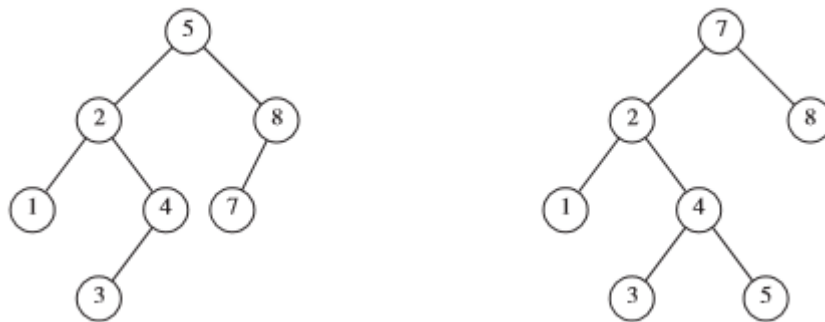
1. What is a binary search tree? (or) When does a binary tree become a binary search tree?
2. Differentiate a binary tree from a binary search tree. (or) Compare binary tree and binary search tree.
3. List out the steps involved in deleting a node from a binary search tree.

### 31.1 INTRODUCTION

An AVL (Adelson-Velskii and Landis) tree is a binary search tree with a balance condition. The balance condition must be easy to maintain, and it ensures that the depth of the tree is  $O(\log N)$ .

An AVL tree is identical to a binary search tree, except that for every node in the tree, the height of the left and right subtrees can differ by at most 1. (The height of an empty tree is defined to be -1.) In Figure 31.1 the tree on the left is an AVL tree but the tree on the right is not. Height information is kept for each node (in the node structure).

A balance factor is the height of the left subtree minus height of the right subtree. For an AVL tree all balance factor should be +1, 0, or -1. If the balance factor of any node in an AVL tree becomes less than -1 or greater than 1, the tree has to be balanced by making either single or double rotations.



**Figure 31.1 Two binary search trees. Only the left tree is AVL.**

Thus, all the tree operations can be performed in  $O(\log N)$  time, except possibly insertion and deletion. When we do an insertion, we need to update all the balancing information for the nodes on the path back to the root, but the reason that insertion is potentially difficult is that inserting a node could violate the AVL tree property. (For instance, inserting 6 into the AVL tree in Figure 4.31 would destroy the balance condition at the node with key 8.) If this is the case, then the property has to be restored before the insertion step is considered over. It turns out that this can always be done with a simple modification to the tree, known as a rotation.

After an insertion, only nodes that are on the path from the insertion point to the root might have their balance altered because only those nodes have their subtrees altered. As we follow the path up to the root and update the balancing information, we may find a node whose new balance violates the AVL condition. We will show how to rebalance the tree at the first (i.e., deepest) such node, and we will prove that this rebalancing guarantees that the entire tree satisfies the AVL property.

Let us call the node that must be rebalanced  $\alpha$ . Since any node has at most two children, and a height imbalance requires that  $\alpha$ 's two subtrees' heights differ by two, it is easy to see that a violation might occur in four cases:

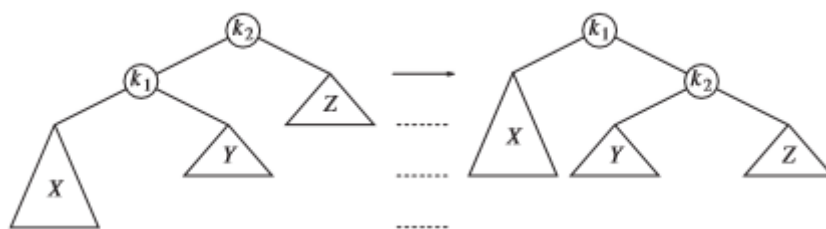
1. An insertion into the left subtree of the left child of  $\alpha$
2. An insertion into the right subtree of the left child of  $\alpha$
3. An insertion into the left subtree of the right child of  $\alpha$
4. An insertion into the right subtree of the right child of  $\alpha$

Cases 1 and 4 are mirror image symmetries with respect to  $\alpha$ , as are cases 2 and 3. Consequently, as a matter of theory, there are two basic cases. From a programming perspective, of course, there are still four cases.

The first case, in which the insertion occurs on the “outside” (i.e., left–left or right–right), is fixed by a single rotation of the tree. The second case, in which the insertion occurs on the “inside” (i.e., left–right or right–left) is handled by the slightly more complex double rotation.

#### 4.4.1 Single Rotation

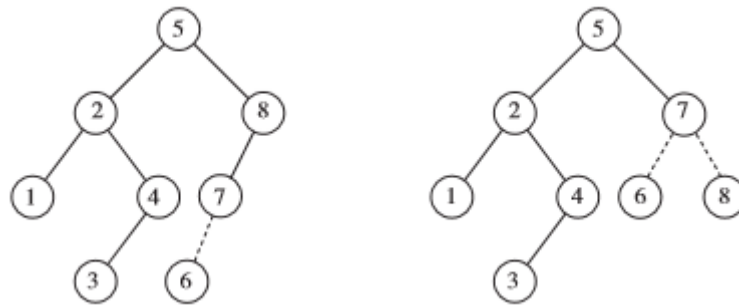
Figure 31.2 shows the single rotation that fixes case 1. The before picture is on the left and the after is on the right. Let us analyze carefully what is going on. Node  $k_2$  violates the AVL balance property because its left subtree is two levels deeper than its right subtree (the dashed lines in the middle of the diagram mark the levels). The situation depicted is the only possible case 1 scenario that allows  $k_2$  to satisfy the AVL property before an insertion but violate it afterwards. Subtree X has grown to an extra level, causing it to be exactly two levels deeper than Z. Y cannot be at the same level as the new X because then  $k_2$  would have been out of balance before the insertion, and Y cannot be at the same level as Z because then  $k_1$  would be the first node on the path toward the root that was in violation of the AVL balancing condition.



**Figure 31.2 Single rotation to fix case 1**

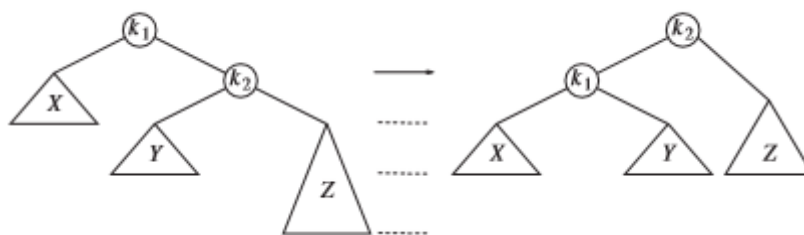
To ideally rebalance the tree, we would like to move X up a level and Z down a level. Note that this is actually more than the AVL property would require. To do this, we rearrange nodes into an equivalent tree as shown in the second part of Figure 32.2. Here is an abstract scenario: Visualize the tree as being flexible, grab the child node  $k_1$ , close your eyes, and shake it, letting gravity take hold. The result is that  $k_1$  will be the new root. The binary search tree property tells us that in the original tree  $k_2 > k_1$ , so  $k_2$  becomes the right child of  $k_1$  in the new tree. X and Z remain as the left child of  $k_1$  and right child of  $k_2$ , respectively. Subtree Y, which holds items that are between  $k_1$  and  $k_2$  in the original tree, can be placed as  $k_2$ 's left child in the new tree and satisfy all the ordering requirements.

As a result of this work, which requires only a few pointer changes, we have another binary search tree that is an AVL tree. This happens because X moves up one level, Y stays at the same level, and Z moves down one level.  $k_2$  and  $k_1$  not only satisfy the AVL requirements, but they also have subtrees that are exactly the same height. Furthermore, the new height of the entire subtree is exactly the same as the height of the original subtree prior to the insertion that caused X to grow. Thus no further updating of heights on the path to the root is needed, and consequently no further rotations are needed. Figure 31.3 shows that after the insertion of 6 into the original AVL tree on the left, node 8 becomes unbalanced. Thus, we do a single rotation between 7 and 8, obtaining the tree on the right.



**Figure 31.3 AVL property destroyed by insertion of 6, then fixed by a single rotation**

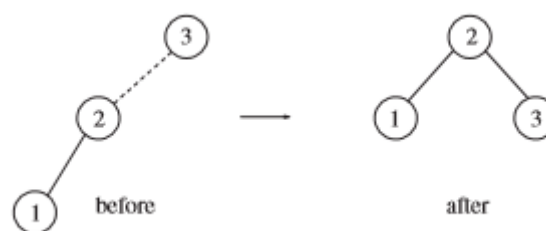
As we mentioned earlier, case 4 represents a symmetric case. Figure 31.4 shows how a single rotation is applied.



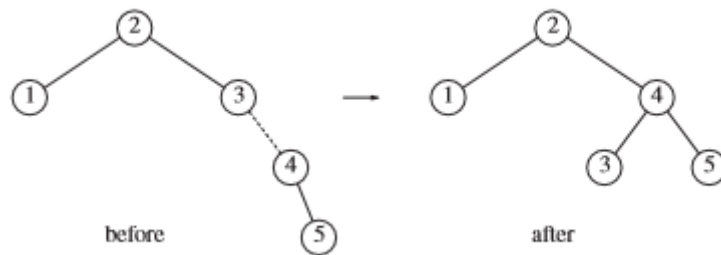
**Figure 31.4 Single rotation fixes case 4**

### Example 1 - Single Rotation

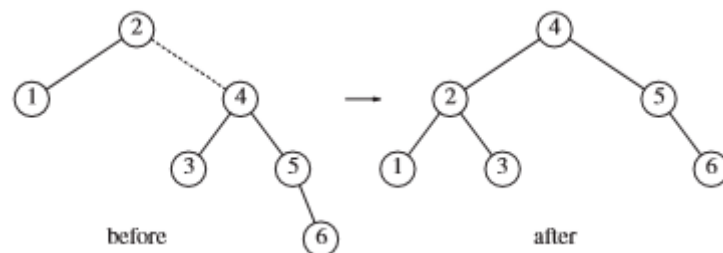
Let us work through a rather long example. Suppose we start with an initially empty AVL tree and insert the items 3, 2, 1, and then 4 through 7 in sequential order. The first problem occurs when it is time to insert item 1 because the AVL property is violated at the root. We perform a single rotation between the root and its left child to fix the problem. Here are the before and after trees:



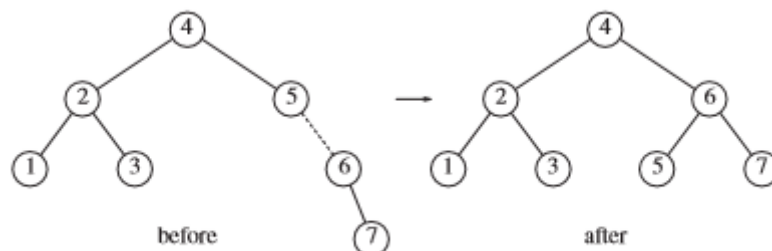
A dashed line joins the two nodes that are the subject of the rotation. Next we insert 4, which causes no problems, but the insertion of 5 creates a violation at node 3 that is fixed by a single rotation. Besides the local change caused by the rotation, the programmer must remember that the rest of the tree has to be informed of this change. Here this means that 2's right child must be reset to link to 4 instead of 3. Forgetting to do so is easy and would destroy the tree (4 would be inaccessible).



Next we insert 6. This causes a balance problem at the root, since its left subtree is of height 0 and its right subtree would be height 2. Therefore, we perform a single rotation at the root between 2 and 4.

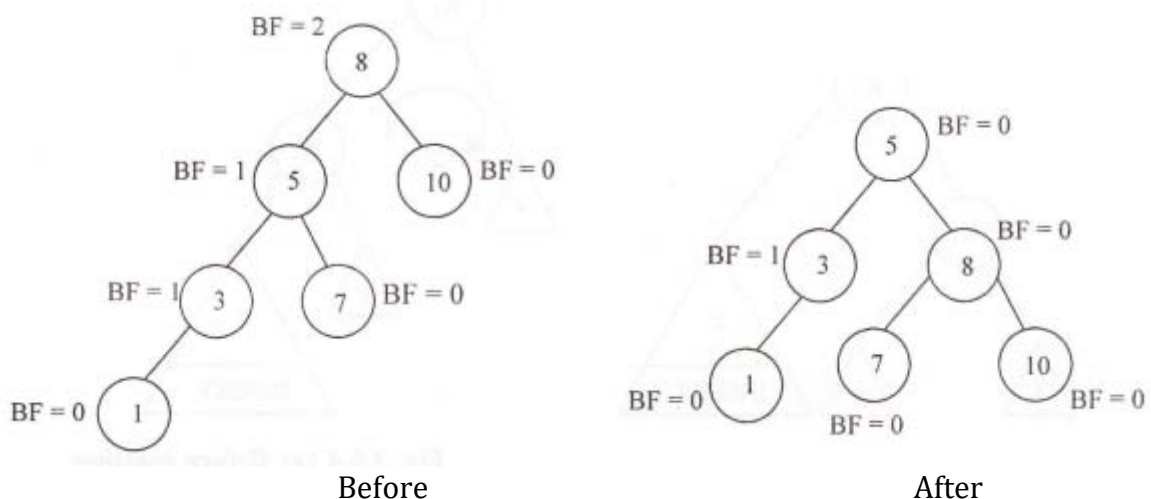


The rotation is performed by making 2 a child of 4 and 4's original left subtree the new right subtree of 2. Every item in this subtree must lie between 2 and 4, so this transformation makes sense. The next item we insert is 7, which causes another rotation:



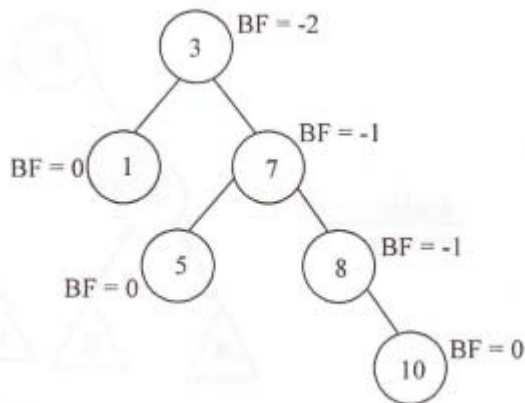
### Example 2 - Single Rotation (Case 1)

Inserting the value 1 in the following AVL Tree makes AVL Tree imbalance.

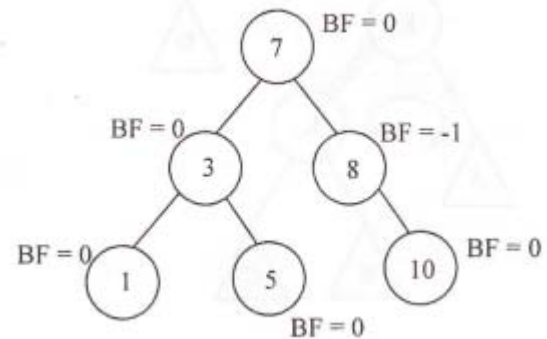


### Example 3 - Single Rotation (Case 4)

Inserting the value 10 in the following AVL Tree makes AVL Tree imbalance.



Before



After

#### 4.4.2 Double Rotation

The algorithm described above has one problem: As Figure 31.5 shows, it does not work for cases 2 or 3. The problem is that subtree Y is too deep, and a single rotation does not make it any less deep. The double rotation that solves the problem is shown in Figure 31.6.

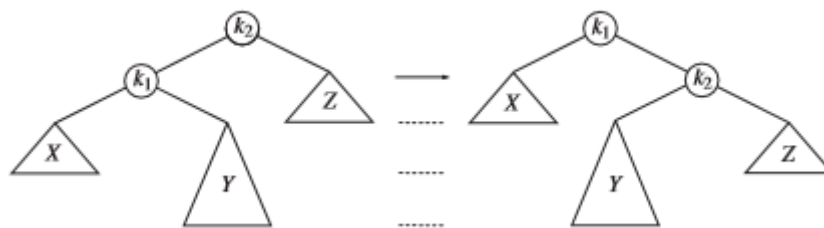


Figure 31.5 Single rotation fails to fix case 2

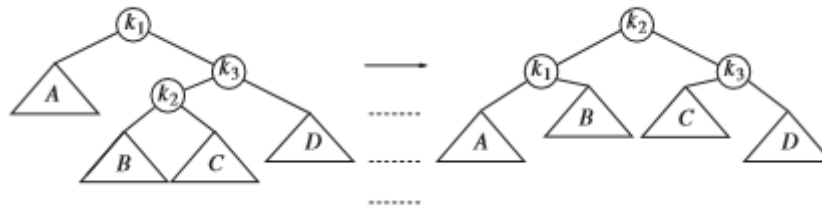


Figure 31.6 Left-right double rotation to fix case 2

The fact that subtree Y in Figure 31.5 has had an item inserted into it guarantees that it is nonempty. Thus, we may assume that it has a root and two subtrees. Consequently, the tree may be viewed as four subtrees connected by three nodes. As the diagram suggests, exactly one of tree B or C is two levels deeper than D (unless all are empty), but we cannot be sure which one. It turns out not to matter; in Figure 31.6, both B and C are drawn at 1 ½ levels below D.



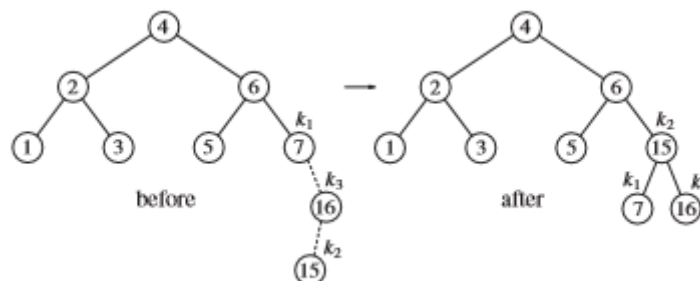
To rebalance, we see that we cannot leave  $k_3$  as the root, and a rotation between  $k_3$  and  $k_1$  was shown in Figure 31.5 to not work, so the only alternative is to place  $k_2$  as the new root. This forces  $k_1$  to be  $k_2$ 's left child and  $k_3$  to be its right child, and it also completely determines the resulting locations of the four subtrees. It is easy to see that the resulting tree satisfies the AVL tree property, and as was the case with the single rotation, it restores the height to what it was before the insertion, thus guaranteeing that all rebalancing and height updating is complete. Figure 31.7 shows that the symmetric case 3 can also be fixed by a double rotation. In both cases the effect is the same as rotating between  $\alpha$ 's child and grandchild, and then between  $\alpha$  and its new child.



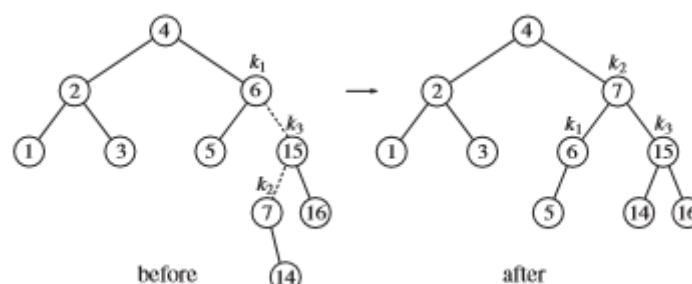
**Figure 31.7 Right-left double rotation to fix case 3**

### Example 1 - Double Rotation

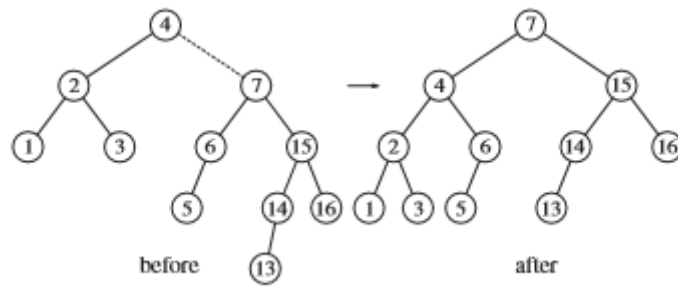
We will continue our previous example by inserting 10 through 16 in reverse order, followed by 8 and then 9. Inserting 16 is easy, since it does not destroy the balance property, but inserting 15 causes a height imbalance at node 7. This is case 3, which is solved by a right-left double rotation. In our example, the right-left double rotation will involve 7, 16, and 15. In this case,  $k_1$  is the node with item 7,  $k_3$  is the node with item 16, and  $k_2$  is the node with item 15. Subtrees A, B, C, and D are empty.



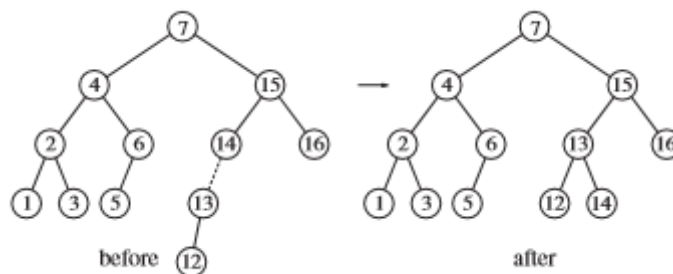
Next we insert 14, which also requires a double rotation. Here the double rotation that will restore the tree is again a right-left double rotation that will involve 6, 15, and 7. In this case,  $k_1$  is the node with item 6,  $k_2$  is the node with item 7, and  $k_3$  is the node with item 15. Subtree A is the tree rooted at the node with item 5; subtree B is the empty subtree that was originally the left child of the node with item 7, subtree C is the tree rooted at the node with item 14, and finally, subtree D is the tree rooted at the node with item 16.



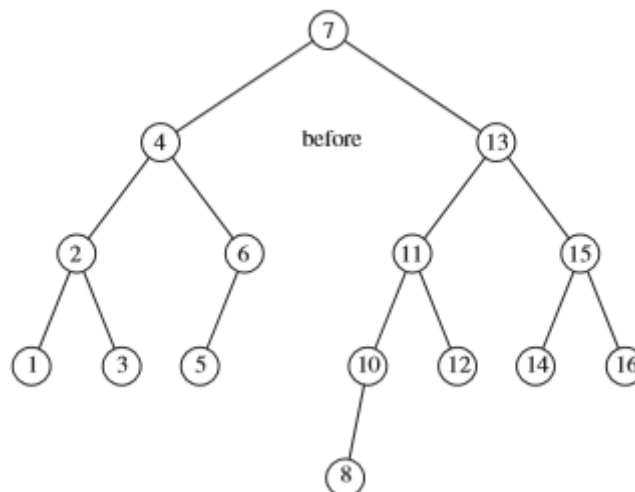
If 13 is now inserted, there is an imbalance at the root. Since 13 is not between 4 and 7, we know that the single rotation will work.



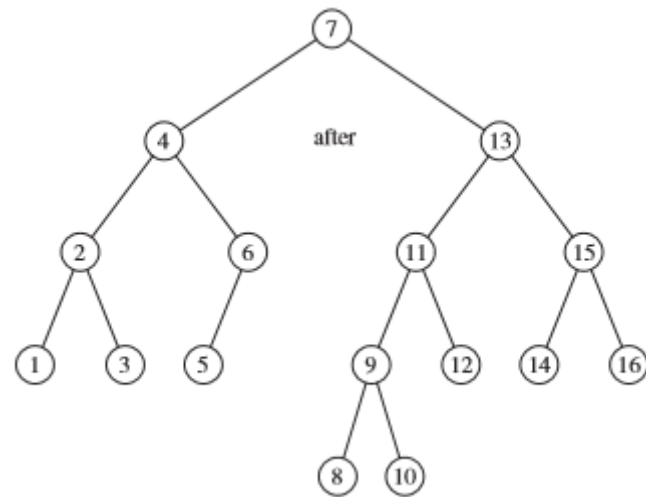
Insertion of 12 will also require a single rotation:



To insert 11, a single rotation needs to be performed, and the same is true for the subsequent insertion of 10. We insert 8 without a rotation, creating an almost perfectly balanced tree:



Finally, we will insert 9 to show the symmetric case of the double rotation. Notice that 9 causes the node containing 10 to become unbalanced. Since 9 is between 10 and 8 (which is 10's child on the path to 9), a double rotation needs to be performed, yielding the following tree:





# RAJALAKSHMI

## ENGINEERING COLLEGE

Website : [www.rajalakshmi.org](http://www.rajalakshmi.org)

Elearning : [www.rajalakshmicolleges.net/moodle](http://www.rajalakshmicolleges.net/moodle)

***Give a man a fish and you feed him for a day.  
Teach him how to fish and you feed him for a lifetime.***