Trees

Outline

- Definitions
- Traversing trees
- Binary trees

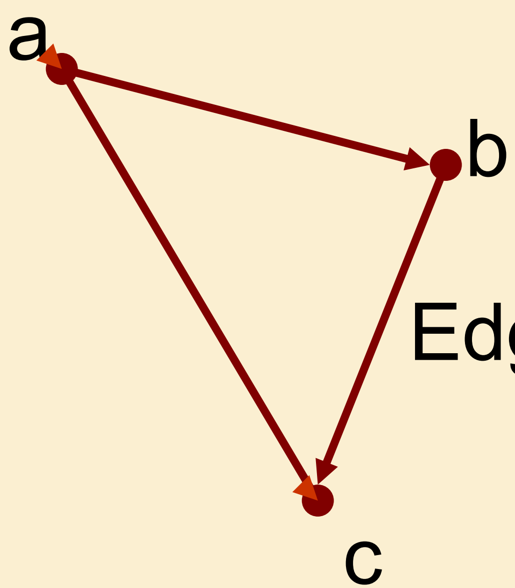
Outcomes

- By understanding this lecture you should be able to:
  - Correctly use terminology associated with trees
  - Explain the purpose of the Position ADT
  - Understand and design ADTs for trees
  - Explain the difference between the 3 common types of tree traversal, and when each might be used
  - Explain what makes a tree a binary tree, and give example applications of binary trees.
  - Implement trees using linked nodes
  - Implement binary trees using arrays.
  - Explain the advantages and disadvantages of linked node and array implementations of binary trees

Outline

- **Definitions**
- Traversing trees
- Binary trees

Graph



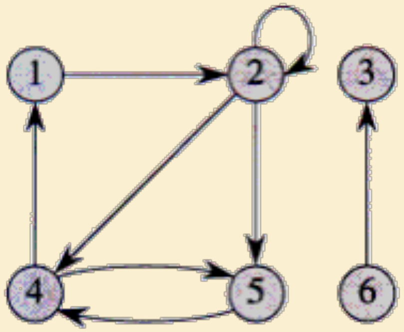
Node ~ city or computer

Edge ~ road or data cable

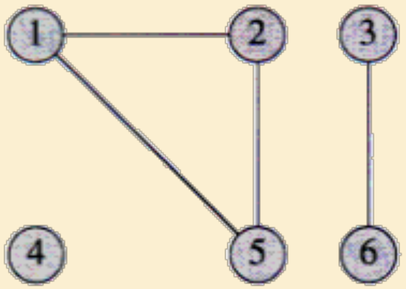
Undirected or Directed

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

Directed and Undirected Graphs



(a)

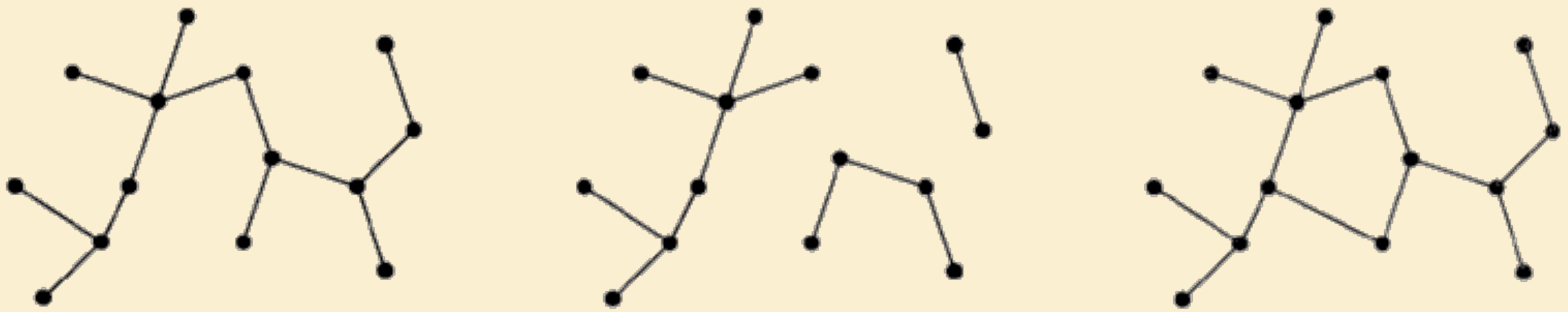


(b)

(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.

Trees



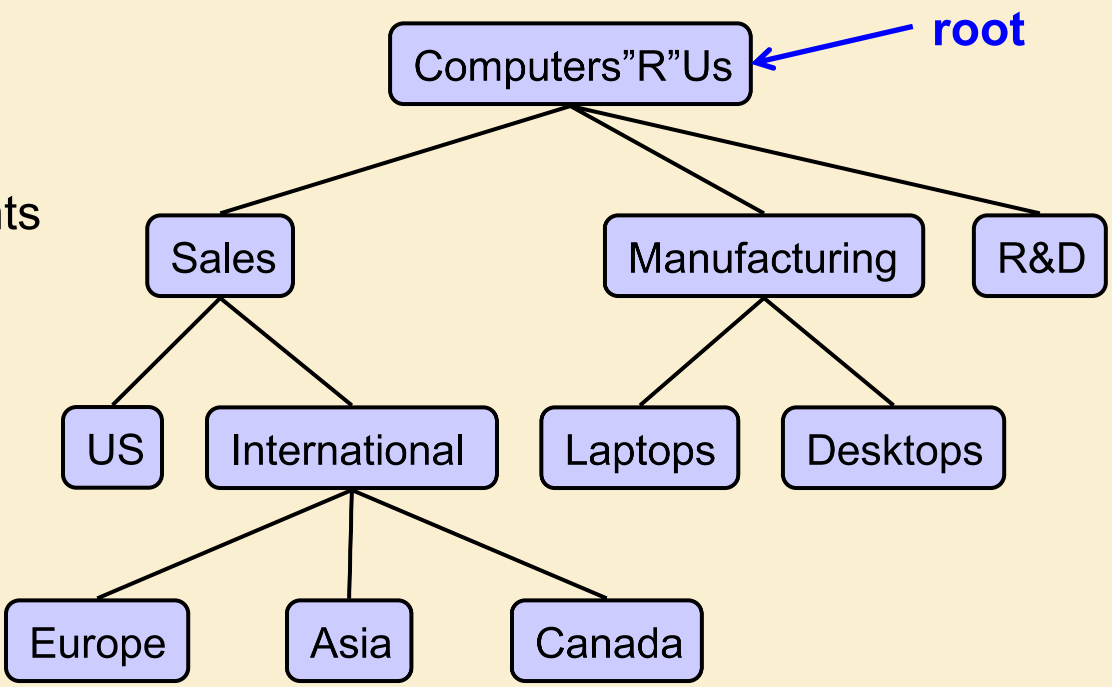
Tree Forest Graph with Cycle

A tree is a connected, acyclic, undirected graph.

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

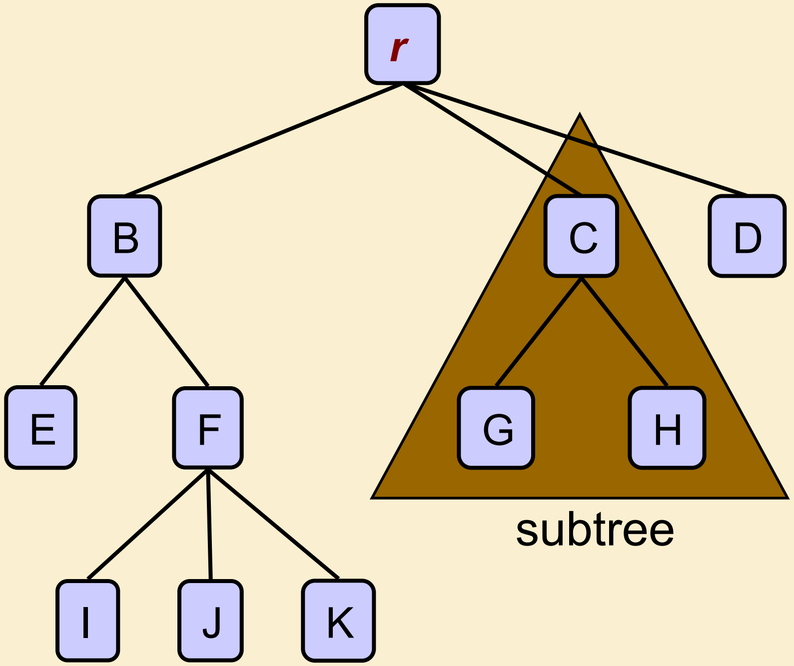
Rooted Trees

- 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



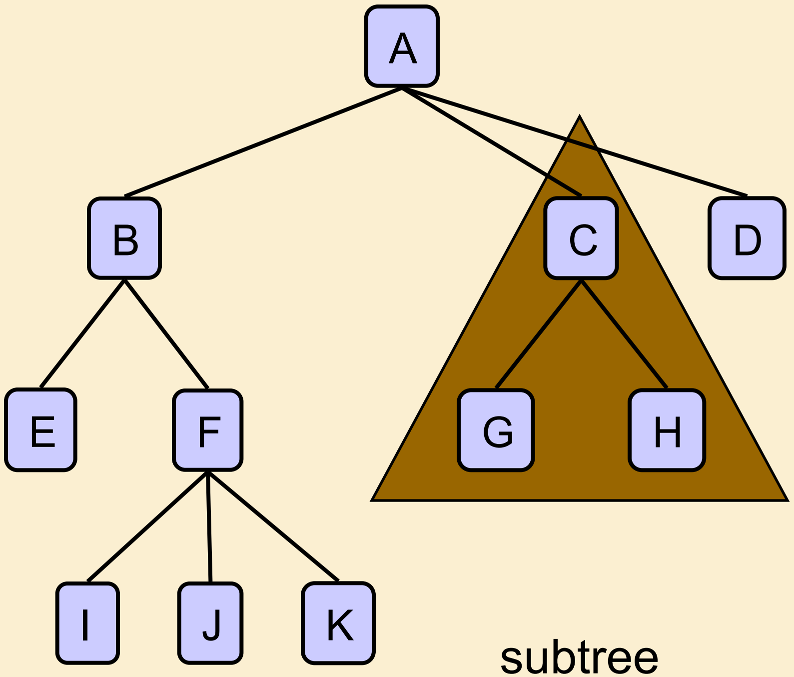
Formal Definition of Rooted Tree

- 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:** self, parent, grandparent, great-grandparent, etc.
  - NB: A node is considered an ancestor of itself!
- **Descendent of a node:** self, child, grandchild, great-grandchild, etc.
  - NB: A node is considered a descendent of itself!
- **Siblings**: two nodes having the same parent
- **Depth of a node:** number of ancestors (excluding the node itself)
- **Height of a tree:** maximum depth of any node (3)
- **Subtree:** tree consisting of a node and its descendents



End of Lecture

Jan 30, 2018

Outline

- Definitions
- **Traversing trees**
- Binary trees

Traversing Trees

- One of the basic operations we require is to be able to traverse over the nodes of a tree.
- To do this, we will make use of a **Position ADT**.

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 **p.getElement()**: returns the element stored at the position **p.**

Tree ADT

- We use positions to abstract the nodes of a tree.
- General 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)
- Update methods:
  - Deferred to specific implementations

Positions vs Elements

- Why have both
  - Iterator iterator()
  - Iterable positions()
- The iterator returned by iterator() provides a means for stepping through the elements stored by the tree.
- The positions() method returns a collection of the nodes of the tree.
- Each node includes the element but also the links connecting the node to its parent and its children.
- This allows you to move around the tree by following links to parents and children.

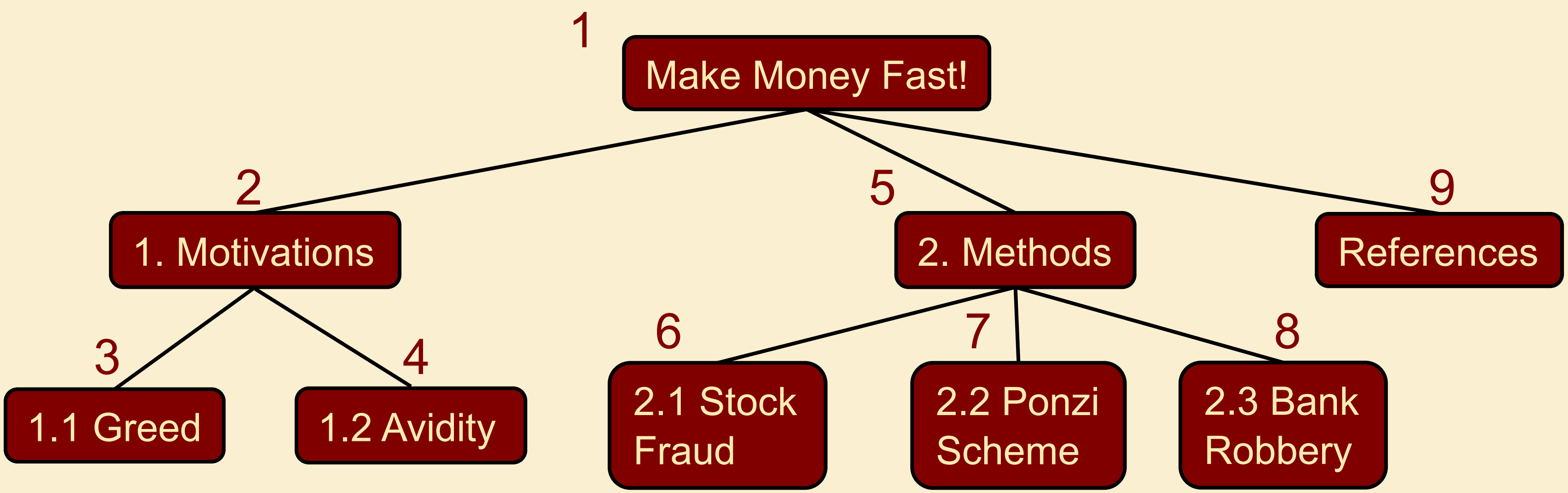
Recursion and Tree Traversals

- Traversing a tree is a natural application of **linear recursion**.
- It is possible to implement iterative traversal algorithms with the same asymptotic run time, but they are a lot more complicated!

Preorder Traversal

- A traversal visits the nodes of a tree in a systematic manner
- Each time a node is visited, an action may be performed.
- Thus the order in which the nodes are visited is important.
- In a preorder traversal, a node is visited before its descendants

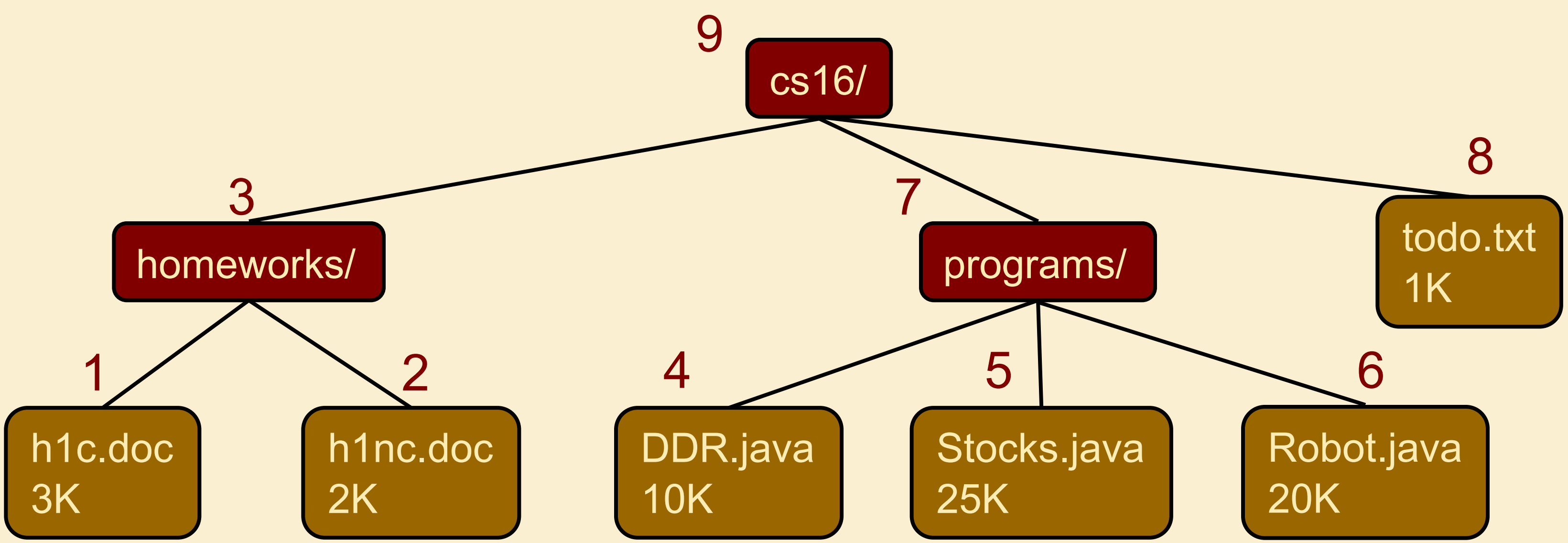
Algorithm ***preOrder***(***v***)
  visit(***v***)
  for each child ***w*** of ***v***
    preOrder (***w***)



Postorder Traversal

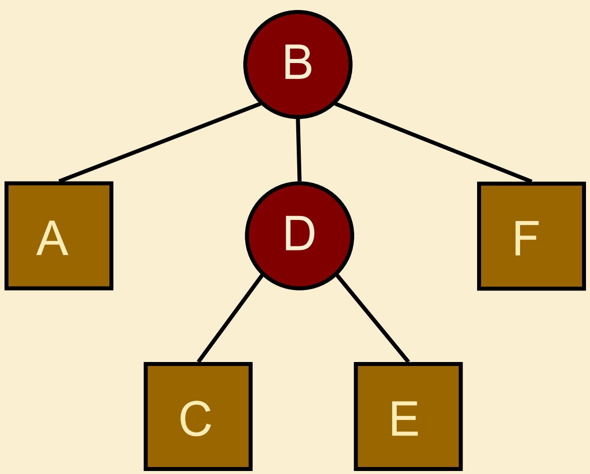
- In a postorder traversal, a node is visited after its descendants

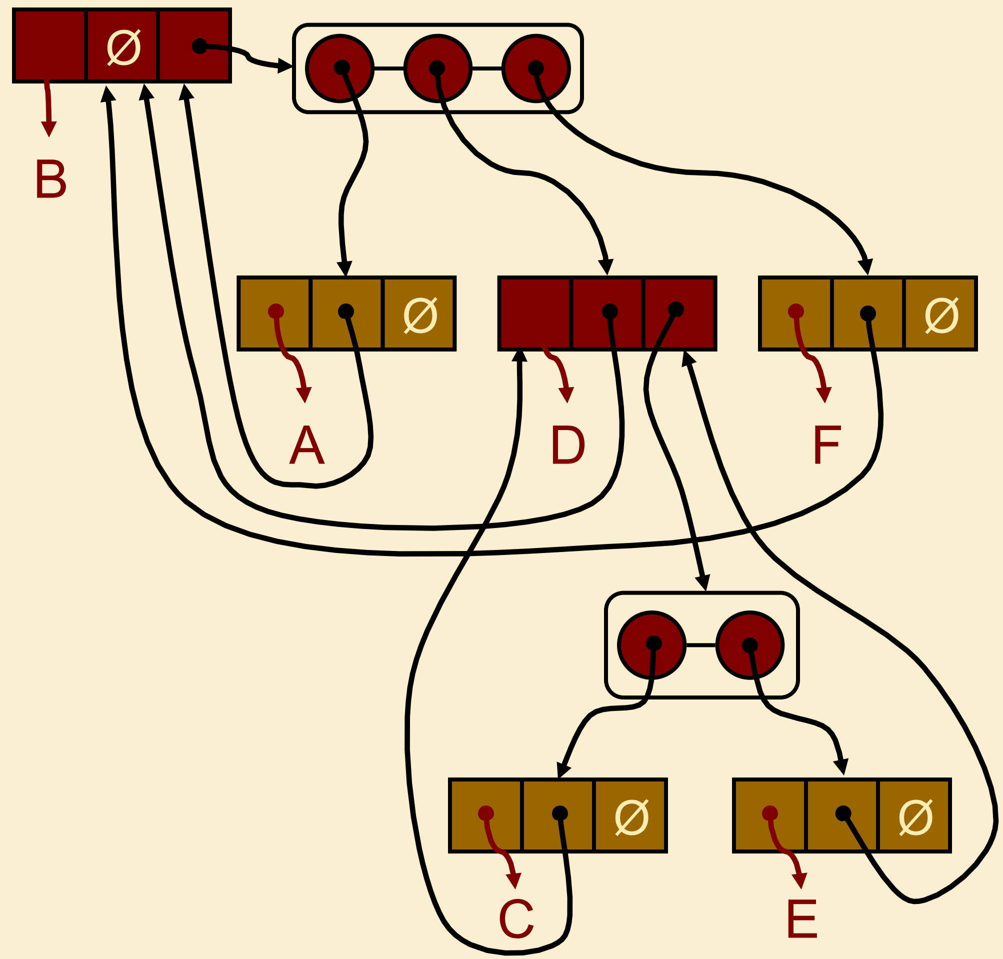
```
Algorithm postOrder(v)
    for each child w of v
        postOrder (w)
    visit(v)
```



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



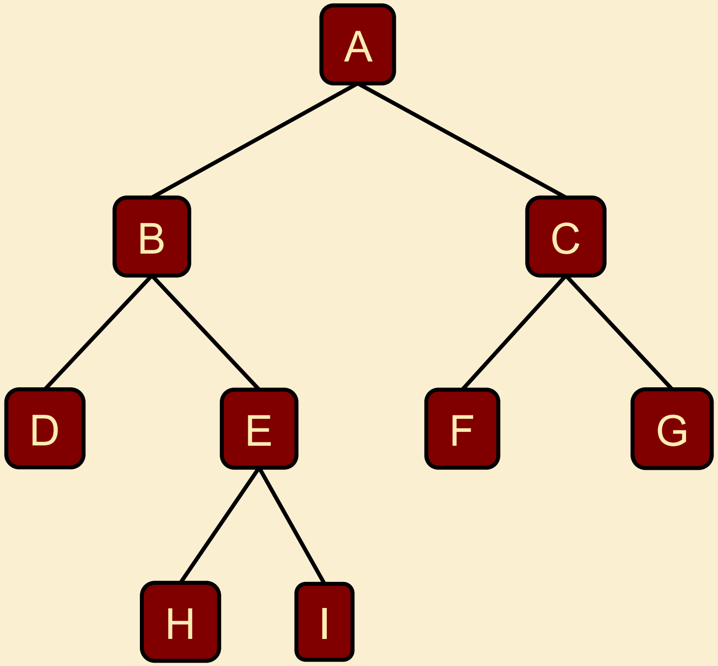


Outline

- Definitions
- Traversing trees
- **Binary trees**

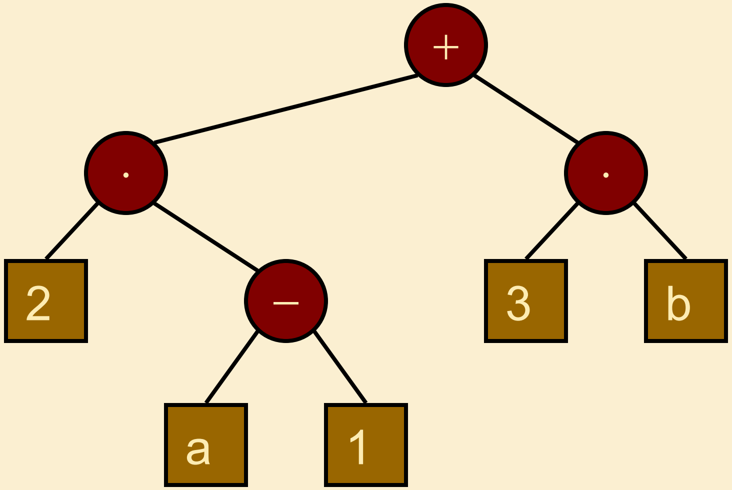
Binary Trees

- 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



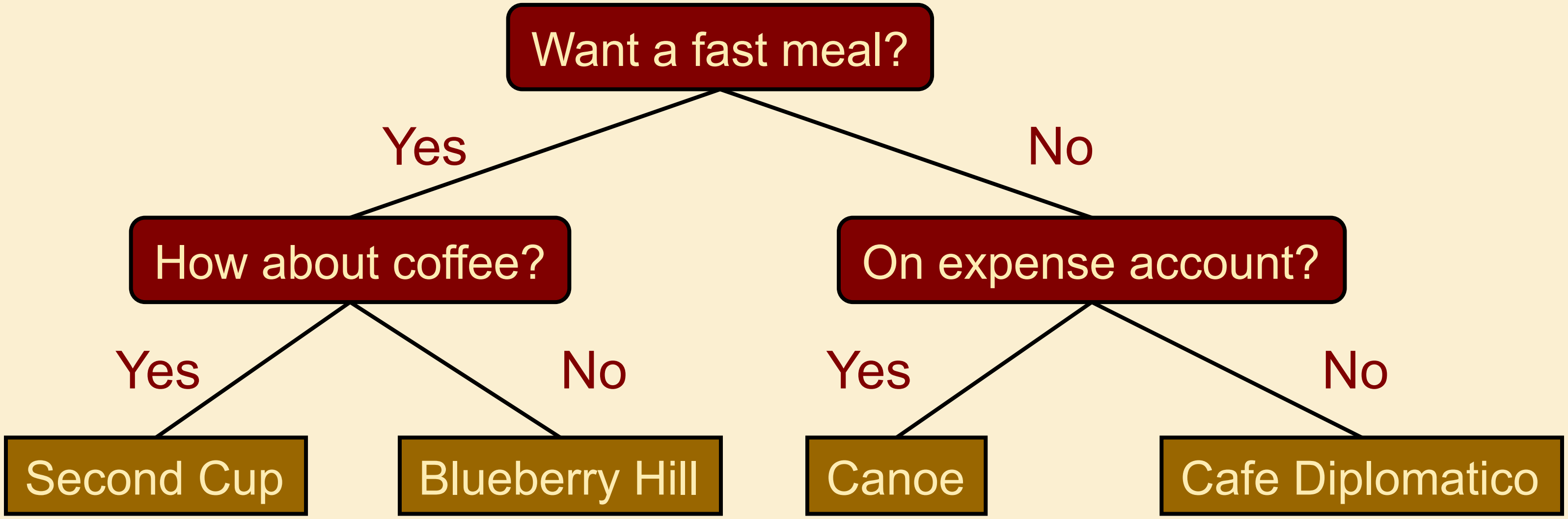
Arithmetic Expression Tree

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



Decision Tree

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



Proper Binary Trees

- A binary tree is said to be **proper** if each node has either 0 or 2 children.

Properties of Proper Binary Trees

- Notation

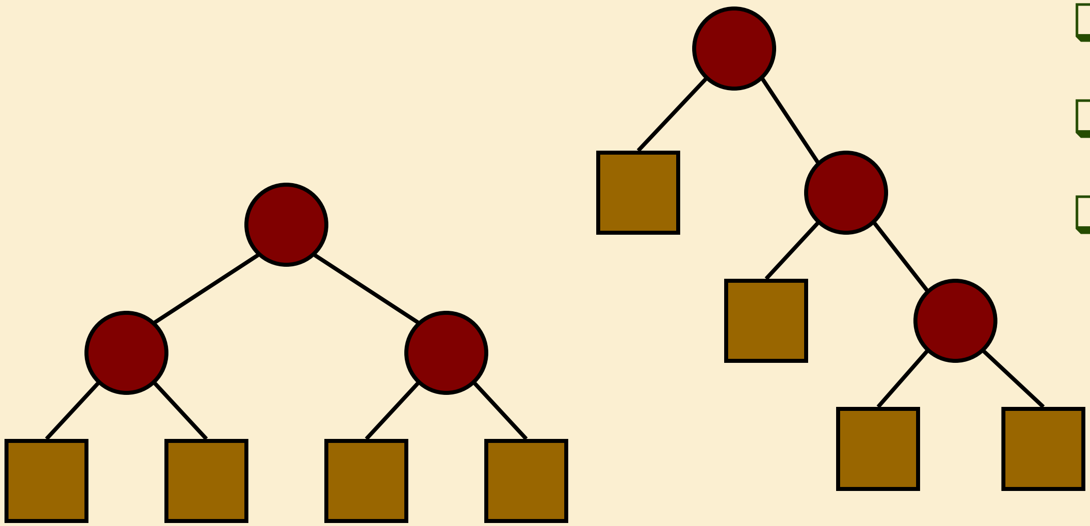
  n number of nodes

  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

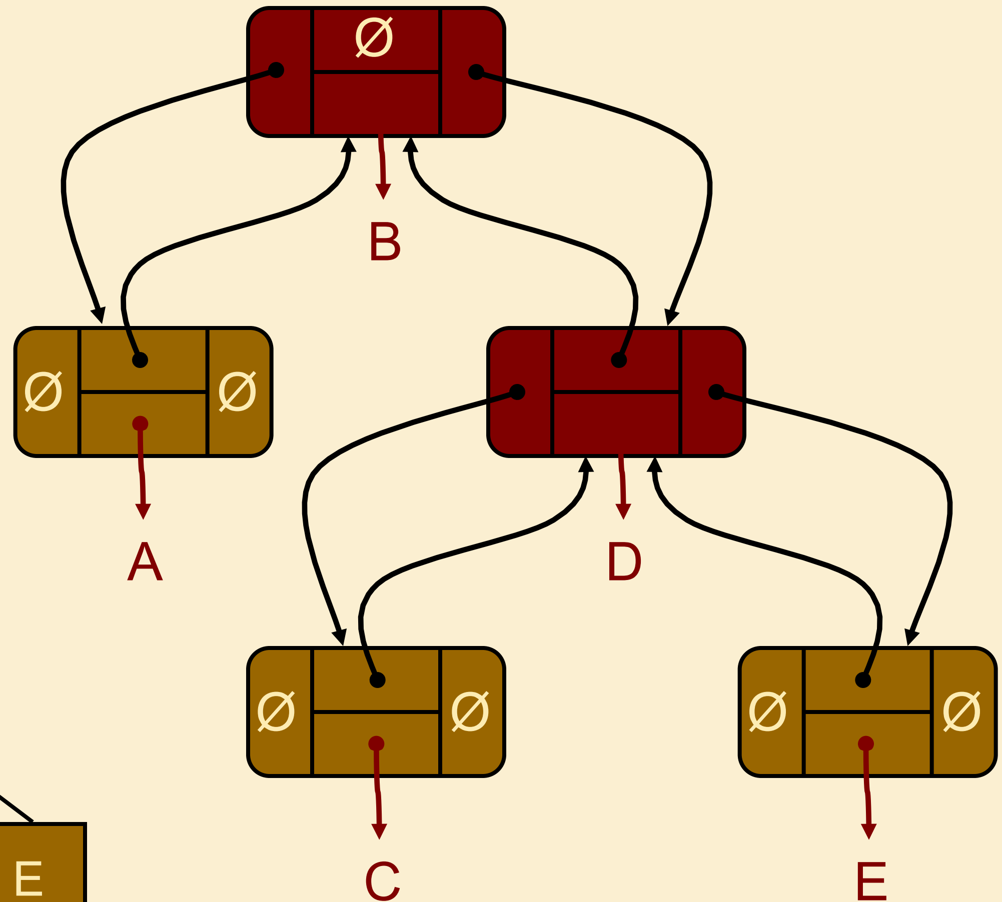
- The BinaryTree ADT extends the Tree ADT, i.e., it inherits all the methods of the Tree ADT
- Additional methods:
  - Position **left**(p)
  - Position **right**(p)
  - boolean **hasLeft**(p)
  - boolean **hasRight**(p)
- Update methods may be defined by data structures implementing the BinaryTree ADT

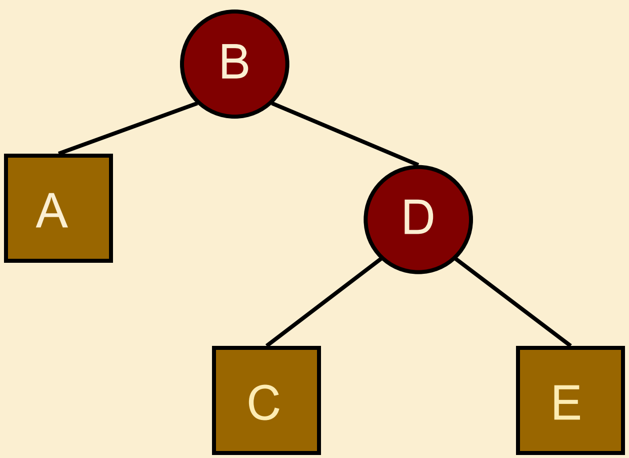
Representing Binary Trees

- Linked Structure Representation
- Array Representation

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

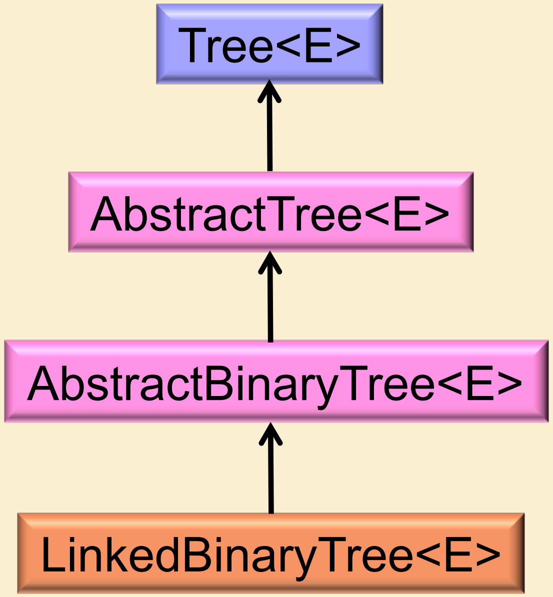




Software Resources

- Goodrich, Tamassia & Goldwasser, the authors of our textbook, provide a java software repository called net.datastructures.
- You can download it at http://net3.datastructures.net/
- We will use pieces of it for Assignment 3.

Implementation of Linked Binary Trees in net.datastructures



We will use this in Assign. 3!

Returns number of ancestors of p.

Returns height of subtree rooted at p.

Query Methods:

- size()
- isEmpty()
- isRoot(p)
- isInternal(p)
- isExternal(p)
- numChildren(p)
- depth(p)
- height(p)

Accessor Methods:

- root()
- left(p)
- right(p)
- parent(p)
- children(p)
- sibling(p)

Modification Methods:

- addRoot(e)
- addLeft(p,e)
- addRight(p,e)
- remove(p)
- set(p,e)
- attach(p,t1,t2)

Traversal Methods:

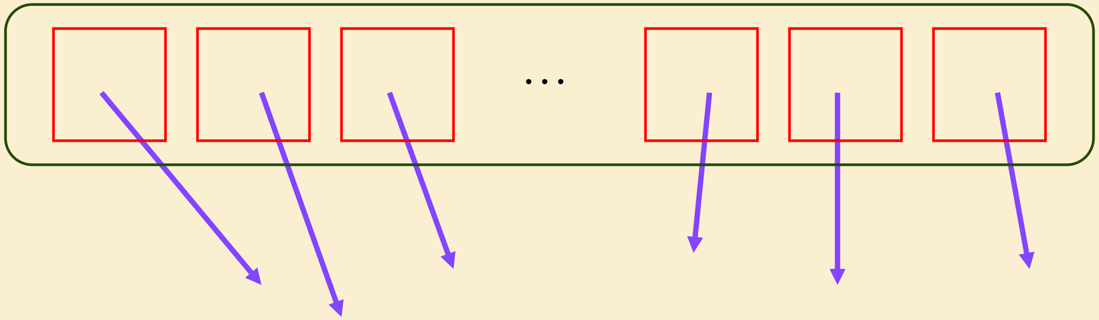
- positions()
- preorder()
- postorder()
- inorder()
- breadthfirst()

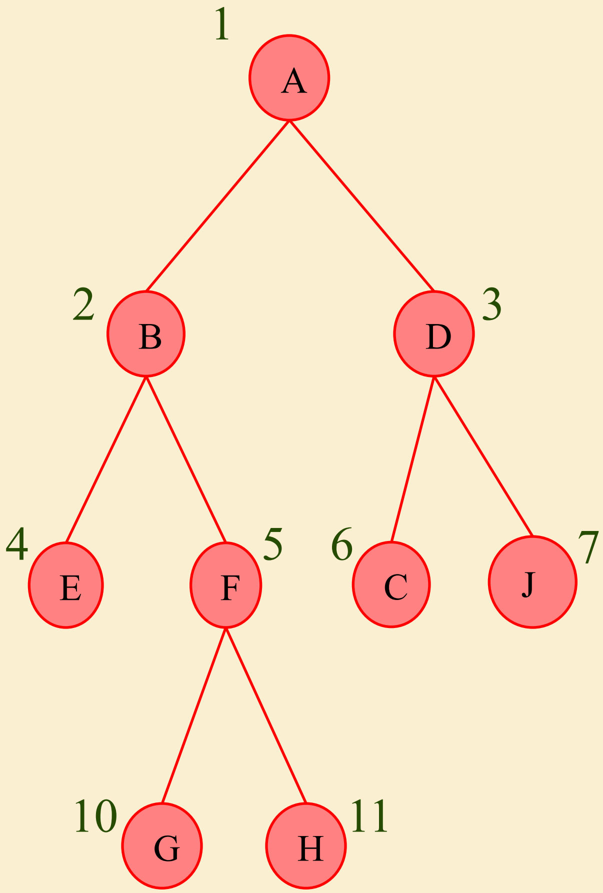
Some Important Exceptions

- addRoot(e): Throws exception if tree is not empty.
- addLeft(p,e): Throws exception if p already has left child.
- addRight(p,e): Throws exception if p already has right child.
- remove(p): Throws exception if p has more than one child.
- attach(p,t1,t2): Throws exception if p is not a leaf node.

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**

Comparison

Linked Structure

- Requires explicit representation of 3 links per position:
  - parent, left child, right child
- Data structure grows as needed – no wasted space.

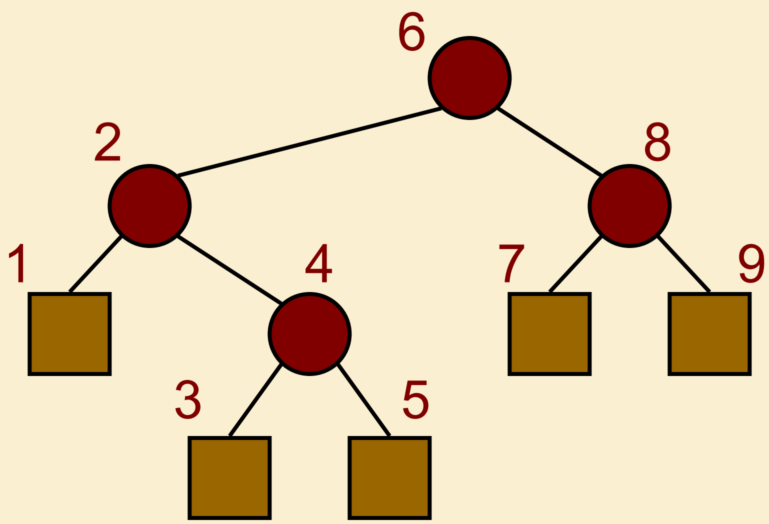
Array

- Parent and children are implicitly represented:
  - Lower memory requirements per position
- Memory requirements determined by height of tree. If tree is **sparse**, this is highly inefficient.

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))
```



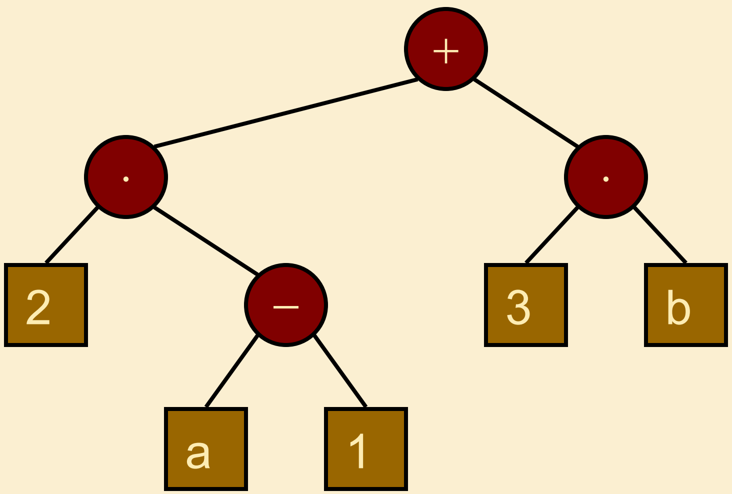
Print Arithmetic Expressions

- 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("(")
    printExpression (left(v))
print(v.element ())
if hasRight (v)
    printExpression (right(v))
    print (")")
```

Input:

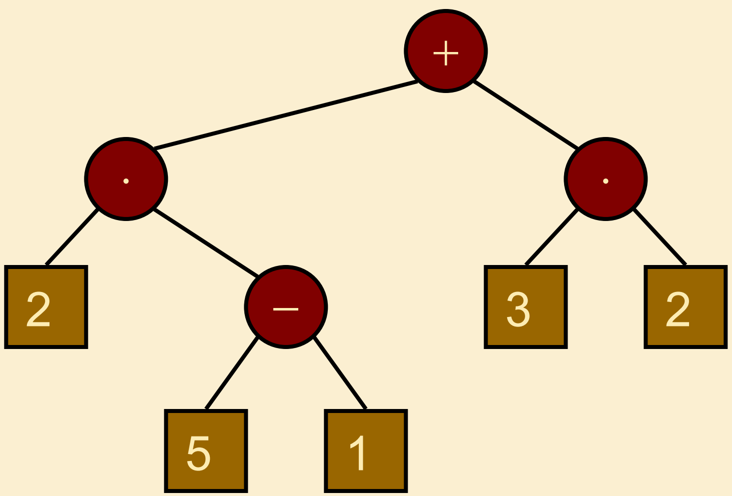


Output:

((2 × (a – 1)) + (3 × b))

Evaluate Arithmetic Expressions

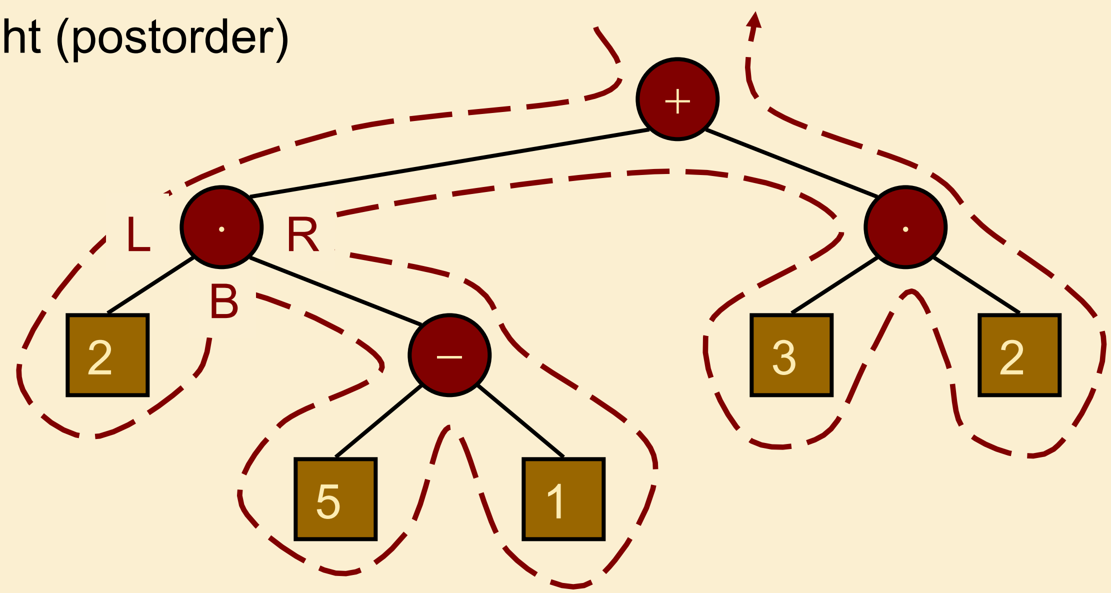
- Specialization of a postorder traversal
  - recursive method returning the value of a subtree
  - when visiting an internal node, combine the values of the subtrees



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)



Outline

- Definitions
- Traversing trees
- Binary trees

Outcomes

- By understanding this lecture you should be able to:
  - Correctly use terminology associated with trees
  - Explain the purpose of the Position ADT
  - Understand and design ADTs for trees
  - Explain the difference between the 3 common types of tree traversal, and when each might be used
  - Explain what makes a tree a binary tree, and give example applications of binary trees.
  - Implement trees using linked nodes
  - Implement binary trees using arrays.
  - Explain the advantages and disadvantages of linked node and array implementations of binary trees