Binary Trees

Reading:
Lewis & Chase 12.1, 12.3
Eck 9.4.1

Objectives

- Tree basics
- Learn the terminology used when talking about trees
- Discuss methods for traversing trees
- Discuss a possible implementation of trees with nodes
- Examine a binary tree example

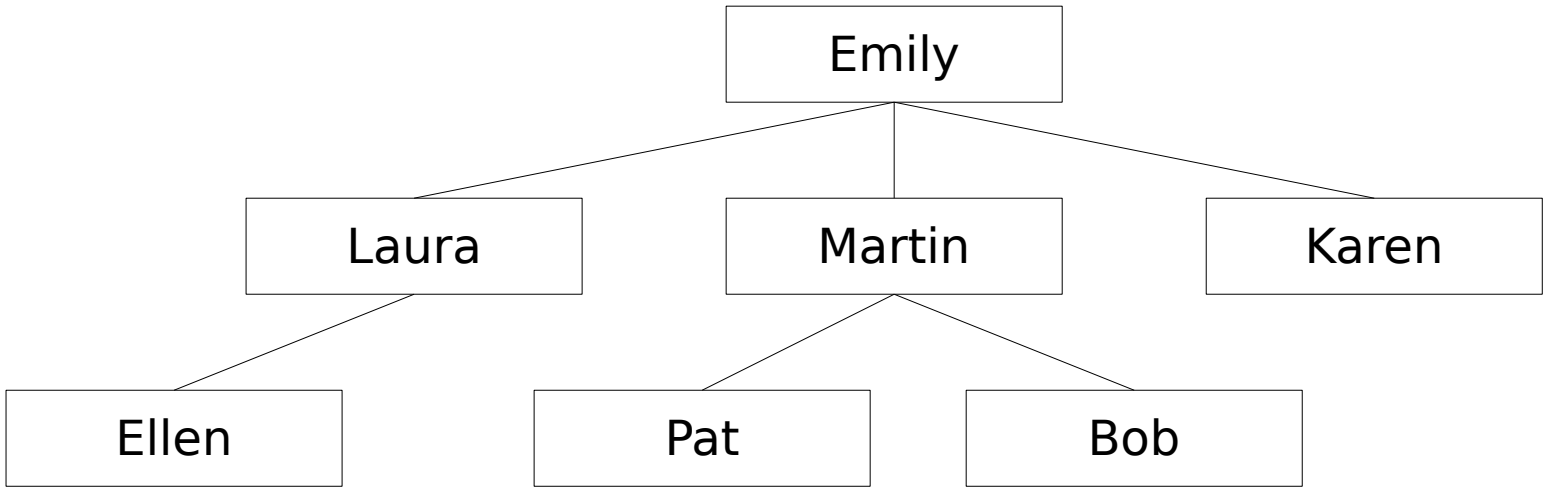
Tree Basics

- So far, all the data structures that we have encountered were linear. Objects in an array, list, stack, or queue are placed one after the other in a line.
- Sometimes it is useful to organize data into groups and subgroups.
- This type of organization of data is hierarchical, or non-linear, since the data appears at various levels.

前面学的dsa都是linear,
有时需要data是(sub)groups,
这种是hierarchical/非线性结构, multilevel data

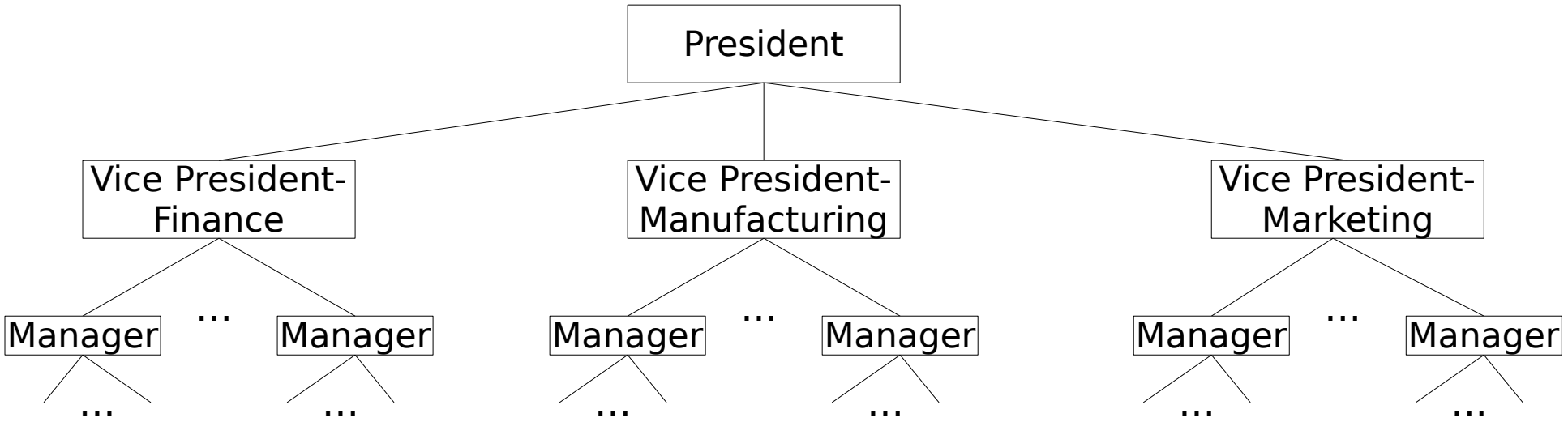
Hierarchical Organizations
Family Trees

- Family trees can be arranged in various ways.
- This diagram shows Emily's children and grandchildren.



Hierarchical Organizations Businesses

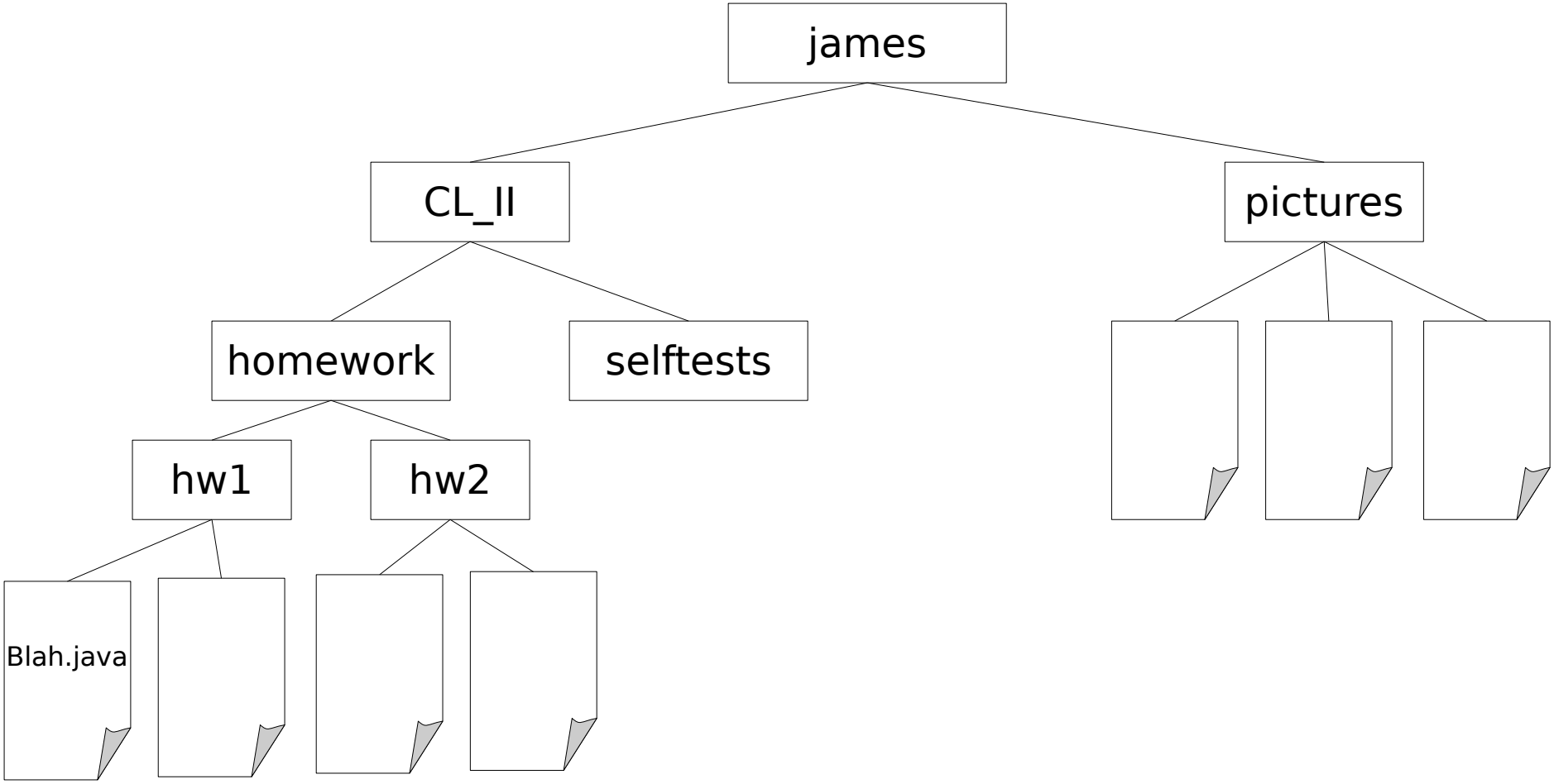
- A hierarchical diagram of a business:



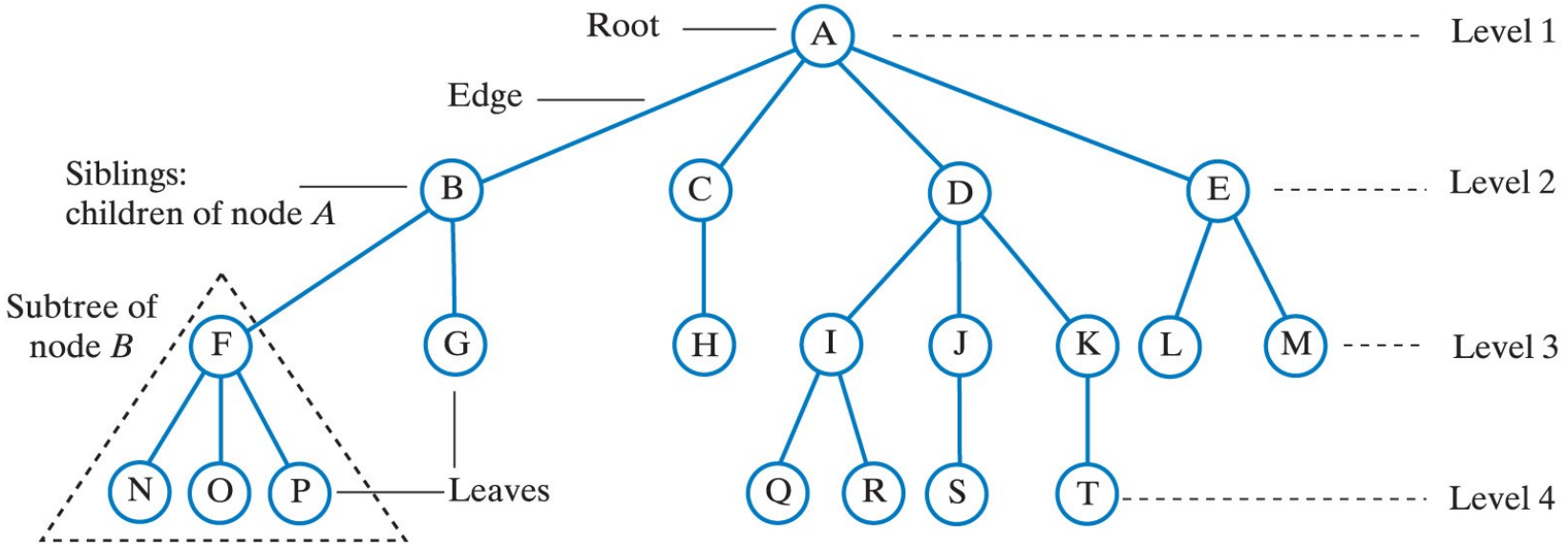
生活中的例子

Hierarchical Organizations Files and Directories

- Files and directories on a computer:

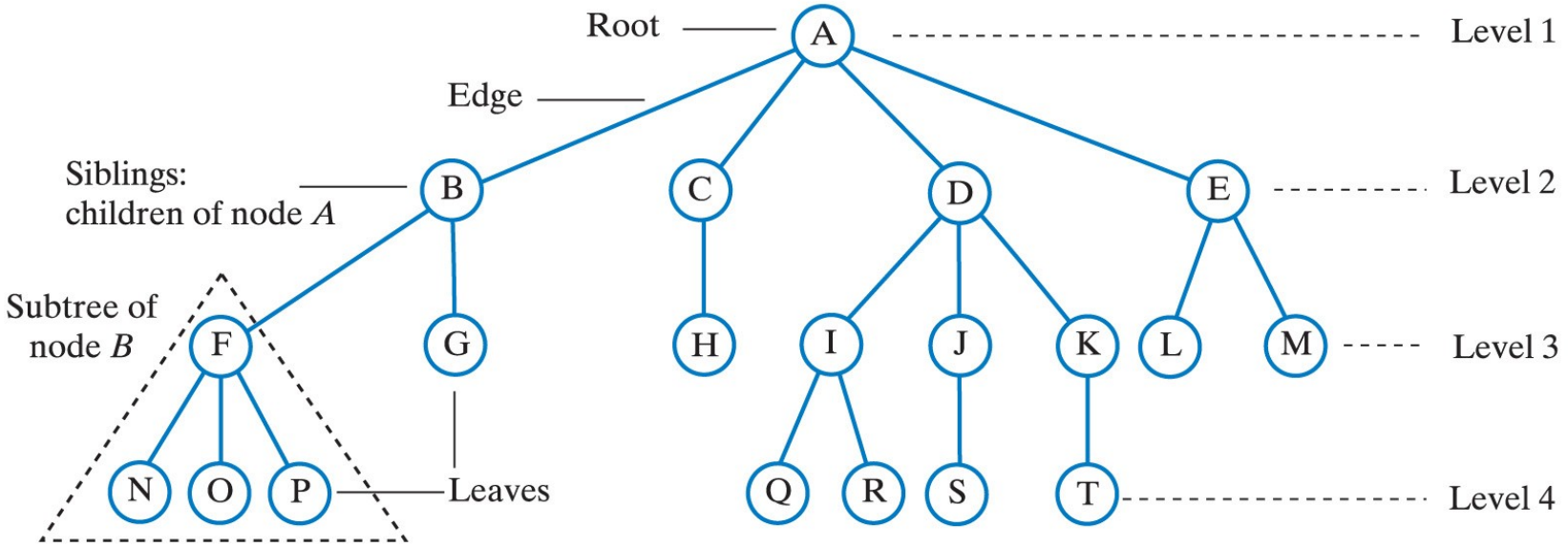


Terminology



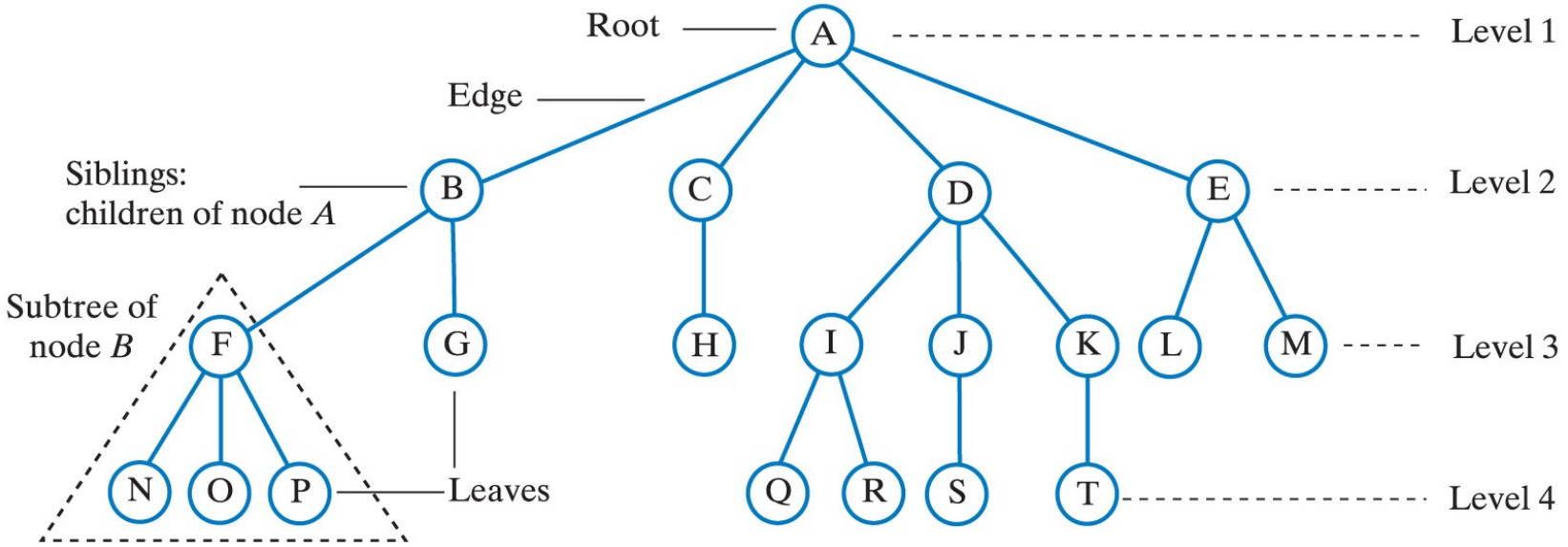
- A **tree** is a set of **nodes** connected by **edges** that show a relationship between the nodes.
- The nodes are arranged in levels that indicate the hierarchy of the nodes. The top level is a single node called the **root**.

Terminology



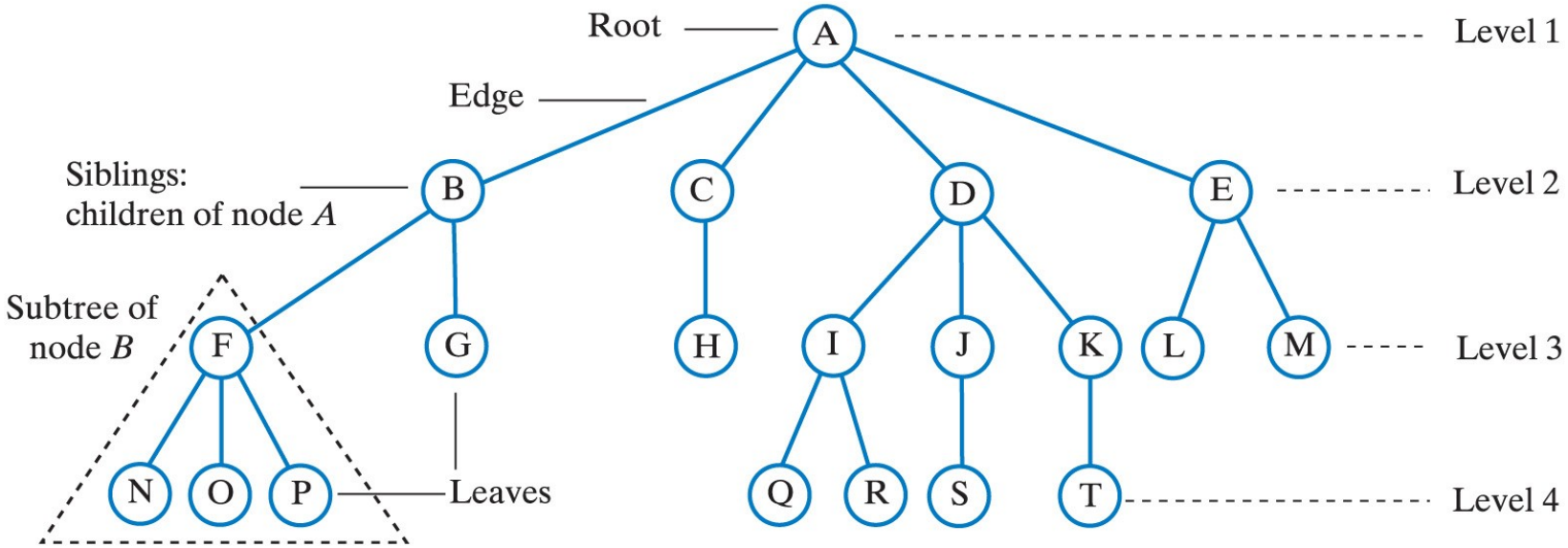
- The **children** of a node are those directly below it. *A* has 4 children: *B, C, D, E*
- The **parent** of a node is the node directly above it. *C*'s parent is *A*

Terminology



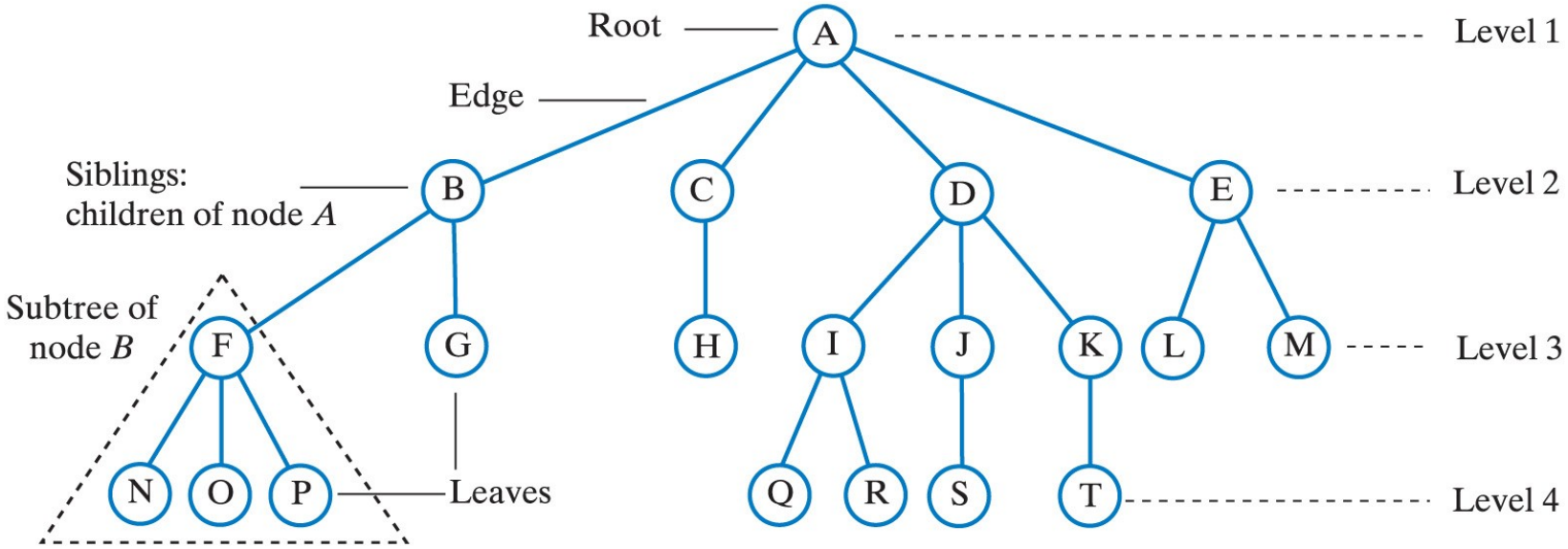
- All nodes have exactly 1 parent, except the root, which has no parent. *H*'s parent is *C*
- Nodes that share a parent are **siblings**. *B, C, D,* and *E* are siblings.

Terminology



- The nodes below a given node (on the downward paths to the leaves) are its **descendants**. 在他之后这条路上的所有
- *D*'s descendants are *I, J, K, Q, R, S* and *T.*

Terminology

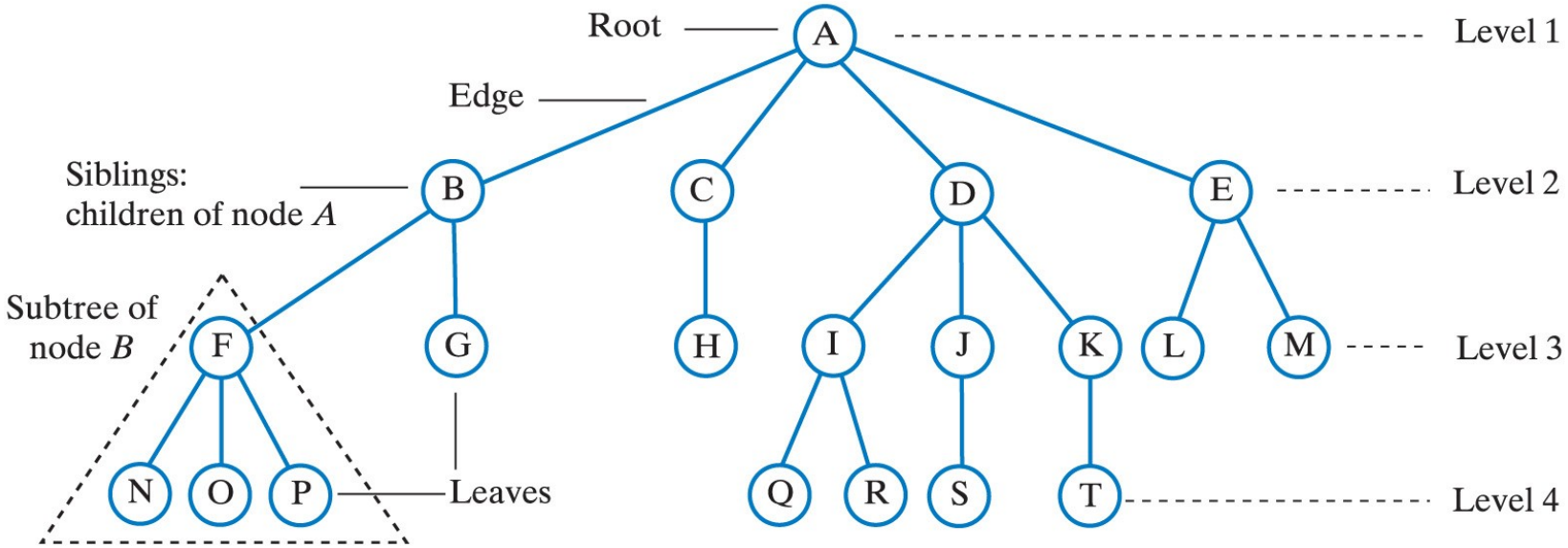


- The nodes above a given node (on the upward path towards the root) are its **ancestors**.

在他之前这条路上的所有

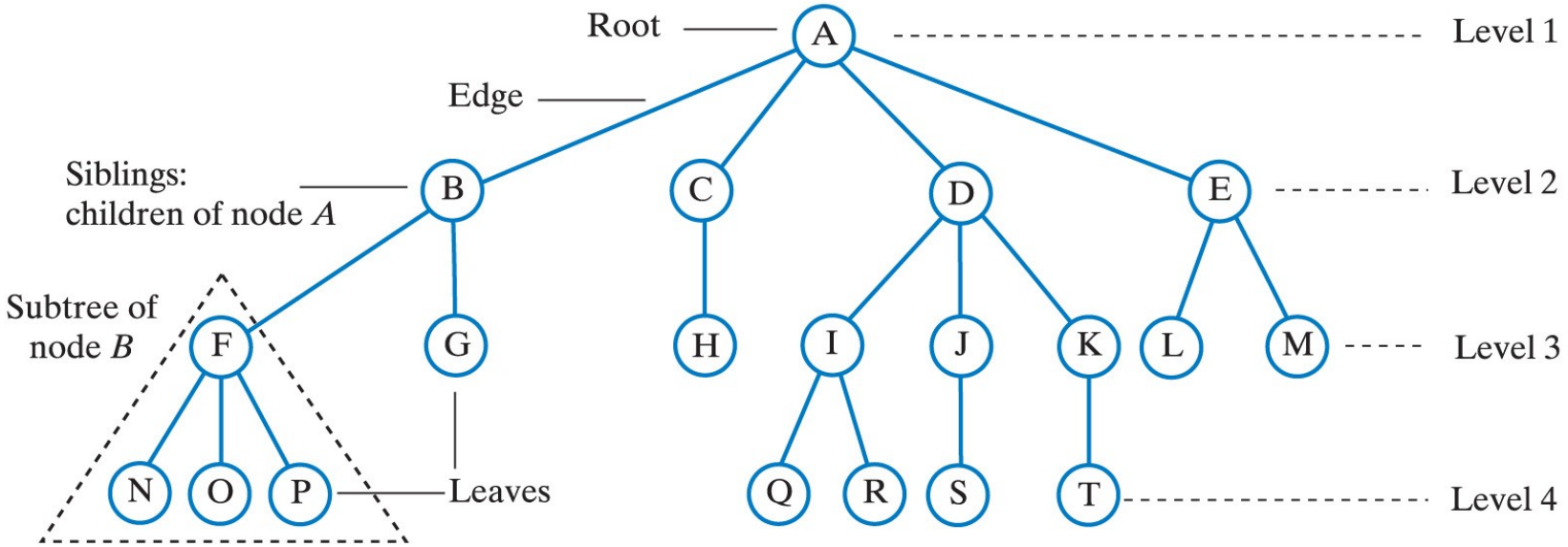
- *Q*'s ancestors are *I*, *D*, and *A*

Terminology



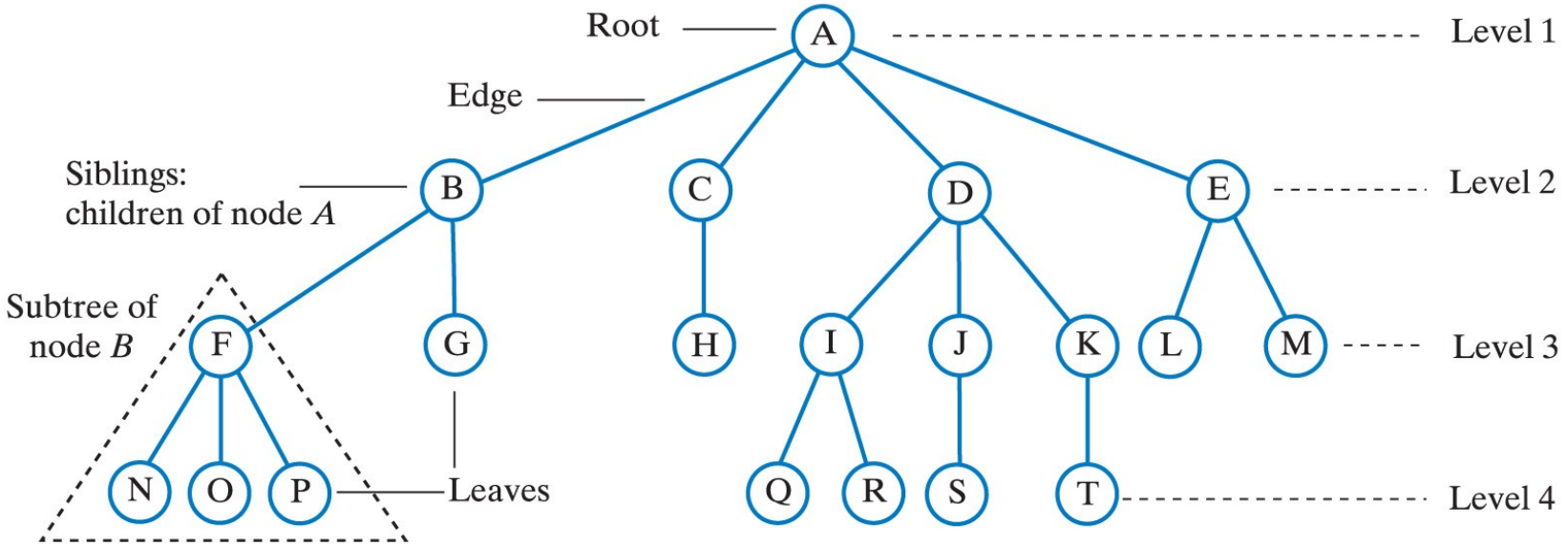
- A **leaf** is a node that has no children.
- Any node that is not a leaf is an **interior node** (or **non-leaf** node).

Exercise



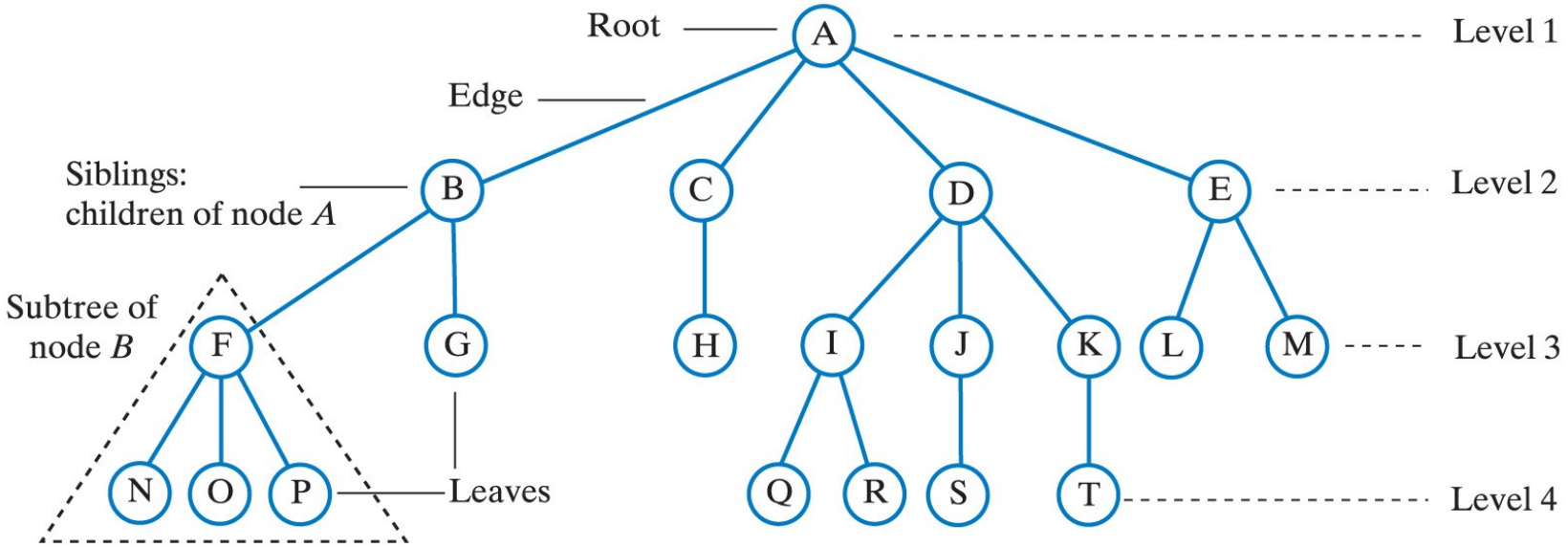
- What are the leaf nodes? N O P G H Q R S T L M
- What are the siblings of J? I K
- What are the children of E? L M
- What are the descendants of B? F G N O P
- What are the ancestors of P? F B A

More Terminology



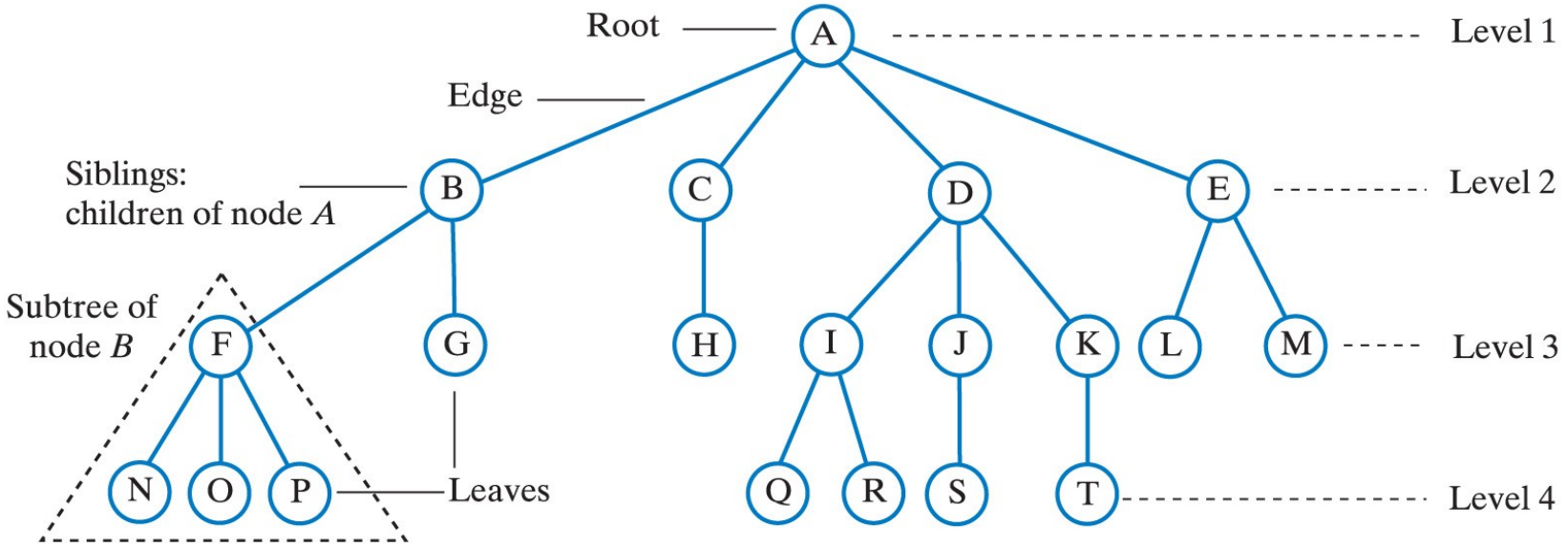
- A **subtree of a node** is a tree rooted at one of that node's children.
- Node *B* has 2 subtrees.
- Node *A* has 4 subtrees.

More Terminology



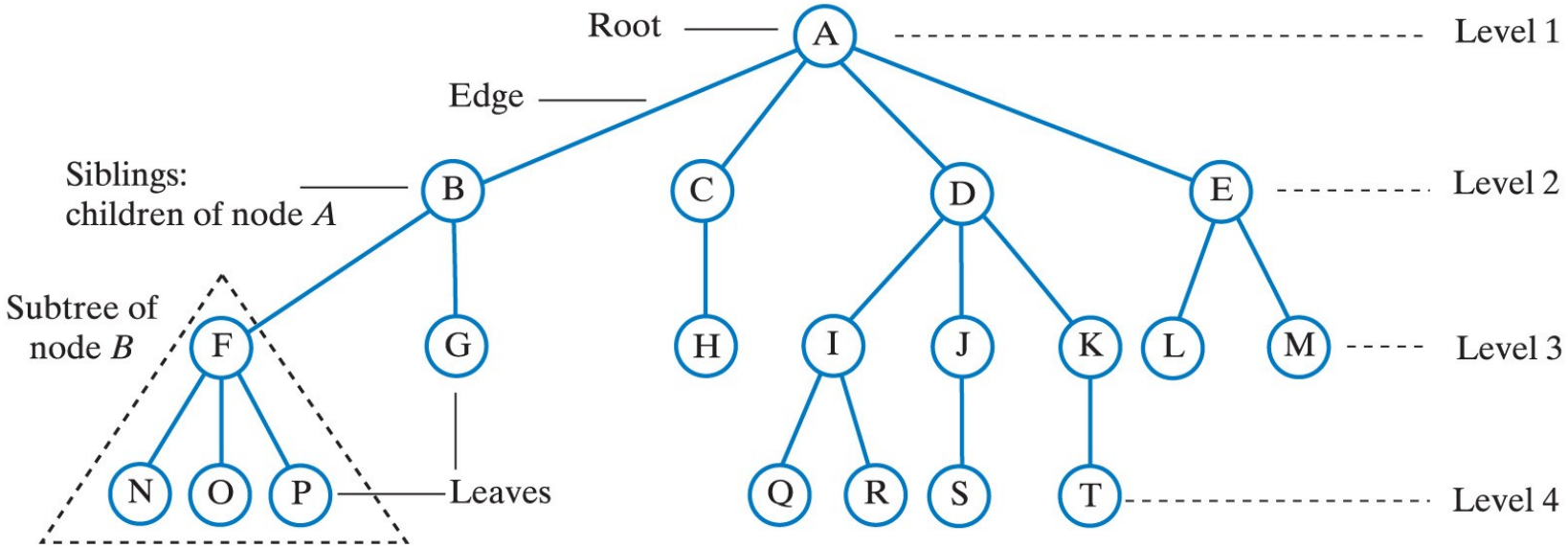
- We can reach any node in a tree by following a **path** that begins at the root and goes from node to node along the edges that connect them.
- The path to *R* is *A D I R*

More Terminology



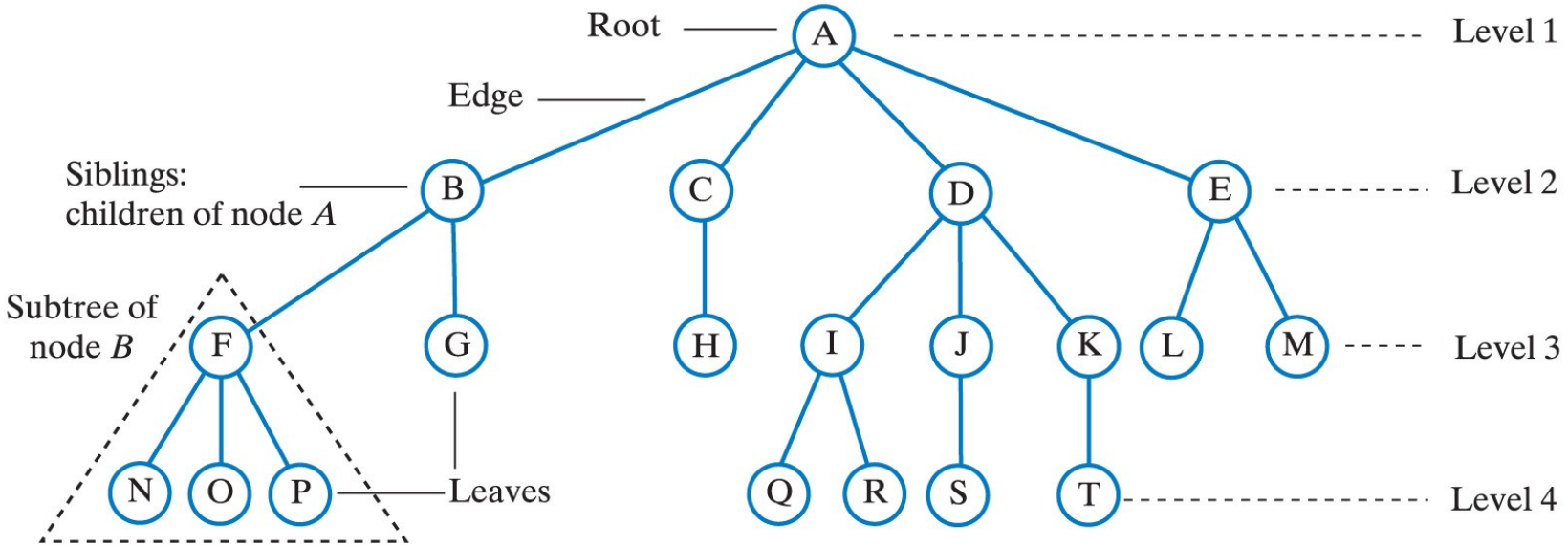
- The **length of a path** is the number of edges that compose it.
- The length of the path to *R* is 3

More Terminology



- The **height** of a tree is the number of levels in the tree, or the number of nodes along the longest path between the root and a leaf.
- This example tree has height 4.

More Terminology



- The path between the root and any other node is unique – there is no circularity in a tree.
- A tree-like data structure that has circularity is called a **graph**.

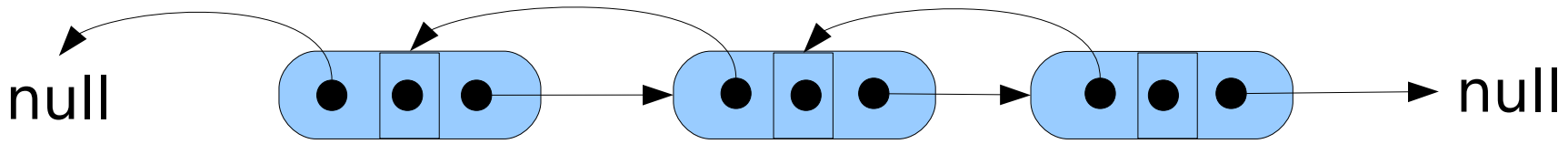
More Terminology

- In a **general tree**, each node can have any number of children.
- A tree in which the nodes have at most *n* children is called an **n-ary** tree.
- In particular, a tree whose nodes have at most 2 children is called a **binary tree**.

Binary Trees

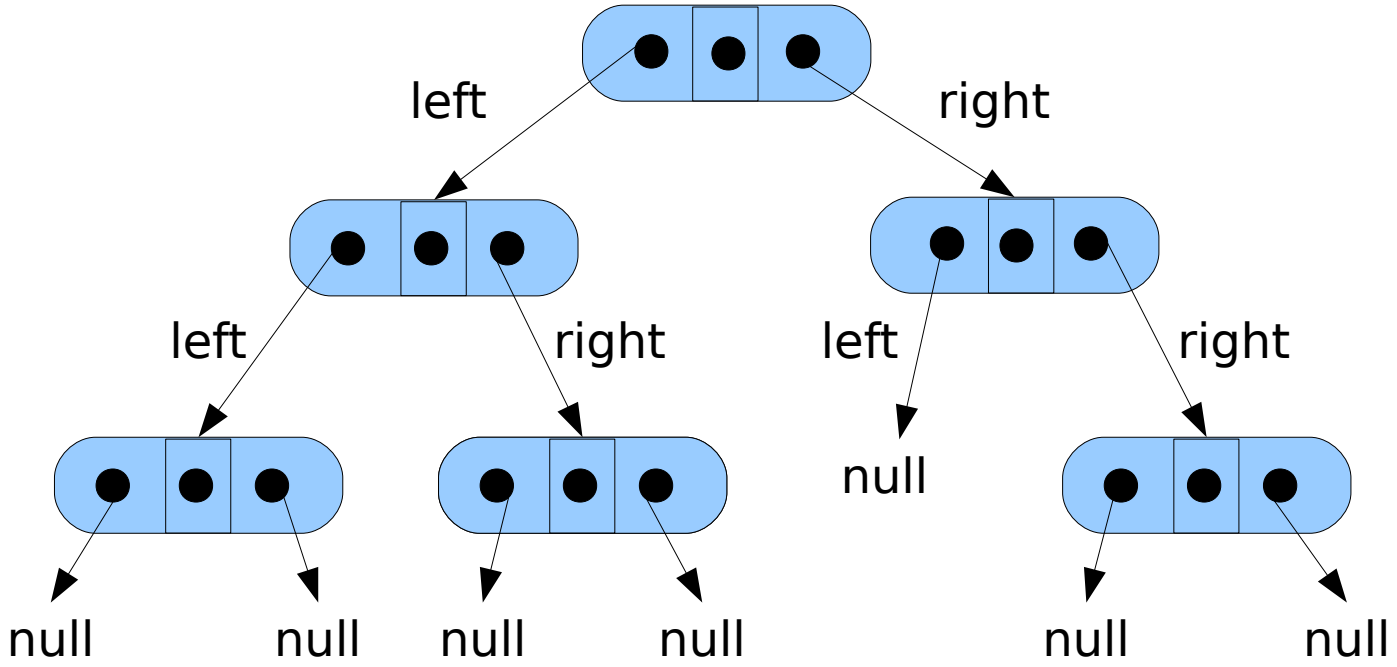
- We know how to create and link nodes together to form a list.
- A doubly-linked list can be formed by using two references in the node object – one to point to the next node and one to point to the previous node in the list.

dynamic ds LINKED LIST中学过



Binary Trees

- By using a similar node class with 2 references, we can form binary trees.
- Each node contains a reference to its *left* and its *right* subtree.



Tree Traversals

- When we iterate a linear data structure, such as a list, the order in which to process the data is clear.
- When we iterate a tree it is called a “穿越” **traversal**, and the order of processing the nodes is not unique.
- Each node must be **visited** (i.e. processed in some way) exactly once.

Tree Traversals

- We know that the subtrees of the root of a binary tree are also binary trees.
- We can use the recursive nature of a binary tree to define its traversal.
- To visit all the nodes in a binary tree we have to
  - visit the root
  - visit all nodes in the left subtree
  - visit all nodes in the right subtree

Tree Traversals

一定是left > right, root都ok

- Visiting the left subtree before the right subtree is simply a convention.
- The root can be visited before, between, or after its two subtrees.

root
left subtree
right subtree

left subtree
root
right subtree

left subtree
right subtree
root

Tree Traversals

- The order in which the root node and its subtrees are visited allows us to define the 3 most common tree traversals
  - preorder
  - inorder
  - postorder
- The *pre-*, *in-*, and *post-* refer to the visitation of the root node.

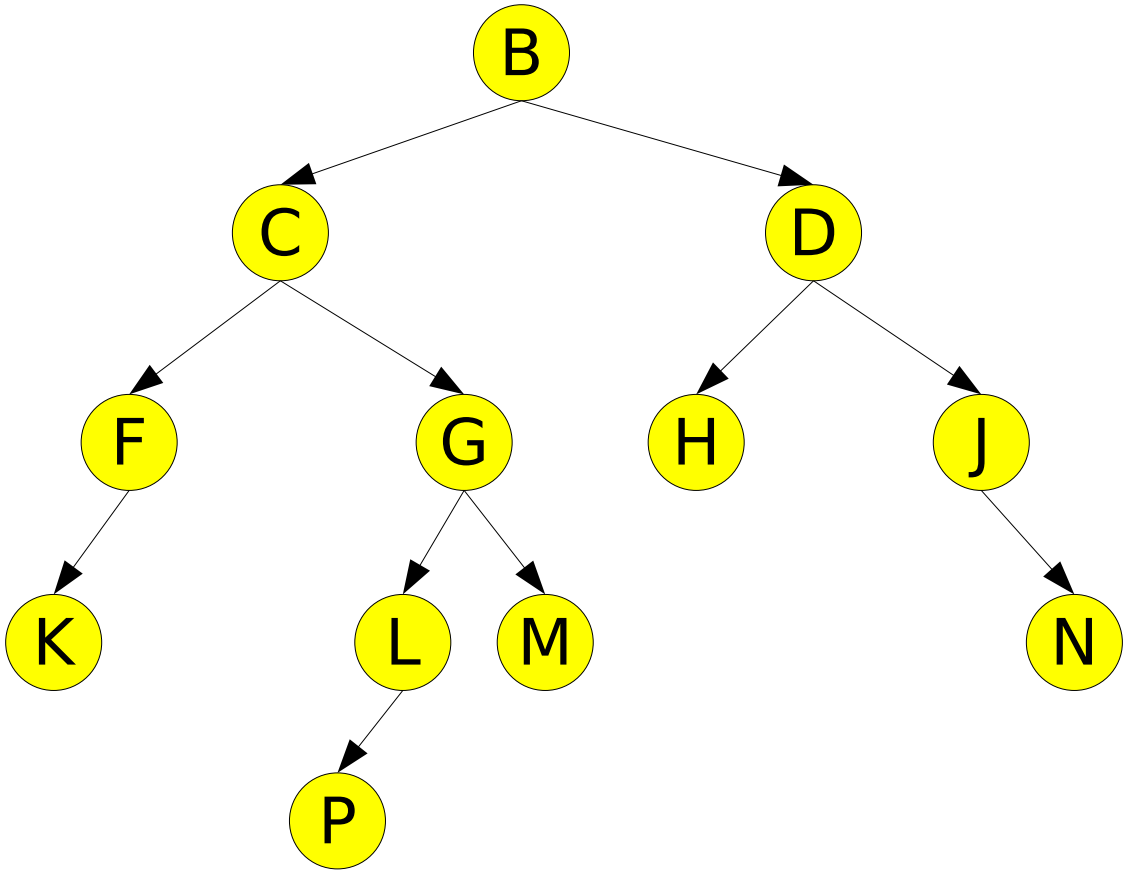
pre-order	**in-order**	**post-order**
root	left subtree	left subtree
left subtree	**root**	right subtree
right subtree	right subtree	**root**

Preorder Traversal

- In a preorder traversal, the root node is visited before its subtrees:
  - **visit the root**
  - visit all nodes in the left subtree
  - visit all nodes in the right subtree

Preorder Traversal Example

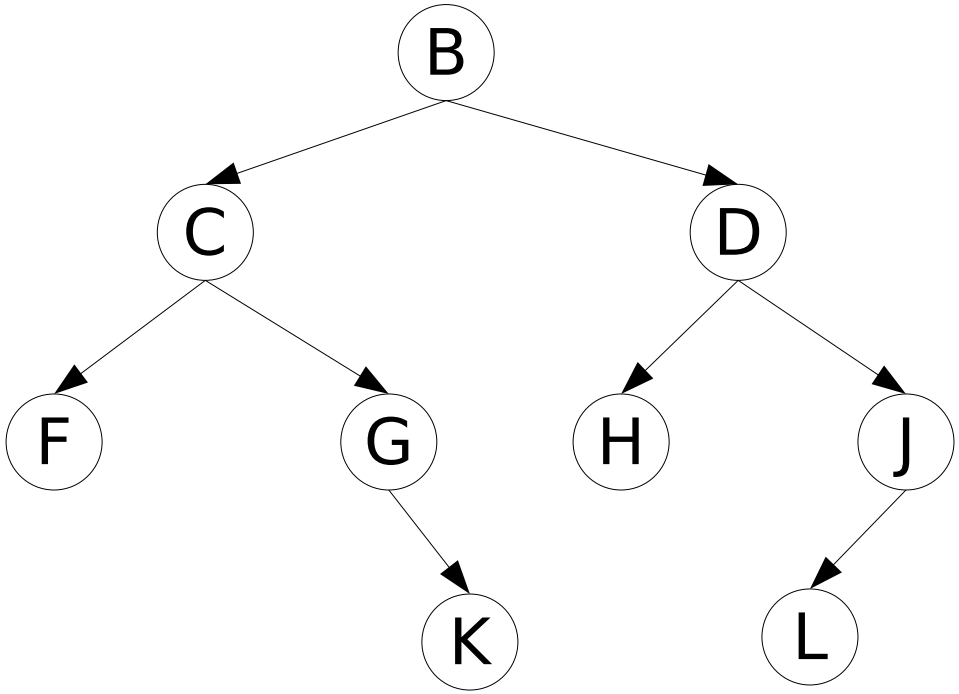
root
left
right



B C F K G L P M D H J N

Preorder Traversal Exercise

What is the preorder traversal of this tree?



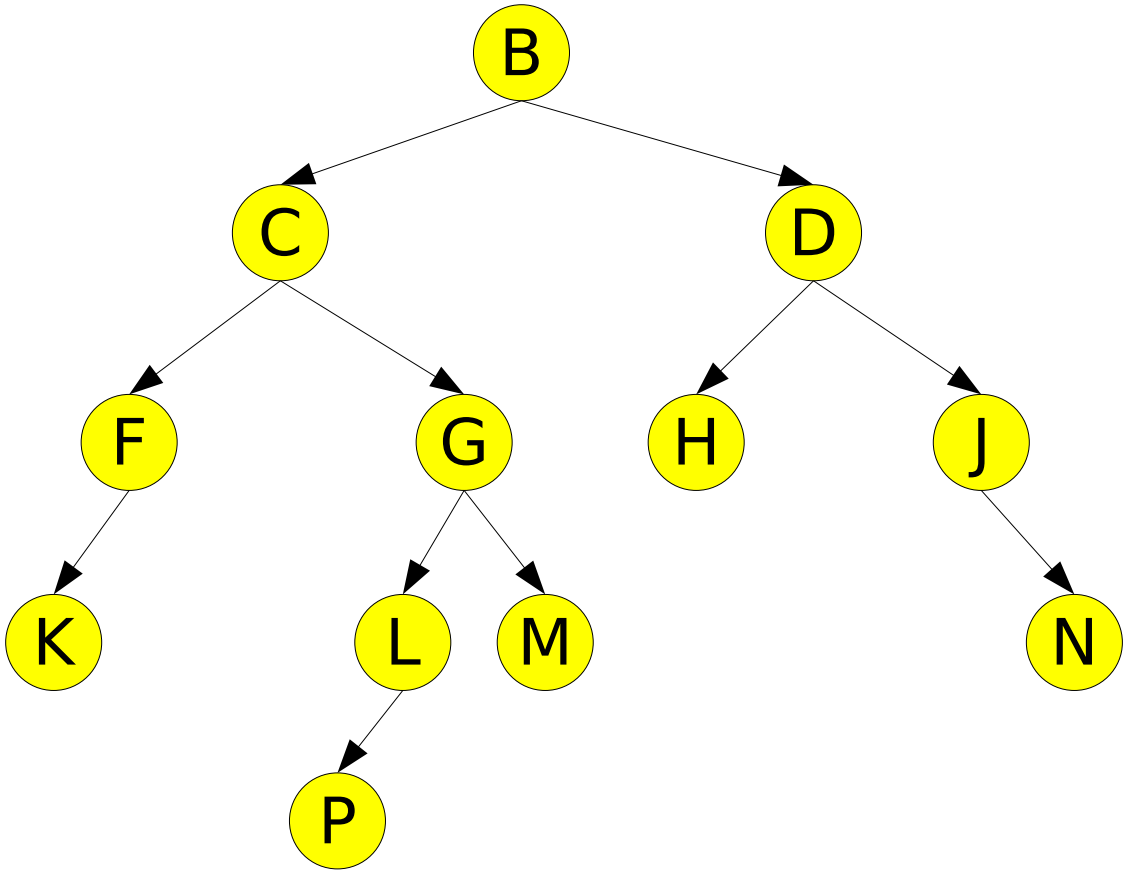
B C F G K D H J L

Inorder Traversal

- In an inorder traversal, the root node is visited between its subtrees:
  - visit all nodes in the left subtree
  - **visit the root**
  - visit all nodes in the right subtree

Inorder Traversal Example

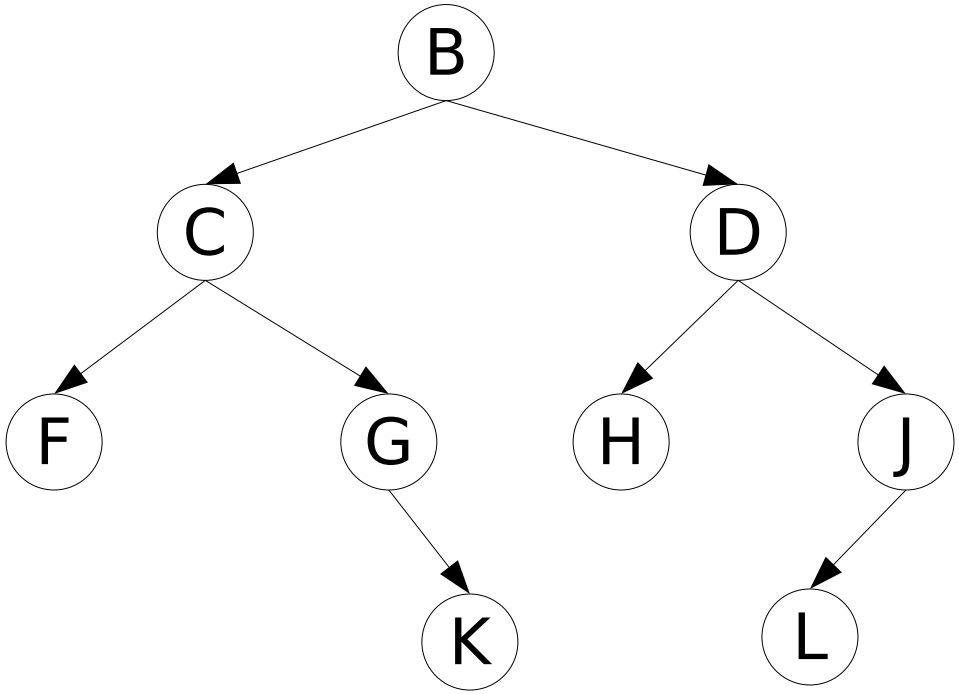
left
root
right



K F C P L G M B H D J N

Inorder Traversal Exercise

What is the inorder traversal of this tree?



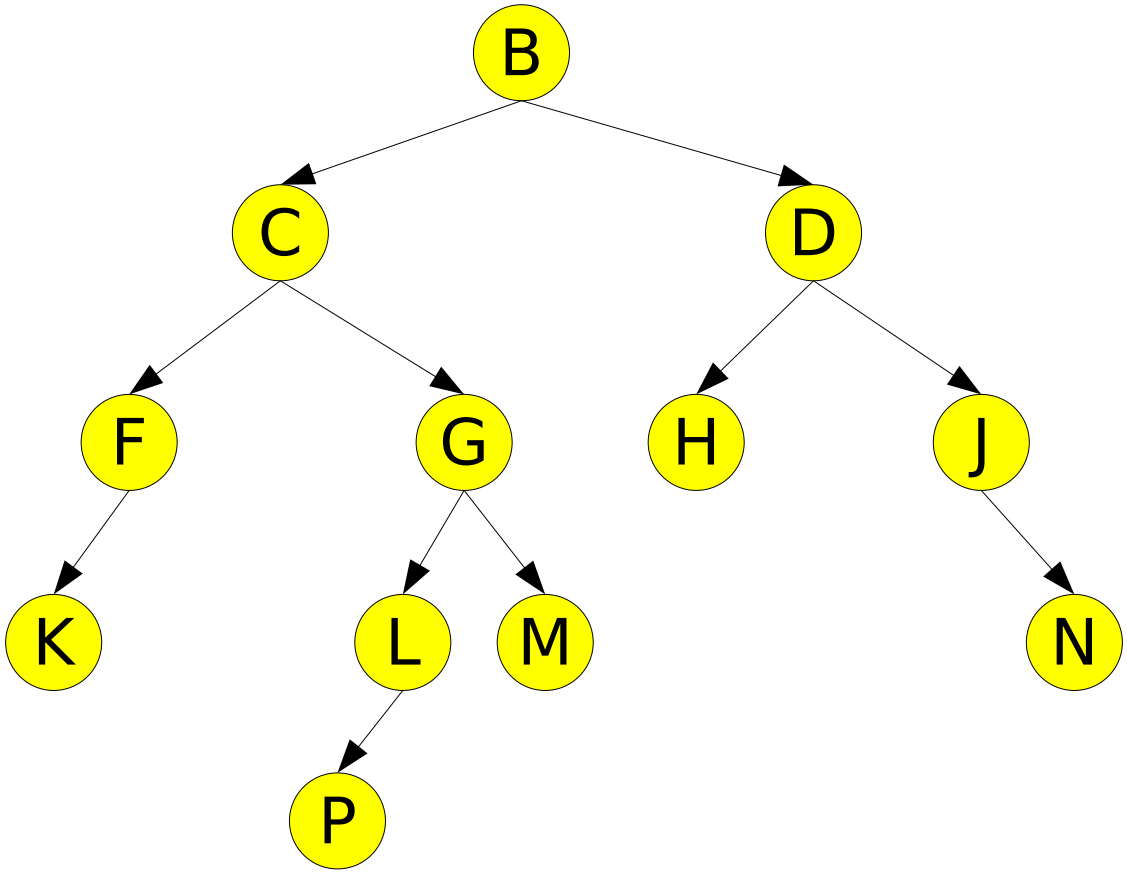
F C G K B H D L J

Postorder Traversal

- In a postorder traversal, the root node is visited after its subtrees:
  - visit all nodes in the left subtree
  - visit all nodes in the right subtree
  - **visit the root**

Postorder Traversal Example

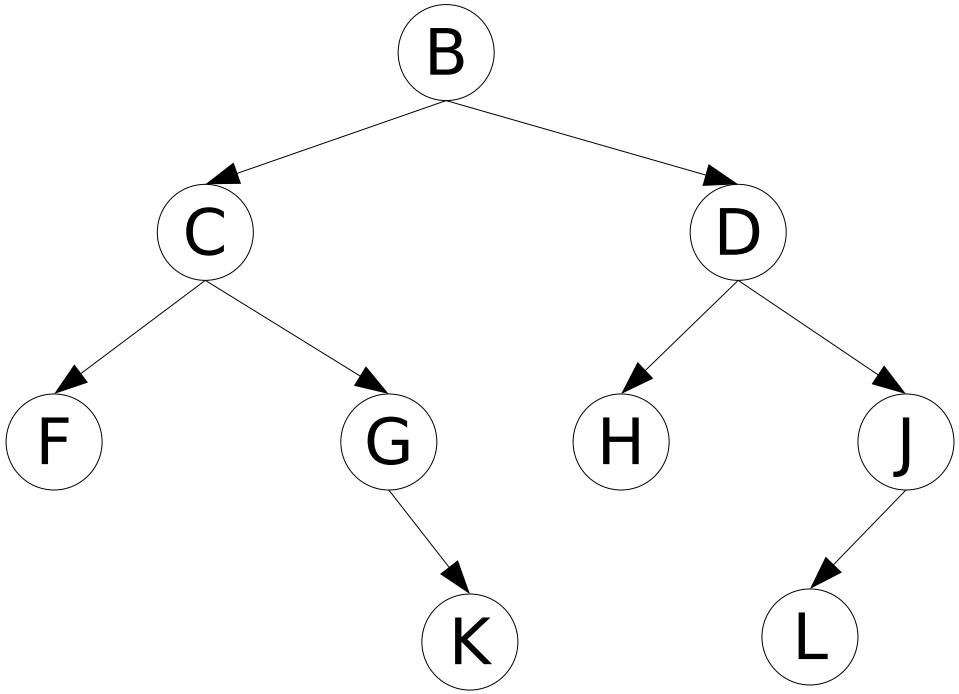
left
right
root



K F P L M G C H N J D B

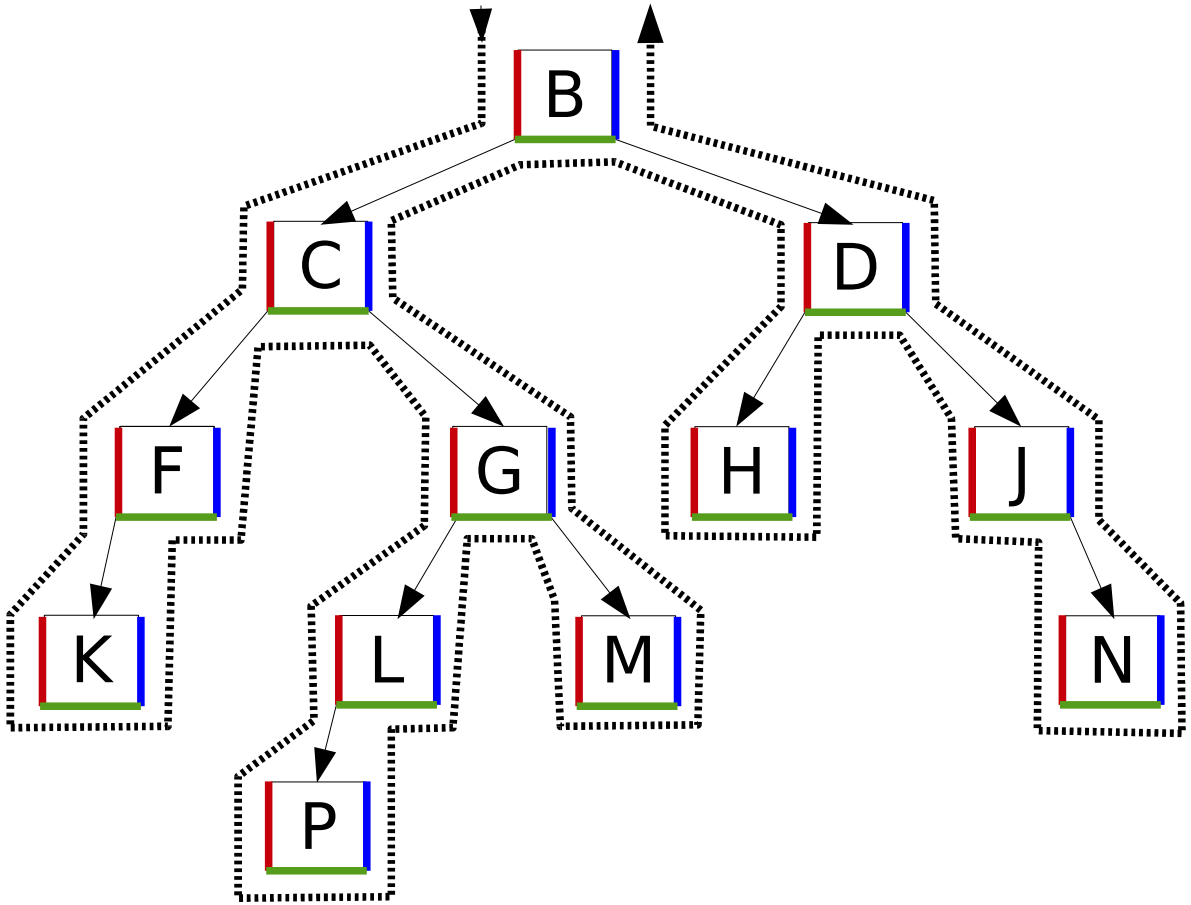
Postorder Traversal Exercise

What is the postorder traversal of this tree?



F K G C H L J D B

Distinguishing the Traversal Types



Preorder (left/red window): B C F K G L P M D H J N

Inorder (bottom/green window): K F C P L G M B H D J N

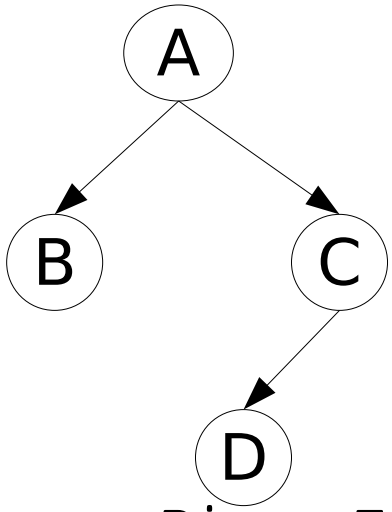
Postorder (right/blue window): K F P L M G C H N J D B

Building a Binary Tree

- Unlike classes that represent lists, stacks, or queues, tree classes often do **not** have methods to add or remove elements.
- There is no obvious place to add a new element.
- Removing a node is even less clear
  - how would you indicate which node should be removed? - can't number them like in a list
  - what happens to the children of the removed node?

Building a Binary Tree

- Trees are built up, node by node.
- Let's build this tree:

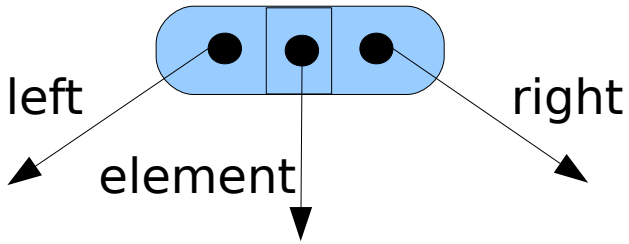



```
BinaryTree<String> b = new BinaryTree<String>("B");
BinaryTree<String> d = new BinaryTree<String>("D");
BinaryTree<String> c = new BinaryTree<String>("C", d, null);
BinaryTree<String> a = new BinaryTree<String>("A", b, c);
```

BinaryTreeNode<T> Instance Variables

a class to represent a node in the tree

```
public class BinaryTreeNode<T> {
    T element;
    BinaryTreeNode<T> left;
    BinaryTreeNode<T> right;

    ...

}
```



a class to represent a node in the tree

BinaryTreeNode<T> Constructors

```
public class BinaryTreeNode<T> {
    T element;
    BinaryTreeNode<T> left;
    BinaryTreeNode<T> right;

    public BinaryTreeNode(T dataObj) {
        element = dataObj;
        left = right = null;
    }

    public BinaryTreeNode(T dataObj,
                          BinaryTreeNode<T> l,
                          BinaryTreeNode<T> r) {
        element = dataObj;
        left = l;
        right = r;
    }
}
```

BinaryTreeNode<T> toString

a class to represent a node in the tree

```
public class BinaryTreeNode<T> {
    T element;
    BinaryTreeNode<T> left;
    BinaryTreeNode<T> right;

    <Constructors>

    public String toString() {
        return element.toString();
    }
}
```

BinaryTreeNode<T> isLeaf - Exercise

a class to represent a node in the tree

```
public class BinaryTreeNode<T> {
    T element;
    BinaryTreeNode<T> left;
    BinaryTreeNode<T> right;

    <Constructors>

    public boolean isLeaf() {

        //*************  To Do *********
                return (left == null) && (right == null);
    }
}
```

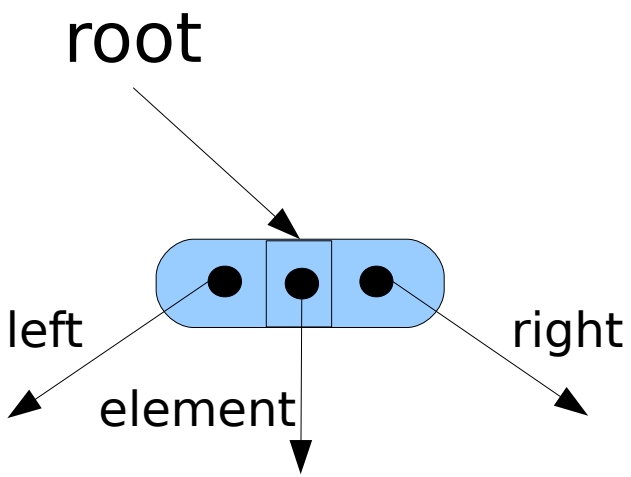
BinaryTree<T>

- Now that we have a class to represent a node in the tree, we can define a class to represent the tree itself.
- The only node that we need a reference to is the root node. All other nodes are accessible from the root.
- We will declare a reference to the root node as an instance variable.

只有root需要ref，其他nodes都可通过root获取

BinaryTree<T> Instance Variable

```
public class BinaryTree<T> {
    BinaryTreeNode<T> root;

    ...

}
```



BinaryTree<T> Constructors

```
public class BinaryTree<T> {
    BinaryTreeNode<T> root;

    public BinaryTree() {
        root = null;
    }

    public BinaryTree(T element) {
        root = new BinaryTreeNode<T> (element);
    }

    ...
}
```

BinaryTree<T> Constructors, cont.

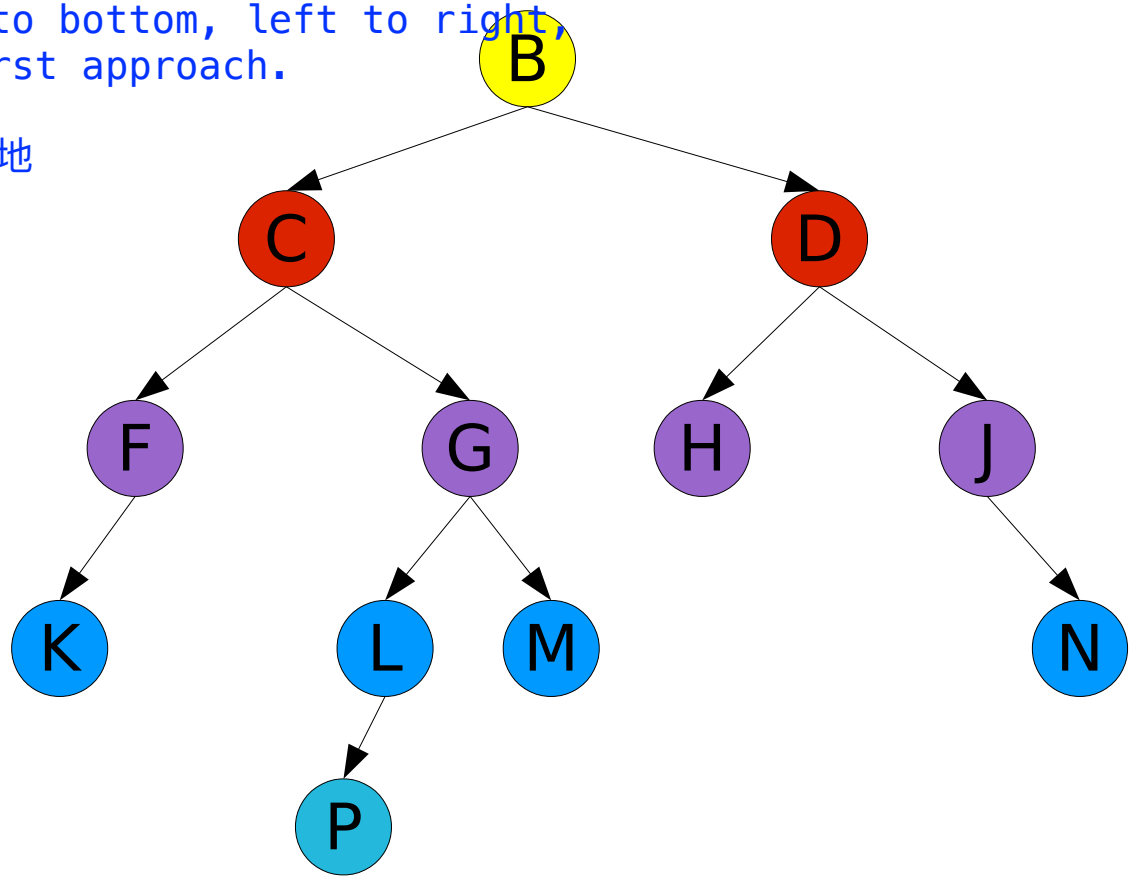
```
public class BinaryTree<T> {
    ...

    public BinaryTree(T element, BinaryTree<T> leftSubtree,
                                 BinaryTree<T> rightSubtree) {
        root = new BinaryTreeNode<T> (element);

        if (leftSubtree != null) {
            root.left = leftSubtree.root;
        } else {
            root.left = null;
        }

        if (rightSubtree != null) {
            root.right = rightSubtree.root;
        } else {
            root.right = null;
        }
    }
}
```

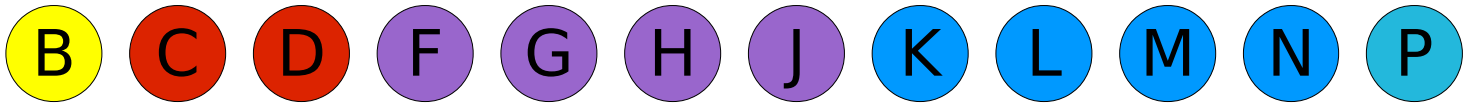
Levelorder traversal of a binary tree

Levelorder tree traversal

a tree traversal method that visits nodes level by level from top to bottom, left to right, using a breadth-first approach.

也就是从上到下一层一层地

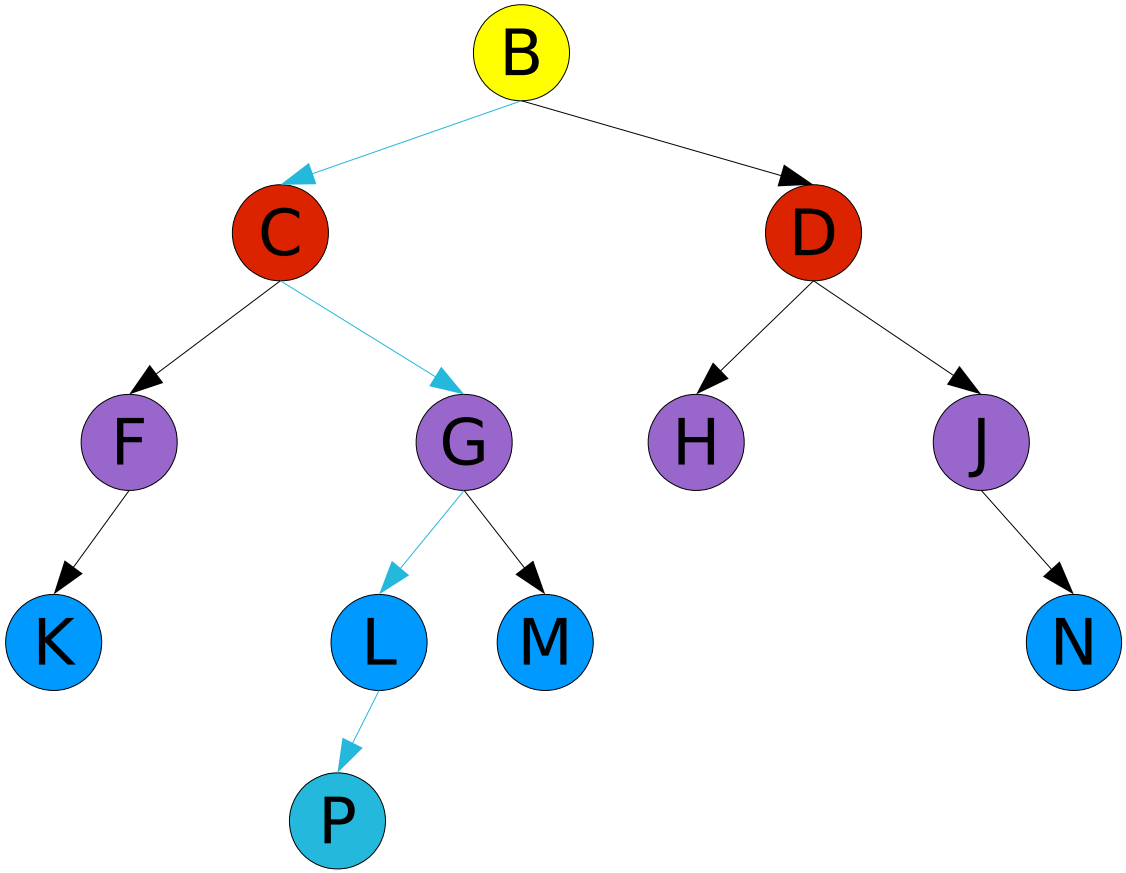


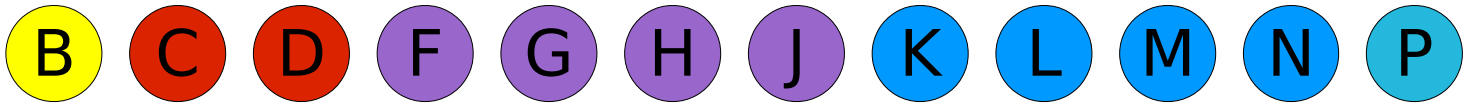


Recursive level order traversal

```
For d = 1 to height of tree
     levelorder(root node of tree, d)

levelorder(node, level)
   if node is NULL then return
   if level is 1 then
      do something with node
   else if level greater than 1 then
     levelorder(leftChild of node, level-1)
     levelorder(rightChild of node, level-1)
```

Levelorder traversal recursive

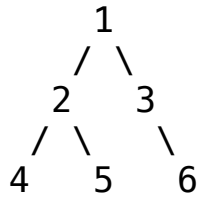




Level order traversal using a queue

```
Create queue aQueue
Add root node to aQueue

While aQueue is not empty
     Get node from queue into aNode
     Do something with aNode
     If exists add left child of aNode to aQueue
     If exists add right child of aNode to aQueue
```

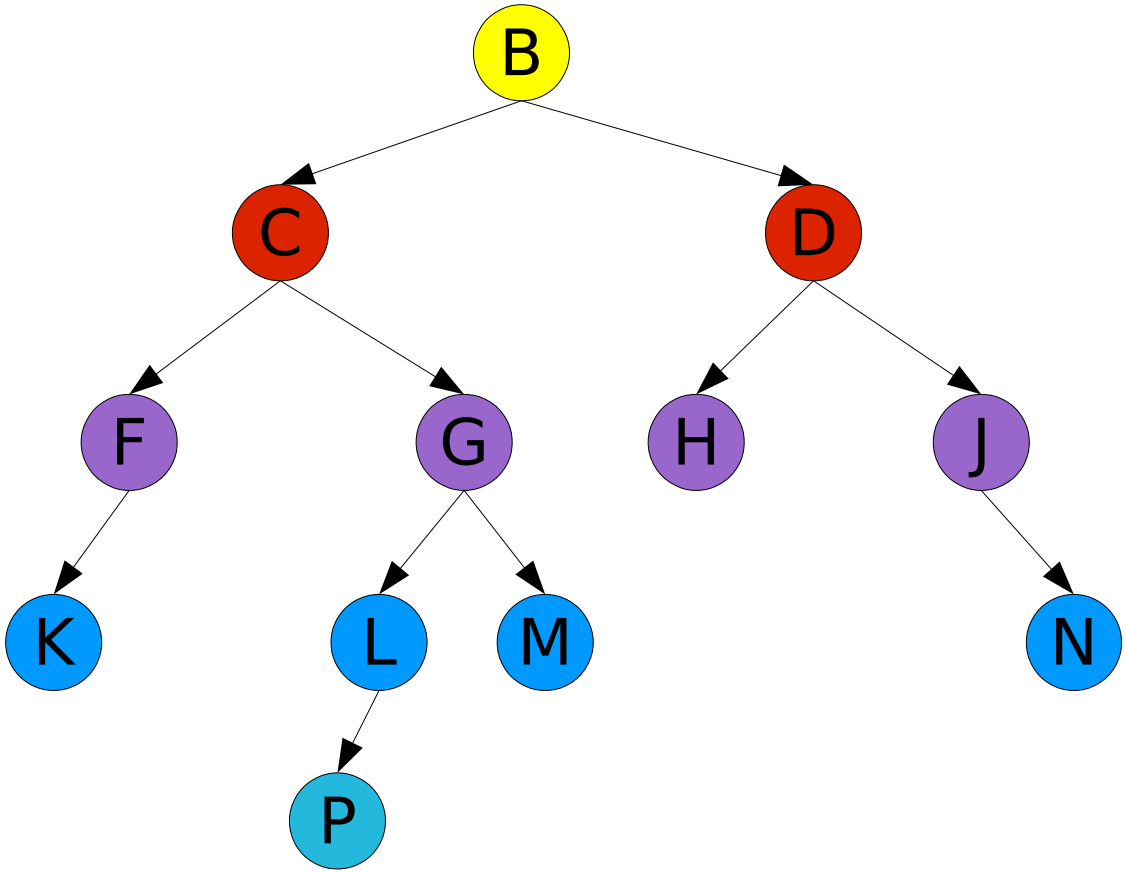
```java
import java.util.LinkedList;
import java.util.Queue;

// 定义树的节点类
class TreeNode {
    int value;
    TreeNode left, right;

    TreeNode(int value) {
        this.value = value;
        left = right = null;
    }
}

public class LevelOrderTraversal {
    // 实现层次遍历的函数
    public static void levelOrder(TreeNode root) {
        if (root == null) return;

        Queue<TreeNode> queue = new LinkedList<>(); // 创建一个队列来存储节点
        queue.offer(root); // 将根节点入队

        // 只要队列不为空就继续遍历
        while (!queue.isEmpty()) {
            TreeNode currentNode = queue.poll(); // 从队列中取出一个节点
            System.out.print(currentNode.value + " "); // 打印该节点的值

            // 如果左子节点不为空，则将左子节点入队
            if (currentNode.left != null) {
                queue.offer(currentNode.left);
            }

            // 如果右子节点不为空，则将右子节点入队
            if (currentNode.right != null) {
                queue.offer(currentNode.right);
            }
        }
    }

    // 测试层次遍历的例子
    public static void main(String[] args) {
        // 构建示例二叉树
        TreeNode root = new TreeNode(1);
        root.left = new TreeNode(2);
        root.right = new TreeNode(3);
        root.left.left = new TreeNode(4);
        root.left.right = new TreeNode(5);
        root.right.right = new TreeNode(6);

        // 执行层次遍历并输出结果
        System.out.print("Level-order Traversal: ");
        levelOrder(root);  // 输出应该为：1 2 3 4 5 6
    }
}
```



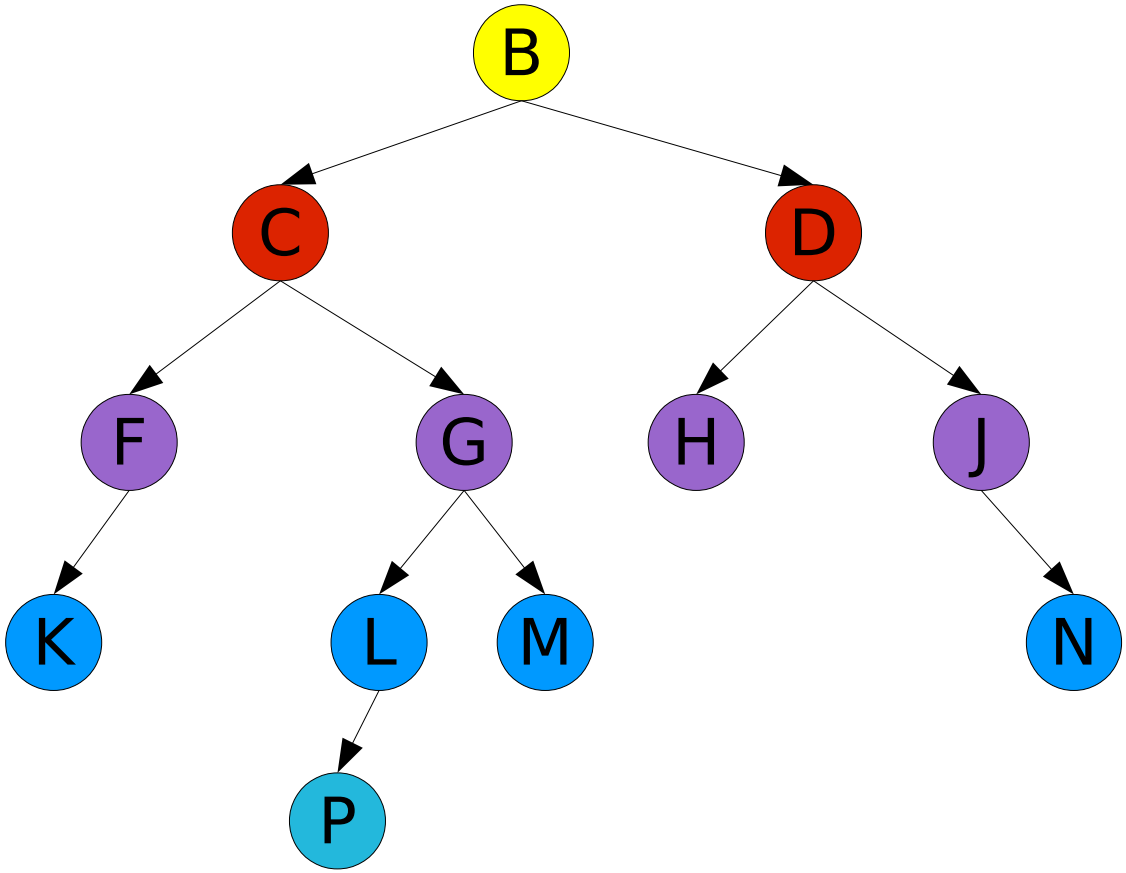
`.left` 和 `.right` 不是方法，而是二叉树节点类（TreeNode）中的属性。它们表示树节点的左子节点和右子节点。
通常我们会在创建 TreeNode 类时，定义这些属性以便表示节点之间的关系。

```
import java.util.LinkedList;
import java.util.Queue;

// 定义树的节点类
class TreeNode {
    int value;          // 节点值
    TreeNode left;      // 左子节点
    TreeNode right;     // 右子节点

    // 构造函数，初始化节点
    TreeNode(int value) {
        this.value = value;
        this.left = null;
        this.right = null;
    }
}

public class LevelOrderTraversal {
    // 实现层次遍历（Level-order traversal）的函数
    public static void levelOrder(TreeNode root) {
        if (root == null) return;  // 如果树为空，直接返回

        Queue<TreeNode> queue = new LinkedList<>();  // 创建队列来存储节点
        queue.offer(root);  // 将根节点入队

        // 只要队列不为空就继续遍历
        while (!queue.isEmpty()) {
            TreeNode currentNode = queue.poll();  // 从队列中取出一个节点
            System.out.print(currentNode.value + " ");  // 打印该节点的值

            // 如果左子节点不为空，将左子节点入队
            if (currentNode.left != null) {
                queue.offer(currentNode.left);
            }

            // 如果右子节点不为空，将右子节点入队
            if (currentNode.right != null) {
                queue.offer(currentNode.right);
            }
        }
    }

    // 测试层次遍历的例子
    public static void main(String[] args) {
        // 构建示例二叉树
        TreeNode root = new TreeNode(1);          // 根节点 1
        root.left = new TreeNode(2);              // 根节点的左子节点 2
        root.right = new TreeNode(3);             // 根节点的右子节点 3
        root.left.left = new TreeNode(4);         // 节点 2 的左子节点 4
        root.left.right = new TreeNode(5);        // 节点 2 的右子节点 5
        root.right.right = new TreeNode(6);       // 节点 3 的右子节点 6

        // 执行层次遍历并输出结果
        System.out.print("Level-order Traversal: ");
        levelOrder(root);  // 输出应该为：1 2 3 4 5 6
    }
}
```

Levelorder traversal using a queue

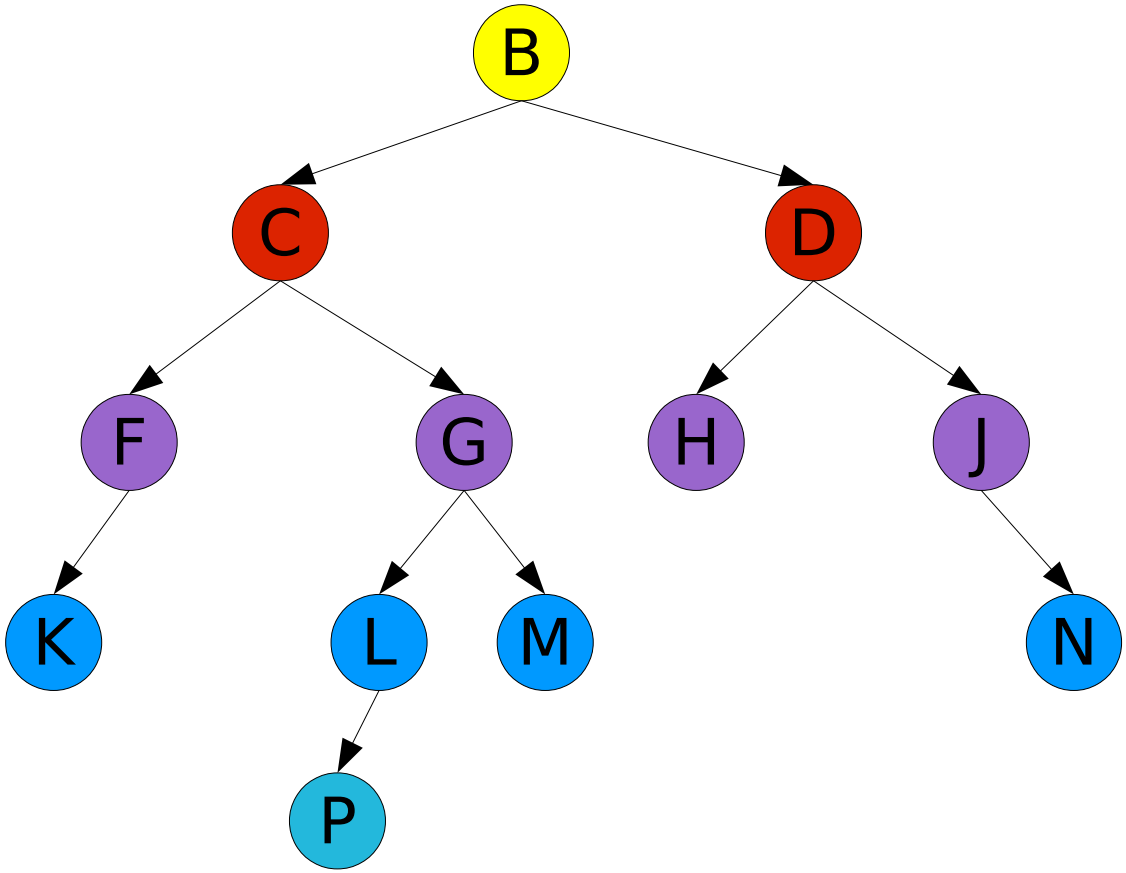


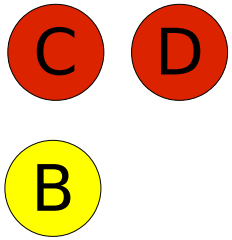
Levelorder traversal using a queue



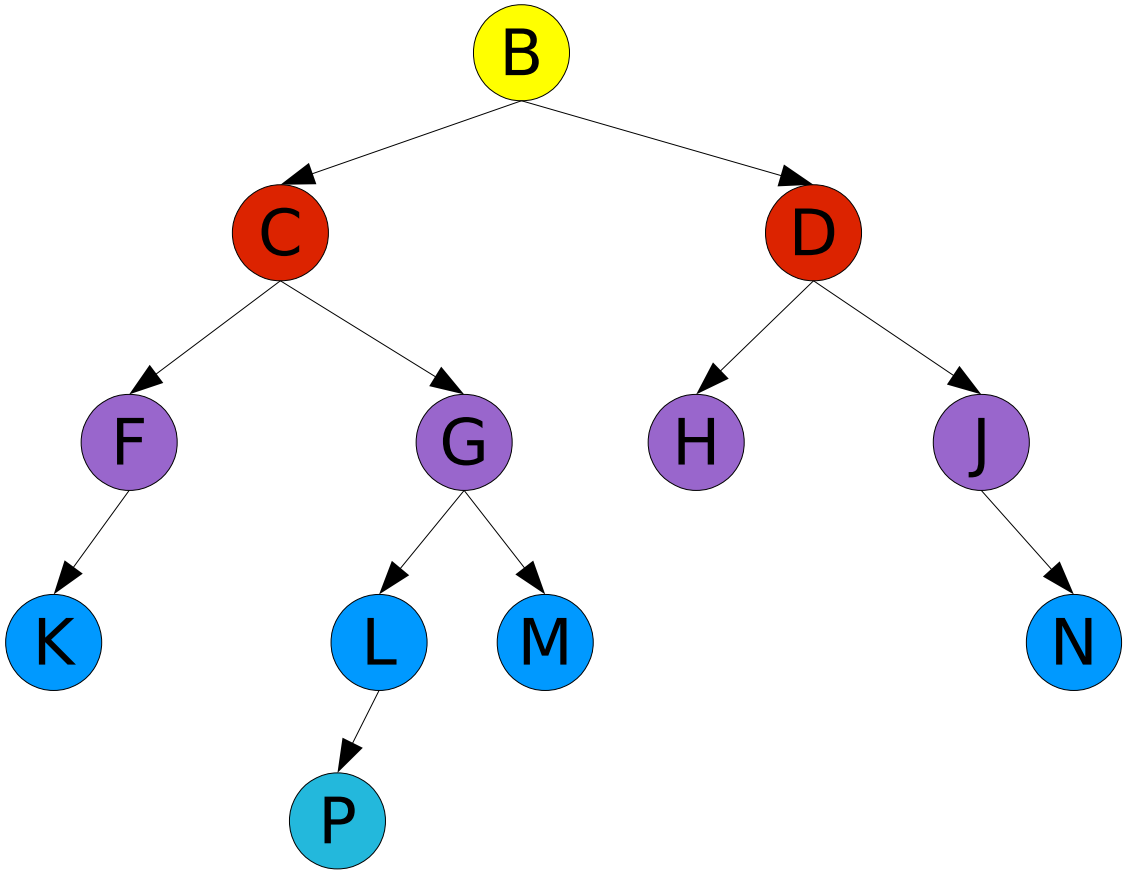
B

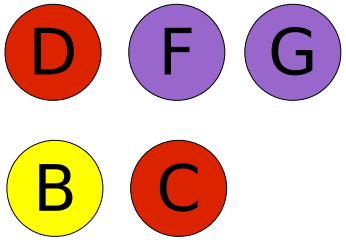
Levelorder traversal using a queue



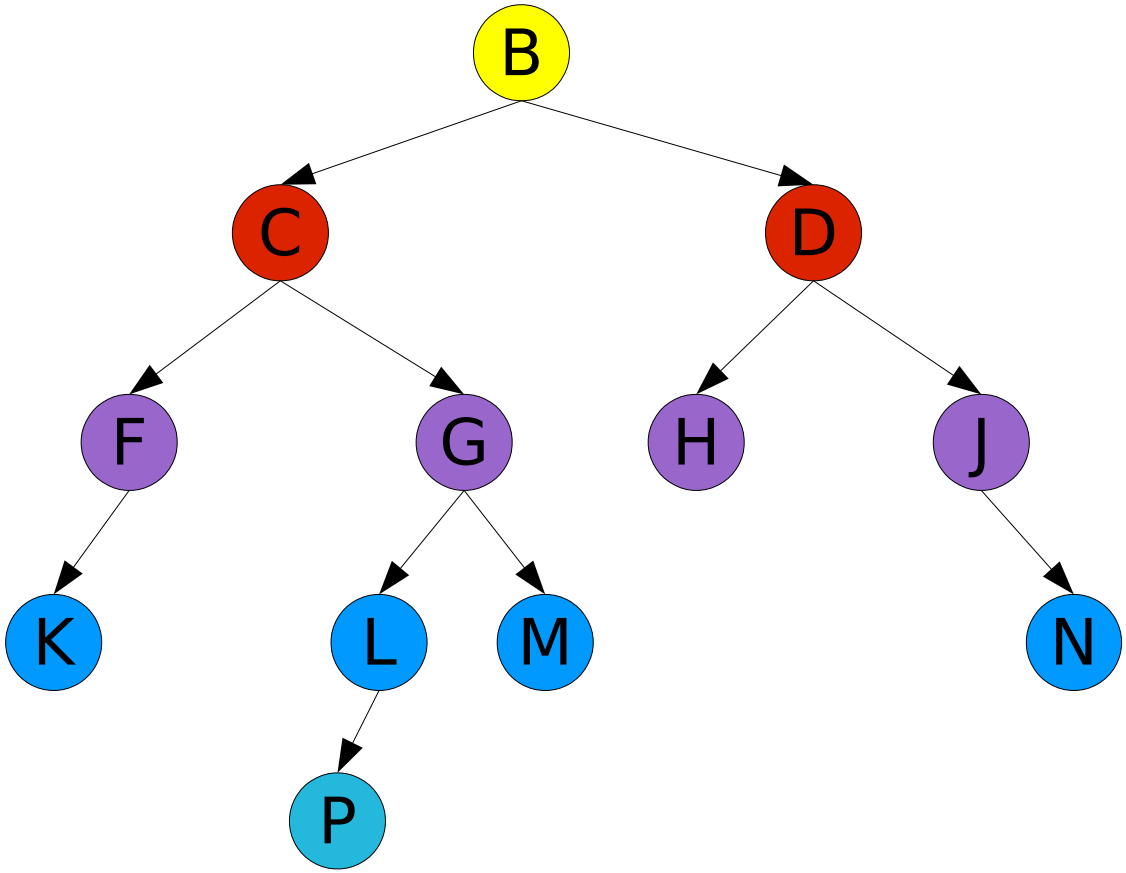


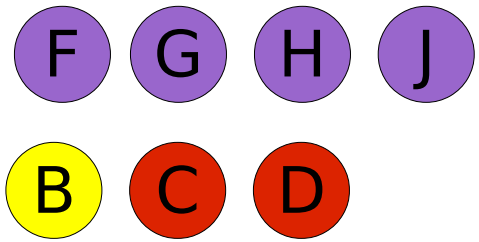
Levelorder traversal using a queue



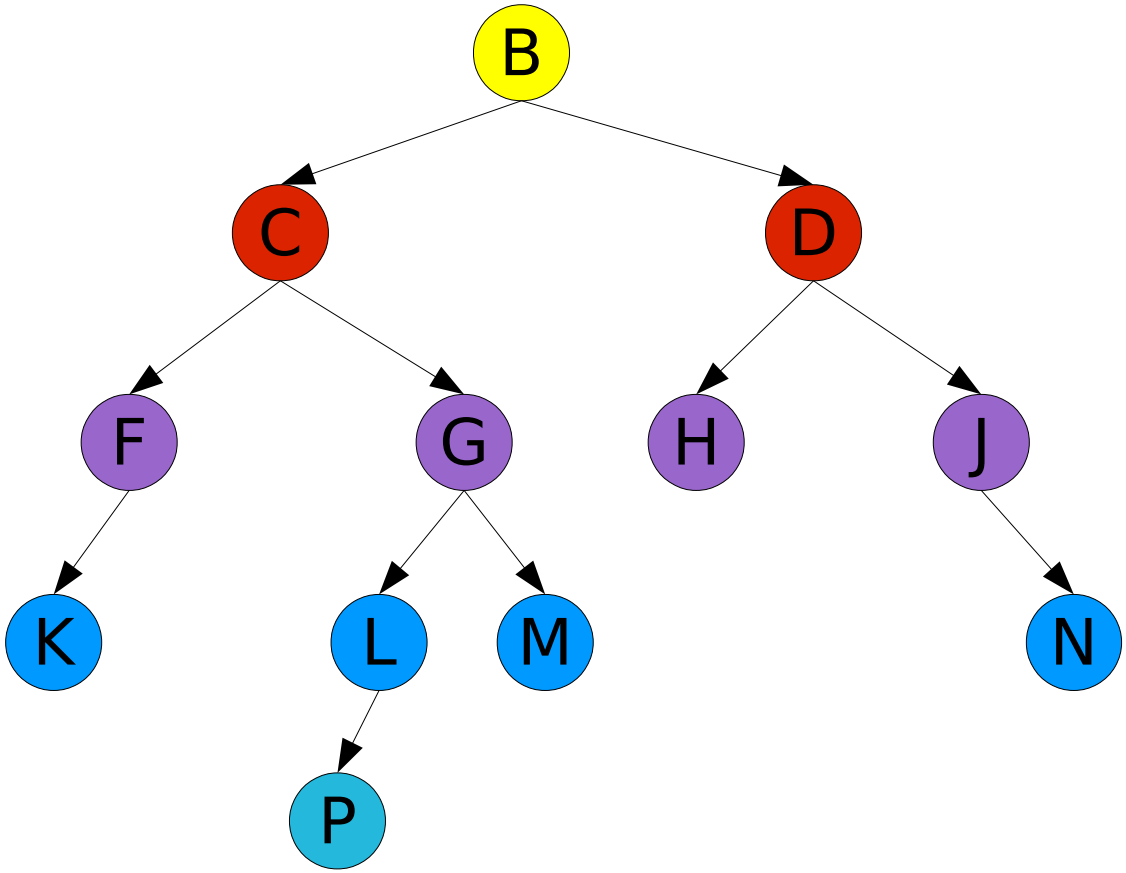


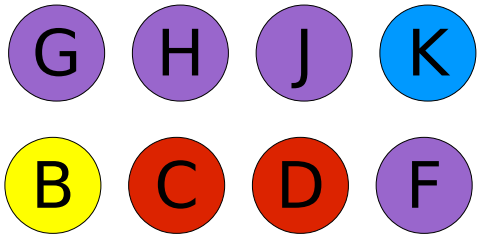
Levelorder traversal using a queue



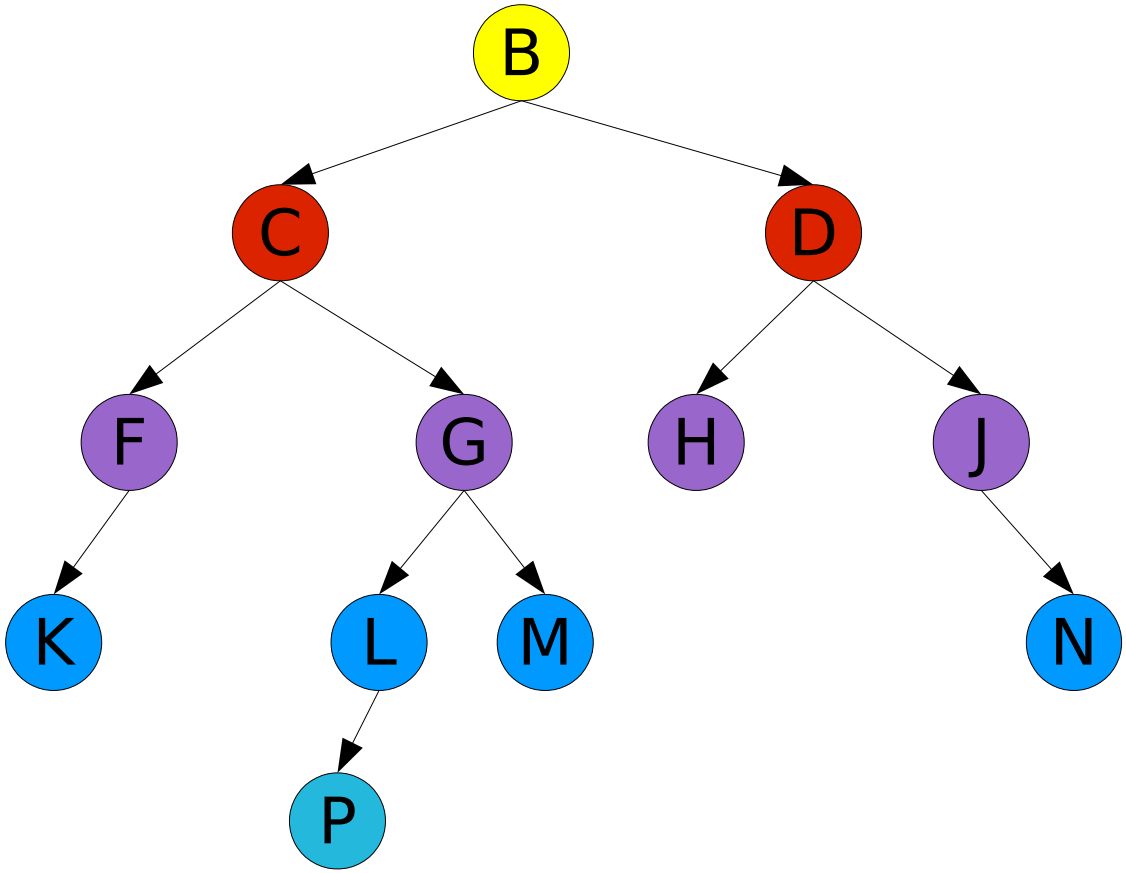


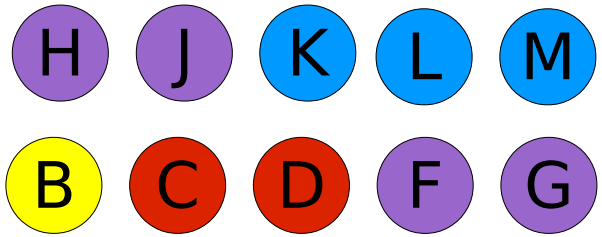
Levelorder traversal using a queue



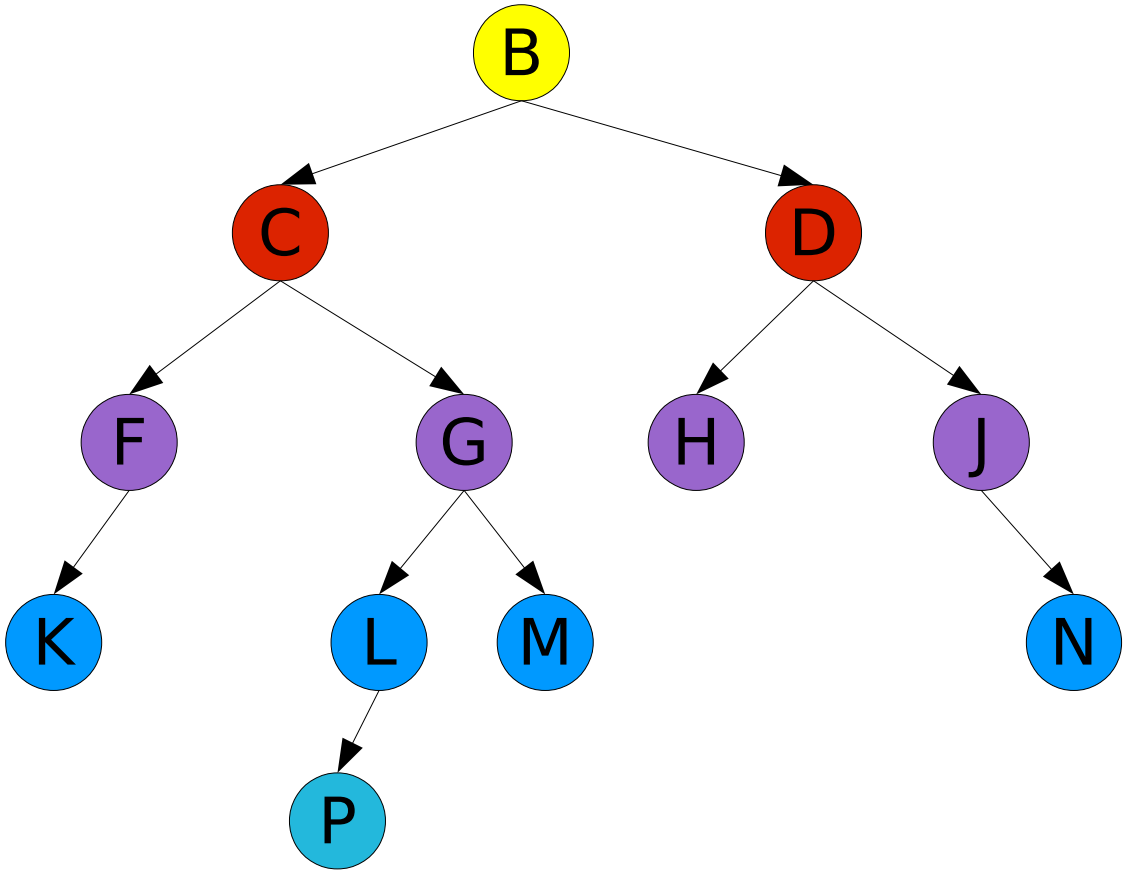


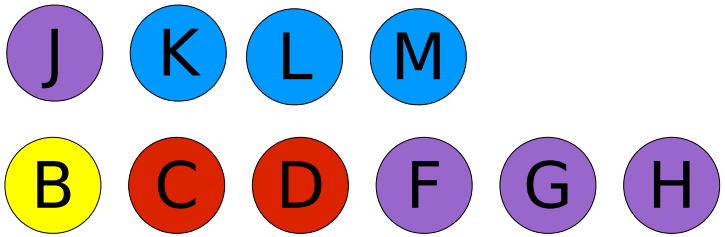
Levelorder traversal using a queue



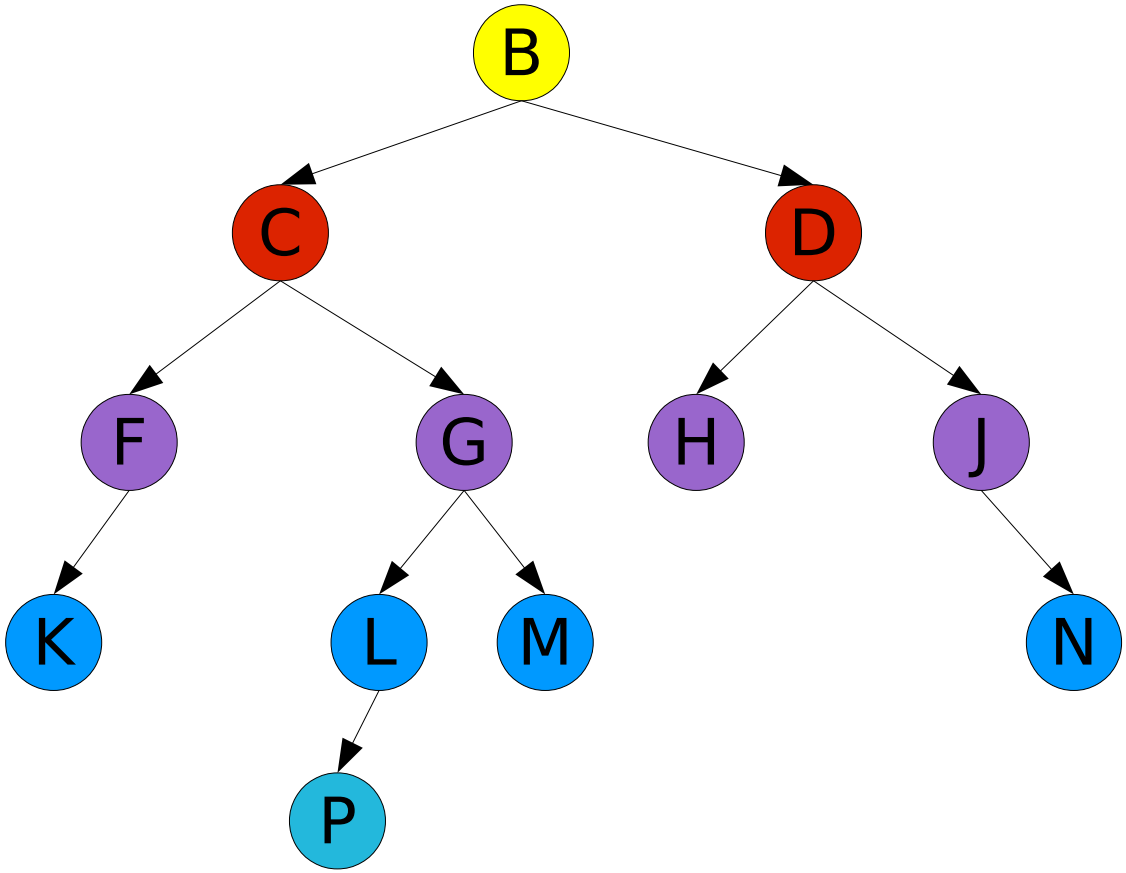


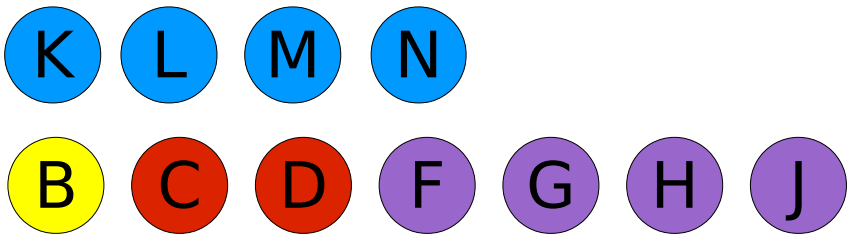
Levelorder traversal using a queue



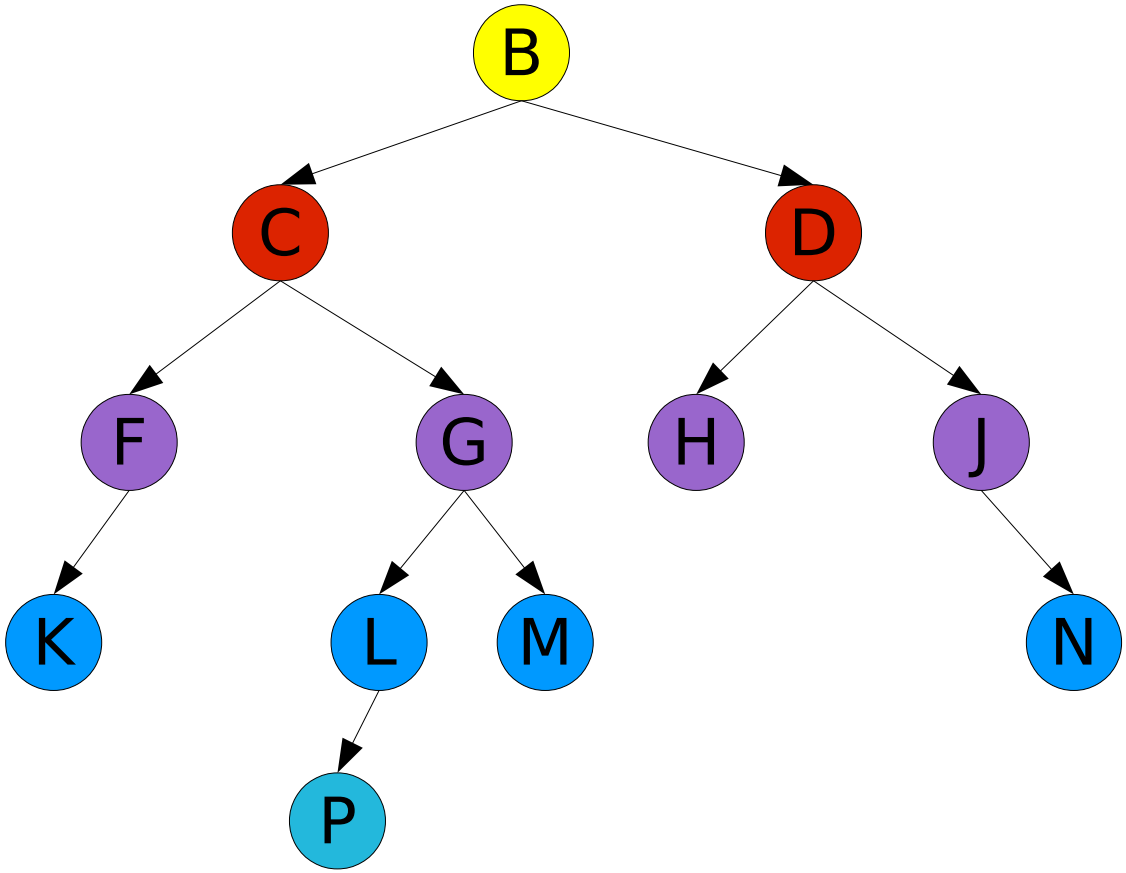


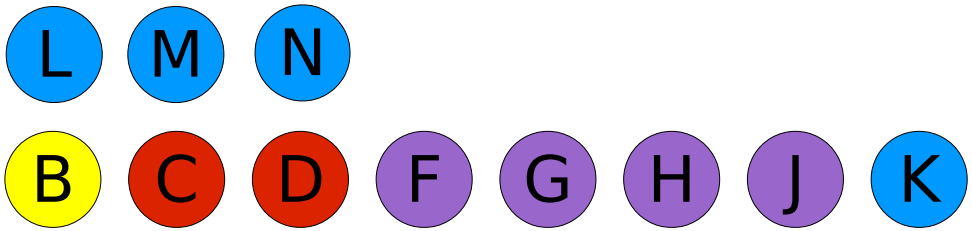
Levelorder traversal using a queue



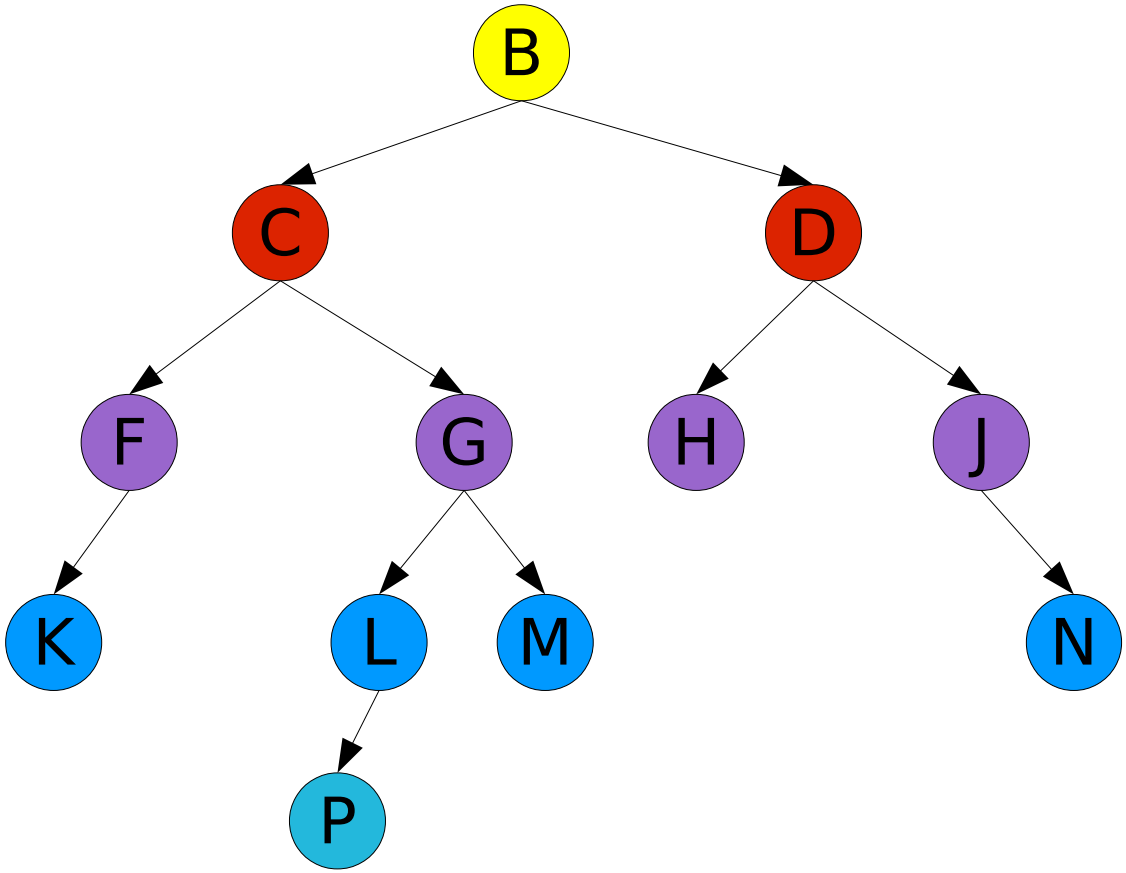


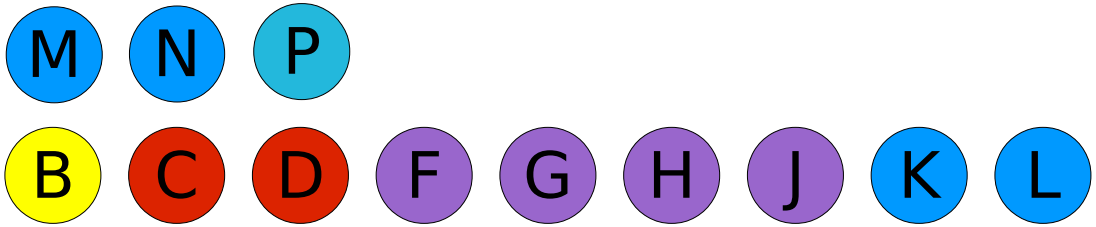
Levelorder traversal using a queue



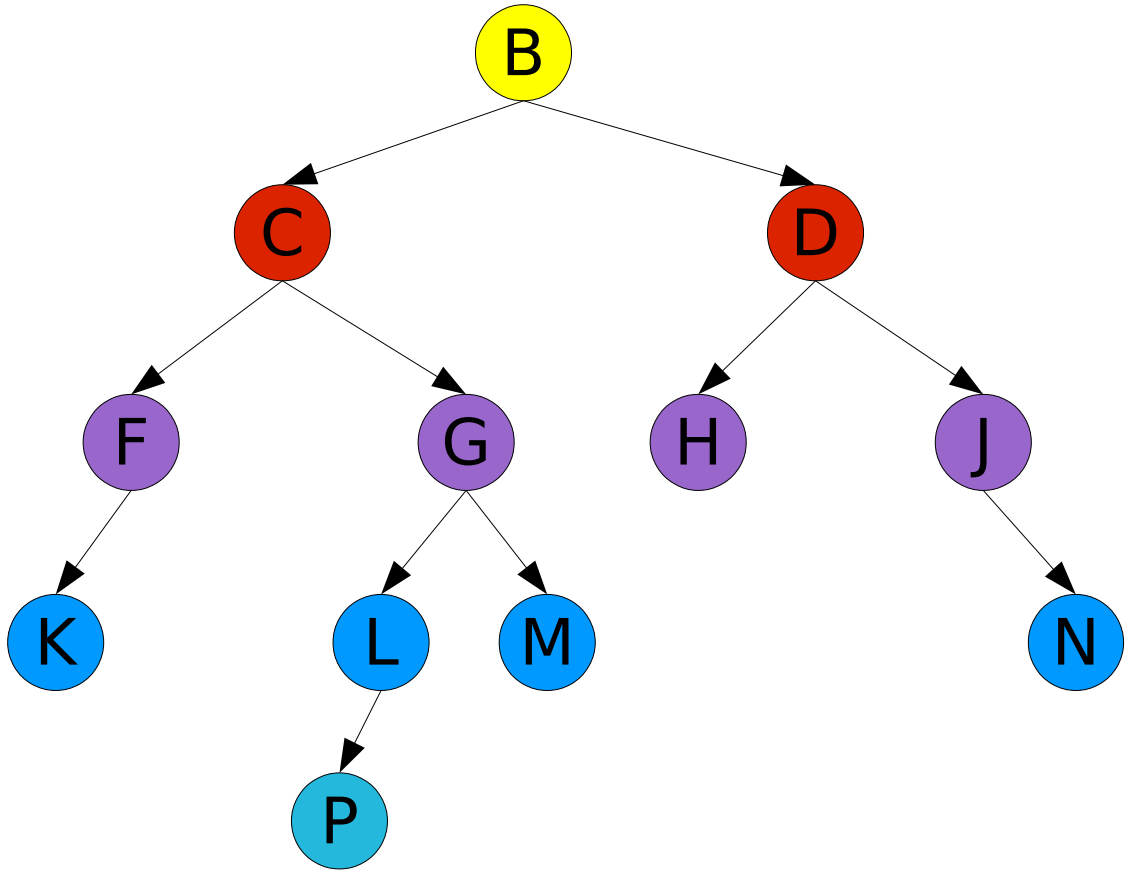


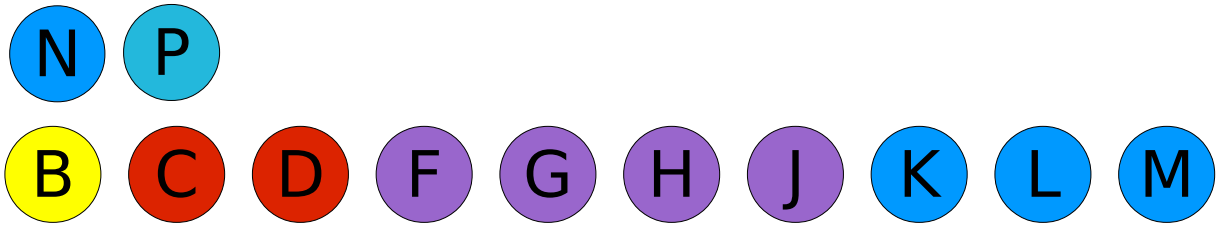
Levelorder traversal using a queue



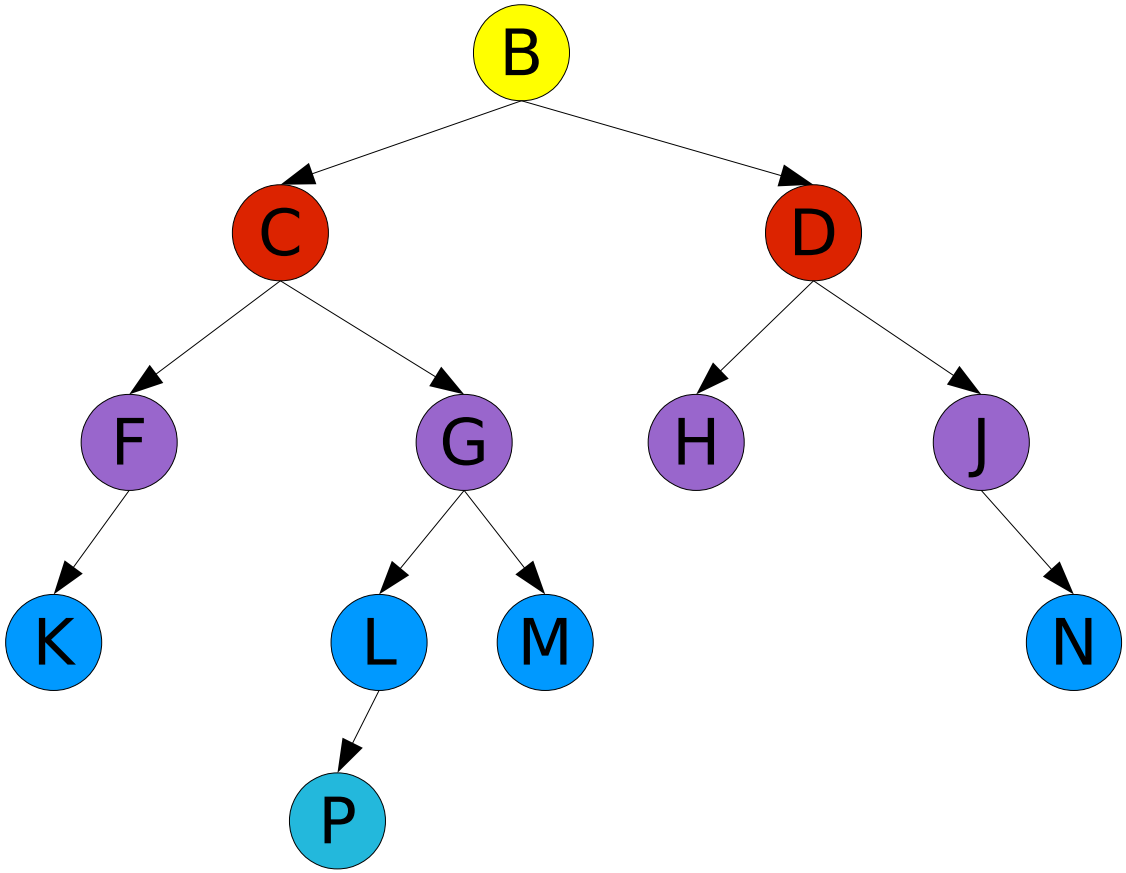


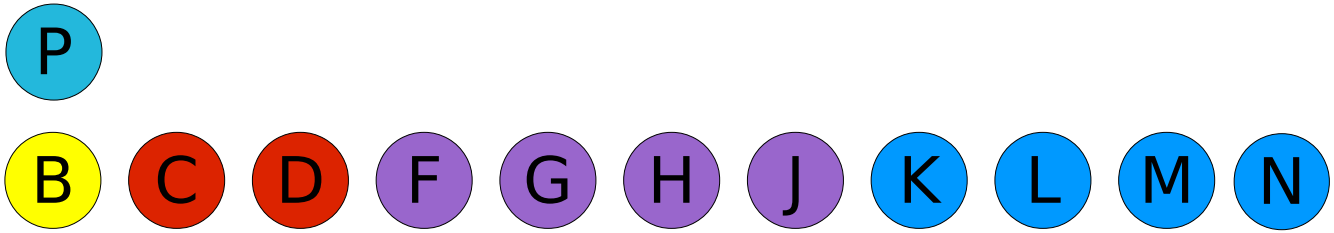
Levelorder traversal using a queue





Levelorder traversal using a queue





Levelorder traversal using a queue