

# Trees

*Dr. Aiman Hanna*

**Department of Computer Science & Software Engineering  
Concordia University, Montreal, Canada**

**These slides have been extracted, modified and updated from original slides of :**

**Data Structures and Algorithms in Java, 5th edition. John Wiley & Sons, 2010. ISBN 978-0-470-38326-1.**

**Data Structures and the Java Collections Framework by William J. Collins, 3rd edition, ISBN 978-0-470-48267-4.**

**Both books are published by Wiley.**

**Copyright © 2010-2011 Wiley**

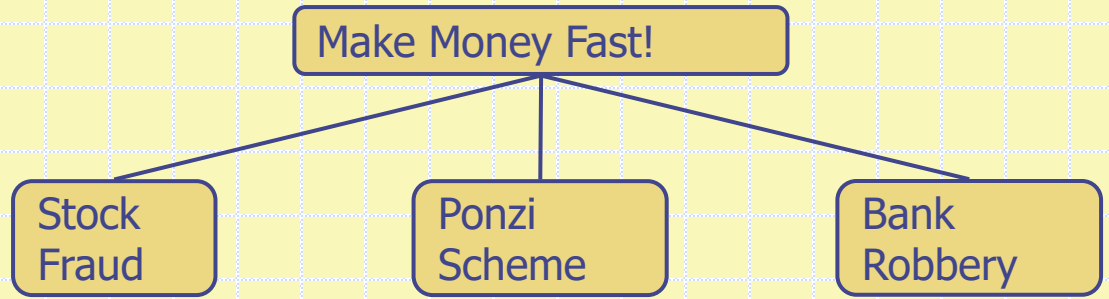
**Copyright © 2010 Michael T. Goodrich, Roberto Tamassia**

**Copyright © 2011 William J. Collins**

**Copyright © 2011-2021 Aiman Hanna**

**All rights reserved**

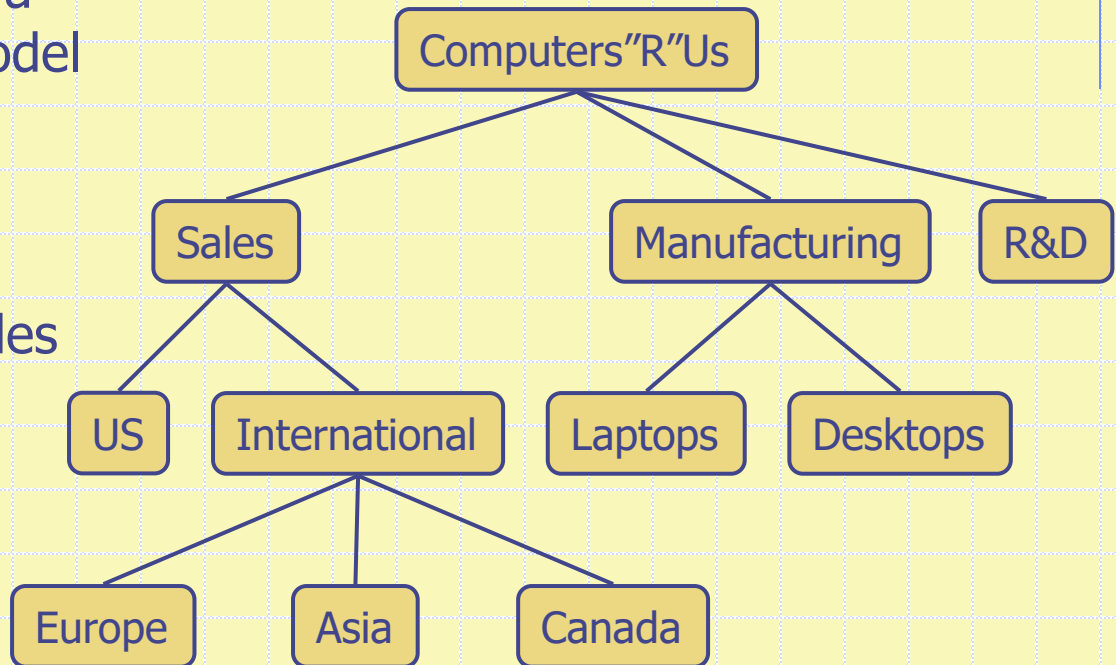
# Coverage



- General Trees
- Tree Traversal Algorithms
- Binary Trees

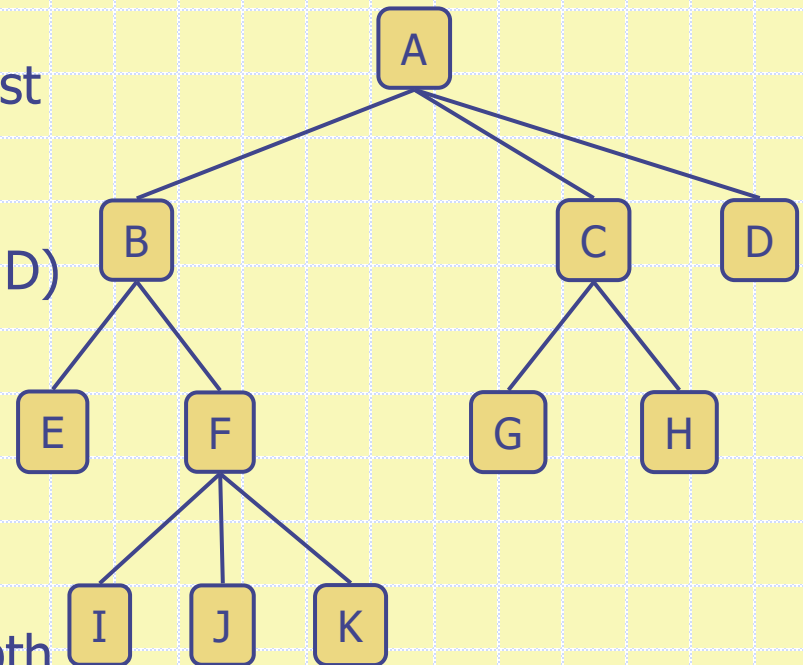
# What is a Tree

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



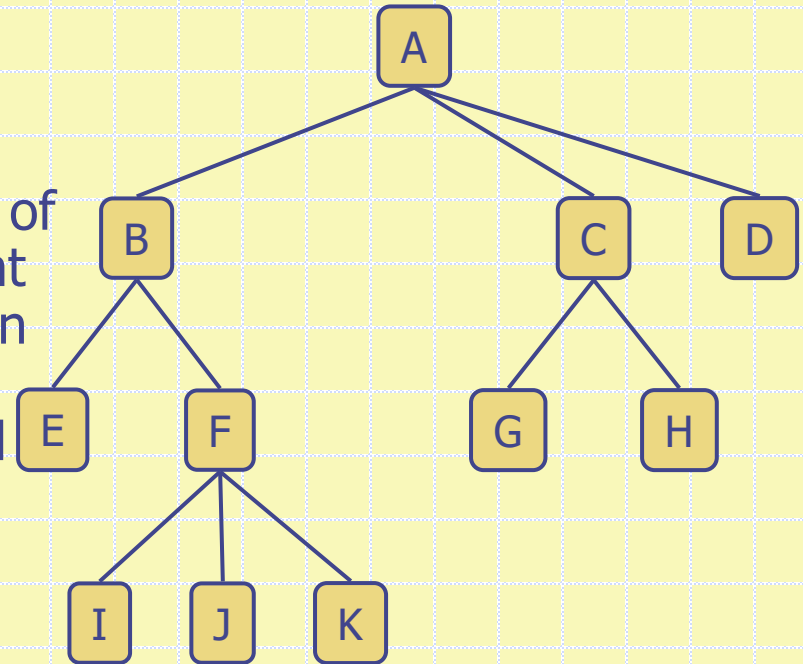
# Tree Terminology

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



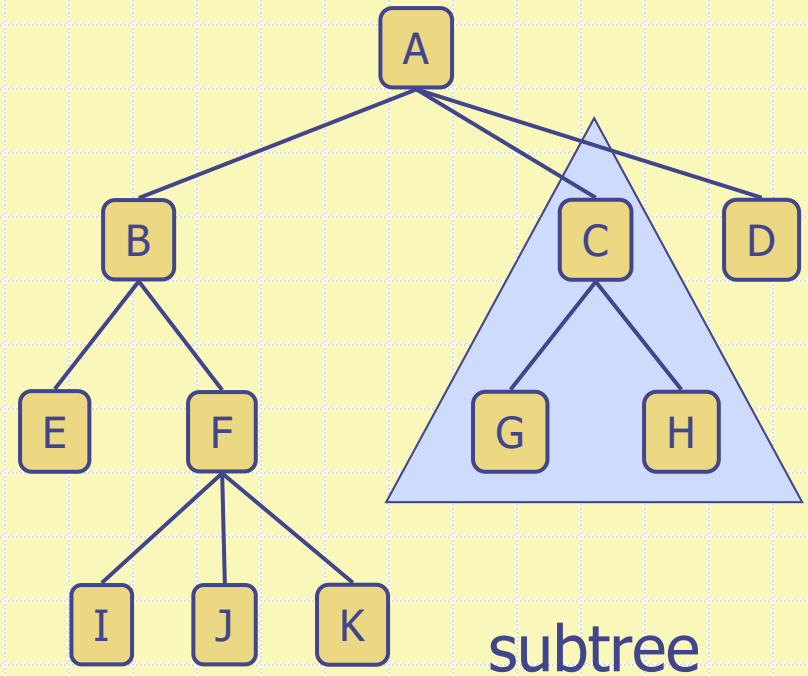
# Tree Terminology

- ❑ **siblings:** Two nodes that are children of the same parent.
- ❑ **edge:** an edge of a tree is a pair of nodes  $(u, v)$ , where  $u$  is the parent and  $v$  is the child, or vice versa. In other words, an edge is a connection between a parent and child in the tree.
- ❑ **Path:** a sequence of nodes such that any two consecutive nodes in the sequence form an edge (for instance: A, B, F, J).



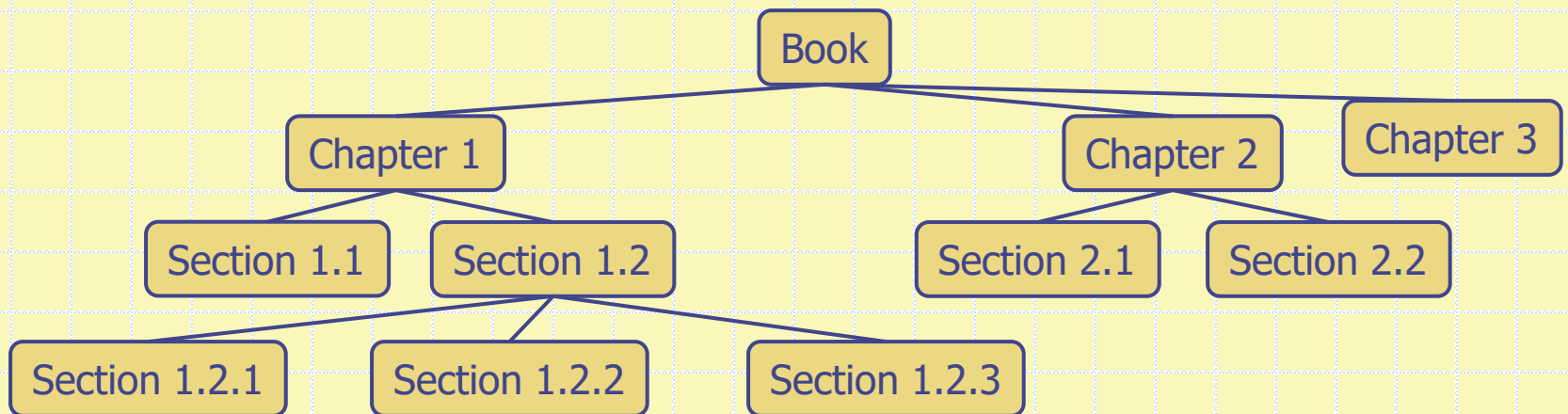
# Tree Terminology

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



# Ordered Trees

- A tree is *ordered* if there is a linear ordering defined for the children of each node.
- That is, with ordered trees we can identify the children of a node as being first, second, third, etc.
- Ordered trees typically indicate the linear order among siblings by listing them in the correct (ordered) order.



# Tree ADT

- We can use positions to abstract nodes. *Positions* of a tree are its *nodes*. The terms “position” and “node” is hence used interchangeably. A position object support the following method:
  - **element()**: Return the object stored in the position.
- Generic methods:
  - integer **size()**: Return the number of nodes in the tree.
  - boolean **isEmpty()**: Tests whether or not the tree has nodes.
  - Iterator **iterator()**: Return an iterator of all the elements stored at nodes of the tree.
  - Iterable **positions()**: Return an iterable collection of all the nodes of the tree.



# Tree ADT

- Accessor methods:

- position **root**(): Return the “root” of tree; error if tree is empty.

- position **parent**( $p$ ): Return the parent of  $p$ ; error if  $p$  is the root.

- Iterable **children**( $p$ ): Return an iterable collection containing all the children of node  $p$ .

- If the tree is ordered, then the iterable collection returned by **children**( $p$ ) stores the children of  $p$  in order.

- If  $p$  is a leaf, then the returned collection is empty.

# Tree ADT

## ◆ Query methods:

- boolean **isInternal**( $p$ ): Tests whether node  $p$  is internal .
- boolean **isExternal**( $p$ ): Tests whether node  $p$  is external.
- boolean **isRoot**( $p$ ): Tests whether node  $p$  is the root.

## ◆ Update method:

- element **replace** ( $p$ ,  $e$ ): Replace the element at node  $p$  with  $e$  and return the original (old) element.

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

# Tree ADT

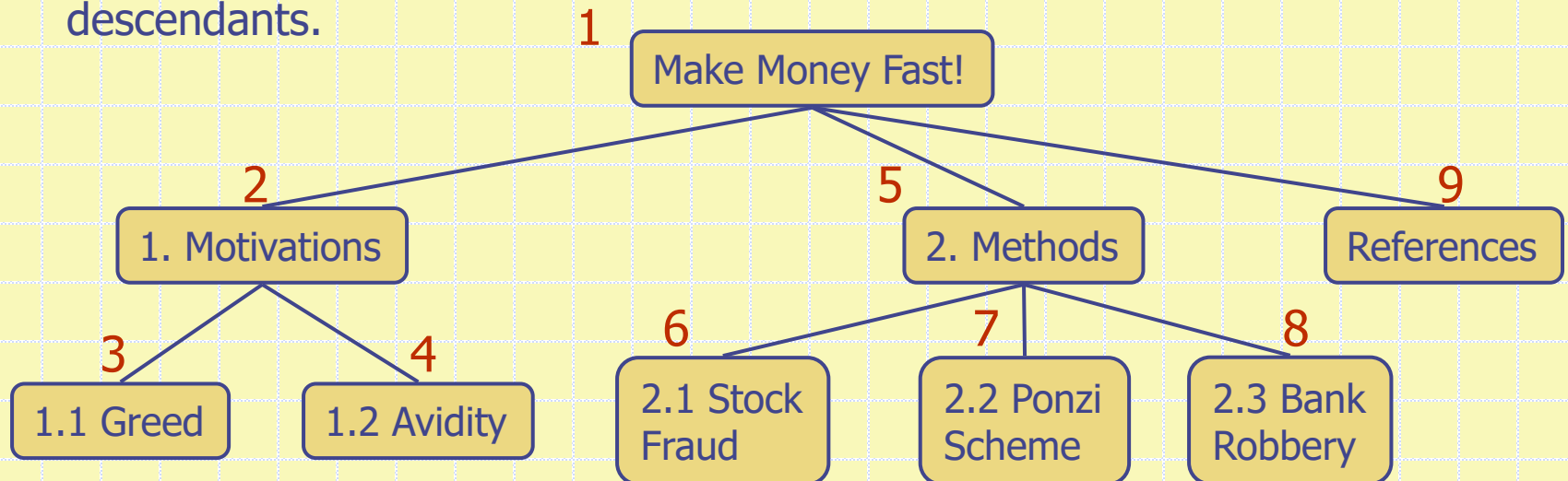
□ Performance of the Tree ADT methods:

Operation	Complexity
size, isEmpty	$O(1)$
Iterator, positions	$O(n)$
replace()	$O(1)$
root, parent	$O(1)$
children(v)	$O(c_v)$ <i><math>c_v</math> denotes the number of children at node C.</i>
isInternal, isExternal, isRoot	$O(1)$

# Preorder Traversal

- A traversal visits the nodes of a tree in a systematic manner
- In a preorder traversal, the root is visited first and then subtrees rooted as its children are traversed recursively in the same manner.
- That is, a node is visited before its descendants.

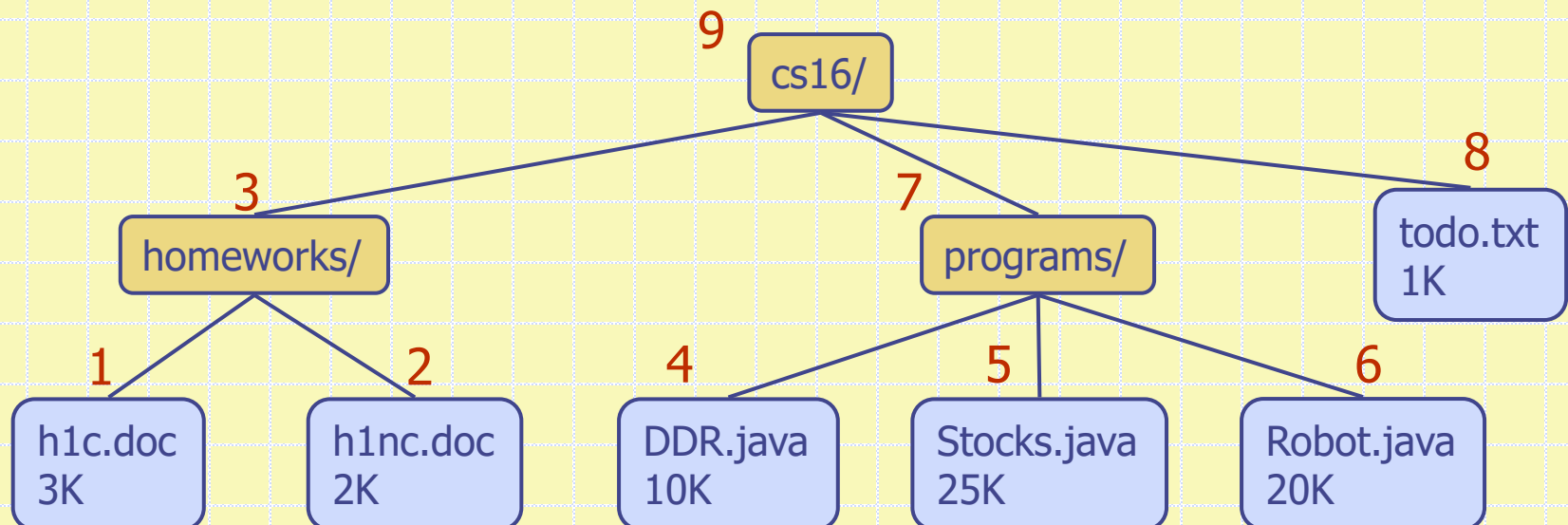
**Algorithm** *preOrder(v)*  
*visit(v)*  
**for each** child *w* of *v*  
*preorder(w)*



# Postorder Traversal

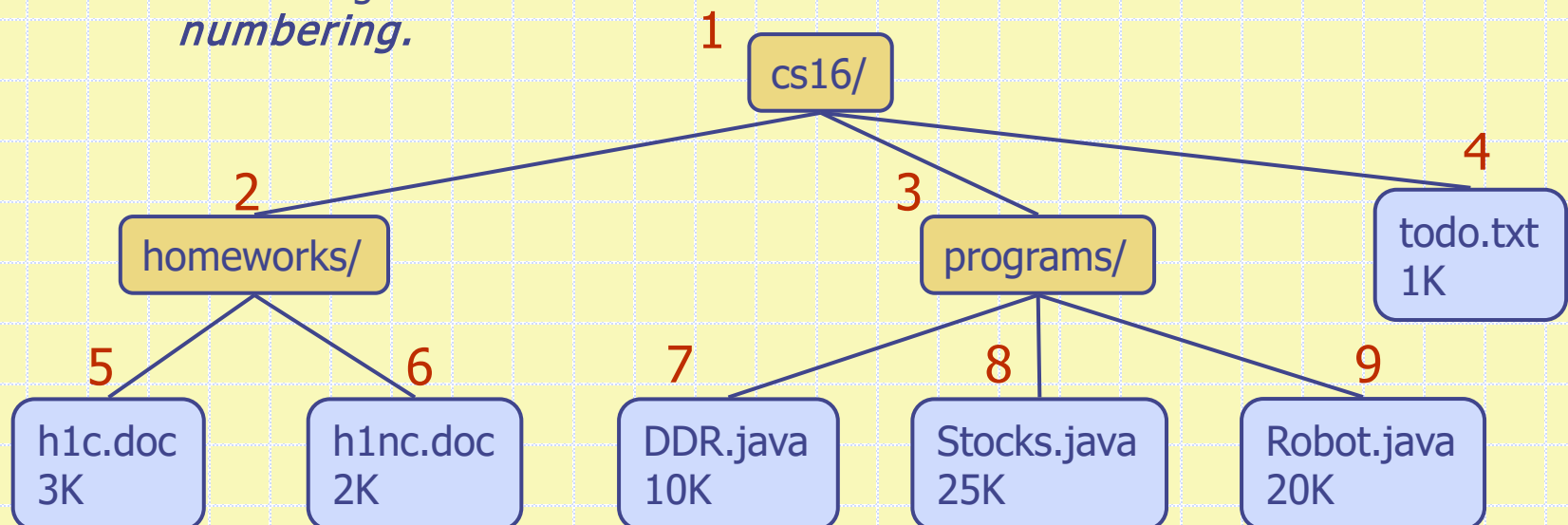
- In a postorder traversal, a node is visited after its descendants
- Application: compute space used by files in a directory and its subdirectories

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



# Other Traversals

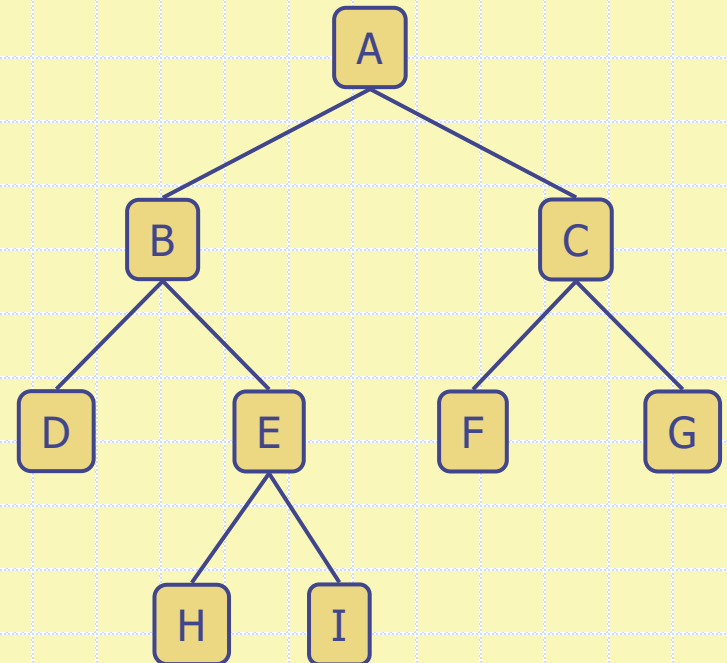
- Preorder and Postorder traversals are common but other traversals are possible.
- For instance, we can visit all the nodes at depth  $d$  before visiting the ones at depth  $d + 1$ .
  - Numbering the nodes as we visit them in such fashion is called *level numbering*.



# Binary Trees

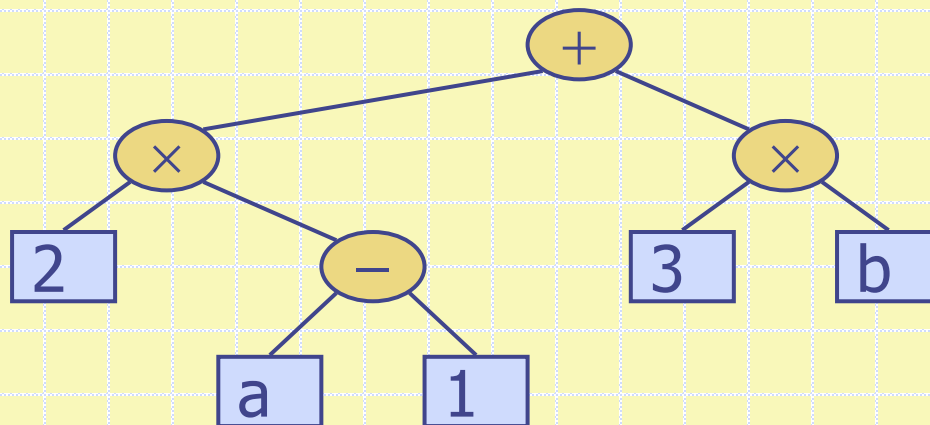
- A binary tree is a tree with the following properties:
  - Each internal node has at most two children (exactly two for **proper** binary trees; otherwise the tree is **improper** )
- We call the children of an internal node **left child** and **right child**
  - The children of a node are an ordered pair, where the left child precedes the right one
- A binary tree can be defined recursively as 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

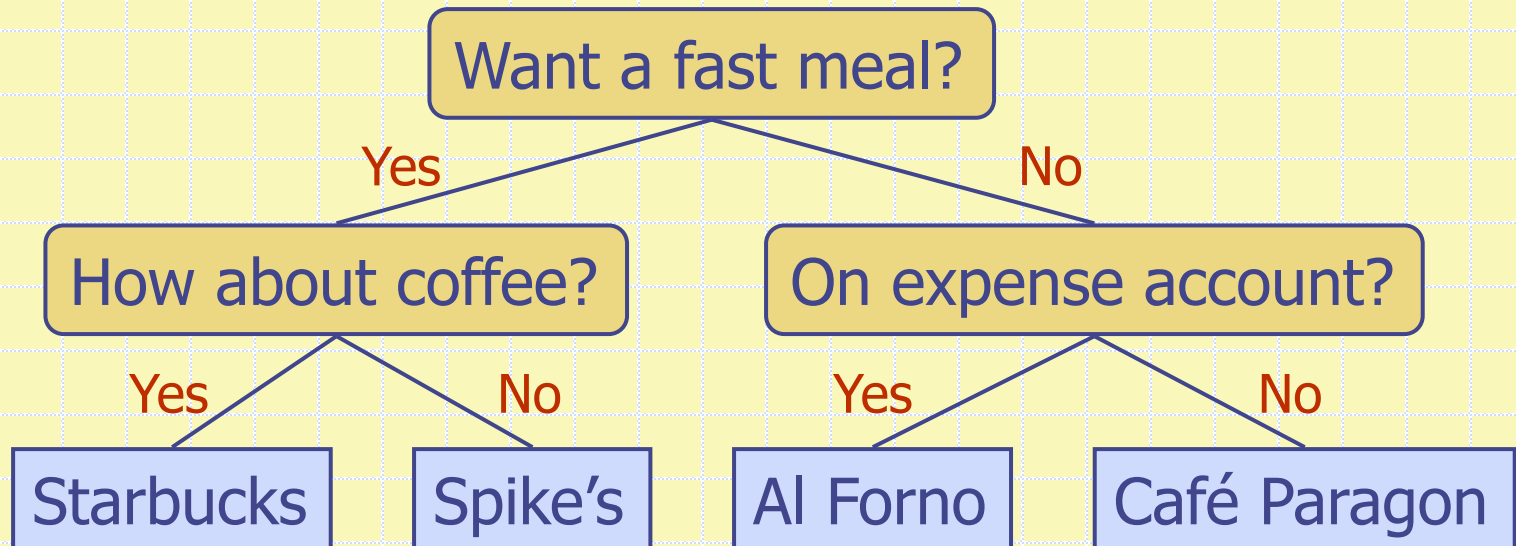
- Binary tree associated with an arithmetic expression is a proper binary tree where:
  - internal nodes: operators
  - external nodes: operands
- Example: arithmetic expression tree for the expression  $(2 \times (a - 1) + (3 \times b))$





# 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

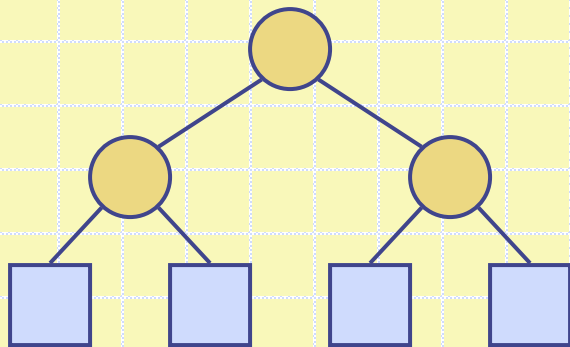
## □ Notation

$n$  number of nodes

$e$  number of  
external nodes

$i$  number of internal  
nodes

$h$  height



## ◆ Properties:

- $e = i + 1$

- $n = 2e - 1 = 2i + 1$

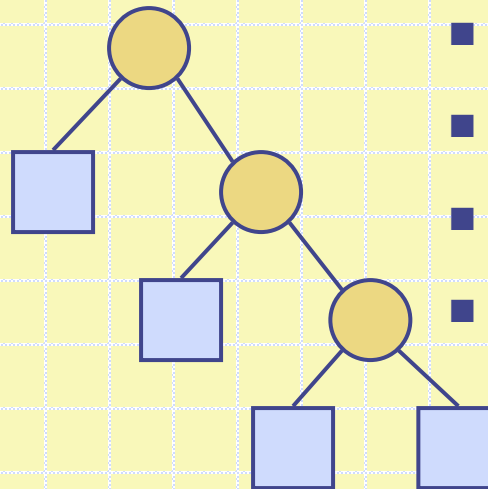
- $h \leq i$

- $h \leq (n - 1)/2$

- $e \leq 2^h$

- $h \geq \log_2 e$

- $h \geq \log_2 (n + 1) - 1$



In a perfectly balanced binary tree:

$$n = 2^0 + 2^1 + \dots + 2^h$$

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

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

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

$$\rightarrow h = \log(n + 1) - 1$$

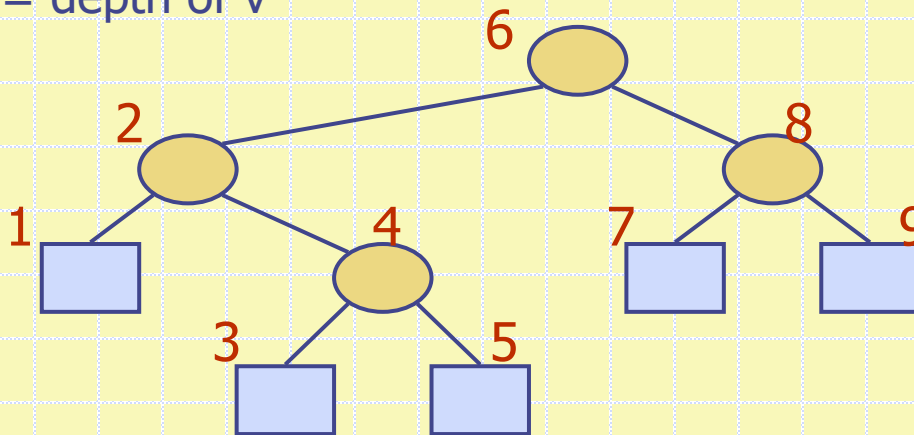
# BinaryTree ADT

- The BinaryTree ADT extends the Tree ADT, i.e., it inherits all the methods of the Tree ADT
- Additional methods:
  - position **left**( $p$ ): Return the left child of  $p$ ; error if  $p$  has no left child
  - position **right**( $p$ ): Return the right child of  $p$ ; error if  $p$  has no right child
  - boolean **hasLeft**( $p$ ): Test whether  $p$  has a left child
  - boolean **hasRight**( $p$ ): Test whether  $p$  has a right child
- Update methods may be defined by data structures implementing the BinaryTree ADT

# Inorder Traversal of Binary Trees

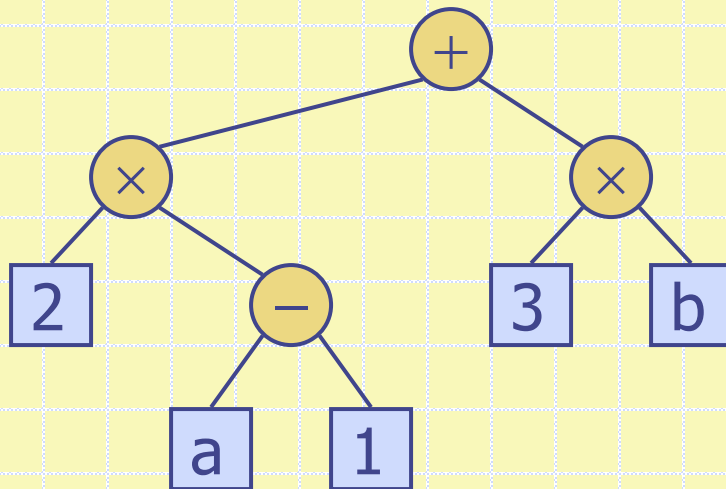
- In an **inorder** traversal a node is visited after its left subtree and before its right subtree
- Application: draw a binary tree
  - $x(v)$  = inorder rank of  $v$
  - $y(v)$  = depth of  $v$

```
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(")")
```

$((2 \times (a - 1)) + (3 \times b))$

# 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* ()

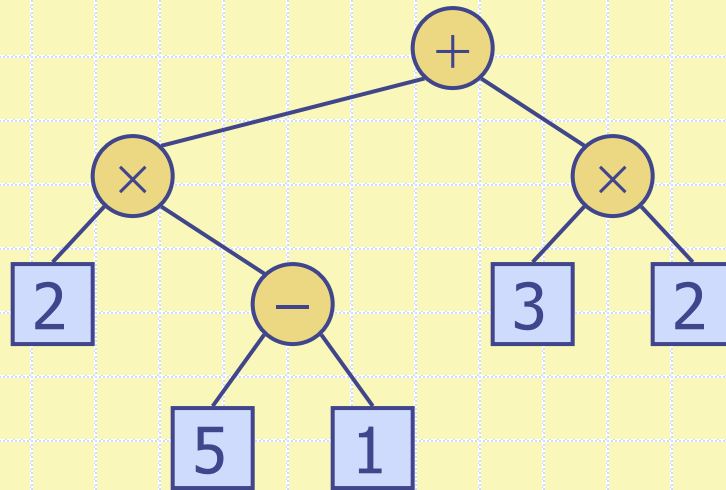
**else**

*x*  $\leftarrow$  *evalExpr*(*leftChild* (*v*))

*y*  $\leftarrow$  *evalExpr*(*rightChild* (*v*))

$\diamond \leftarrow$  operator stored at *v*

**return** *x*  $\diamond$  *y*

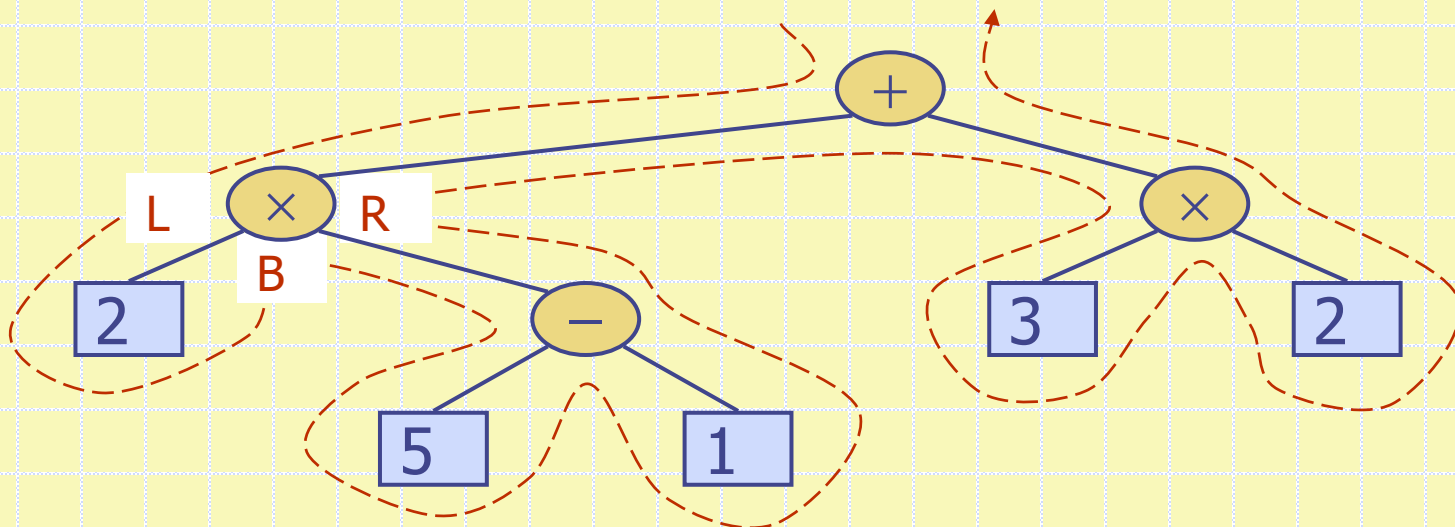


# Euler Tour Traversal

- ❑ All previously discussed traversal algorithms are forms of iterators where each traversal is guaranteed to visit each node in a certain order, and exactly once.
- ❑ Euler Tour traversal relaxes that requirement of the single visit to each node.
- ❑ The advantage of such algorithms is to allow for more general kinds of algorithms to be expressed easily.
- ❑ In other words, it provides a generic traversal of a binary tree which includes the preorder, postorder and inorder traversals.

# Euler Tour Traversal

- Euler Tour traversals walk around the tree and visit each node three times:
  - on the left (preorder – root  $\rightarrow$  left  $\rightarrow$  right)
  - from below (inorder – left  $\rightarrow$  root  $\rightarrow$  right)
  - on the right (postorder – left  $\rightarrow$  right  $\rightarrow$  root)
- If the node is external, all three visits actually happen at the same time.



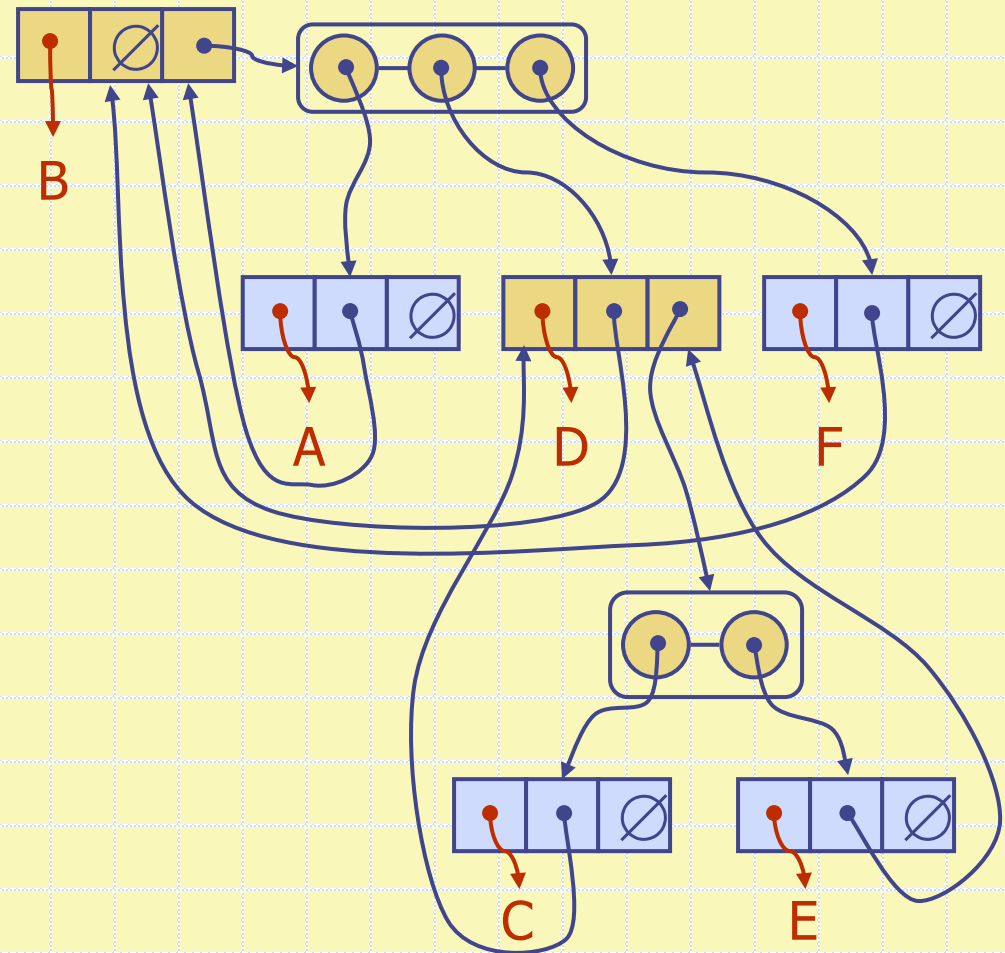
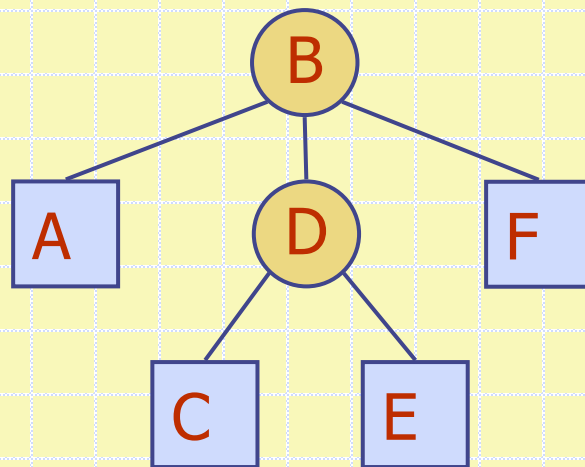


# Euler Tour Traversal

- Complexity of Euler Tour is easy to analyze:
  - Assume each visit takes a constant time  $O(1)$
  - Since we spend a constant time at each node, the overall running time is  $O(n)$
- The Euler Tour can perform other kinds of traversals as well
- For instance, to compute the number of descendants of a node  $v$ , we can perform the following:
  - Start the tour by initializing a counter to 0
  - Record the counter when  $v$  is visited on the left, say  $x$
  - Record the counter when  $v$  is visited on the right, say  $y$
  - Number of descendants =  $y - x + 1$

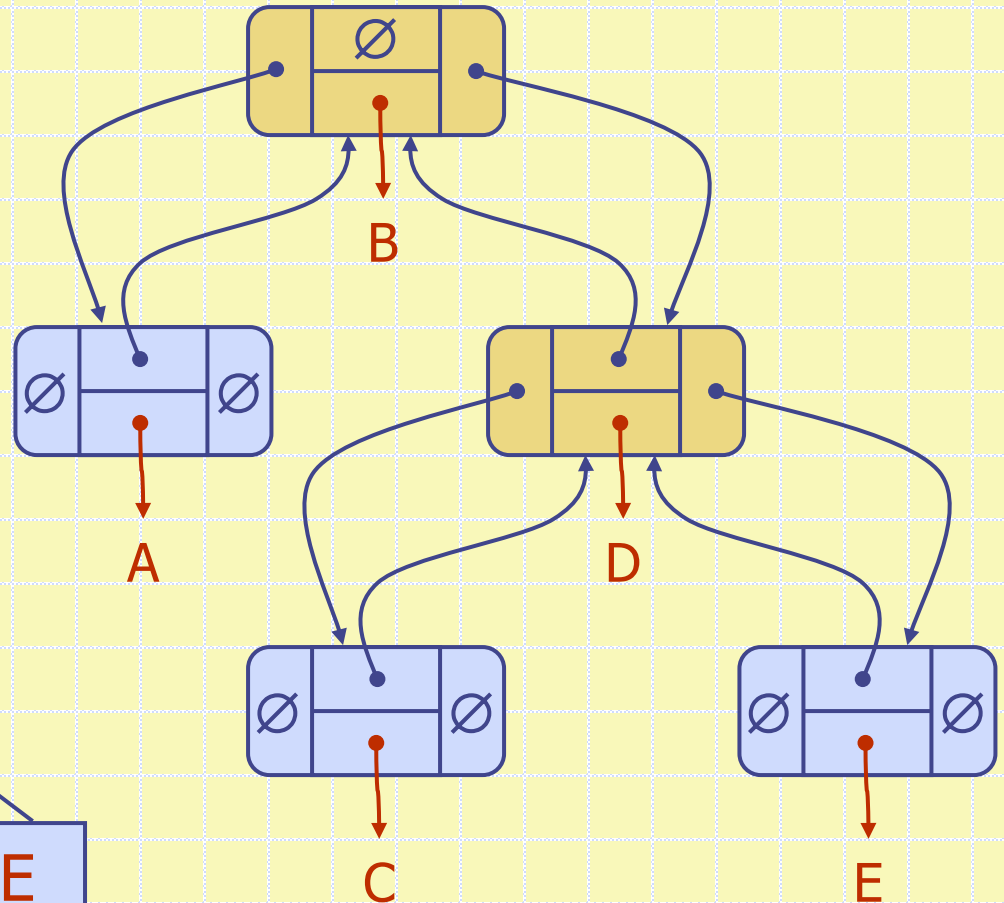
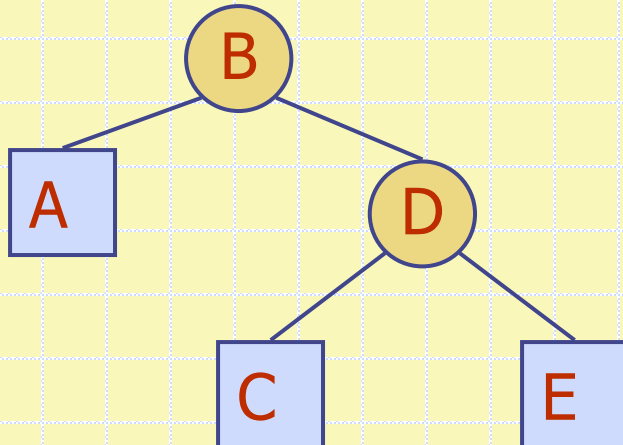
# Linked Structure for General Trees

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

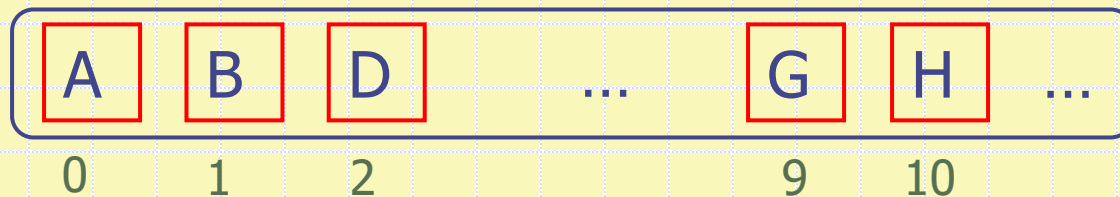


# Performance of List Implementation of Binary Trees

Operation	Complexity
size, isEmpty	$O(1)$
Iterator, positions	$O(n)$
replace	$O(1)$
root, parent, left, right, sibling, children	$O(1)$
children(v)	$O(1)$ <i>Since there are at most two children of any node in a binary tree.</i>
hasLeft, hasRight, isInternal, isExternal, isRoot, remove, insertLeft, insertRight, attach	$O(1)$

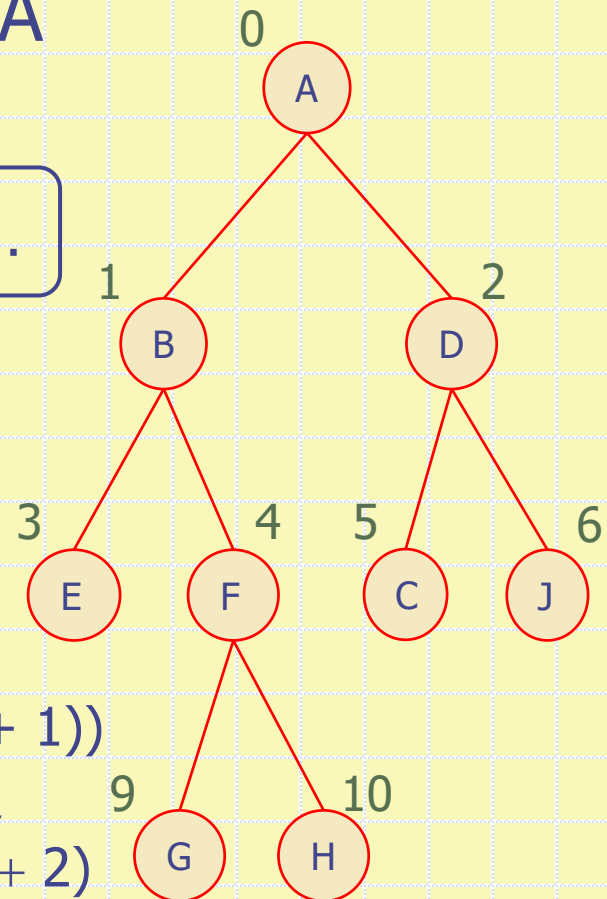
# Array-Based Representation of Binary Trees

- Nodes are stored in an array A

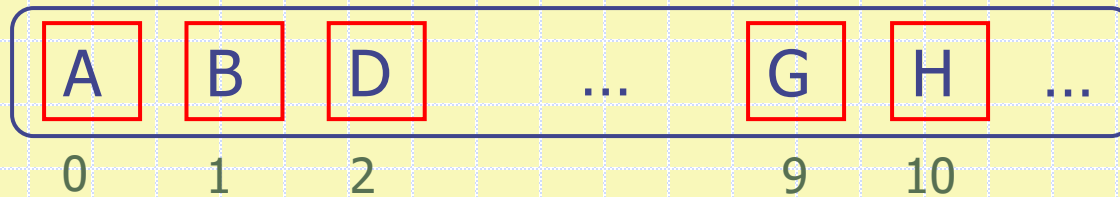


- Use level numbering
- Node v is stored at  $A[\text{rank}(v)]$

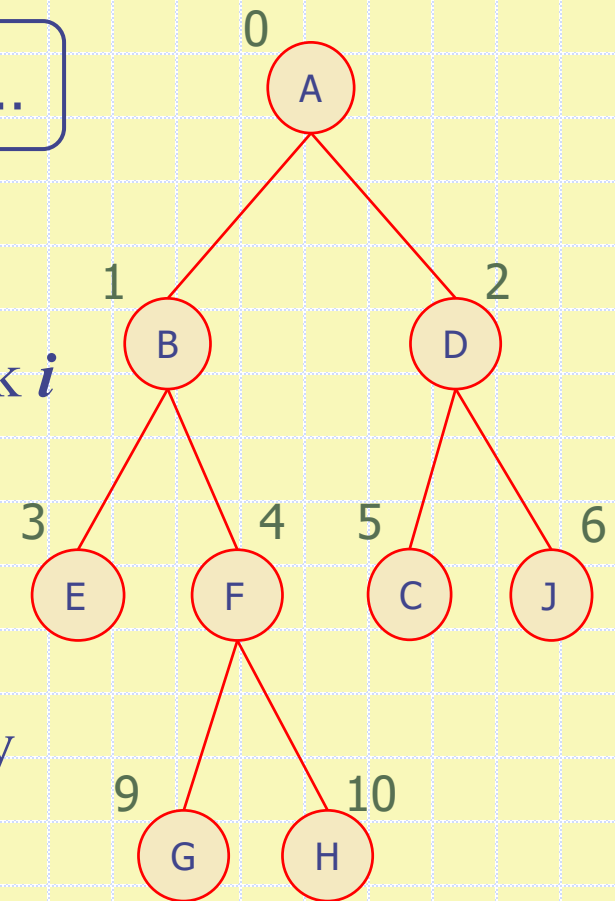
- $\text{rank}(\text{root}) = 0$
- if node is the left child of  $\text{parent}(\text{node})$ ,  
 $\text{rank}(\text{node}) = 2 \cdot (\text{rank}(\text{parent}(\text{node}) + 1))$
- if node is the right child of  $\text{parent}(\text{node})$ ,  
 $\text{rank}(\text{node}) = 2 \cdot (\text{rank}(\text{parent}(\text{node})) + 2)$



# Array-Based Representation of Binary Trees



- So, in general; for any parent/root at rank  $i$ 
  - the left child is at rank  $2i + 1$
  - the right child is at rank  $2i + 2$
  - links between nodes are not explicitly stored



# Performance of Array-based Implementation of Binary Trees

Operation	Complexity
size, isEmpty	$O(1)$
Iterator, positions	$O(n)$
replace()	$O(1)$
root, parent, left, right, children	$O(1)$
children(v)	$O(1)$ <i>Since there are at most two children of any node in a binary tree.</i>
hasLeft, hasRight, isInternal, isExternal, isRoot	$O(1)$

# Template Method Pattern

- Generic algorithm
- Implemented by abstract Java class
- Visit methods redefined by subclasses
- Template method `eulerTour`
  - Recursively called on left and right children
  - A `TourResult` object with fields `left`, `right` and `out` keeps track of the output of the recursive calls to `eulerTour`

```
public abstract class EulerTour <E, R> {  
    protected BinaryTree<E> tree;  
    public abstract R execute(BinaryTree<E> T);  
    protected void init(BinaryTree<E> T) { tree = T; }  
    protected R eulerTour(Position<E> v) {  
        TourResult<R> r = new TourResult<R>();  
        visitLeft(v, r);  
        if (tree.hasLeft(p))  
            { r.left=eulerTour(tree.left(v)); }  
        visitBelow(v, r);  
        if (tree.hasRight(p))  
            { r.right=eulerTour(tree.right(v)); }  
        return r.out;  
    }  
    protected void visitLeft(Position<E> v, TourResult<R> r) {}  
    protected void visitBelow(Position<E> v, TourResult<R> r) {}  
    protected void visitRight(Position<E> v, TourResult<R> r) {}  
}
```



# Specializations of EulerTour

- Specialization of class EulerTour to evaluate arithmetic expressions
- Assumptions
  - Nodes store ExpressionTerm objects with method getValue
  - ExpressionVariable objects at external nodes
  - ExpressionOperator objects at internal nodes with method setOperands(Integer, Integer)

```
public class EvaluateExpressionTour
    extends EulerTour<ExpressionTerm, Integer> {
    public Integer execute
        (BinaryTree<ExpressionTerm> T) {
        init(T);
        return eulerTour(tree.root());
    }
    protected void visitRight
        (Position<ExpressionTerm> v,
         TourResult<Integer> r) {
        ExpressionTerm term = v.element();
        if (tree.isInternal(v)) {
            ExpressionOperator op = (ExpressionOperator) term;
            op.setOperands(r.left, r.right);
            r.out = term.getValue();
        }
    }
}
```