Data Structures and Algorithms

Binary Trees

3.1 Trees

3.2 Binary Trees

3.3 Traversing a Binary Tree

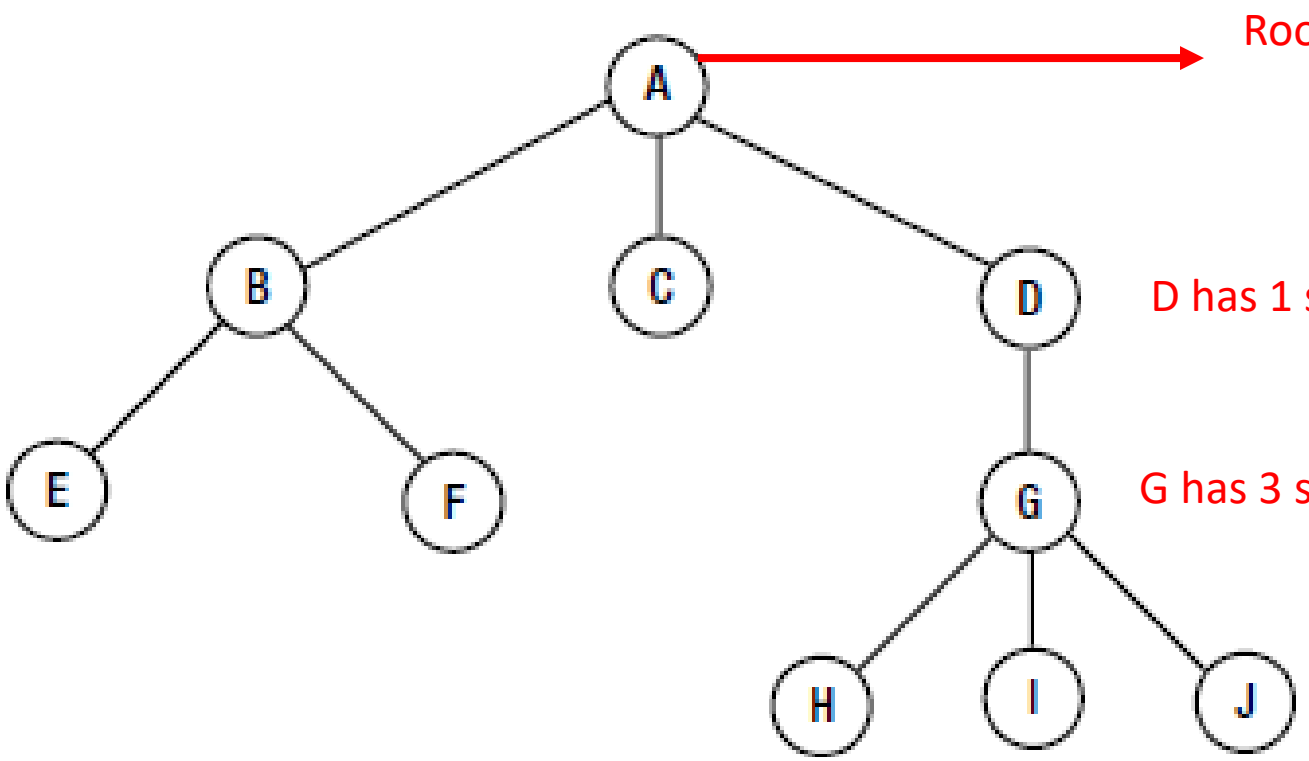
3.4 How to represent a Binary Tree

3.1 Trees

A *tree* is a finite set of nodes such that the following are both true:

- There is one specially designated node called the *root* of the tree.
- The remaining nodes are partitioned into $m \geq 0$ disjoint sets $T_1, T_2, \ldots, T_m$, and each of these sets is a tree.

The trees $T_1, T_2, \ldots, T_m$, are called the *subtrees* of the root. We use a recursive definition since recursion is an innate characteristic of tree structures. Figure 8-1 illustrates a tree. By convention, the root is drawn at the top, and the tree grows downward.



Root A with 3 subtrees => Degree of A = 3

D has 1 subtree => Degree of D = 1

G has 3 subtrees => Degree of G = 3

The *degree* of a node is the number of subtrees of the node. Think of it as the number of lines leaving the node.
For example, degree(A) = 3, degree(B) = 2, degree(C) = 0, degree(D) = 1, and degree(G) = 3.

We use the terms *parent*, *child*, and *sibling* to refer to the nodes of a tree.
For example, the parent A has three children, which are B, C, and D; the parent B has two children, which are E and F.

Sibling nodes are child nodes of the same parent.
For example, B, C, and D are siblings; E and F are siblings; and H, I, and J are siblings.

- In a tree, a node may have several children but, only one parent. The root has no parent.
- A *leaf /terminal* node is a node of degree 0 (i.e. a node with no children).
- A *branch* node is a non leaf node.
- The examples below are based on the tree in slide 3.
- C, E, F, H, I, and J are leaves, while A, B, D, and G are branch nodes.
- The *moment* of a tree is the number of nodes in the tree. The moment = 10.
- The *weight* of a tree is the number of leaves in the tree. The weight = 6.
- The *level* (or *depth*) of a node is the number of branches that must be traversed on the path to the node from the root.:
  - The root is at level 0;
  - B, C, and D are at level 1;
  - E, F, and G are at level 2;
  - H, I, and J are at level 3.
- The *height* of a tree is the number of levels in the tree. The height = 4.

3.2 Binary Trees

A *binary tree* is a classic example of a nonlinear data structure—compare this to a linear list where we identify a first item, a next item, and a last item. A binary tree is a special case of the more general *tree* data structure, but it is the most useful and most widely used kind of tree. A binary tree is best defined using the following recursive definition:

A *binary tree*

a. is empty

or

b. consists of a root and two subtrees—a left and a right—with each subtree being a binary tree

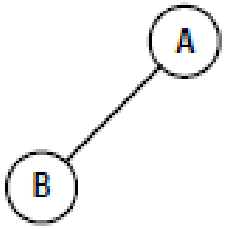
In a binary tree, a node may have 0, 1, or 2 children maximum.

Examples of Binary Trees

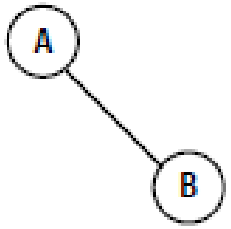
Example 1:

Here's a binary tree with one node, the root:

Example 2:

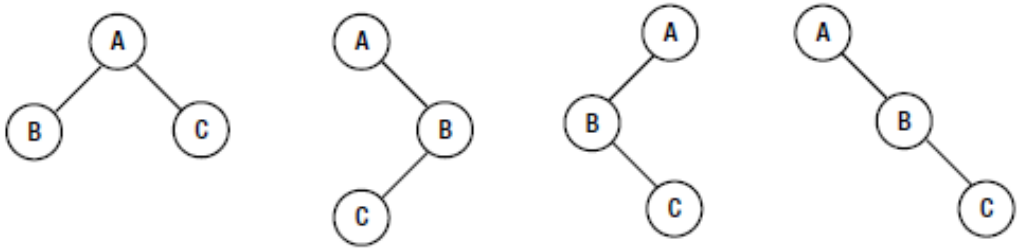


is a different binary tree from



Example 3:

Here are binary trees with three nodes:

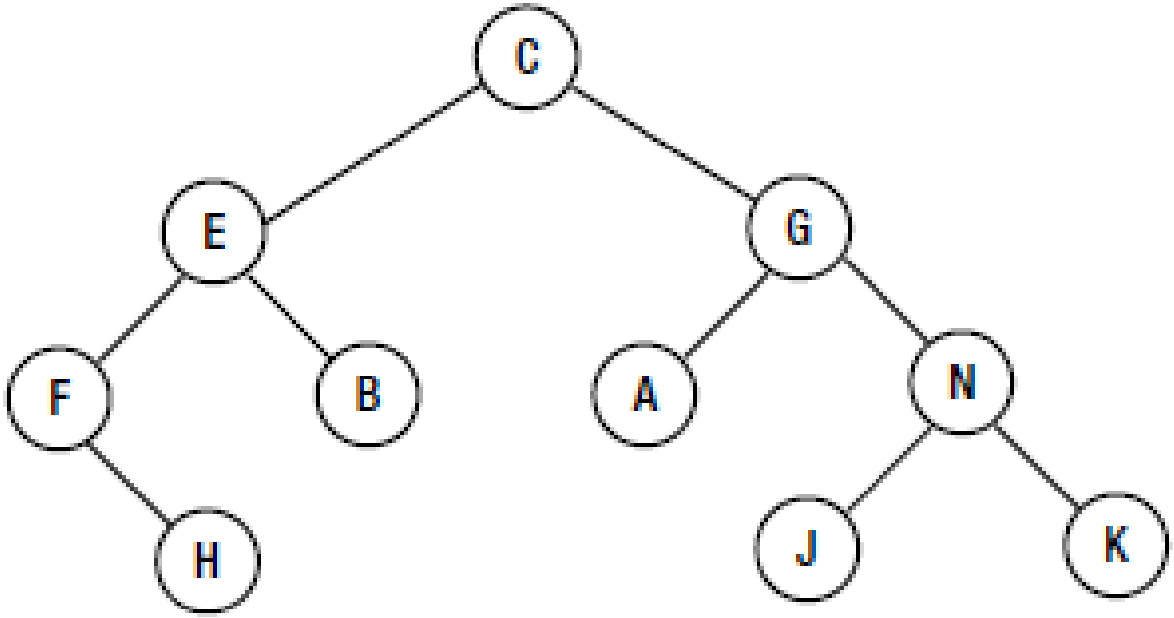


Example 4:

Here are binary trees with all left subtrees empty and all right subtrees empty:

Examples of Binary Trees

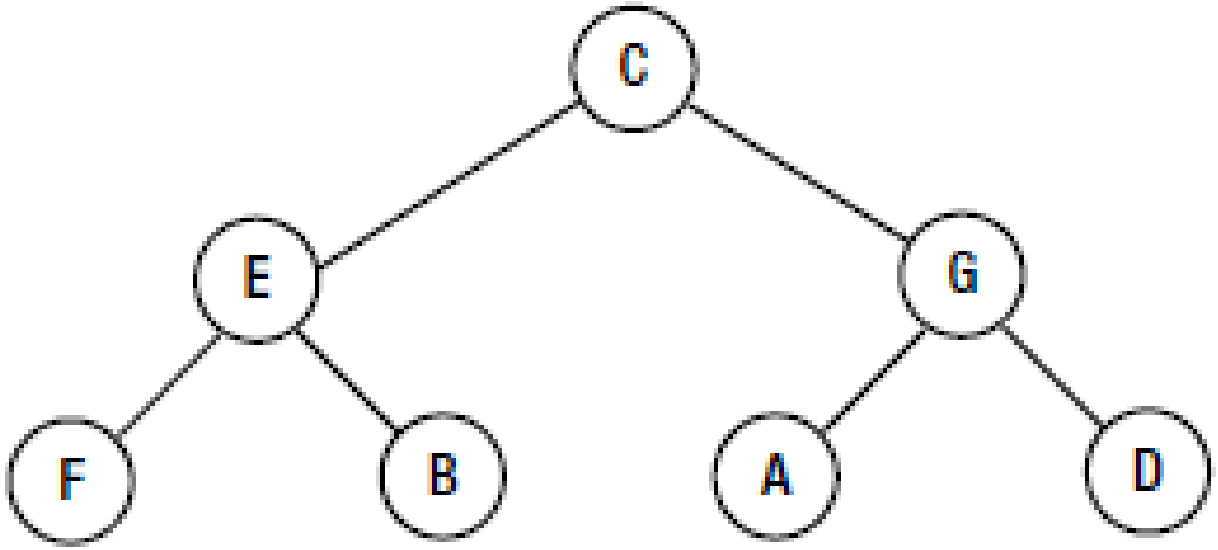
Example 5: General Binary Tree



Examples of Binary Trees

Example 5: A complete binary tree

A complete binary tree is a binary tree where each node, except the leaves, has exactly two subtrees.



3.3 Traversing a Binary Tree

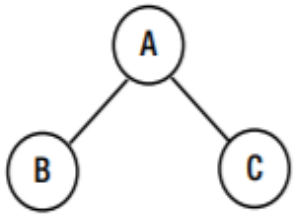
- In many applications, we need to traverse a binary tree which means visit all the nodes of the binary tree in some systematic way. For now, we'll think of "visit" as simply printing the data in the node.
- For a tree of n nodes, there are n! (n factorial) ways to visit them, assuming that each node is visited once.
- For example, we can visit the nodes of a tree with the three nodes A, B, and C, in any of the following ways: ABC, ACB, BCA, BAC, CAB, and CBA.
- We will discuss three ways to visit a binary tree: pre-order, in-order, and post-order—that are useful.

Pre-order traversal

1. Visit the root.
2. Traverse the left subtree in pre-order.
3. Traverse the right subtree in pre-order.

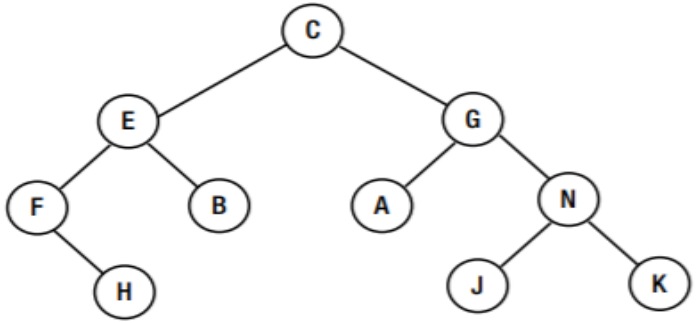
Note that the traversal is defined recursively. In steps 2 and 3, we must reapply the definition of pre-order traversal, which says “visit the root, then the left subtree, then the right subtree.”

The *pre-order* traversal of this tree



is A B C.

The *pre-order* traversal of this tree



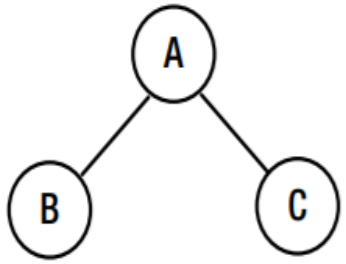
is C E F H B G A N J K.

In-order traversal

1. Traverse the left subtree in in-order.
2. Visit the root.
3. Traverse the right subtree in in-order.

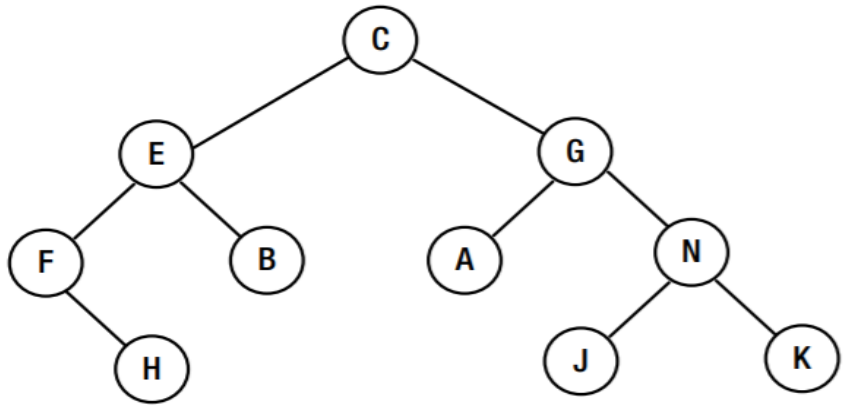
Here we traverse the left subtree first, then the root, and then the right subtree.

The *in-order* traversal of this tree



is B A C.

The *in-order* traversal of this tree



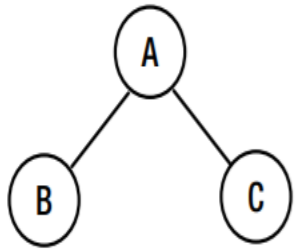
is F H E B C A G J N K

Post-order traversal

1. Traverse the left subtree in post-order.
2. Traverse the right subtree in post-order.
3. Visit the root.

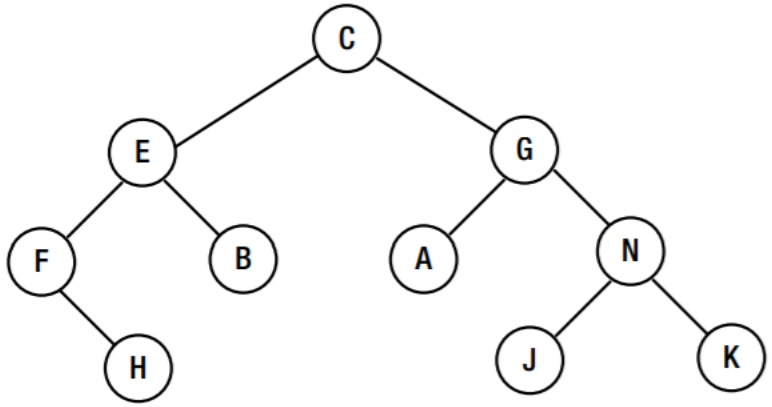
Here we traverse the left and right subtrees before visiting the root.

The *post-order* traversal of this tree



is B C A.

The *post-order* traversal of this tree



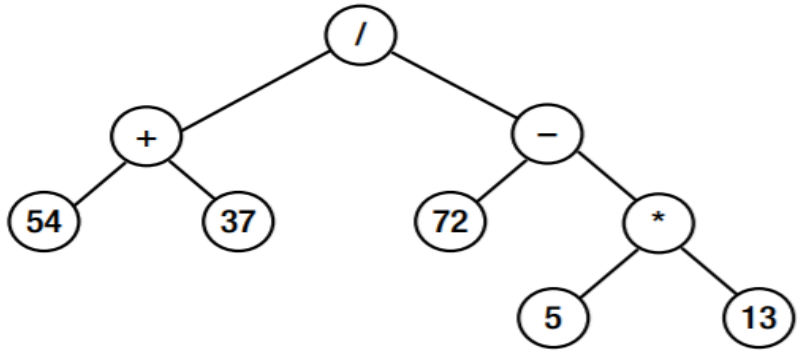
is H F B E A J K N G C.

Example

consider a binary tree that can represent the following arithmetic expression:

(54 + 37) / (72 – 5 * 13)

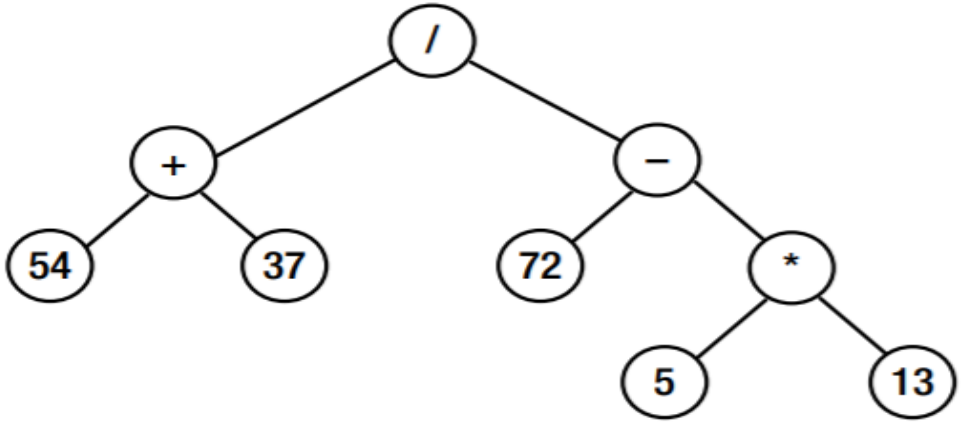
Here is the tree:



The leaves of the tree contain the operands, and the branch nodes contain the operators.

Given a node containing an operator, the left subtree represents the first operand, and the right subtree represents the second operand.

Example



The pre-order traversal is: / + 54 37 – 72 * 5 13

The in-order traversal is: 54 + 37 / 72 – 5 * 13

The post-order traversal is: 54 37 + 72 5 13 * - /

Animation

Animation of the pre-order, in-order, and post-order traversal algorithms can be found at this webpage:

https://towardsdatascience.com/4-types-of-tree-traversal-algorithms-d56328450846

Click the link and scroll down till around the middle of the page where you can find the name of each of the above three algorithms.

PreOrder Traversal (Code)

```
// preOrder Traversal --> Node Left Right (NLR)
public void preOrder() {
        preOrderTraversal(root);
} // end preOrder
// preOrderTraversal --> NLR
private void preOrderTraversal(TreeNode current) {
   if (current!=null) {
      System.out.print(current.data + " "); // visit the node  (N)
      preOrderTraversal(current.left);    // traverse the left subtree in preorder (L)
      preOrderTraversal(current.right);   // traverse the right subtree in preorder (R)
   }
} // end preOrderTraversal
```

InOrder Traversal (Code)

```
// inOrder Traversal --> Left Node Right (LNR)
public void inOrder() {
    inOrderTraversal(root);
} // end inOrder
// inOrderTraversal  --> LNR
private void inOrderTraversal(TreeNode current) {
    if (current!=null) {
       inOrderTraversal(current.left);     // traverse the left subtree in inorder (L)
       System.out.print(current.data + " "); // visit the node  (N)
       inOrderTraversal(current.right);    // traverse the right subtree in inorder (R)
    }
} // end inOrderTraversal
```

PostOrder Traversal (Code)

```
// postOrder Traversal --> Left Right Node (LRN)
public void postOrder() {
        postOrderTraversal(root);
} // end postOrder
// postOrderTraversal --> LRN
private void postOrderTraversal(TreeNode current) {
   if (current!=null) {
      postOrderTraversal(current.left);    // traverse the left subtree in postorder (L)
      postOrderTraversal(current.right);   // traverse the right subtree in postorder (R)
      System.out.print(current.data + " ");  // visit the node  (N)
   }
} // end postOrderTraversal
```

3.4 How to represent a Binary Tree

Each node of a binary tree consists of three fields:

- a field containing the data at the node
- a reference to the left subtree
- a reference to the right subtree.

We can begin by writing the class TreeNode which is a bit similar to the class Node used with the LinkedList class.

```
public class TreeNode {
   NodeData data;
   TreeNode left, right;
   public TreeNode(NodeData d) {
      data = d;
      left = right = null;
   }
} // end class TreeNode
```

3.4 How to represent a Binary Tree

we have defined TreeNode in terms of a general data type called NodeData. Any program that needs to use TreeNode must provide its own definition of NodeData.

For example, if the data in the node is an integer, NodeData could be defined as follows:

```
public class NodeData {
    int value;
    public NodeData(int v) {
        value = v;
    }
} //end class NodeData
```

3.4 How to represent a Binary Tree

In addition to the nodes of the tree, we will need to know the root of the tree. Keep in mind that once we know the root, we have access to all the nodes in the tree via the left and right references.

We will develop a BinaryTree class to work with binary trees. The class will start as follows:

```
public class BinaryTree {
   private TreeNode root;
   public BinaryTree() {
      root = null;
   }

   // + other methods

} //end class BinaryTree
```

3.5 Binary Search Trees (BST)

3.9 Level-order traversal

3.7 Non-recursive traversal (non-recursive in-order traversal only**)**

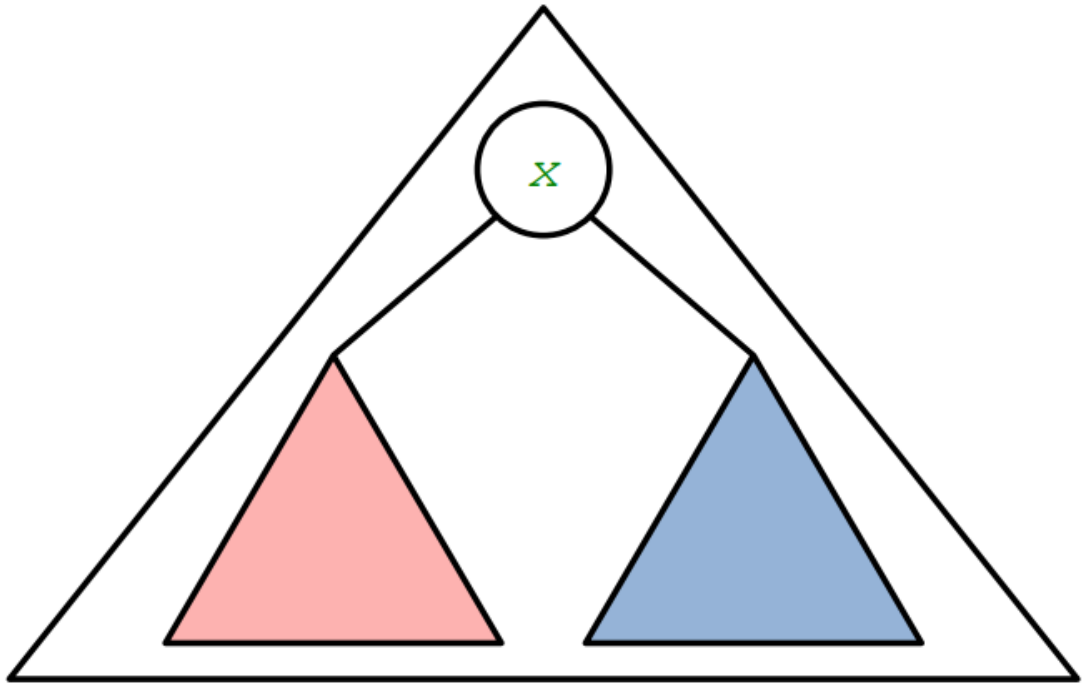
3.5 Binary Search Trees (BST)

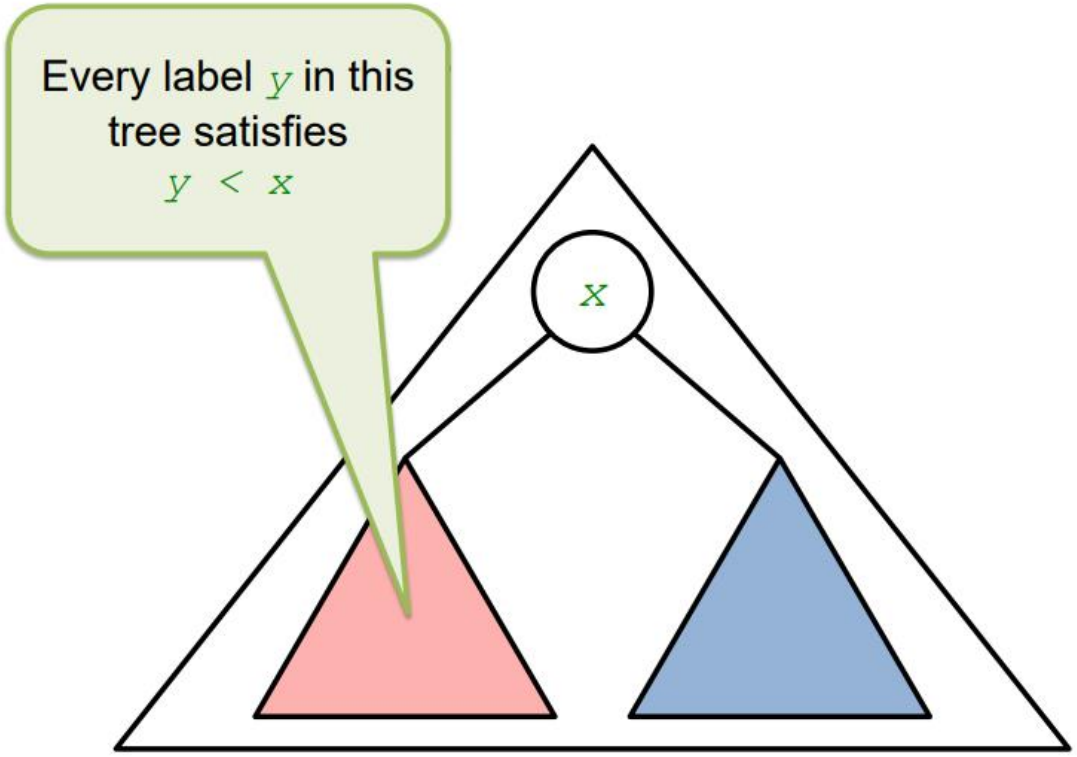
A **Binary Search Tree** (BST) is a binary tree which has the following properties:

- The left sub-**tree** of a node has a key less than to its parent node's key.
- The right sub-**tree** of a node has a key greater than to its parent node's key.

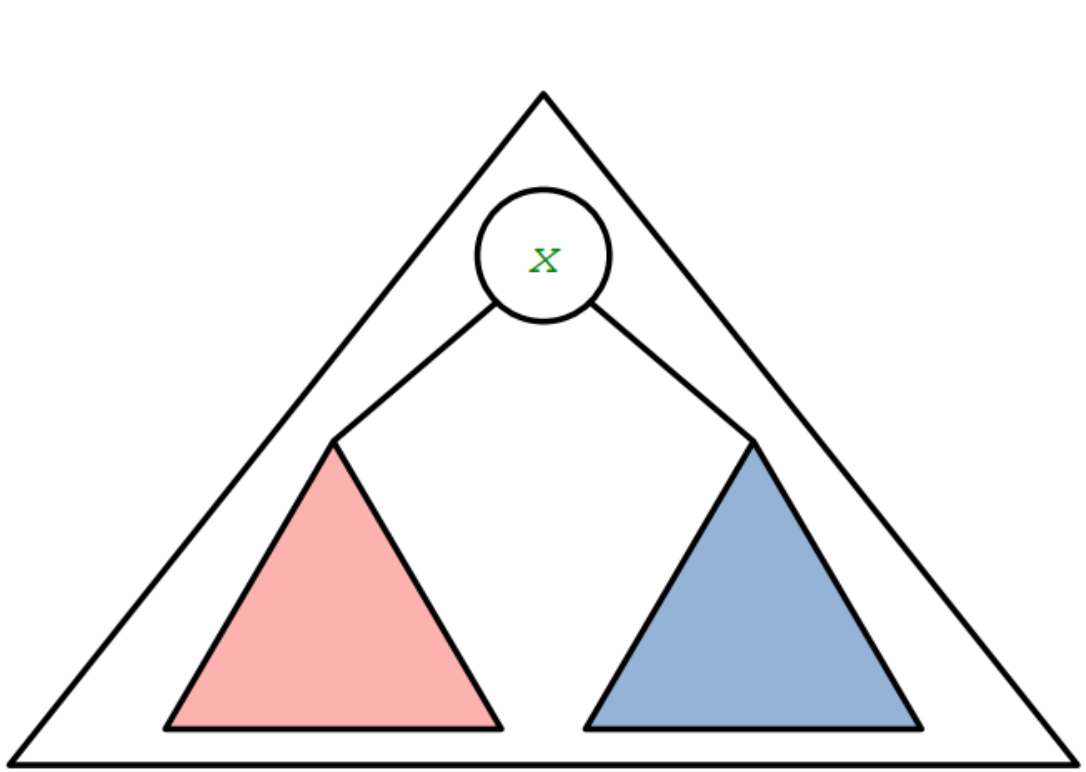
In other words:

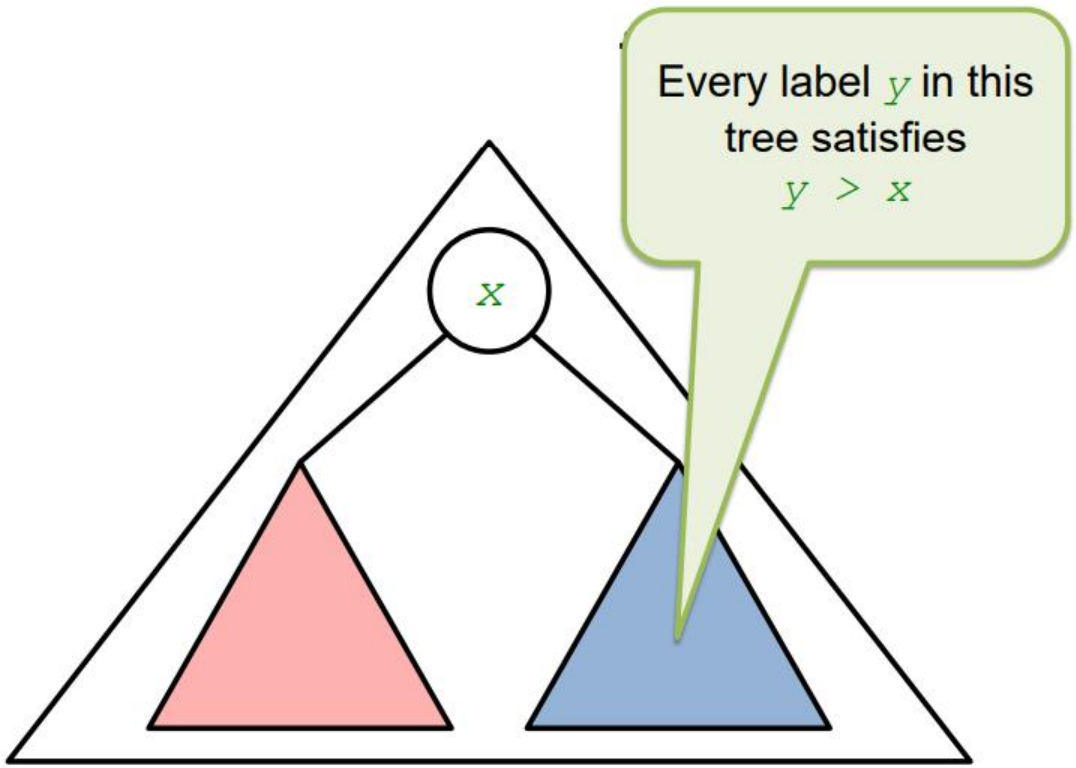
- For every node x in the tree, if y is in that node’s left subtree, then y < x.
- For every node x in the tree, if y is in that node’s right subtree, then y > x.



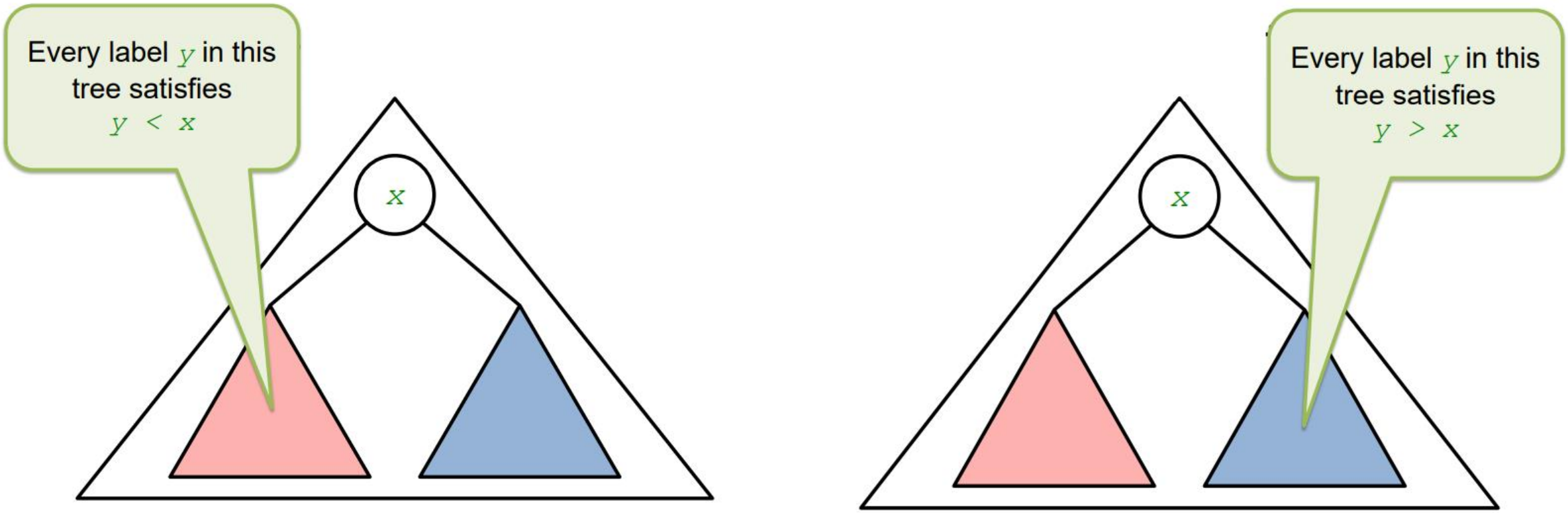


1. For every node x in the tree, if y is in that node's left subtree, then y < x





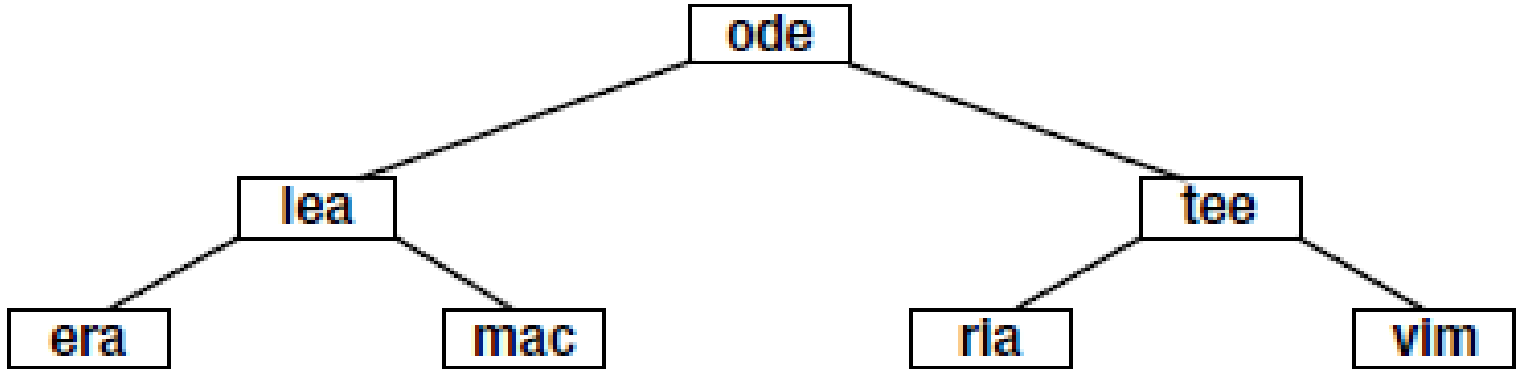
2. For every node x in the tree, if y is in that node's right subtree, then y > x



1. For every node x in the tree, if y is in that node's left subtree, then y < x

2. For every node x in the tree, if y is in that node's right subtree, then y > x

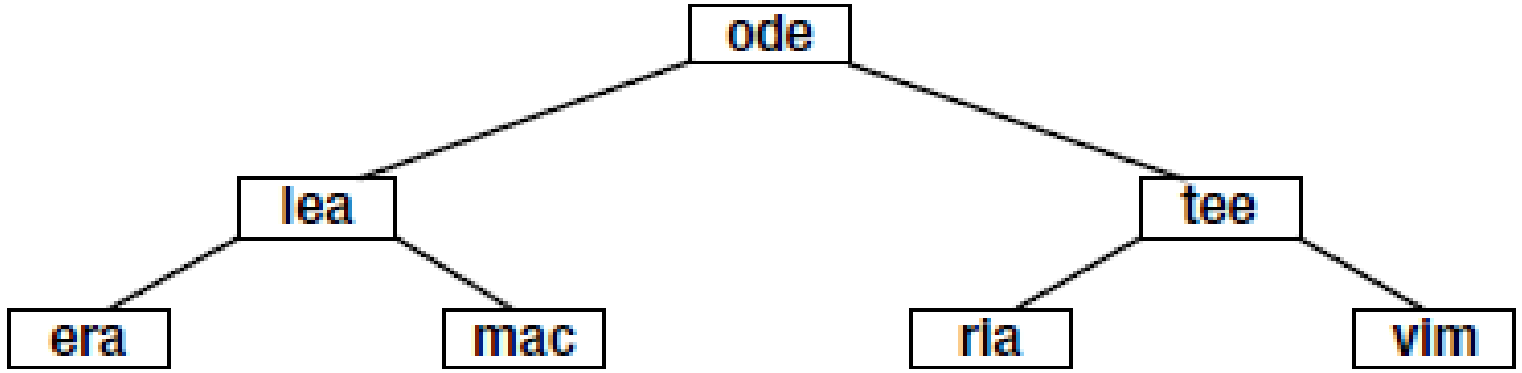
BST Example



- Given any node, a word in the left subtree is “smaller,” and a word in the right subtree is “greater” than the word at the node. (Here, smaller and greater refer to **alphabetical** order based on the first letter.)

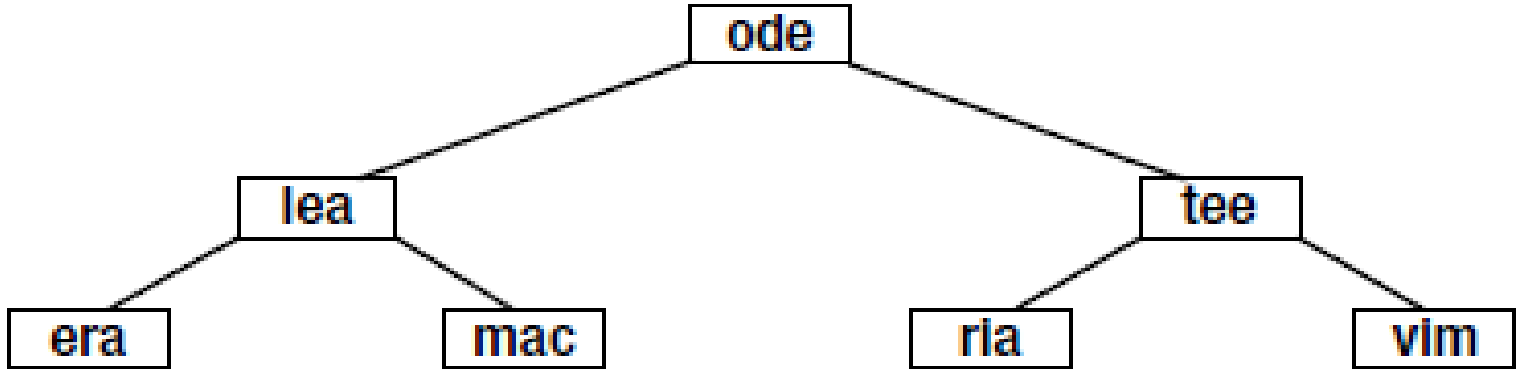
- It facilitates the search for a given key using a method of searching similar to the binary search of an array.

Searching in BST



- Consider the search for **ria**. Starting at the root, ria is compared with **ode**. Since **ria** is greater (in alphabetical order) than **ode**, we can conclude that if it is in the tree, it must be in the right subtree. It must be so since all the nodes in the left subtree are smaller than **ode**.
- Following the right subtree of **ode**, we next compare **ria** with **tee**. Since **ria** is smaller than **tee**, we follow the left subtree of **tee**.
- We then compare **ria** with **ria**, and the search ends successfully.

Searching for Non Existing Node



- But what if we were searching for **fun**?

1. **fun** is smaller than **ode**, so we go left.
2. **fun** is smaller than **lea**, so we go left again.
3. **fun** is greater than **era**, so we must go right.

- But since the right subtree of **era** is empty, we can conclude that **fun** is not in the tree.

Code for the search operation

```
public boolean search(NodeData item) {
        TreeNode current = root;
        if (isEmpty())
            System.out.print("The search operation failed. The BST is empty.");
        else
            while(current != null) {
                if (item.compareTo(current.data) == 0)
                    return true;
                else if (item.compareTo(current.data) < 0)
                    current = current.left;
                else
                    current = current.right;
            }
        return false;
} // end search
```

Other Possible Arrangements

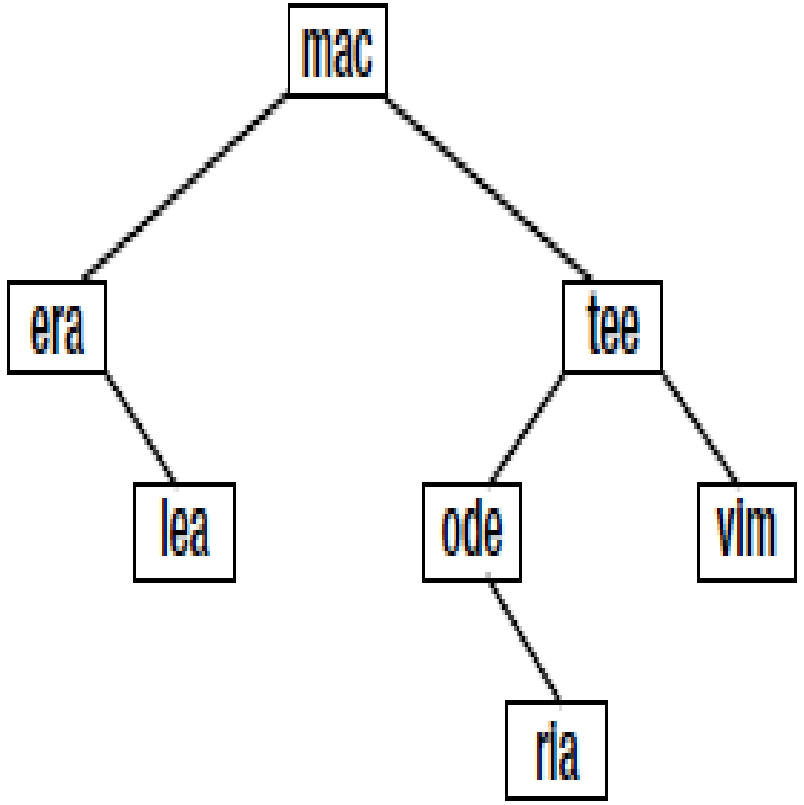
- The arrangement of the words in the previous example of the BST is not the only possible arrangement for these words.
- Suppose the words came in one at a time, and as each word came in, it was added to the tree in such a way that the tree remained a binary search tree.
- The final tree built will depend on the order in which the words came in.

Other Possible Arrangements Example

- For example, suppose the words came in this order:

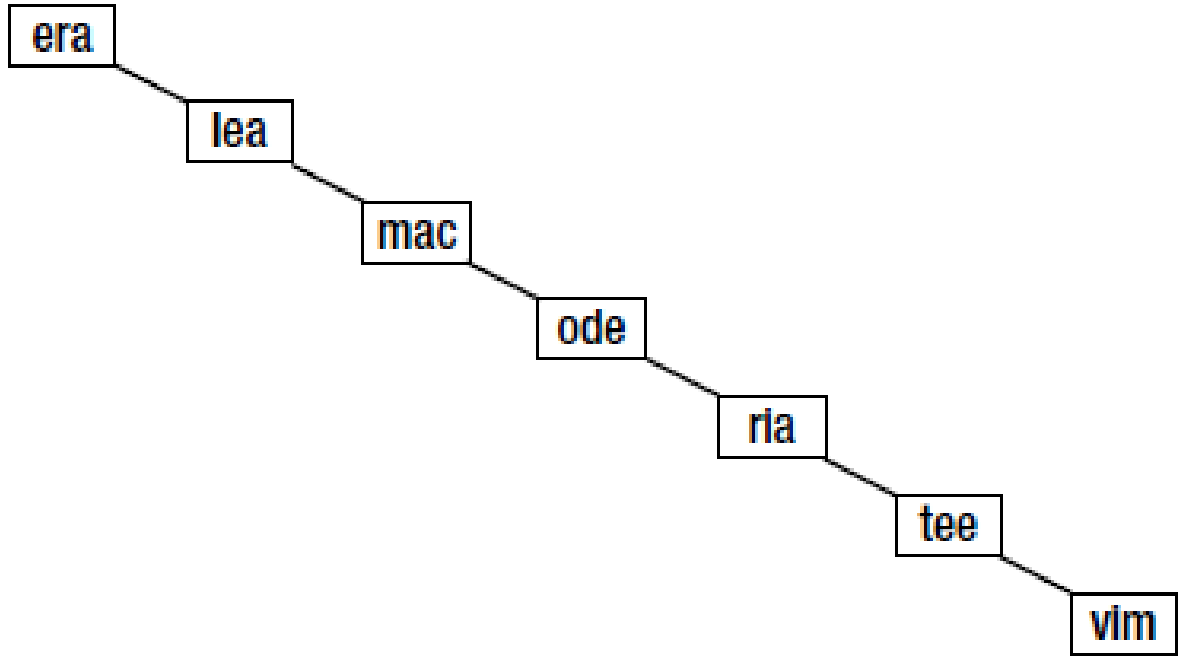
mac tee ode era ria lea vim

- Initially the tree is empty. When **mac** comes in, it becomes the root of the tree.
- **tee** comes next and is compared with **mac**. Since **tee** is greater, it is inserted as the right subtree of **mac**.
- **ode** comes next and is greater than **mac**, so we go right; **ode** is smaller than **tee**, so it inserted as the left subtree of **tee**.
- **era** is next and is smaller than **mac**, so it is inserted as the left subtree of **mac**.
- And so on …



The worst case (*Degenerate* Tree)

- If the words come in alphabetical order, then the tree built will be like this.
- Searching such a tree is reduced to a sequential search of a linked list.
- This kind of tree is called a *degenerate* tree.

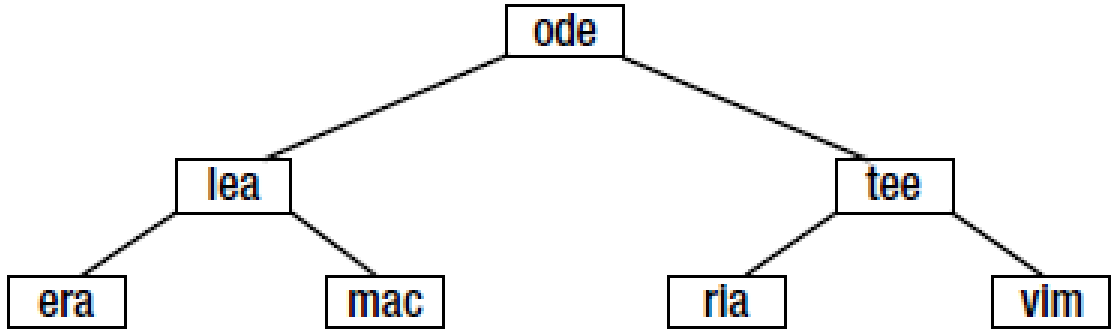


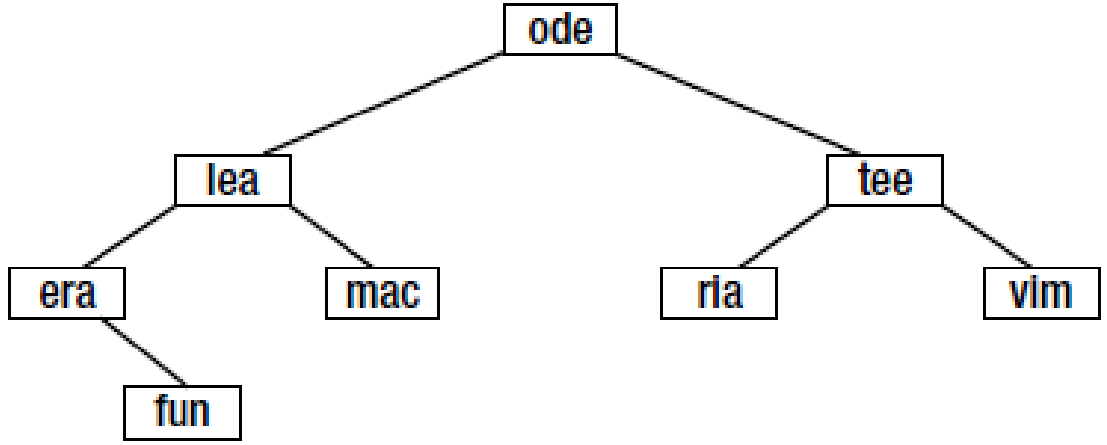
Inserting an element into a BST

- In the previous slides we covered the method `search` used to search for an item in a binary search tree (BST).

- In the next slides we look at the method `findOrInsert` used to insert an item into a binary search tree (BST) if it is not already found there.

Adding the Non Existing word “fun”



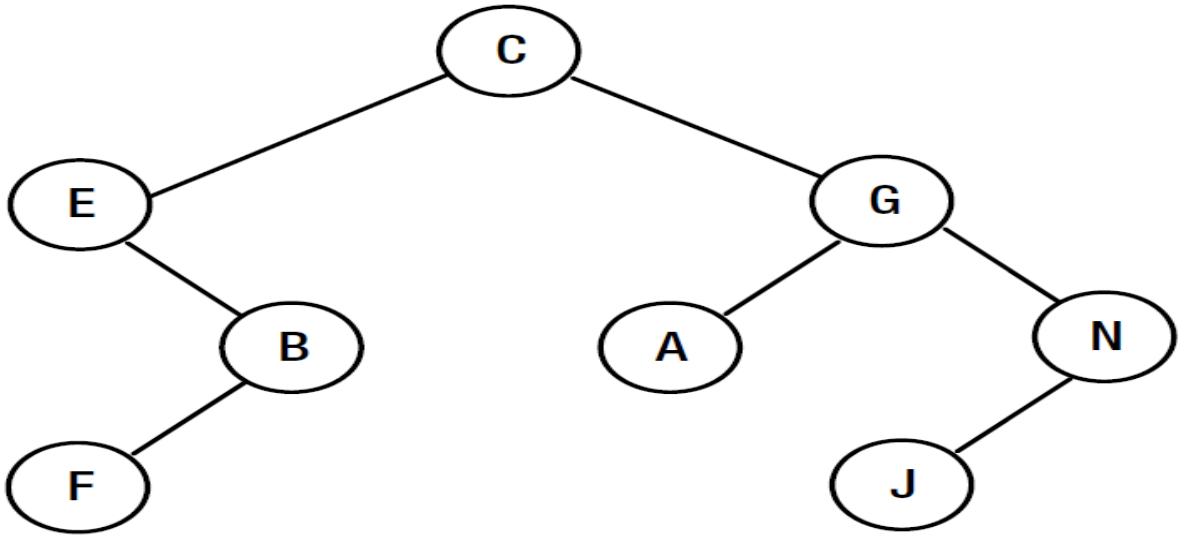


- When explaining the search algorithm, we discovered that if **fun** is to exist in the BST it must be on the right of **era**.
- Since it is not there, if we need to add it, then it must be added to the right of **era**.
- Not only does the binary search tree facilitates searching, but if an item is not found, it can be easily inserted.
- We will see the code of the insert operation in the next slide.

```
public void findOrInsert(NodeData item) {
        TreeNode current = root, parent = null, newNode;
        boolean found = false;
        if (isEmpty()) // empty BST
            root = new TreeNode(item); // insert item at the root
        else {
            while(current!= null) {
                if (item.compareTo(current.data) == 0) { // item is found
                    System.out.println("Insert Operation failed. The element is already in the BST.");
                    found = true;
                    break;
                }
                else if (item.compareTo(current.data) < 0) {
                    parent = current;
                    current = current.left; // go left
                }
                else { // item > current.data)
                    parent = current;
                    current = current.right;  // go right
                }
            } // end while
            if (!found) { // current is null in this case
                newNode = new TreeNode(item);
                if (item.compareTo(parent.data) < 0)
                    parent.left = newNode; // insert to the left of parent
                else
                    parent.right= newNode; // insert to the right of parent
            }// end if
        }// end else
}// end findOrInsert
```

3.9 Level-Order Traversal

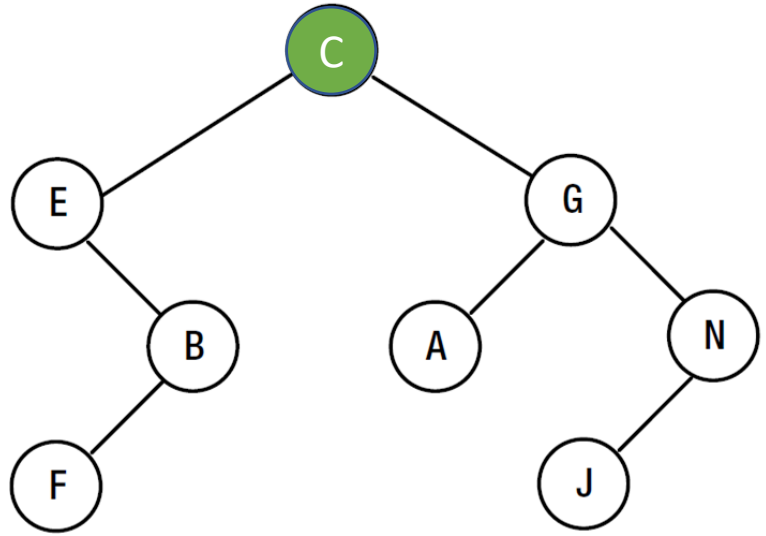
- An alternative to pre-order, in-order, and post-order traversals is *level-order*.
- We traverse the tree level by level (from left to right in each level).
- To perform a level-order traversal, we use a **queue** of TreeNode objects.



Its level-order traversal is C E G B A N F J.

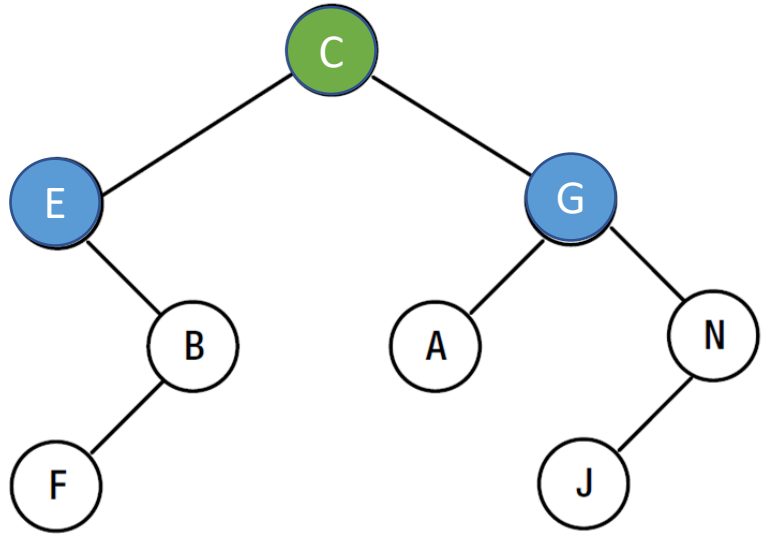
Level-Order Traversal using a queue

- Enqueue C in Q



Level-Order Traversal using a queue

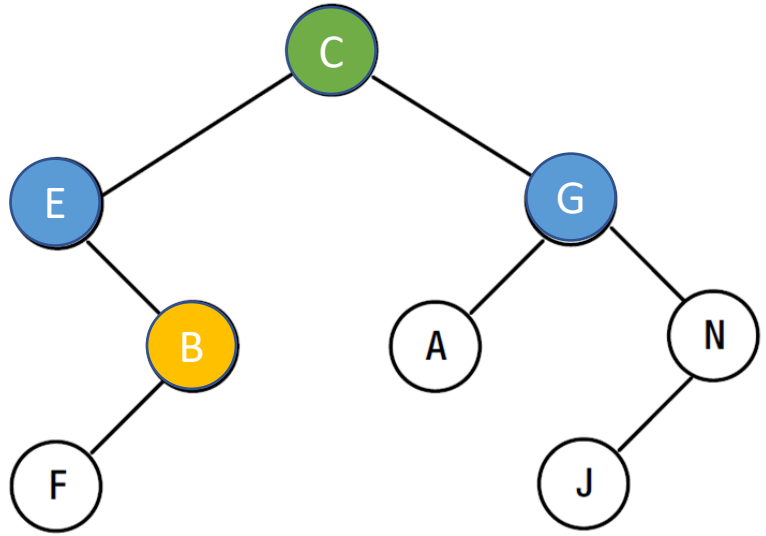
- Q is not empty, so dequeue and visit (output) C, then add (enqueue) its children E and G to Q.



Output: C

Level-Order Traversal using a queue

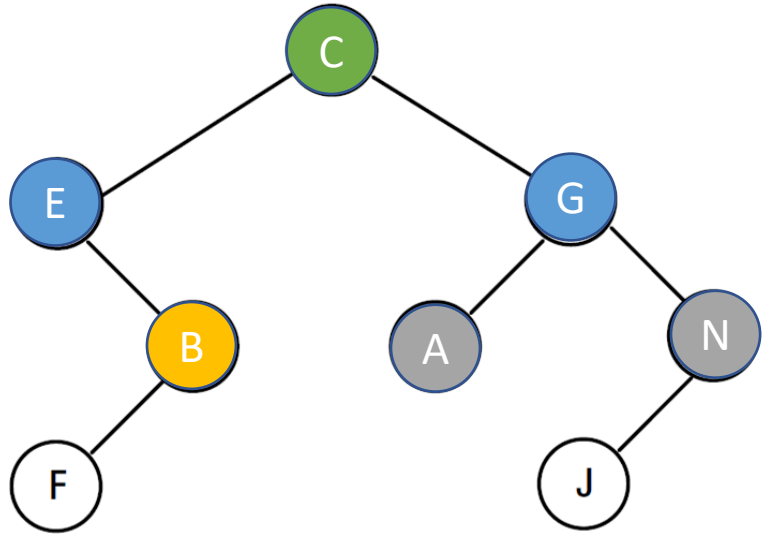
- Q is not empty, so dequeue and visit (output) E, then add its child B to Q.



Output: C E

Level-Order Traversal using a queue

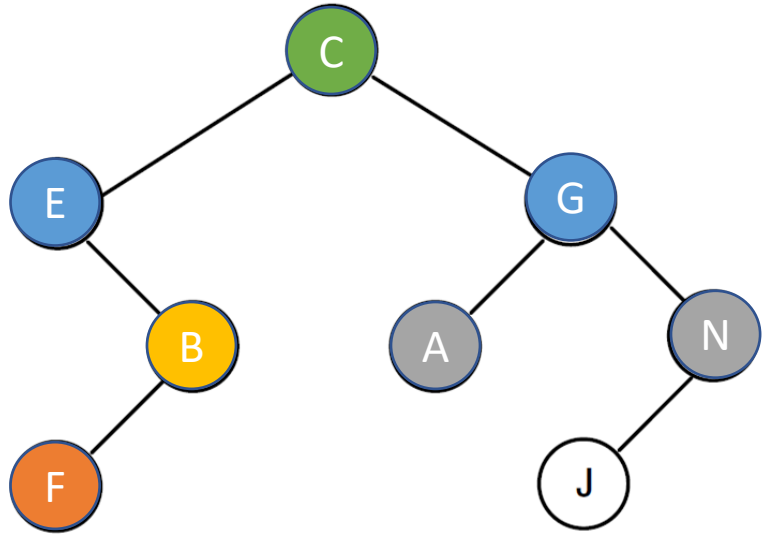
- Q is not empty, so dequeue and visit (output) G, then add its children A and N to Q.



Output: C E G

Level-Order Traversal using a queue

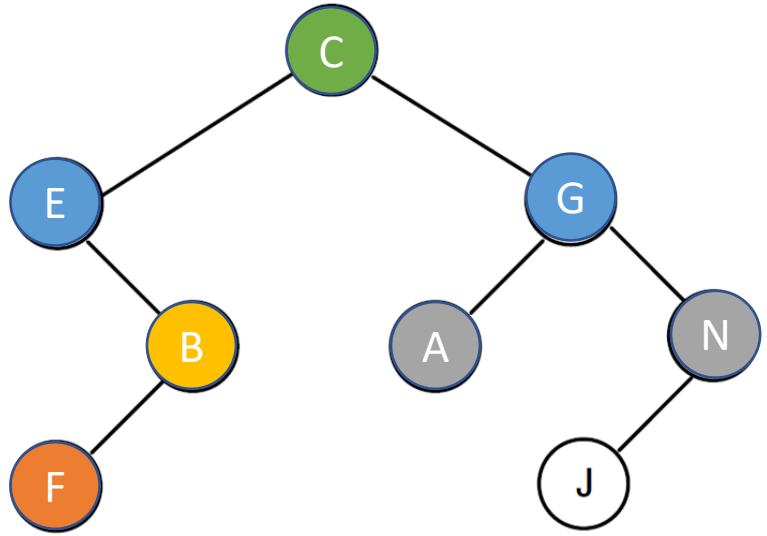
- Q is not empty, so dequeue and visit (output) B, then add its child F to Q.



Output: C E G B

Level-Order Traversal using a queue

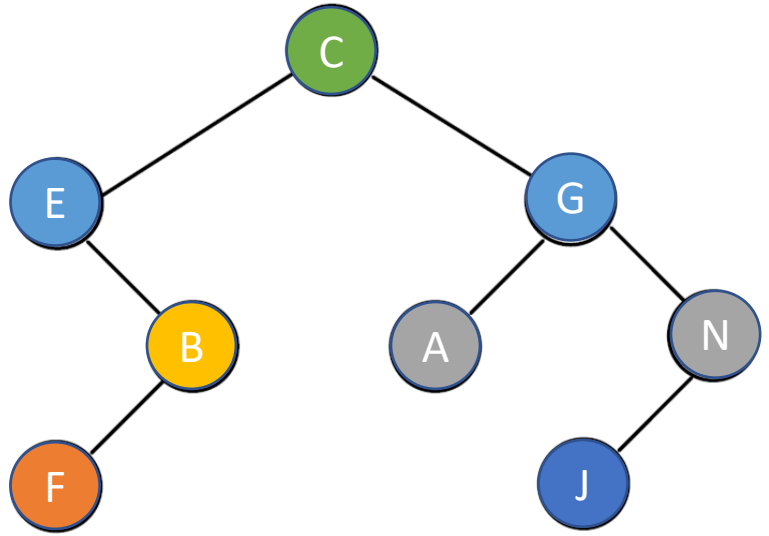
- Q is not empty, so dequeue and visit (output) A. Since A has no children then nothing is added to Q.



Output: C E G B A

Level-Order Traversal using a queue

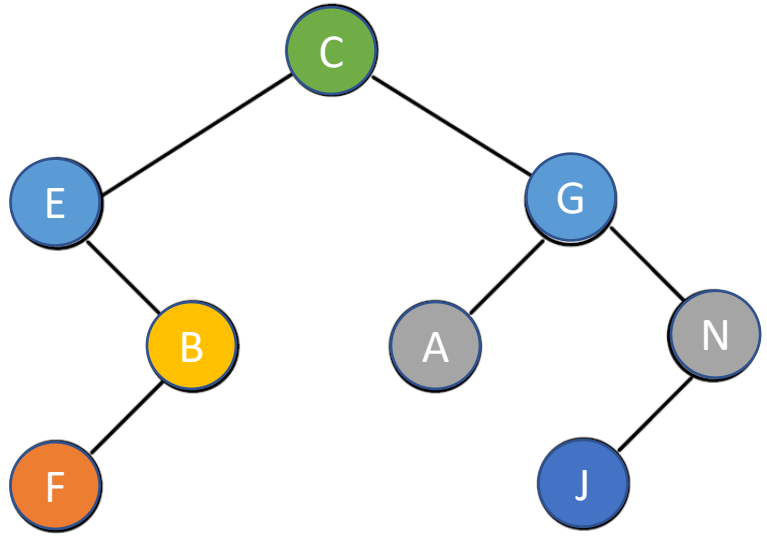
- Q is not empty, so dequeue and visit (output) N, then add its child J to Q.



Output: C E G B A N

Level-Order Traversal using a queue

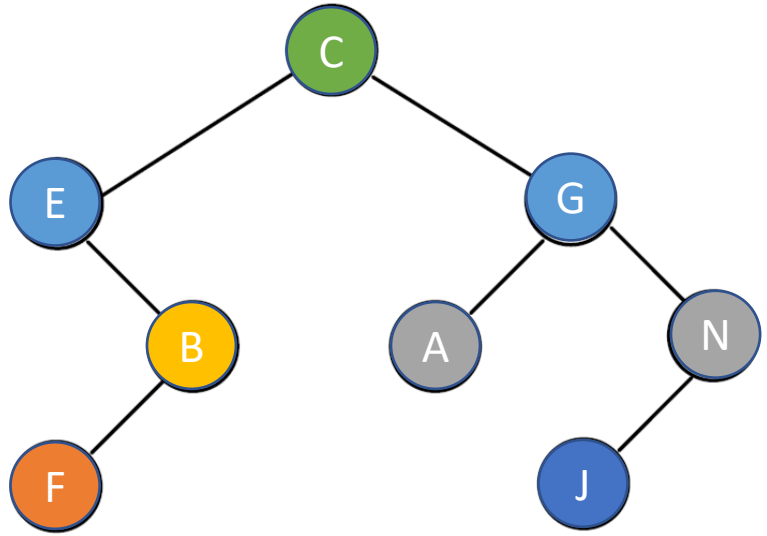
- Q is not empty, so dequeue and visit (output) F. Since F has no children then nothing is added to Q.



Output: C E G B A N F

Level-Order Traversal using a queue

- Q is not empty, so dequeue and visit (output) J. Since J has no children then nothing is added to Q.



- Now Q is empty. The traversal ends having visited the nodes in the order C E G B A N F J.

Output: C E G B A N F J

Level Order Traversal code

```
public void levelOrder() {
  Queue queue = new Queue();              // A queue of TreeNode objects
  TreeNode curr;
  if (root == null)                       // if the tree is empty then exit
      return;
  queue.enqueue(root);                    // enqueue the element at the root
  while (!queue.isEmpty()) {              // As long as the queue is not empty
    curr = queue.dequeue();               // dequeue the element at the head
    System.out.print(curr.data.value  + " "); //output the data
    if (curr.left != null)                // if there is a left child
       queue.enqueue(curr.left);          // enqueue the left child
    if (curr.right != null)               // if there is a right child
       queue.enqueue(curr.right);         // enqueue the right child
  }
} //end levelOrder
```

3.7 Non-recursive Traversals

- The pre-order, in-order, and post-order traversals can be done without the use of recursion.
- This can be done using a **stack** of TreeNode objects.
- We will discuss the non-recursive in-order traversal only.

Non Recursive InOrder Traversal Algorithm

1) Create an empty stack S.
2) Initialize current node as root
3) Push the current node to S and set current = current->left until current is NULL
4) If current is NULL and stack is not empty then
    a) Pop the top item from stack.
    b) Print the popped item, set current = popped_item->right
    c) Go to step 3.
5) If current is NULL and stack is empty then we are done.

Non Recursive InOrder Traversal code

```
public void nonRecursiveInorder() {
      Stack s = new Stack(); // A stack of TreeNode objects
      TreeNode curr = root;
      boolean stop = false;
      while (!stop) {
         while (curr != null) {
                  s.push(curr);
                  curr = curr.left;
                }
                if (s.empty())
                    stop = true;
                else
                {
                   curr =  s.pop();
                   System.out.println(curr.data.value + " ");
                   curr = curr.right;
                }
            }
      }
}
```

BST Animation Links

- Animation of the Binary Search Tree (BST) Search, Insert, Remove, Inorder, Preorder, and Postorder algorithms.

  http://liveexample.pearsoncmg.com/liang/animation/web/BST.html

- Animation of the four traversal algorithms of binary tree (preorder, inorder, postorder, and levelorder).

  https://towardsdatascience.com/4-types-of-tree-traversal-algorithms-d56328450846

- Animation in parallel with the code of the three traversal algorithms of binary tree (preorder, inorder, postorder).

  https://opendsa-server.cs.vt.edu/ODSA/Books/Everything/html/BinaryTreeTraversal.html

3.10 Some useful tree functions

3.11 BST deletion

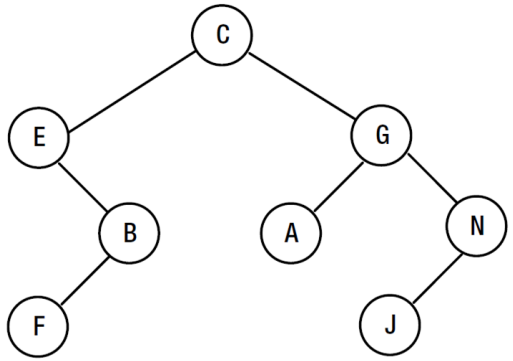
3.10 Some Useful Binary Tree Functions

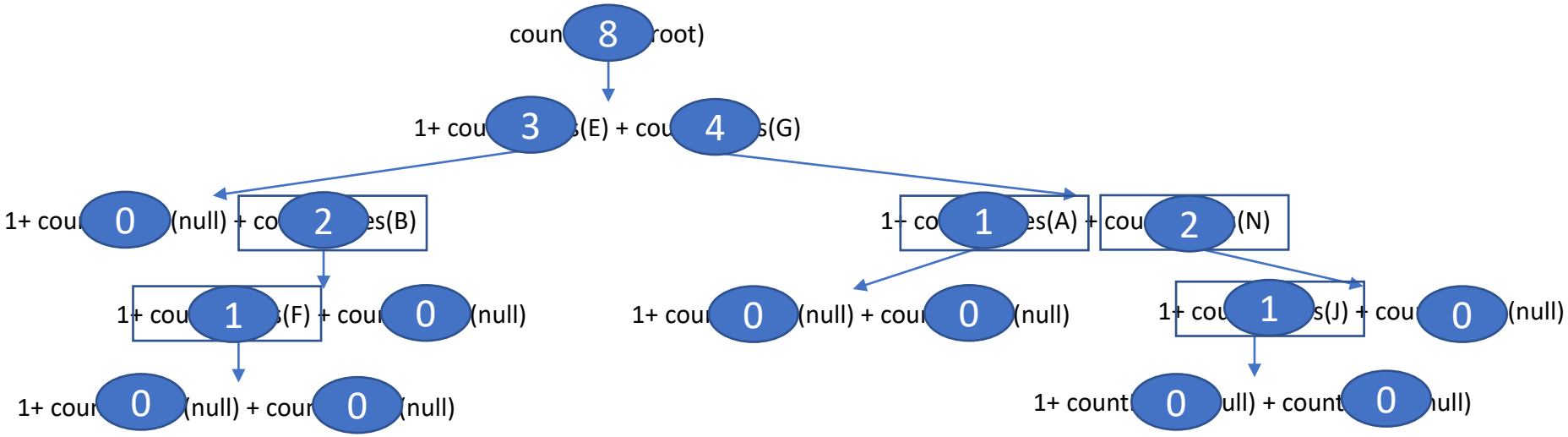
- Count the number of nodes in a tree:

```
public int numNodes() {
    return countNodes(root);
}
private int countNodes(TreeNode current) {
    if (current == null)
       return 0;
    return 1 + countNodes(current. left) + countNodes(current.right);
}
```

- Note: the public method is not the recursive one.

Count the number of nodes in a tree





Some Useful Binary Tree Functions

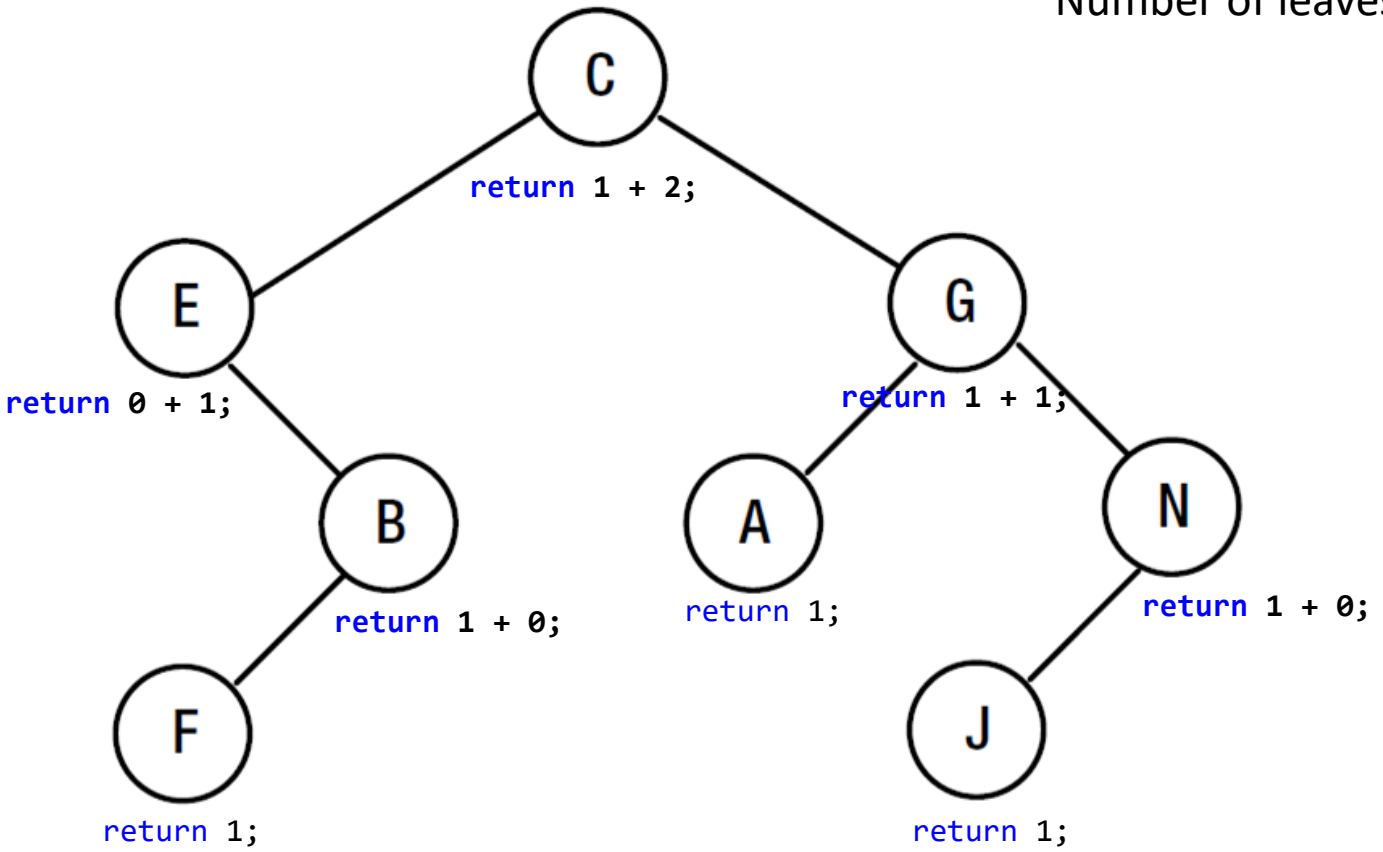
- Count the number of leaves in the tree. That is the number of nodes that have their left and right nodes both null.

```
public int numLeaves() {
    return countLeaves(root);
}
private int countLeaves(TreeNode current) {
    if (current== null)
        return 0;
    if (current.left == null && current.right == null)
        return 1;
    return countLeaves(current.left) + countLeaves(current.right);
}
```

- Note: Same recursion principle as the countNode method.

Count the number of leaves in a tree

Number of leaves = 3

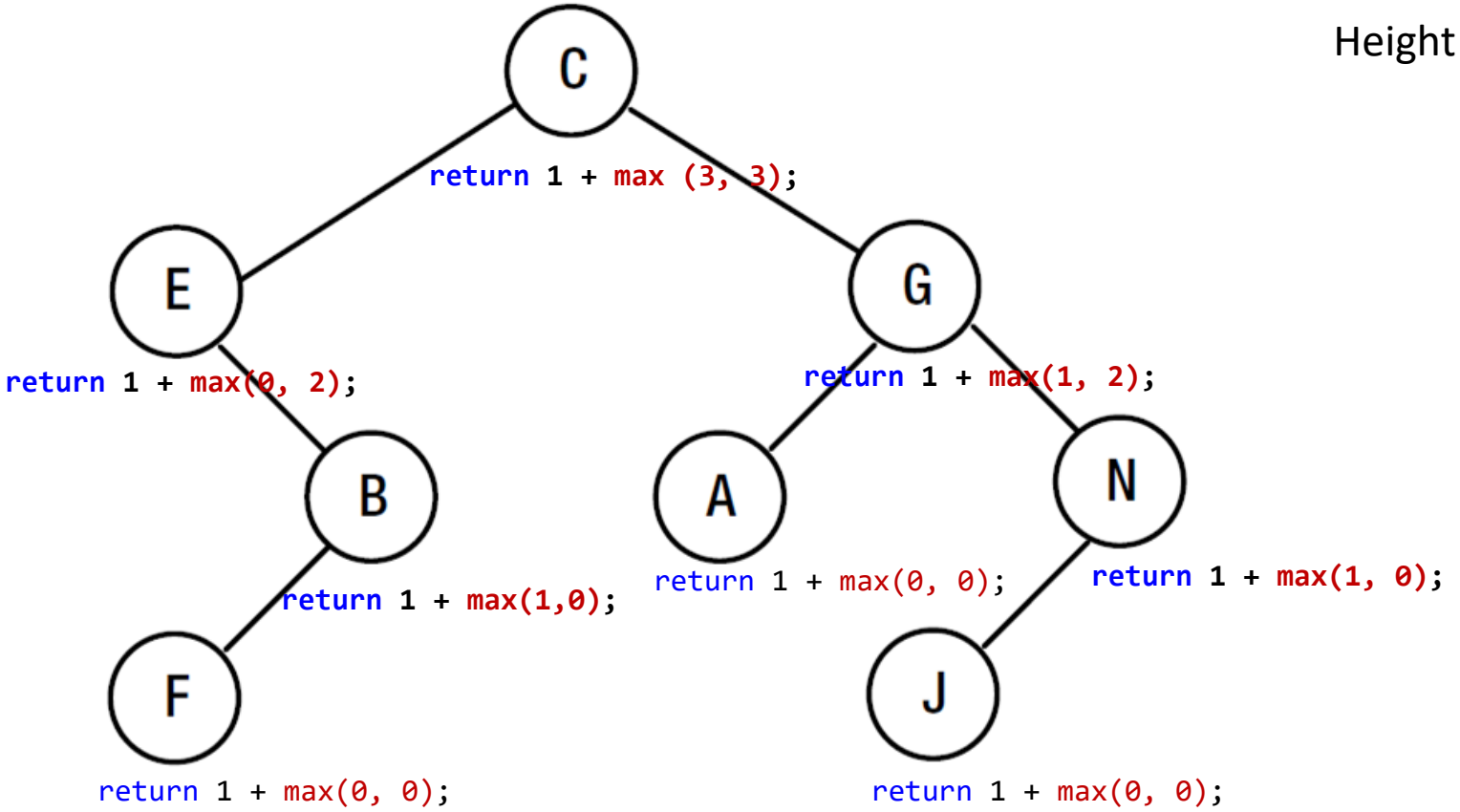


Some Useful Binary Tree Functions

- Calculate the height (number of levels) of the tree

```
public int height() {
    return countLevels(root);
}
private int countLevels(TreeNode current) {
    if (current == null)
        return 0;
    return 1 + Math.max(countLevels(current.left), countLevels(current.right));
}
```

Calculate the height of the tree



Height = 4

3.11 BST deletion

- The method `delete`(NodeData item) is used to delete an item from the binary search tree (BST) if it is found.

- This semester you will be only responsible for knowing how delete works. This will be explained in the next slides.

- You will not be responsible for the code of the method delete. However, the code will be provided in the complete implementation of the class BST posted with this lecture.

8.11 Binary Search Tree Deletion

Consider the problem of deleting a node from a binary search tree (BST) so that it remains a BST. There are three cases to consider:

1. The node is a leaf.
2. (a) The node has no left subtree.

   (b) The node has no right subtree.
3. The node has non-empty left and right subtrees.

We illustrate these cases using the BST shown in Figure 8-9.

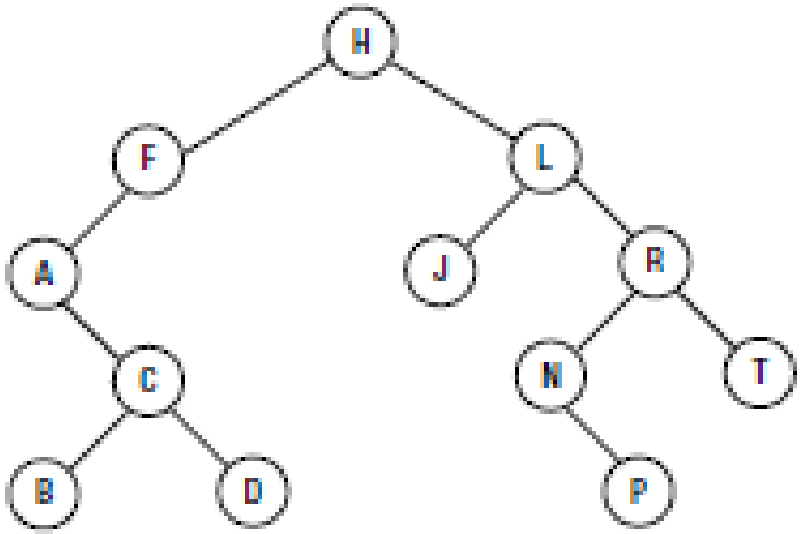
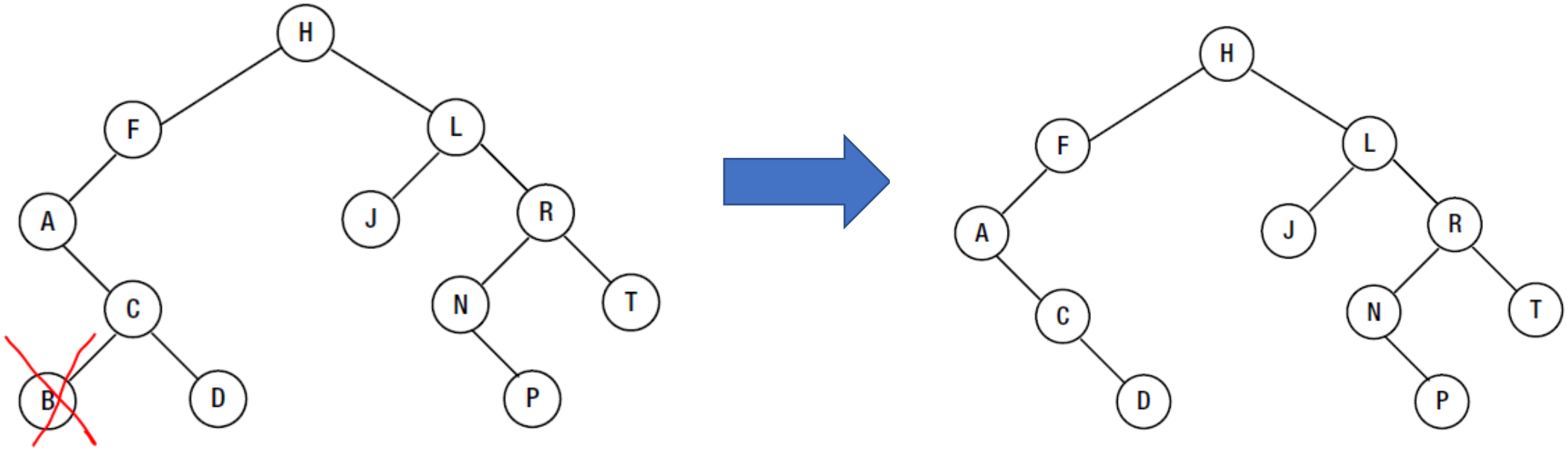


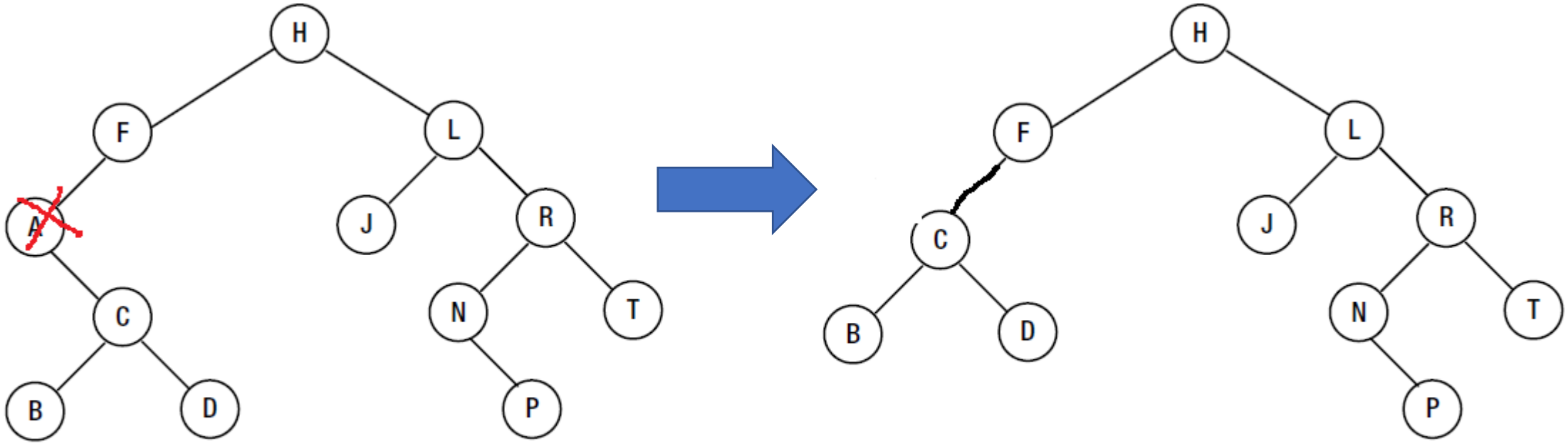
Figure 8-9. A binary search tree

Example 1: delete a node with 0 children (leaf node) such as Node B



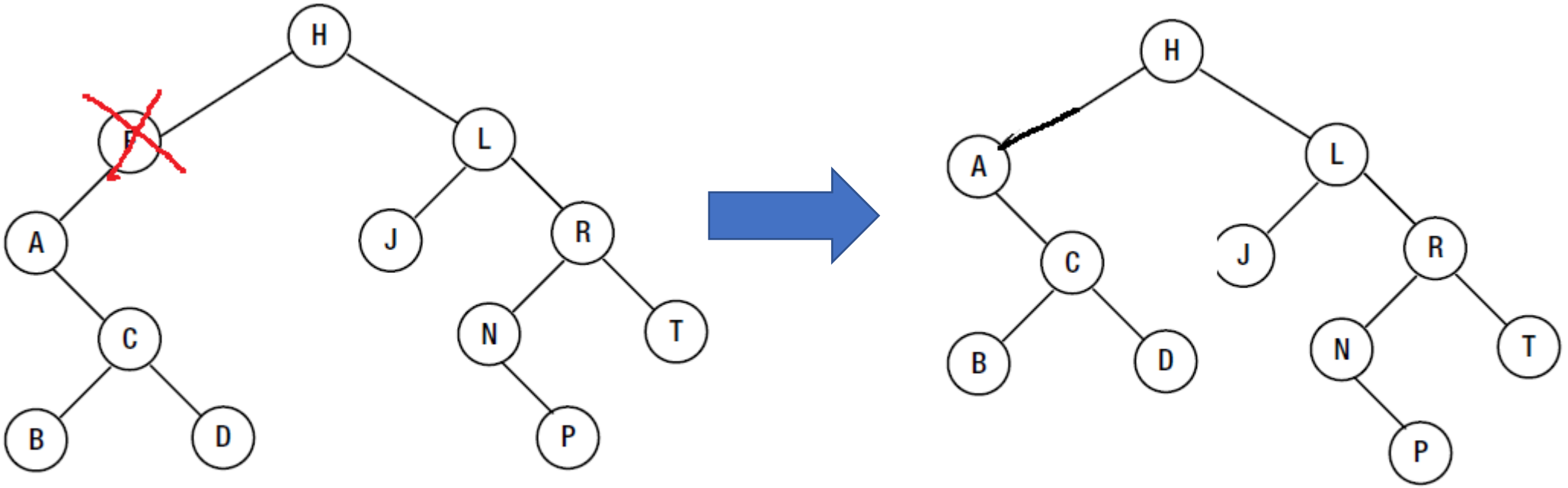
Same if we delete one of the other leaf nodes (nodes with no children): D , P or T

Example 2: delete a node with only one child (no left subtree) such as node A

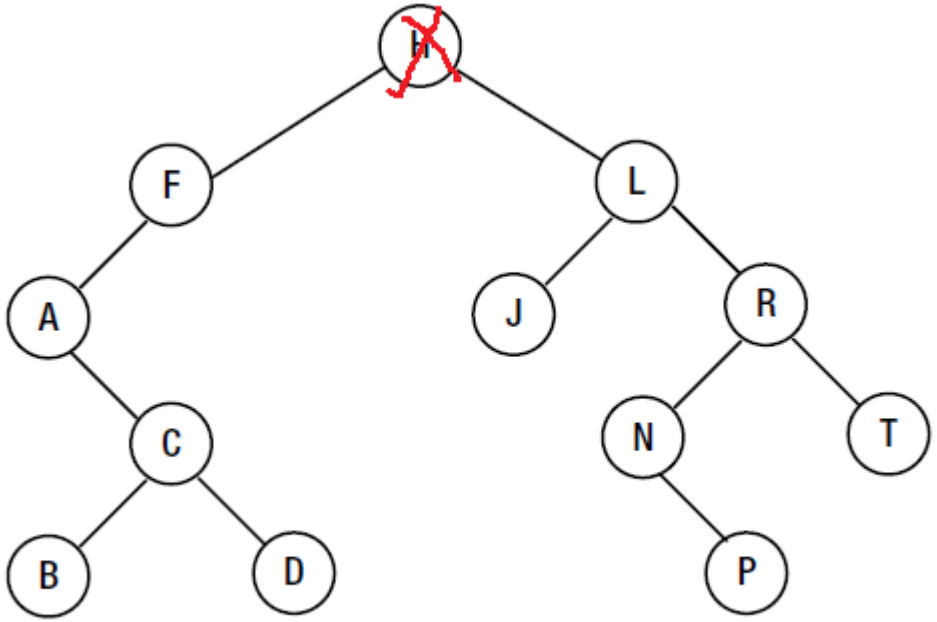


Same if we delete one of the nodes with only one child: F or N

Example 3: delete a node with only one child (no right subtree) such as node F



Example 4: Delete a node with two children (non-empty left and right subtrees) such as node H

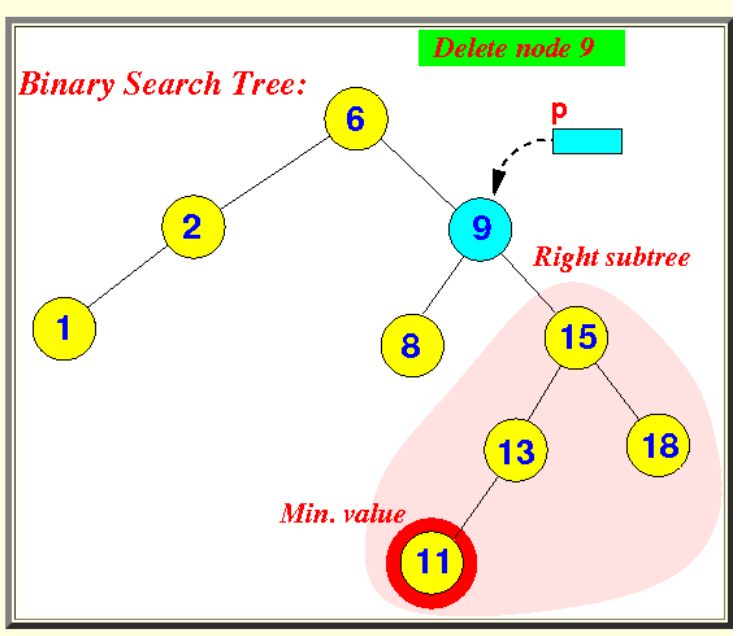


If we want to delete H, we will have 2 options:

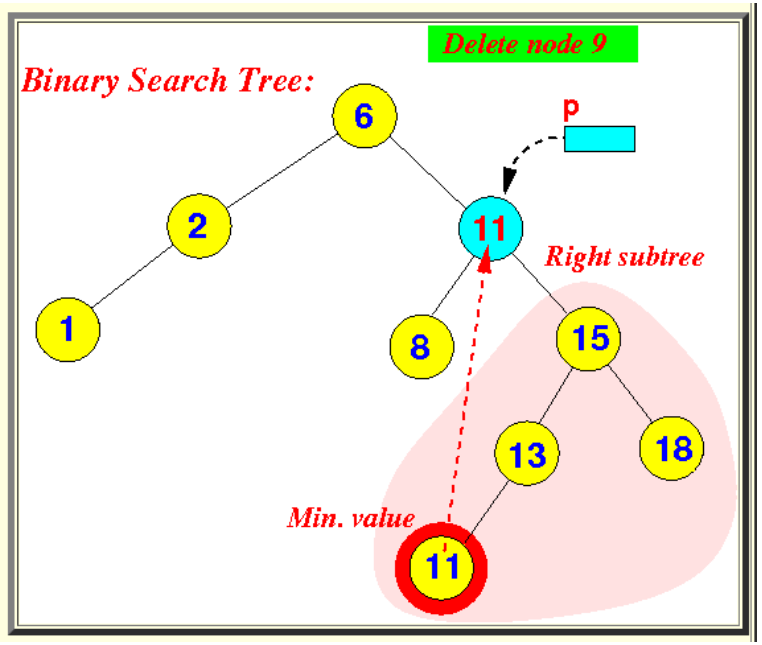
1. **replace H with J called in-order successor then delete the successor.**
2. **replace H with F called in-order predecessor then delete the predecessor**

We will adopt option1 which is replacing the deleted node with its successor (i.e. replace H with the first element greater then H which is J), then we delete the node that contained the successor (J) initially which is a node with either one child or no children.

Example 5: Delete the node that contains 9 => replace it with its successor which is 11

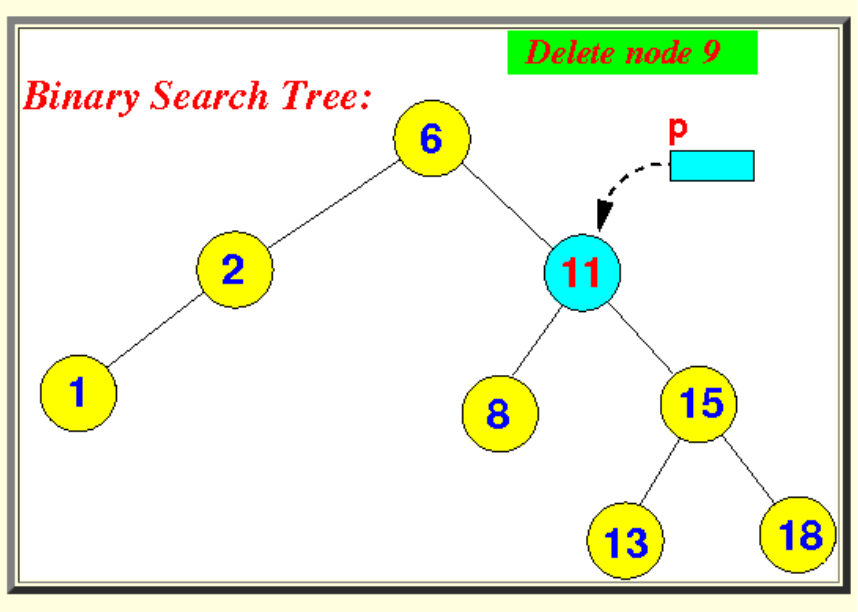


Step 1: Find the successor of 9 which is 11 by going right once and then left as much as possible.

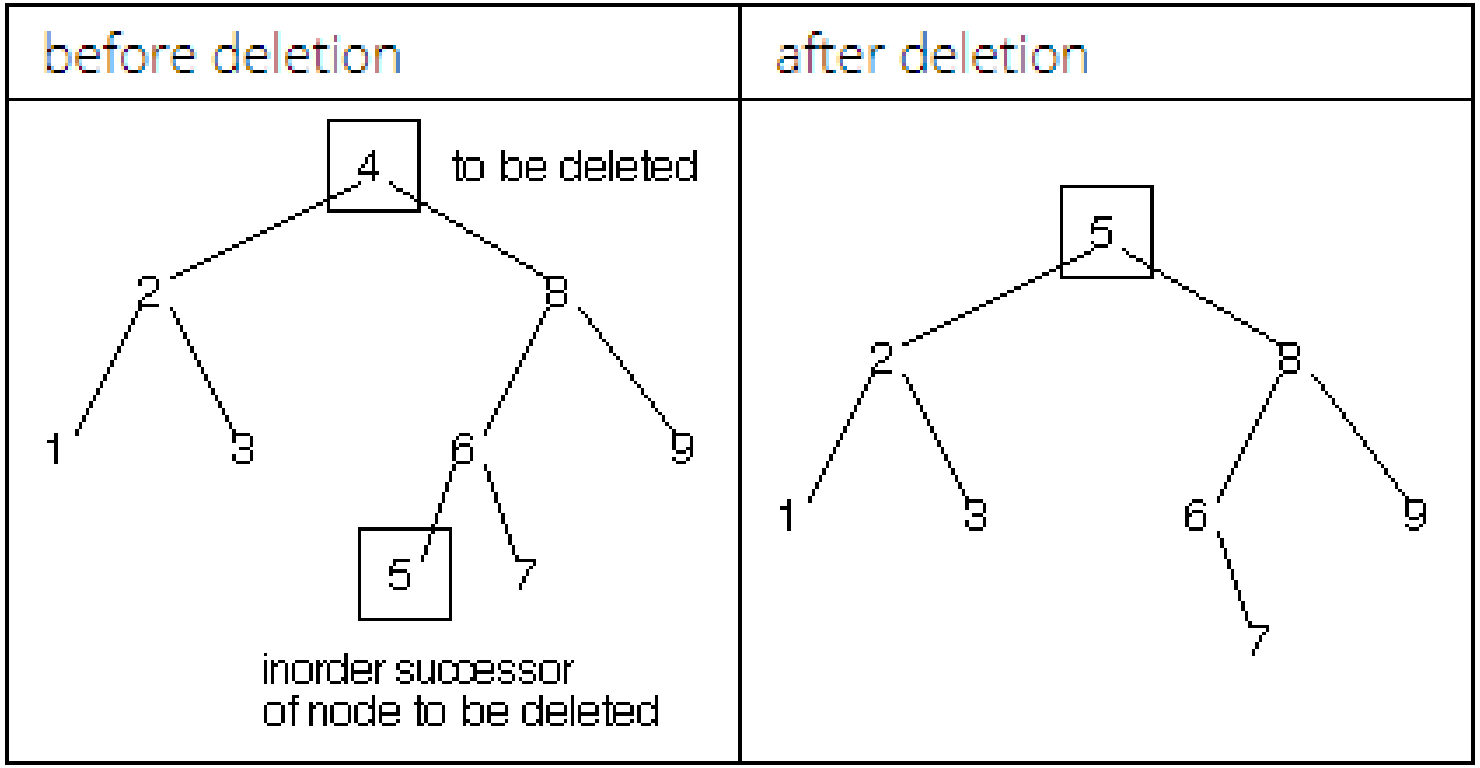


Step 2: Replace the value in the node to be deleted with value in the successor node. The node that contained 9 now contains 11.

Step 3: Delete the successor node so that we do not have the value 11 twice in the BST.



Example 6: Delete the node that contains 4 => replace it with its successor which is 5



Code for method delete

```
public void delete(NodeData item) {
      TreeNode parent = null, child, temp, current = root;
      boolean found = false;

      if (root == null) { // empty BST
            System.out.println("Delete operation failed. The BST is empty.");
            return;
      }
      // loop to check if the element is already in the tree
      while (current != null && !found) {
            if (item.compareTo(current.data) == 0) // item equals node's data
                found= true;
            else if (item.compareTo(current.data) < 0) { // item < node's data
            {
                parent = current;
                current = current.left; // go left
            }
```

Code for method delete

```
    else // item > than node's data
    {
      parent = current;
      current = current.right; // go right
    }
} // end while

if (!found)
     System.out.println("Delete failed. Element is not in the BST.");
else // found
{
   if (current.left != null && current.right != null) { // two children
      temp = current; // keep a pointer to the node containing the item
      // find the successor of item
      parent = current;
      current = current.right; // go right once
```

Code for method delete

```
    while (current.left != null) { // go left all the way
         parent = current;
          current = current.left;
    } // end while
    // current now points to the node containing the successor of item
    temp.data = current.data; // exchange the data of temp and that of current
  } // end if two children - now current has one child or no children
 // case of a node with one child (left or right) or no children
    child = current.left; // get the left child of the node to be deleted
    if (child == null) // if it has no left child
      child = current.right; // get its right child
    if (current == root) // deleting the root
        root = child;
    else if (parent.left == current) // if current is a left child
        parent.left = child; // connect parent of node to the child of node
    else
      parent.right = child;
  } // end else found
} // end delete
```

BST Animation Links

- Animation of the Binary Search Tree (BST) Remove (Delete) Algorithm.
  http://liveexample.pearsoncmg.com/liang/animation/web/BST.html