

# CSE 201: Data Structures and Algorithms

## Lecture 8:Trees

Dr. Vidhya Balasubramanian

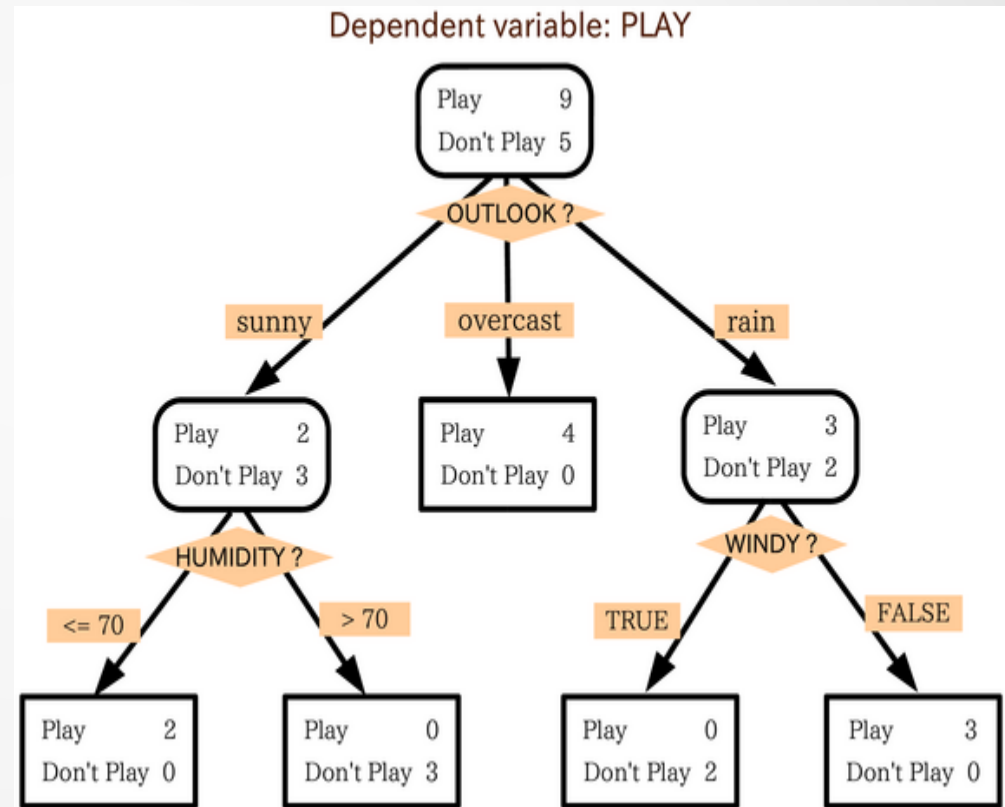
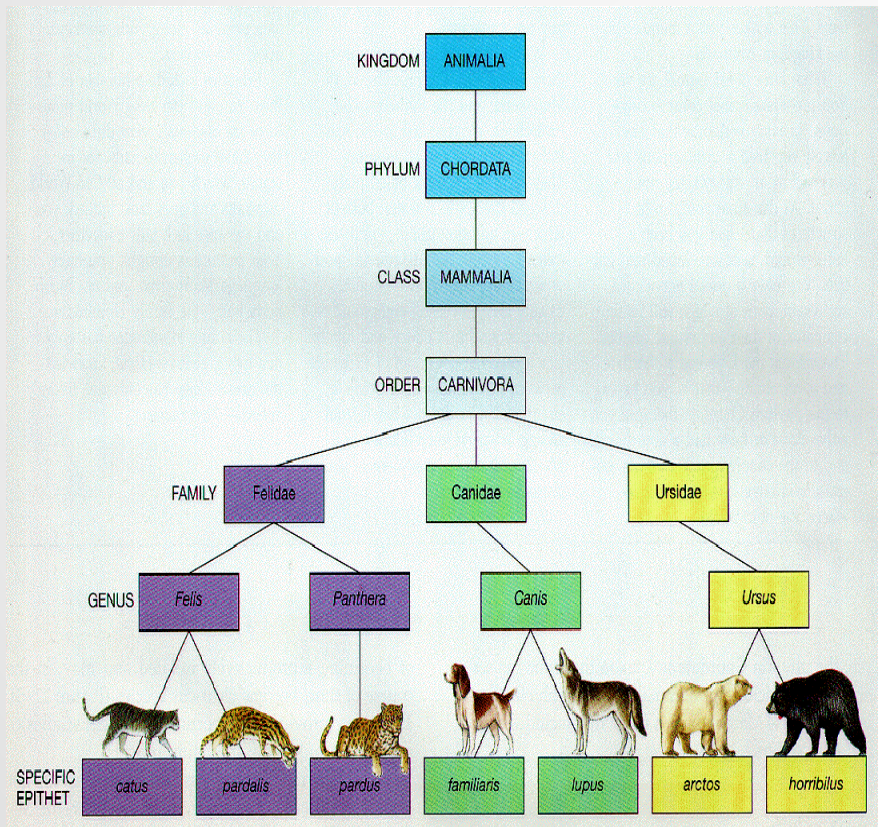
# Non Linear Data Structures

- Represent relationships more richer than “before” and “after” in sequences
- Examples
  - Trees
  - Hash Table
  - Dictionaries
  - Skip Lists



# Basic Concept

- Root : Primary category, starting point etc
- Children: Subcategory, descendants etc



<http://ridge.icu.ac.jp/gen-ed/classif-gifs/animal-class-example.gif>



# Applications/Scenarios Trees Model

- Trees capture situations where branching happens
- Hierarchies
  - Captures types subtype relationships
  - e.g Family trees or hierarchies
  - Animal or Plant Taxonomies
- Decisions
  - Root denotes the primary start point, and every branch leads to an alternate decision
- Organization into ranges
  - Each child represents one range

# Trees: Basic Definitions

- Tree is an abstract model of a hierarchical structure
- A tree  $T$  is a set of **nodes** storing elements in a **parent-child** relationship with the following properties
  - $T$  has special node  $r$  called the “**root**” which has no parent node
  - Each node  $v$  of  $T$  different from  $r$  has a unique parent node  $u$
  - If  $u$  is the parent node,  $v$  is the **child** of  $u$
  - Two nodes that are children of the same parent are **siblings**

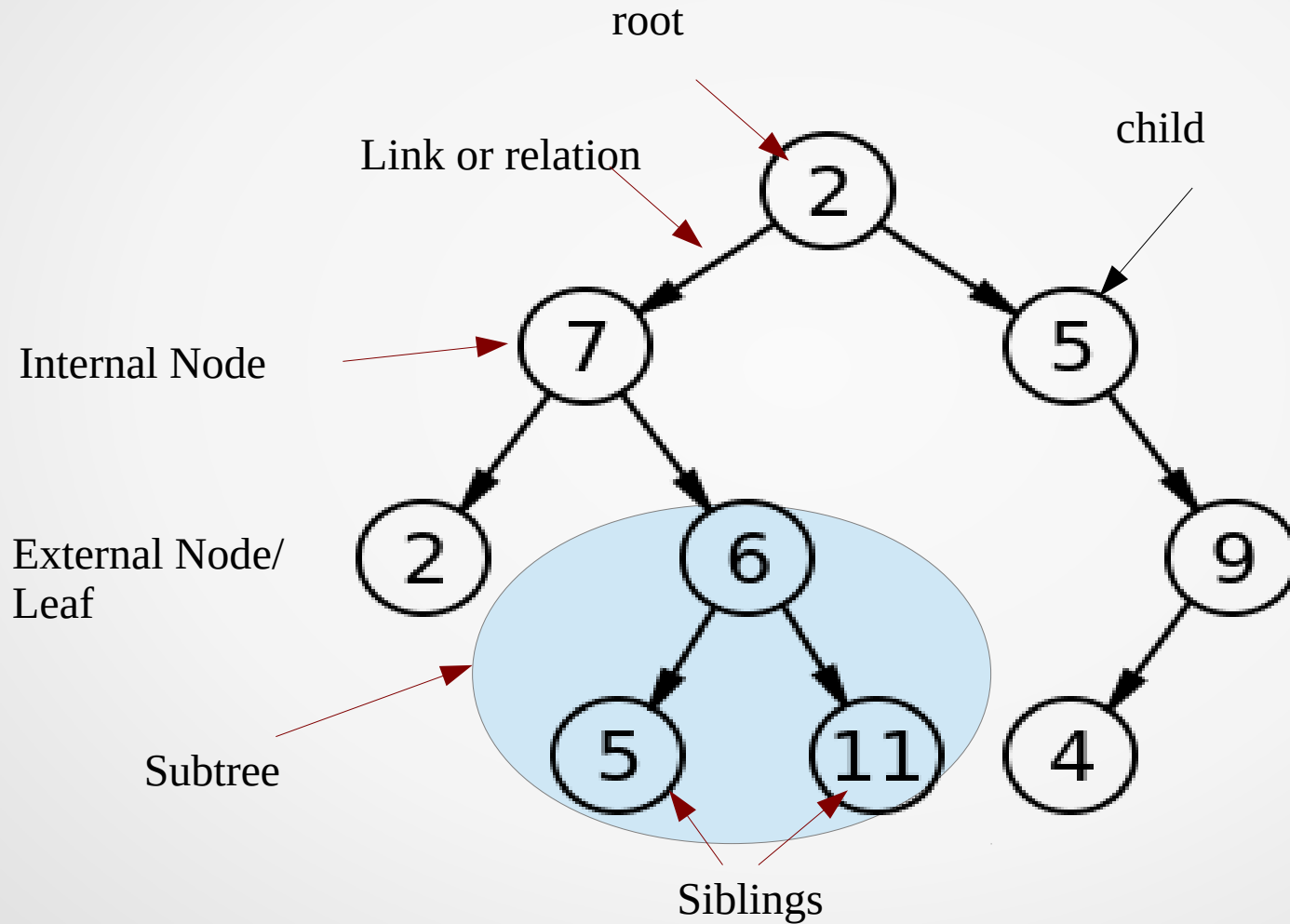
# Trees: Basic Definitions

- **External node**: node that has no children
  - Also called **leaf** node
- **Internal node** has one or more children
- **Subtree** of  $T$  rooted at a node  $v$  is the tree consisting of all the descendants of  $v$  in  $T$  (including  $v$ )
- **Ancestor** of a node is
  - Node itself or
  - ancestor of the parent of the node
  - $v$  is **descendent** of  $u$  if  $u$  is ancestor of  $v$

# Trees: Basic Definition

- **Depth** of a node
  - length of the path to the root
    - Indicates the level of the node
  - Depth of the root is 0, and is at level 0
- **Height** of a node
  - length of the longest downward path to a leaf from that node
  - Height of the tree is the height of the root
    - Equal to the depth of the deepest node in the tree
      - If depth of the deepest node is 3, height is 3

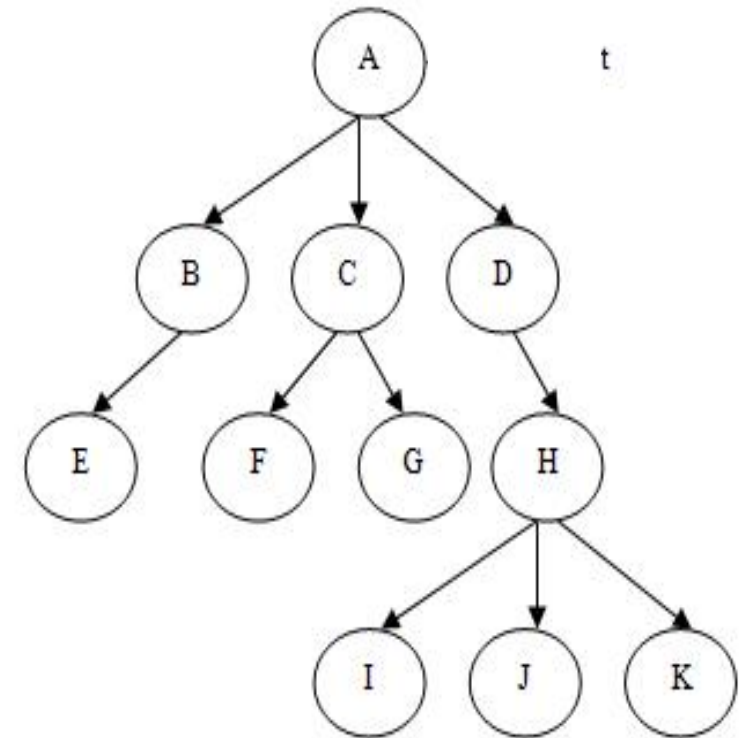
# Trees: Basic Definitions





# Exercise

- Consider the tree given in figure
  - Which node is the root
  - List the internal nodes
  - How many descendants does node C have
  - How many ancestors does node J and node G have
  - Which nodes are in the subtree rooted at D
  - What is the height of the tree
  - What is the depth of node F



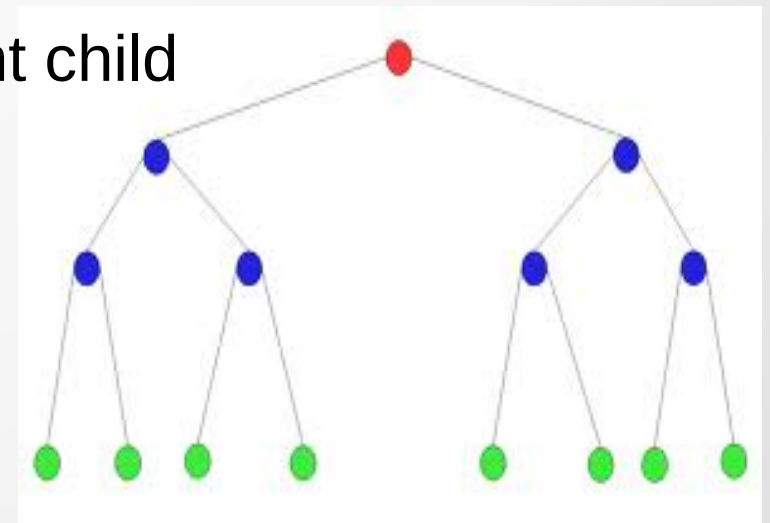
[http://lh4.ggpht.com/\\_1SkEgLzvHUY/SwgZ9Mqv0jI/AAAAAAAAARK/BSN\\_Zs-ANYI/s400/1.JPG](http://lh4.ggpht.com/_1SkEgLzvHUY/SwgZ9Mqv0jI/AAAAAAAAARK/BSN_Zs-ANYI/s400/1.JPG)

# Ordered Trees

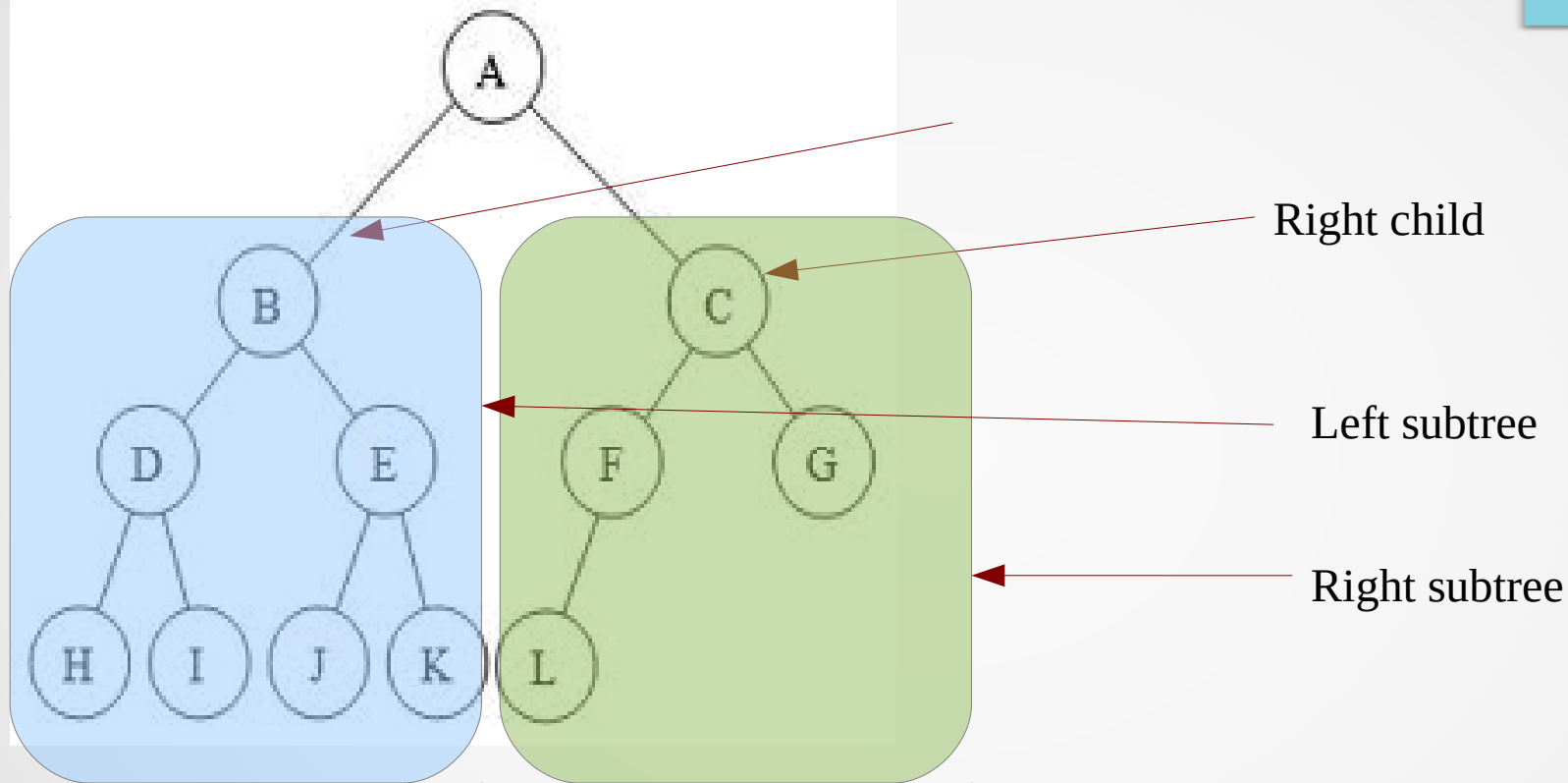
- A tree is **ordered** if there is a linear ordering defined for the children of each node
  - Each child of a node can be identified as the first, second, third etc
  - Usually done by arranging siblings left to right
- Ordered trees represent the linear order relationship between siblings
  - e.g in exercise : Children of node H can be ordered as follows
    - I, J, K

# Binary Trees

- Ordered tree in which every node atmost two children
- Tree is proper if
  - Each node has either zero or two children
  - ie every internal node has exactly two children
  - Also known as full binary tree
- Each child is labeled as left child or right child
- Subtrees
  - Left subtree rooted at a left child
  - Right subtree rooted at a right child



# Binary trees



Src: [web.cecs.pdx.edu](http://web.cecs.pdx.edu)

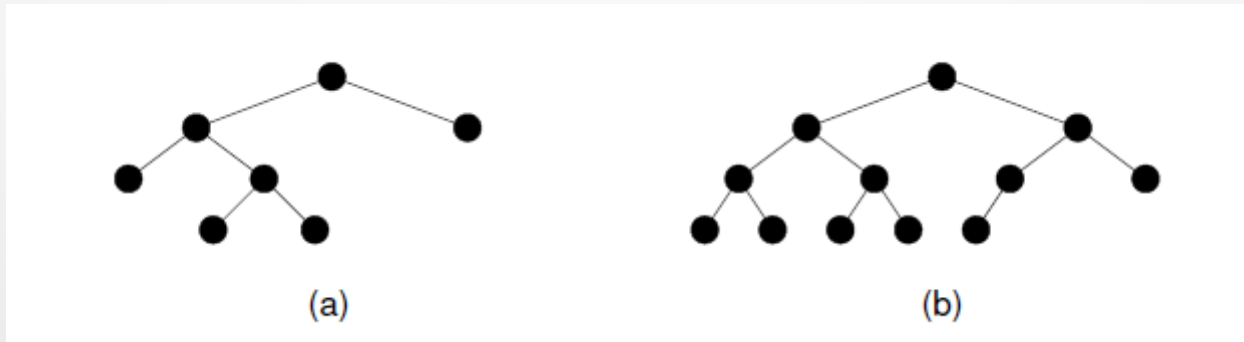
# Binary Trees: Types

- Rooted Binary Tree
  - tree with a root node in which every node has at most two children.
- Full Binary Tree (Proper or strictly binary tree)
- Perfect Binary tree
  - full binary tree in which all leaves are at the same depth and in which every parent has two children
- Balanced Binary tree
  - depth of the two subtrees of every node differ by 1 or less



# Binary Trees

- Complete Binary Trees
  - A complete binary tree has a restricted shape obtained by starting at the root and filling the tree by levels from left to right
    - all levels except possibly level  $d-1$  are completely full



Src : Clifford A. Shaffer, A Practical Introduction to Data Structures and Algorithm Analysis

# Binary Tree ADT

- Stores values at nodes, and are defined relative to the neighboring node
  - Relationships satisfy the parent child relationship
- Accessor Functions
  - root(): returns the root of the tree
  - parent(v): returns the parent of the node v; returns error if v is root.
  - children(v): returns an iterator of the children of node v
    - children can be stored in an array or list
      - For instance left child is at index 0, right at index 1

# Binary Tree ADT

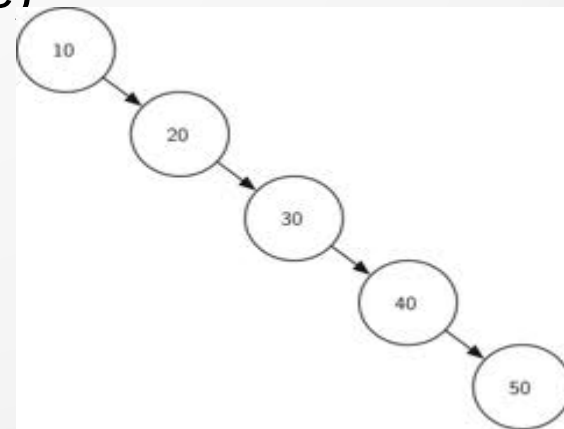
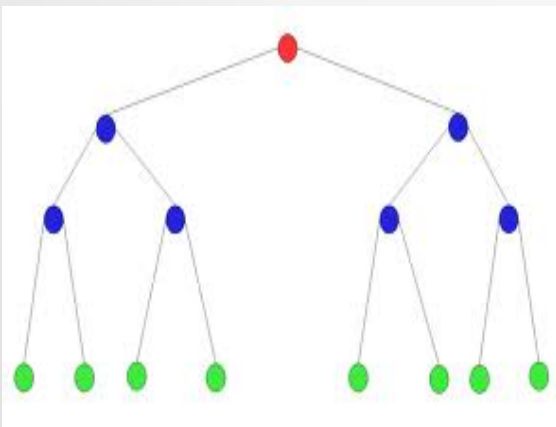
- Query Functions
  - `isInternal(v)`: Test whether node  $v$  is internal; output is Boolean
  - `isExternal(v)`: Test whether node  $v$  is external; output is Boolean
  - `isRoot(v)`: Test whether node  $v$  is a root: output is Boolean
- Update Methods
  - `swapElements(v, w)`: swap elements stored at nodes  $v$  and  $w$
  - `replaceElement(v, e)`: replace the element at node  $v$  with element  $e$

# Binary Tree ADT

- Generic methods:
  - Size()
  - isEmpty()
  - elements(): returns an iterator of all elements stored at the nodes of the tree
  - nodes(): return an iterator of all nodes of the tree

# Properties of a Binary Tree

- The number of nodes  $n$  in a perfect binary tree can be found as follows
  - $n = 2^{h+1} - 1$  where  $h$  is height of the tree
  - In figure  $n = 2^{3+1} - 1 = 16 - 1 = 15$
- The number of nodes  $n$  in a binary tree of height  $h$  is
  - at least  $n = h + 1$  ( $2h + 1$  for a proper binary tree) and
    - See figure
  - at most  $n = 2^{h+1} - 1$  (in a perfect binary tree)


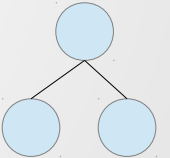




# Properties of a Binary tree

- Number of leaves in a perfect binary tree
  - $L = 2^h$ , where  $h$  is height of the tree
  - Therefore  $n = 2L - 1$
- Number of internal nodes in a complete binary tree of  $n$  nodes
  - $\lfloor n/2 \rfloor$
- For any non-empty binary tree with  $n_0$  leaf nodes and  $n_2$  nodes of degree 2,
  - $n_0 = n_2 + 1$
- $h \leq (n-1)/2$ , and  $\leq$  number of internal nodes
- $h \geq \log_2 L$ , and  $h \geq \log_2 (n + 1) - 1$

# Properties of Binary Trees

- Full Binary Tree Theorem
  - The number of leaves in a non-empty full binary tree is one more than the number of internal nodes
  - Proof: Use induction
    - Base Cases:
      - The non-empty tree with zero internal nodes has one leaf node. 
      - A full binary tree with one internal node has two leaf nodes. 
      - Thus, the theorem holds for the base cases ie for  $n = 0$  and  $n = 1$

# Full Binary Tree Theorem Proof

- Induction Hypothesis
  - Assume that, any full binary tree  $T$  containing  $n - 1$  internal nodes has  $n$  leaves
- Induction Step
  - Given  $T$  with  $n$  internal nodes, select an internal node  $u$  whose children are both leaf nodes
  - Remove both children of  $u$ , making it a leaf node, and let the resulting tree be  $T'$ 
    - $T'$  has  $n-1$  internal nodes, and by the theorem,  $n$  leaves
  - Restore the children of  $u$ , hence resulting in  $T$  with  $n$  internal nodes
    - Since  $T'$  has  $n$  leaves, adding 2 makes it  $n+2$
    - However node  $u$  which was earlier counted as leaf, now is an internal node
- Thus, tree  $T$  has  $n + 1$  leaf nodes and  $n$  internal nodes. Hence proved

# More Properties

- Theorem:
  - The number of empty subtrees in a non-empty binary tree is one more than the number of nodes in the tree.
- Proof
  - every node in binary tree  $T$  has two children, for a total of  $2n$  children in a tree of  $n$  nodes
  - Every node except the root node has one parent, for a total of  $n - 1$  nodes with parents
    - there are  $n - 1$  non-empty children
    - Of the  $2n$  children, the remaining  $n+1$  children must be empty
  - Can also be proved using the Full Binary Tree Theorem

# Algorithms on Trees

- Depth is recursively defined
  - If  $v$  is the root, depth of  $v$  is 0
  - Otherwise, depth of  $v$  is one plus depth of parent of  $v$
- **Algorithm**  $\text{depth}(T, v)$ :
  - if**  $T.\text{isRoot}(v)$  **then**
    - return** 0
  - else**
    - return**  $1 + \text{depth}(T, T.\text{parent}(v))$
- Running time:
  - $O(1 + d_v)$ , where is  $d_v$  depth of node  $v$  in  $T$
  - Worst case  $O(n)$



# Algorithms on Trees

- Height is also recursively defined
  - If  $v$  is an external node, height of  $v$  is 0
  - Otherwise, height of  $v$  is one plus height of child of  $v$
- **Algorithm** height( $T, v$ ): //for tree
  - for** each  $v$  in  $T.nodes()$  **do**
  - if**  $T.isExternal(v)$  **then**
  - $h = \max(h, \text{depth}(T, v))$
  - return**  $h$
- Running time:
  - $O(n + \sum_{e \in E} (1 + d_v))$ , where is  $d_v$  depth of node  $v$  in  $T$
  - Worst case  $O(n^2)$

# Algorithms on Trees

- **Algorithm** height2( $T, v$ ):

**if**  $T.isExternal(v)$  **then**

**return** 0

**else**

$h = 0$

**for** each  $w$  in  $T.children(v)$  **do**

$h = \max(h, \text{height2}(T, w))$

**return**  $1+h$

- Running time:

$O(\sum_{v \in T} (1+c_v))$ , where  $c_v$  is order of computation of function  $children(v)$

Worst case  $O(n)$

$\sum_{v \in T} (c_v) = n-1$  ie each node, except root is a child of another node

# Exercise: Properties of Binary trees

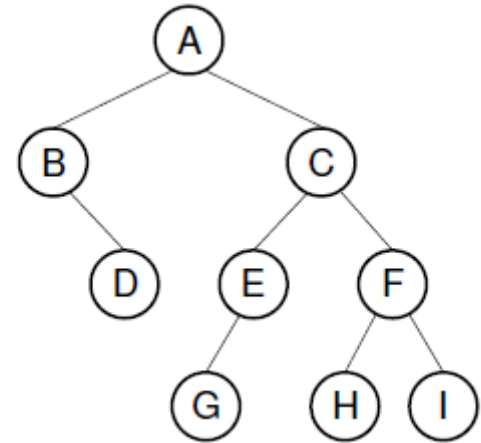
- Draw a binary tree with height 4 and maximum number of external nodes
  - What is the minimum number of external nodes for a binary tree with height  $h$ . Justify
  - What is the maximum number of external nodes for a binary tree with height  $h$ . Justify
  - Let  $T$  be a binary tree with height  $h$  and  $n$  nodes. Show that
    - $\log(n+1)-1 \leq h \leq (n-1)/2$
  - For which values of  $n$  and  $h$  can the above lower and upper bounds on  $h$  be attained with equality

# Tree Traversal

- A traversal of a tree  $T$  is a systematic way of visiting or accessing all the nodes of  $T$
- Types of Traversals
  - Preorder traversal
    - Node is visited before its descendants
  - Postorder traversal
    - Node is visited after its descendants
  - Inorder Traversal
    - node is visited after its left subtree and before its right subtree

# Preorder Traversal

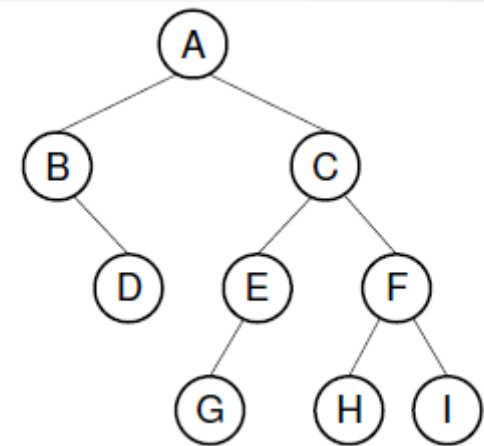
- First node accessed/visited is the root or parent
- Then nodes of the left subtree are visited (in preorder) before any node of the right subtree
- **Algorithm**  $\text{preorder}(T, v)$ 
  - $\text{visit}(v)$
  - for** each child  $w$  of  $v$  **do**
    - $\text{preorder}(T, w)$
- For example shown
  - A B D C E G F H I





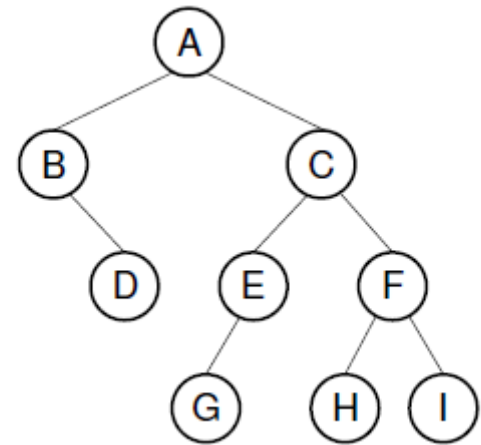
# Postorder Traversal

- First node is visited after its descendants
- Used commonly compute space used by files in a directory and its subdirectories
- **Algorithm**  $\text{postorder}(T, v)$ 
  - for** each child  $w$  of  $v$  **do**
    - $\text{postorder}(T, w)$
  - $\text{visit}(v)$
- For example shown
  - D B G E H I F C A



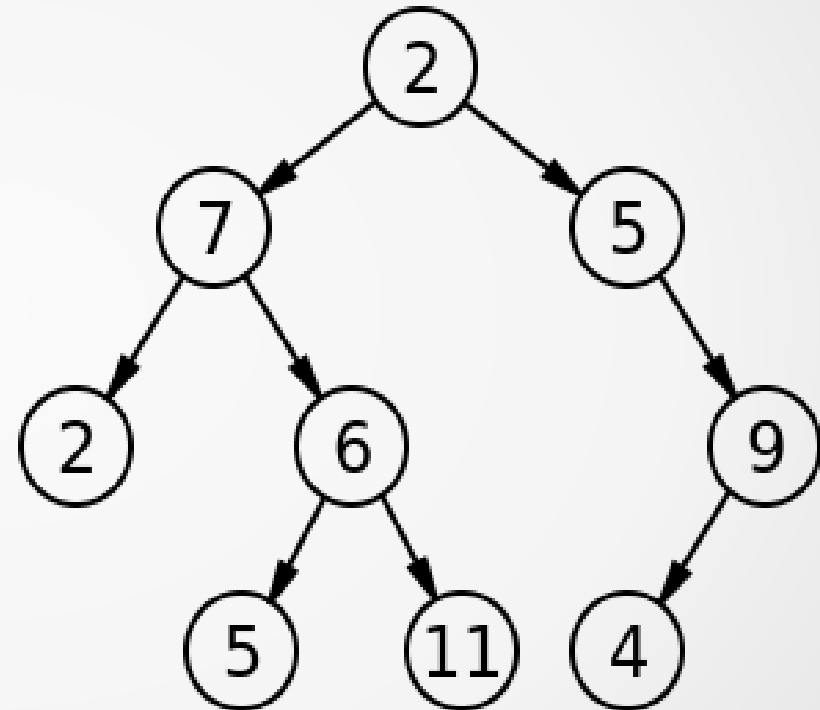
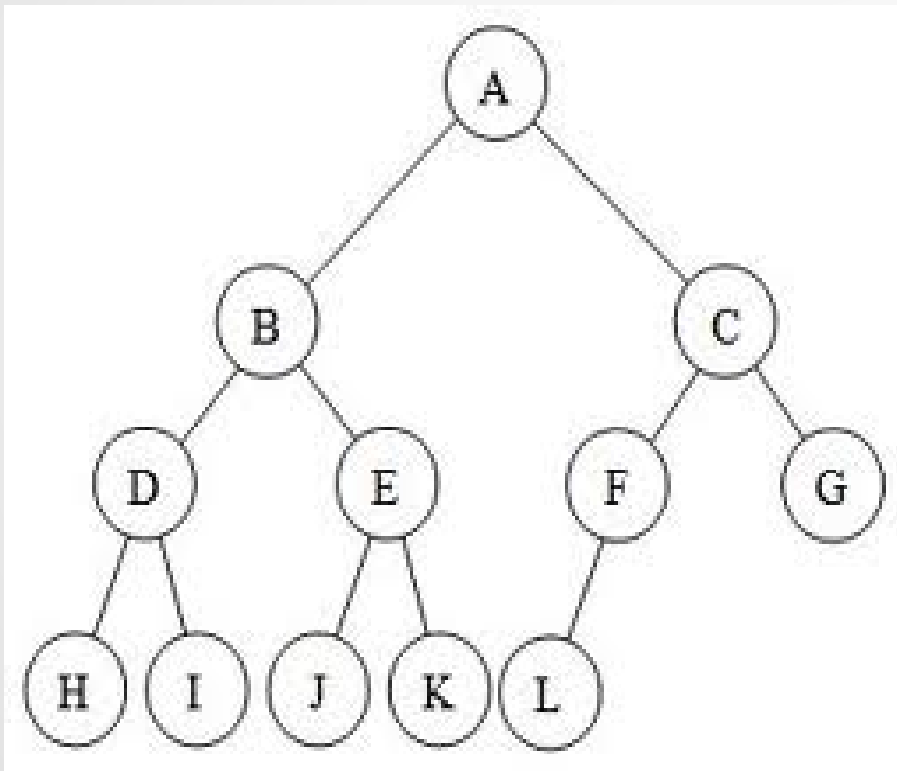
# Inorder Traversal

- First visit the left child (including its entire subtree)
- Then visit the node, and finally visit the right child (including its entire subtree)
- **Algorithm**  $\text{inorder}(T, v)$ 
  - if isInternal ( $v$ )
    - $\text{inOrder}(\text{leftChild}(v))$
  - $\text{visit}(v)$
  - if isInternal ( $v$ )
    - $\text{inOrder}(\text{rightChild}(v))$
- For example shown
  - B D A G E C H F I



# Exercise

- Show the different traversals on the following trees

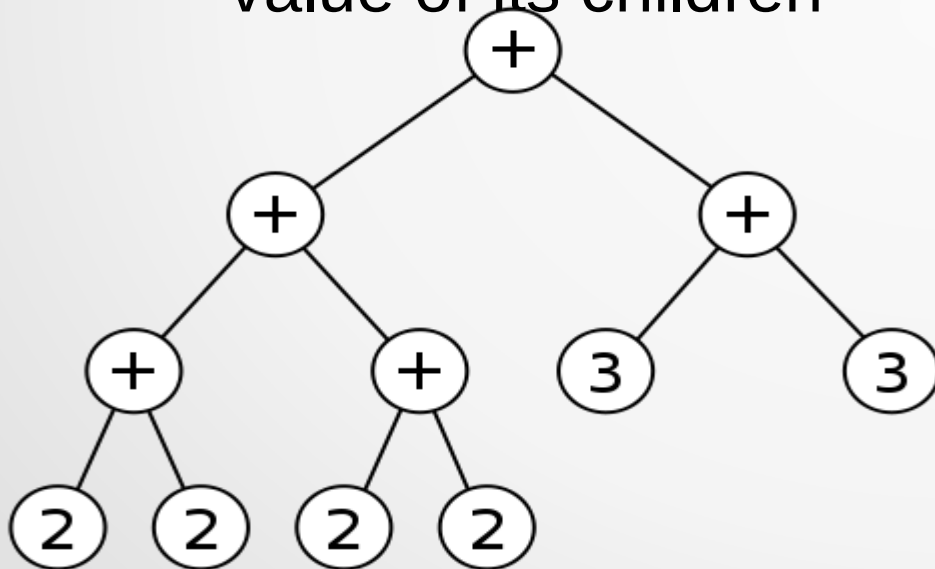


# Exercise

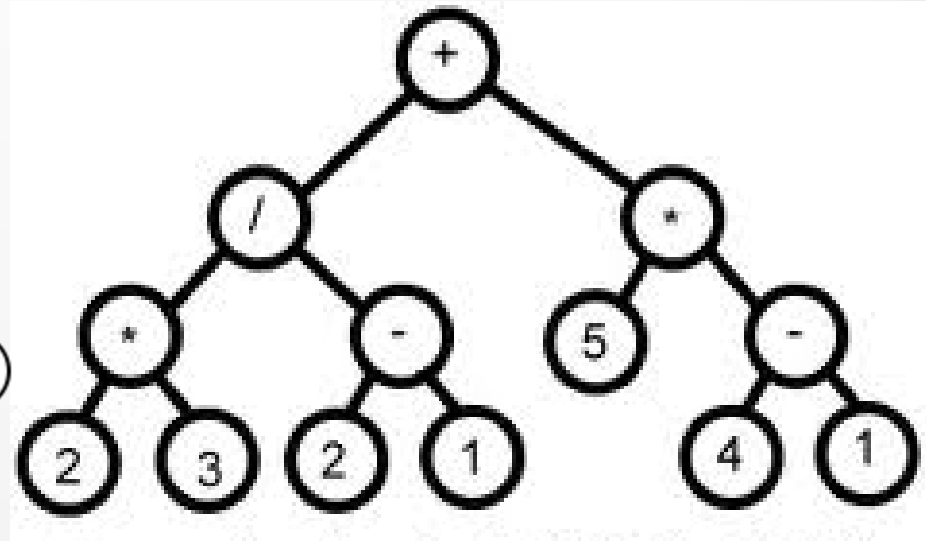
- Draw a binary tree T such that
  - Each internal node of T stores a single character
  - A preorder traversal of T yields EXAMFUN
  - An inorder traversal of T yields MAFXUEN

# Application: Binary Expression Tree

- Arithmetic expression can be represented by a binary tree
  - External nodes are variables or constants
  - Internal nodes are operators
  - Its value is defined by applying its operation to the value of its children



Src: commons.wikimedia.org



Src: public.arnau-sanchez.com

# Printing Arithmetic Expressions

- Uses specialization of inorder traversal
  - print operand or operator when visiting node
  - print “(“ before traversing left subtree
  - print “)” after traversing right subtree
- Algorithm printExpression(v)

if isInternal (v)

print("(")

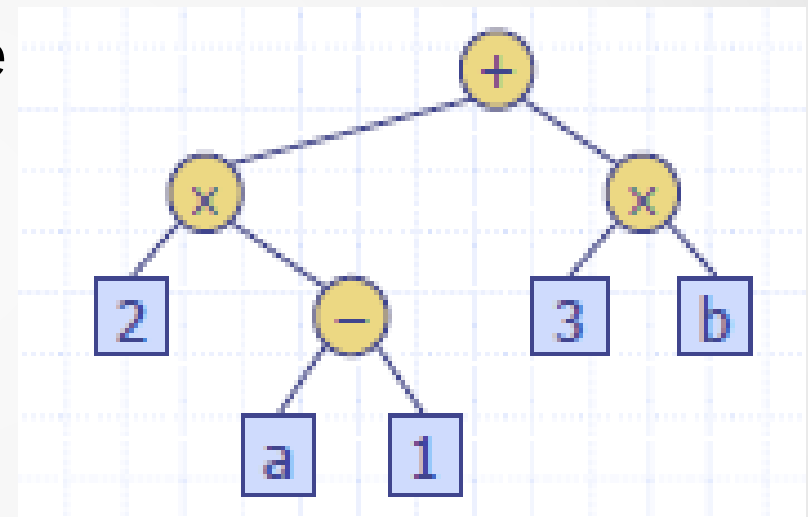
inOrder (leftChild (v))

print(v.element ())

if isInternal (v)

inOrder (rightChild (v))

print (")")



$((2*(a-1))+(3*b))$

Src: goodrich notes

# Evaluate Arithmetic Expressions

- Specialization of a postorder traversal
  - recursive method returning the value of a subtree
  - if an internal node is visited, combine the values of the subtrees

- Algorithm evalExpr(v)

if isExternal (v) return v.element ()

else

$x \leftarrow \text{evalExpr}(\text{leftChild}(v))$

$y \leftarrow \text{evalExpr}(\text{rightChild}(v))$

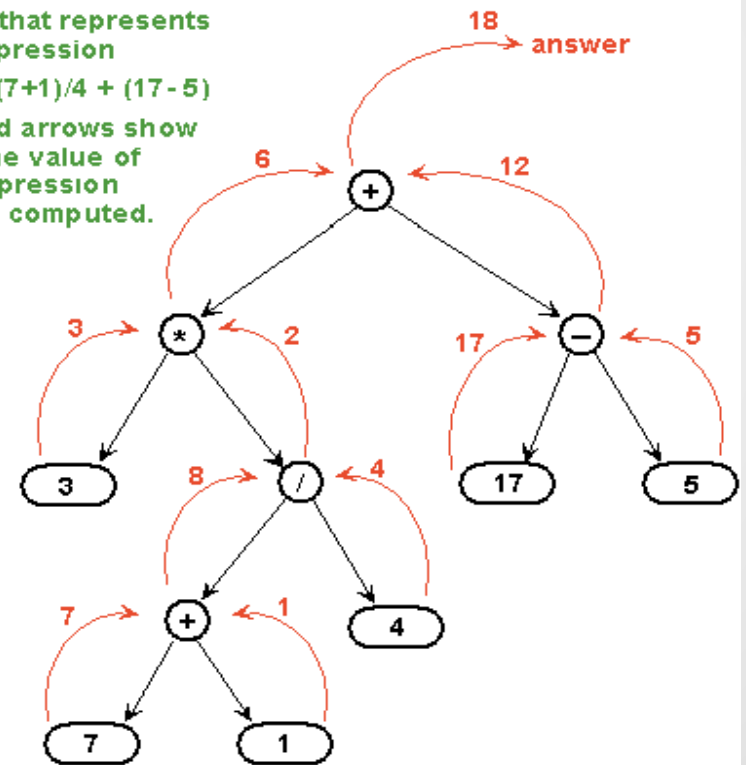
$\diamond \leftarrow \text{operator stored at } v$

return  $x \diamond y$

A tree that represents  
the expression

$3 * (7+1)/4 + (17-5)$

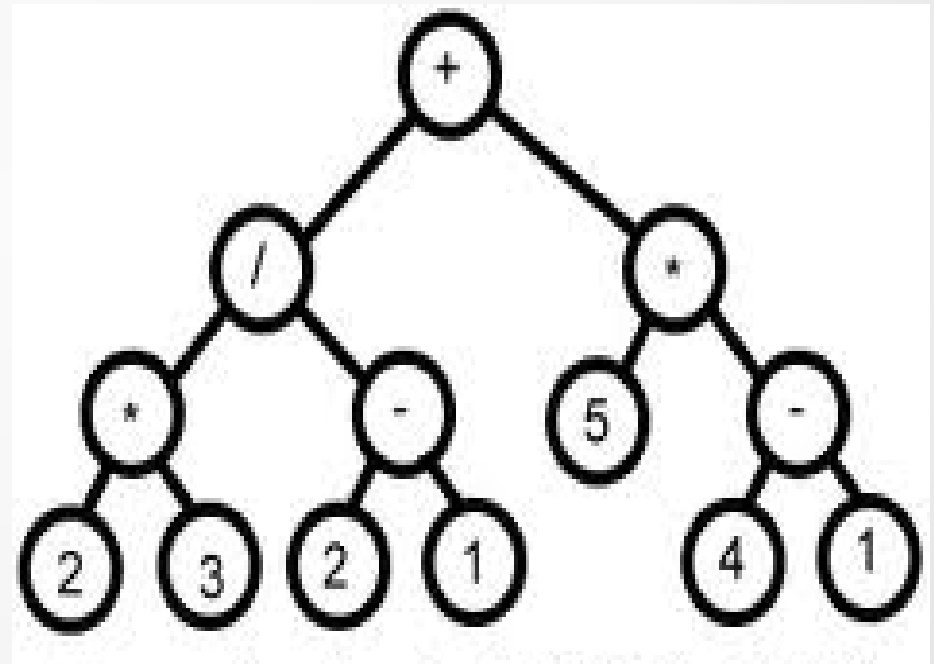
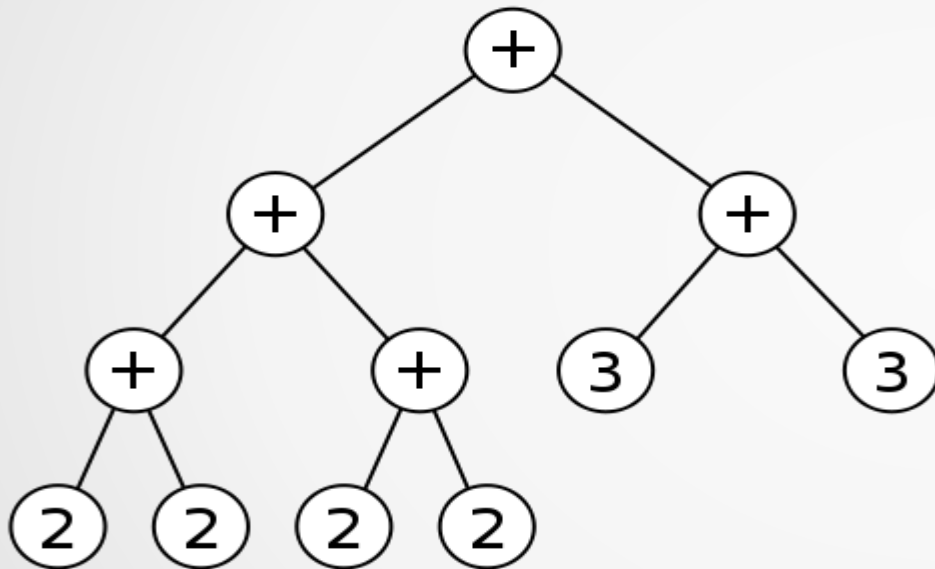
The red arrows show  
how the value of  
the expression  
can be computed.



Src: [math.hws.edu](http://math.hws.edu)

# Exercise

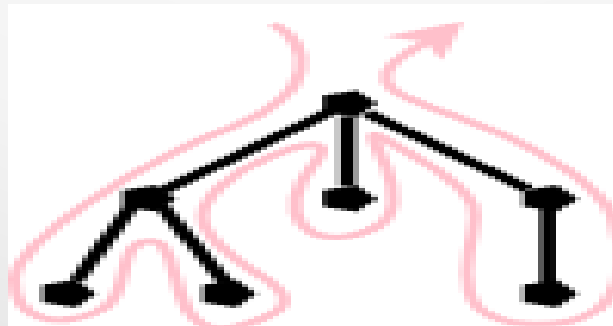
- Print and evaluate the expressions represented by the following trees





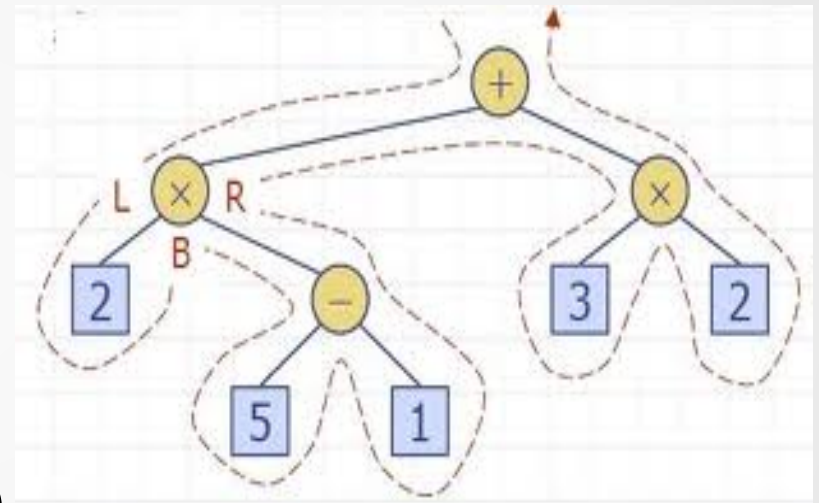
# Euler Tour Traversal

- An Euler tour is a trail in a tree which visits every edge exactly once
  - If it is undirected, it visits once in each direction
  - Walk around T from root towards left child viewing edges of T as being walls that we always keep to our left
- Walk around the tree and visit each node three times:
  - on the left (preorder ie before Euler tour of v's left subtree)
  - from below (inorder ie between Euler's tour of both subtrees)
  - on the right (postorder ie after Euler tour of v's right subtree)



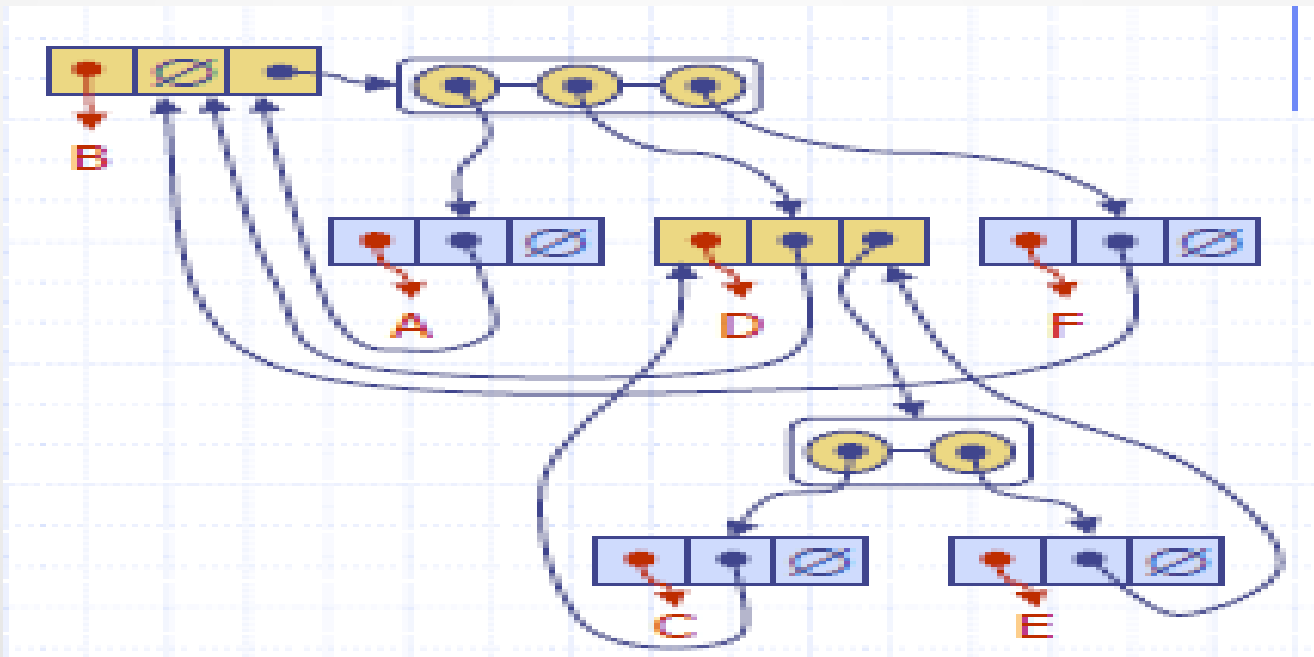
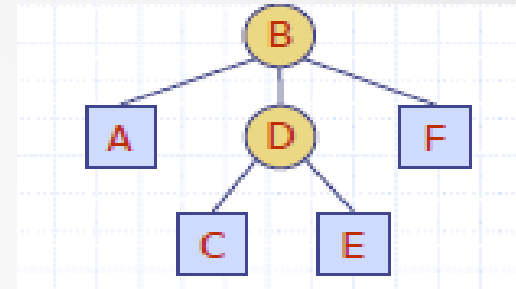
# Euler Tour Traversal

- Algorithm EulerTour( $T, v$ )  
    visitLeft( $T, v$ )  
    if  $T$ .hasLeft( $v$ )  
        eulerTour( $T$ , leftChild( $v$ ))  
    visitBelow( $T, v$ )  
    if  $T$ .hasRight( $v$ )  
        eulerTour( $T$ , rightChild( $v$ ))  
    visitRight( $T, v$ )



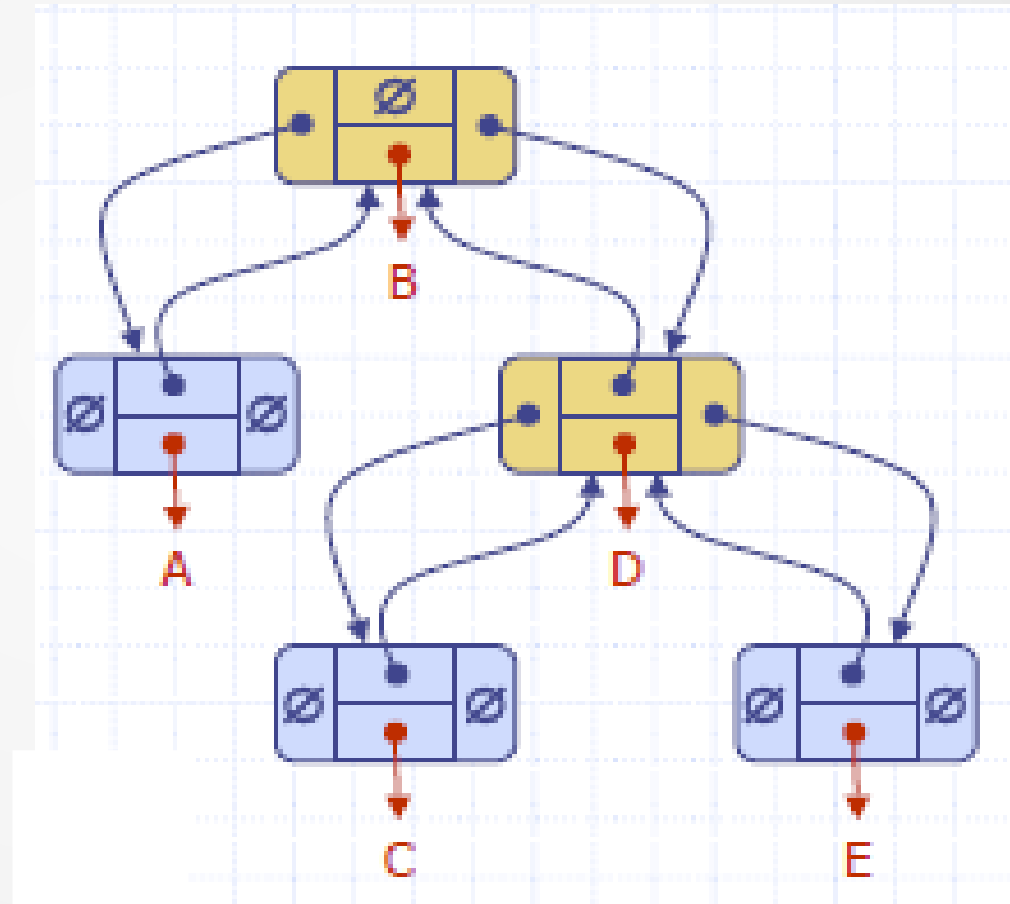
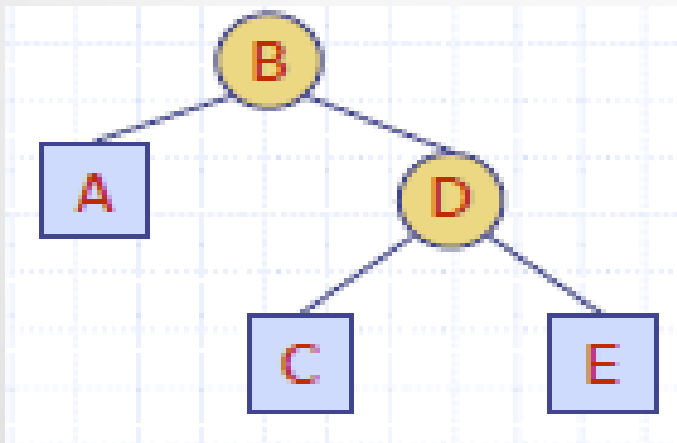
# Data Structure for Trees: Linked Representation

- Node represented as follows (Method 1)
  - Element
  - Parent node
  - Sequence of children nodes



# Linked Representation

- Node represented as follows (Method 2)
  - Element
  - Parent node
  - Left Child Node
  - Right Child Node

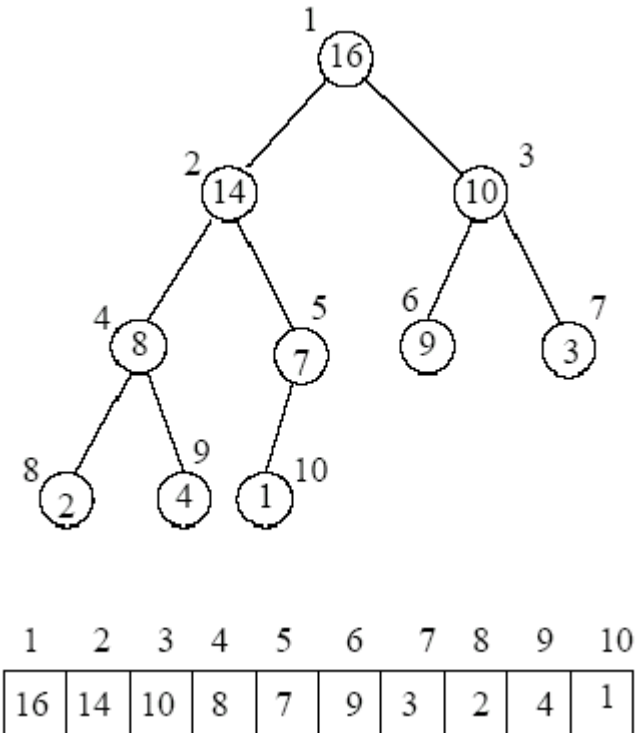


# Vector based representation

- Based on number nodes
- For each node  $v$ ,  $p(v)$  defined as follows
  - If  $v$  is the root,  $p(v) = 1$
  - If  $v$  is left child of node  $u$ , then  $p(v) = 2p(u)$
  - If  $v$  is right child of node  $u$ , then  $p(v) = 2p(u) + 1$
- Numbering function  $p$  is the “level numbering” of the nodes in a binary  $T$ 
  - Numbers in each level increase from left to right
  - Numbers can be skipped

# Vector based representation

- The different functions can be computed using arithmetic operations
- Extendible array representations can be used to manage the size



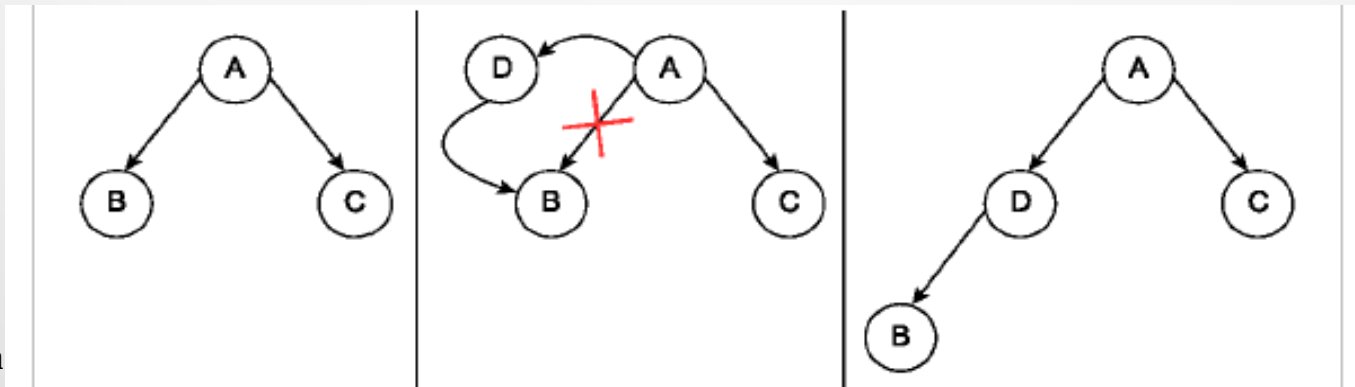
Src: [www.cse.hut.fi](http://www.cse.hut.fi)

# Binary Tree ADT

- leftChild(v):
  - Return left child of v
  - Error if v is external node
- rightChild(v):
  - Return right child of v
  - Error if v is external node
- sibling(v)
  - Return sibling of node v
  - Error if v is the root

# Insertion and Deletion

- Insertion: InsertAfter(node A)
  - A is an internal node
    - Let node B be child of A
    - A assigns its child to the new node and the new node assigns its parent to A.
    - New node assigns its child to B and B assigns its parent as the new node
  - A is an external node
    - A assigns the new node as one of its children
    - The new node assigns node A as its parent.





# Deletion

- Delete(node A)
  - A is an external node
    - set the corresponding child of A's parent to null
  - A has one child
    - set the parent of A's child to A's parent
    - set the child of A's parent to A's child.
  - A has two children
    - Usually not recommended
    - Involves more complex operations

