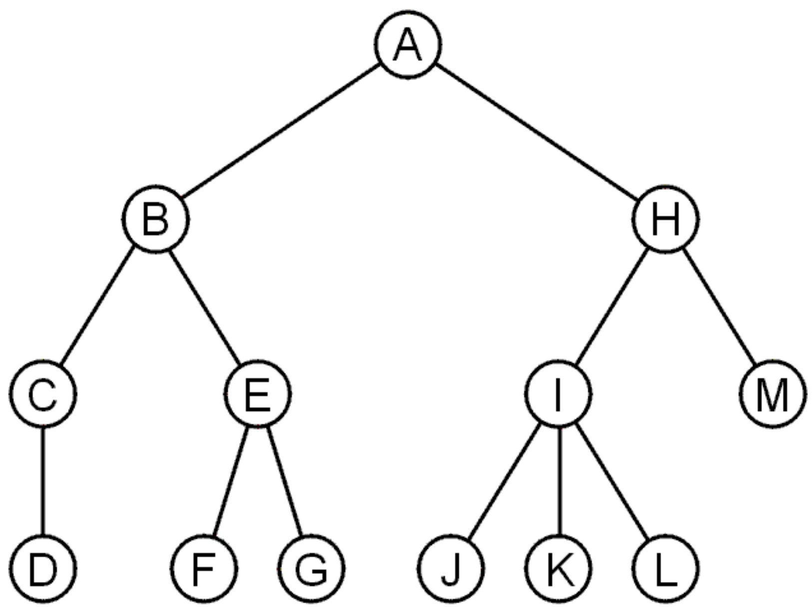
Trees
COL 106

Acknowledgement :Many slides are courtesy
Douglas Harder, UWaterloo

Trees

A rooted tree data structure stores information in *nodes*

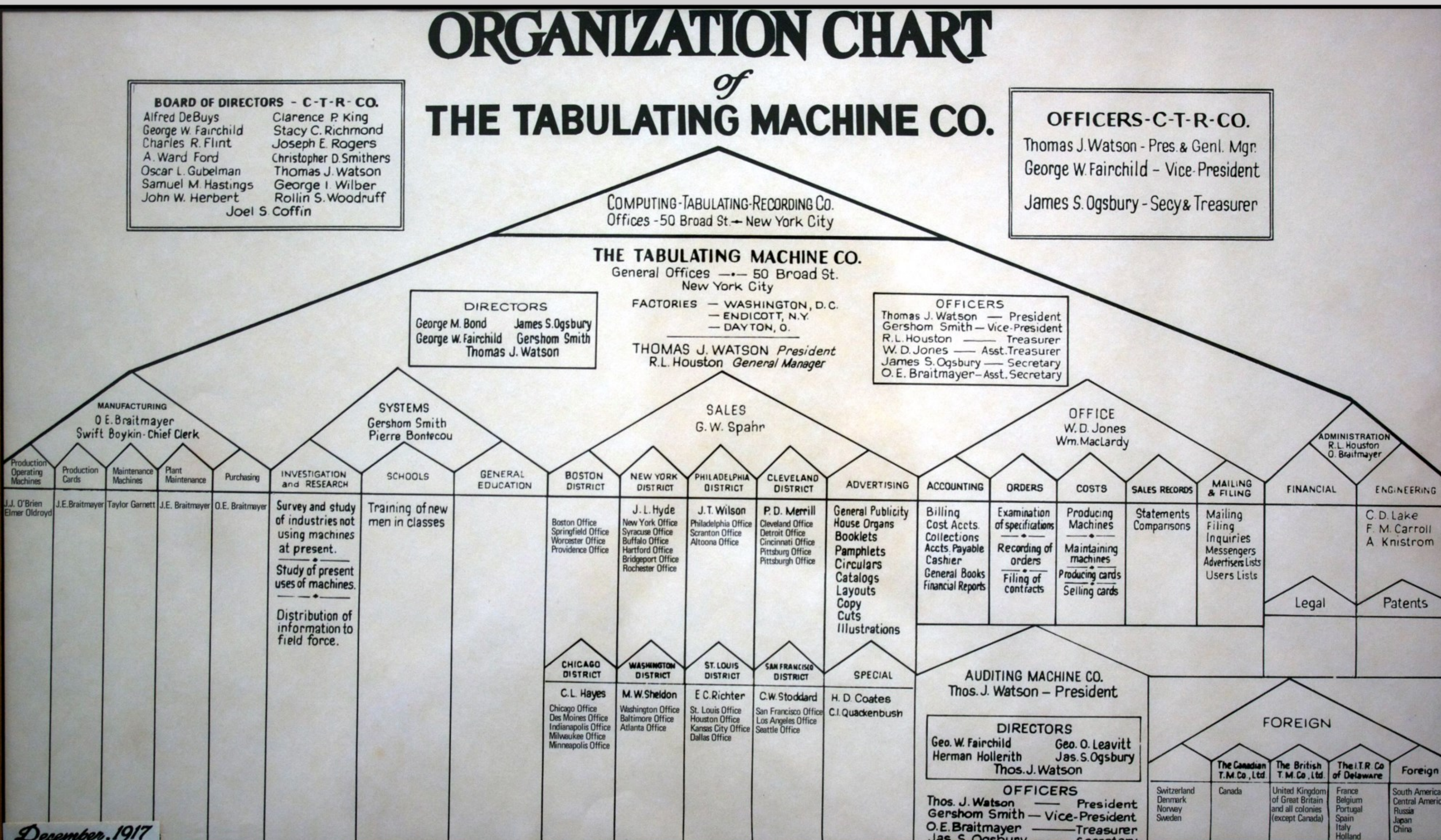
- Similar to linked lists:
    - There is a first node, or *root*
    - Each node has variable number of references to successors (children)
    - Each node, other than the root, has exactly one node as its predecessor (or parent)



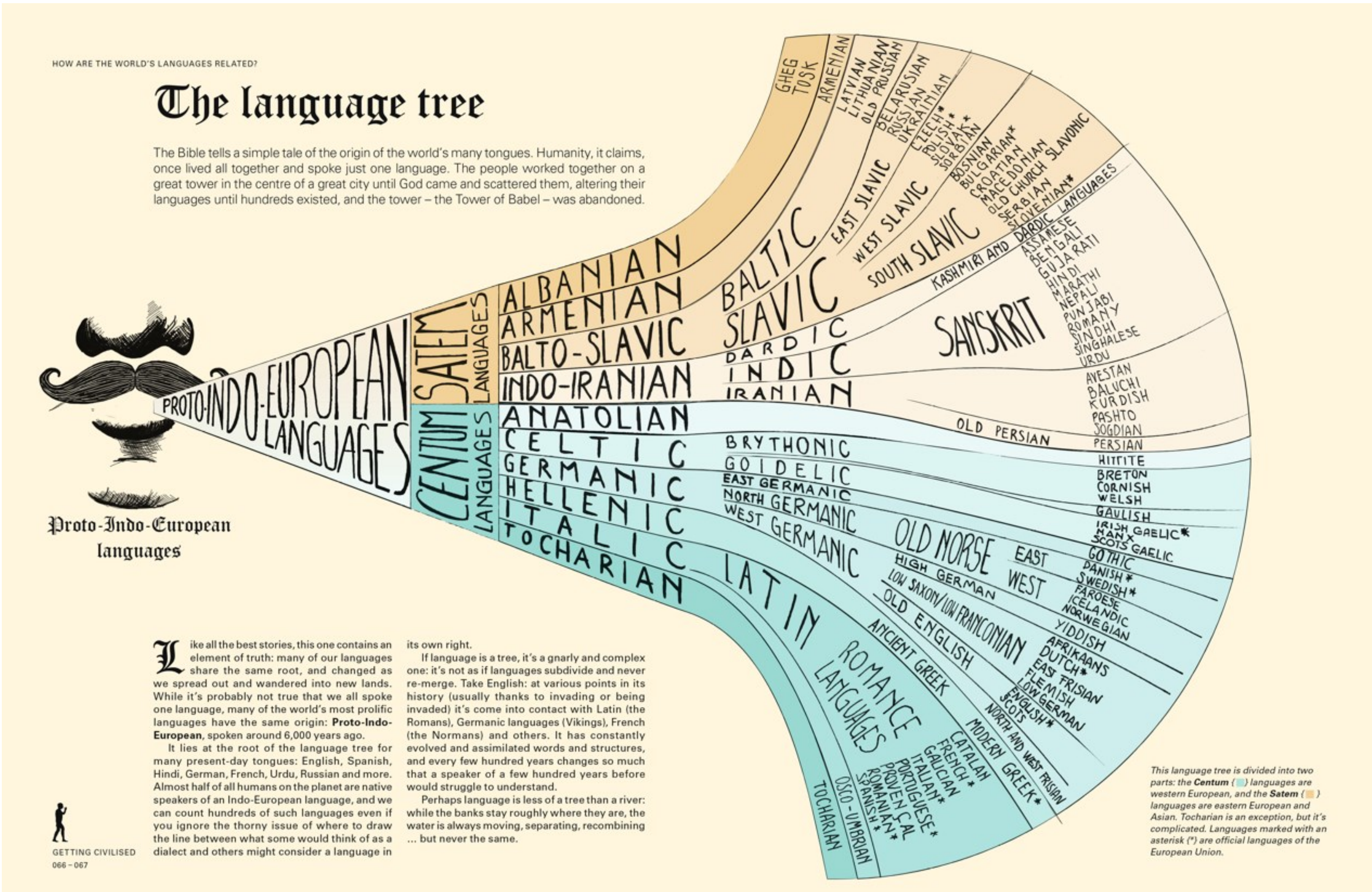
What are trees suitable for ?

To store hierarchy of people

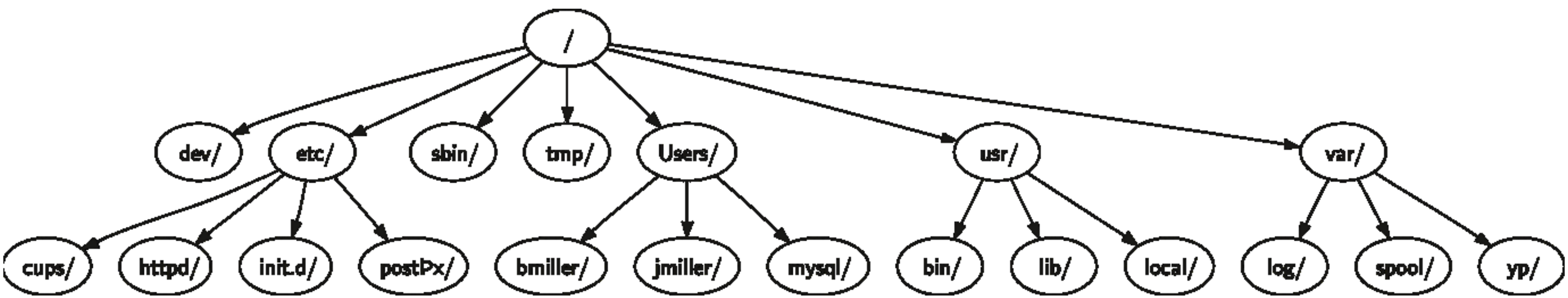
To store organization of departments



To capture the evolution of languages

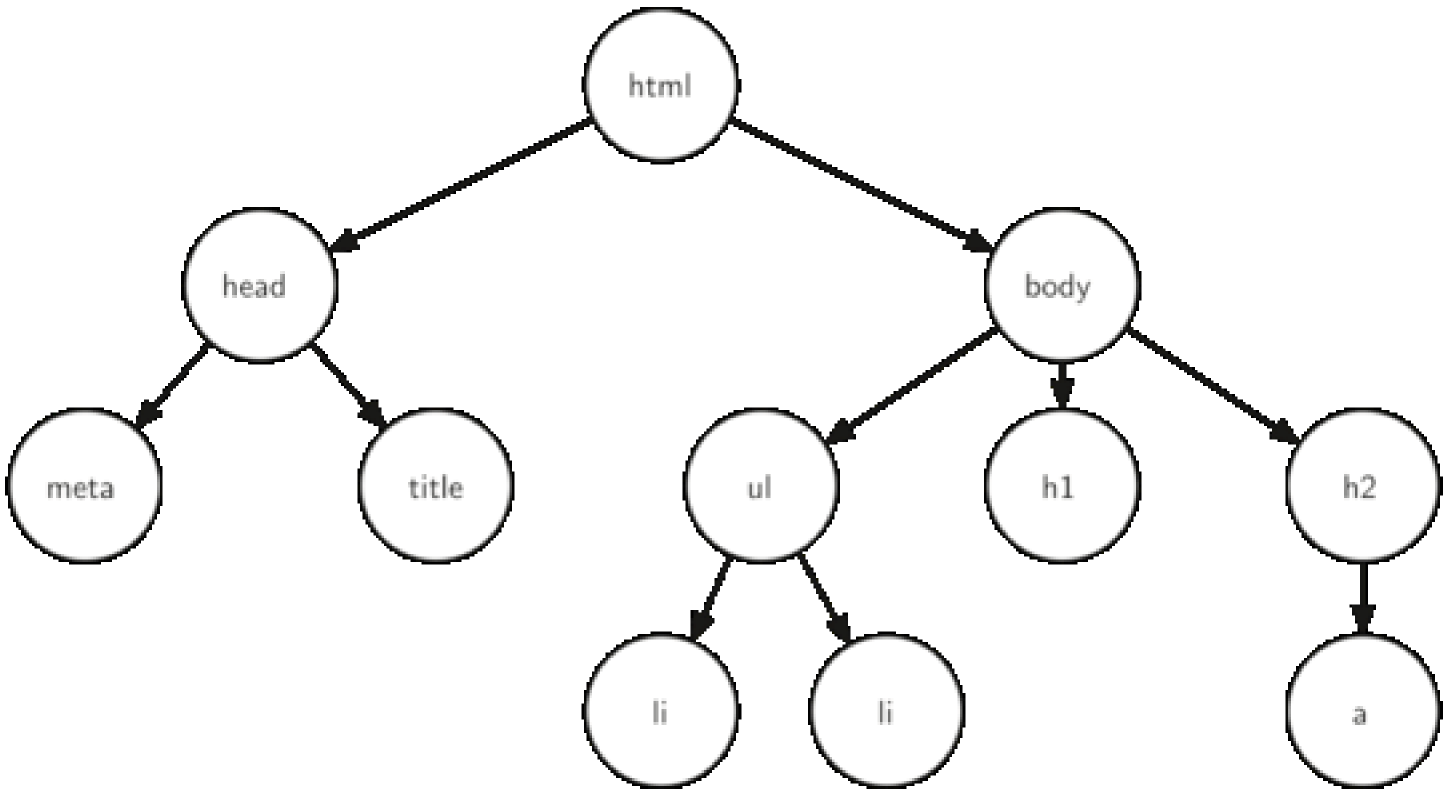


To organize file-systems

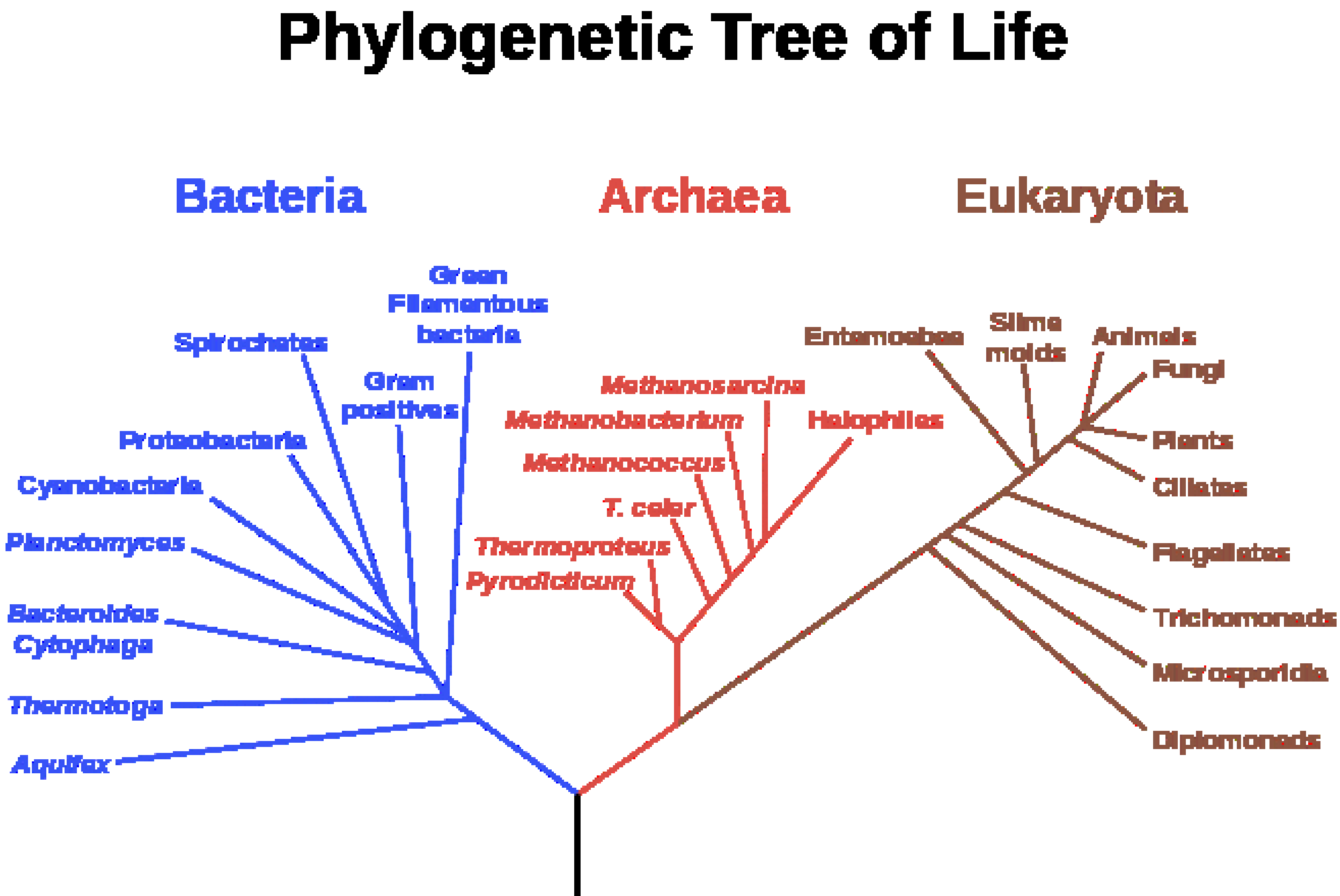


Unix file system

Markup elements in a webpage



To store phylogenetic data



This will be our running example. Will illustrate tree concepts using actual phylogenetic data.

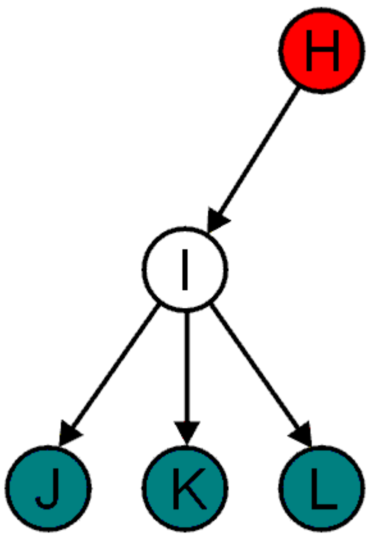
Terminology

All nodes will have zero or more child nodes or *children*

– I has three children: J, K and L

For all nodes other than the root node, there is one parent node

– H is the parent of I

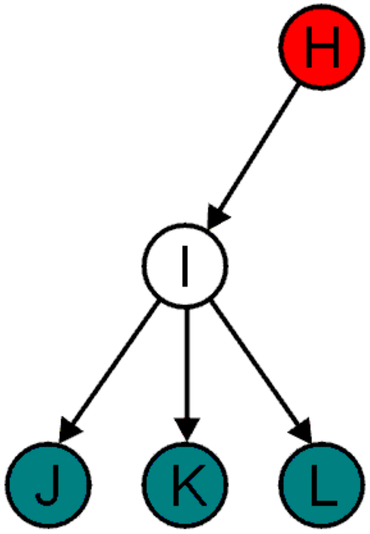


Terminology

The *degree* of a node is defined as the number of its children: $\deg(\mathrm{I}) = 3$

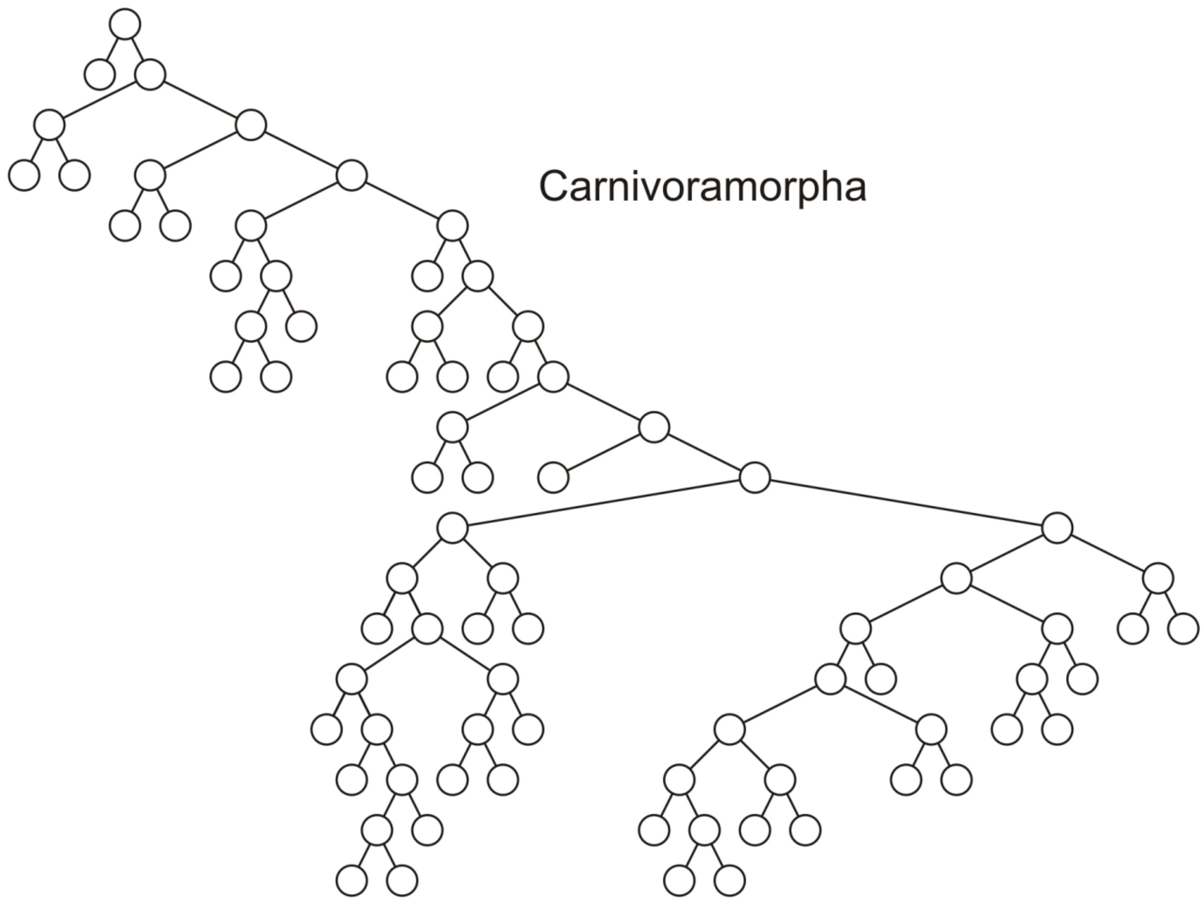
Nodes with the same parent are *siblings*

– J, K, and L are siblings



Terminology

Phylogenetic trees have nodes with degree 2 or 0:

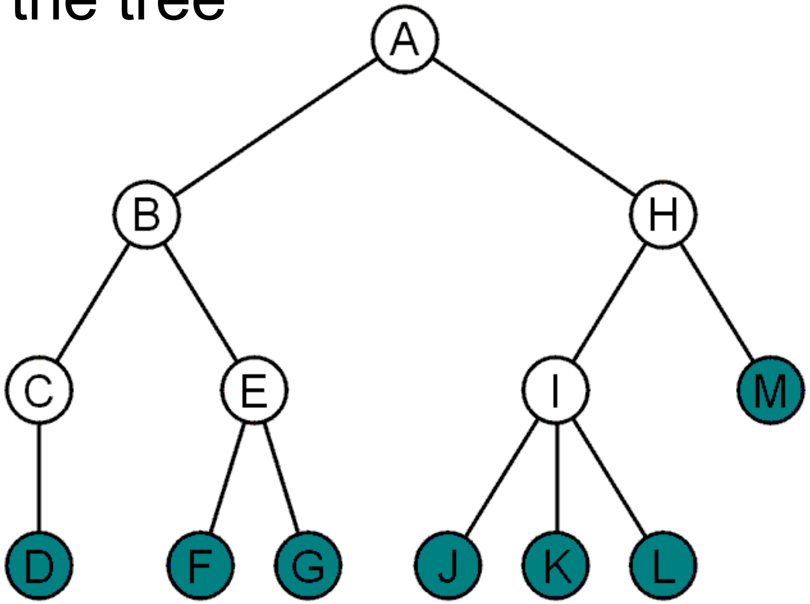


Wesley-Hunt, G. D.; Flynn, J. J. “Phylogeny of the Carnivora: basal relationships among the Carnivoramorphans, and assessment of the position of ‘Miacoidea’

Terminology

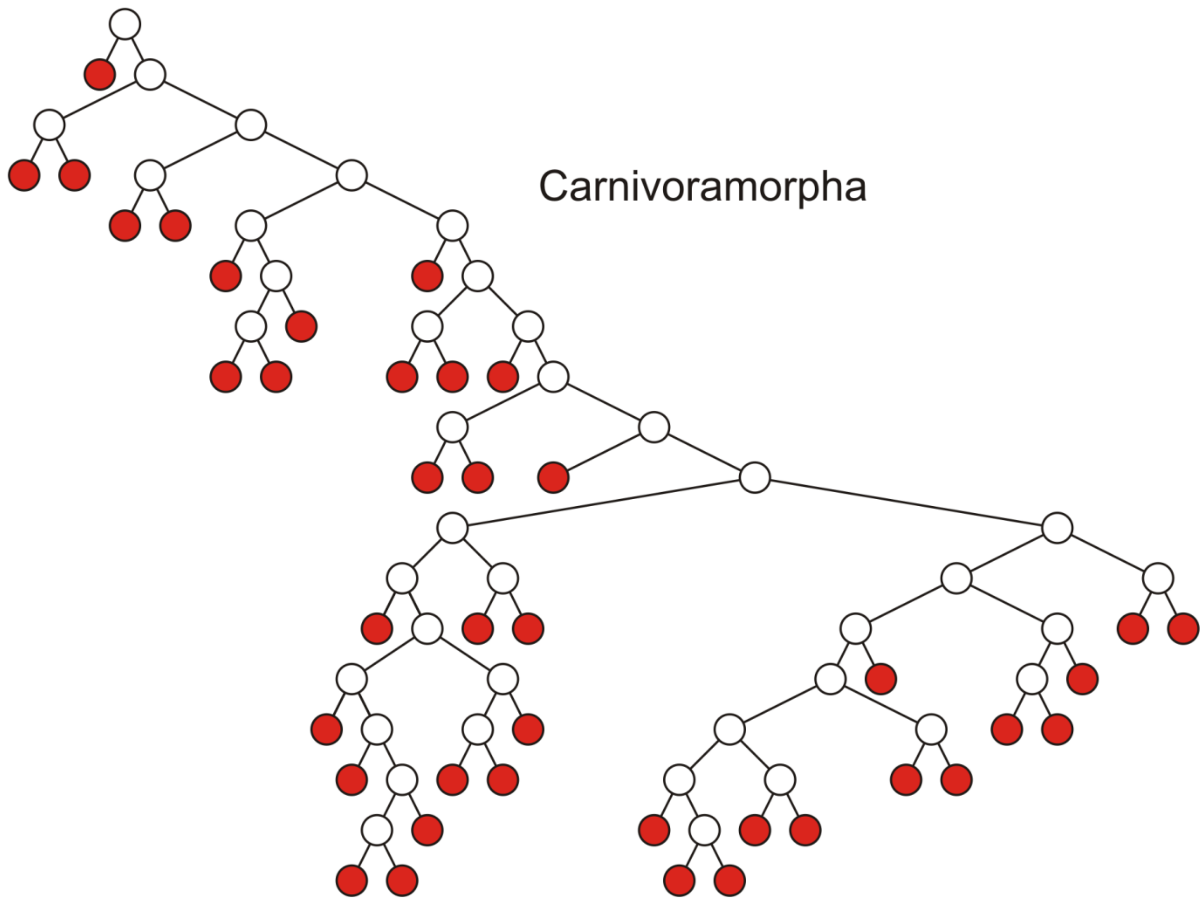
Nodes with degree zero are also called *leaf nodes*

All other nodes are said to be *internal nodes*, that is, they are internal to the tree



Terminology

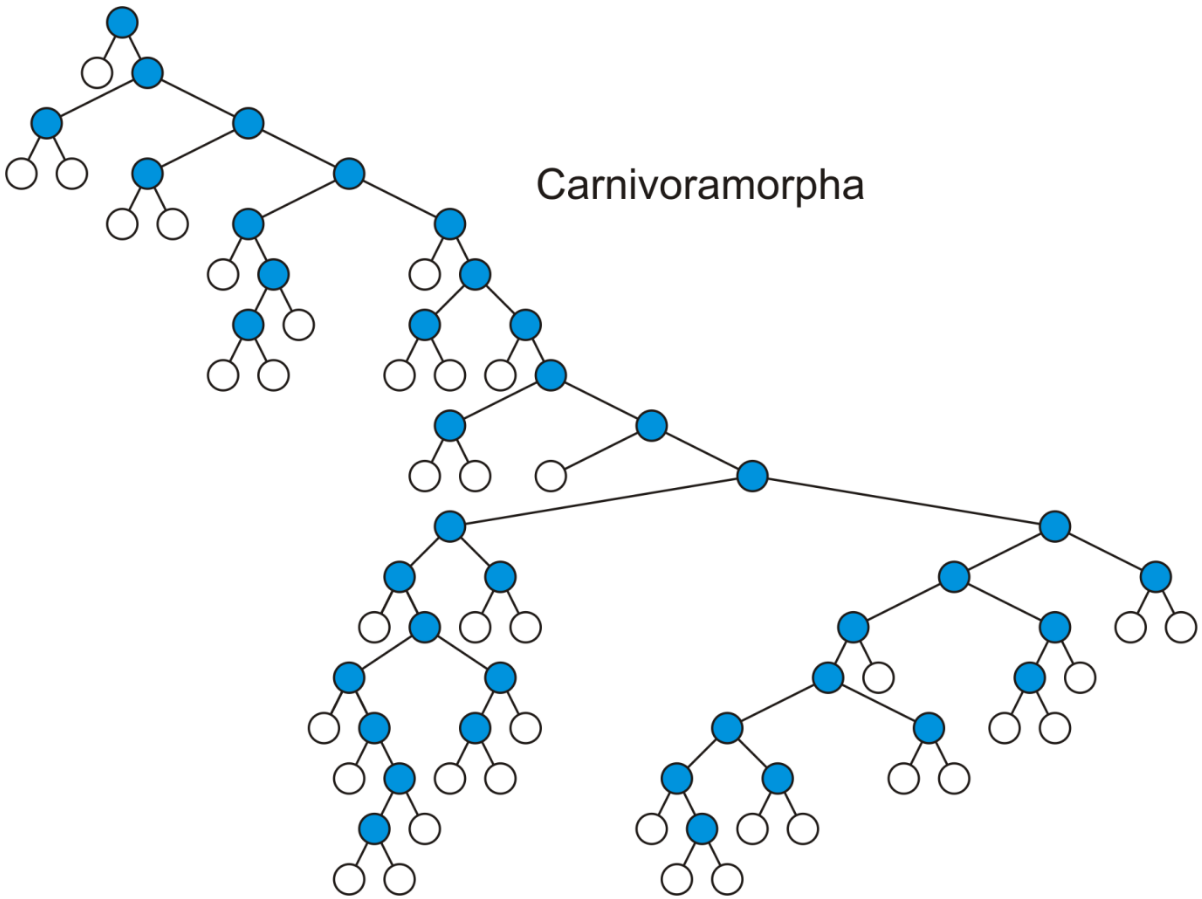
Leaf nodes:



Wesley-Hunt, G. D.; Flynn, J. J. “Phylogeny of the Carnivora: basal relationships among the Carnivoramorphans, and assessment of the position of ‘Miacoidea’

Terminology

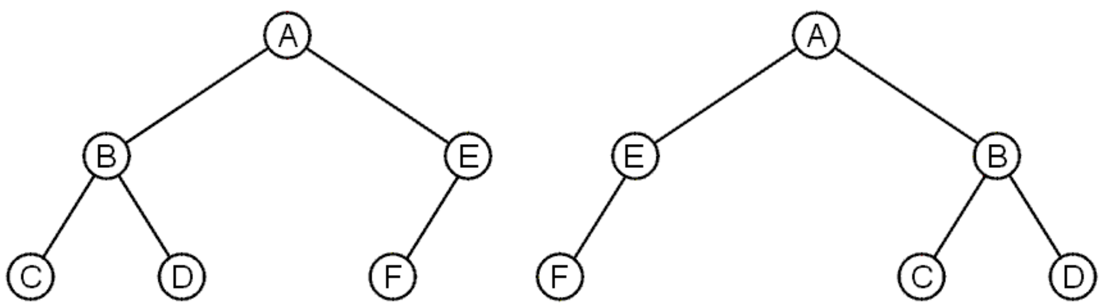
Internal nodes:



Wesley-Hunt, G. D.; Flynn, J. J. “Phylogeny of the Carnivora: basal relationships among the Carnivoramorphans, and assessment of the position of ‘Miacoidea’

Terminology

These trees are equal if the order of the children is ignored (*Unordered trees*)

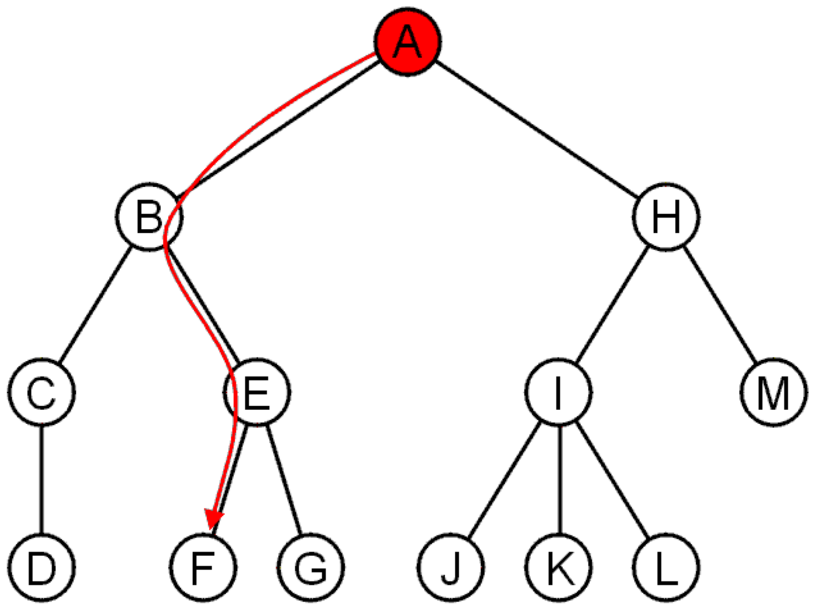


They are different if order is relevant (*ordered trees*)

– We will usually examine ordered trees (linear orders)
– In a hierarchical ordering, order is not relevant

Terminology

The shape of a rooted tree gives a natural flow from the *root node*, or just *root*



Terminology

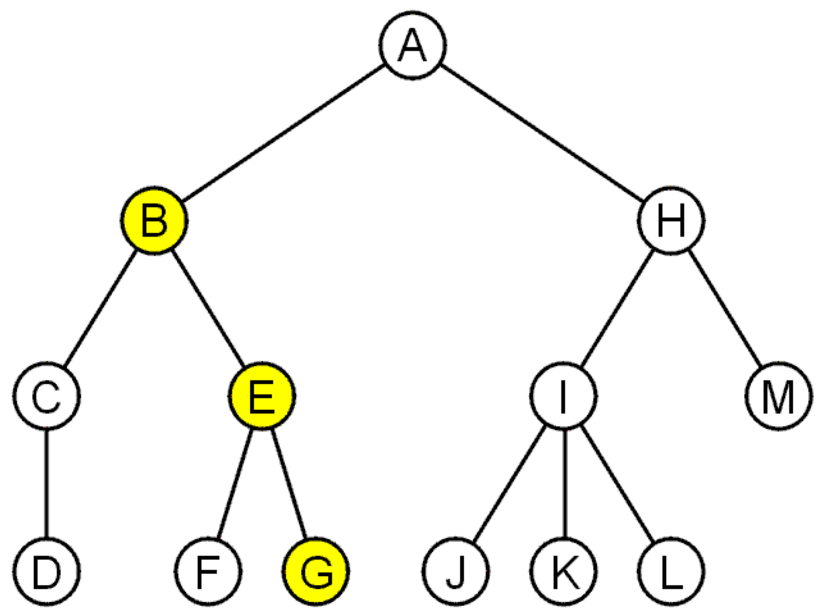
A path is a sequence of nodes

$$(a_0, a_1, ..., a_n)$$

where a_{k+1} is a child of a_k is

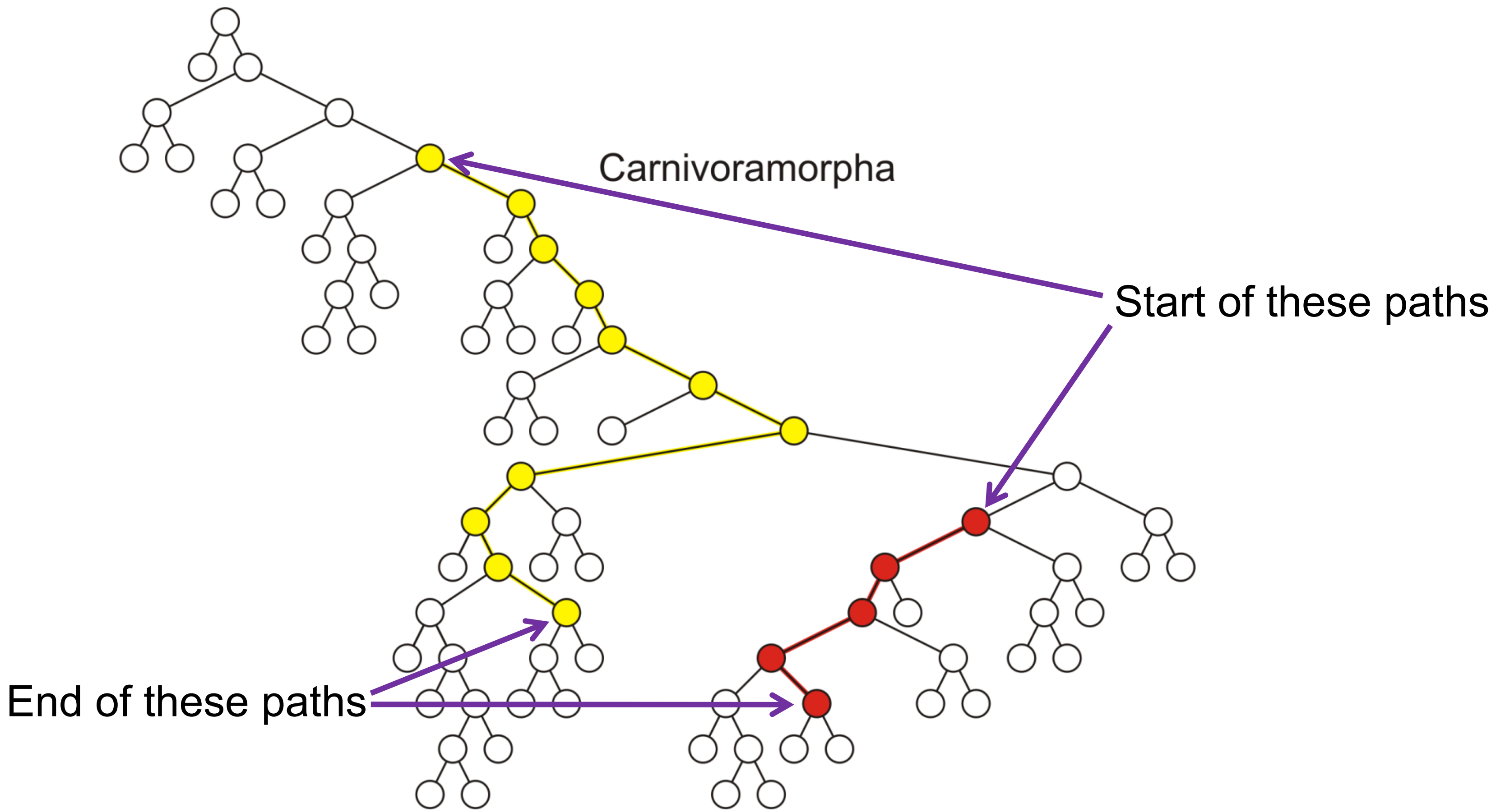
The length of this path is n

E.g., the path (B, E, G) has length 2



Terminology

Paths of length 10 (11 nodes) and 4 (5 nodes)

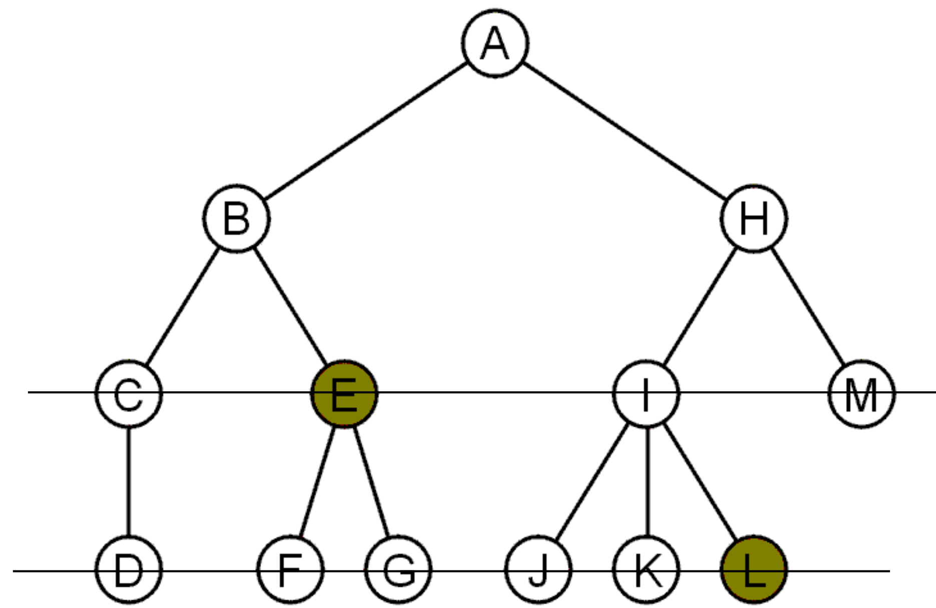


Wesley-Hunt, G. D.; Flynn, J. J. “Phylogeny of the Carnivora: basal relationships among the Carnivoramorphans, and assessment of the position of ‘Miacoidea’

Terminology

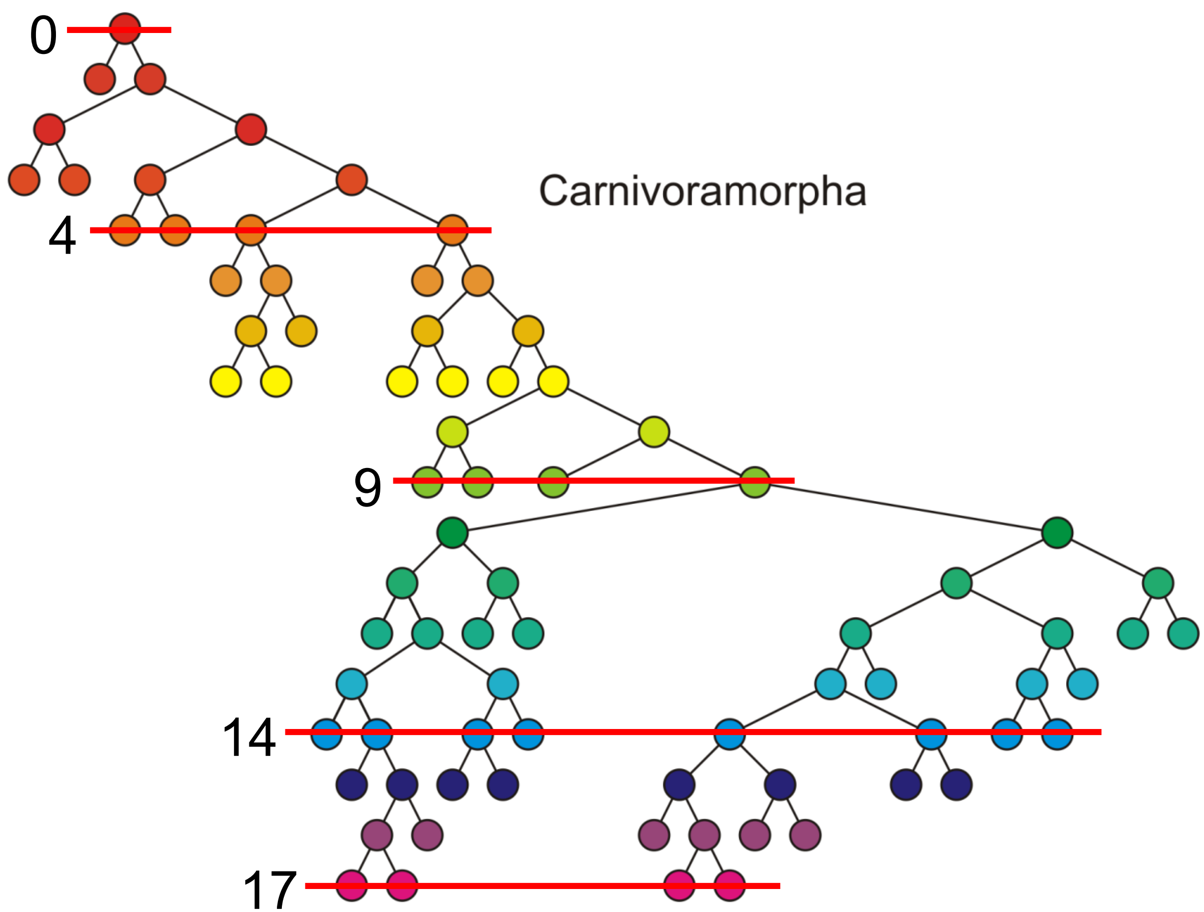
For each node in a tree, there exists a unique path from the root node to that node

The length of this path is the *depth* of the node, *e.g.*,
- E has depth 2
- L has depth 3



Terminology

Nodes of depth up to 17



Wesley-Hunt, G. D.; Flynn, J. J. "Phylogeny of the Carnivora: basal relationships among the Carnivoramorphans, and assessment of the position of 'Miacoidea'

Terminology

The *height* of a tree is defined as the maximum depth of any node within the tree

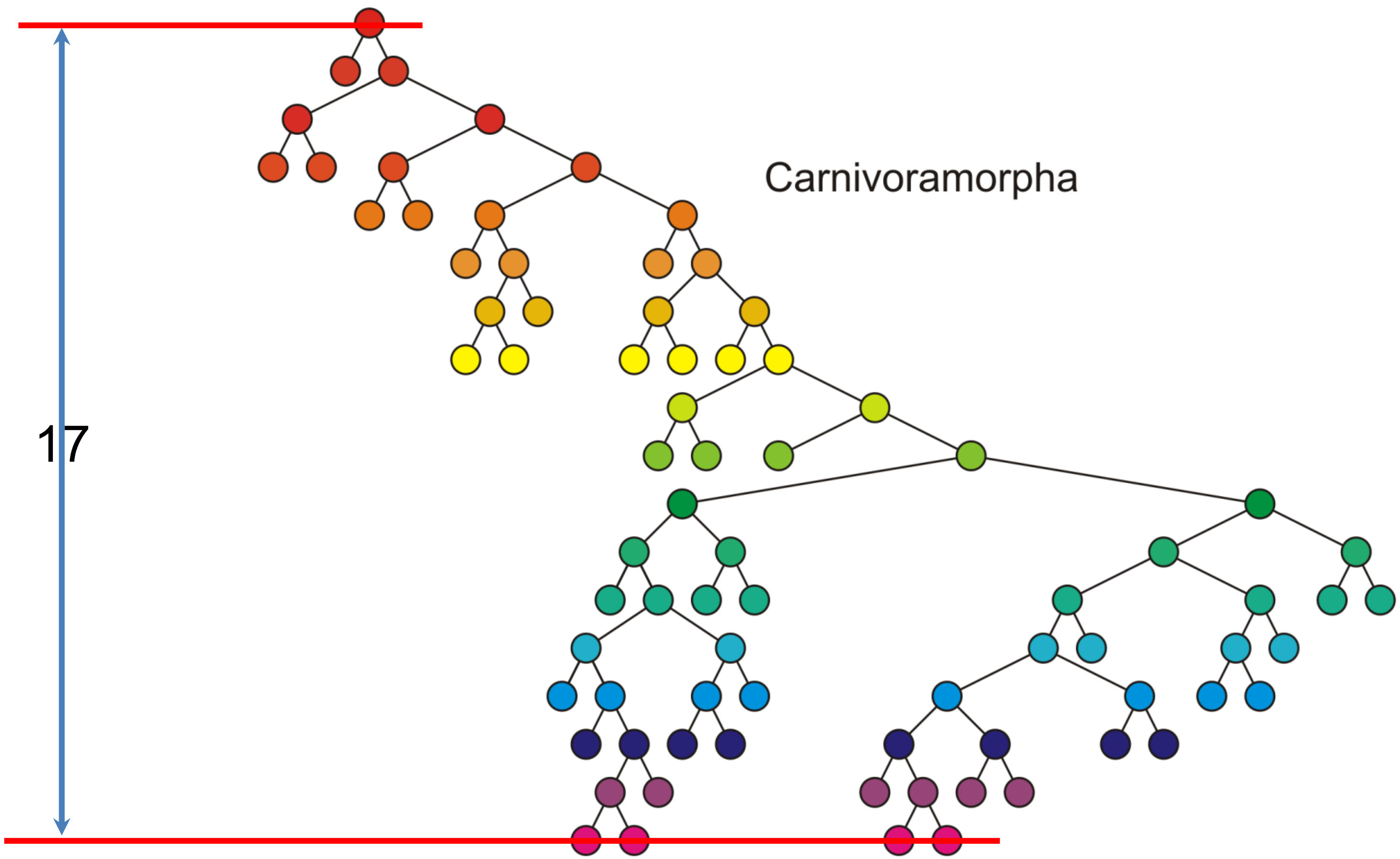
The height of a tree with one node is 0

- Just the root node

For convenience, we define the height of the empty tree to be -1

Terminology

The height of this tree is 17



Wesley-Hunt, G. D.; Flynn, J. J. “Phylogeny of the Carnivora: basal relationships among the Carnivoramorphans, and assessment of the position of ‘Miacoidea’

Terminology

If a path exists from node a to node b:

- a is an *ancestor* of b
- b is a *descendent* of a

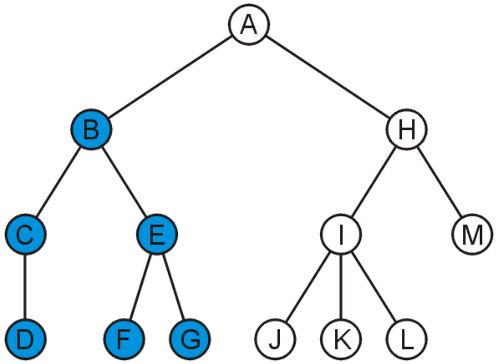
Thus, a node is both an ancestor and a descendant of itself

- We can add the adjective *strict* to exclude equality: a is a *strict descendent* of b if a is a descendant of b but $a \neq b$

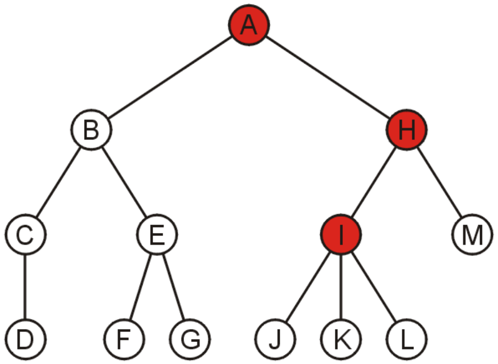
The root node is an ancestor of all nodes

Terminology

The descendants of node B are B, C, D, E, F, and G:

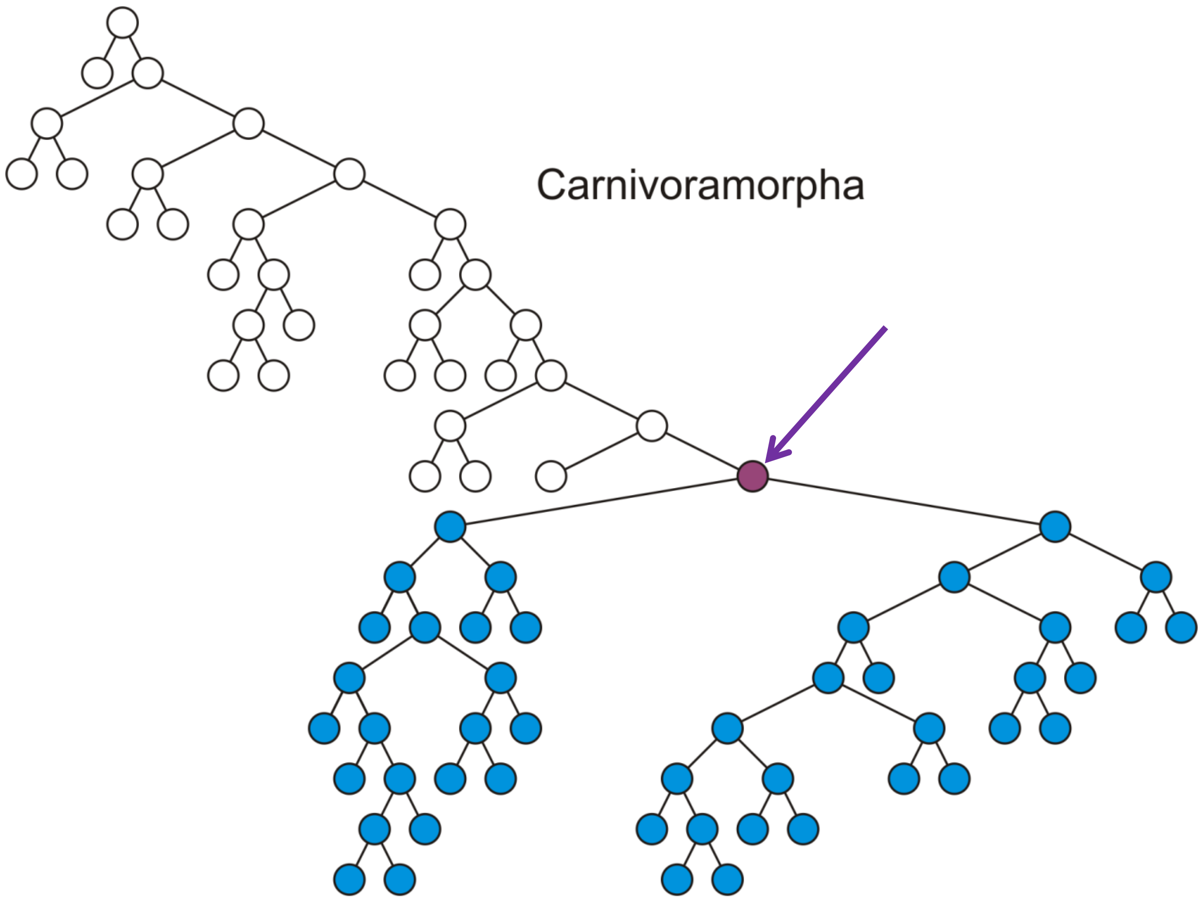


The ancestors of node I are I, H, and A:



Terminology

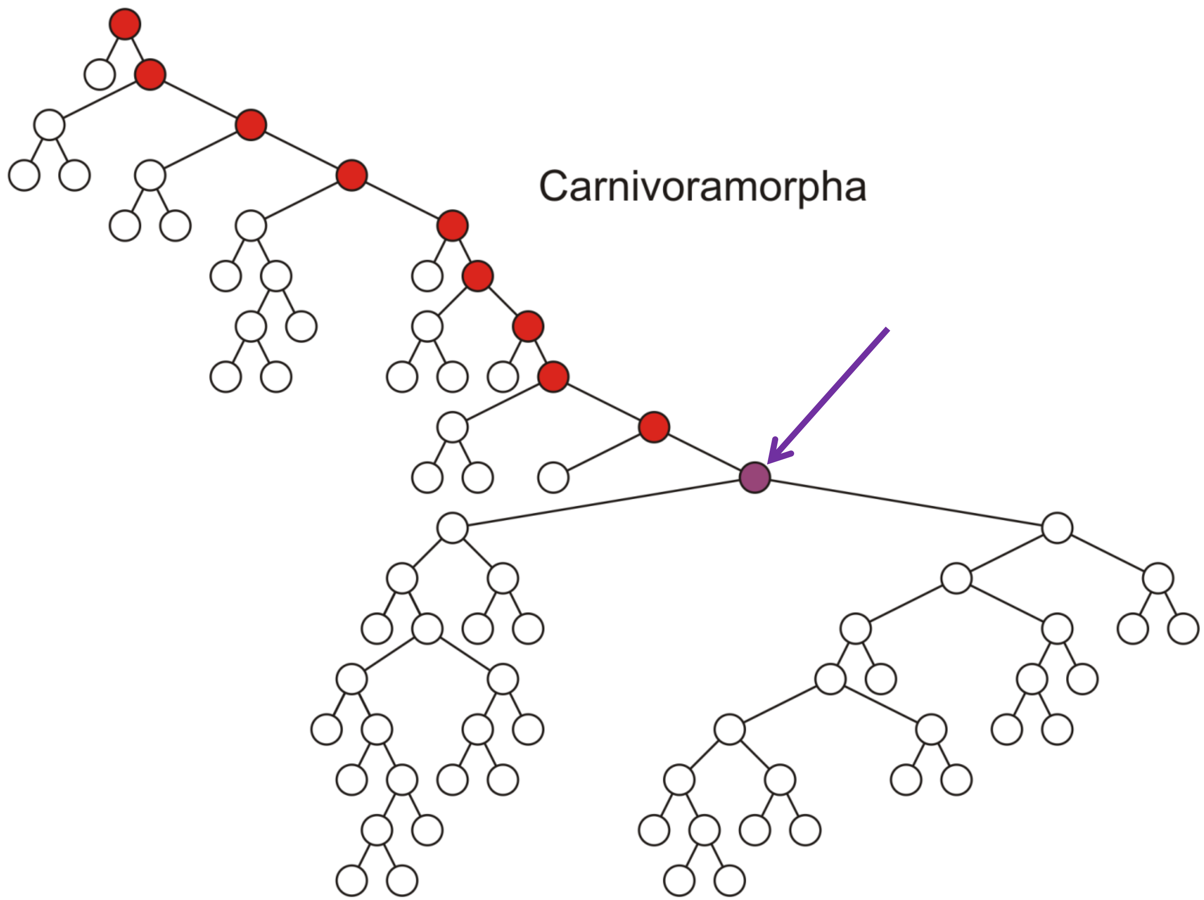
All descendants (including itself) of the indicated node



Wesley-Hunt, G. D.; Flynn, J. J. "Phylogeny of the Carnivora: basal relationships among the Carnivoramorphans, and assessment of the position of 'Miacoidea'

Terminology

All ancestors (including itself) of the indicated node



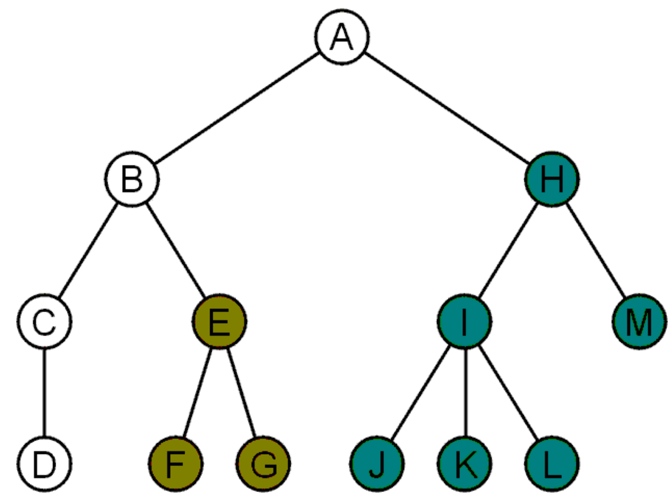
Wesley-Hunt, G. D.; Flynn, J. J. "Phylogeny of the Carnivora: basal relationships among the Carnivoramorphans, and assessment of the position of 'Miacoidea'

Terminology

Another approach to a tree is to define the tree recursively:

- A degree-0 node is a tree
- A node with degree n is a tree if it has n children and all of its children are disjoint trees (*i.e.*, with no intersecting nodes)

Given any node a within a tree with root r, the collection of a and all of its descendants is said to be a *subtree of the tree with root* a



Example: XHTML

Consider the following XHTML document

```
<html>
    <head>
        <title>Hello World!</title>
    </head>
    <body>
        <h1>This is a <u>Heading</u></h1>

        <p>This is a paragraph with some
        <u>underlined</u> text.</p>
    </body>
</html>
```

Example: XHTML

Consider the following XHTML document

title

```
<html>
    <head>
        <title>Hello World!</title>
    </head>
    <body>
        <h1>This is a <u>Heading</u></h1>

        <p>This is a paragraph with some
        <u>underlined</u> text.</p>
    </body>
</html>
```

heading

body of page

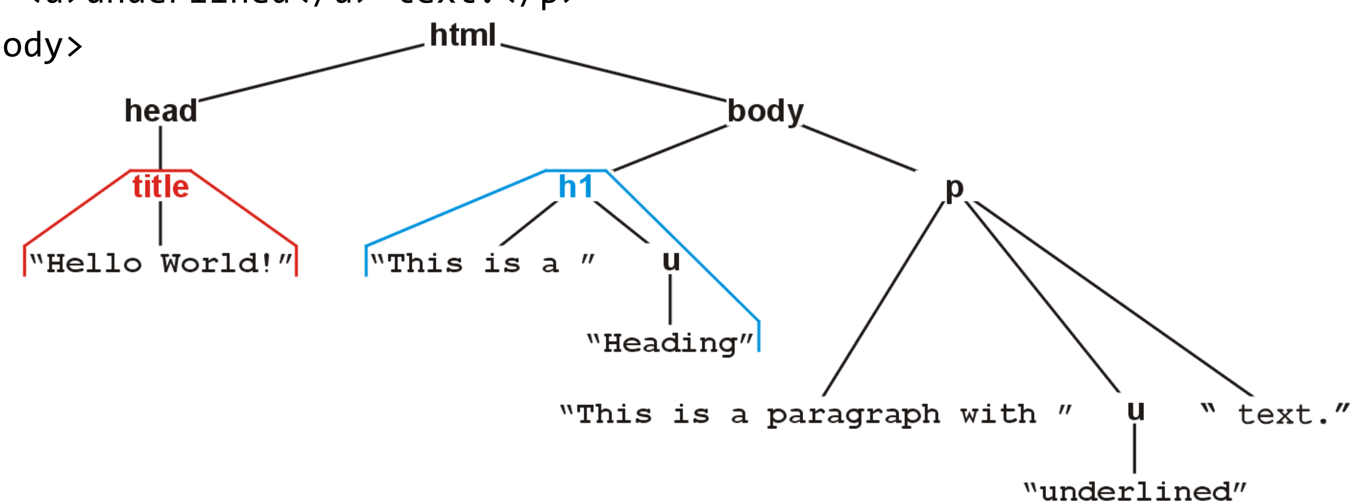
paragraph

underlining

Example: XHTML

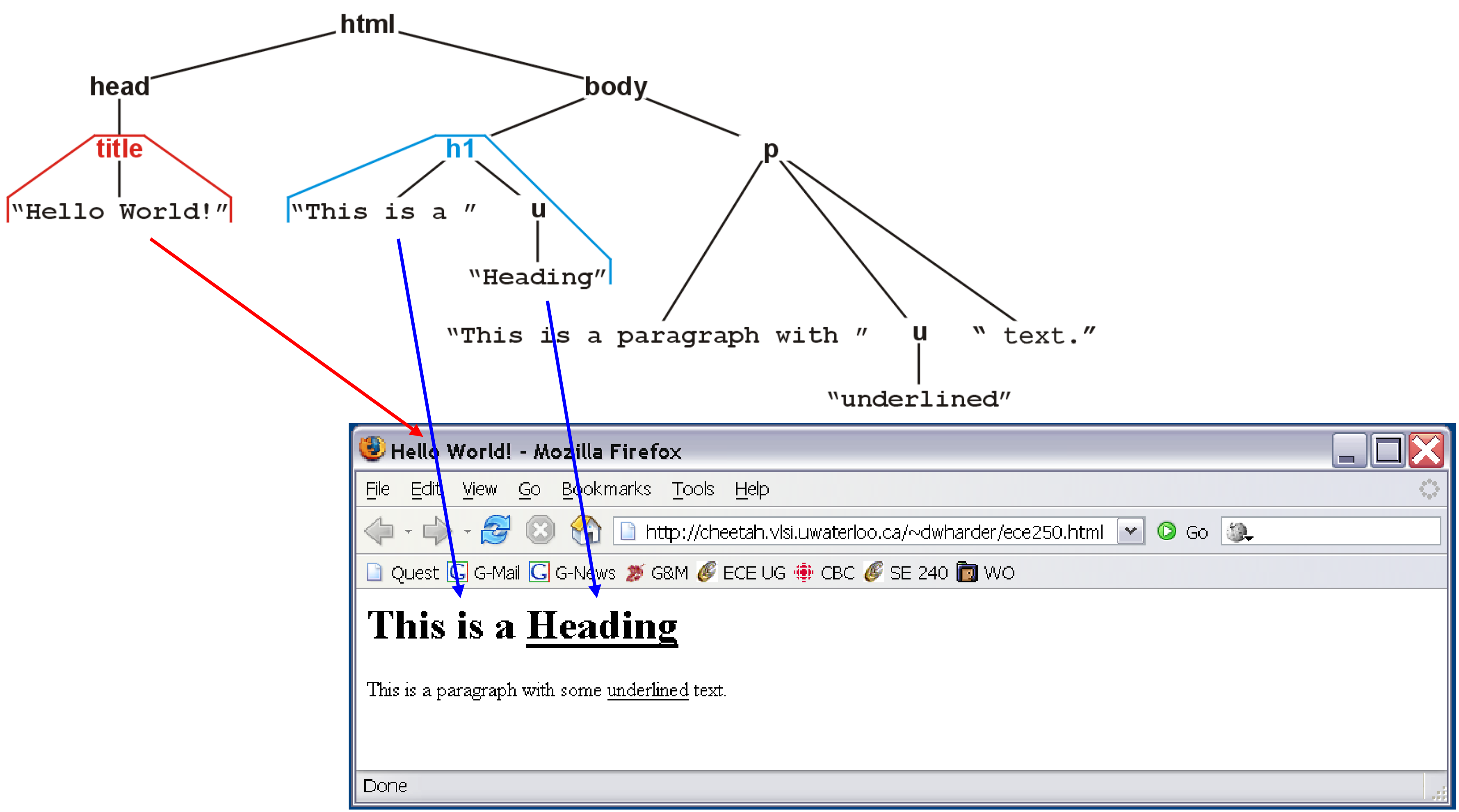
The nested tags define a tree rooted at the HTML tag

```
<html>
    <head>
        <title>Hello World!</title>
    </head>
    <body>
        <h1>This is a <u>Heading</u></h1>

        <p>This is a paragraph with some
        <u>underlined</u> text.</p>
    </body>
</html>
```



Example: XHTML

Web browsers render this tree as a web page



Iterator ADT

Most ADTs in Java can provide an iterator object, used to traverse all the data in any linear ADT.

Iterator Interface

```
public interface Iterator<E>{
   boolean hasNext();
   E next();
  void remove(); // Optional
}
```

Getting an Iterator

You get an iterator from an ADT by calling the method `iterator();`

```
Iterator<Integer> iter = myList.iterator();
```

Now a simple while loop can process each data value in the ADT:

```
while(iter.hasNext()) {
    process iter.next()
}
```

Adding Iterators to SimpleArrayList is easy

First, we add the `iterator()` method to `SimpleArrayList`:

```
public Iterator<E> iterator(){
    return new
      ArrayListIterator<E>(this);
}
```

Then we implement the iterator class for Lists:

```
import java.util.*;
public class ArrayListIterator<E>
 implements Iterator<E> {
    // *** fields ***
    private SimpleArrayList<E> list;
    private int curPos;

    public ArrayListIterator(
          SimpleArrayList<E> list) {
        this.list = list;
        curPos = 0;
    }
```

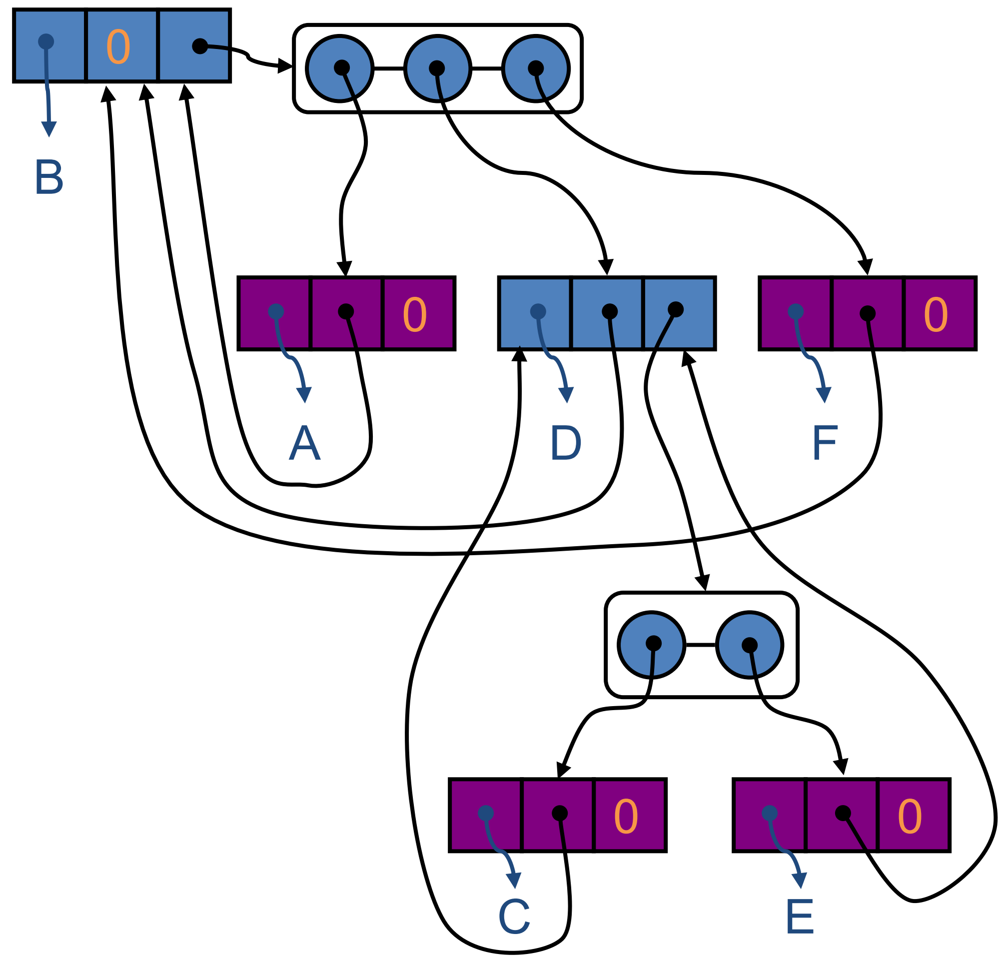
```
 public boolean hasNext() {
     return curPos < list.size();
  }

 public E next() {
    if (!hasNext()) throw
       new NoSuchElementException();

    E result = list.get(curPos);
    curPos++;
    return result;
 }

  public void remove() {
      throw new UnsupportedOperationException();
  }

}
```

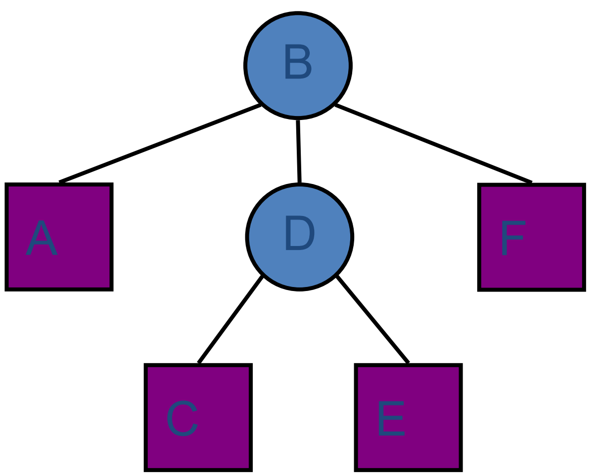
Tree ADT

- We use positions to abstract nodes
- Generic methods:
  - integer size()
  - boolean isEmpty()
  - objectIterator elements()
- Accessor methods:
  - node root()
  - node parent(p)
  - nodeIterator children(p)
- Query methods:
  - boolean isInternal(p)
  - boolean isLeaf (p)
  - boolean isRoot(p)
- Update methods:
  - swapElements(p, q)
  - object replaceElement(p, o)
- Additional update methods may be defined by data structures implementing the Tree ADT

A Linked Structure for General Trees

- A node is represented by an object storing
  - Element
  - Parent node
  - Sequence of children nodes





A Linked Structure for General Trees

```
Class Node {
    Object element;
    Node parent;
    List<Node> Children; // or array of Nodes

}
```

Tree using Array

- Each node contains a field for data and an array of pointers to the children for that node
  - Missing child will have null pointer
- Tree is represented by pointer to root
- Allows access to i^{th} child in O(1) time
- Very wasteful in space when only few nodes in tree have many children (most pointers are null)

<table>
<tr><td colspan="4">info</td></tr>
<tr><td>p_0</td><td>p_1</td><td>$\cdots$</td><td>p_{bf-1}</td></tr>
</table>

Tree Traversals

- A *traversal* visits the nodes of a tree in a systematic manner

- We will see three types of traversals
  - Pre-order
  - Post-order
  - In-order

Flavors of (Depth First) Traversal

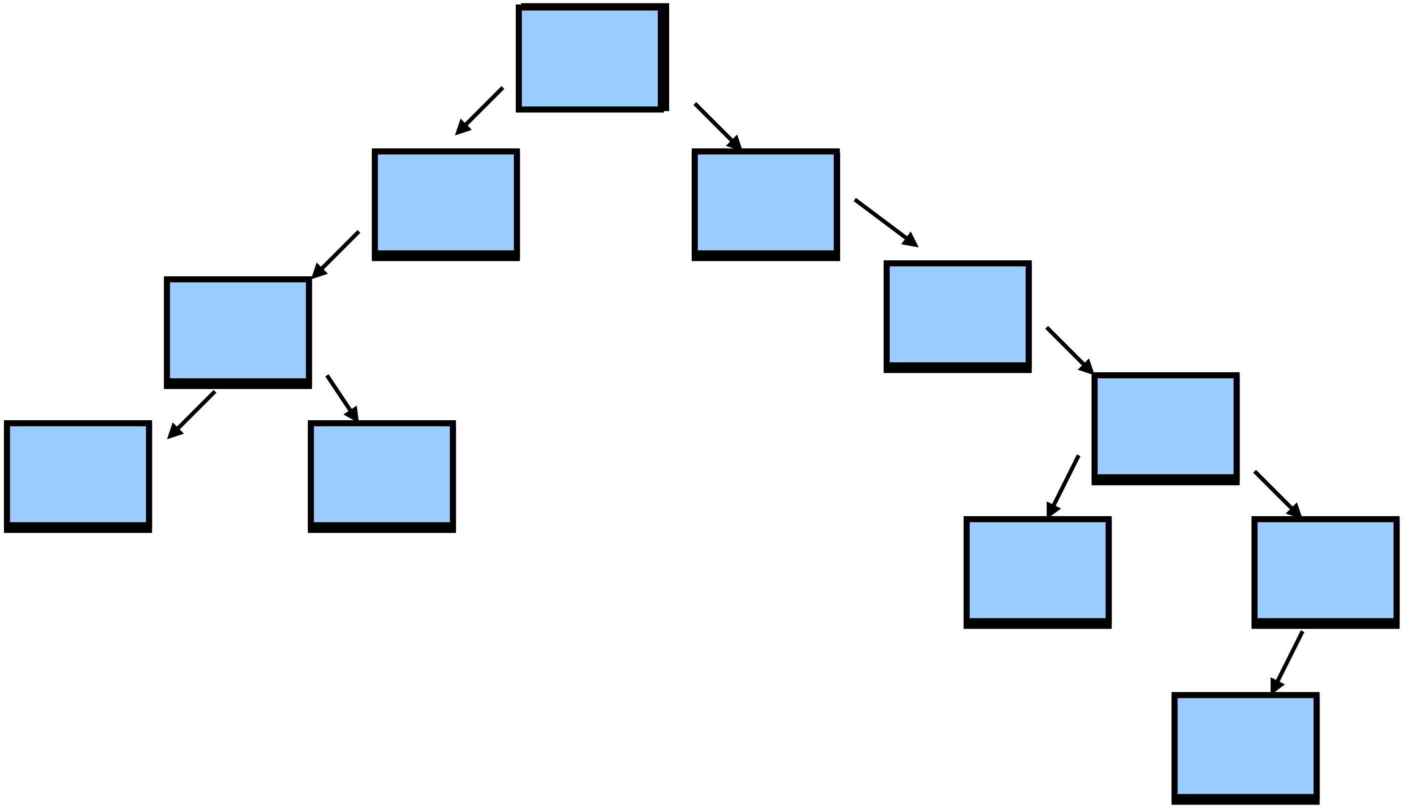
- In a *preorder traversal*, a node is visited before its descendants
- In a *postorder traversal, a node is visited after its* descendants
- In an *inorder traversal* a node is visited after its left subtree and before its right subtree

Preorder Traversal

- Process the root
- Process the nodes in the all subtrees in their order

```
Algorithm preOrder(v)
    visit(v)
    for each child w of v
        preOrder(w)
```

Preorder Traversal



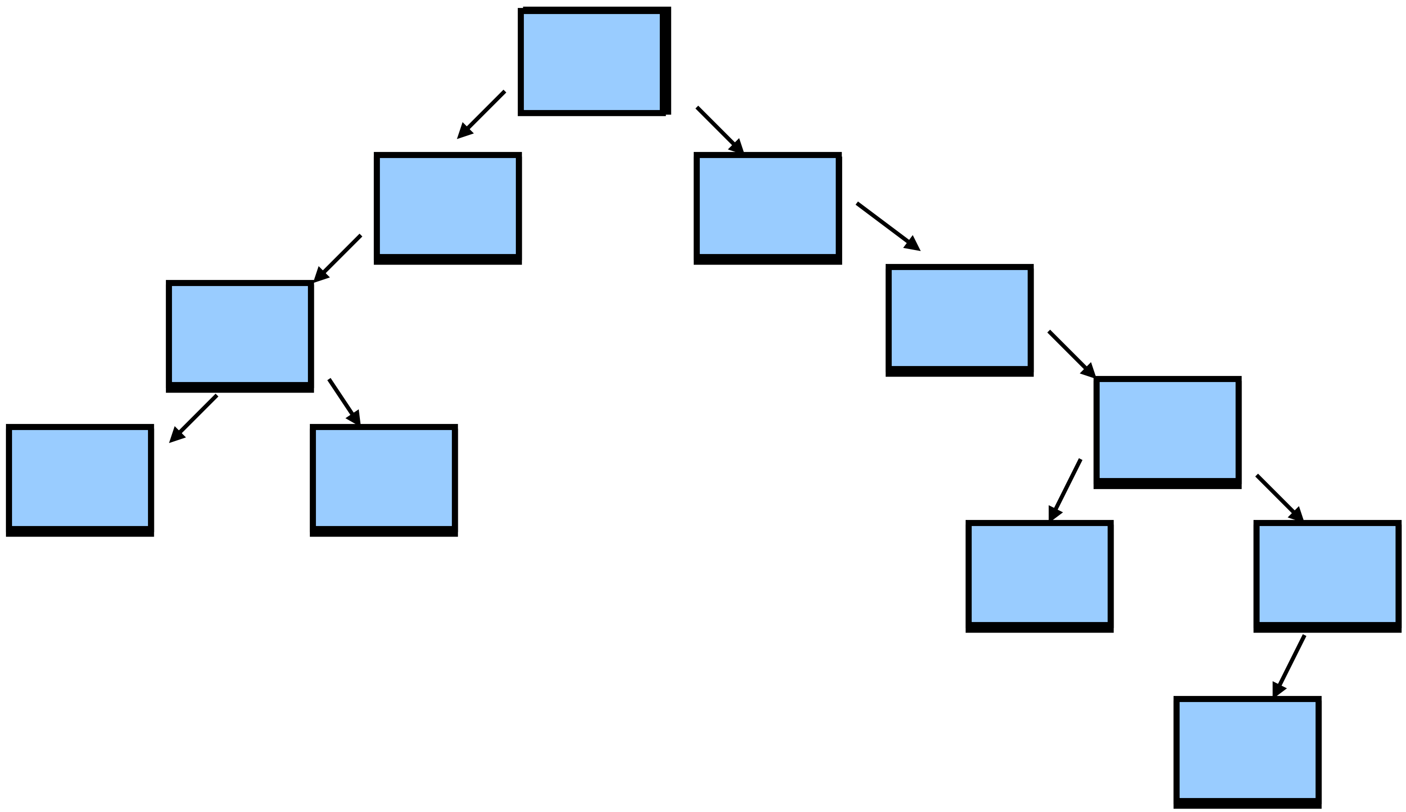
Preorder traversal: node is visited before its descendants

Postorder traversal

1. Process the nodes in all subtrees in their order
2. Process the root

```
Algorithm postOrder(v)
    for each child w of v
        postOrder(w)
    visit(v)
```

Postorder Traversal



Postorder traversal: node is visited before its descendants

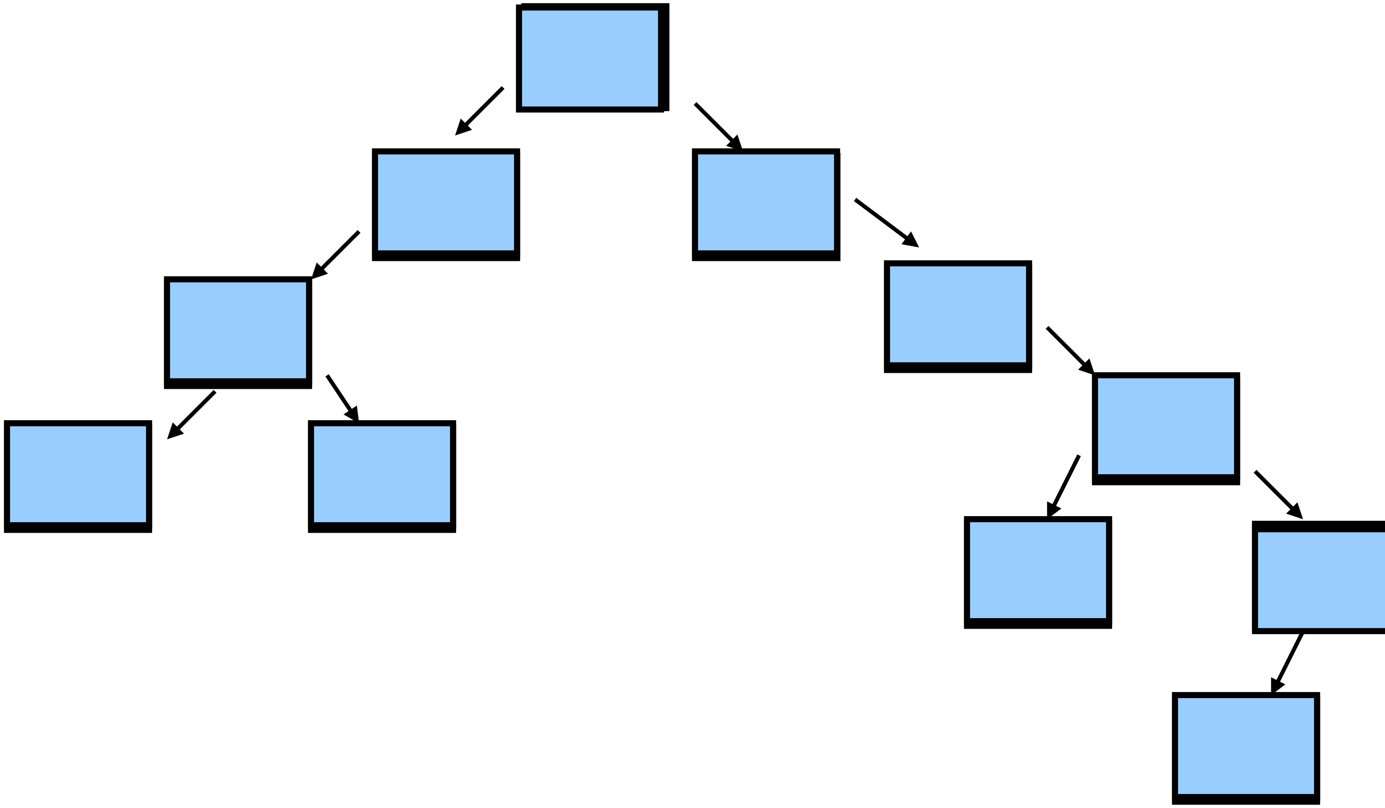
Inorder traversal

1. Process the nodes in the left subtree
2. Process the root
3. Process the nodes in the right subtree

```
Algorithm InOrder(v)
    InOrder(v->left)
    visit(v)
    InOrder(v->right)
```

For simplicity, we consider tree having at most 2 children, though it can be generalized.

Inorder Traversal



Inorder traversal: node is visited after its left subtree

and before its right subtree

Computing Height of Tree

Can be computed using the following idea:

1. The height of a leaf node is 0
2. The height of a node other than the leaf is the maximum of the height of the left subtree and the height of the right subtree plus 1.

   Height(v) = max[height(v→left) + height(v→right)] + 1

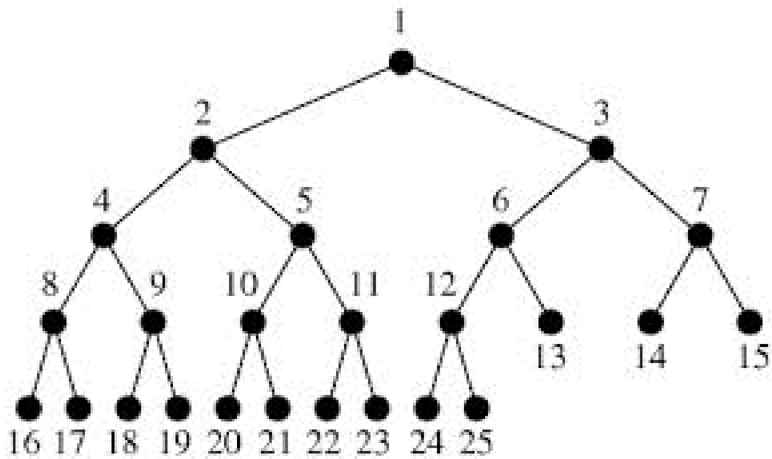
Details left as exercise.

More examples

Which traversal will use if:

1. Want to evaluate the depth of every node ?
2. Given a tree representing arithmetic expression, print it in postfix notation ?
3. Given the directory structure of files, figure out the total memory usage ?
4. Given the directory structure of files, print the complete file names for each file ?

Binary Trees

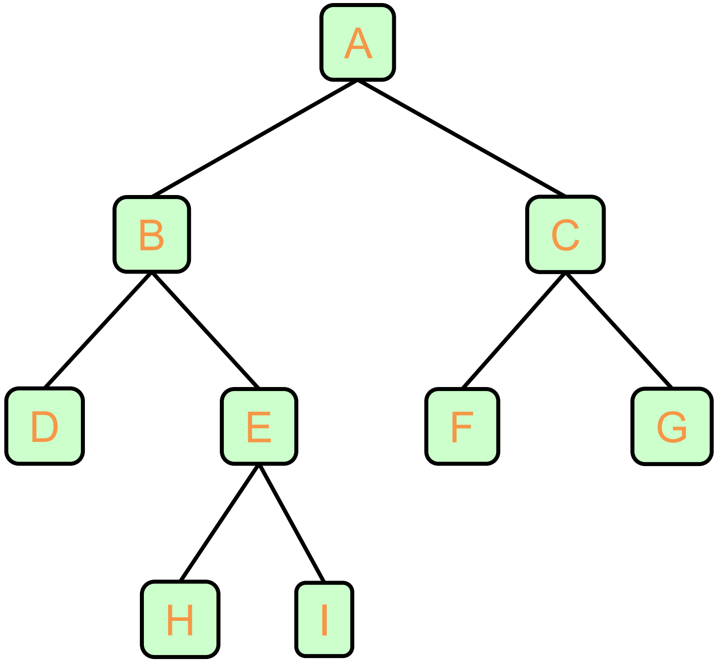


Every node has degree up to 2.
Proper binary tree: each internal node has degree exactly 2.

Binary Tree

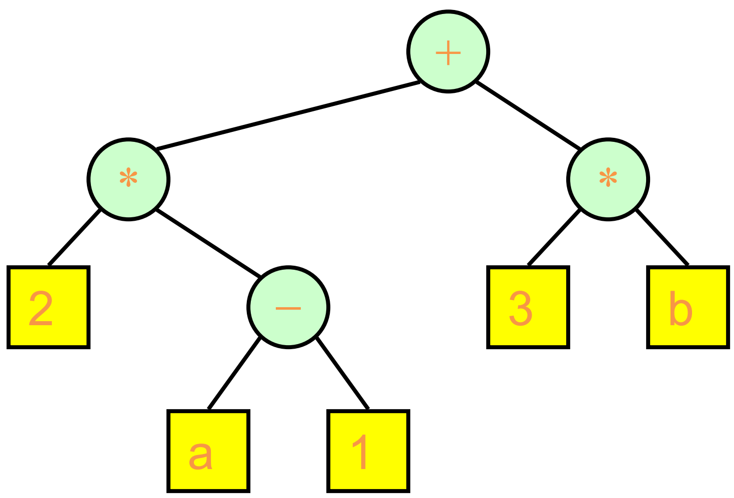
- A *binary tree* is a tree with the following properties:
  - Each internal node has two children
  - The children of a node are an ordered pair
- We call the children of an internal node *left child* and *right child*
- Alternative recursive definition: a binary tree is either
  - a tree consisting of a single node, or
  - a tree whose root has an ordered pair of children, each of which is a disjoint binary tree

- Applications:
  - arithmetic expressions
  - decision processes
  - searching

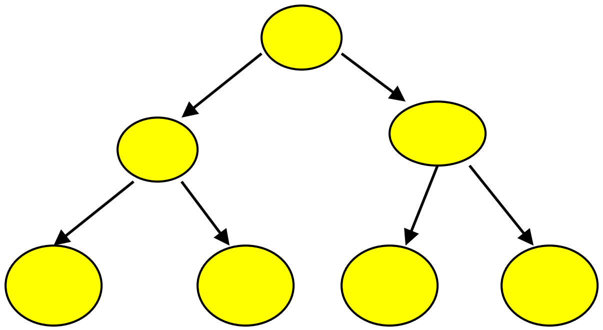


Arithmetic Expression Tree

- Binary tree associated with an arithmetic expression
  - internal nodes: operators
  - leaves: operands
- Example: arithmetic expression tree for the expression $(2 * (a - 1) + (3 * b))$



How many leaves L does a complete binary tree of height h have?



The number of leaves at depth d $= 2^d$

If the height of the tree is h it has 2^h leaves.

$L = 2^{h.}$

What is the height h of a complete binary tree with L leaves?

leaves = 1	height = 0
leaves = 2	height = 1
leaves = 4	height = 2
leaves = L	height = Log_2L

Since $L = 2^h$

$\log_2 L = \log_2 2^h$

$h = \log_2 L$

The number of internal nodes of a complete binary tree of height *h* is ?

Internal nodes = 0 height = 0

Internal nodes = 1 height = 1

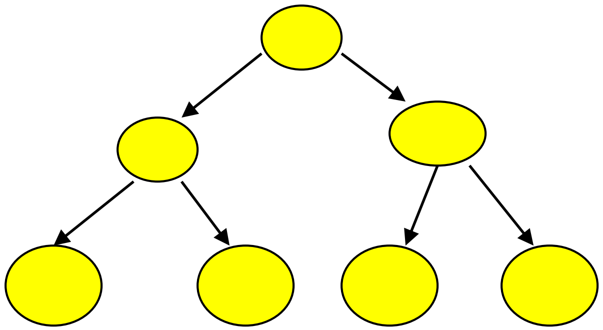
Internal nodes = 1 + 2 height = 2

Internal nodes = 1 + 2 + 4 height = 3

$1 + 2 + 2^2 + . . . + 2^{h-1} = 2^h - 1$ Geometric series

Thus, a complete binary tree of height = h has 2^h-1 internal nodes.

The number of nodes n of a complete binary tree of height *h* is ?



nodes = 1 height = 0

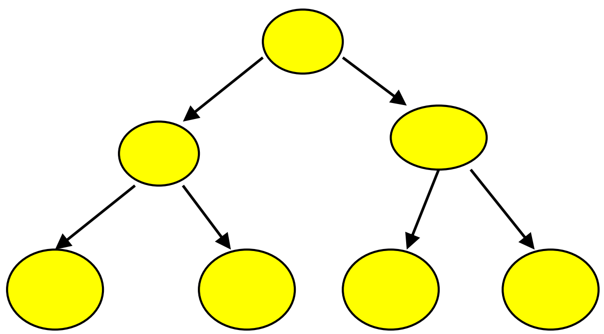
nodes = 3 height = 1

nodes = 7 height = 2

nodes = 2^{h+1}- 1 height = h

Since L = 2^h
and since the number of internal nodes = 2^{h}-1 the total number of nodes n = $2^h + 2^{h} - 1 = 2(2^h) – 1 = 2^{h+1} - 1$.

If the number of nodes is n then what is the height?



nodes = 1	height = 0
nodes = 3	height = 1
nodes = 7	height = 2
nodes = n	height = $\text{Log}_2(n+1) - 1$

Since $n = 2^{h+1}-1$

$n + 1 = 2^{h+1}$

$\text{Log}_2(n+1) = \text{Log}_2 2^{h+1}$

$\text{Log}_2(n+1) = h+1$

$h = \text{Log}_2(n+1) - 1$

What if the tree is not complete (but proper) ?

Height could lie in the range [log n, n/2]

Number of leaves = Number of internal nodes + 1

BinaryTree ADT

- The BinaryTree ADT extends the Tree ADT, i.e., it inherits all the methods of the Tree ADT
- Additional methods:
  - node leftChild(p)
  - node rightChild(p)
  - node sibling(p)
- Update methods may be defined by data structures implementing the BinaryTree ADT