CS280 - Data Structures

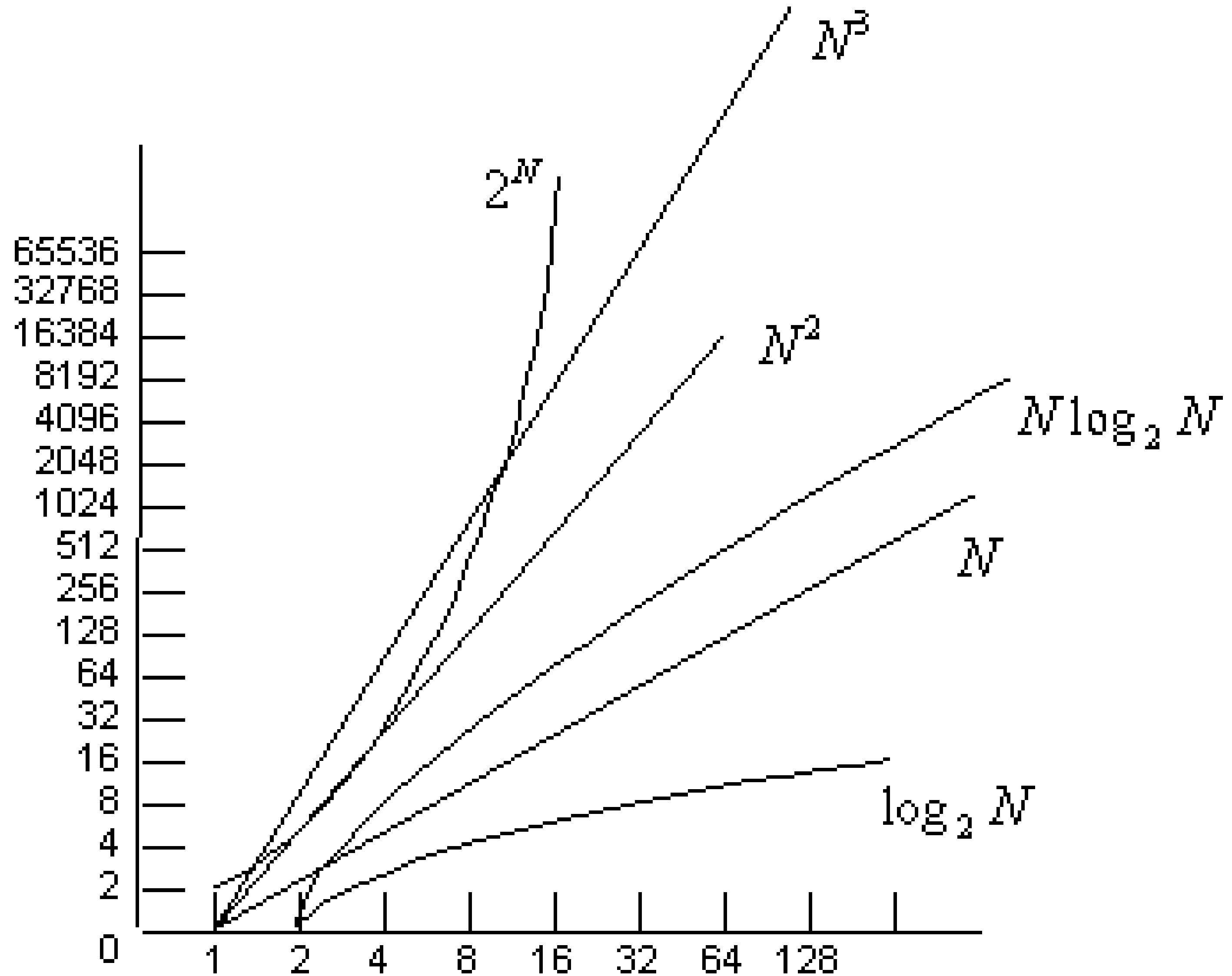
Trees - part I

Overview

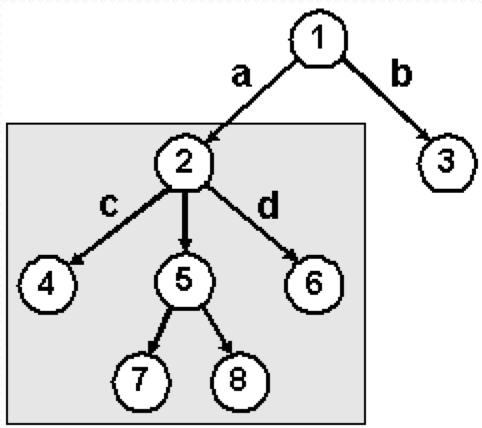
- Introduction to Trees
- Terminology
- Basic Properties
- Binary Trees
  - Basic Properties.
  - Traversing Binary Trees.
  - Implementing Tree Algorithms.
  - More Tree Algorithms
  - Level Order Traversal

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 tress are **O(lgN)**
  - Remember binary search on a linked list.


N^3
2^N
N^2
$N \log_2 N$
N
$\log_2 N$
65536
32768
16384
8192
4096
2048
1024
512
256
128
64
32
16
8
4
2
0
1
2
4
8
16
32
64
128

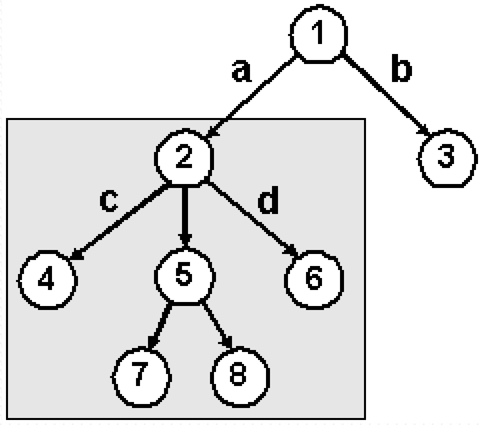
Terminology



A generic Tree

- Trees consist of *vertices* and *edges*
- *Vertex*: An object that carries associated information[1,2,3].
  - In other word, a *node*.
- *Edge* – A connection between two vertices. A *link* to from one node to another [a,b,c].

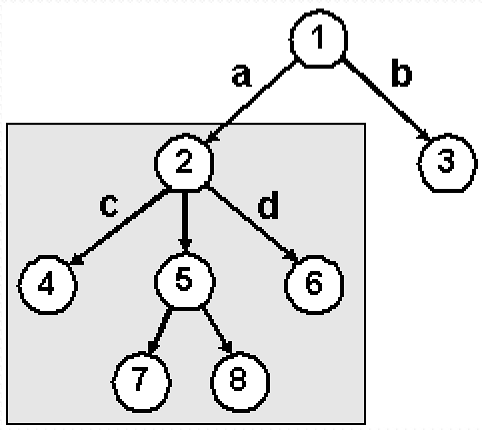
Terminology



A generic Tree

- *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**.
  - 4,5, and 6 are children of 2; 2 is the parent of 4,5, and 6
- *Sibling*: Nodes that have the same parent. [2,3] have the same parent.
- *Root*: A node that has no parent[1]. Ther is only one root in any given tree.

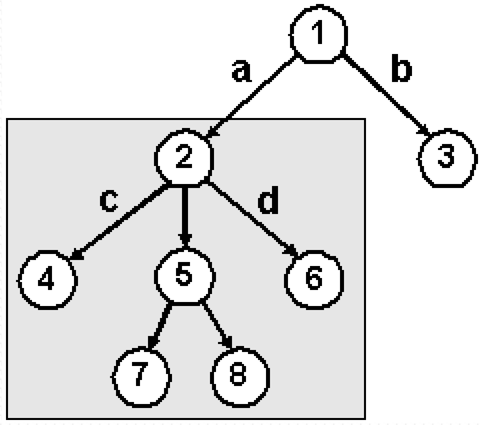
Terminology



A generic Tree

- *Path*: Al list of vertices [1-2-4]
- *Leaf*: A node with no children [4, 7]
  - *External node*
  - *Terminal node*
  - *Terminal*
- *Non-Leaf*: A node with at least one child [1,2,5[
  - *Internal node*
  - *Non-terminal node*
  - *Non-terminal*

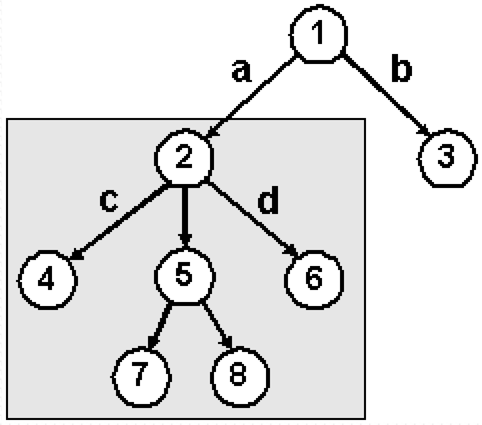
Terminology



A generic Tree

- *Depth(or height)*: The length of the longest path from the root to a leaf. [1,2,5,7] = 3
  - *The number of edges in the path is the length*
  - *A tree consisting of 1 node (the root) has a height of* ***<u>0</u>***.
- *SubTree*: Any given node, with all of its descendants (children). [5,7,8] is a subtree, 5 is the root.

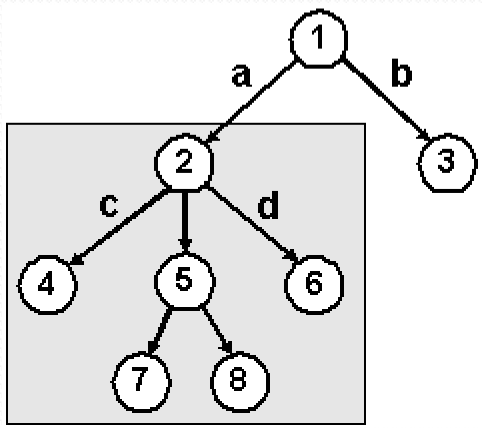
Terminology



A generic Tree

- *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).

Terminology



A generic Tree

- *M-ary tree:* A tree which must have specific number of children in a specific order.
  - Binary tree – An M-ary tree where:
    - All internal nodes have one or two children
    - All external nodes (leaves) have no children
    - The two childrent are sorted and 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 on path from one node to any other node.
  - If there are multiple paths, it's a graph and not a tree.
- There is *exactly* one path from the root to any leaf

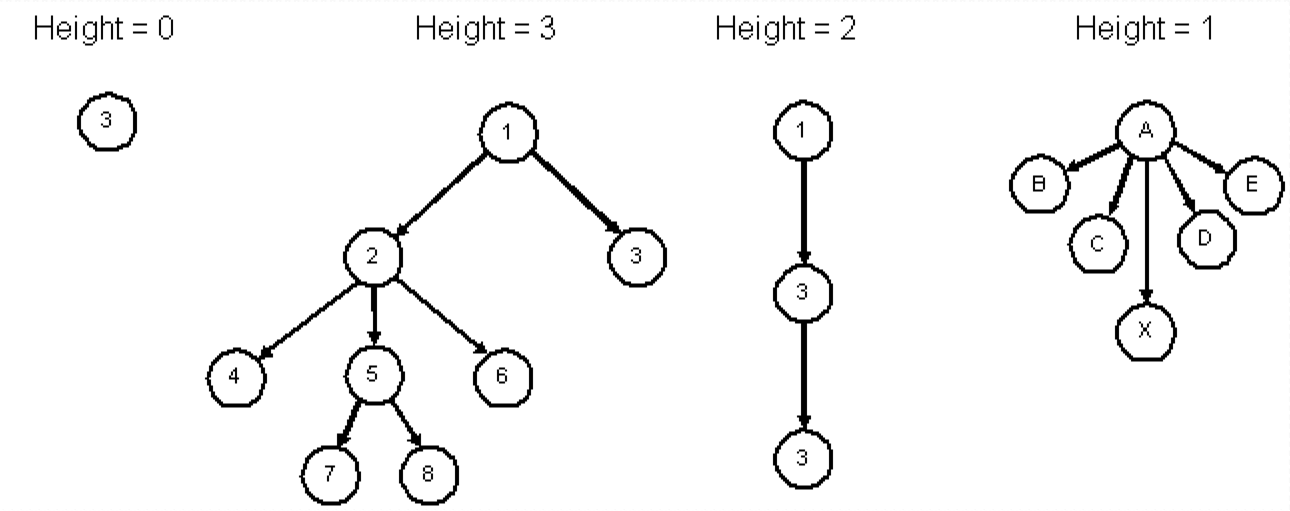
Other Properties

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

0	if node is a root
Level(parent) + 1	if node is a child of parent.

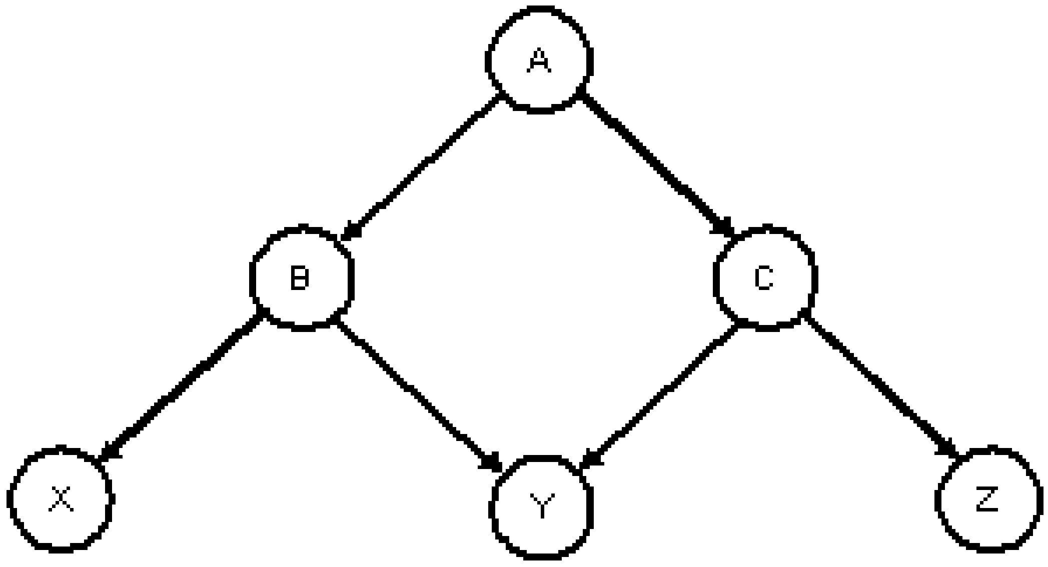
Two interpretations of HEIGHT

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



SELF CHECK

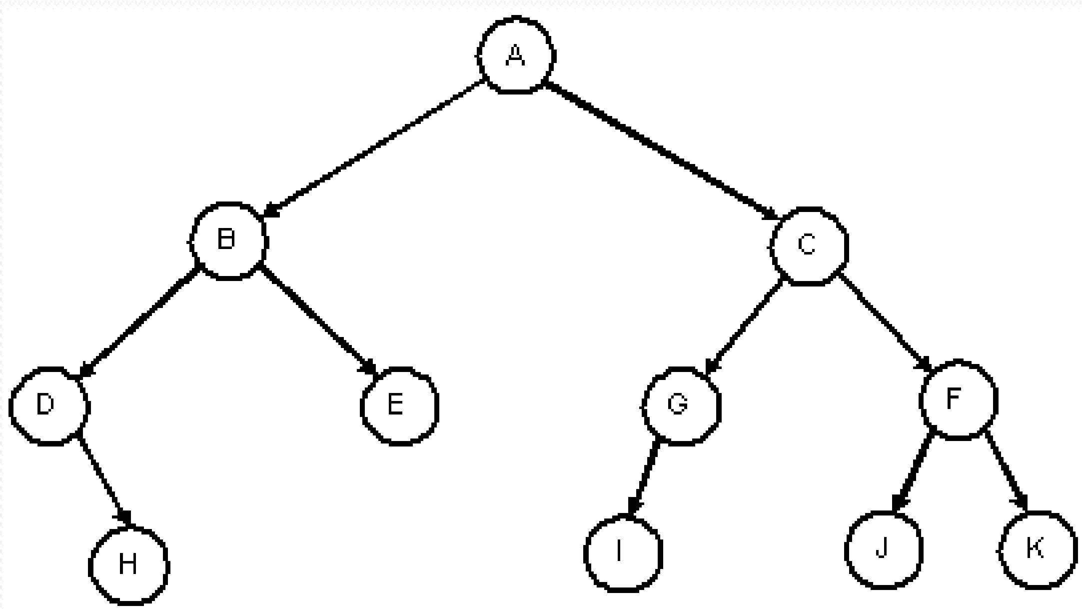
- This is not a tree. Why?



Binary Trees

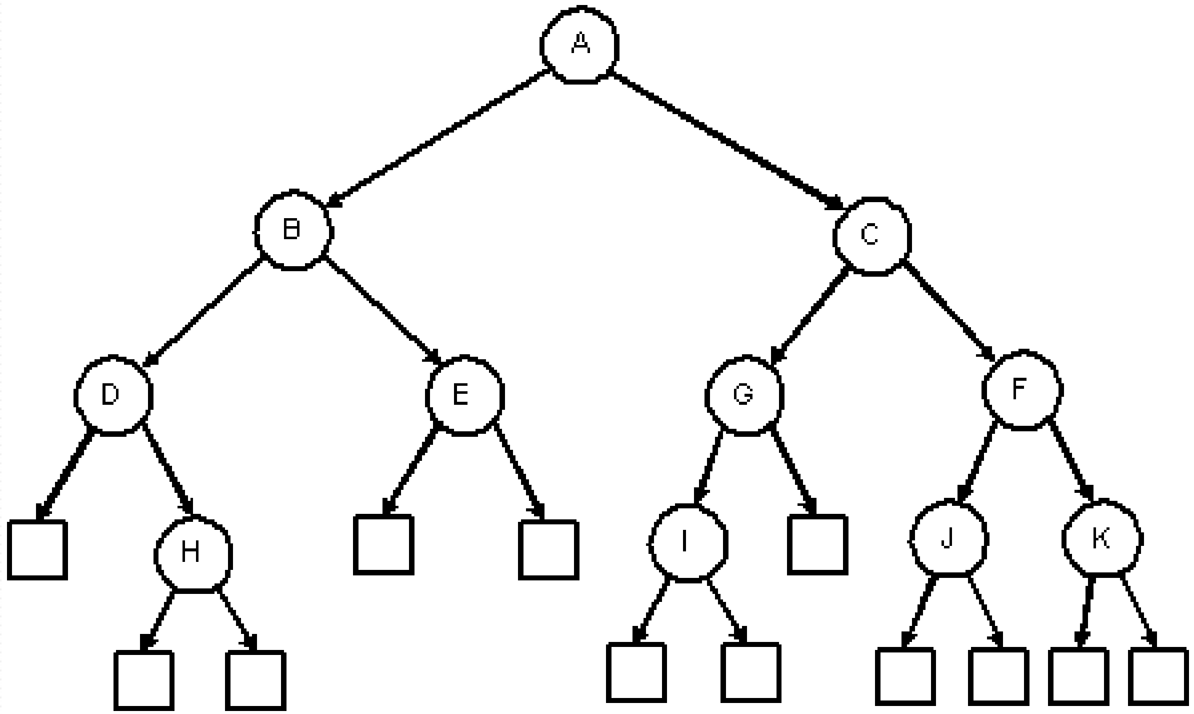
Properties of Binary Trees

- There are two distinct types of nodes: *internal* and *external*.
- An internal node contains *exactly* two links, *left* and *right*



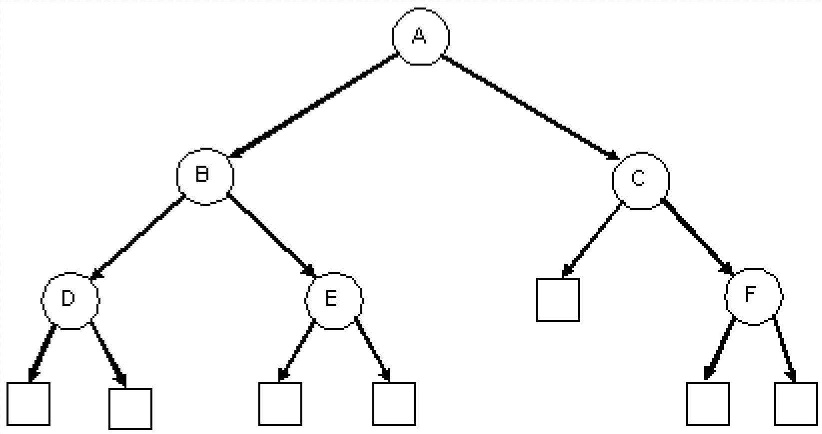
Properties of Binary Trees

- The two links are *disjoint* binary trees (they have no nodes in common)
- 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.

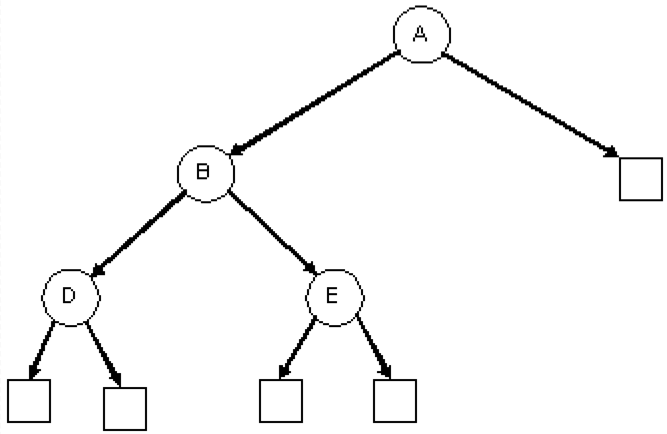


Properties of Binary Trees

- 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.

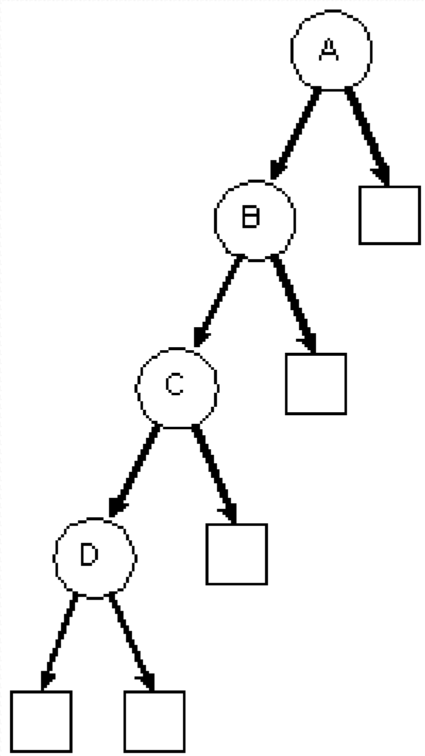


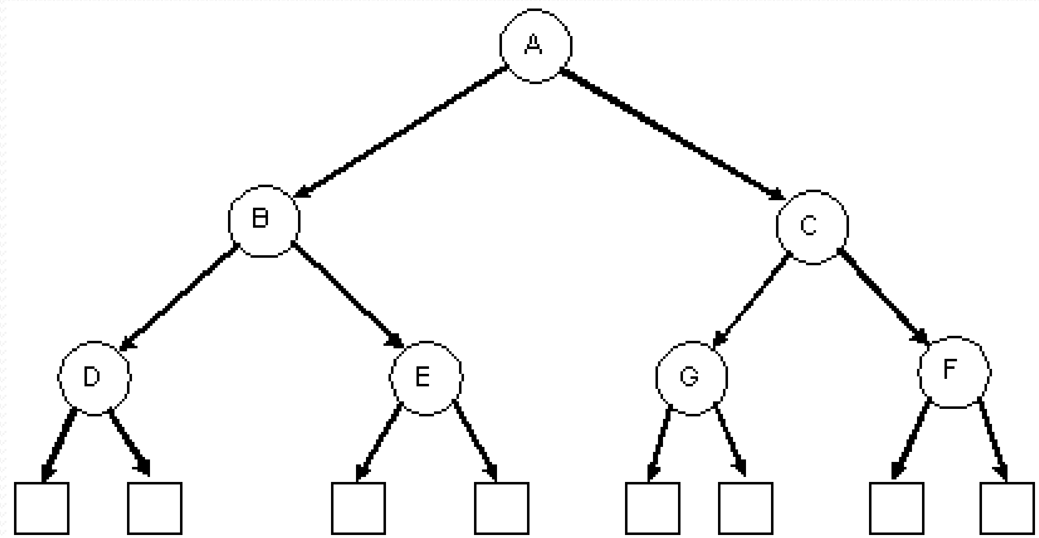
A balanced binary tree



An unbalanced binary tree

A degenerate binary tree

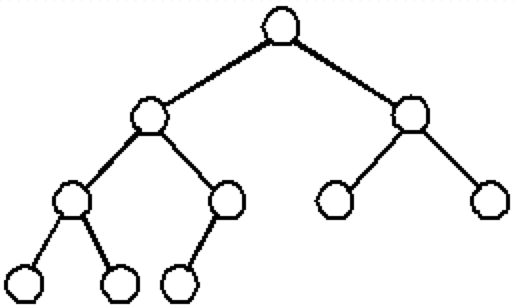




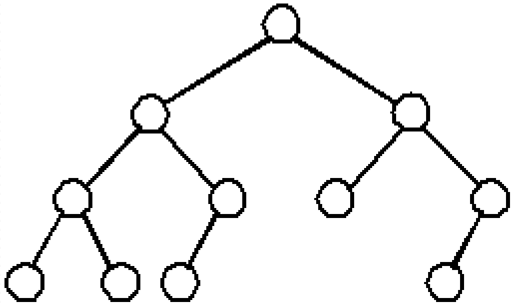
A balanced binary tree
(it's also a complete binary tree)

Properties of Binary Trees

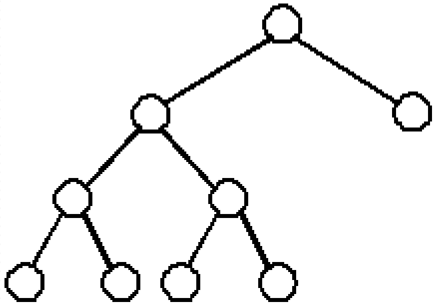
- A *complete* binary tree is similar to a balanced tree except that all the leaves must be placed as far to the left as possible. (The leaves must be "filled-in" from the left to right, one level at a time.)



A complete binary tree



An incomplete binary tree



An incomplete binary tree

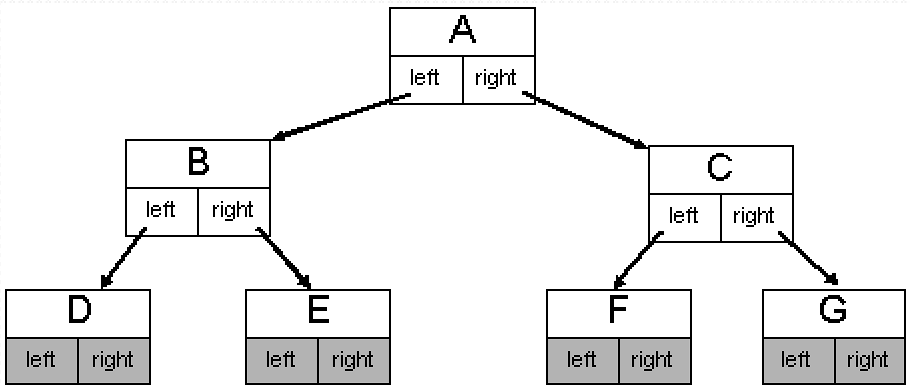
Trees vs. Linked List

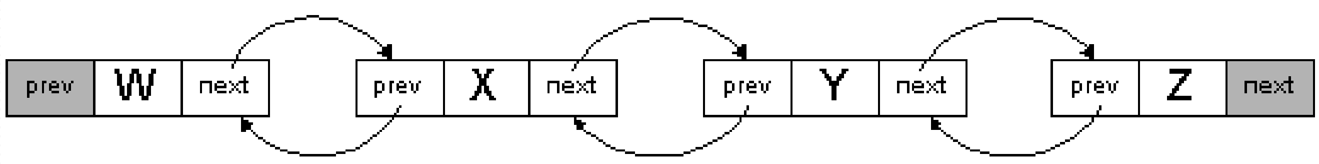
- Realize that the two links in a binary tree are not quite the same as the two links in a doubly linked list:
  - Trees have *left* and *right* link.
  - Lists have *previous* and *next* link.
  - Both imply ordering, but a different kind of ordering.
  - Structurally speaking, they are equivalent

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

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

Trees vs. Linked List





Traversing Binary Trees

- Trees are inherently recursive data structures.
- Recursive algorithms are quite appropriate.
  - In some cases, iterative algorithms can be significantly more complicated.

Traversal Order

Pre-order traversal

1. ***Visit the node***
2. Traverse the left subtree.
3. Traverse the right subtree.

In-order traversal

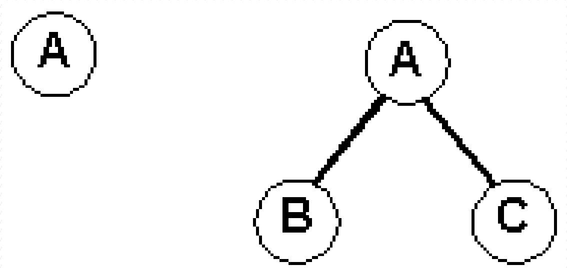
1. Traverse the left subtree.
2. ***Visit the node***
3. Traverse the right subtree.

Post-order traversal

1. Traverse the left subtree.
2. Traverse the right subtree.
3. ***Visit the node***

Traversal order

- Given these binary trees



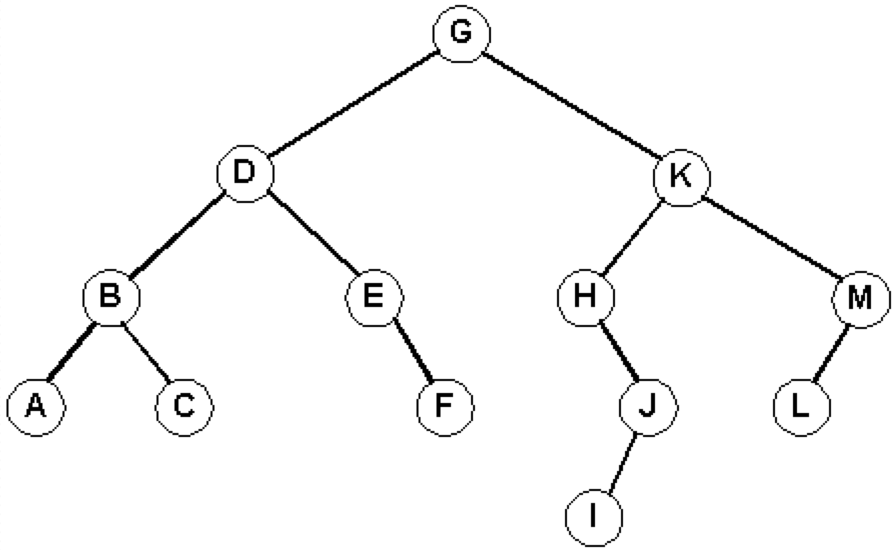
Assuming that *visiting* a node means printing the letter of the node.
The result of traversing the first tree is **A** in all 3 cases.

For the second tree we have:

- Pre-order traversal: ***ABC*** (**visit,** traverse left, traverse right).
- In-order traversal: ***BAC***(traverse left, **visit**, traverse right).
- Post-order traversal: **BCA**(traverse left, traverse right, **visit**).

Traversal order

- A larger example

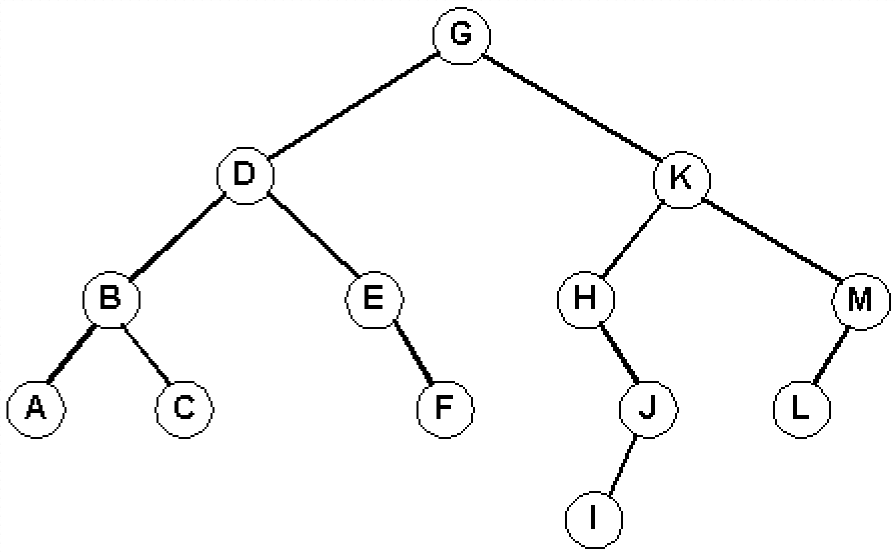


For the second tree we have:

- Pre-order traversal:
- In-order traversal:
- Post-order traversal:

Traversal order

- A larger example



For the second tree we have:

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

Implementing 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;
}
```

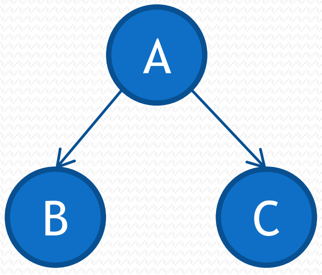
```
void FreeNode(Node *node)
{
     delete node;
}

typedef Node* Tree;
```

- We can construct random binary trees by providing a height for the final tree.

```
Tree BuildRandBinTreePre(int height);
```

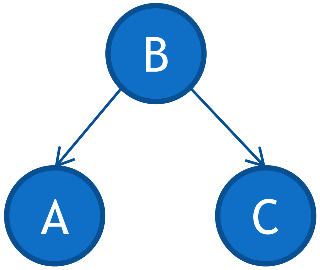
I- Building a Tree

```
int Count = 0;

Tree BuildRandBinTreePre(int height)
{
    if (height == -1)
        return 0;

    Node *node = MakeNode('A' + Count++);       // "visiting" the node
    node->left = BuildRandBinTreePre(height - 1);  // build the left tree
    node->right = BuildRandBinTreePre(height - 1); // build the right tree
    return node;
}
void main(void)
{
    Tree t = BuildRandBinTreePre(1);
}
```

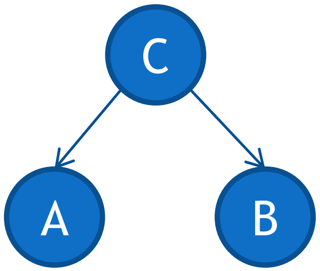


The resulting tree

Building a Tree




```
Tree BuildRandBinTreeIn(int height)
{
      if (height == -1)
            return 0;

      Node *node = new Node;
      node->left = BuildRandBinTreeIn(height - 1);  // build left subtree
      node->data = 'A' + Count++;                   // visit node
      node->right = BuildRandBinTreeIn(height - 1); // build right subtree
      return node;
}
```




```
Tree BuildRandBinTreePost(int height)
{
      if (height == -1)
            return 0;

      Node *node = new Node;
      node->left = BuildRandBinTreePost(height - 1);  // build left subtree
      node->right = BuildRandBinTreePost(height - 1); // build right subtree
      node->data = 'A' + Count++;                     // visit node
      return node;
}
```

II- Finding the number of nodes

- As always, start by stating the algorithm in english:
  - 0 if the tree is empty
    - 0
  - if 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);
}
```

III- Finding the height of a tree

- If the tree is empty
  - -1
- If $height\ of\ left\ subtree\ >\ height\ of\ right\ subtree$
  - $1 + height\ of\ left\ subtree$
- otherwise
  - $1 + height\ of\ right\ subtree$

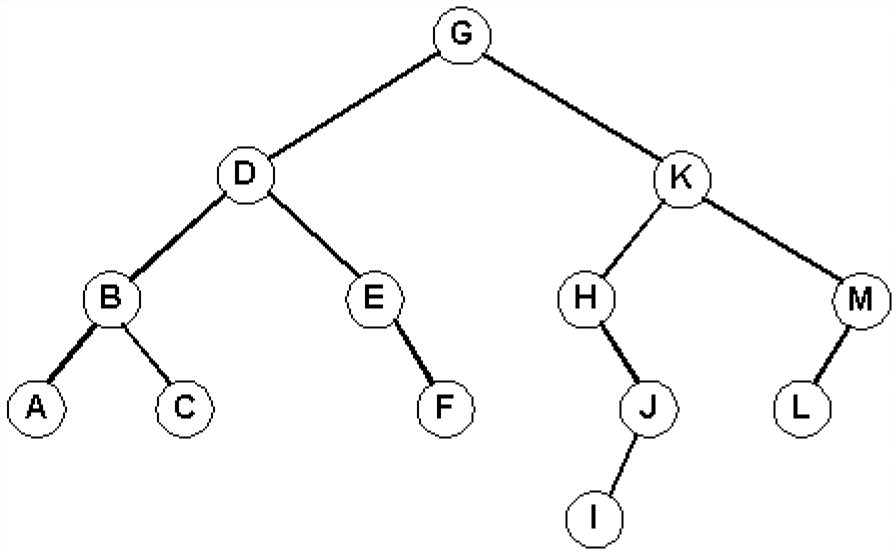
```
int Height(Tree tree)
{
     if (tree == 0)
          return -1;

     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 lh = Height(tree->left);
    int rh = Height(tree->right);

    if (lh > rh)
        return lh + 1;
    else
        return rh + 1;
}
```

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

- If level to visit is 0
  - Visit the node
- If level to visit is > 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
	{
		TraverseLevelOrder2(tree->left, level - 1);
		TraverseLevelOrder2(tree->right, level - 1);
	}
}
```

Level-Order Traversal: Exercise

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**