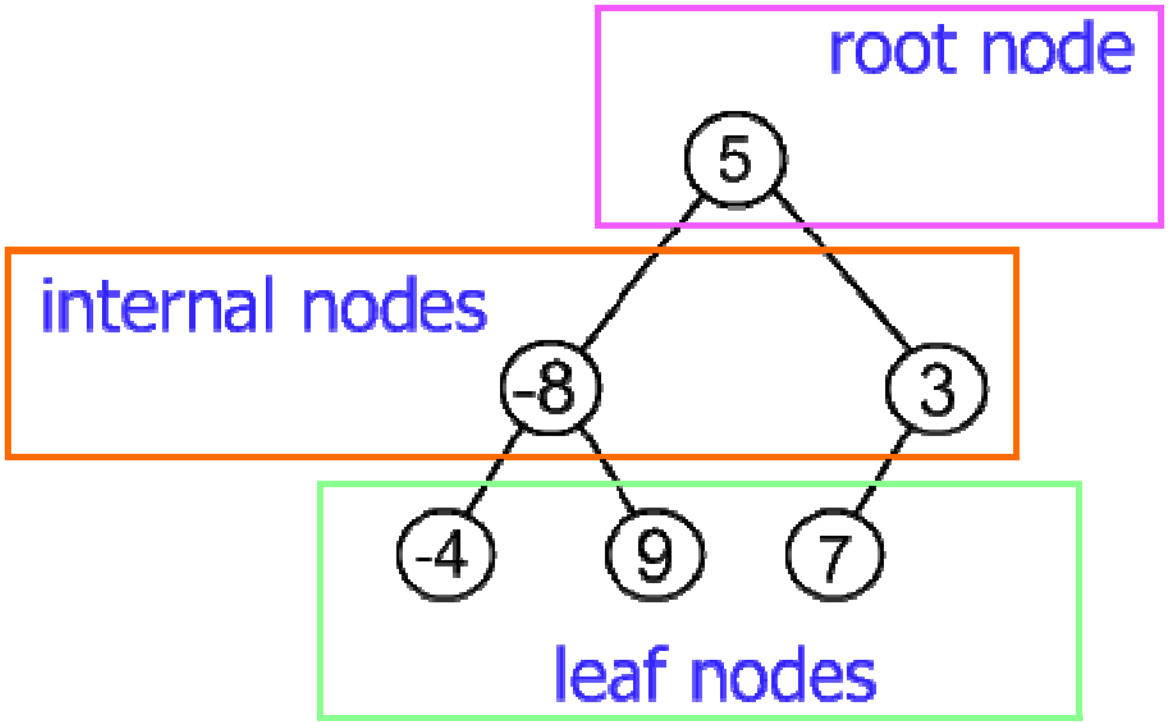
5: Binary Trees

In computer science, a ***binary tree*** is a tree data structure in which each node has at most two children. Typically the first node is known as the ***parent*** and the child nodes are called ***left*** and ***right***.

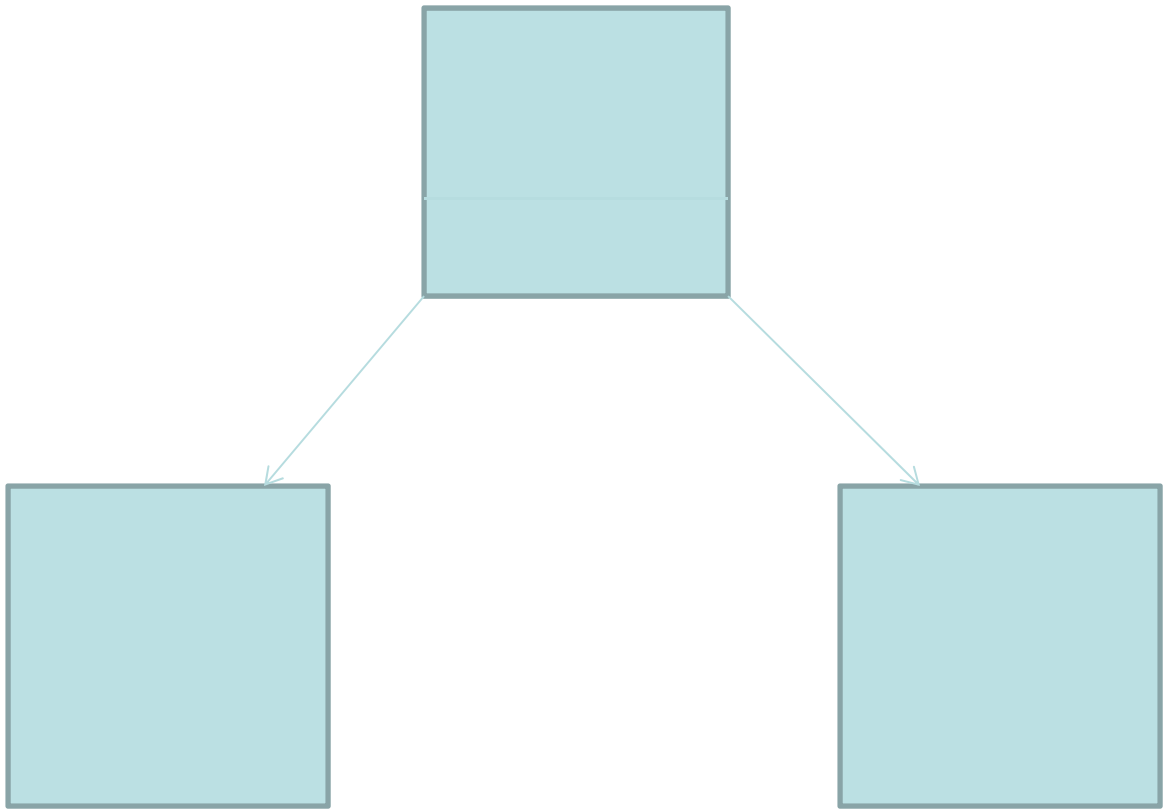
Widespread usage of binary tree is as a basic structure for *binary search tree*. Each binary tree has following groups of nodes:

- **Root:** the topmost node in a tree. It is a kind of "main node" in the tree, because all other nodes can be reached from root. Also, root has no parent. It is the node, at which operations on tree begin (commonly).
- **Internal nodes:** these nodes has a parent (root node is not an internal node) and **at least one** child.
- **Leaf nodes:** these nodes has a parent, but has no children.

Binary Tree Example

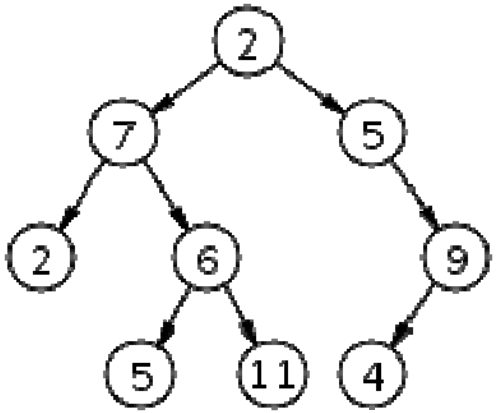


Binary Tree Example



Definitions

root: The root node of a tree is the node with no parents.

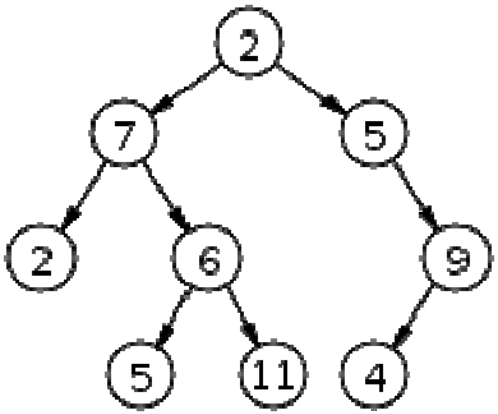


leaf: A leaf node has no children.

depth: The depth of a node n is the length of the path from the root to the node.

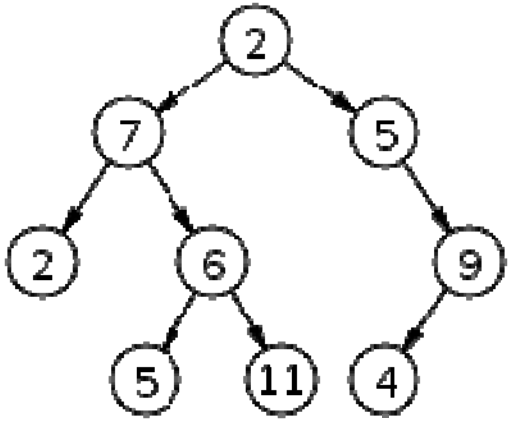
Definitions

height: The height of a tree is the length of the path from the root to the deepest node in the tree. A (rooted) tree with only a node (the root) has a height of zero.



sibling: Siblings are nodes that share the same parent node.

Definitions

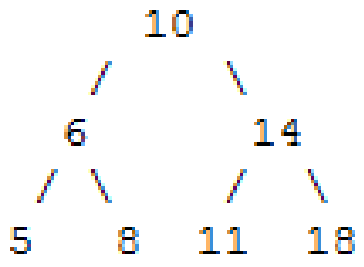


ancestor or descendant : If a path exists from node p to node q, where node p is closer to the root node than q, then p is an **ancestor** of q and q is a **descendant** of p.

Binary Tree: Definitions

full binary tree: A full binary tree (sometimes proper binary tree or 2-tree) is a tree in which every node other than the leaves has two children.

perfect binary tree (complete binary tree) A perfect binary tree is a *full binary tree* in which all *leaves* are at the same *depth* or same *level*.



Binary Tree: Definitions

Height balanced tree (AVL tree): A balanced binary tree is where the depth of all the *leaves* differs by at most 1. Balanced trees have a predictable depth (how many nodes are traversed from the root to a leaf, root counting as node 0 and subsequent as 1, 2, ..., depth). This depth is equal to the integer part of $log_2(n)$ where n is the number of nodes on the balanced tree. Example 1: balanced tree with 1 node, $log_2(1) = 0$ (depth = 0). Example 2: balanced tree with 3 nodes, $log_2(3) = 1.59$ (depth=1). Example 3: balanced tree with 5 nodes, $log_2(5) = 2.32$ (depth of tree is 2 nodes).

Typical Binary Tree Code in C/C++

The binary tree is built with a node type like this:

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

Or simply in C++:

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

Typical Binary Tree Code in C/C++

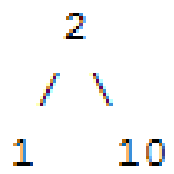
Lookup( )

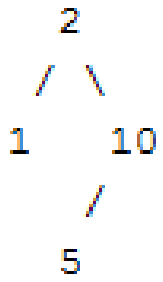
```
/*
    Given a binary tree, return true if a node
    with the target data is found in the tree. Recurs
    down the tree, chooses the left or right
    branch by comparing the target to each node.
*/
```

```
int lookup(node* node, int target) {
      // 1. Base case == empty tree
      if (node == NULL) {
        return(false);
      }
      else {
        // 2. see if found here
        if (target == node->data) return(true);
        else {
          // 3. otherwise recur down the correct subtree
          if (target < node->data)
              return(lookup(node->left, target));
          else
              return(lookup(node->right, target));
        }
      }
    }
```

Insert( )

Given a binary search tree and a number, insert a new node with the given number into the tree in the correct place.





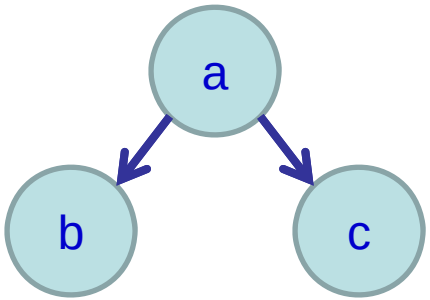
Example of inserting 5: before (left) and after (right).

Insert( )

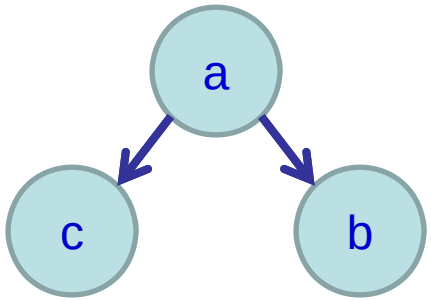
```
/*   Helper function that allocates a new node
     with the given data and NULL left and right
     pointers.  */


node* NewNode(int data) {
      node* Node = new(struct node); // "new" is like "malloc"
      Node->data = data;
      Node->left = NULL;
      Node->right = NULL;
      return(Node);
}
```

```
/*  Give a binary search tree and a number, inserts a new node with the given
    number in the correct place in the tree. Returns the new root pointer which
    the caller should then use (the standard trick to avoid using reference
    parameters).  */
node* insert(node* Node, int data) {
  // 1. If the tree is empty, return a new, single node
  if (Node == NULL) {
    return(newNode(data));
  }
  else {
    // 2. Otherwise, recur down the tree
    if (data <= Node->data)
        Node->left = insert(Node->left, data);
    else
        Node->right = insert(Node->right, data);
    return(Node); // return the (unchanged) node pointer
  }
}
```

Order of a tree



Tree 1



Tree 2

Order of Tree: Tree 1 and Tree 2 are distinctly two trees.

Three orders of tree traversals

1. Preorder
2. Inorder
3. Postorder

- In a *preorder* traversal, the root node of the tree is processed first, then the left subtree is traversed, then the right subtree.

- In a *postorder* traversal, the left subtree is traversed, then the right subtree, and then the root node is processed.

- In an *inorder* traversal, the left subtree is traversed first, then the root node is processed, then the right subtree is traversed.

preorder

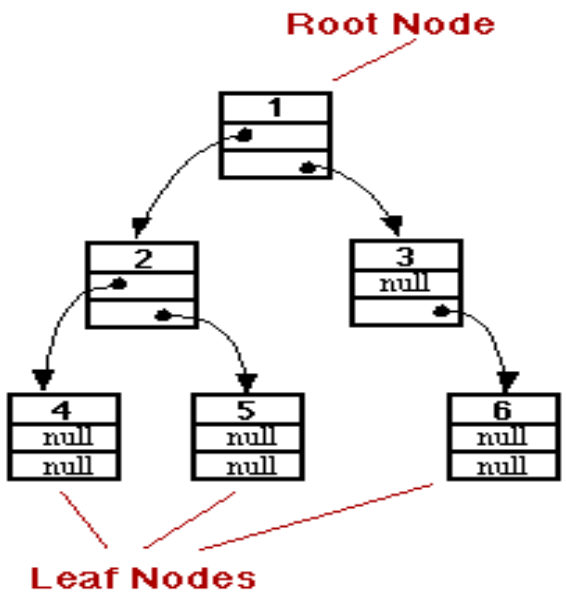
```
void preorder(node *tree)
        {
            if(tree!=NULL)
            {
                cout<<tree->data;    // print info
                preorder(tree->left);
                preorder(tree->right);
                getch();
            }
        }
```

inorder

```
void inorder(node *tree)
        {
            if(tree!=NULL)
            {
                inorder(tree->left);
                cout<<tree->data;     // print info
                inorder(tree->right);
                getch();
            }
        }
```

postorder

```
void postorder(node *tree)
        {
            if(tree!=NULL)
            {
                postorder(tree->left);
                postorder(tree->right);
                cout<<tree->data;     // print info
                getch();
            }
        }
```



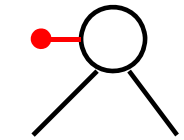
preorder outputs: 1 2 4 5 3 6

postorder outputs: 4 5 2 6 3 1

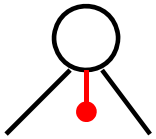
inorder outputs: 4 2 5 1 3 6

Tree traversals using “flags”

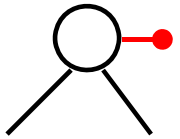
- The order in which the nodes are visited during a tree traversal can be easily determined by imagining there is a “flag” attached to each node, as follows:



preorder

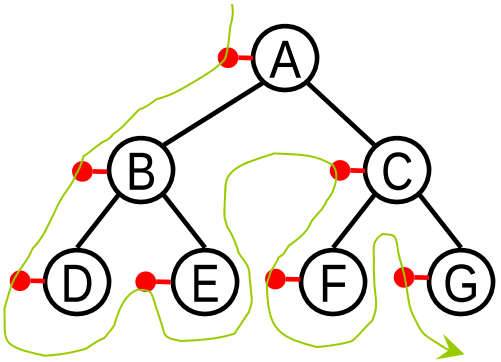


inorder

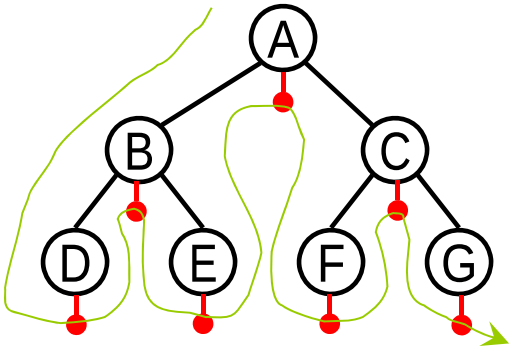


postorder

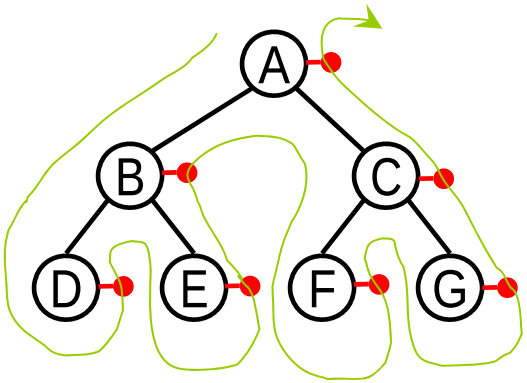
- To traverse the tree, collect the flags:



A B D E C F G



D B E A F C G



D E B F G C A

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

node *tree=NULL;
node *insert(node *tree,int ele);

void preorder(node *tree);
void inorder(node *tree);
void postorder(node *tree);
int count=1;

void main()
{
    clrscr();   // clean screen
    int ch,ele;
    do
    {
        clrscr();
        cout<<"\n\t\a\a1----INSERT A NODE IN A BINARY TREE.\a\a";
        cout<<"\n\t\a\a2----PRE-ORDER TRAVERSAL.\a\a";
        cout<<"\n\t\a\a3----IN-ORDER TRAVERSAL.\a\a";
        cout<<"\n\t\a\a4----POST-ORDER TRAVERSAL.\a\a";
        cout<<"\n\t\a\a5----EXIT.\a\a";
        cout<<"\n\t\a\aENTER CHOICE::\a\a";
        cin>>ch;
```

```
switch(ch) {
case 1:     cout<<"\n\t\a\aENTER THE
            ELEMENT::\a\a";
             cin>>ele;
             tree=insert(tree,ele);
             break;

case 2:     cout<<"\n\t\a\a****PRE-ORDER
            TRAVERSAL OF A TREE****\a\a";
             preorder(tree);
             break;

case 3:     cout<<"\n\t\a\a****IN-ORDER
            TRAVERSAL OF A TREE****\a\a";
             inorder(tree);
             break;

case 4:     cout<<"\n\t\a\a****POST-ORDER
            TRAVERSAL OF A TREE****\a\a";
            postorder(tree);
             break;

case 5:     exit(0);
}
}while(ch!=5);

} //main( )
```

C++ (java-like tree node)

```
struct TreeNode {
        // An object of type TreeNode represents one node
        // in a binary tree of strings.

    string item;      // The data in this node.
    TreeNode left;    // Pointer to left subtree.
    TreeNode right;   // Pointer to right subtree.

    TreeNode(string str) {
          // Constructor.  Make a node containing str.
      item = str;
      left = NULL;
      right = NULL;
    }

};  // end struct Treenode
```

A recursive function (treeContains): search for a given item in the tree

```
TreeNode *root;   // Pointer to the root node in the tree.
root = NULL;      // Start with an empty tree.
```

```
bool treeContains( TreeNode *root, string item ) {
        // Return true if item is one of the items in the binary
        // sort tree to which root points.   Return false if not.
    if ( root == NULL ) {
        // Tree is empty, so it certainly doesn't contain item.
      return false;
    }
    else if ( item == root->item ) {
        // Yes, the item has been found in the root node.
      return true;
    }
    else if ( item < root->item ) {
        // If the item occurs, it must be in the left subtree.
      return treeContains( root->left, item );
    }
    else {
        // If the item occurs, it must be in the right subtree.
      return treeContains( root->right, item );
    }
  }  // end treeContains()
```

Non recursive:

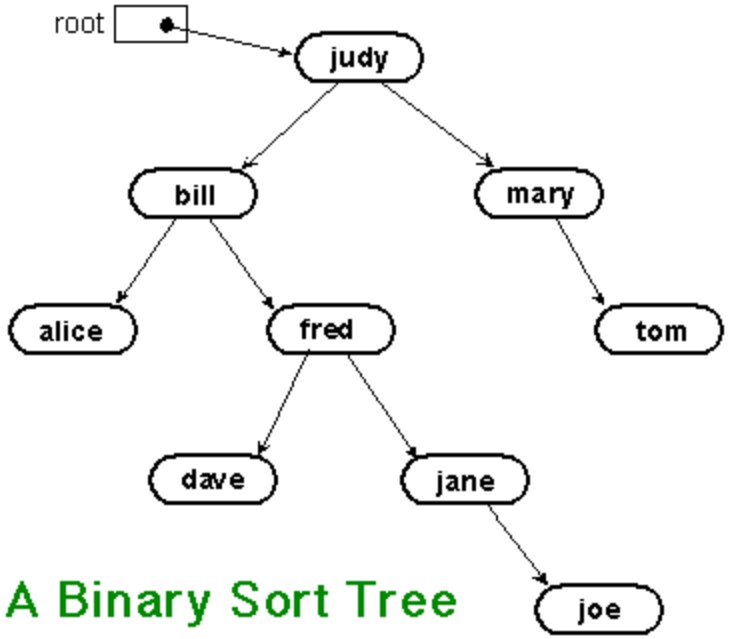
```
bool treeContainsNR( TreeNode *root, string item ) {
      // Return true if item is one of the items in the binary
      // sort tree to which root points.   Return false if not.
   TreeNode *runner;          // For "running" down the tree.
   runner = root;                              // Start at the root node.
   while (true) {
      if (runner == NULL) {   // We've fallen off the tree without finding item.
         return false;
      }
      else if ( item == runner->item ) {  // We've found the item.
         return true;
      }
      else if ( item < runner->item ) {    // If the item occurs, it must be in the left subtree,
                                           // So, advance the runner down one level to the left.
         runner = runner->left;
      }
      else {                       // If the item occurs, it must be in the right subtree.
                                   // So, advance the runner down one level to the right.
         runner = runner->right;
      }
   }  // end while
} // end treeContainsNR();
```

Insertion (Recursive)

```
void treeInsert(TreeNode *&root, string newItem) {
      // Add the item to the binary sort tree to which the parameter
      // "root" refers.  Note that root is passed by reference since
      // its value can change in the case where the tree is empty.
   if ( root == NULL ) {
         // The tree is empty.  Set root to point to a new node containing
         // the new item.  This becomes the only node in the tree.
      root = new TreeNode( newItem );
            // NOTE:  The left and right subtrees of root
            // are automatically set to NULL by the constructor.
            // This is important!
      return;
   }
   else if ( newItem < root->item ) {
      treeInsert( root->left, newItem );
   }
   else {
      treeInsert( root->right, newItem );
   }
}  // end treeInsert()
```

Binary Sort Tree

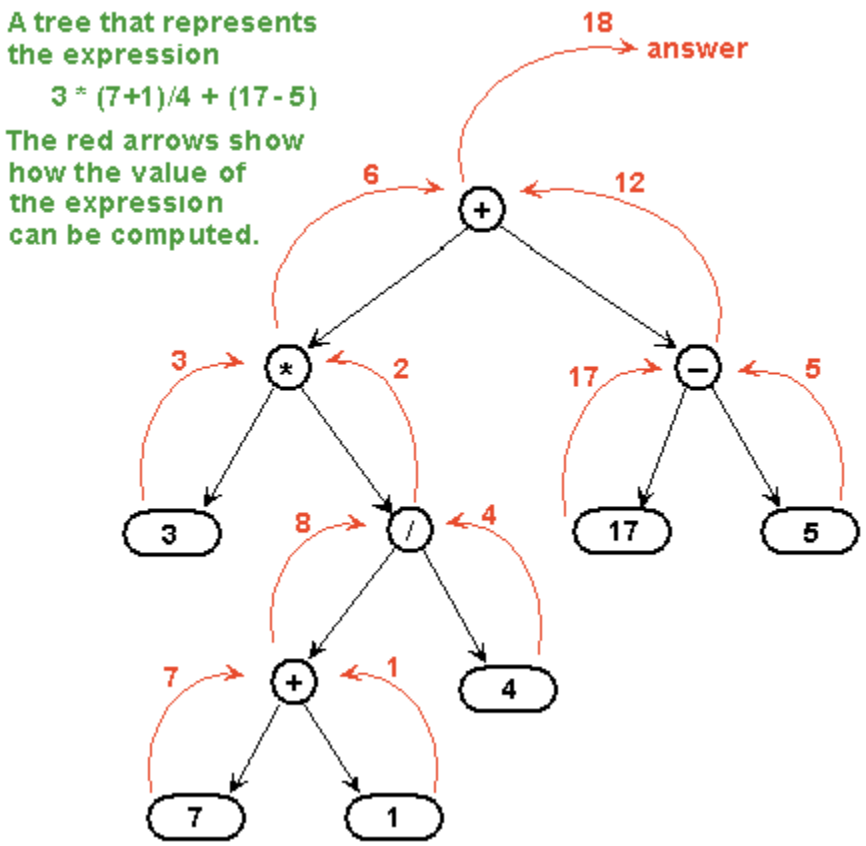
- For every node in the tree, the item in that node is greater than (or less than) every item in the left subtree of that node, and it is less than or equal to (or great than or equal to) all the items in the right subtree of that node.



A Binary Sort Tree

Expression Tree

- Another application of trees is to store mathematical expressions
- Every node in the tree holds either a number or an operator. A node that holds a number is a leaf node of the tree. A node that holds an operator has two subtrees representing the operands to which the operator applies.



An expression tree contains two types of nodes: nodes that contain numbers and nodes that contain operators.

```
const int NUMBER = 0, OPERATOR = 1; // Values representing two kinds of nodes.

    struct ExpNode {  // A node in an expression tree.
       int kind;        // Which type of node is this?
                        //   (Value is NUMBER or OPERATOR.)
       double number;   // The value in a node of type NUMBER.
       char op;         // The operator in a node of type OPERATOR.
       ExpNode *left;   // Pointers to subtrees,
       ExpNode *right;  //     in a node of type OPERATOR.

       ExpNode( double val ) {
             // Constructor for making a node of type NUMBER.
          kind = NUMBER;
          number = val;
       }

       ExpNode( char op, ExpNode *left, ExpNode *right ) {
             // Constructor for making a node of type OPERATOR.
          kind = OPERATOR;
          this->op = op;
          this->left = left;
          this->right = right;
       }

    }; // end ExpNode
```

```
double getValue( ExpNode *node ) {
        // Return the value of the expression represented by
        // the tree to which node refers.  Node must be non-NULL.
    if ( node->kind == NUMBER ) {
        // The value of a NUMBER node is the number it holds.
      return node->number;
    }
    else {      // The kind must be OPERATOR.
                // Get the values of the operands and combine them
                //    using the operator.
      double leftVal = getValue( node->left );
      double rightVal = getValue( node->right );
      switch ( node->op ) {
        case '+':  return leftVal + rightVal;
        case '-':  return leftVal - rightVal;
        case '*':  return leftVal * rightVal;
        case '/':  return leftVal / rightVal;
      }
    }
  } // end getValue()
```

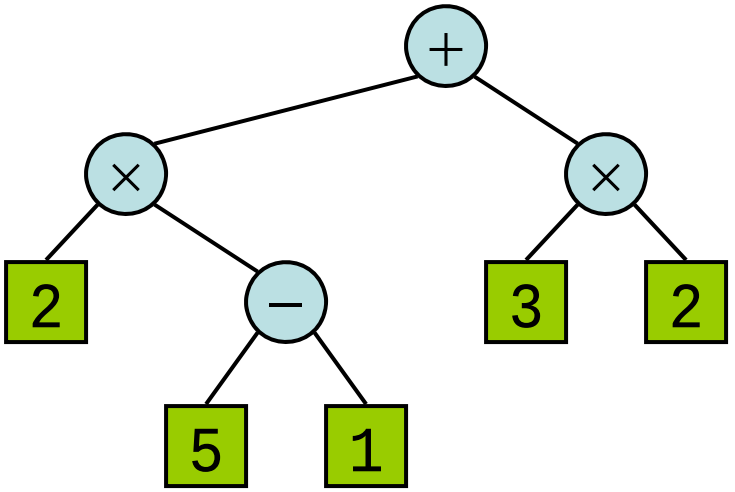
Arithmetic Expression Tree

Binary tree for an arithmetic expression

- internal nodes: operators
- leaves: operands

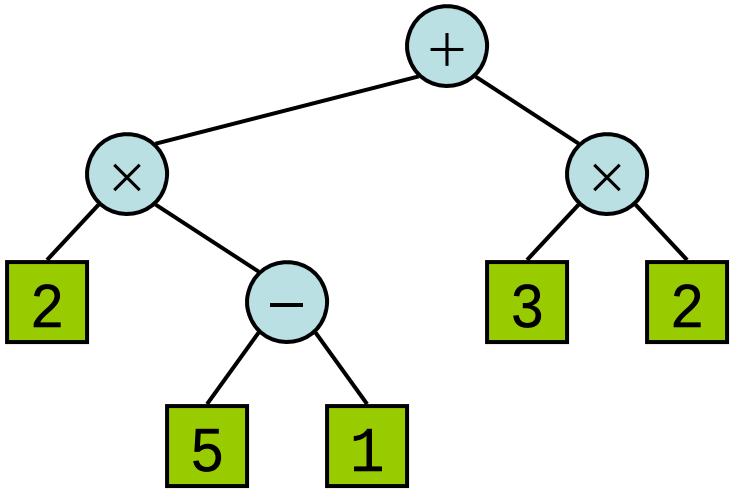
- Example: arithmetic expression tree for the expression

$$((2 \times (5 - 1)) + (3 \times 2))$$



Print Arithmetic Expressions

- inorder traversal:
  - print “(“ before traversing left subtree
  - print operand or operator when visiting node
  - print “)“ after traversing right subtree




```
void printTree(t)
//binary operands only
   if (t.left != null){
      print("(");
      printTree (t.left);}
   print(t.element );
   if (t.right != null){
      printTree (t.right);
      print (")");}
```

((2 × (5 - 1)) +
(3 × 2))

Height Balanced Tree:
AVL Tree (Adelson-Velskii and Landis)

- A balanced binary search tree where the height of the two subtrees (children) of a node differs by at most one. Look-up, insertion, and deletion are O(log n), where n is the number of nodes in the tree.

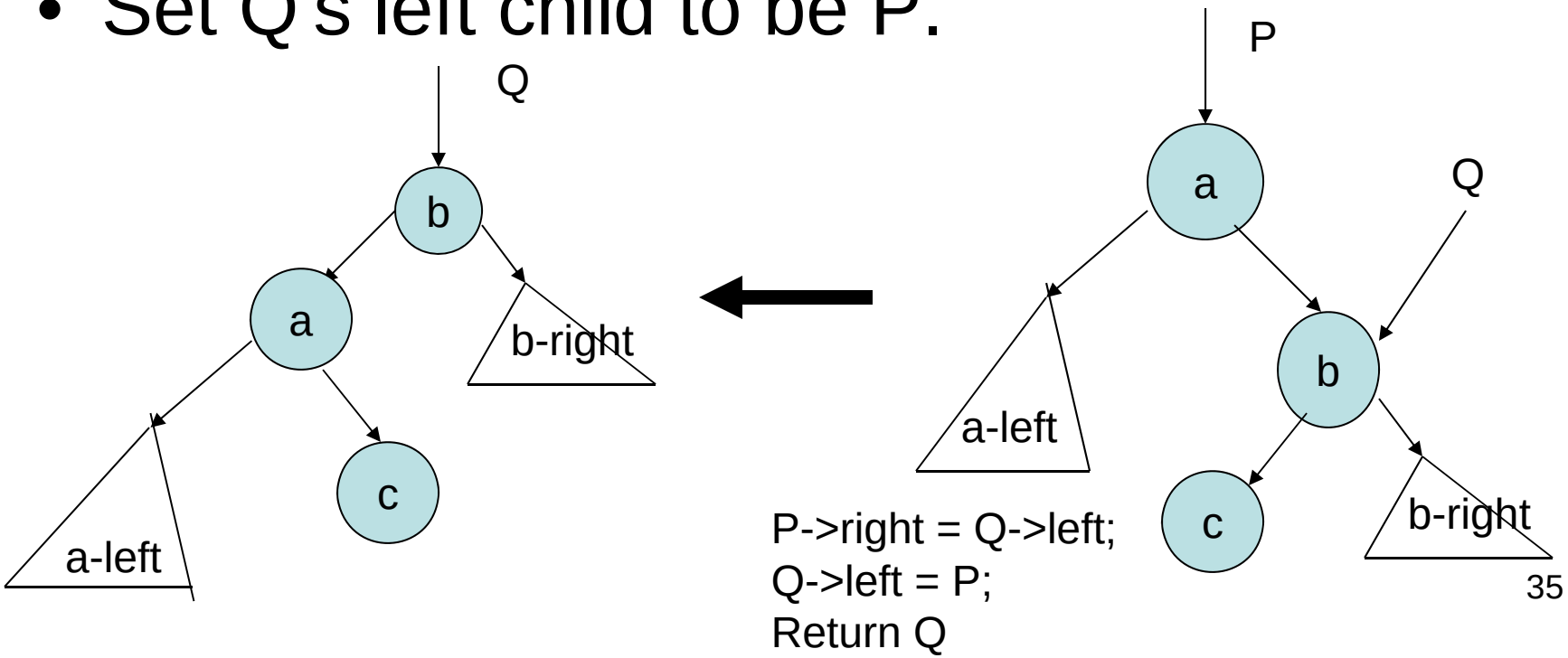
Height: the length of the longest path from a node to a leaf.

All leaves have a height of **0**

An empty tree has a height of **–1**

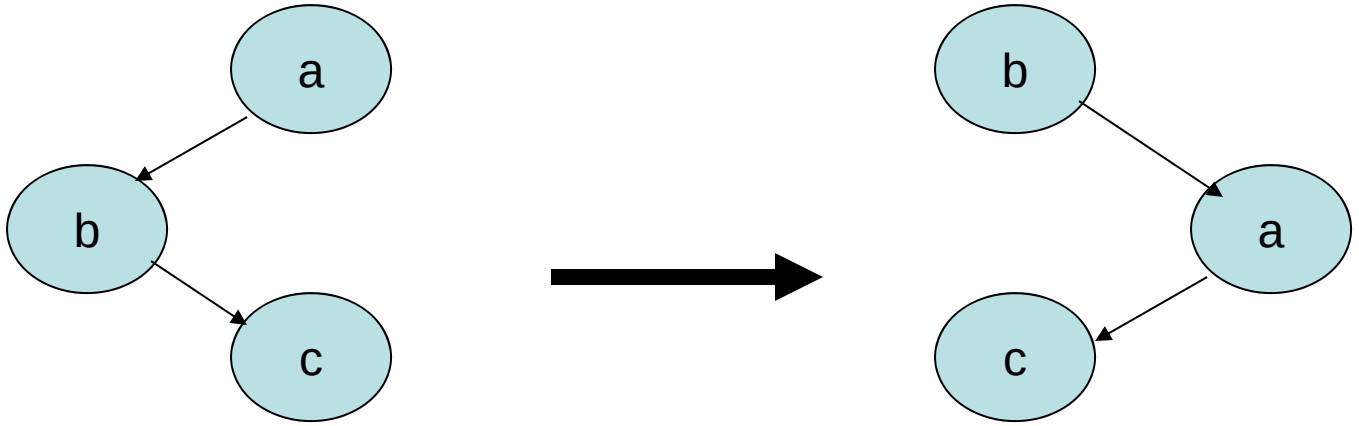
Left Rotation of node P:

- Let Q be P's right child.
- Set Q to be the new root.
- Set P's right child to be Q's left child.
- Set Q's left child to be P.

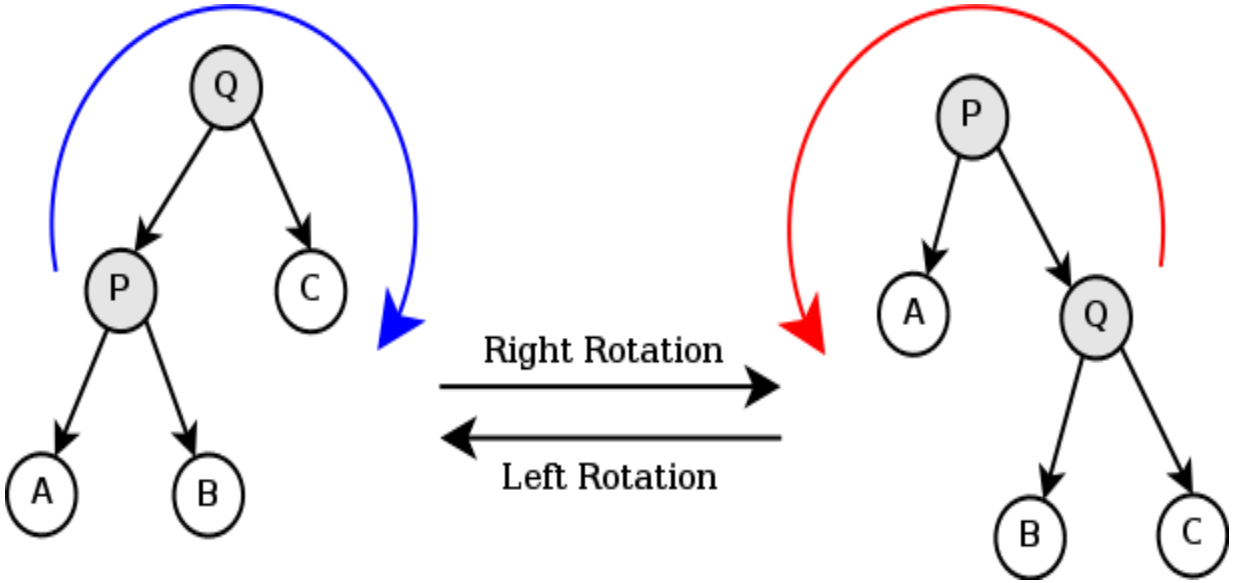


Right Rotation of node Q:

- Let P be Q's left child.
- Set P to be the new root.
- Set Q's left child to be P's right child.
- Set P's right child to be Q.

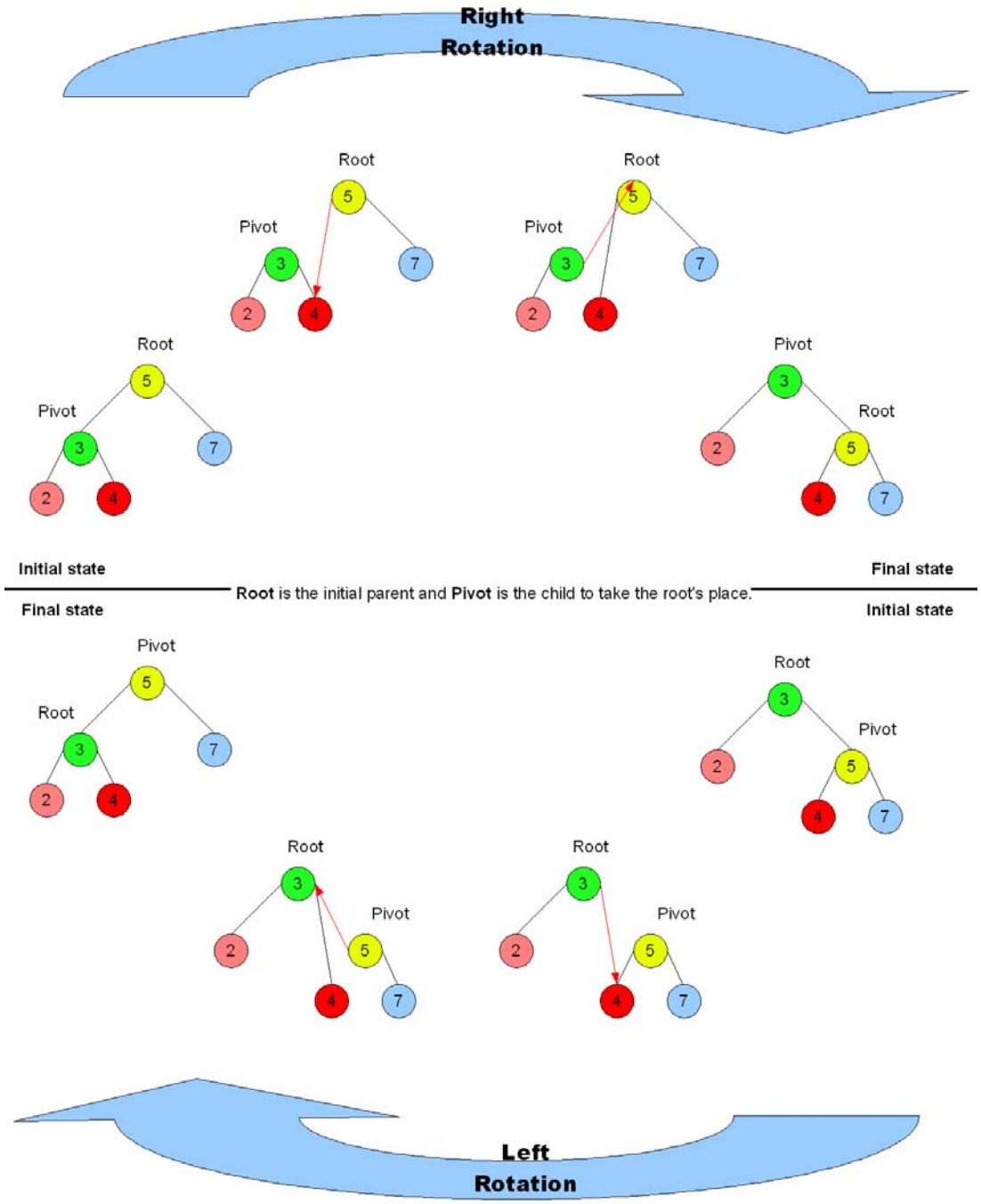


Tree rotation

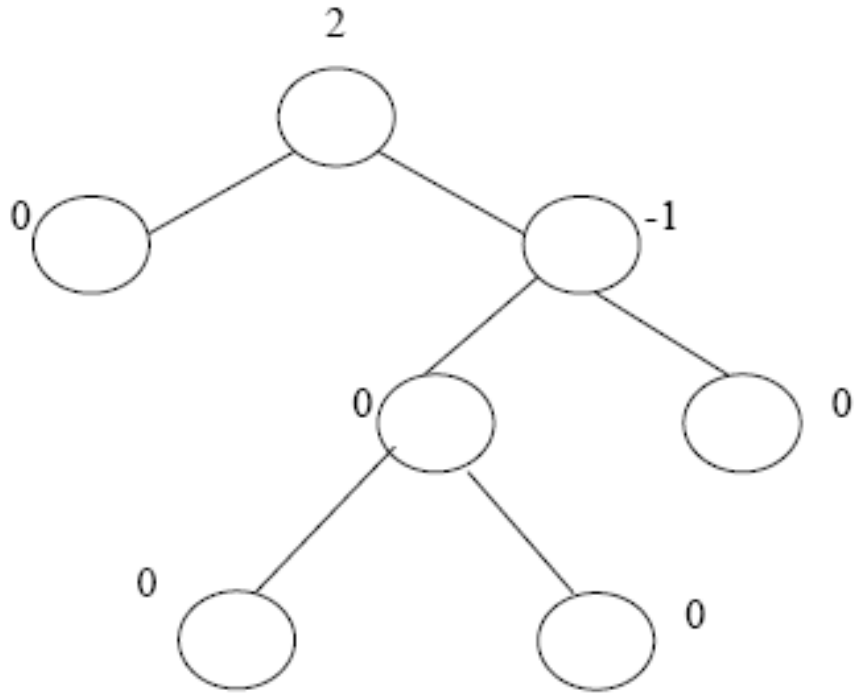


In-order listing:
APBQC

In-order listing:
APBQC


Right
Rotation
Root
Pivot
Root
Pivot
Root
Pivot
Pivot
Root
Initial state
Final state
Root is the initial parent and Pivot is the child to take the root's place.
Final state
Initial state
Pivot
Root
Root
Pivot
Root
Root
Pivot
Pivot
Left
Rotation

AVL TREE showing the Balance Factor



Steps for insertion

1. Insert the numbers in Binary Search Tree (BST) pattern.
2. Before the next number is inserted, perform the following operations :
   a) Find out the balance factor of each node.
   b) Balance Factor of a node = (Maximum number of levels that can be reached in right sub-tree)– (Maximum number of levels that can be reached in left sub tree)

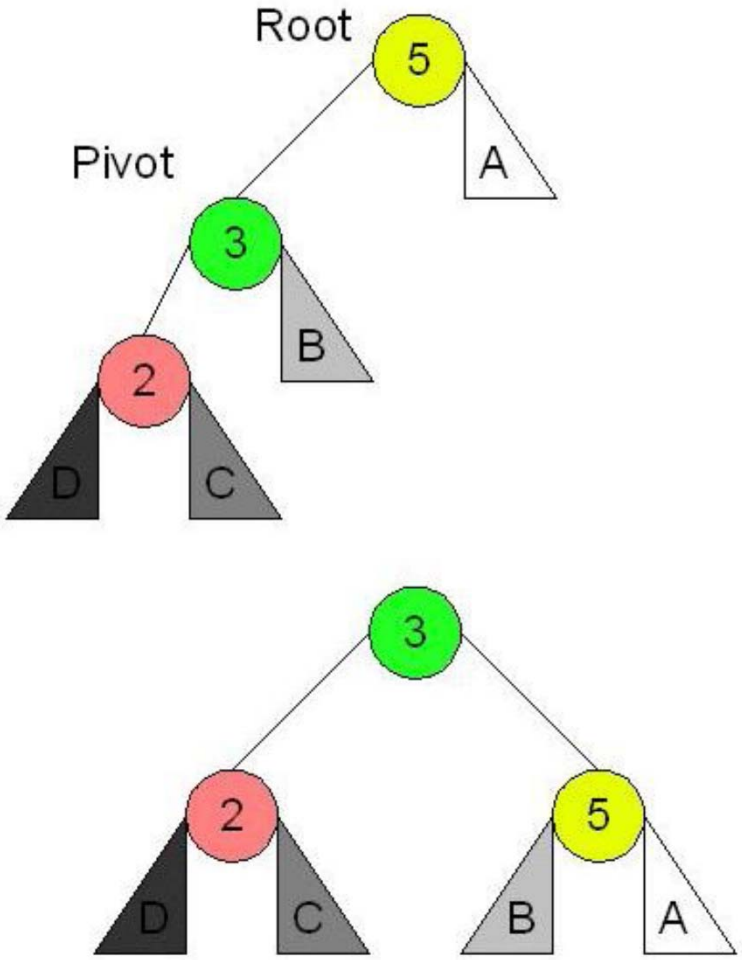
      [WE TAKE LEVEL OF THE STARTING NODE AS 0]
   c) If any of the Balance Factor is (-2) or (2), perform required rotation according to the algorithm given below. If two nodes have (2/-2) then select the lowest node in level.
   d) The tree now become balanced, i.e. no node has Balance Factor (-2) or (2). Continue from step 1.

Rotation Algorithm

```
[Right heavy (B.F. = 2), Left heavy (B.F. = -2)]
IF tree is right heavy (B.F. = 2)
{
    IF tree's right sub tree is left heavy (B.F. = -2 / -1)
        Perform Left-Right (LR) rotation
    ELSE
        Perform Left rotation
}
ELSE IF tree is left heavy (B.F. = -2)
{
    IF tree's left sub tree is right heavy (B.F. = 2 / 1)
        Perform Right-Left Rotation (RL) rotation
    ELSE
        Perform Right rotation
}
```

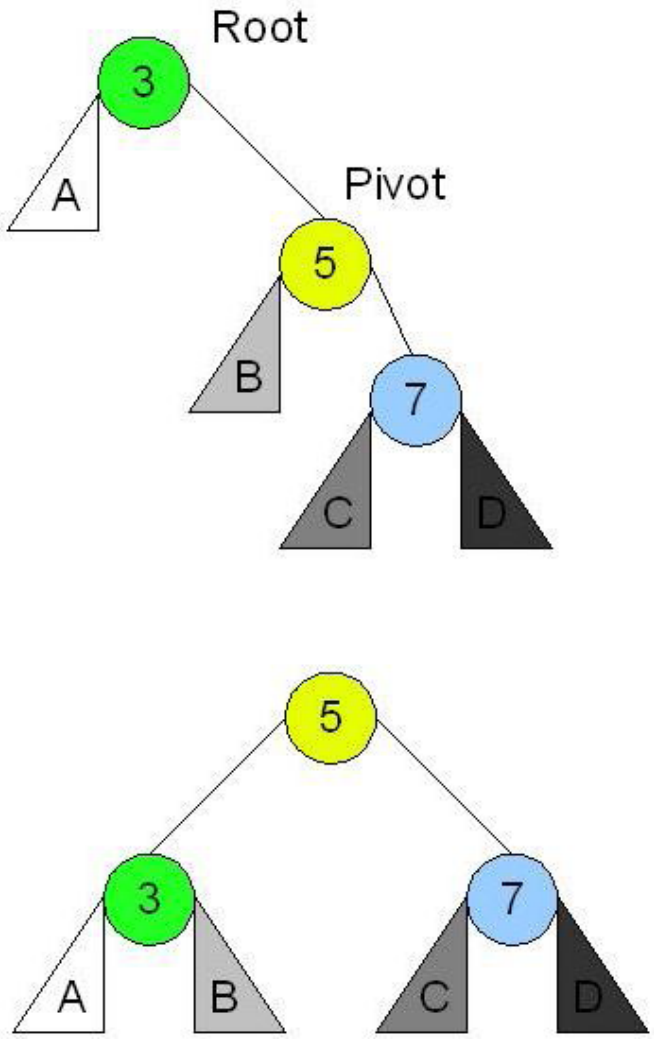
ROTATIONS

Left – Left Case (Right Rotation)



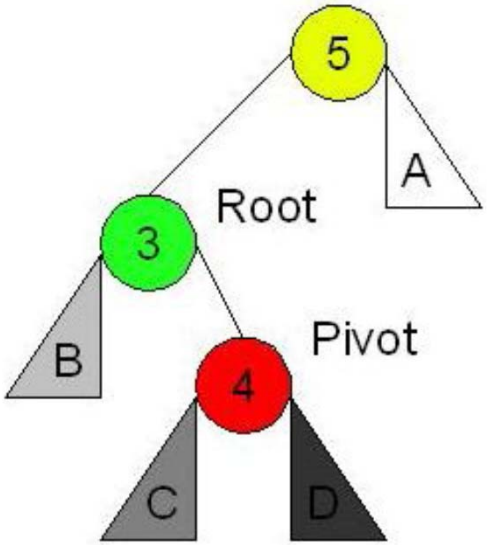
ROTATIONS

Right – Right Case (Left Rotation)

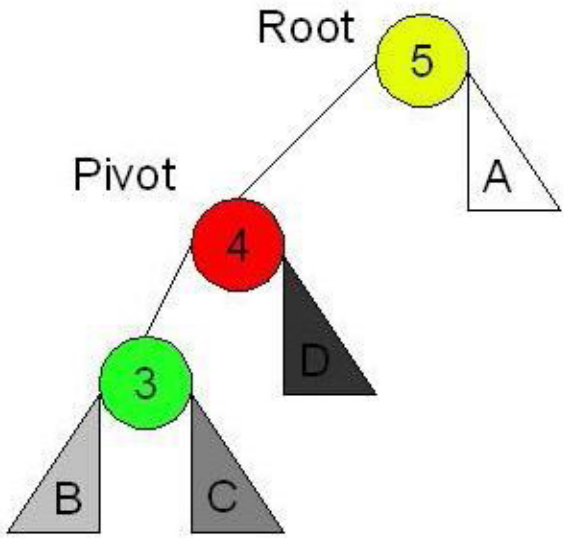


ROTATIONS

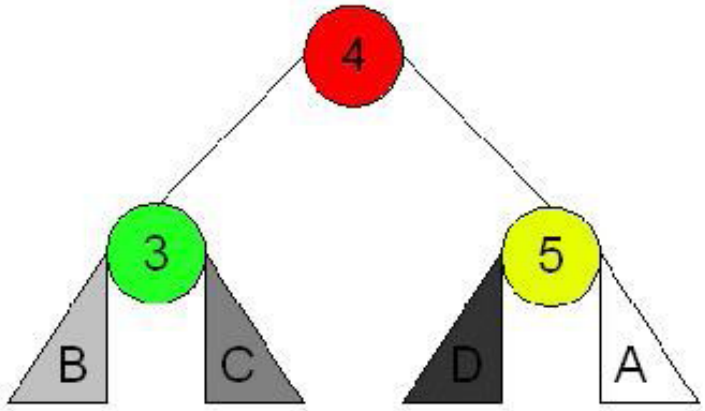
Left - Right Case (LR Rotation)



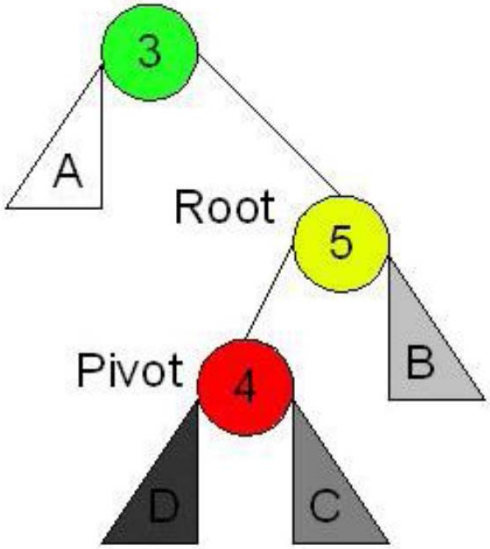
Left rotation



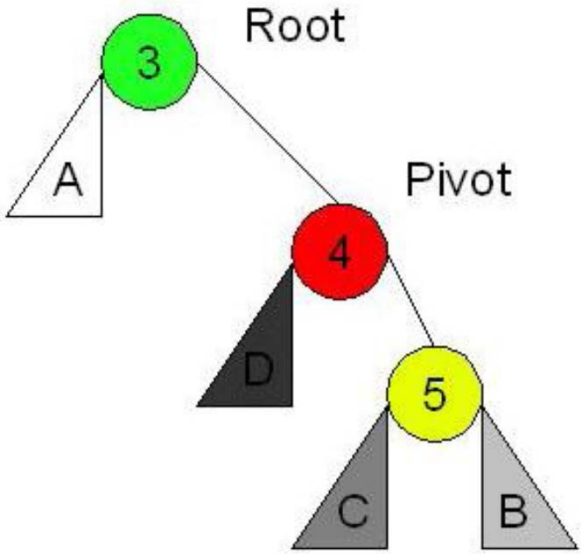
Right rotation



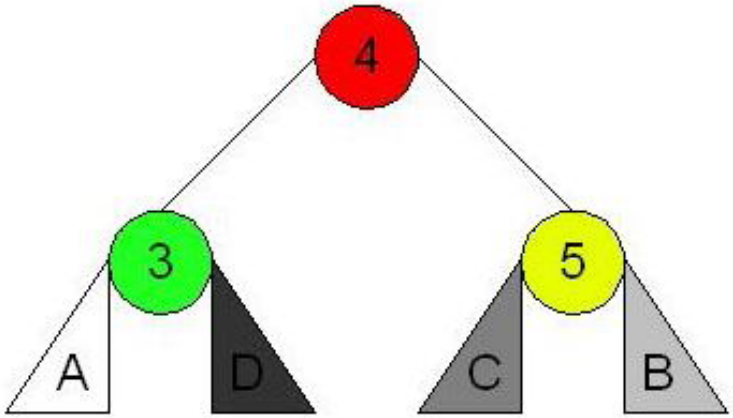
Right - Left Case (RL Rotation)



Right rotation



Left rotation



Steps for deletion

1. Delete the number in Binary Search Tree (BST) pattern. This can be done by the following operations:

   a) If the node has no leaf node, just remove it from the tree.

   b) If the node has only one sub-tree (left/right), just remove the node and replace it by its immediate child node.

   c) If the node has two sub-trees (left/right), remove the node and replace it by its immediate inorder traversal node (i.e. the node that will be written just after the deleted node in inorder traversal). When the deletion of node and reordering of the tree is done, continue from the next step.

Steps for deletion

2. Perform the following operations:

a) Find out the balance factor of each node.

b) Balance Factor of a node = (Maximum number of levels that can be reached in right sub-tree) – (Maximum number of levels that can be reached in left sub tree)

[WE TAKE LEVEL OF THE STARTING NODE AS 0]

c) If any of the Balance Factor is (-2) or (2), perform required rotation according to the algorithm given above. If two nodes have (2/-2) then select the lowest node in level. **If no nodes have (2/-2) then the AVL Tree is still balanced and no further rearranging is not required.**

69, 17, 62, 58, 65, 8, 96, 51, 32, 24