CS 2233-01

Data Structures and Algorithms

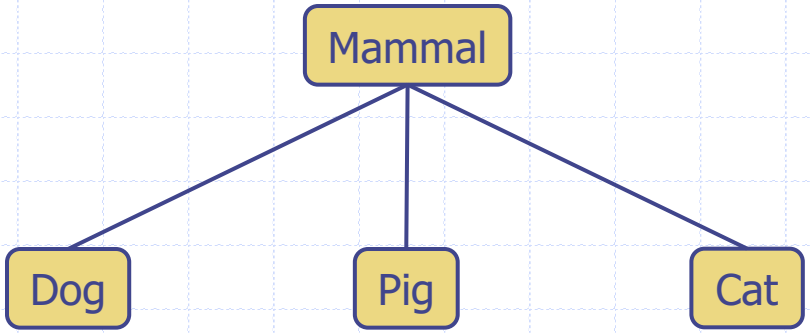
Instructor: Dr. Qingguo Wang

College of Computing & Technology

Lipscomb University

Presentation for use with the textbook Data Structures and Algorithms in Java, 6th edition, by M. T. Goodrich, R. Tamassia, and M. H. Goldwasser, Wiley, 2014

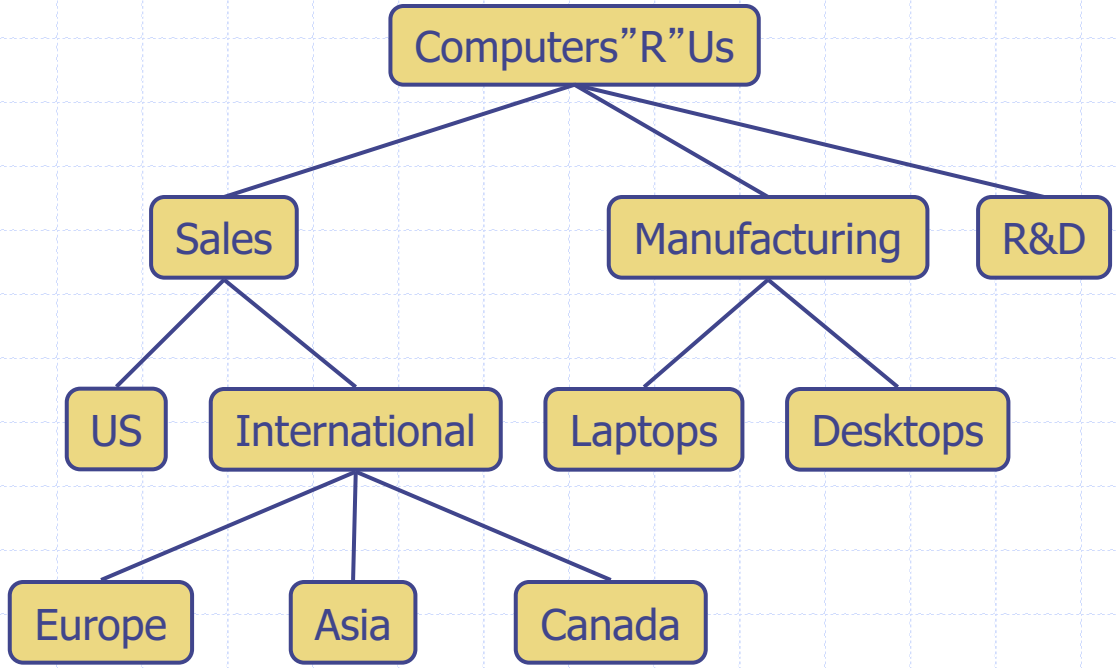
Chapter 8 Trees



What is a Tree?

What is a Tree

- In computer science, a tree is an abstract model of a hierarchical structure
- A tree consists of nodes with a parent-child relation
- Applications:
  - Organization charts
  - File systems
  - Programming environments



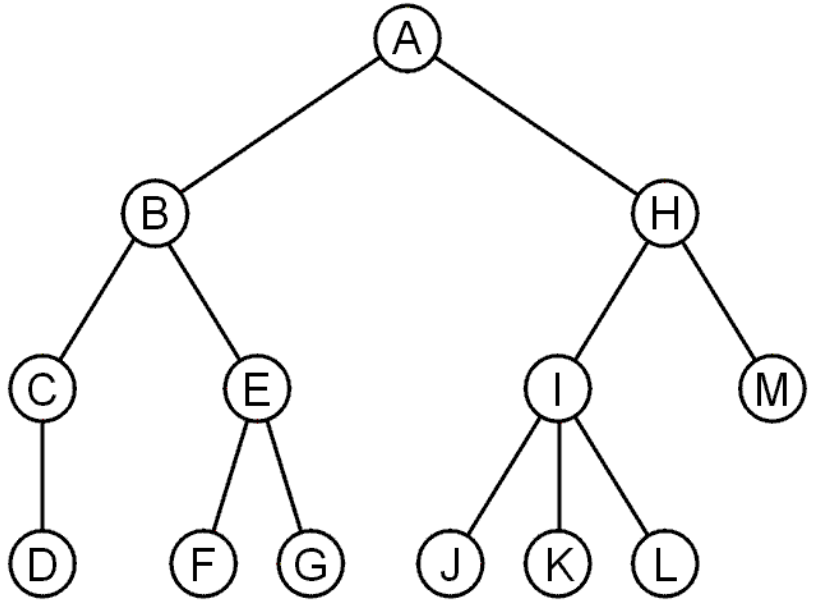
Outline

- Tree terminology
- Tree traversals
- Binary trees

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



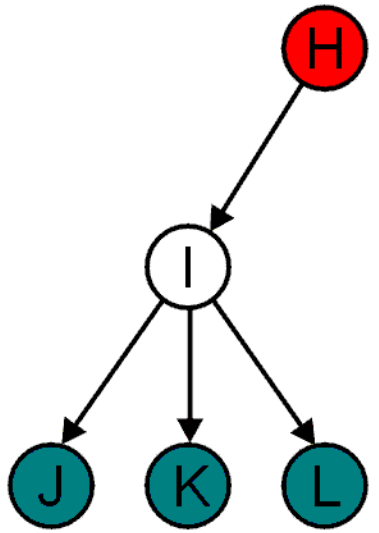
Terminology: Children, parent

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

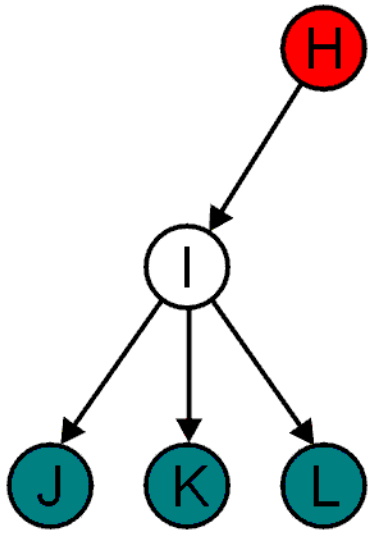


Terminology: degree, siblings

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

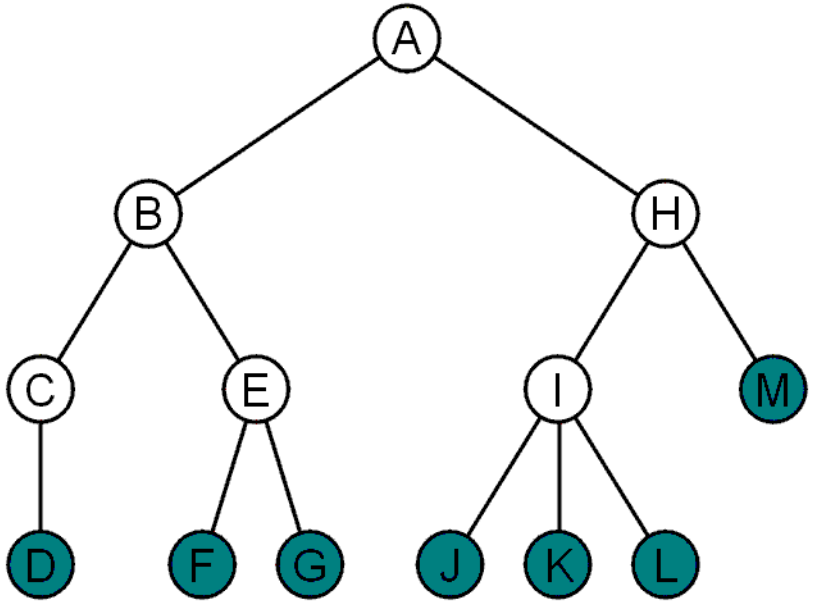
Nodes with the same parent are ***siblings***

- J, K, and L are siblings



Terminology: leaf nodes, internal nodes

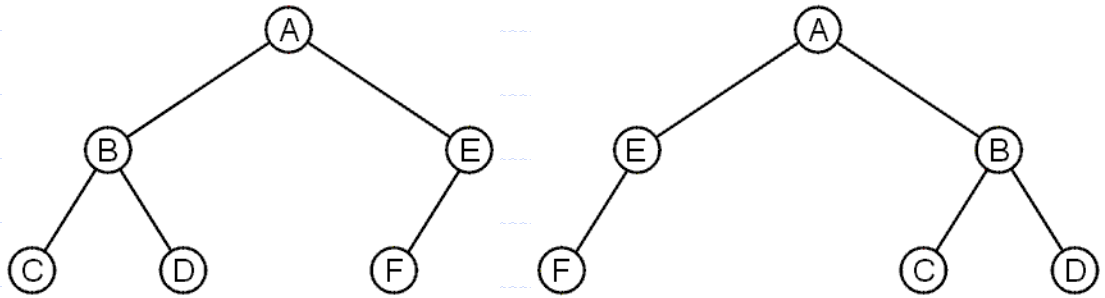
- Nodes with degree zero are also called ***leaf nodes***
- All other nodes are said to be ***internal nodes***, that is, they are internal to the tree



Terminology: order tree

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

- ***unordered trees***

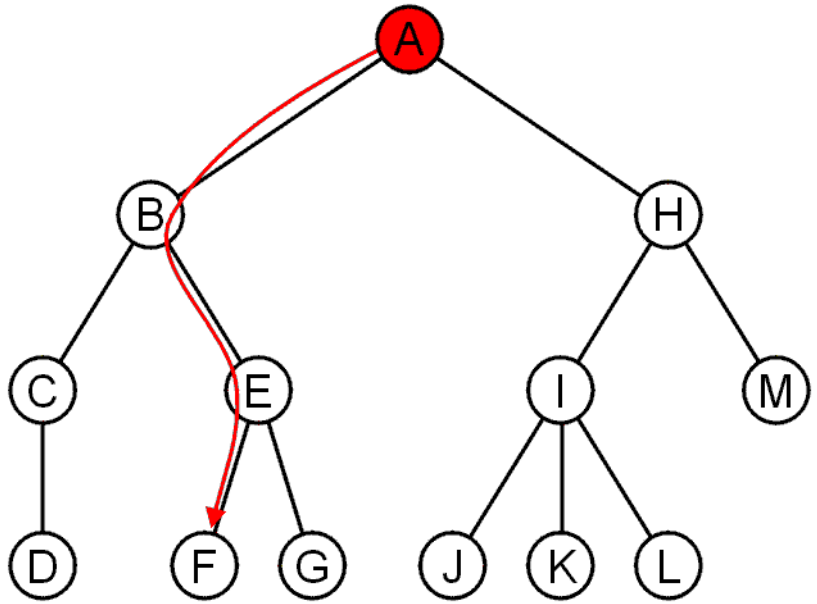


They are different if order is relevant (***ordered trees***)

- 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: path

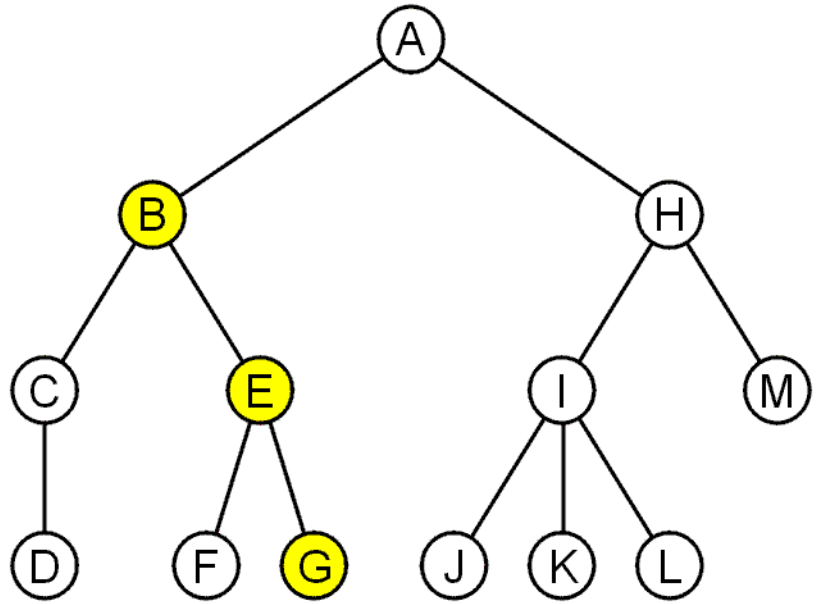
A **path** is a sequence of nodes

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

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

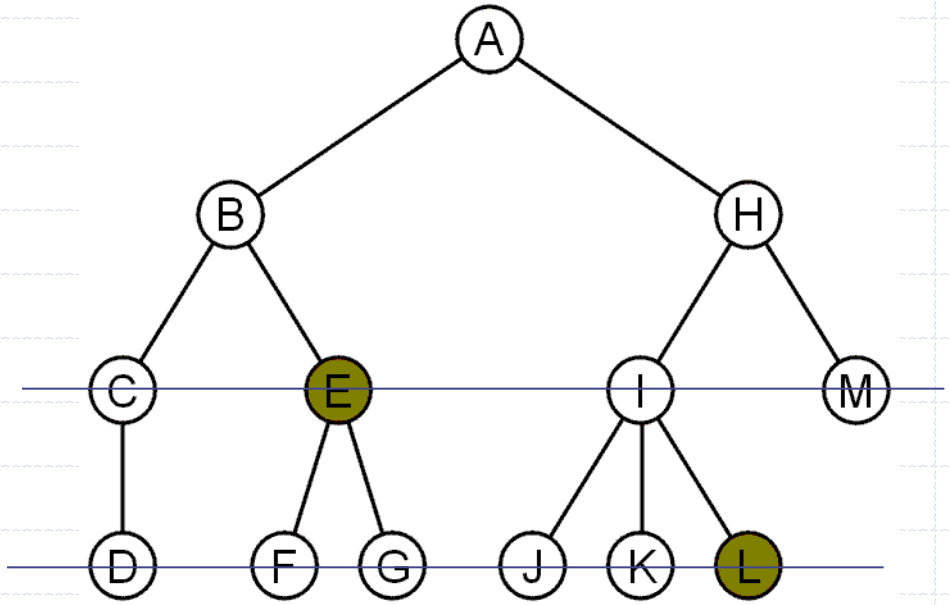
The length of this path is n

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



Terminology: depth

- 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: height

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: ancestor, descendent

If a path exists from node a to node b:

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

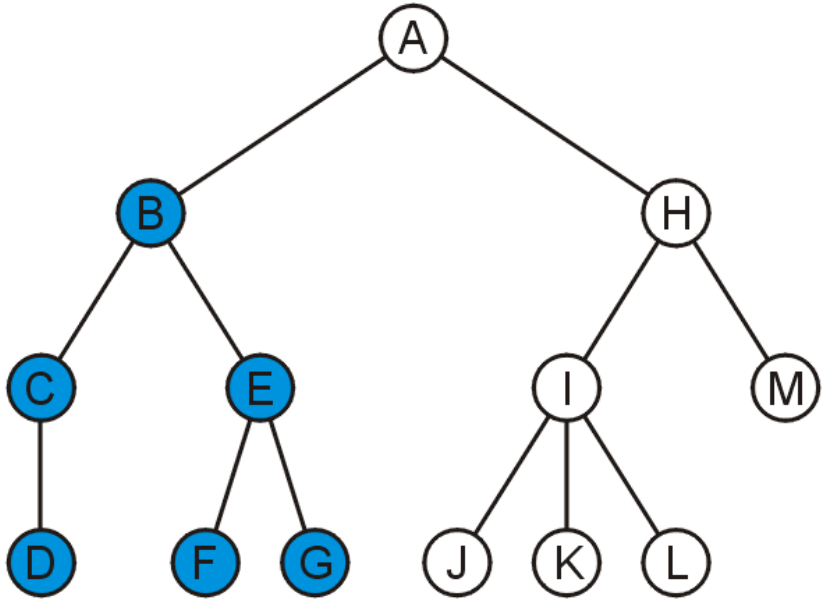
Thus, a node is both an ancestor and a descendant of itself

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

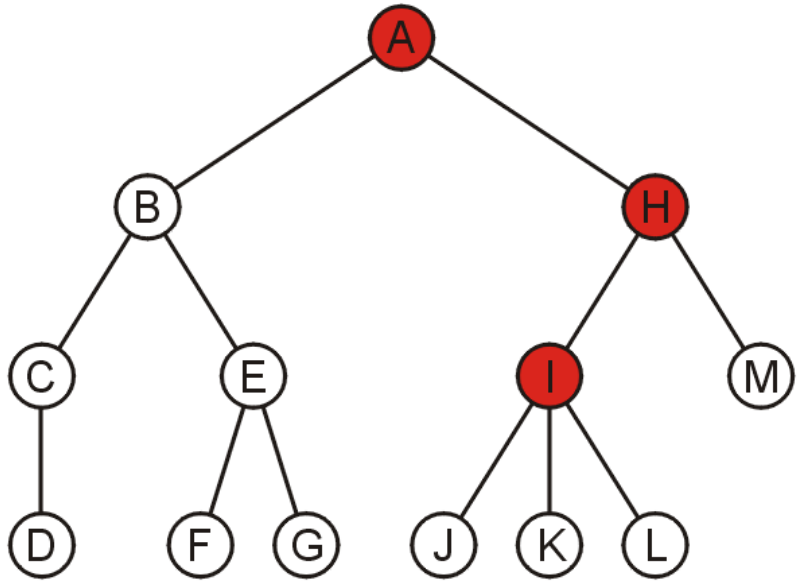
The root node is an ancestor of all nodes

Terminology: ancestor, descendent

What are the descendants of node B?

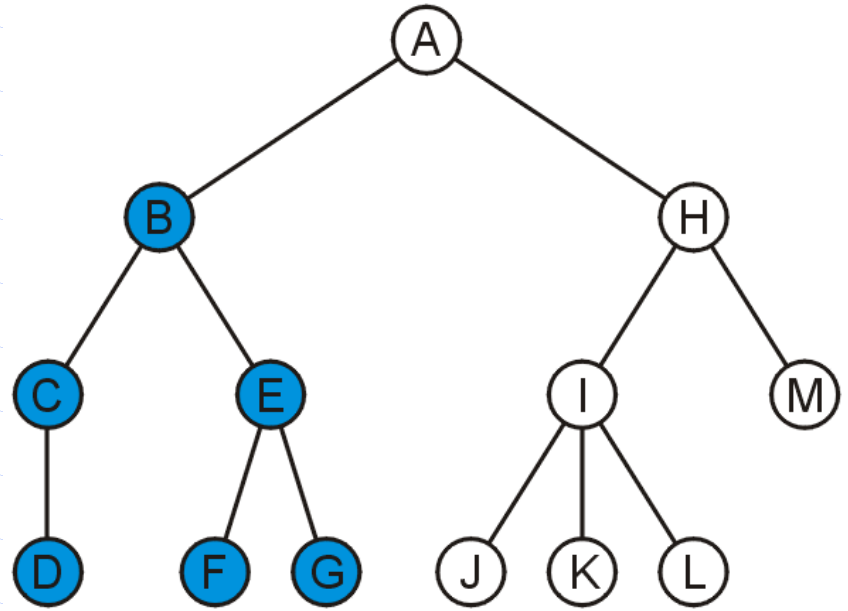


What are the ancestors of node I?

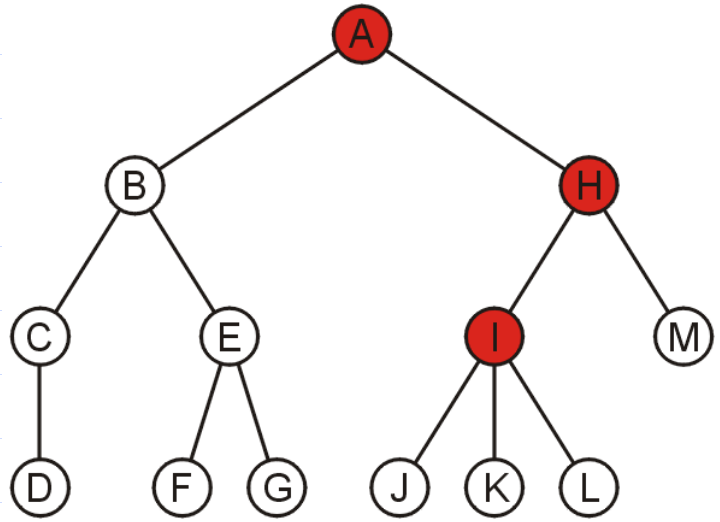


Terminology: ancestor, descendent

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



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

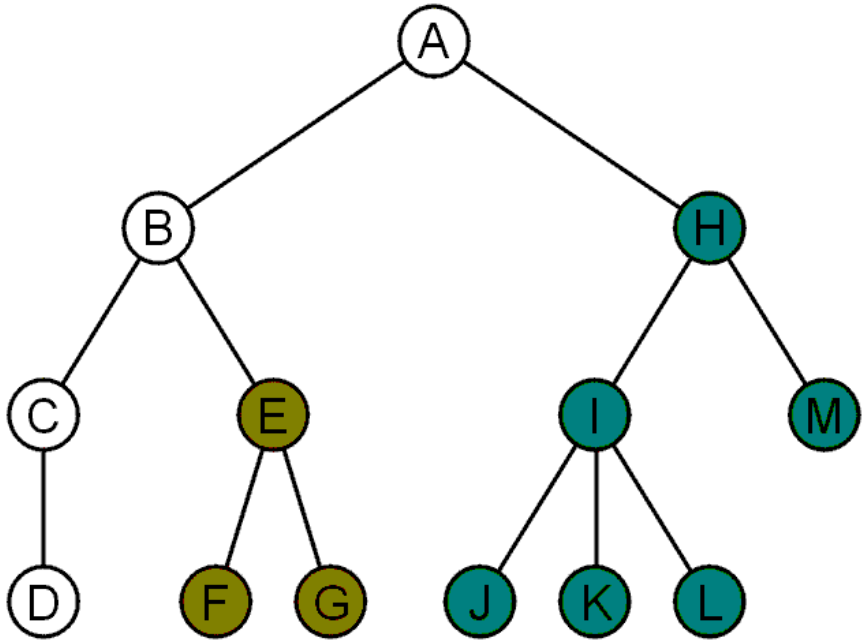


Terminology: subtree

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

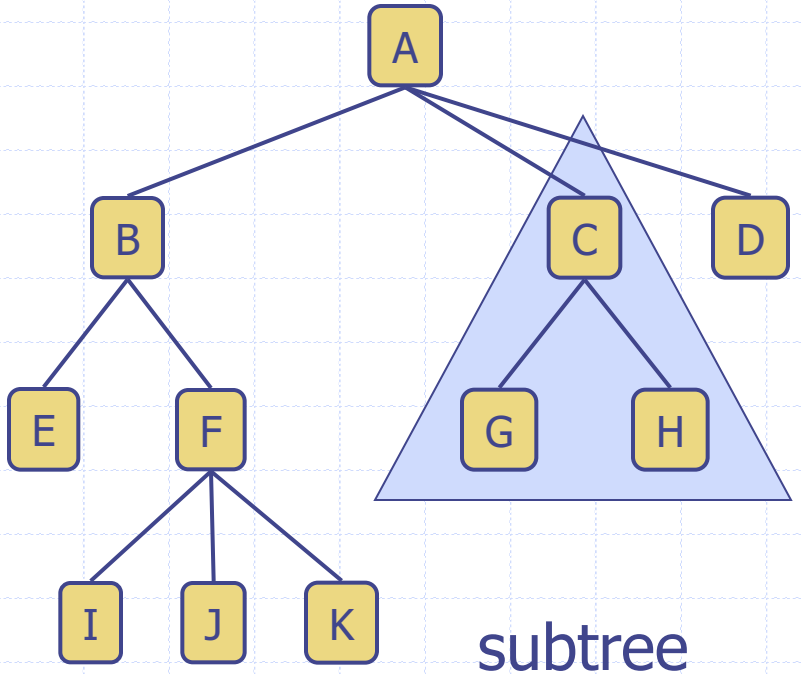
- A degree-0 node is a tree
- A node with degree n is a tree if it has n children and all of its children are disjoint trees (*i*.e., with no intersecting nodes)

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



Summary of Tree Terminology

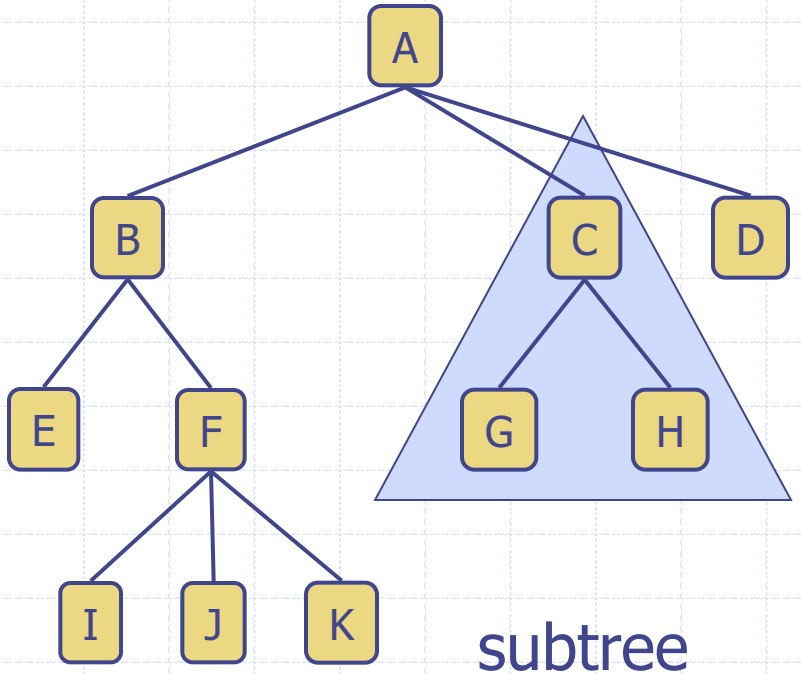
- **Root**:
- **Internal** node
- **External** node (a.k.a. leaf)
- **Ancestors** of a node
- **Depth** of a node
- **Height** of a tree
- **Descendant** of a node
- **Subtree**



Summary of Tree Terminology

- **Root**: node without parent (A)
- **Internal** node: node with at least one child (A, B, C, F)
- **External** node (a.k.a. leaf): node without children (E, I, J, K, G, H, D)
- **Ancestors** of a node: parent, grandparent, great-grandparent, etc.
- **Depth** of a node: number of ancestors
- **Height** of a tree: maximum depth of any node (3)
- **Descendant** of a node: child, grandchild, great-grandchild, etc.

- **Subtree**: tree consisting of a node and its descendants



Tree ADT

- We use positions to abstract nodes
- Generic methods:
  - integer size()
  - boolean isEmpty()
  - Iterator iterator()
  - Iterable positions()
- Accessor methods:
  - position root()
  - position parent(p)
  - Iterable children(p)
  - Integer numChildren(p)

- Query methods:
  - boolean isInternal(p)
  - boolean isExternal(p)
  - boolean isRoot(p)

- Additional update methods may be defined by data structures implementing the Tree ADT

Java Interface

Methods for a Tree interface:

```
1  /** An interface for a tree where nodes can have an arbitrary number of children. */
2  public interface Tree<E> extends Iterable<E> {
3    Position<E> root( );
4    Position<E> parent(Position<E> p) throws IllegalArgumentException;
5    Iterable<Position<E>> children(Position<E> p)
6                                          throws IllegalArgumentException;
7    int numChildren(Position<E> p) throws IllegalArgumentException;
8    boolean isInternal(Position<E> p) throws IllegalArgumentException;
9    boolean isExternal(Position<E> p) throws IllegalArgumentException;
10   boolean isRoot(Position<E> p) throws IllegalArgumentException;
11   int size( );
12   boolean isEmpty( );
13   Iterator<E> iterator( );
14   Iterable<Position<E>> positions( );
15 }
```

Computing Depth

- Let p be a position within tree T. The ***depth*** of p is the number of ancestors of p, other than p itself.

Computing Depth

- Let p be a position within tree T. The ***depth*** of p is the number of ancestors of p, other than p itself.

- The depth of p can also be recursively defined as follows:
  - If p is the root, then the depth of p is 0.
  - Otherwise, the depth of p is one plus the depth of the parent of p.

Computing Depth

```
1    /** Returns the number of levels separating Position p from the root. */
2    public int depth(Position<E> p) {
3      if (isRoot(p))
4        return 0;
5      else
6        return 1 + depth(parent(p));
7    }
```

Code Fragment 8.3: Method depth, as implemented within the AbstractTree class.

Computing Height

- The ***height*** of a tree to be equal to the maximum of the depths of its positions (or zero, if the tree is empty).

Computing Height (cont.)

- The ***height*** of a tree to be equal to the maximum of the depths of its positions (or zero, if the tree is empty).

```
private int height (p) {
    int h = 0;
    for (Position<E> p: positions())
        if (isExternal(p))
            h = Math.max(h, depth(p));
    return h;
}
```

Computing Height (cont.)

- We define the ***height*** of a position p in a tree T as follows:
  - If p is a leaf, then the height of p is 0.
  - Otherwise, the height of p is one more than the maximum of the heights of p's children.

- **Proposition 8.3**: The height of the root of a nonempty tree T, according to the recursive definition, equals the maximum depth among all leaves of tree T

Computing Height

```
1    /** Returns the height of the subtree rooted at Position p. */
2    public int height(Position<E> p) {
3      int h = 0;                                  // base case if p is external
4      for (Position<E> c : children(p))
5        h = Math.max(h, 1 + height(c));
6      return h;
7    }
```

Code Fragment 8.5: Method height for computing the height of a subtree rooted at a position p of an AbstractTree.

Outline

- Tree terminology
- Tree traversals
- Binary trees

Preorder Traversal

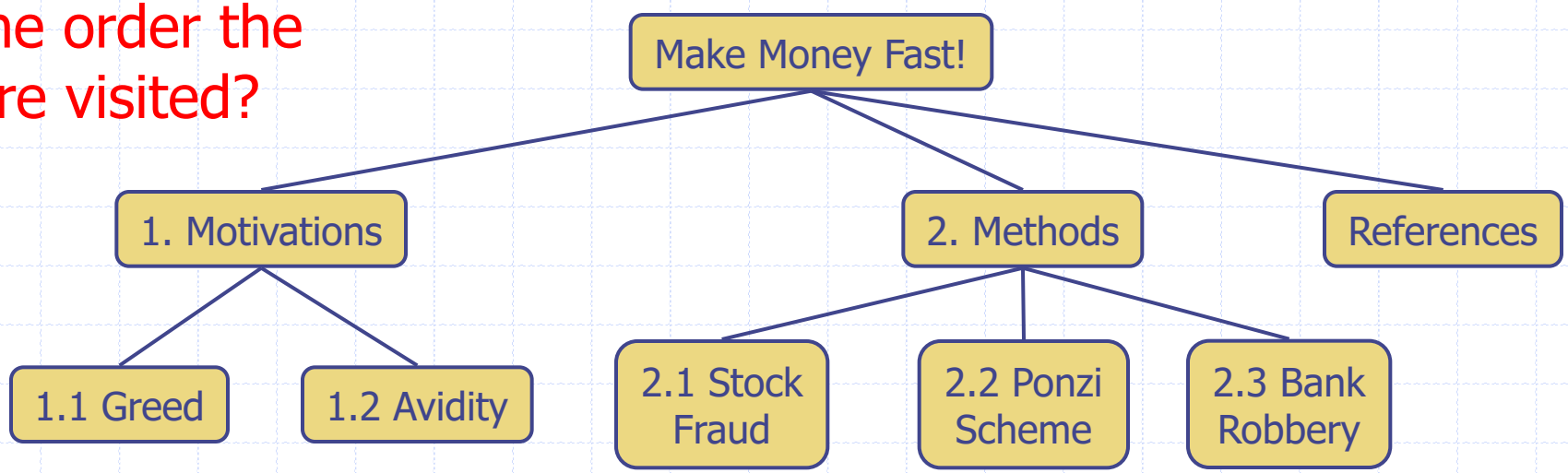
- A traversal visits the nodes of a tree in a systematic manner
- In a preorder traversal, a node is visited **before** its descendants
- Application: print a structured document

Preorder Traversal

- A traversal visits the nodes of a tree in a systematic manner
- In a preorder traversal, a node is visited **before** its descendants
- Application: print a structured document

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

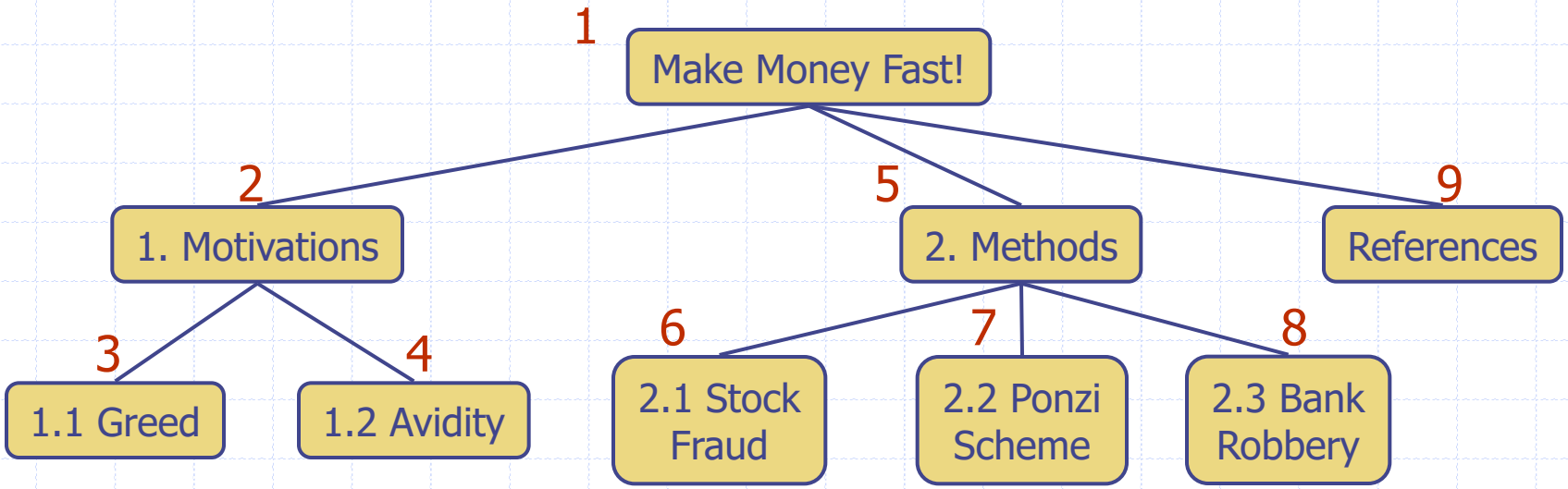
What is the order the nodes are visited?



Preorder Traversal (cont.)

- A traversal visits the nodes of a tree in a systematic manner
- In a preorder traversal, a node is visited **before** its descendants
- Application: print a structured document

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



Postorder Traversal

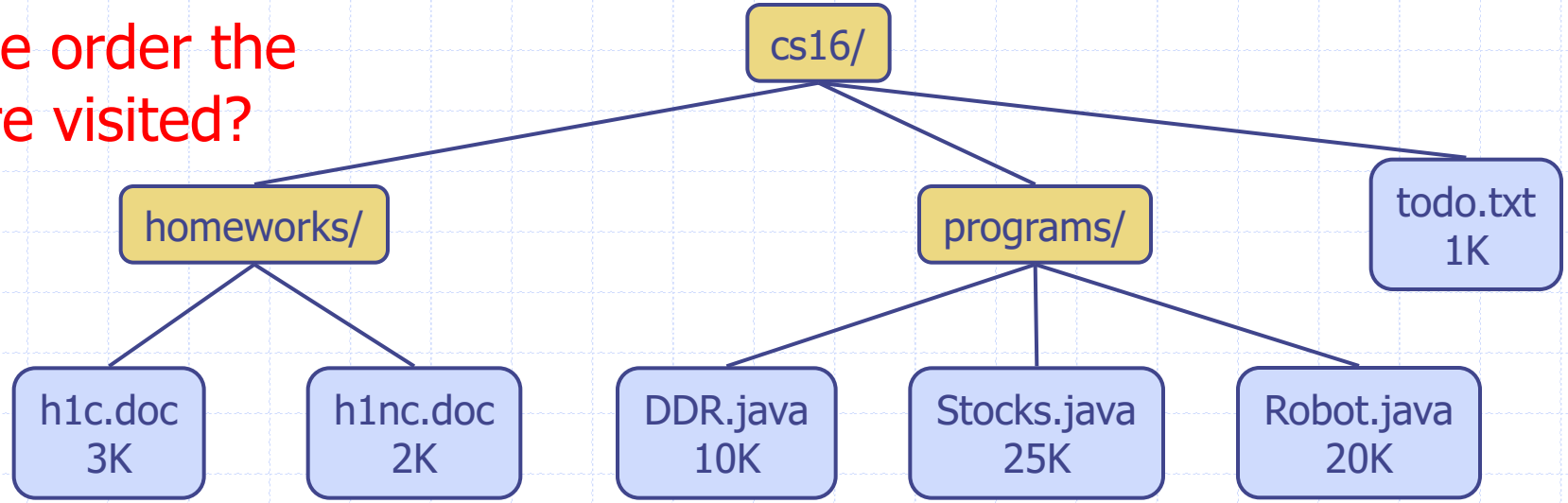
- In a postorder traversal, a node is visited **after** its descendants
- Application: compute space used by files in a directory and its subdirectories

Postorder Traversal

- In a postorder traversal, a node is visited **after** its descendants
- Application: compute space used by files in a directory and its subdirectories

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

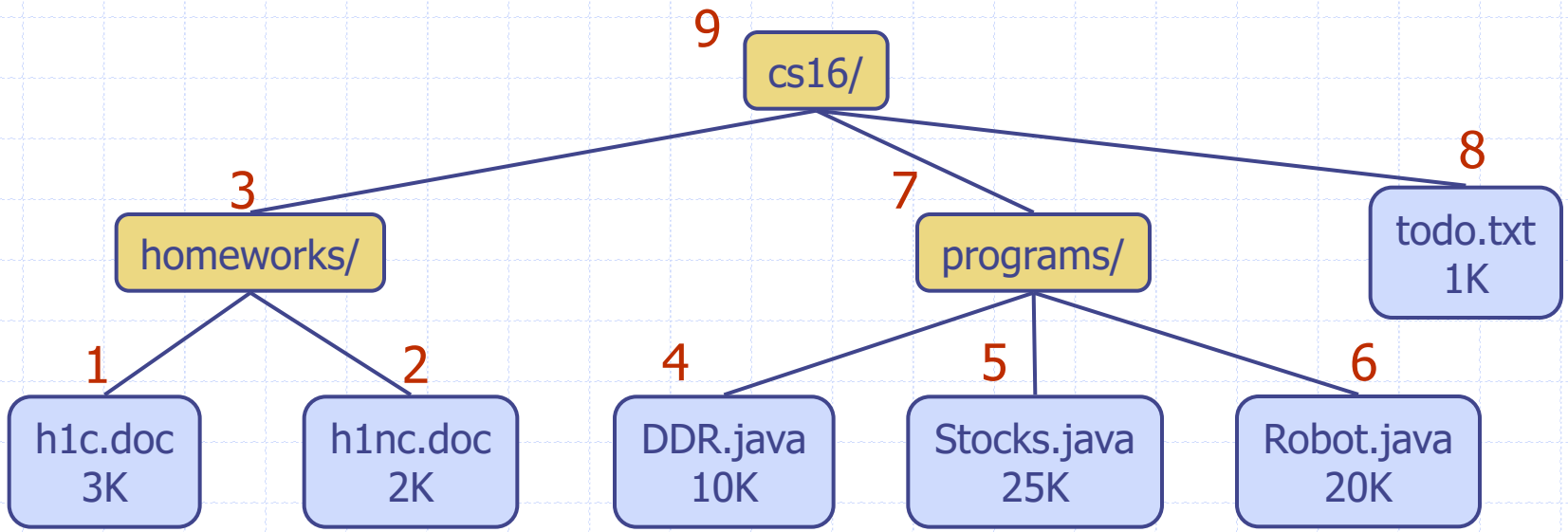
What is the order the nodes are visited?



Postorder Traversal (cont.)

- In a postorder traversal, a node is visited **after** its descendants
- Application: compute space used by files in a directory and its subdirectories

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



Outline

- Tree terminology
- Tree traversals
- Binary trees

Binary Trees

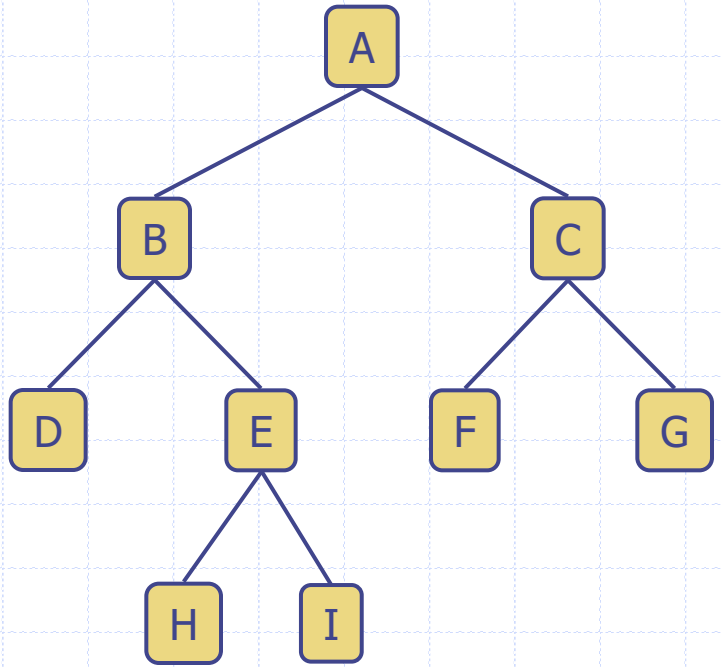
The arbitrary number of children in general trees is often unnecessary—many real-life trees are restricted to two branches

- Expression trees using binary operators
- Lossless encoding algorithms

A **binary tree** is a restriction where each node has exactly two children

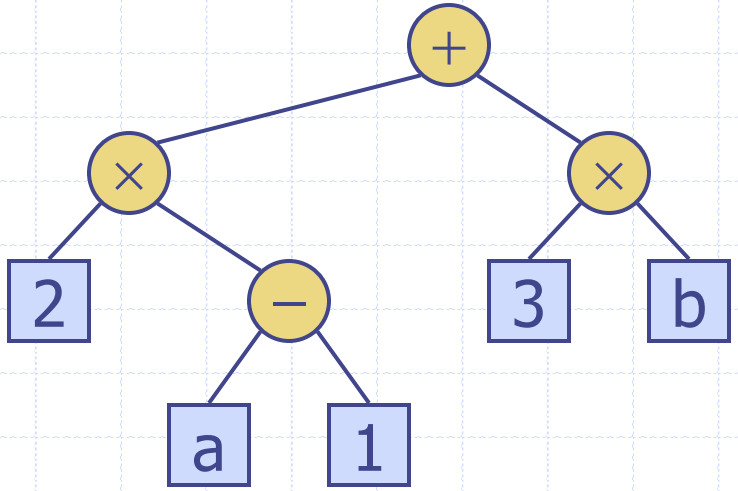
Binary Trees (cont.)

- A binary tree is a tree with the following properties:
  - Each internal node has at most **two** children (exactly two for proper binary trees)
  - 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 binary tree
- Applications:
  - arithmetic expressions
  - decision processes
  - searching



Arithmetic Expression Tree

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



Application: Expression Trees

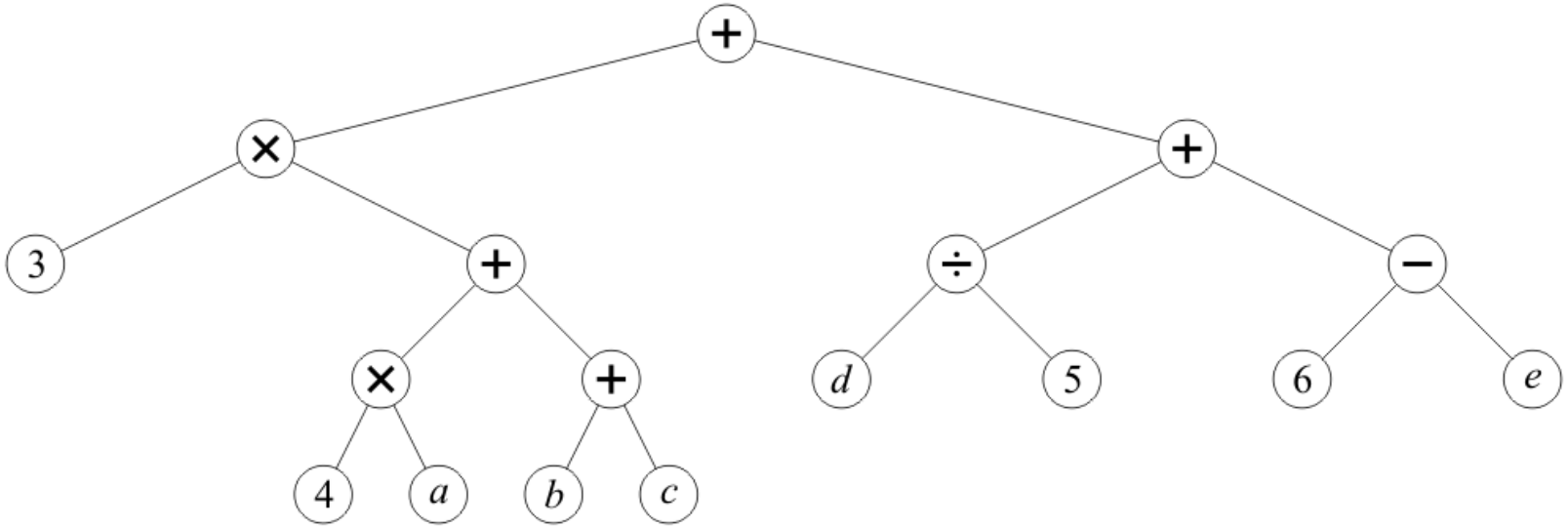
Any basic mathematical expression containing binary operators may be represented using a binary tree

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

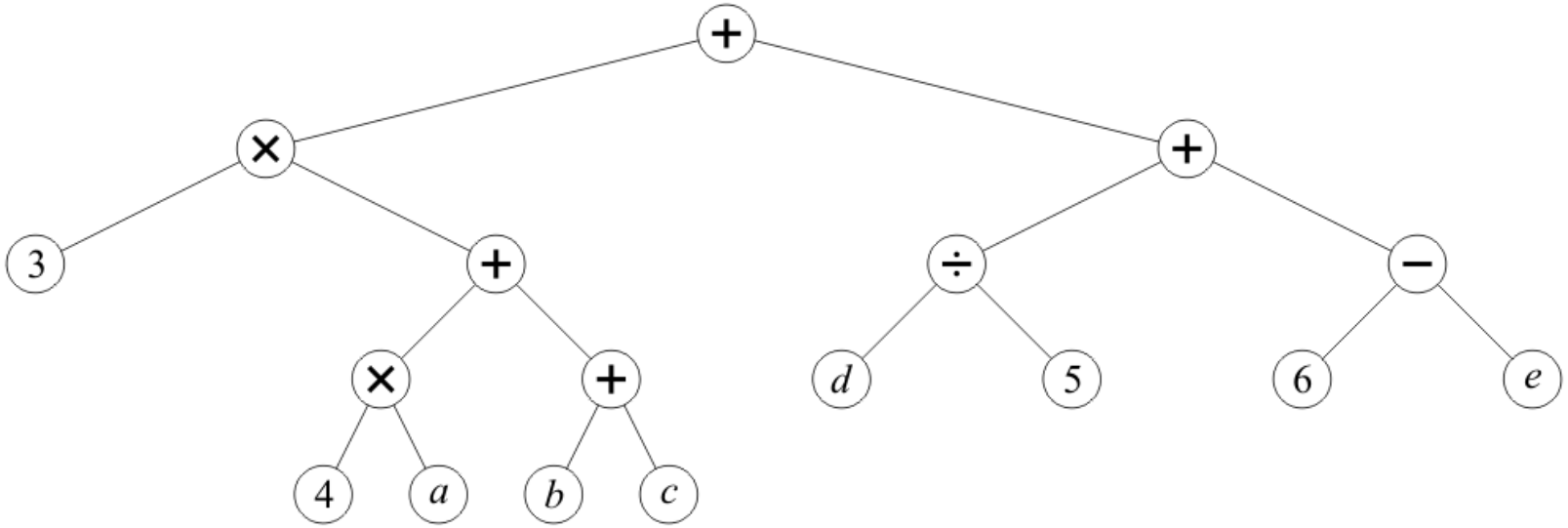
Application: Expression Trees

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

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

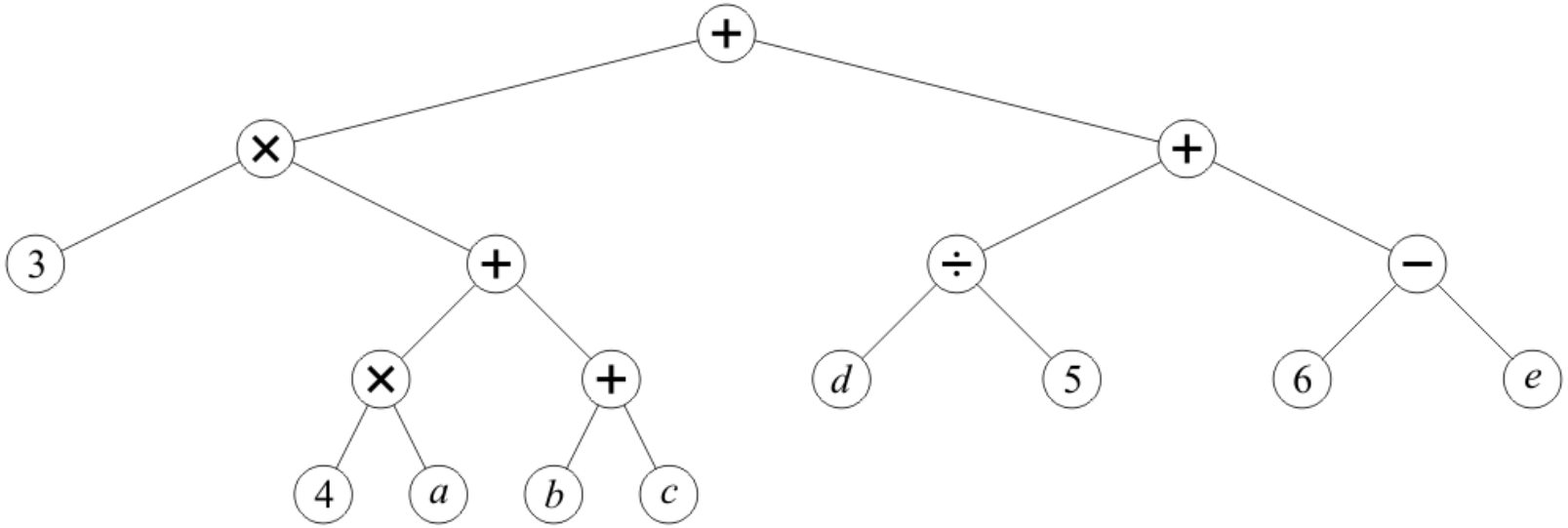


Post-order traversal of expression trees?



Application: Expression Trees

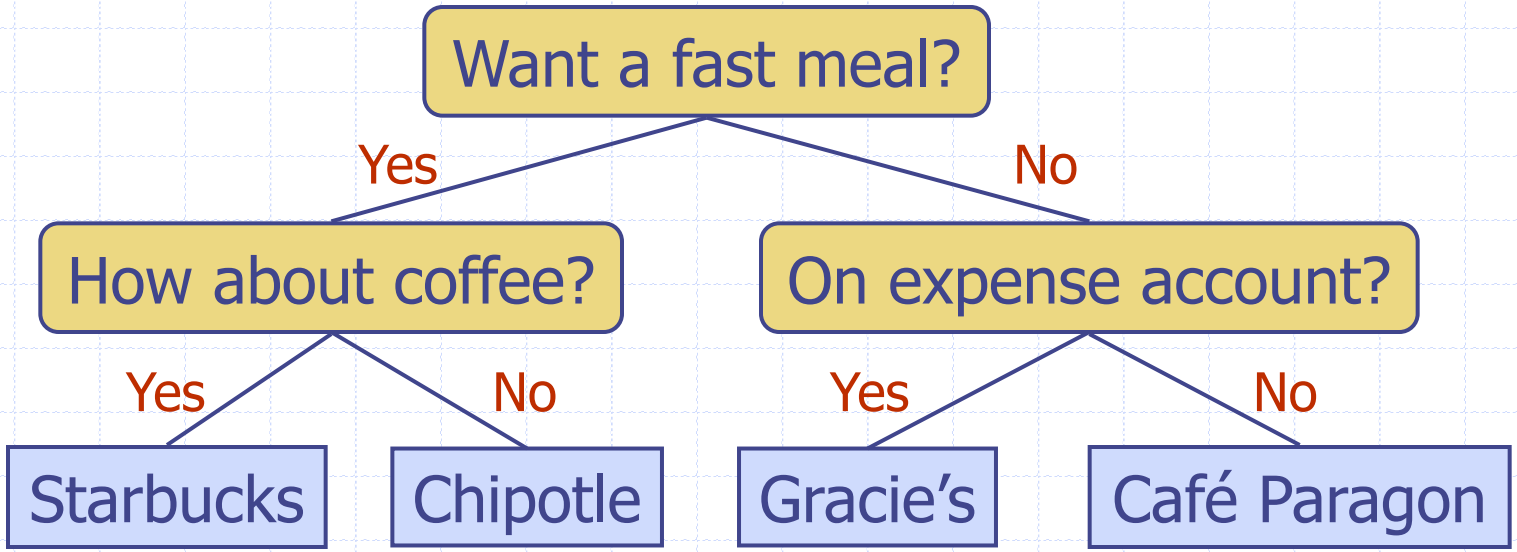
A post-order traversal converts such a tree to the postfix (reverse-Polish) format



3 4 a × b c + + × d 5 ÷ 6 e − + +

Decision Tree

- Binary tree associated with a decision process
  - internal nodes: questions with yes/no answer
  - external nodes: decisions
- Example: dining decision

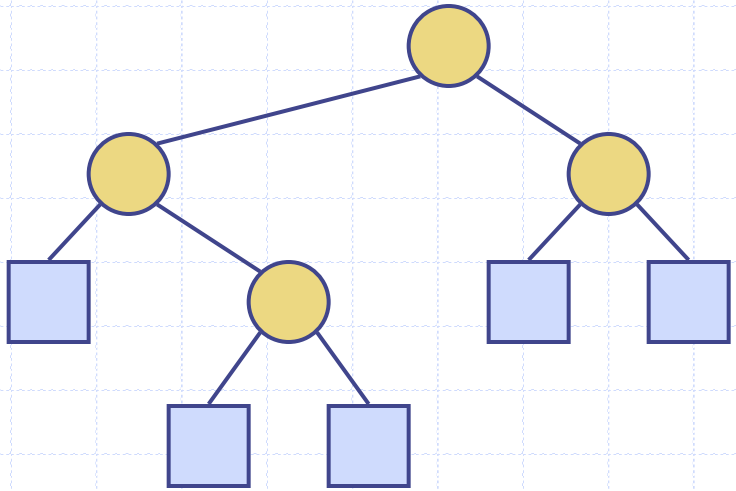


BinaryTree ADT

- The BinaryTree ADT extends the Tree ADT, i.e., it inherits all the methods of the Tree ADT
- Additional methods:
  - position left(p)
  - position right(p)
  - position sibling(p)
- The above methods return null when there is no left, right, or sibling of p, respectively
- Update methods may be defined by data structures implementing the BinaryTree ADT

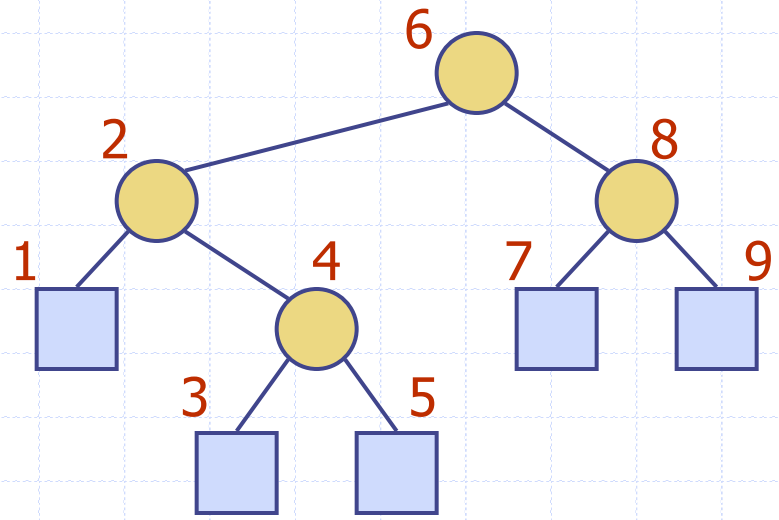
Inorder Traversal

- In an inorder traversal a node is visited **after** its left subtree and **before** its right subtree
- Application: draw a binary tree
  - x(v) = inorder rank of v
  - y(v) = depth of v



Inorder Traversal

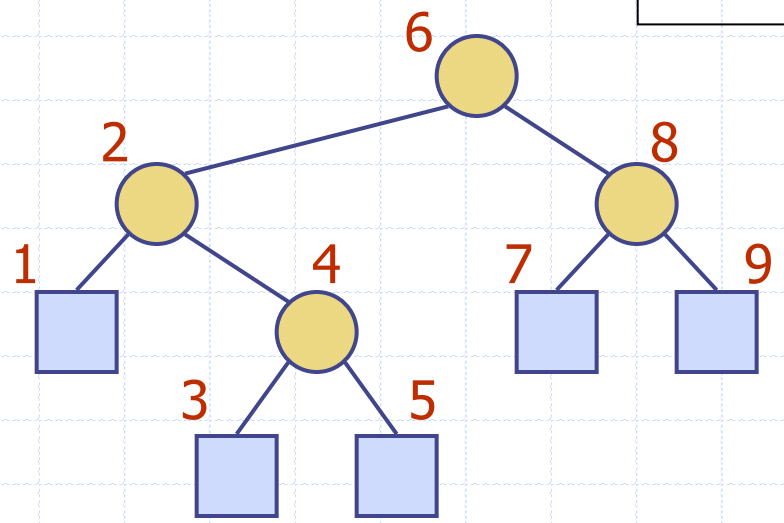
- In an inorder traversal a node is visited **after** its left subtree and **before** its right subtree
- Application: draw a binary tree
  - x(v) = inorder rank of v
  - y(v) = depth of v



Inorder Traversal

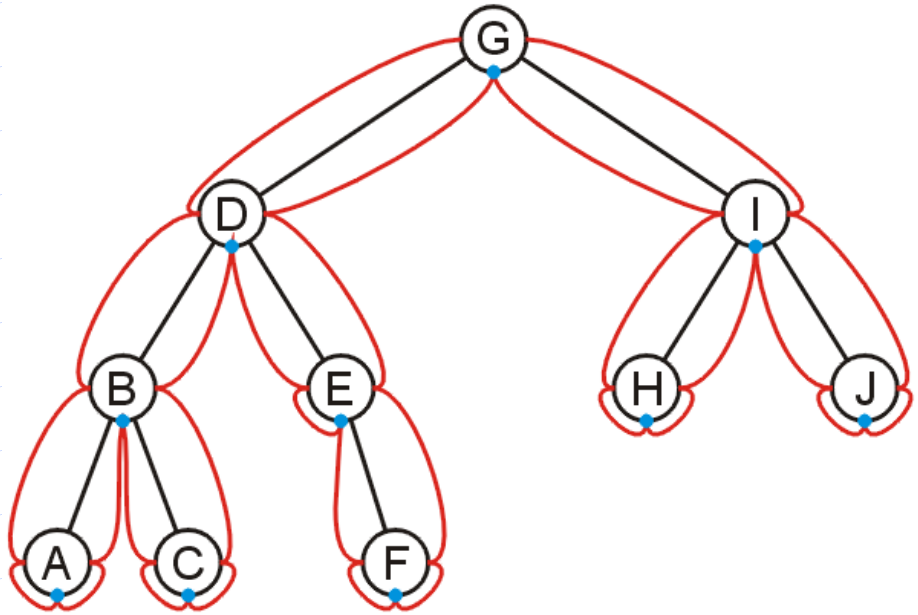
- In an inorder traversal a node is visited **after** its left subtree and **before** its right subtree
- Application: draw a binary tree
  - x(v) = inorder rank of v
  - y(v) = depth of v

```
Algorithm inOrder(v)
    if left (v) ≠ null
        inOrder (left (v))
    visit(v)
    if right(v) ≠ null
        inOrder (right (v))
```



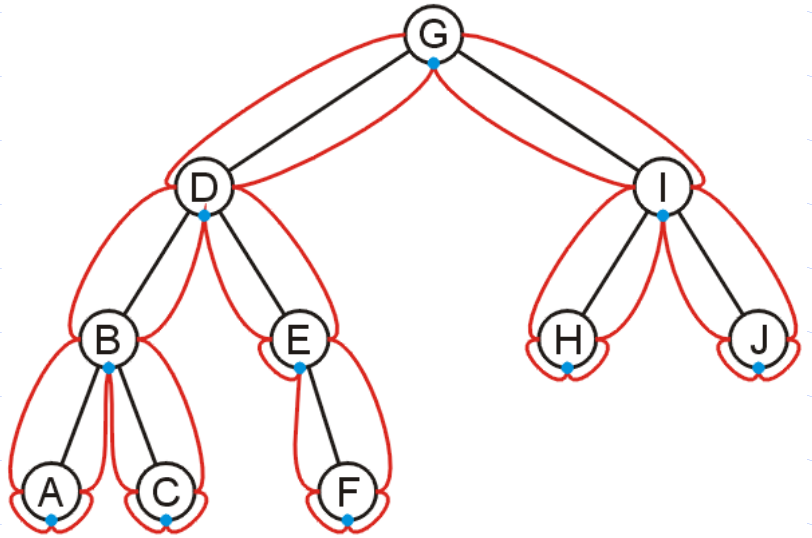
Exercise

- In the **Algorithm** *inOrder(v)*, suppose *visit (v)* prints the element of node. What is the sequence of letters printed by *inOrder(G)*?



Exercise

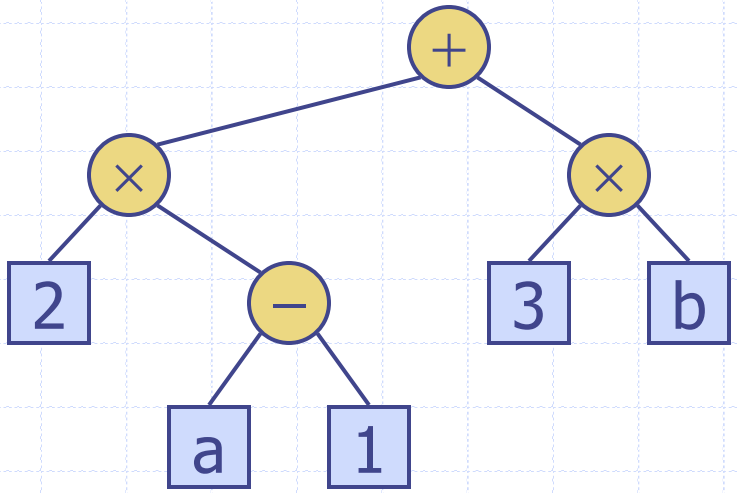
- In the **Algorithm** *inOrder(v)*, suppose *visit (v)* prints the element of node. What is the sequence of letters printed by *inOrder(G)*?



A, B, C, D, E, F, G, H, I, J

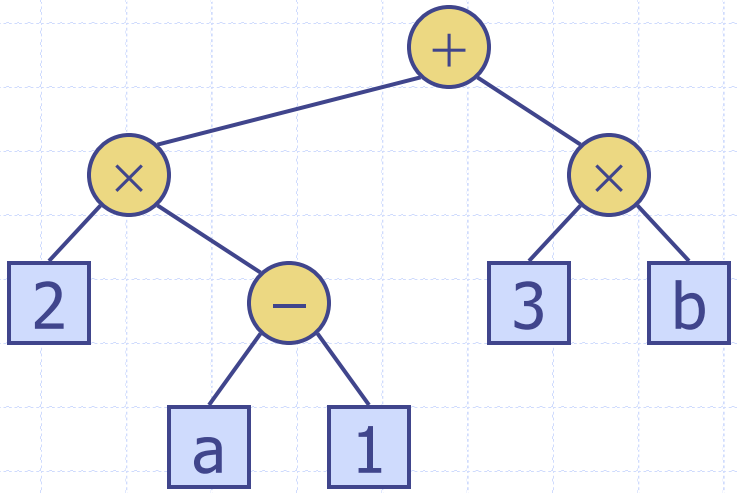
Print Arithmetic Expressions

- Specialization of an inorder traversal
  - print operand or operator when visiting node
  - print "(" before traversing left subtree
  - print ")" after traversing right subtree



Print Arithmetic Expressions

- Specialization of an inorder traversal
  - print operand or operator when visiting node
  - print "(" before traversing left subtree
  - print ")" after traversing right subtree



Algorithm *printExpression*(*v*)

```
if left (v) ≠ null
    print("(" )
    inOrder (left(v))
print(v.element ())
if right(v) ≠ null
    inOrder (right(v))
    print (")" )
```

((2 × (a – 1)) + (3 × b))

Application: Expression Trees

Humans think in in-order

Computers think in post-order:

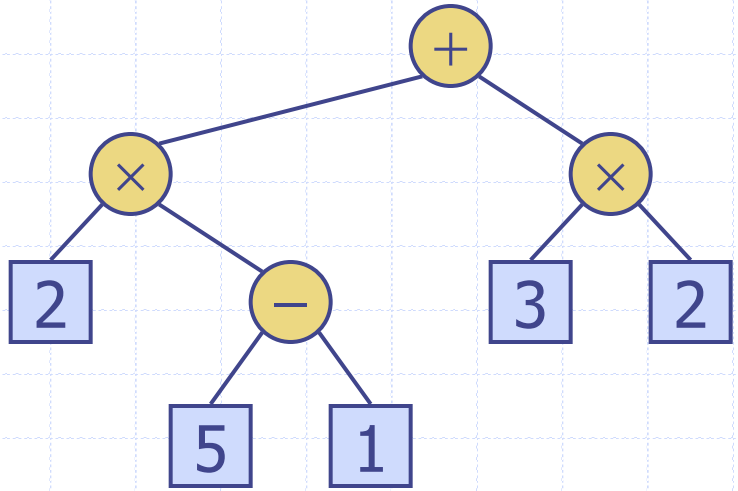
- Both operands must be loaded into registers
- The operation is then called on those registers

Most use in-order notation (C, C++, Java, C#, etc.)

- Necessary to translate in-order into post-order

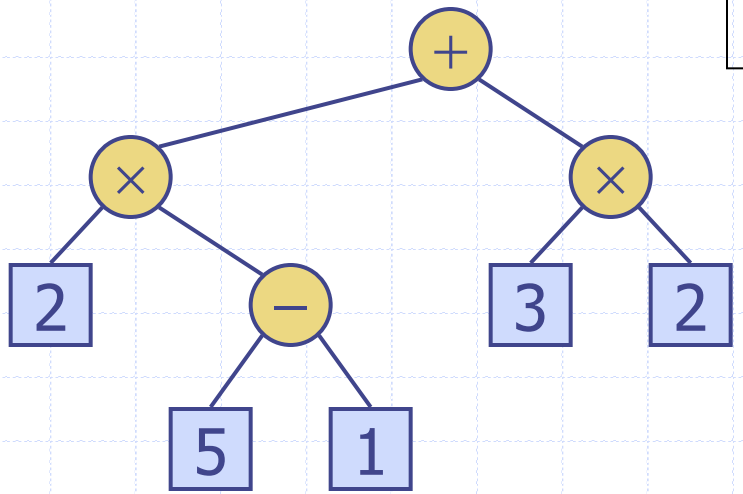
Evaluate Arithmetic Expressions

- Specialization of a postorder traversal
  - recursive method returning the value of a subtree
  - when visiting an internal node, combine the values of the subtrees



Evaluate Arithmetic Expressions

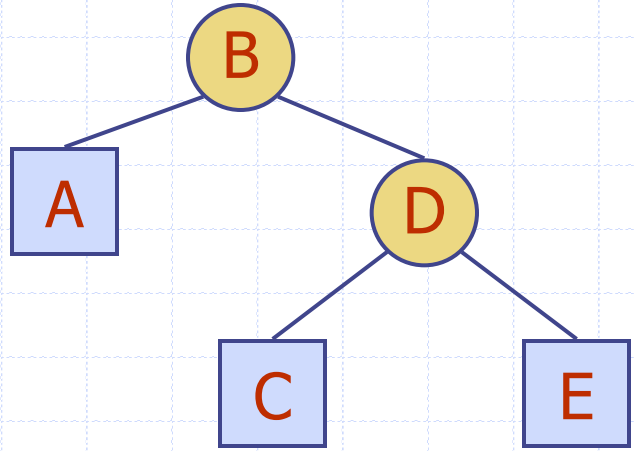
- Specialization of a postorder traversal
  - recursive method returning the value of a subtree
  - when visiting an internal node, combine the values of the subtrees

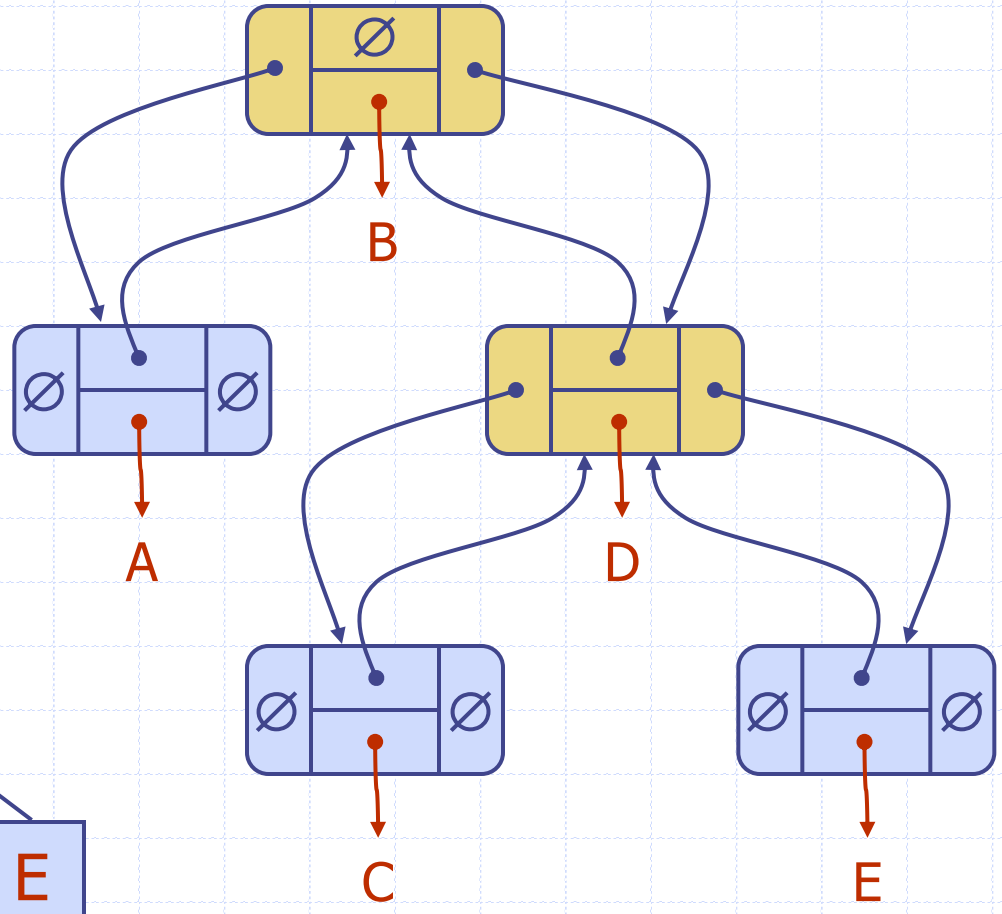



```
Algorithm evalExpr(v)
    if isExternal (v)
        return v.element ()
    else
        x ← evalExpr(left(v))
        y ← evalExpr(right(v))
        ◊ ← operator stored at v
        return x ◊ y
```

Linked Structure for Binary Trees

- A node is represented by an object storing
  - Element
  - Parent node
  - Left child node
  - Right child node
- Node objects implement the Position ADT





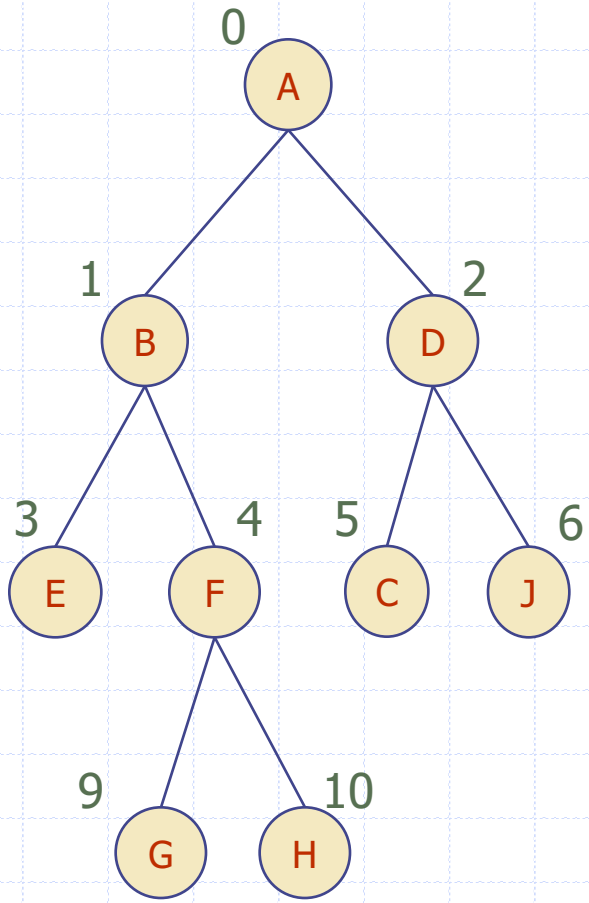
Efficiency

Method	Running Time
size, isEmpty	$O(1)$
root, parent, left, right, sibling, children, numChildren	$O(1)$
isInternal, isExternal, isRoot	$O(1)$
addRoot, addLeft, addRight, set, attach, remove	$O(1)$
depth(p)	$O(d_p + 1)$
height	$O(n)$

Table 8.1: Running times for the methods of an n-node binary tree implemented with a linked structure. The space usage is $O(n)$.

Array-Based Representation of Binary Trees

- Nodes are stored in an array A

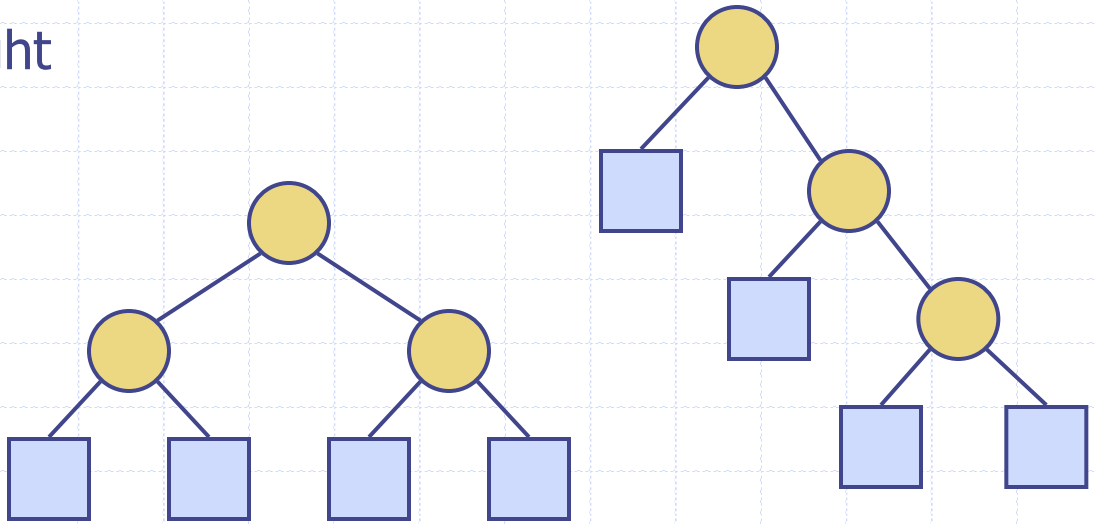


Node v is stored at A[rank(v)]

- rank(root) = 0
- if node is the left child of parent(node),
  rank(node) = 2 · rank(parent(node)) + 1
- if node is the right child of parent(node),
  rank(node) = 2 · rank(parent(node)) + 2

Properties of Proper Binary Trees

- **Proper Binary Trees**: each internal node has exactly two children
  - Notation
    - n number of nodes
    - e number of external nodes
    - i number of internal nodes
    - h height



- Properties:
  - $e = i + 1$
  - $n = 2e - 1$
  - $h \leq i$
  - $h \leq (n - 1)/2$
  - $e \leq 2^h$
  - $h \geq \log_2 e$
  - $h \geq \log_2 (n + 1) - 1$

Homework 4: Incorrect use of stacks, queues, & deques

- D.queF(S.pop);
- S.push(D.dequeF);

- D.addFirst(S.dequeue());
- S.enqueue(D.dequeue());

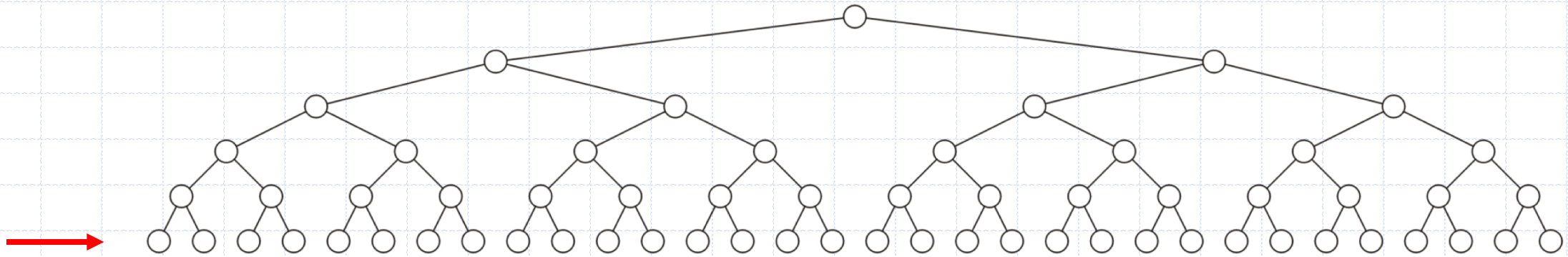
- Q[i] = D[i];

- Q.add(D.pop());

Perfect binary tree

A perfect binary tree of height h is a binary tree where

- All leaf nodes have the same depth h
- All other nodes are full



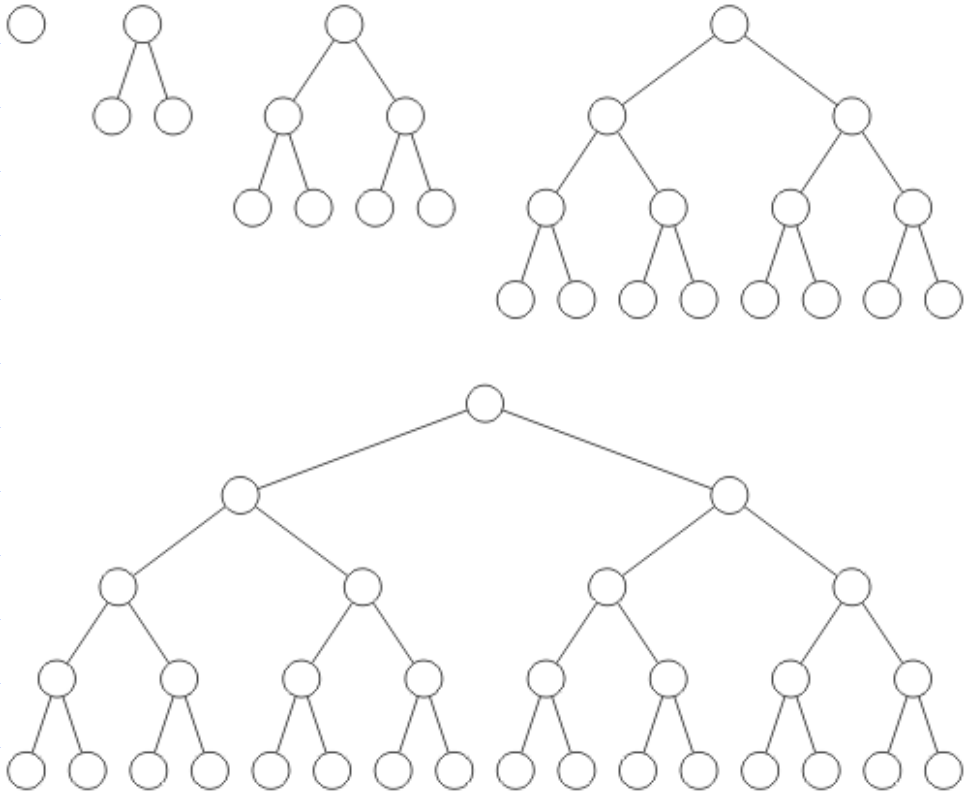
Definition

Recursive definition:

- A binary tree of height $h = 0$ is perfect
- A binary tree with height $h > 0$ is a perfect if both sub-trees are prefect binary trees of height $h - 1$

Examples

Perfect binary trees of height $h = 0, 1, 2, 3$ and 4



Theorems

The properties of perfect binary trees:

- A perfect tree has $2^{h+1} - 1$ nodes
- The height is $\Theta(\ln(n))$
- There are 2^h leaf nodes
- The average depth of a node is $\Theta(\ln(n))$

$2^{h+1} - 1$ Nodes

Theorem

A perfect binary tree of height h has $2^{h+1} - 1$ nodes

Proof:

We will use mathematical induction:

1. Show that it is true for $h = 0$
2. Assume it is true for an arbitrary h
3. Show that the truth for h implies the truth for $h + 1$

$2^{h+1} - 1$ Nodes

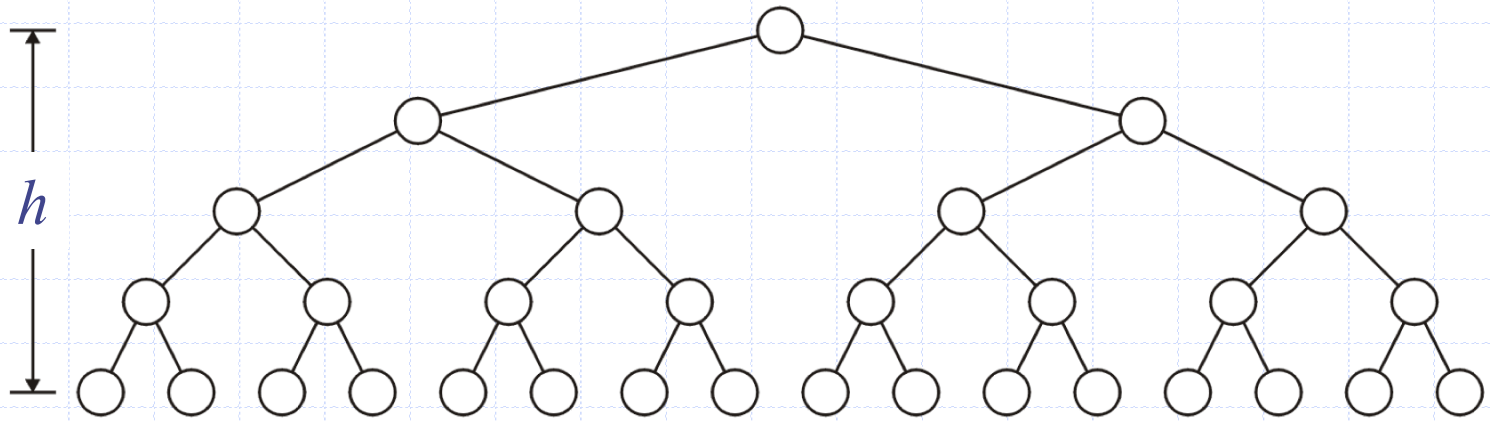
The base case:

- When $h = 0$ we have a single node $n = 1$
- The formula is correct: $2^{0+1} - 1 = 1$

$2^{h+1} - 1$ Nodes

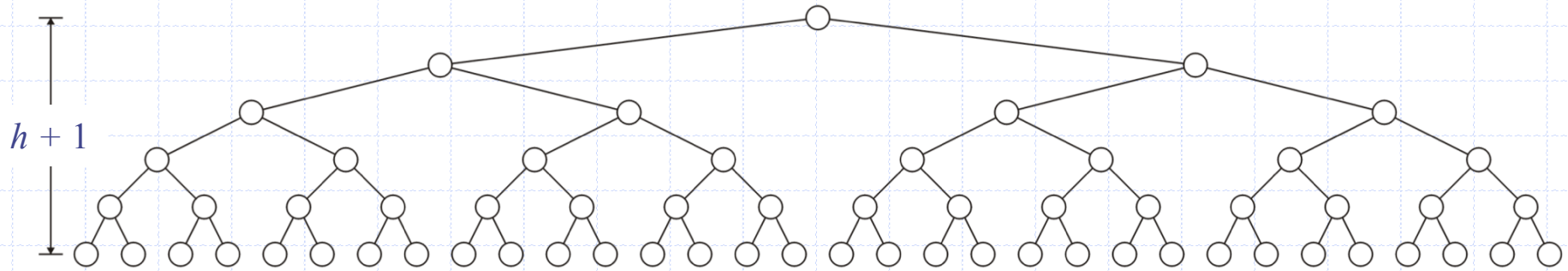
The inductive step:

- If the height of the tree is h, then assume that the number of nodes is $n = 2^{h+1} - 1$



$2^{h+1} - 1$ Nodes

We must show that a tree of height $h + 1$ has $n = 2^{(h+1)+1} - 1 = 2^{h+2} - 1$ nodes

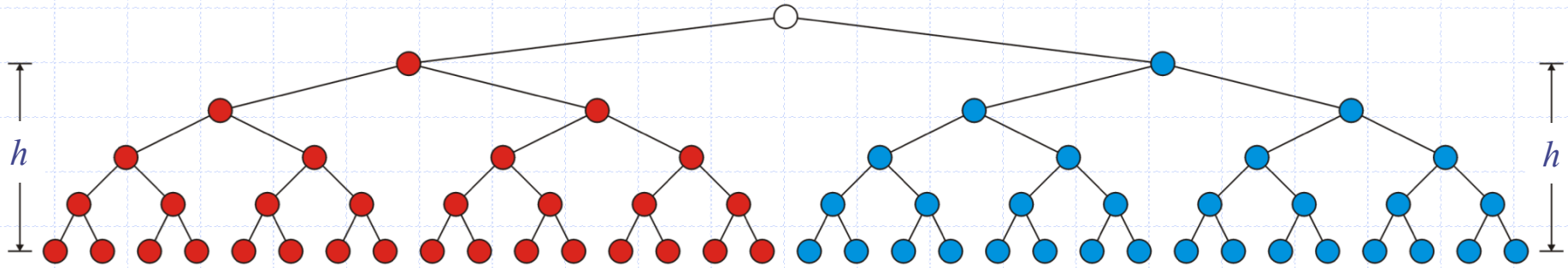


$2^{h+1} - 1$ Nodes

Using the recursive definition, both sub-trees are perfect trees of height h

- By assumption, each sub-tree has $2^{h+1} - 1$ nodes
- Therefore the total number of nodes is

$$(2^{h+1} - 1) + 1 + (2^{h+1} - 1) = 2^{h+2} - 1$$



$2^{h+1} - 1$ Nodes

Consequently

The statement is true for $h = 0$ and the truth of the statement for an arbitrary h implies the truth of the statement for $h + 1$.

Therefore, by the process of mathematical induction, the statement is true for all $h \geq 0$

Logarithmic Height

Theorem

A perfect binary tree with n nodes has height $\lg(n + 1) - 1$

Logarithmic Height

Theorem

A perfect binary tree with n nodes has height $\lg(n+1) - 1$

Proof

Solving $n = 2^{h+1} - 1$ for h:

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

$$\lg(n+1) = h + 1$$

$$h = \lg(n+1) - 1$$

Logarithmic Height

Lemma

$$\lg(n+1) - 1 = \Theta(\ln(n))$$

Proof

$$\lim_{n\to\infty} \frac{\lg(n+1)-1}{\ln(n)} = \lim_{n\to\infty} \frac{\frac{1}{(n+1)\ln(2)}}{\frac{1}{n}} = \lim_{n\to\infty} \frac{n}{(n+1)\ln(2)} = \lim_{n\to\infty} \frac{1}{\ln(2)} = \frac{1}{\ln(2)}$$

2^h Leaf Nodes

Theorem

A perfect binary tree with height h has 2^h leaf nodes

Proof (by induction):

When $h = 0$, there is $2^0 = 1$ leaf node.

2^h Leaf Nodes

Theorem

A perfect binary tree with height h has 2^h leaf nodes

Proof (by induction):

When $h = 0$, there is $2^0 = 1$ leaf node.

Assume that a perfect binary tree of height h has 2^h leaf nodes and observe that both sub-trees of a perfect binary tree of height $h + 1$ have 2^h leaf nodes.

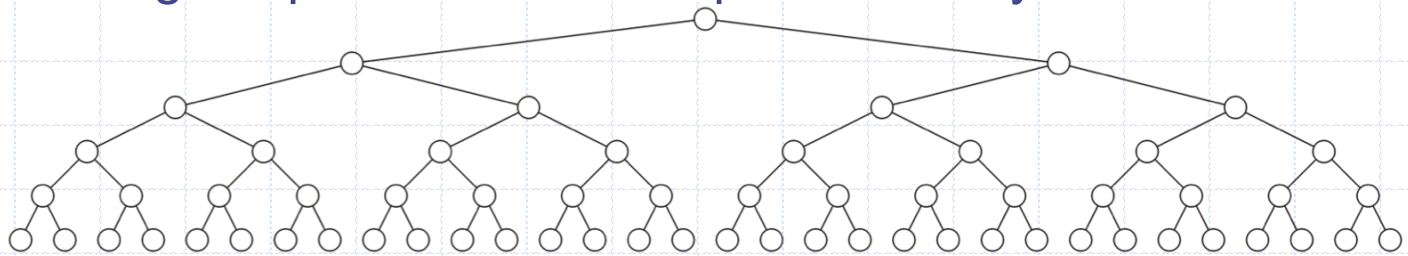
Consequence: Over half all nodes are leaf nodes:

The Average Depth of a Node

Consequence:

- 50-50 chance that a randomly selected node is a leaf node

The average depth of a node in a perfect binary tree is



Depth	Count
0	1
1	2
2	4
3	8
4	16
5	32

Sum of the depths

Number of nodes

$$\frac{\sum_{k=0}^{h} k2^k}{2^{h+1}-1} = \frac{h2^{h+1}-2^{h+1}+2}{2^{h+1}-1} = \frac{h(2^{h+1}-1)-(2^{h+1}-1)+1+h}{2^{h+1}-1}$$

$$= h-1+\frac{h+1}{2^{h+1}-1} \approx h-1 = \Theta\left(\ln(n)\right)$$

Applications

Perfect binary trees are considered to be the *ideal* case

- The height and average depth are both $\Theta(\ln(n))$

We will attempt to find trees which are as close as possible to perfect binary trees

Summary of perfect binary trees

The number of nodes: $n = 2^{h+1} - 1$

- The height: $\log_2(n+1) - 1$
- The number of leaves: 2^h
- Half the nodes are leaves
  - Average depth is $\Theta(\ln(n))$
- It is an ideal case