

CSI2110

Data Structures and Algorithms

Prof. WonSook Lee



CSI2110

Data Structures and Algorithms

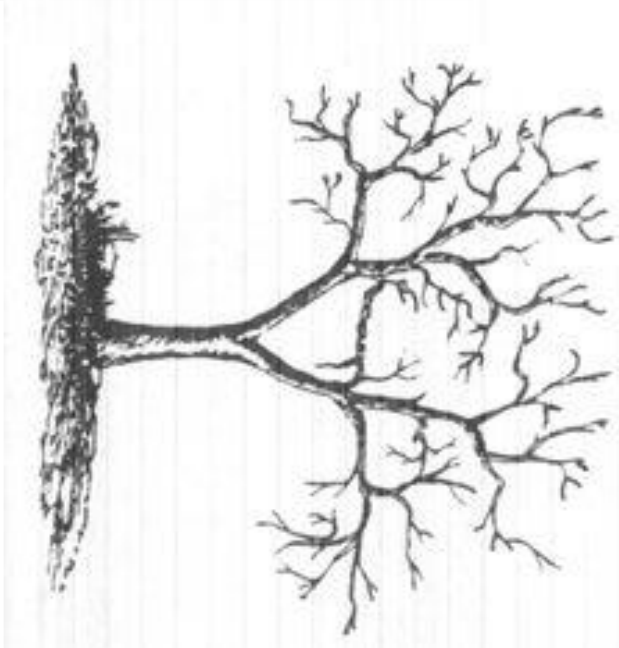
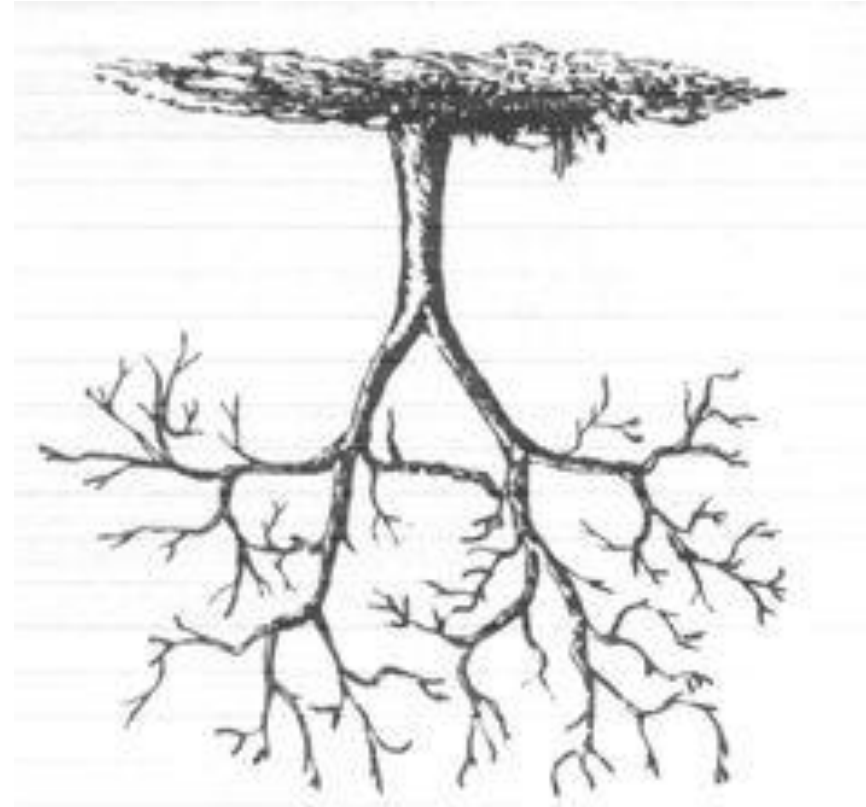
Prof. WonSook Lee

Trees

- Trees
- Binary Trees
- Properties of Binary Trees
- Traversals of Trees
- Data Structures for Trees

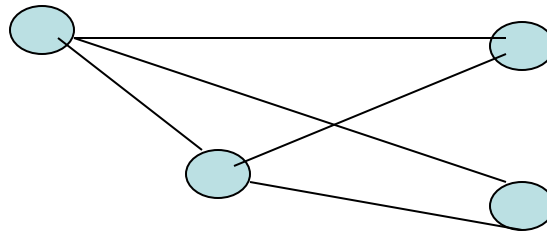


a Tree



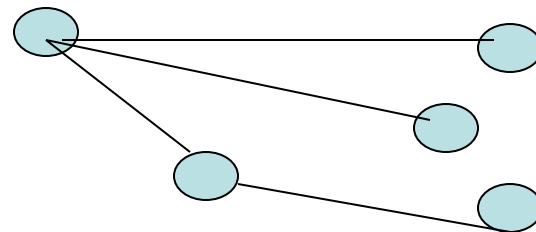
Trees

A **graph** $G = (V, E)$ consists of a set V of VERTICES
and a set E of edges, with $E = \{(u, v): u, v \in V, u \neq v\}$



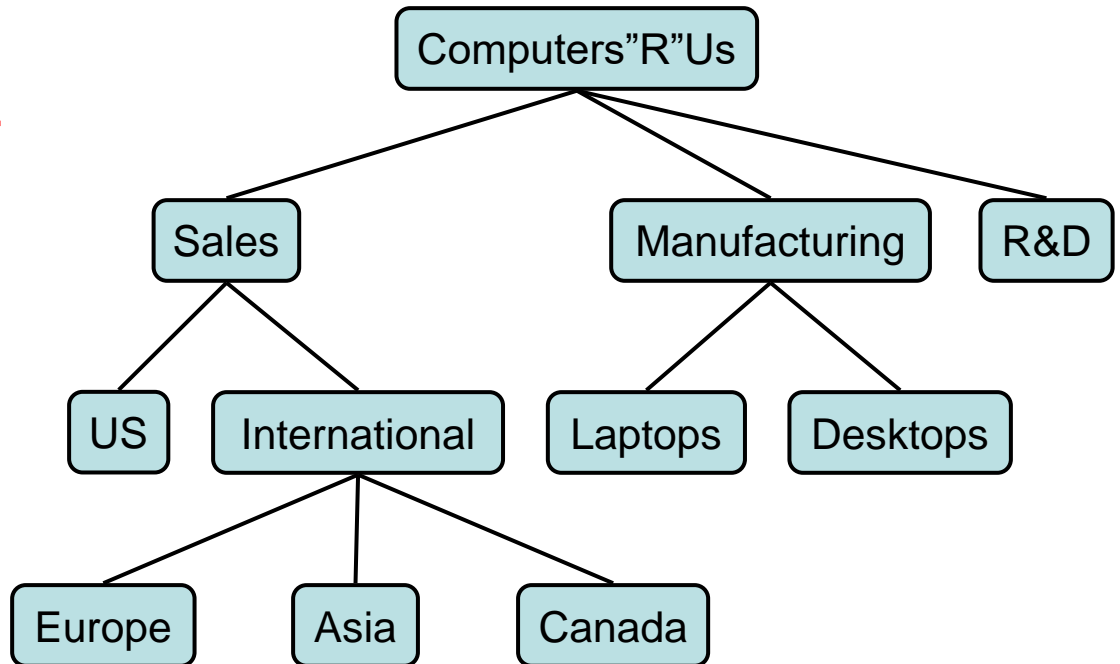
A **tree** is a connected graph with no cycles.

→ \exists a path between each pair of vertices.

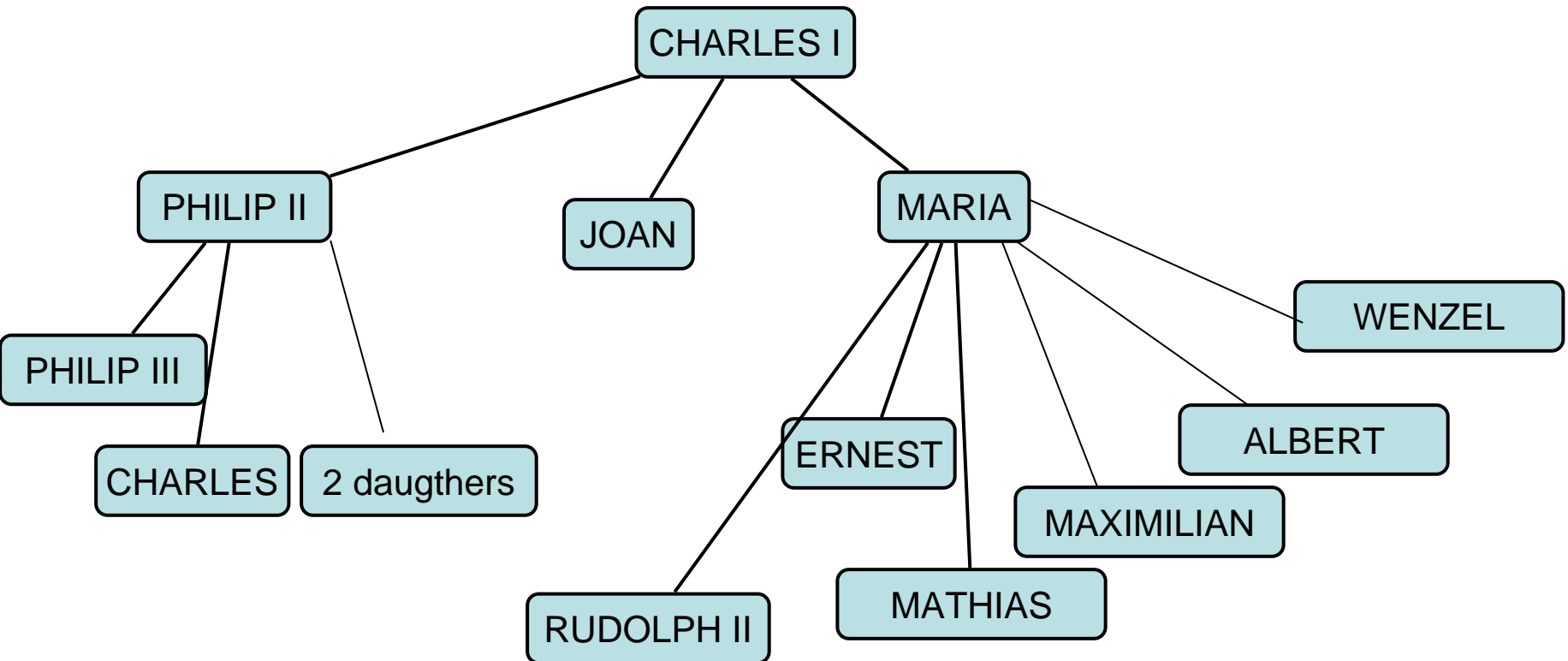


What is a Tree

- Abstract model of a hierarchical structure
- A tree consists of nodes with a parent-child relation
- Applications:
 - Organization charts
 - File systems
 - Programming environments



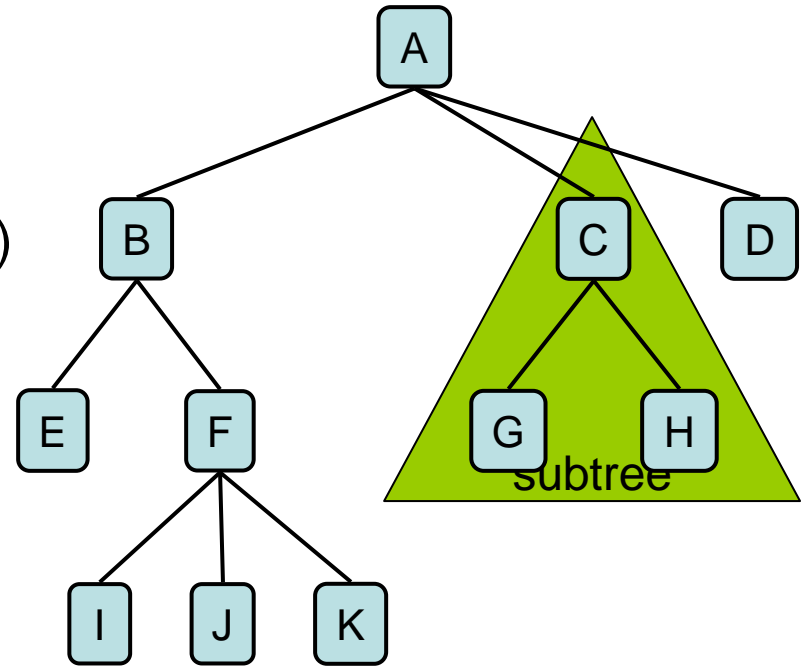
Example: Genealogical Tree



Hasburg Family

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

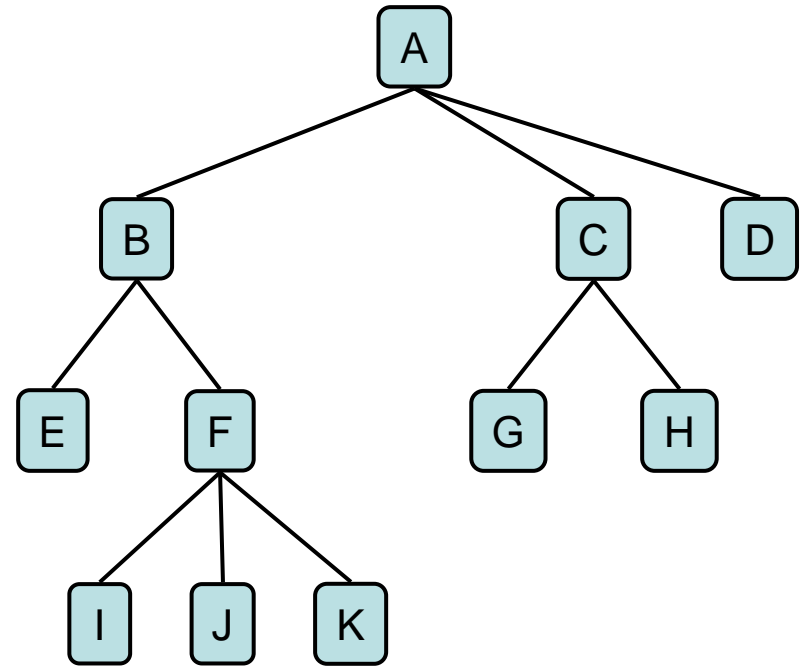


Tree Terminology

Distance between two nodes: number of “edges” between them

- **Depth** of a node: number of ancestors (= distance from the root)

- **Height** of a tree: maximum depth of any node (3)



ADTs for Trees

- generic container methods
 - `size()`, `isEmpty()`, `elements()`
- positional container methods
 - `positions()`, `swapElements(p,q)`, `replaceElement(p,e)`
- query methods
 - `isRoot(p)`, `isInternal(p)`, `isExternal(p)`
- accessor methods
 - `root()`, `parent(p)`, `children(p)`
- update methods
 - application specific

Computing the depth of a node

If v is the root the depth is 0

If v is an internal node the depth is $1 +$ the depth of its parent

Algorithm $\text{depth}(T, v)$

if $T.\text{isRoot}(v)$ then

 return 0

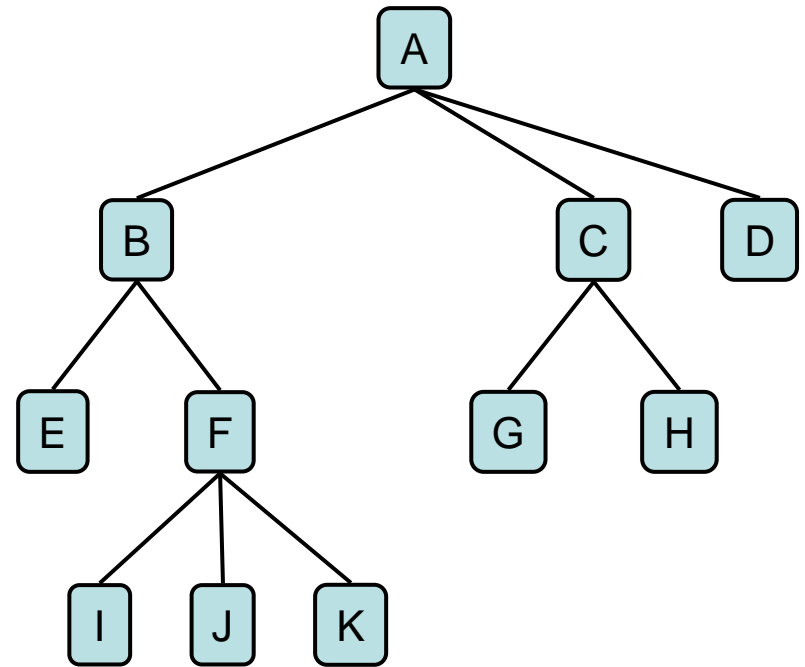
else

 return $1 + \text{depth}(T, T.\text{parent}(v))$

Complexity ?

Now, Traversing a tree!

How to visit all the nodes in a tree?

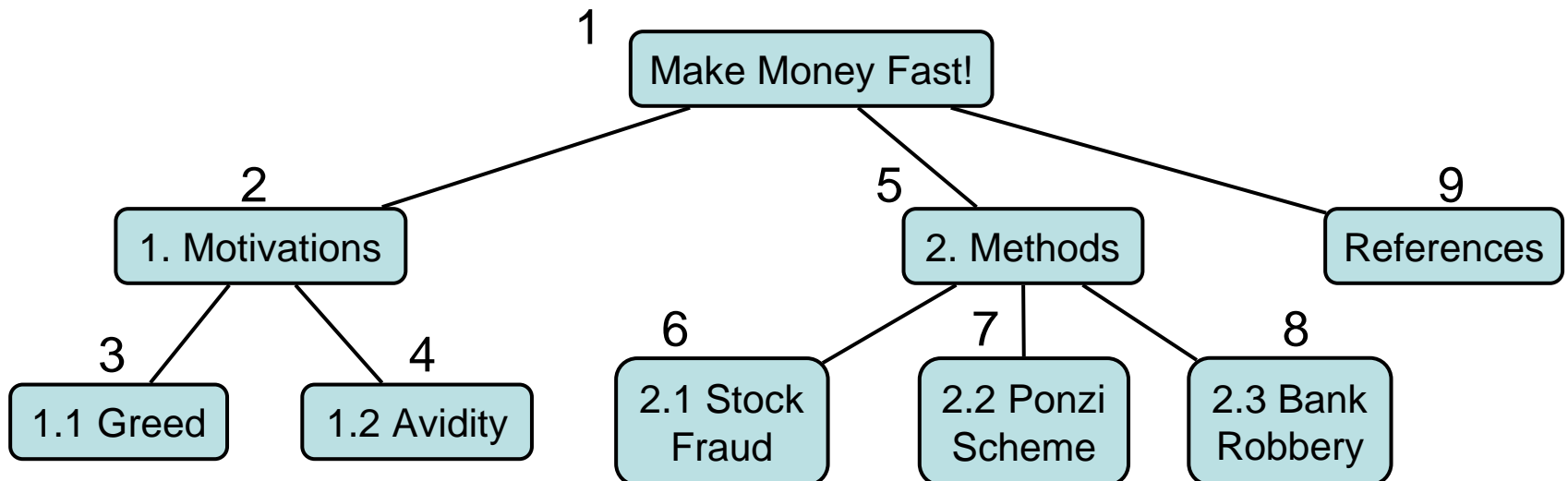


Traversing Trees

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**
- Application: print a structured document

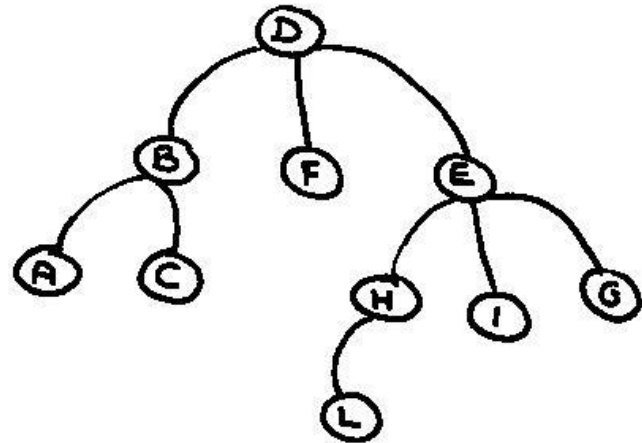
```
Algorithm preOrder(v)  
  visit(v)  
  for each child w of v  
    preOrder (w)
```



Traversing Trees

Preorder Traversal

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



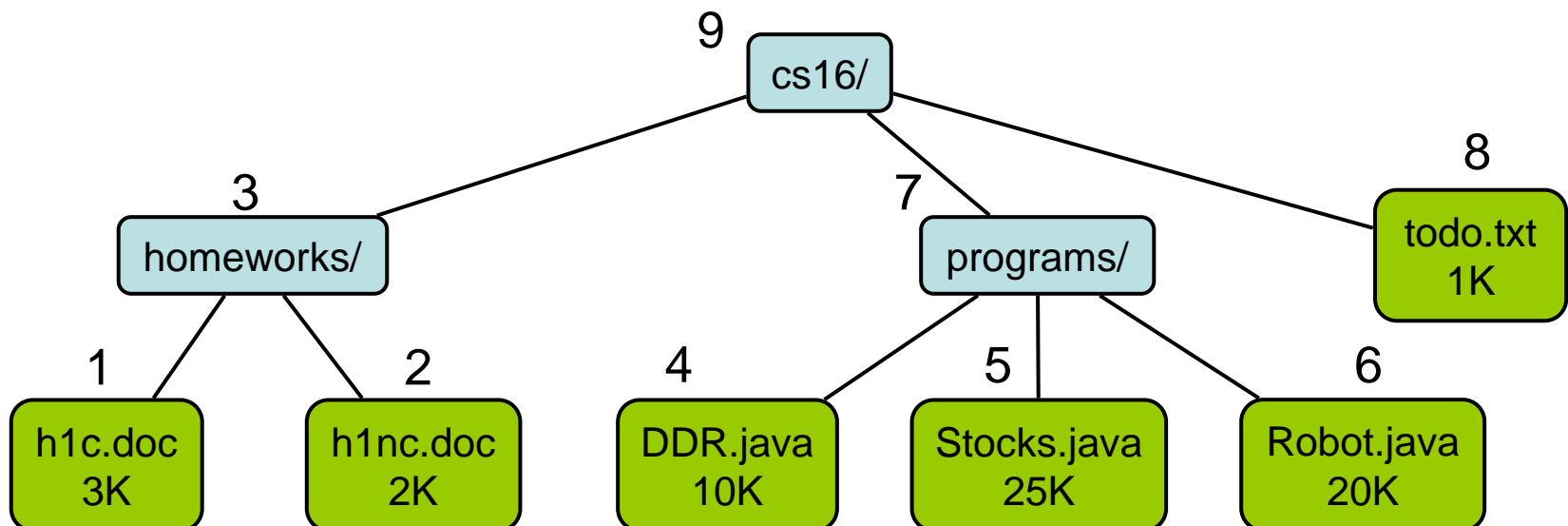
D B A C F E H L I G

Traversing Trees

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)
```

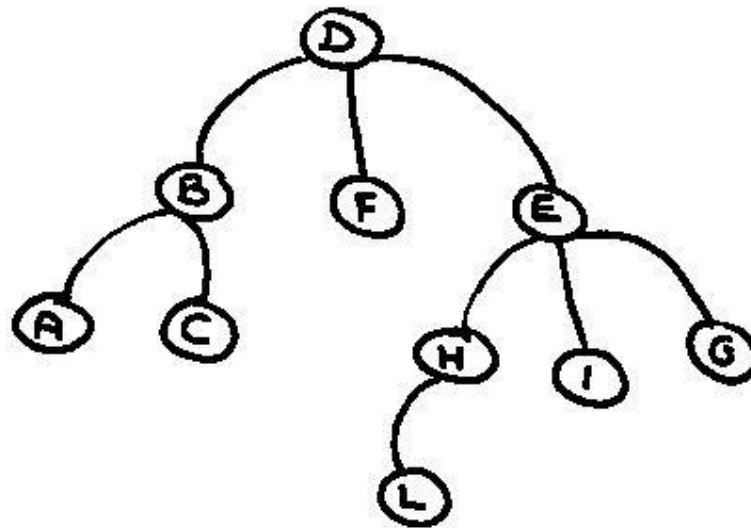


Traversing Trees

Postorder Traversal

Algorithm `postOrder(v)`
 for each child `w` of `v` **do**
 recursively perform `postOrder(w)`
 “visit” node `v`

A C B F L H I G E D



Traversing Trees

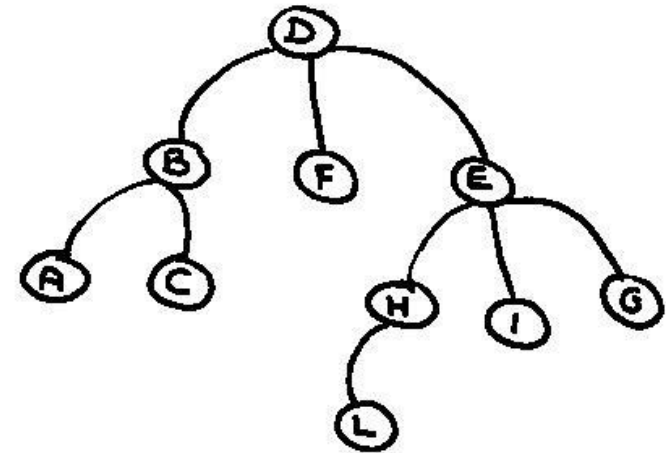
Inorder Traversal of a tree (Depth-first)

Let $d(x)$ be the number
of sub-trees of node x .

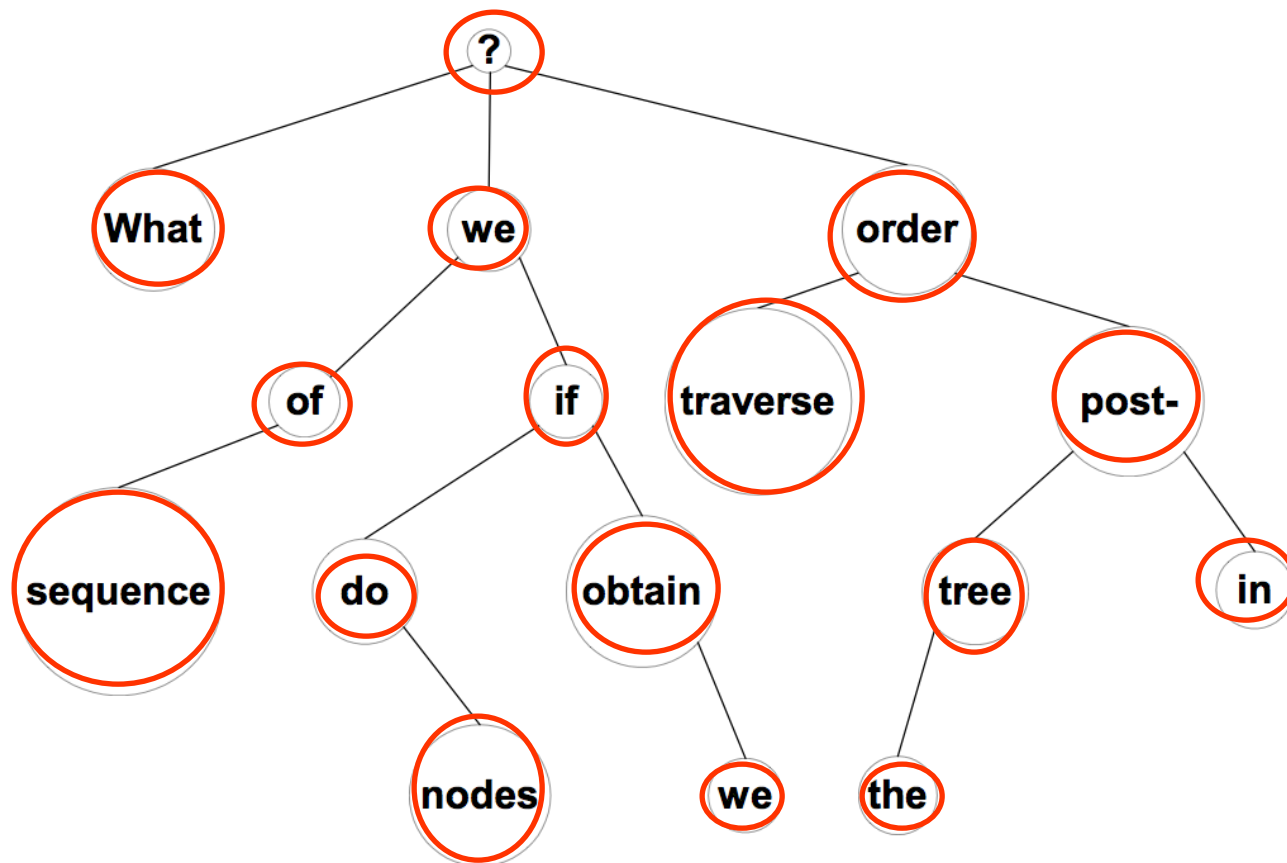
Start: $x = \text{root}$

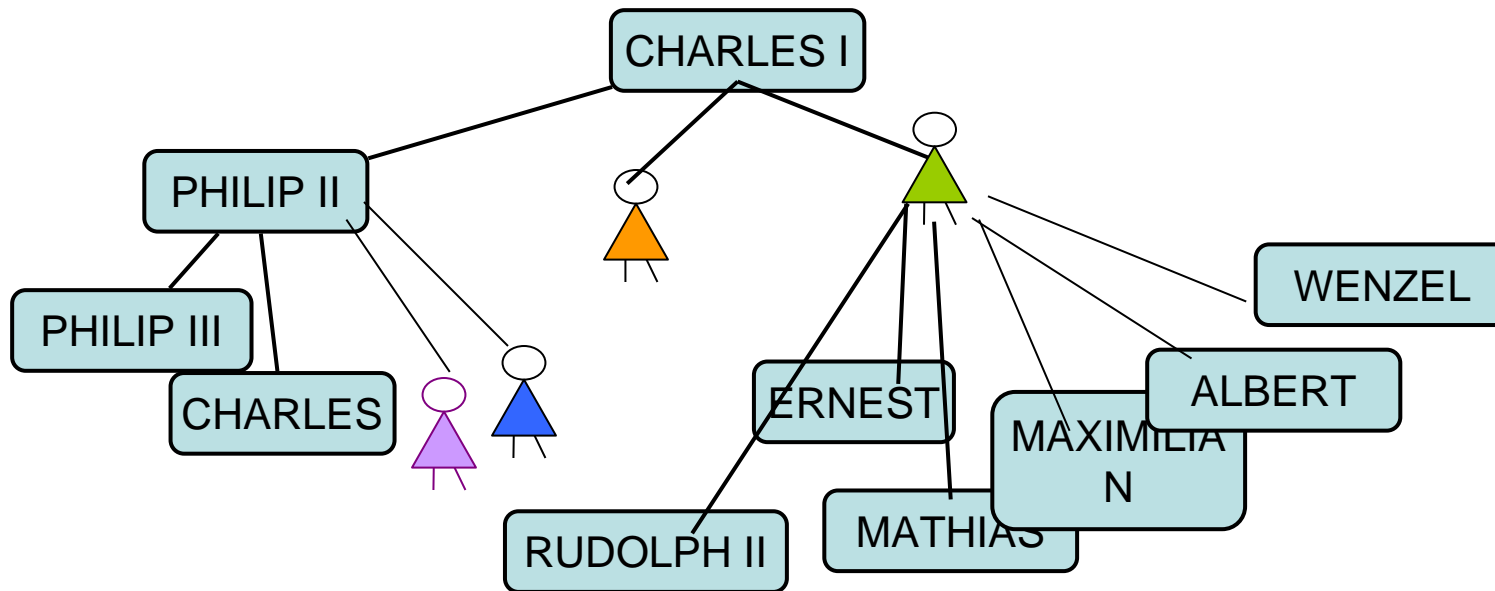
IN-ORDER VISIT

1. Visit the first sub-tree (inorder)
2. Visit the root
3. Visit the second sub-tree (inorder)
- \vdots
- $d(x)+1$. Visit the $d(x)^{\text{th}}$ sub-tree (inorder)



A B C D F L H E I G





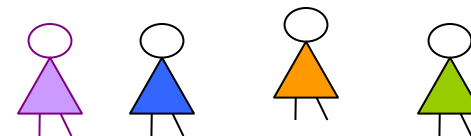
When Charles dies, Philip II becomes King.
If Philip II dies as well

Charles I,

Philip II,

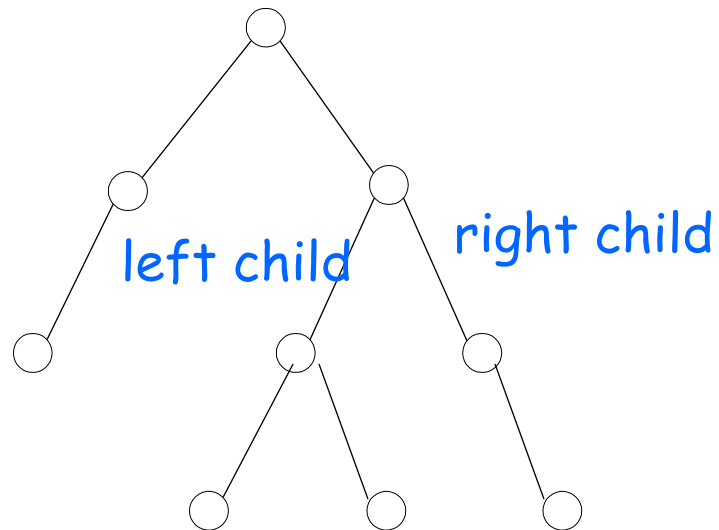
Philip III,

Charles,



Rudolph II, Ernest, Mathias, Max, Albert, Wenzel,

Binary Trees



Children are ordered

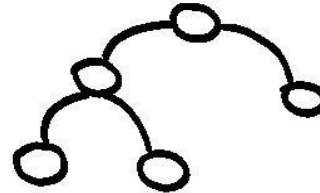
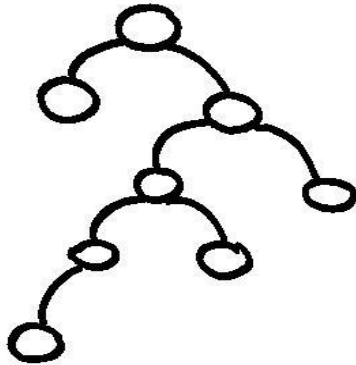
Each node has at most two children:

[0, 1, or 2]

"Full" Binary Trees (or "Proper")

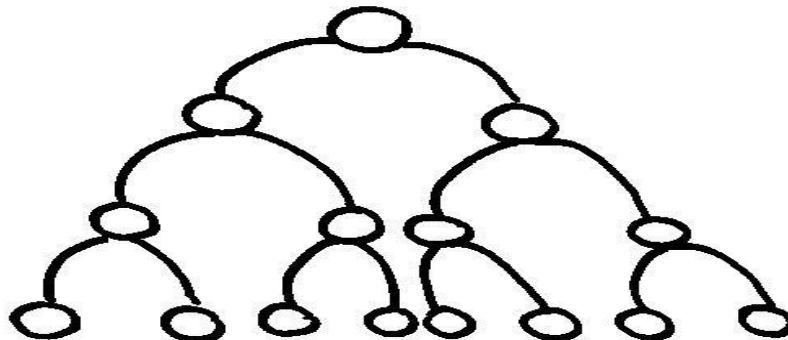
Each node:

{ is a leaf, or
has two children



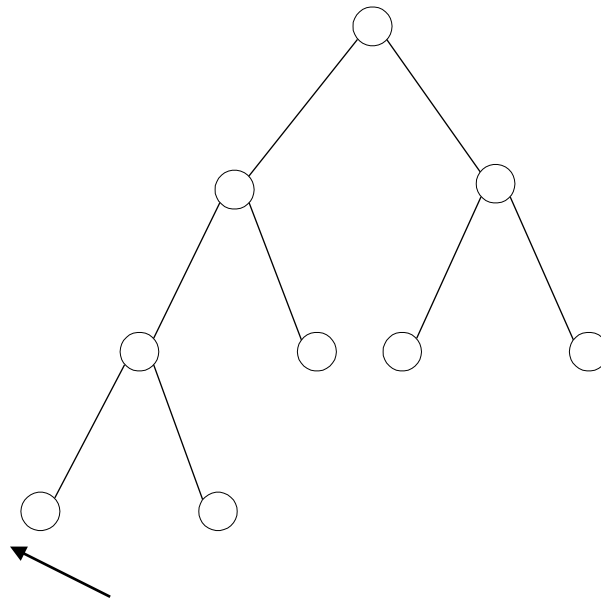
Perfect Binary Trees

Full binary trees with all leaves at the same level:

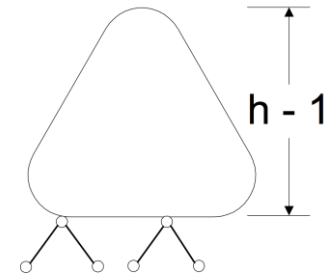


Complete Binary Trees

of depth h = Perfect trees of depth $(h-1)$
+
one or more leaves at level h .



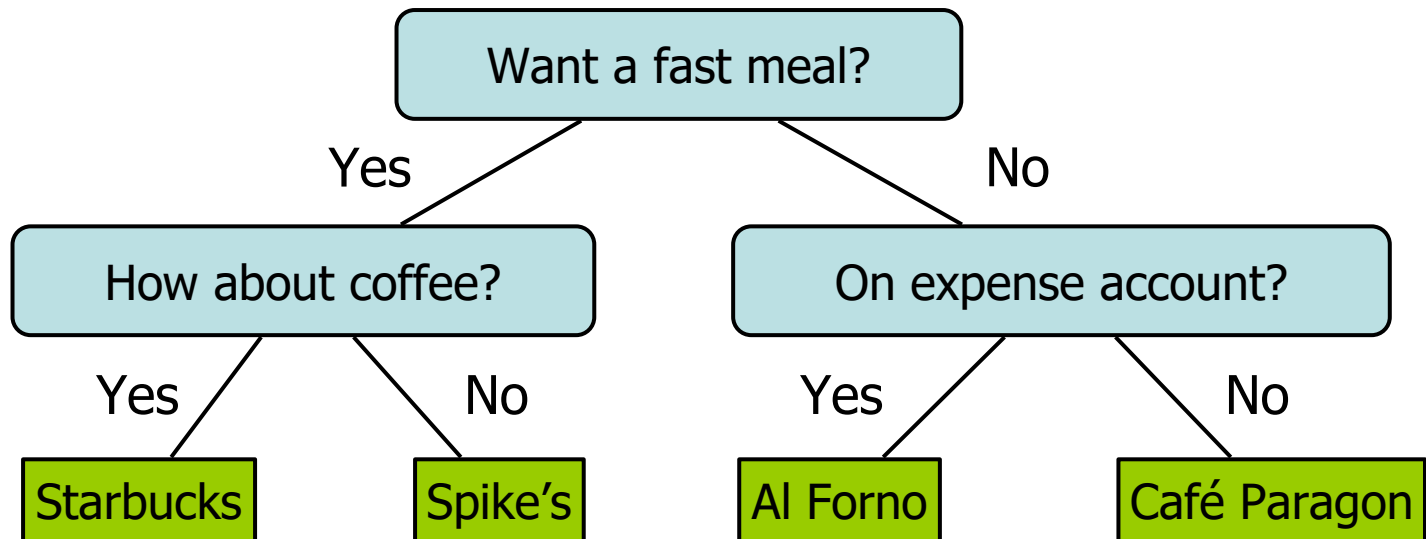
Leaves go at the left



Examples of Binary Trees

Decision Tree

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



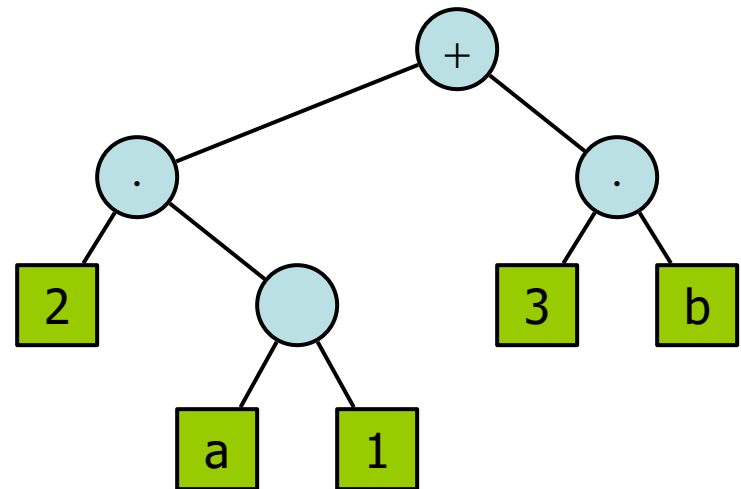
Examples of Binary Trees

Arithmetic Expression Tree

- Binary tree associated with an arithmetic expression
 - internal nodes: operators
 - external nodes: operands

Example: arithmetic expression
tree for the expression

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

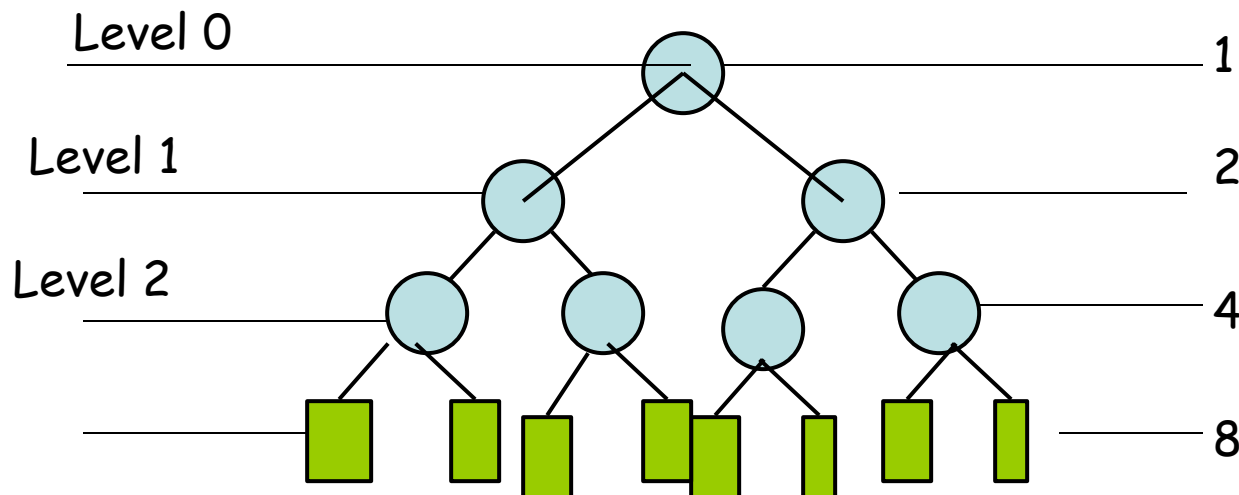


Properties of Binary Trees

- Notation

n	# of nodes	e	# of leaves
i	# of internal nodes	h	height

Maximum number of
nodes at each level ?



level i ----- 2^i

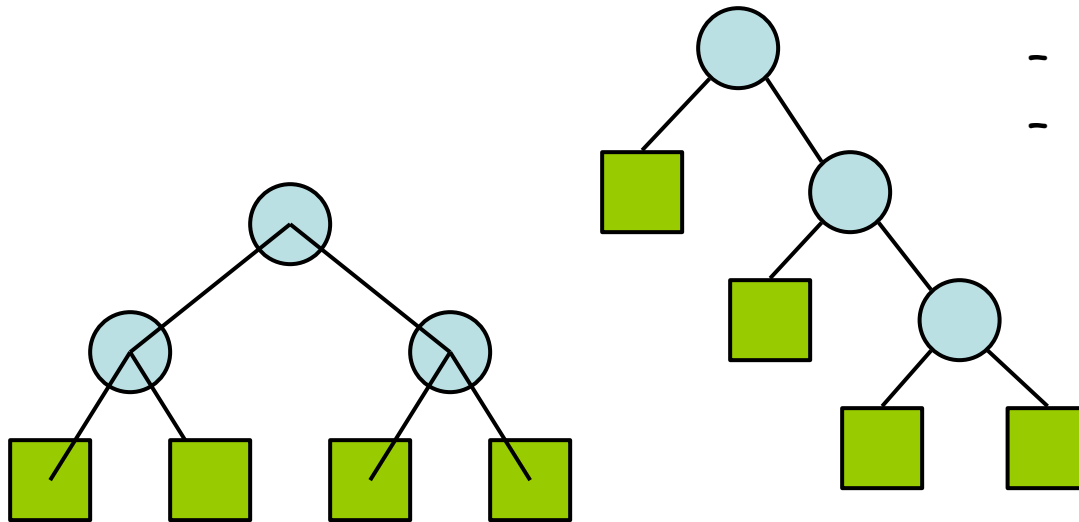
Properties of Full Binary Trees

- Notation

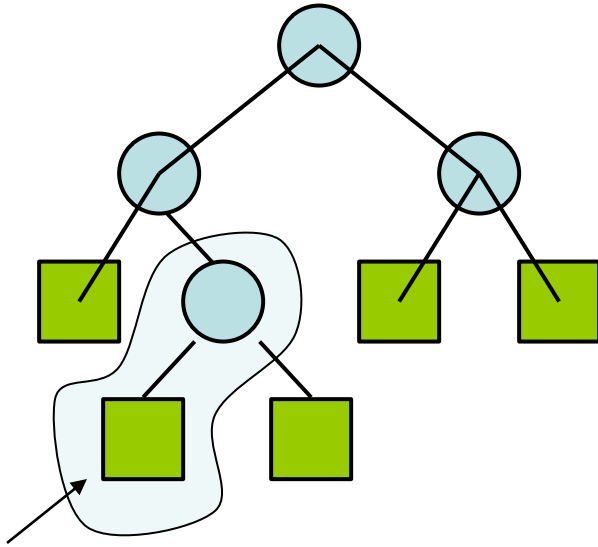
n number of nodes
 e number of leaves
 i number of internal nodes
 h height

- Properties:

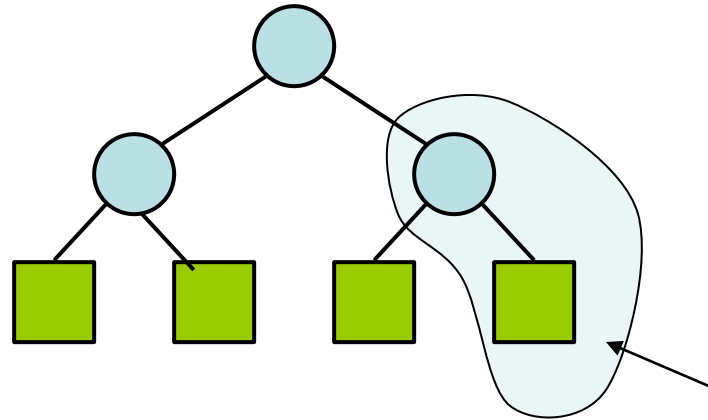
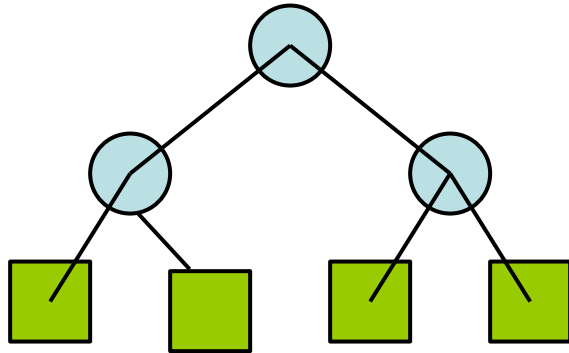
- $e = i + 1$
- $n = 2e - 1$
- $h \leq i$
- $h \leq (n - 1)/2$
- $e \leq 2^h$
- $h \geq \log_2 e$
- $h \geq \log_2 (n + 1) - 1$



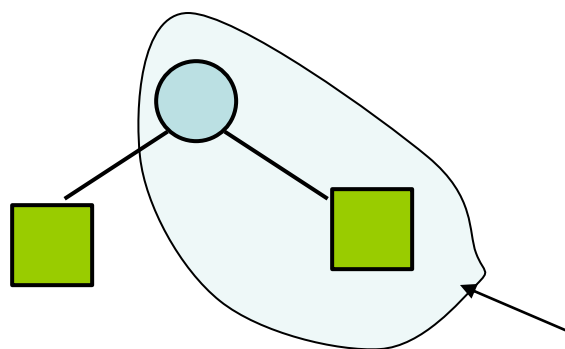
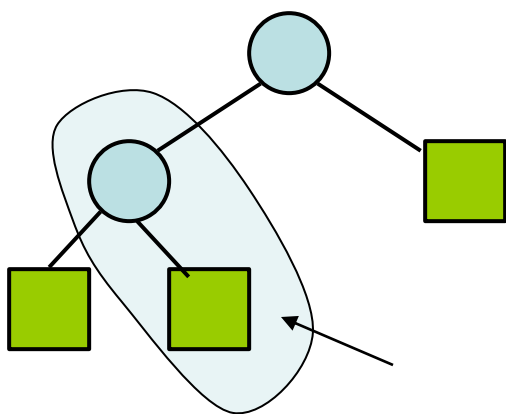
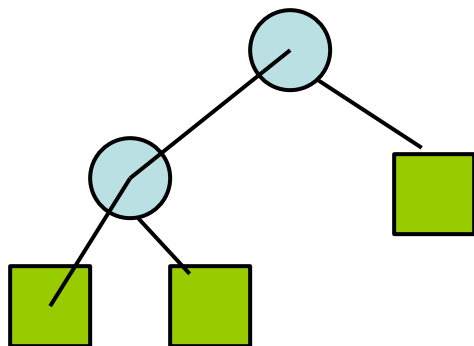
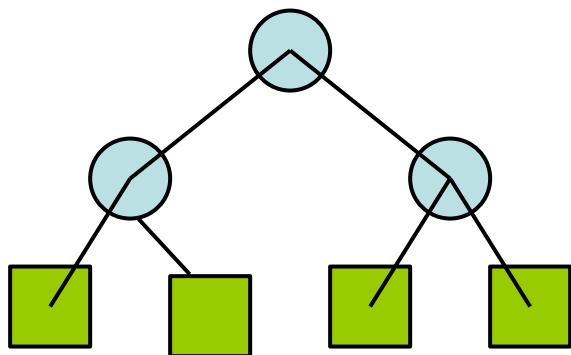
$$e = i + 1$$



$$e = i + 1$$



$$e = i + 1$$



$$n = 2e - 1$$

$$n = i + e$$

$$e = i + 1 \text{ (just proved)}$$

$$i = e - 1$$

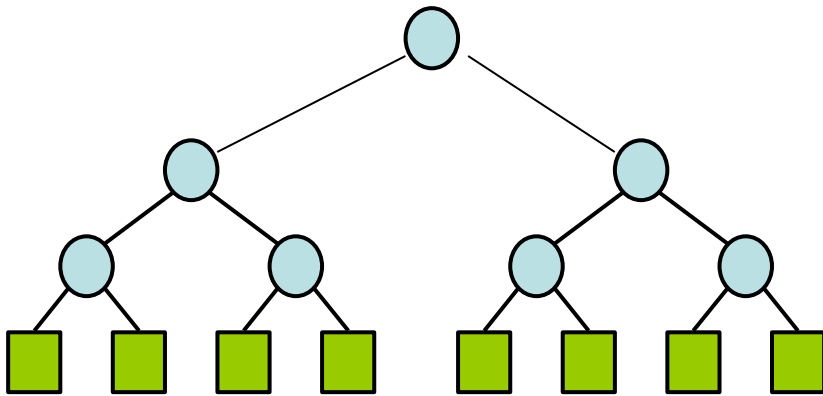
$$n = 2e - 1$$

$$h \leq i$$

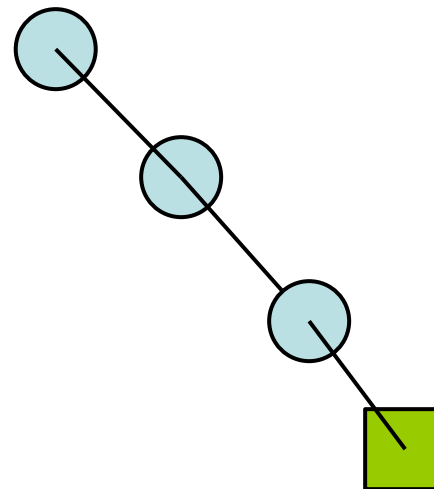
(h = max num of ancestors)

There must be at least one internal node for each level
(except the last) !

Ex: $h=3, i=7$

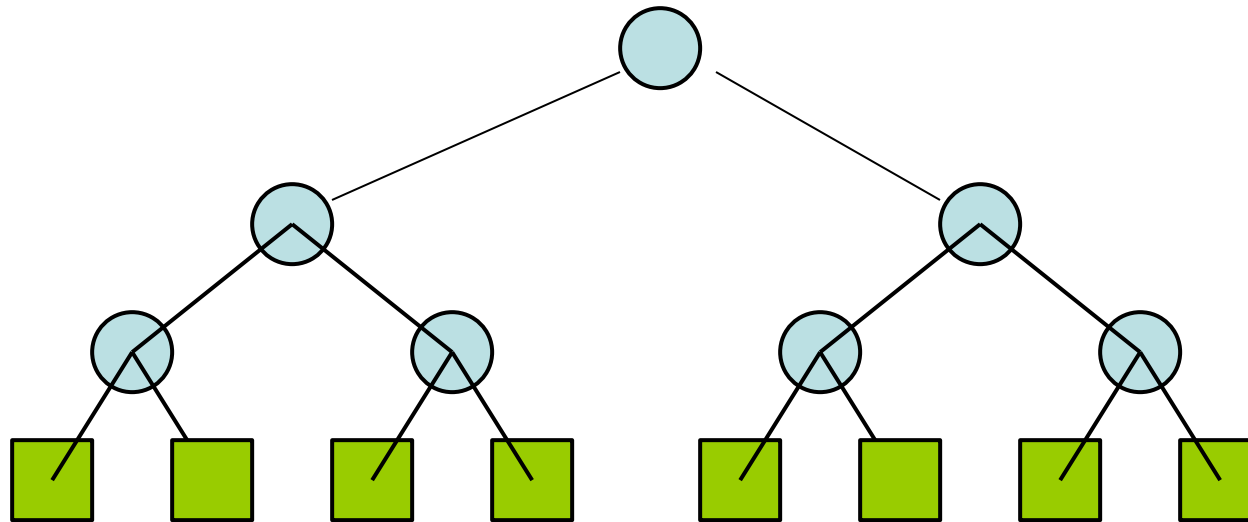


Ex: $h=3, i=3$



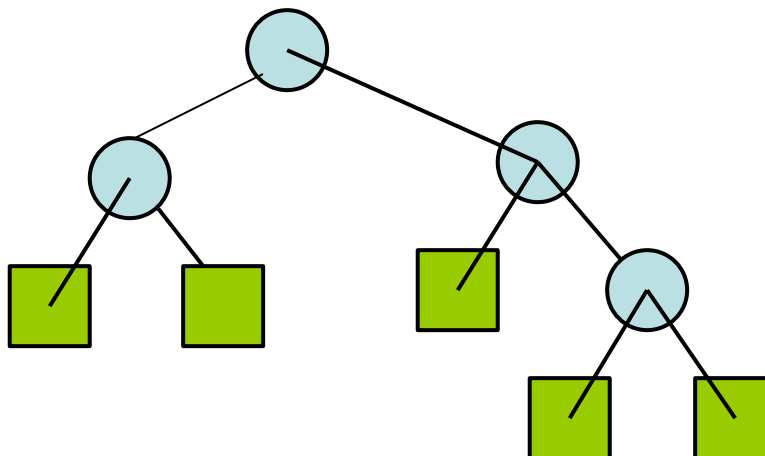
$$e \leq 2^h$$

level i ----- max num of nodes is 2^i



$h = 3$

2^3 leaves
if all at last
level h



otherwise less

Since $e \leq 2^h$

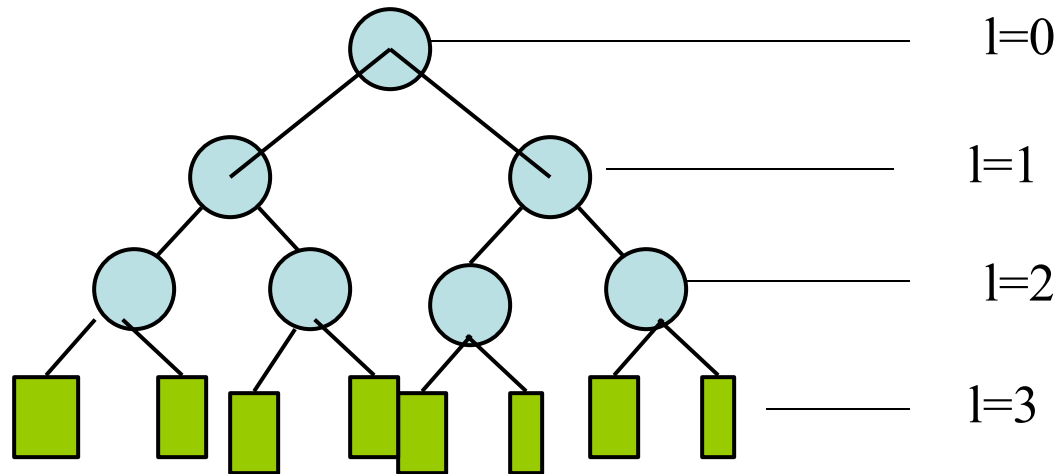
$$\log_2 e \leq \log_2 2^h$$

$$\log_2 e \leq h$$

$$h \geq \log_2 e$$

In Perfect Binary Trees...
with height h there are $2^{h+1} - 1$ nodes

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



At each level there are 2^l nodes, so the tree has:

$$\sum_{l=0}^h 2^l = 1 + 2 + 4 + \dots + 2^h = 2^{h+1} - 1$$

As a consequence:

In **Binary trees**:

obviously $n \leq 2^{h+1} - 1$

$$n \leq 2^{h+1} - 1$$

$$n+1 \leq 2^{h+1}$$

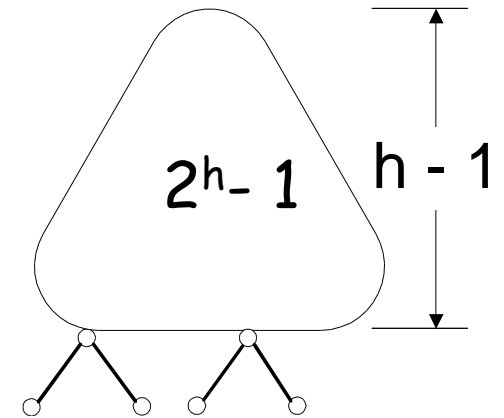
$$\log(n+1) \leq h+1$$

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

Complete Binary Trees

A complete binary tree of height h is composed by a perfect binary tree of height $h-1$ plus some leaves.

$$\longrightarrow 2^h \leq n \leq 2^{h+1} - 1$$



$$2^h \leq n \leq 2^{h+1} - 1$$

$$2^h \leq n < 2^{h+1}$$

$$h \leq \log_2(n) < h+1$$

h is an integer $\Rightarrow h = \text{integer part of } \log_2(n)$

$$\lfloor \log n \rfloor$$

Summary of Important Properties

log is
 \log_2

Binary Trees

$$h + 1 \leq n \leq 2^{h+1} - 1$$

$$1 \leq e \leq 2^h$$

$$h \leq i \leq 2^h - 1$$

$$\log(n+1) - 1 \leq h \leq n-1$$

Full Binary Trees

$$2h + 1 \leq n \leq 2^{h+1} - 1$$

$$h+1 \leq e \leq 2^h$$

$$h \leq i \leq 2^h - 1$$

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

Binary Trees: properties of the height

Height h of a tree:

✓ Binary:

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

✓ Binary - Full :

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

✓ Binary - Complete :

$$n \geq 2^h \quad h = \text{floor}(\log n)$$

(integer part of $\log n$)

✓ Binary - perfect:

$$n = 2^{h+1} - 1 \quad h = \log(n+1) - 1$$

ADTs for Binary Trees

- accessor methods
 - leftChild(p), rightChild(p), sibling(p)
- update methods
 - expandExternal(p), removeAboveExternal(p)

other application specific methods

Traversing Binary Trees

Pre-, post-, in- (order)

- Refer to the place of the parent relative to the children
- **pre** is before: parent, child, child
- **post** is after: child, child, parent
- **in** is in between: child, parent, child

Traversing Binary Trees

Preorder, Postorder,

Algorithm preOrder(T,v)

visit(v)

if v is internal:

preOrder (T,T.LeftChild(v))

preOrder (T,T.RightChild(v))

Algorithm postOrder(T,v)

if v is internal:

postOrder (T,T.LeftChild(v))

postOrder(T,T.RightChild(v))

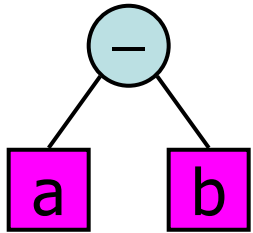
visit(v)

Traversing Binary Trees

Inorder
(Depth-first)

```
Algorithm inOrder(T,v)
    if v is internal:
        inOrder (T,T.LeftChild(v))
    visit(v)
    if v is internal:
        inOrder(T,T.RightChild(v))
```

Arithmetic Expressions



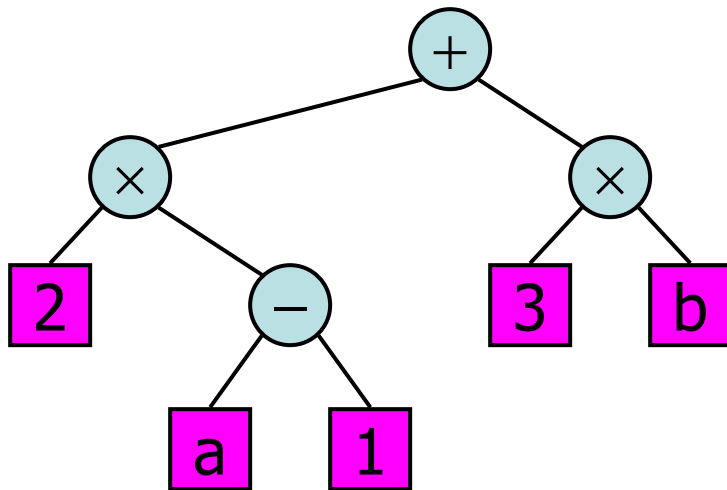
Inorder: $a - b$

Postorder: $a b -$

Preorder: $- a b$

Inorder:

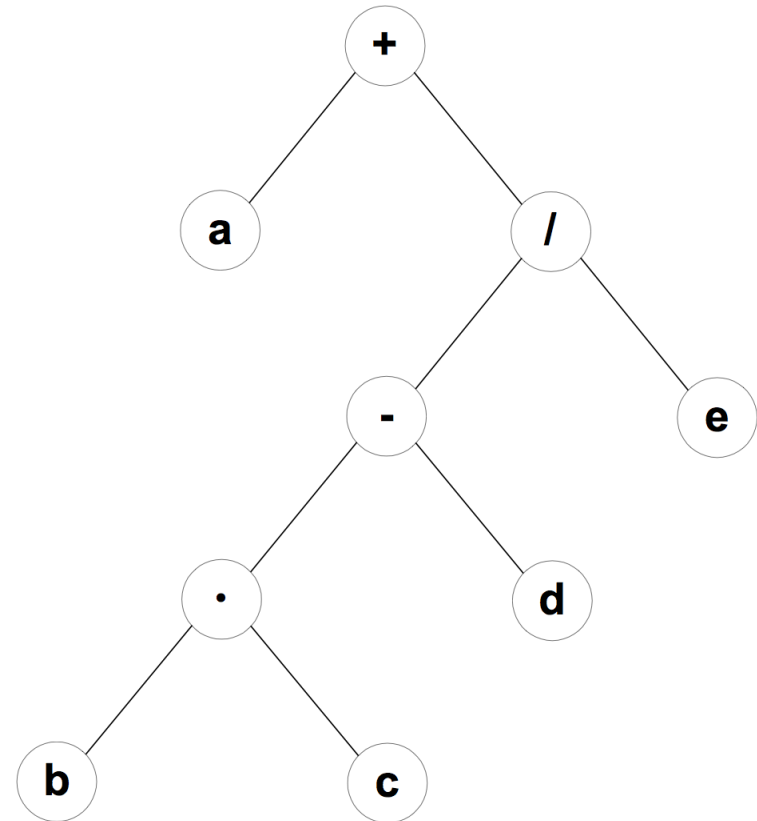
$2 \times a - 1 + 3 \times b$



Postorder:

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

$$a + (b \cdot c - d) / e$$



PRE-ORDER:

$+ a / - \cdot b c d e$

POST-ORDER:

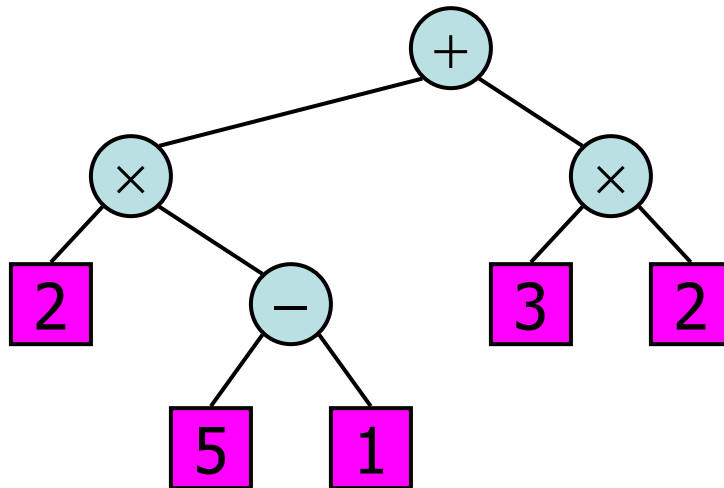
$a b c \cdot d - e / +$

IN-ORDER:

$a + b \cdot c - d / e$

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* ()

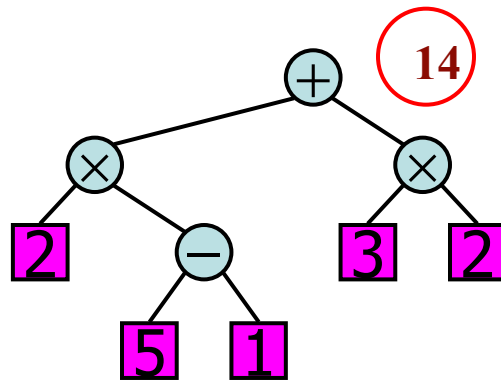
else

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

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

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

return *x* \diamond *y*



Algorithm *evalExpr*(*v*)

if *isExternal* (*v*)

return *v.element* ()

else

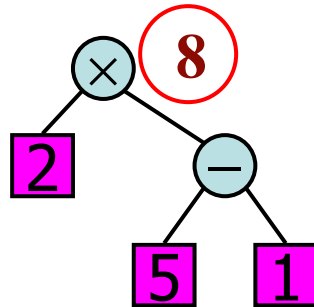
x \leftarrow *evalExpr*(*leftChild* (*v*))

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

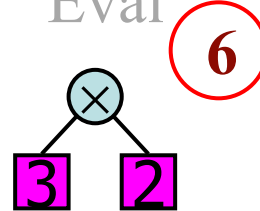
$\diamond \leftarrow$ operator stored at *v*

return *x* \diamond *y*

Eval



Eval



+

Eval

3

Eval

×

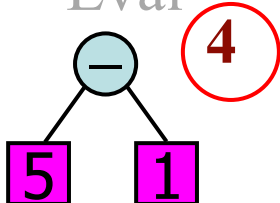
2

Eval

2

×

Eval



Eval

5

Eval

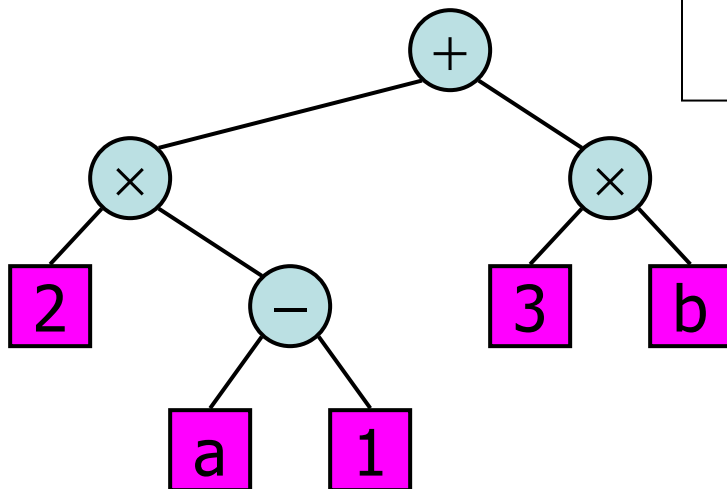
-

1

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 *printInOrder*(*v*)

if *isInternal* (*v*)

print (“(”)

printInOrder (*leftChild*(*v*))

print(*v.element* ())

if *isInternal* (*v*)

printInOrder(*rightChild*(*v*))

print (“)”)

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

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

Algorithm *preOrderTraversalwithStack(T)*

Stack *S*

TreeNode *N*

S.push(T) // push the reference to *T* in the empty stack

While (not *S.empty()*)

N = S.pop()

 if (*N != null*) {

print(N.elem) // print information

S.push(N.rightChild) // push the reference to
 the right child

S.push(N.leftChild) // push the reference to
 the left child

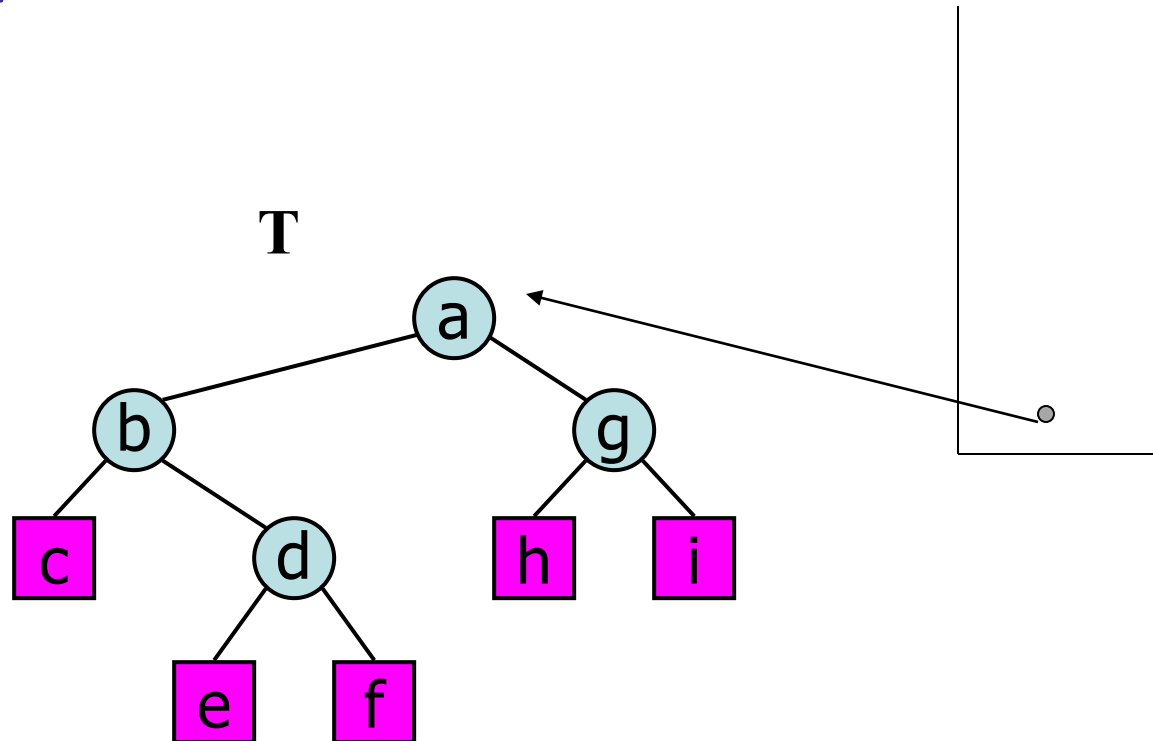
 }

Algorithm *preOrderTraversalwithStack(T)*

S.push(T) // push the reference to *T* in the empty stack

N = S.pop()

print(N.elem)

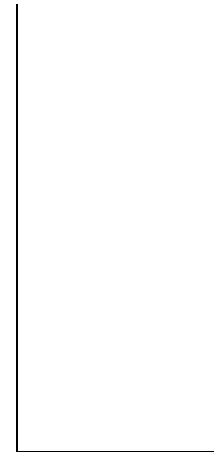
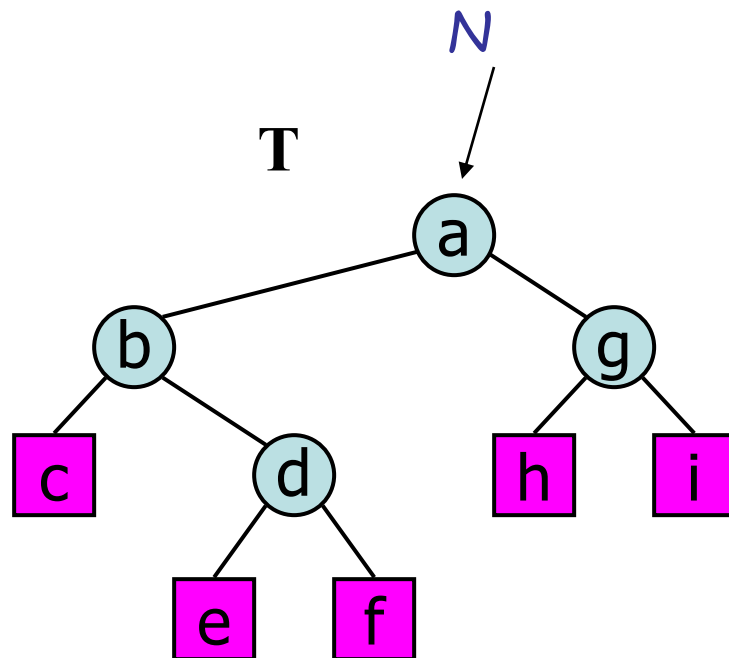


Algorithm *preOrderTraversalwithStack(T)*

S.push(T) // push the reference to *T* in the empty stack

N = S.pop()

print(N.elem)

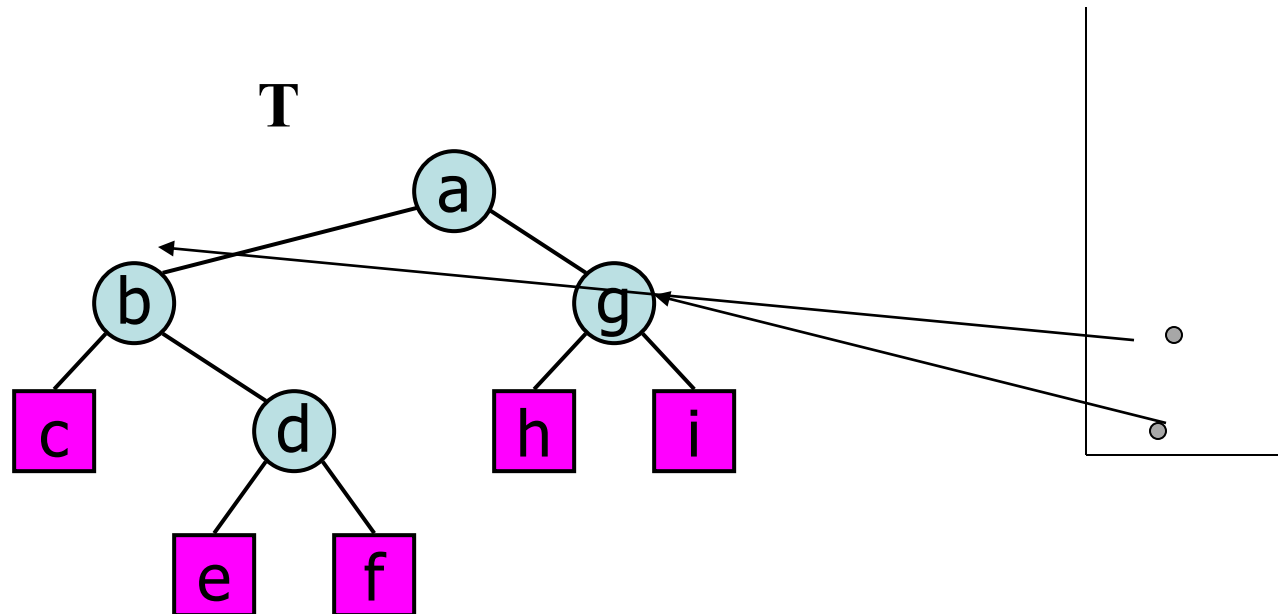


a

Algorithm *preOrderTraversalwithStack(T)*

S.push(N.rightChild) // push the reference to
the right child

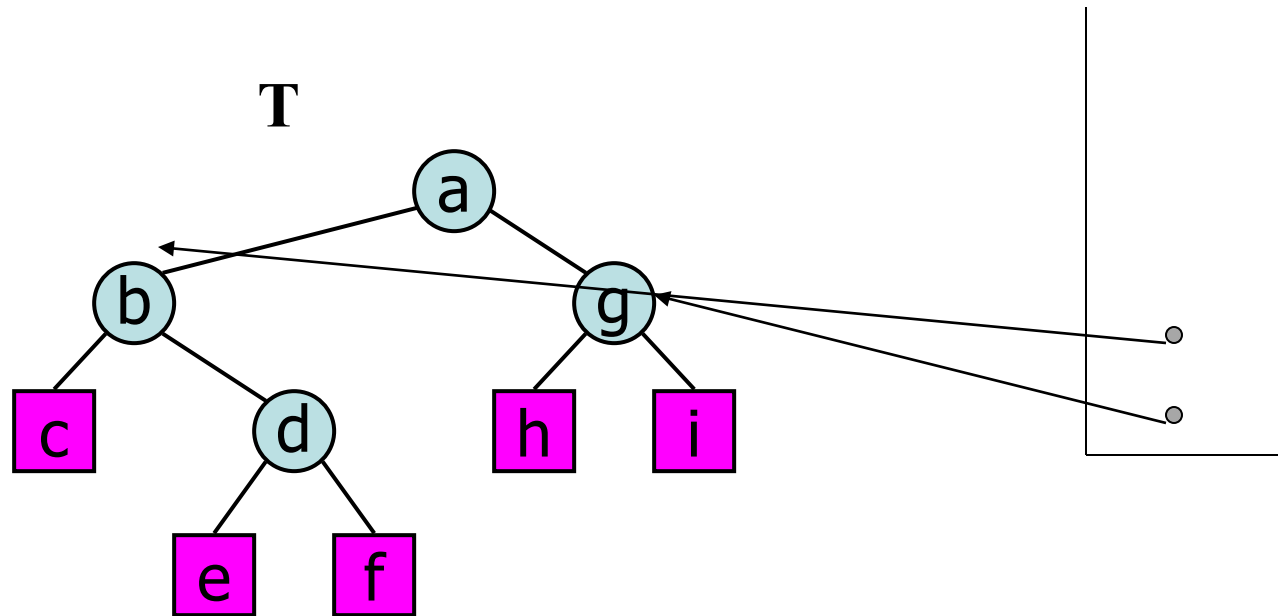
S.push(N.leftChild) // push the reference to
the left child



a

Algorithm *preOrderTraversalwithStack(T)*

$N = S.pop()$

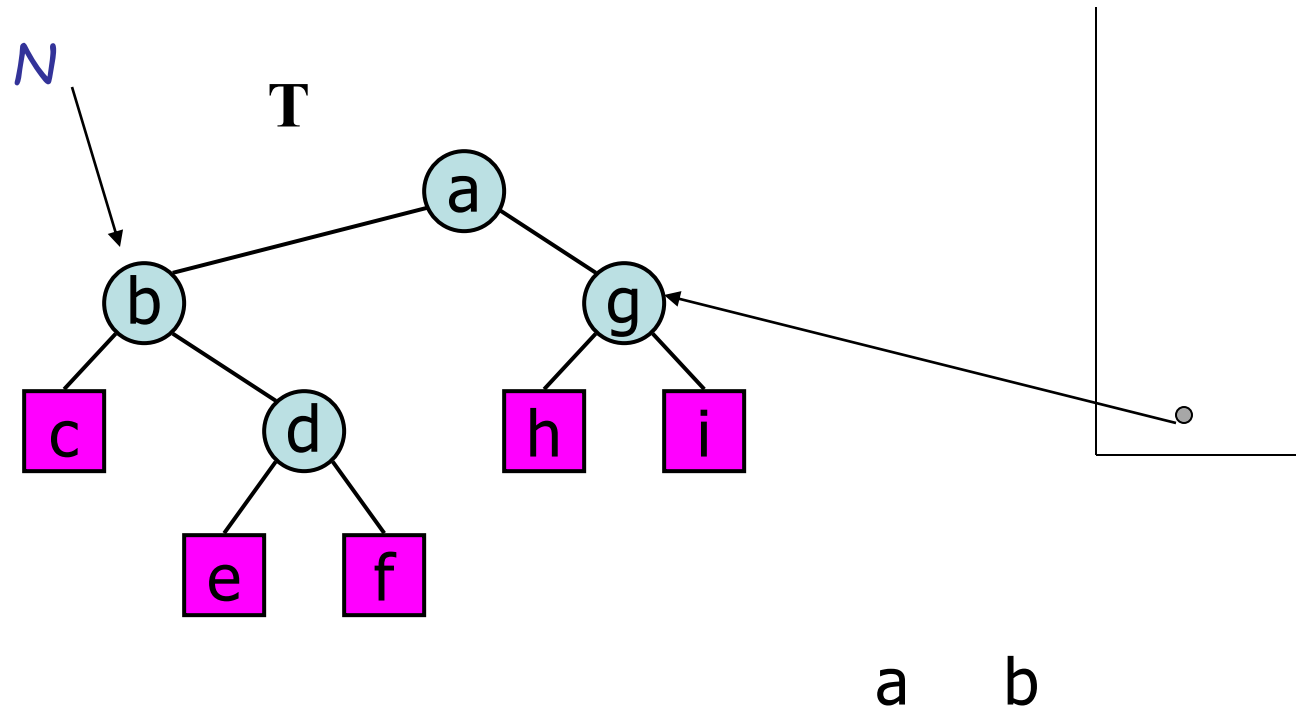


a

Algorithm *preOrderTraversalwithStack(T)*

$N = S.pop()$

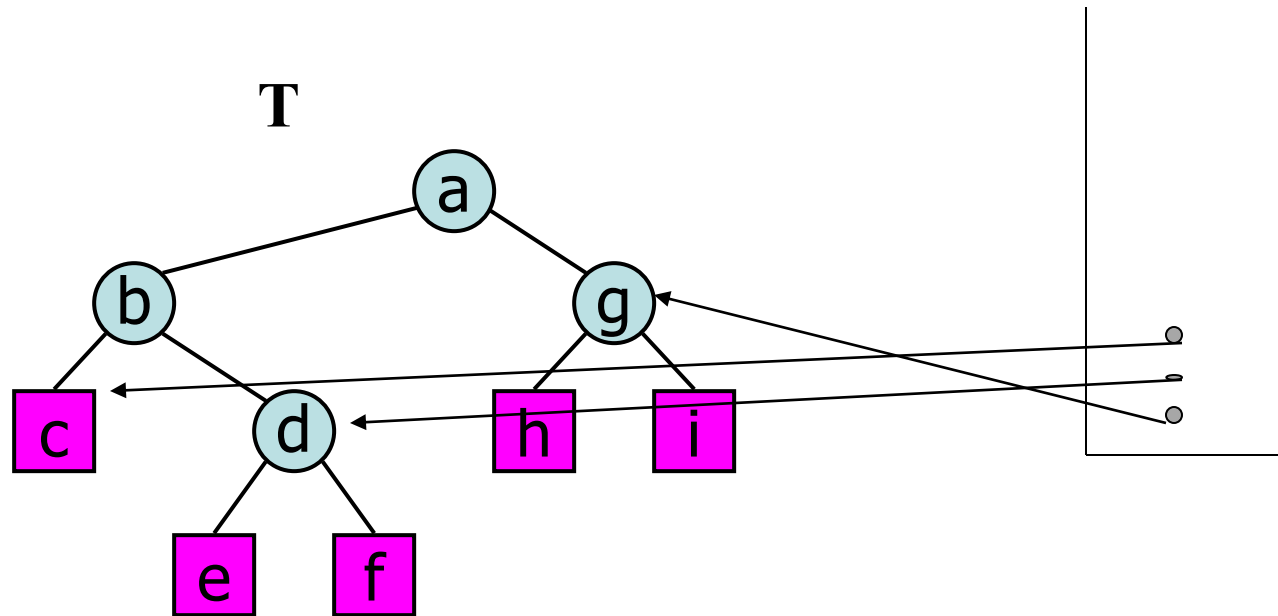
$print(N.elem)$



Algorithm *preOrderTraversalwithStack(T)*

S.push(N.rightChild)

S.push(N.leftChild)

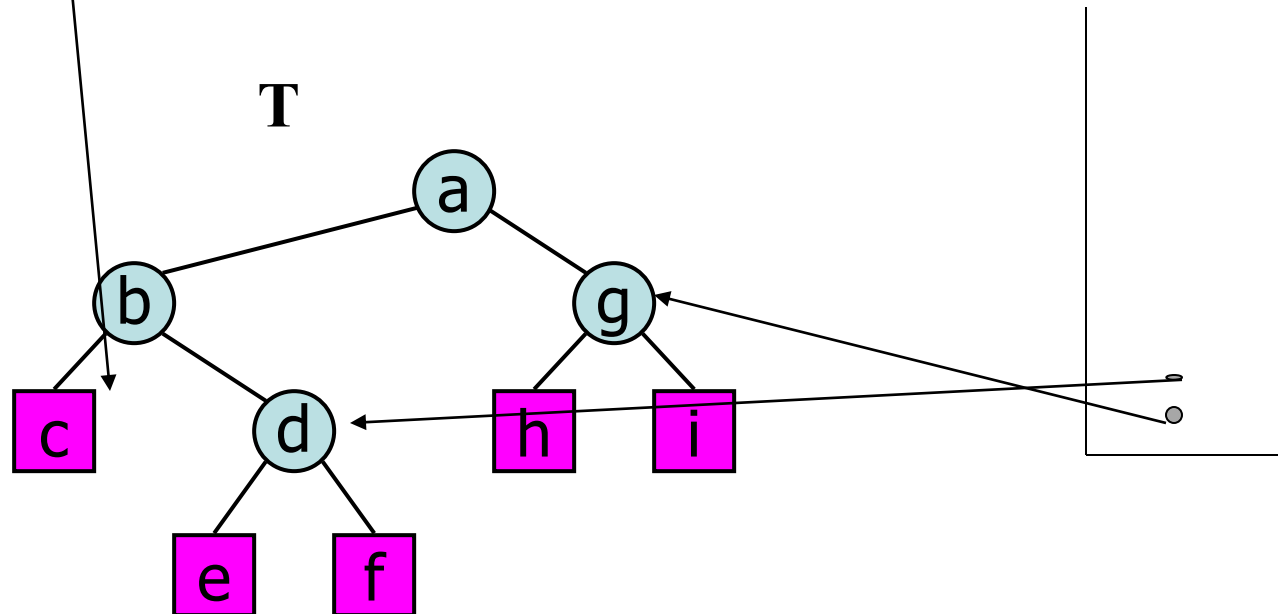


a b

Algorithm *preOrderTraversalwithStack(T)*

$N = S.pop()$

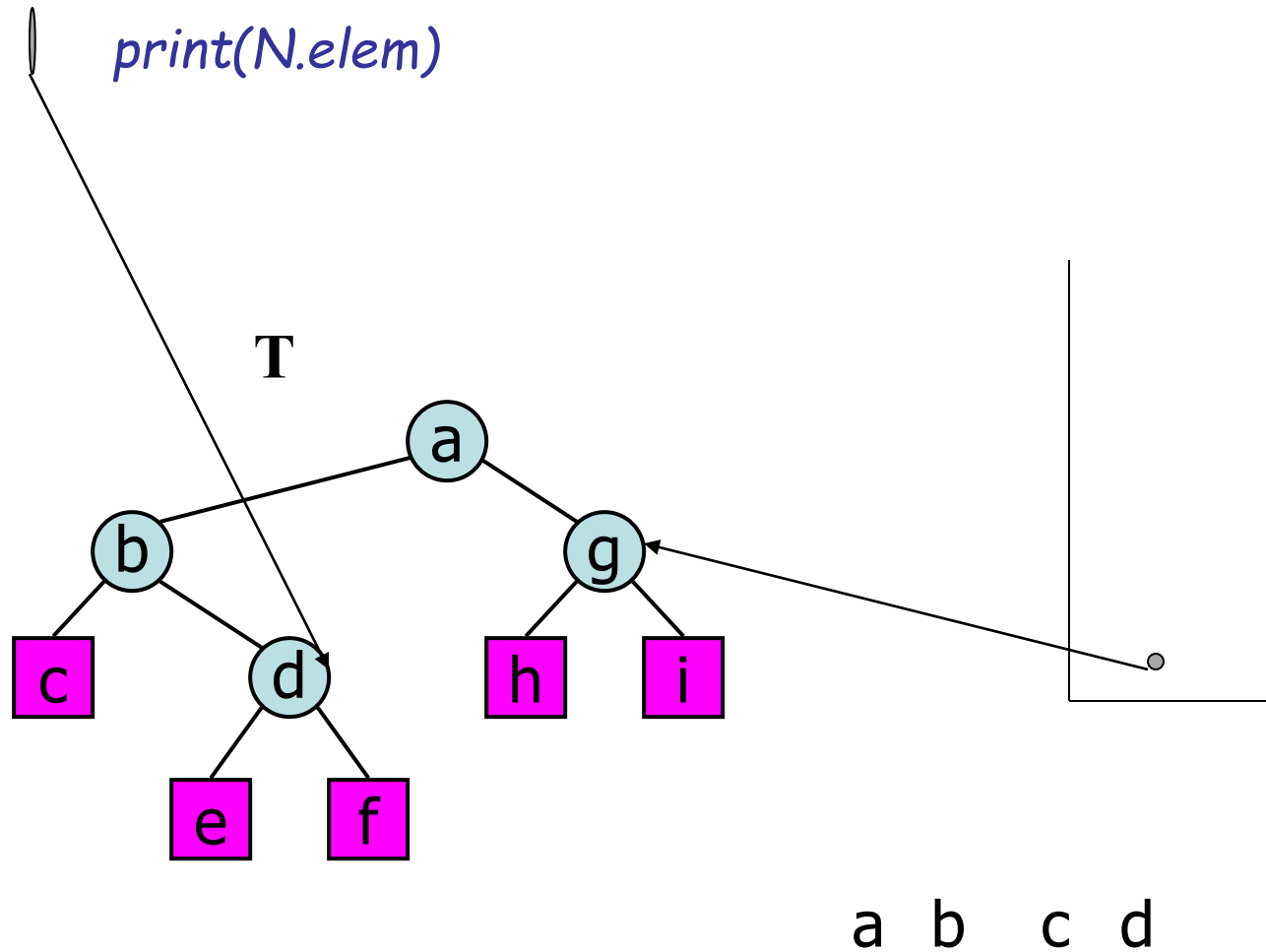
print(N.elem)



Algorithm *preOrderTraversalwithStack(T)*

$N = S.pop()$

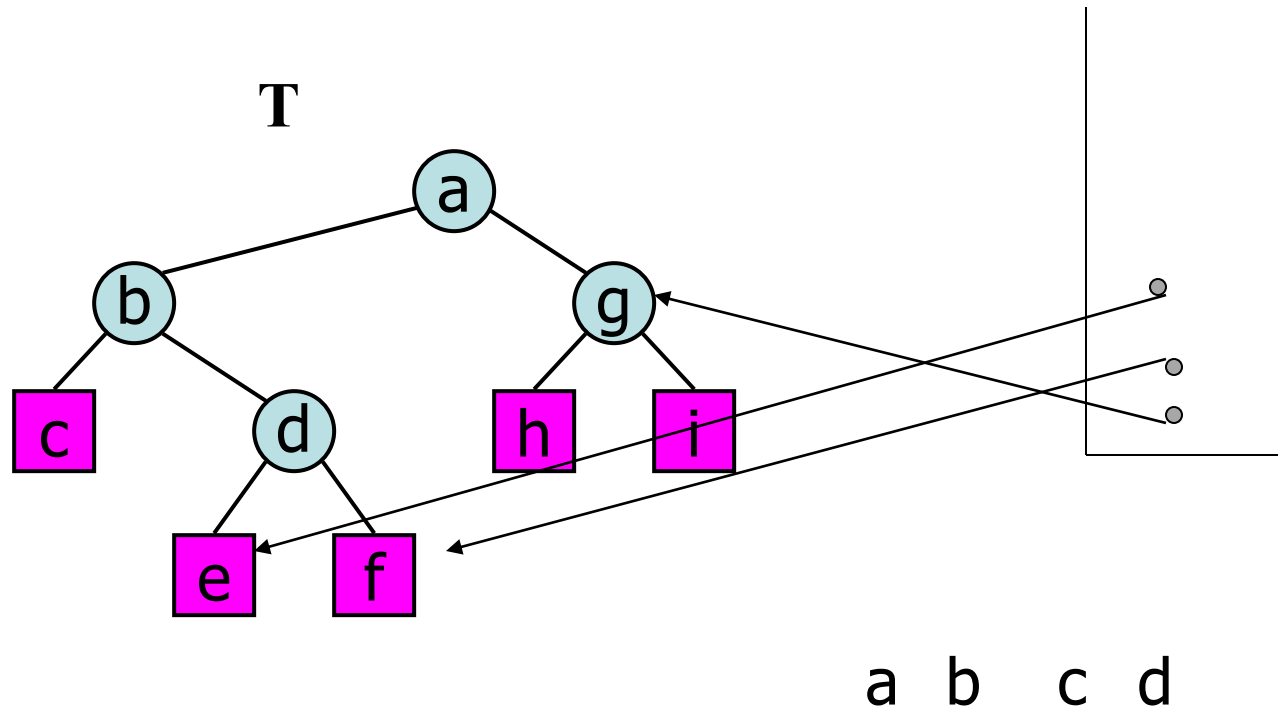
$print(N.elem)$



Algorithm *preOrderTraversalwithStack(T)*

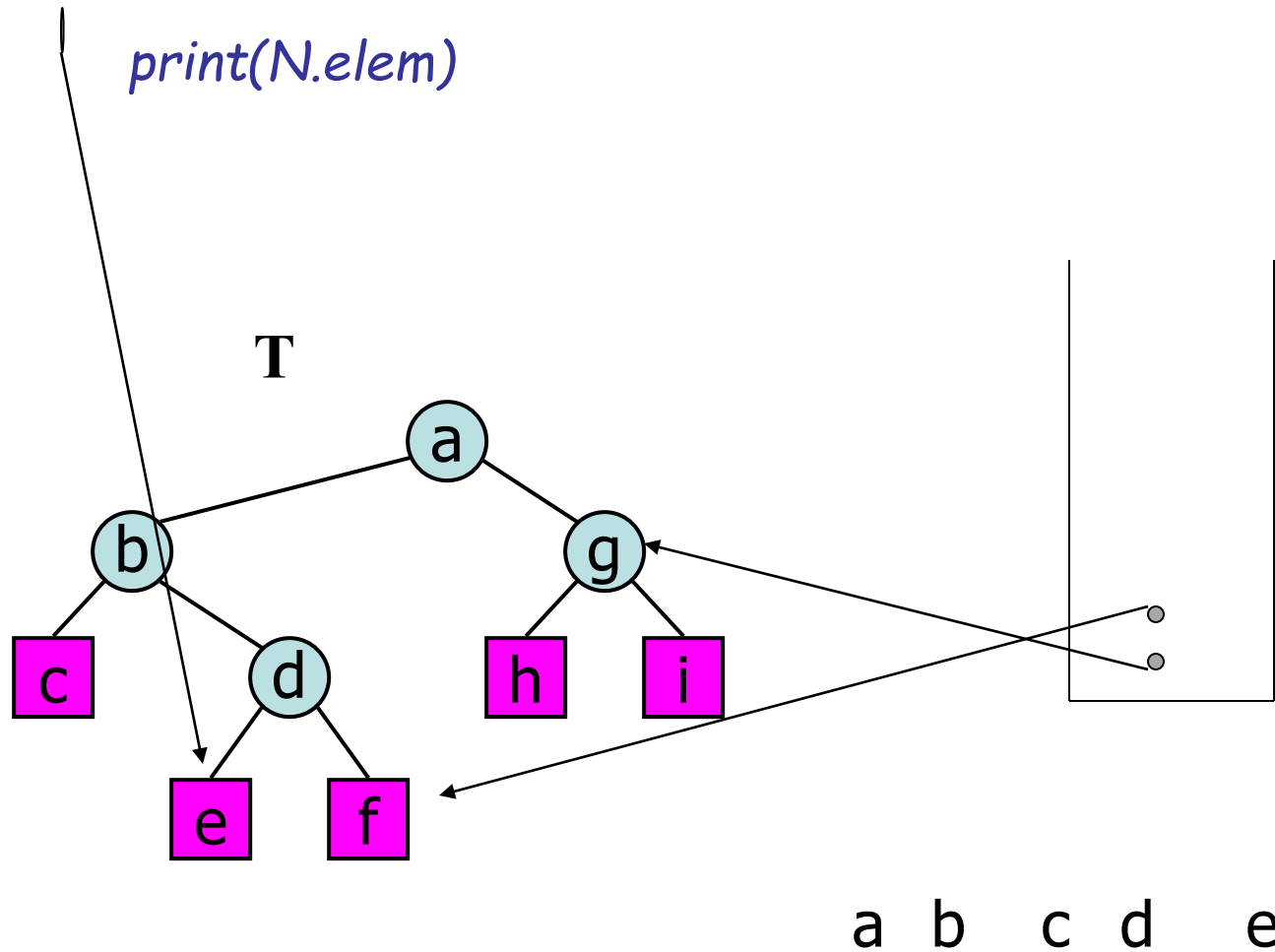
S.push(N.rightChild)

S.push(N.leftChild)



Algorithm *preOrderTraversalwithStack(T)*

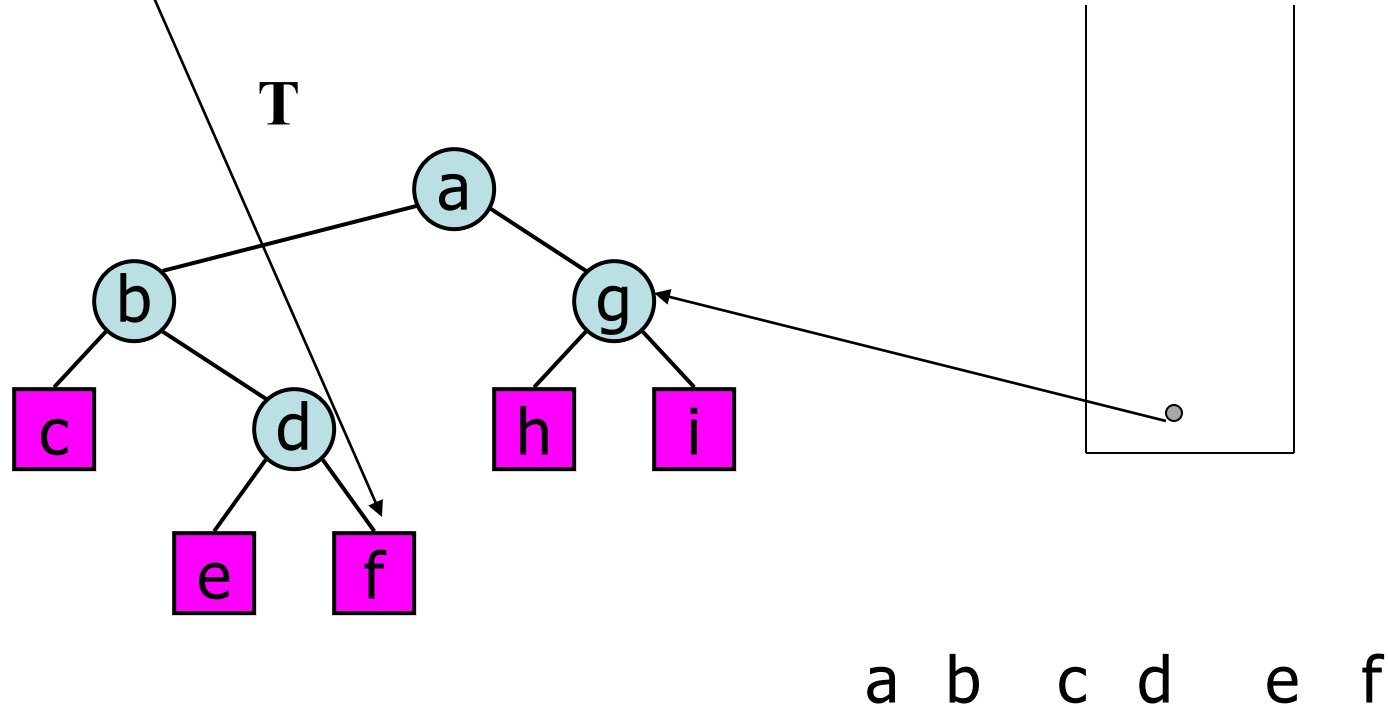
$N = S.pop()$



Algorithm *preOrderTraversalwithStack(T)*

$N = S.pop()$

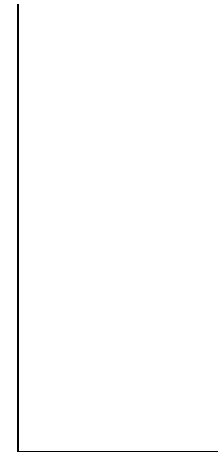
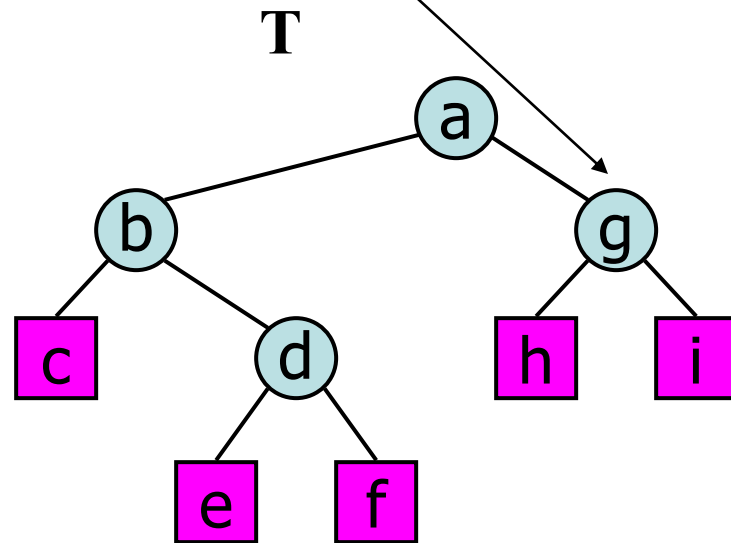
$print(N.elem)$



Algorithm *preOrderTraversalwithStack(T)*

$N = S.pop()$

$print(N.elem)$

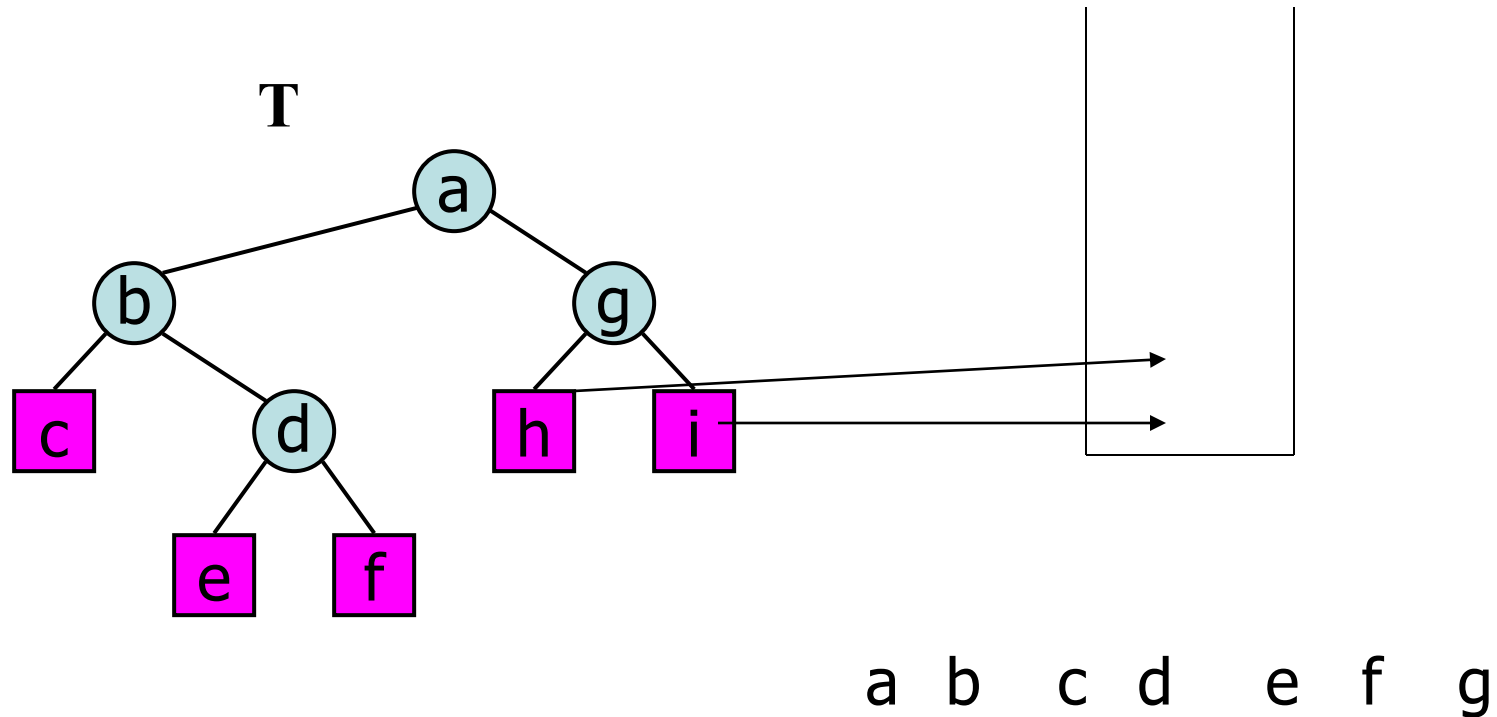


a b c d e f g

Algorithm *preOrderTraversalwithStack(T)*

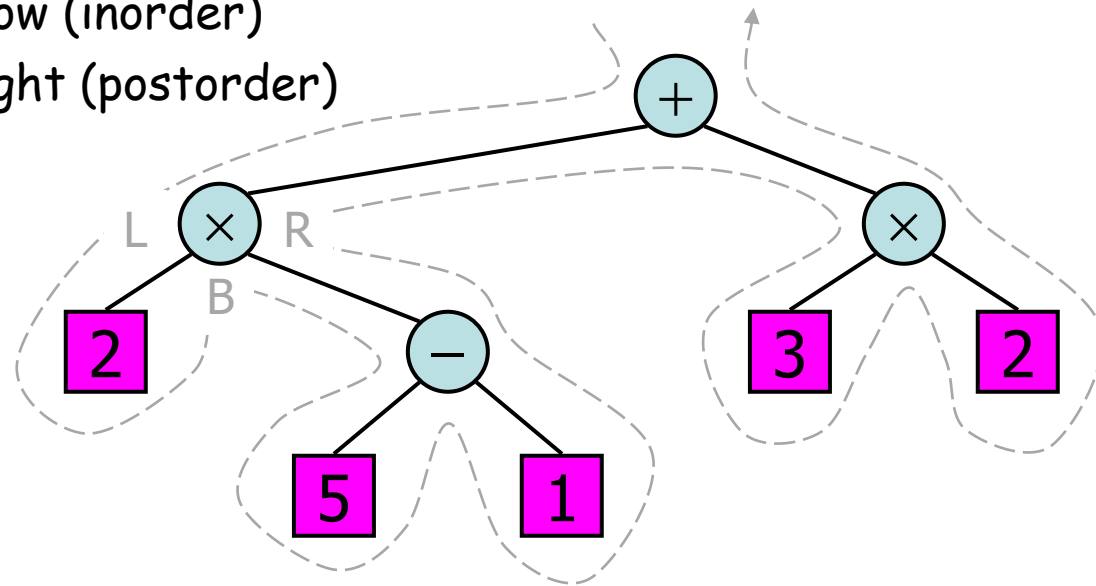
S.push(N.rightChild)

S.push(N.leftChild)



Euler Tour Traversal

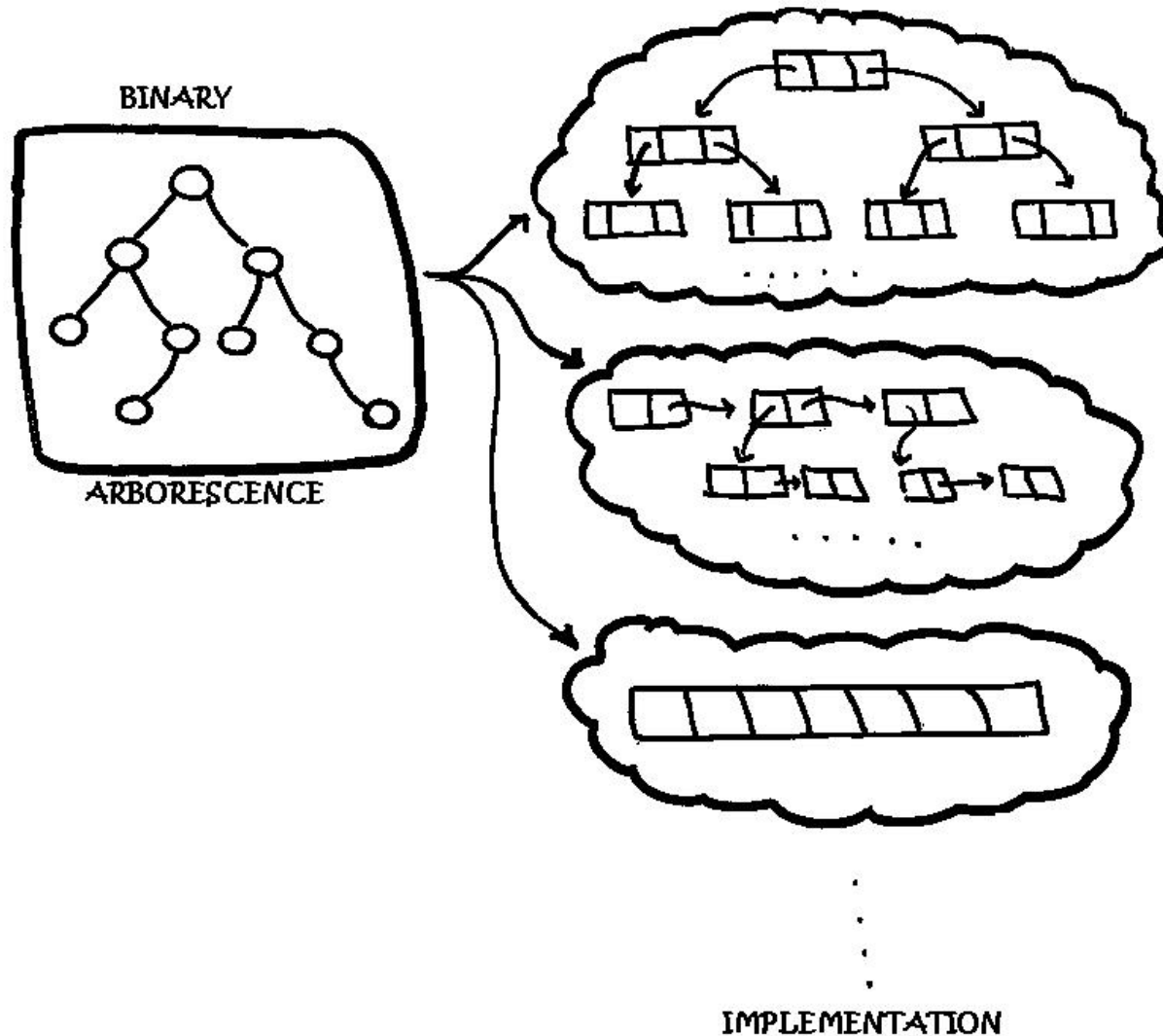
- » Generic traversal of a binary tree
- » Includes a special cases the preorder, postorder and inorder traversals
- » Walk around the tree and visit each node three times:
 - on the left (preorder)
 - from below (inorder)
 - on the right (postorder)



Algorithm eulerTour(T, v)

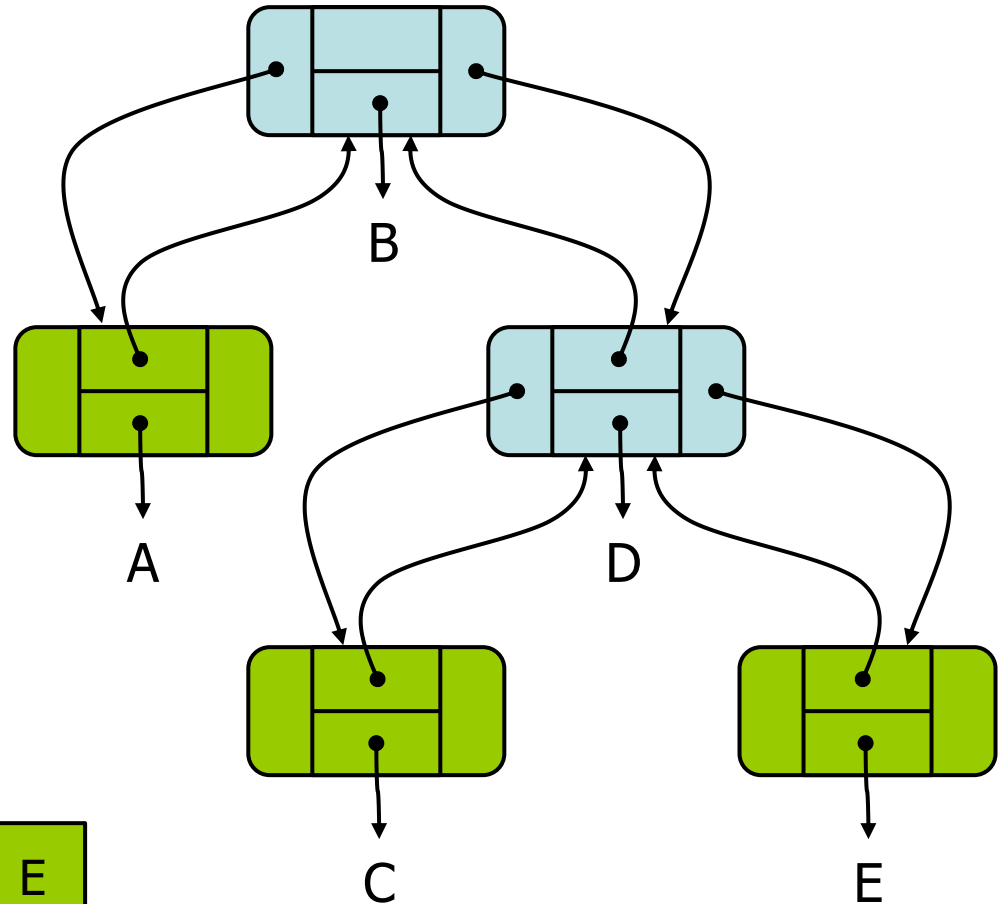
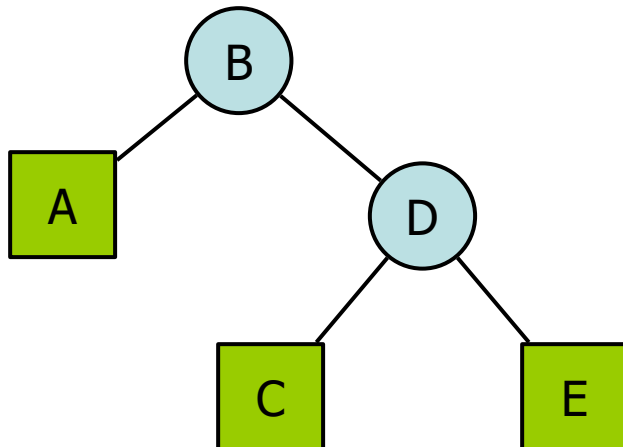
```
leftVisit v                                (from the left)
  if v is internal:
    eulerTour (T, T.LeftChild(v))
belowVisit v                              (from below)
  if v is internal:
    eulerTour(T, T.RightChild(v))
rightVisit v                              (from the right)
```

Implementations of Binary trees....

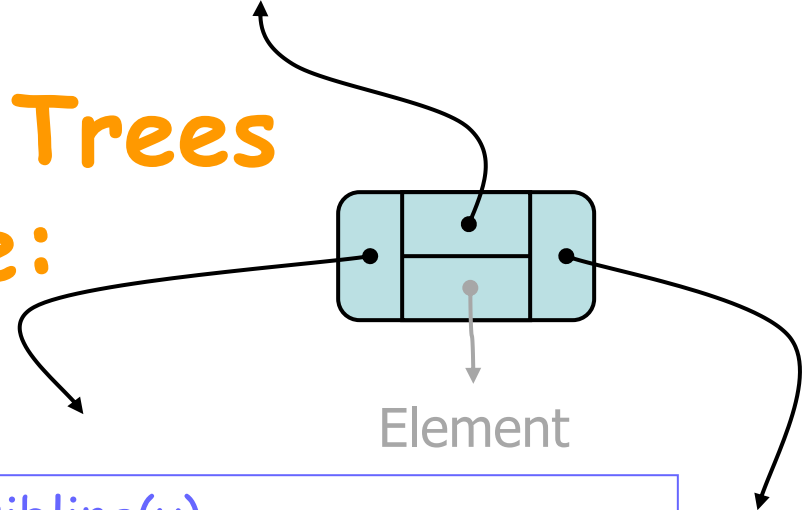


Implementing Binary Trees with a Linked Structure

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



Implemented Binary Trees with linked structure: ADT BTNode



```
left(v)  return v.left
```

```
right(v) return v.right
```

```
swapElements(v,w)
  temp ← w.element
  w.element ← v.element
  v.element ← temp
```

```
sibling(v)
  p ← parent(v)
  q ← left(p)
  if (v = q) return right(p)
  else return q
```

```
replaceElement(v,obj)
  temp ← v.element
  v.element ← obj
  return temp
```

leftChild(p), rightChild(p), sibling(p):

Input: Position Output: Position

swapElements(p,q) Input: 2 Positions Output: None

replaceElement(p,e) Input: Position and an object Output: Object

isRoot(p) Input: Position Output: Boolean

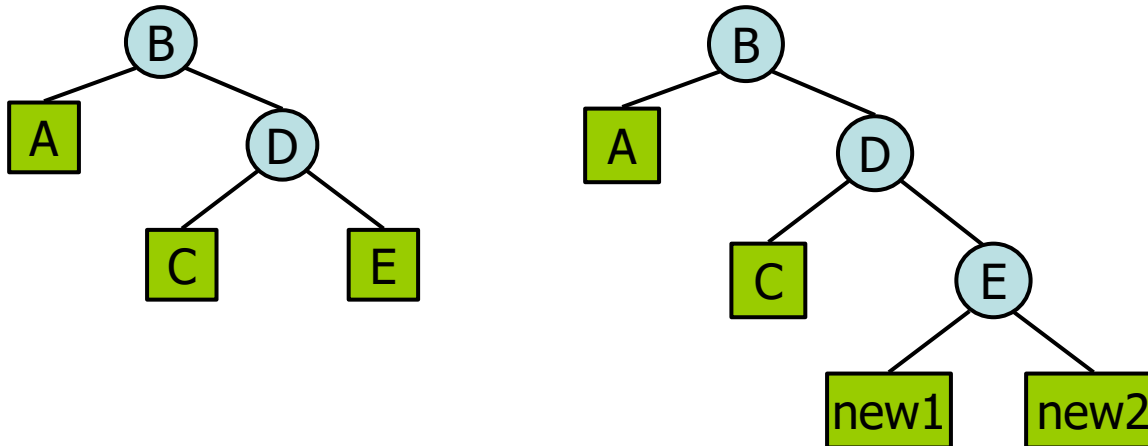
isInternal(p) Input: Position Output: Boolean

isExternal(p) Input: Position Output: Boolean

$O(1)$

Other interesting methods for the ADT Binary Tree:

expandExternal(v): Transform v from an external node into an internal node by creating two new children



expandExternal(v):

new1 and new 2 are the new nodes

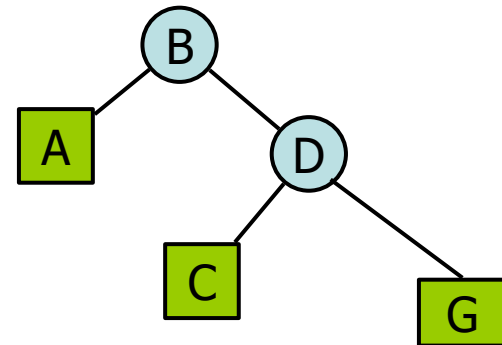
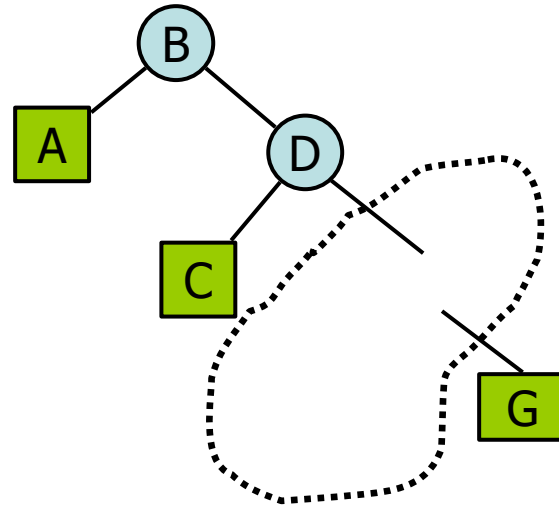
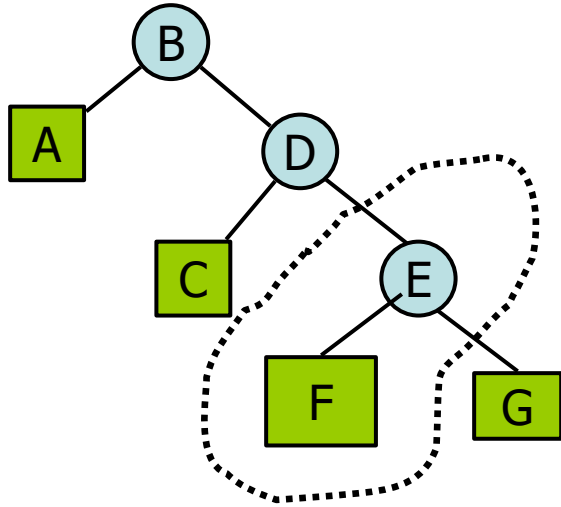
if isExternal(v)

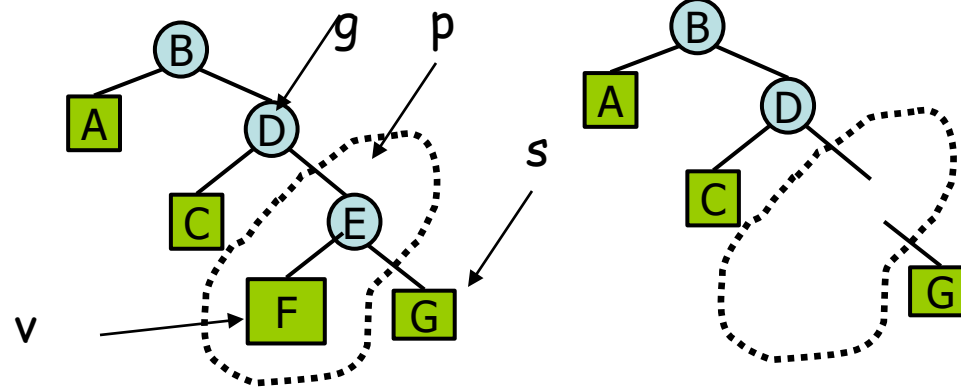
v.left \leftarrow new1

v.right \leftarrow new2

size \leftarrow size +2

removeAboveExternal(v):





removeAboveExternal(v):

if isExternal(v)

{ p ← parent(v)

s ← sibling(v)

if isRoot(p) s.parent ← null and root ← s

else

{ g ← parent(p)

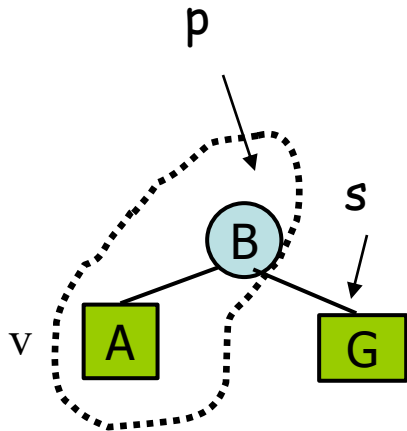
if p is leftChild(g) g.left ← s

else g.right ← s

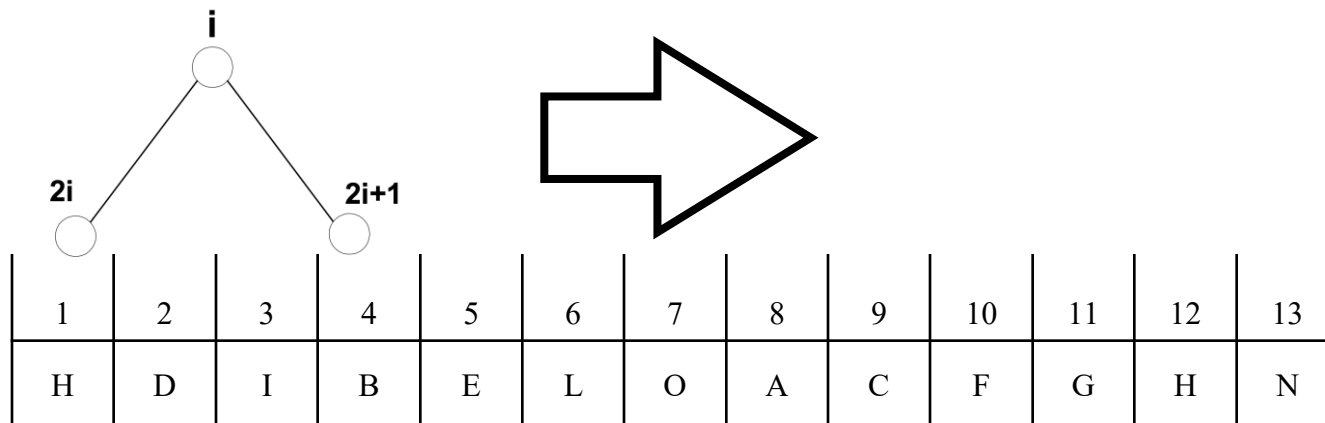
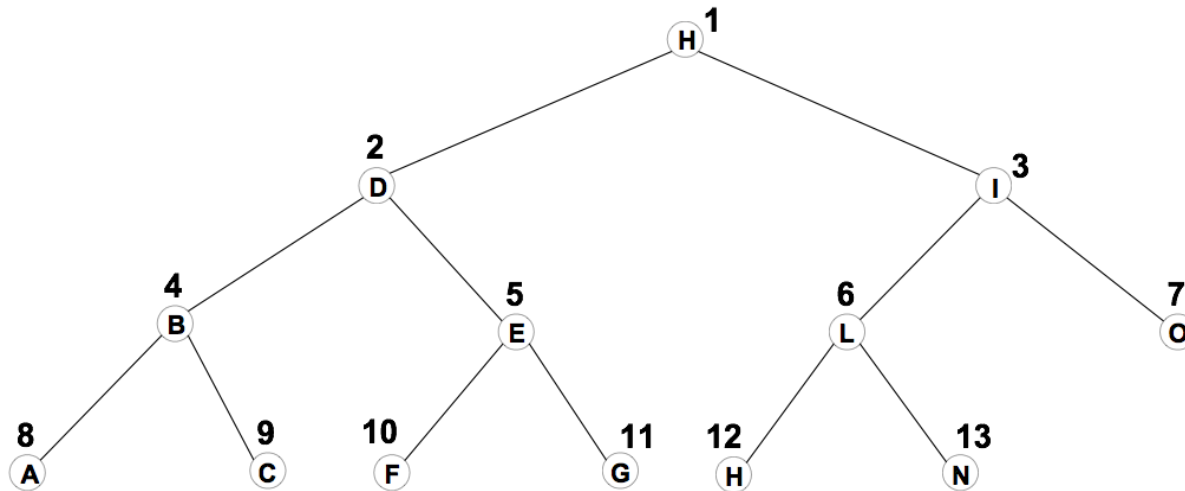
s.parent ← g

}

size ← size - 2 }



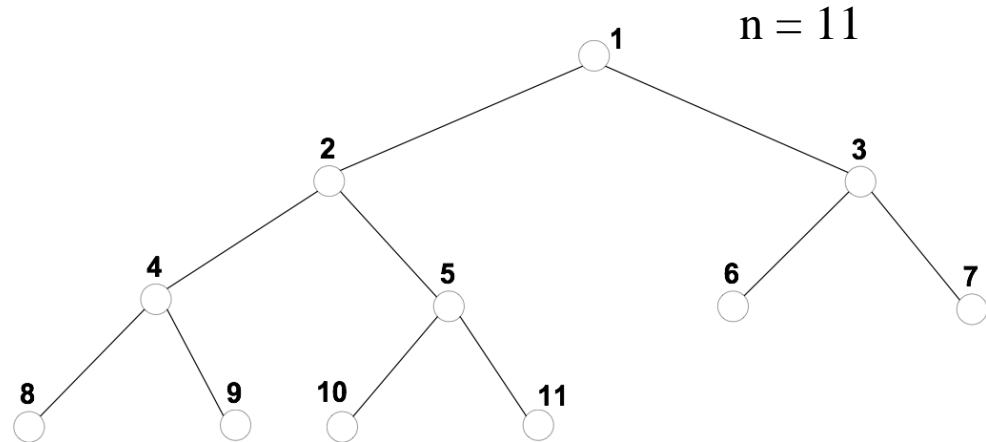
Implementing Complete Binary Trees with Vectors (Array-based)



leftChild(p), rightChild(p), sibling(p):

swapElements(p,q),
replaceElement(p,e)
isRoot(p), isInternal(p),
isExternal(p)

They all have
complexity $O(1)$



Left child of $T[i]$	$T[2i]$	if	$2i \leq n$
Right child of $T[i]$	$T[2i+1]$	if	$2i + 1 \leq n$
Parent of $T[i]$	$T[i \text{ div } 2]$	if	$i > 1$
The Root	$T[1]$	if	$T \neq 0$
Leaf? $T[i]$	TRUE	if	$2i > n$

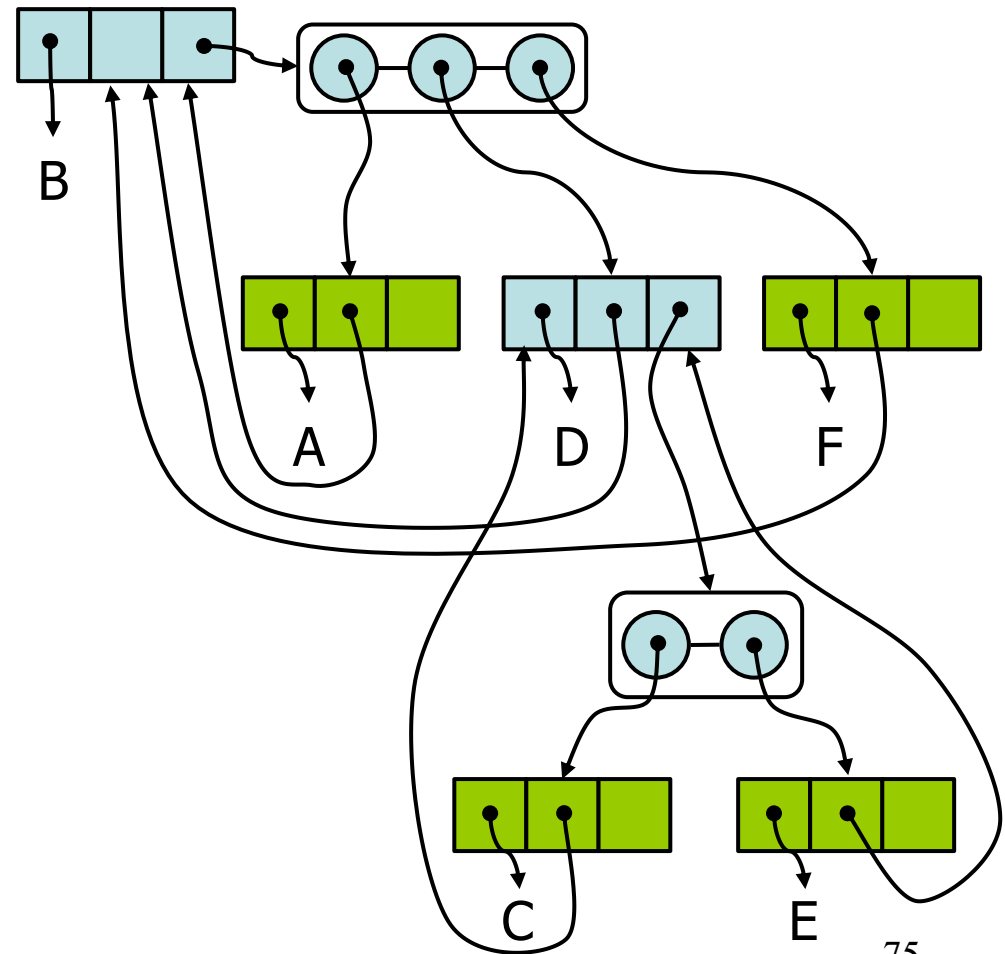
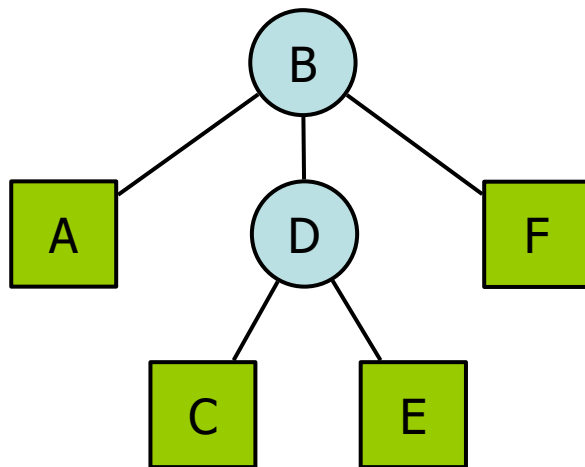
leftChild(p), rightChild(p), sibling(p):

swapElements(p,q),
replaceElement(p,e)
isRoot(p), isInternal(p),
isExternal(p)

They all have
complexity $O(1)$

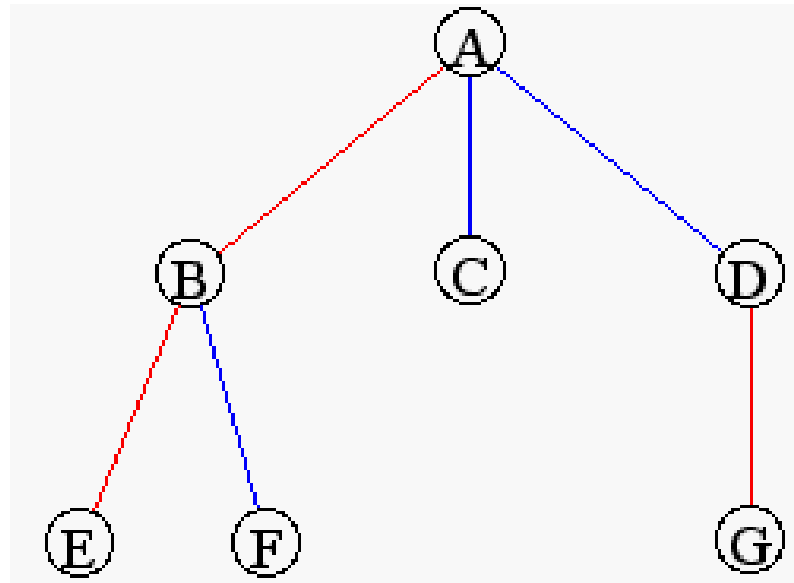
Implementing General Trees with a Linked Structure

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

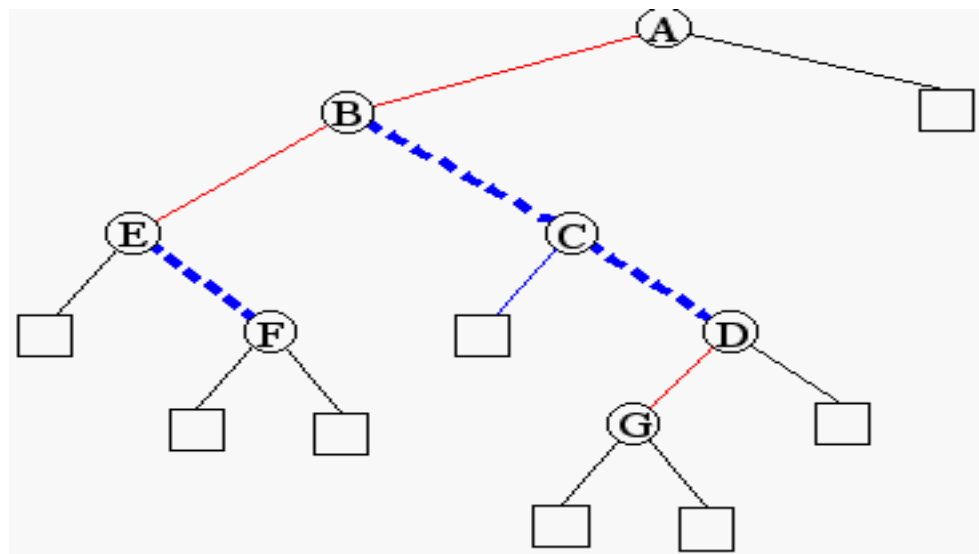


Representing General Trees

tree T



binary tree T' representing T



RULES for representing general tree T using binary tree T'

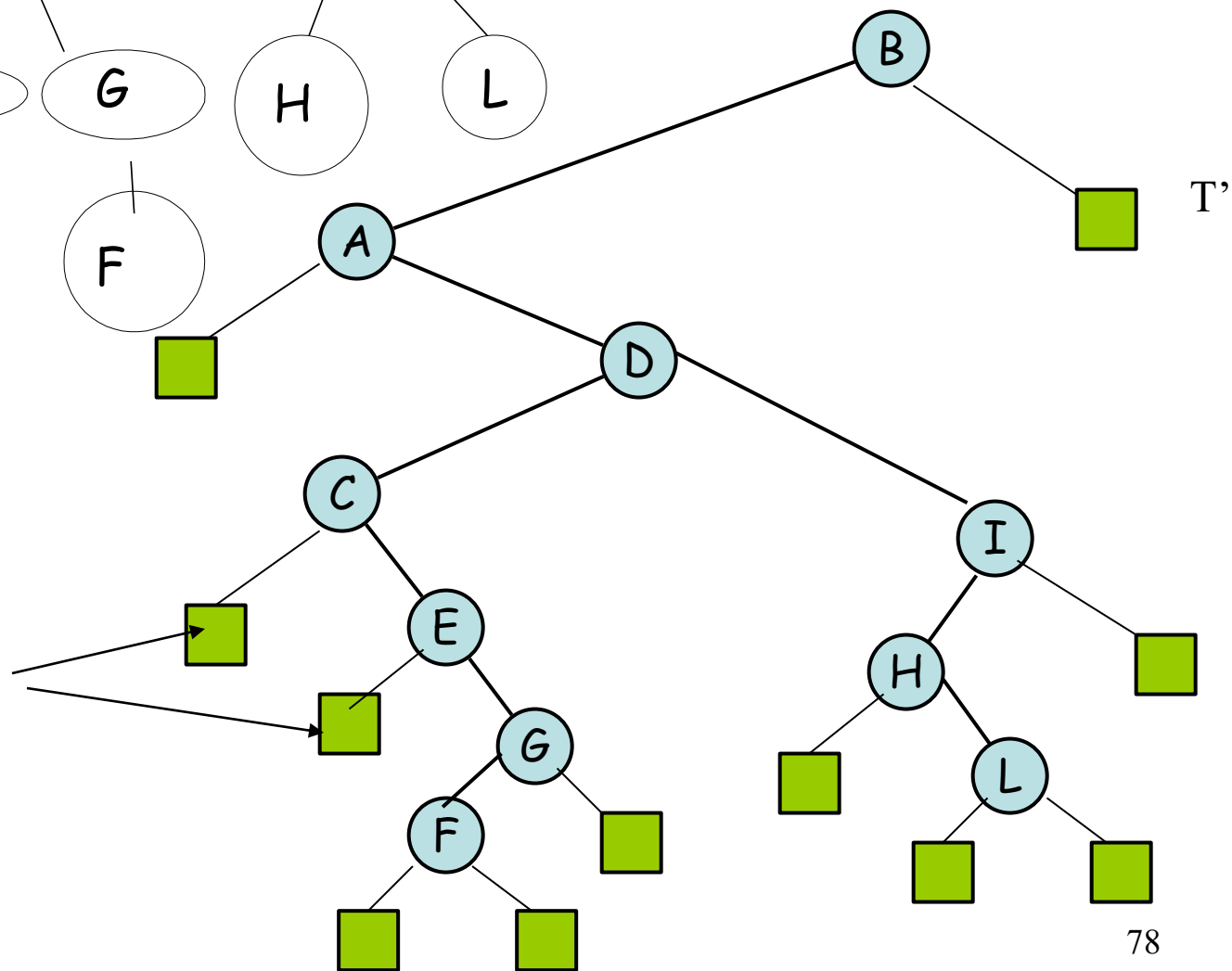
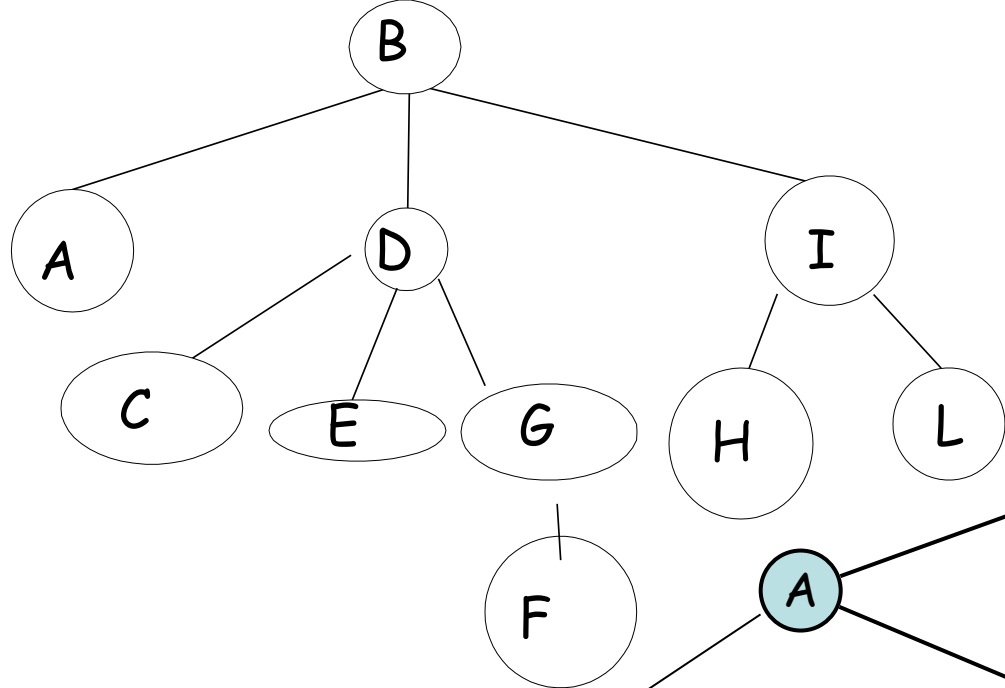
u in T

u' in T'

first child of u in T is **left child** of u' in T'

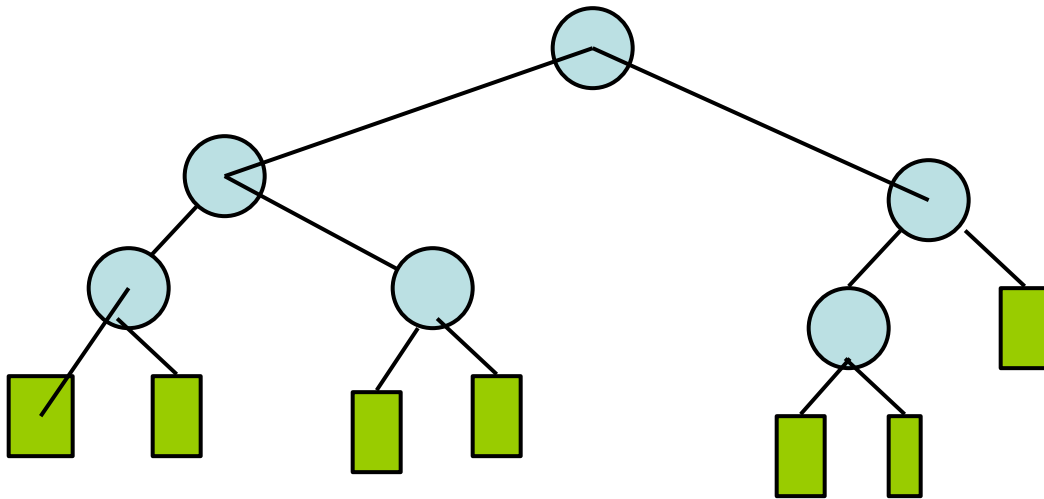
first sibling of u in T is **right child** of u' in T'

T



Place holder

children are "completed" with "fake" nodes



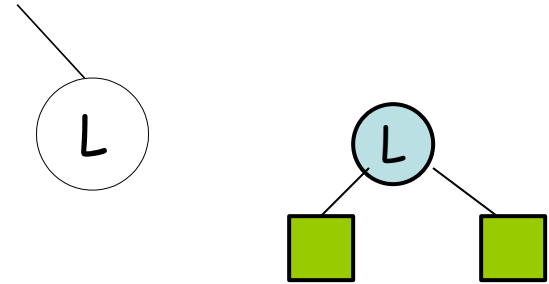
The green squared nodes are the dummy nodes.

In this way ALL the original nodes are internal.
The leaves are the fake green nodes.

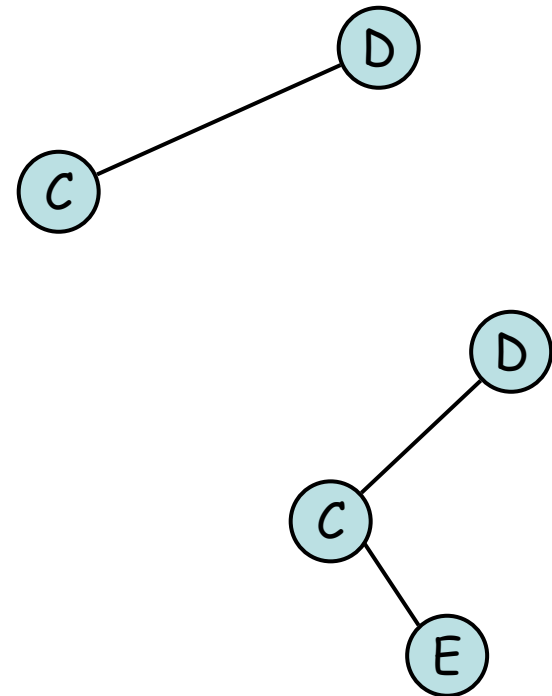
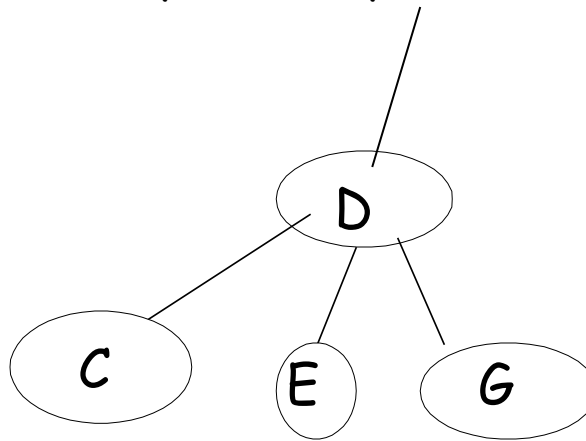
RULE:

to u in T corresponds u' in T'

if u is a leaf in T and has no siblings,
then the children of u' are leaves



If u is internal in T and v is its first child
then v' is the left child of u' in T'



If v has a sibling w immediately following it,
 w' is the right child of v' in T'

