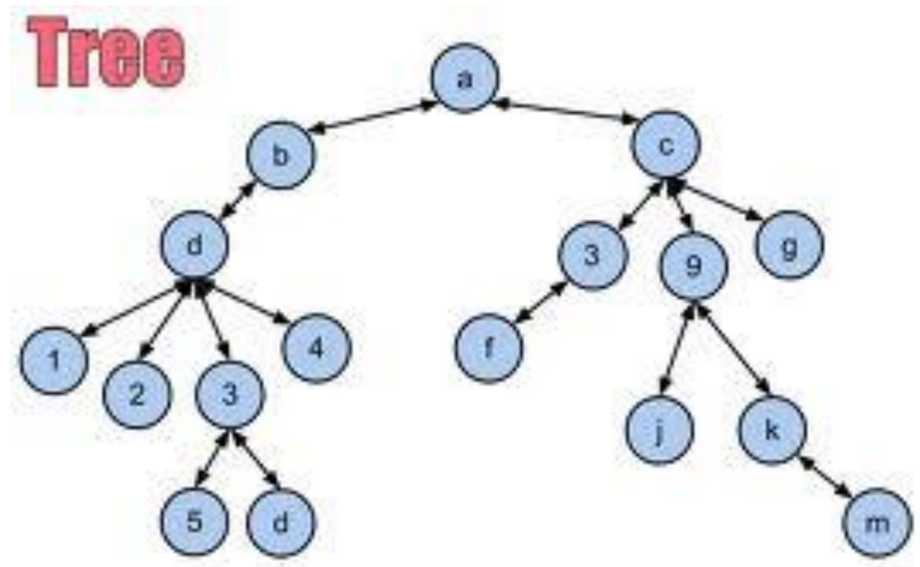
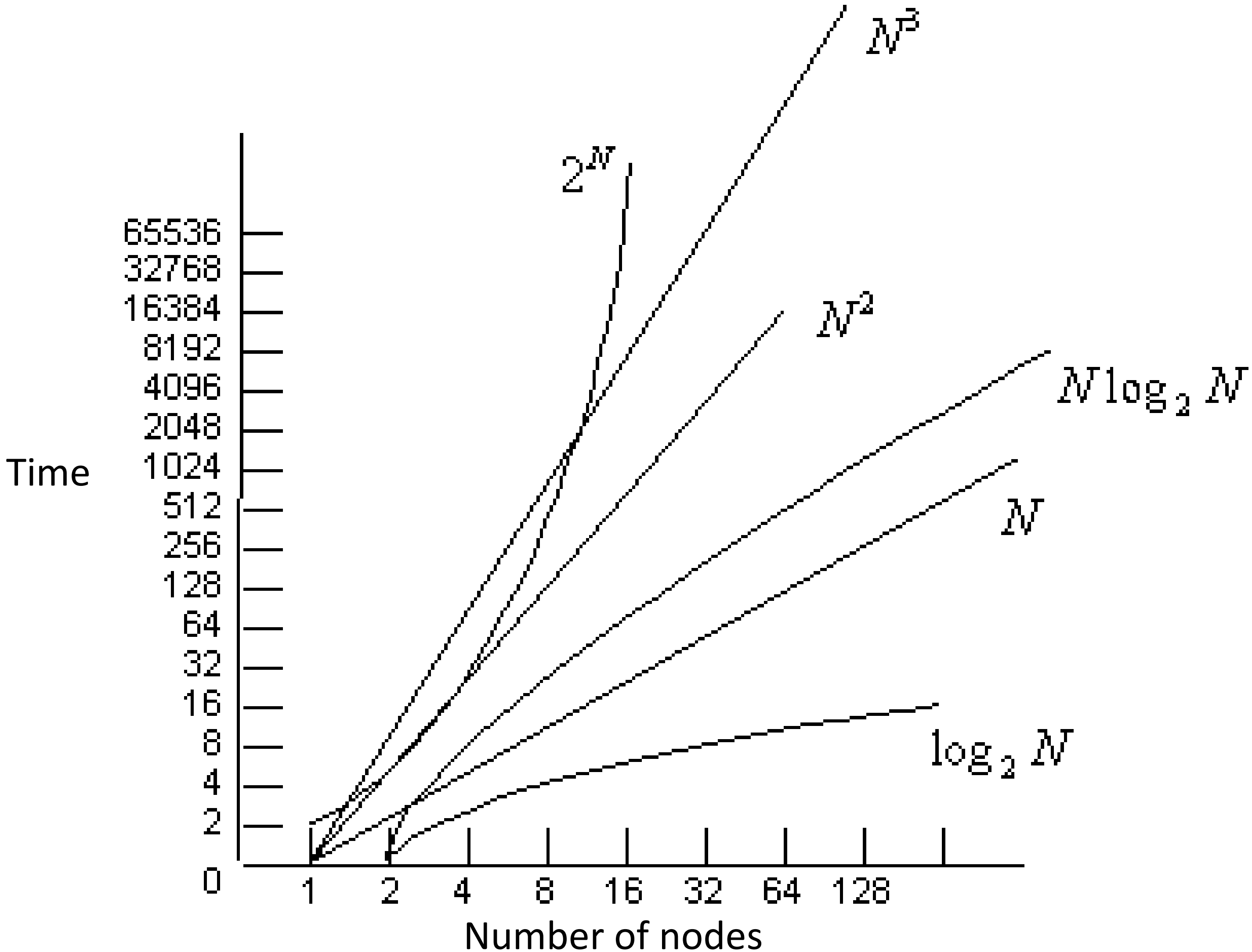
Binary Trees



Introduction

- Tress are one of the fundamental data structures in computer science.
- Trees are a specific type of graph, but simpler.
- They are constructed so as to retrieve information rapidly
- Typical search times for trees are O(log N)
  - Remember binary search on a linked list.

Recall: Common Growth Rate



Terminology

- Trees consist of vertices and edges.
- Vertex: An object that carries associated information. (node)
- Edge: A connection between two vertices. A link from one node to another.

Terminology

- Child/Parent: If either the right or left link of A is a link to B, then B is a child of A and A is a parent of B.

- Sibling: Nodes that have the same parent.

- Root: A node that has no parent. There is only one root in a tree.

Terminology

- Path: A list of vertices

- Leaf: A node with no children
  - External node, terminal node, terminal

- Non-Leaf: A node with at least one child
  - Internal node, non-terminal node, non-terminal

Terminology

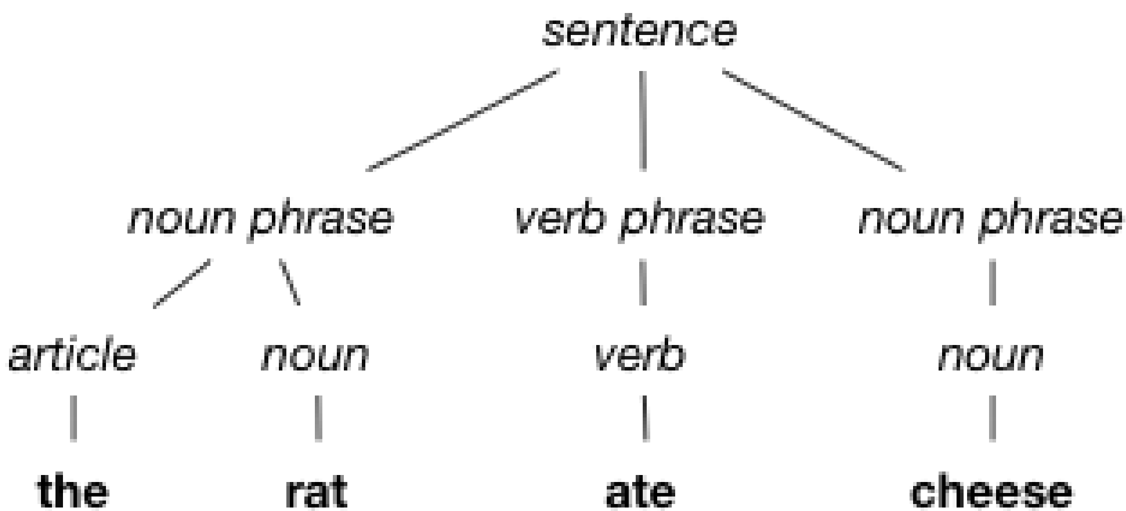
- Depth (or height): The length of the longest path from the root to a leaf.
  - The number of edges in the path is the length
  - A tree consisting of 1 node (the root) has a height of 0.

- SubTree: Any given node, with all of its descendants (children).

Terminology

- Trees can be ordered or unordered.
  - Ordered trees specify the order of the children (example parse tree).
  - Unordered trees place no criteria on the ordering of the children (example: file system directories).

Parse Tree


sentence
noun phrase
verb phrase
noun phrase
article
noun
verb
noun
the
rat
ate
cheese

Terminology

- M-ary tree: A tree which must have specific number of children (M) in a specific order.

- Binary tree – An M-ary tree where:
  - All internal nodes have at most two children.
  - All external nodes (leaves) have no children.
  - The two children are called the left child and right child.

Basic Properties

- A node has at most one edge leading to it.
  - Each node has exactly one parent, except the root which has no parent.
- There is at most one path from one node to any other node.
  - If there are multiple paths, there will be cycles. It's a graph and not a tree.
- There is exactly one path from the root to any leaf

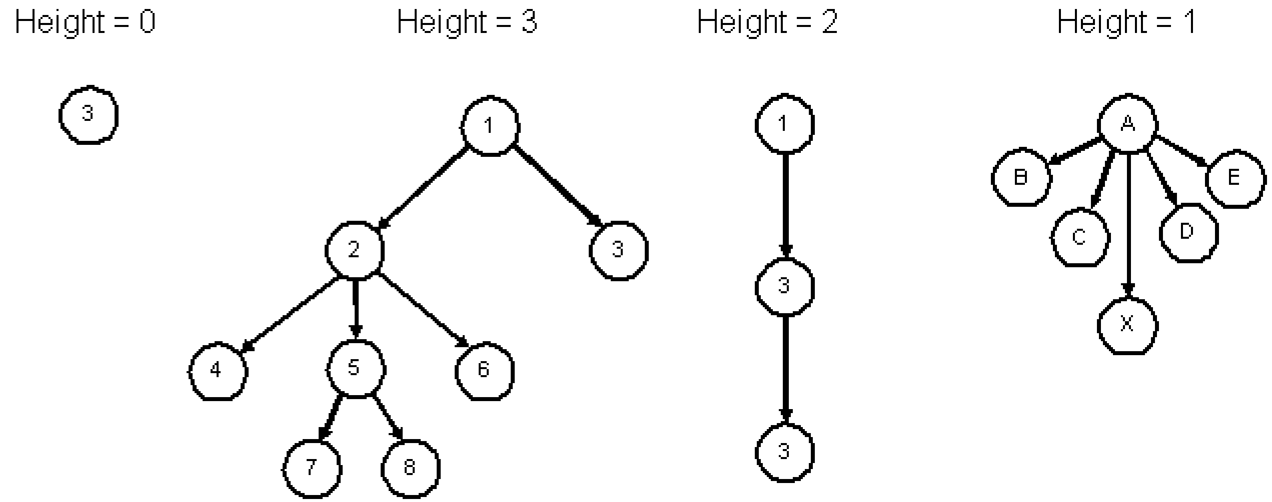
Other Properties

- The level of a given node in a tree is defined recursively as:

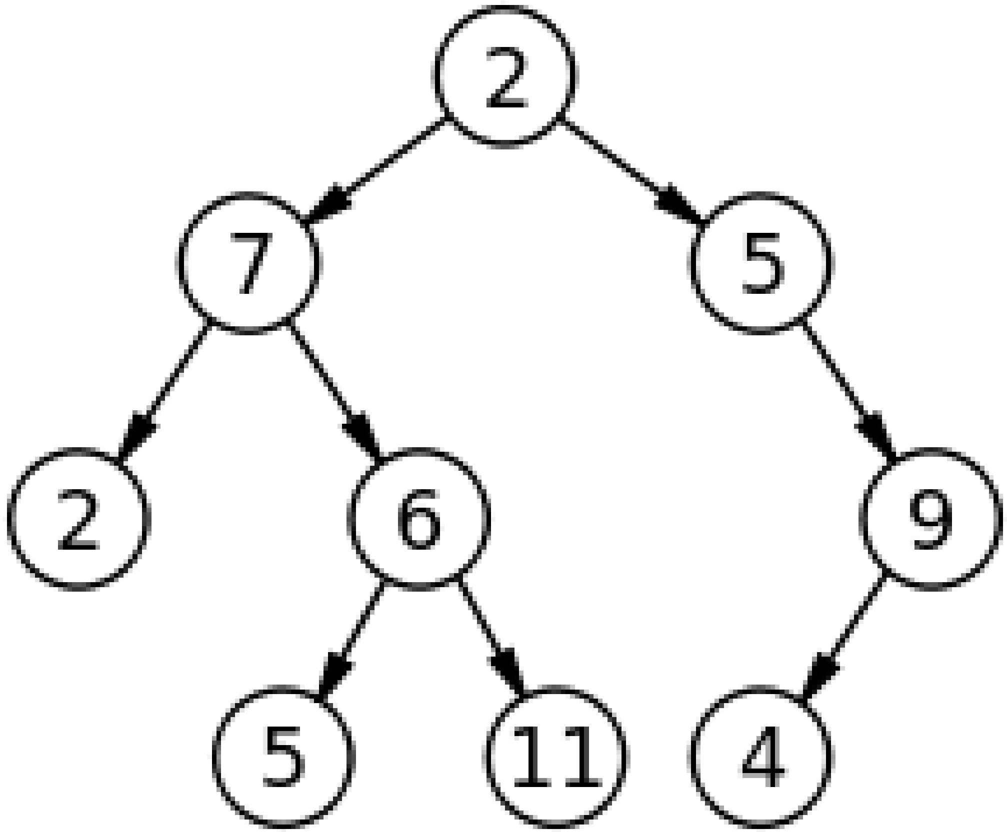
- Level = **0,** if node is a root.
- Level = (**Level(parent) + 1),** if node is a child of parent.

Two Interpretations of Height (Depth)

- The height (depth) of a tree is the length of the longest path from the root to a leaf
- The height (depth) is the maximum of the levels of the tree's nodes

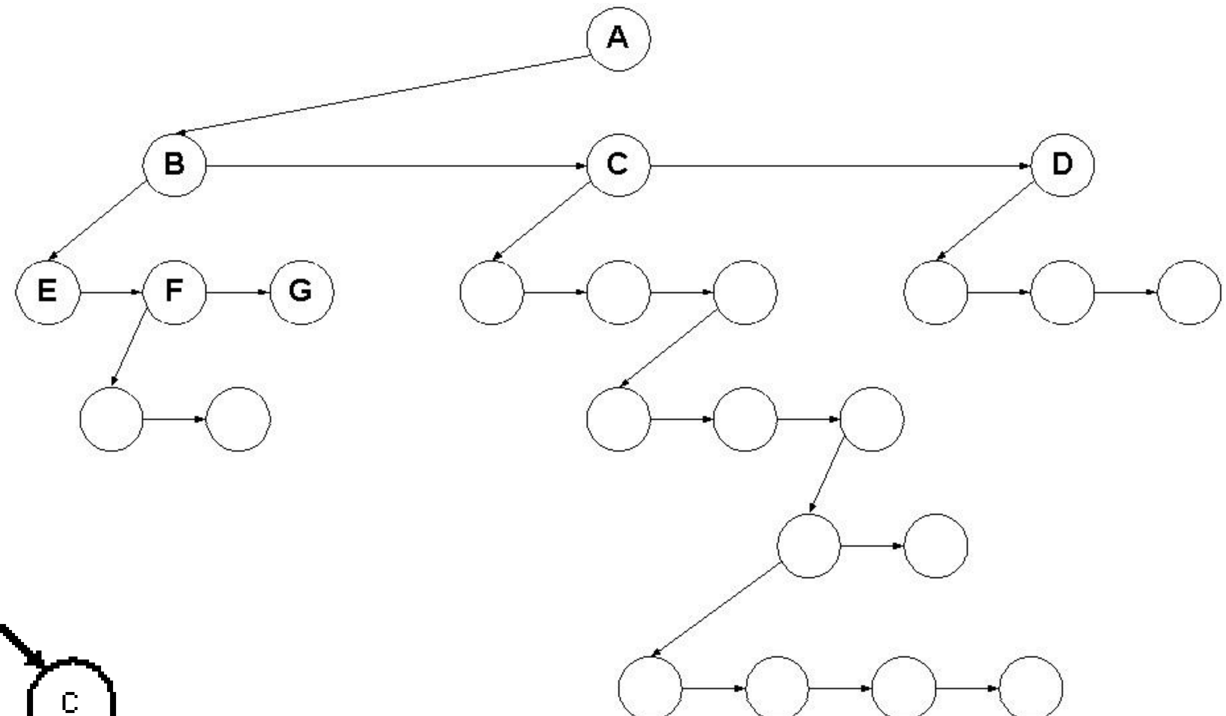


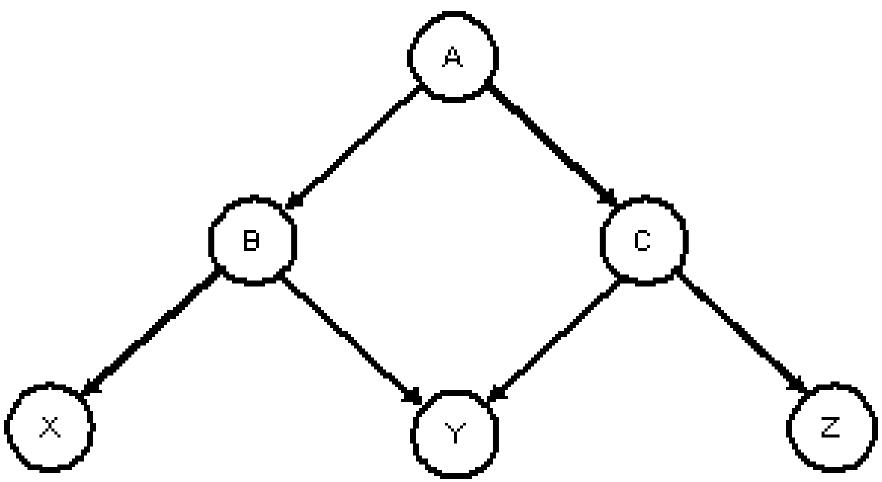
Self Check



Self Check

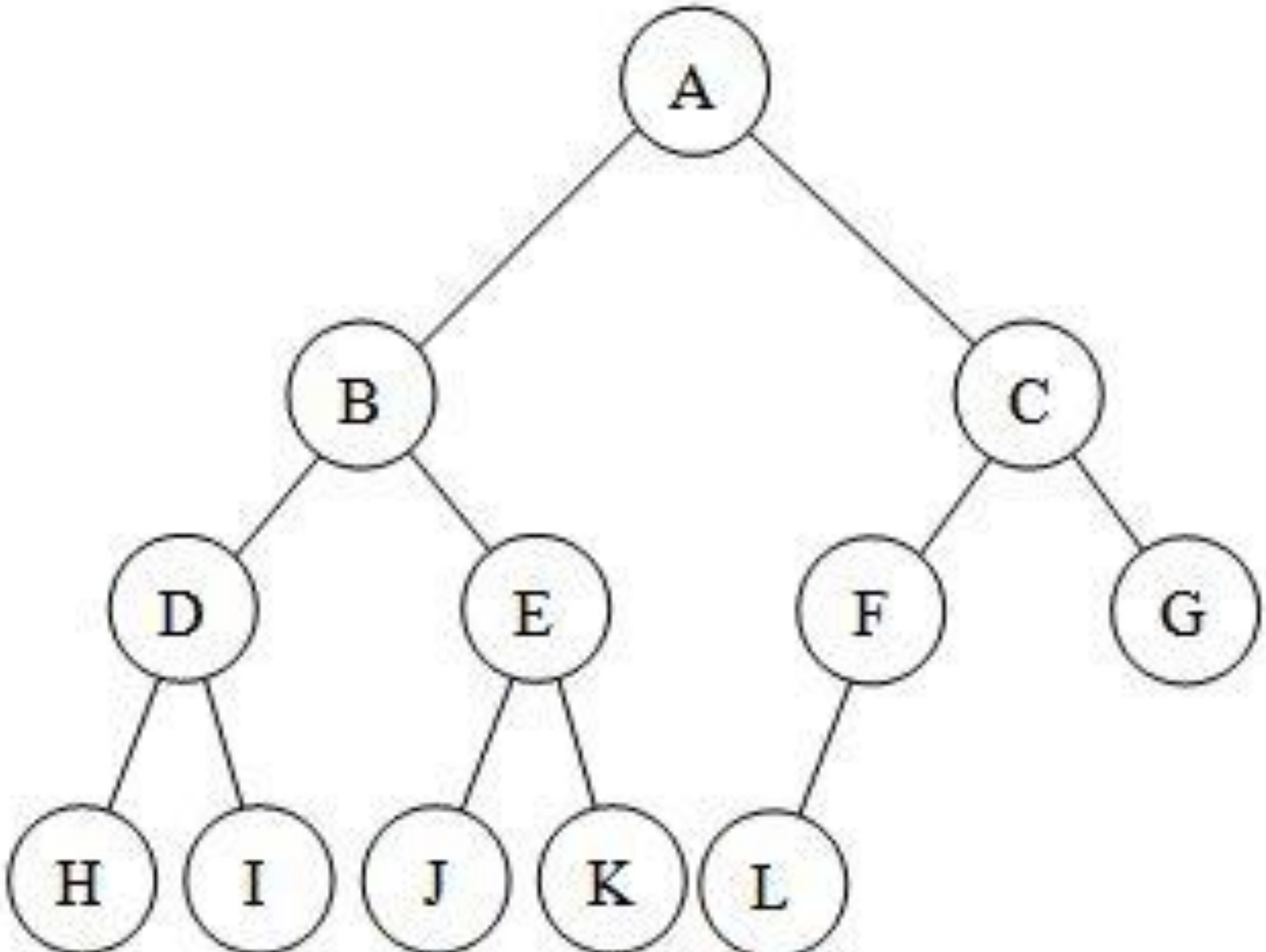
- Are these trees?





Binary Trees

Binary Trees



Properties of Binary Trees

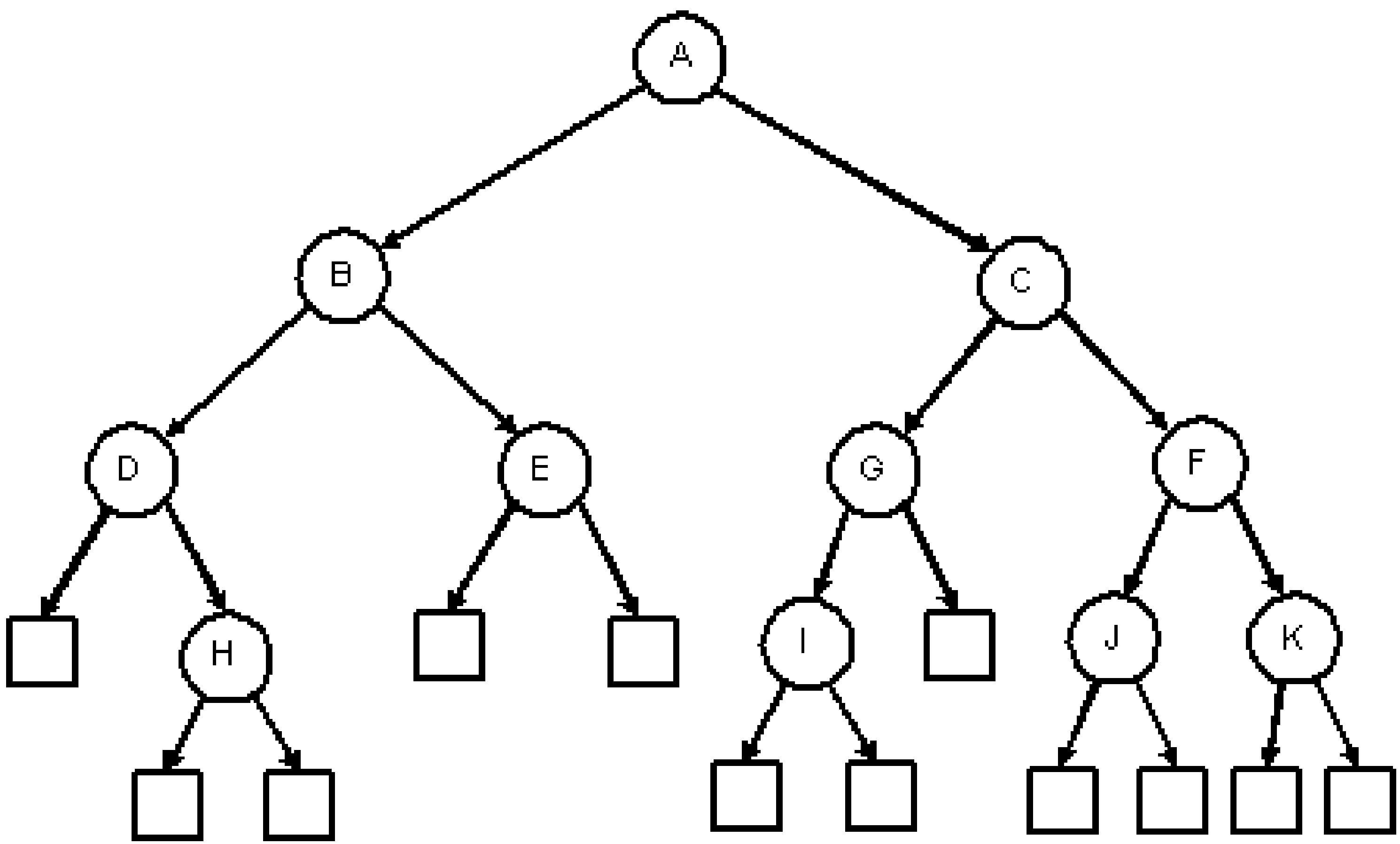
- There are two distinct types of nodes: internal and external.
- An internal node contains two links, left and right.
- One or both links can be NULL (an empty tree)

Properties of Binary Trees

- A binary tree with N internal nodes has N+1 external nodes. (some may be empty/NULL)

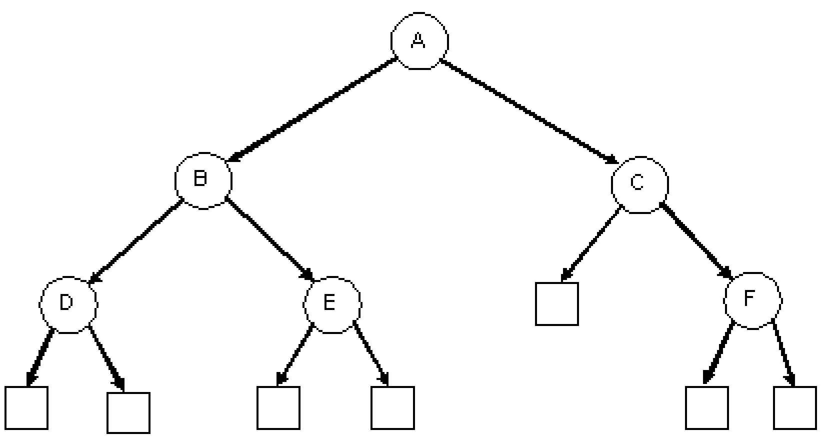
- A binary tree with N internal nodes has 2N links.

Properties of Binary Trees

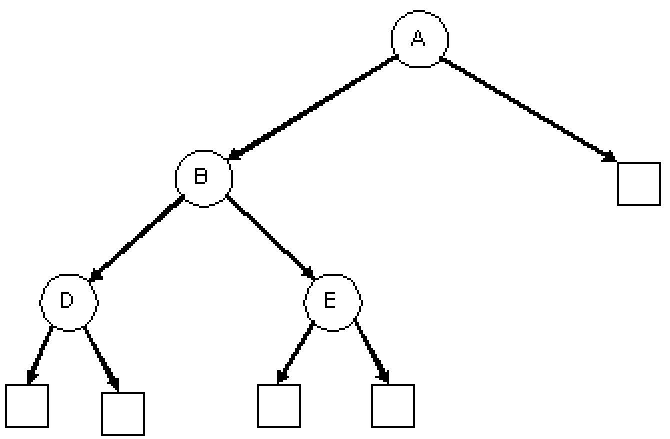


Properties of Binary Trees

- A balanced binary tree (height-balanced) is a tree where for each node the depth of the left and right subtrees differ by no more than 1.



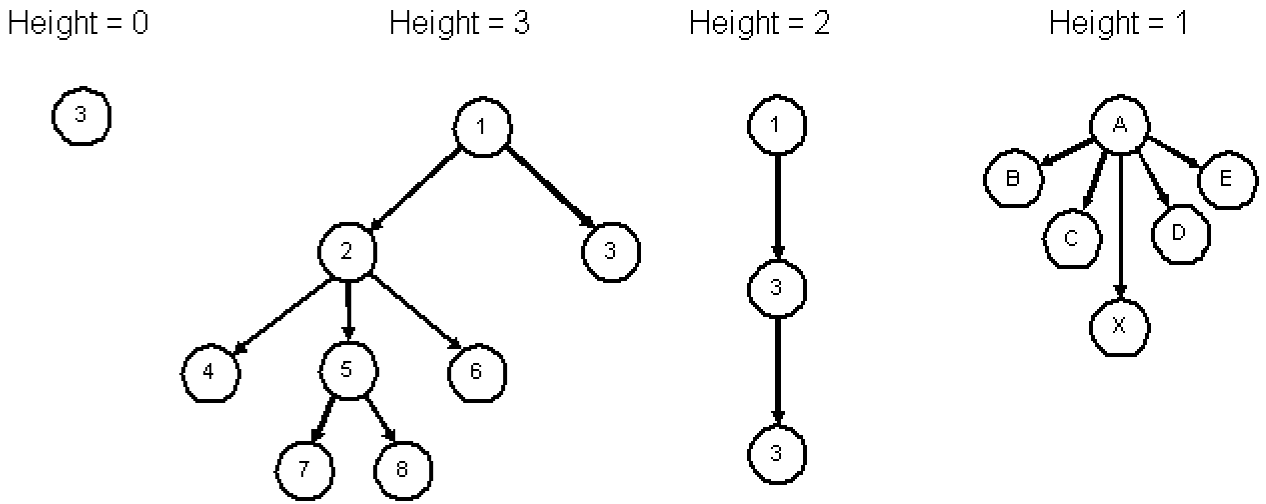
A balanced binary tree



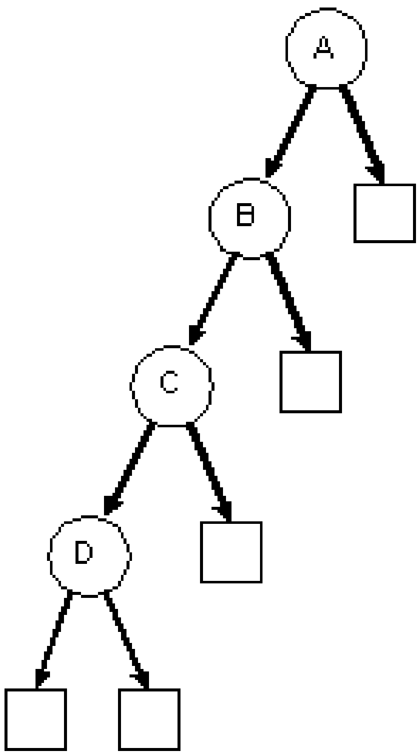
An unbalanced binary tree

Recall: Depth (or Height)

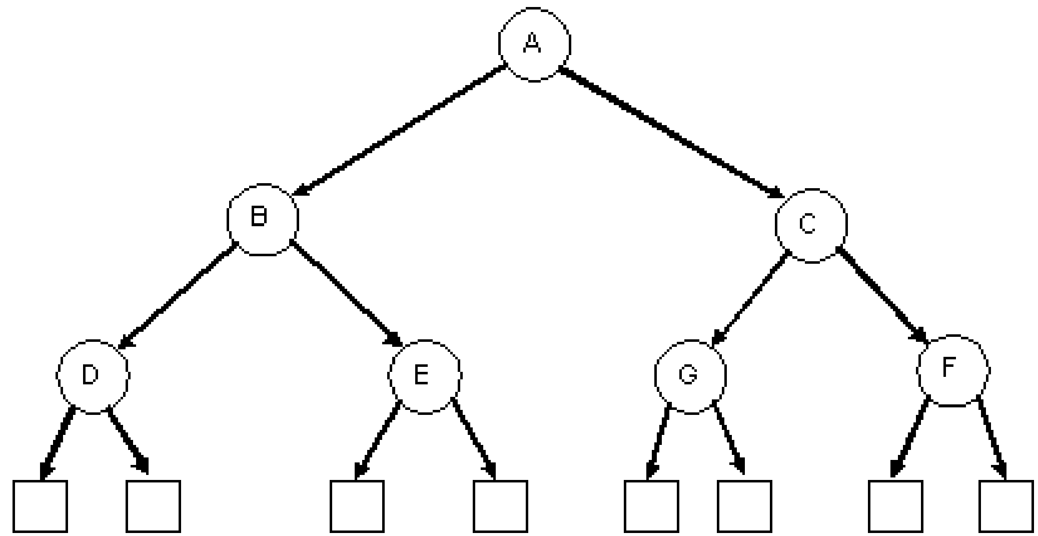
- The length of the longest path from the root to a leaf.
  - The number of edges in the path is the length
  - A tree consisting of 1 node (the root) has a height of 0.



Properties of Binary Trees



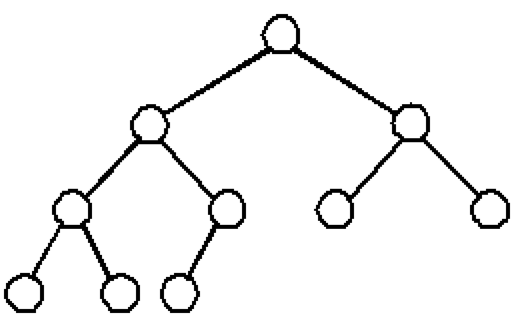
A degenerate binary tree



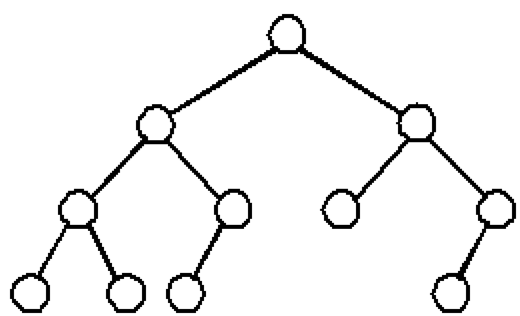
A balanced binary tree
(it’s also a full binary tree)

Properties of Binary Trees

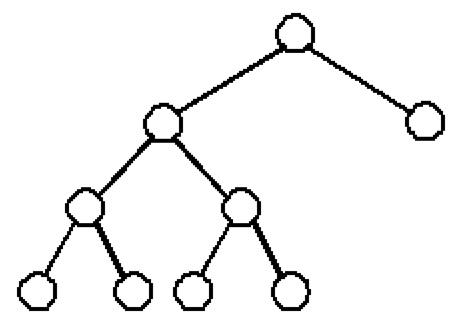
- A complete binary tree is similar to a balanced binary tree except that all of the leaves must be placed as far to the left as possible.
  - The leaves must be filled in from left to right, one level at a time.



A complete binary tree



An incomplete binary tree



An incomplete binary tree

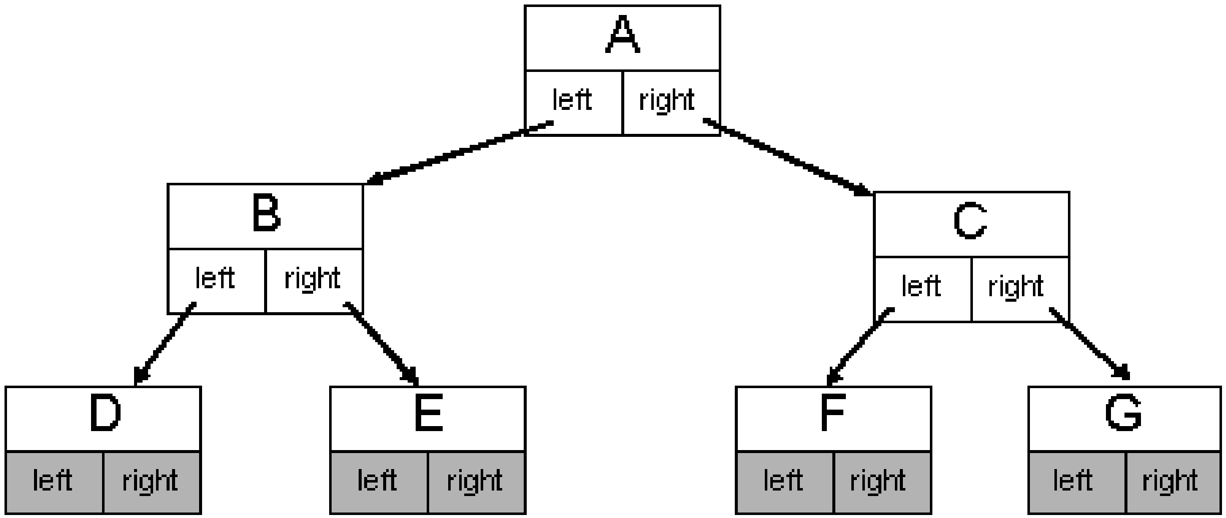
Trees v.s. Linked List

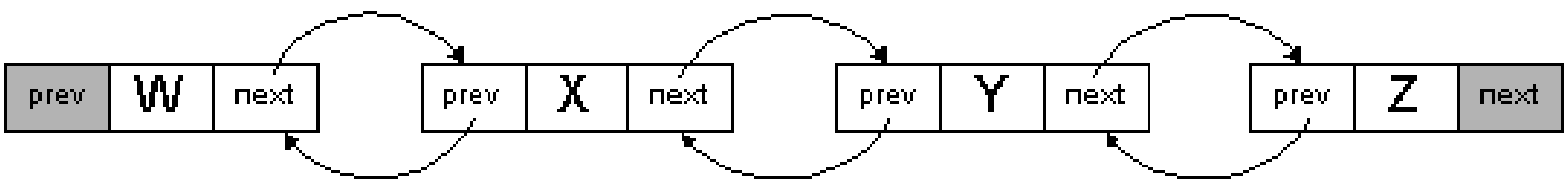
```
struct ListNode
{
  ListNode *next;
  ListNode *prev;
  Data *data;
};


struct TreeNode
{
  TreeNode *left;
  TreeNode *right;
  Data *data;
};
```

- The two links in a binary tree are not quite the same as the two links in a doubly linked list
  - Trees have left and right link.
  - Lists have previous and next link.
  - Both imply ordering, but a different kind of ordering.

Trees v.s. Linked List

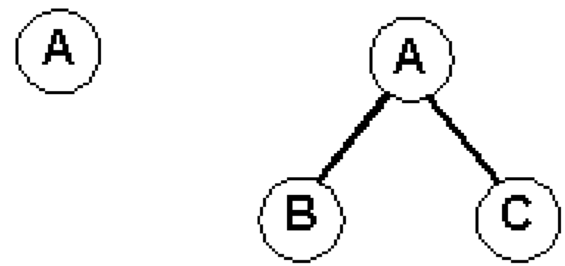


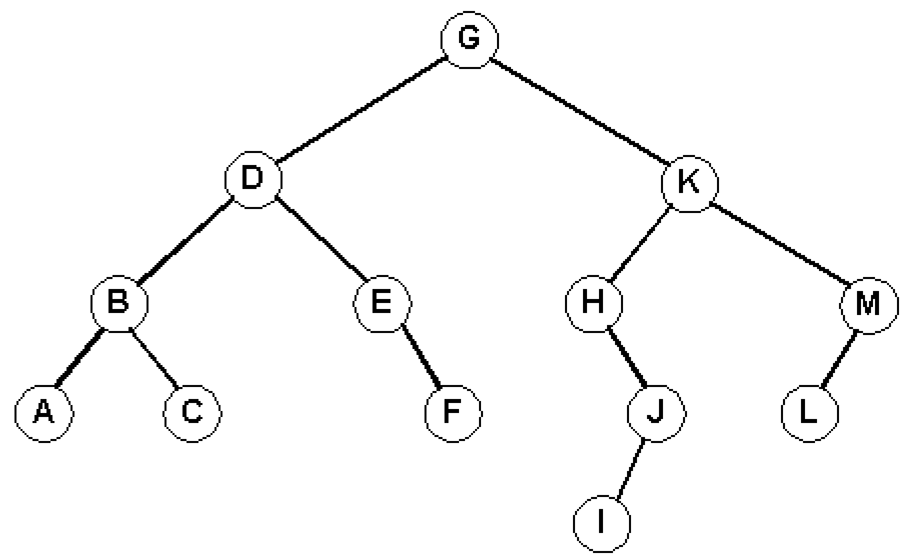


Binary Tree Traversal

Traversal Order

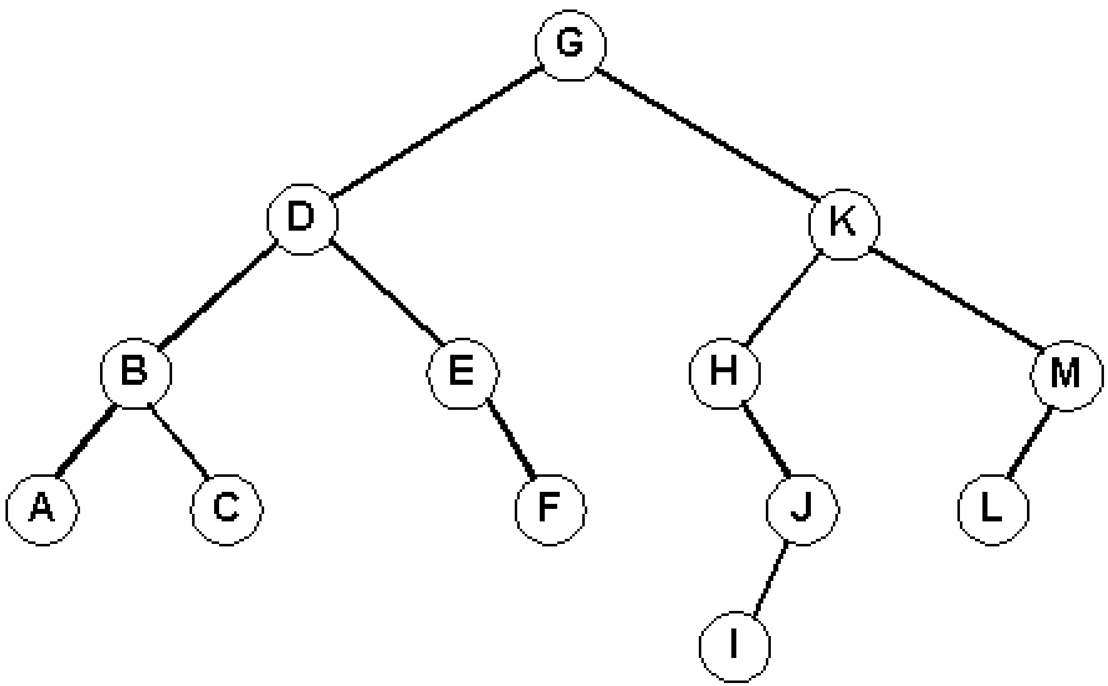
- Preorder traversal
  1. **Visit the node**
  2. Traverse the left subtree.
  3. Traverse the right subtree.
- Inorder traversal
  1. Traverse the left subtree.
  2. **Visit the node**
  3. Traverse the right subtree.
- Postorder traversal
  1. Traverse the left subtree.
  2. Traverse the right subtree.
  3. **Visit the node**





Traversal Order

- Pre-order traversal: GDBACEFKHJIML
- In-order traversal: ABCDEFGHIJKLM
- Post-order traversal: ACBFEDIJHLMKG



Traversing Binary Trees

- Binary trees are inherently recursive data structures.

- Recursive algorithms are quite appropriate.
  - In some cases, iterative algorithms can be significantly more complicated.

Tree Algorithms Implementation

Implementing 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 the Number of Nodes

- State the algorithm in English:
  - If the tree is empty: 0
  - If the tree is not empty: 1 + (nodes in left subtree) + (nodes in right subtree)

```
int NodeCount(Tree tree){
    if (tree == 0)
        return 0;
    else
        return 1 + NodeCount(tree->left) + NodeCount(tree->right);
}
```

Find the Height of a Tree

- State the algorithm in English:
  - If the tree is empty: -1
  - If the tree is not empty: 1 +max((height of left subtree),(height of right subtree))

```
int Height(Tree tree){
    if (tree == 0)
        return -1;
    if (Height(tree->left) > Height(tree->right))
        return Height(tree->left) + 1;
    else
        return Height(tree->right) + 1;
}
```

Better Implementation

```
int Height(Tree tree){
    if (tree == 0)
        return -1;
    int hl=Height(tree->left);
    int hr=Height(tree->right);
    if (hl > hr)
        return hl + 1;
    else
        return hr + 1;
}
```

Implement Tree Traversal Algorithms

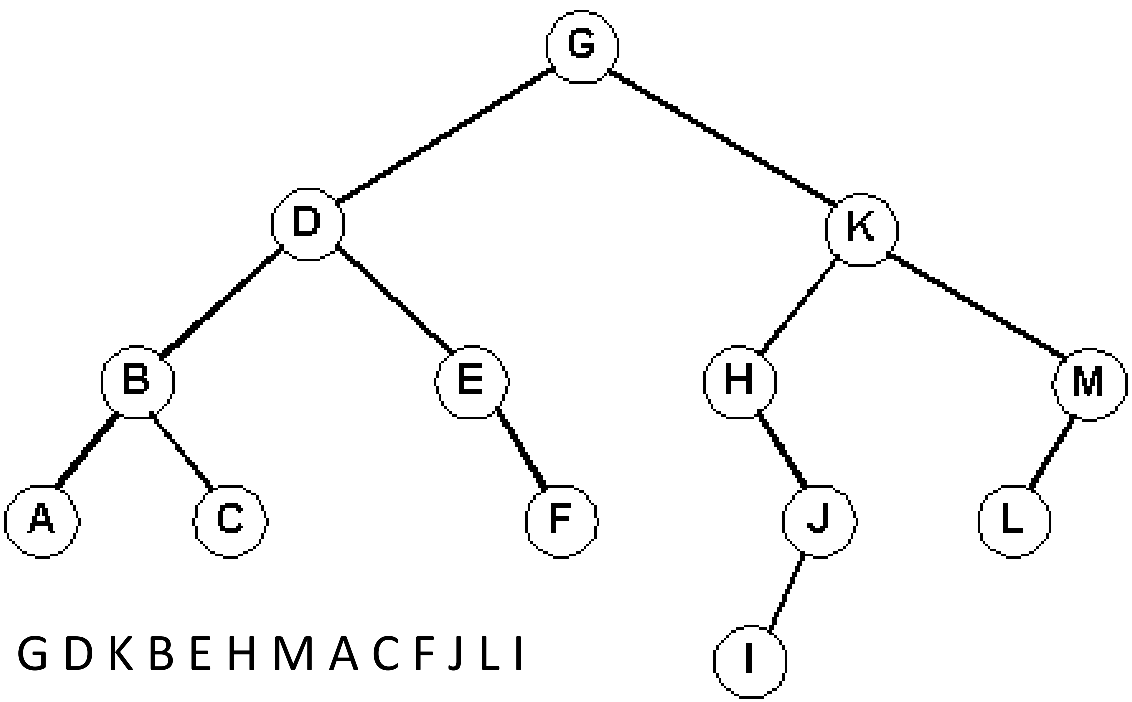
```
void TraversePreOrder(Tree tree){
  if (tree == 0)
    return;
  else{
    VisitNode(tree);
    TraversePreOrder(tree->left);
    TraversePreOrder(tree->right);
  }
}
```

```
void TraversePostOrder(Tree tree){
  if (tree == 0)
    return;
  else{
    TraversePostOrder(tree->left);
    TraversePostOrder(tree->right);
    VisitNode(tree);
  }
}
```

```
void TraverseInOrder(Tree tree){
  if (tree == 0)
    return;
  else{
    TraverseInOrder(tree->left);
    VisitNode(tree);
    TraverseInOrder(tree->right);
  }
}
```

Level-Order Traversal

- Traversing all nodes on level 0 from left to right, then all nodes on level 1 (left to right), then nodes on level 2(left to right), etc…



G D K B E H M A C F J L I

Level-Order Traversal

- State the algorithm in English:
  - If the level to visit = 0, visit the node
  - If the level to visit > 0, traverse the left subtree, traverse the right subtree

```
void TraverseLevelOrder(Tree tree){
     int height = Height(tree);
     for (int i = 0; i <= height; ++i)
          TraverseLevelOrder2(tree, i);
}

void TraverseLevelOrder2(Tree tree, int level){
     if (level == 0)
          VisitNode(tree);
     else {
          // visit the subtrees…
          TraverseLevelOrder2(tree->left, level - 1);
          TraverseLevelOrder2(tree->right, level - 1);
     }
}
```

Level-Order Traversal Using a Queue

1. If the tree isn't empty
    1. Push the node onto the Queue
    2. While the Queue isn't empty
        1. Pop a node from the Queue
        2. Visit the node
        3. If the node's left child is not NULL
            1. Push the left child onto the Queue
        4. If the node's right child is not NULL
            1. Push the right child onto the Queue
    3. End While
2. End If

Level-Order Traversal

- **EXERCISE:** Modify the algorithm above so it prints the nodes in reverse level-order: **I L J F C A M H E B K D G**