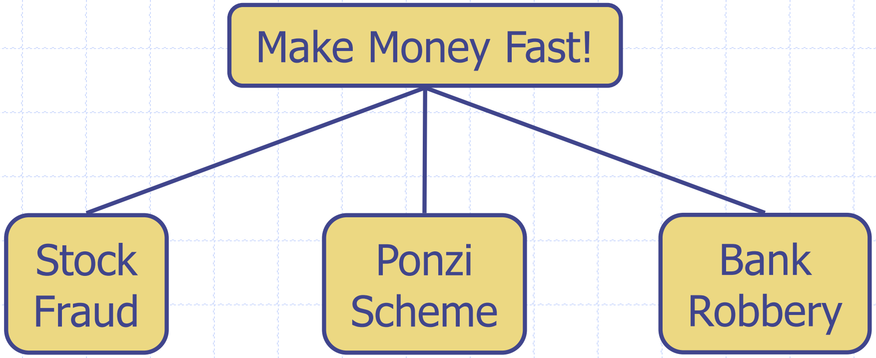
Lecture 6: Tree Structures

(Chapter 7 from the book)

Agenda

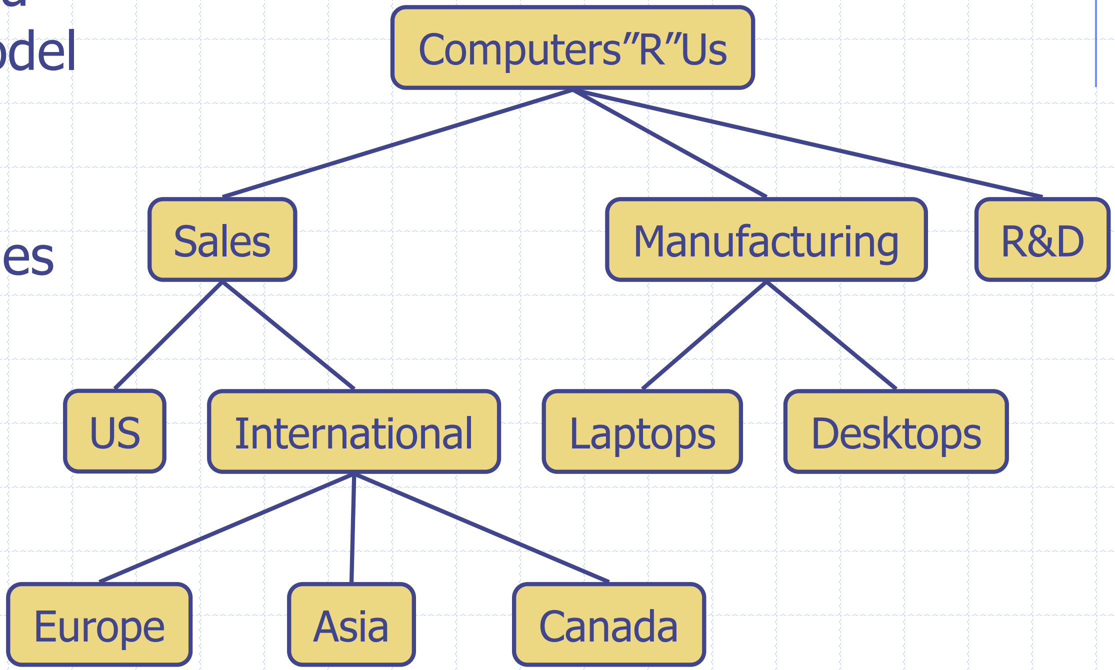
- General Trees
- Tree Traversal Algorithms
- Binary Trees
- Binary Search Tree

General Trees



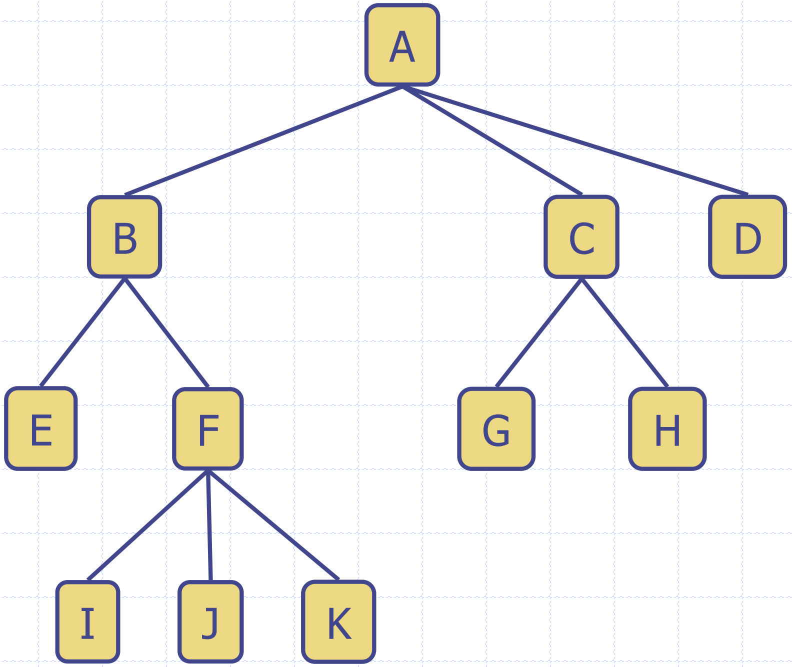
What is a Tree

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



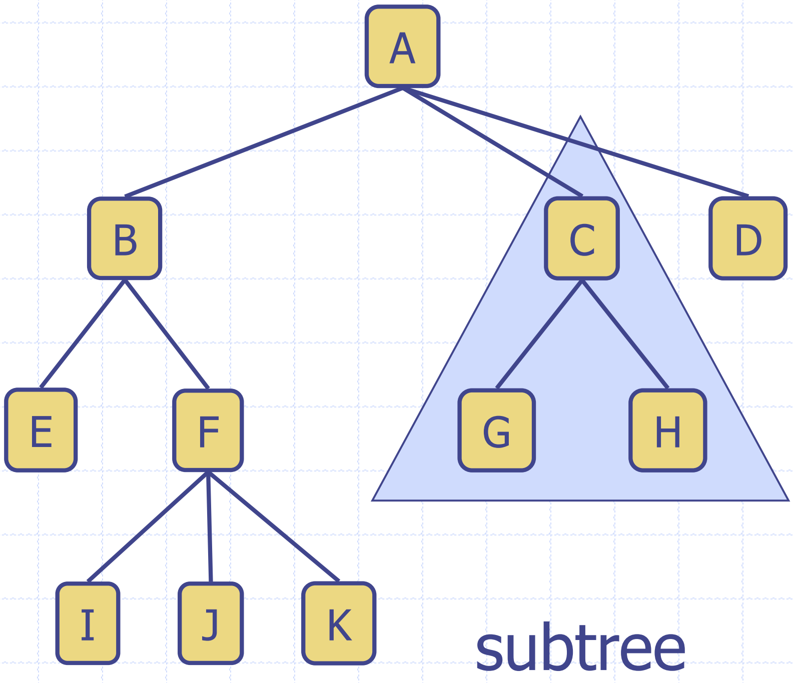
Tree Terminology

- Each node (except root – top node) has parent and zero or more children
- 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.
- Siblings: children of the same parent (e.g. I, J, K)



Tree Terminology

- Depth of a node: number of ancestors (e.g. for F it is 2)
- Height of a tree: maximum depth of any node (3)
- Subtree: tree consisting of a node and its descendants



Formal Tree Definition

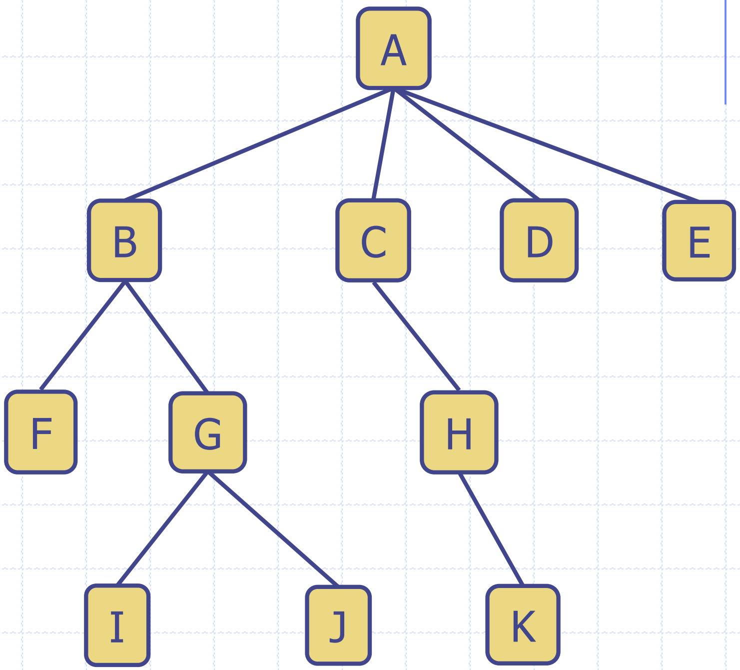
- Tree *T* is a set of nodes storing elements such that the nodes have a parent-child relationship, that satisfies the following properties:
  - If *T* is nonempty, it has a special node, called the root of *T*, that has no parent.
  - Each node *v* of *T* different from the root has a unique parent node *w*. Every node with parent *w* is a child of *w*.
- Note that tree can be empty!

Exercise 1 – Tree

- Draw a tree that has 11 nodes.
  6 of those nodes are leaves.
- For each node:
  - name its ancestors and descendants,
  - Name its siblings,
  - say if the node is internal or external,
  - give its depth.
- Evaluate the height of the tree

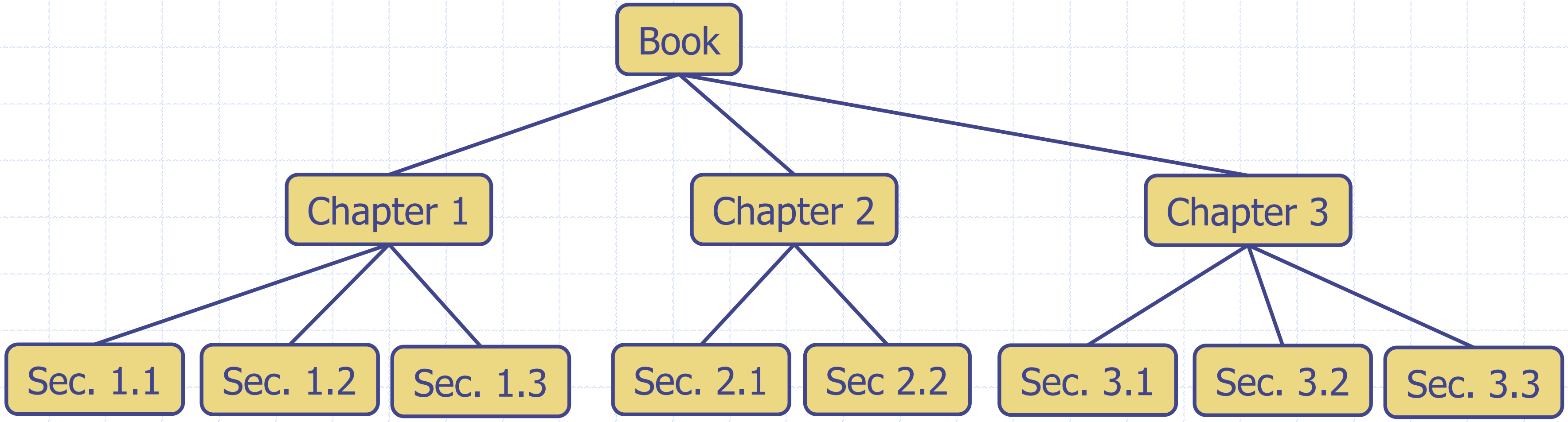
Exercise 1 – Tree – Answer

- Consider node G:
  - its ancestors – A, B
  - its descendants – I, J
  - its siblings – F
  - say if the node is internal or external – internal
  - give its depth – 2
- Evaluate the height of the tree – 3



Ordered Tree

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



Tree ADT

- Tree ADT stores elements at positions, which, as with positions in a list, are defined relative to neighbouring positions.
- As with a list position, a position object for a tree supports the method
  - element() – return the object stored at this position
- The positions in a tree are its nodes, and neighbouring positions satisfy the parent-child relationships that define a valid tree.
- Accessor methods:
  - position root() – return the tree's root; an error occurs if the tree is empty;
  - position parent(v) – return the parent of v, an error occurs if v is the root;
  - Iterable children(v) – return an iterable collection containing the children of node v.

Tree ADT (cont.)

- Query methods:
  - boolean isInternal(v) – test whether node v is internal;
  - boolean isExternal(v) – test whether node v is external;
  - boolean isRoot(v) – test whether node v is a root.
- These methods make programming with trees easier and more readable as we can use them in the conditionals of ***if*** statements and ***while*** loops

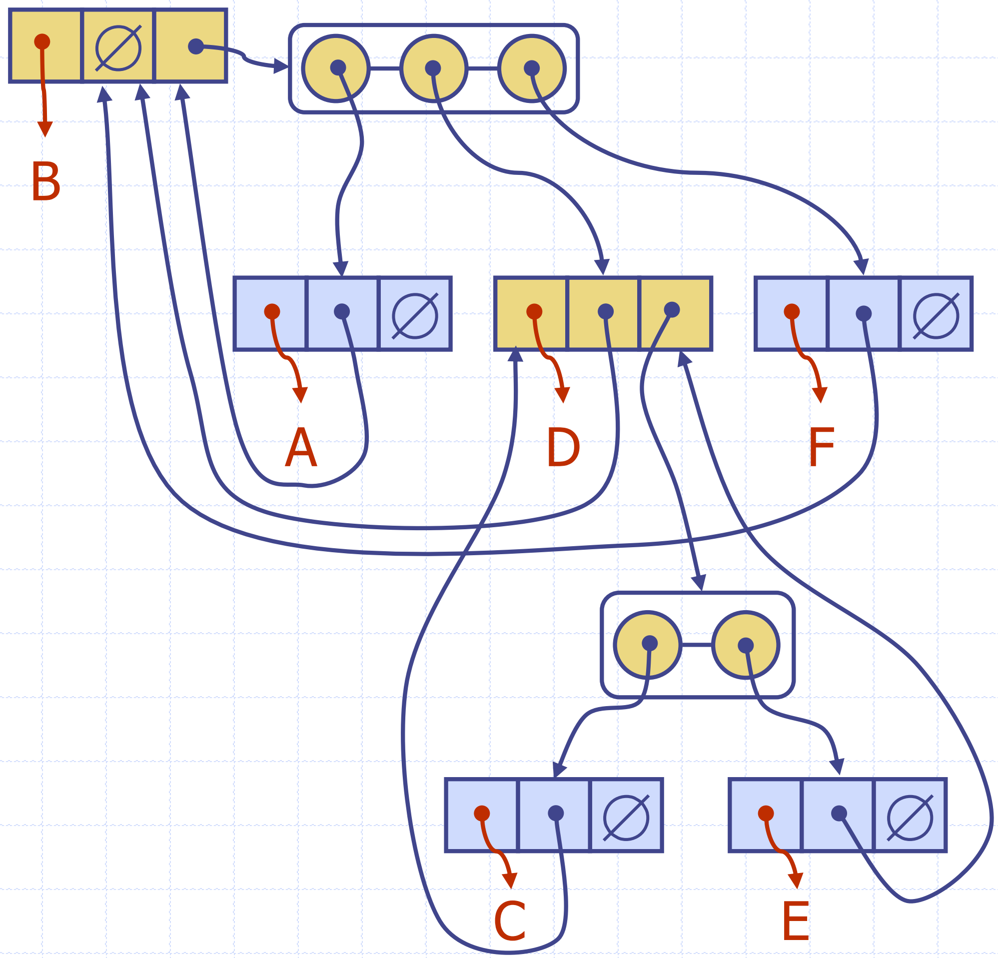
Tree ADT (cont.)

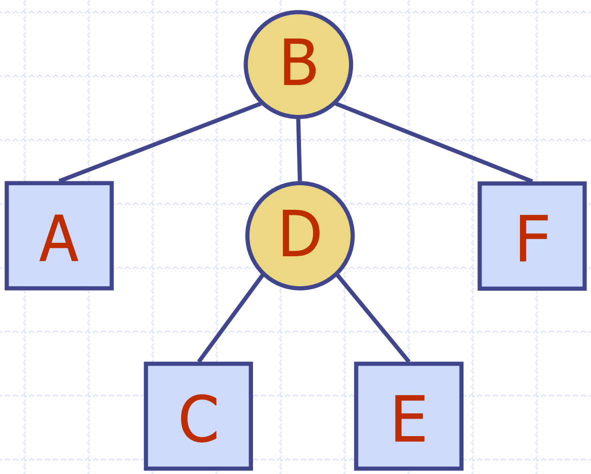
- Generic methods:
  - integer size() – return the number of nodes in the tree;
  - boolean isEmpty() – test whether the tree has any nodes or not;
  - Iterator iterator() – return an iterator of all the elements stored at nodes of the tree;
  - Iterable positions() – return an iterable collection of all the nodes of the tree;
  - element replace (*v*, *e*) – replace with *e* and return the element stored at node *v*.

© 2010 Goodrich, Tamassia

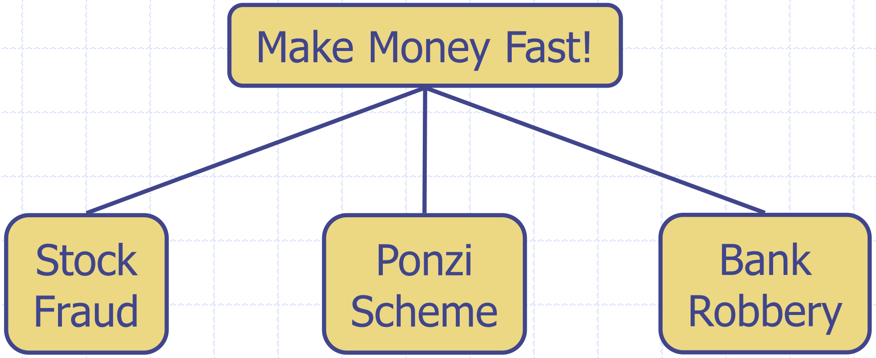
Linked Structure for Trees

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





Tree Traversal Algorithms

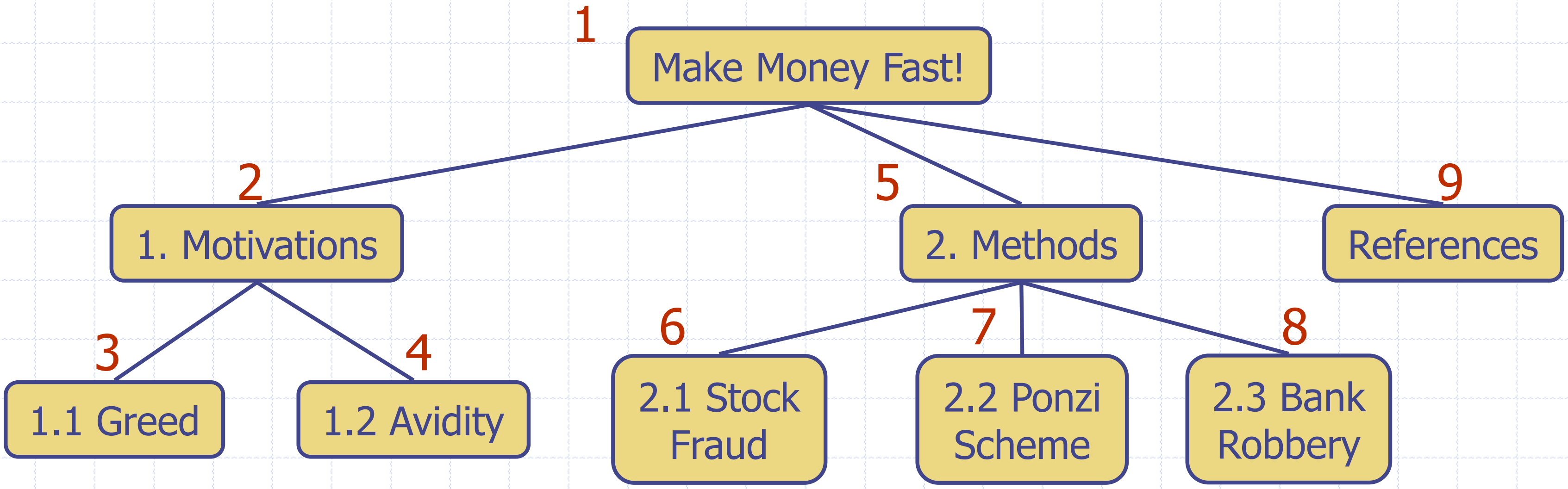


Traversal of a Tree

- A traversal of a tree *T* is a systematic way of accessing, or “visiting”, all the nodes of *T*.
- Traversal schemes:
  - Preorder traversal
  - Postorder traversal

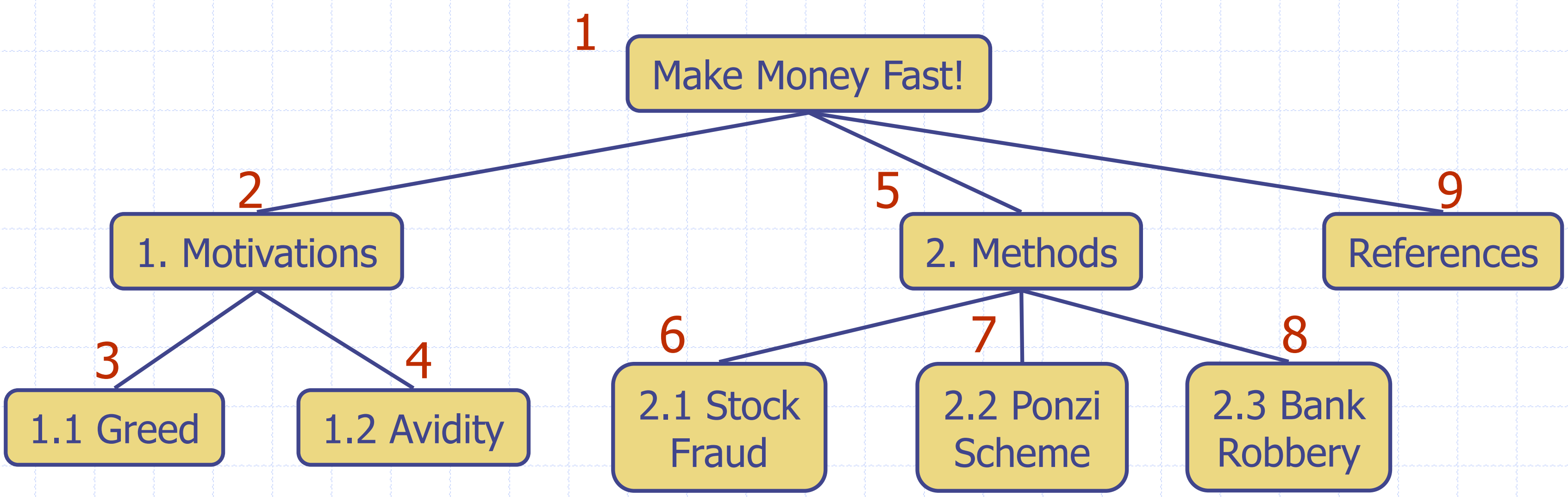
Preorder Traversal

- In a preorder traversal, a node is visited before its descendants.
- Parents always come before their children.
- Note that if the tree is ordered, then the subtrees are traversed according to the order of children.
- Application: print a structured document.
- Running time for the tree with *n* nodes: O(*n*)



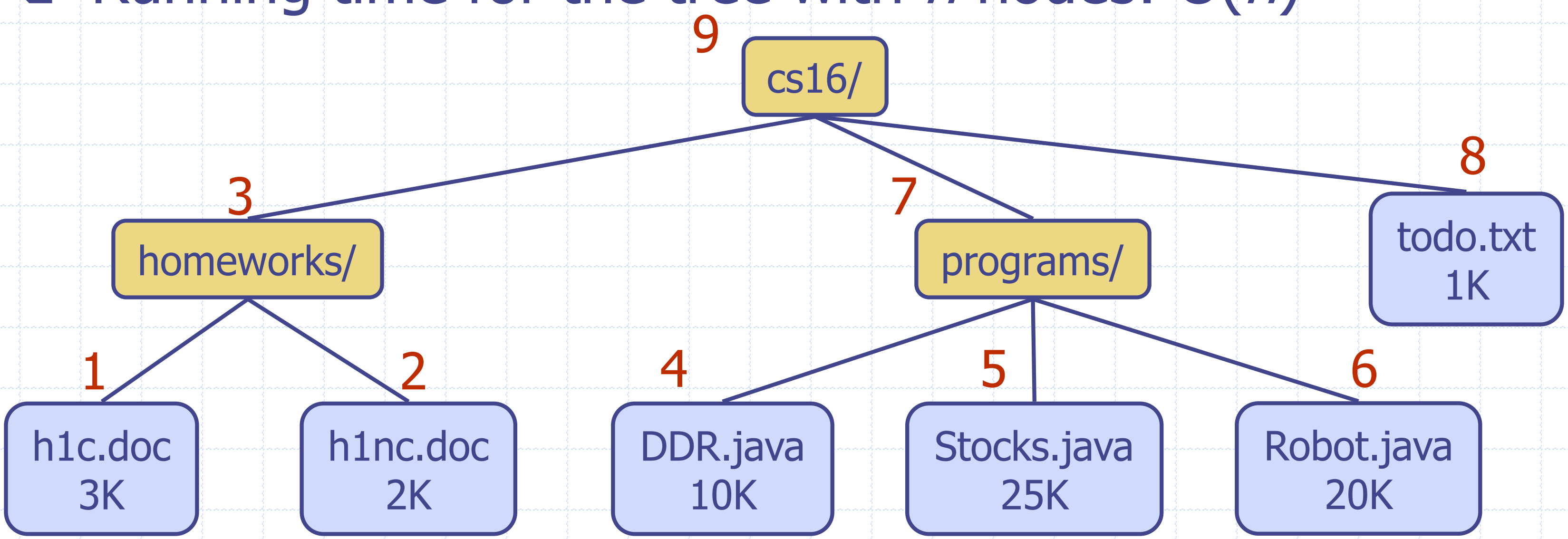
Preorder Traversal

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



Postorder Traversal

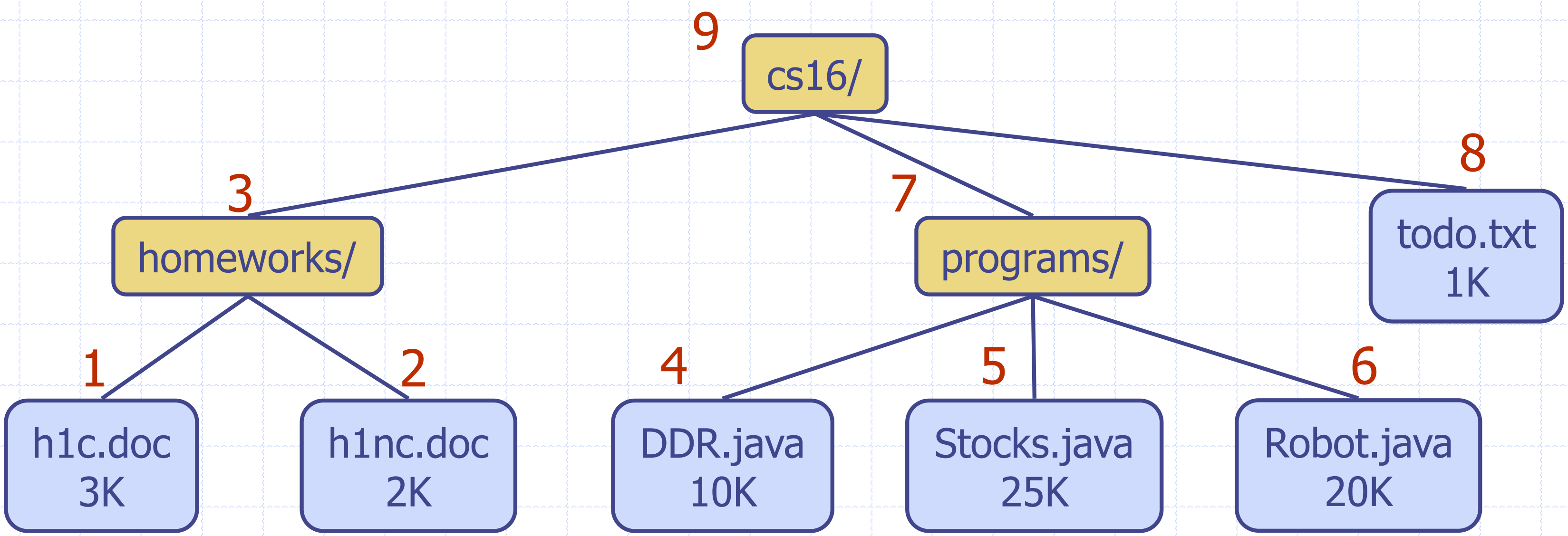
- In a postorder traversal, a node is visited after its descendants
- Application: compute space used by files in a directory and its subdirectories
- Running time for the tree with *n* nodes: O(*n*)



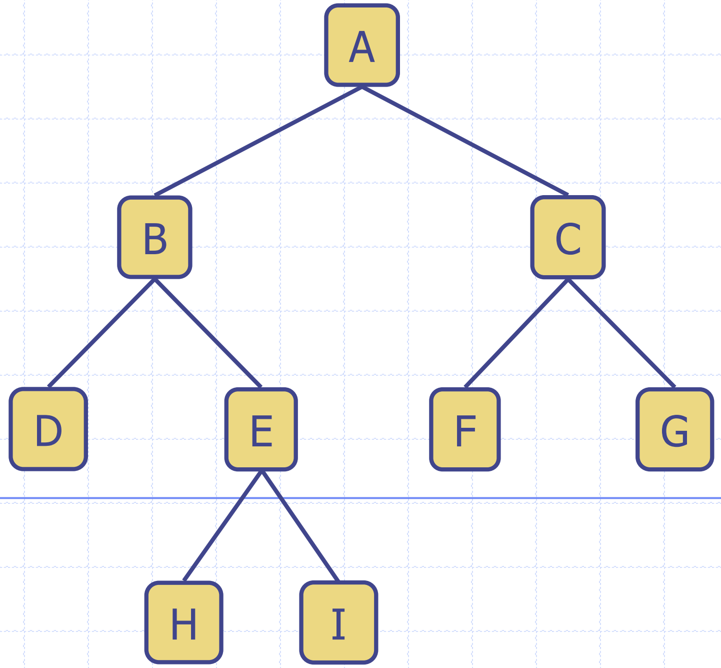
© 2010 Goodrich, Tamassia

Postorder Traversal

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

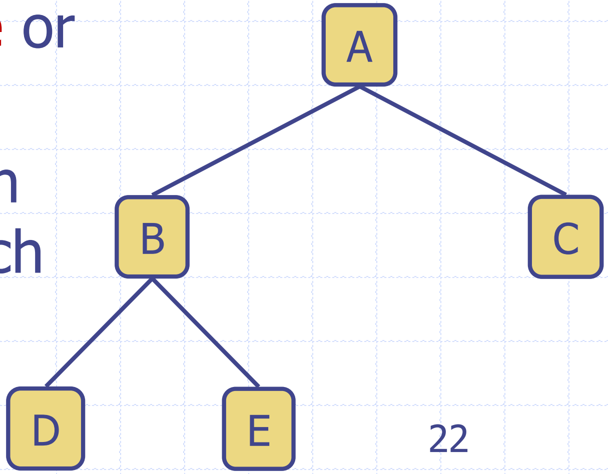


Binary Trees



Binary Trees – Definition

- A binary tree is an ordered tree with the following properties:
  - Each internal node has at most two children
  - Each child node is labeled as being either a left child or a right child
  - A left child precedes a right child in the ordering of children of a node
- The subtree rooted at a left or right child of an internal node *v* is called a left subtree or right subtree, respectively, of *v*.
- A binary tree is proper (a.k.a. full) if each node has either zero or two children. Each internal node has exactly two children.

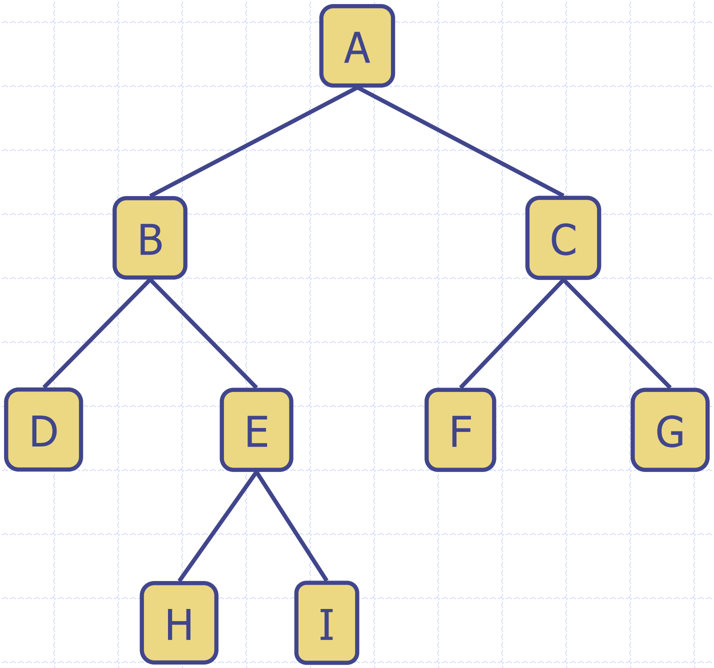


Binary Trees – Recursive Def.

- A binary tree *T* is either empty or consists of
  - A node *r*, called the root of *T* and storing the element
  - A binary tree, called the left subtree of *T*
  - A binary tree, called the right subtree of *T*.

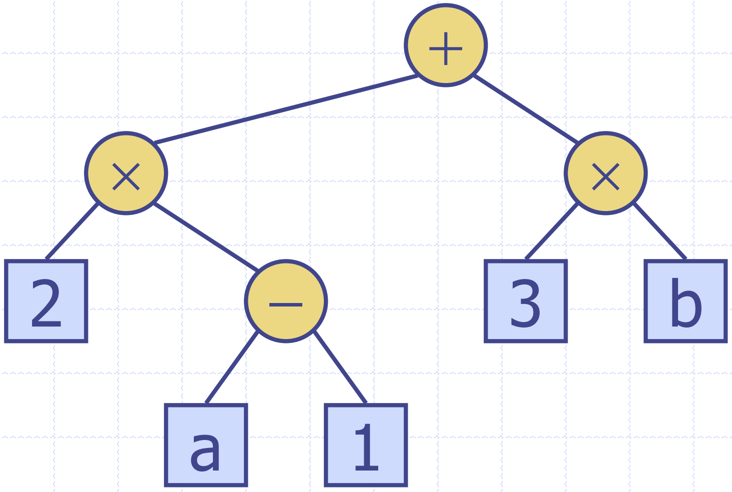
Binary Trees – Applications

- Applications:
  - arithmetic expressions
  - decision processes
  - searching



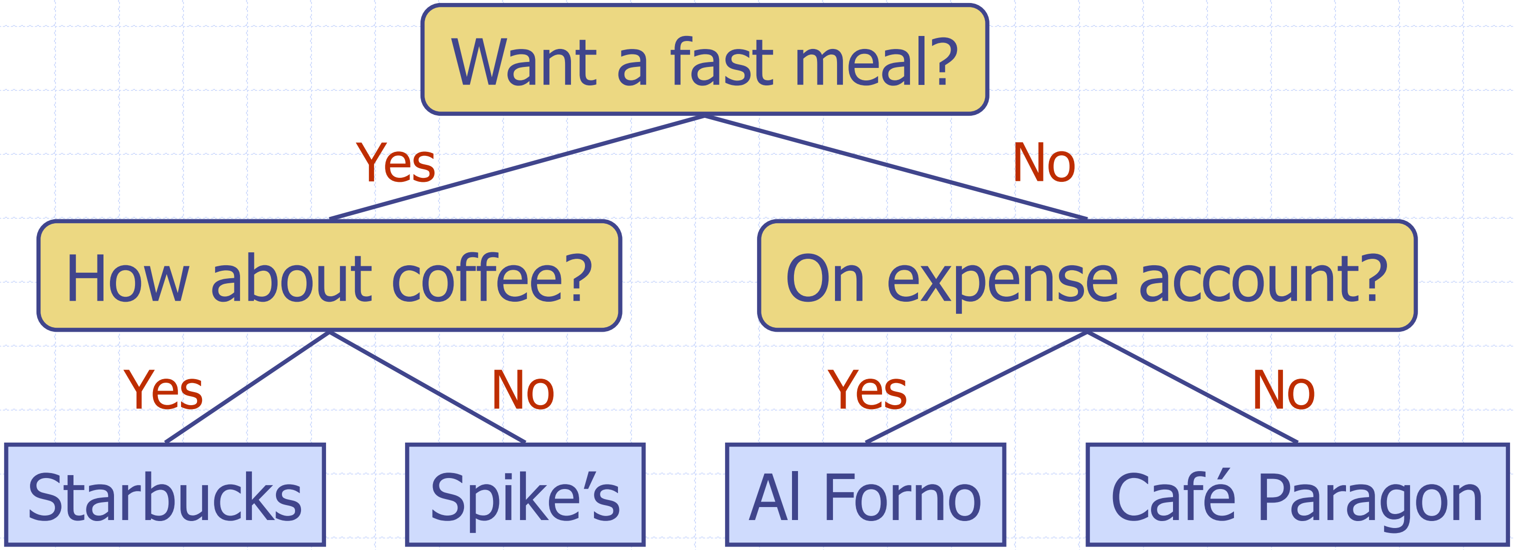
Arithmetic Expression Tree

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



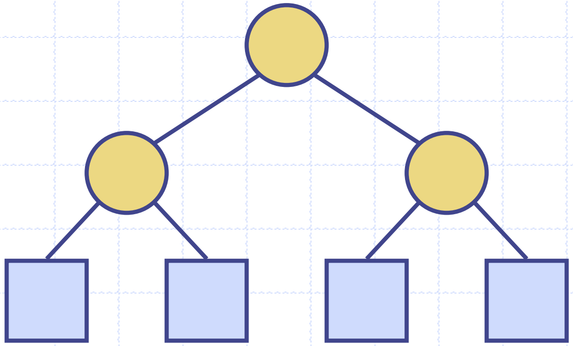
Decision Tree

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



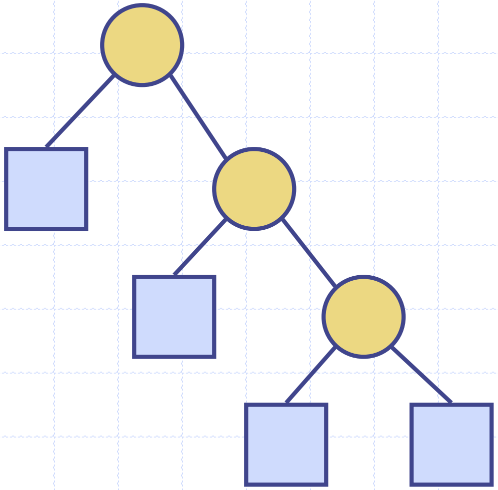
Properties of Binary Trees

- Notation
  - n – number of nodes
  - n_e – number of external nodes
  - n_i – number of internal nodes
  - h – height



- Properties:
  - a) $h+1 \le n \le 2^{h+1} - 1$
  - b) $1 \le n_e \le 2^h$
  - c) $h \le n_i \le 2^h - 1$
  - d) $\log_2 (n + 1) - 1 \le h \le n - 1$

  For $n \ge 1$



Properties of Binary Trees

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

b) $1 \leq n_e \leq 2^h$

c) $h \leq n_i \leq 2^h - 1$

d) $\log_2 (n + 1) - 1 \leq h \leq n - 1$

For $n \geq 1$

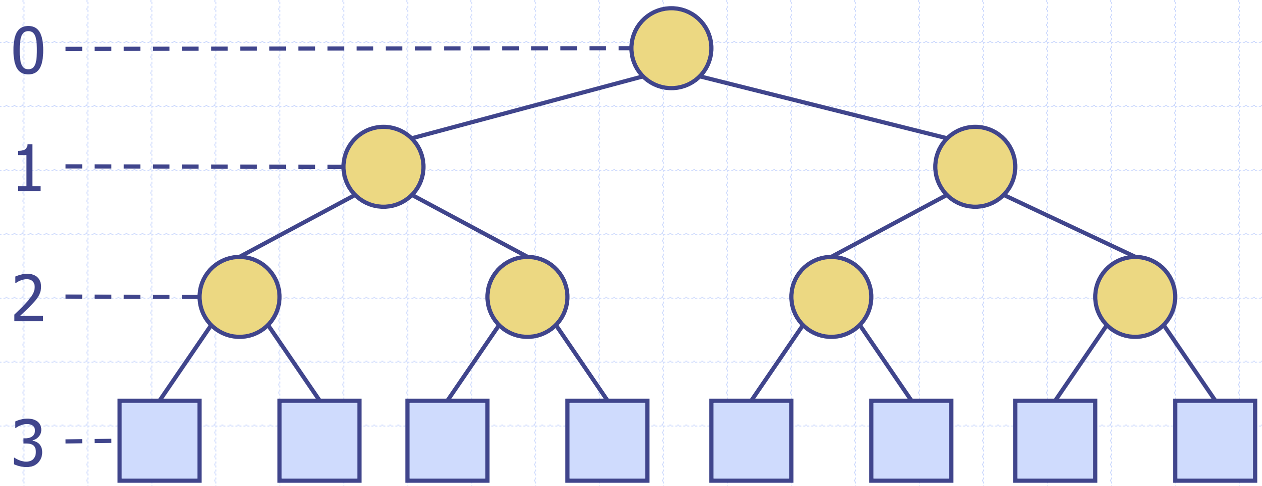
a) $4 \leq \mathbf{15} \leq \mathbf{15}$

b) $1 \leq \mathbf{8} \leq \mathbf{8}$

c) $3 \leq \mathbf{7} \leq \mathbf{7}$

d) $\mathbf{3} \leq \mathbf{3} \leq 14$

Height – h



$h=3$

$n_e=8$

$n_i=7$

$n=15$

Properties of Binary Trees

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

b) $1 \leq n_e \leq 2^h$

c) $h \leq n_i \leq 2^h - 1$

d) $\log_2 (n + 1) - 1 \leq h \leq n - 1$

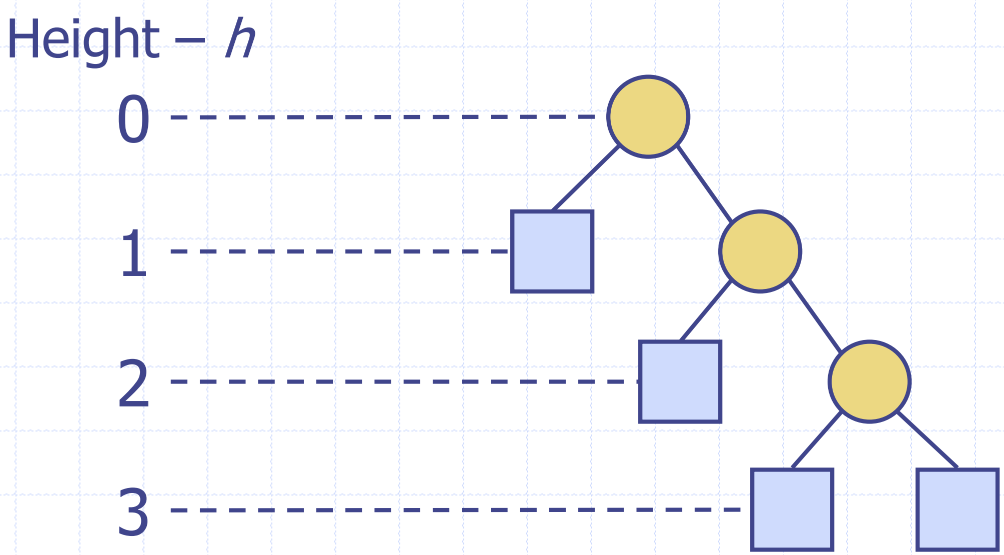
For $n \geq 1$

a) $4 \leq 7 \leq 15$

b) $1 \leq 4 \leq 8$

c) $3 \leq 3 \leq 7$

d) $2 \leq 3 \leq 6$



h=3

n_e=4

n_i=3

n=7

Properties of Binary Trees

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

b) $1 \le n_e \le 2^h$

c) $h \le n_i \le 2^h - 1$

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

For $n \ge 1$

a) $1 \le 1 \le 1$

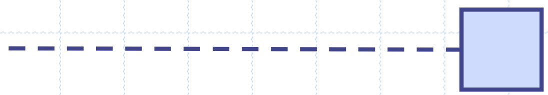
b) $1 \le 1 \le 1$

c) $0 \le 0 \le 0$

d) $0 \le 0 \le 0$

Height – *h*

0



$h=0$

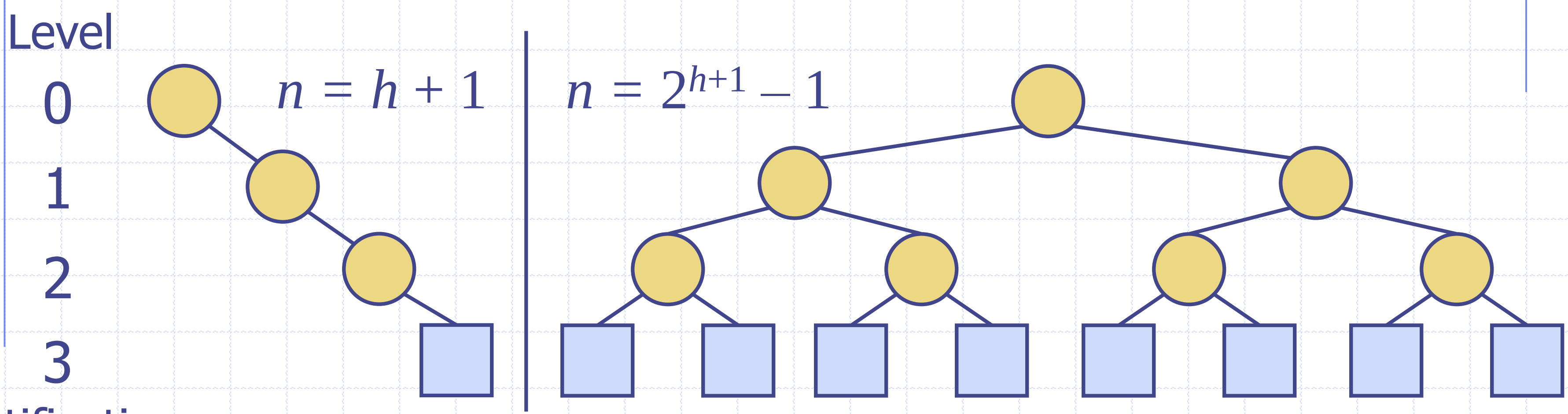
$n_e=1$

$n_i=0$

$n=1$

Properties of Binary Trees

a) Let $n \geq 1$ be the number of elements in a binary tree of height $h \geq 0$ then: $h+1 \leq n \leq 2^{h+1} - 1$



Justification

- We must have at least one element at each level $0, 1, \ldots, h$, so $n \geq h+1$
- At each level i, for i=0, 1, …, h there are at most 2^i elements at this level, so we have:

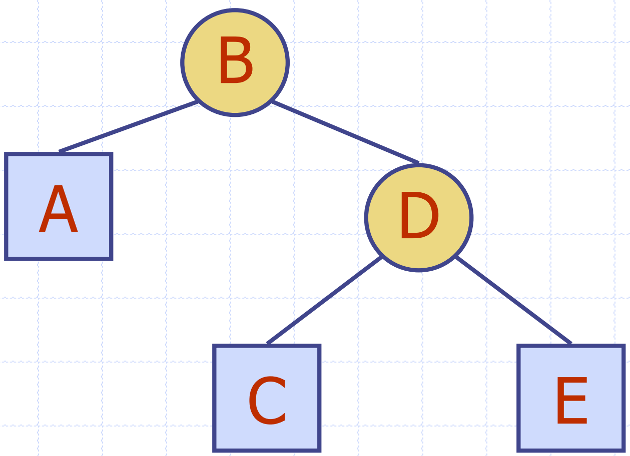
$$n \leq \sum_{i=0}^{h} 2^i = 1 + 2 + 4 + \cdots + 2^h = 1 \cdot \frac{1 - 2^{h+1}}{1 - 2} = 2^{h+1} - 1$$

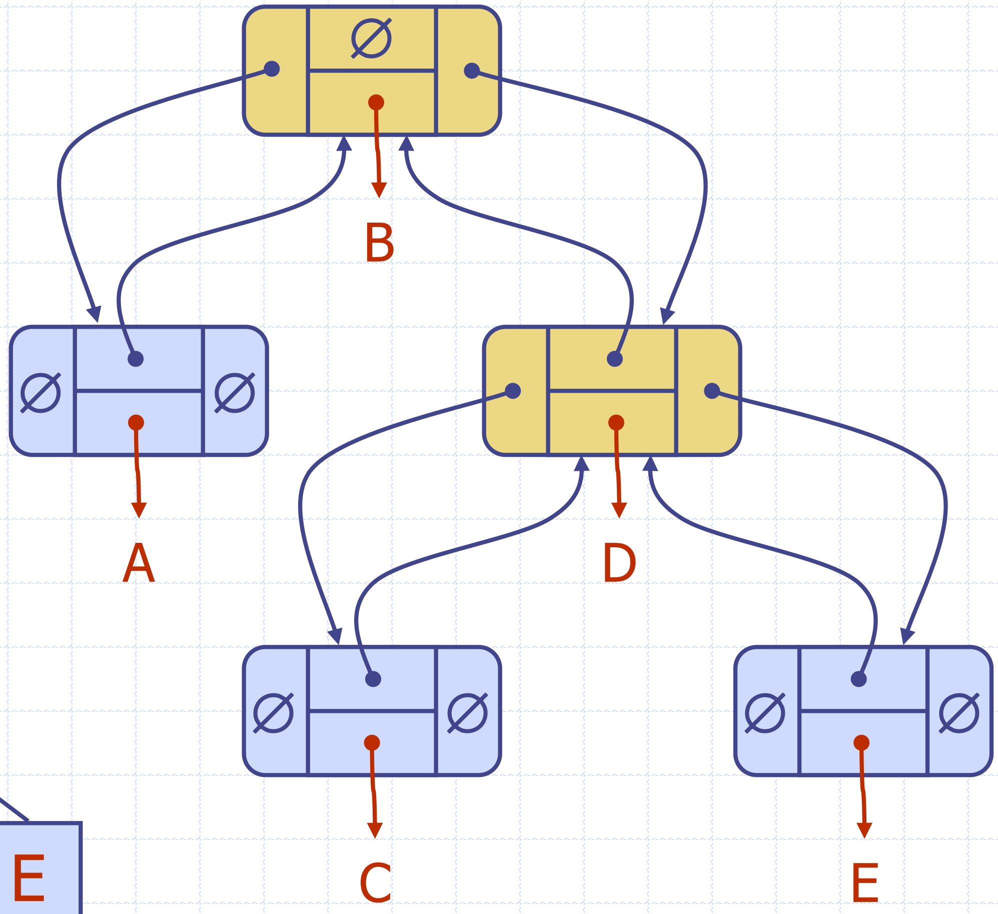
BinaryTree ADT

- The BinaryTree ADT extends the Tree ADT, i.e., it inherits all the methods of the Tree ADT
- Additional methods:
  - position left(*v*) – return the left child of *v*;
  - position right(*v*) – return the right child of *v*;
  - boolean hasLeft(*v*) – test whether *v* has a left child
  - boolean hasRight(*v*) – test whether *v* has a right child

Linked Structure for Binary Trees

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

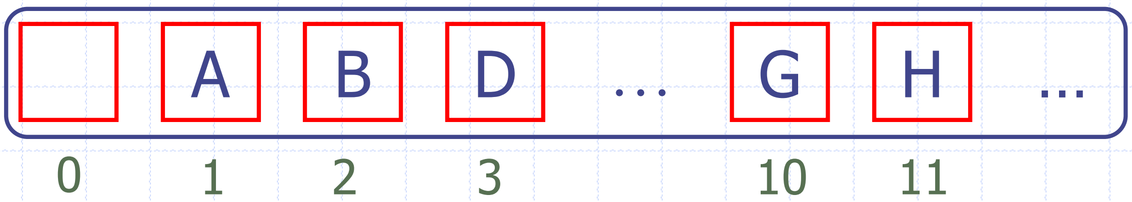


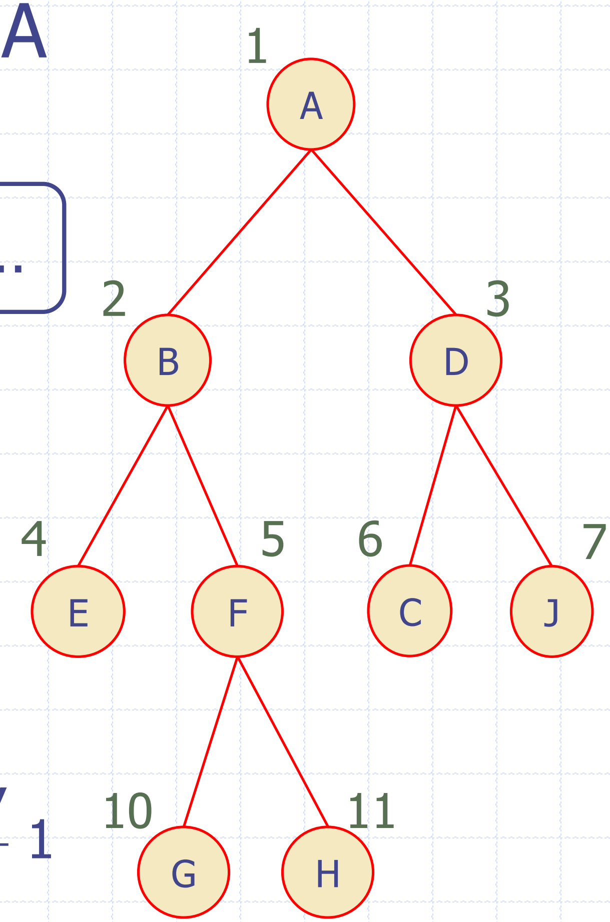


© 2010 Goodrich, Tamassia

Array-Based Representation of Binary Trees

- Nodes are stored in an array A





- Node v is stored at A[rank(v)]
  - rank(root) = 1
  - if node is the left child of parent(node), rank(node) = 2 · rank(parent(node))
  - if node is the right child of parent(node), rank(node) = 2· rank(parent(node)) + 1

Preorder Traversal

- The preorder traversal for general trees can be applied to any binary tree.
- However, the algorithm can be simplified:

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

⇨

Algorithm *binaryPreorder*(*T*, *v*)
  visit(v)
  if *hasLeft* (*v*) **then**
    binaryPreorder(*T*, *left*(*v*))
  if *hasRight* (*v*) **then**
    binaryPreorder(*T*, *right*(*v*))c

Postorder Traversal

- The postorder traversal for general trees can be applied to any binary tree.
- However, the algorithm can be simplified:

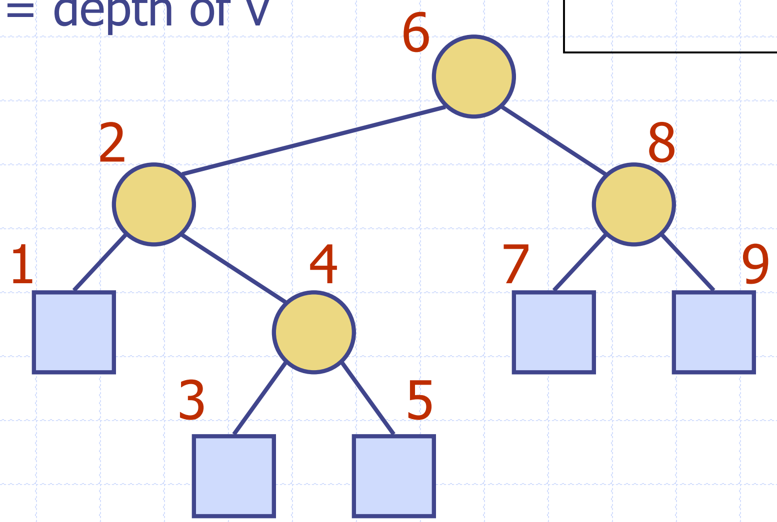
Algorithm *postOrder*(*T*, *v*)
  for each child *w* of *v* in *T*
    postOrder (*T*, *w*)
  visit(*v*)

Algorithm *binaryPostorder*(*T*, *v*)
  if *hasLeft* (*v*) **then**
    binaryPostorder(*T*, *left*(*v*))
  if *hasRight* (*v*) **then**
    binaryPostorder(*T*, *right*(*v*))
  visit(v)

Inorder Traversal

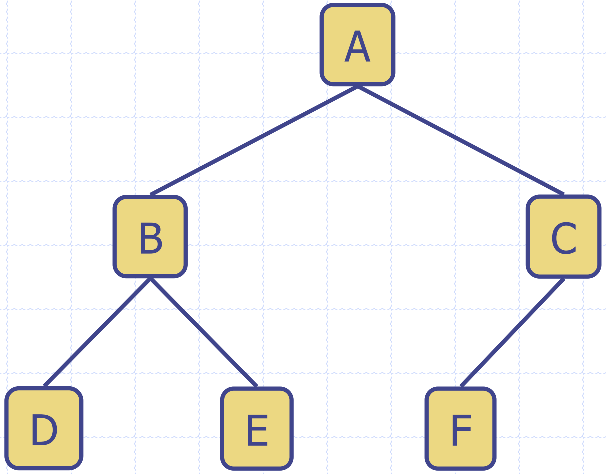
- In an inorder traversal a node is visited after its left subtree and before its right subtree
- Application: draw a binary tree
  - x(v) = inorder rank of v
  - y(v) = depth of v

Algorithm *inOrder*(*T*, *v*)
    if *hasLeft* (*v*)
        inOrder (*T*, *left* (*v*))
    visit(*v*)
    if *hasRight* (*v*)
        inOrder (*T*, *right* (*v*))



Binary Tree Traversals – To sum up

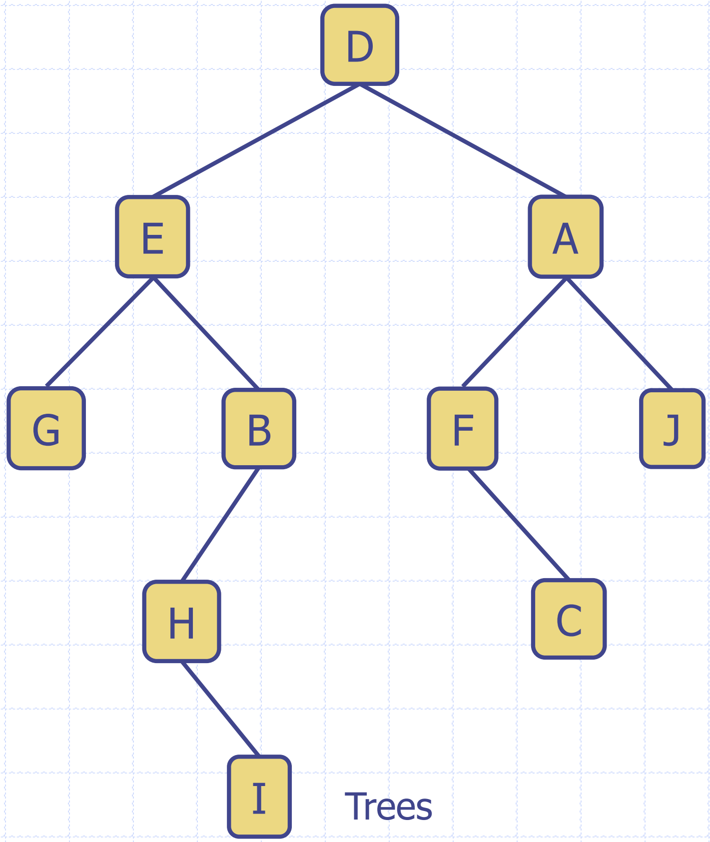
- Basic operations of the Binary Tree data structure are systematic traversals of the nodes of the tree.
- The common ways of traverse a binary tree are:
  - Preorder: Visit-Left-Right
  - Inorder: Left-Visit-Right
  - Postorder: Left-Right-Visit
- Example
  - Preorder: A [left] [right] = A B D E C F
  - Inorder: [left] A [right] = D B E A F C
  - Postorder: [left] [right] A = D E B F C A



- See also:
  - http://www.khanacademy.org/cs/depth-first-traversals-of-binary-trees/934024358

Exercise 2 – Binary tree traversals

- List the nodes of the following binary tree in preorder, postorder and inorder traversals.



Exercise 2 – Binary tree traversals – Answer

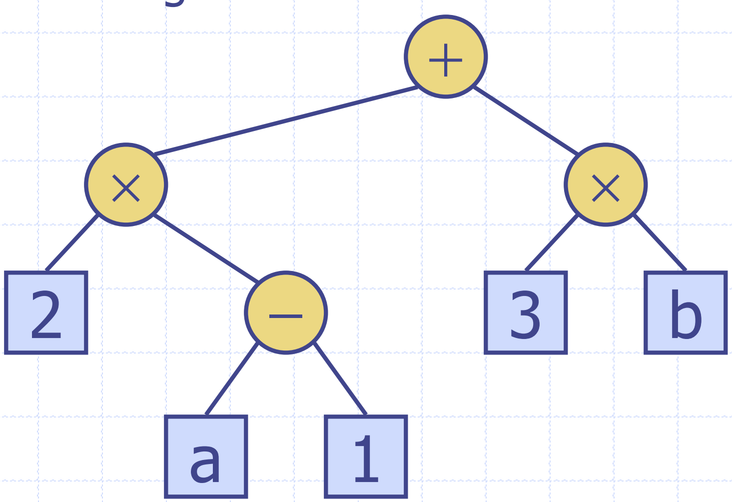
- Preorder: D E G B H I A F C J
- Postorder: G I H B E C F J A D
- Inorder: G E H I B D F C A J

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*(T, v)

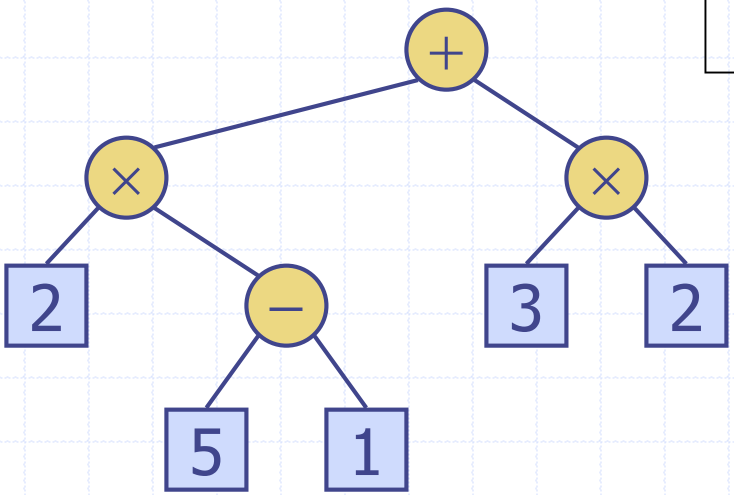
    if *hasLeft* (v)
        print("(")
        inOrder $(T, left(v))$
    print(*v.element* ())
    if *hasRight* (v)
        inOrder $(T, right(v))$
        print (")")



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

Evaluate Arithmetic Expressions

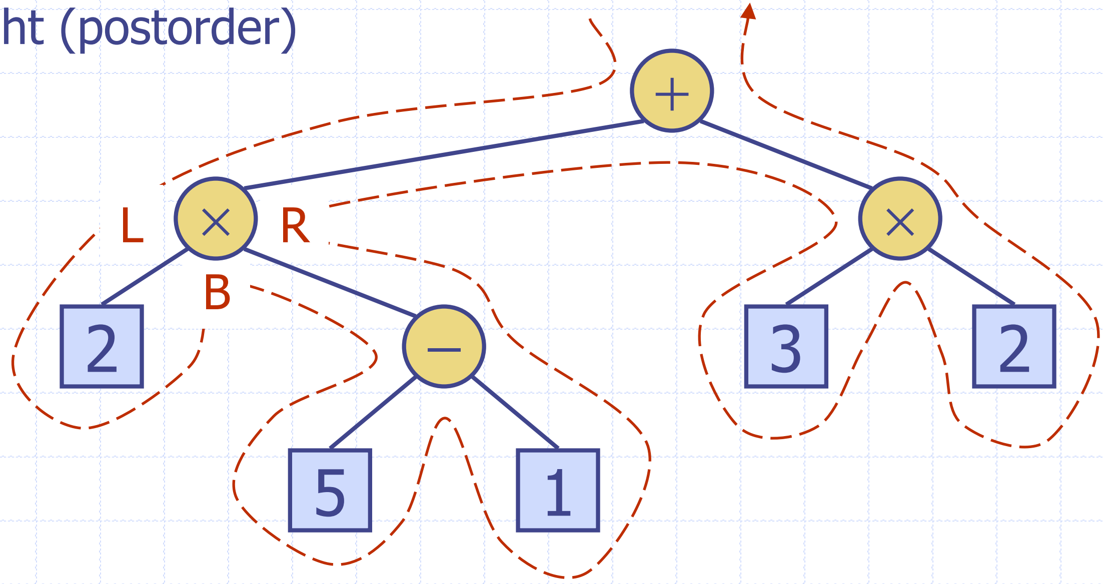
- Specialization of a postorder traversal
  - recursive method returning the value of a subtree
  - when visiting an internal node, combine the values of the subtrees



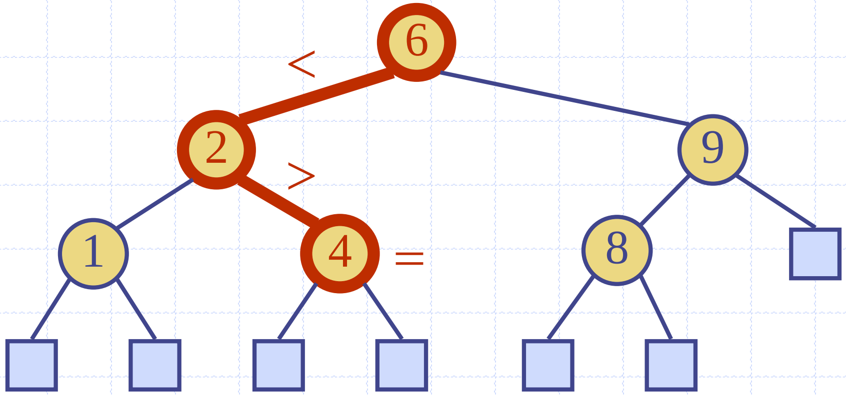
Algorithm *evalExpr(T, v)*
  if *isExternal* (*v*)
    return *v.element* ()
  else
    x ← *evalExpr*(*T*, *leftChild* (*v*))
    y ← *evalExpr*(*T*, *rightChild* (*v*))
    ◊ ← operator stored at *v*
    return *x* ◊ *y*

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)



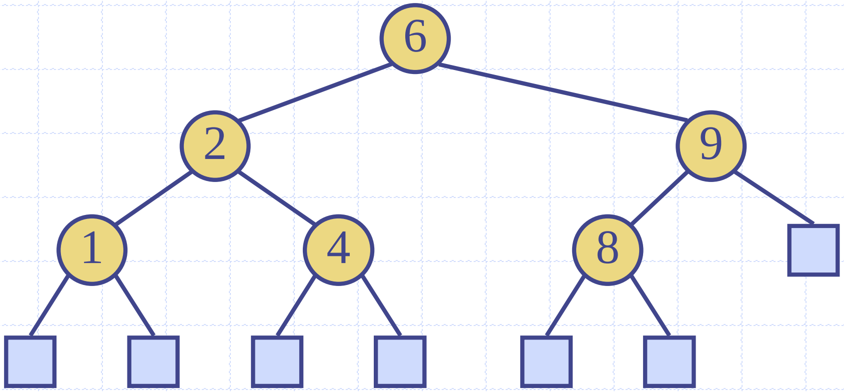
Binary Search Trees



Binary Search Trees

- A binary search tree is a binary tree storing keys (or key-value entries) at its internal nodes and satisfying the following property:
  - Let u, v, and w be three nodes such that u is in the left subtree of v and w is in the right subtree of v. We have $key(u) \leq key(v) \leq key(w)$
- External nodes do not store items

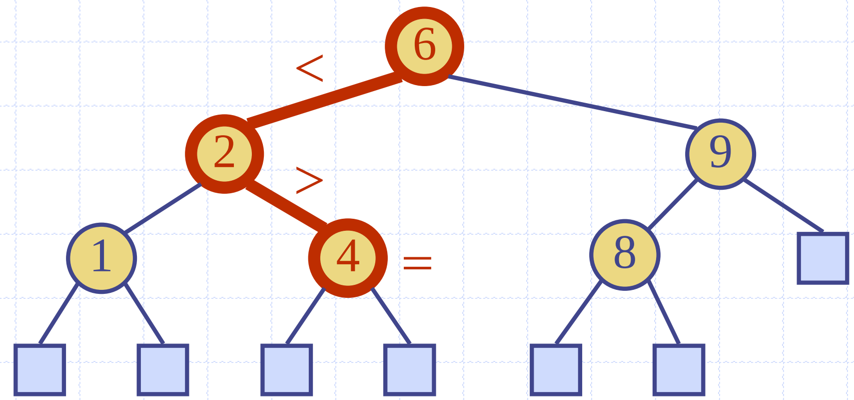
- An inorder traversal of a binary search trees visits the keys in increasing order



Search

- To search for a key k, we trace a downward path starting at the root
- The next node visited depends on the comparison of k with the key of the current node
- If we reach a leaf, the key is not found
- Example: TreeSearch(4,root)

```
Algorithm TreeSearch(k, v)
    if isExternal (v)
        return v
    if k < key(v)
        return TreeSearch(k, left(v))
    else if k = key(v)
        return v
    else { k > key(v) }
        return TreeSearch(k, right(v))
```

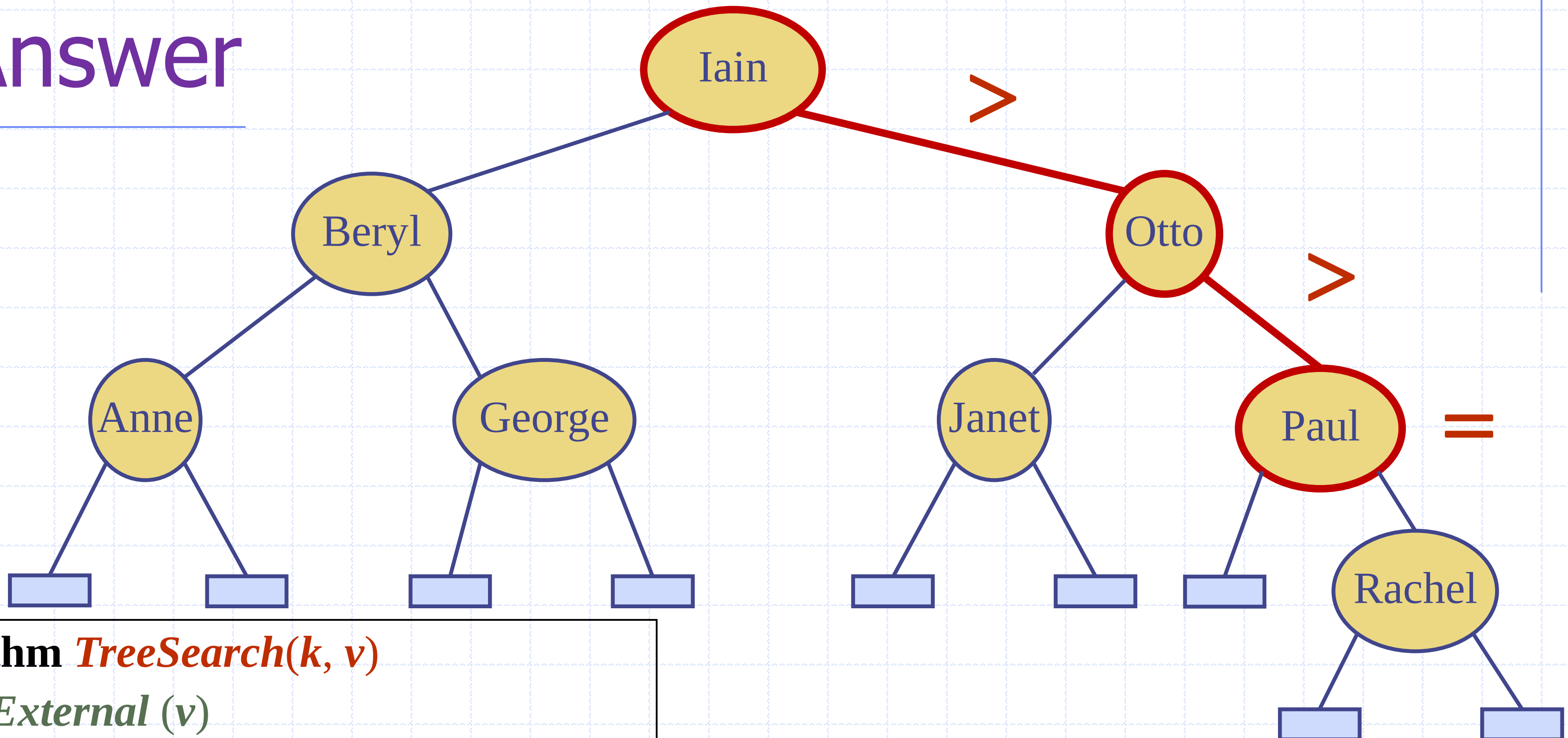


Exercise 3 – Binary search tree

- Represent the below array as a binary search tree and show the execution of *TreeSearch* algorithm.
- Give the set of steps that have to be performed in order to find element "Paul" in an ordered array presented below

0	1	2	3	4	5	6	7
Iain	Beryl	Otto	Anne	George	Janet	Paul	Rachel

Exercise 3 – Binary search tree – Answer



Algorithm *TreeSearch*(k, v)
    if *isExternal* (v)
        return v
    if $k < key(v)$
        return *TreeSearch*(k, *left*(v))
    else if $k = key(v)$
        return v
    else { $k > key(v)$ }
        return *TreeSearch*(k, *right*(v))