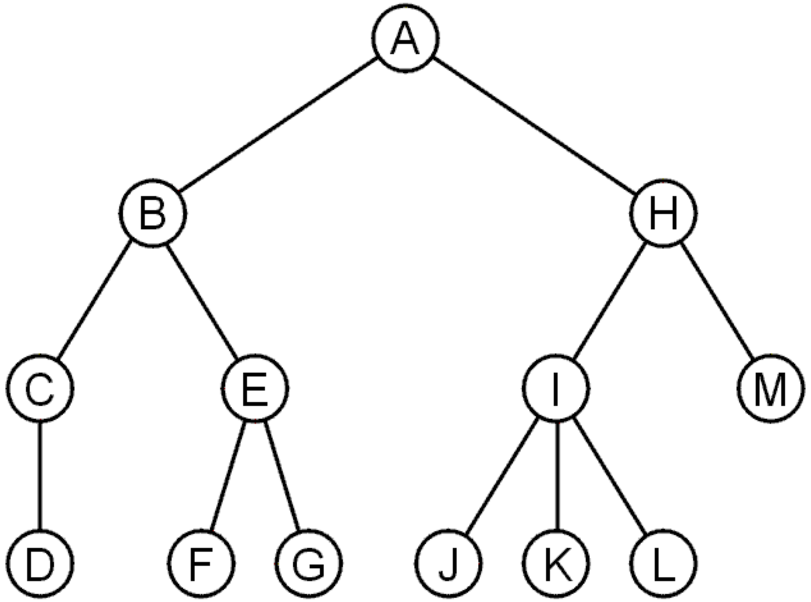
Binary Tree

Outline

- Tree
- Example: XHTML and CSS
- Binary Tree
- Traversal
- Application: Expression Tree

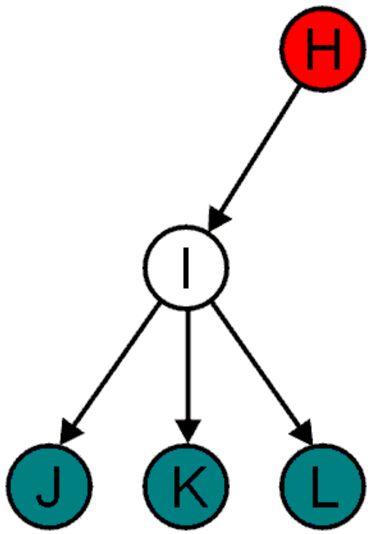
Trees

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



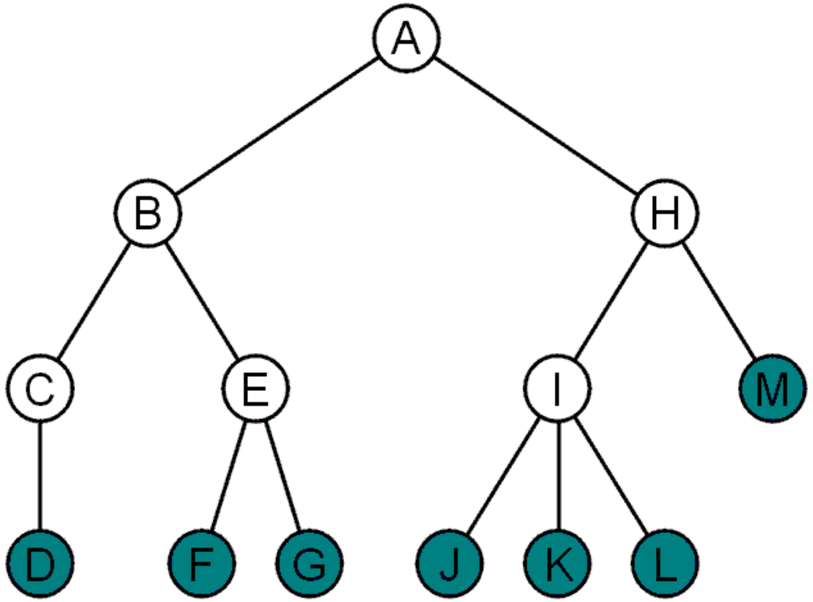
Trees

- The degree of a node is defined as the number of its children:
  - deg(I) = 3
- Nodes with the same parent are siblings
  - J, K, and L are siblings



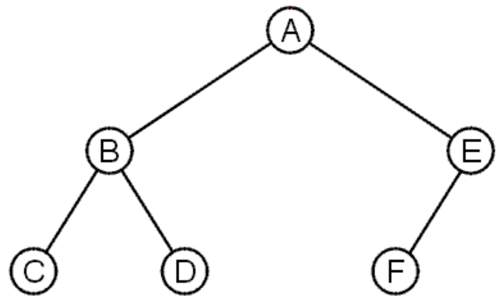
Trees

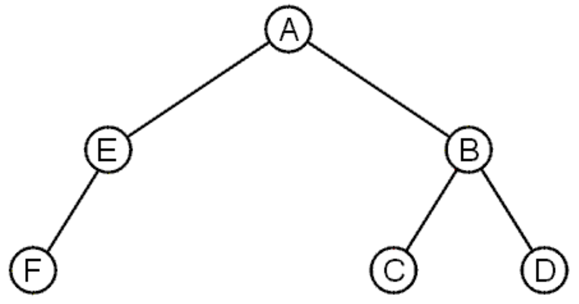
- Nodes with degree zero are also called leaf nodes
- All other nodes are said to be internal nodes, that is, they are internal to the tree



Trees

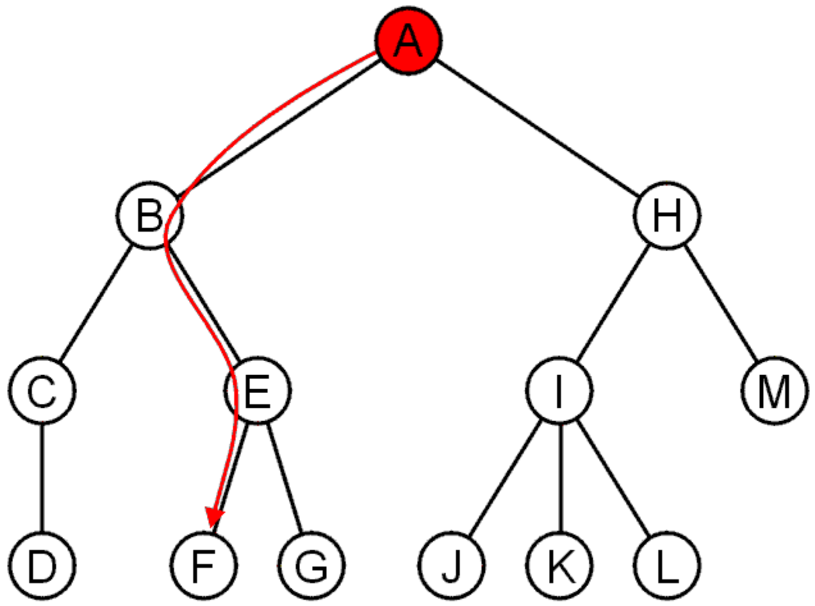
- These trees are equal if the order of the children is ignored
  - unordered trees
- They are different if order is relevant (ordered trees)
  - We will usually examine ordered trees (linear orders)
  - In a hierarchical ordering, order is not relevant





Trees

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



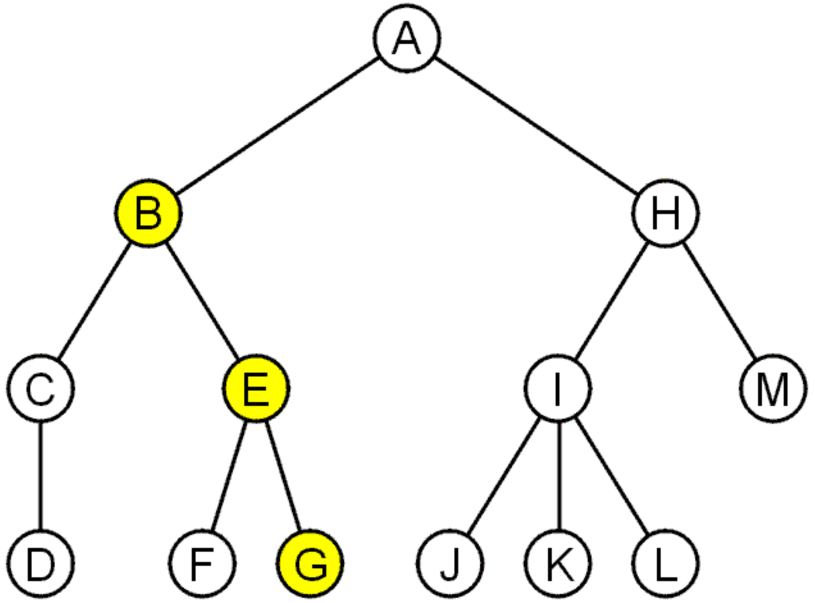
Trees

- A path is a sequence of nodes

  (a0, a1, ..., an)

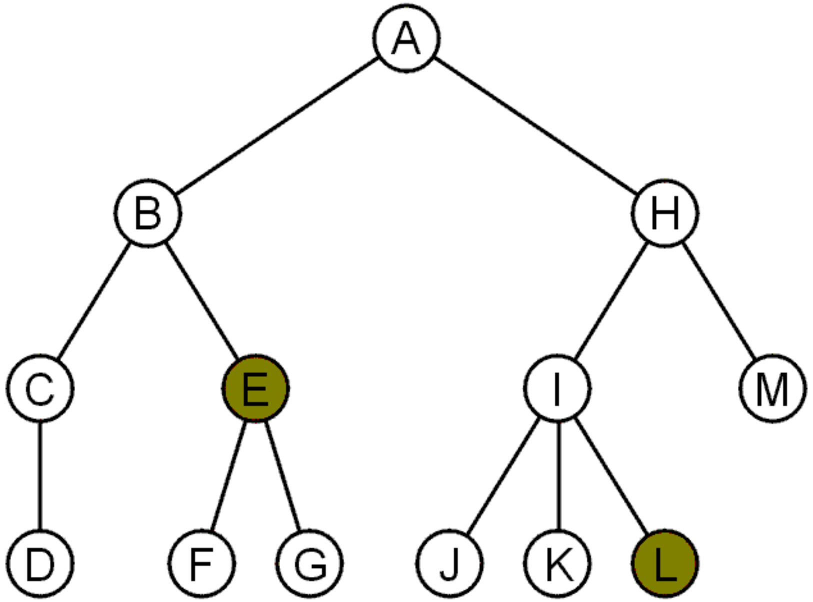
  where ak + 1 is a child of ak is
- The length of this path is n
- E.g., the path (B, E, G)

  has length 2



Trees

- 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

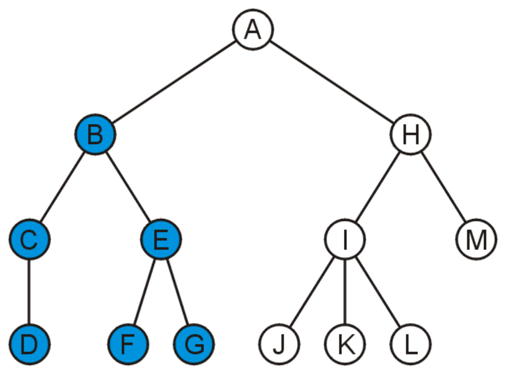


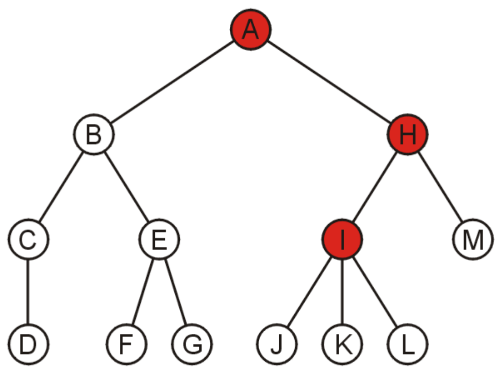
Trees

- 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

Trees

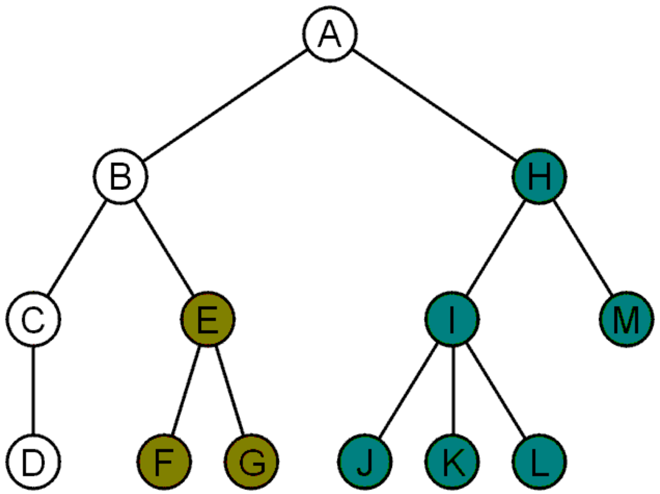
- The descendants of node B are B, C, D, E, F, and G:
- The ancestors of node I are I, H, and A:



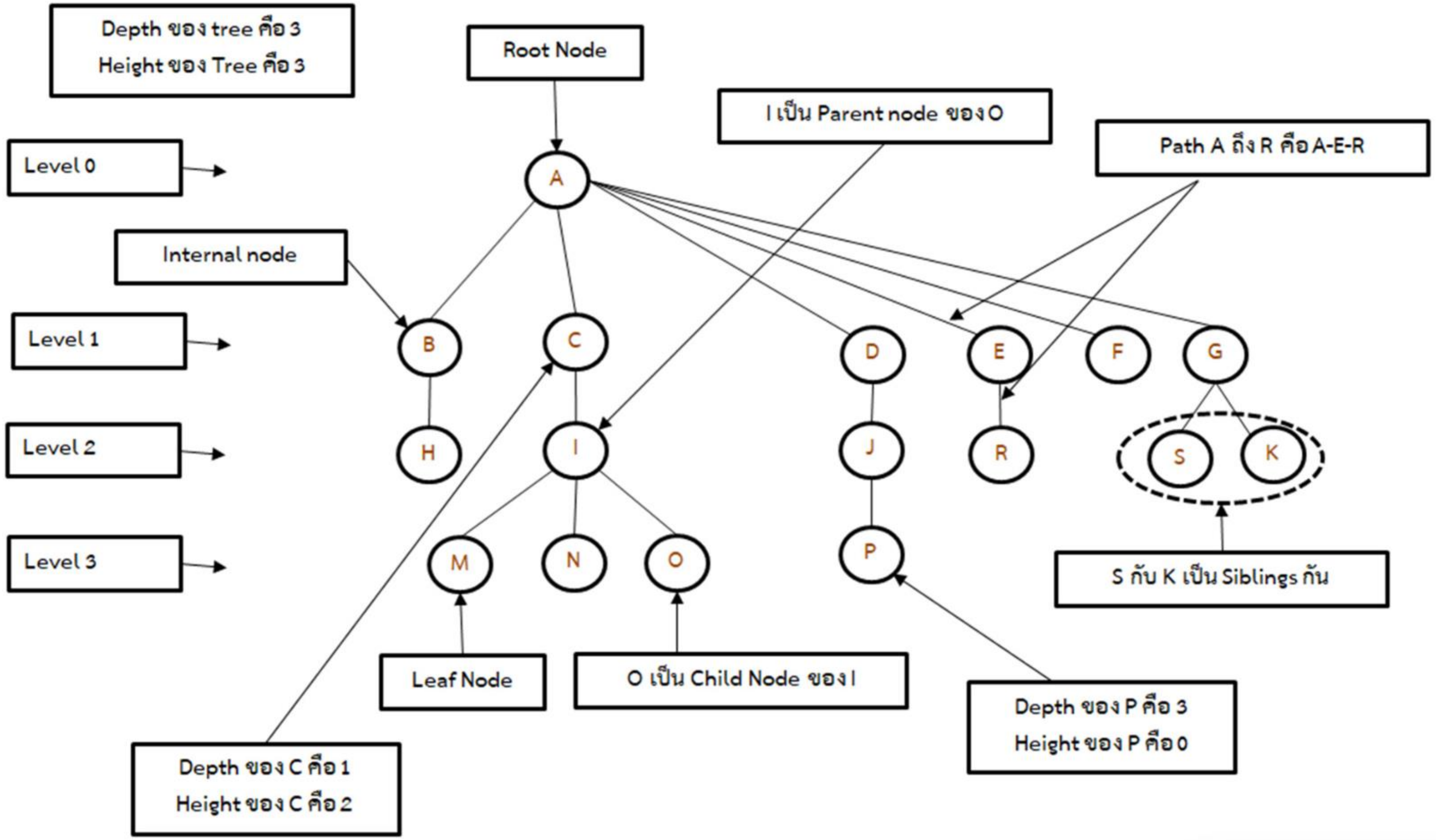


Trees

- 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



Trees



Depth ของ tree คือ 3
Height ของ Tree คือ 3
Root Node
I เป็น Parent node ของ O
Path A ถึง R คือ A-E-R
Level 0
A
Internal node
Level 1
B
C
D
E
F
G
Level 2
H
I
J
R
S
K
Level 3
M
N
O
P
S กับ K เป็น Siblings กัน
Leaf Node
O เป็น Child Node ของ I
Depth ของ P คือ 3
Height ของ P คือ 0
Depth ของ C คือ 1
Height ของ C คือ 2

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

title

heading

body of page

underlining

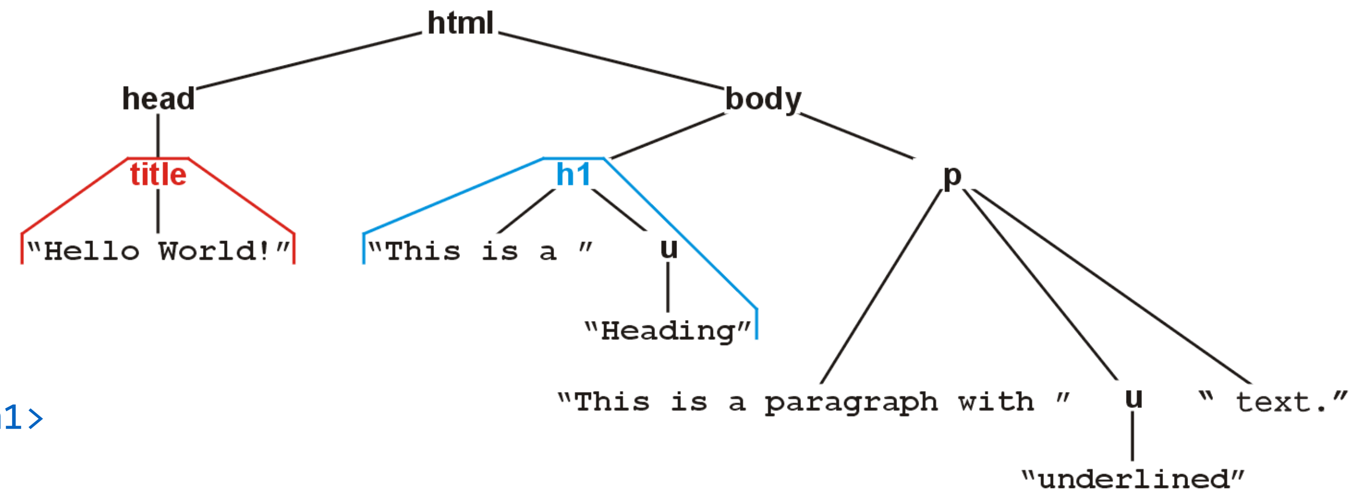
paragraph

```
<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

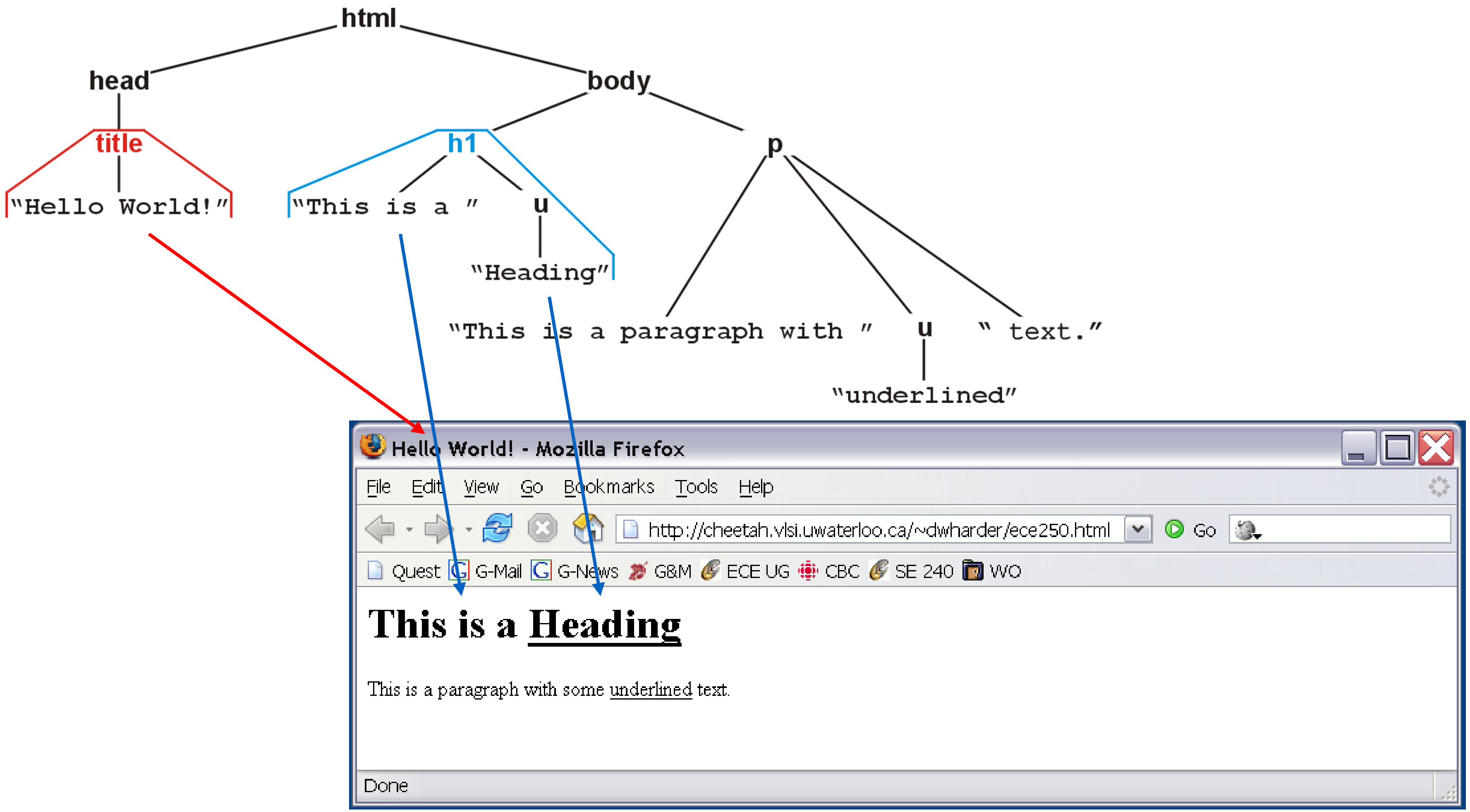
The nested tags define a tree rooted at the HTML tag

```
<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>
```



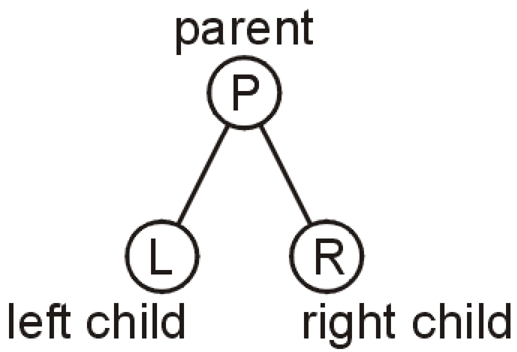
Example: XHTML and CSS

Web browsers render this tree as a web page



Binary Tree

- A binary tree is a restriction where each node has exactly two children:
  - Each child is either empty or another binary tree
  - This restriction allows us to label the children as left and right subtrees

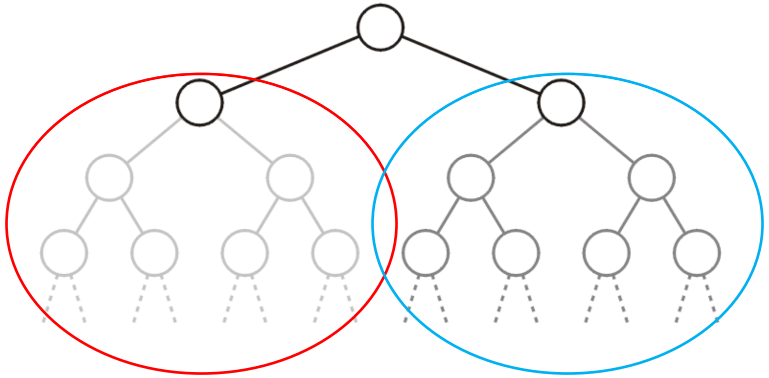


- At this point, recall that lg(n) = Q(logb(n)) for any b

Binary Tree

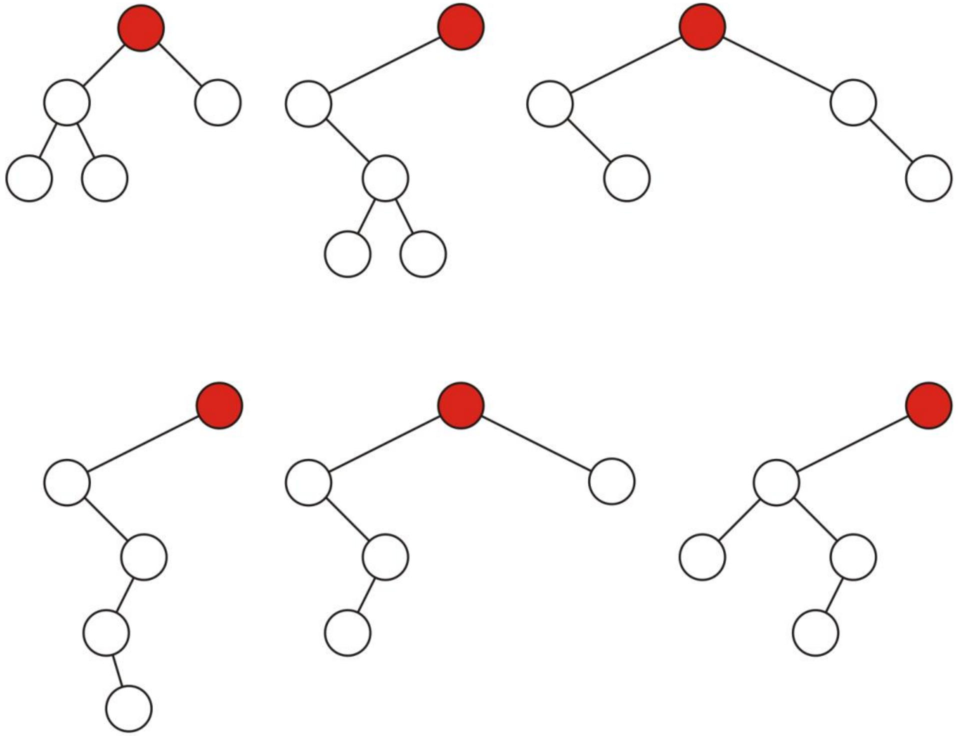
We will also refer to the two sub-trees as

- The left-hand sub-tree, and
- The right-hand sub-tree



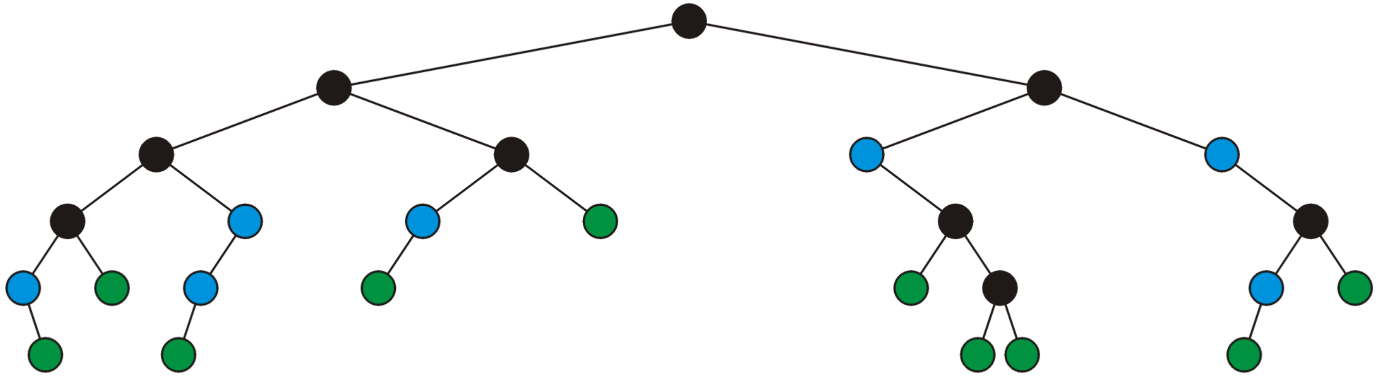
Binary Tree

Sample variations on binary trees with five nodes:



Binary Tree

A *full* node is a node where both the left and right sub-trees are non-empty trees

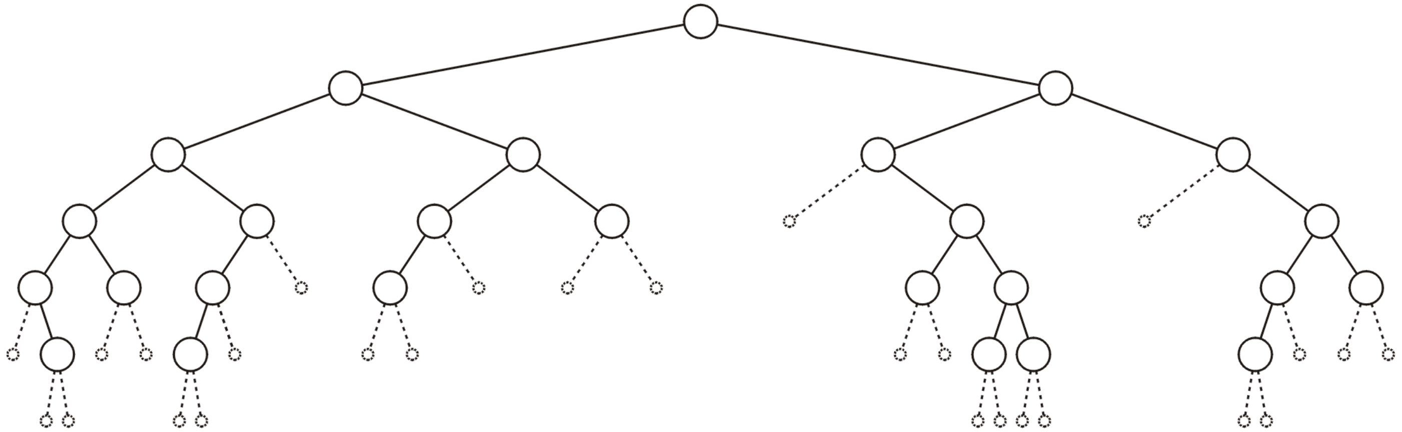


Legend:

full nodes neither leaf nodes

Binary Tree

An *empty node* or a *null sub-tree* is any location where a new leaf node could be appended



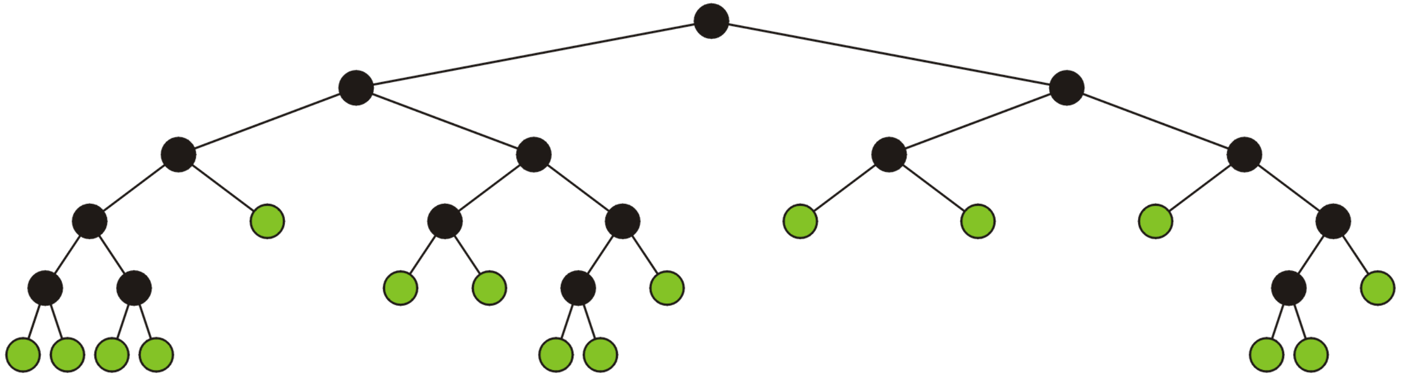
Binary Tree

A *full binary tree* is where each node is:

- A full node, or
- A leaf node

These have applications in

- Expression trees
- Huffman encoding



Binary Node class

The binary node class is similar to the single node class:

```
#include <algorithm>

template <typename Type>
class Binary_node {
    protected:
        Type node_value;
        Binary_node *p_left_tree;
        Binary_node *p_right_tree;

    public:
        Binary_node( Type const & );

        Type value() const;
        Binary_node *left() const;
        Binary_node *right() const;

        bool is_leaf() const;
        int size() const;
        int height() const;
        void clear();
}
```

Binary Node class

We will usually only construct new leaf nodes

```
template <typename Type>
Binary_node<Type>::Binary_node( Type const &obj ):
node_value( obj ),
p_left_tree( nullptr ),
p_right_tree( nullptr ) {
    // Empty constructor
}
```

Binary Node class

The accessors are similar to that of `Single_list`

```
template <typename Type>
Type Binary_node<Type>::value() const {
    return node_value;
}
template <typename Type>
Binary_node<Type> *Binary_node<Type>::left() const {
    return p_left_tree;
}
template <typename Type>
Binary_node<Type> *Binary_node<Type>::right() const {
    return p_right_tree;
}
```

Binary Node class

Much of the basic functionality is very similar to `Simple_tree`

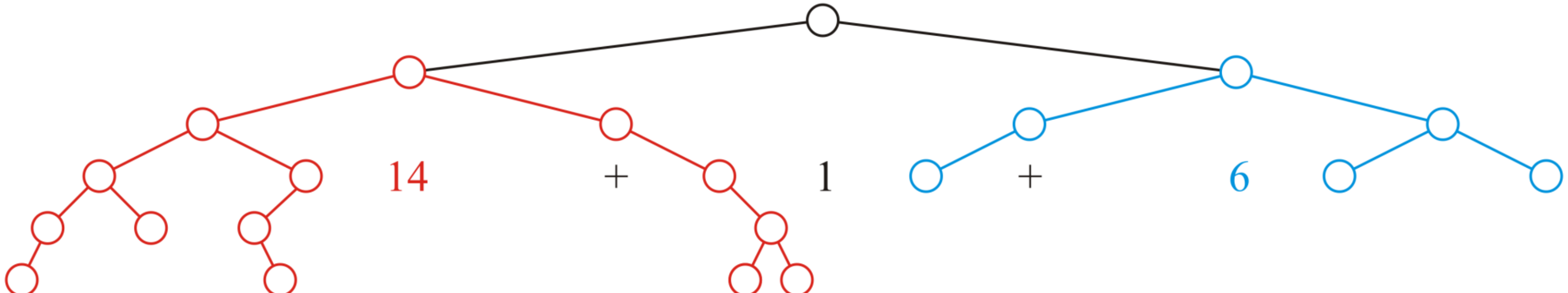
```
template <typename Type>
bool Binary_node<Type>::is_leaf() const {
    return (left() == nullptr) && (right() == nullptr);
}
```

Size

The recursive size function runs in $\Theta(n)$ time and $\Theta(h)$ memory

- These can be implemented to run in $\Theta(1)$

```
template <typename Type>
int Binary_node<Type>::size() const {
    if ( left() == nullptr ) {
        return ( right() == nullptr ) ? 1 : 1 + right()->size();
    } else {
        return ( right() == nullptr ) ?
            1 + left()->size() :
            1 + left()->size() + right()->size();
}
```

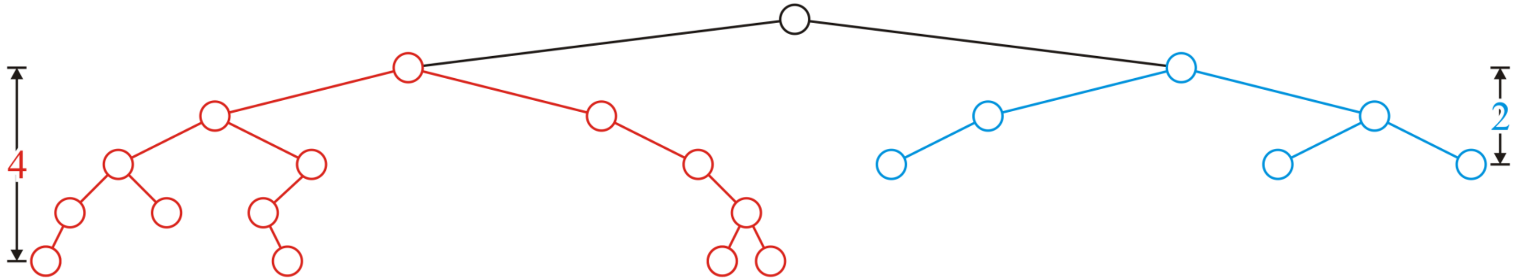


Height

The recursive height function also runs in $\Theta(n)$ time and $\Theta(h)$ memory

- Later we will implement this in $\Theta(1)$ time

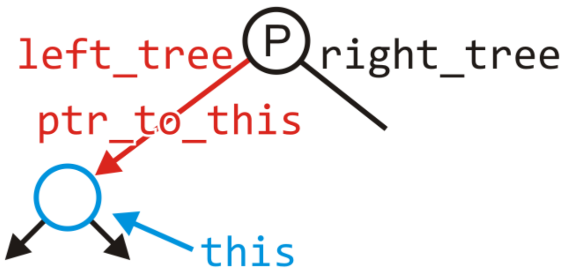
```
int Binary_node<Type>::height() const {
    if ( left() == nullptr ) {
        return ( right() == nullptr ) ? 0 : 1 + right()->height();
    } else {
        return ( right() == nullptr ) ?
            1 + left()->height() :
            1 + left()->height() + right()->height();
}
```



Clear

Removing all the nodes in a tree is similarly recursive:

```
template <typename Type>
void Binary_node<Type>::clear( Binary_node *&p_to_this ) {
    if ( left() != nullptr ) {
        left()->clear( p_left_tree );
    }

    if ( right() != nullptr ) {
        right()->clear( p_right_tree );
    }

    delete this;
    p_to_this = nullptr;
}
```



Traversal

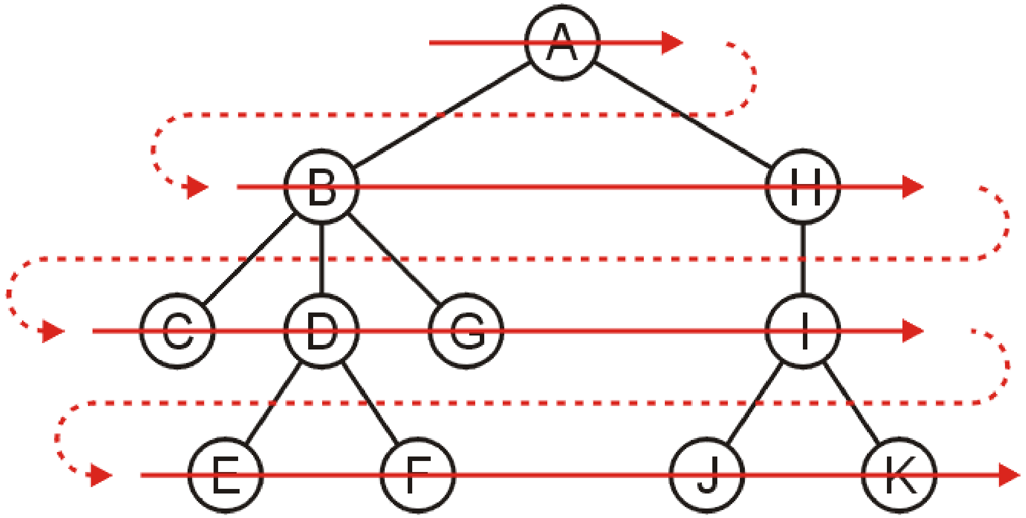
There are 2 types of traversal:

- The breadth-first traversal visits all nodes at depth *k* before proceeding onto depth *k* + 1
  - Easy to implement using a queue
- Another approach is to visit always go as deep as possible before visiting other siblings: *depth-first traversals*

Breadth-First Traversal

Breadth-first traversals visit all nodes at a given depth

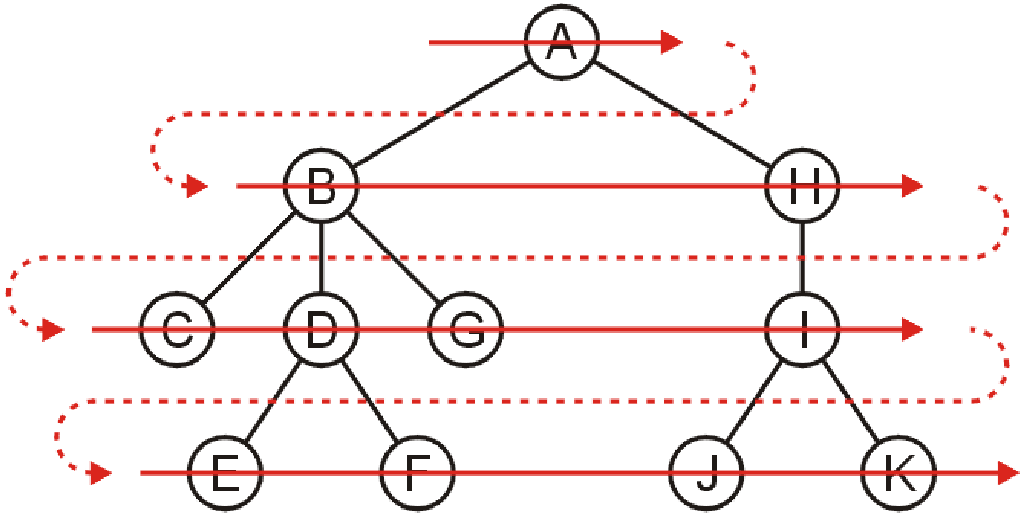
- Can be implemented using a queue
- Run time is $\Theta(n)$
- Memory is potentially expensive: maximum nodes at a given depth
- Order: A B H C D G I E F J K



Breadth-First Traversal

The implementation was already discussed:

- Create a queue and push the root node onto the queue
- While the queue is not empty:
  - Push all of its children of the front node onto the queue
  - Pop the front node

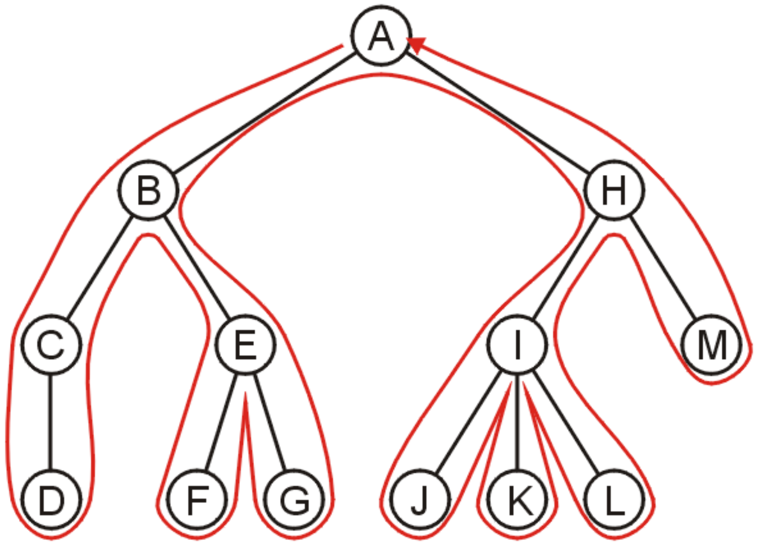


Backtracking

To discuss depth-first traversals, we will define a backtracking algorithm for stepping through a tree:

- At any node, we proceed to the first child that has not yet been visited
- Or, if we have visited all the children (of which a leaf node is a special case), we backtrack to the parent and repeat this decision making process

We end once all the children of the root are visited

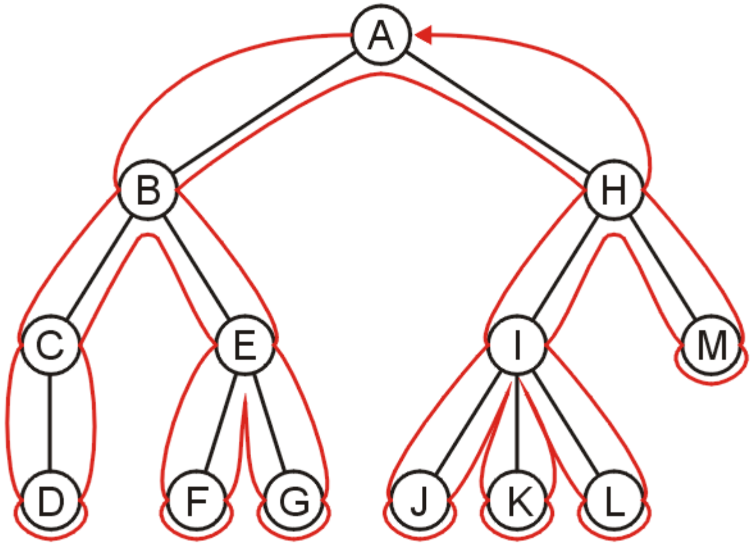


depth-first traversal

We define such a path as a *depth-first traversal*

We note that each node could be visited twice in such a scheme

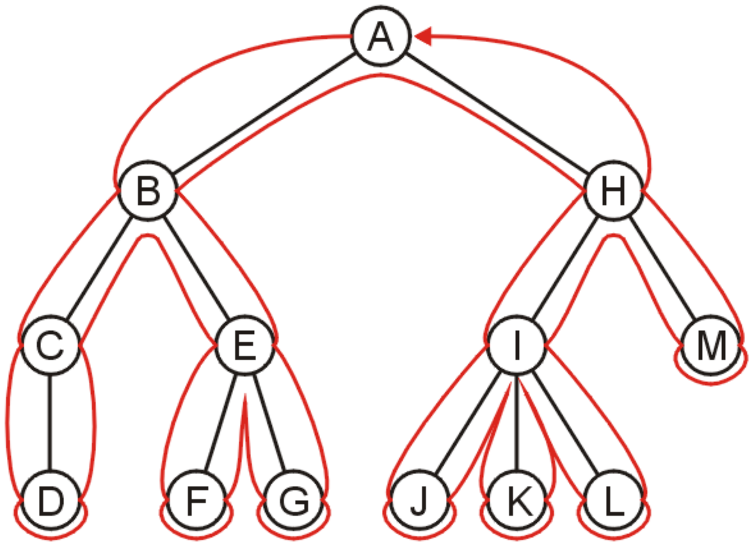
- The first time the node is approached (before any children)
- The last time it is approached (after all children)



Implementing depth-first traversal

Performed on this tree, the output would be

```
<A><B><C><D></D></C><E><F></F><G></G></E></B><H><I><J></J><K></K><L></L></I><M></M></H></A>
```



Implementing depth-first traversal

Alternatively, we can use a stack:

- Create a stack and push the root node onto the stack
- While the stack is not empty:
  - Pop the top node
  - Push all of the children of that node to the top of the stack in reverse order
- Run time is $\Theta(n)$
- The objects on the stack are all unvisited siblings from the root to the current node
  - If each node has a maximum of two children, the memory required is $\Theta(h)$: the height of the tree

With the recursive implementation, the memory is $\Theta(h)$: recursion just hides the memory

Guidelines

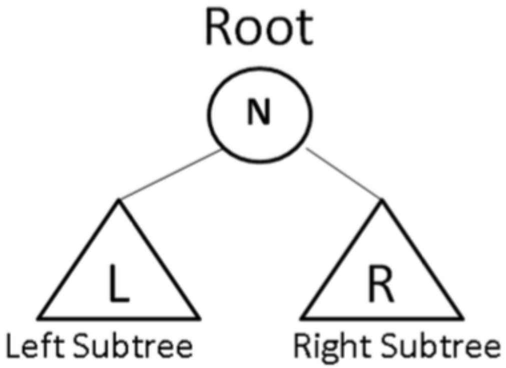
Depth-first traversals are used whenever:

- The **parent** needs information about all its children or **descendants**, or
- The **children** require information about all its parent or **ancestors**

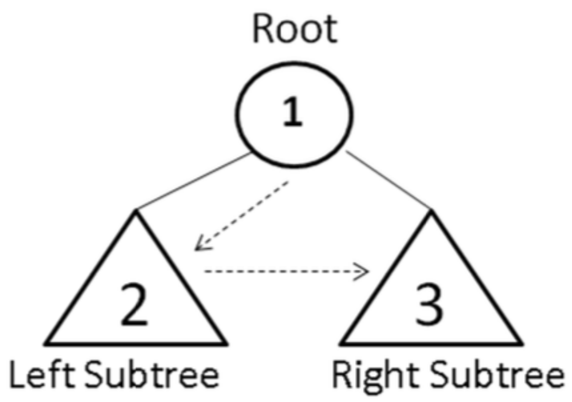
In designing a depth-first traversal, it is necessary to consider:

1. Before the children are traversed, what initializations, operations and calculations must be performed?
2. In recursively traversing the children:
   a) What information must be passed to the children during the recursive call?
   b) What information must the children pass back, and how must this information be collated?
3. Once all children have been traversed, what operations and calculations depend on information collated during the recursive traversals?
4. What information must be passed back to the parent?

Preorder-Inorder-Postorder Traversal

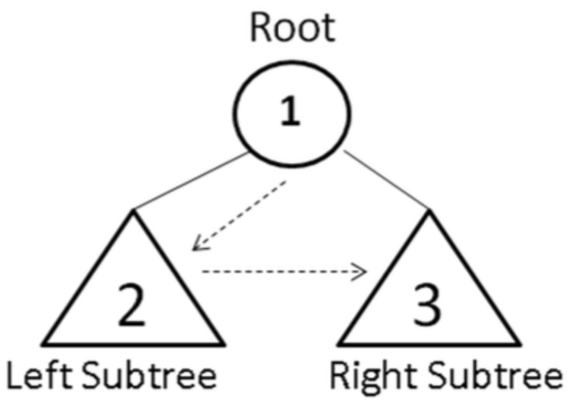


Preorder Traversal

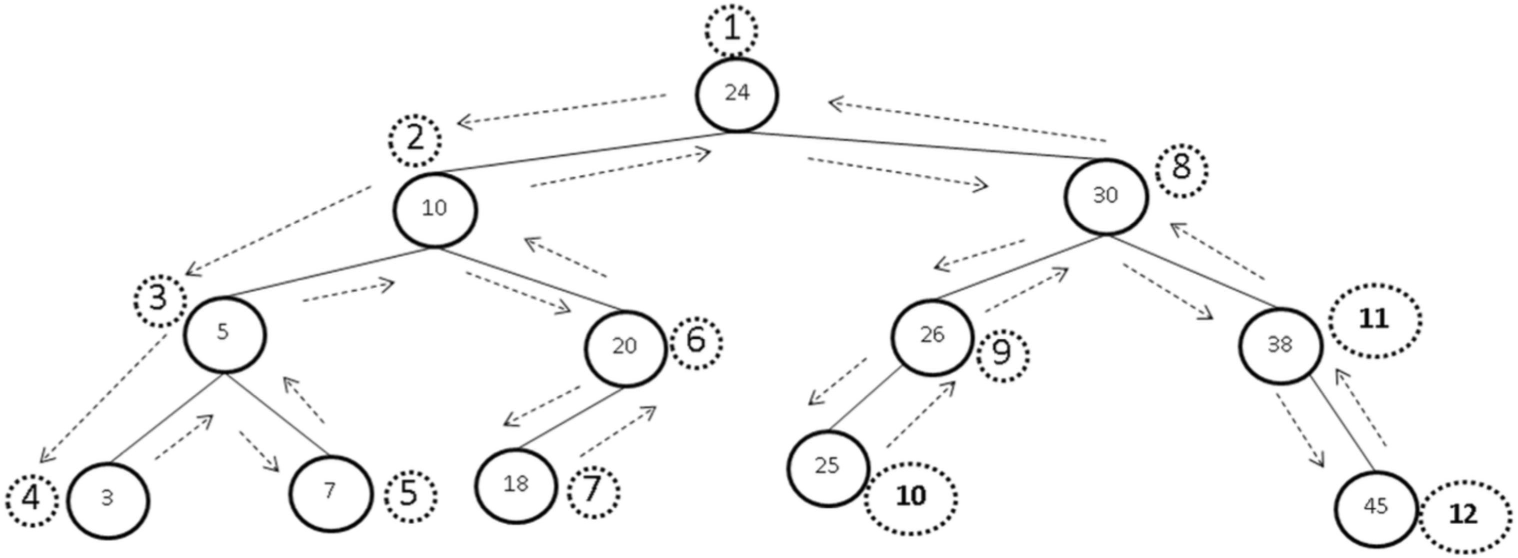


Root – Left - Right

Preorder Traversal

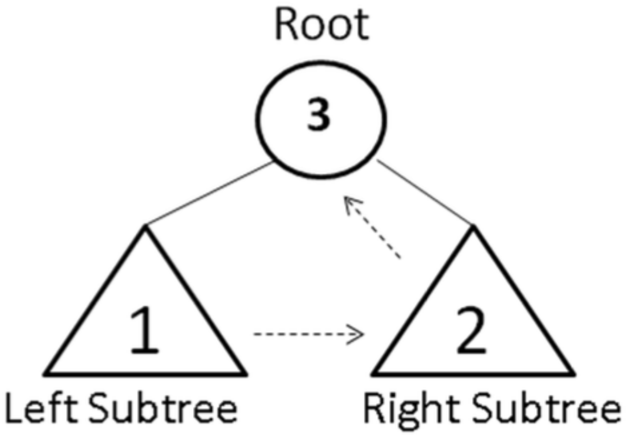


Root – Left - Right

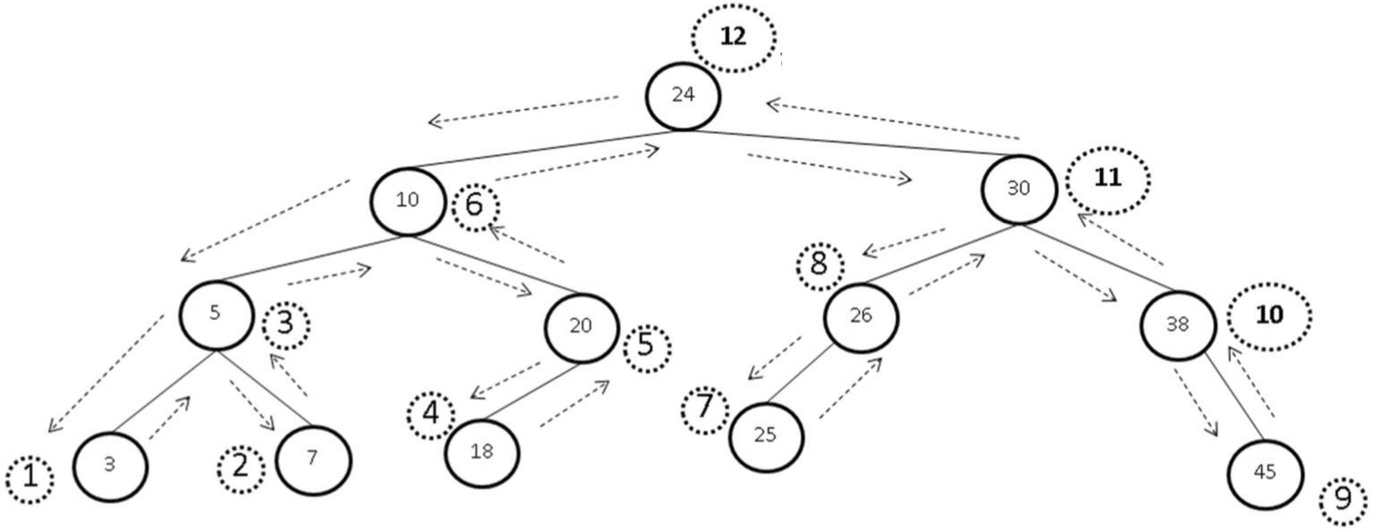


Example

Postorder Traversal

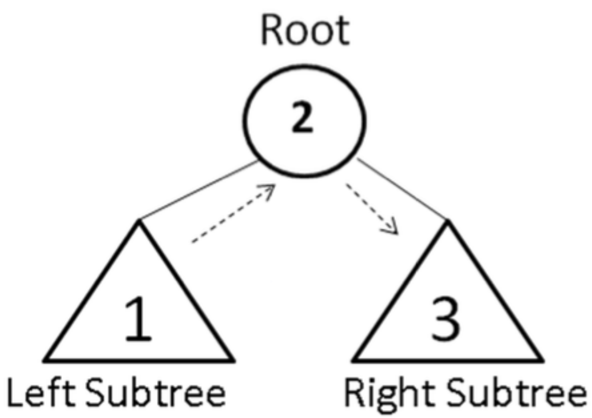


Left - Right - Root

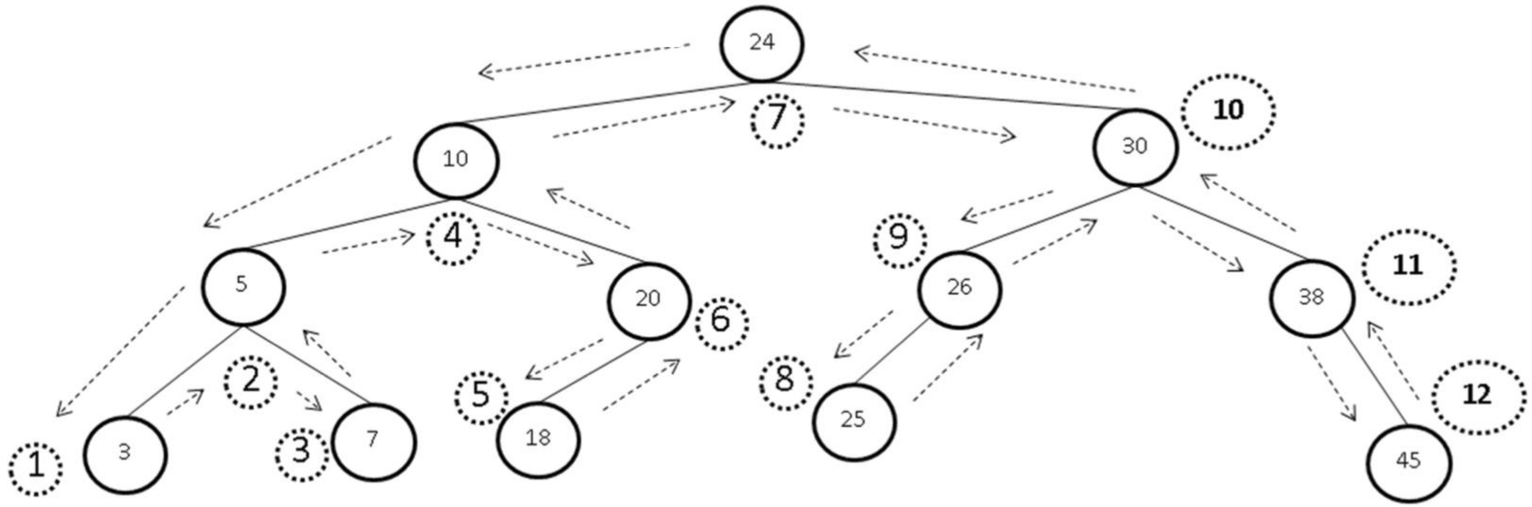


Example

Inorder Traversal



Left – Root - Right



Example

Run times

Recall that with linked lists and arrays, some operations would run in $\Theta(n)$ time

The run times of operations on binary trees, we will see, depends on the height of the tree

We will see that:

- The worst is clearly $\Theta(n)$
- Under average conditions, the height is $\Theta\left(\sqrt{n}\right)$
- The best case is $\Theta(\ln(n))$

Run times

If we can achieve and maintain a height $\Theta(\lg(n))$, we will see that many operations can run in $\Theta(\lg(n))$ we

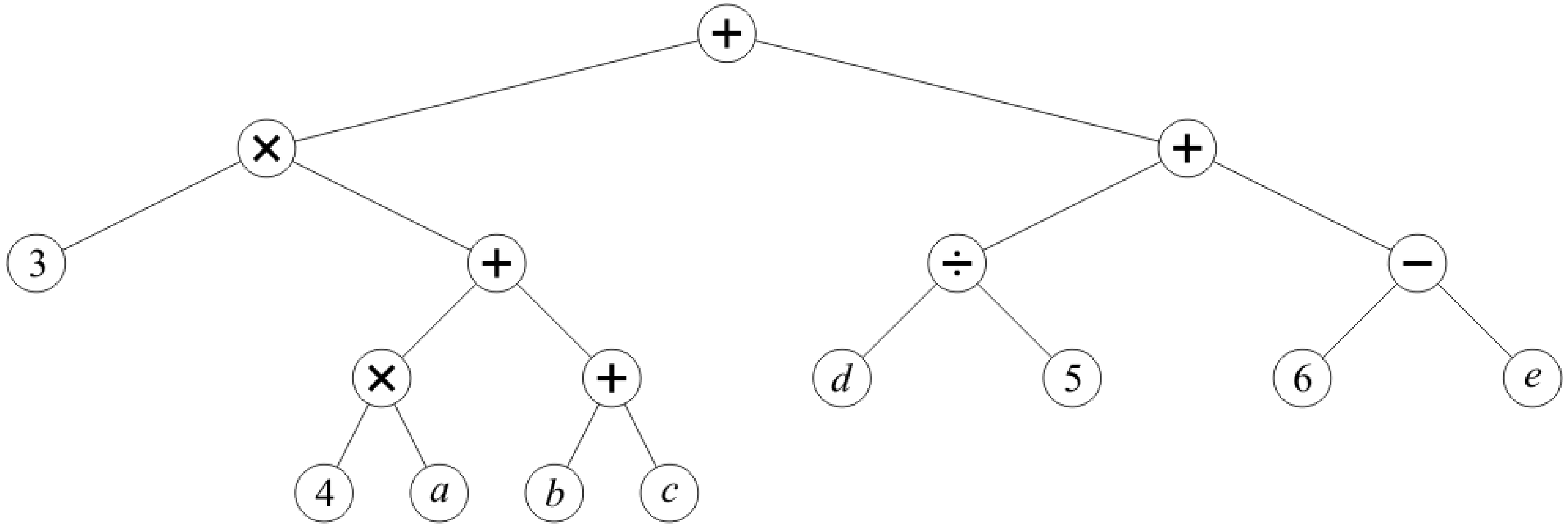
Logarithmic time is not significantly worse than constant time:

$\lg(1000) \approx 10$	kB
$\lg(1\,000\,000) \approx 20$	MB
$\lg(1\,000\,000\,000) \approx 30$	GB
$\lg(1\,000\,000\,000\,000) \approx 40$	TB
$\lg(1000^n) \approx 10\,n$	

Application: Expression Tree

Any basic mathematical expression containing binary operators may be represented using a binary tree

For example, $3(4a + b + c) + d/5 + (6 - e)$



Application: Expression Tree

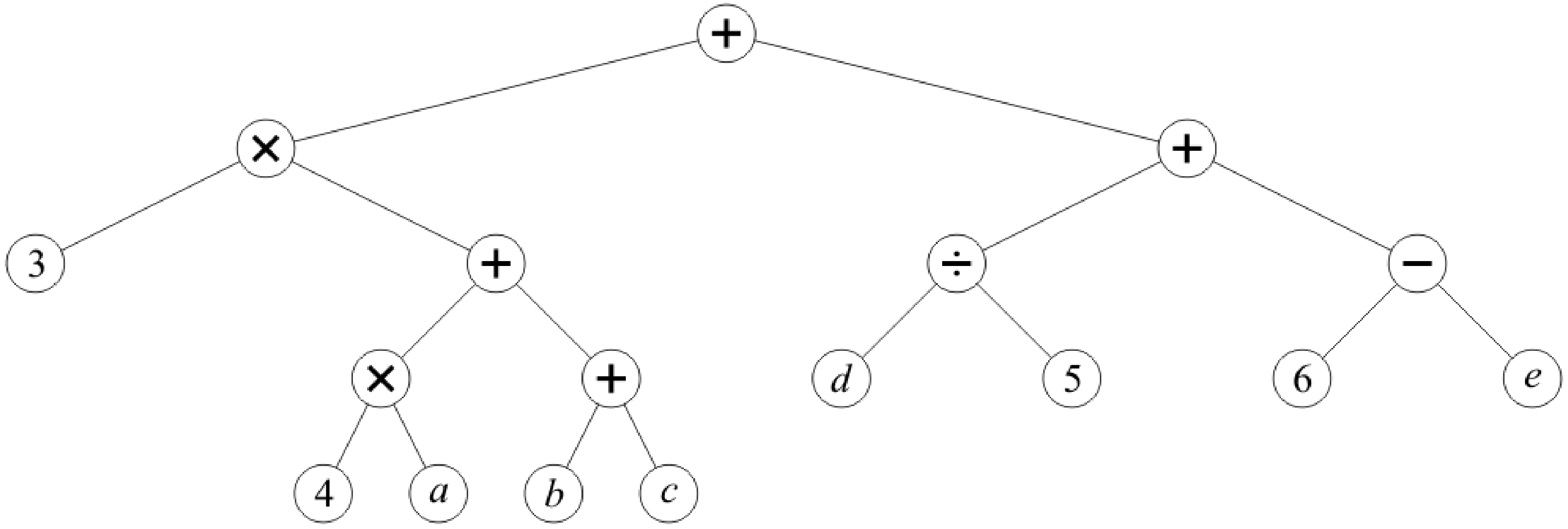
Observations:

- Internal nodes store operators
- Leaf nodes store literals or variables
- No nodes have just one sub tree
- The order is not relevant for
  - Addition and multiplication (commutative)
- Order is relevant for
  - Subtraction and division (non-commutative)
- It is possible to replace non-commutative operators using the unary negation and inversion:

$$(a/b) = a\, b^{-1} \qquad (a - b) = a + (-b)$$

Application: Expression Tree

A post-order depth-first traversal converts such a tree to the reverse-Polish format



$$3\ 4\ a\ \times\ b\ c\ +\ +\ \times\ d\ 5\ \div\ 6\ e\ -\ +\ +$$

Reference

Allen, W. M. (2007). *Data structures and algorithm analysis in C++*. Pearson Education India.

Nell B. Dale. (2003). *C++ plus data structures*. Jones & Bartlett Learning.

เฉียบวุฒิ รัตนวิลัยสกุล. (2023). โครงสร้างข้อมูล. มหาวิทยาลัยเทคโนโลยีพระจอมเกล้าพระนครเหนือ

https://ece.uwaterloo.ca/~dwharder/aads/Lecture_materials/