TREE

1-The Concept of The Tree

It implies that we organize the data so that items of information are related by the branches.

Definition: A tree is a finite set of one or more nodes such that:

1. There is a specially designated node called the root.
2. The remaining nodes are partitioned into n ≥ 0 disjoint sets $T1,\ldots, T_n$, where each of these sets is a tree. We call $T1,\ldots, T_n$ the subtrees of the root.

2- Terminology

Root of the tree: The top node of the tree that is not a subtree to other node, and has two children of subtrees.

Node: It is stands for the item of information and the branches to other nodes.

The degree of a node: It is the number of subtrees of the node.

The degree of a tree: It is the maximum degree of the nodes in the tree

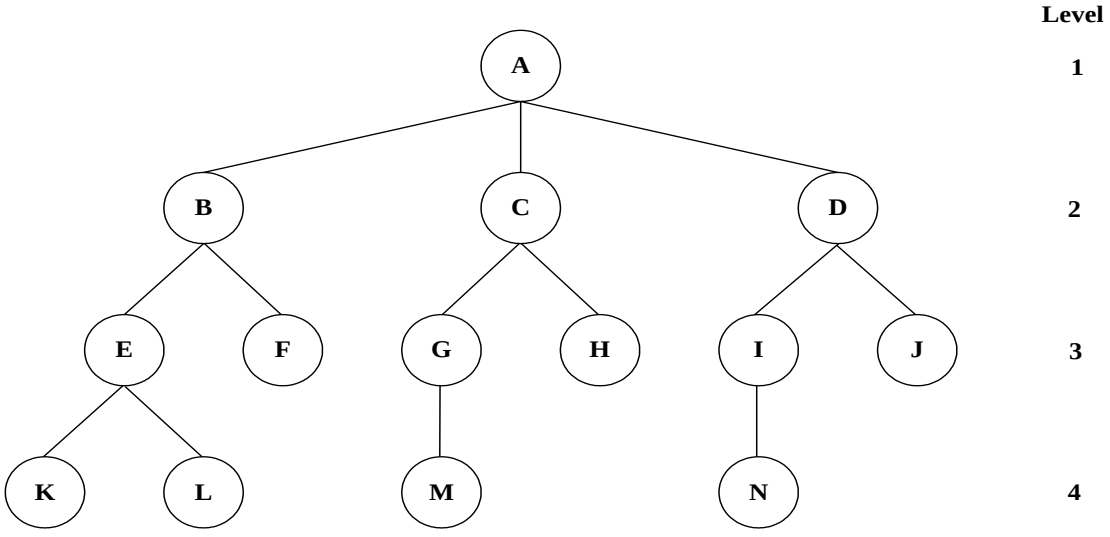
The parent node: a node that has subtrees is the parent of the roots of the subtrees

The child node: a node that is the roots of the subtrees are the children of the node

The Level of the tree: We define the level of a node by initially letting the root be at level one

The depth of a tree: It also called height of a tree. It is the maximum level of any node in the tree

General Tree Graphical Picture:



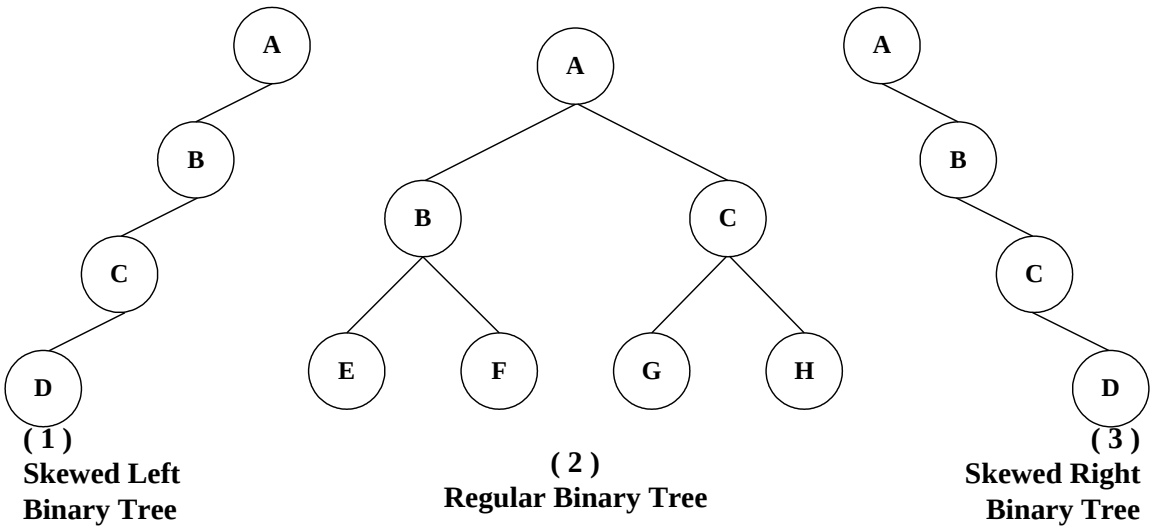
3- Binary Tree

Definition: A binary tree is a finite set of nodes that is either empty or consists of a root and two disjoint binary trees called the left subtree and the right subtree.

Binary Tree Types:

Regular Binary Tree (2)
Skewed Left Binary Tree (1)
Skewed Right Binary Tree (3)

Three Graphical Pictures of the Binary Tree:



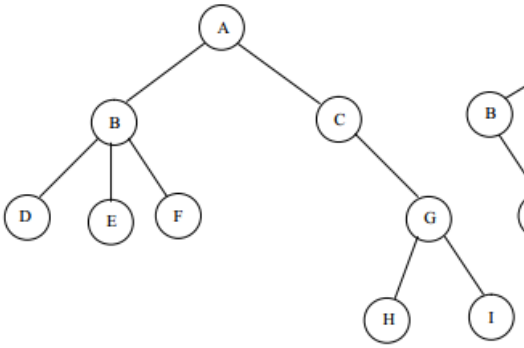
4- Properties of Binary Trees

In particular, we want to find out the maximum number of nodes in a binary tree of depth k, and the number of leaf nodes and the number of nodes of degree two in a binary tree. We present both these observations as lemma.

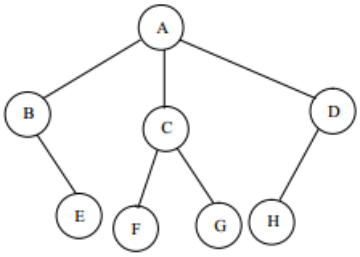
Lemma 1 [Maximum number of nodes]:

(1) The maximum number of nodes on level i of a binary tree is 2^{i-1}, $i \geq 1$
(2) The maximum number of nodes in a binary tree of depth k is $2^k - 1$, $k \geq 1$

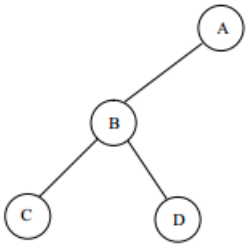
Balanced tree: the tree that all leaves in the same level:



not Balanced tree

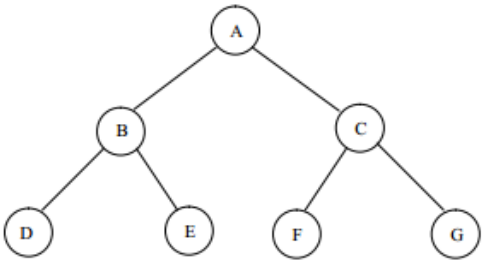


Balanced tree

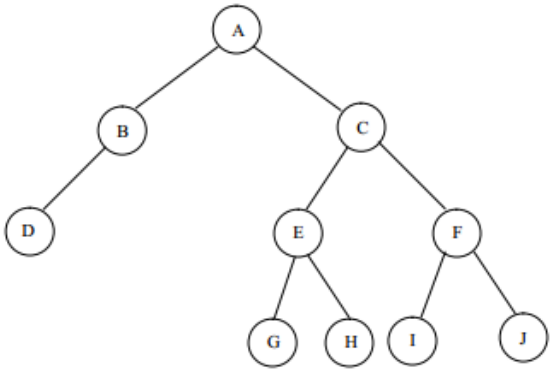


Balanced tree

Full Binary tree: the tree that all leaves in the same level and each node have exactly two leaves:

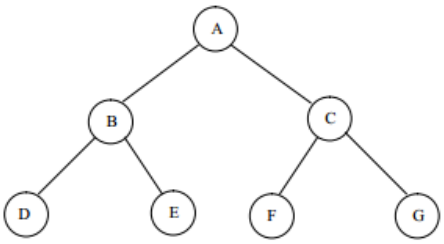


Full Binary tree

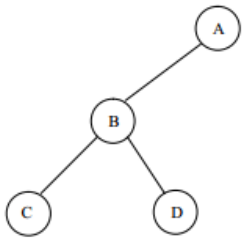


not full binary tree

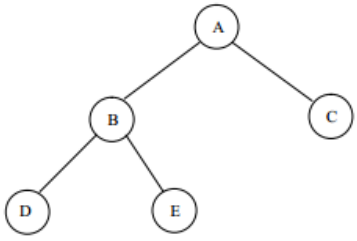
Binary Complete tree: the tree may be full or full to the last level where the leaves at the left side



Full and Binary Complet



not Full and not Binary Complet



not Full and Binary Complet

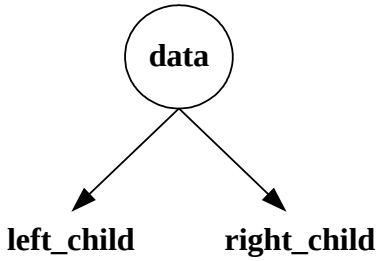
5- Binary Tree Representations

Array Representation

The numbering scheme used in it suggests out first representation of a binary tree in memory. Since the nodes are numbered from 1 to *n*, we can use a one-dimensional array to store the nodes. (We do not use the 0^{th} position of the array.) Using Lemma 1 we can easily determine the locations of the parent, left child, and right child of any node, *i*, in the binary tree.

Linked Representation

While the sequential representation is acceptable for complete binary trees, it wastes space for many other binary trees. In, addition, this representation suffers from the general inadequacies of other sequential representations. Thus, insertion or deletion of nodes from the middle of a tree requires the movement of potentially many nodes to reflect the change in the level of these nodes. We can easily overcome these problems by using a linked representation. Each node has three fields, left_child, data, and right_child as two pictures show the node representation of the binary tree below:



The Node Class

First, we need a class of node objects. These objects contain the data representing the objects being stored and also references to each of the node's two children. Here's how that looks:

```
class Node
{
Int        iData;            // data used as key value
double   fData;           // other data
node      leftChild;       // this node's left child
node      rightChild;      // this node's right child
public void displayNode()
{
   // ( for method body)
}
```

6- Binary Tree Traversals

There are many operations that we can perform on tree, but one that arises frequently is traversing a tree, that is, visiting each node in the tree exactly once. A full traversal produces a linear order for the information in a tree.

Binary Tree Traversals Types

Inorder Traversal (Left, Parent, Right) LNR
Preorder Traversal (Parent, Left, Right) NLR
Postorder Traversal (Left, Right, Parent) LRN

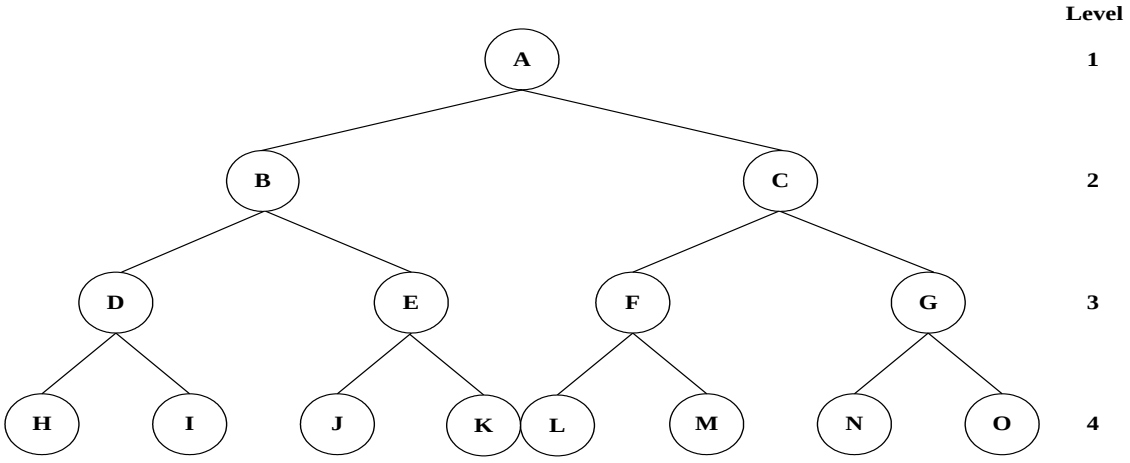
Binary Tree Traversals Functions:

***Inorder Tree Traversal**

Recursive function:

```
private void addInOrder(Node node, Set<K> set)
    {
      if (node == null) return;
      addInOrder(node.left, set);
      set.add(node.key);
      addInOrder(node.right, set);
    }
```

For the following tree



Result of binary tree example:
H, D, I, B, J, E, K, A, L, F, M, C, N, G, O

Preorder Tree Traversal

Recursive function:

```
private void preorder (Node node, Set<K> set)
        {
          if (node == null) return;
         set.add(node.key);
          preorder (node.left, set);
             preorder (node.right, set);
        }
```

Result of binary tree example:
A, B, D, H, I, E, J, K, C, F, L, M, G, N, O

Postorder Tree Traversal

Recursive function:

```
private void postorder (Node node, Set<K> set)
        {
          if (node == null) return;
            postorder (node.left, set);
             postorder (node.right, set);
             set.add(node.key);
        }
```

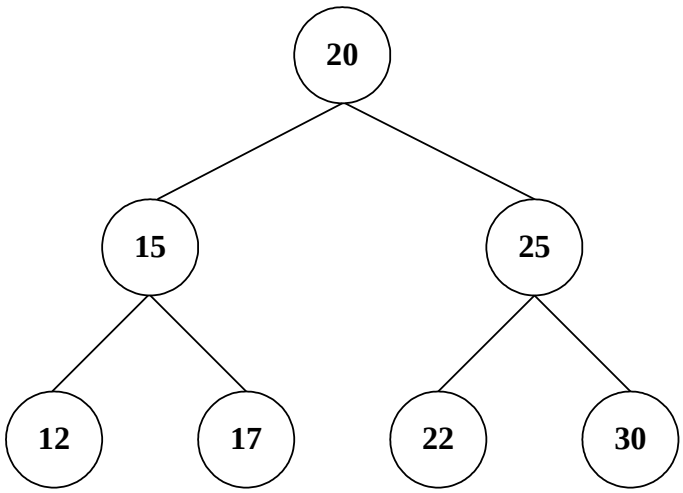
Result of binary tree example:
H, I, D, J, K, E, B, L, M, F, N, O, G, C, A

7- Binary Search Tree(BST)

Definition: A binary search tree is a binary tree. It may be empty. If it is not empty, it satisfies the following properties:

(1) Every element has a key, and no two elements have the same key, that is, the keys are unique.
(2) The keys in a nonempty left subtree must be smaller than the key in the root of the subtree.
(3) The keys in a nonempty right subtree must be larger than the key in the root of the subtree.
(4) The left and right subtrees are also binary search trees.

Example of the Binary Search Tree:



Searching A Binary Search Tree

Suppose we wish to search for an element with a key. We begin at the root. If the root is NULL, the search tree contains no elements and the search is unsuccessful. Otherwise, we compare key with the key value in root. If key equals root's key value, then the search terminates successfully. If key is less than root's key value, then no elements in the right subtree subtree can have a key value equal to key. Therefore, we search the left subtree of root. If key is larger than root's key value, we search the right subtree of root.

Recursive Function for Binary Search Tree:

```
Search (  root, int   key )
{
        if ( !=root )
                return   NULL;
        if ( key = = root.data )
                return   root;
        if ( key < root.data )
                return   search ( root.left_child, key );
        return   search ( root.right_child, key );
}
```

Inserting Into A Binary Search Tree

To insert a new element, key, we must first verify that the key is different from those of existing elements. To do this we search the tree. If the search is unsuccessful, then we insert the element at the point the search terminated.

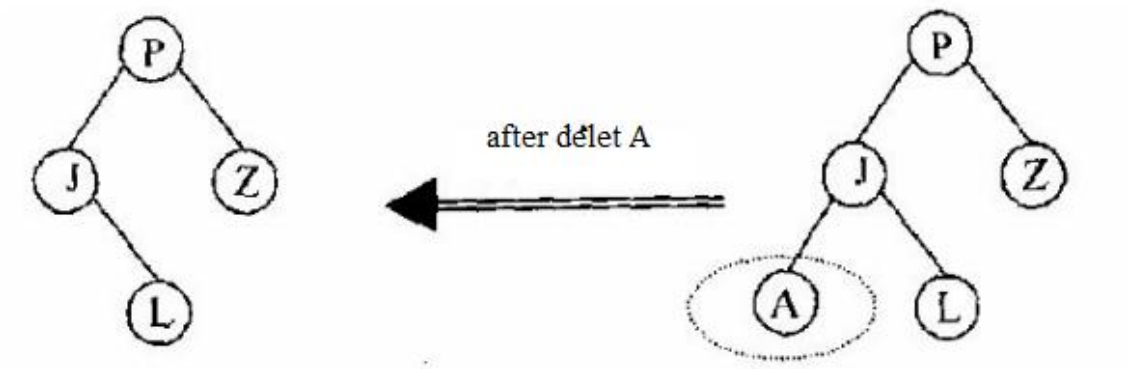
Insert Function:

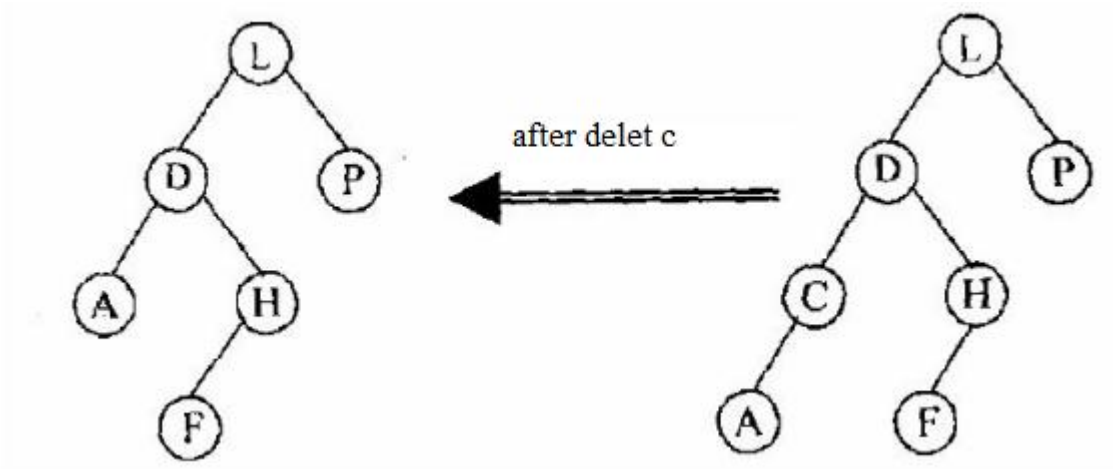
```
void   insert_node ( tree node, int   num )
{
        tree   ptr,
         temp = modified_search ( node, num );  /**
        if ( temp || ! ( node ) )
        {
                ptr = new node;
                if ( ptr = = NULL)
                {
                        print "The memory is full \n";
                        exit ( 1 );
                }
                Ptr. data = num;
                Ptr. eft_child = ptr. right_child = NULL;
                if ( node )
                {
                        if ( num < temp.data )
                                temp.left_child = ptr;
                        else
                                temp.right_child = ptr;
                }
                else
                        node = ptr;
        }
}
```

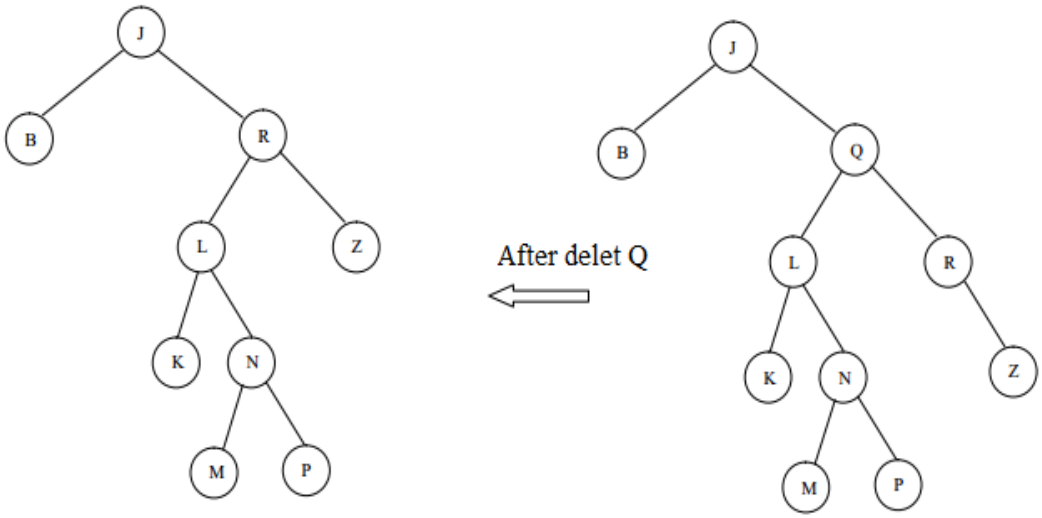
Note ** The function modified_search that is slightly modified version of function search. This function searches the binary search tree *node for the key num. If the tree is empty or if num is present, it returns NULL. Otherwise, it returns a pointer to the last node of the tree that was encountered during the search. The new element is to be inserted as a child of this node.

Deletion from a Binary Search Tree

- Deletion of a leaf node is easy. For example, if a leaf node is left child, we set the left child field of its parent to NULL and free the node.
- The deletion of a nonleaf node that has only a single child is also easy. We erase the node and then place the single child in the place of the erased node.
- When we delete a nonleaf node with two children, we replace the node with either the largest element in its left subtree or the smallest elements in its right subtree. Then we proceed by deleting this replacing element from the subtree from which it was taken.


P
J
Z
L
after delet A
P
J
Z
A
L


L
D
P
A
H
F
after delet c
L
D
P
C
H
A
F

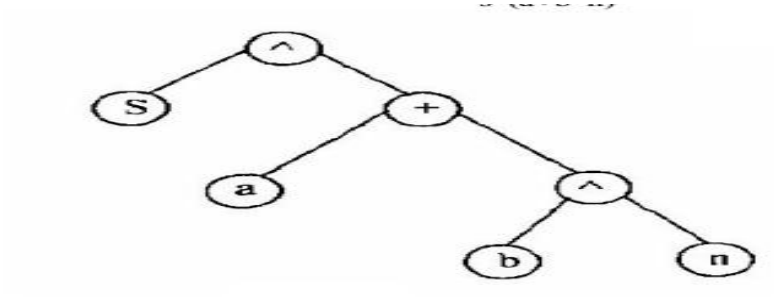

J
B
R
L
Z
K
N
M
P
After delet Q
J
B
Q
L
R
K
N
Z
M
P

8- Representation Expressions Using Binary Tree

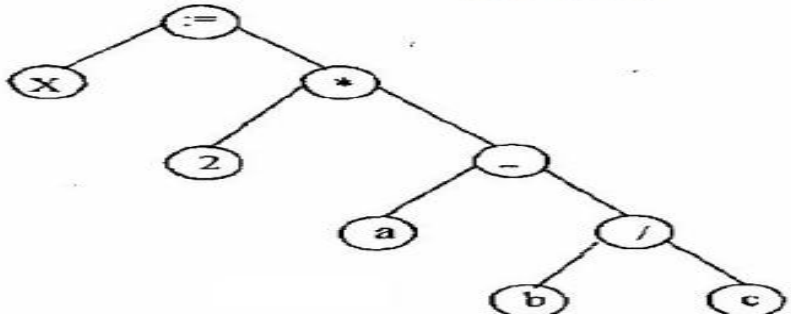
An important application for binary trees is the use of arithmetic expression ,where the operation (+,-,*,, etc.) represent The tree node levels and the operand is leaves in that arithmetic expression, where the level of tree reflect the primacy of computational operation.

Trees Examples: Use the binary tree to represent the following arithmetic expression:

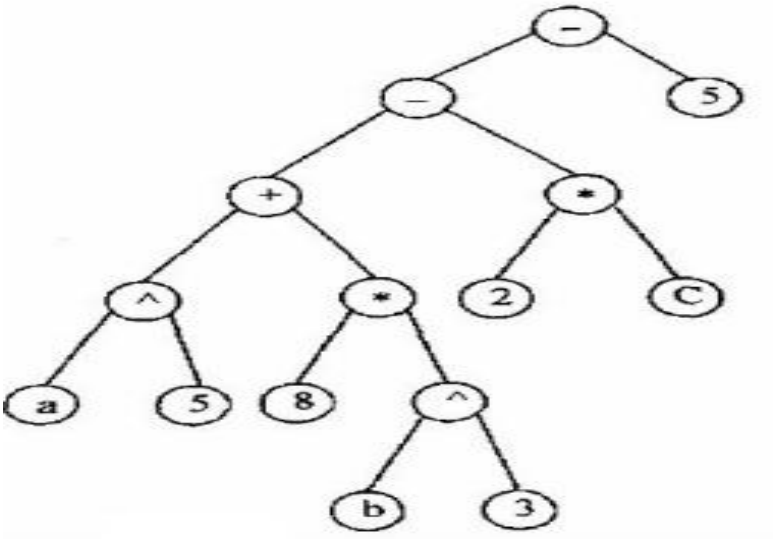
S^(a+b^n):



x:=2* (a-b/c)



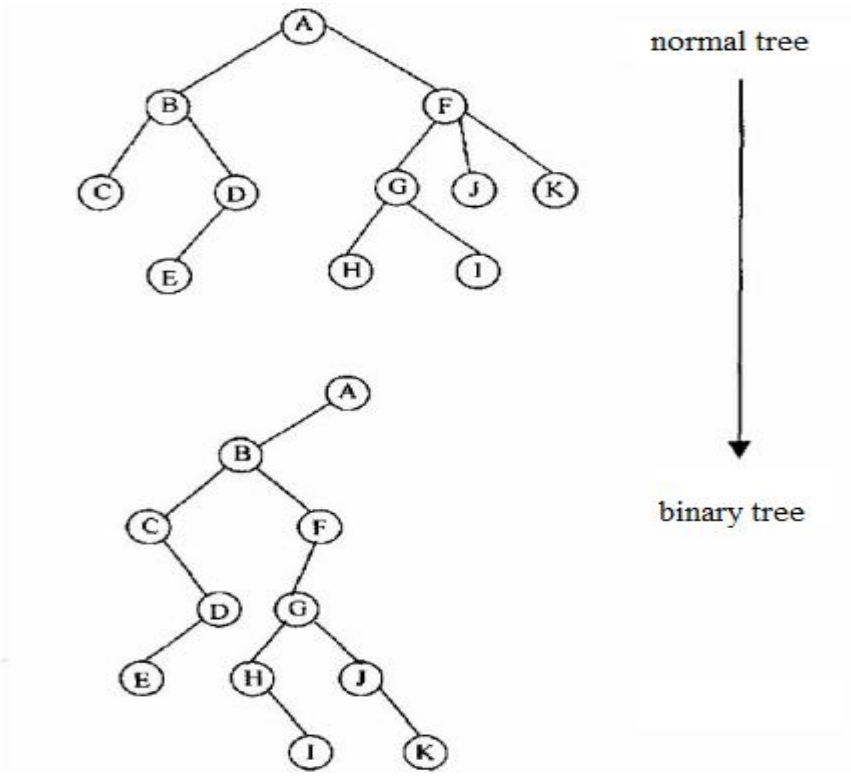
a^5+8*b^3-2*c-5



9- Transfer the tree to binary tree

To transfer the tree to binary tree ,following the steps:

a. The left child for binary tree is the same left child of tree
b. The brothers of this left child of tree are became the right child in binary tree
c. Repeat the same steps and the left child represent now the root.

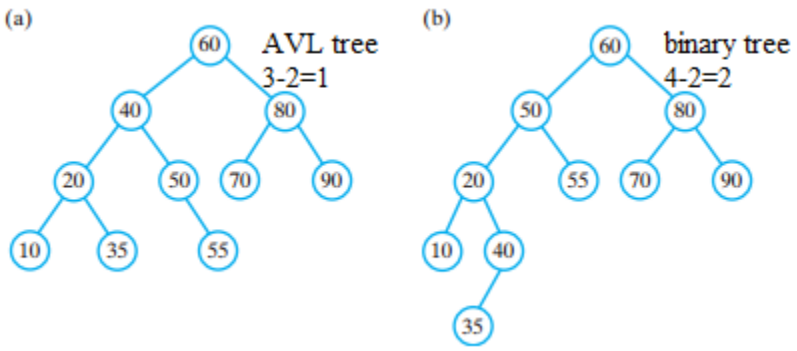


10- AVL TREE(BLANCED TREE)

In this part that you can form several differently shaped binary search trees from the same collection of data. Some of these trees will be balanced and some will not. You could take an unbalanced binary search tree and rearrange its nodes to get a balanced binary search tree. Recall that every node in a balanced binary tree has subtrees whose heights differ by no more than 1.

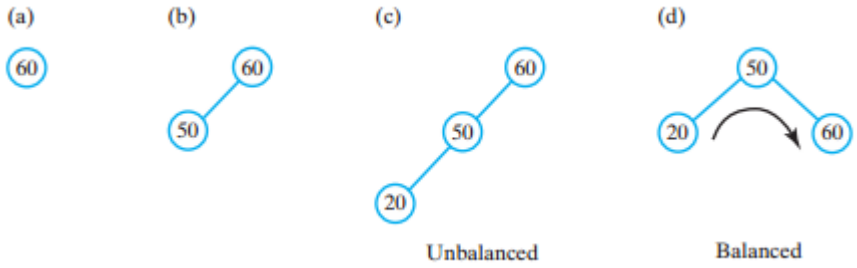
This idea of rearranging nodes to balance a tree was first developed in 1962 by two mathematicians, Adel'son-Vel'skii and Landis. Named after them, the AVL tree is a binary search tree that rearranges its nodes whenever it becomes unbalanced. The balance of a binary search tree is upset only when you add or remove a node. Thus, during these operations, the AVL tree rearranges nodes as necessary to maintain its balance**.**

Note: A node is balanced if it is the root of a balanced tree, that is, if its two subtrees differ in height by no more than 1.See the following figure :

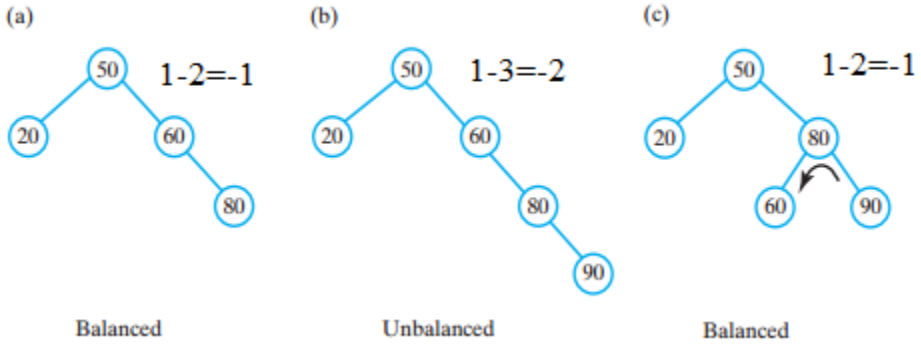


Single Rotations

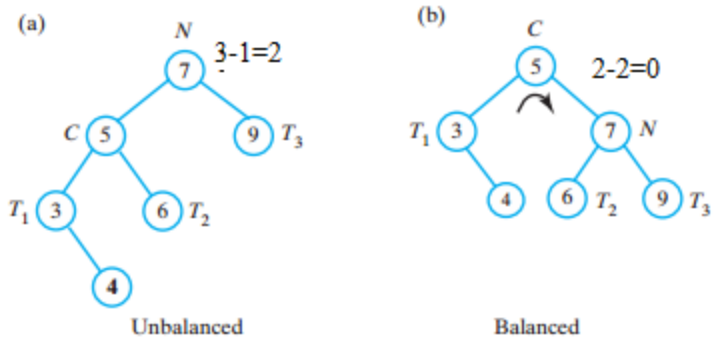
Right rotations. : See the following figure , After inserting (a) 60; (b) 50; and (c) 20 into an initially empty binary search tree, the tree is not balanced; (d) a corresponding AVL tree rotates its nodes to restore balance



Lift rotation: See the following figure , (a) Adding 80 to the above tree in (d) does not change the balance of the tree; (b) a subsequent addition of 90 makes the tree unbalanced ; (c) a left rotation restores its balance

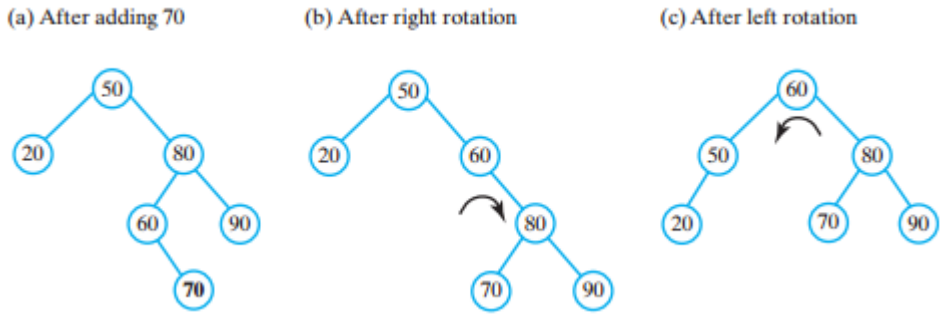


Another example of right rotation:



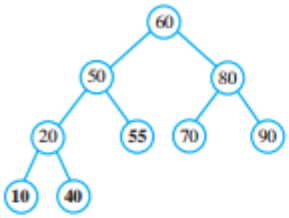
Double Rotations

The following rotations are called a right-left double rotation : (a) Adding 70 to the tree destroys its balance; to restore the balance, perform both (b) a right rotation and (c) a left rotation

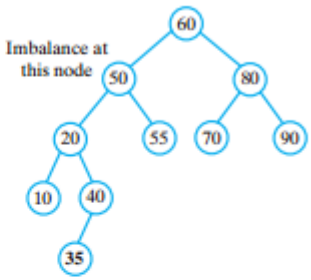


Left-right double rotations.: (a) The AVL tree after additions that maintain its balance; (b) after an addition that destroys the balance; (c) after a left rotation; (d) after a right rotation:

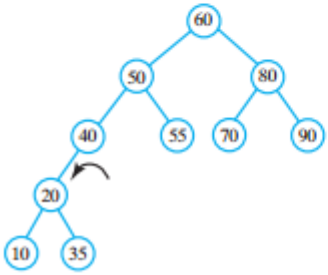
(a) After adding 55, 10, and 40



(b) After adding 35



(c) After left rotation about 40



(d) After right rotation about 40