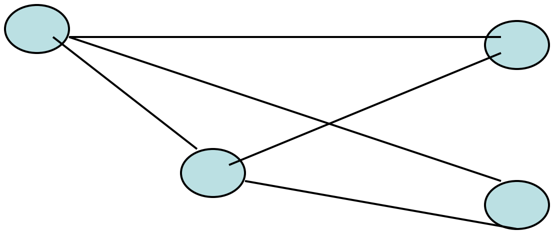
Trees

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

Trees

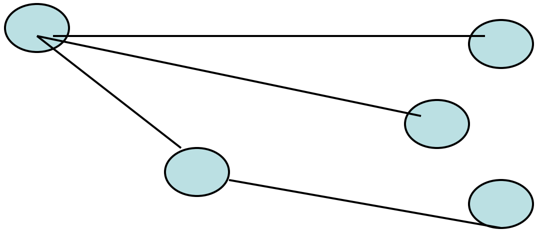
A graph G = (V,E) consists of an set V of VERTICES

and a set E of edges, with $E = \{(u,v): u, v \in V, u \neq v\}$

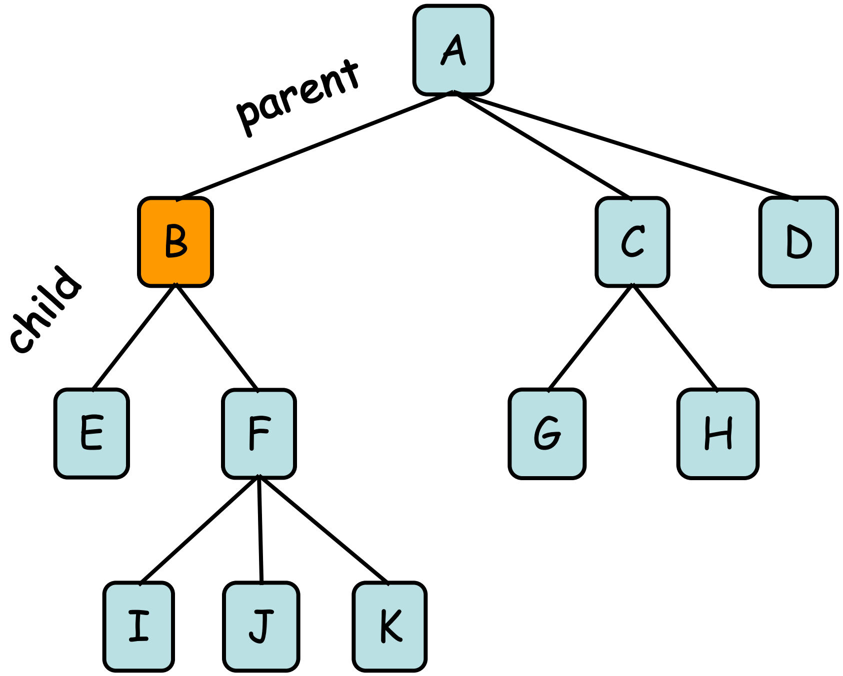


A tree is a connected graph with no cycles.

→ ∃ a path between each pair of vertices.



(Rooted) Trees



Tree Terminology

- **Root**: node without parent (A)

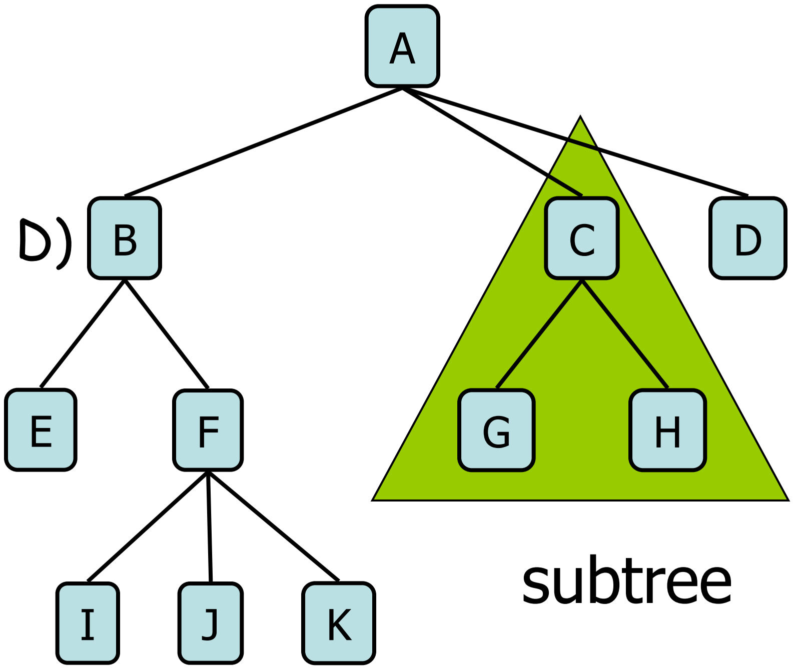
- **Internal node**: node with at least one child (A, B, C, F)

- **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.

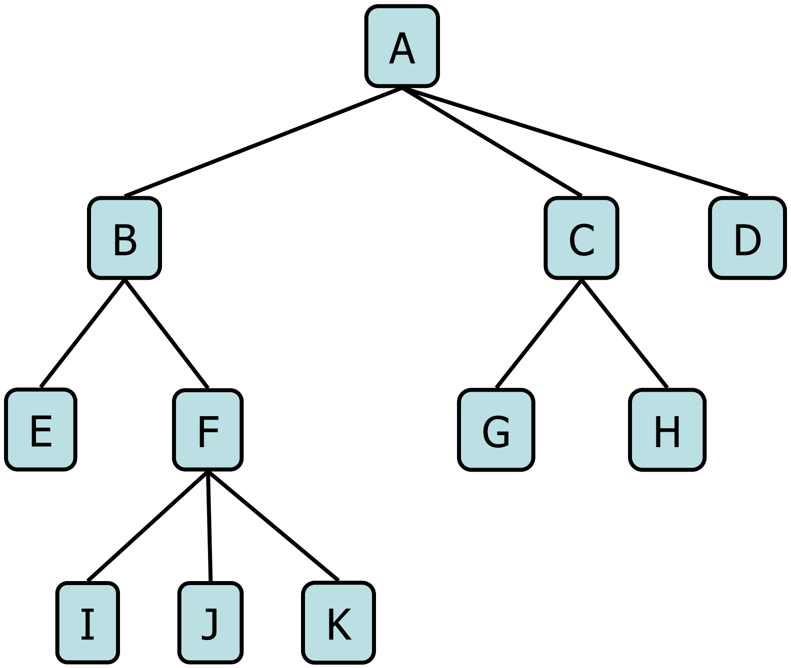
- **Subtree**: tree consisting of a node and its descendants



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)



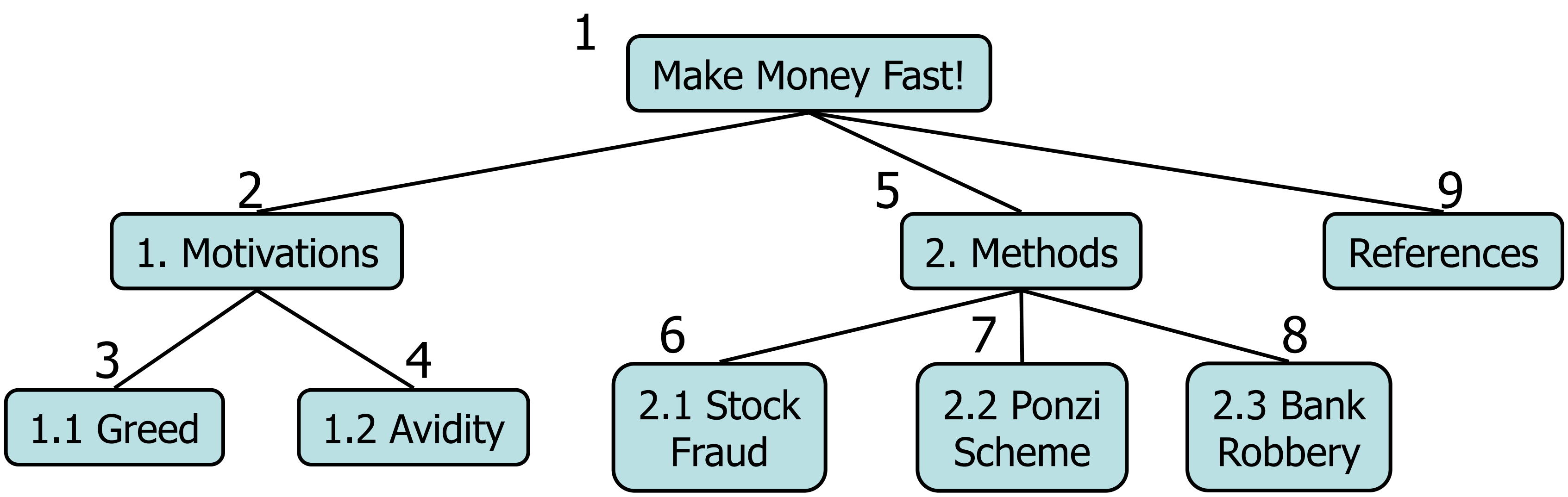
ADTs for Trees

- 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

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)

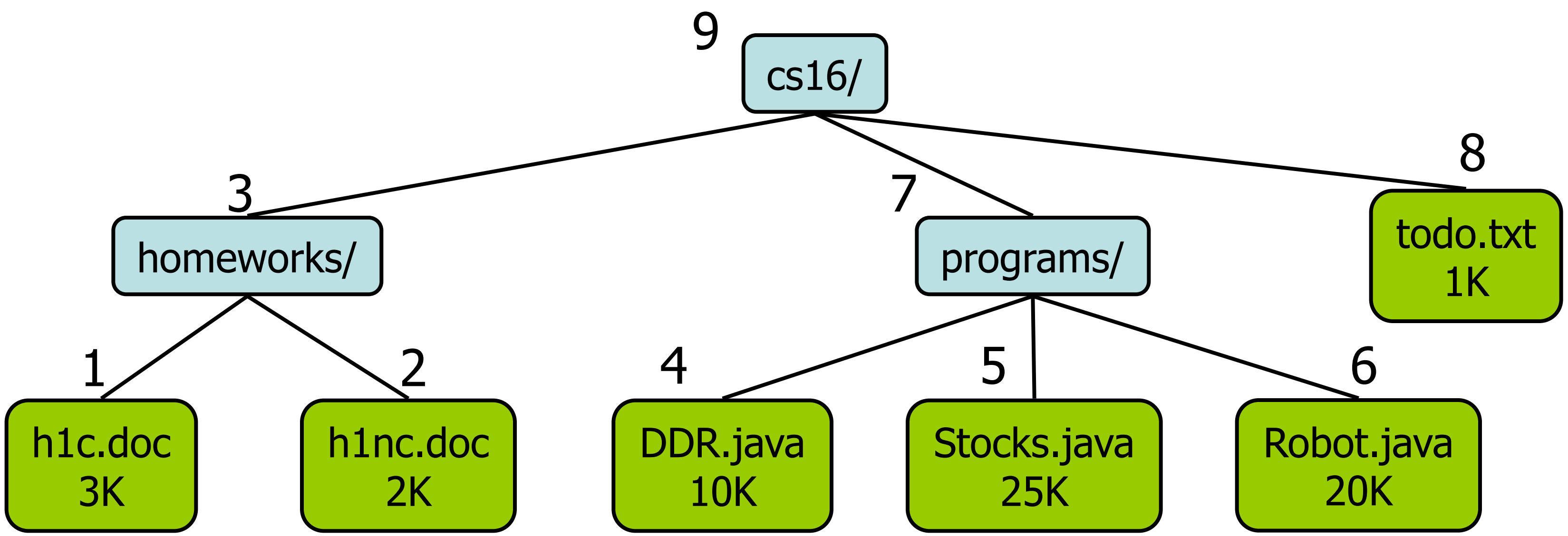


Traversing Trees

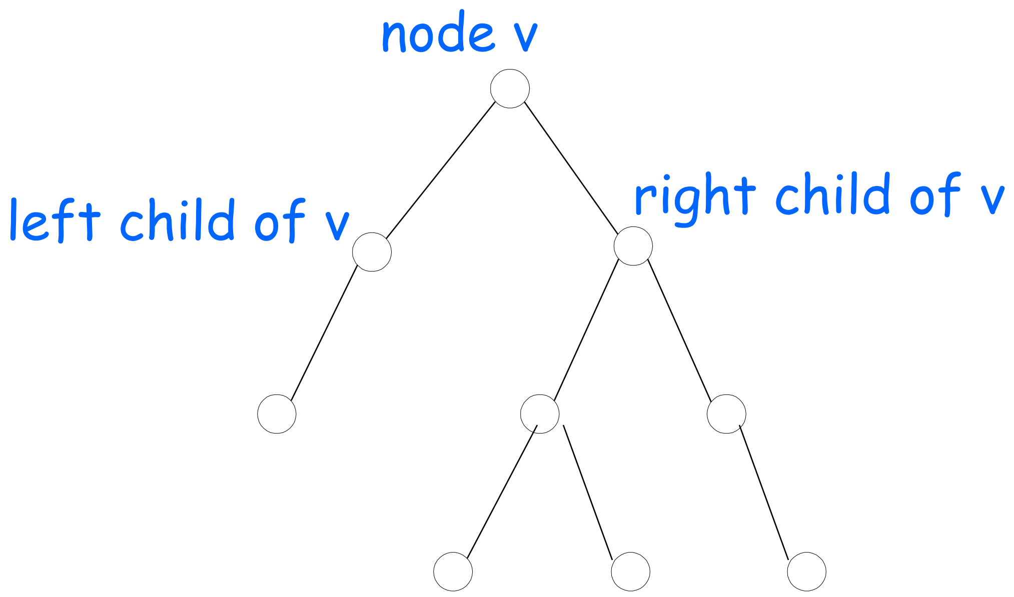
Postorder Traversal

- 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)



Binary Trees



Children are ordered

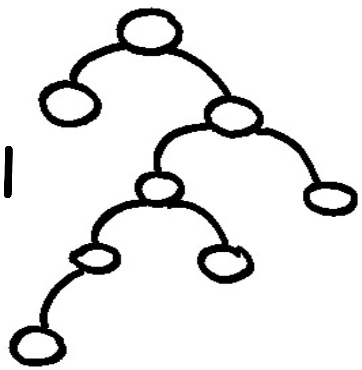
Each node has at most two children:

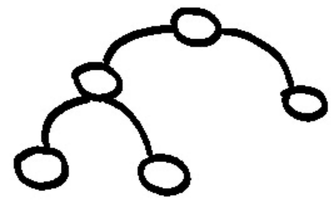
[0, 1, or 2]

"Full" Binary Trees (or "Proper")

Each node: is a leaf, or has two children

not full

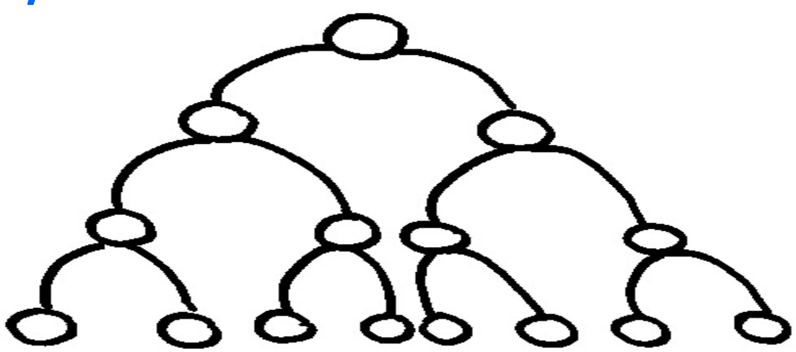




full

Perfect Binary Trees

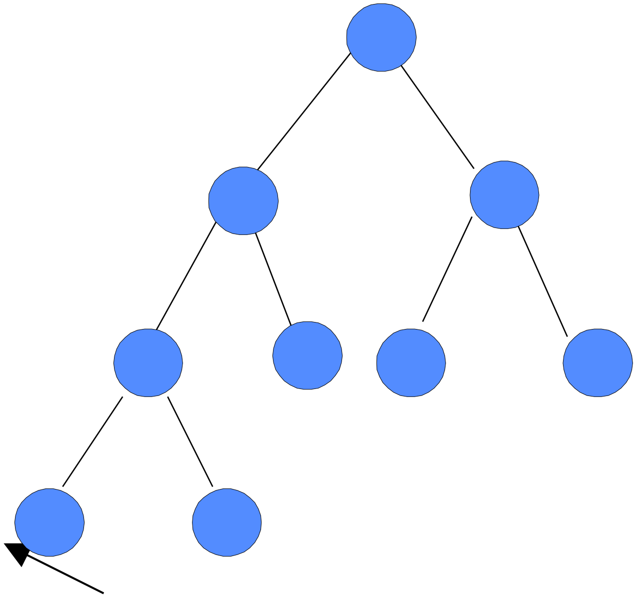
Full binary trees with all leaves at the same level:



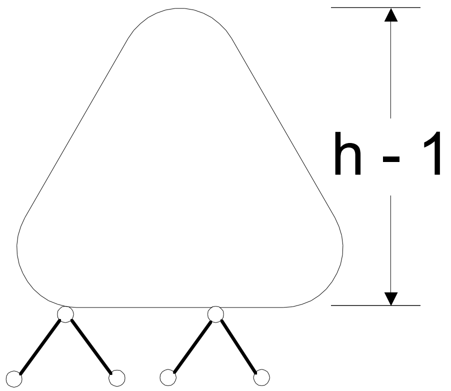
Complete Binary Trees

Complete binary tree of depth h =

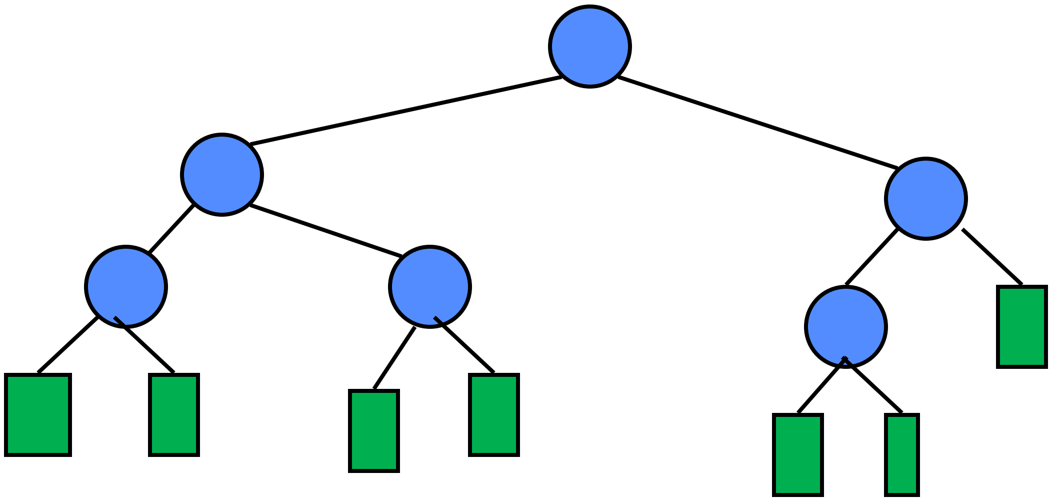
Perfect tree of depth (h-1) with one or more leaves at level h which are placed as left as possible.



Leaves go at the left



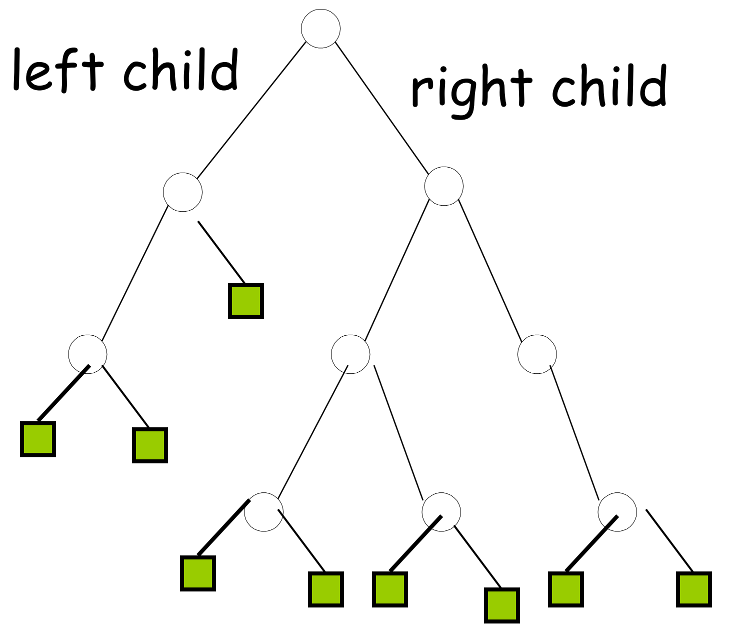
Binary Trees



In the book children are "completed" with "dummy" nodes

and all trees are considered FULL

Binary Trees + dummy leaves

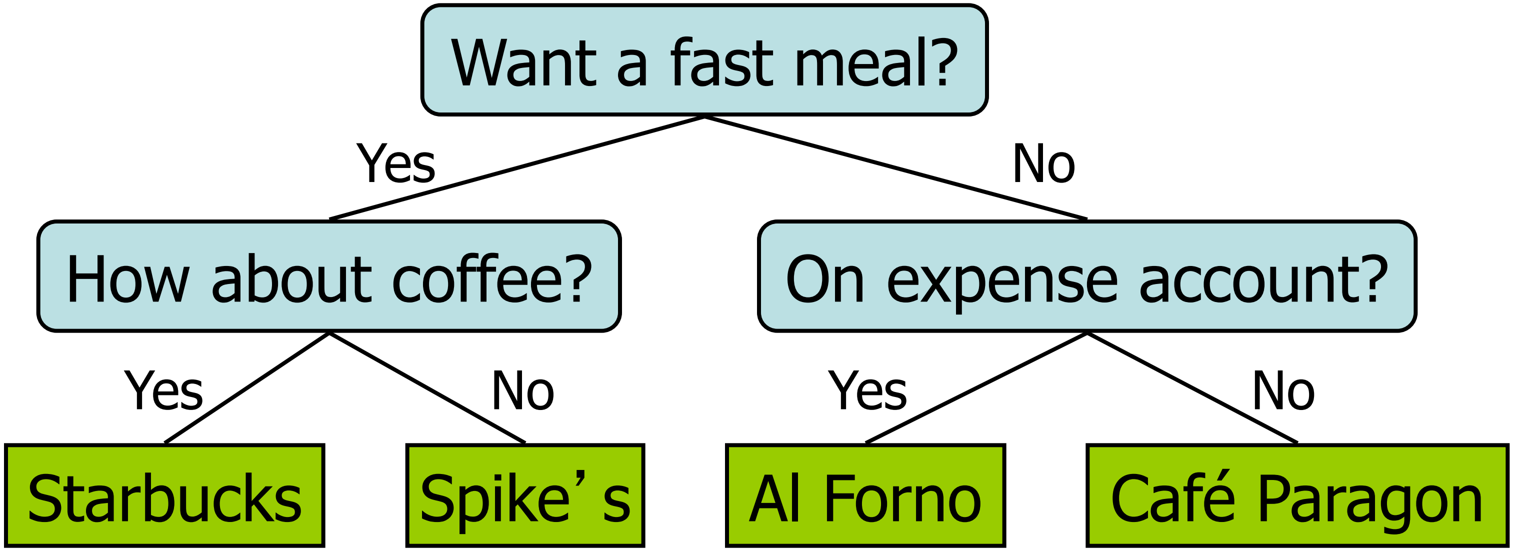


Each internal node has two children

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

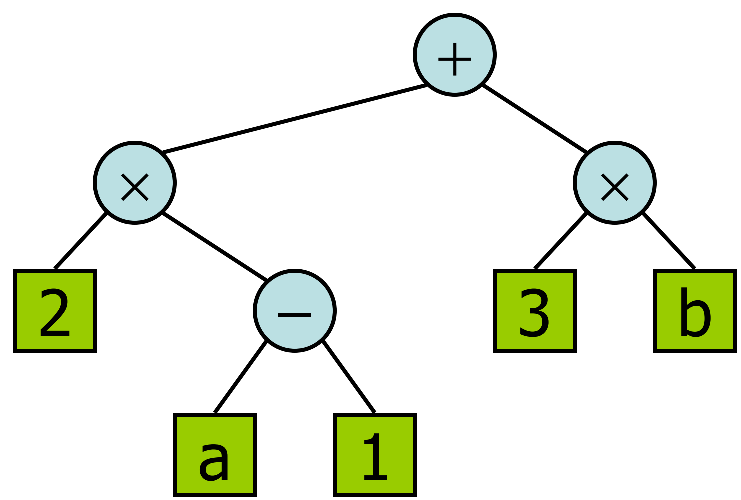


Examples of Binary Trees

Arithmetic Expression Tree

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

Example: arithmetic expression tree for the expression $((2 \times (a - 1)) + ((3 \times b)))$



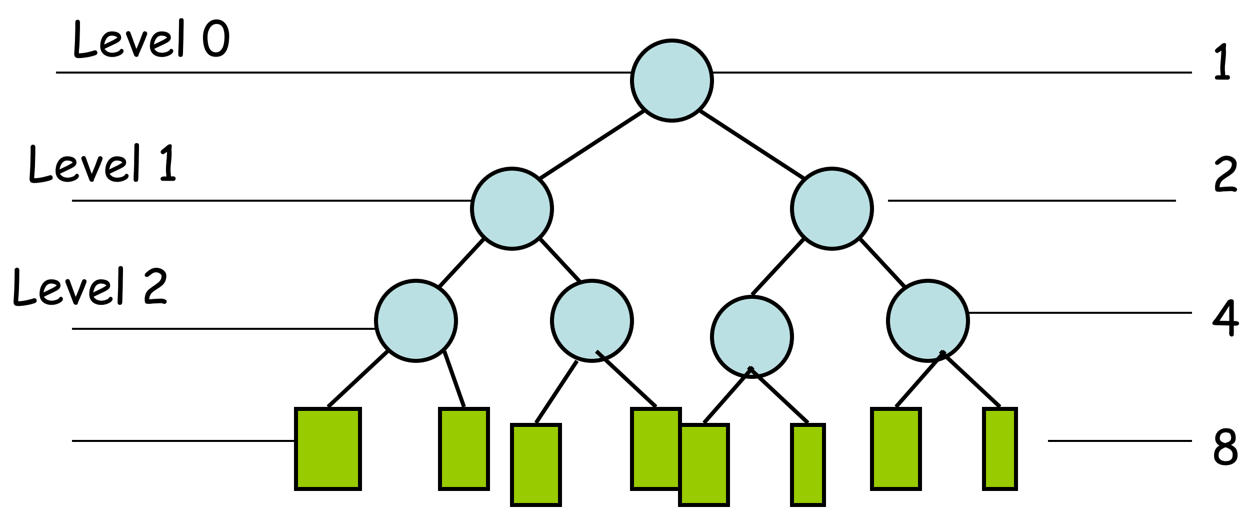
Properties of Binary Trees

- Notation

  n # of nodes   *e* # of leaves

  i # of internal nodes   *h* height

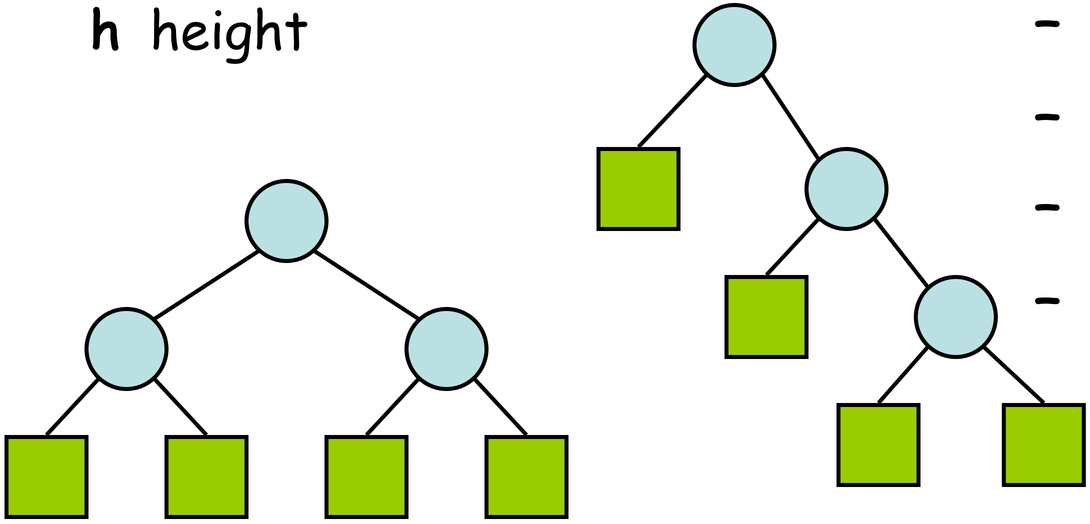
Maximum number of nodes at each level ?



level i ------- 2^i

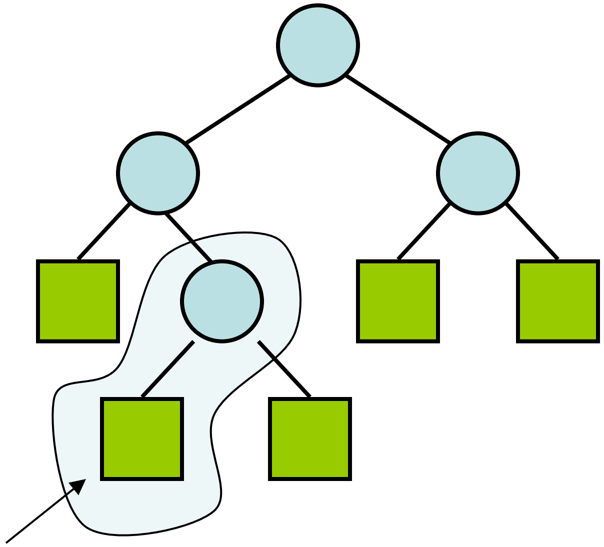
Properties of Full Binary Trees

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

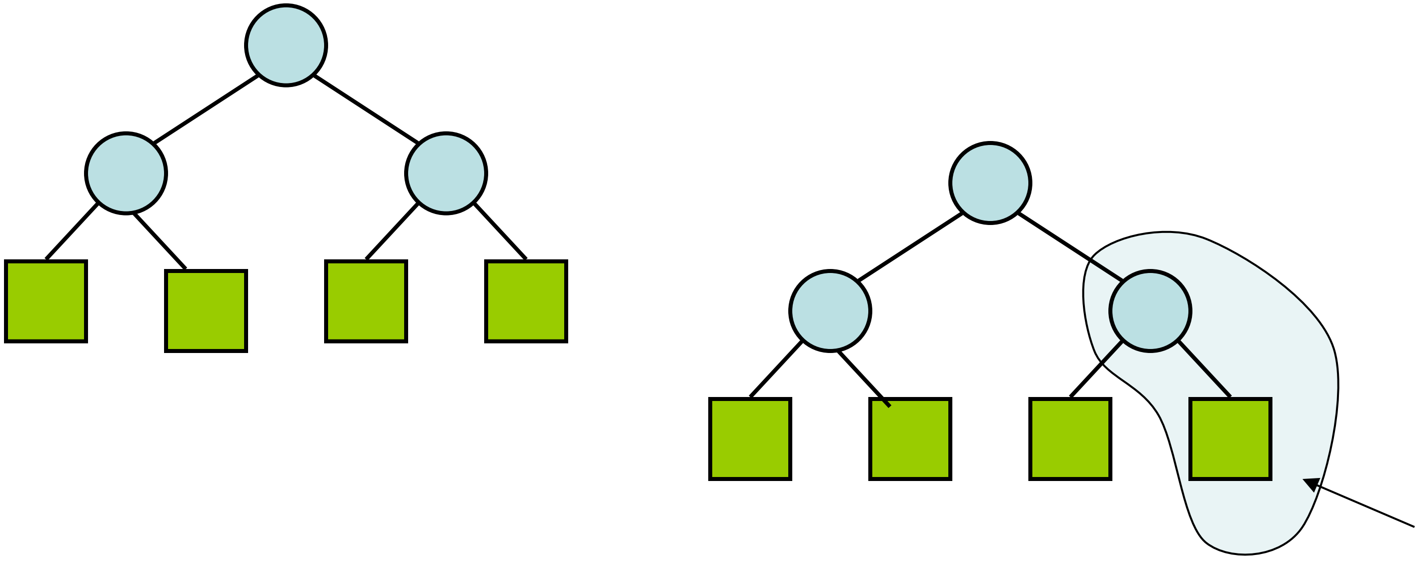


- Some Properties:
  - $e = i + 1$
  - $n = 2e - 1$
  - $h \leq i$
  - $h \leq (n - 1)/2$
  - $e \leq 2^h$
  - $h \geq \log_2 e$
  - $h \geq \log_2 (n + 1) - 1$

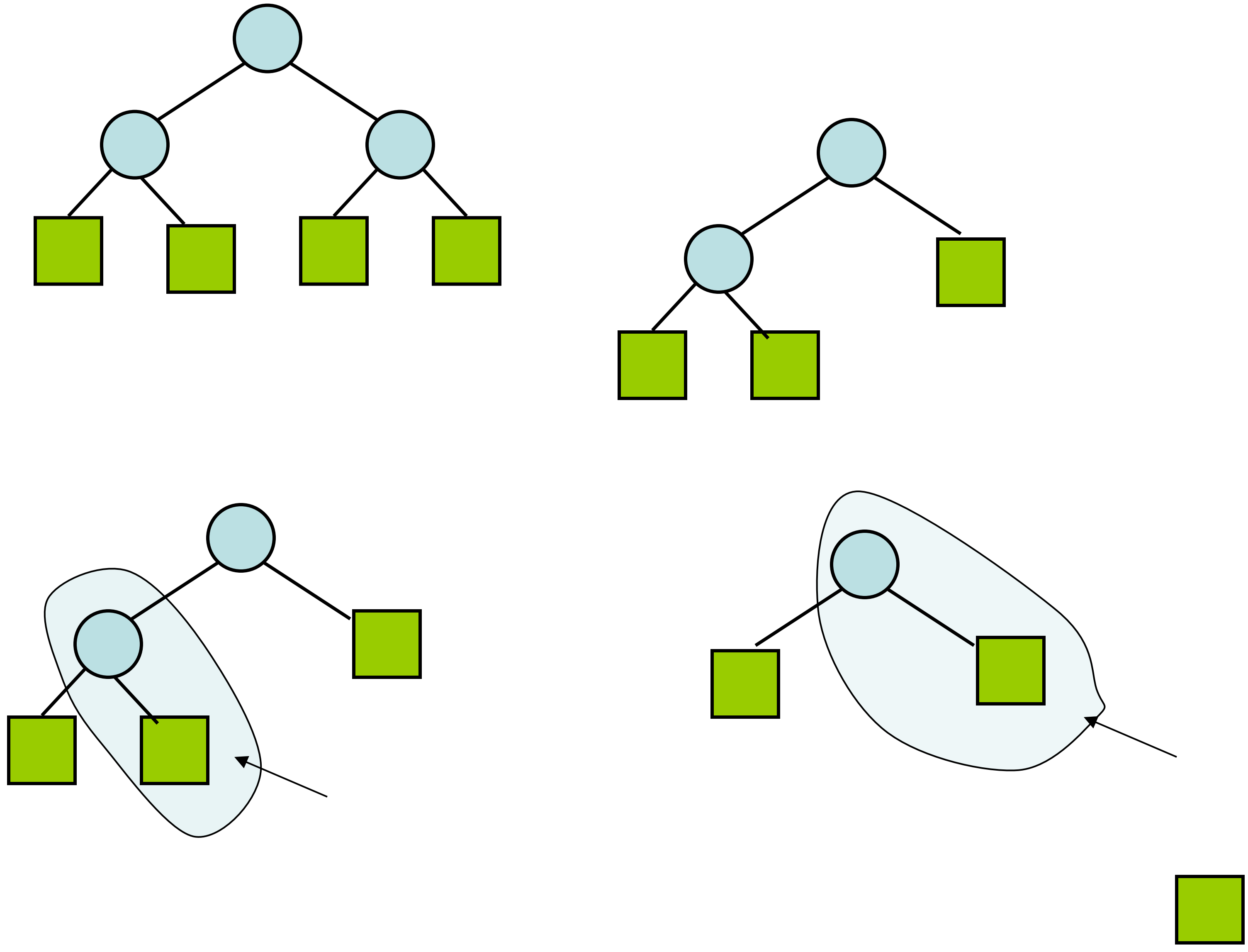
$e = i + 1$



$$e = i + 1$$



$$e = i + 1$$



$$\boxed{n = 2e - 1}$$

$n = i + e$

$e = i + 1$ (just proved)

$i = e - 1$

$n = e - 1 + e = 2e - 1$ $\longrightarrow$ $e = (n+1)/2$

also: $i + e = n$

$\Rightarrow\ i = n - e$

$= n - (n+1)/2$

$\Rightarrow\ i = (2n - n - 1)/2$

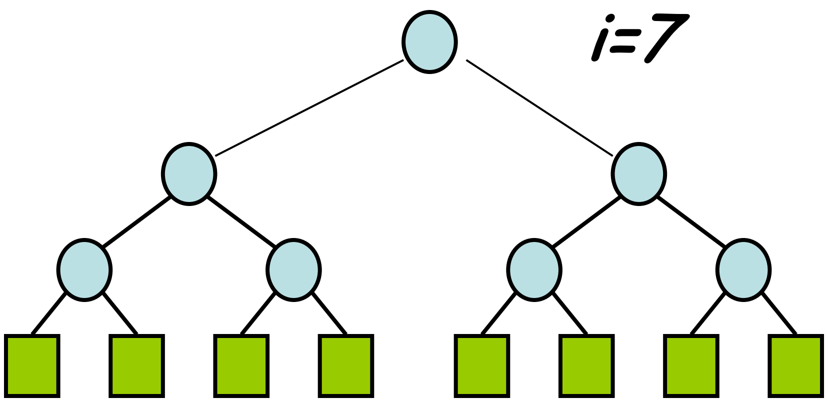
$$\boxed{i = (n-1)/2}$$

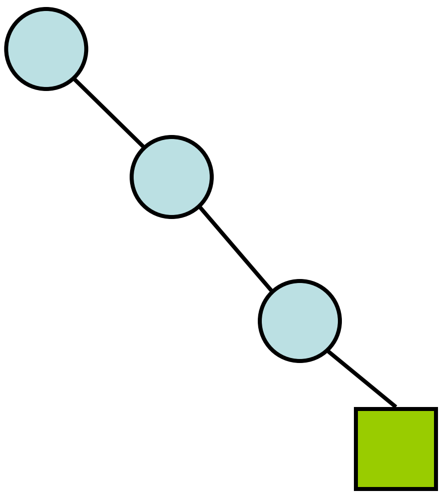
$\boldsymbol{h \leq i}$

(h = max n. of ancestors)

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

Ex: h=3, i=7

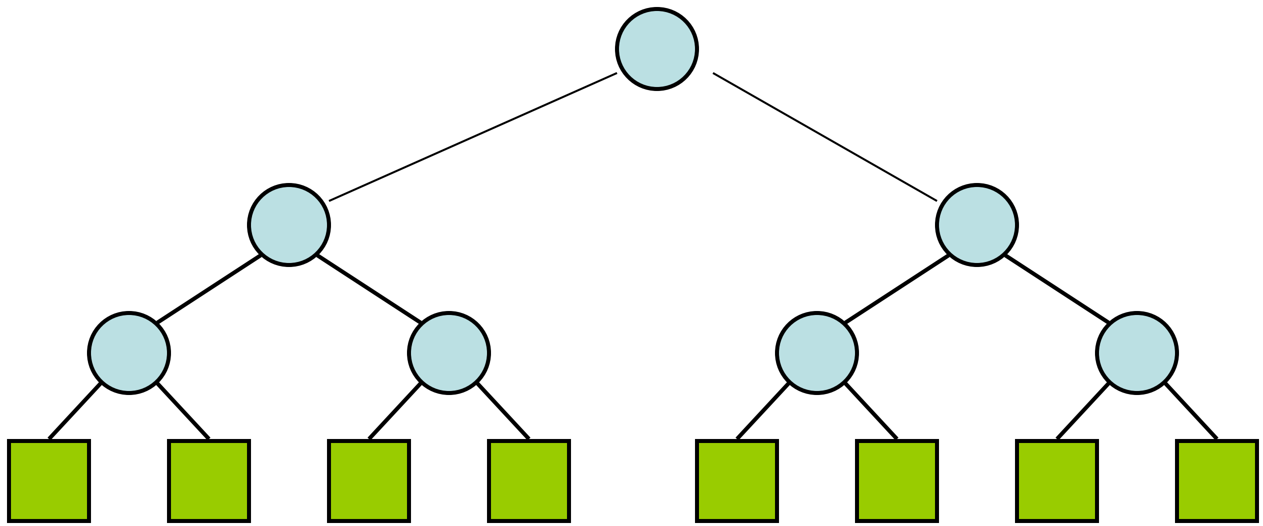




Ex: h=3, i=3

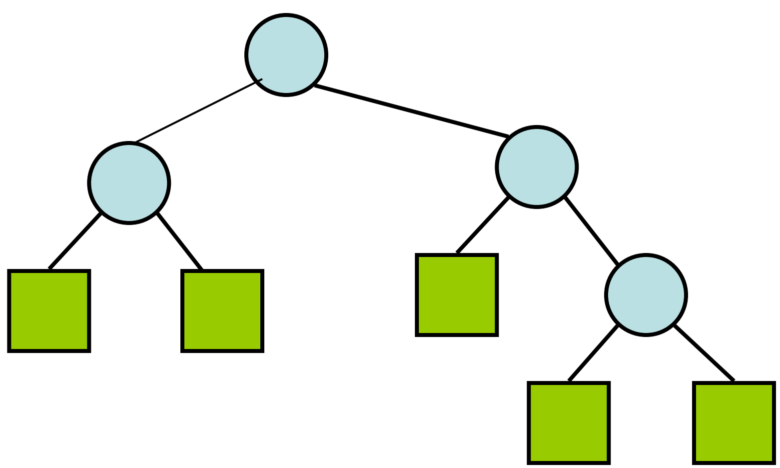
$e \le 2^h$

level i ------- max n. of nodes is 2^i



h = 3

2^3 leaves
if all at last
level h



otherwise less

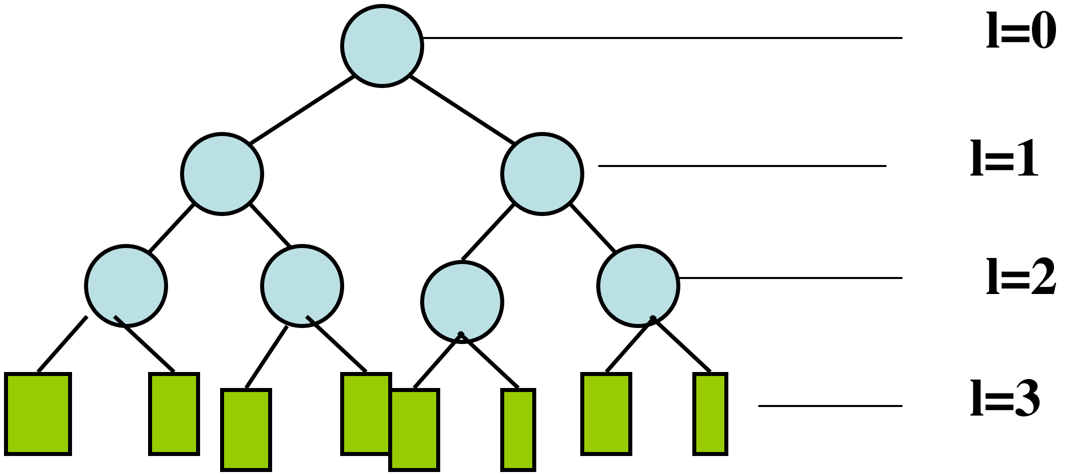
Since $e \le 2^h$

$\log_2 e \le \log_2 2^h$

$\log_2 e \le h \longrightarrow \boxed{h \ge \log_2 e}$

In Perfect Binary Trees

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



WHY ?

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

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

As a consequence:

In Binary trees:

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

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

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

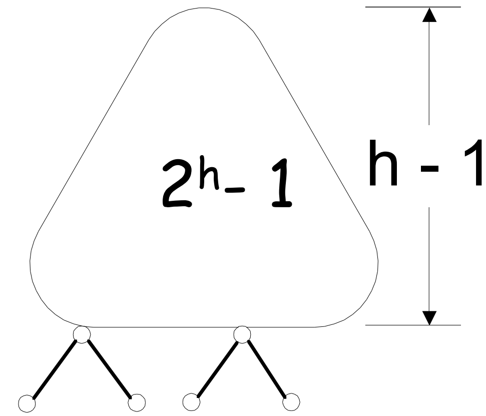
$$\log_2 (n+1) \le h+1$$

$$\boxed{h \ge \log_2 (n+1) - 1}$$

Complete Binary Trees

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

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



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

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

$$\boxed{h \leq \log_2(n) < h+1}$$

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

$\lfloor \log n \rfloor$

Summary of Important Properties

log is $\log_2$

Binary Trees

$$h+1 \le n \le 2^{h+1} - 1$$
$$1 \le e \le 2^{h}$$
$$h \le i \le 2^{h} - 1$$
$$\log(n+1) - 1 \le h \le n-1$$

Full Binary Trees

$$2h+1 \le n \le 2^{h+1} - 1$$
$$h+1 \le e \le 2^{h}$$
$$h \le i \le 2^{h} - 1$$
$$\log(n+1) - 1 \le h \le (n-1)/2$$

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 : $n \geq 2^h$ $\quad h = floor(\log n)$
  (integer part of log n)
- ✓Binary - perfect: $n = 2^{h+1} - 1$ $\quad h = \log(n+1) - 1$

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

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)
```

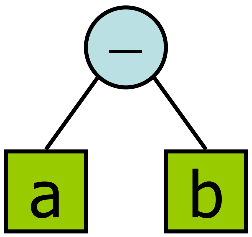
Traversing Binary Trees

Inorder

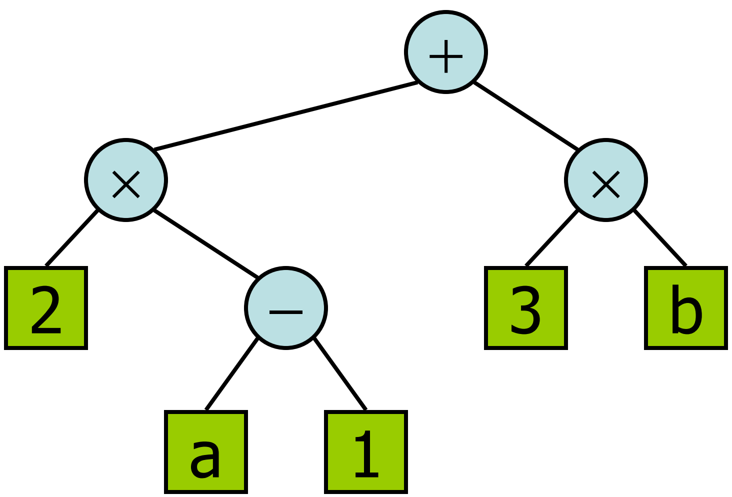
(Depth-first)

```
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:

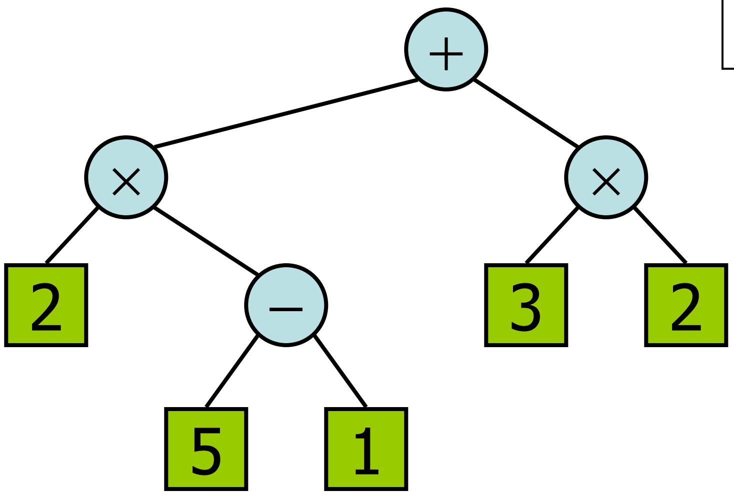
$2 \times a - 1 + 3 \times b$

Postorder:

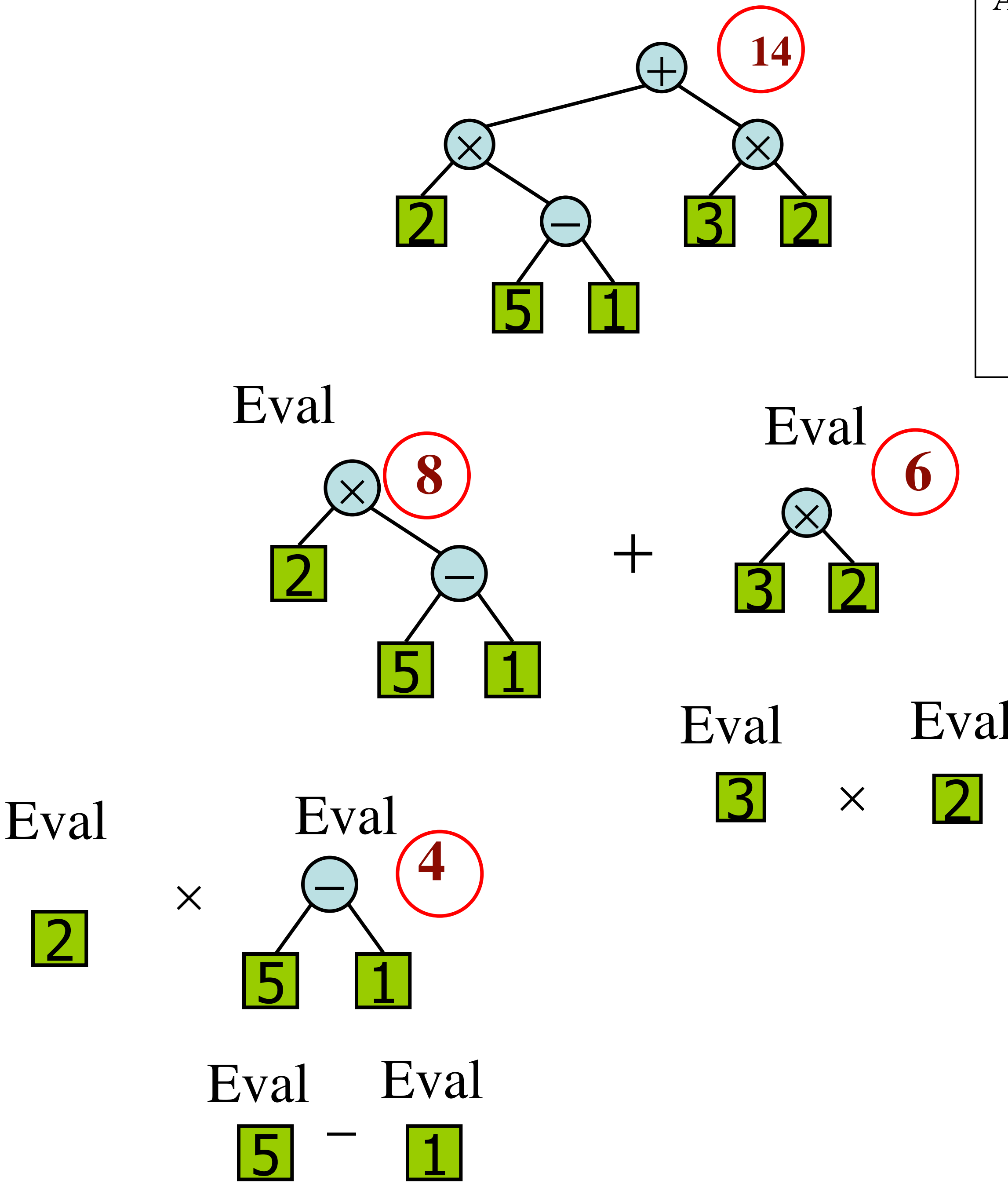
2 a 1 - $\times$ 3 b $\times$ +

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 ← *evalExpr*(*leftChild* (*v*))
        y ← *evalExpr*(*rightChild* (*v*))
        ◊ ← operator stored at *v*
        return *x* ◊ *y*

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(“(’ ’)

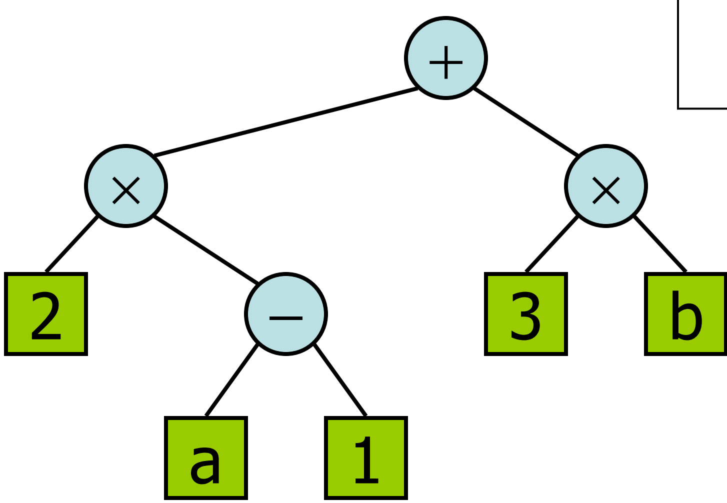
        printInOrder (*leftChild*(*v*))

    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)

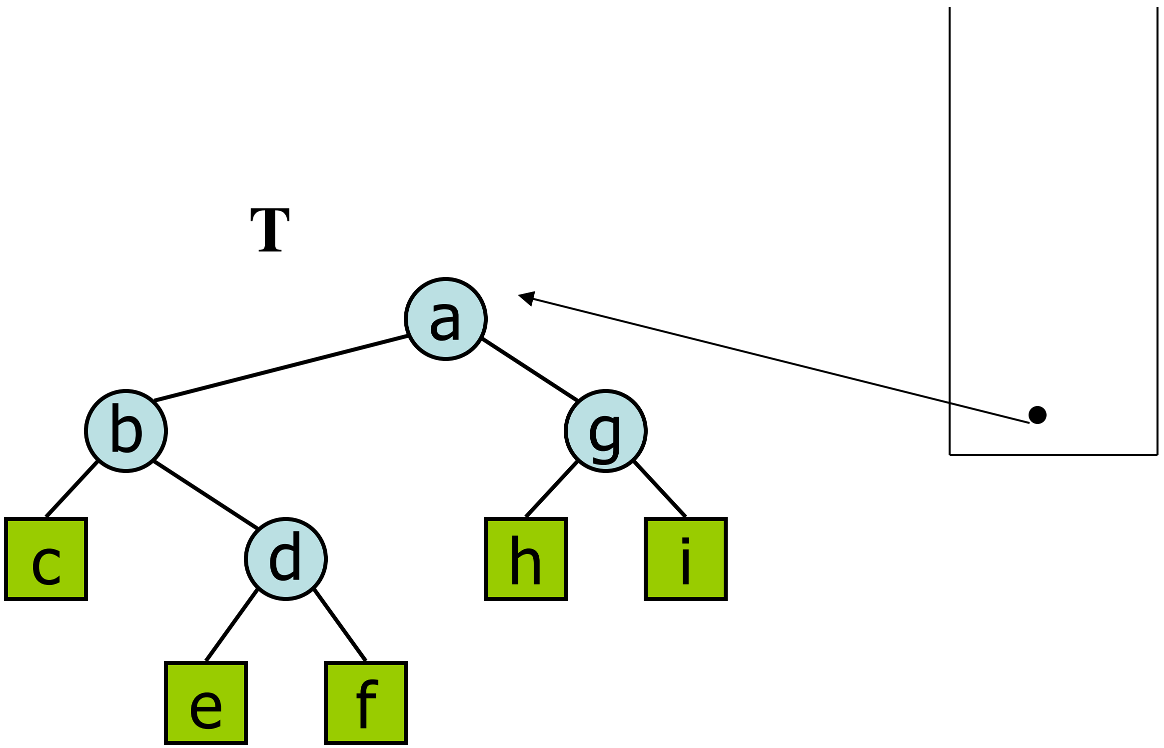
```
    Stack S
    TreeNode N

    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)

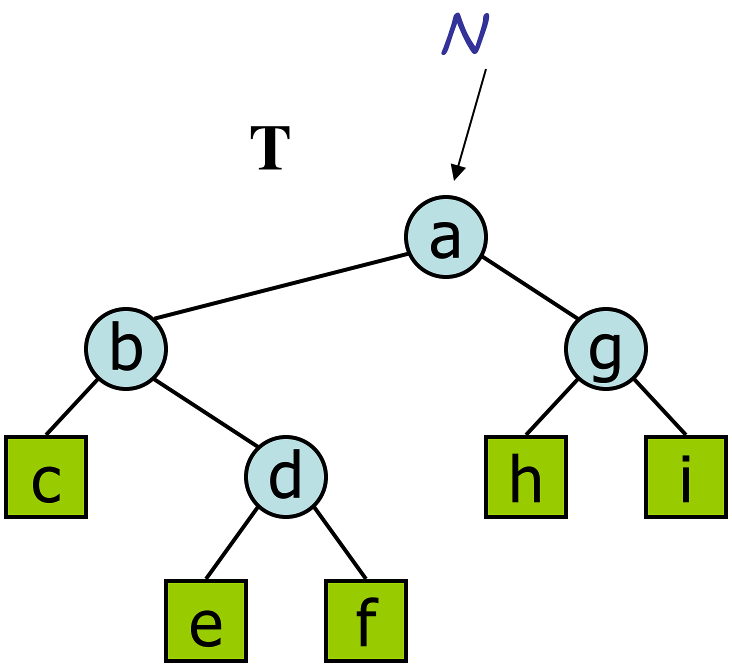


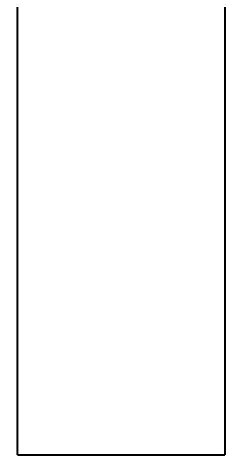
Algorithm *preOrderTraversalwithStack*(T)

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

N = S.pop()

print(N.elem)



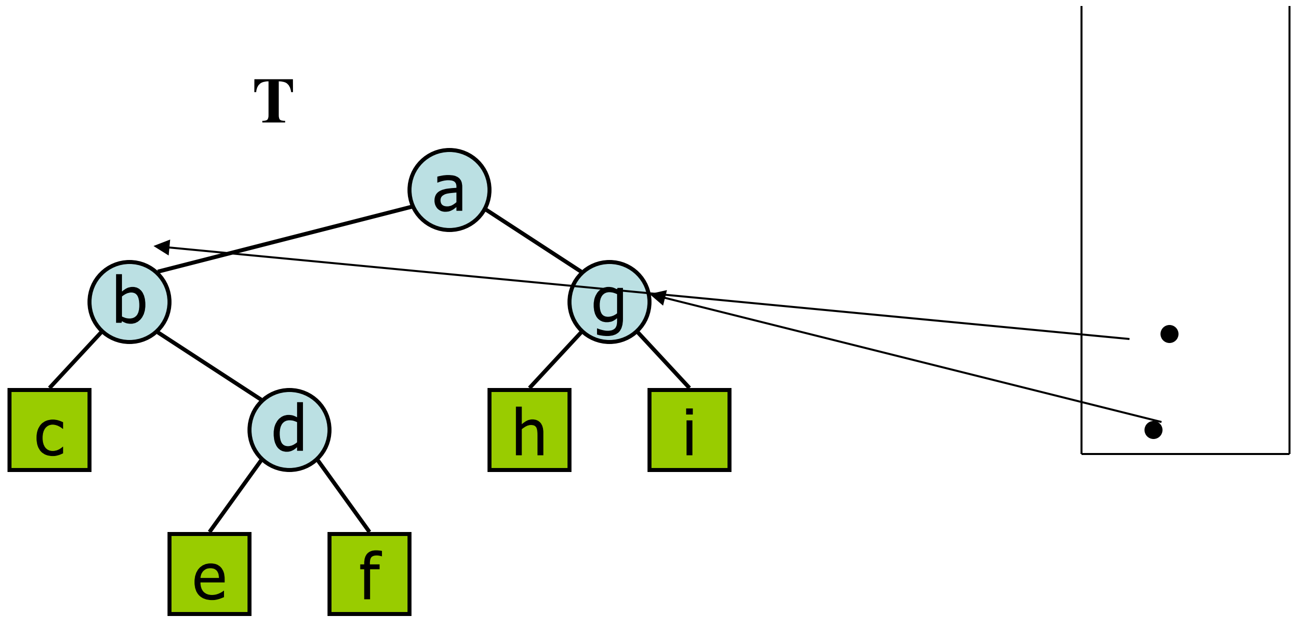


a

Algorithm *preOrderTraversalwithStack*(T)

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

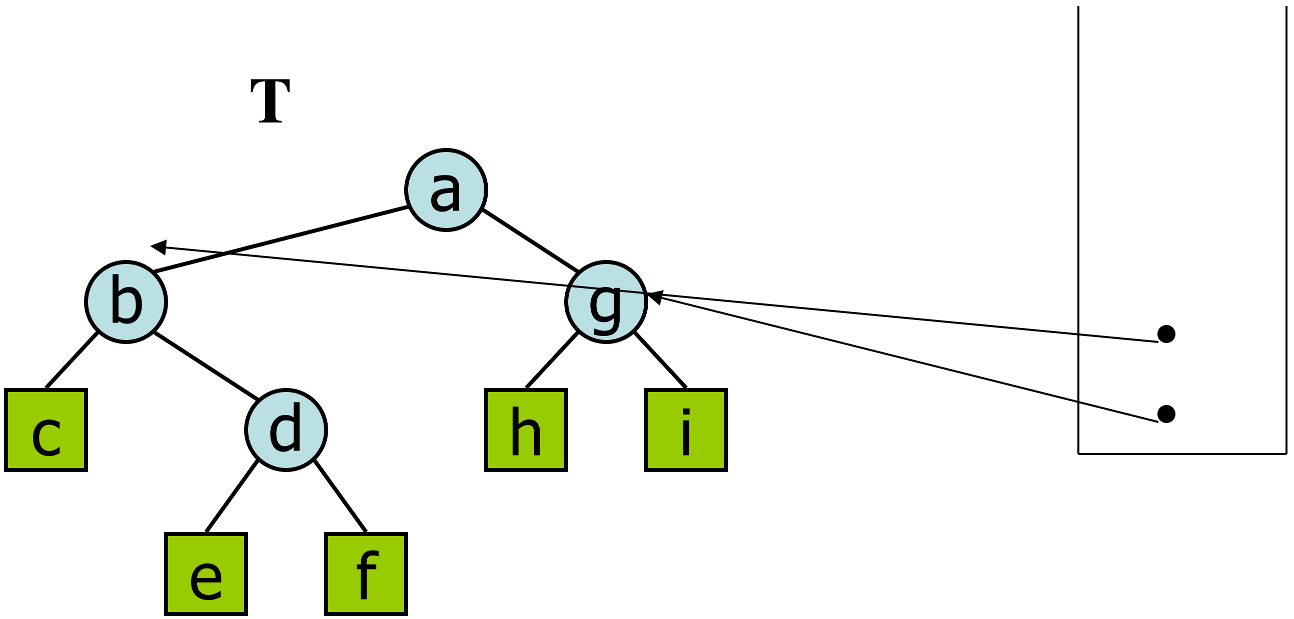
S.push(N.leftChild) // push the reference to the left child



a

Algorithm *preOrderTraversalwithStack*(T)

N = S.pop()

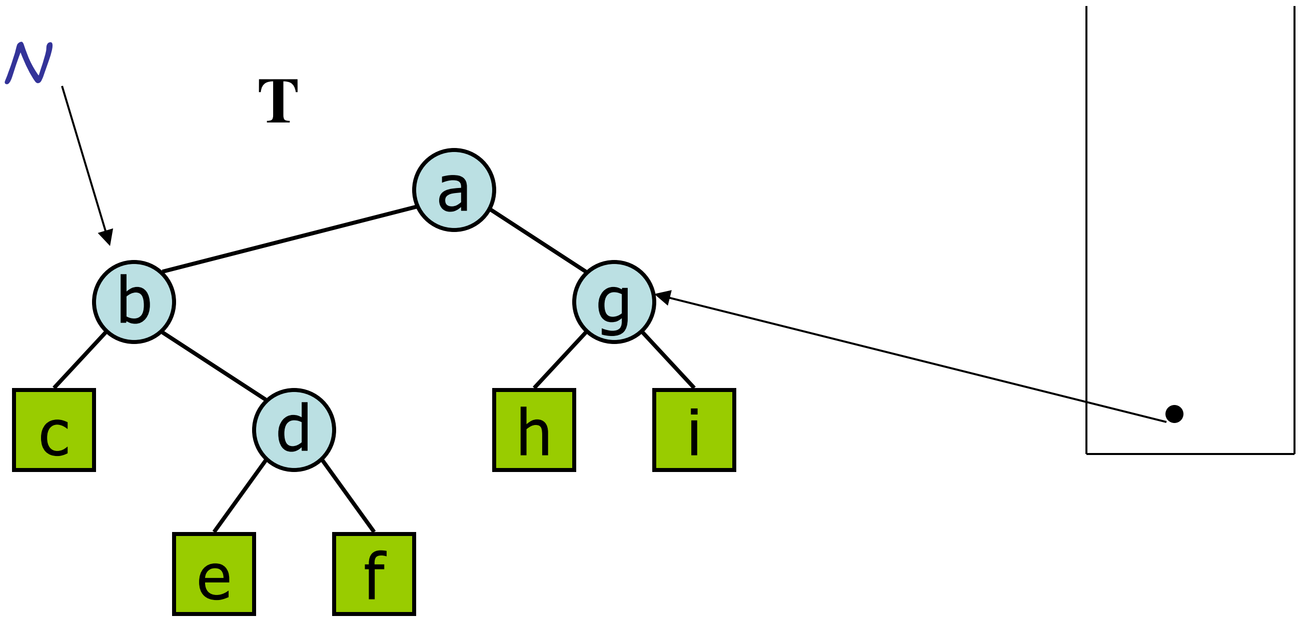


a

Algorithm *preOrderTraversalwithStack*(T)

N = S.pop()

print(N.elem)

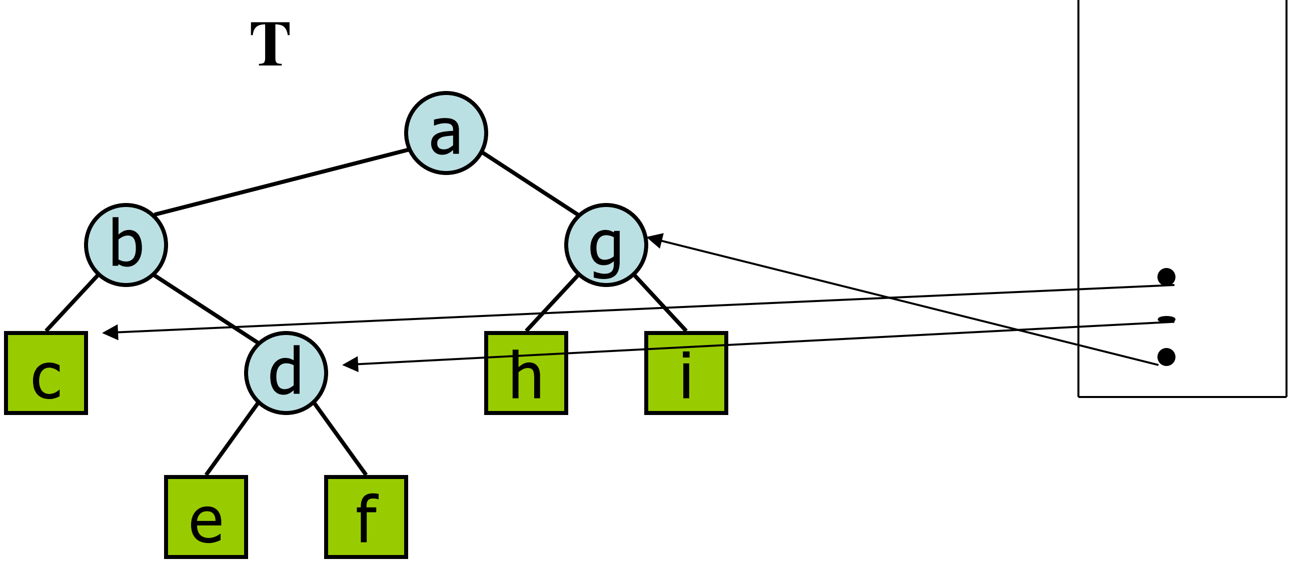


a b

Algorithm *preOrderTraversalwithStack*(T)

S.push(N.rightChild)

S.push(N.leftChild)

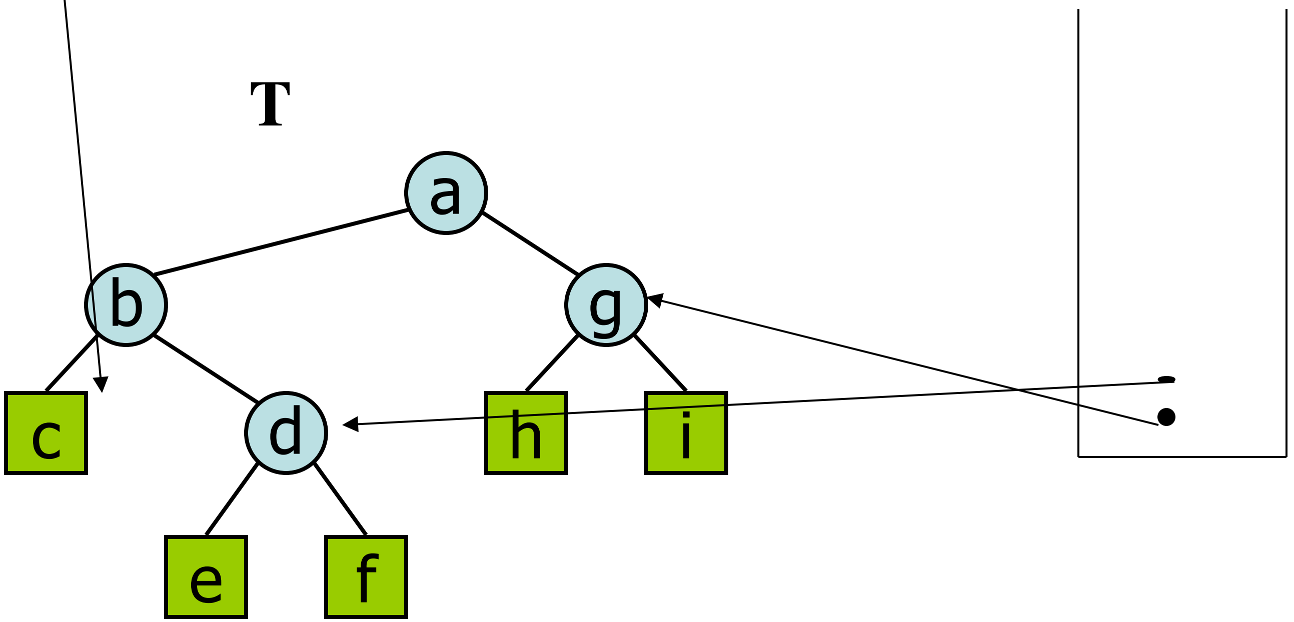


a b

Algorithm *preOrderTraversalwithStack*(T)

N = S.pop()

print(N.elem)

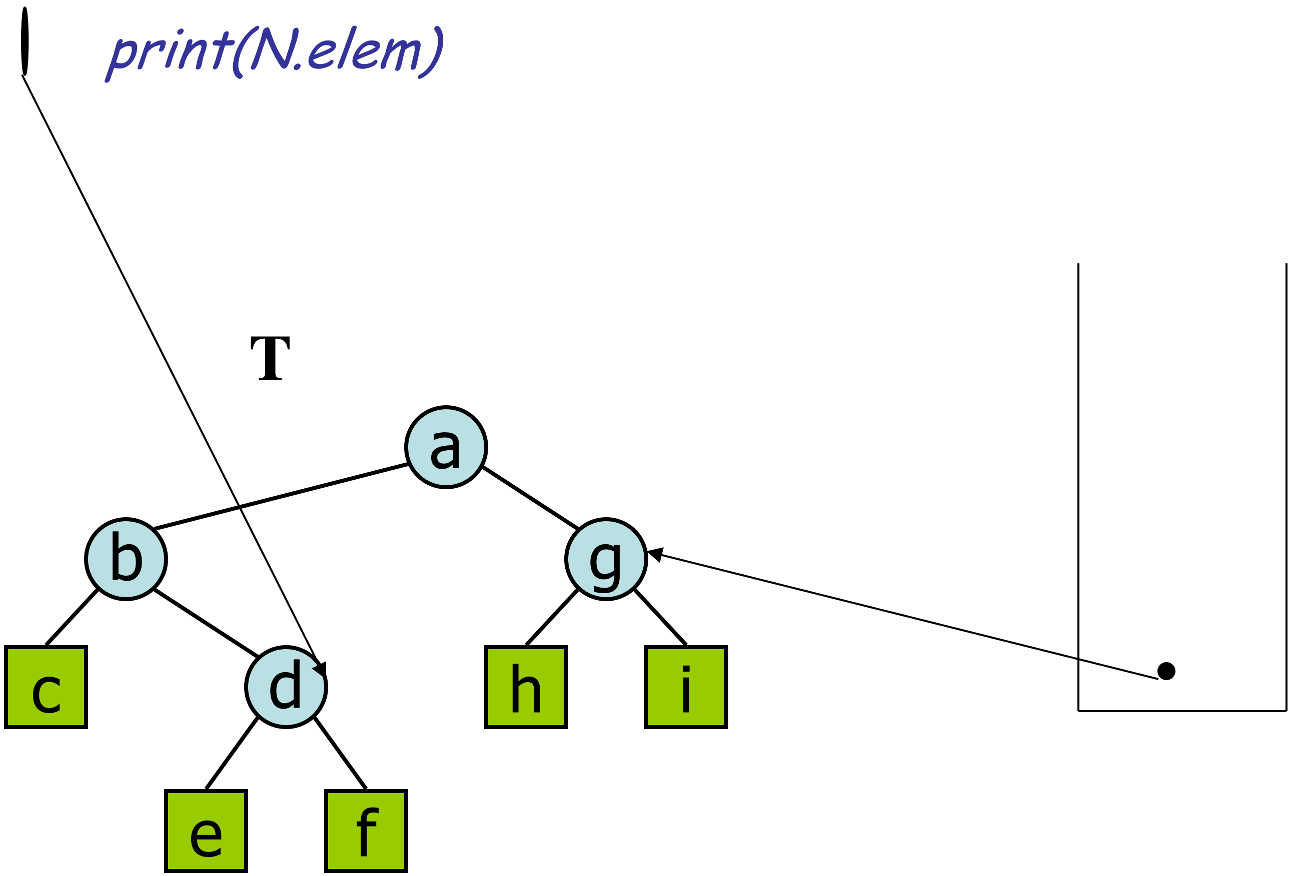


a b c

Algorithm *preOrderTraversalwithStack*(T)

N = S.pop()

print(N.elem)

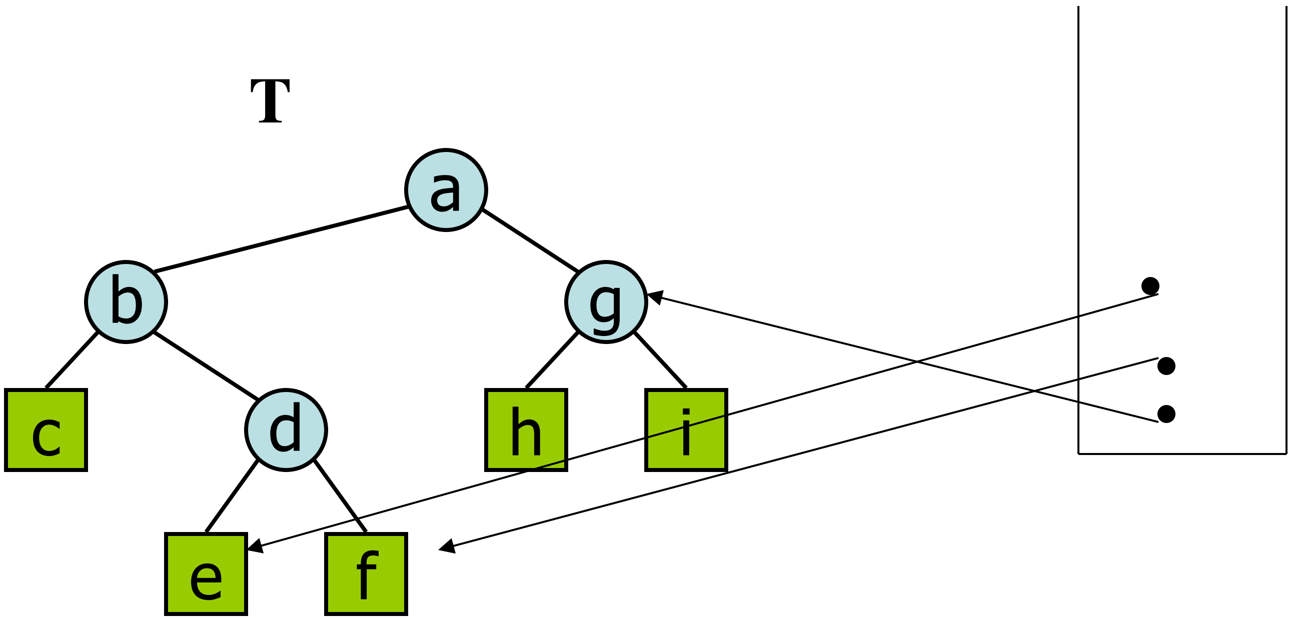


a b c d

Algorithm *preOrderTraversalwithStack*(T)

S.push(N.rightChild)

S.push(N.leftChild)

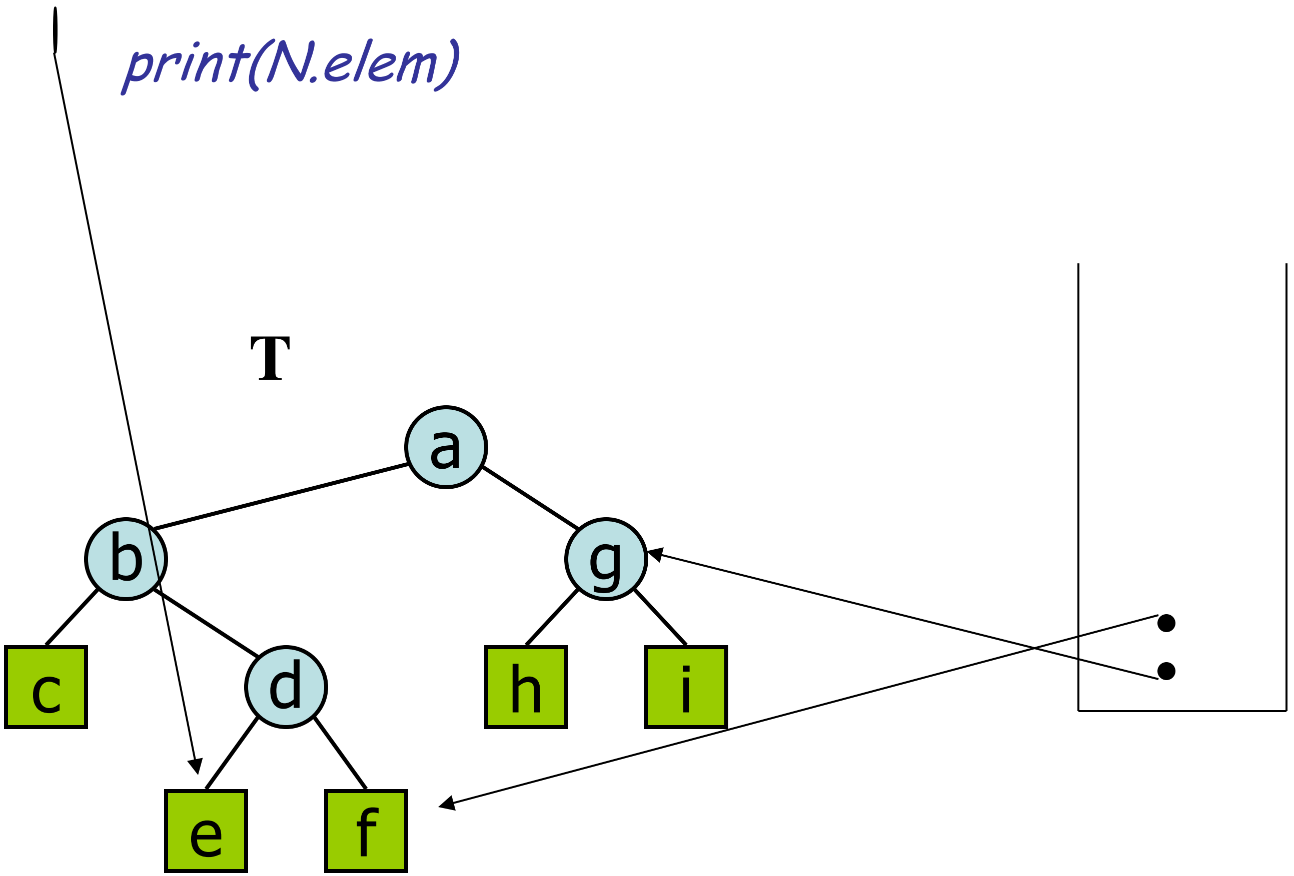


a b c d

Algorithm *preOrderTraversalwithStack*(T)

N = S.pop()

print(N.elem)

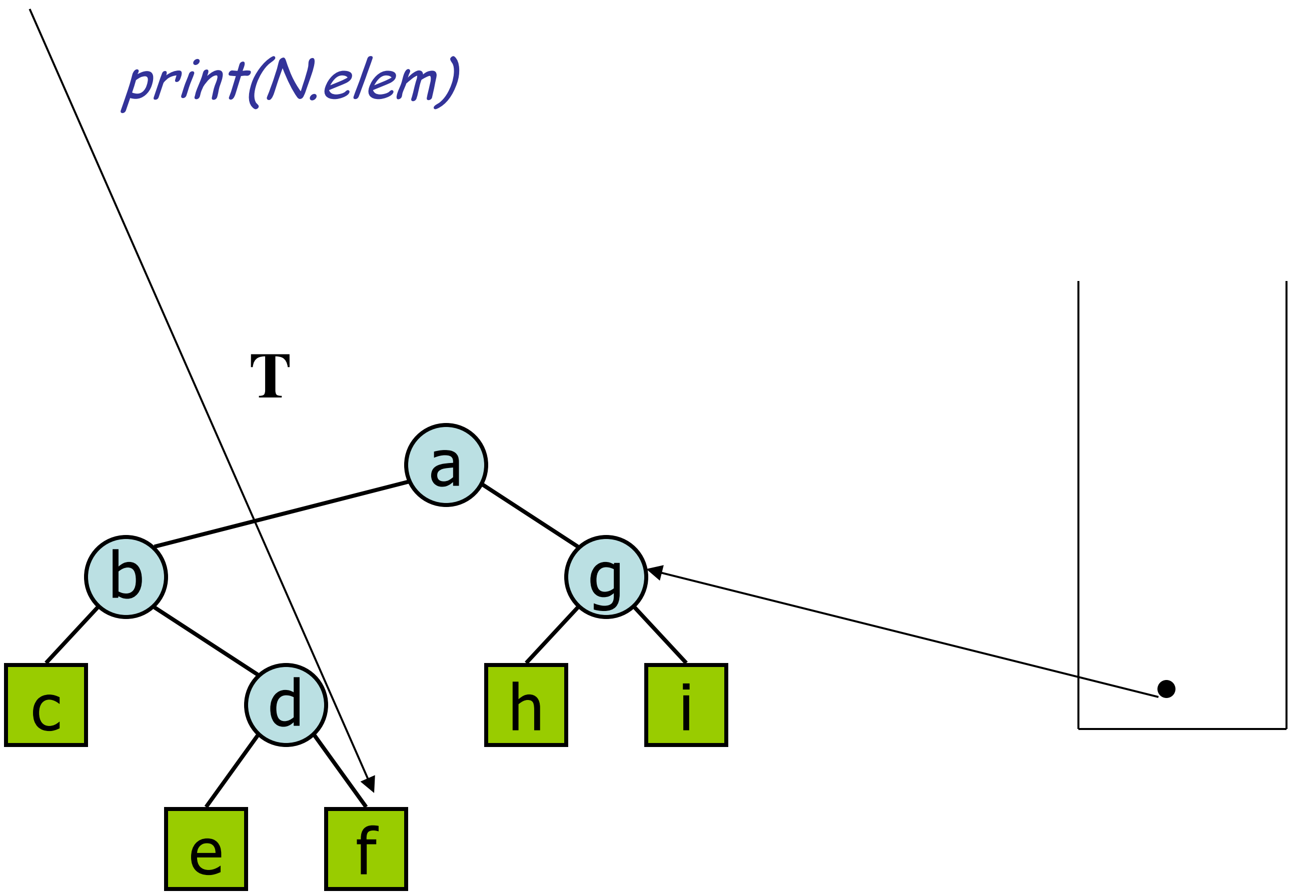


a b c d e

Algorithm *preOrderTraversalwithStack*(T)

N = S.pop()

print(N.elem)

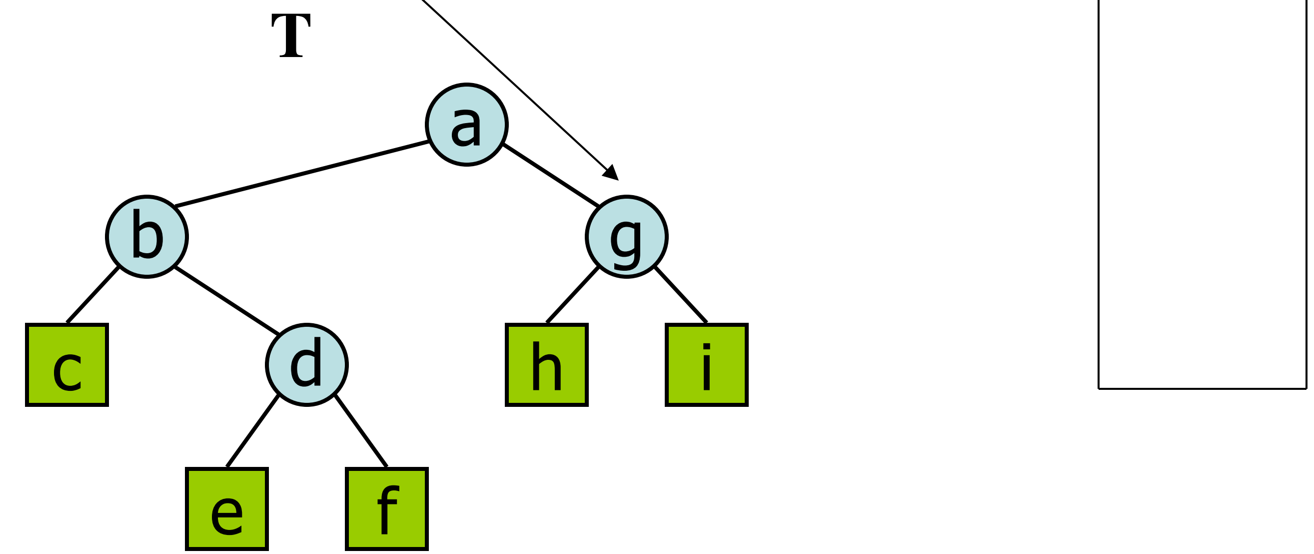


a b c d e f

Algorithm *preOrderTraversalwithStack*(T)

N = S.pop()

print(N.elem)

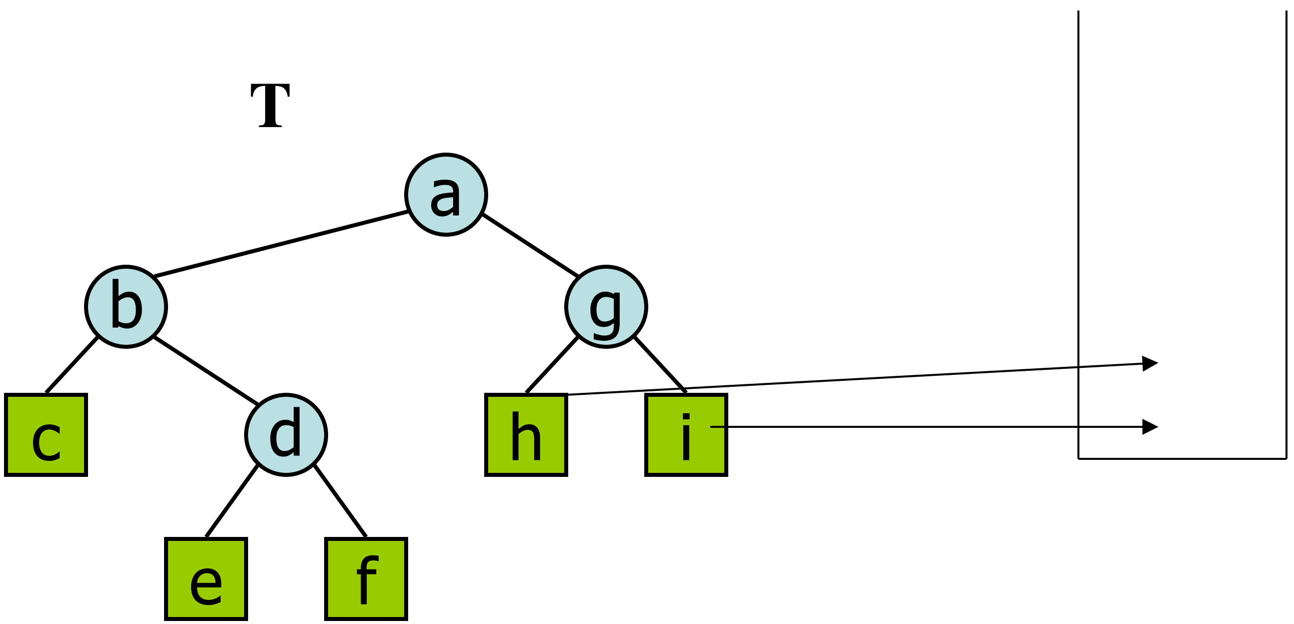


a b c d e f g

Algorithm *preOrderTraversalwithStack*(T)

S.push(N.rightChild)

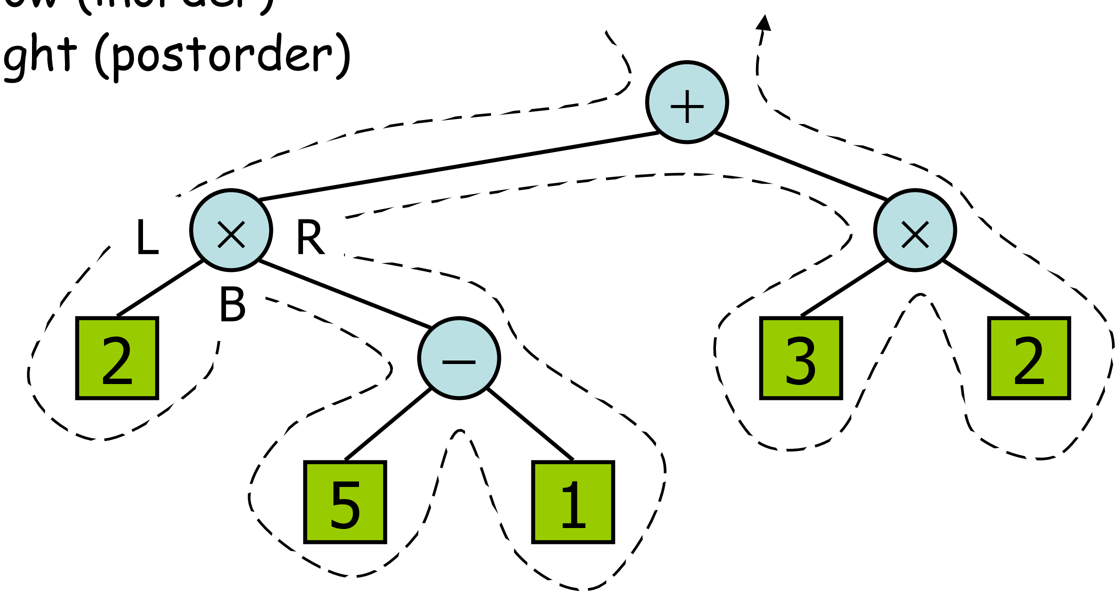
S.push(N.leftChild)



a b c d e f g

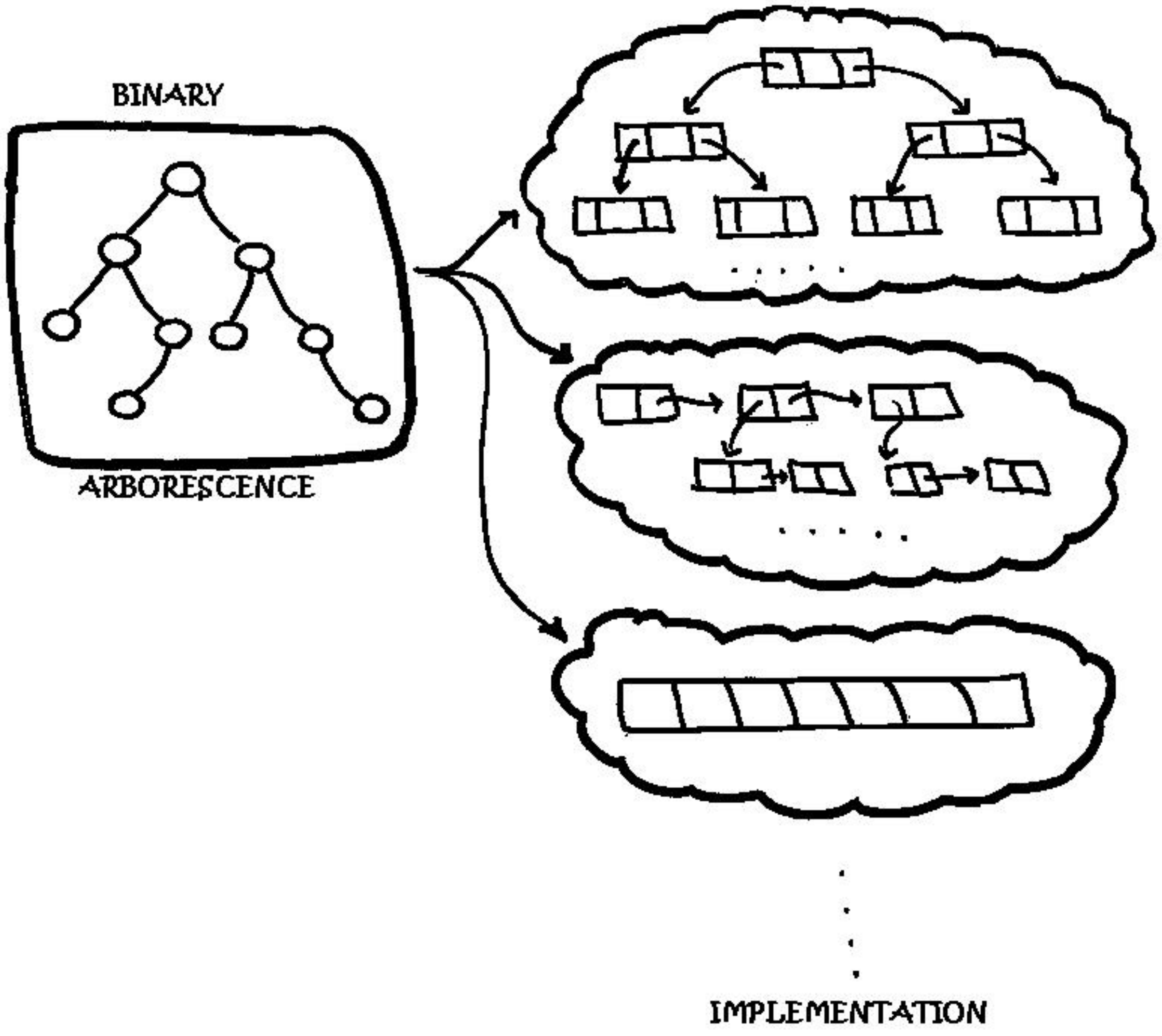
Euler Tour Traversal

- 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)



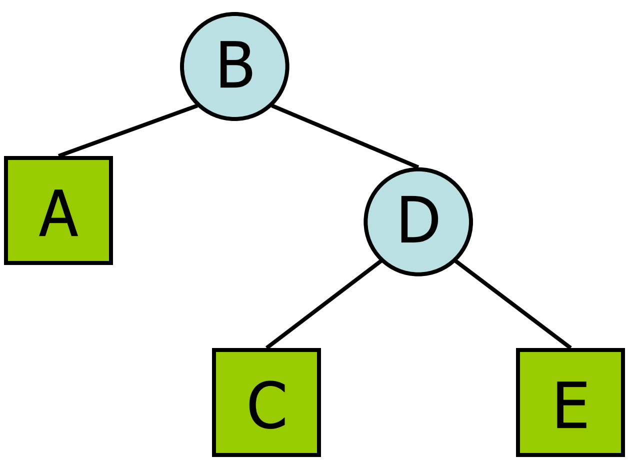
```
Algorithm eulerTour(T,v)

    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)
```

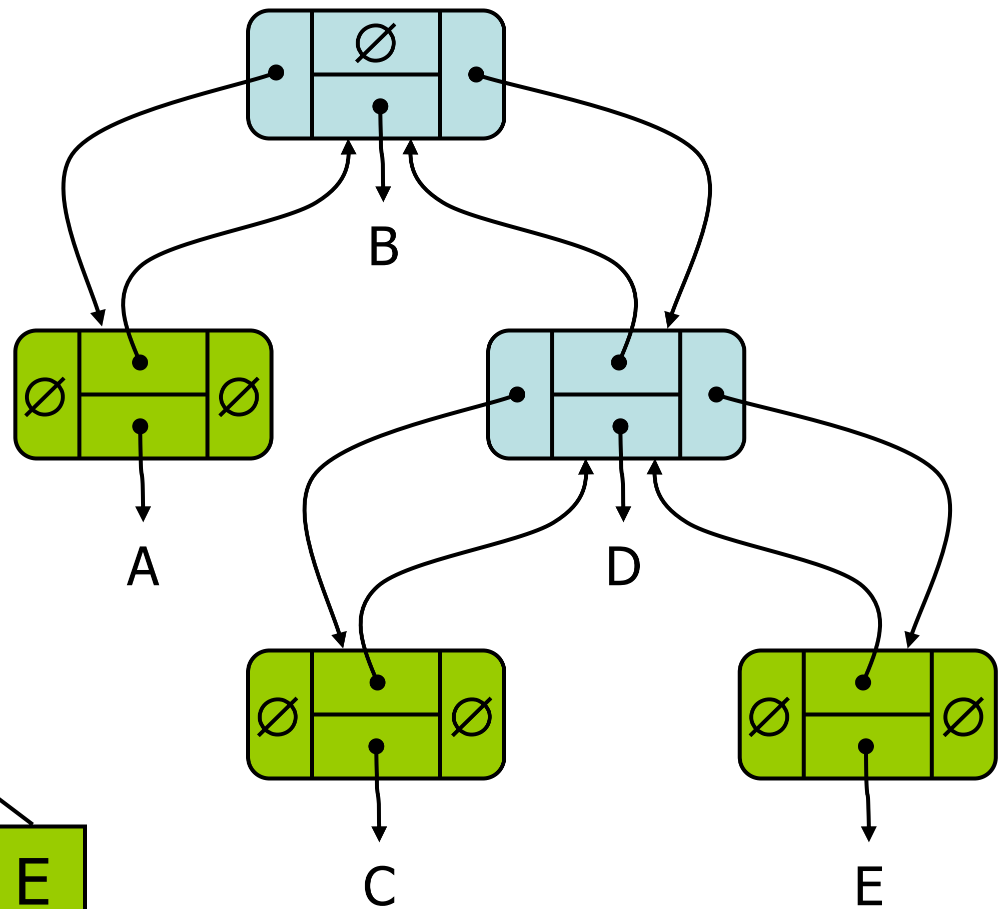
Implementations of Binary trees….



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





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

Input: Position Output: Position

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

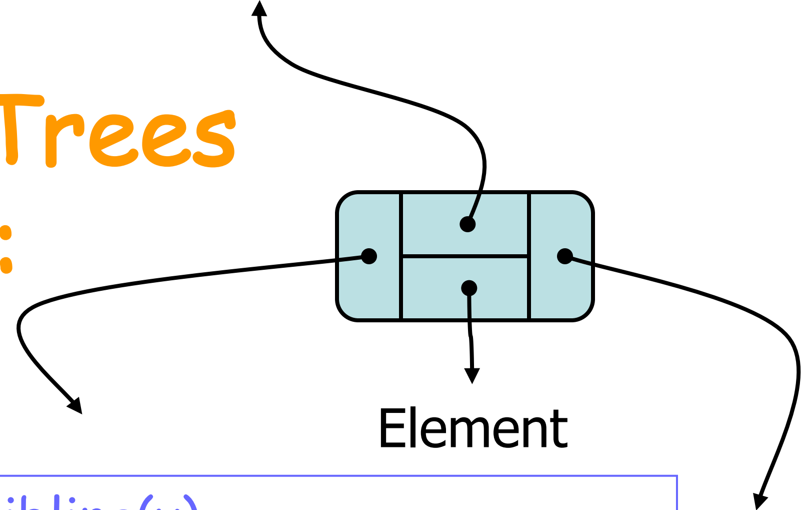
replaceElement(p,e) Input: Position and an object Output: Object

isRoot(p) Input: Position Output: Boolean

isInternal(p) Input: Position Output: Boolean

isExternal(p) Input: Position Output: Boolean

Implemented Binary Trees with linked structure: ADT BTNode




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

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

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

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

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

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

swapElements(p,q),

replaceElement(p,e)

isRoot(p),

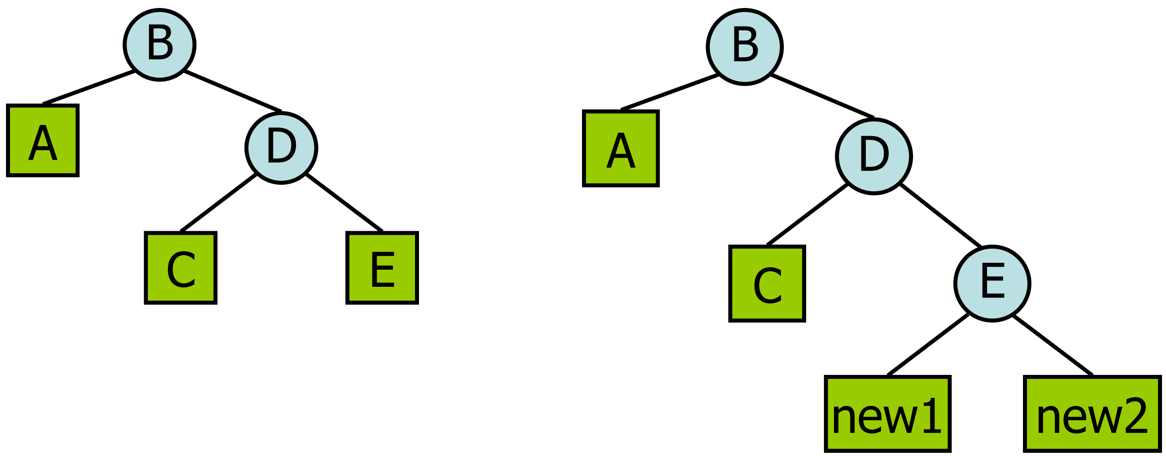
isInternal(p),

isExternal(p)

$O(1)$

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):

if isExternal(v)

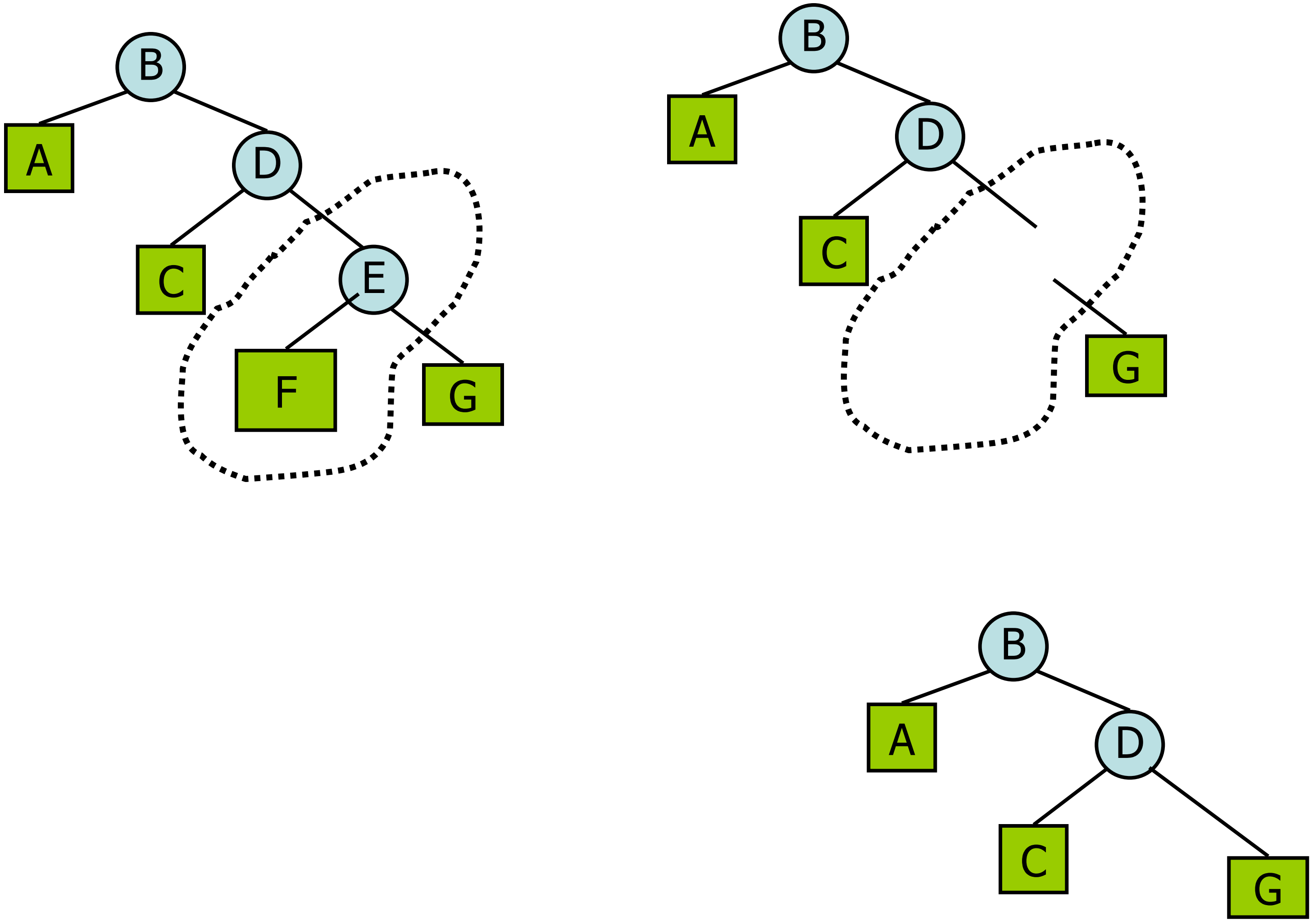
create new nodes new1 and new 2

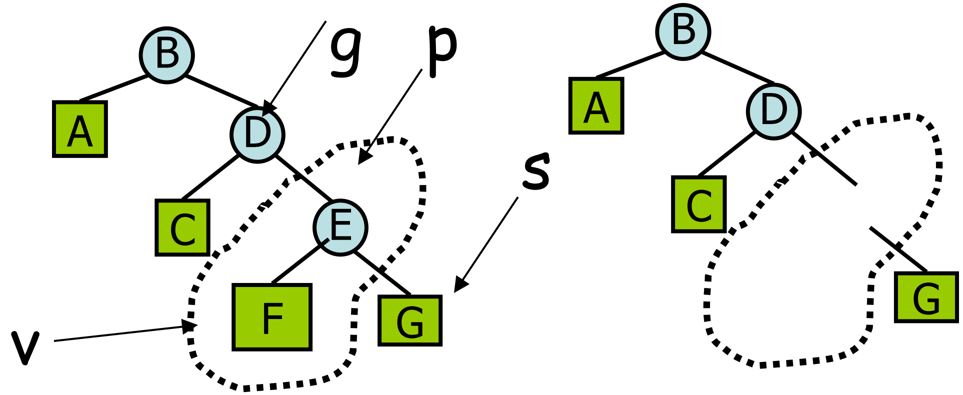
v.left ← new1

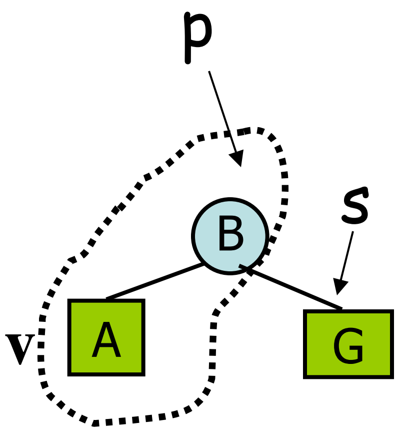
v.right ← new2

size ← size +2

removeAboveExternal(v):





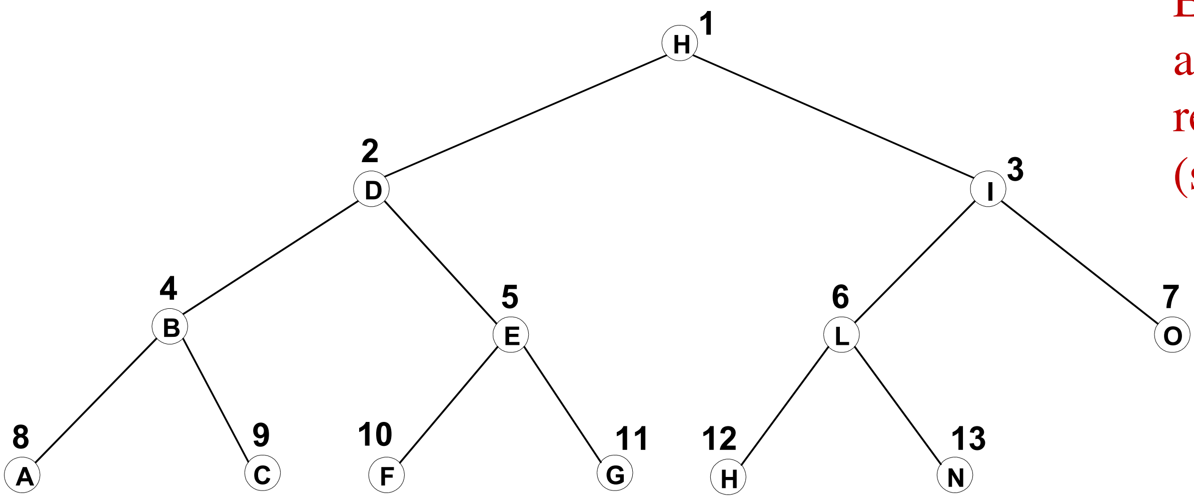


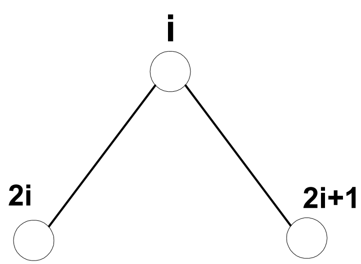
removeAboveExternal(v):

```
if isExternal(v) and (size >= 3) {
    p ← parent(v)
    s ← sibling(v)
    if isRoot(p)  {
        s.parent  ←  null
        root ← s
        }
    else {
        g ← parent(p)
        if p is leftChild(g)  g.left ← s
        else  g.right ← s
        s.parent ← g
        }
    size ← size - 2
}
```

Implemented Binary Trees with Array List

Exercise: Start the numbering at 0 instead of 1and derive the respective formulas!
(see class on heaps)





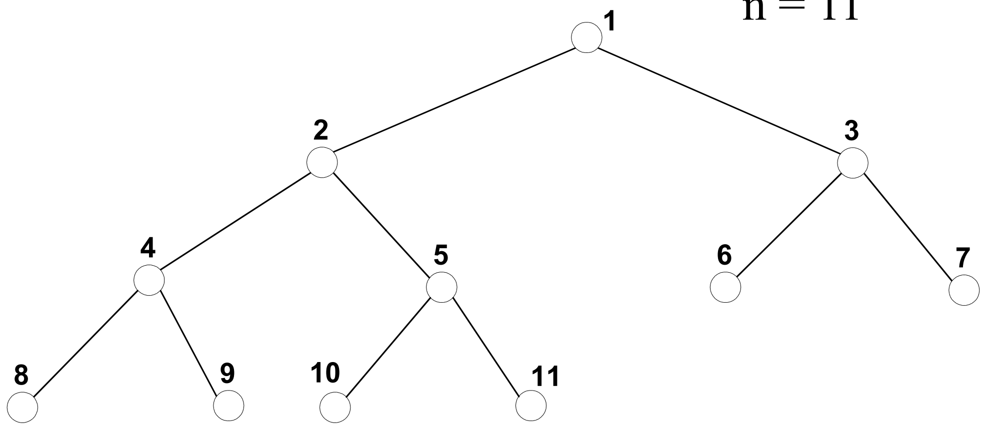
1	2	3	4	5	6	7	8	9	10	11	12	13
H	D	I	B	E	L	O	A	C	F	G	H	N

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

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

They all have complexity O(1)

$n = 11$



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 div 2]	if	$i > 1$
The Root	T[1]	if	$T \neq 0$
Leaf? T[i]	TRUE	if	$2i > n$

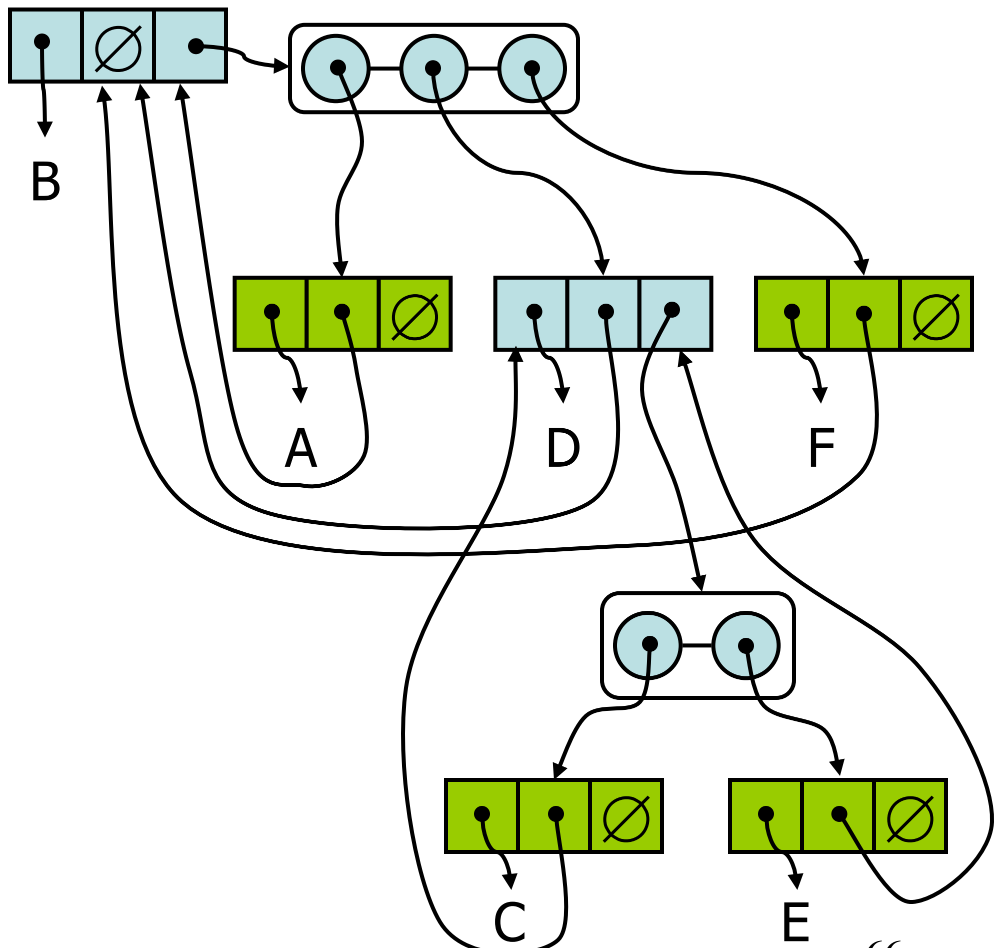
leftChild(p), rightChild(p), sibling(p):

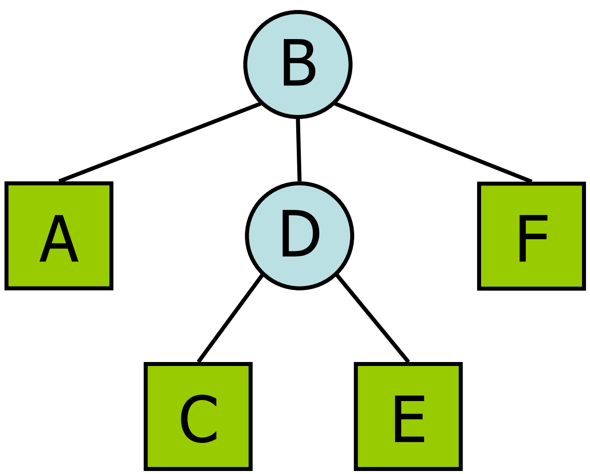
swapElements(p,q),
replaceElement(p,e)
isRoot(p), isInternal(p),
isExternal(p)

They all have complexity O(1)

Implementing General Trees with a Linked Structure

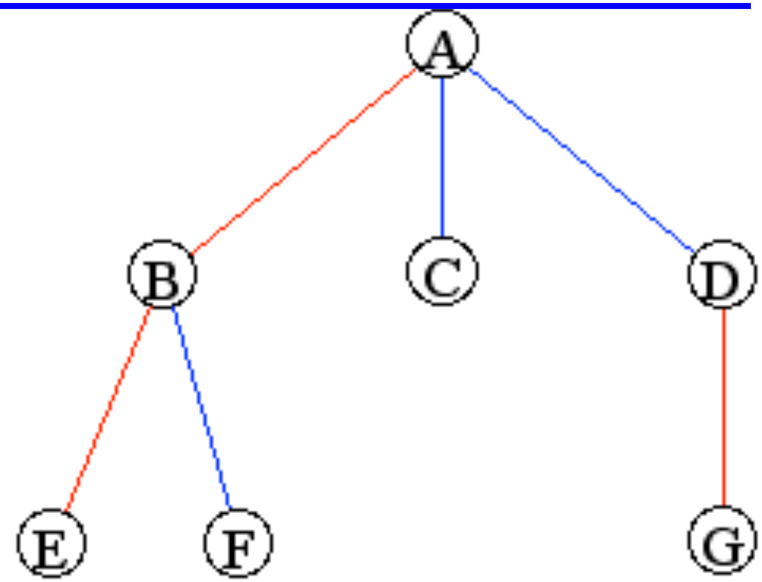
- A node is represented by an object storing
  - Element
  - Parent node
  - Sequence of children nodes
- Node objects implement the Position ADT



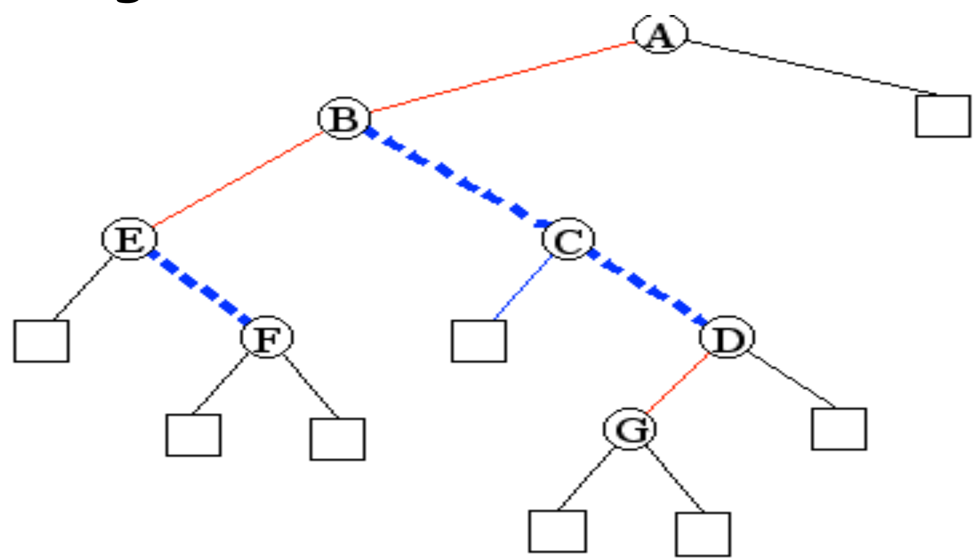


Representing General Trees using Binary Trees

general tree T (not binary)



binary tree T' representing T



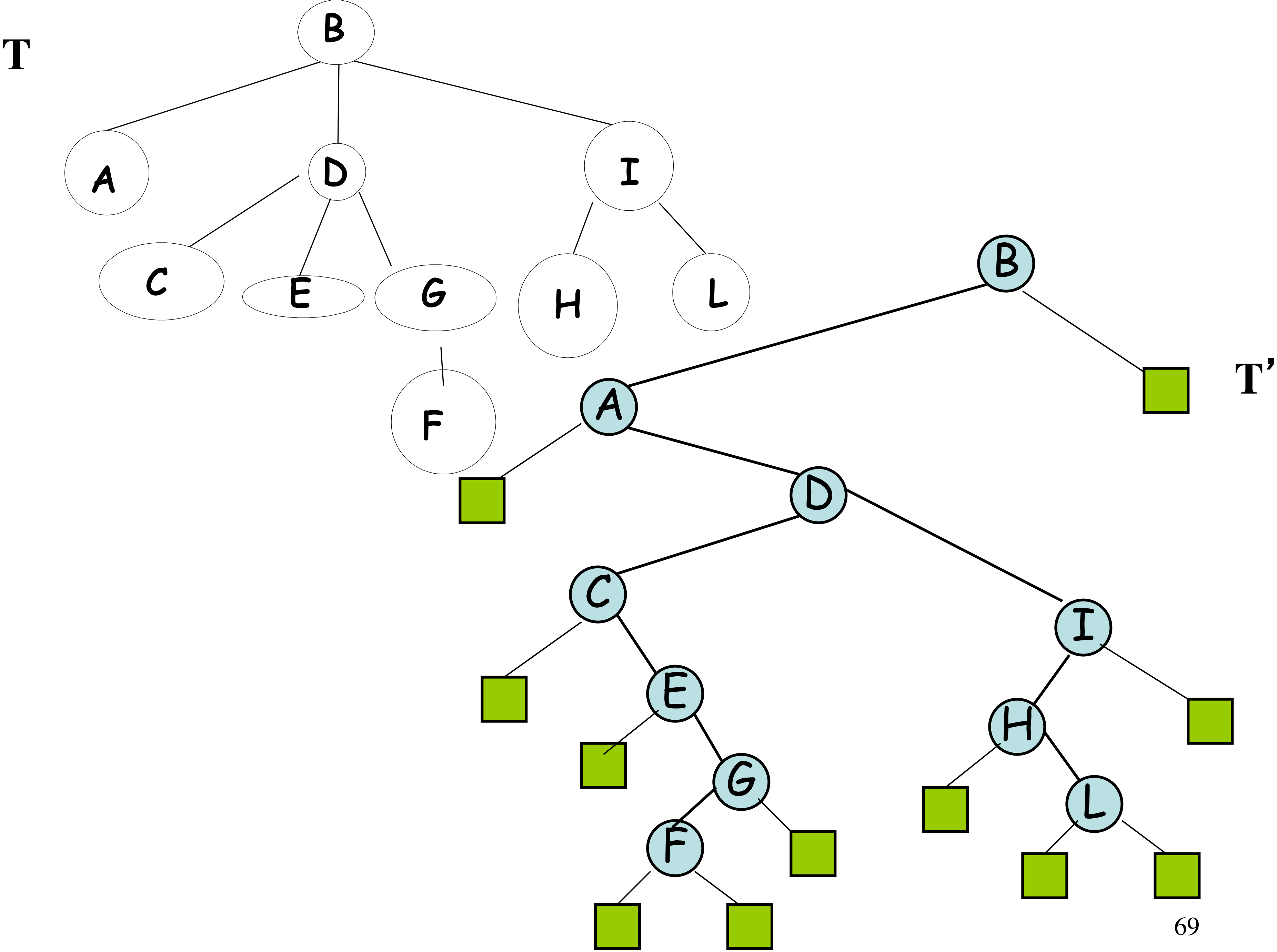
RULES for representing general tree T using binary tree T’

u in T

u’ in T’

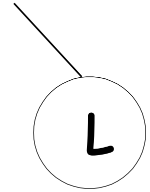
first child of u in T is **left child** of u’ in T’

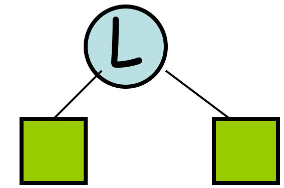
first sibling of u in T is **right child** of u’ in T’


T
B
A
D
I
C
E
G
H
L
F
T'
B
A
D
C
I
E
H
G
L
F

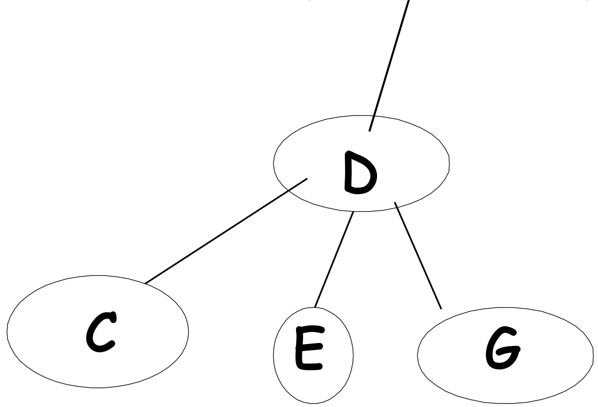
RULE:
to u in T corresponds u’ in T’

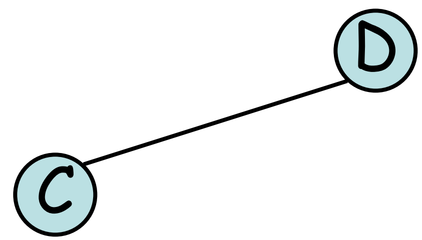
if u is a leaf in T and has no siblings,
then the children of u’ are leaves





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’