CS280 – Data Structures

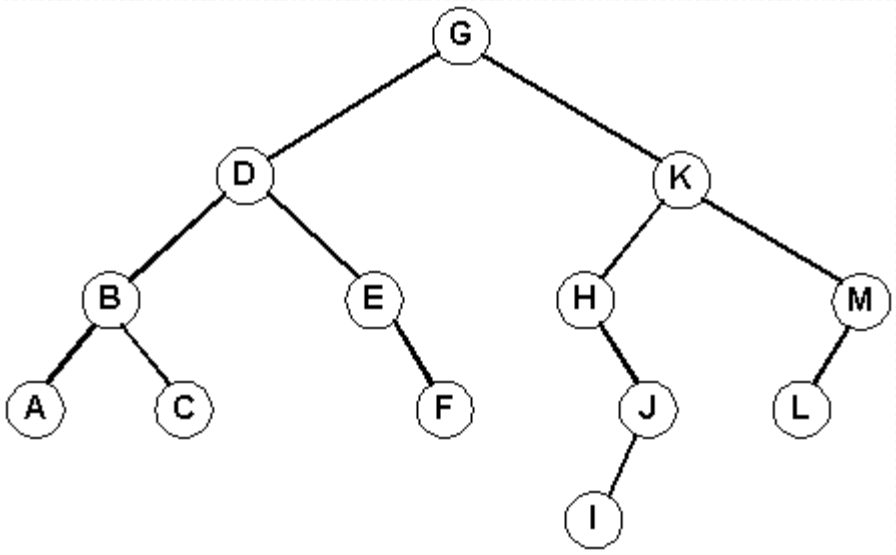
Binary Search Trees

Overview

- Introduction
- Deleting A Node
- Rotating Nodes
- Splay Trees
- Expression Trees

Definition

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, and 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.



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
- Count and Height (Depth) are straight-forward
- The traverse routines are usual.
- ItemExists and InsertItem are also trivial.

Binary Tree Algorithms

- Assume we have these definitions:

```
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;
```

I-Finding an item in a BST:

As always,

- 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);
}
```

II- Insert an item in a BST:

As always,

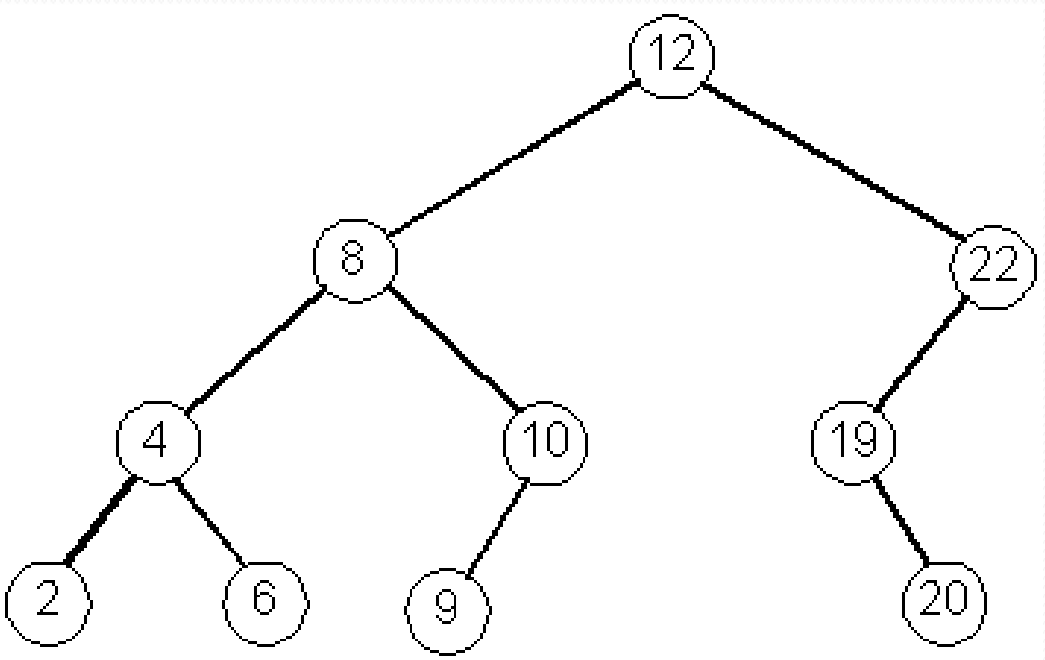
- State the recursive algorithm in English for insert an item.

```
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;
}
```

Exercise:

- Create a tree using these values (in this order):

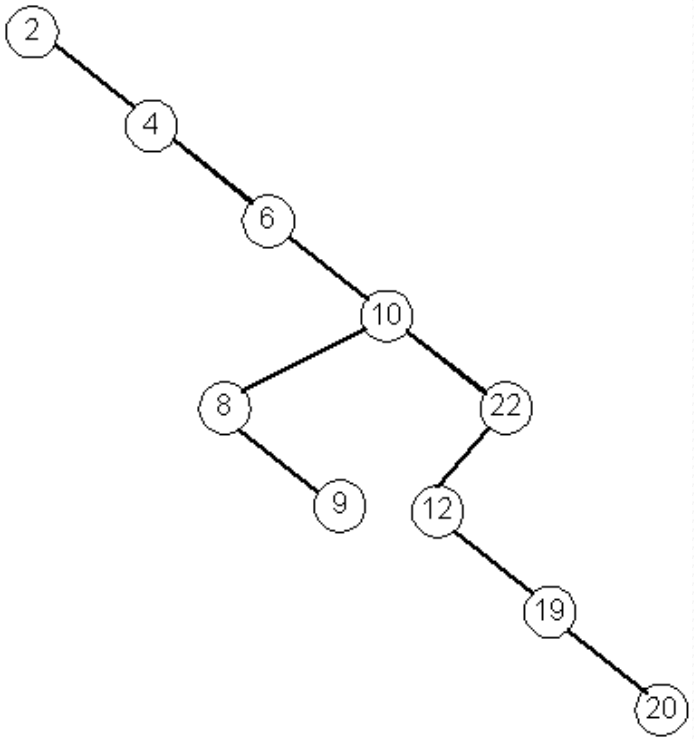
12, 22, 8, 19, 10, 9, 20, 4, 2, 6



Exercise:

- Create a tree using these values (in this order):

2, 4, 6, 10, 8, 22, 12, 9, 19, 20

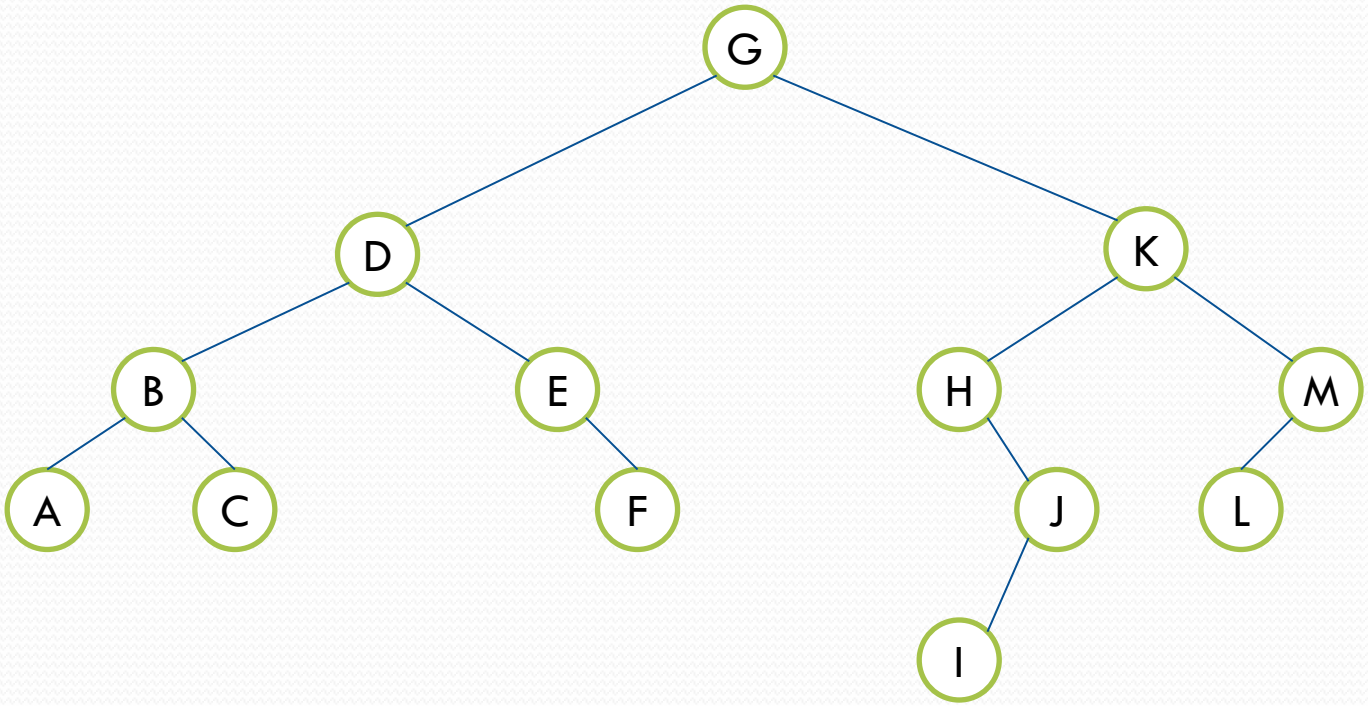


Search Times

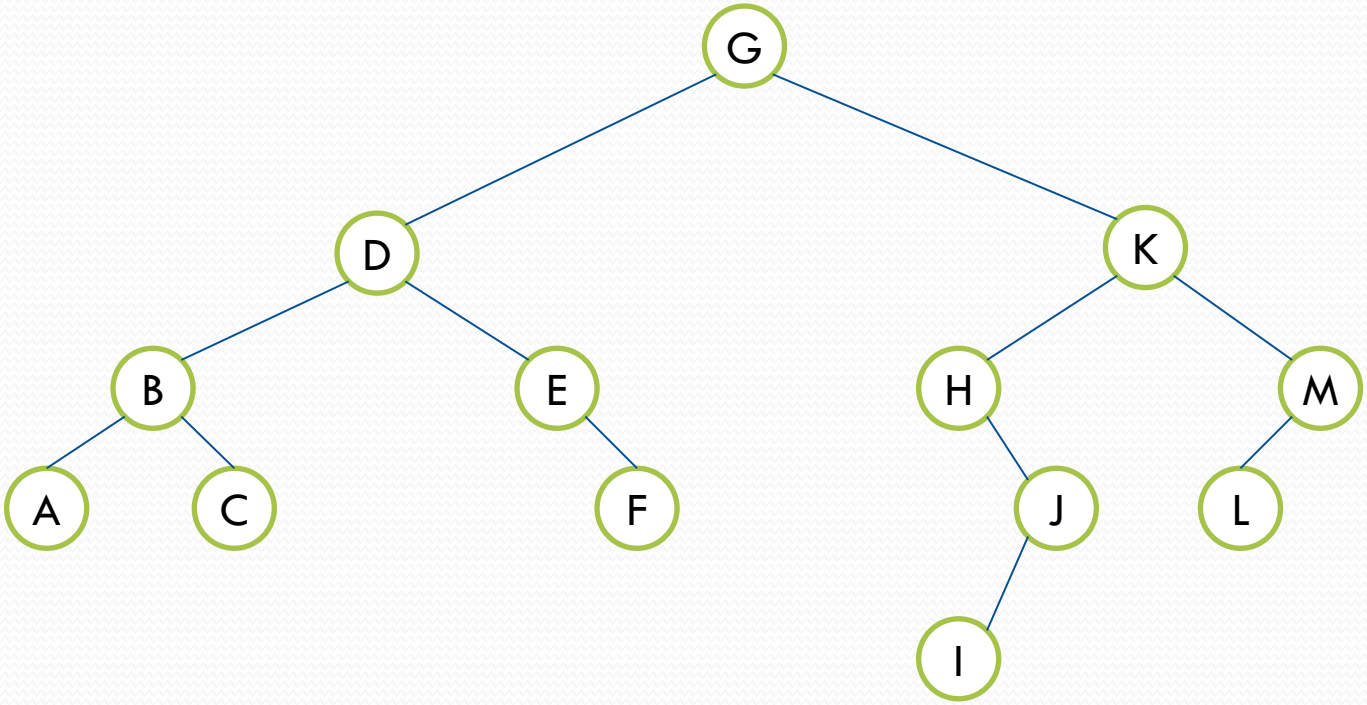
- What is the worst case time complexity for searching a BST?
  - Best?
- What causes the best/worst cases?
- BST algorithms are highly dependent on the input data.
- One solution is to balance the tree.
- Balancing BSTs can be expensive and the algorithms are more complex than what we've seen so far.

Deleting a Node

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

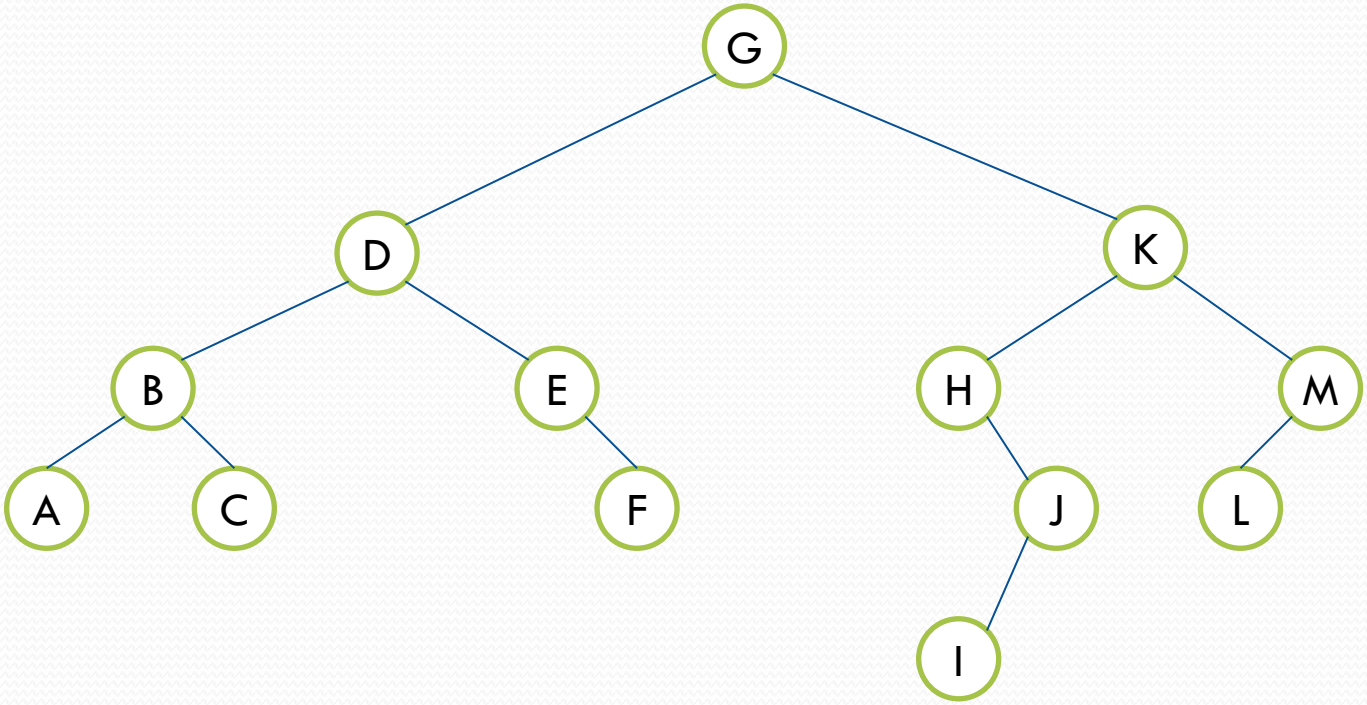


Deleting a Leaf Node



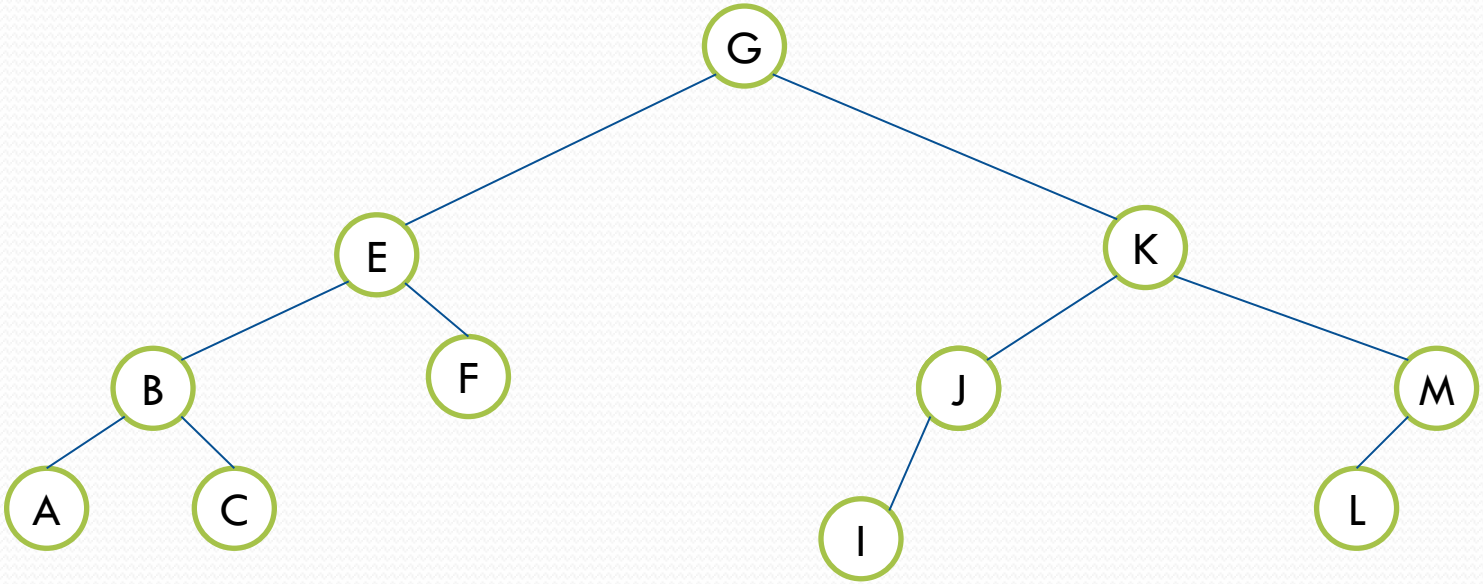
Set the parent's pointer to this node to NULL.

The Node has empty left, non-empty right child



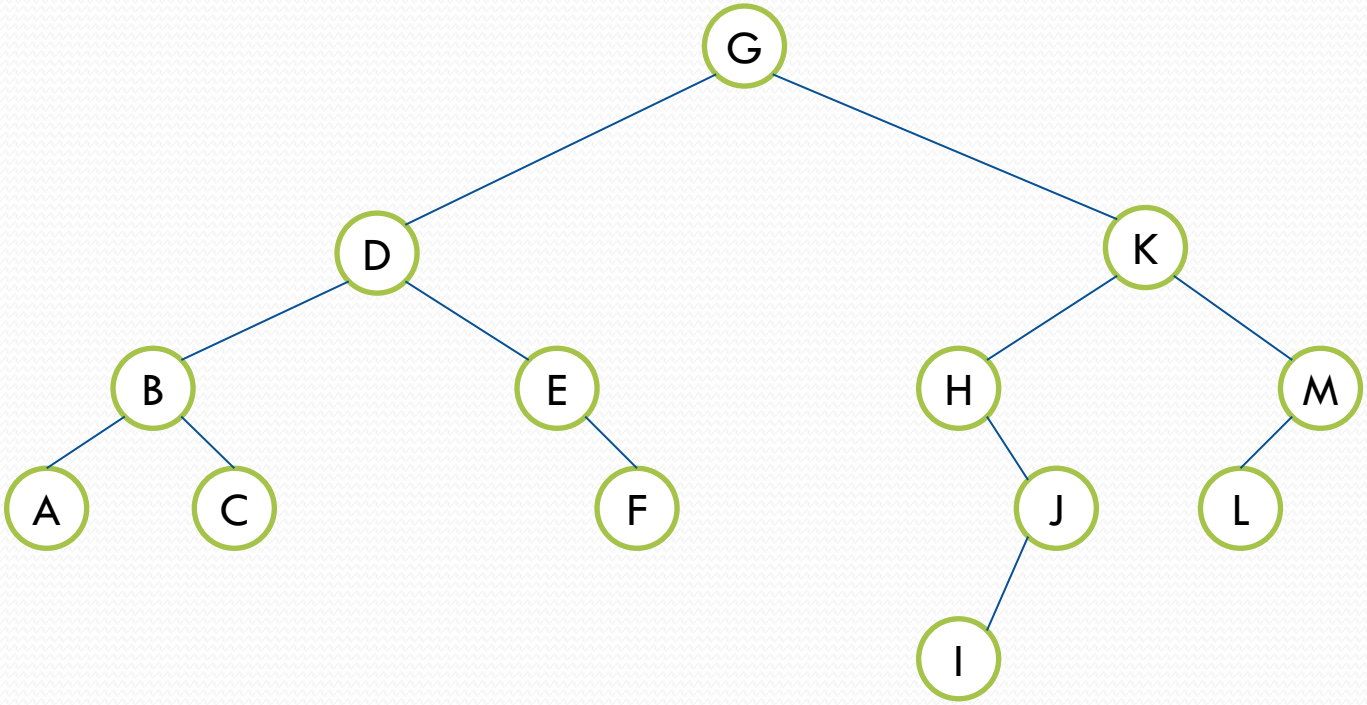
- Replace the deleted node with its right child.
- "Promote" the right child

The Node has empty left, non-empty right child



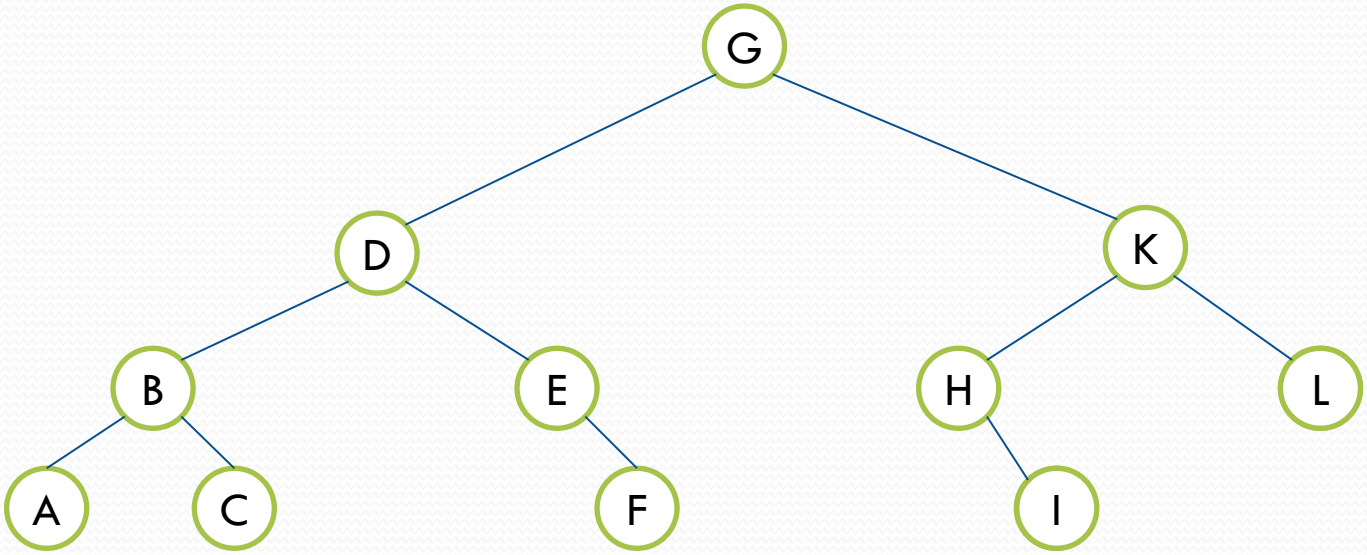
- Replace the deleted node with its right child.
- "Promote" the right child

The Node has empty right, non-empty left child



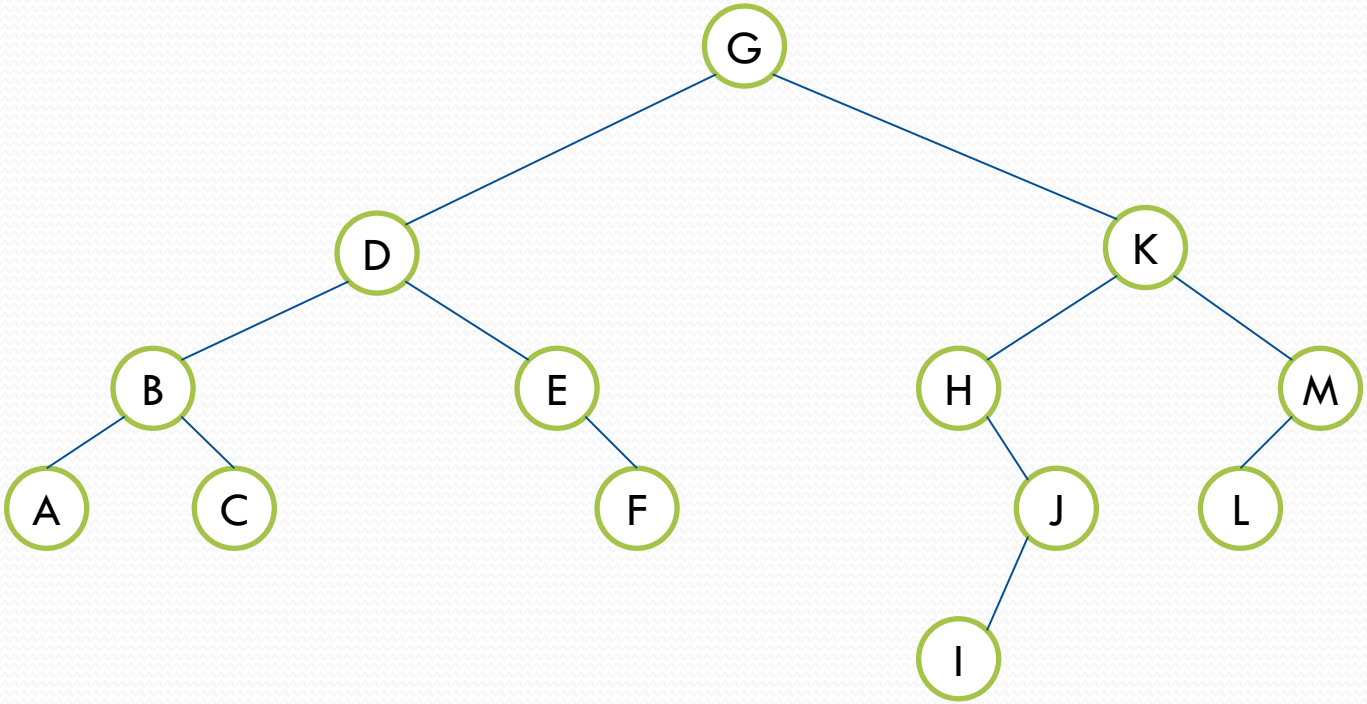
- Replace the deleted node with its left child.
- "Promote" the left child

The Node has empty right, non-empty left child



- Replace the deleted node with its left child.
- "Promote" the left child

The node to be deleted has both children non-empty



- Replace the data in the deleted node with its predecessor under in-order traversal.
- Delete the node that held the predecessor.
- The predecessor will be the rightmost node in the left subtree of the deleted node.
- In the diagram, this is **A** if **B** is deleted, **C** if **D** is deleted, **F** is **G** is deleted, and **J** if **K** is deleted.

```
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;
}
```

Notes:

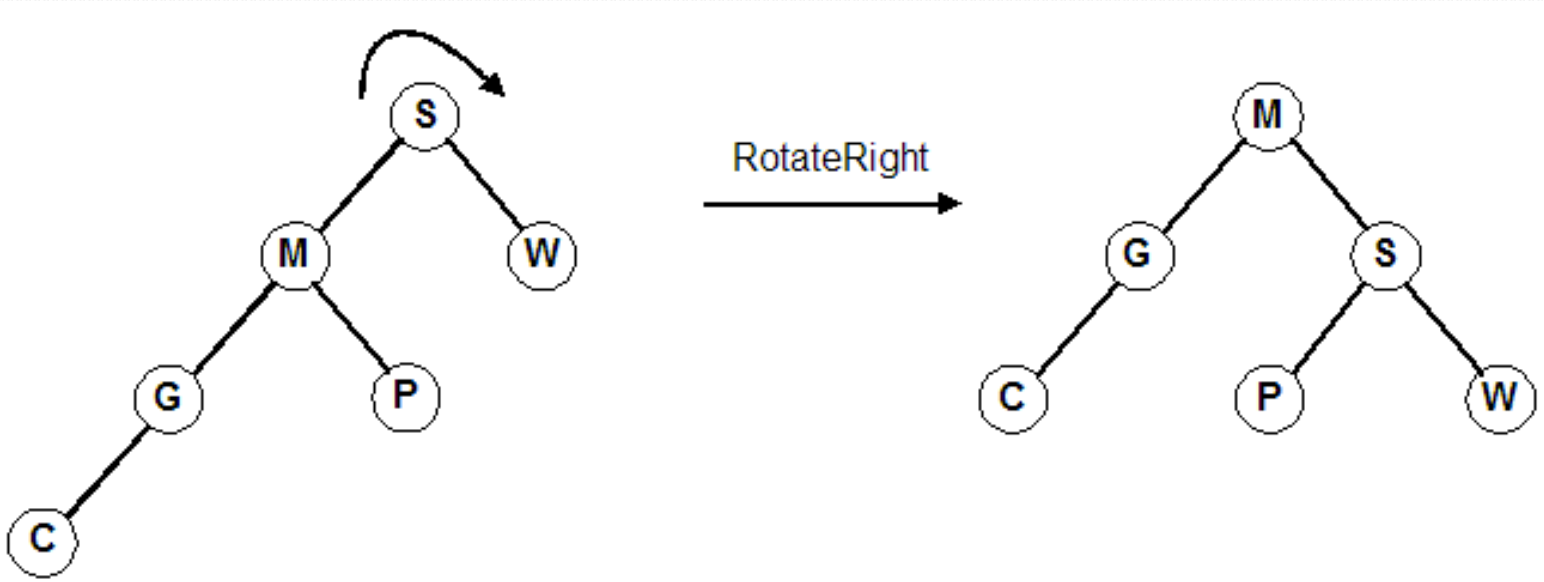
- We are replacing the data in the node, *not* the node itself
- Because the predecessor comes from the left subtree:
  - It must be ***less*** than everything in the right subtree.
  - It must be ***greater*** than everything in the left subtree.
- The recursive deletion of the predecessor's node will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

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, promotion is unambiguous)
- You can rotate about any node that has children
- *After the rotation, the sort order is preserved.*
  - *The resulting is STILL a BST*

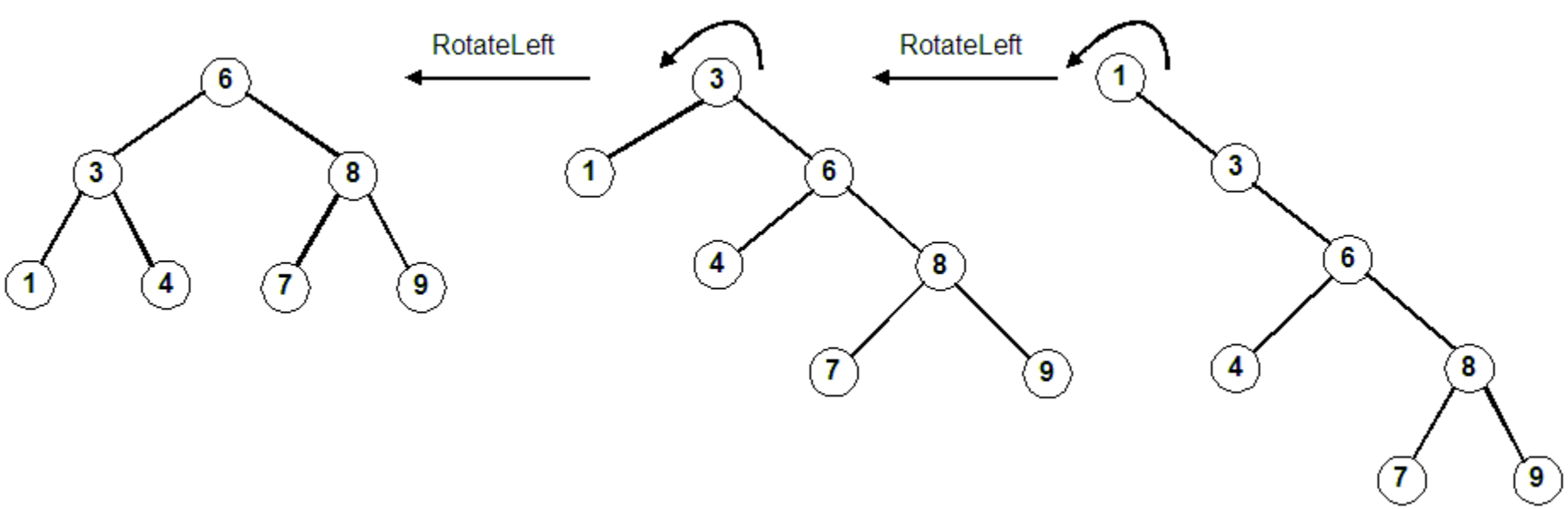
Rotating Right

- In this example we are rotating ***right*** around the root S
  - In other words, we are promoting M



Rotating Left

- In this example we are rotating ***left*** twice around the root (1 then 3).
  - In other words, we are promoting 3 then 6

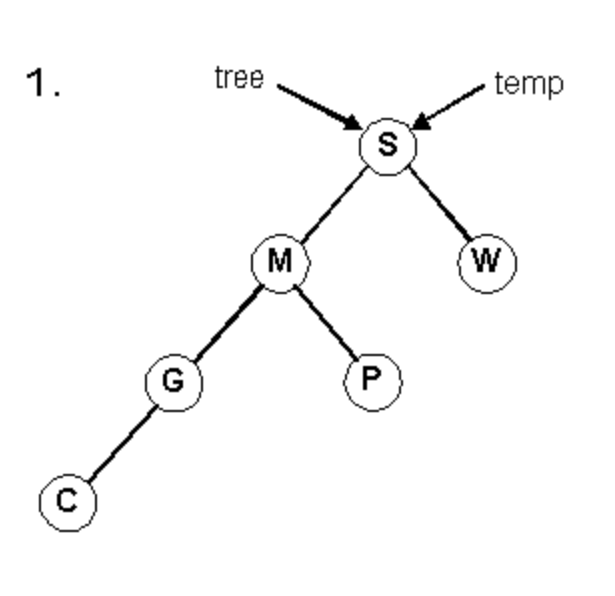


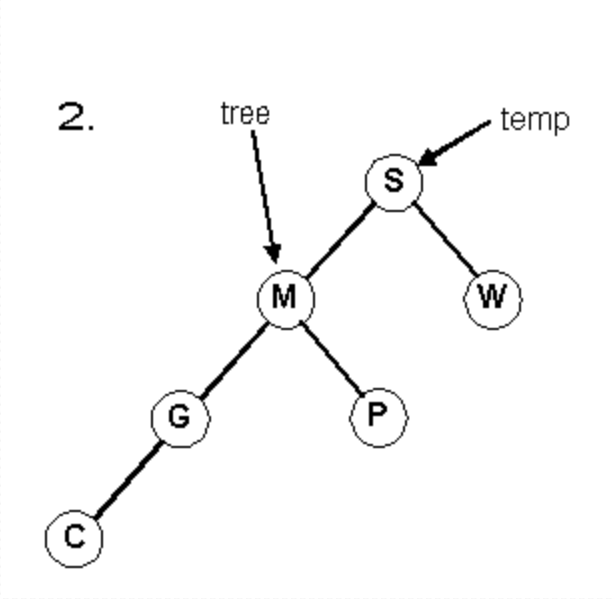
Rotating Left and Right

```
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

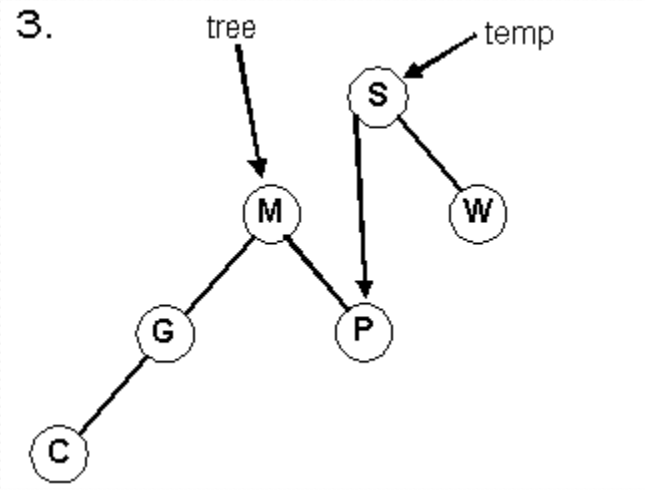




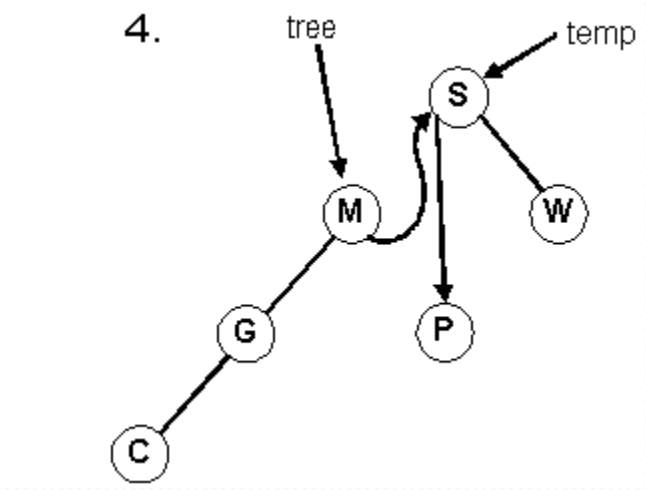

```
1.Tree temp = tree;
```

```
2.tree = tree->left;
```

Step by step




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




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

Adjusting the diagram