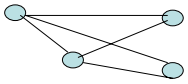
Trees

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

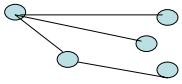
Trees

A graph G = (V,E) consists of an set V of VERTICES and a set E of edges, with E = {(u,v): u,v ∈V, u ≠ v



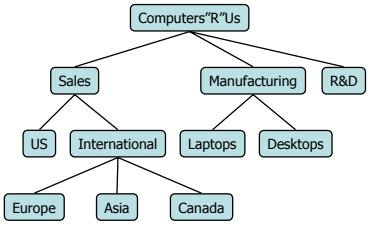
A tree is a connected graph with no cycles.

→ ∃ a path between each pair of vertices.

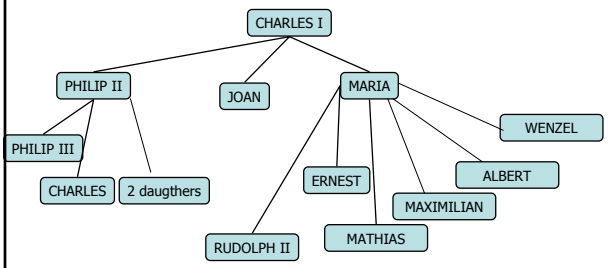


What is a Tree

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



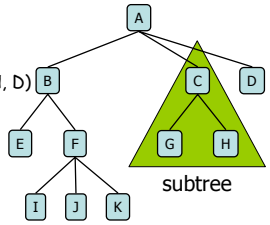
Example: Genealogical Tree



Hasburg Family

Tree Terminology

- Root: node without parent (A)
- Subtree: tree consisting of a node and its descendants
- 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.

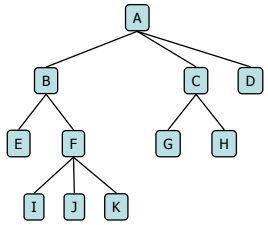


- Descendant of a node: child, grandchild, grand-grandchild, etc.

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)



Methods for Tree ADT

- generic methods
  - size(), isEmpty(), elements(), positions(), replace(p,e)
- query methods
  - isRoot(p), isInternal(p), isExternal(p)
- accessor methods
  - root(), parent(p), children(p)
- update methods
  - application specific

Note: p stands for position which is a node in the tree

Performance

size()	O(1)
isEmpty()	O(1)
elements()	O(n)
positions()	O(n)
replace(p,e)	O(1)
isRoot(p)	O(1)
isInternal(p)	O(1)
isExternal(p)	O(1)
root()	O(1)
parent(p)	O(1)
children(p)	$O(C_p)$, where C_p is the #children of p.

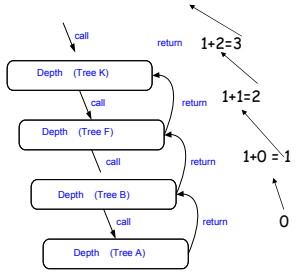
Computing the depth of a node

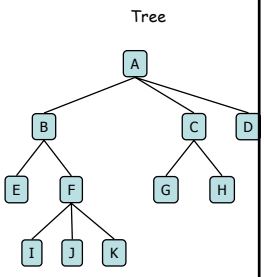
If v is the root the depth is 0
If v is an internal node the depth is 1 + the depth of its parent

```
Algorithm depth(T,v)
if T.isRoot(v) then
   return 0
else
return 1 + depth(T, T.parent(v))
```

Complexity ?

Example: Depth(Tree,K)

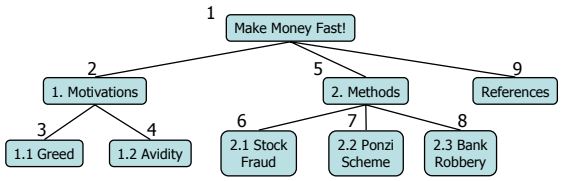




Traversing Trees
Preorder Traversal

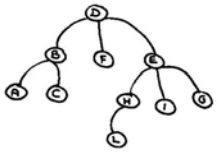
- A traversal visits the nodes of a tree in a systematic manner
- In a preorder traversal, a **node is visited before its descendants**
- Application: print a structured document

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*)



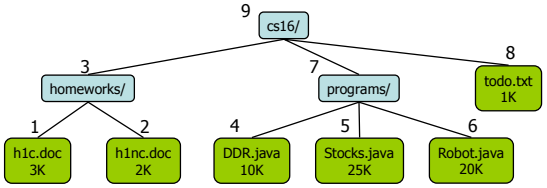
D B A C F E H L I G

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*)

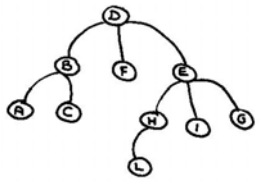


Traversing Trees

Postorder Traversal

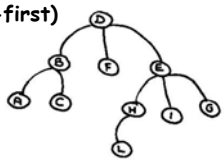
Algorithm postOrder(v)
  for each child w of v **do**
  recursively perform postOrder(w)
    "visit" node v

A C B F L H I G E D



Traversing Trees

Inorder Traversal of a tree (Depth-first)



A B C D F L H E I G

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

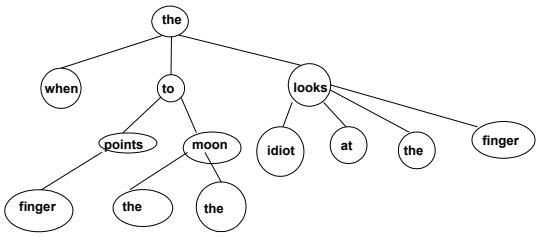
Start: x = root

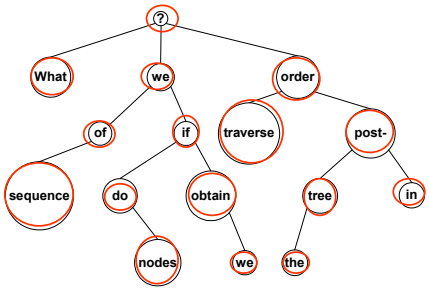
IN-ORDER VISIT

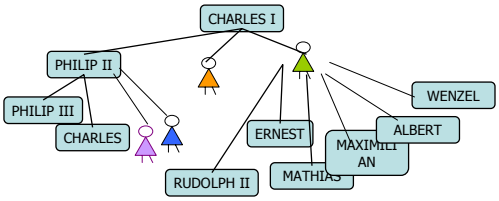
1. Visit the first sub-tree (inorder)
2. Visit the root
3. Visit the second sub-tree (inorder)

⋮ ⋮

d(x)+1. Visit the d(x)th sub-tree (inorder)





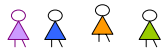


When Charles dies, Philip II becomes King.
If Philip II dies as well ....

Charles I,

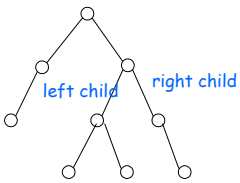
Philip II,

Philip III, Charles,



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

Binary Trees



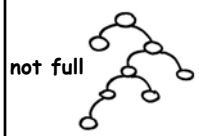
Children are ordered

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

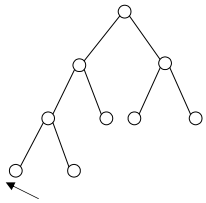


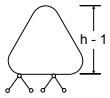
Perfect Binary Trees

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.

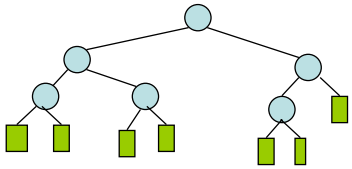




Leaves go at the left

Notation for binary tree.

In the book: children are "completed" with "fake" nodes

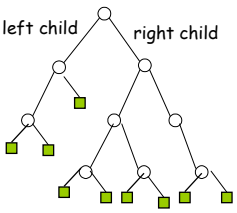


The green squared nodes are the dummy nodes.

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

All trees are FULL

Binary Trees + dummy leaves

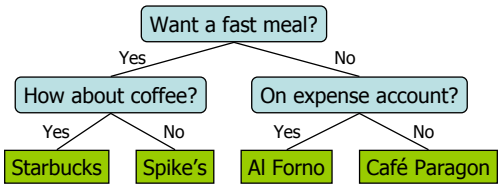


Each internal node has two children

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

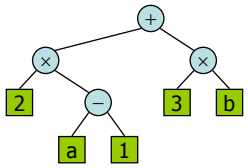


Examples of Binary Trees

Arithmetic Expression Tree

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

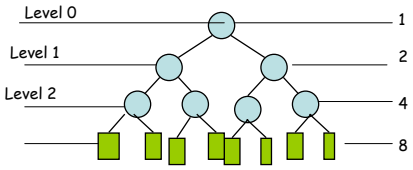
Example: arithmetic expression tree for the expression ((2 × (a - 1)) + ((3 × b)))



Properties of Binary Trees

- Notation
  - ***n*** # of nodes ***e*** # of leaves
  - ***i*** # of internal nodes ***h*** height

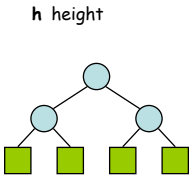
Maximum number of nodes at each level ?

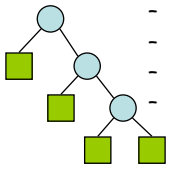


level i ------- 2^i

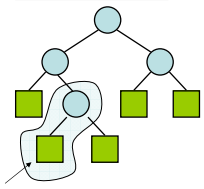
Properties of Full Binary Trees

- Notation
  - **n** number of nodes
  - **e** number of leaves
  - **i** number of internal nodes
  - **h** height
- Some Properties:
  - $e = i + 1$
  - $n = 2e - 1$
  - $h \leq i$
  - $h \leq (n - 1)/2$
  - $e \leq 2^h$
  - $h \geq \log_2 e$
  - $h \geq \log_2 (n + 1) - 1$

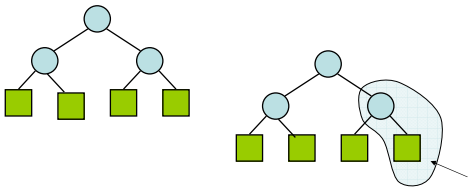




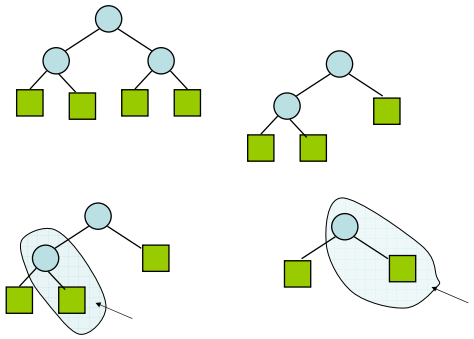
$e = i + 1$



$e = i + 1$



$e = i + 1$



$n = 2e - 1$

$n = i + e$

$e = i + 1$ (just proved)

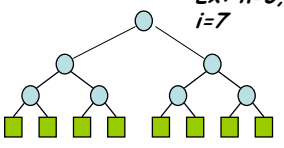
$i = e - 1$

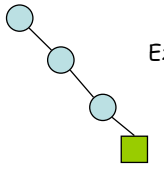
$n = 2e + i$

$h \leq i$ (h = max n. of ancestors)

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

Ex: h=3, i=7

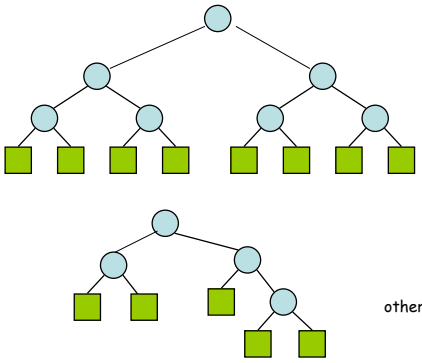




Ex: h=3, i=3

$e \leq 2^h$

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



h = 3

2^3 leaves if all at last level h

otherwise less

Since $e \leq 2^h$

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

$\log_2 e \leq h$

$h \geq \log_2 e$

Summary & some more properties:

In binary trees

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

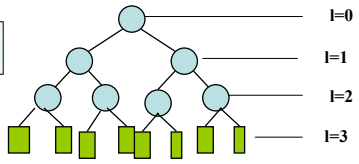
In FULL binary trees

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

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:

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

As a consequence:

In Binary trees: obviously $n \leq 2^{h+1} - 1$

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

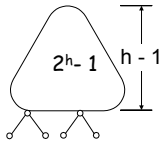
$n+1 \leq 2^{h+1}$

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

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

In Complete Binary Trees …

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



From previous observation: $n \leq 2^{h+1} - 1$

A complete binary tree is a perfect binary tree of height h-1 plus some more leaves …

$n \geq 2^h$

$n \geq 2^h$

It follows that:

Height of a **complete binary tree** with n nodes:

$\lfloor \log n \rfloor$

Methods for Binary Tree ADT

- accessor methods
  - left(p), right(p), hasLeft(p), hasRight(p)
- update methods
  - addRoot(p), insertLeft(p,e), insertRight(p,e)
  - remove(p), attach(p,T_1,T_2)

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.Left(v))
        preOrder (T,T.Right(v))
```

```
Algorithm postOrder(T,v)
    if v is internal:
        postOrder (T,T.Left(v))
        postOrder(T,T.Right(v))
    visit(v)
```

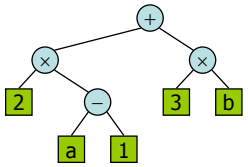

Traversing Binary Trees

Inorder
(Depth-first)

```
Algorithm inOrder(T,v)
    if v is internal:
        inOrder (T,T.Left(v))
    visit(v)
    if v is internal:
        inOrder(T,T.Right(v))
```


Arithmetic Expressions

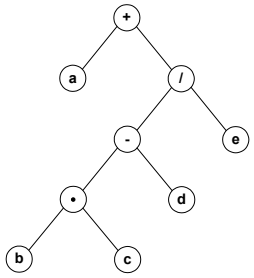
Inorder: a – b
Postorder: a b –
Preorder – a b



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

Postorder:
2 a 1 - × 3 b × +

a + (b • c – d)/e



PRE-ORDER:

+ a / – • b c d e

POST-ORDER:

a b c • d – e / +

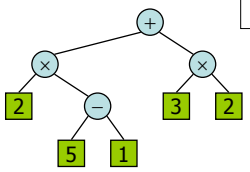
IN-ORDER:

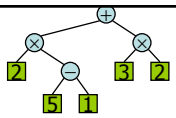
a + b • c – d / e

Evaluate Arithmetic Expressions

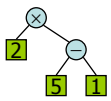
- Specialization of a **postorder** traversal
  - recursive method returning the value of a subtree
  - when visiting an internal node, combine the values of the subtrees

Algorithm *evalExpr*(*v*)
  if *isExternal* (*v*)
    return *v.element* ()
  else
    x ← *evalExpr*(*left*(*v*))
    y ← *evalExpr*(*right*(*v*))
    ◊ ← operator stored at *v*
    return *x* ◊ *y*





Eval



\+

Eval

Eval

2

×

Eval

Eval

3

×

Eval

2

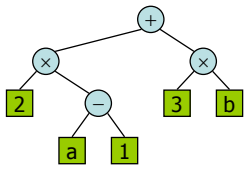
–

5 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 *printExpression*(*v*)
  if *isInternal* (*v*)
    print("(")
    inOrder (*left*(*v*))
  print(*v.element* ())
  if *isInternal* (*v*)
    inOrder (*right*(*v*))
    print (")")



2 × a – 1 + 3 × b

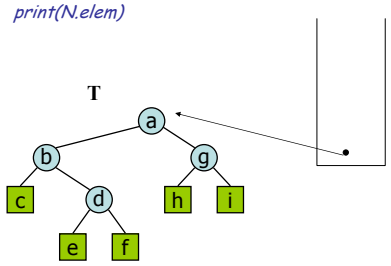
((2 × (a – 1)) + (3 × b))

```
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
                 }
```

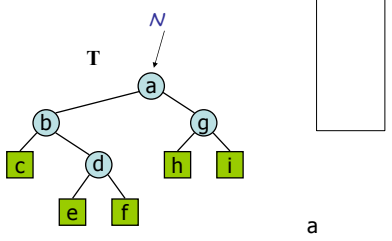

Algorithm *preOrderTraversalwithStack*(T)

```
S.push(T)   // push the reference to T in the empty stack
N = S.pop()
 print(N.elem)
```



Algorithm *preOrderTraversalwithStack*(T)

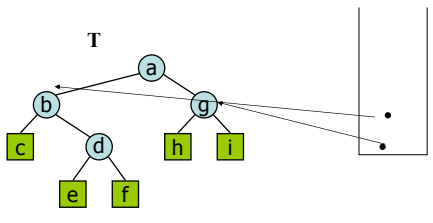
```
S.push(T)   // push the reference to T in the empty stack
N = S.pop()
 print(N.elem)
```



a

Algorithm *preOrderTraversalwithStack*(T)

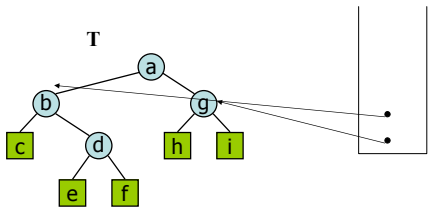
```
S.push(N.rightChild) // push the reference to
                                the right child
S.push(N.leftChild) // push the reference to
                                the left child
```



a

Algorithm *preOrderTraversalwithStack*(T)

N = S.pop()

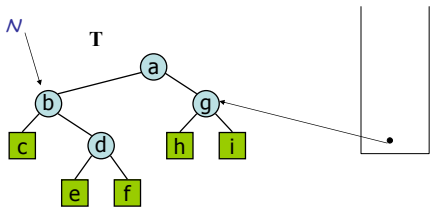


a

Algorithm *preOrderTraversalwithStack*(T)

N = S.pop()

print(N.elem)

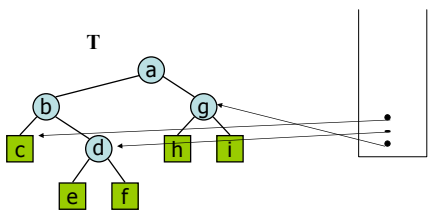


a b

Algorithm *preOrderTraversalwithStack*(T)

S.push(N.rightChild)

S.push(N.leftChild)

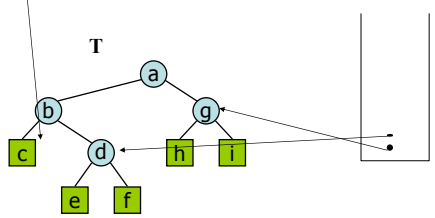


a b

Algorithm *preOrderTraversalwithStack*(T)

N = S.pop()

print(N.elem)

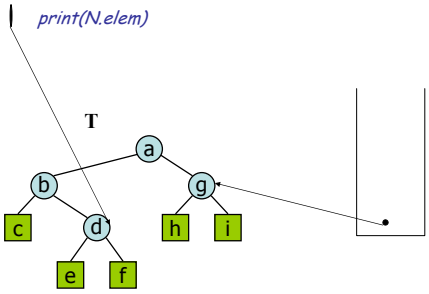


a b c

Algorithm *preOrderTraversalwithStack*(T)

N = S.pop()

print(N.elem)

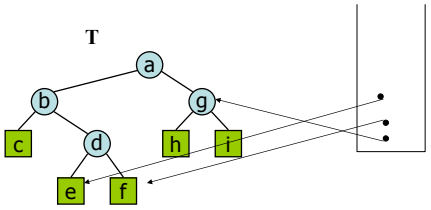


a b c d

Algorithm *preOrderTraversalwithStack*(T)

S.push(N.rightChild)

S.push(N.leftChild)

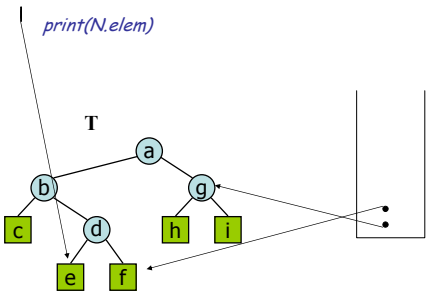


a b c d

Algorithm *preOrderTraversalwithStack*(T)

N = S.pop()

print(N.elem)

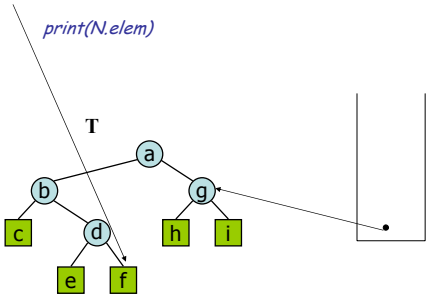


a b c d e

Algorithm *preOrderTraversalwithStack*(T)

N = S.pop()

print(N.elem)

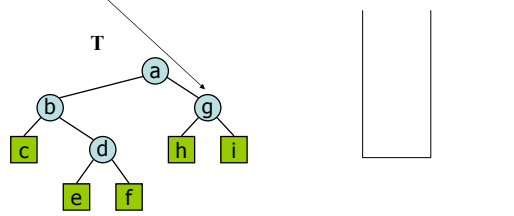


a b c d e f

Algorithm *preOrderTraversalwithStack*(T)

N = S.pop()

print(N.elem)

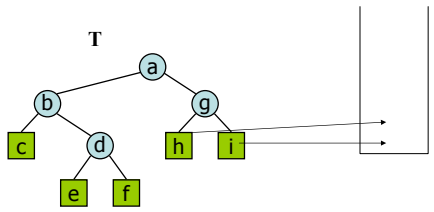


a b c d e f g

Algorithm *preOrderTraversalwithStack*(T)

S.push(N.rightChild)

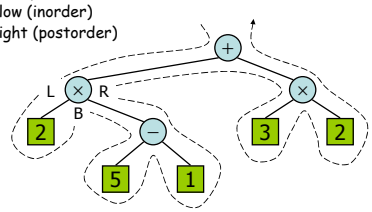
S.push(N.leftChild)



a b c d e f g

Euler Tour Traversal

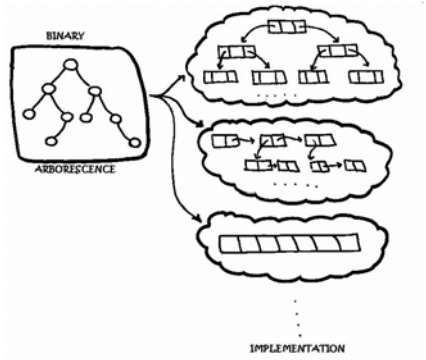
- Generic traversal of a binary tree
- Includes a special cases the preorder, postorder and inorder traversals
- Walk around the tree and visit each node three times:
  - on the left (preorder)
  - from below (inorder)
  - on the right (postorder)



Algorithm eulerTour(T,v)

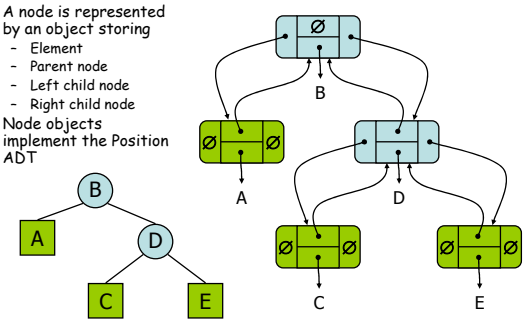
```
visit v                        (from the left)
if v is internal:
    eulerTour (T,T.Left(v))
visit v                        (from below)
if v is internal:
    eulerTour(T,T.Right(v))
visit v                        (from the right)
```

Implementations of Binary trees….

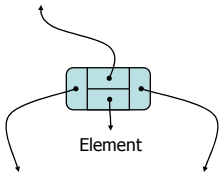


Implementing Binary Trees with a Linked Structure

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



BTNode
Object Element
BTNode left, right, parent



left(v): return v.left

right(v): return v.right

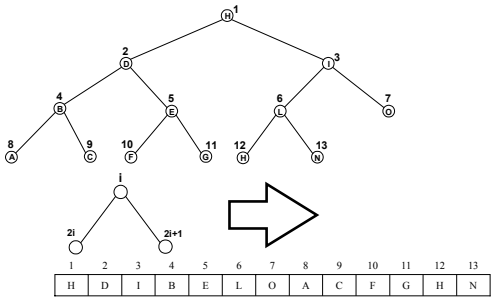
sibling(v):
p ← parent(v)
q ← left(p)
if (v = q) return right(p)
else return q

replace(v, obj)
temp ← v.element
v.element ← obj
return temp

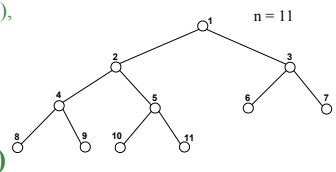
Performance of LinkedBinary Implementation

- size, isEmpty, replace, root, parent, children, left, right, sibling, hasLeft, hasRight, isInternal, isExternal, isRoot, insertLeft, insertRight, attach, remove
  → All take O(1) time.
- elements, positions
  → All take O(n) time

Implementing Complete Binary Trees with Vectors (Array-based)



left(p), right(p), sibling(p),
replace(p,e), isRoot(p),
isInternal(p),
isExternal(p)

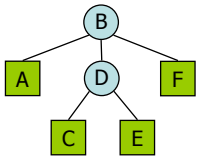


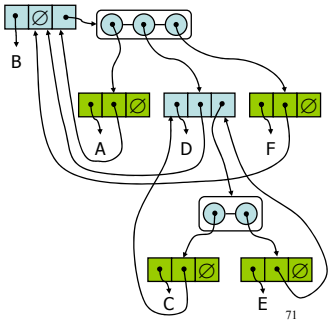
They all have complexity O(1)

Left child of T[i]	T[2i]	if	$2i \le n$
Right child of T[i]	T[2i+1]	if	$2i + 1 \le n$
Parent of T[i]	T[i div 2]	if	$i > 1$
The Root	T[1]	if	$T \ne 0$
Leaf? T[i]	TRUE	if	$2i > n$

Implementing General Trees with a Linked Structure

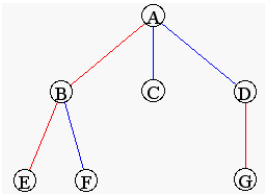
- A node is represented by an object storing
  - Element
  - Parent node
  - Sequence of children nodes
- Node objects implement the Position ADT



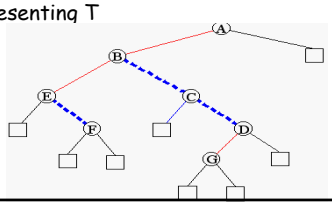


Representing General Trees

tree T



binary tree T' representing T

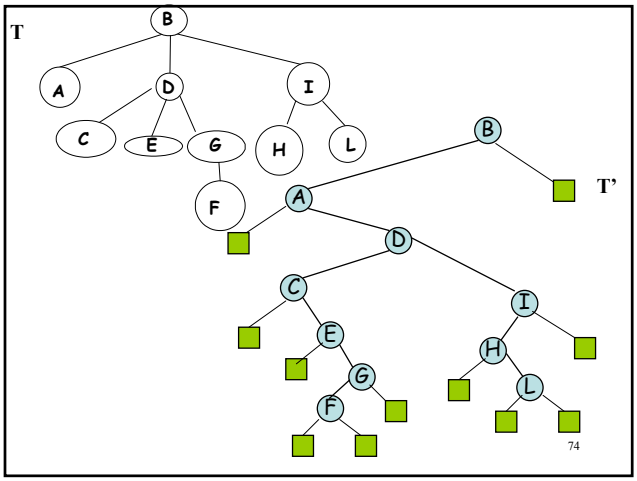


RULES

u in T u' in T'

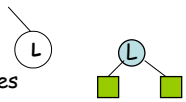
first child of u in T is left child of u' in T'

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

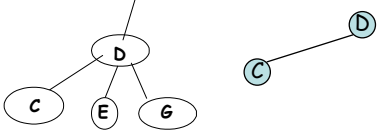


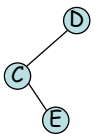
RULE:
to u in T corresponds u' in T'

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'