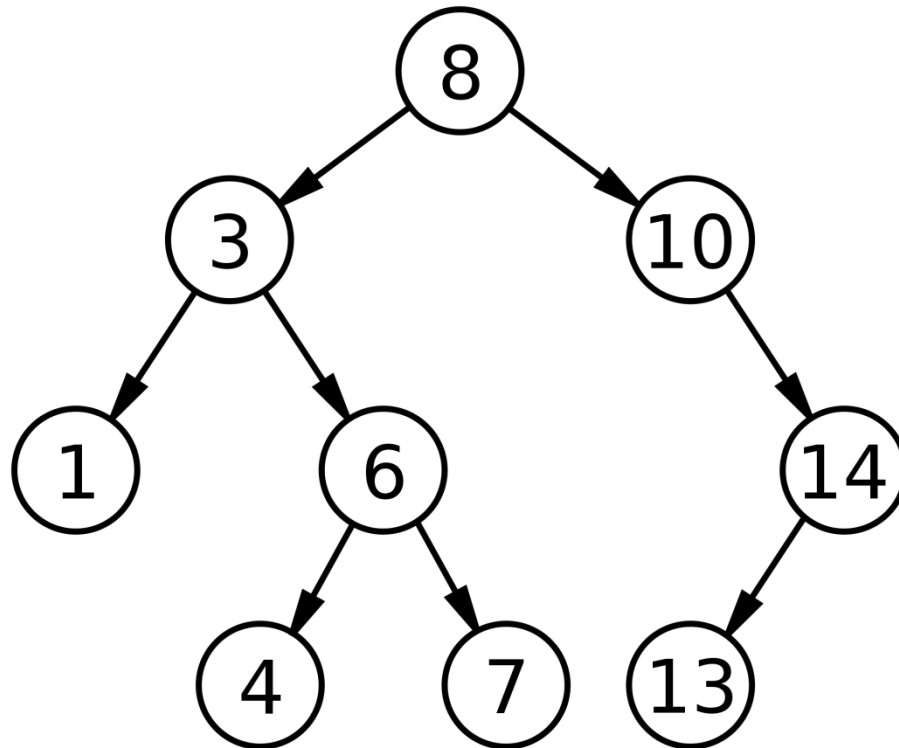


# Binary Search Tree

# Binary Search Tree

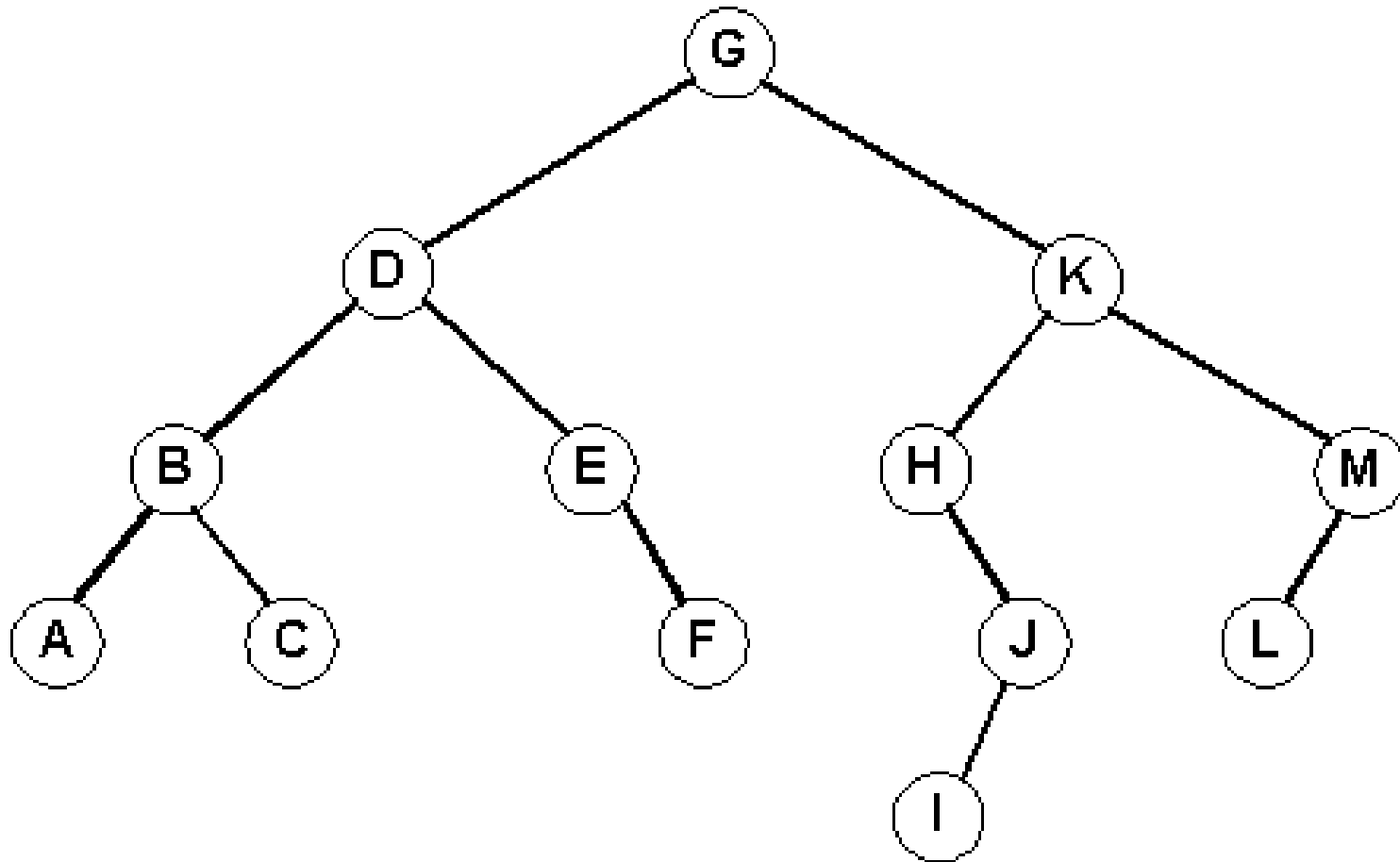
- A binary search tree (BST) is a binary tree in which
  - The values in the **left subtree** of a node are all **less than** the value in the node
  - The values in the **right subtree** of a node are all **greater than** the value of the node.
  - The subtrees of a binary search tree must themselves be binary search trees.

# Binary Search Tree



Left  $\leq$  Node  $\leq$  Right

# Binary Search Tree



# Properties and Operations

- Note that under this definition, a BST never contains duplicate nodes.
- Some operations for BSTs:
  - InsertItem
  - DeleteItem
  - ItemExists
  - Traverse(Pre, In, Post)
  - Count
  - Height

# Binary Tree Algorithms

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

```
Node *MakeNode(int Data)  
{  
    Node *node = new Node;  
    node->data = Data;  
    node->left = 0;  
    node->right = 0;  
    return node;  
}
```

```
void FreeNode(Node *node){  
    delete node;  
}  
  
typedef Node* Tree;
```

# Finding an Item in a BST

- State the recursive algorithm in English for finding an item.

```
bool ItemExists(Tree tree, int Data){  
    if (tree == 0)  
        return false;  
    else if (Data == tree->data)  
        return true;  
    else if (Data < tree->data)  
        return ItemExists(tree->left, Data);  
    else  
        return ItemExists(tree->right, Data);  
}
```

Complexity

Best case?

Worst case?

# Insert an Item in a BST

- State the recursive algorithm in English for inserting an item.

```
void InsertItem(Tree &tree, int Data){  
    if (tree == 0)  
        tree = MakeNode(Data);  
    else if (Data < tree->data)  
        InsertItem(tree->left, Data);  
    else if (Data > tree->data)  
        InsertItem(tree->right, Data);  
    else  
        cout << "Error, duplicate item" << endl;  
}
```

Complexity

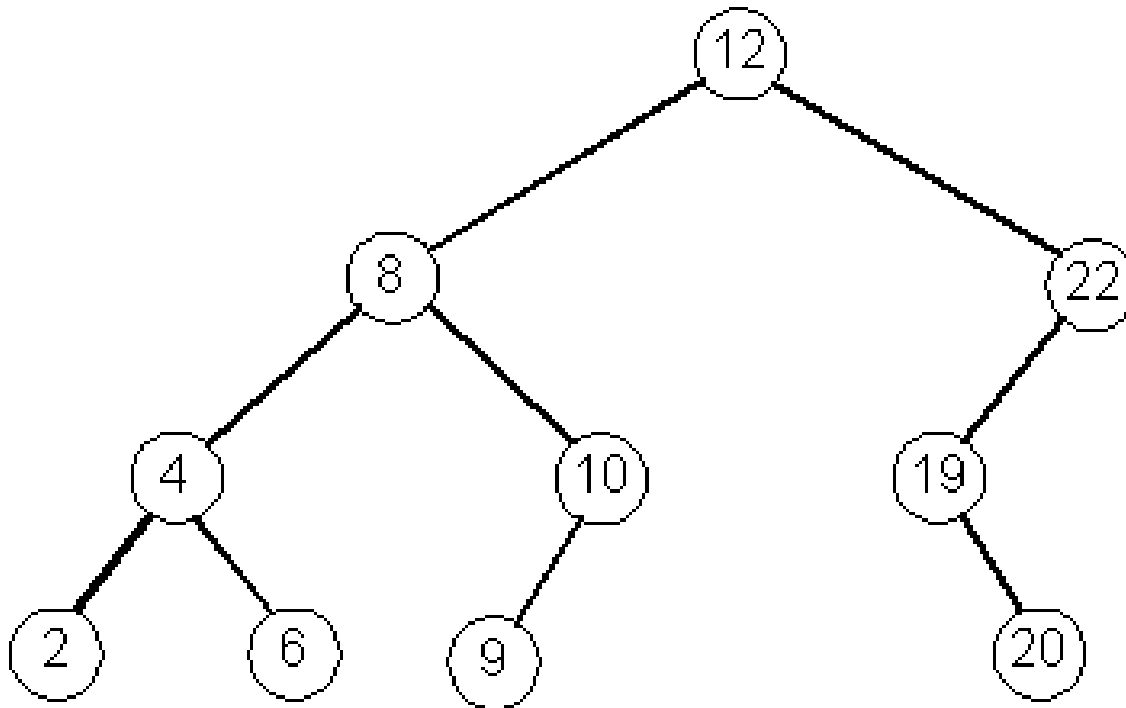
Best case?

Worst case?



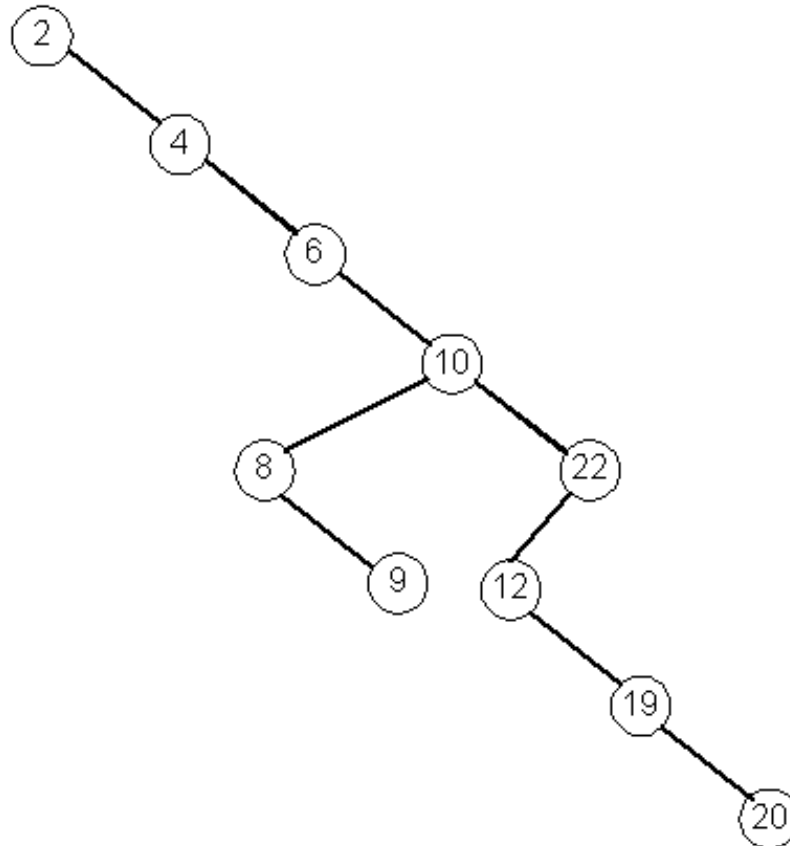
# Insert an Item in a BST

- Create a tree using these values (in this order):
  - 12, 22, 8, 19, 10, 9, 20, 4, 2, 6



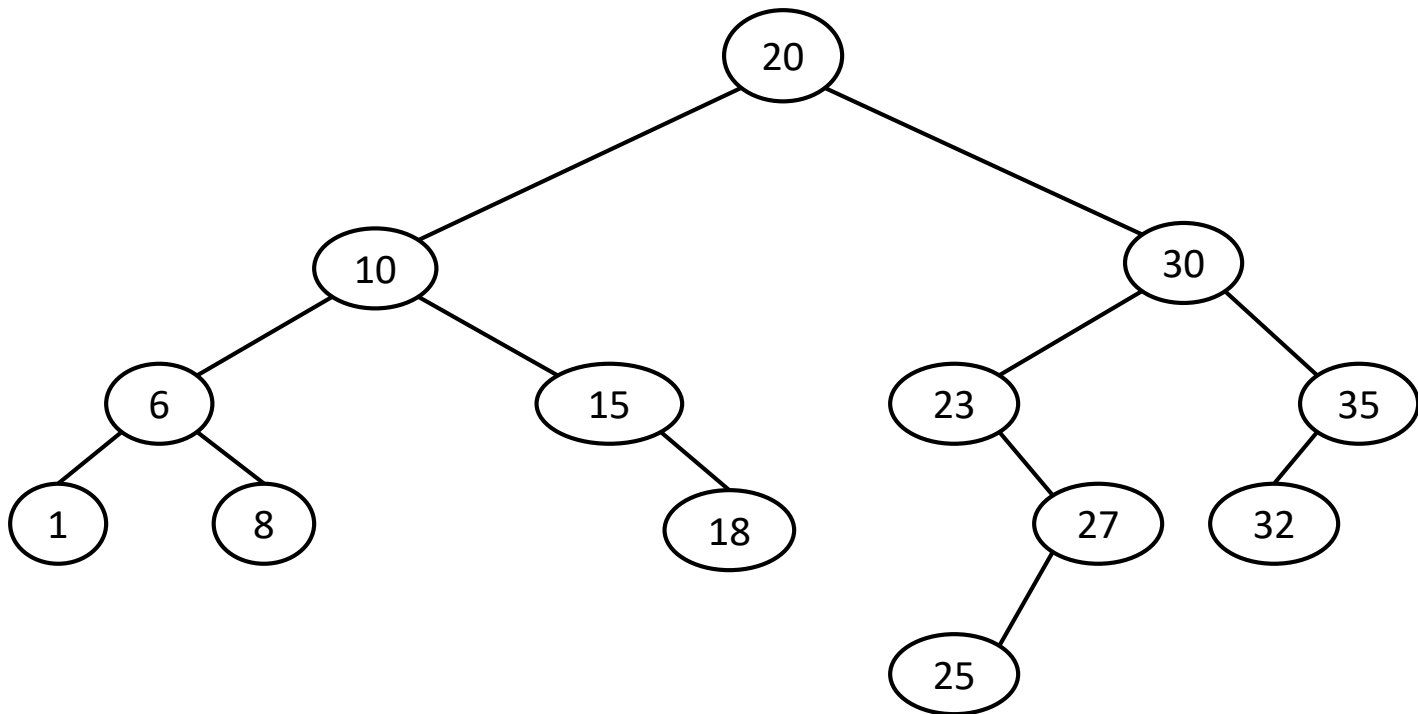
# Insert an Item in a BST

- Create a tree using these values (in this order):
  - 2, 4, 6, 10, 8, 22, 12, 9, 19, 20



# Deleting a Node

- The caveat of deleting a node is that, after deletion, the tree must still be a BST.



# Deleting a Node

- Case 1: The node to be deleted is a leaf node.
- Case 2: The node to be deleted has an empty left child but non-empty right child.
- Case 3: The node to be deleted has an empty right child but non-empty left child.
- Case 4: The node has both children non-empty.

# Case 1: Leaf Node

- Set the parent's pointer to this node to NULL.
- Release the memory of the leaf node.

## Case 2: Empty Left Child

- Replace the deleted node with its right child.

## Case 3: Empty Right Child

- Replace the deleted node with its left child.

## Case 4: Non-empty Left and Right Child

- Replace the data in the deleted node with its **predecessor** under in-order traversal.
- Delete the node that holds the predecessor.



# Deleting a Node

```
void DeleteItem(Tree &tree, int Data){
    if (tree == 0) return;
    else if (Data < tree->data)
        DeleteItem(tree->left, Data);
    else if (Data > tree->data)
        DeleteItem(tree->right, Data);
    else { // (Data == tree->data)
        if (tree->left == 0){
            Tree temp = tree;
            tree = tree->right;
            FreeNode(temp);
        }
        else if (tree->right == 0){
            Tree temp = tree;
            tree = tree->left;
            FreeNode(temp);
        }
        else{
            Tree pred = 0;
            FindPredecessor(tree, pred);
            tree->data = pred->data;
            DeleteItem(tree->left, tree->data);
        }
    }
}
```

```
void FindPredecessor(Tree tree, Tree
&predecessor){
    predecessor = tree->left;
    while (predecessor->right != 0)
        predecessor = predecessor->right;
}
```

- We are replacing the data in the node, not the node itself
- Because the predecessor comes from the left subtree:
  - It must be less than everything in the right subtree.
  - It must be greater than everything in the left subtree.
- The recursive deletion of the predecessor's node will lead to one of the simpler cases.

# Summary

- Binary tree
- Binary search tree
  - Finding an item
  - Insertion
  - Deletion

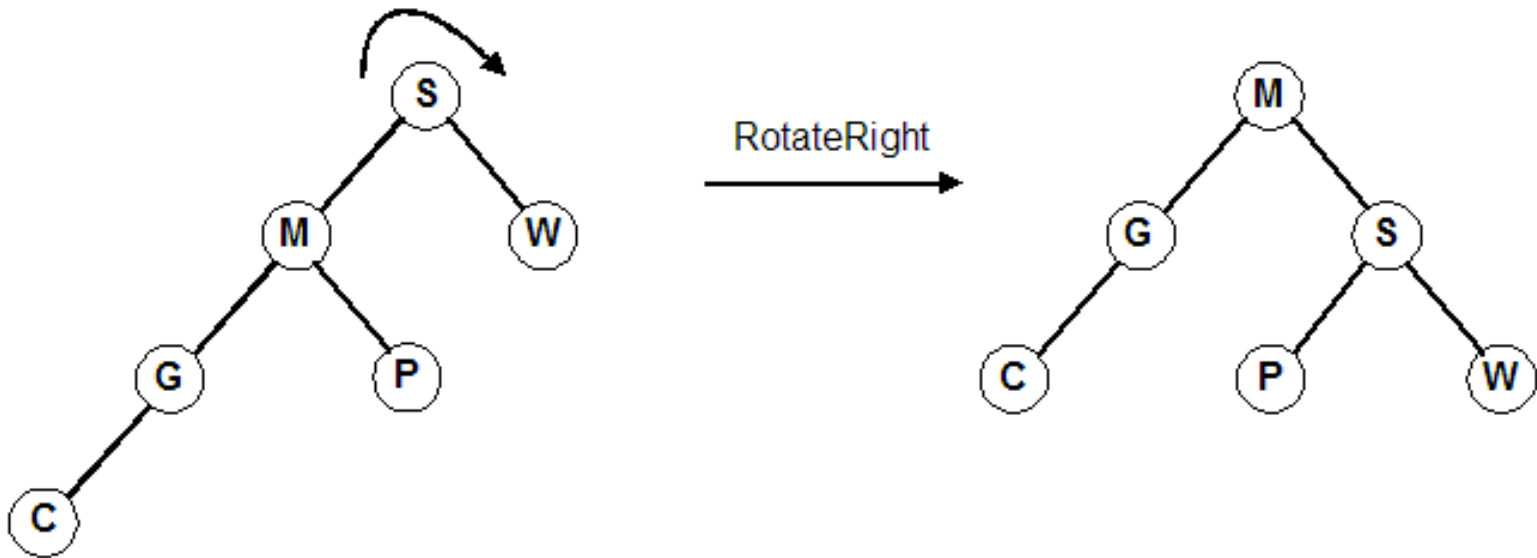
# Rotating Nodes

# Rotating Nodes

- Rotation is a fundamental technique performed on BSTs.
- Two types of rotations:
  - Left
  - Right
- Promoting a node is the same as rotating around the node's parent.
- There is no direction in promotion.
- You can rotate about any node that has children.
- After the rotation, the sort order is preserved.
  - The resulting is STILL a BST

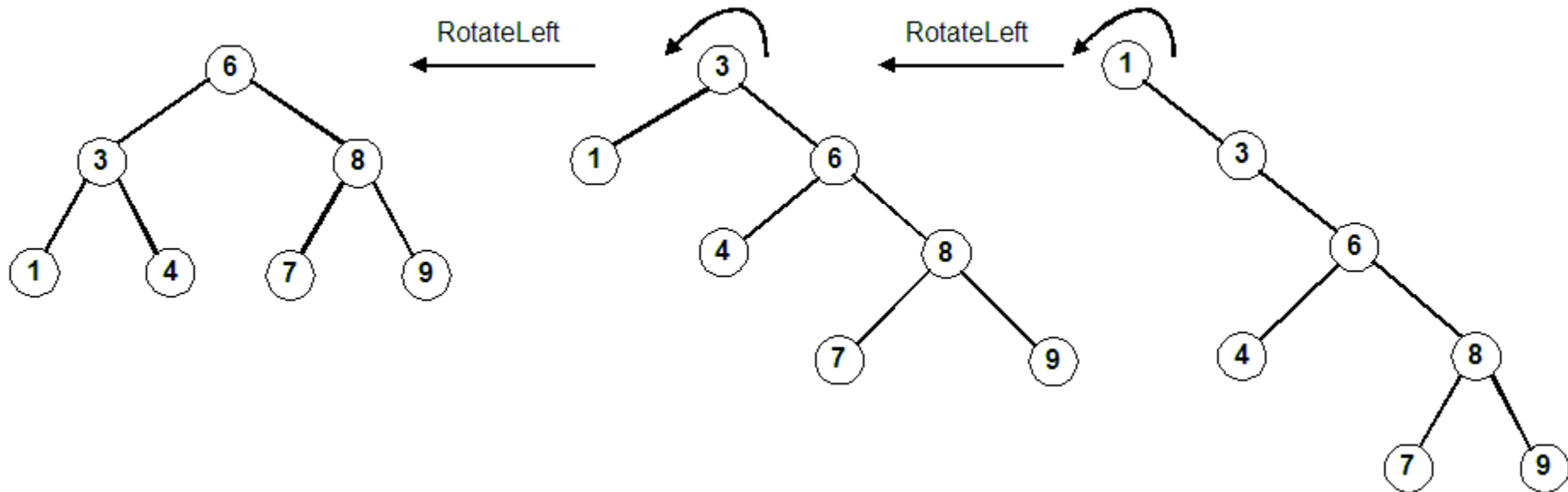
# Right Rotation

- Rotate right around the root, S (Same as promoting M)



# Left Rotation

- Rotate left twice around the root. First around 1, then around 3. (Same as promoting 3 then 6)



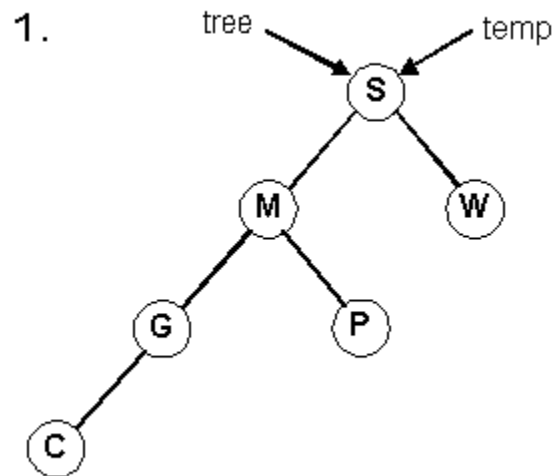
# Rotating Left and Right

```
void RotateRight(Tree &tree){  
    Tree temp = tree;  
    tree = tree->left;  
    temp->left = tree->right;  
    tree->right = temp;  
}
```

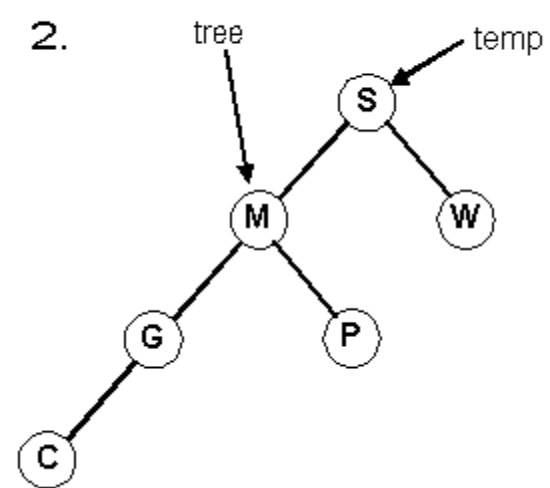
```
void RotateLeft(Tree &tree){  
    Tree temp = tree;  
    tree = tree->right;  
    temp->right = tree->left;  
    tree->left = temp;  
}
```

# Step by Step

1. `Tree temp = tree;`



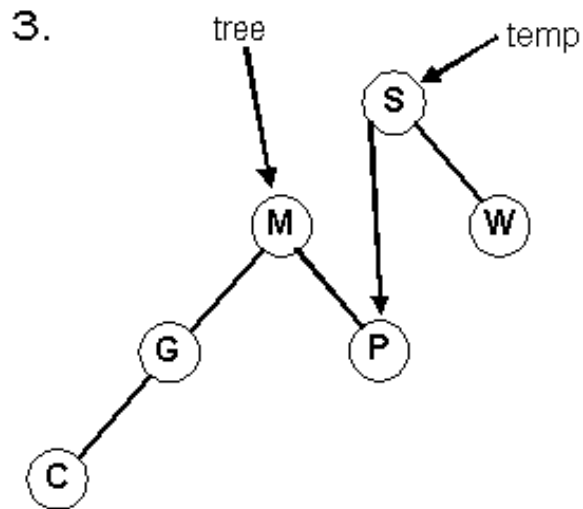
2. `tree = tree->left;`



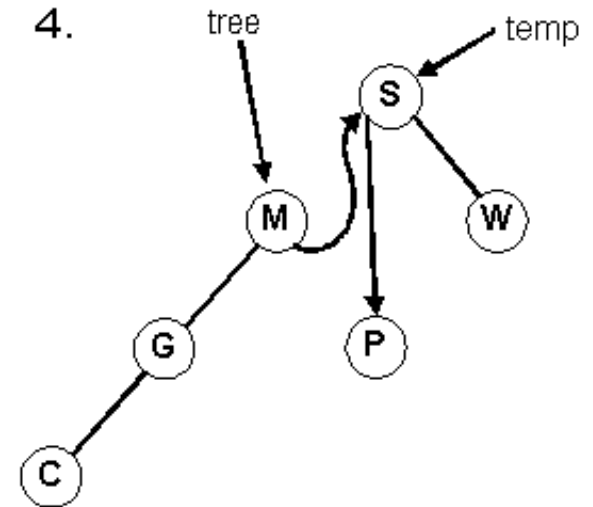


# Step by Step

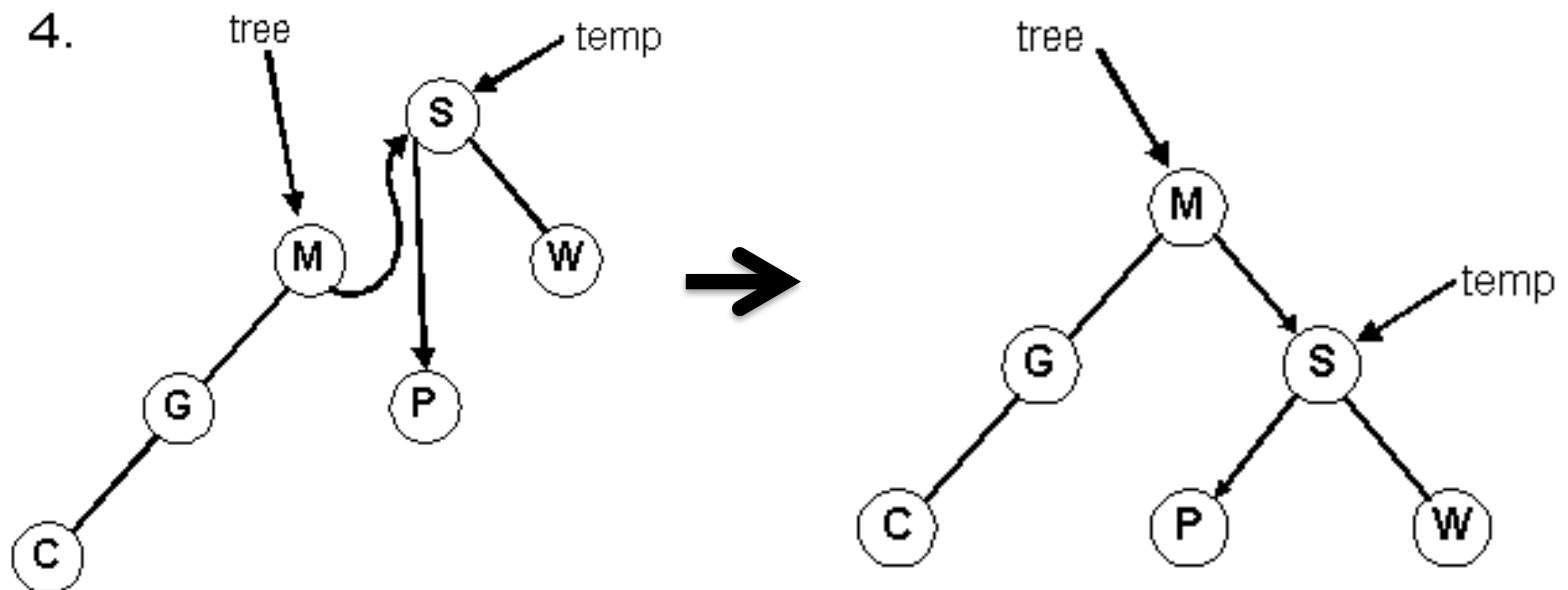
`temp->left = tree->right;`



`tree->right = temp;`



# Adjusting the Diagram

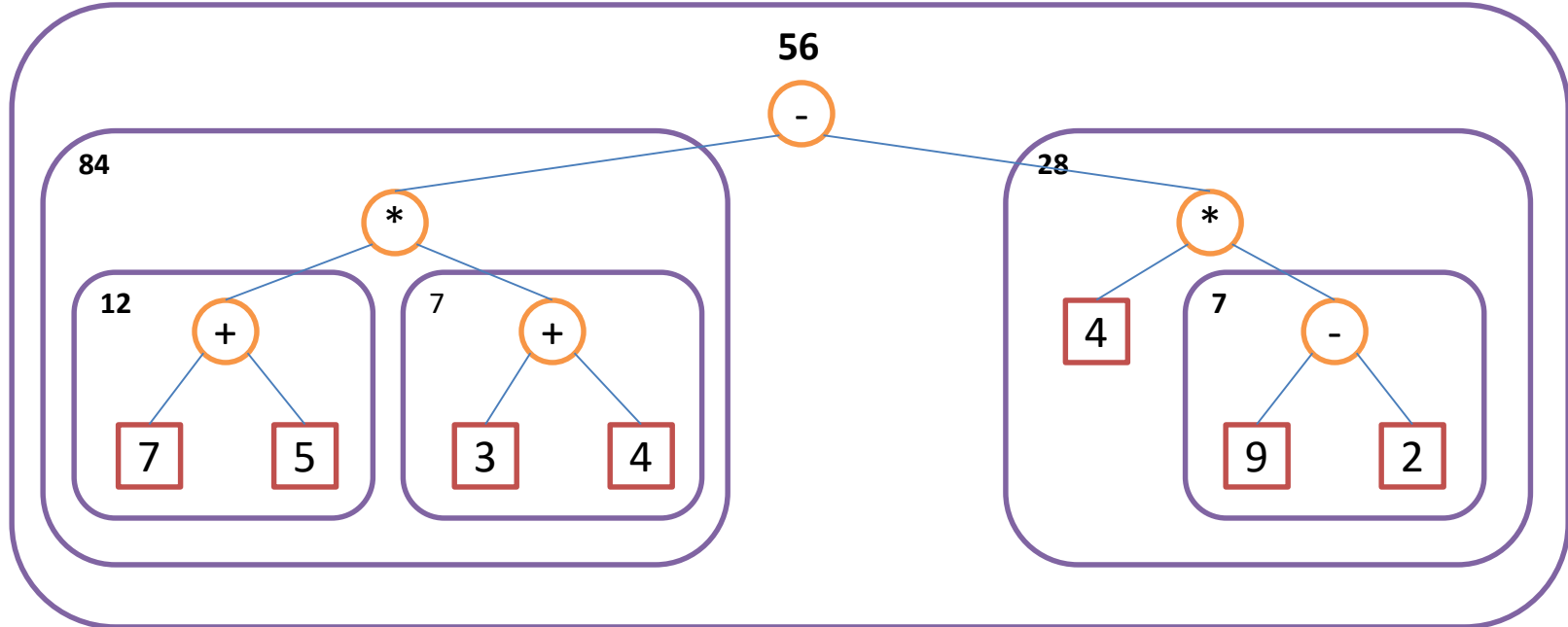


# Expression Trees

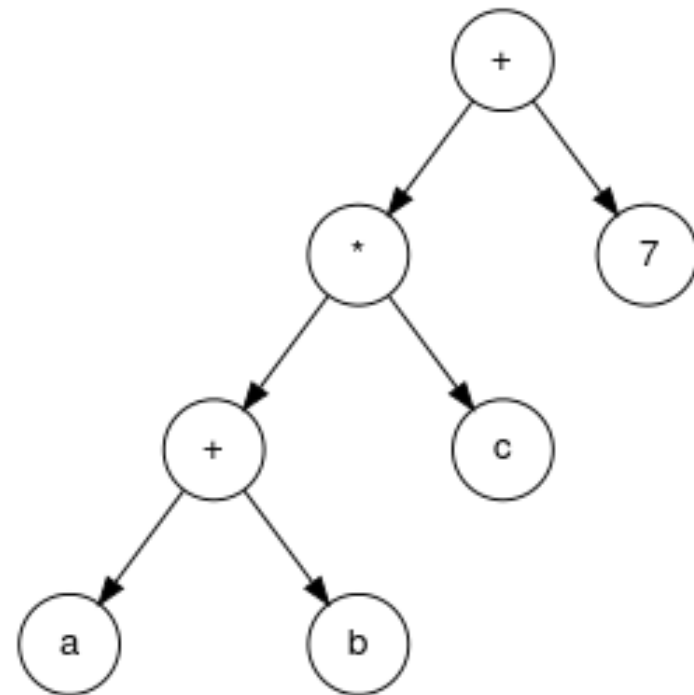
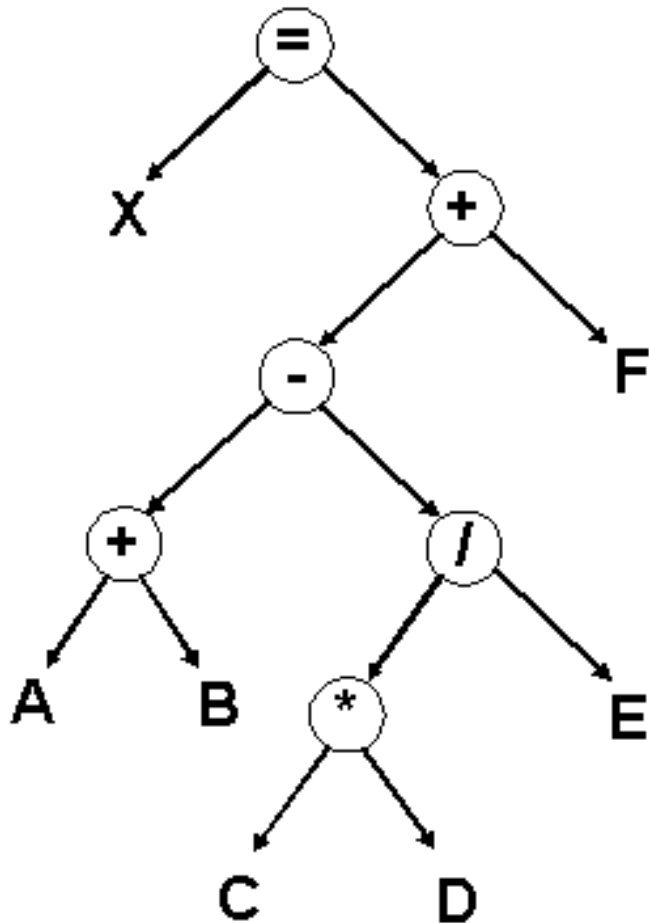
# Expression Trees

- Expression trees are like binary trees, but they are not “sorted” in the usual way.
- Expression trees are a way to
  - Solve arithmetic expressions.
  - Compile a language

# Expression Tree



# Expression Trees



# Evaluate an expression tree

```
1.  int evaluate(Tree node){
2.      if(node->type == NODE_OPERAND)
3.          return node->opd;
4.      else {
5.          int left = evaluate(node->left);
6.          int right = evaluate(node->right);
7.
8.          switch(node->opt){
9.              case '+': return (left+right);
10.             case '-': return (left-right);
11.             case '*': return (left*right);
12.             case '/': return (left/right);
13.             default: return -1;
14.          }
15.      }
16.  }
```

- We only consider binary operators here and the list of operators you need to consider includes “+”, “-”, “\*” and “/”.