# COMP20010: Algorithms and Imperative Programming

## Lecture 1

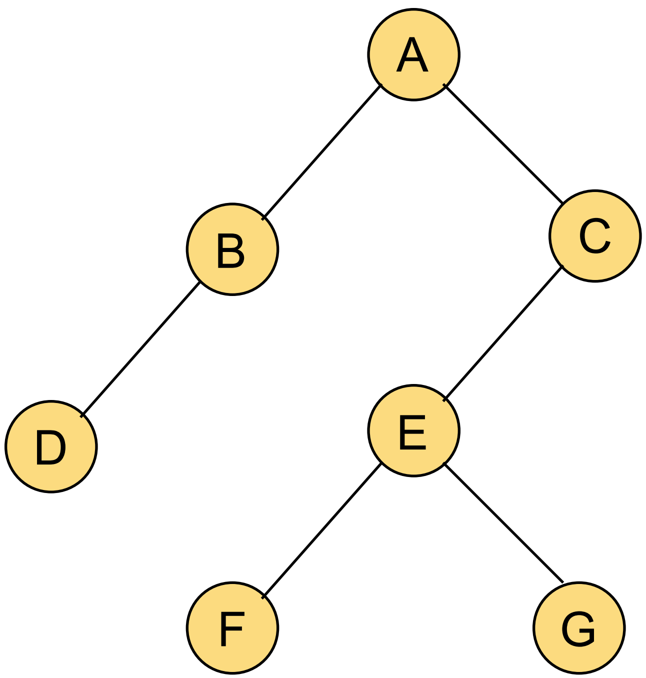
Trees

# Lecture outline

- Motivation
- Definitions
- Ordered trees
- Generic methods for tree operations
- Tree traversal (preorder, postorder, inorder)
- Binary trees – tree traversal

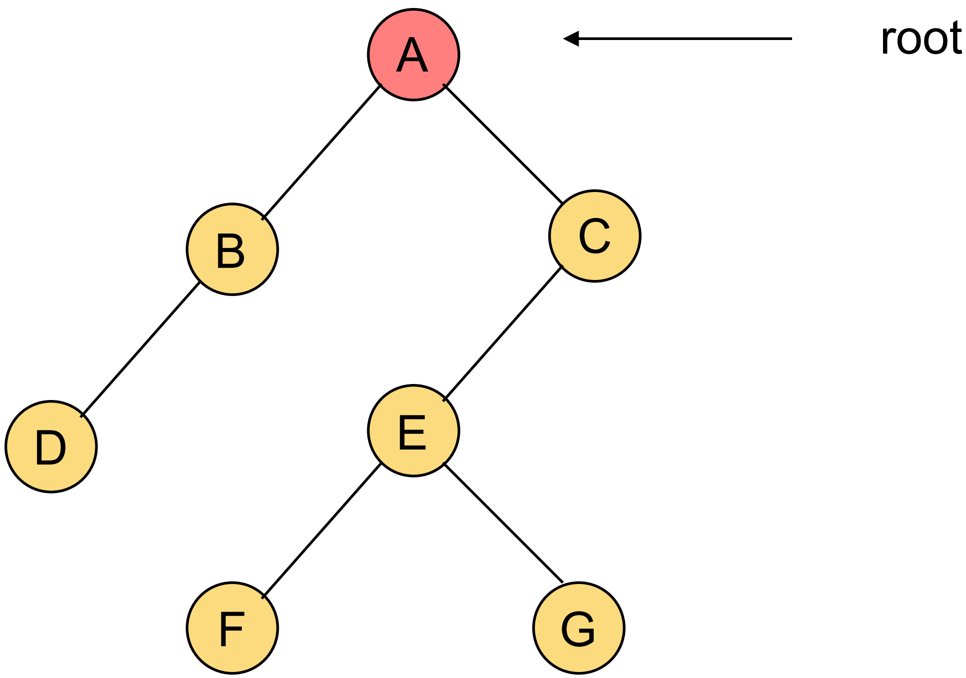
# Definitions

- An abstract data type for hierarchical storage of information;



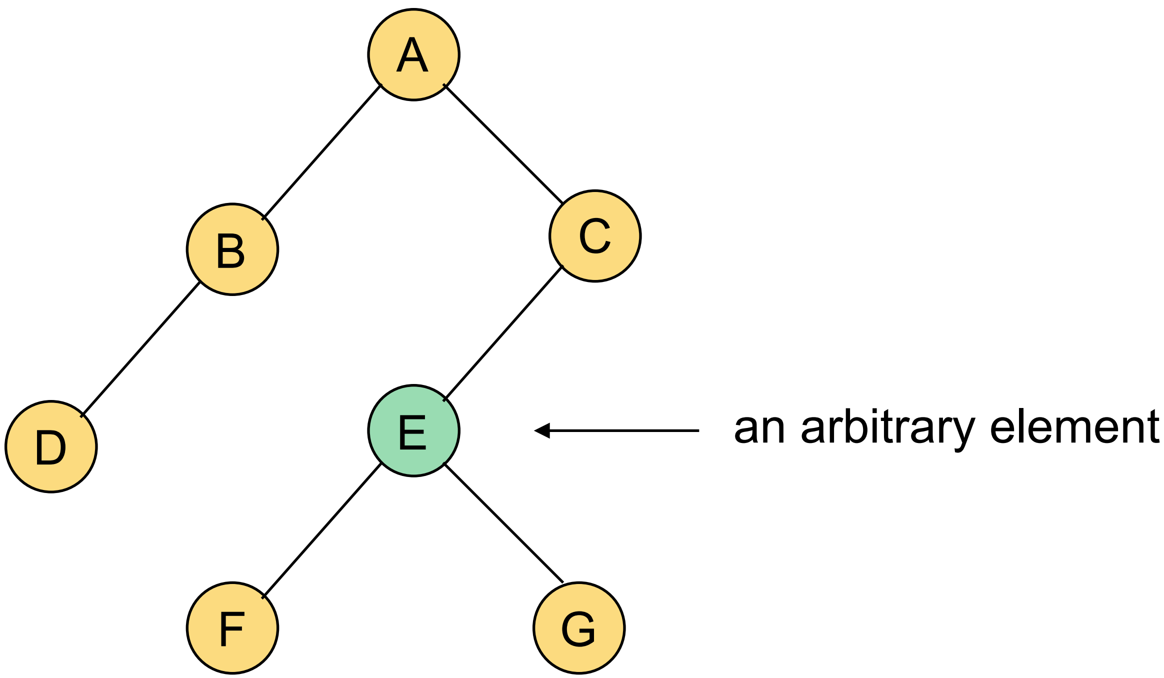
# Definitions

- The top element of a tree is referred to as the root of a tree;



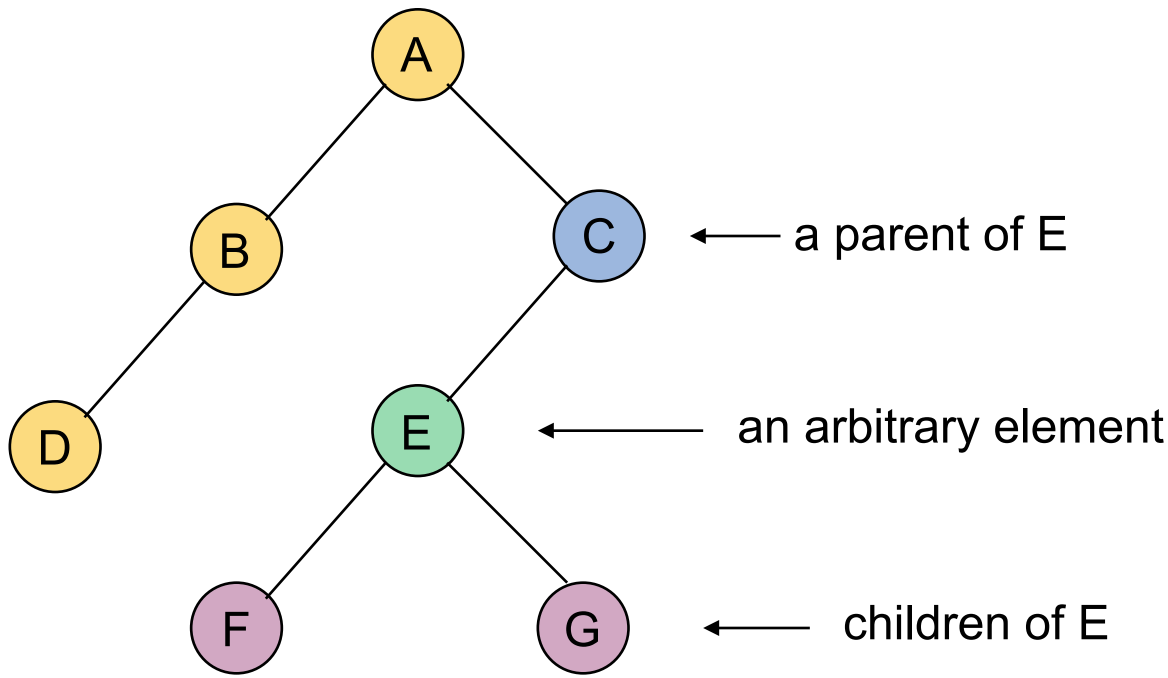
# Definitions

- Each element in a tree (except the root)

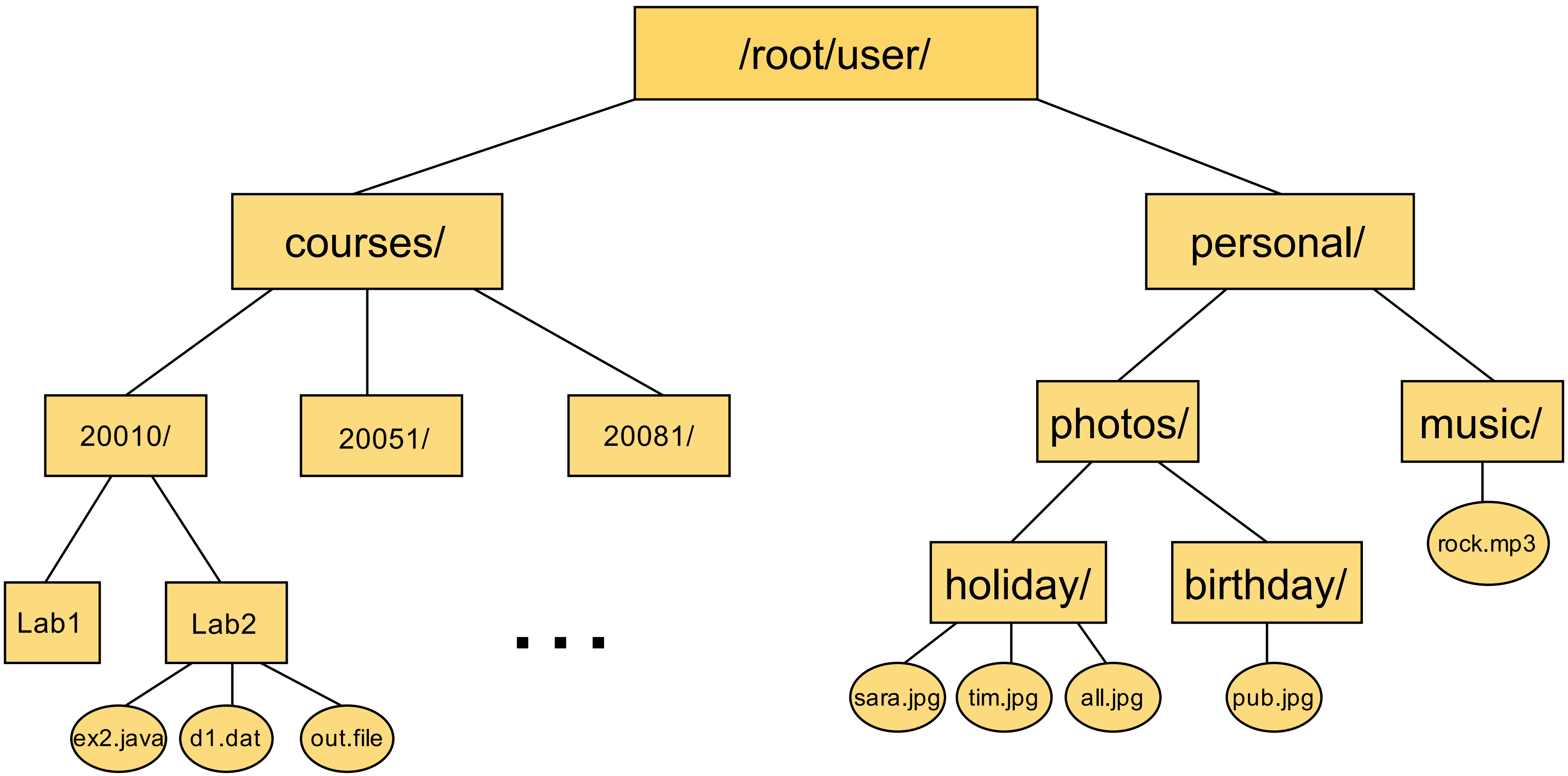


# Definitions

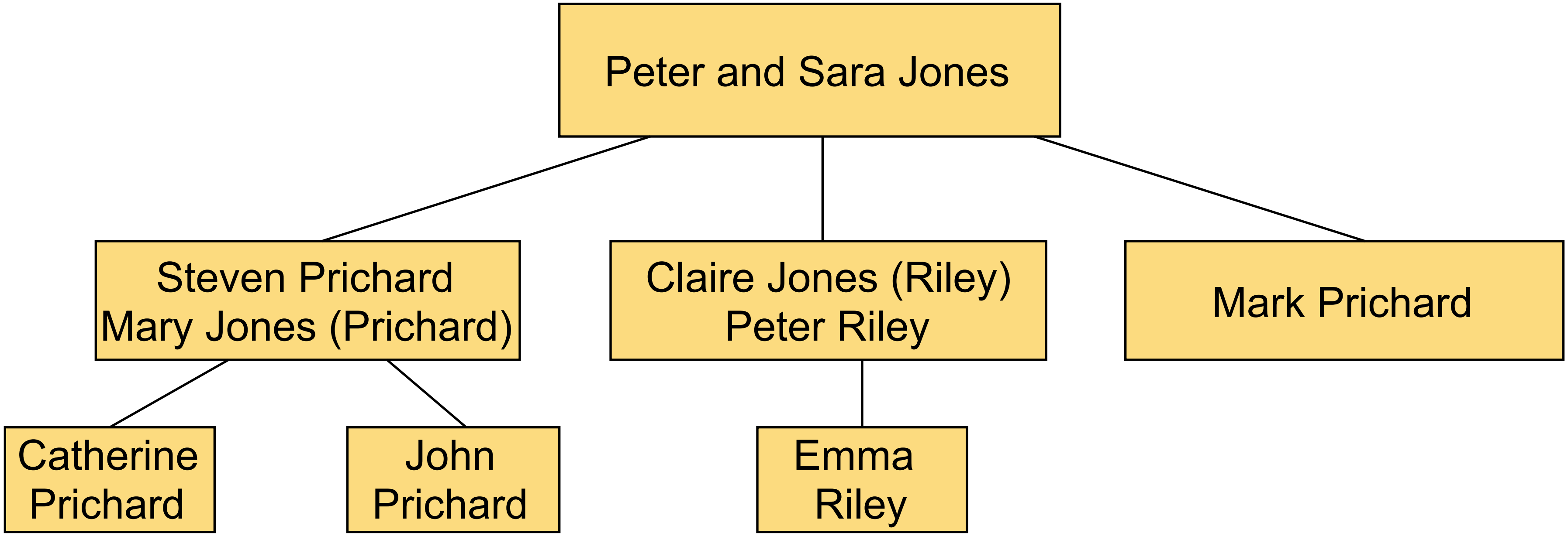
- Each element in a tree (except the root) has a parent and zero or more children elements;



# Examples
# Computer disc directory structure



# Examples
# A family tree

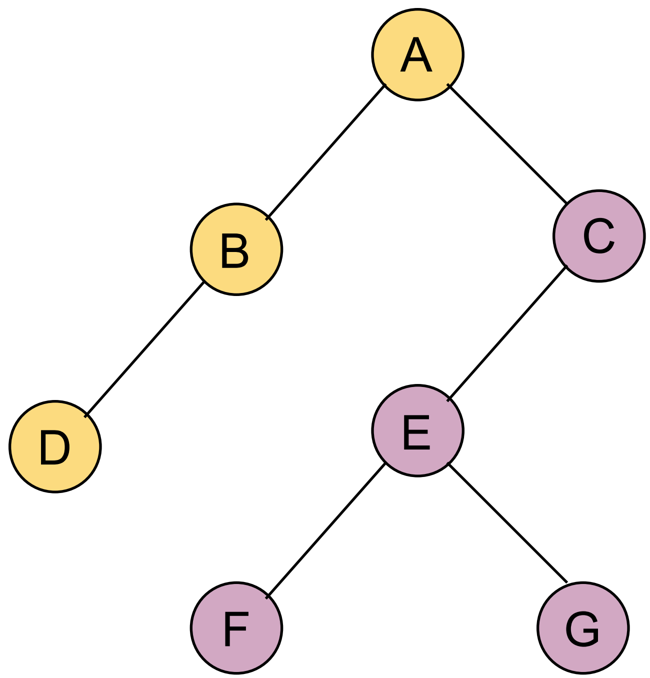


# Definitions

- A tree *T* is a non-empty set of nodes storing useful information in a parent-child relationship with the following properties:
  - *T* has a special node *r* referred to as the root;
  - Each node *v* of *T* different from *r* has a parent node *u*;
- If the node *u* is the parent (ancestor) node of *v*, then *v* is a child (descendent) of *u*. Two children of the same parent are siblings.
- A node is external (a leaf node) if it has no children and internal if it has one or more children.

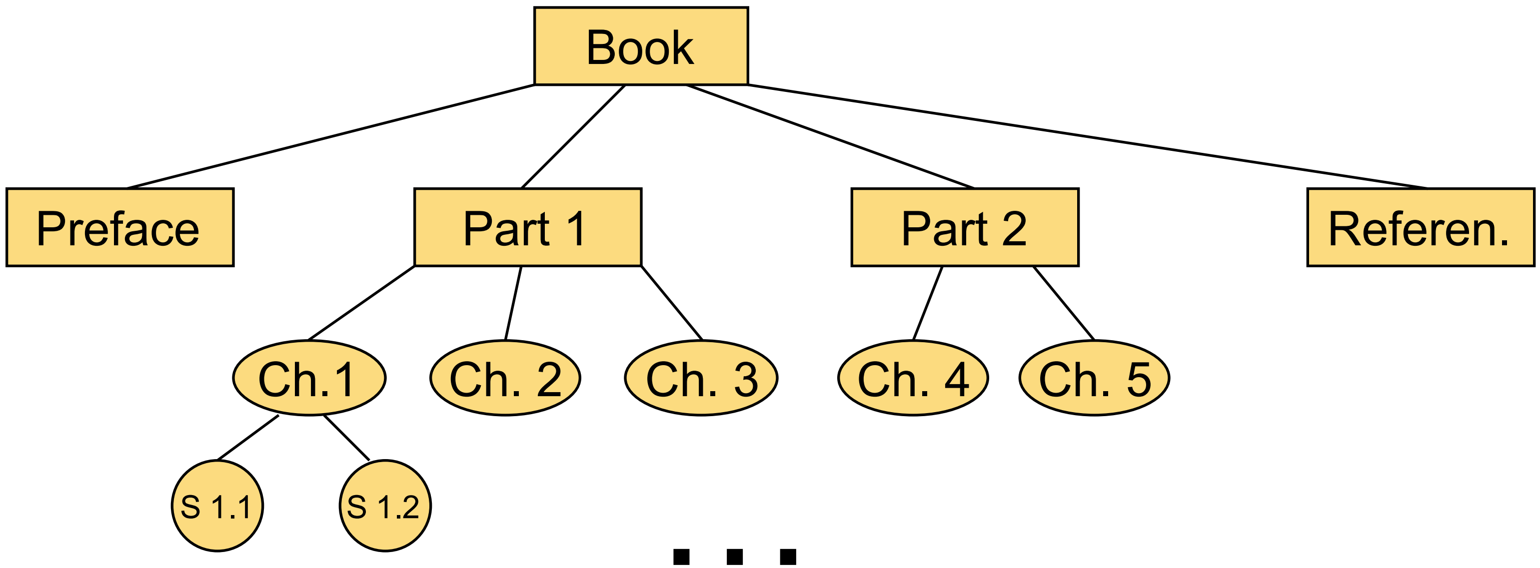
# Definitions

- A sub-tree of *T* rooted at the node *v* is a tree consisting of all the descendants of *v* in *T*, including *v* itself.



# Ordered trees

- A tree is ordered if a linear ordering relation is defined for the children of each node, that is, we can define an order among them.
- Example: a book



# Binary trees

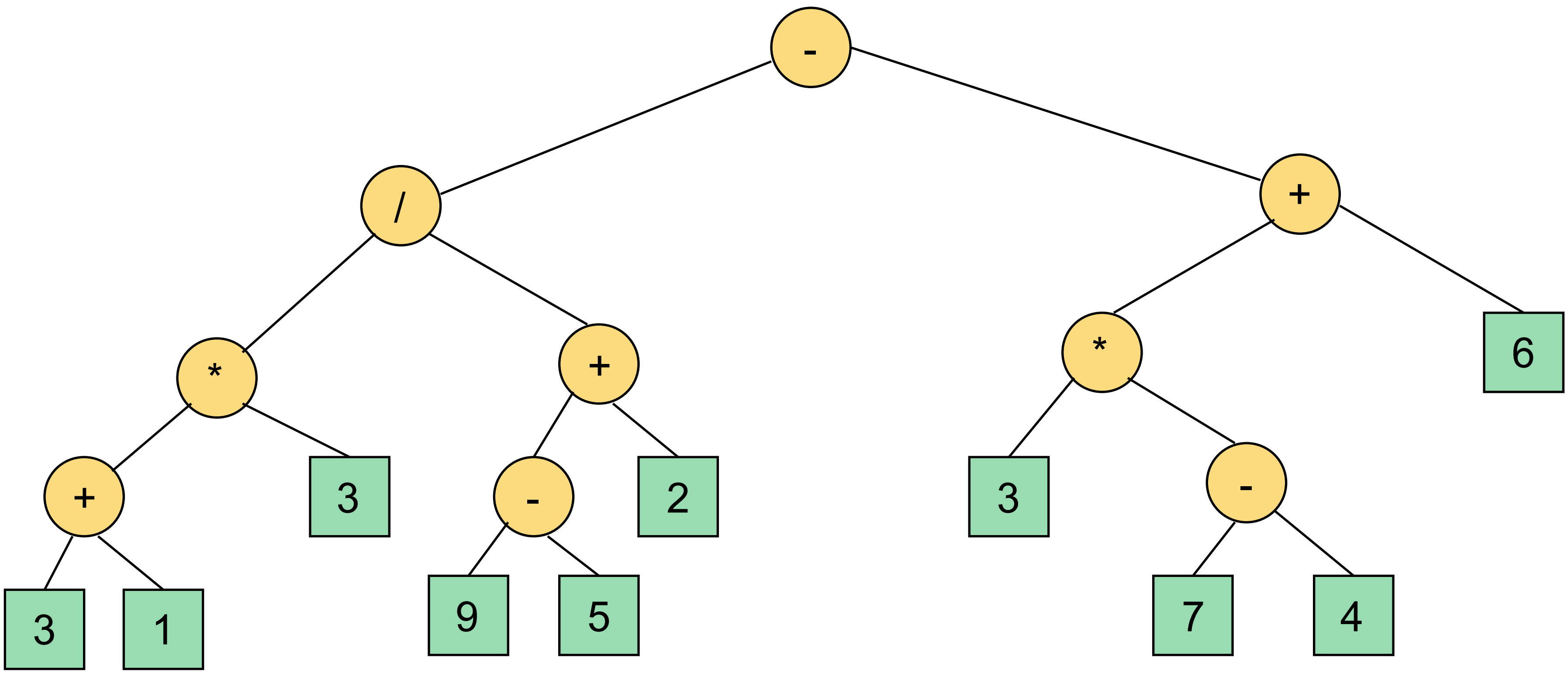
- A binary tree is an ordered tree in which each node has at most two children.
- A binary tree is proper if each internal node has two children.
- For each internal node its children are labelled as a left child and a right child.
- The children are ordered so that a left child comes before a right child.

# An example of a binary tree

- Representing an arithmetic expression by a binary tree in which the external nodes are associated with variables or constants and the internal nodes are associated with 4 arithmetic operations.

# An example of a binary tree

- ((((3+1)*3)/((9-5)+2))-((3*(7-4))+6))



# The tree abstract data type

- Elements of a tree are stored at positions (tree nodes) which are defined relative to neighbouring positions (parent-child relationships).
- Accessor methods for the tree ADT:
  - root() – returns the root of the tree;
  - parent(*v*) – returns the parent node of *v* (an error if *v* is the root);
  - children(*v*) – returns an iterator of the children of the node *v* (if *v* is a leaf node it returns an empty iterator);

# The tree abstract data type

- Querry methods for the tree ADT:
  - isInternal(*v*) – tests whether the node *v* is internal;
  - isExternal(*v*) – tests whether the node *v* is external;
  - isRoot(*v*) – tests whether the node *v* is the root;
- Generic methods (not necessarily related to the tree structure):
  - size() – returns the number of nodes of the tree;
  - elements() – returns the an iterator of all the elements stored at the nodes of the tree;
  - positions() – returns an iterator of all the nodes of the tree;
  - swapElements(*v,w*) – swaps the elements stored at the nodes *v* and *w*;
  - replaceElement(*v,e*) – returns the element *v* and replaces it with the element *e*;

# Tree traversal

- Complexity of the methods of the tree ADT:
  - root() and parent(*v*) take O(1) time;
  - isInternal(*v*), isExternal(*v*), isRoot(*v*) take O(1) time;
  - children(*v*) takes O($c_v$) time, where $c_v$ is the number of children of *v;*
  - swapElements(*v,w*) and replaceElement(*v,e*) take O(1) time;
  - elements() and positions() take O(*n*) time, where *n* is the number of elements in the tree;

# Depth of a node

- Let *v* be a node of a tree *T*. The depth of *v* is a number of ancestors of *v*, excluding *v* itself.
- The depth of the root is 0;
- Recursive definition:
  - If *v* is a root, then the depth of *v* is 0;
  - Otherwise, the depth of *v* is one plus the depth of its parent;

**Algorithm** depth(*T,v*)
  **if** *T.*isroot(*v*) **then**
    **return** 0
  **else**
    **return** 1+depth(*T,T.*parent(*v*))

# The height of a tree

- It is equal to the maximum depth of an external node of *T.*
- If the previous depth-finding algorithm is applied, the complexity would be $O(n^2)$ .
- A recursive definition of height of a node *v* in a tree *T:*
  - If *v* is an external node, the height of *v* is 0;
  - Otherwise, the height of *v* is one plus the maximum height of a child of *v*;

**Algorithm** height(*T,v*)
  **if** T.isExternal(*v*) **then**
    **return** 0
  **else**
    h=0
    **for** each *w* in T.children(*v*) **do**
      h=max(h,height(*T,w*))
    **return** 1+h

# A traversal of a tree

- A traversal of a tree is a sustematic way of accessing (visiting) all the nodes of *T.*
- There are two different traversal schemes for trees referred to as preorder and postorder.
- In a preorder traversal of a tree *T* the root is visited first, and then the subtrees rooted at its children are traversed recursively.

**Algorithm** preorder(*T,v)*

perform the action on the node *v*

**for** each child *w* of *v*

**call** preorder(*T,w*)

# Preorder traversal of a tree

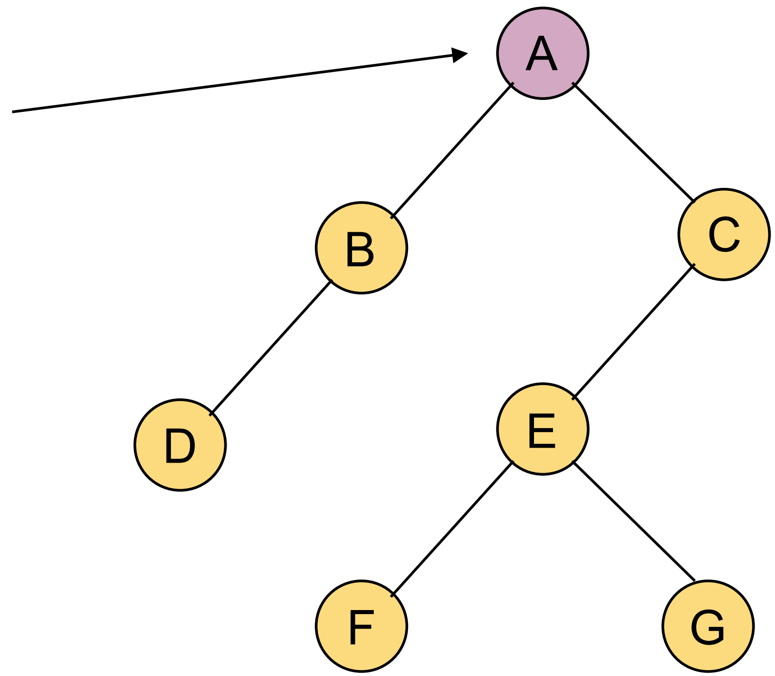
**call** preorder(*T*, *T*.root)

**Algorithm** preorder(*T*,*v*)

perform the action on the node *v*

**for** each child *w* of *v*

**call** preorder(*T*,*w*)



# Preorder traversal of a tree

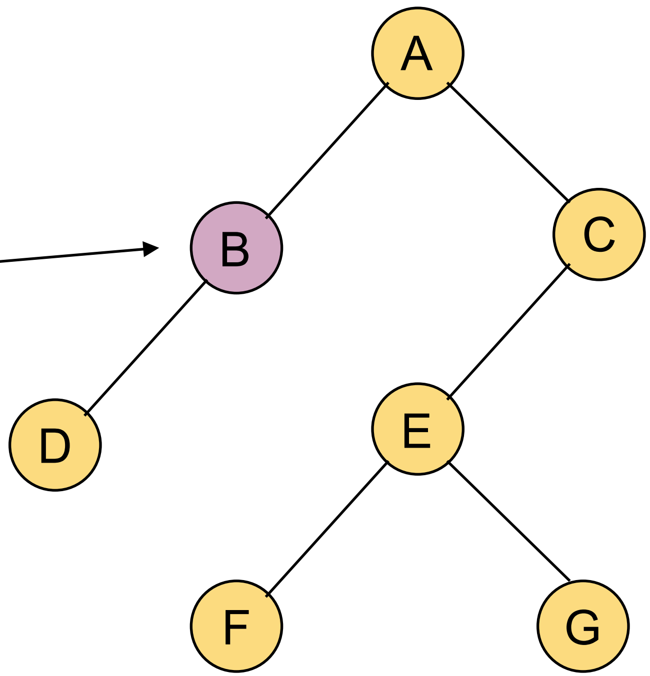
**call** preorder(*T*, *T*.root)

**Algorithm** preorder(*T*,*v*)

perform the action on the node *v*

**for** each child *w* of *v*

**call** preorder(*T*,*w*)



# Preorder traversal of a tree

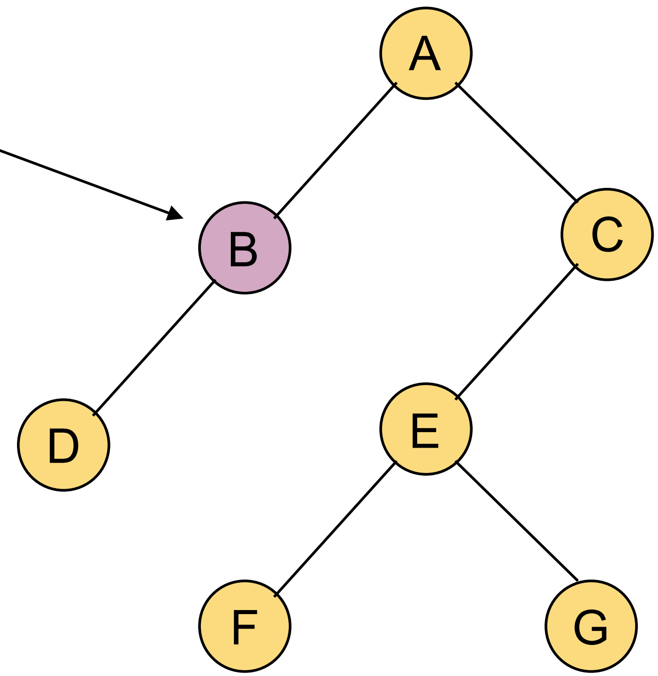
**call** preorder(*T*, *T*.root)

**Algorithm** preorder(*T*,*v*)

perform the action on the node *v*

**for** each child *w* of *v*

**call** preorder(*T*,*w*)



# Preorder traversal of a tree

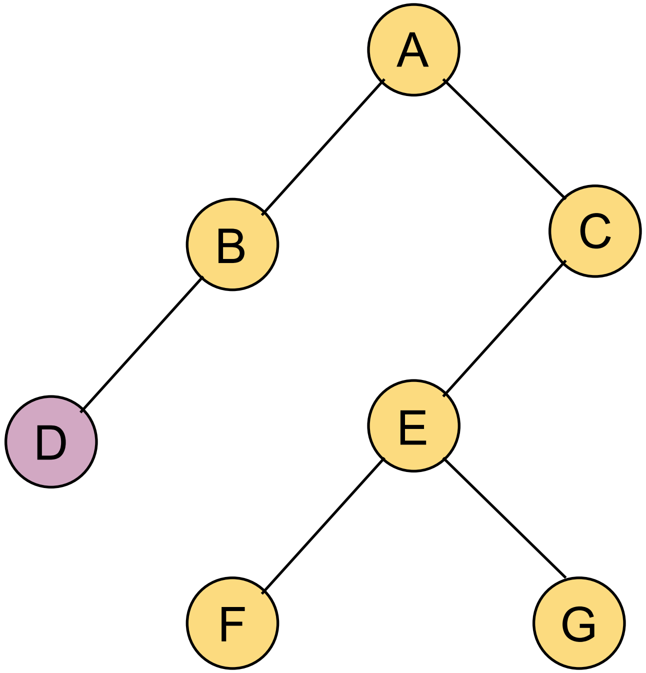
**call** preorder(*T*, *T*.root)

**Algorithm** preorder(*T*,*v*)

perform the action on the node *v*

**for** each child *w* of *v*

**call** preorder(*T*,*w*)



# Preorder traversal of a tree

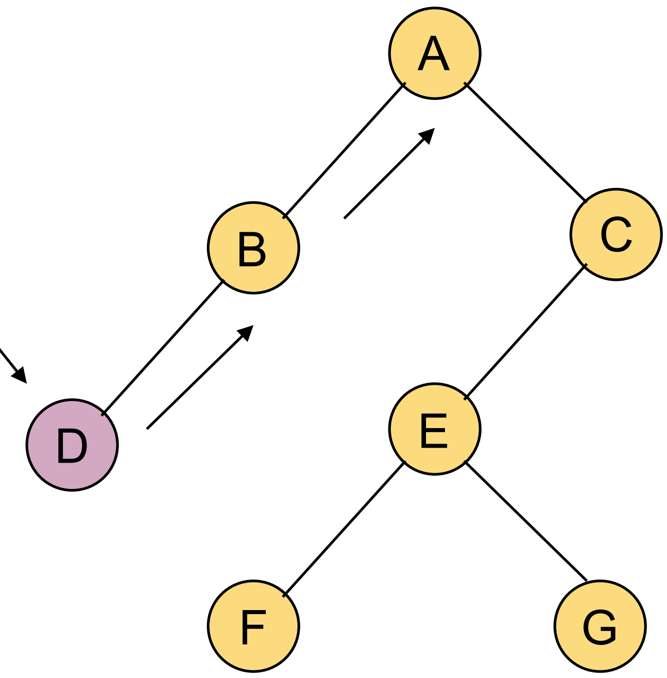
**call** preorder(*T*, *T*.root)

**Algorithm** preorder(*T*,*v*)

perform the action on the node *v*

**for** each child *w* of *v*

**call** preorder(*T*,*w*)



# Preorder traversal of a tree

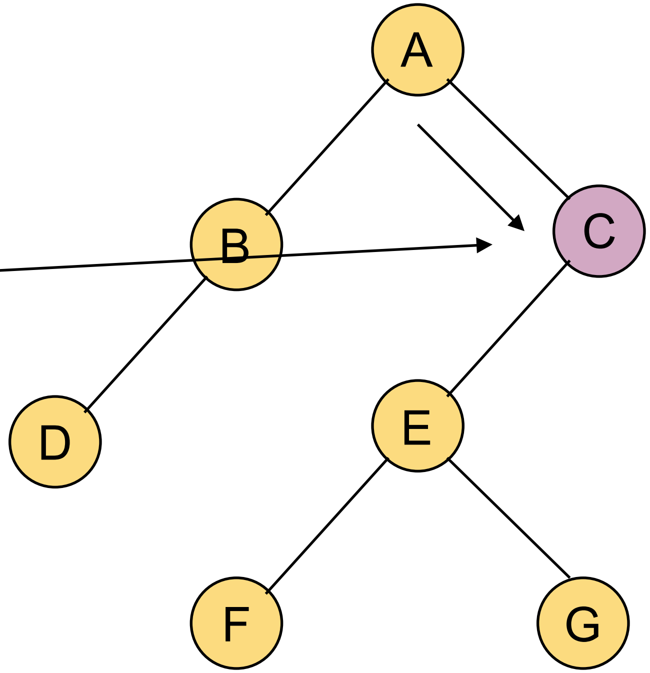
**call** preorder(*T*, *T*.root)

**Algorithm** preorder(*T*, *v*)

perform the action on the node *v*

**for** each child *w* of *v*

**call** preorder(*T*, *w*)



# Preorder traversal of a tree

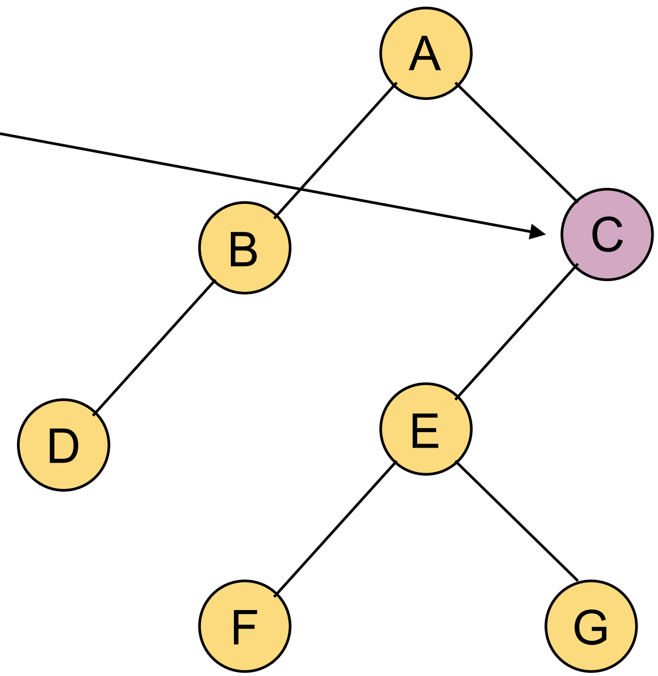
**call** preorder(*T*, *T*.root)

**Algorithm** preorder(*T*,*v*)

perform the action on the node *v*

**for** each child *w* of *v*

**call** preorder(*T*,*w*)



# Preorder traversal of a tree

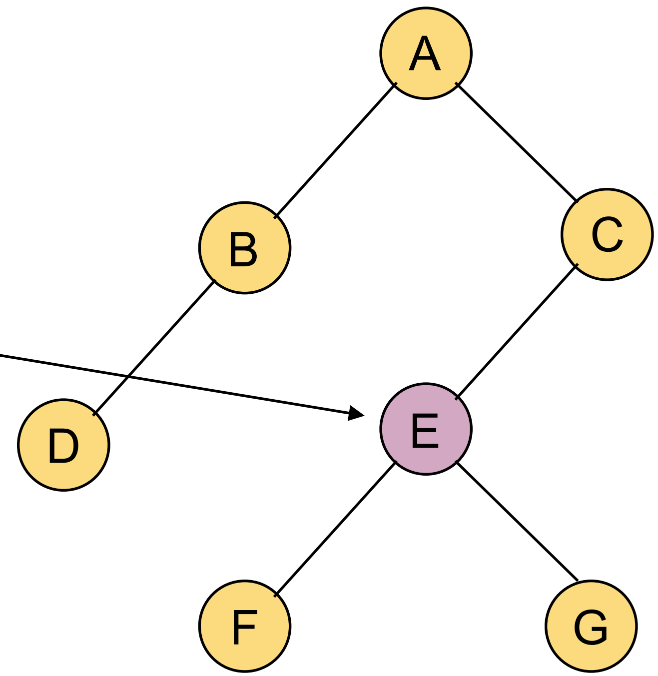
**call** preorder(*T*, *T*.root)

**Algorithm** preorder(*T*,*v*)

perform the action on the node *v*

**for** each child *w* of *v*

**call** preorder(*T*,*w*)



# Preorder traversal of a tree

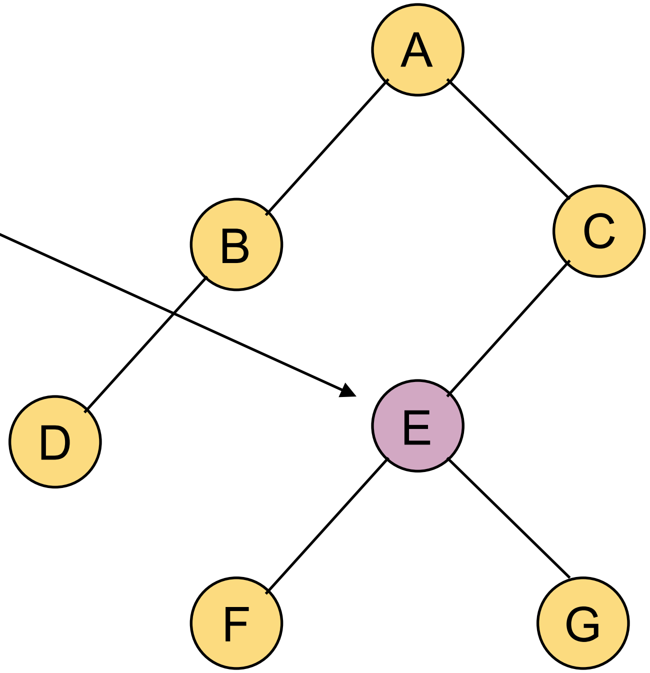
**call** preorder(*T*, *T*.root)

**Algorithm** preorder(*T*,*v*)

perform the action on the node *v*

**for** each child *w* of *v*

**call** preorder(*T*,*w*)



# Preorder traversal of a tree

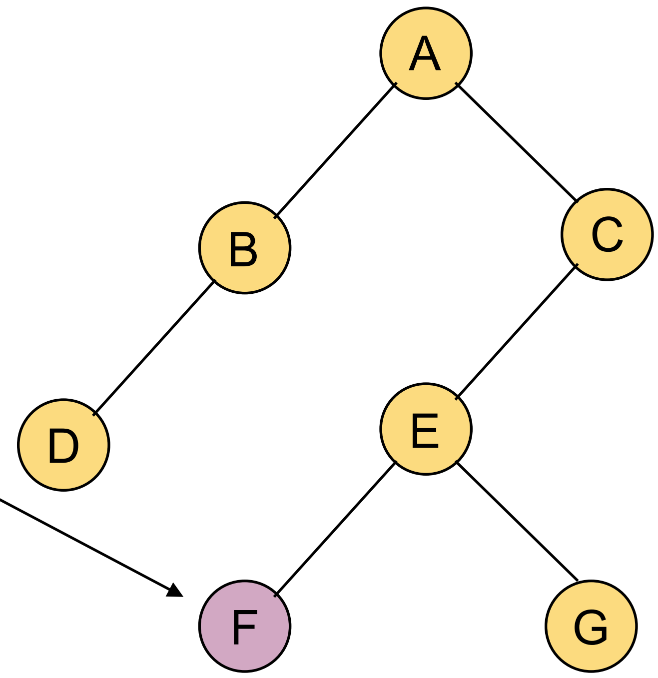
**call** preorder(*T*, *T*.root)

**Algorithm** preorder(*T*,*v*)

perform the action on the node *v*

**for** each child *w* of *v*

**call** preorder(*T*,*w*)



# Preorder traversal of a tree

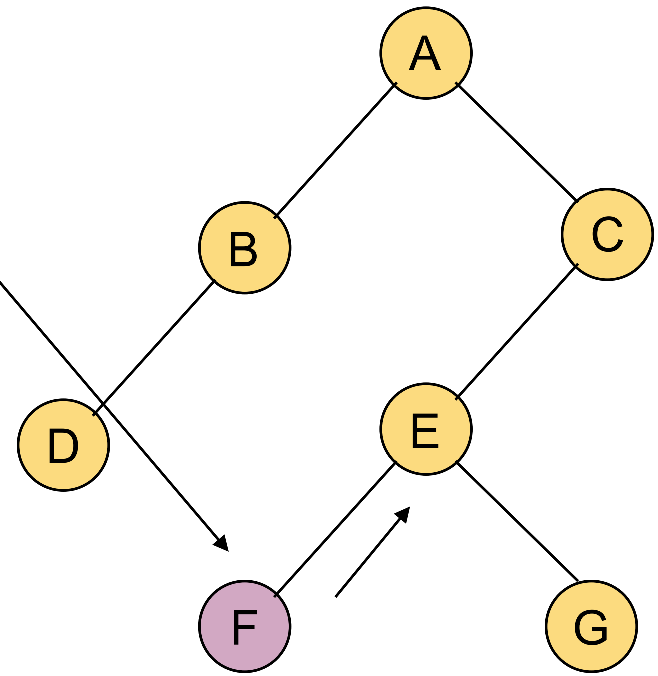
**call** preorder(*T*, *T*.root)

**Algorithm** preorder(*T*, *v*)

perform the action on the node *v*

**for** each child *w* of *v*

**call** preorder(*T*, *w*)



# Preorder traversal of a tree

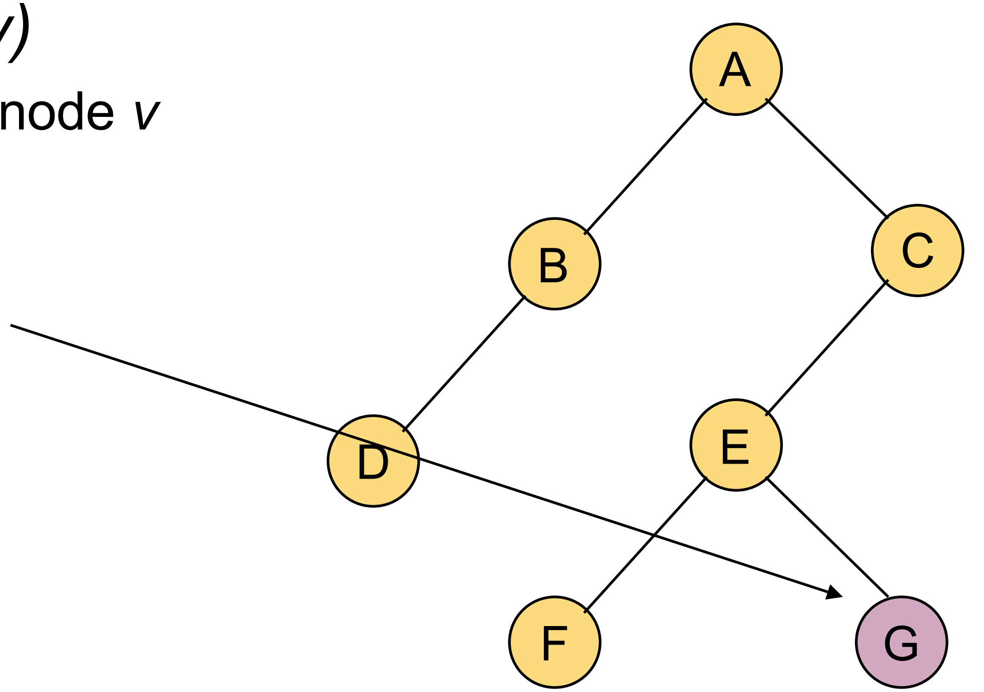
**call** preorder(*T*, *T*.root)

**Algorithm** preorder(*T*,*v*)

perform the action on the node *v*

**for** each child *w* of *v*

**call** preorder(*T*,*w*)



# Preorder traversal of a tree

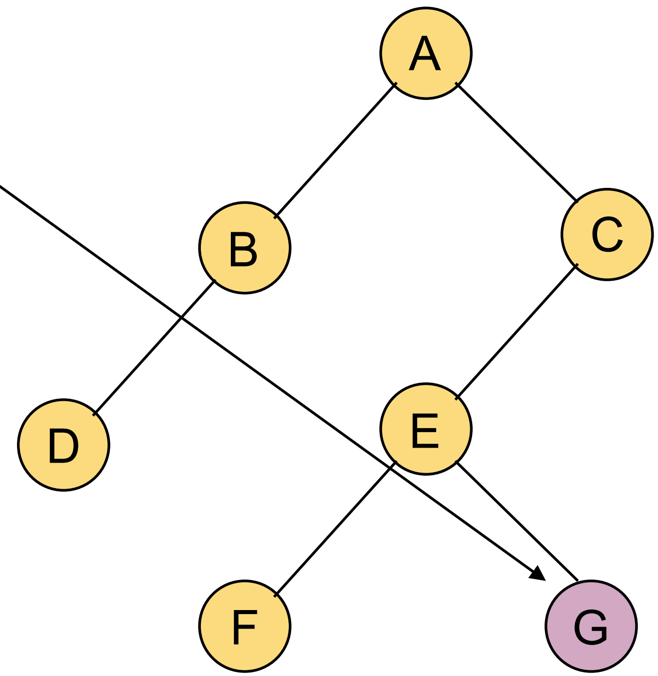
**call** preorder(*T*, *T*.root)

**Algorithm** preorder(*T*,*v*)

perform the action on the node *v*

**for** each child *w* of *v*

**call** preorder(*T*,*w*)



# Preorder traversal of a tree

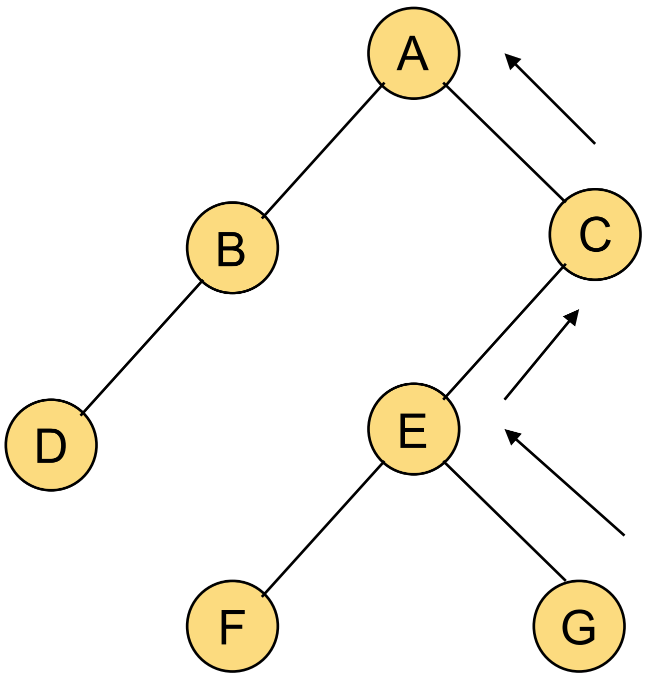
**call** preorder($T$, $T$.root)

**Algorithm** preorder($T$,$v$)

perform the action on the node $v$

**for** each child $w$ of $v$

**call** preorder($T$,$w$)



# Preorder traversal of a tree

- It is useful for producing a linear ordering of the nodes in a tree where parents are always before their children.
- If a document is represented as a tree, the preorder traversal examines the document sequentially.
- The overall running time of the preorder traversal is O($n$).

# Postorder traversal of a tree

- It is a complementary algorithm to preorder traversal, as it traverses recursively the subtrees rooted at the children of the root first before visiting the root.

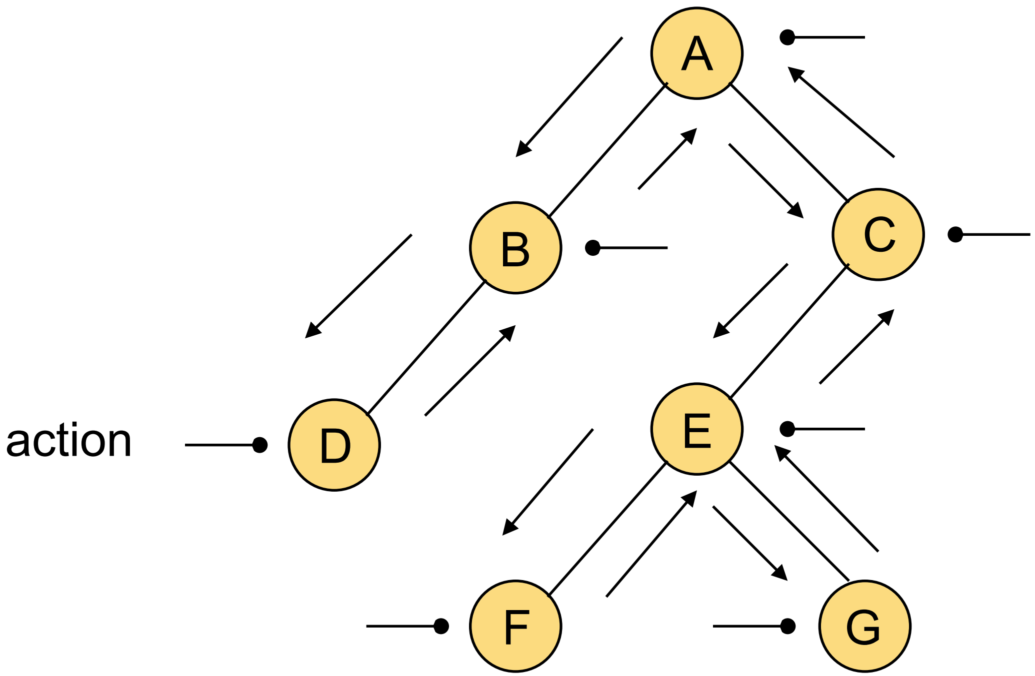
**Algorithm** postorder(*T,v*)

  **for** each child *w* of *v* **do**

    **call** postorder(*T,w*)

  perform the action on the node *v*

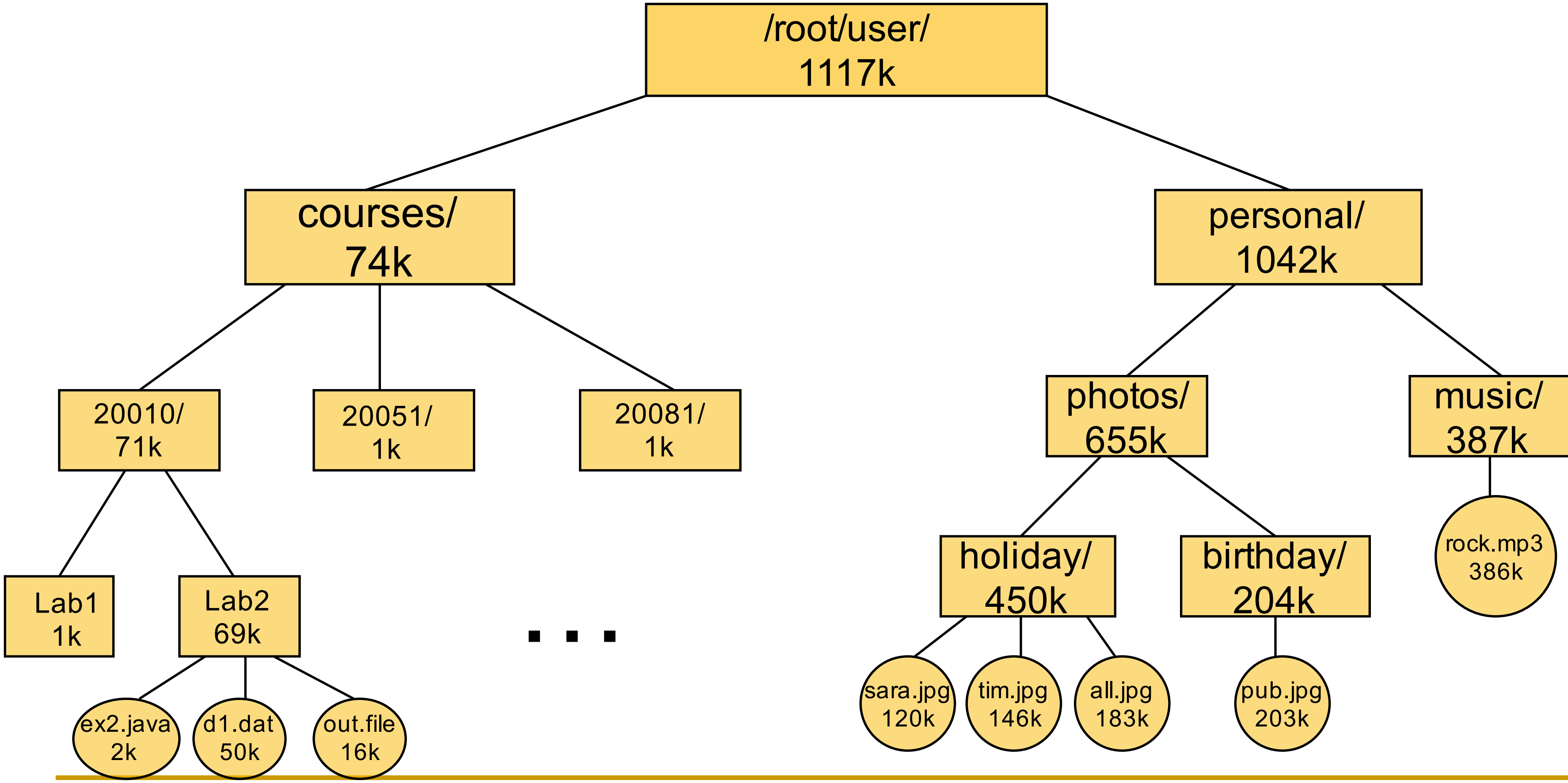
# Postorder traversal of a tree



# Postorder traversal of a tree

- The postorder traversal of a tree with *n* nodes takes O(*n*) time, assuming that visiting each node takes O(1) time.
- The algorithm is useful if computing a certain property of a node in a tree requires that this property is previously computed for all its children.

# Postorder traversal of a tree



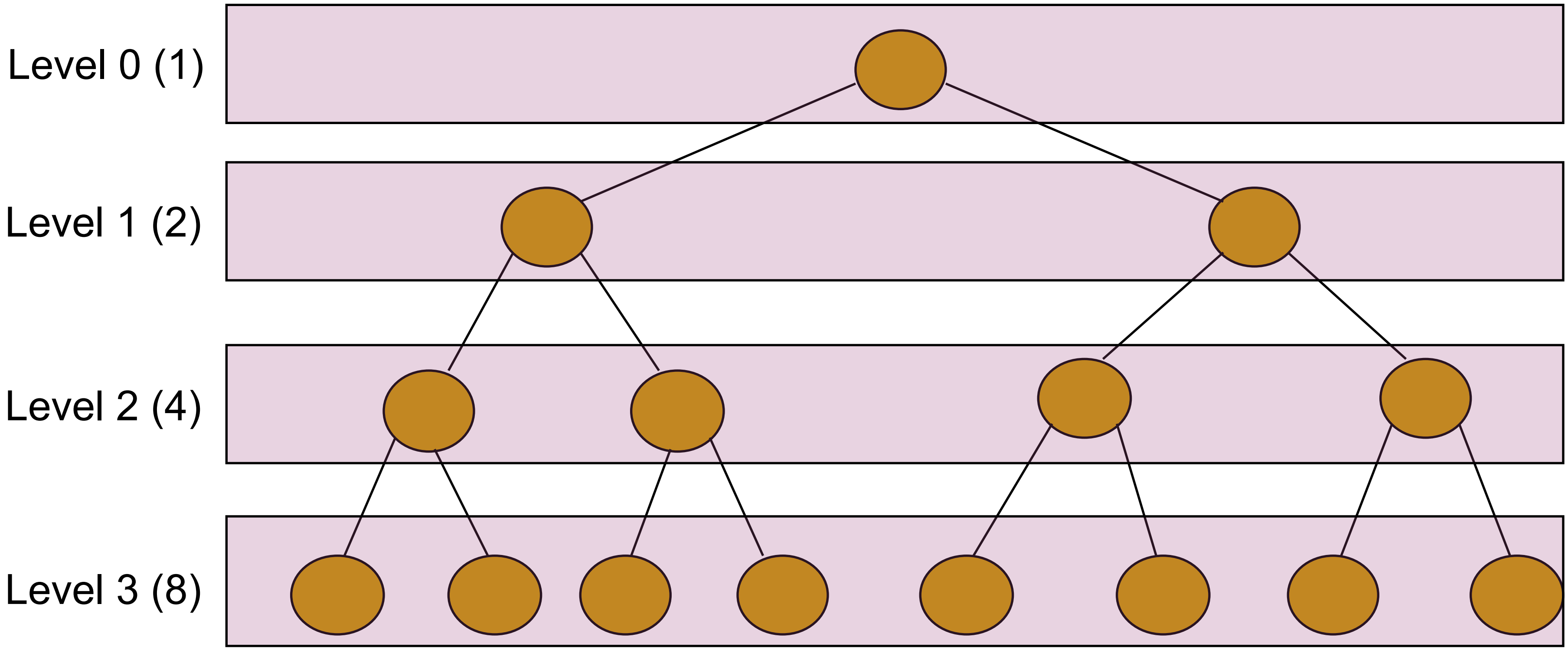
# Binary trees

- A proper binary tree is an ordered tree in which each internal node has exactly two children.
- As an ADT, a binary tree supports 3 additional accessor methods:
  - leftChild(*v*) – returns the left child of *v*; if *v* is an external node, an error occurs;
  - rightChild(*v*) – returns the right child of *v*; if *v* is an external node, an error occurs;
  - sibling(*v*) – returns the sibling of *v*; an error occurs if *v* is the root;

# Binary trees

- Denote all the nodes of a binary tree *T* at the same depth *d* as the level *d* of *T*;
- Level 0 has 1 node (the root), level 1 has at most 2 nodes, etc. In general, level *d* has at most $2^d$ nodes;
- In a proper binary tree the number of external nodes is 1 more than the number of internal nodes;

# Binary trees



# Preorder traversal of a binary tree

**Algorithm** binaryPreorder(*T,v*)

perform the action on the node *v*

**if** *v* is an internal node **then**

**call** binaryPreorder(*T,T.*leftChild(*v*))

**call** binaryPreorder(*T,T.*rightChild(*v*))

# Postorder traversal of a binary tree

**Algorithm** binaryPostorder(*T,v*)

  **if** *v* is an internal node **then**

    **call** binaryPostorder(*T,T.*leftChild(*v*))

    **call** binaryPostorder(*T,T.*rightChild(*v*))

  perform the action on the node *v*

# Inorder traversal of a binary tree

- In this method the action on a node *v* is performed in between the recursive traversals on its left and right subtrees.
- The inorder traversal of a binary tree can be viewed as visiting the nodes from left to right.

**Algorithm** binaryInorder(*T,v*)
  **if** *v* is an internal node **then**
    **call** binaryInorder(*T,T.*leftChild(*v*))
  perform the action on the node *v*
  **if** *v* is an internal node **then**
    **call** binaryInorder(*T,T.*leftChild(*v*))