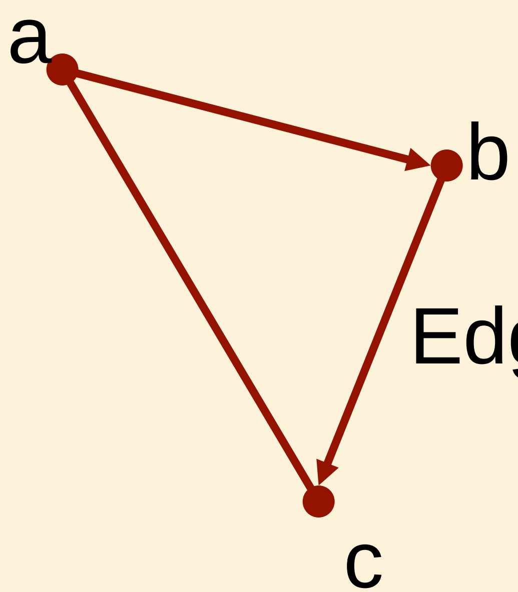
# Trees

## Chapter 7

# Graph



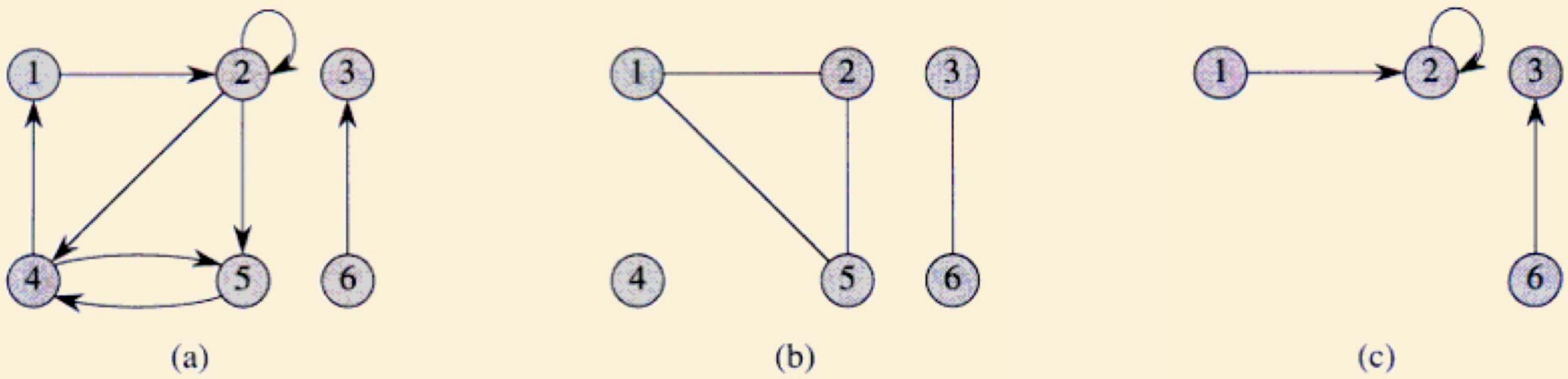

Node ~ city or computer

Edge ~ road or data cable

Undirected or Directed

A surprisingly large number of computational problems can be expressed as graph problems.

# Directed and Undirected Graphs

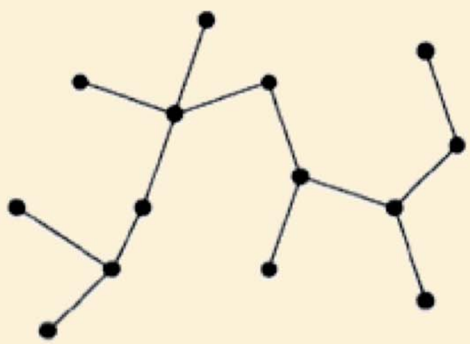



(a) A directed graph $G = (V, E)$, where $V = \{1,2,3,4,5,6\}$ and $E = \{(1,2), (2,2), (2,4), (2,5), (4,1), (4,5), (5,4), (6,3)\}$. The edge (2,2) is a self-loop.

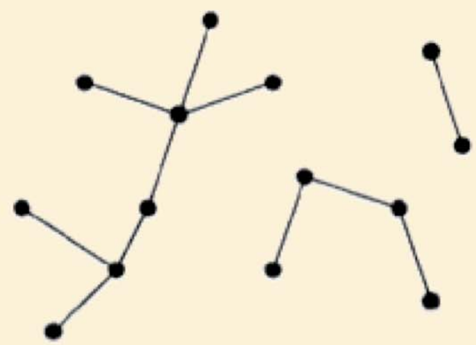
(b) An undirected graph $G = (V,E)$, where $V = \{1,2,3,4,5,6\}$ and $E = \{(1,2), (1,5), (2,5), (3,6)\}$. The vertex 4 is isolated.

(c) The subgraph of the graph in part (a) induced by the vertex set $\{1,2,3,6\}$.

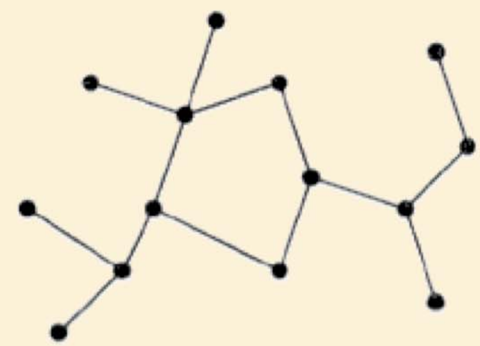
# Trees



Tree



Forest



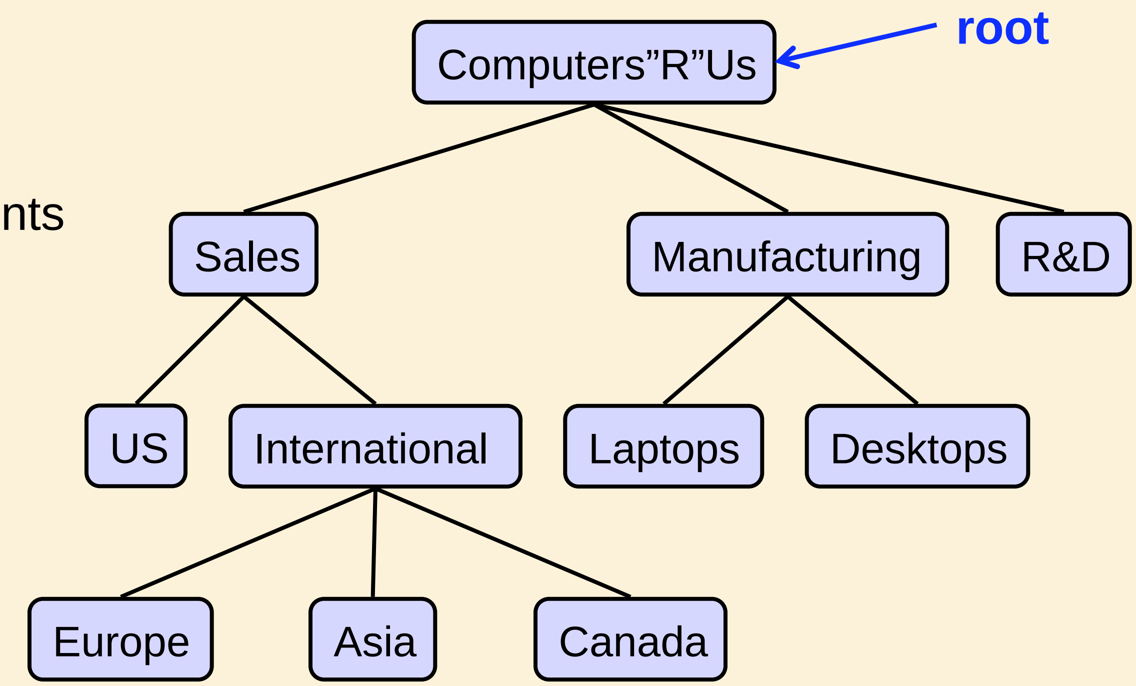
Graph with Cycle

A tree is a connected, acyclic, undirected graph.

A forest is a set of trees (not necessarily connected)

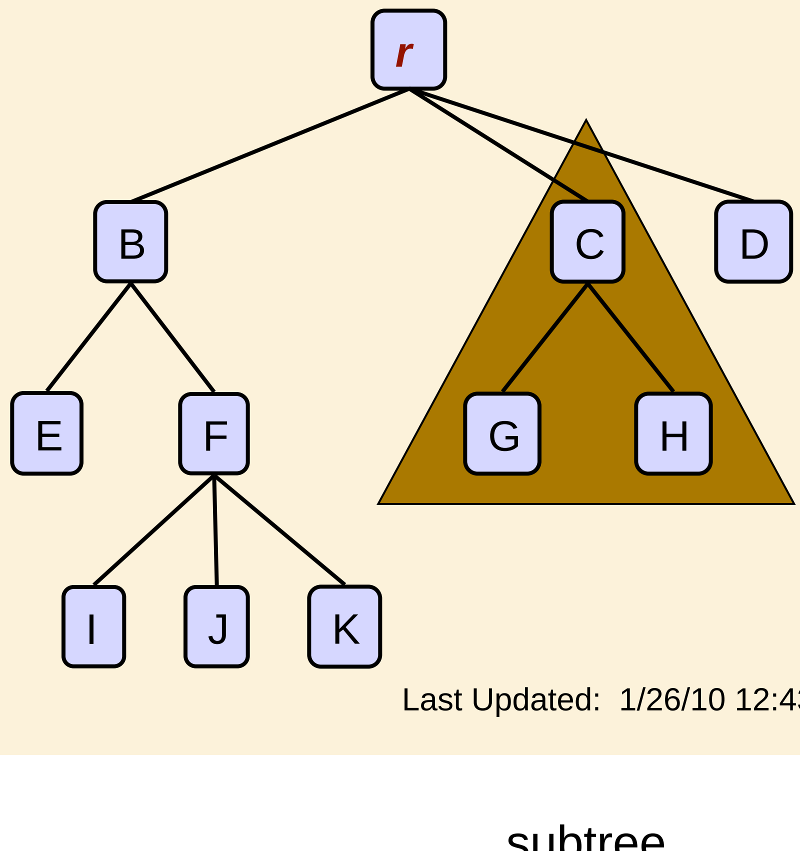
# Rooted Trees

- Trees are often used to represent hierarchical structure
- In this view, one of the vertices (nodes) of the tree is distinguished as the root.
- This induces a parent-child relationship between nodes of the tree.
- Applications:
  - Organization charts
  - File systems
  - Programming environments



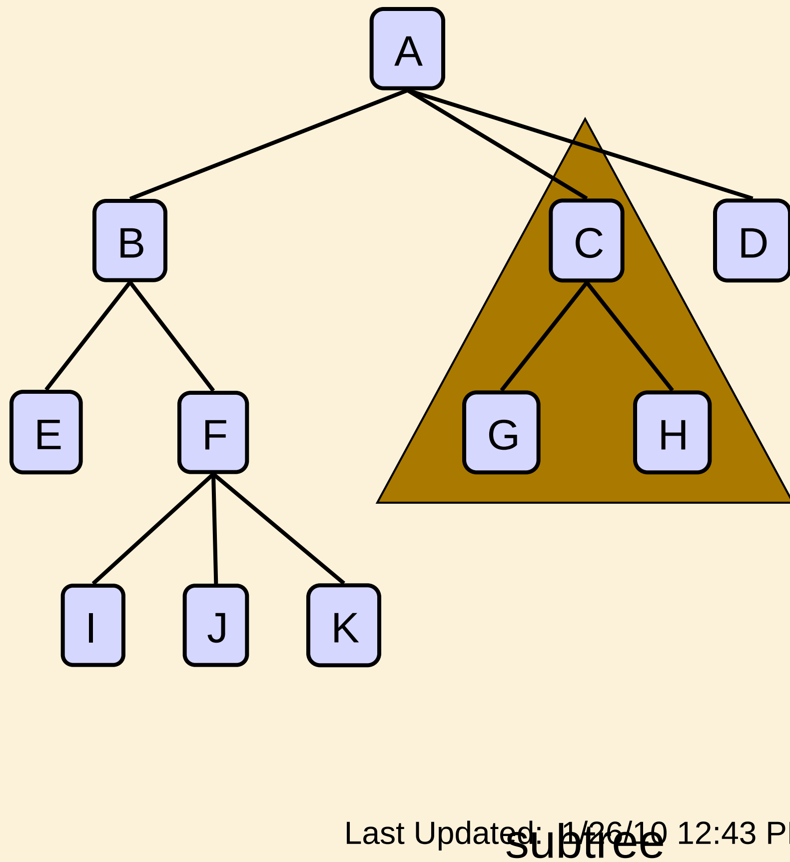
# Formal Definition of Rooted Tree

- A rooted tree may be empty.
- Otherwise, it consists of
  - A root node ***r***
  - A set of **subtrees** whose roots are the children of ***r***



# 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, grand-grandparent, etc.
- **Descendant of a node:** child, grandchild, grand-grandchild, etc.
- **Siblings**: two nodes having the same parent
- **Depth of a node:** number of ancestors
- **Height of a tree:** maximum depth of any node (3)
- **Subtree:** tree consisting of a node and its descendants



# Position ADT

- The Position ADT models the notion of place within a data structure where a single object is stored
- It gives a unified view of diverse ways of storing data, such as
  - a cell of an array
  - a node of a linked list
  - a node of a tree
- Just one method:
  - object element(): returns the element stored at the position

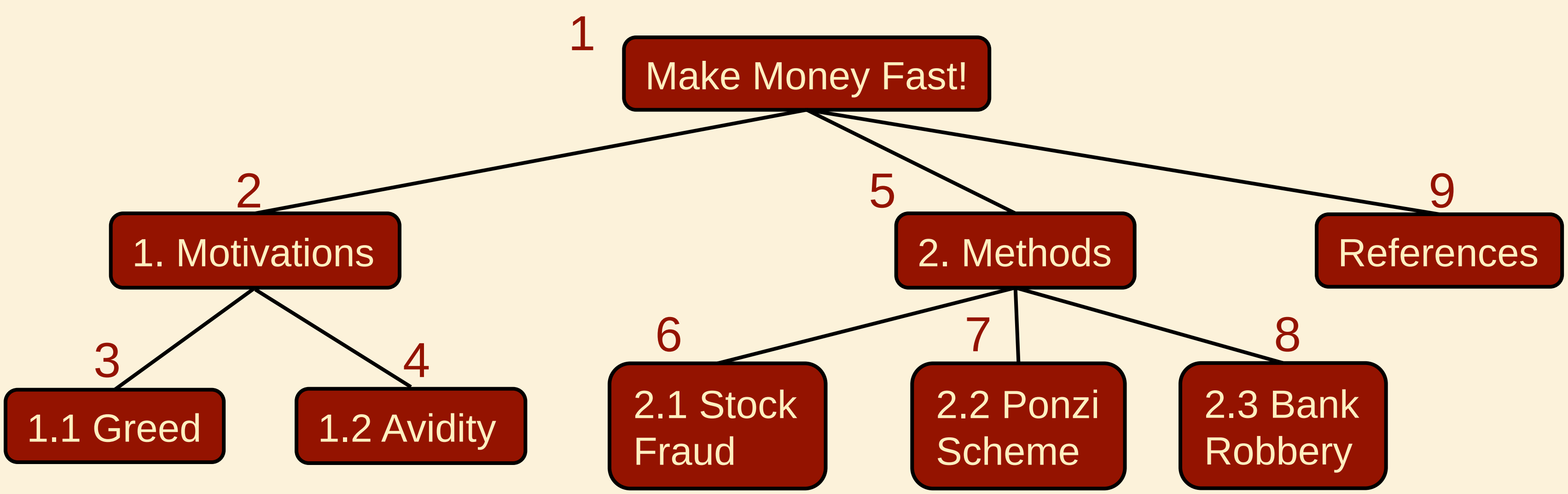
# Tree ADT

- We use positions to abstract nodes
- Generic methods:
  - integer size()
  - boolean isEmpty()
  - Iterator iterator()
  - Iterable positions()
- Accessor methods:
  - position root()
  - position parent(p)
  - positionIterator children(p)
- Query methods:
  - boolean isInternal(p)
  - boolean isExternal(p)
  - boolean isRoot(p)
- Update method:
  - object replace(p, o)
  - Additional update methods may be defined by data structures implementing the Tree ADT

# 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

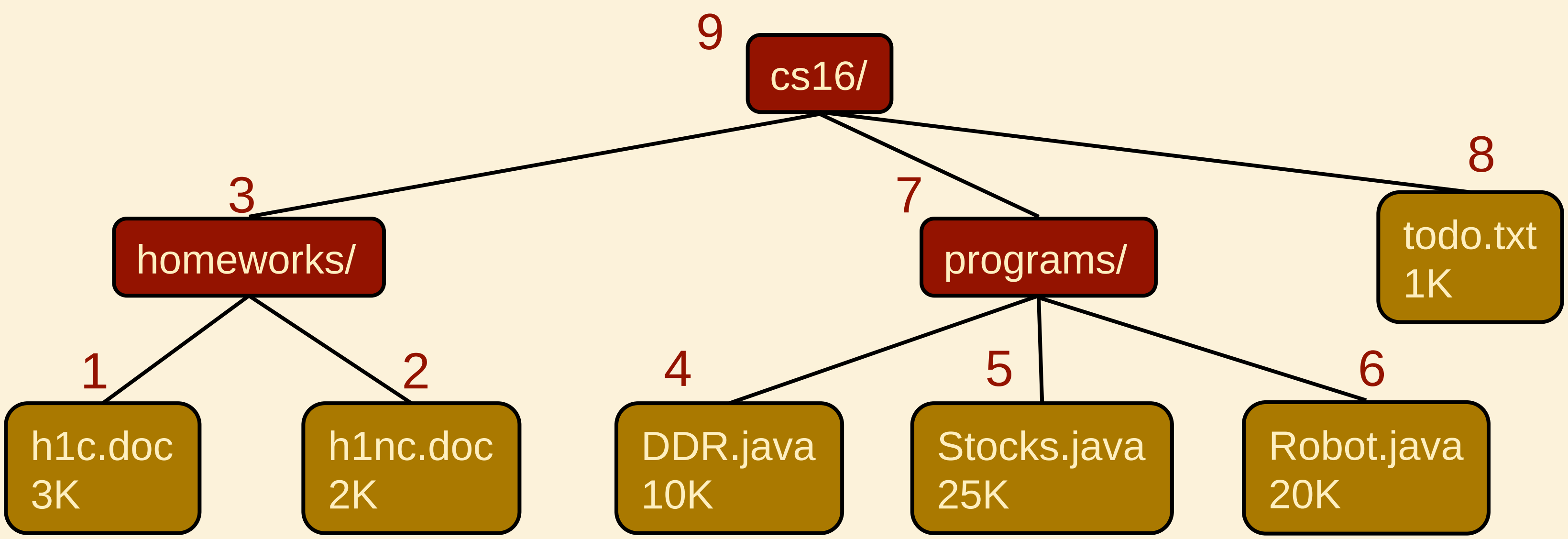
**Algorithm** ***preOrder***(***v***)
    ***visit***(***v***)
    **for each** child ***w*** of ***v***
        ***preOrder*** (***w***)



# Postorder Traversal

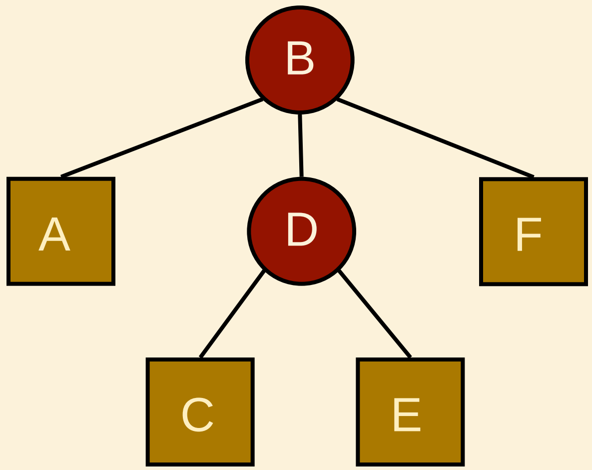
- In a postorder traversal, a node is visited after its descendants

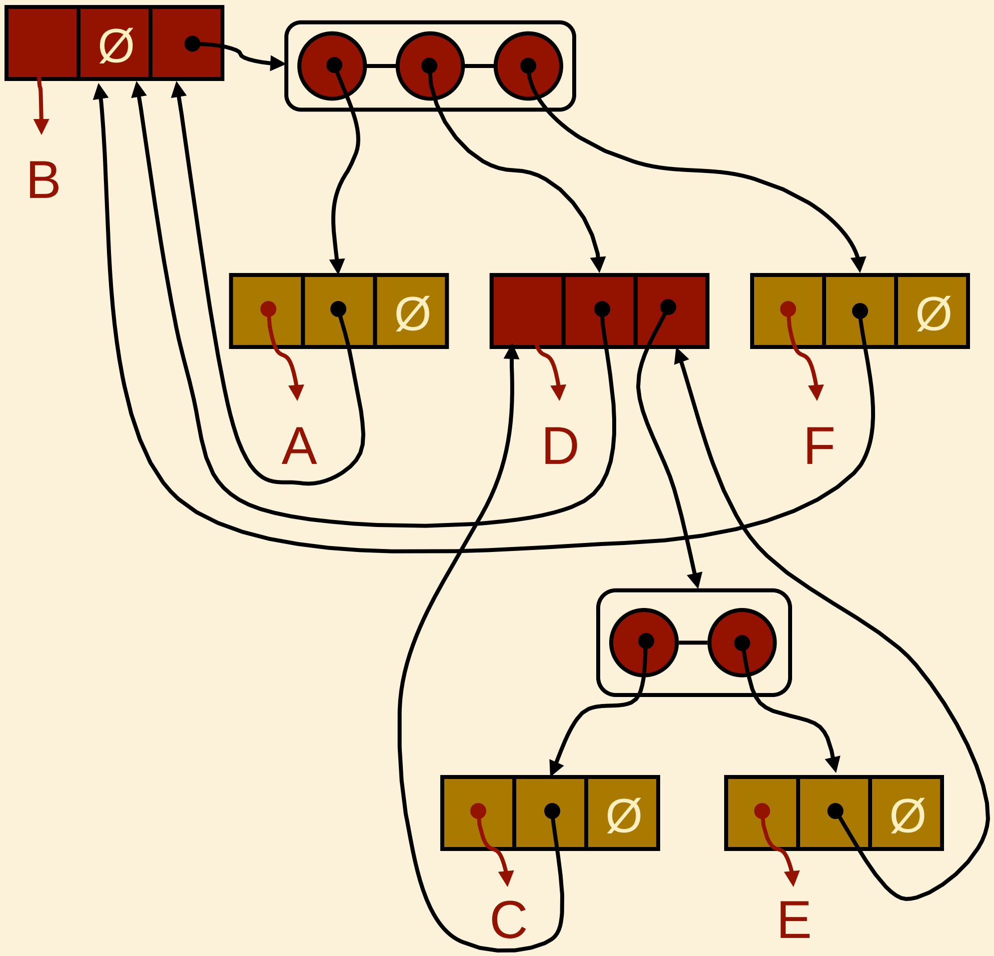
**Algorithm** ***postOrder***(***v***)
    **for each** child ***w*** of ***v***
        ***postOrder*** (***w***)
    ***visit***(***v***)



# Linked Structure for Trees

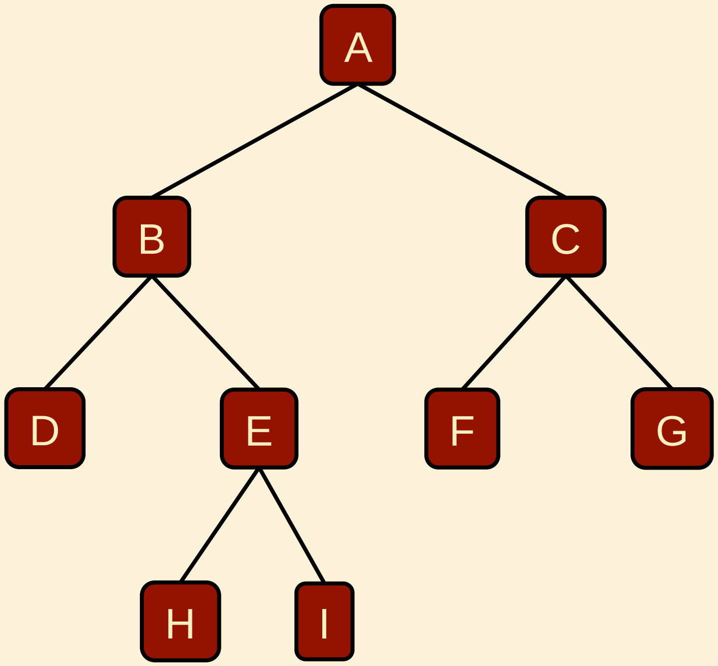
- A node is represented by an object storing
  - Element
  - Parent node
  - Sequence of children nodes
- Node objects implement the Position ADT



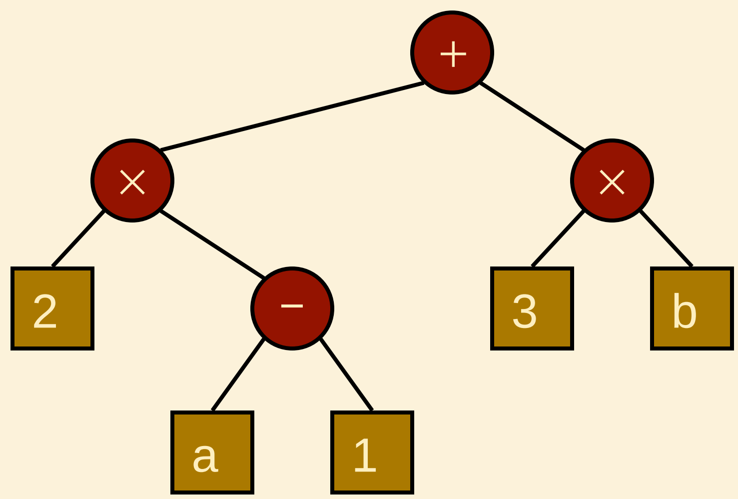
# Binary Trees

- 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**
- Applications:
  - arithmetic expressions
  - decision processes
  - searching



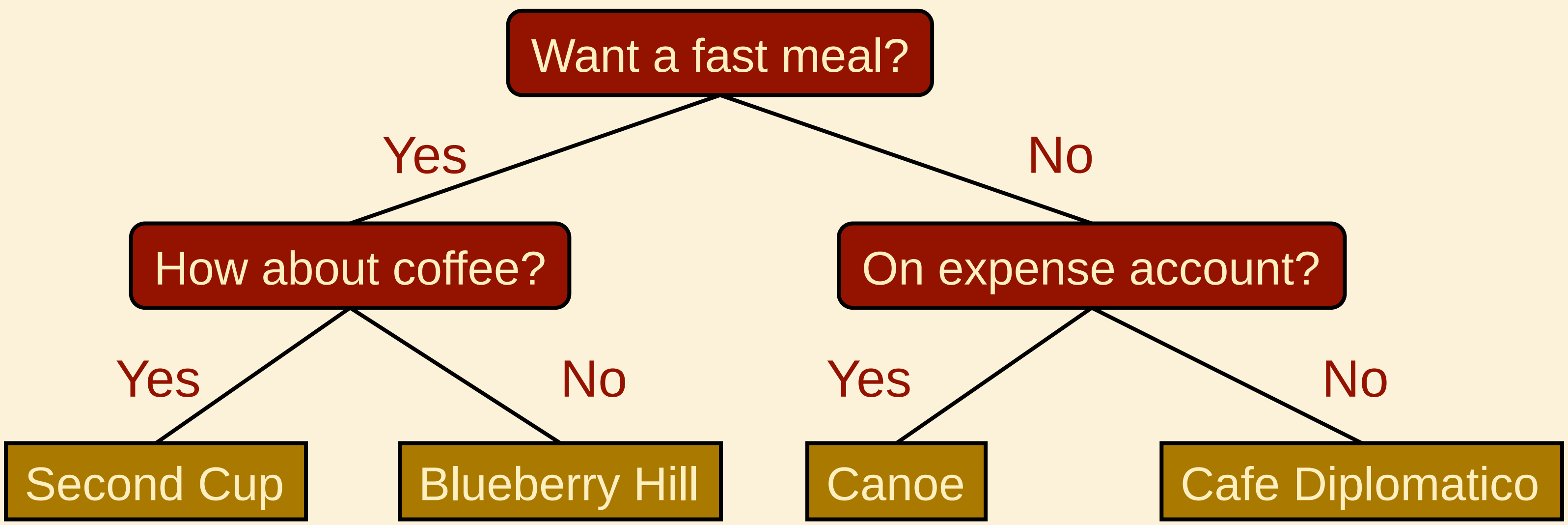
# Arithmetic Expression Tree

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



# Decision Tree

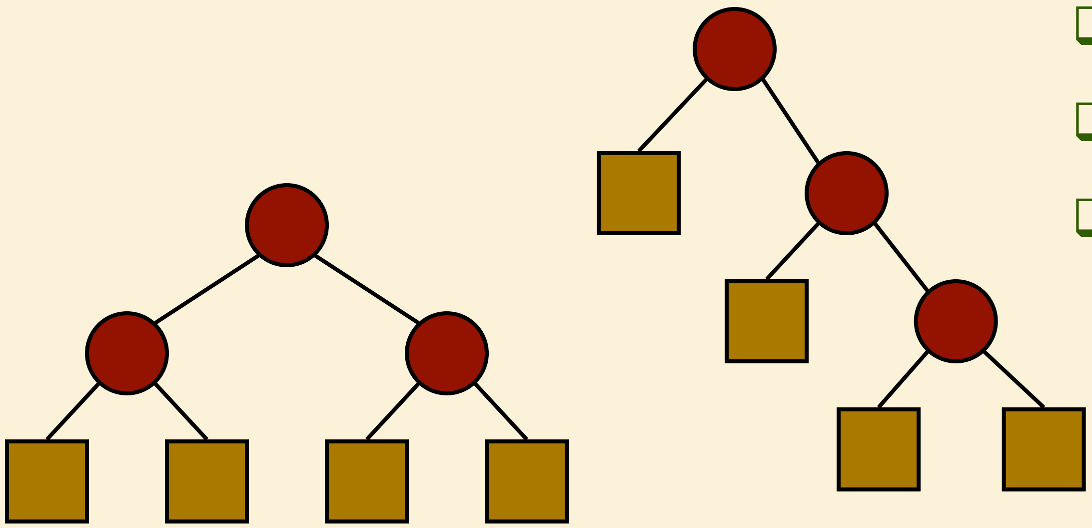
- Binary tree associated with a decision process
  - internal nodes: questions with yes/no answer
  - external nodes: decisions
- Example: dining decision



# Properties of Proper Binary Trees

- Notation
  - ***n*** number of nodes
  - ***e*** number of external nodes
  - ***i*** number of internal nodes
  - ***h*** height

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

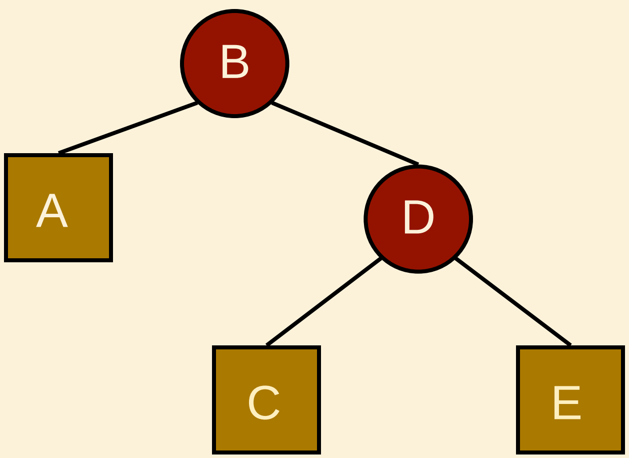


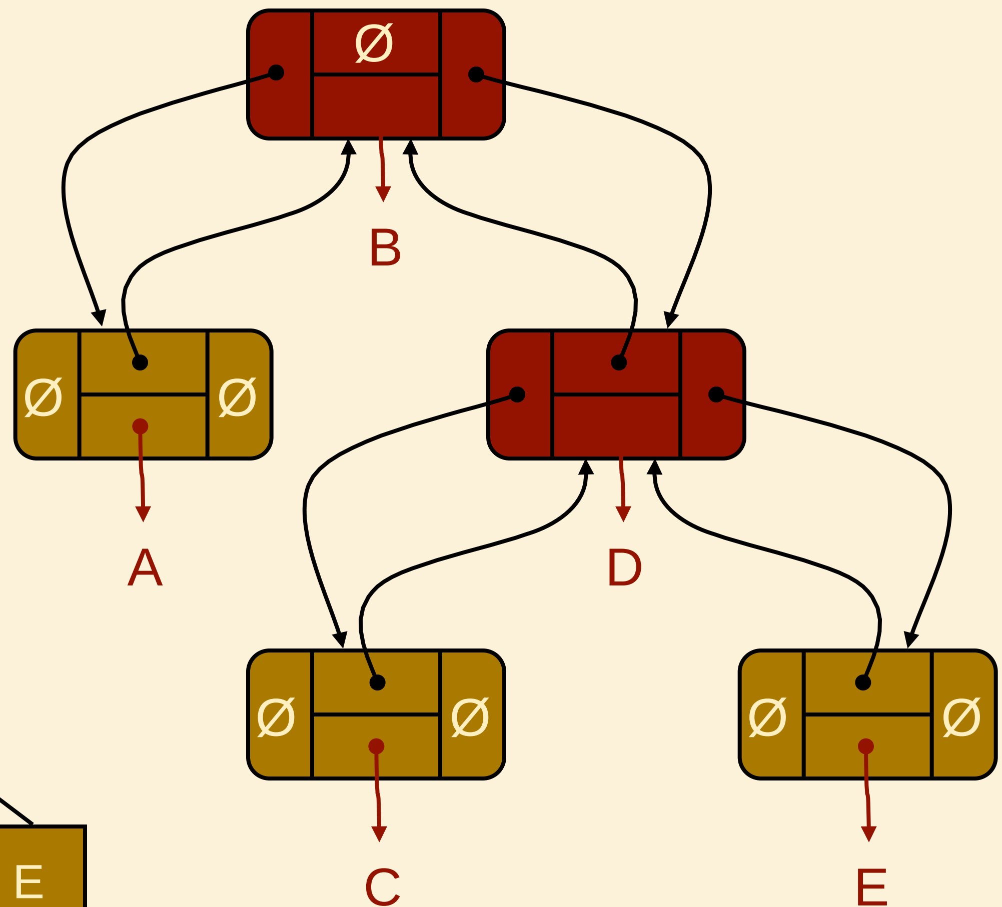
# BinaryTree ADT

- 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)
  - boolean **hasLeft**(p)
  - boolean **hasRight**(p)
- Update methods may be defined by data structures implementing the BinaryTree ADT

# Linked Structure for Binary Trees

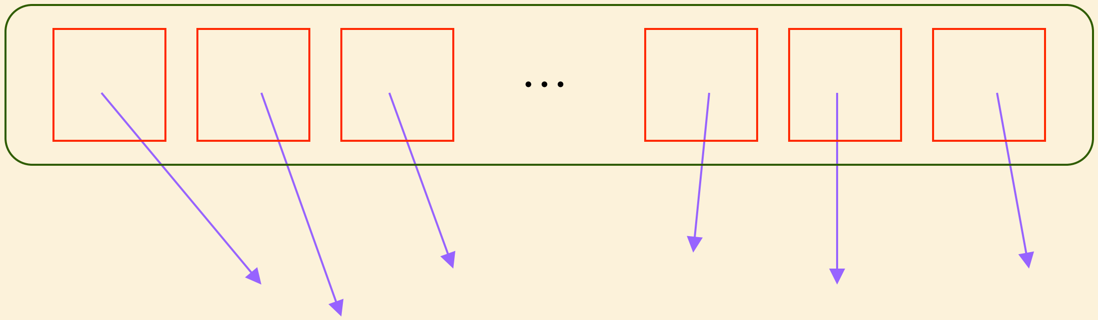
- A node is represented by an object storing
  - Element
  - Parent node
  - Left child node
  - Right child node
- Node objects implement the Position ADT

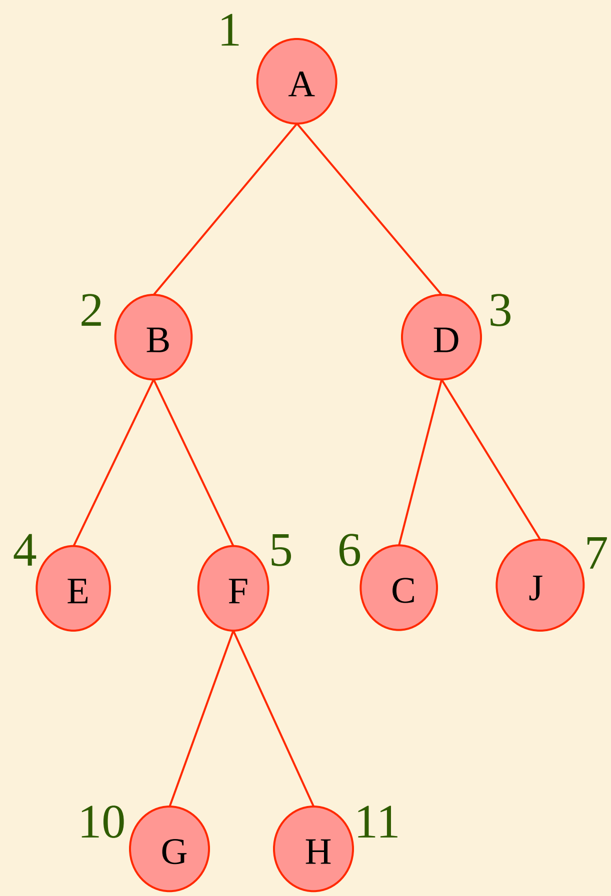


# Array-Based Representation of Binary Trees

- nodes are stored in an array, using a **level-numbering** scheme.



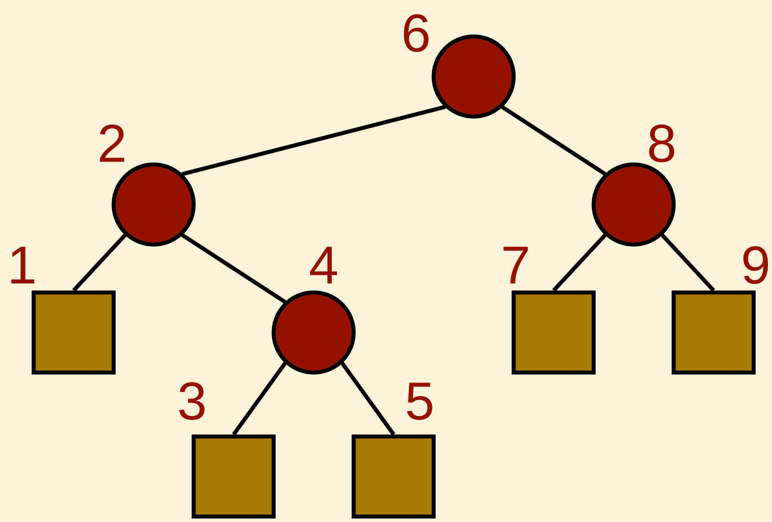




- let rank(node) be defined as follows:
  - rank(root) = 1
  - if node is the **left** child of parent(node),
    rank(node) = **2*rank(parent(node))**
  - if node is the **right** child of parent(node),
    rank(node) = **2*rank(parent(node))+1**

# Inorder Traversal of Binary Trees

- 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** ***hasLeft*** **(*v*)**
        ***inOrder*** **(*left*** **(*v*))**
    ***visit*(*v*)**
    **if** ***hasRight*** **(*v*)**
        ***inOrder*** **(*right*** **(*v*))**

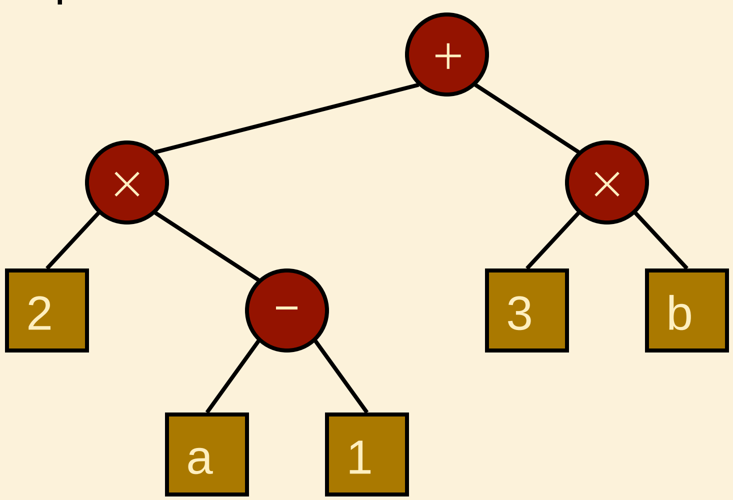


# Print Arithmetic Expressions

- 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 *hasLeft* (*v*)
        *print*("(")
        *inOrder* (*left*(*v*))
    *print*(*v.element* ())
    if *hasRight* (*v*)
        *inOrder* (*right*(*v*))
        *print* (")")

Input:

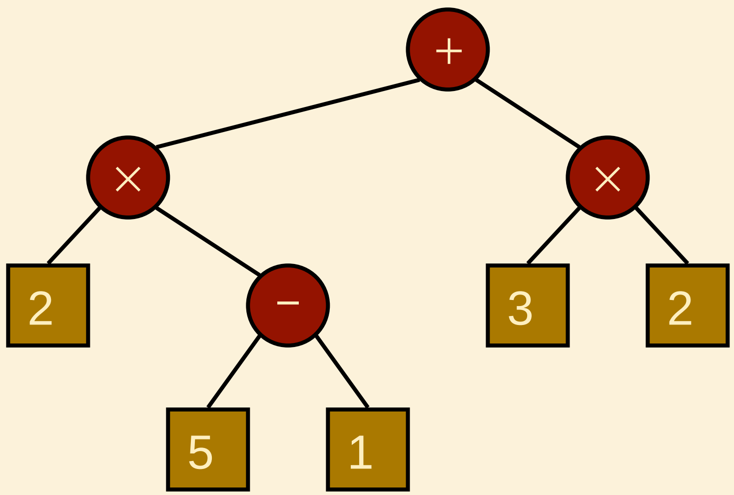



Output:

((2 × (a − 1)) + (3 × b))

# Evaluate Arithmetic Expressions

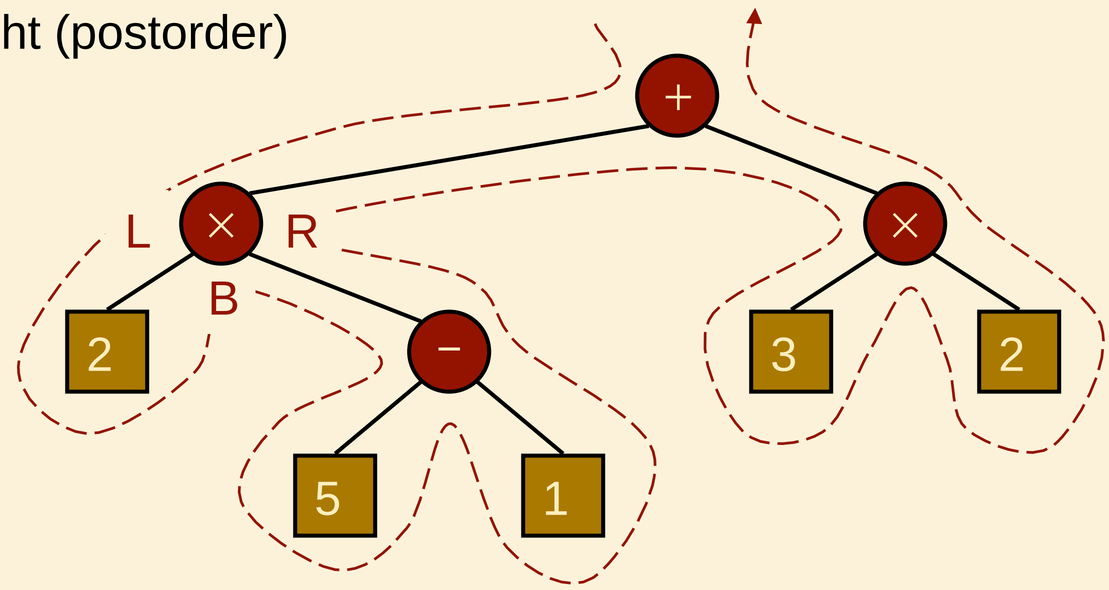
- Specialization of a postorder traversal
  - recursive method returning the value of a subtree
  - when visiting an internal node, combine the values of the subtrees




Algorithm *evalExpr*(*v*)
    if *isExternal* (*v*)
        return *v.element* ()
    else
        *x* ← *evalExpr* (*leftChild* (*v*))
        *y* ← *evalExpr* (*rightChild* (*v*))
        ◆ ← operator stored at *v*
        return *x* ◆ *y*

# Euler Tour Traversal

- Generic traversal of a binary tree
- Includes as special cases the preorder, postorder and inorder traversals
- Walk around the tree and visit each node three times:
  - on the left (preorder)
  - from below (inorder)
  - on the right (postorder)



# Template Method Pattern

- Generic algorithm that can be specialized by redefining certain steps
- Implemented by means of an abstract Java class
- Visit methods that can be redefined by subclasses
- Template method eulerTour
  - Recursively called on the left and right children
  - A Result object with fields leftResult, rightResult and finalResult keeps track of the output of the recursive calls to eulerTour

```
public abstract class EulerTour {
    protected BinaryTree tree;
    protected void visitExternal(Position p, Result r) { }
    protected void visitLeft(Position p, Result r) { }
    protected void visitBelow(Position p, Result r) { }
    protected void visitRight(Position p, Result r) { }
    protected Object eulerTour(Position p) {
        Result r = new Result();
        if tree.isExternal(p) { visitExternal(p, r); }
            else {
                visitLeft(p, r);
                r.leftResult = eulerTour(tree.left(p));
                visitBelow(p, r);
                r.rightResult = eulerTour(tree.right(p));
                visitRight(p, r);
                return r.finalResult;
            } …
```

# Specializations of EulerTour

- We show how to specialize class EulerTour to evaluate an arithmetic expression
- Assumptions
  - External nodes store Integer objects
  - Internal nodes store Operator objects supporting method

    operation (Integer, Integer)

```
public class EvaluateExpression
                        extends EulerTour {
    protected void visitExternal(Position p, Result r) {
        r.finalResult = (Integer) p.element();
    }
    protected void visitRight(Position p, Result r) {
        Operator op = (Operator) p.element();
        r.finalResult = op.operation(
                            (Integer) r.leftResult,
                            (Integer) r.rightResult
                            );
    }
    …
}
```