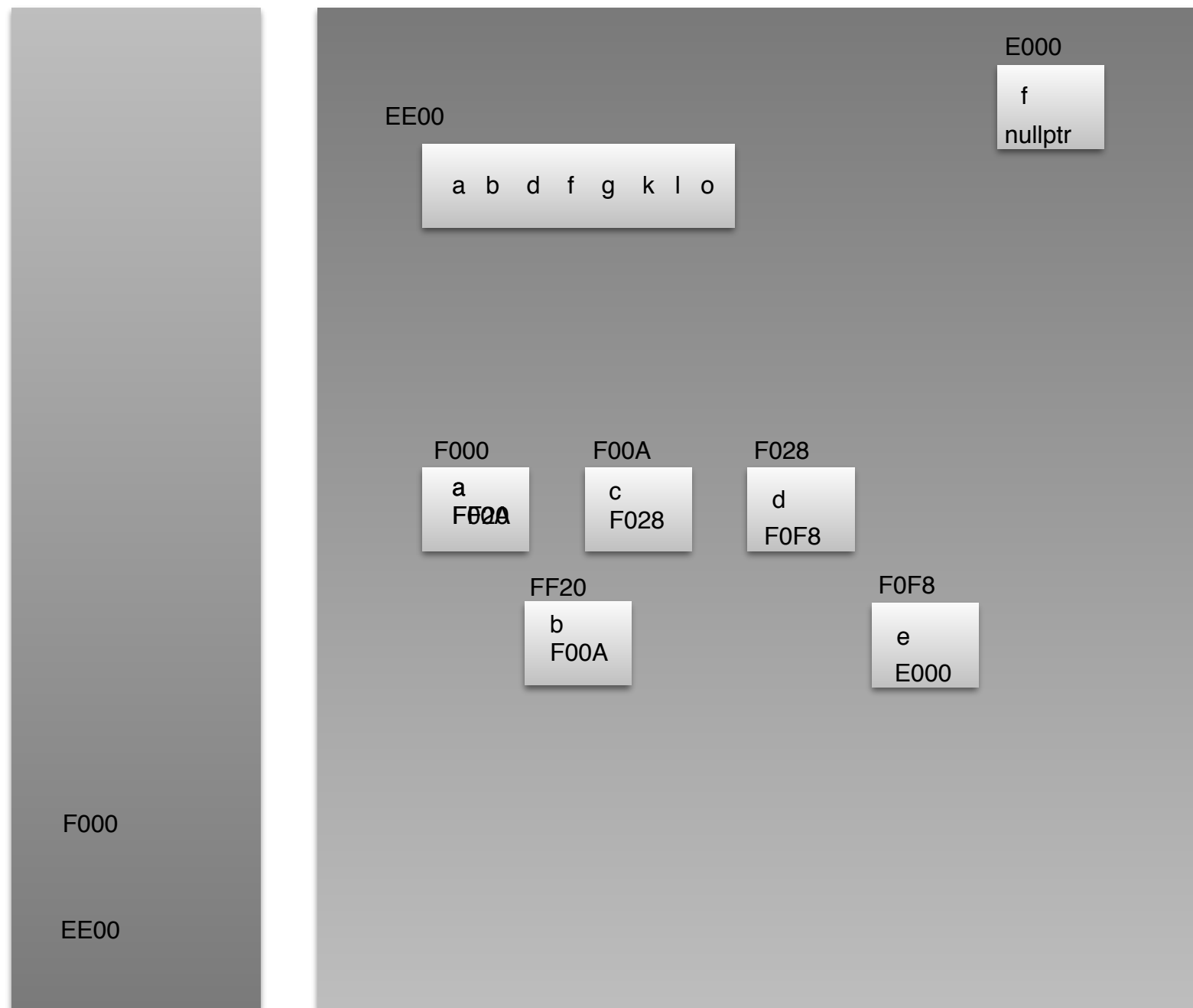


# Lecture 15

## Trees

# Thinking about storing items...

## What could we do differently?



The best time to plant a tree was 20 years ago.  
The next best time is now. ~Chinese Proverb

I willingly confess to so great a partiality  
for trees as tempts me to respect a man in  
exact proportion to his respect for them.  
~James Russell Lowell

# Trees

– Up to now, the data structures we've studied (vectors, stacks queues, lists) have stored sequences of data

- each item has a single successor

– Often, data is organized hierarchically, not sequentially

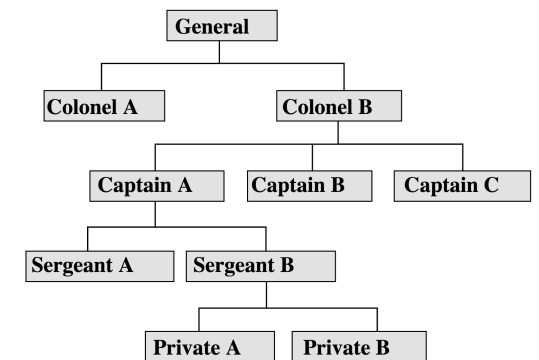
- organization chart for a company
- classification of the animal kingdom
- file system

– Trees can be used to characterize such relationships:

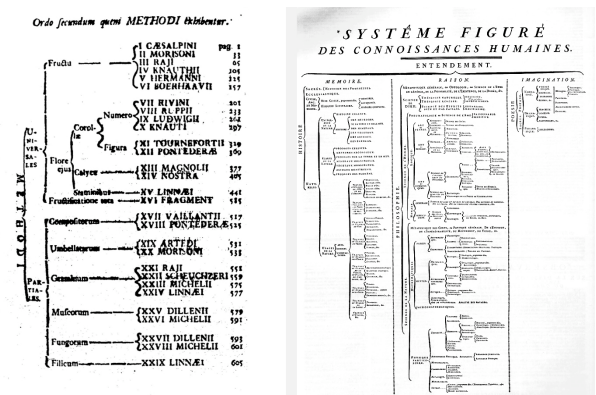
- items may have several successors, called children

– We'll focus on binary trees, in which each node has at most two children

- in addition to applications to hierarchical data, these have applications to searching and sorting.

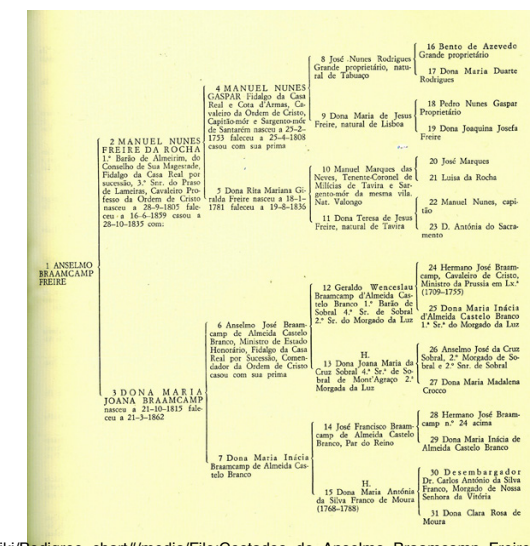


[https://en.wikipedia.org/wiki/Organizational\\_chart#/media/File:Organizational\\_chart.svg](https://en.wikipedia.org/wiki/Organizational_chart#/media/File:Organizational_chart.svg)



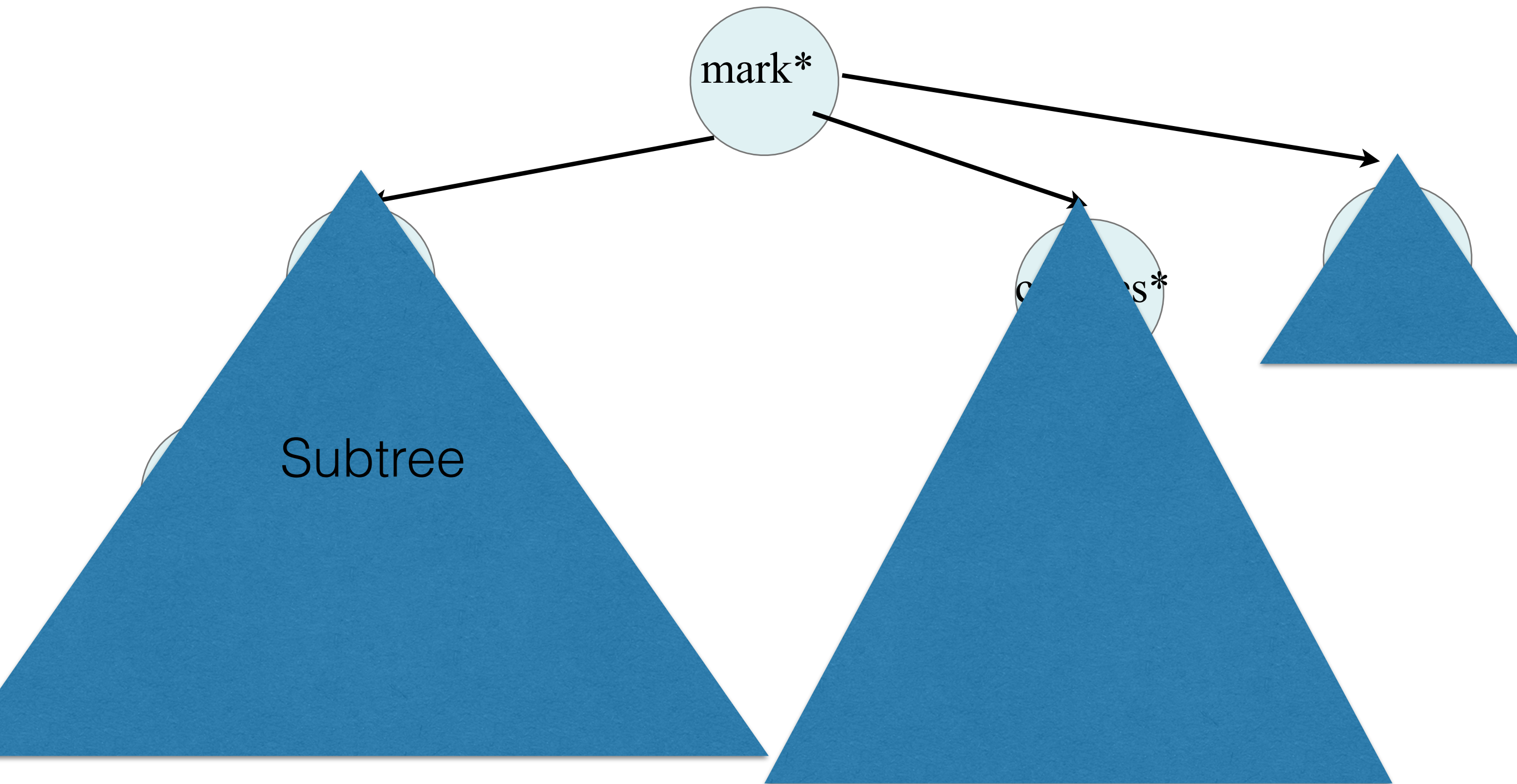
“Linné's method for classification of plants in Classes Plantarum 1738, and the Figurative system of human knowledge from Diderot's Encyclopédie , 1752.”

[https://upload.wikimedia.org/wikipedia/commons/7/70/Taxonomy\\_Linn%C3%A9\\_%26\\_Diderot.jpg](https://upload.wikimedia.org/wikipedia/commons/7/70/Taxonomy_Linn%C3%A9_%26_Diderot.jpg)



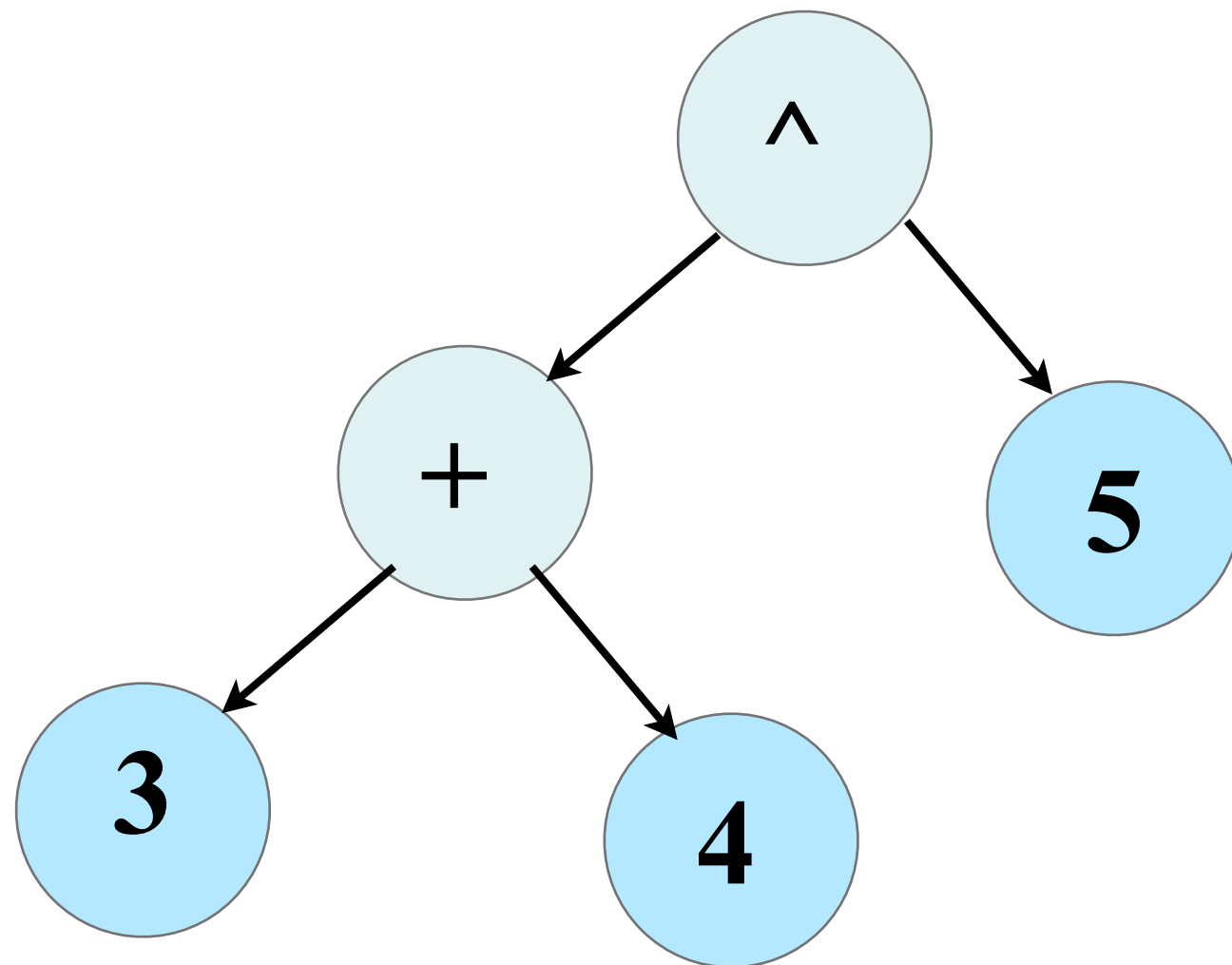
[https://en.wikipedia.org/wiki/Pedigree\\_chart#/media/File:Costados\\_de\\_Anselmo\\_Braamcamp\\_Freire.jpg](https://en.wikipedia.org/wiki/Pedigree_chart#/media/File:Costados_de_Anselmo_Braamcamp_Freire.jpg)

# Unix Directory Tree



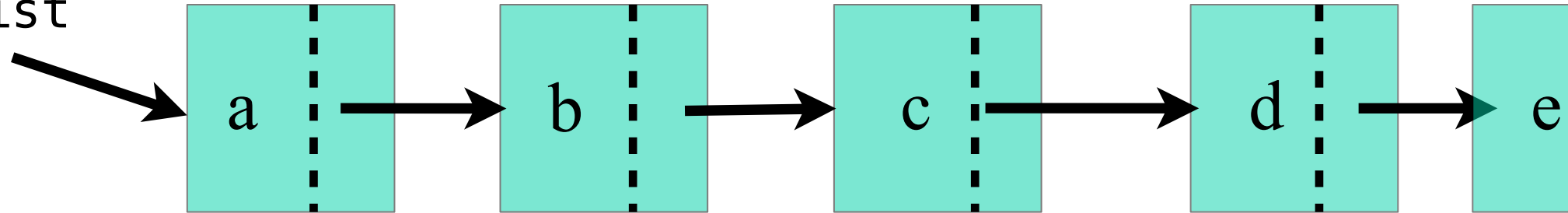
# Expression Tree

$$(3 + 4)^5$$

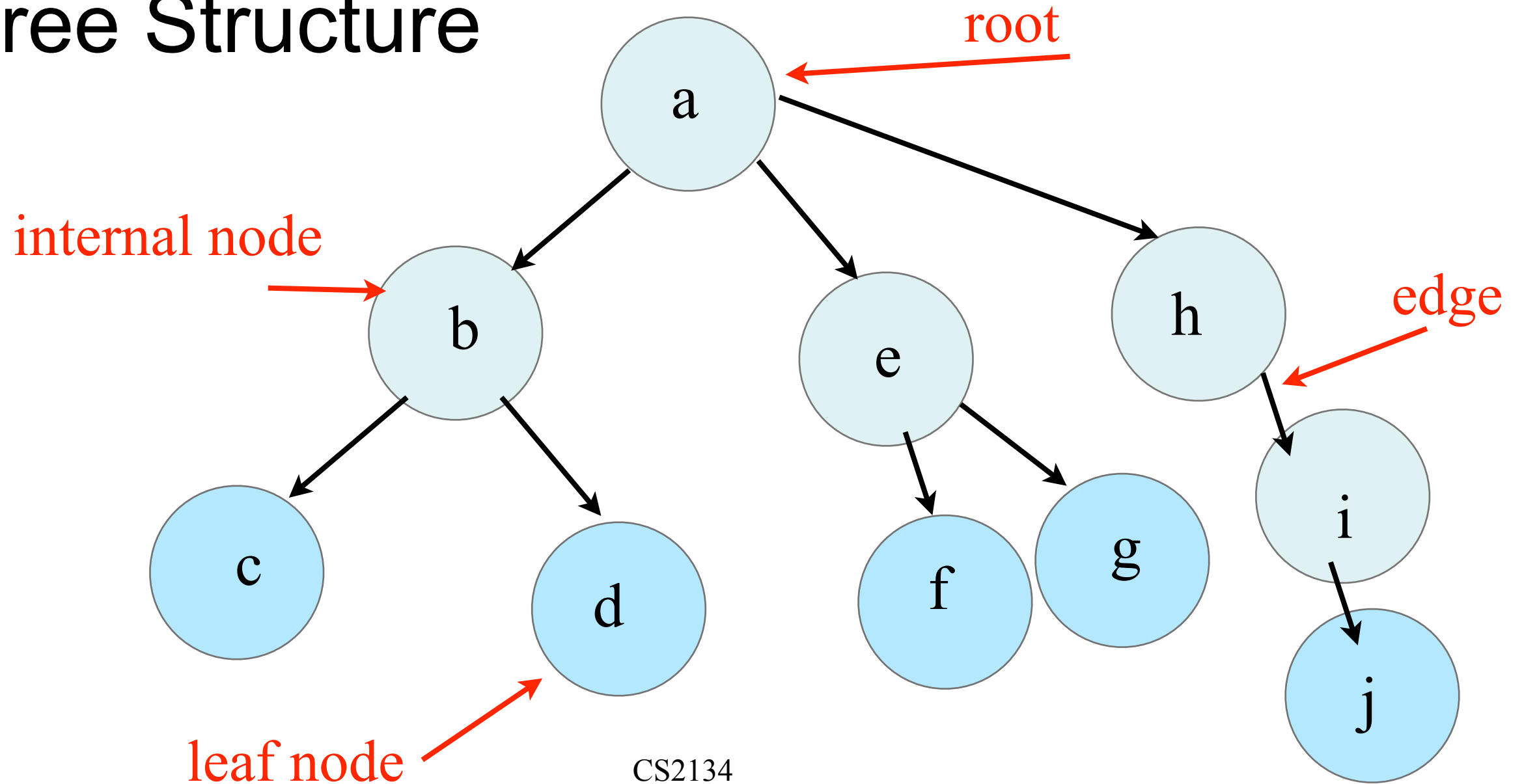


# List Structure

frontOfList



# Tree Structure



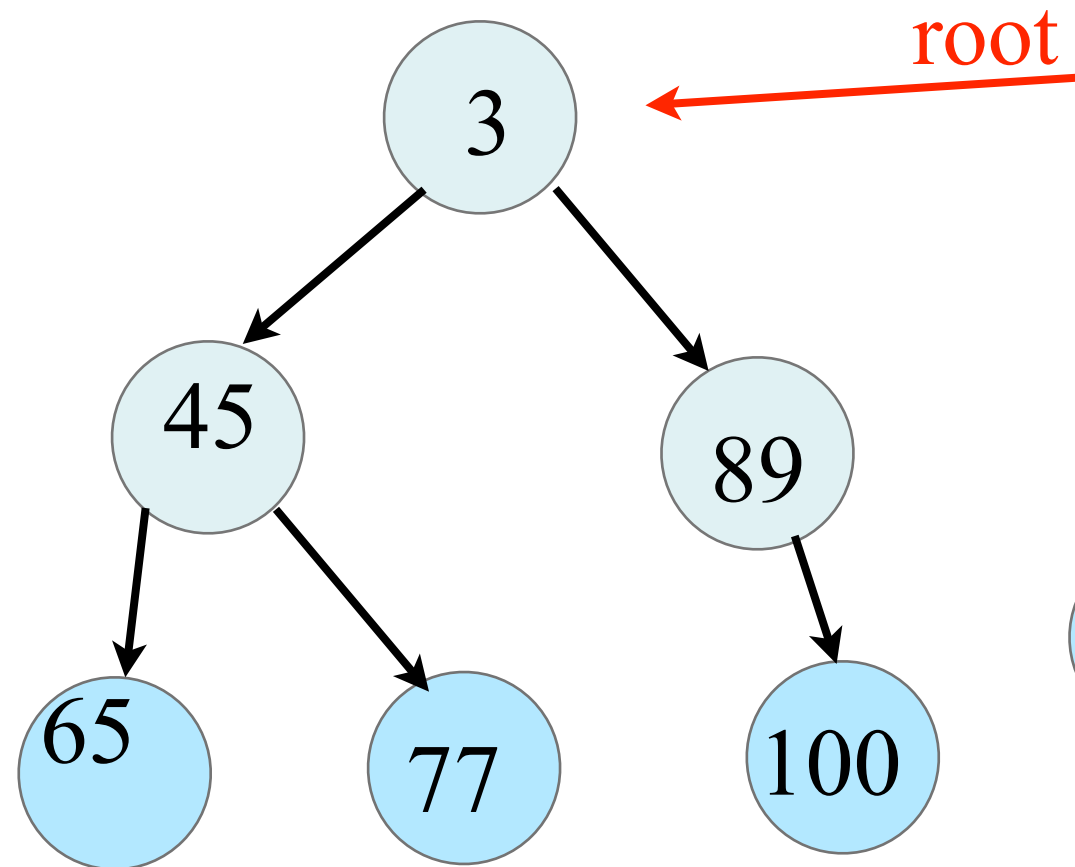
CS2134



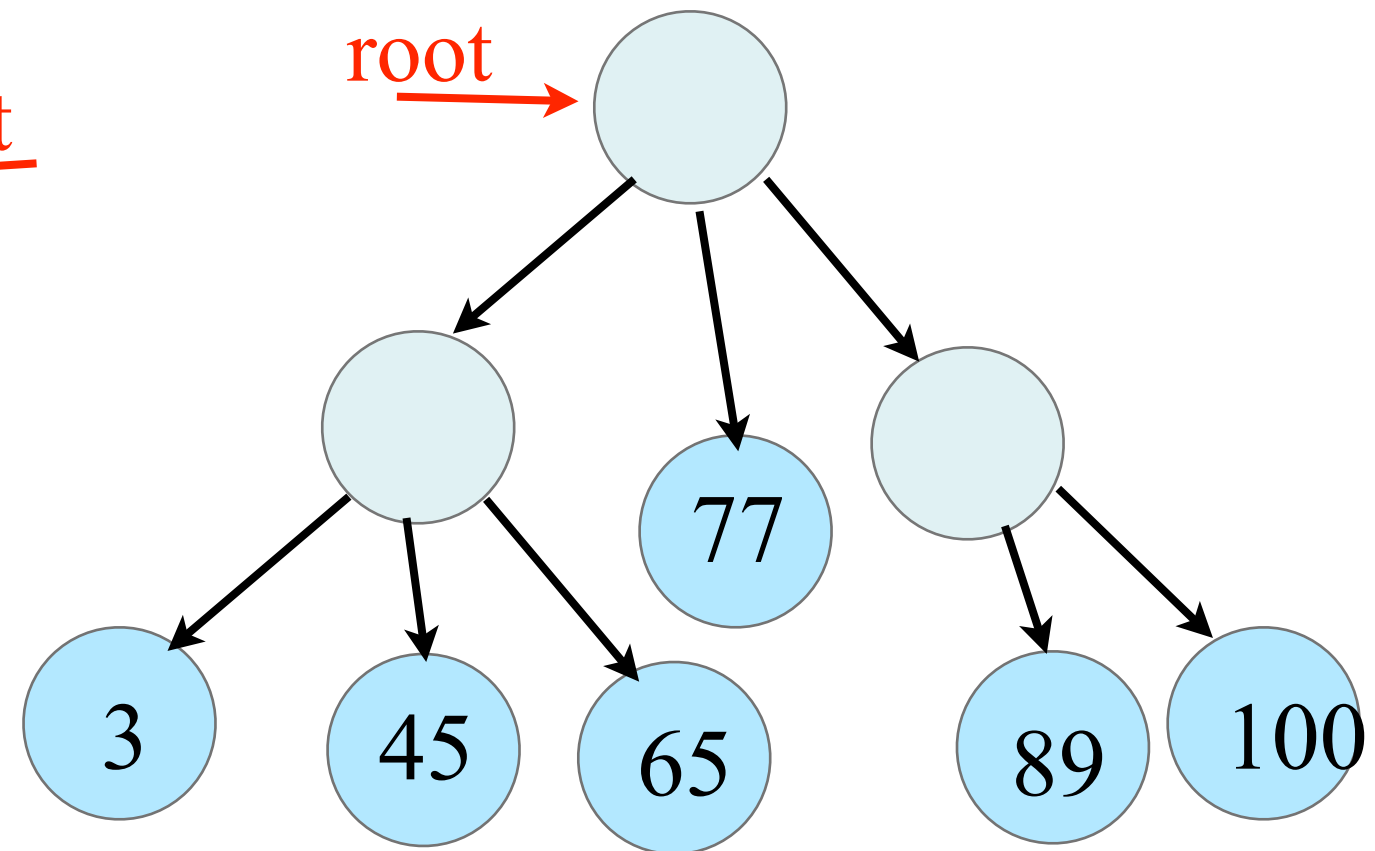
*Family Tree, Biological classifications, Organization chart*

- parent
- child
- sibling
- ancestor
- descendent
- root
- leaf
- internal node
- size
  - of node
  - of tree
- height

# Storing #'s 3, 45, 65, 77, 89, 100



Information is stored in every node



Information is stored only in the leaf nodes

# Definitions:

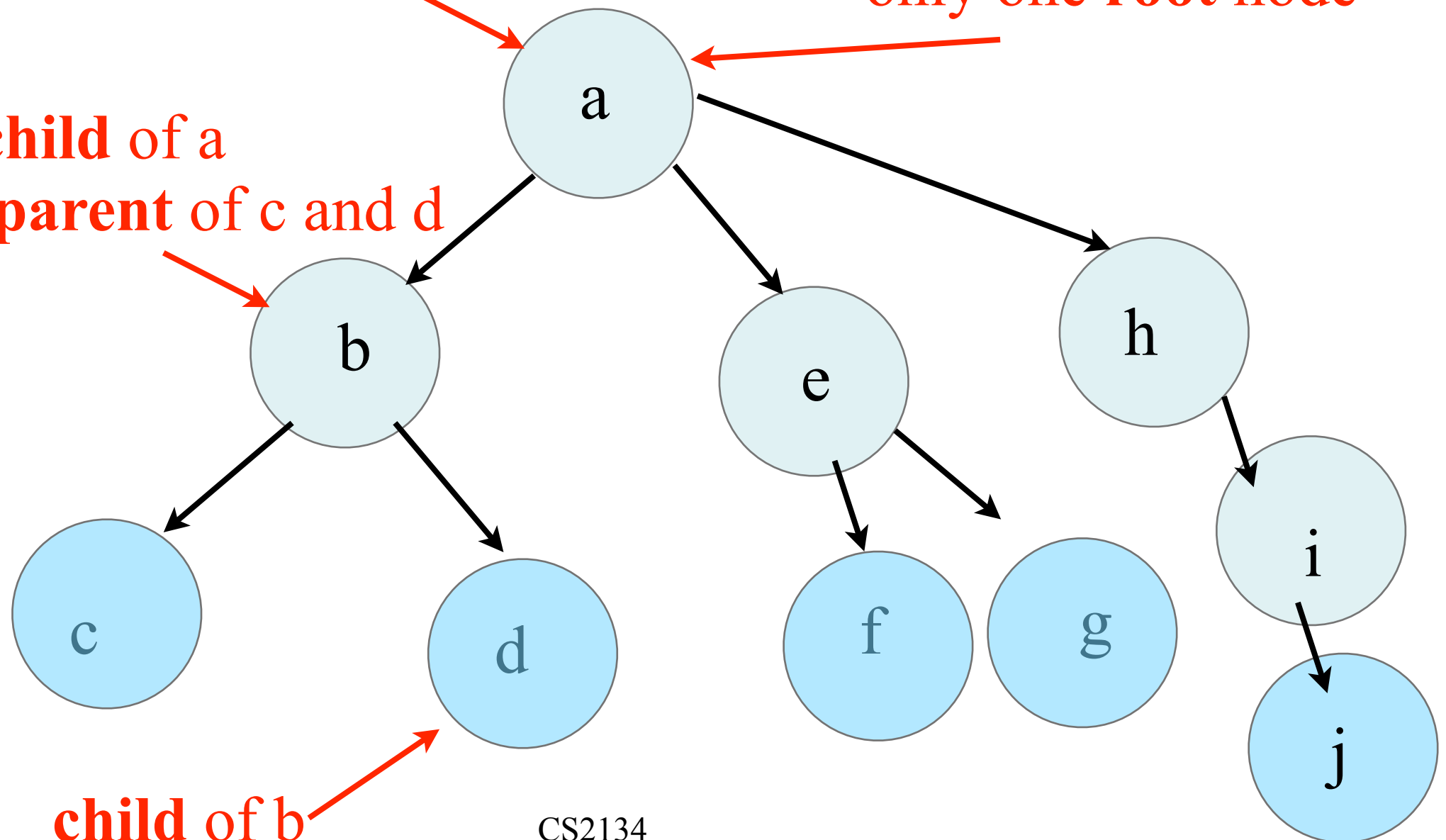
Node a is the **parent** of nodes b, e and h

only one **root** node

b is the **child** of a

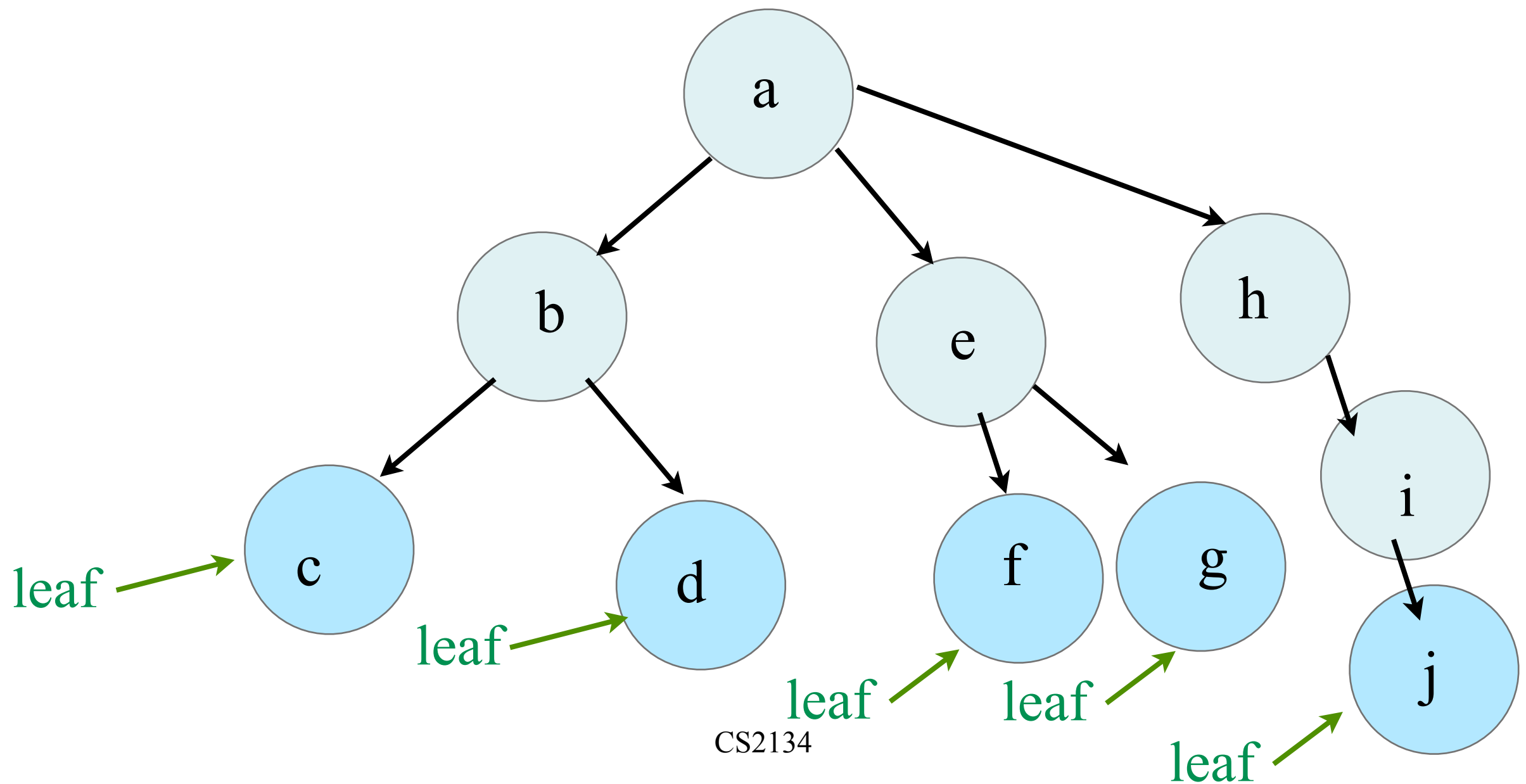
b is the **parent** of c and d

**child** of b



CS2134

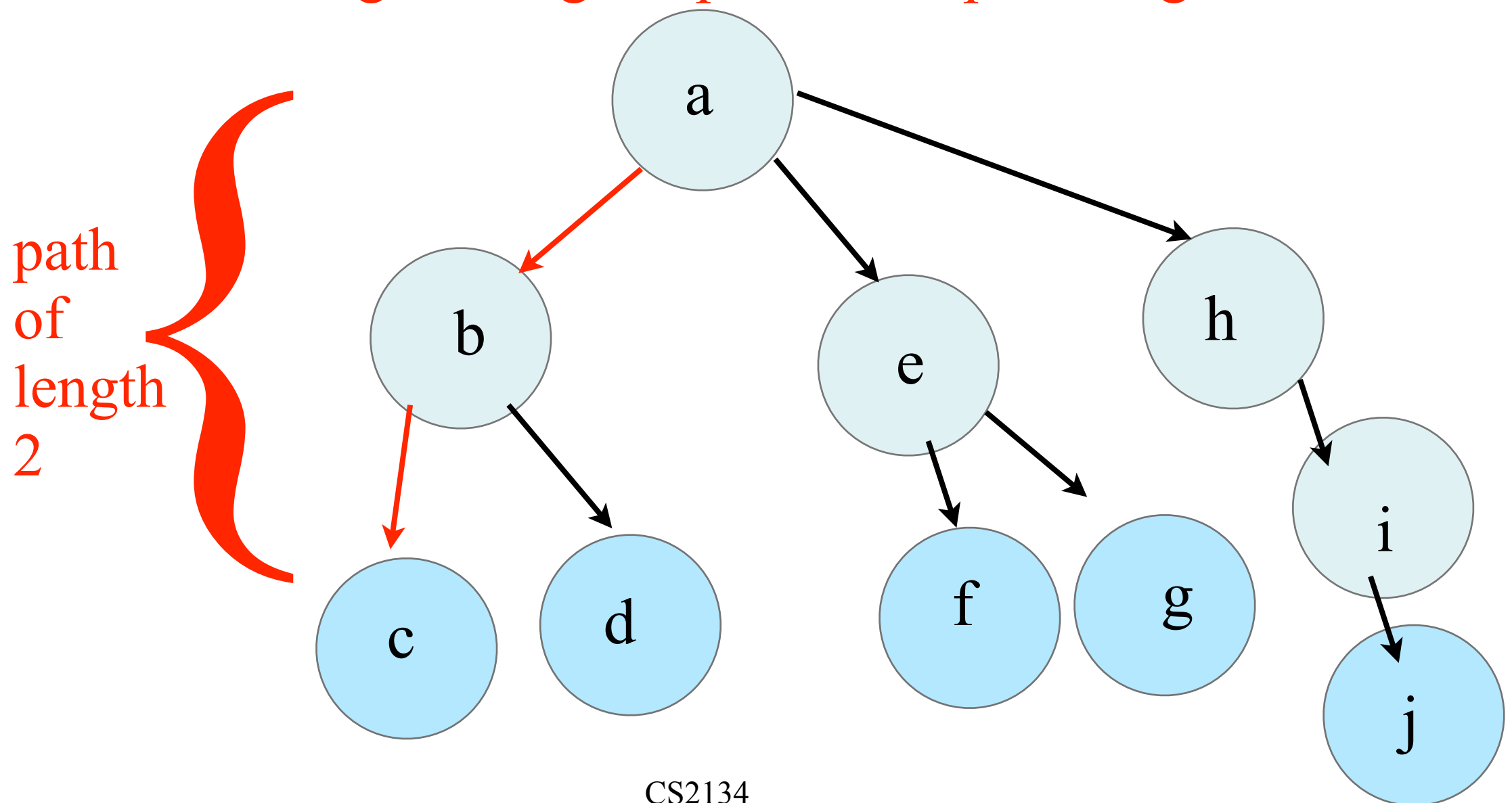
# What is the definition of a leaf?



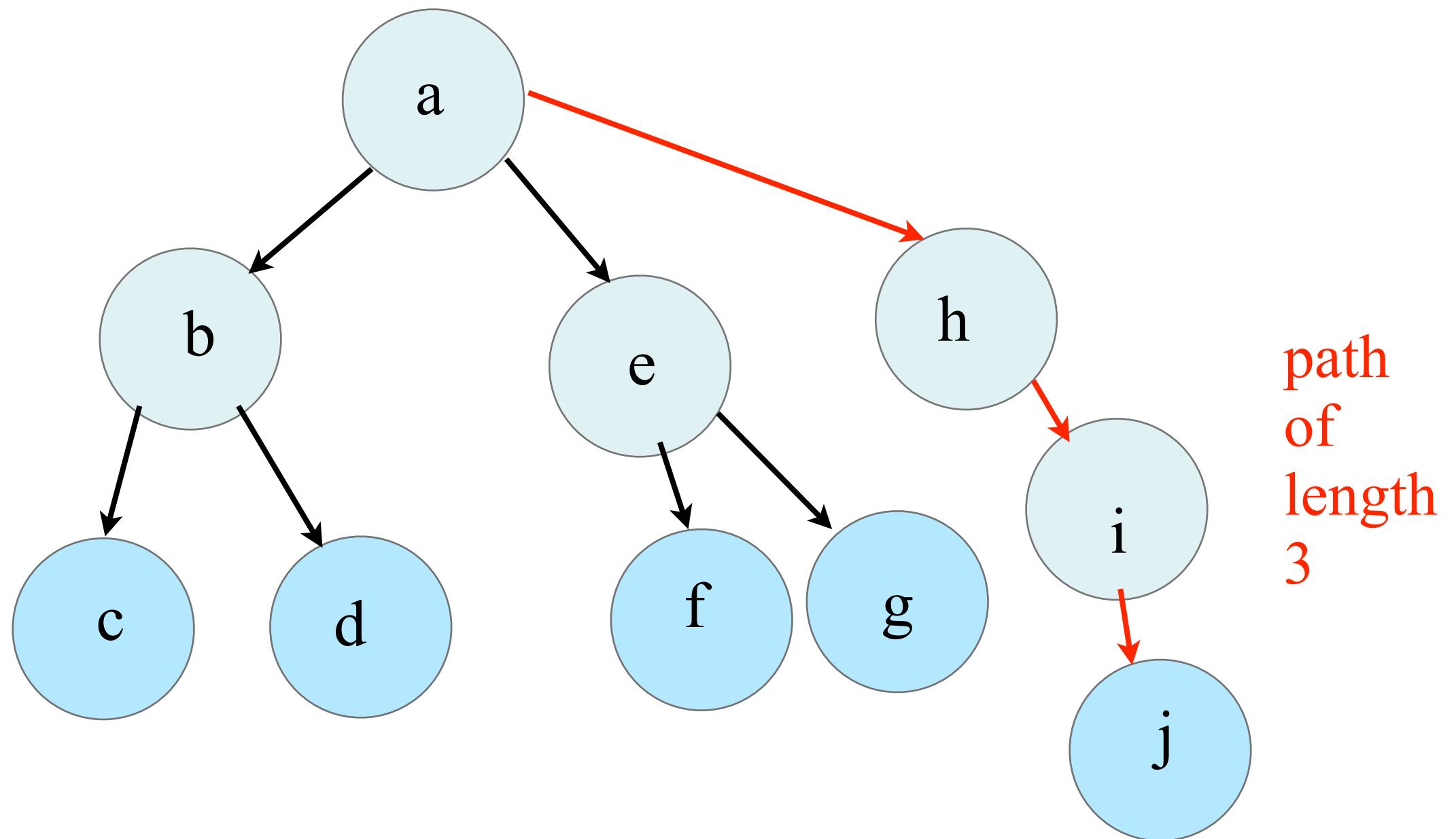
# What is the definition of the length of a path between two nodes?

There is one unique path from the root to any node in the tree.

The number of edges along the path is the path length.

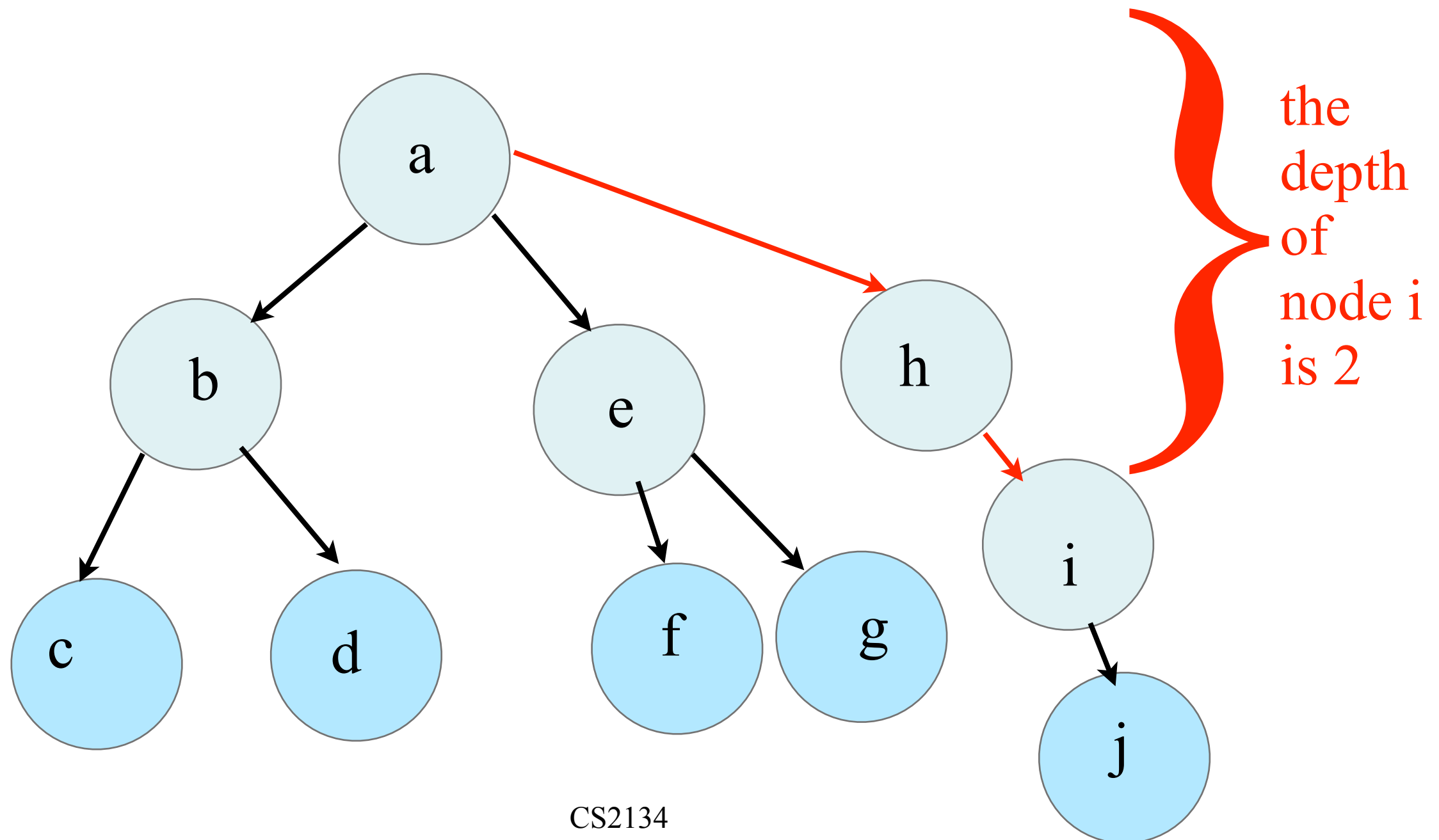


# What is the path length from a to j?



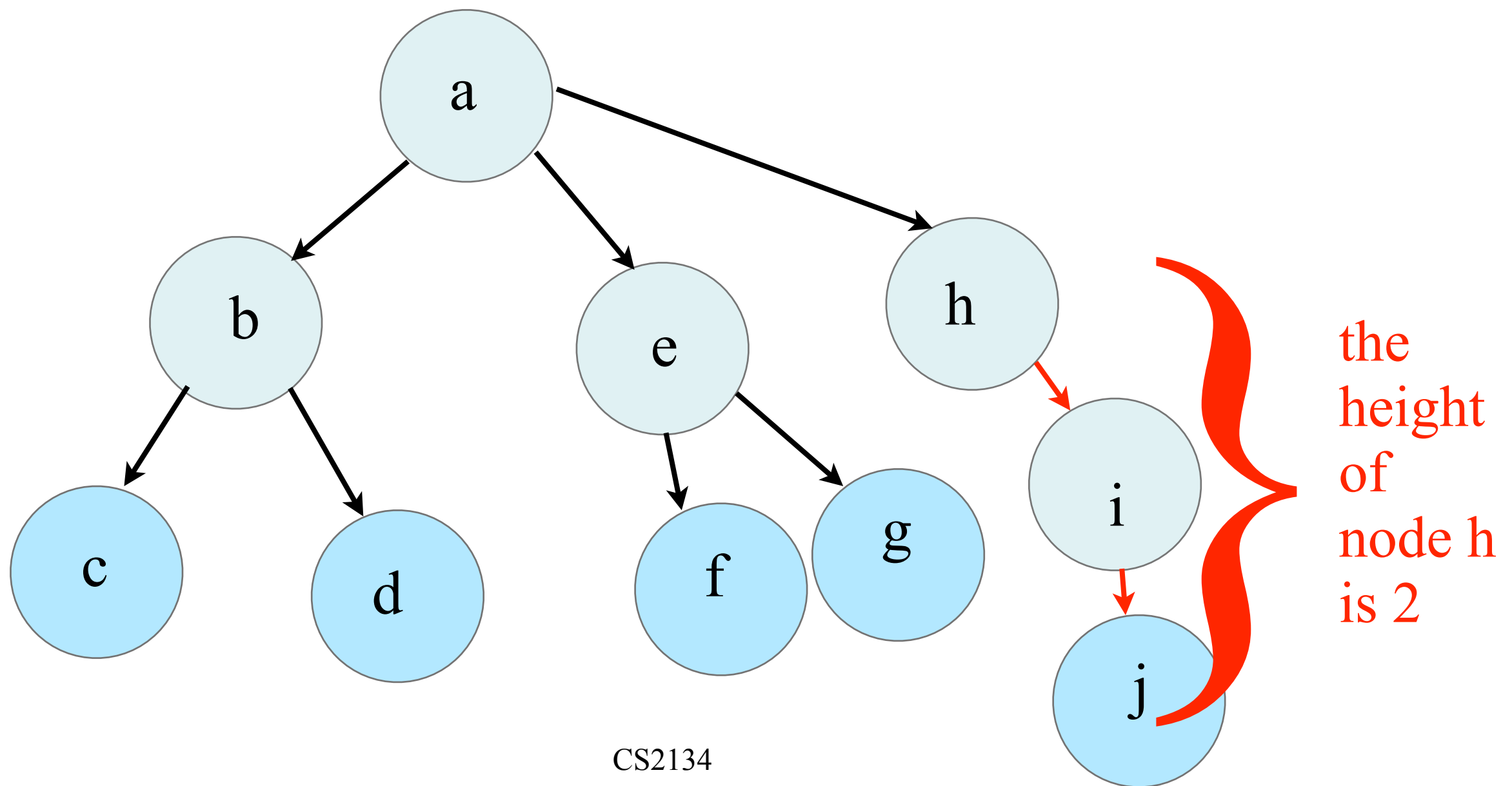
# What is the definition of the depth of a node?

The *depth* of a node is the number of edges from the root to the node



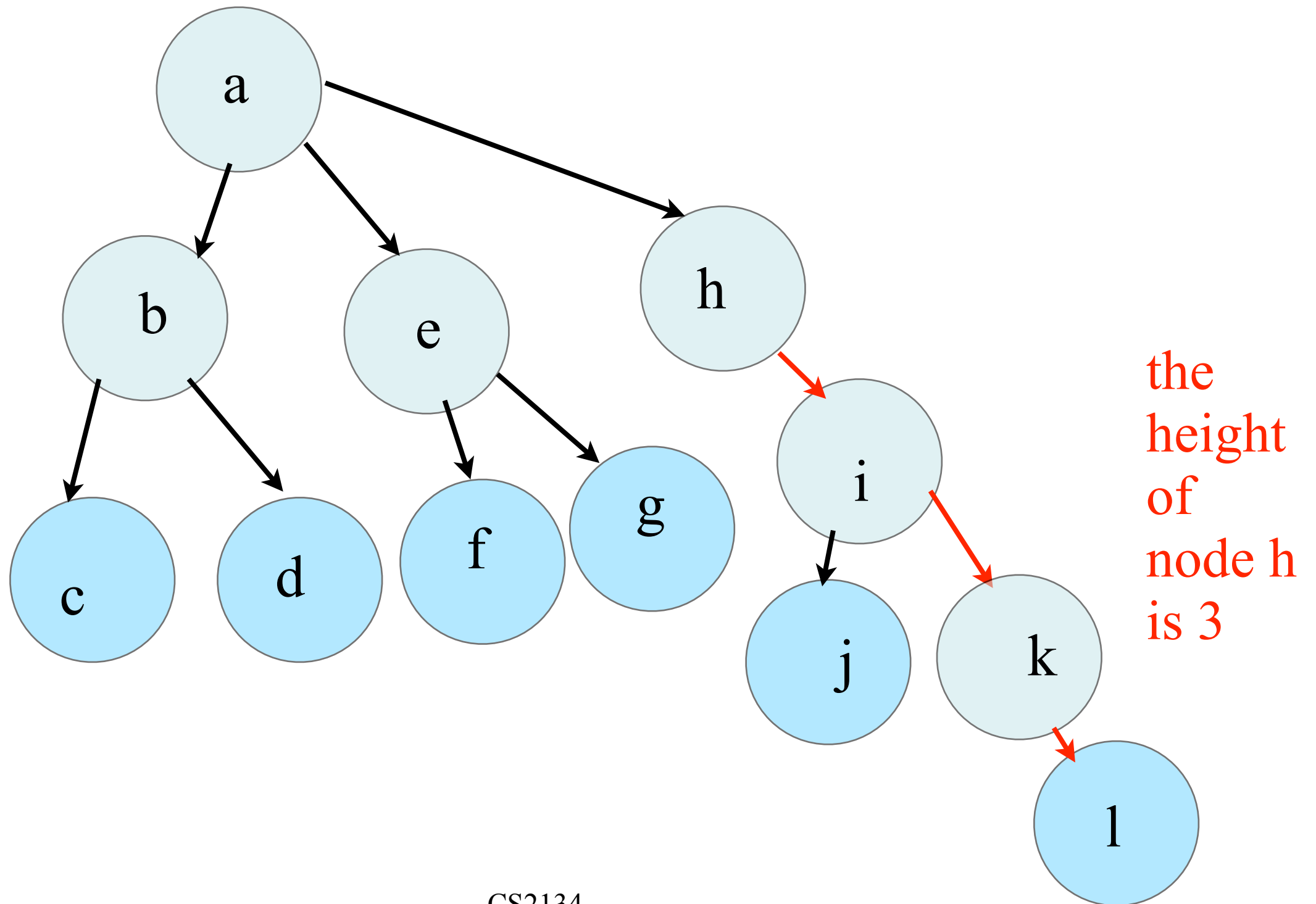
# What is the definition of the height of a node?

The *height* of a node is the number of edges from the node to the deepest leaf.

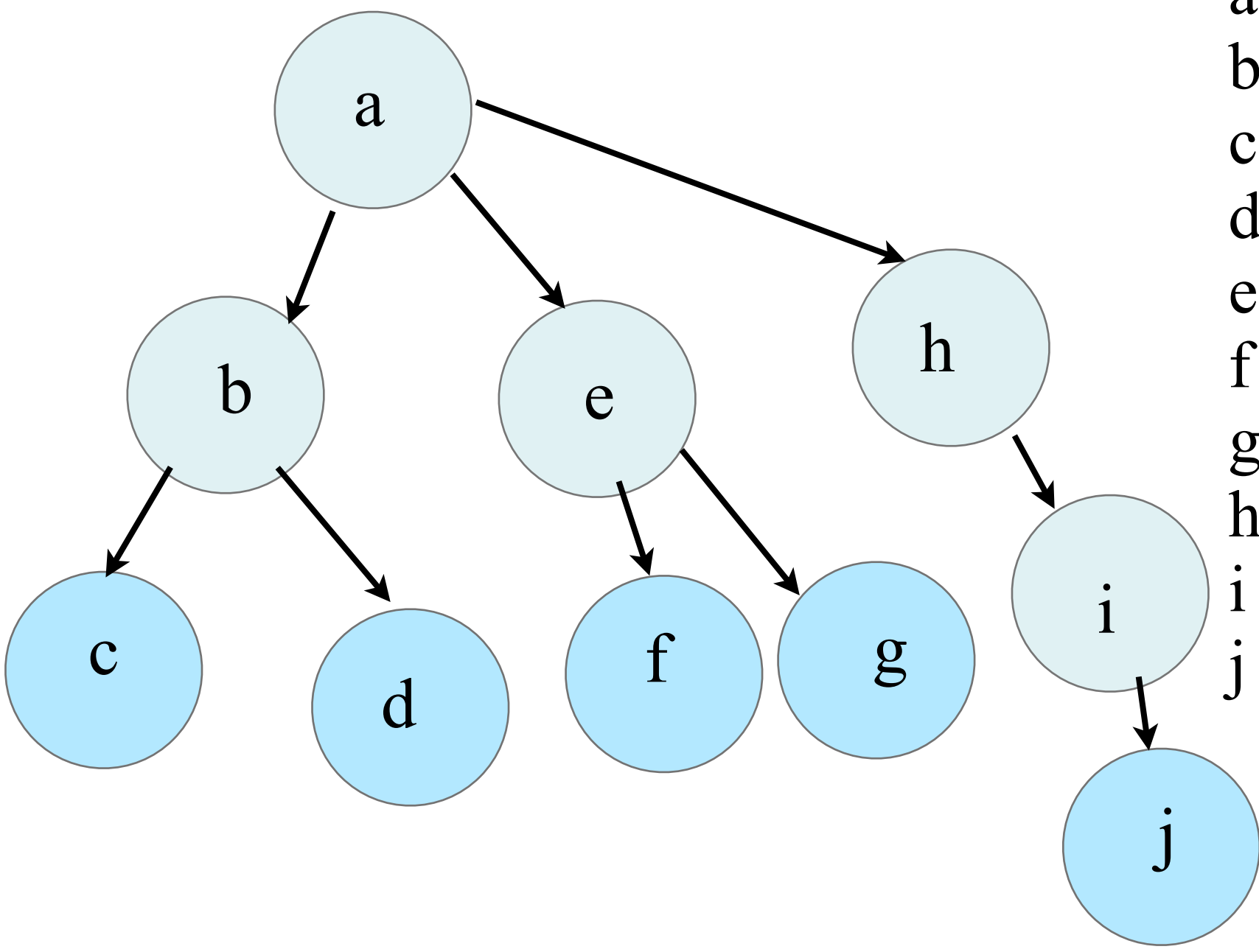




# What is the height of node h?



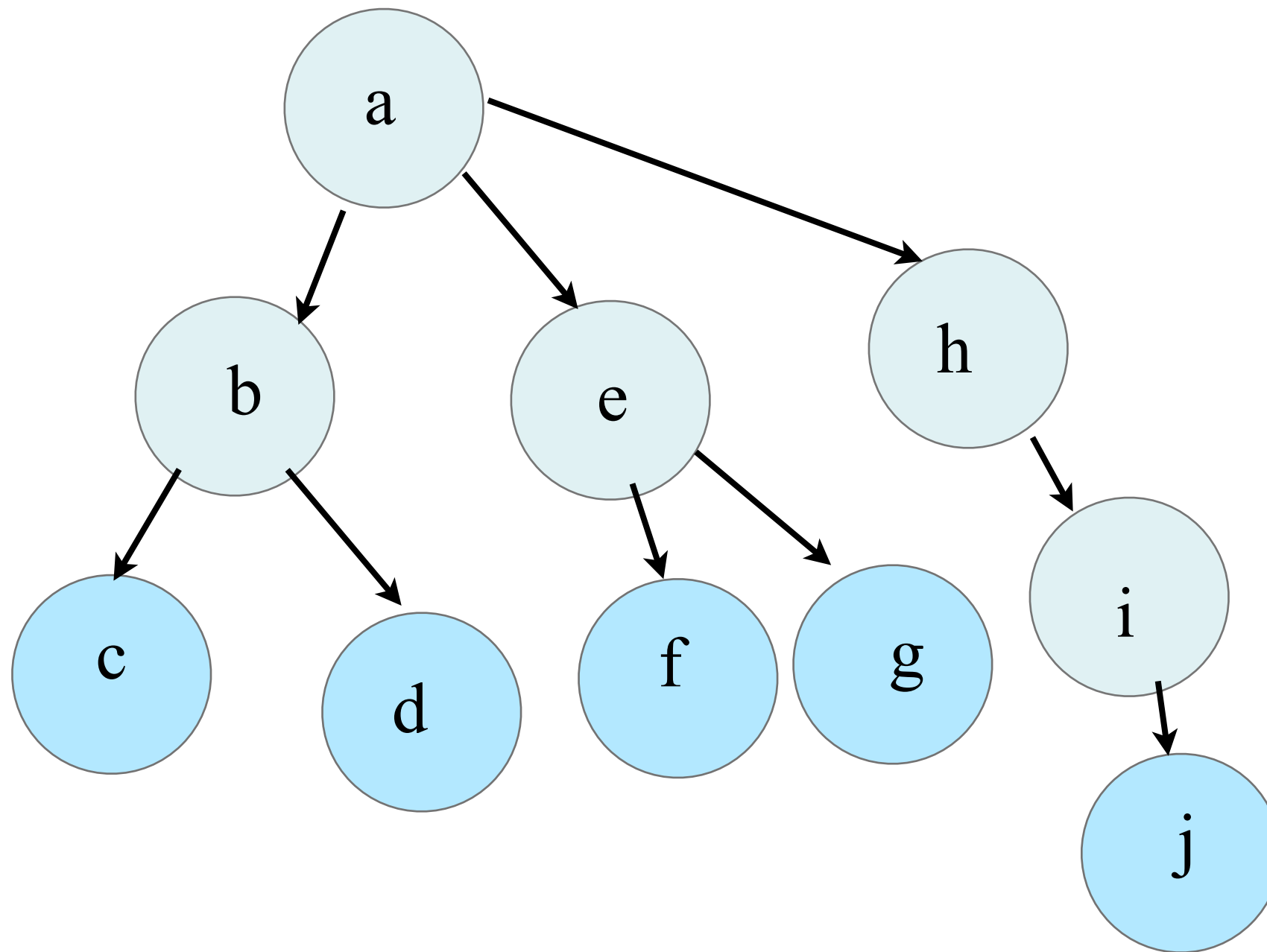
What are the height and depth of nodes a, b and c?



Node	Height	Depth
a	3	0
b	1	1
c	0	2
d	0	2
e	1	1
f	0	2
g	0	2
h	2	1
i	1	2
j	0	3

# What is the definition of the size of a node?

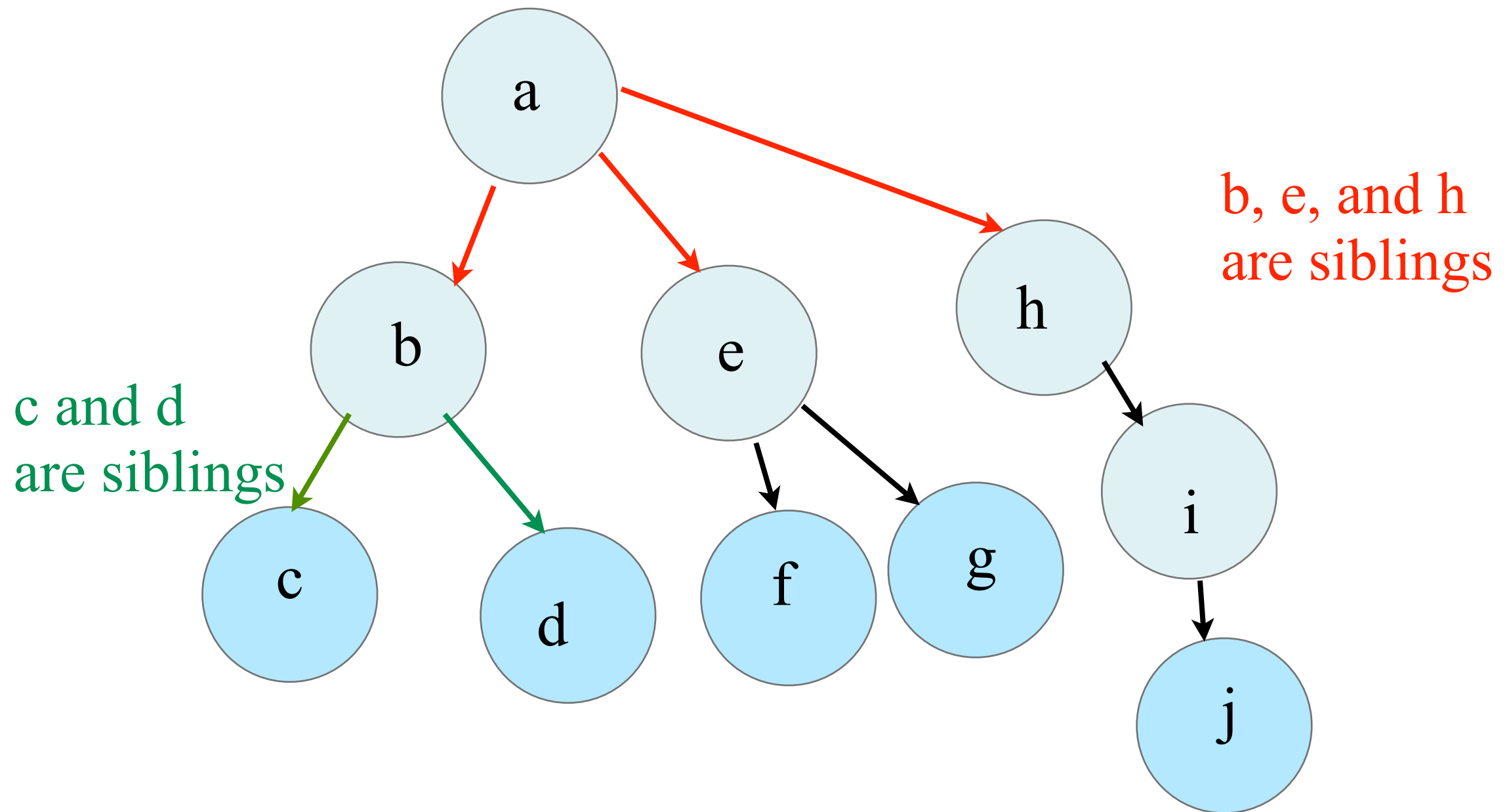
The *size* of a node is the number of descendants the node has (including the node itself.)



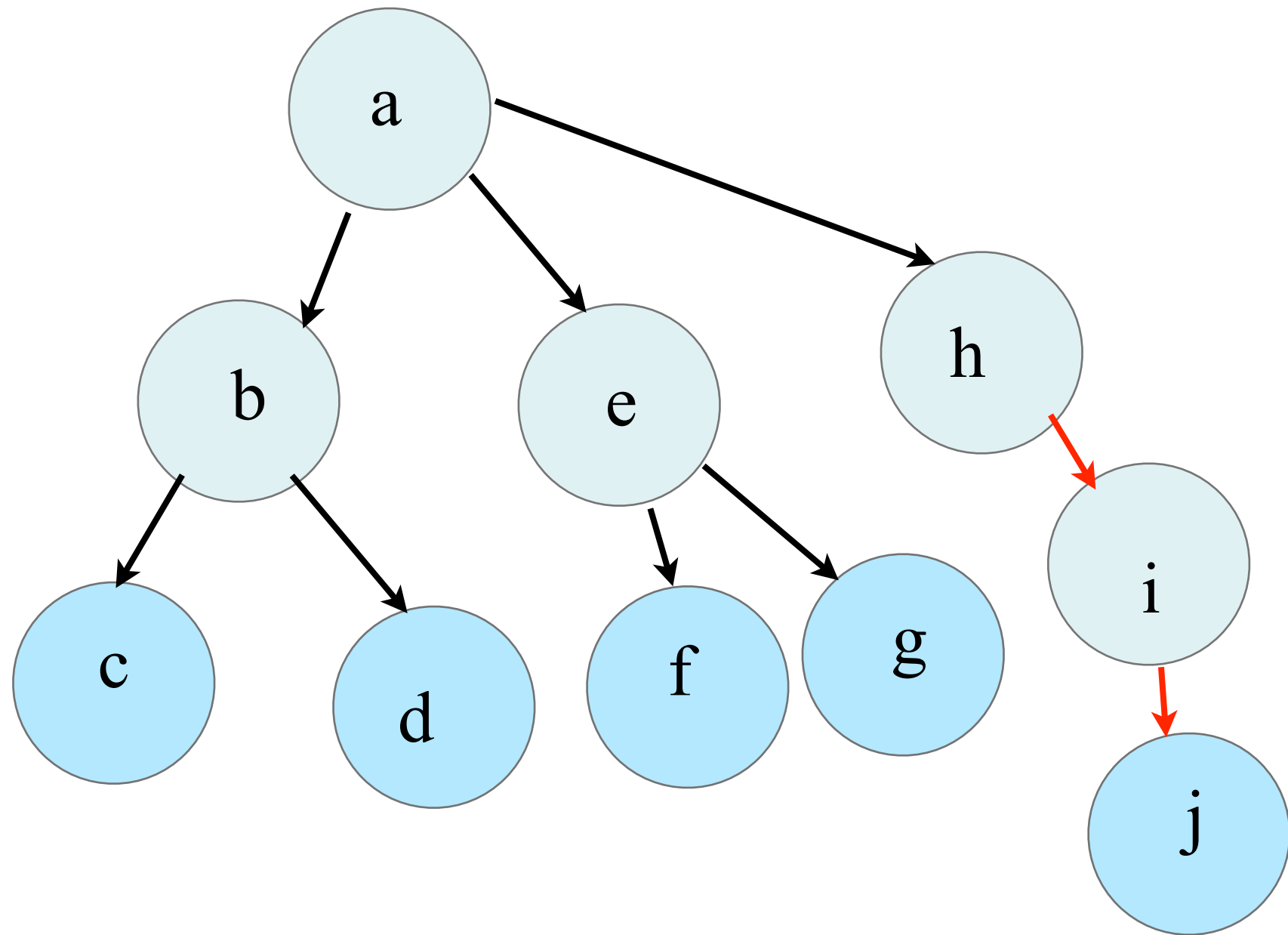
Node	Size
a	10
b	3
c	1
d	1
e	3
f	1
g	1
h	3
i	2
j	1

The *size* of a tree is the size of the root node.

Nodes with the same parent are siblings



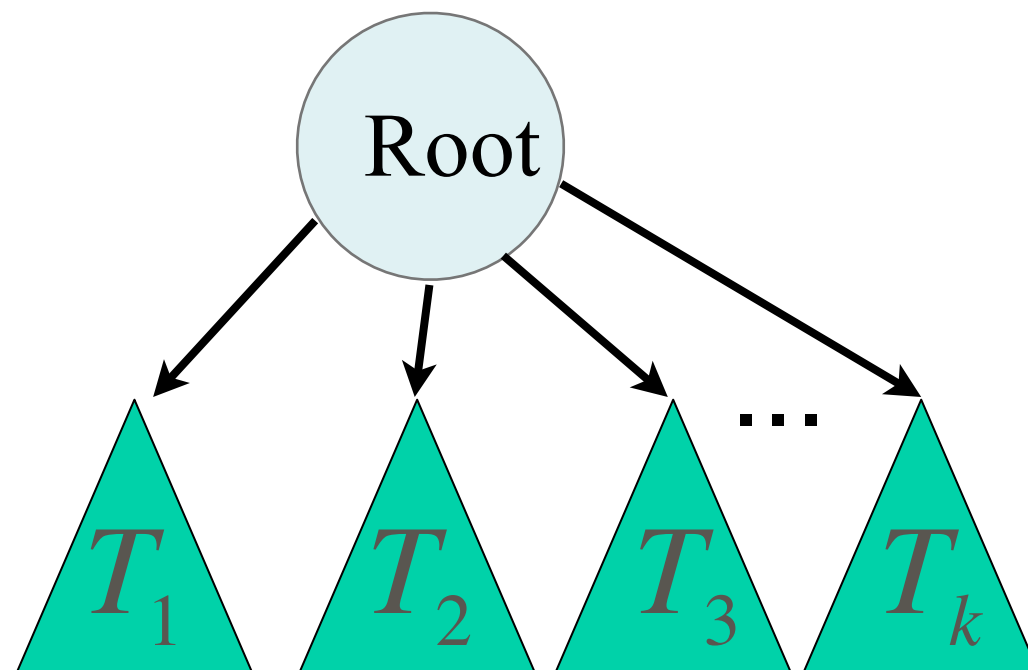
If there is a path from node h to node j, then  
h is the *ancestor* of node j, and j is the *descendent* of node h



# Recursive Definition of a Tree

- A tree is empty
- **or** it consists of a root and zero or more nonempty subtrees  $T_1, T_2, \dots, T_k$ .

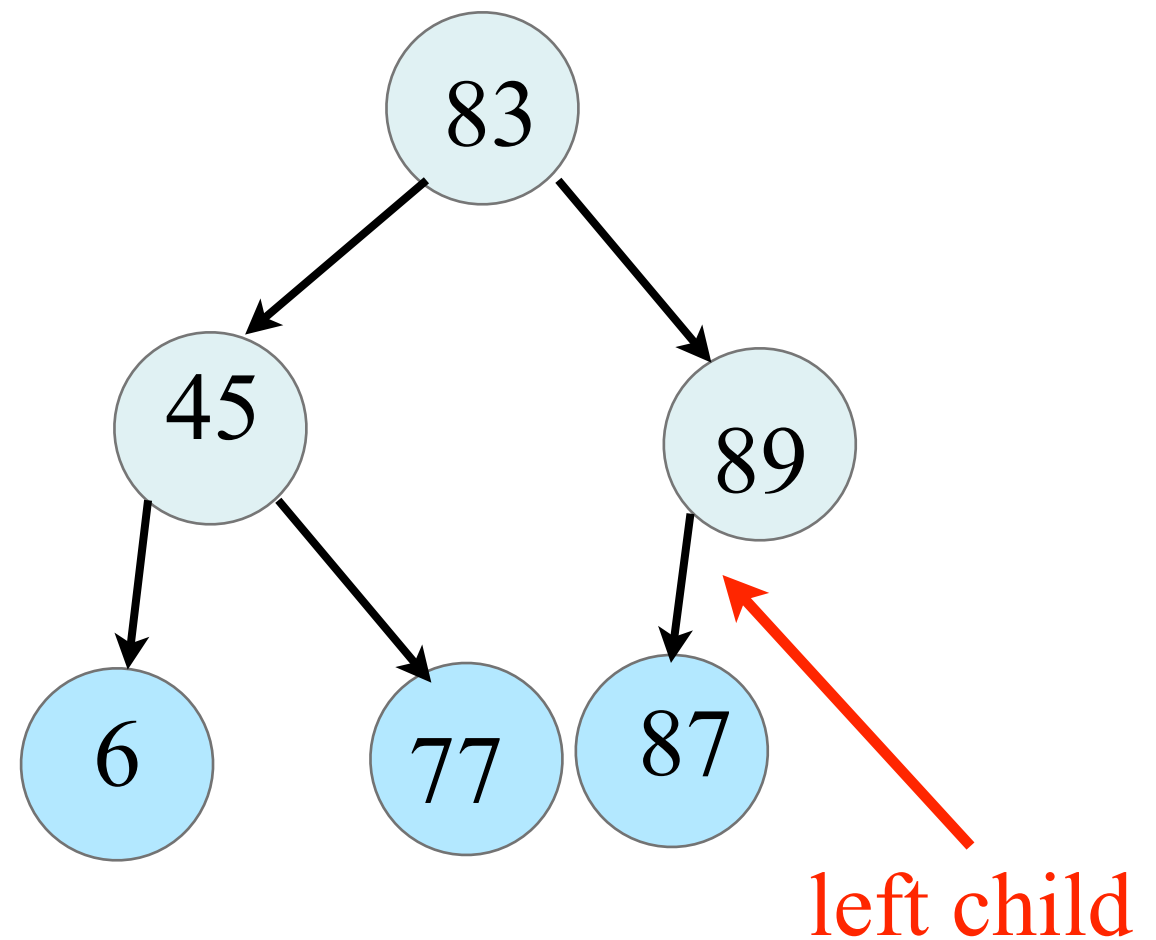
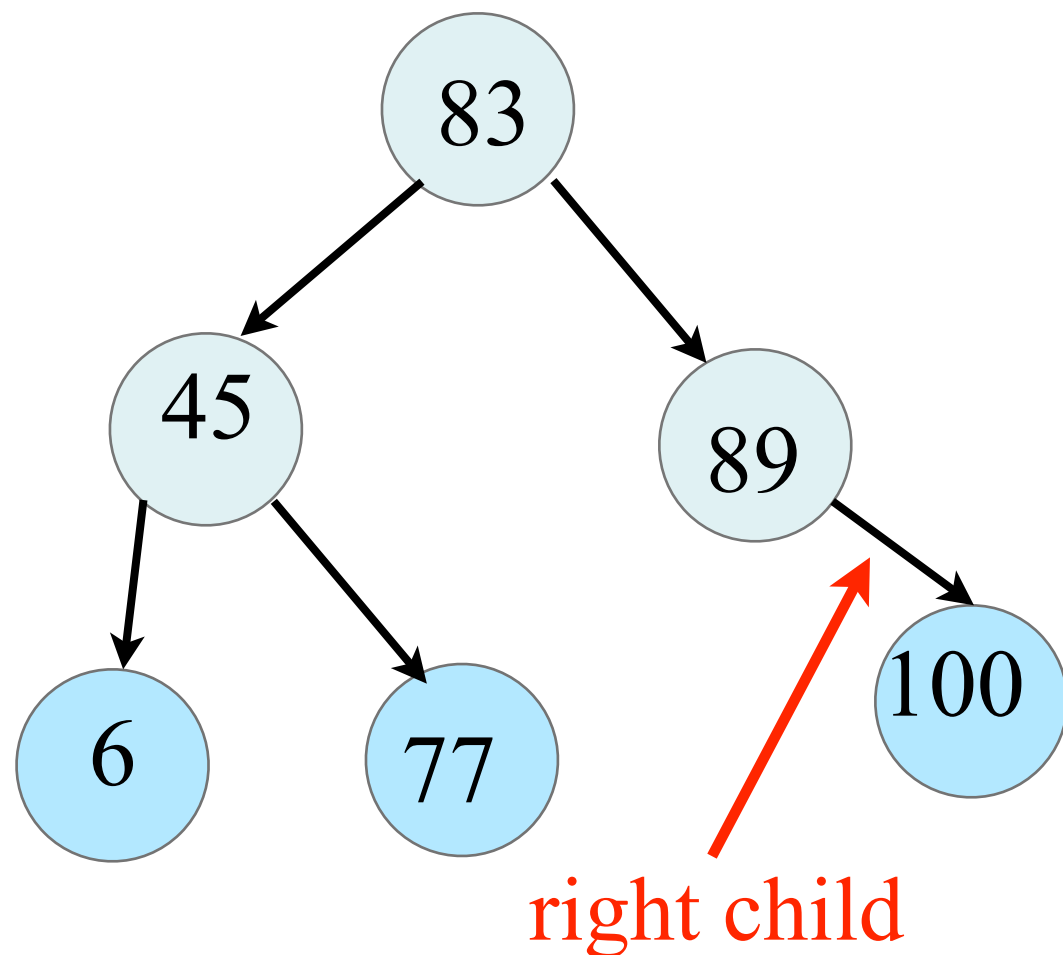
Each subtree is connected by an edge from the root.



If a tree has  $n$  nodes how many edges does it have?  $n-1$

# Binary Trees

- We will focus on binary trees.
- 0, 1 or 2 children per node.
- Sometimes an “order” is imposed on the tree in the tree below. Can you see what it is?



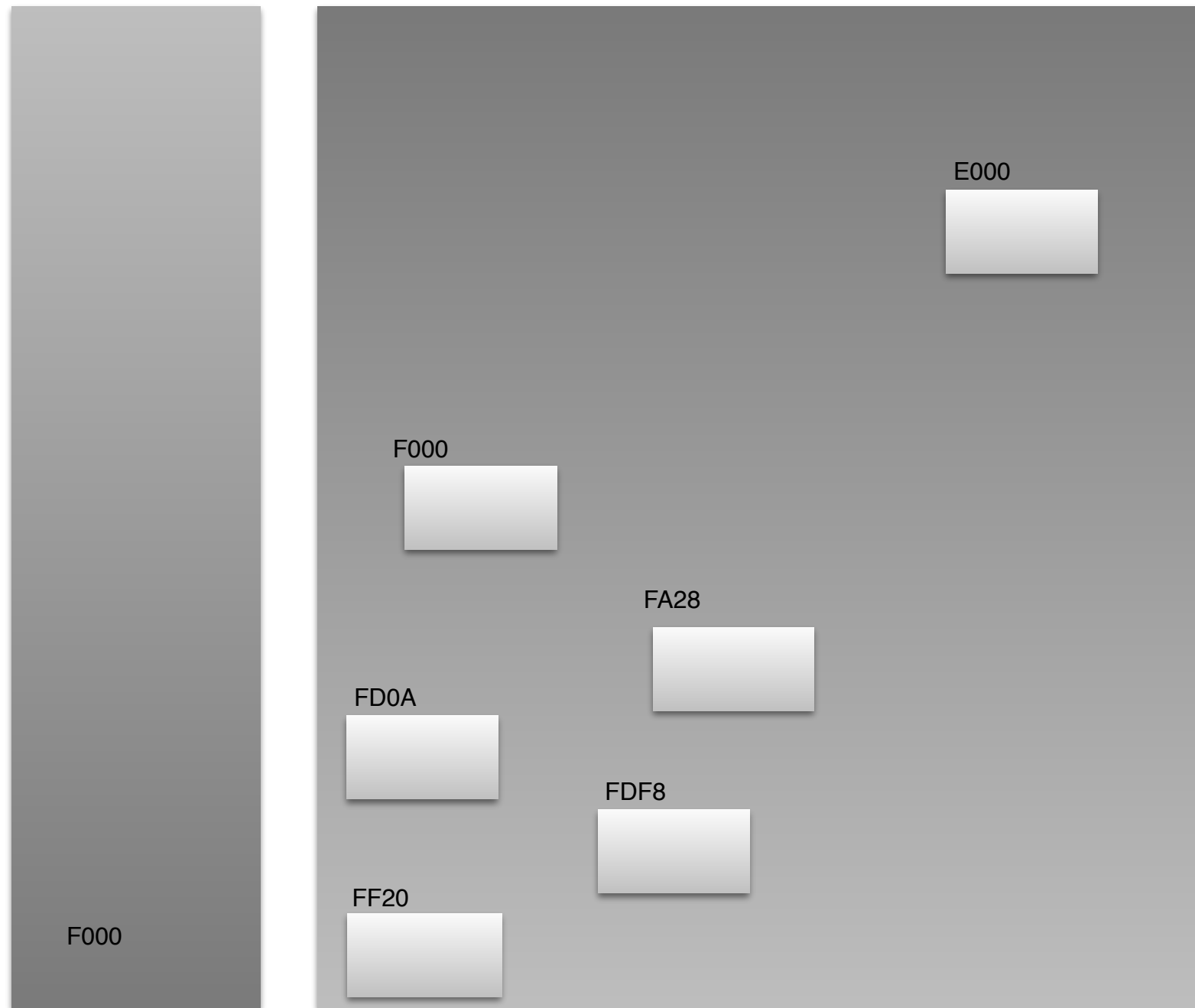


# Binary tree Definition

- Def 1:
  - **empty**, or
  - unique root node, and
  - each node has at most two children, the left child and the right child, and
  - for each node there a unique path from the root “down” to the node.
- Def 2:
  - **empty**, or
  - a **node**, or
  - a node along with two trees, the right subtree and the left subtree

# Thinking about storing items as a tree

## How could we implement this?



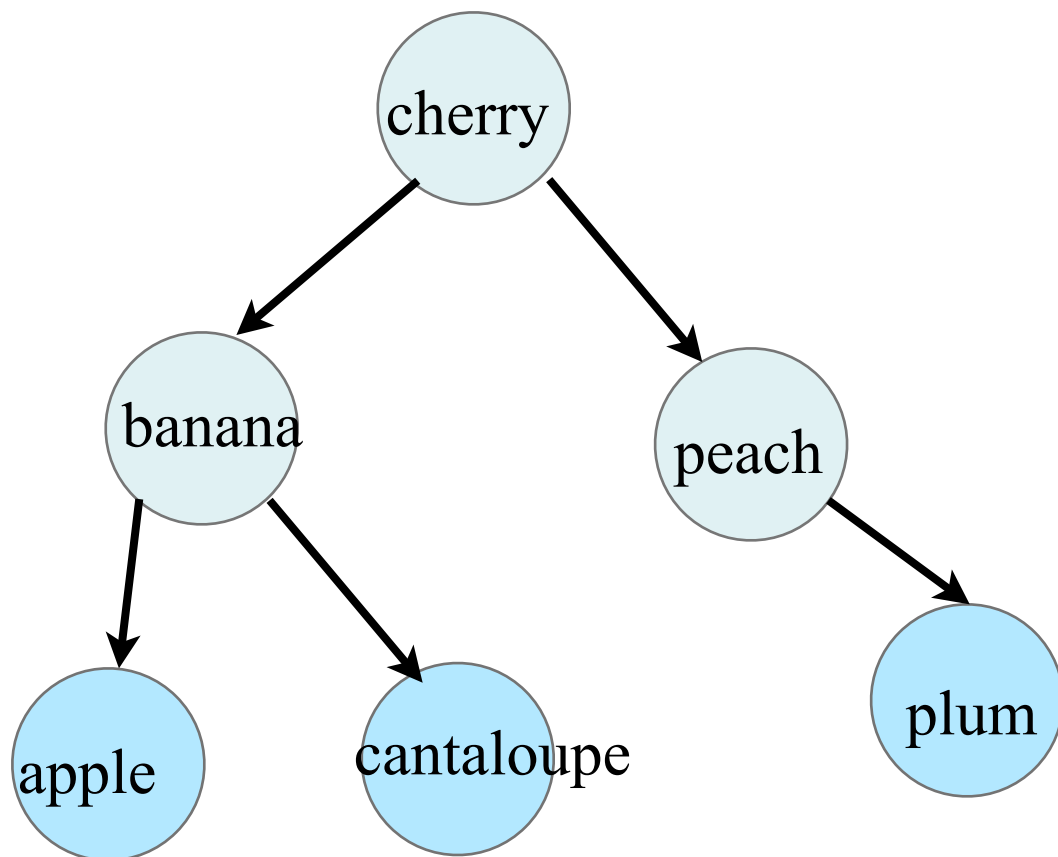
# Implementing a node

```
template <class Object>
class BinaryNode {

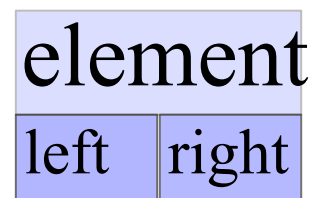
public:
    BinaryNode( const Object & theElement = Object( ),
               BinaryNode *lt = nullptr, BinaryNode *rt = nullptr );
    BinaryNode( Object && theElement, BinaryNode *lt = nullptr, BinaryNode *rt = nullptr );

    // Some Methods

public: // To keep things simple
    Object element;
    BinaryNode *left;
    BinaryNode *right;
};
```



A node in the tree



```

template <class Object>
class BinaryNode {

public:
    BinaryNode( const Object & theElement = Object( ), BinaryNode *lt = nullptr,
               BinaryNode *rt = nullptr );

    BinaryNode( Object && theElement, BinaryNode *lt = nullptr, BinaryNode *rt = nullptr );

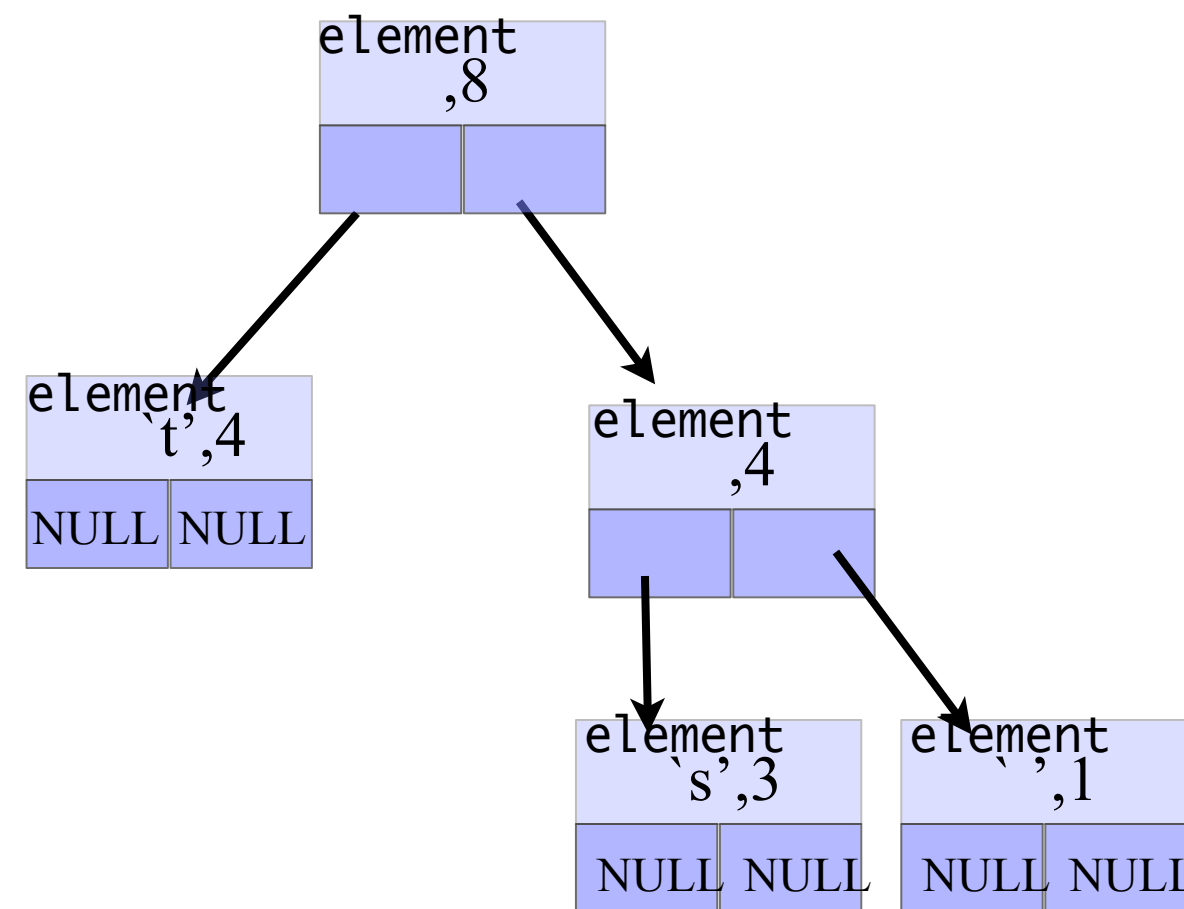
    // Some Methods

public: // To keep things simple
    Object element;
    BinaryNode *left;
    BinaryNode *right;
};

```

- pointer structure for tree
- dynamic growing and shrinking
- resides in heap

## Tree example

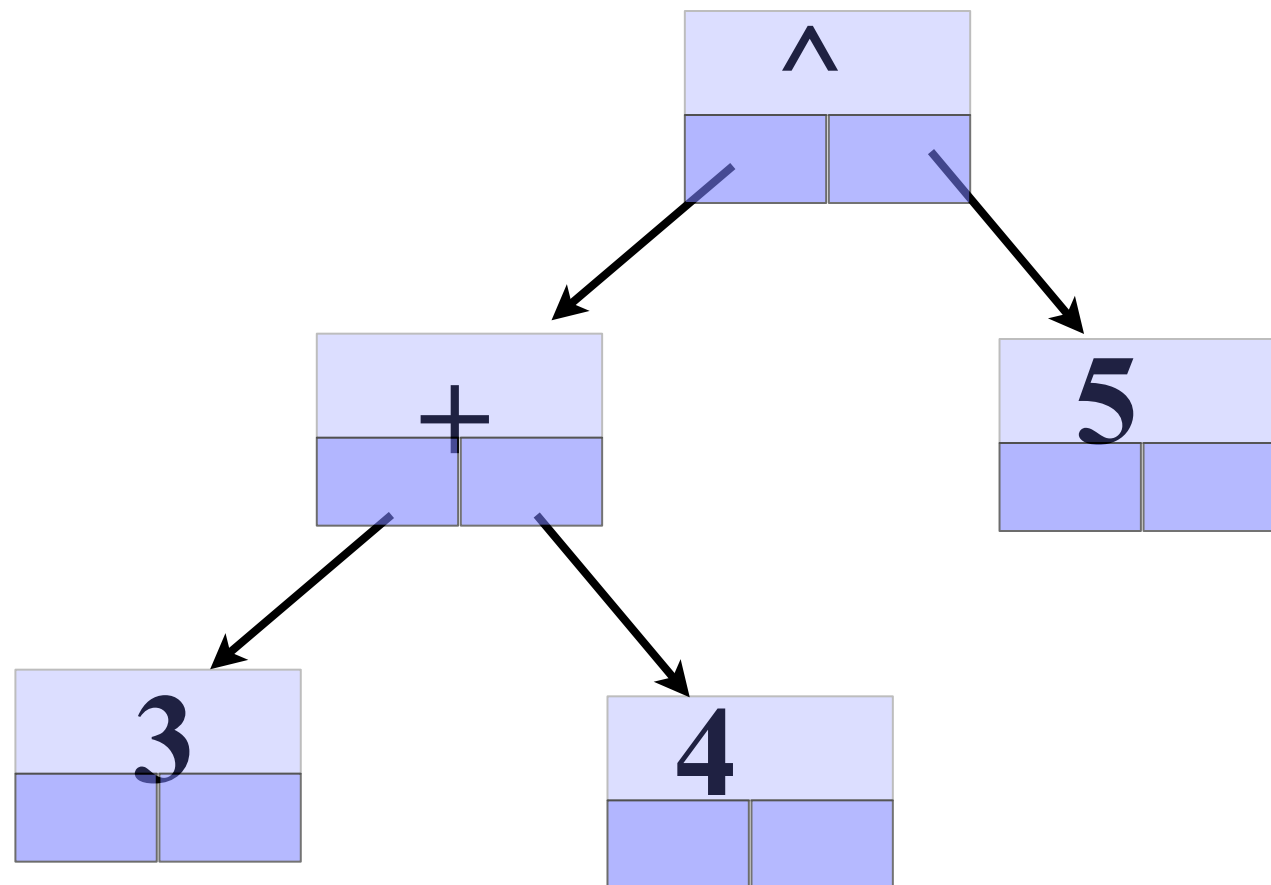


# Example: Expression Trees

- Internal nodes: operator symbols
- Leaves: numbers (or other operand tokens)
- Left and right subtrees of a node represent the left and right operands for the operator stored in that node

# Expression Tree

$$(3 + 4)^5$$



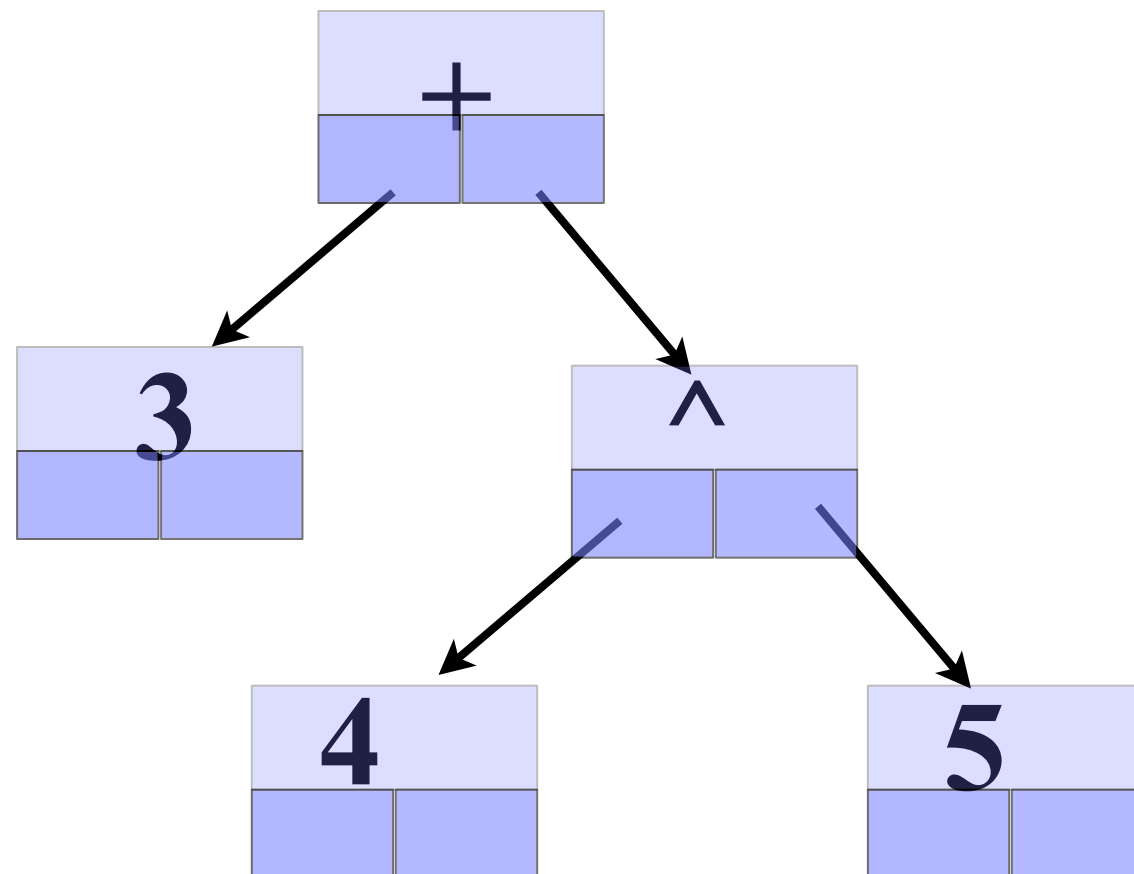
inorder:  $(3 + 4) \wedge (5)$

preorder:  $\wedge + 3 4 5$

postorder:  $3 4 + 5 \wedge$

# Expression Tree

$$3 + 4^5$$



inorder:  $(3 + (4 ^ 5))$

preorder:  $+ 3 ^ 4 5$

postorder:  $3 4 5 ^ +$

# Expression Trees

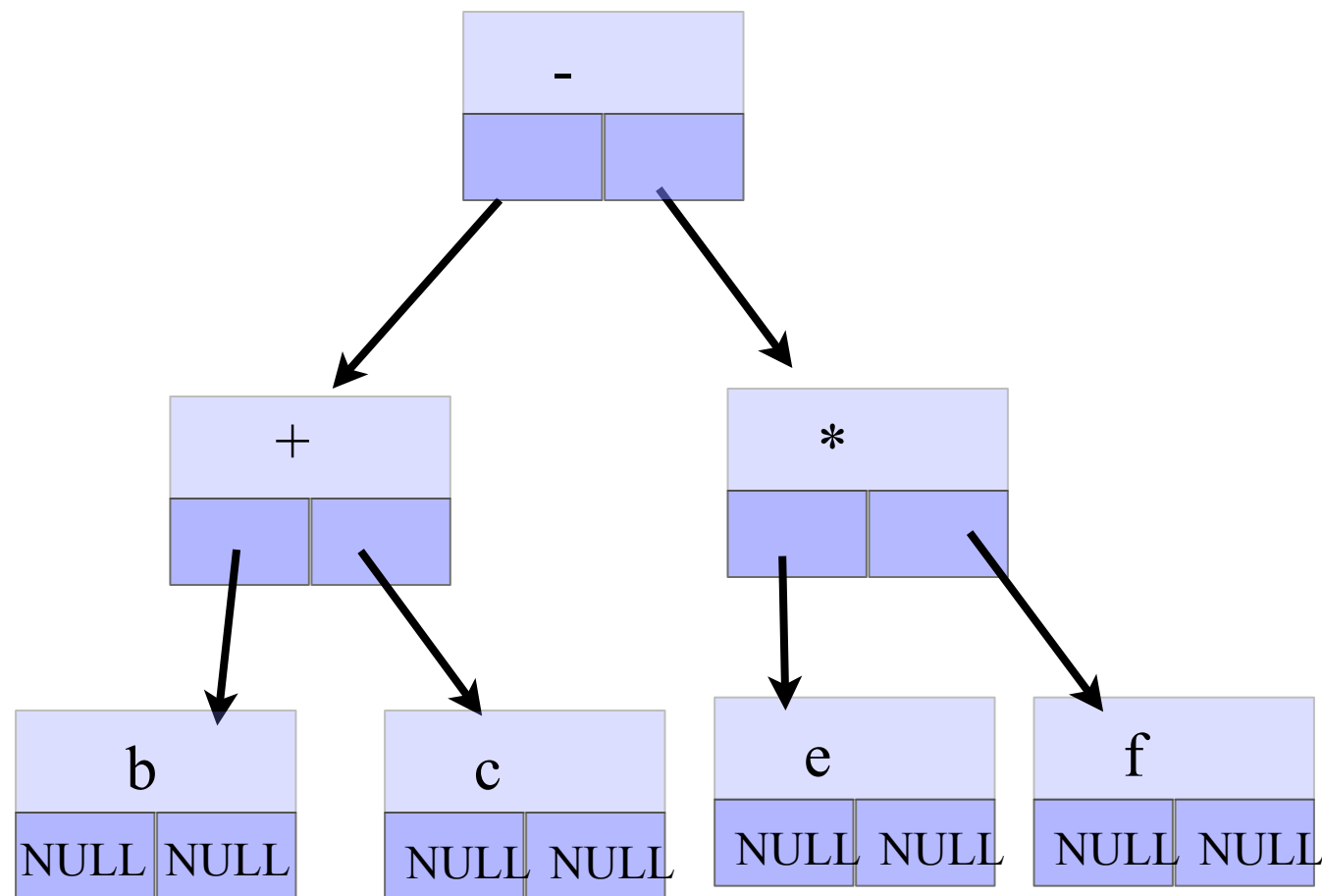
- leaves are operands
- other nodes are operators
- binary operators implies binary tree
- a node would have only one child if unary (e.g. -)
- evaluate by applying the operator at the root and recursively evaluating the left and right subtrees



# Expression Trees

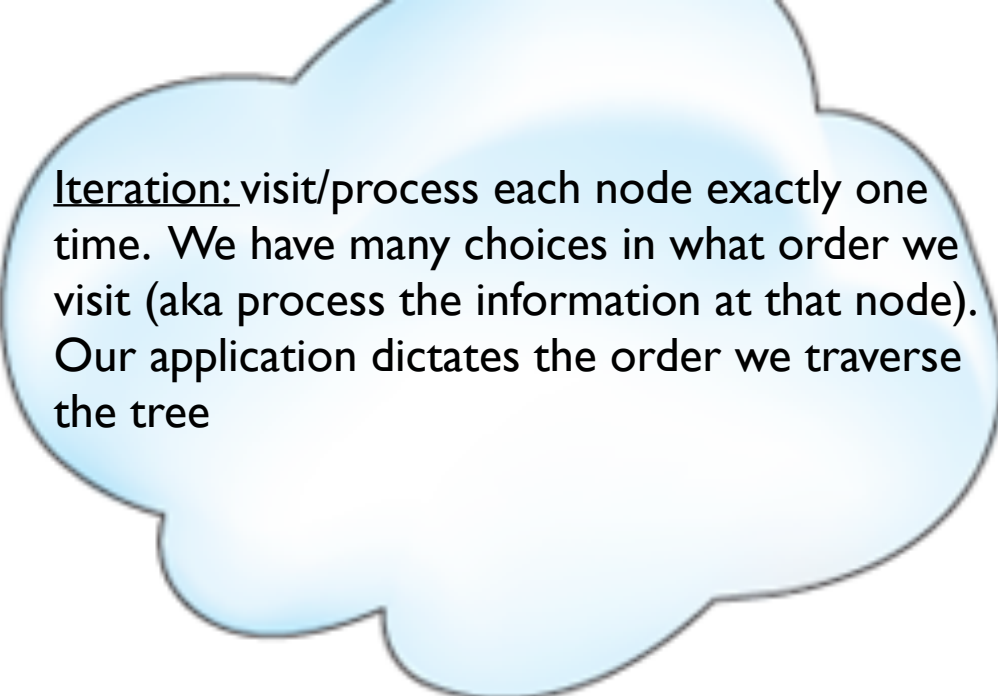
Construct an expression tree from a postfix expression. Create a stack<pointer> see an operand create a single node tree and push a pointer to it on our stack, when we seen an operator, pop and merge the two top trees on the stack

Given a postfix expression,  $b\ c\ +\ e\ f\ *\ -$ ,  
we can represent it as a tree



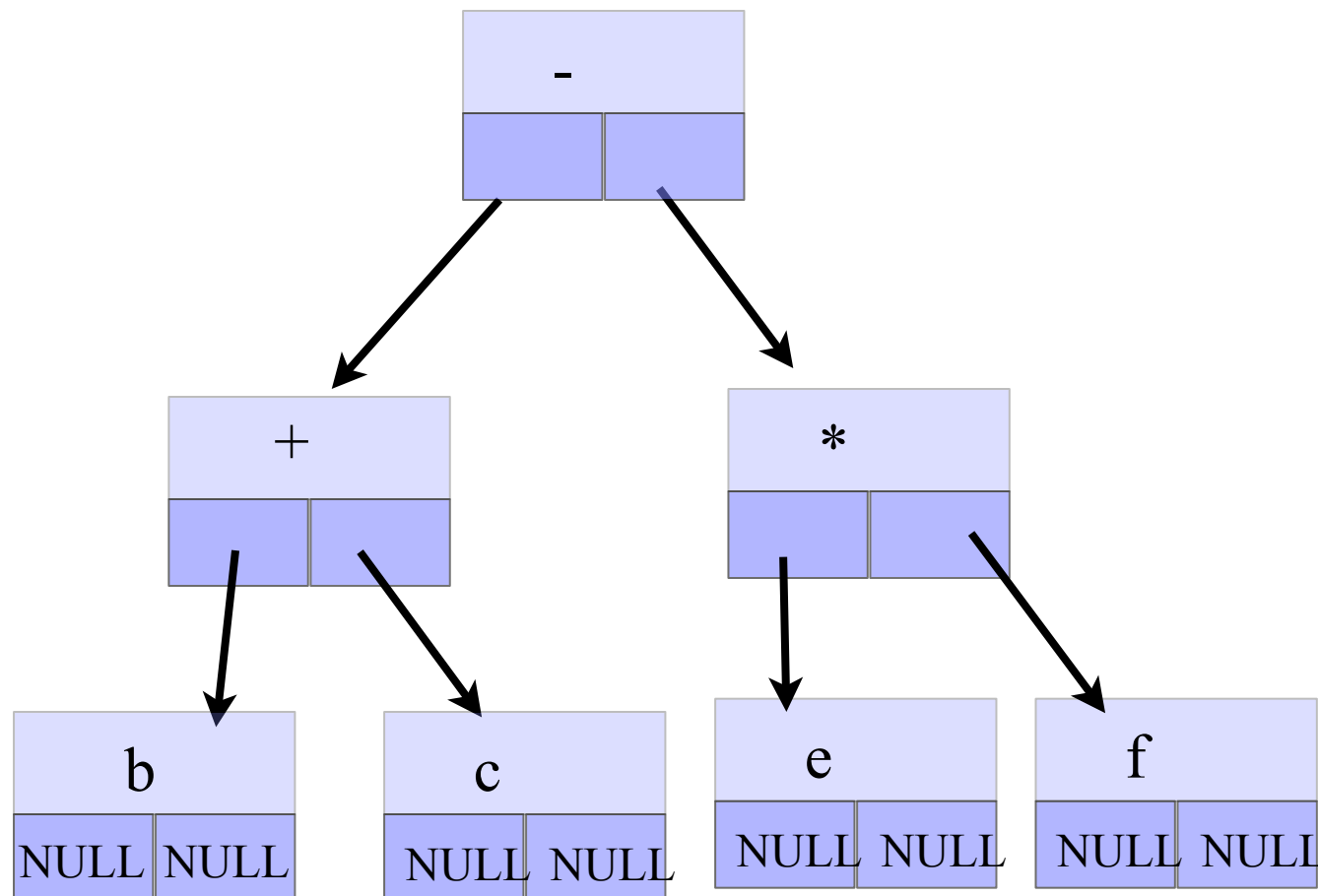
# Tree Traversals

- pre-order
  - visit root
  - traverse left subtree
  - traverse right subtree
- post-order
  - traverse left subtree
  - traverse right subtree
  - visit root
- in-order
  - traverse left subtree
  - visit root
  - traverse right subtree



Iteration: visit/process each node exactly one time. We have many choices in what order we visit (aka process the information at that node). Our application dictates the order we traverse the tree

The expression tree for  $b + c - e * f$



Inorder  $(b + c) - (e * f)$

Postorder  $b c + e f * -$

Preorder  $- + b c * e f$

# The node class

```
template <class Object>
class BinaryNode
{
public:
    BinaryNode( const Object & theElement = Object( ),
                BinaryNode *lt = NULL, BinaryNode *rt = NULL )
        : element( theElement ), left( lt ), right( rt ){ }
    BinaryNode( Object && theElement,
                BinaryNode *lt = NULL, BinaryNode *rt = NULL )
        : element( std::move(theElement) ), left( lt ), right( rt ){ }

    static int size( BinaryNode *t );
    static int height( BinaryNode *t );
    BinaryNode *duplicate( ) const;

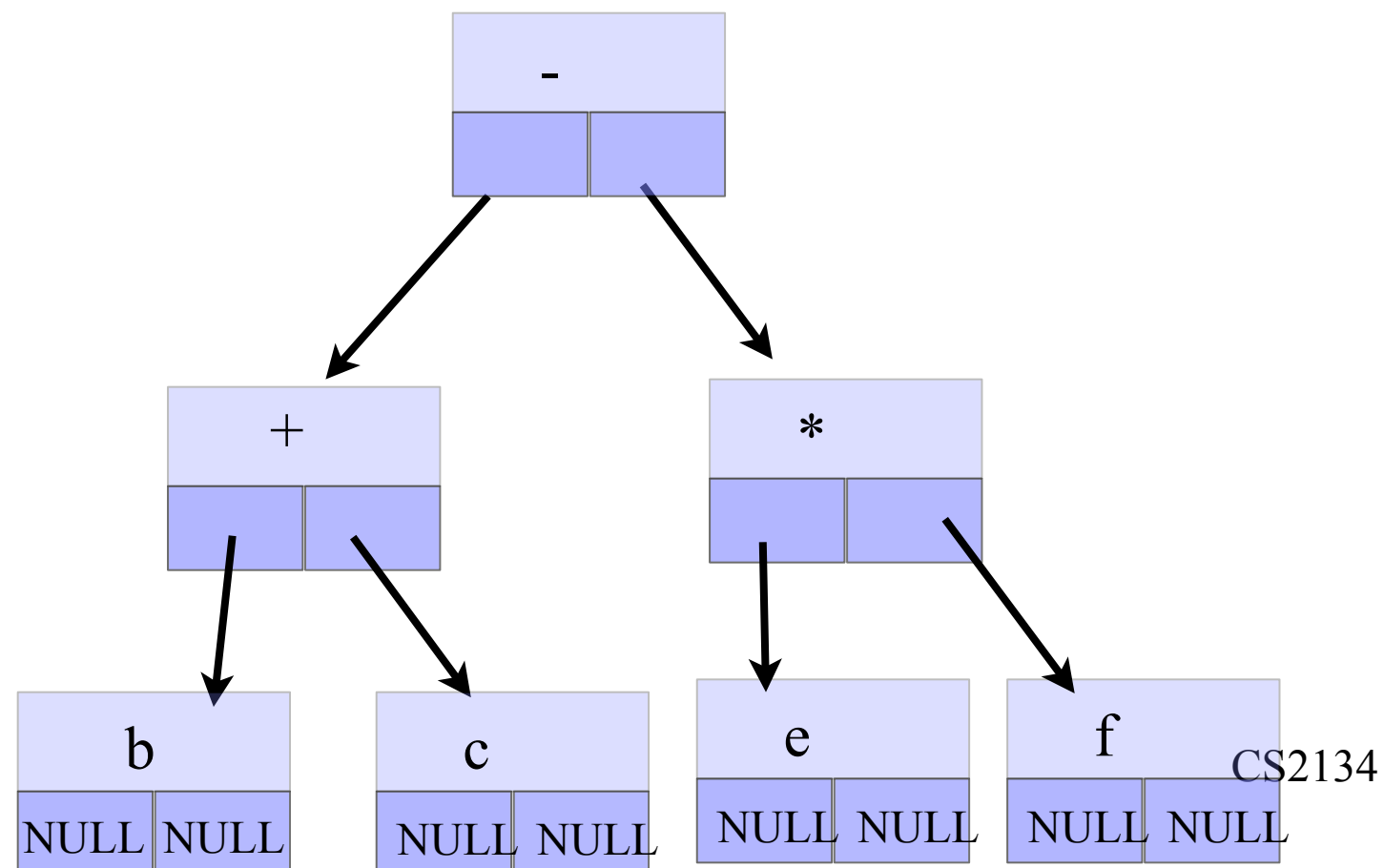
    void printPreOrder( ) const;
    void printPostOrder( ) const;
    void printInOrder( ) const;

public: // To keep things simple
    Object element;
    BinaryNode *left;
    BinaryNode *right;
};
```

# Code to print a tree in preorder

// Print the tree rooted at current node using preorder traversal.

```
template <class Object>
void BinaryNode<Object>::printPreOrder( ) const
{
    cout << element << " ";           // Node
    if( left != NULL )
        left->printPreOrder( );         // Left
    if( right != NULL )
        right->printPreOrder( );        // Right
}
```

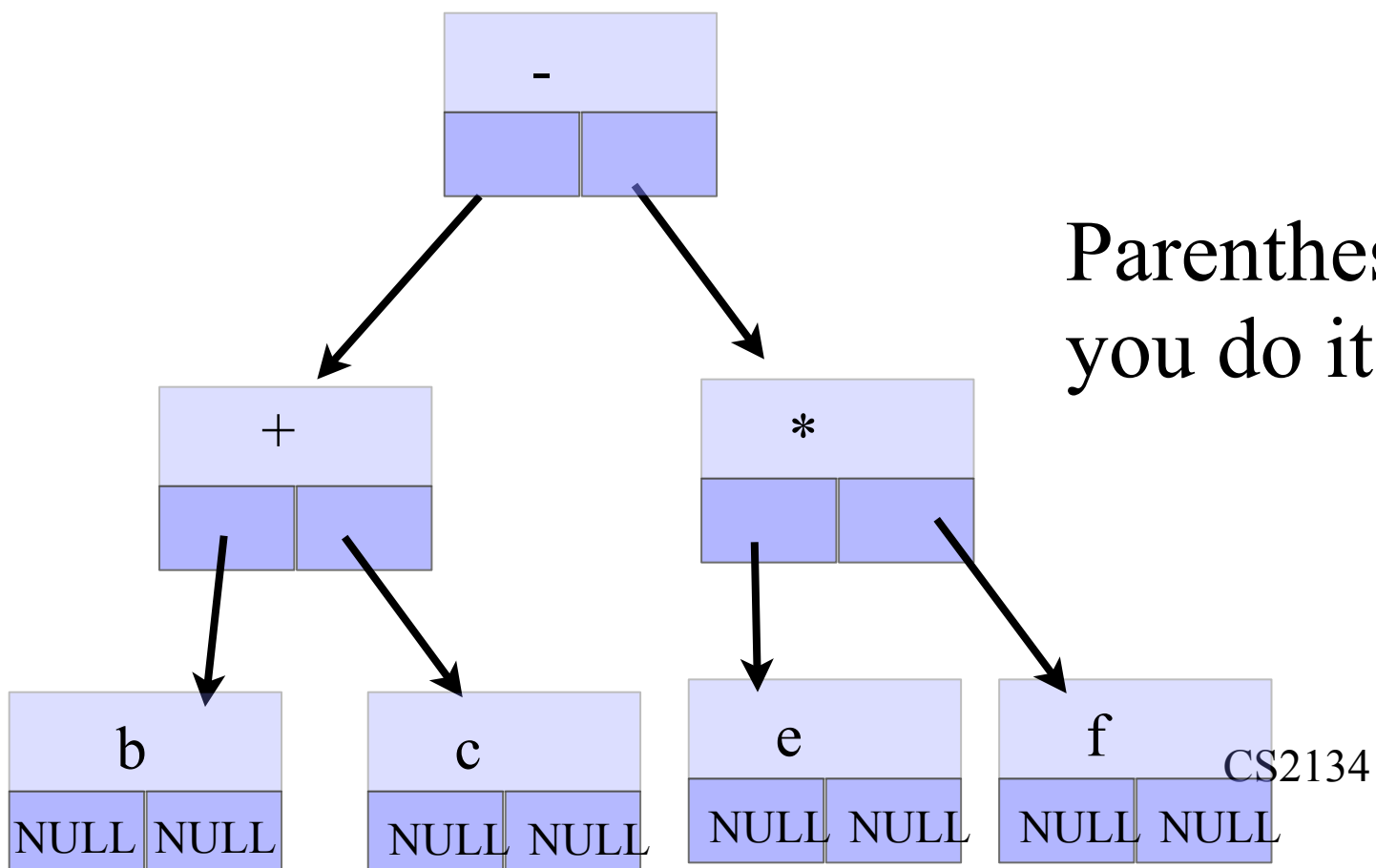


# Code to print a tree in inorder

// Print the tree rooted at current node using inorder traversal.

```
template <class Object>
void BinaryNode<Object>::printlnOrder( ) const
{
    if( left != NULL )           // Left
        left->printlnOrder( );
    cout << element << " ";    // Node
    if( right != NULL )         // Right
        right->printlnOrder( );
}
```

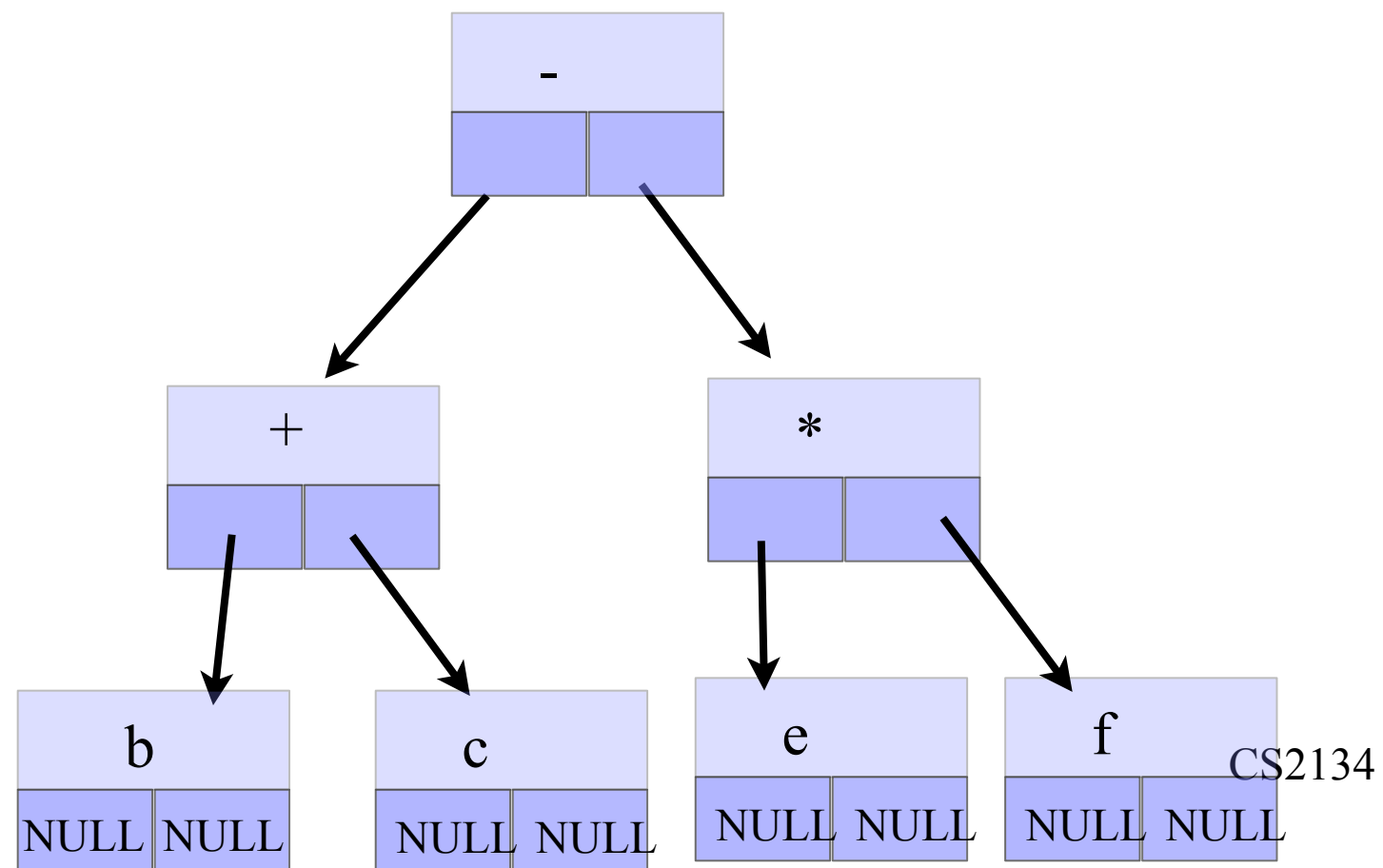
Parenthesis is not written here how would you do it?



# Code to print a tree in postorder

// Print the tree rooted at current node using postorder traversal.

```
template <class Object>
void BinaryNode<Object>::printPostOrder( ) const
{
    if( left != NULL )           // Left
        left->printPostOrder( );
    if( right != NULL )          // Right
        right->printPostOrder( );
    cout << element << " ";    // Node
}
```





## The node class

```
template <class Object>
class BinaryNode
{
public:
    BinaryNode( const Object & theElement = Object( ),
                BinaryNode *lt = NULL, BinaryNode *rt = NULL )
        : element( theElement ), left( lt ), right( rt ){ }
```

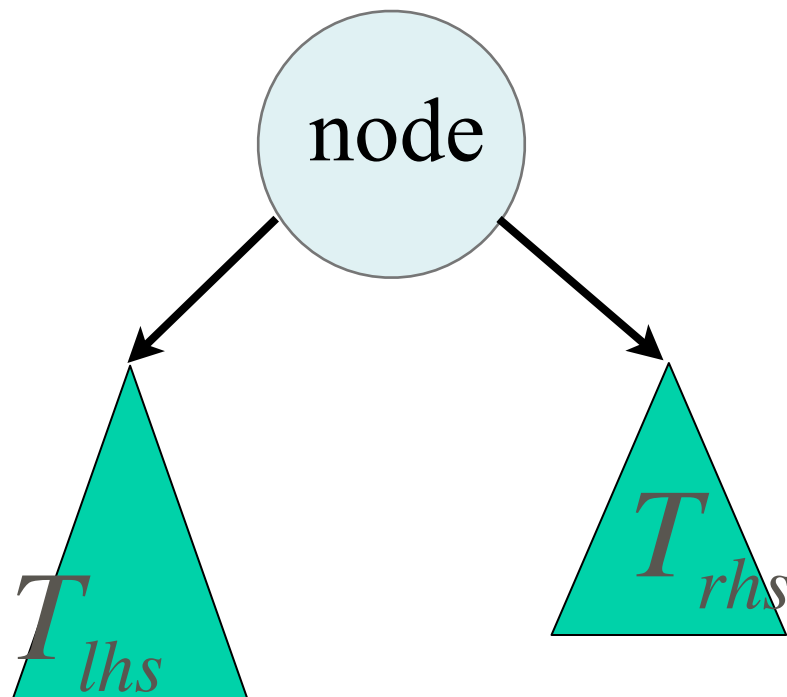
```
    static int size( BinaryNode *t );
    static int height( BinaryNode *t );
    BinaryNode *duplicate( ) const;
```

```
void printPreOrder( ) const;
void printPostOrder( ) const;
void printInOrder( ) const;
```

```
public: // To keep things simple
    Object    element;
    BinaryNode *left;
    BinaryNode *right;
};
```

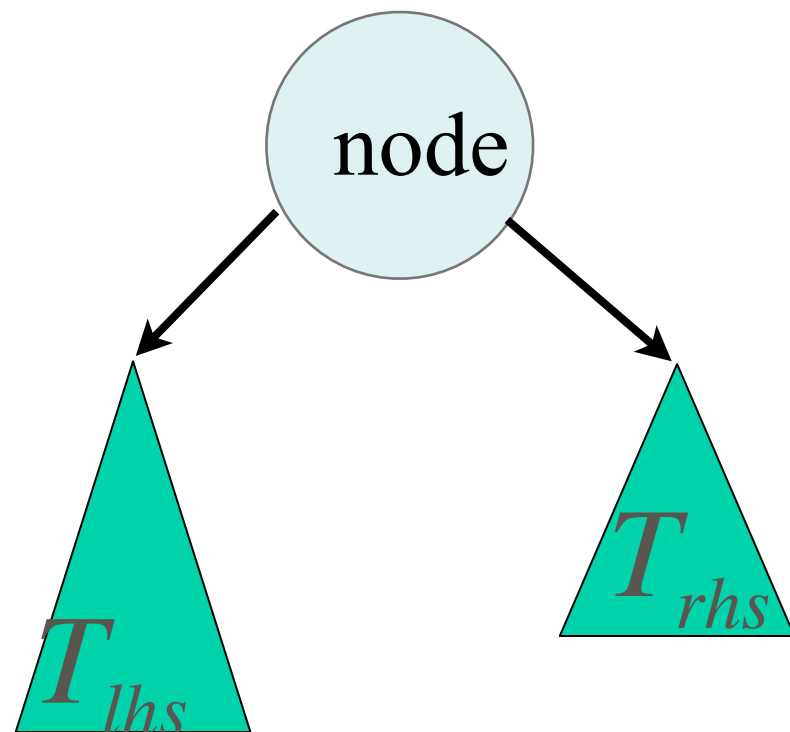
# Code to compute the size of a tree

```
// Return size of tree rooted at t.  
template <class Object>  
int BinaryNode<Object>::size( BinaryNode<Object> * t )  
{  
    if( t == NULL )  
        return 0;  
    else  
        return 1 + size( t->left ) + size( t->right );  
}
```



# Code to compute the height of a tree

```
// Return height of tree rooted at t.  
template <class Object>  
int BinaryNode<Object>::height( BinaryNode<Object> * t )  
{  
    if( t == NULL )  
        return -1;  
    else  
        return 1 + max( height( t->left ),  
                        height( t->right ) );  
}
```

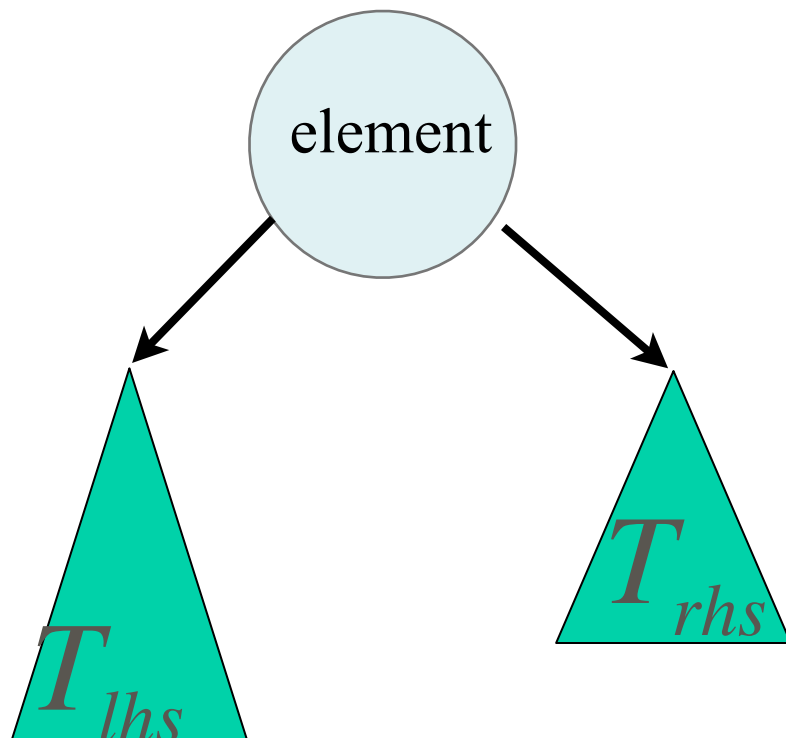


# Code to recursively copy a tree

```
// Return a pointer to a node that is the root of a
// duplicate of the tree rooted at the current node.
template <class Object>
BinaryNode<Object> * BinaryNode<Object>::duplicate( ) const
{
    BinaryNode<Object> *root = new BinaryNode<Object>( element );

    if( left != NULL )    // If there's a left subtree
        root->left = left->duplicate( );// Duplicate; attach
    if( right != NULL )   // If there's a right subtree
        root->right = right->duplicate( );// Duplicate; attach

    return root;         // Return resulting tree
}
```



CS2134

