# CS 2420: Tree

Dr. Tsung-Wei (TW) Huang

Department of Electrical and Computer Engineering

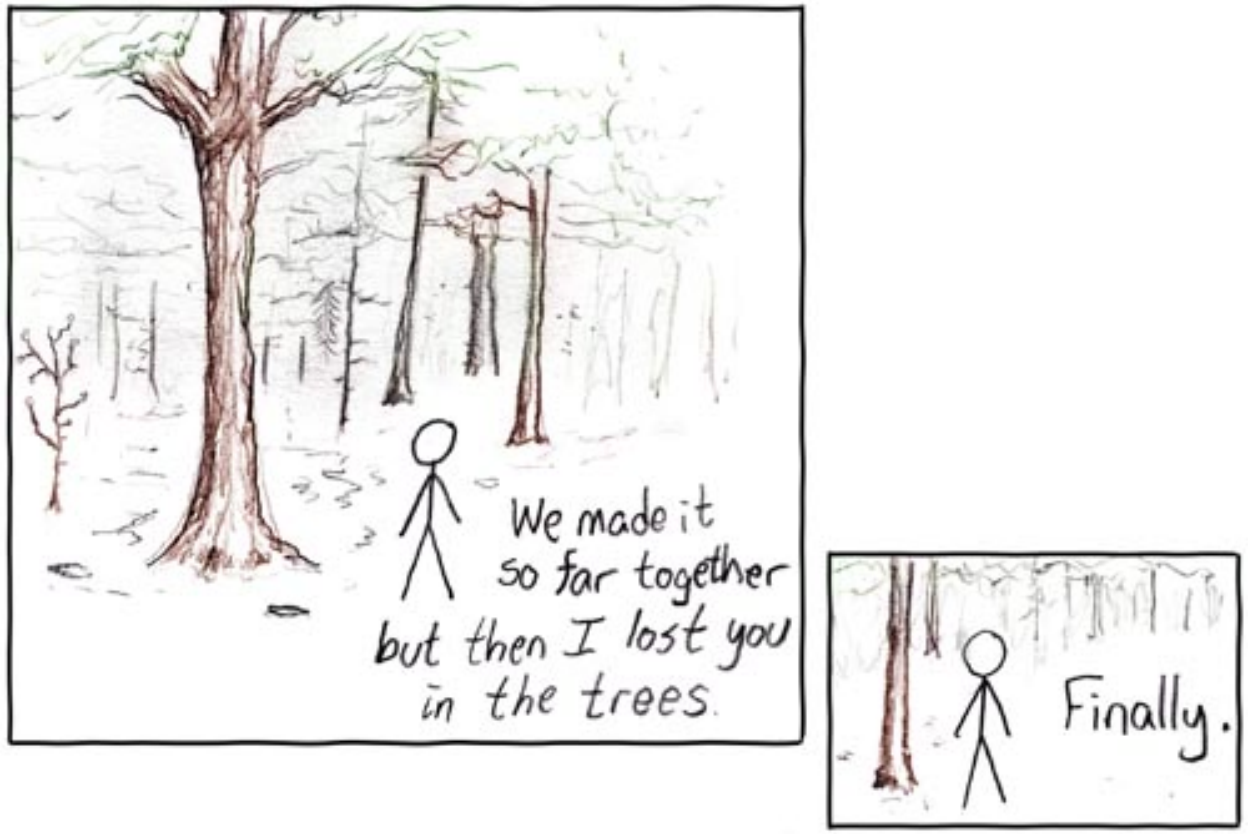
University of Utah, Salt Lake City, UT

# Outline

- In this topic, we will cover:
  - Definition of a tree data structure and its components
  - Concepts of:
    - Root, internal, and leaf nodes
    - Parents, children, and siblings
    - Paths, path length, height, and depth
    - Ancestors and descendants
    - Ordered and unordered trees
    - Subtrees
- Examples
    - XHTML and CSS

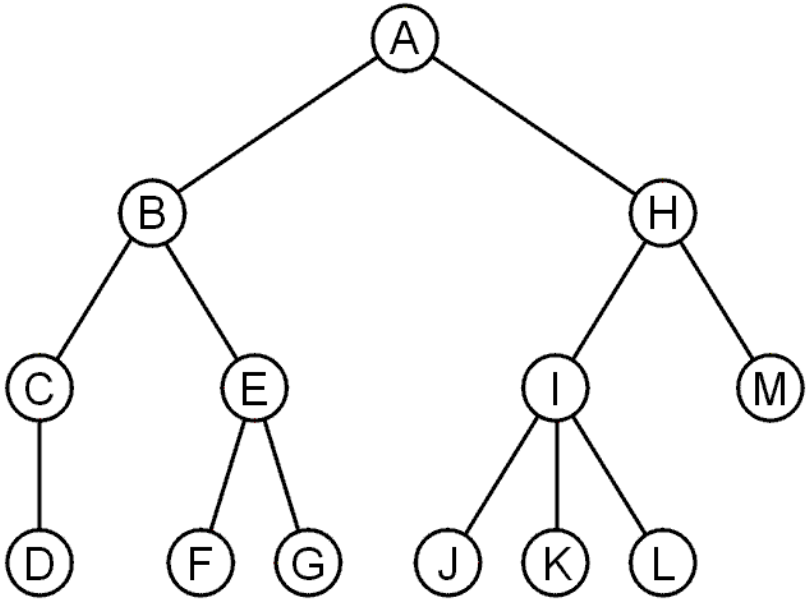
# Abstract Deque

- Trees are the first data structure different from what you've seen in your first-year programming courses



# Tree

- 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
    - Each node, other than the root, has exactly one node pointing to it



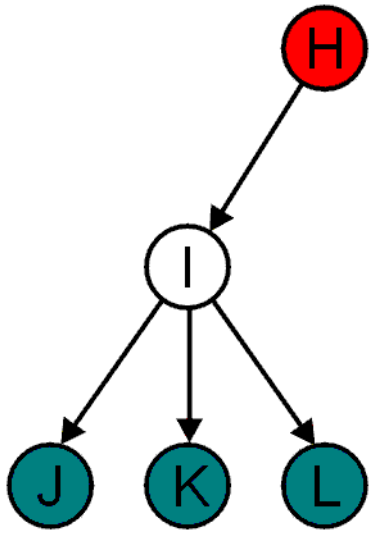
# Terminology

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 I



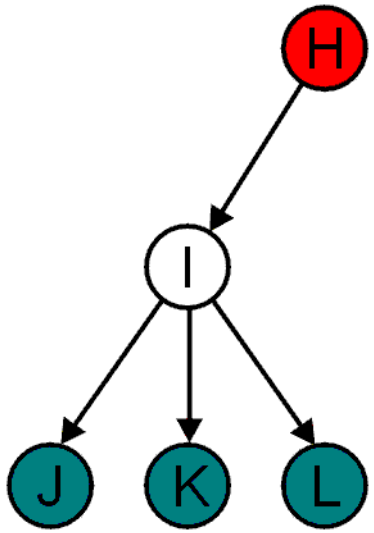
# Terminology

The *degree* of a node is defined as the number of its children:

$$\deg(\mathrm{I}) = 3$$

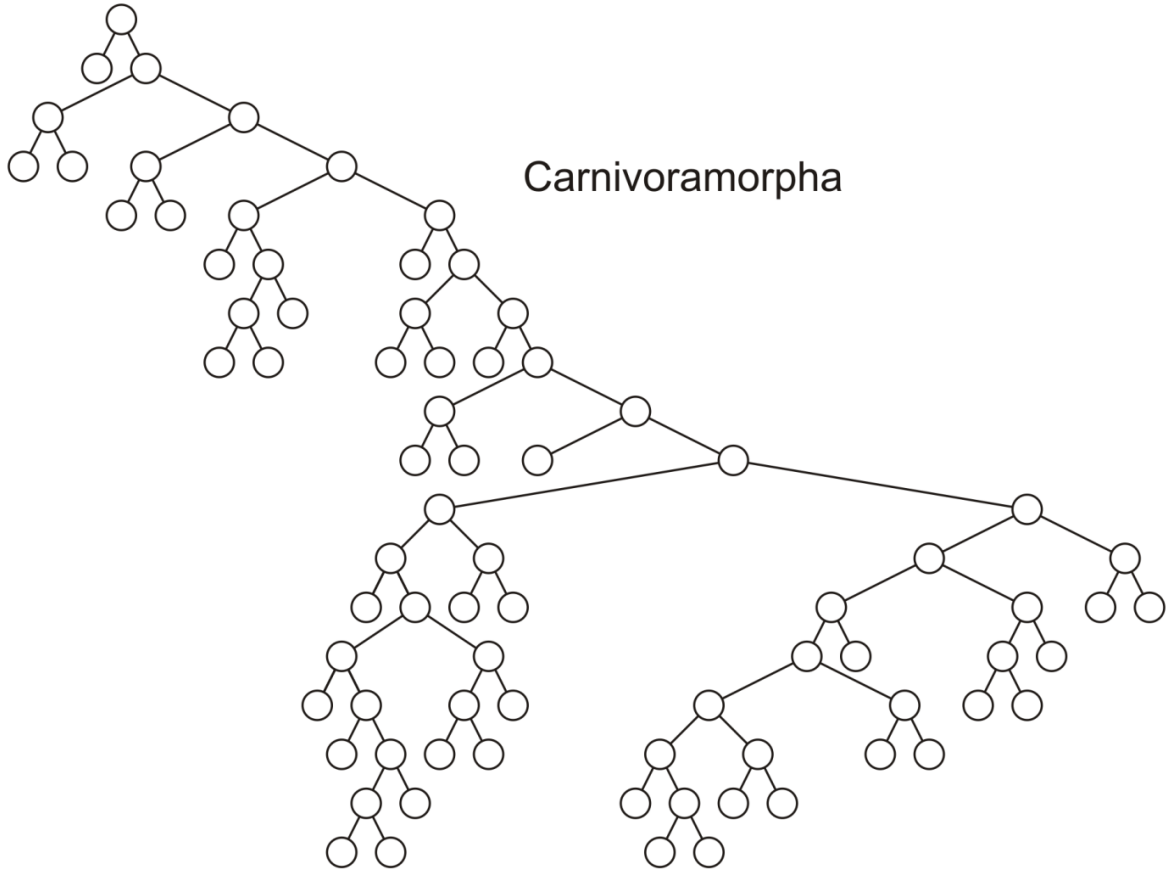
Nodes with the same parent are *siblings*

- J, K, and L are siblings



# Terminology

Phylogenetic trees have nodes with degree 2 or 0:

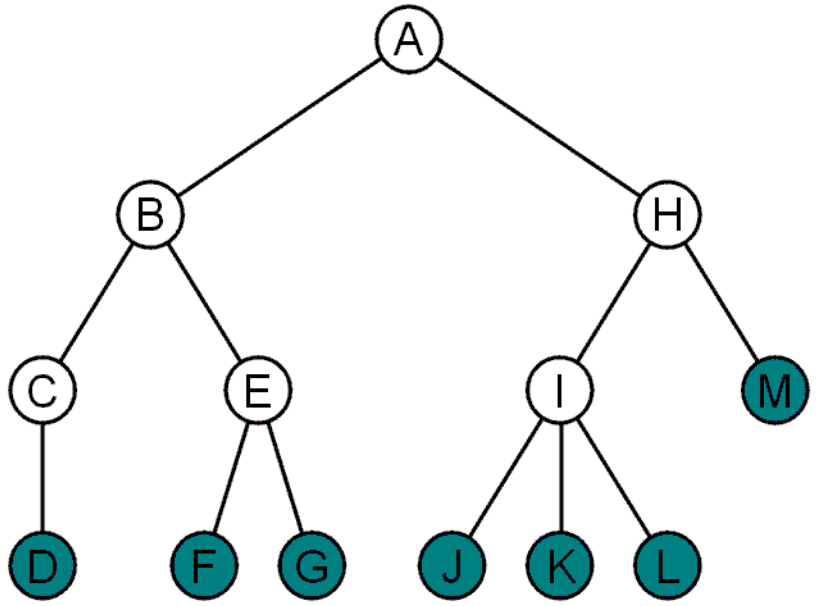



Wesley-Hunt, G. D.; Flynn, J. J. “Phylogeny of the Carnivora: basal relationships among the Carnivoramorphans, and assessment of the position of ‘Miacoidea’

# Terminology

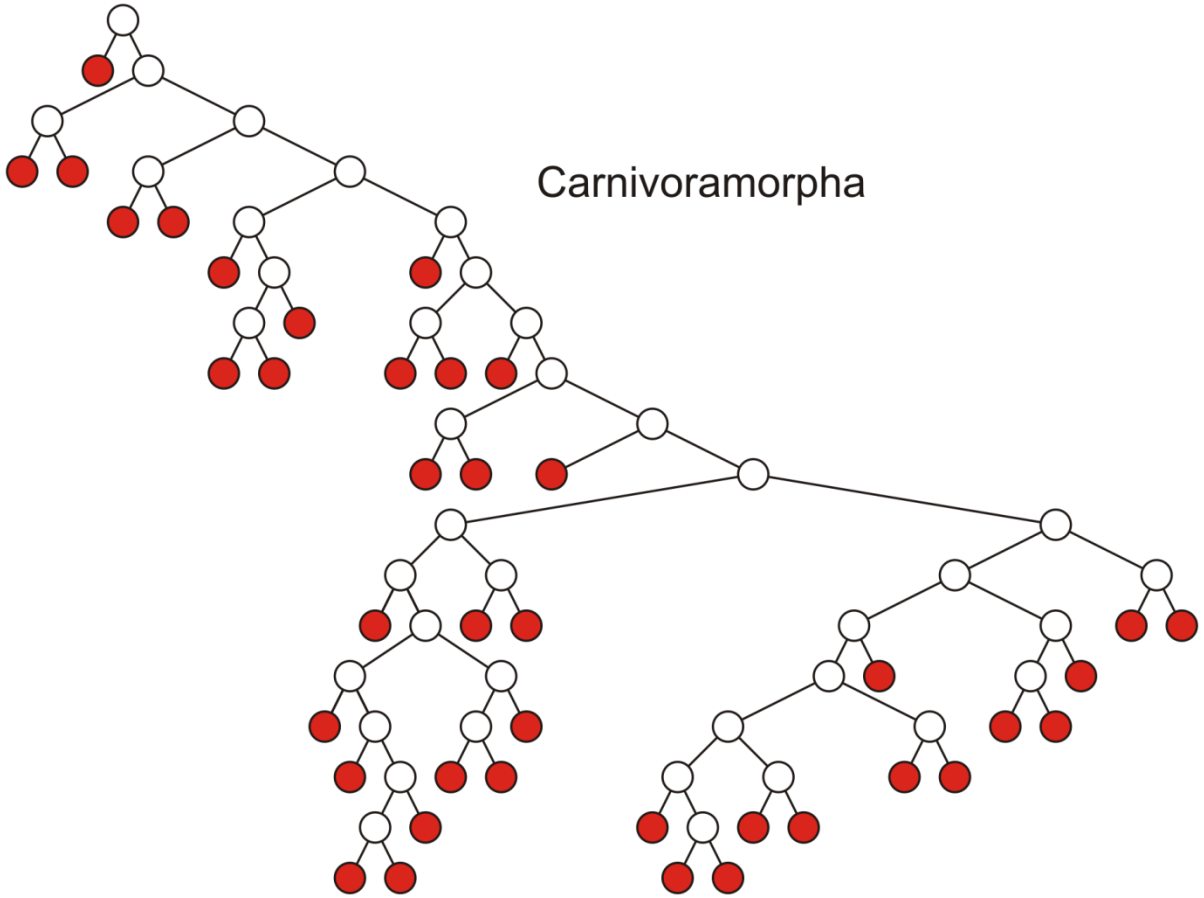
Nodes with degree zero are also called *leaf nodes*

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



# Terminology

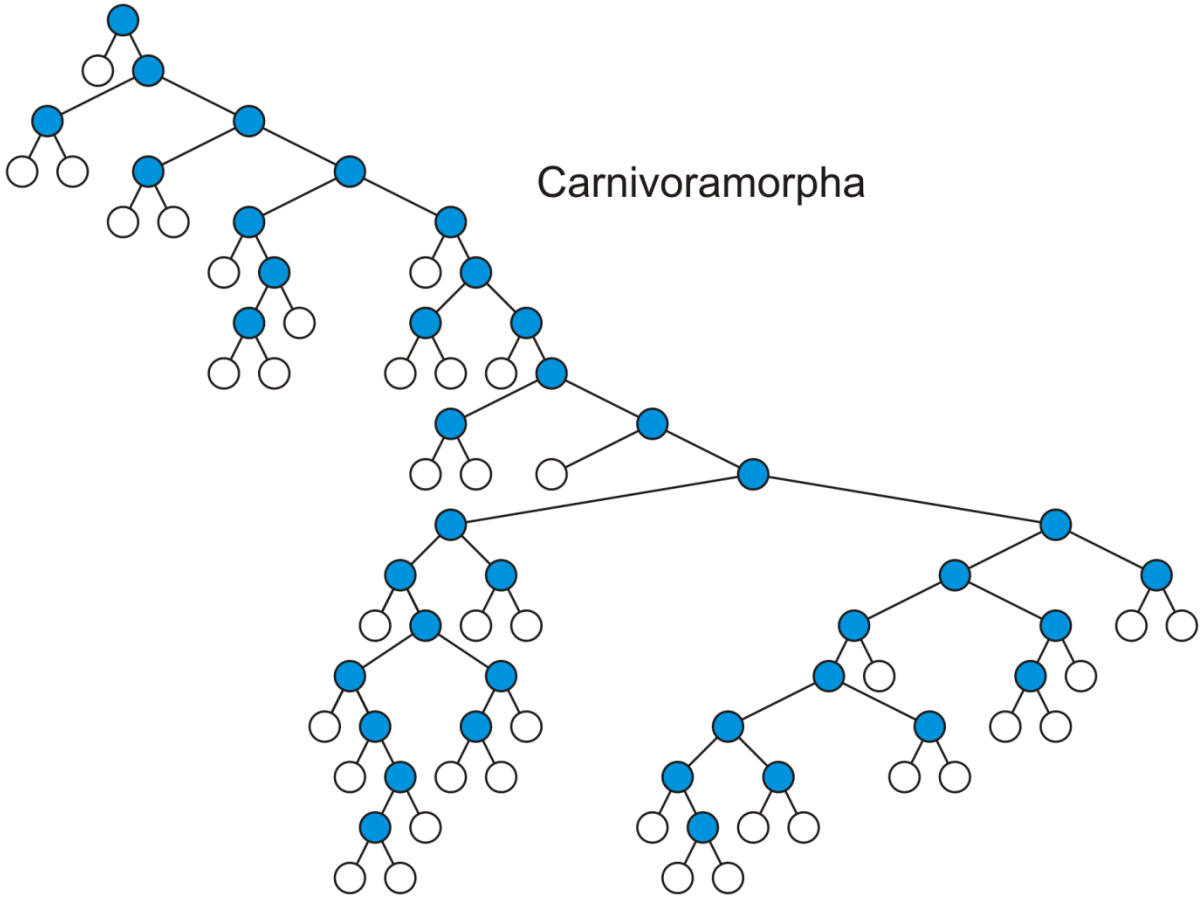
Leaf nodes:




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:

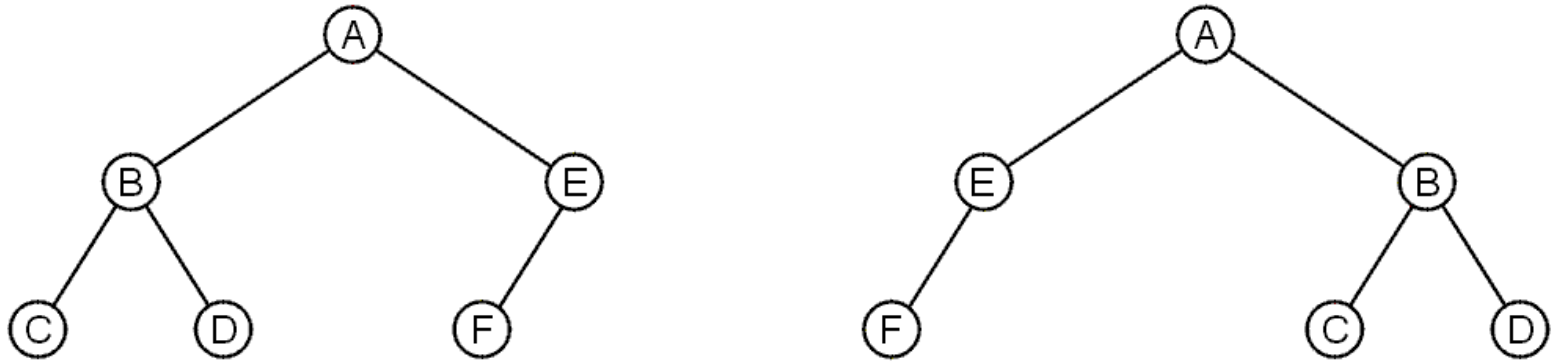



Wesley-Hunt, G. D.; Flynn, J. J. “Phylogeny of the Carnivora: basal relationships among the Carnivoramorphans, and assessment of the position of ‘Miacoidea’

# Terminology

These trees are equal if the order of the children is ignored

- *unordered trees*

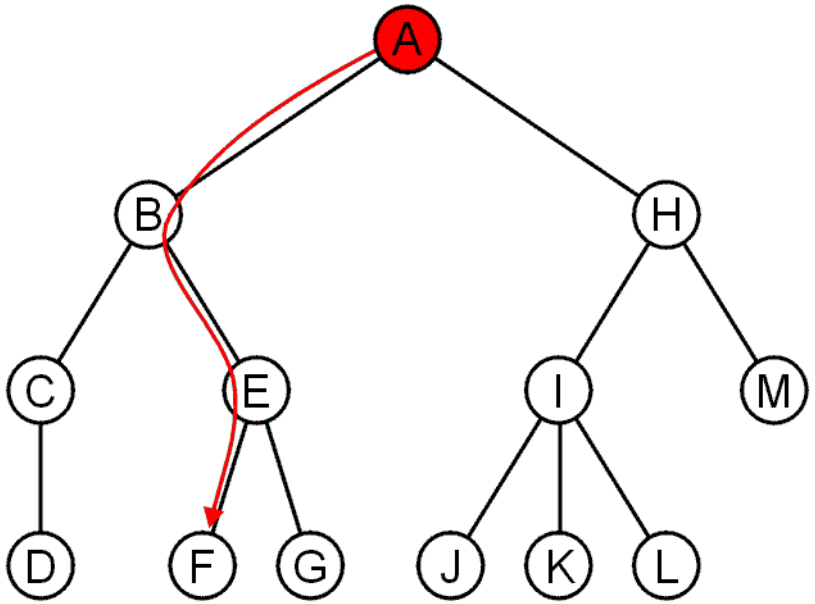



They are different if order is relevant (*ordered trees*)

- 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

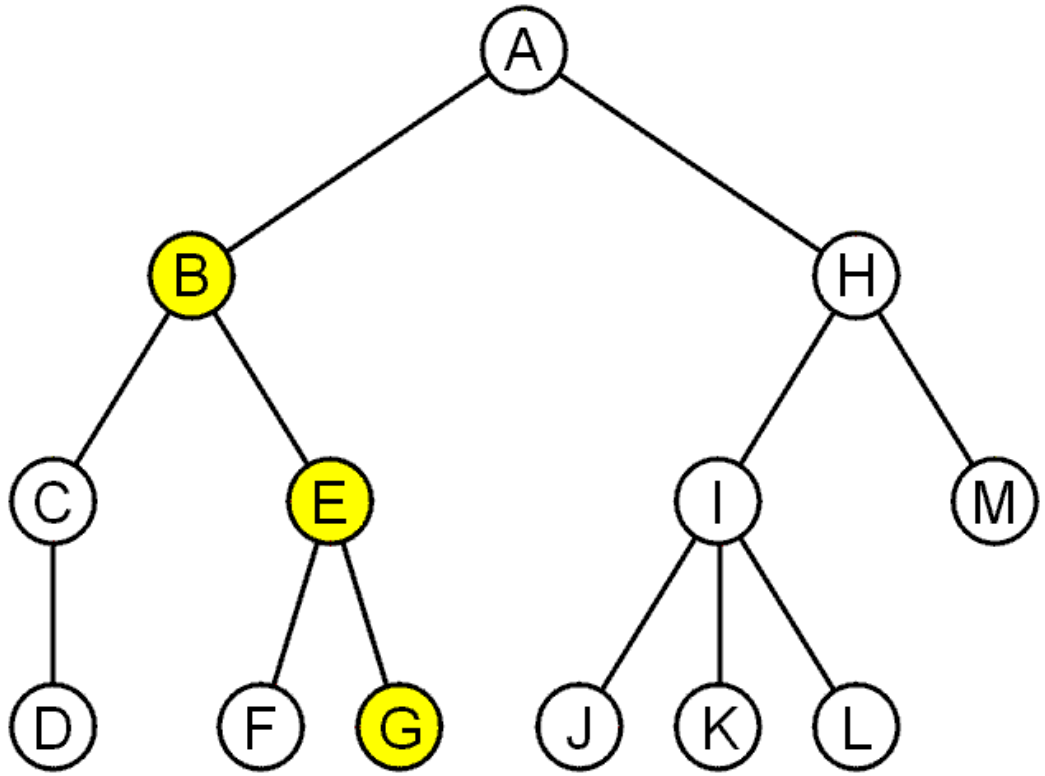
A path is a sequence of nodes

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

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

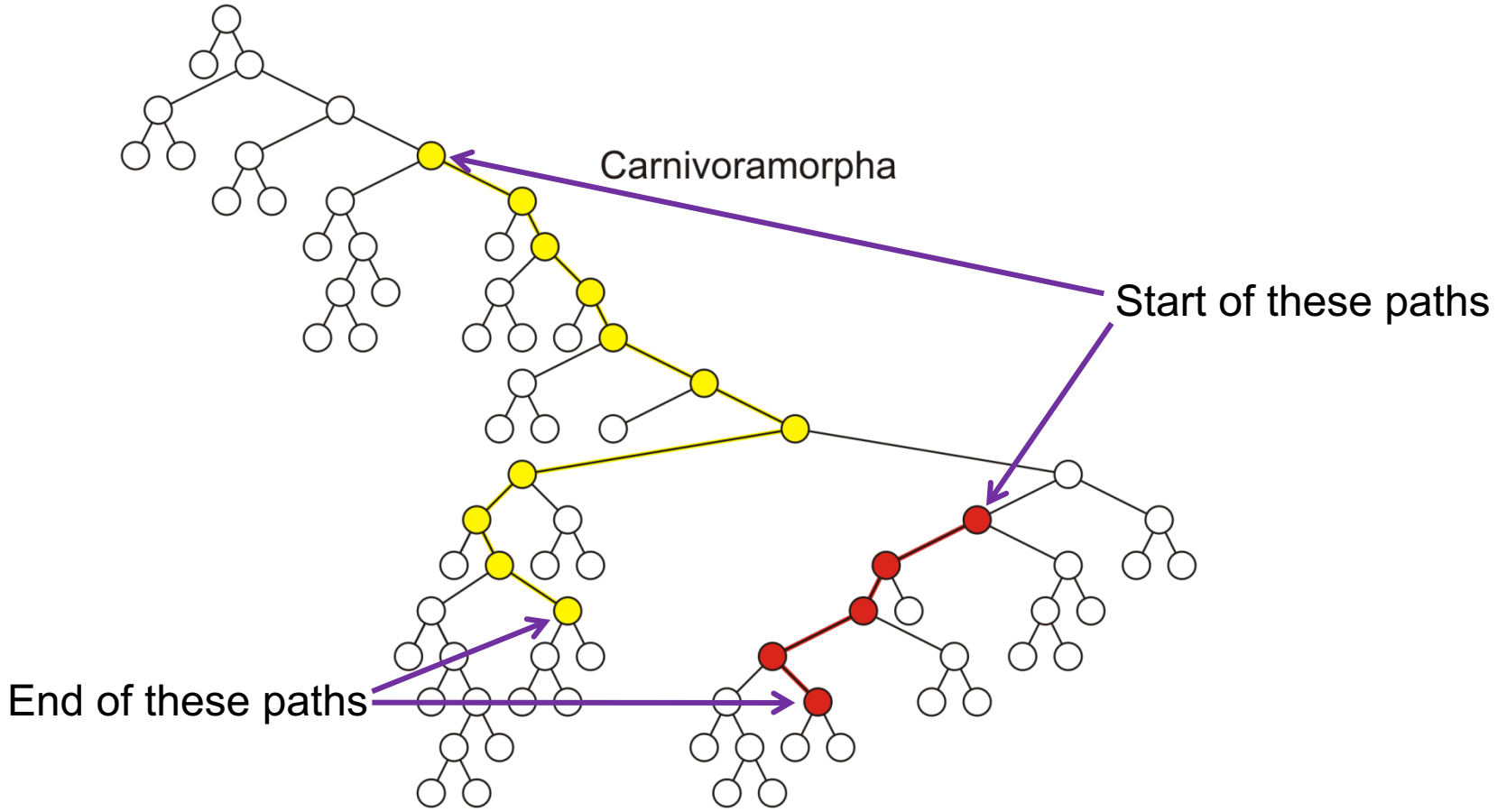
The length of this path is $n$

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



# Terminology

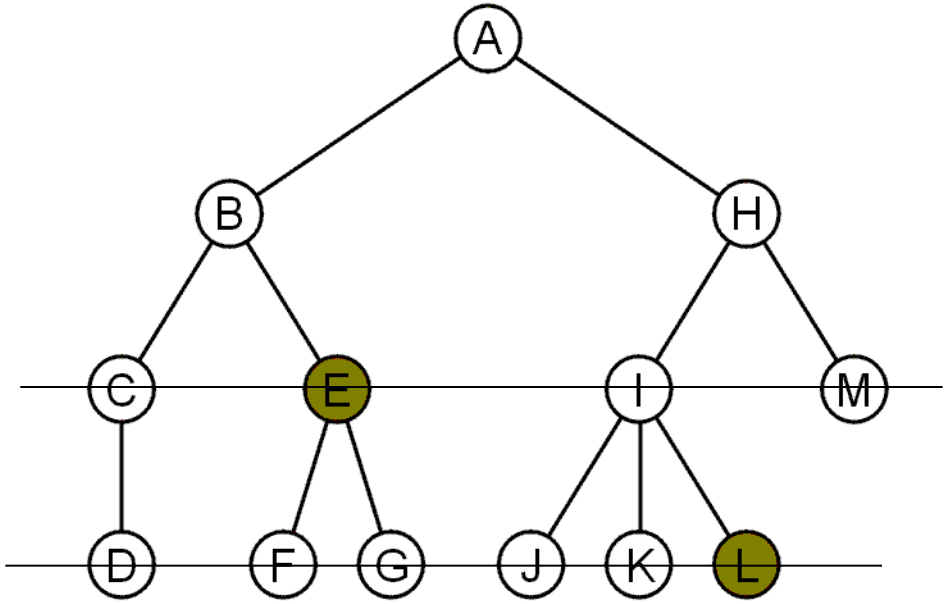
Paths of length 10 (11 nodes) and 4 (5 nodes)




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

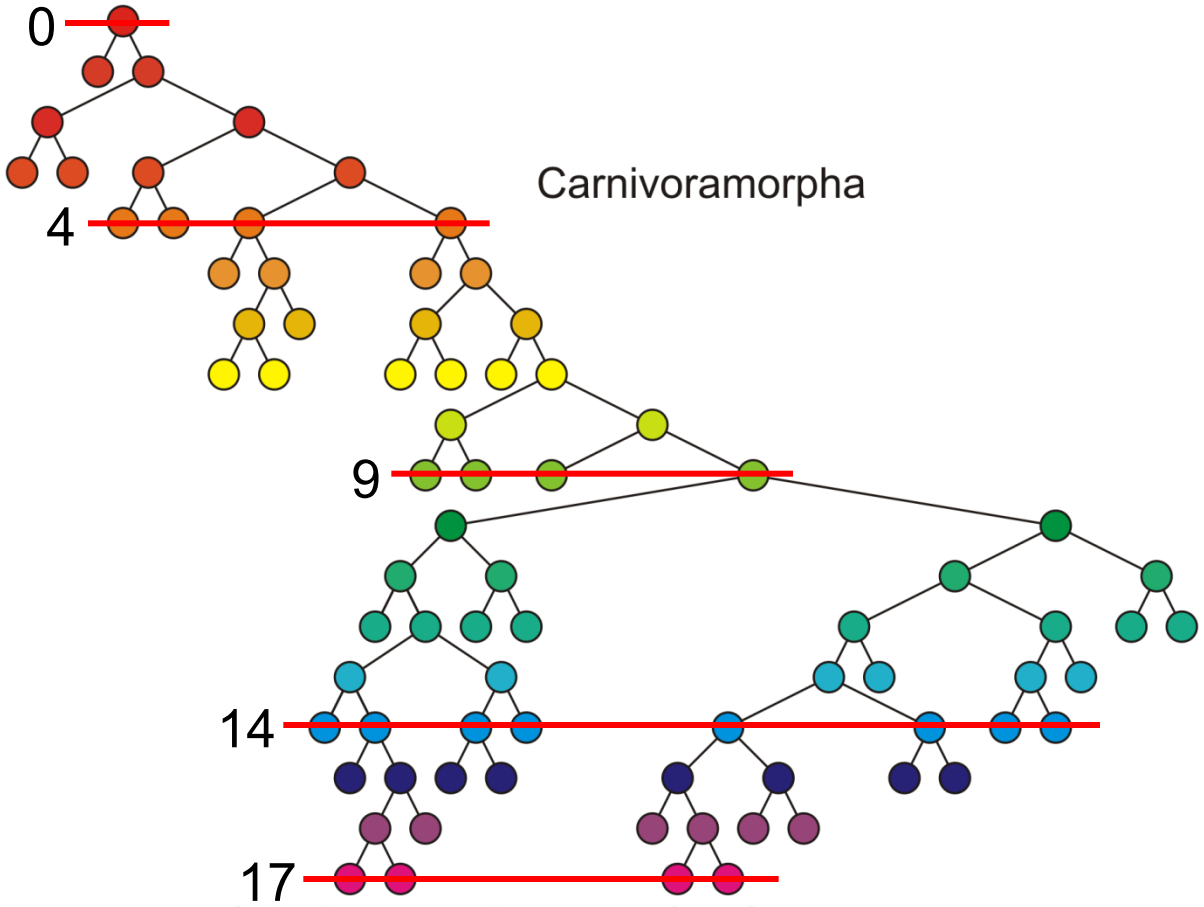
# Terminology

- 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

- Depth 17



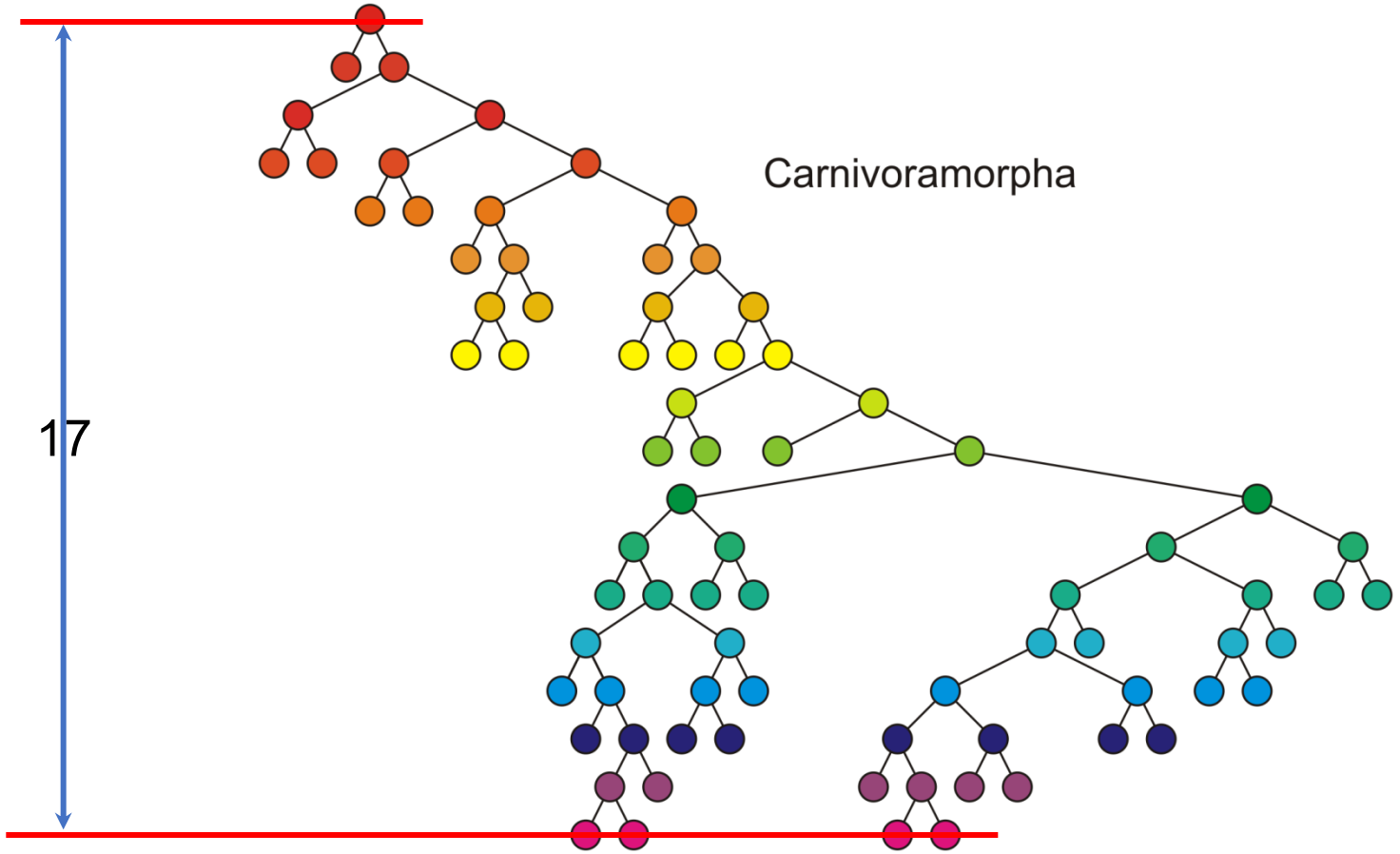

Wesley-Hunt, G. D.; Flynn, J. J. “Phylogeny of the Carnivora: basal relationships among the Carnivoramorphans, and assessment of the position of ‘Miacoidea’

# 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



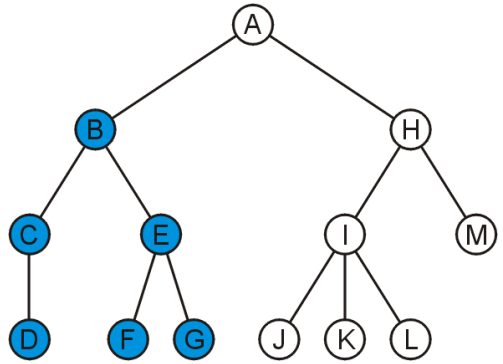

Wesley-Hunt, G. D.; Flynn, J. J. “Phylogeny of the Carnivora: basal relationships among the Carnivoramorphans, and assessment of the position of ‘Miacoidea’

# Terminology

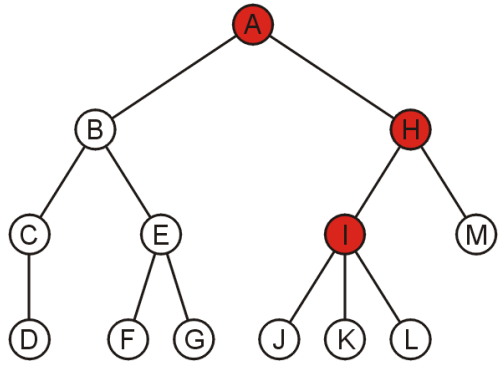
- If a path exists from node $a$ to node $b$:
  - $a$ is an *ancestor* of $b$
  - $b$ is a *descendent* of $a$
- Thus, a node is both an ancestor and a descendant of itself
  - We can add the adjective *strict* to exclude equality: $a$ is a *strict descendent* of $b$ if $a$ is a descendant of $b$ but $a \neq b$
- The root node is an ancestor of all nodes

# Terminology

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

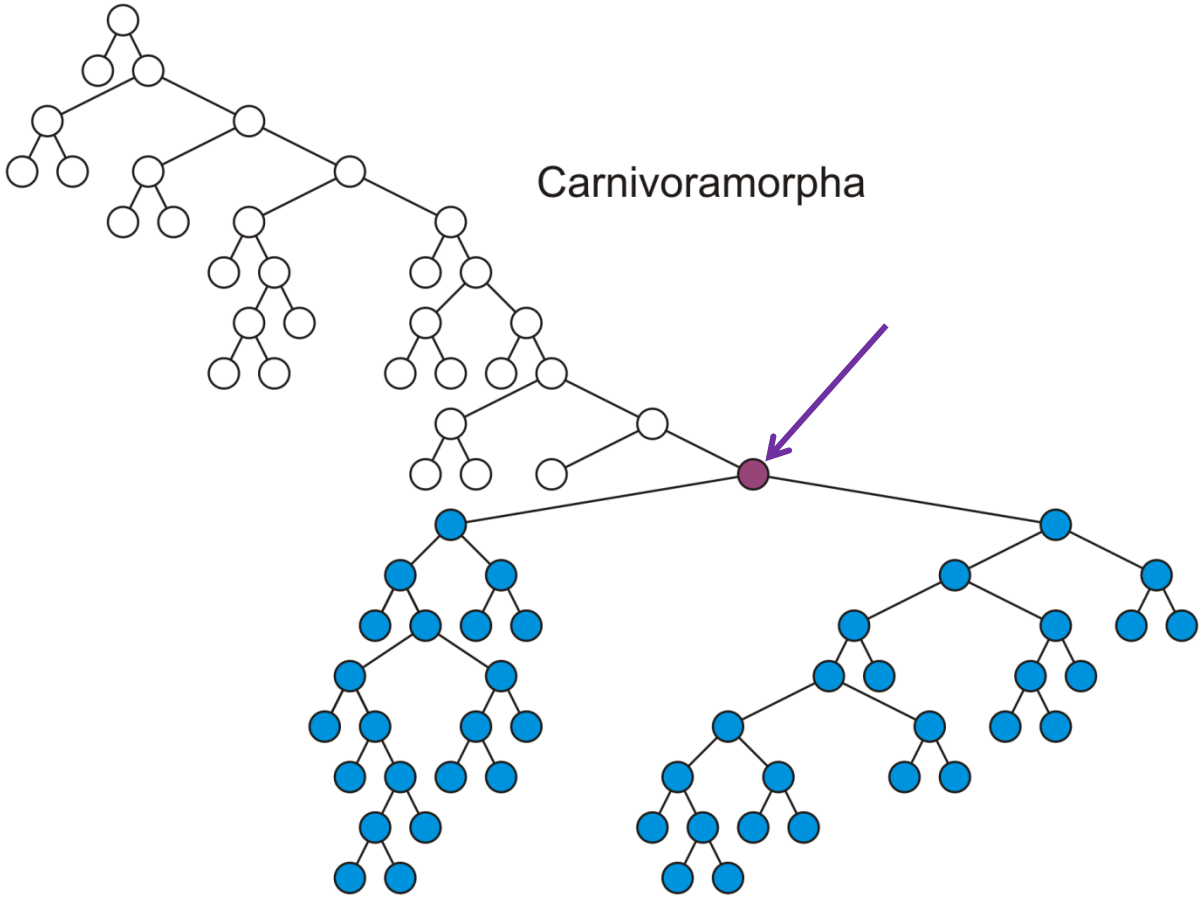



- The ancestors of node I are I, H, and A:



# Terminology

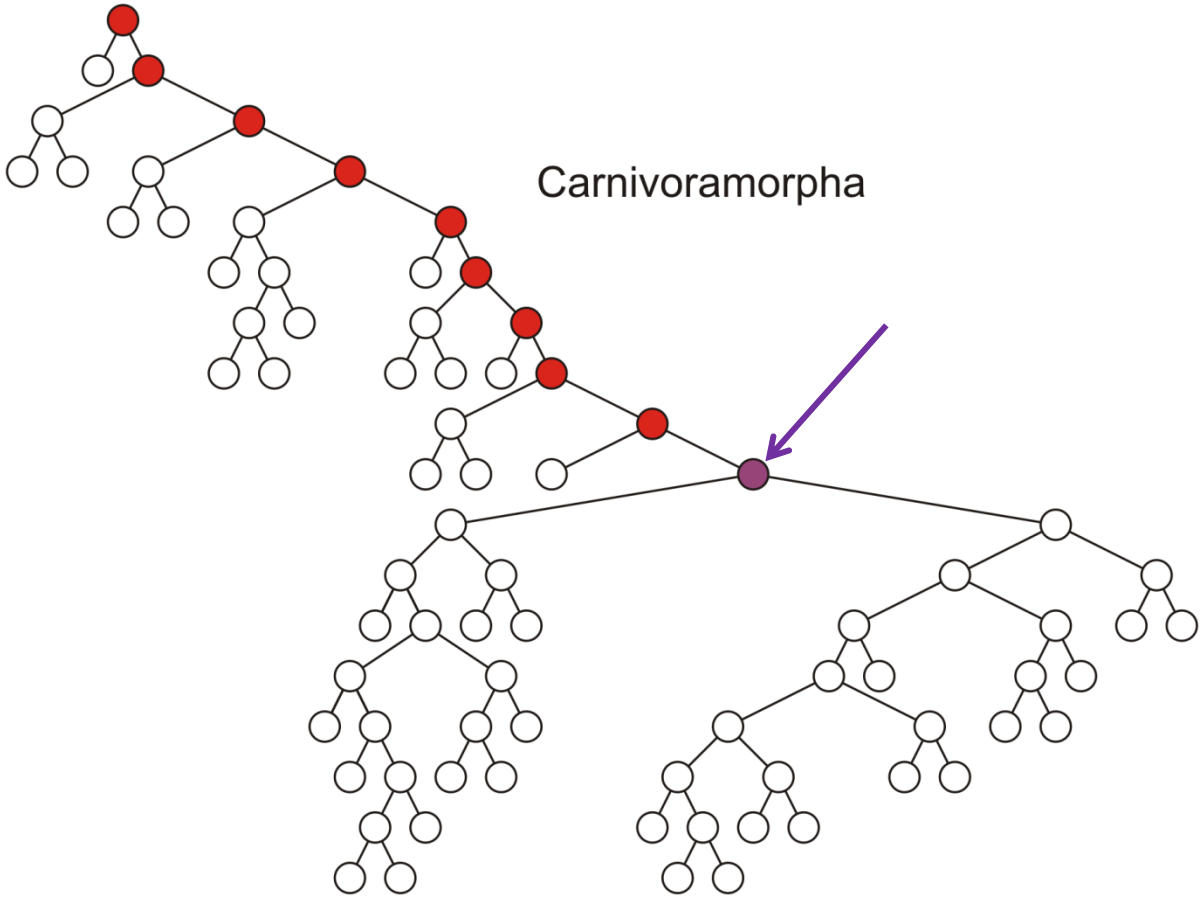
- All descendants (including itself) of the indicated node




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

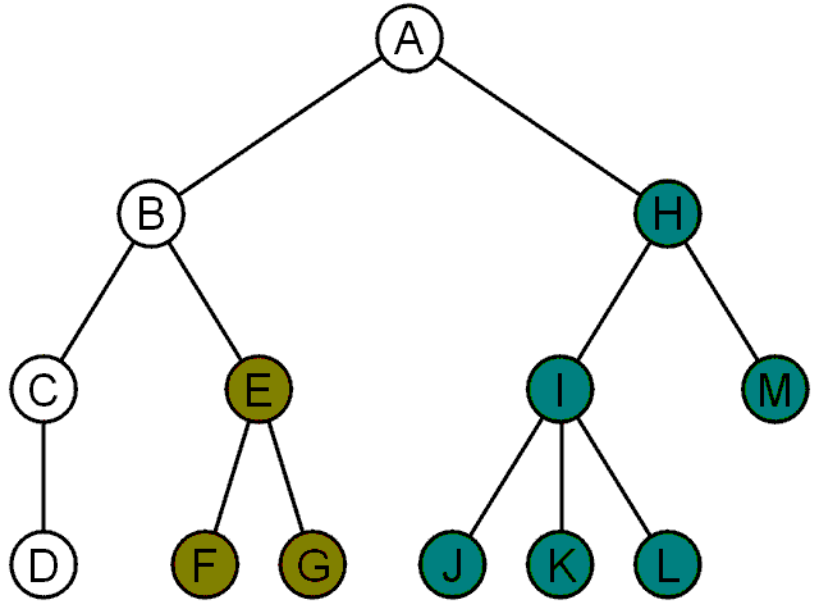



Wesley-Hunt, G. D.; Flynn, J. J. “Phylogeny of the Carnivora: basal relationships among the Carnivoramorphans, and assessment of the position of ‘Miacoidea’

# 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 and CSS

- The XML of XHTML has a tree structure

- Cascading Style Sheets (CSS) use the tree structure to modify the display of HTML

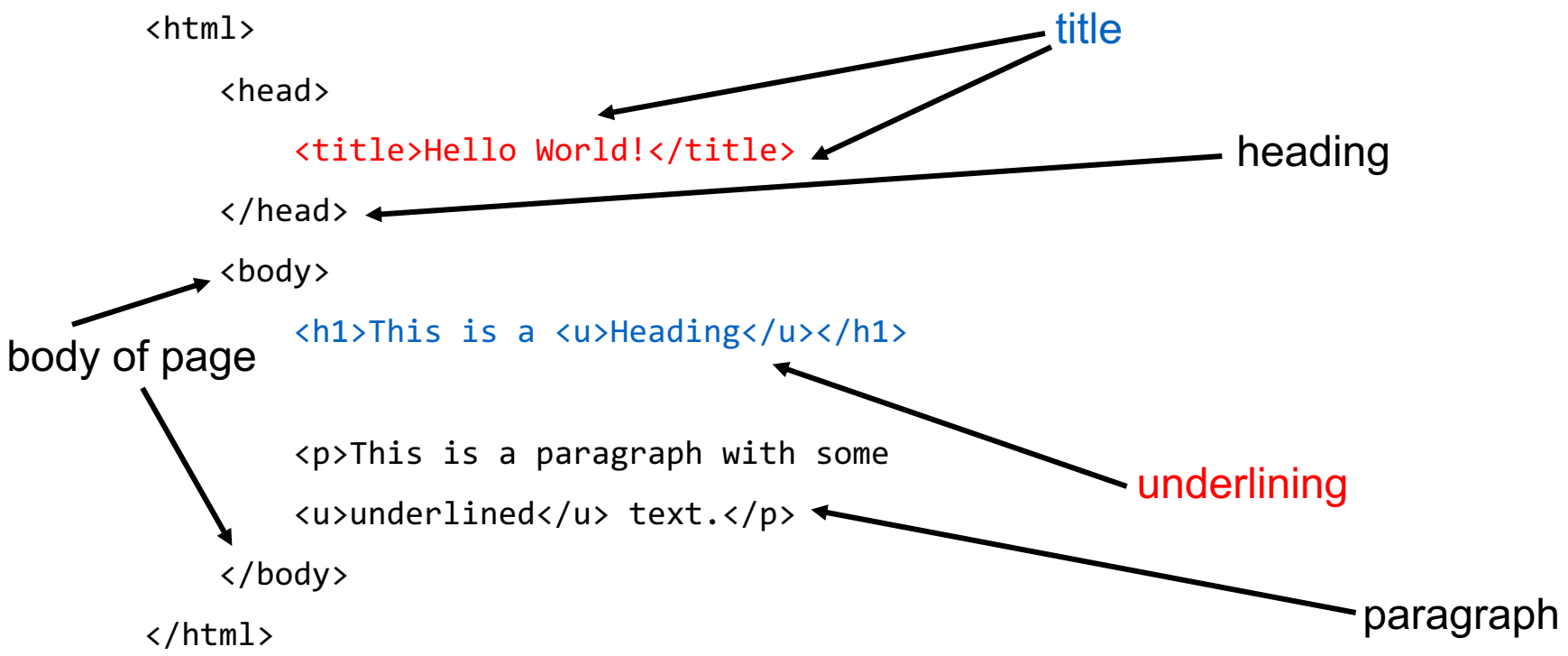
# Example: XHTML and CSS

- 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 and CSS

- Consider the following XHTML document

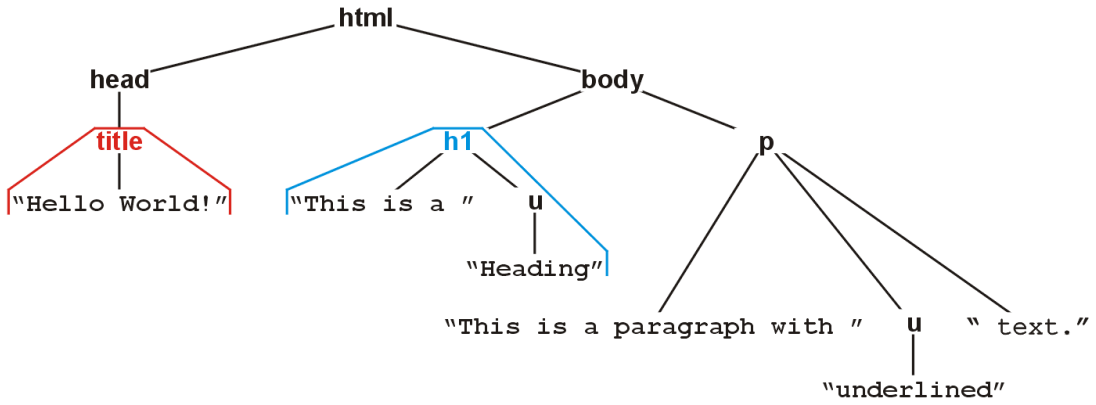



```
<html>
    <head>
        <title>Hello World!</title>
    </head>
    <body>
        <h1>This is a <u>Heading</u></h1>

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

# Example: XHTML and CSS

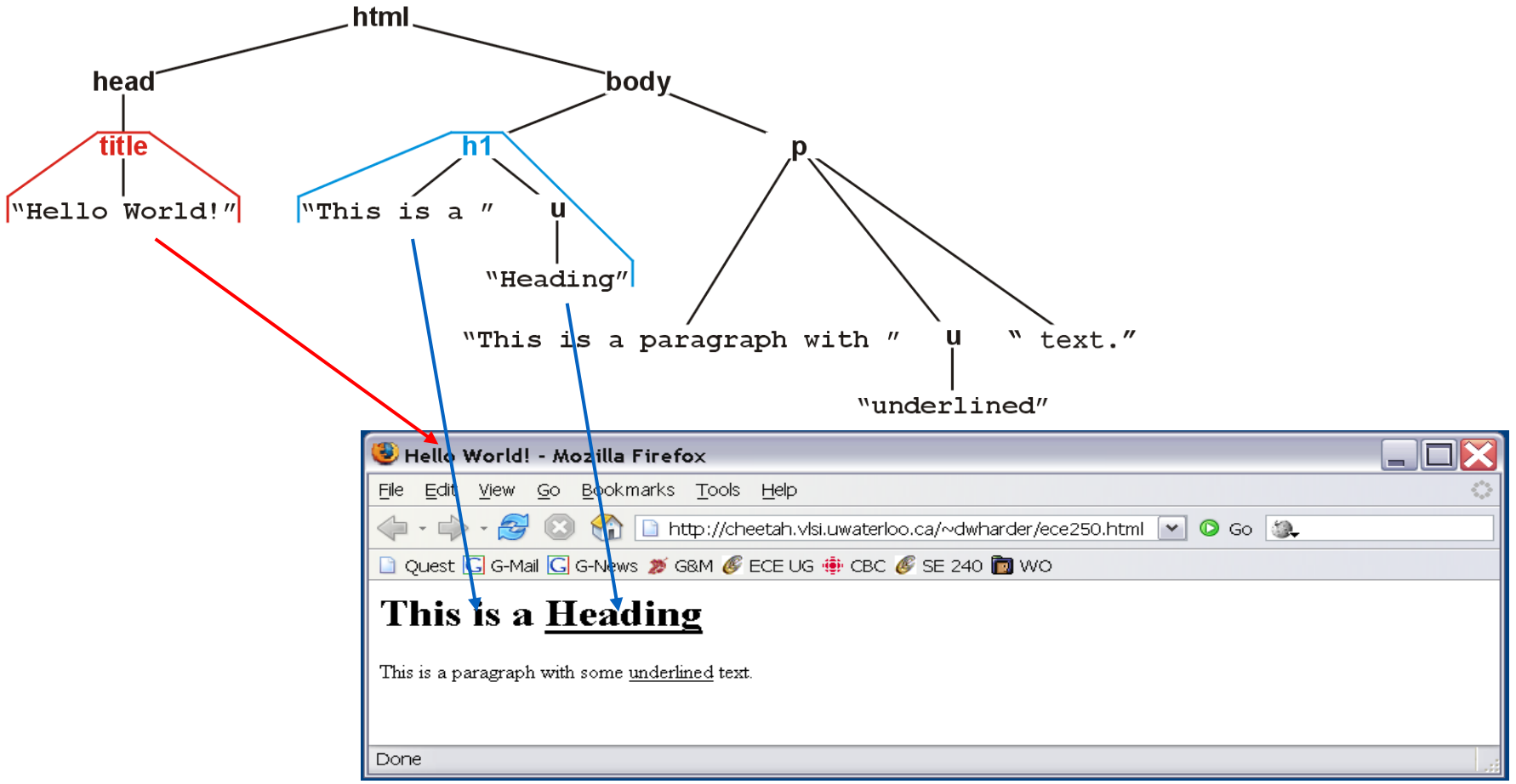
- The nested tags define a tree rooted at the HTML tag

```
<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 and CSS

- Web browsers render this tree as a web page



# Example: XHTML and CSS

XML tags `<tag>...</tag>` must be nested

For example, to get the following effect:

1 2 3 **4 5 6** **7 8 9**

you may use

```
<u>1 2 3 <b>4 5 6</b></u><b> 7 8 9</b>
```

You may not use:

```
<u>1 2 3 <b>4 5 6</u> 7 8 9</b>
```

# Example: XHTML and CSS

Cascading Style Sheets (CSS) make use of this tree structure to describe how HTML should be displayed

- For example:

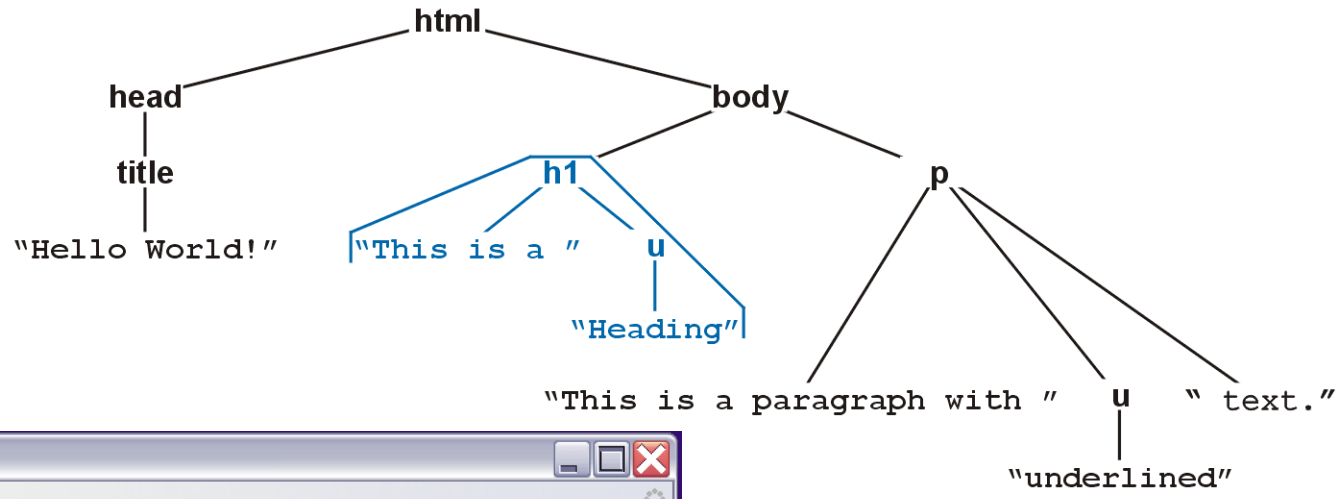
```
<style type="text/css">
   h1 { color:blue; }
</style>
```

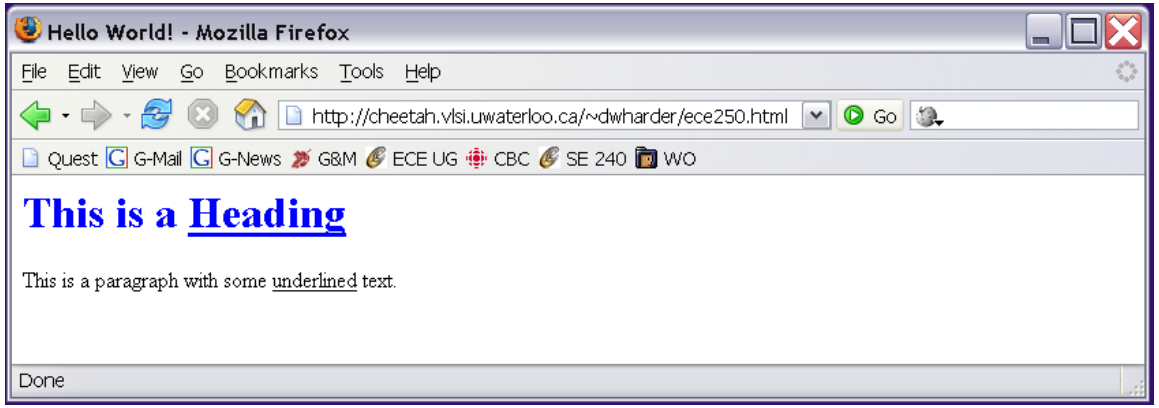
indicates all text/decorations descendant from an `h1` header should be blue

# Example: XHTML and CSS

For example, this style renders as follows:

```
<style type="text/css">
   h1 { color:blue; }
</style>
```

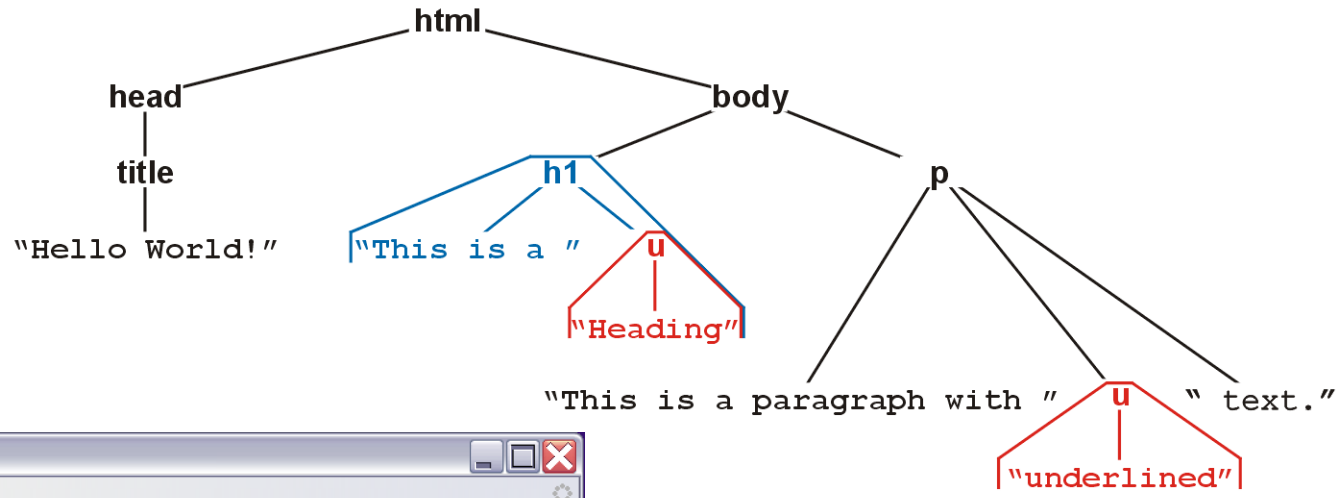


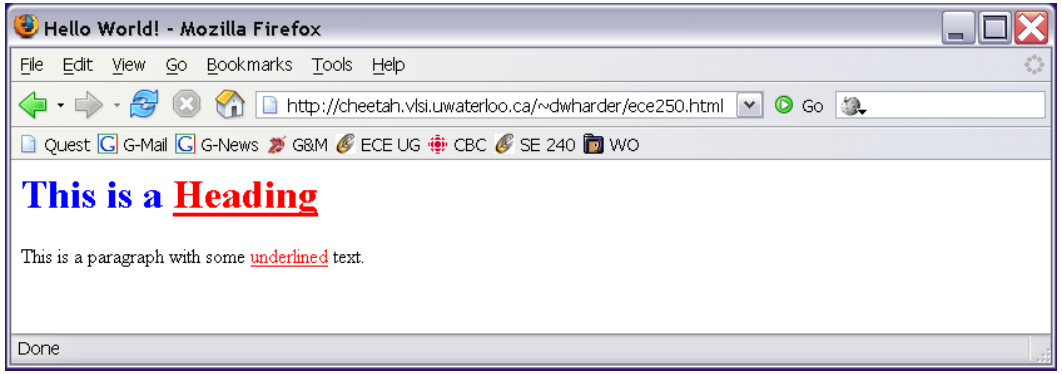



# Example: XHTML and CSS

For example, this style renders as follows:

```
<style type="text/css">
   h1 { color:blue; }
   u { color:red; }
</style>
```



# Example: XHTML and CSS

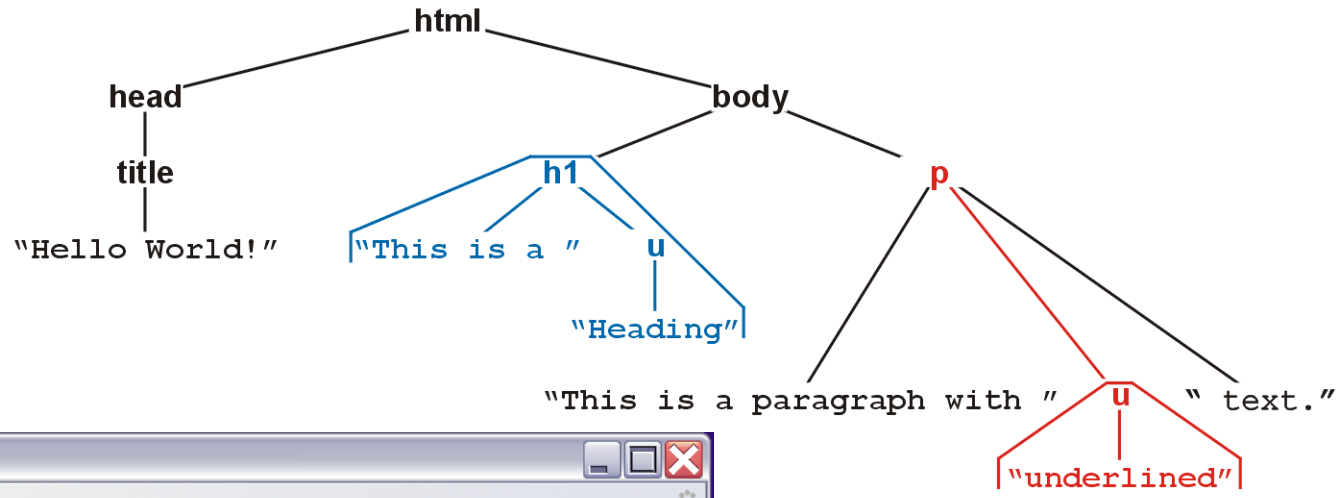
- Suppose you don’t want underlined items in headers (h1) to be red
  - More specifically, suppose you want any underlined text within paragraphs to be red

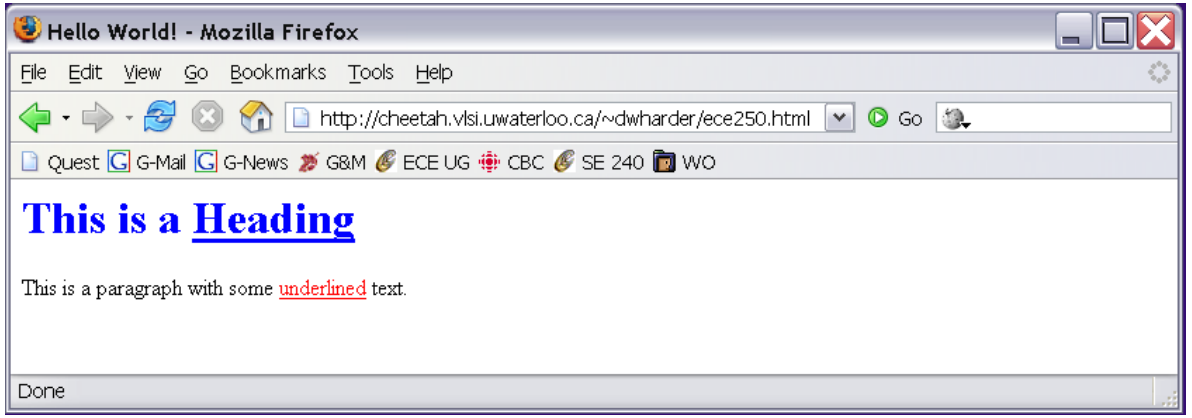
- That is, you only want text marked as `<u>text</u>` to be underlined if it is a descendant of a `<p>` tag

# Example: XHTML and CSS

For example, this style renders as follows:

```
<style type="text/css">
   h1 { color:blue; }
   p u { color:red; }
</style>
```



# Example: XHTML and CSS

You can read the second style

```
<style type="text/css">
   h1 { color:blue; }
   p u { color:red; }
</style>
```

as saying “text/decorations descendant from the underlining tag (<u>) which itself is a descendant of a paragraph tag should be coloured red”

# Example: XML

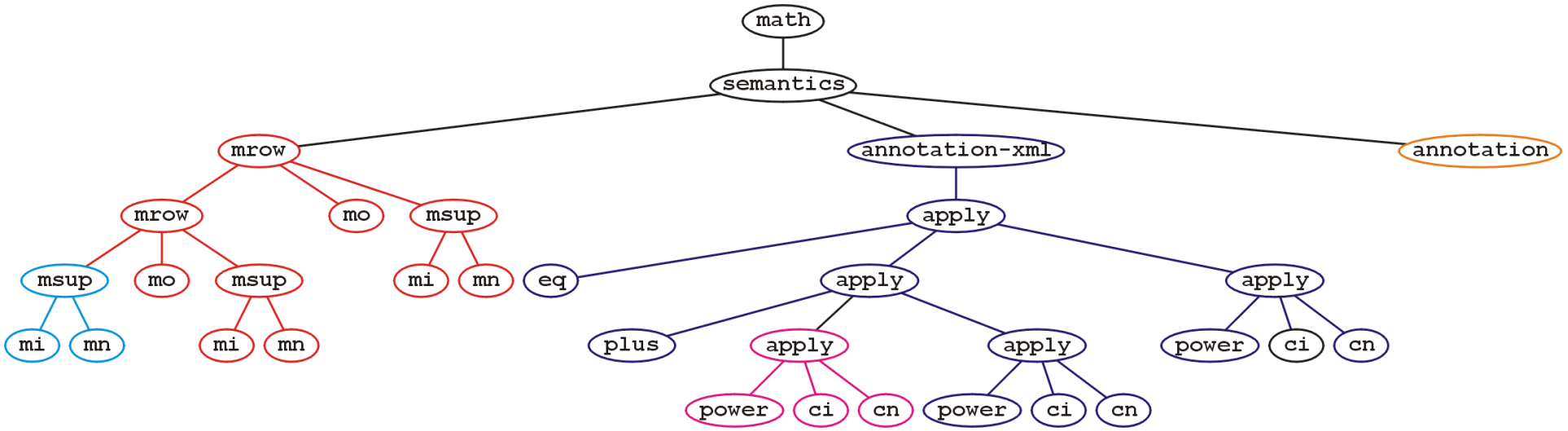
- In general, any XML can be represented as a tree
  - All XML tools make use of this feature
  - Parsers convert XML into an internal tree structure
  - XML transformation languages manipulate the tree structure
    - *E.g.*, XMLT

# MathML: $x^2 + y^2 = z^2$

```
<math xmlns="http://www.w3.org/1998/Math/MathML">
  <semantics>
    <mrow><mrow><msup><mi>x</mi><mn>2</mn></msup><mo>+</mo>
    <msup><mi>y</mi><mn>2</mn></msup></mrow>
    <mo>=</mo><msup><mi>z</mi><mn>2</mn></msup></mrow>
    <annotation-xml encoding="MathML-Content">
      <apply><eq/>
        <apply><plus/>
          <apply><power/><ci>x</ci><cn>2</cn></apply>
          <apply><power/><ci>y</ci><cn>2</cn></apply>
        </apply>
        <apply><power/><ci>z</ci><cn>2</cn></apply>
      </apply>
    </annotation-xml>
    <annotation encoding="Maple">x^2+y^2 = z^2</annotation>
  </semantics>
</math>
```

# MathML: $x^2 + y^2 = z^2$

- The tree structure for the same MathML expression is



# MathML: $x^2 + y^2 = z^2$

- Why use 500 characters to describe the equation

$$x^2 + y^2 = z^2$$

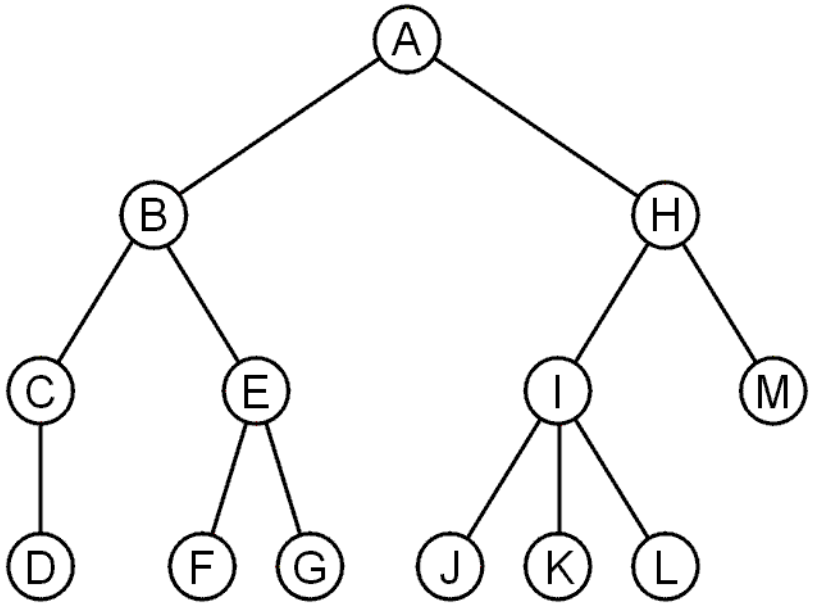
- which, after all, is only twelve characters (counting spaces)?

- The root contains three children, each different codings of:
  - How it should look (presentation),
  - What it means mathematically (content), and
  - A translation to a specific language (Maple)

# Abstract Trees

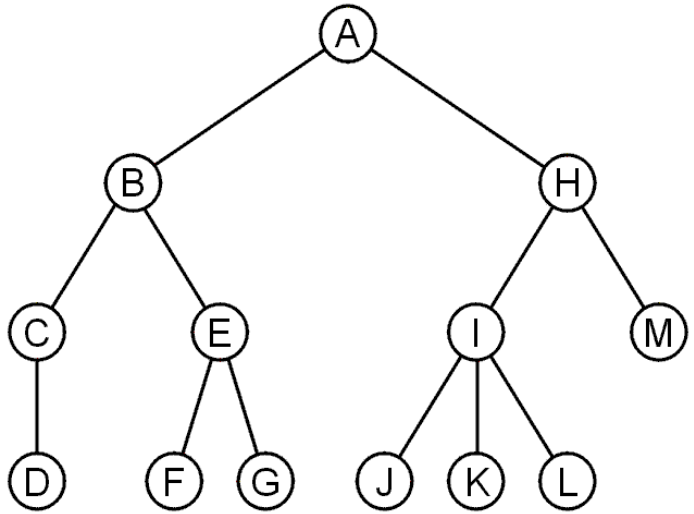
- An abstract tree (or abstract hierarchy) does not restrict the number of nodes
  - In this tree, the degrees vary:

| Degree | Nodes |
|---|---|
| 0 | D, F, G, J, K, L, M |
| 1 | C |
| 2 | A, B, E, H |
| 3 | I |



# Abstract Trees: Design

- We implement an abstract tree or hierarchy by using a class that:
  - Stores a value
  - Stores the children in a linked-list



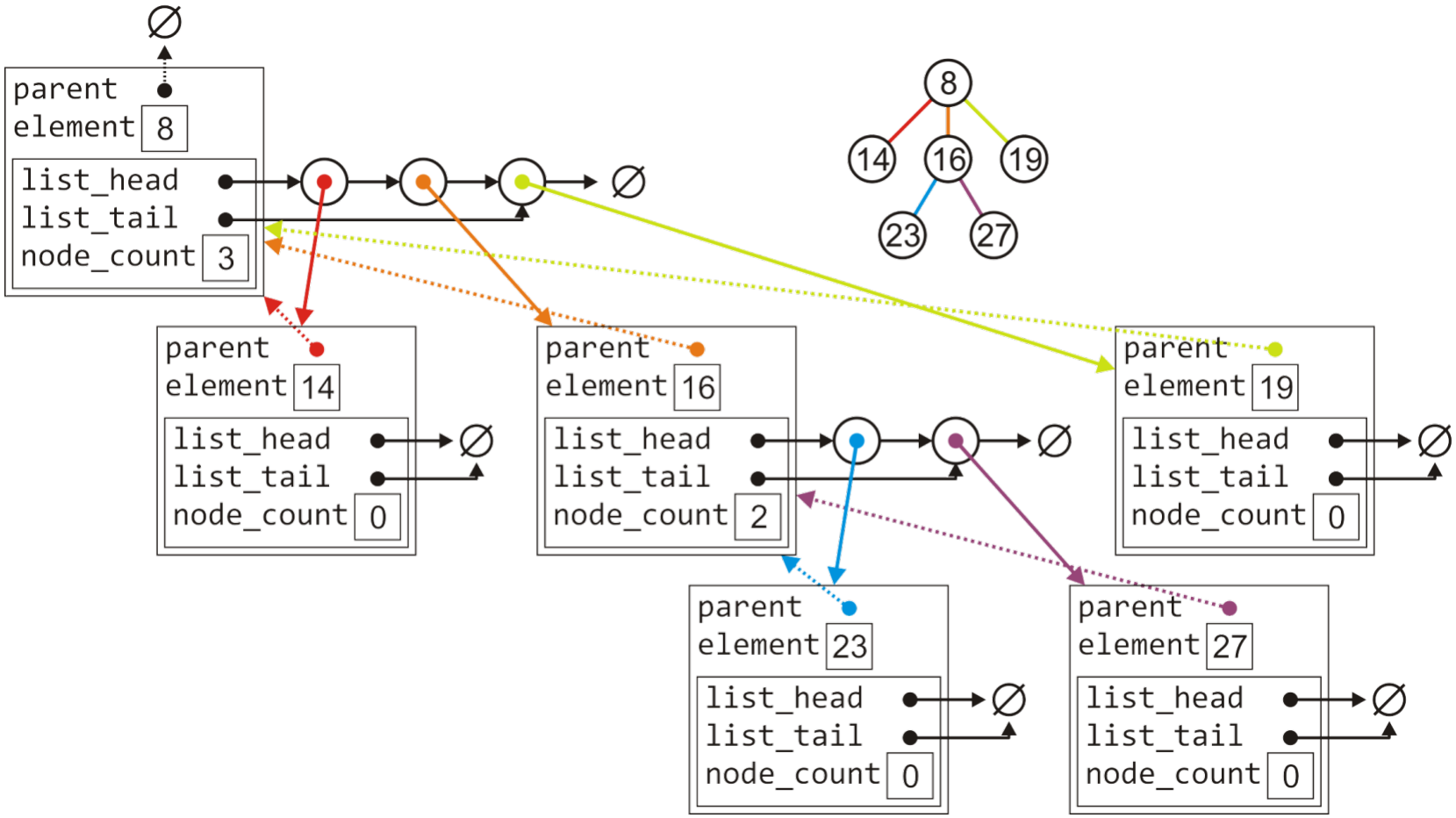
# Implementation

- The class definition would be:

```
template <typename Type>
class Simple_tree {
    private:
        Type node_value;
        Simple_tree *parent_node;
        std::list<Simple_tree *> children;
    public:
        Simple_tree( Type const & = Type(), Simple_tree * = nullptr );
        Type value() const;
        Simple_tree *parent() const;
        int degree() const;
        bool is_root() const;
        bool is_leaf() const;
        Simple_tree *child( int n ) const;
        int height() const;
        void insert( Type const & );
        void attach( Simple_tree * );
        void detach();
};
```

# Implementation

- The tree with six nodes would be stored as follows:



# Implementation

- Much of the functionality is similar to that of the list class:

```
template <typename Type>
Simple_tree<Type>::Simple_tree( Type const &obj, Simple_tree *p ):
node_value( obj ),
parent_node( p ) {
    // Empty constructor
}

template <typename Type>
Type Simple_tree<Type>::value() const {
    return node_value;
}

template <typename Type>
Simple_tree<Type> *Simple_tree<Type>::parent() const {
    return parent_node;
}
```

# Implementation

- Much of the functionality is similar to that of the list class:

```
template <typename Type>
bool Simple_tree<Type>::is_root() const {
    return ( parent() == nullptr );
}

template <typename Type>
int Simple_tree<Type>::degree() const {
    return children.size();
}

template <typename Type>
bool Simple_tree<Type>::is_leaf() const {
    return ( degree() == 0 );
}
```

# Implementation

- Accessing the $n^{\text{th}}$ child requires a for loop ($\Theta(n)$):

```
template <typename Type>
Simple_tree<Type> *Simple_tree<Type>::child( int n ) const {
    if ( n < 0 || n >= degree() ) {
        return nullptr;
    }
    auto *child = children.head();
    // Skip the first n - 1 children
    for ( int i = 1; i < n; ++i ) {
        child = child->next();
    }
    return child->value();
}
```

# Implementation

- Attaching a new object to become a child is similar to a linked list:

```
template <typename Type>
void Simple_tree<Type>::attach( Type const &obj ) {
    children.push_back( new Simple_tree( obj, this ) );
}
```

# Implementation

- To detach a tree from its parent:
  - If it is already a root, do nothing
  - Otherwise, erase this object from the parent's list of children and set the parent pointer to zero

```
template <typename Type>
void Simple_tree<Type>::detach() {
    if ( is_root() ) {
        return;
    }

    parent()->children.erase( this );
    parent_node = nullptr;
}
```

# Implementation

- Attaching an entirely new tree as a sub-tree, however, first requires us to check if the tree is not already a sub-tree of another node:
  - If so, we must detach it first and only then can we add it

```
template <typename Type>
void Simple_tree<Type>::attach( Simple_tree<Type> *tree ) {
    if ( !tree->is_root() ) {
        tree->detach();
    }

    tree->parent_node = this;
    children.push_back( tree );
}
```

# Implementation

- Suppose we want to find the size of a tree:
  - An empty tree has size 0, a tree with no children has size 1
  - Otherwise, the size is one plus the size of all the children

```
template <typename Type>
int Simple_tree<Type>::size() const {
    if ( this == nullptr ) {
        return 0;
    }
    int tree_size = 1;

    for ( auto *child = children.begin(); child != children.end(); child = child->next()
  ) {
        tree_size += child->value()->size();
    }
    return tree_size ;
}
```

# Implementation

- Suppose we want to find the height of a tree:
  - An empty tree has height –1 and a tree with no children is height 0
  - Otherwise, the height is one plus the maximum height of any sub tree

```
#include <algorithm>
template <typename Type>
int Simple_tree<Type>::height() const {
    if ( this == nullptr ) {
        return -1;
    }
    int tree_height = 0;
    for ( auto *child = children.begin(); child != children.end(); child = child->next()
  ) {
        tree_height = std::max( h, 1 + child->value()->height() );
    }
    return tree_height;
}
```

# Implementation

- Implementing a tree by storing the children in an array is similar, however, we must deal with the full structure
  - A general tree using an array would have a constructor similar to:

```
template <typename Type>
Simple_tree<Type>::Simple_tree( Type const &obj, Simple_tree *p ):
node_value( obj ),
parent_node( p ),
child_count( 0 ),
child_capacity( 4 ),
children( new Simple_tree *[child_capacity] ) {
    // Empty constructor
}
```

# Summary

- In this topic, we have:
  - Introduced the terminology used for the tree data structure
  - Discussed various terms which may be used to describe the properties of a tree, including:
    - root node, leaf node
    - parent node, children, and siblings
    - ordered trees
    - paths, depth, and height
    - ancestors, descendants, and subtrees
  - We looked at XHTML and CSS
- We have looked at one implementation of a general tree:
  - store the value of each node
  - store all the children in a linked list
  - not an easy ($\Theta(1)$) way to access children
  - if we use an array, different problems...