



AE2ADS

Algorithms Data Structures & Efficiency


Lecturer: Heshan Du

Email: Heshan.Du@nottingham.edu.cn

University of Nottingham Ningbo China



Abstract Data Types vs. Concrete Data Structures



Abstract Data Type (ADT)	Concrete Data Structure
Stack	Array
Queue	Singly Linked List
List	Doubly Linked List
Positional List	

Abstract Data Types vs. Concrete Data Structures

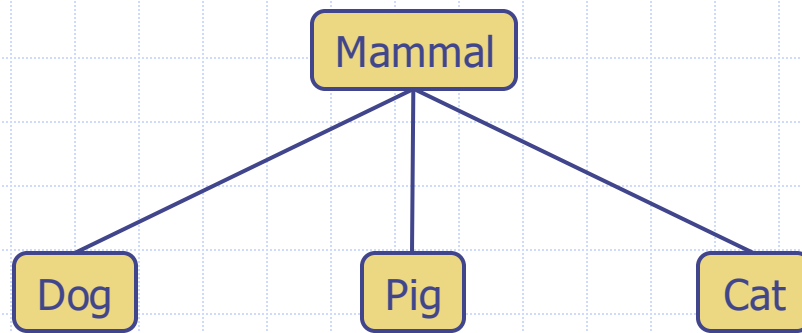
Abstract Data Type (ADT)	Concrete Data Structure
Stack	Array
Queue	Singly Linked List
List	Doubly Linked List
Positional List	
Tree	
Binary Tree	

Trees



The photo is from Pixabay, under CC0 License.

Trees



Aim and Learning Objectives

- ❑ To be able to *understand* and describe the tree ADT and the binary tree ADT, as well as related important concepts and properties.
- ❑ To be able to *implement* the tree ADT and the binary tree ADT, and analyze the complexity of implemented methods.
- ❑ To be able to *apply* tree structures to solve problems.

Aim and Learning Objectives

- ❑ To be able to *understand* and describe tree traversal algorithms.
- ❑ To be able to *implement* tree traversal algorithms and analyze their complexity.
- ❑ To be able to *apply* tree traversal algorithms to solve problems.

Reading

M. T. Goodrich, R. Tamassia and M. H. Goldwasser,
Data Structures and Algorithms in Java, 6th Edition,
2014.

- Chapter 8. Tree Structures

Overview of Contents

1. Tree definitions and tree ADT
2. Tree traversal algorithms
3. Binary tree definitions and binary tree ADT
4. Binary tree traversal algorithms
5. Implementation of tree and binary tree ADTs using concrete data structures

Overview of Contents

1. **Tree definitions and tree ADT**
2. Tree traversal algorithms
3. Binary tree definitions and binary tree ADT
4. Binary tree traversal algorithms
5. Implementation of tree and binary tree ADTs using concrete data structures

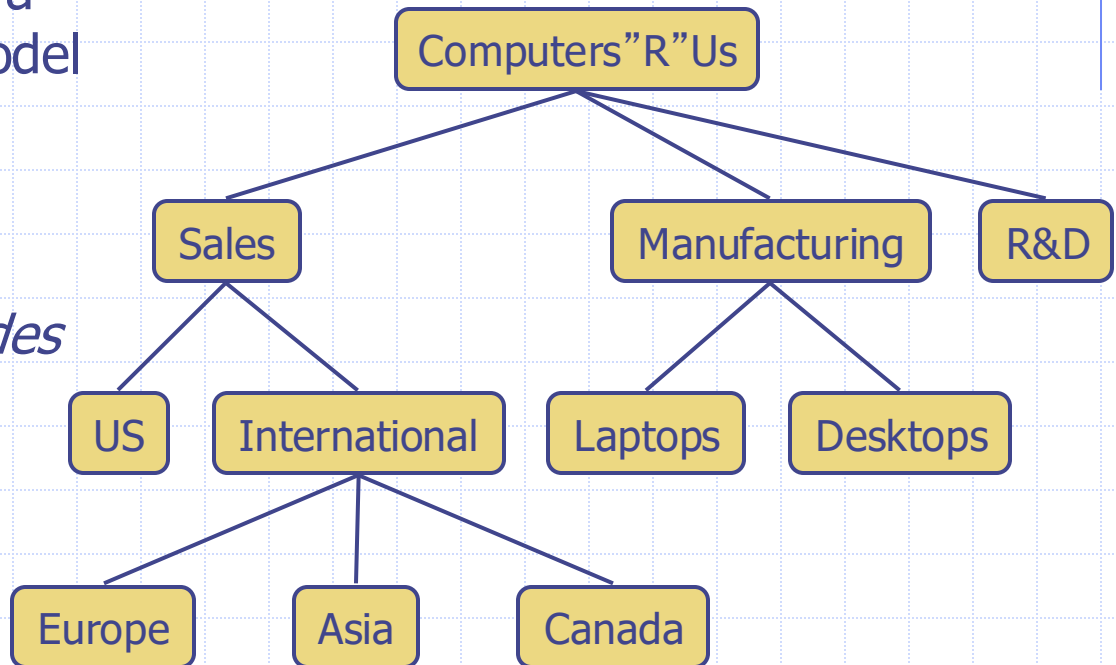
What is a Tree

- In computer science, a tree is an abstract model of a **hierarchical structure** 层次结构

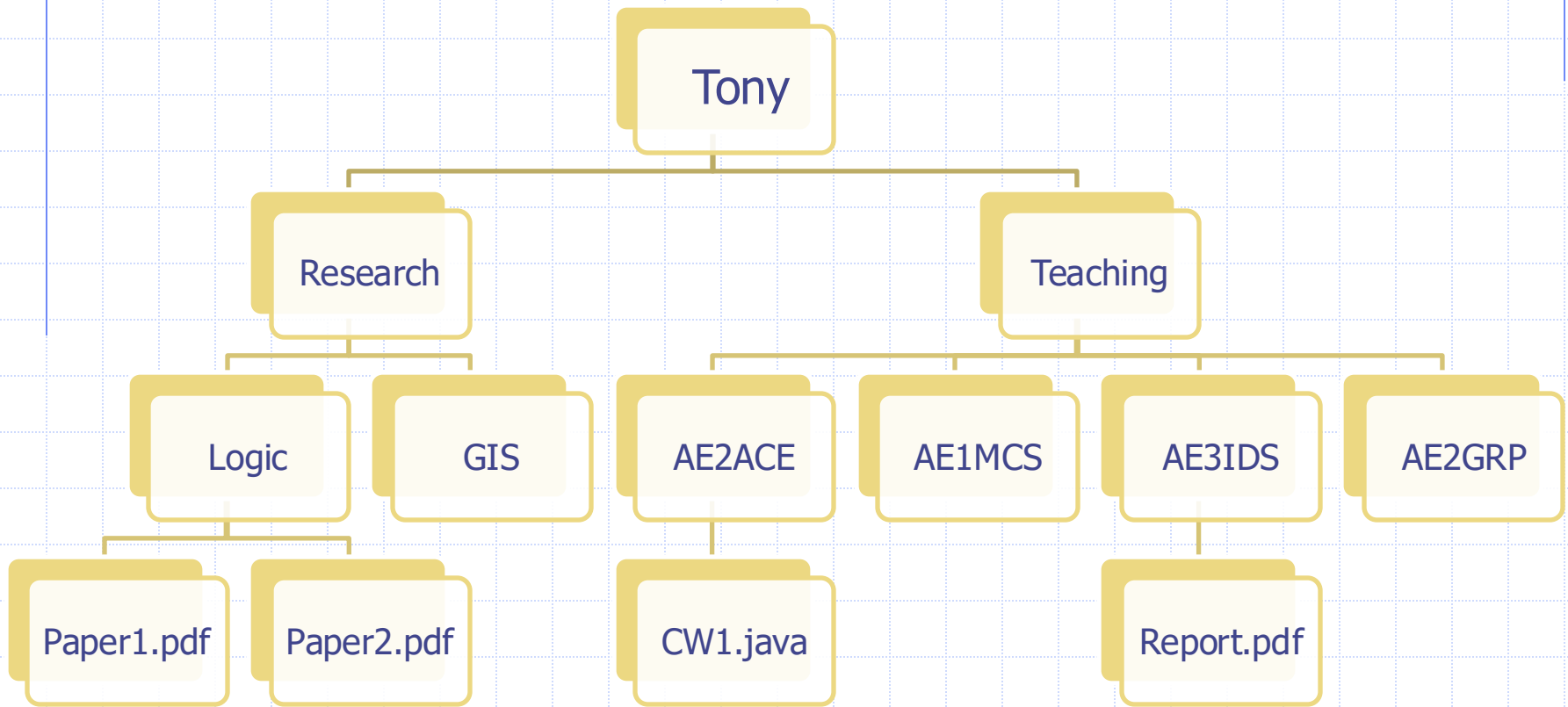
树由多个节点 (nodes) 构成, 每个节点有一个父节点和一个或多个子节点, 这就是父子关系 (parent-child relation)。

- A tree consists of *nodes* with a **parent-child relation**

- Applications:
 - **Organization charts**
 - **File systems**

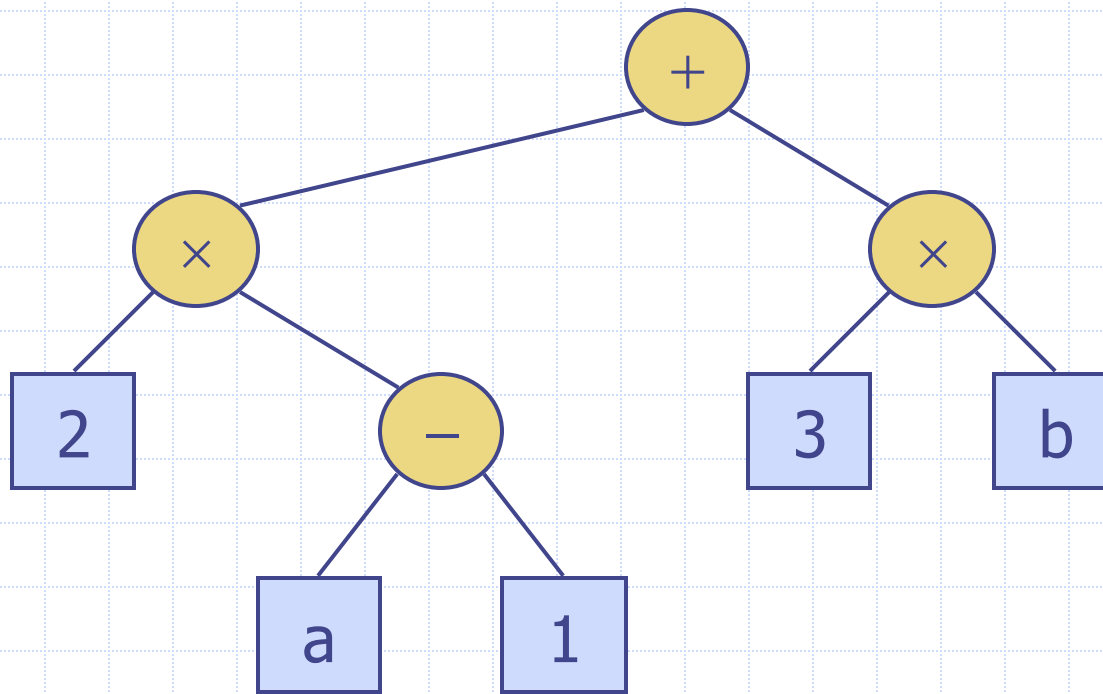


File System



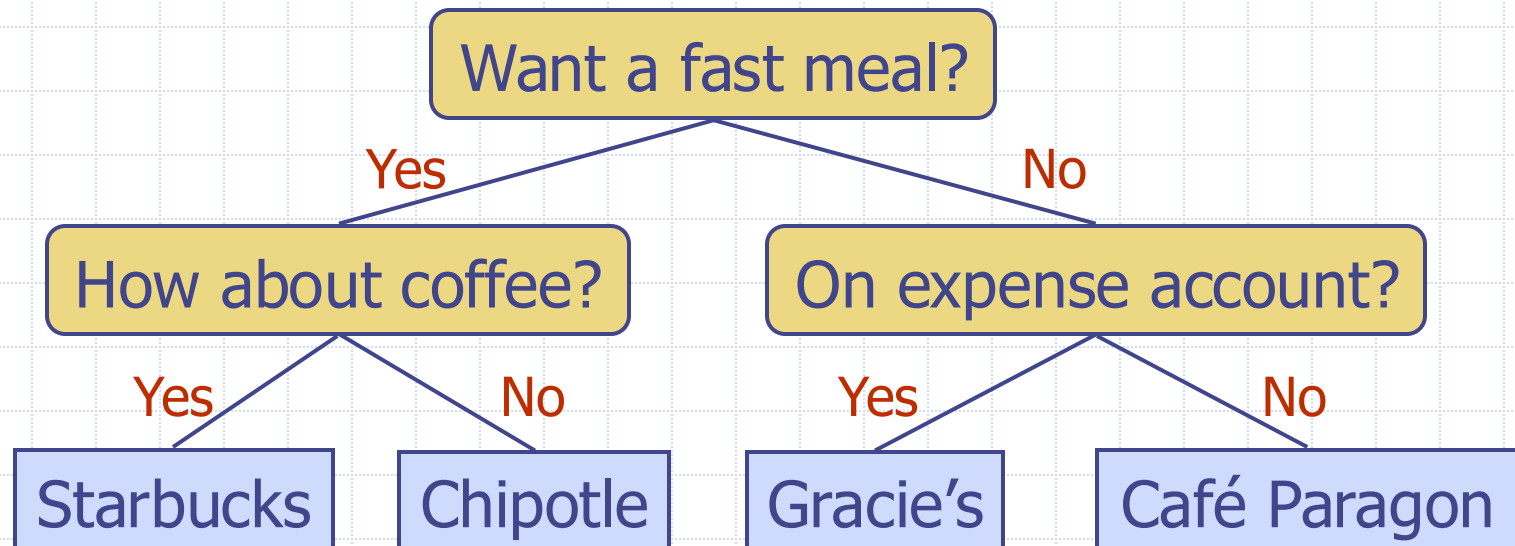
Arithmetic Expression Tree

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



Decision Tree

Example: dining decision



What is a tree?

- What are the *common properties* of these example trees?
- How to decide whether a given structure is a tree or not?
- What are the main criteria?

树的主要特点：

层级结构：树结构由根节点和多个层级的节点组成，每个节点有一个父节点和零个或多个子节点。

无环结构：树是一种无环的结构，意味着没有节点是自己或自己的子节点的祖先。

连通性：树的节点是连通的，即从根节点出发，能够访问所有的其他节点。

如何判断一个结构是否是树：

有根节点：树必须有一个根节点，且只有一个根节点。

父子关系：每个节点都应有一个父节点（除了根节点），并且每个节点可以有多个子节点。

无环：树结构不能有环，即不存在节点自指或回到自身的情况。

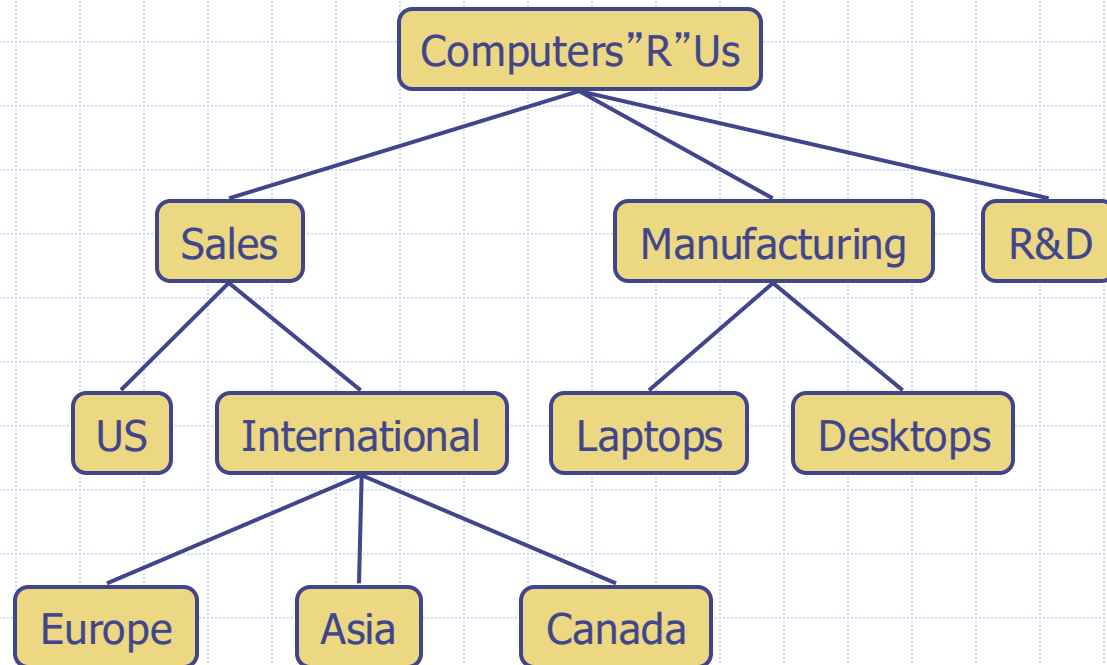
主要标准：

树的结构：树是由节点组成的，每个节点有且只有一个父节点。

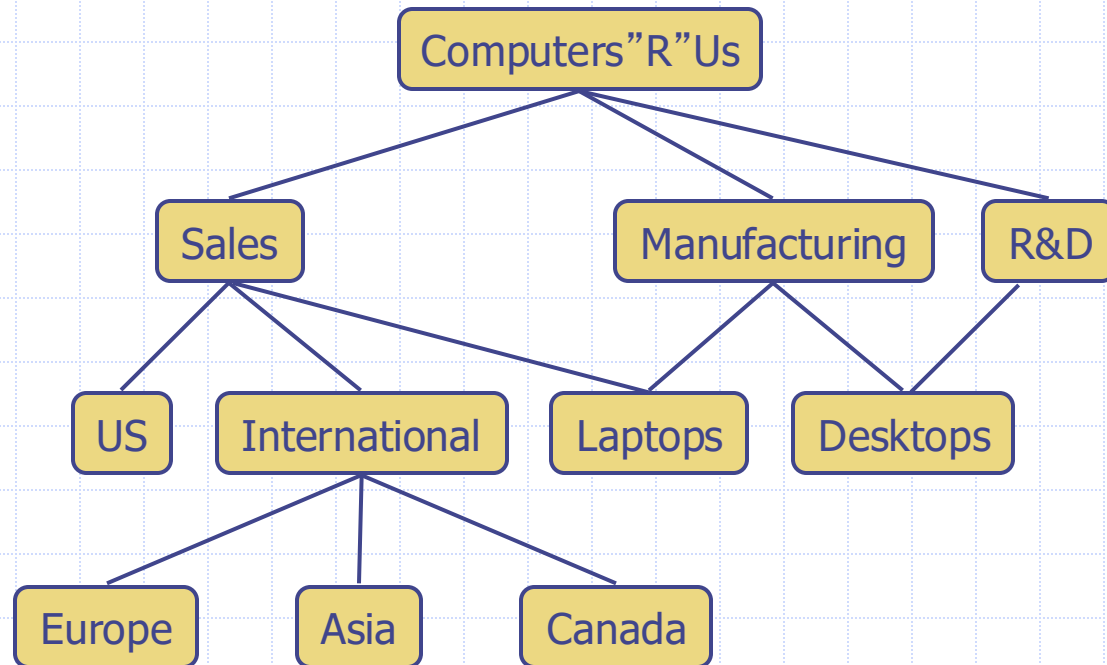
无环性：树中没有环的存在，不能形成回路。

连通性：树中的所有节点都能通过父子关系被访问到。

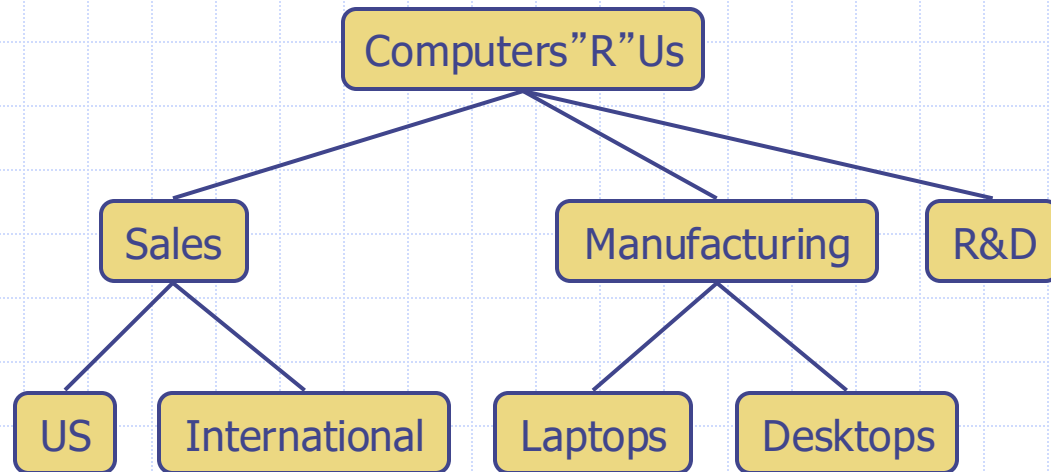
Is this a tree?



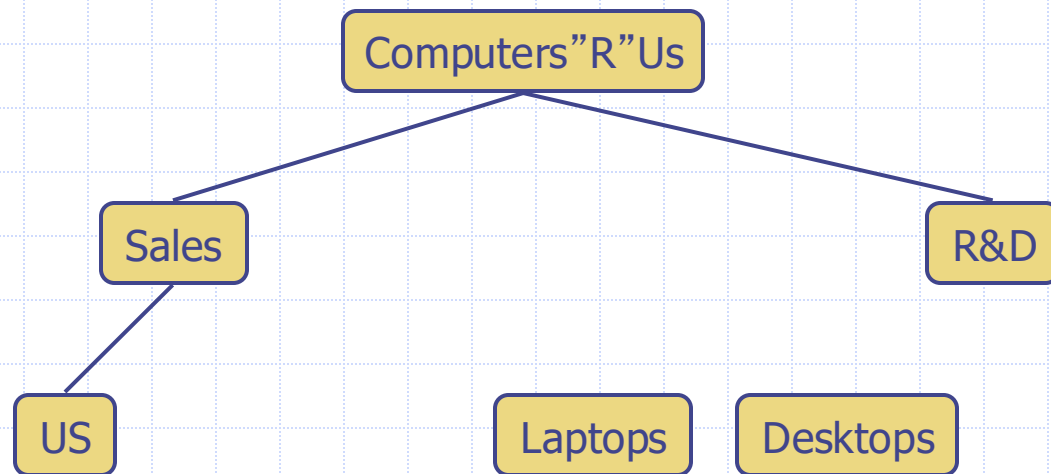
Is this a tree?



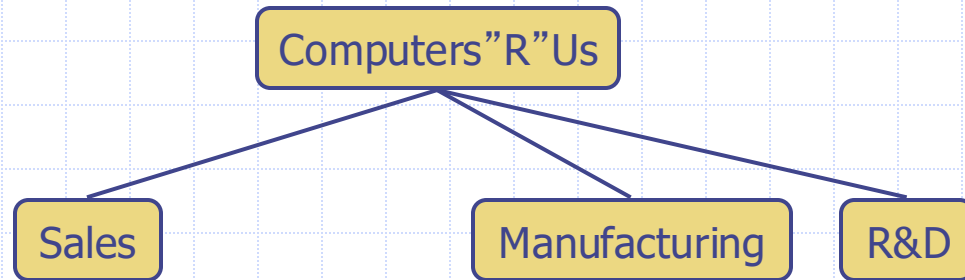
Is this a tree?



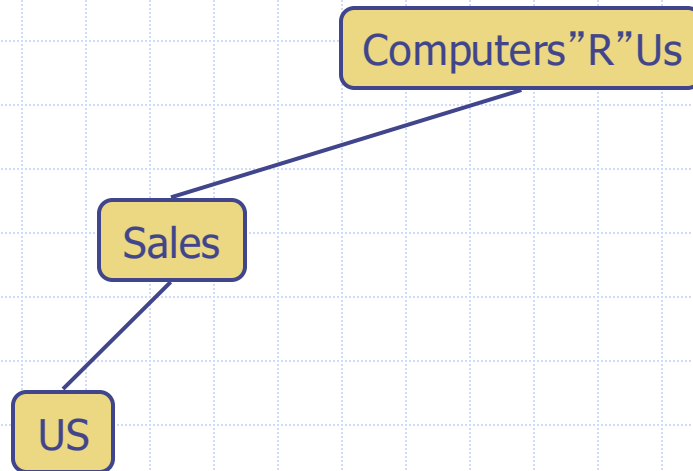
Is this a tree?



Is this a tree?



Is this a tree?



Is this a tree?

Computers"R"Us

What is a tree?

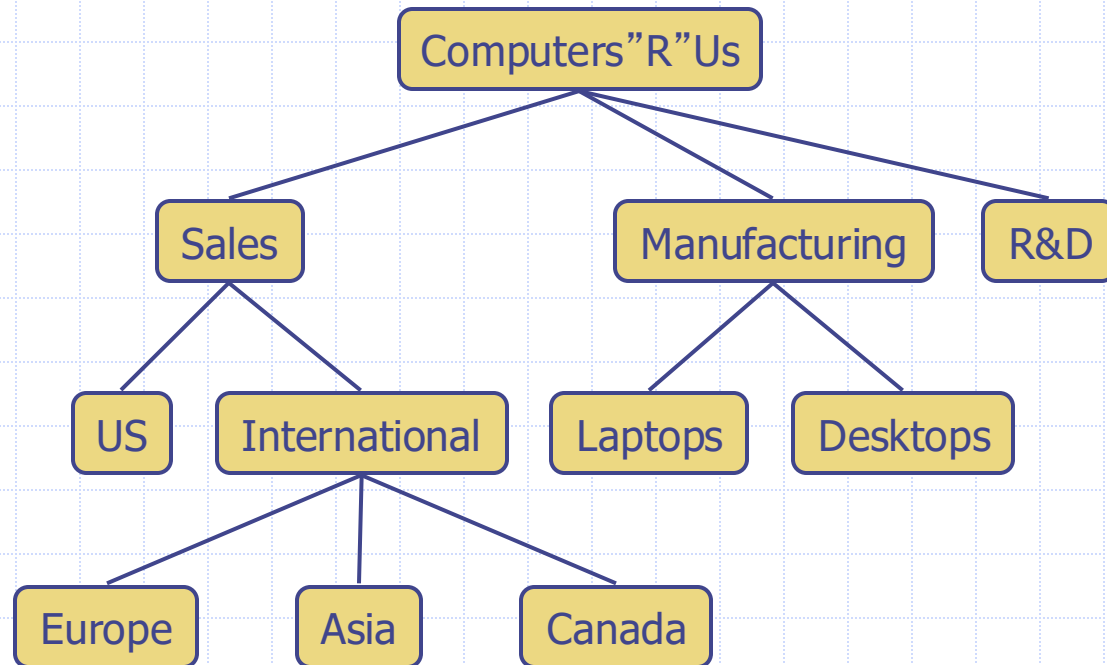
- ❑ A tree can be defined regarding to properties of its nodes.
- ❑ What property does a node in a tree have?
- ❑ Is there any special node in a tree?
- ❑ Write down a definition of a tree in your own words.

Tree

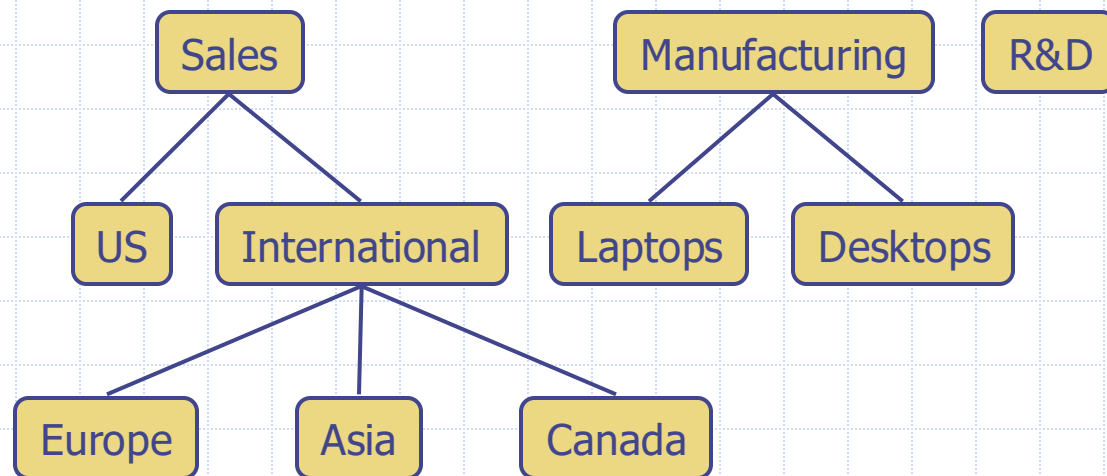
A **tree** T is defined as a set of **nodes** storing elements such that the nodes have a **parent-child** relationship that satisfies the following properties:

- 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 .

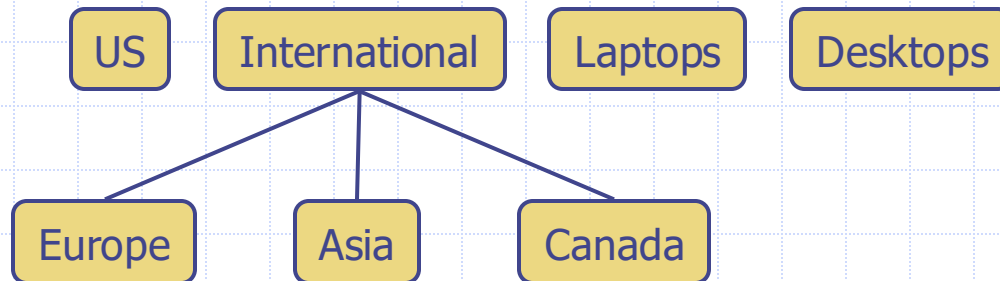
Is this a tree?



Are they trees?



Are they trees?



Are they trees?



Europe

Asia

Canada

What is a tree?

递归

- A tree can be defined *recursively*.
- Let us allow a tree to be *empty*, meaning that it does not have any nodes.
- Write down a *recursive definition* of a tree in your own words.

Recursive Definition of a Tree

- *A tree can be empty*, meaning that it does not have any nodes.
- This convention also allows us to define a tree recursively such that
 - a tree T is either empty
 - or consists of a node r , called the root of T , and a (possibly empty) set of *trees* whose roots are the children of r .

Siblings (兄弟节点) : 两个节点是同一父节点的孩子节点, 它们被称为兄弟节点。

Internal nodes (内部节点) : 如果一个节点有一个或多个孩子节点, 那么这个节点就是内部节点。

External nodes (外部节点) : 如果一个节点没有孩子节点, 那么这个节点是外部节点。外部节点也称为 叶子节点。

Other Node Relationships

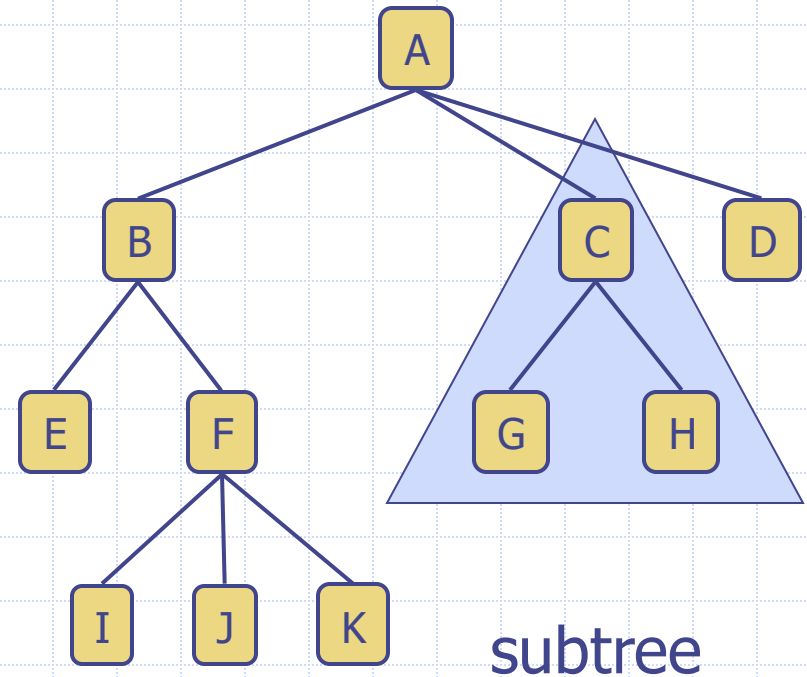
- ❑ Two nodes that are children of the same parent are ***siblings***.
- ❑ A node v is ***internal*** if it has one or more children.
- ❑ A node v is ***external*** if v has no children.
- ❑ External nodes are also known as ***leaves***.

Other Node Relationships

- A node u is an 祖先 **ancestor** of a node v if
 - $u = v$
 - or u is an ancestor of the parent of v .
- Conversely, we say that a node v is a 后代 **descendant** of a node u if u is an ancestor of v .
- The **subtree** of T **rooted** at a node v is the tree consisting of all the descendants of v in T (including v itself).

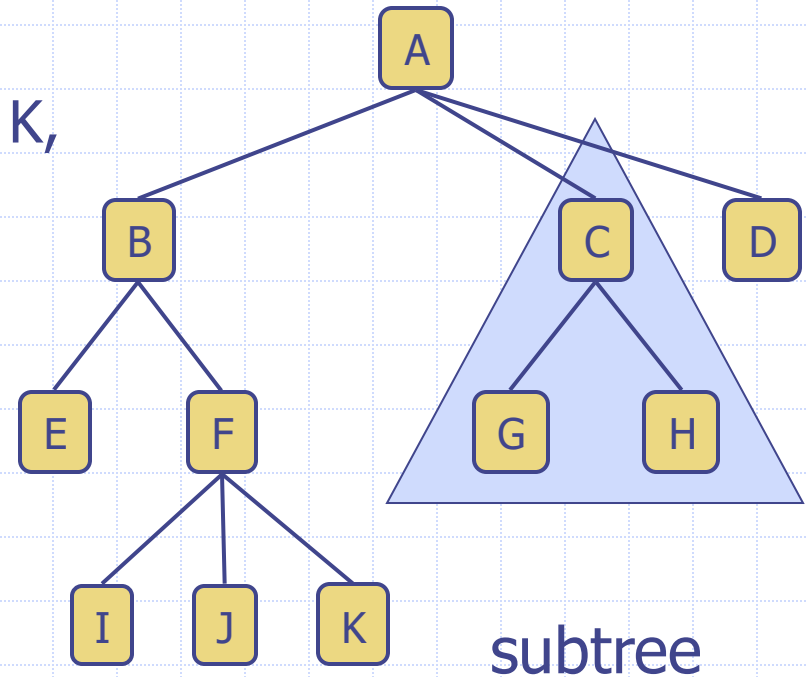
Tree Terminology

- ❑ **Root**: node without parent ()
- ❑ **Internal node**: node with at least one child (, ,)
- ❑ **External node** (a.k.a. leaf): node without children (, , , ,)
- ❑ **Ancestors** of a node v : v , parent, grandparent, grand-grandparent, etc.
- ❑ **Descendant** of a node v : v , child, grandchild, grand-grandchild, etc.
- ❑ **Subtree**: tree consisting of a node and its descendants



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 v : v , *parent*, grandparent, grand-grandparent, etc.
- ❑ **Descendant** of a node v : v , *child*, grandchild, grand-grandchild, etc.
- ❑ **Subtree**: tree consisting of a node and its descendants



size() : 返回树中节点的数量。
isEmpty() : 判断树是否为空。
iterator() : 返回树的迭代器, 用于遍历树中的元素。
positions() : 返回树中所有节点的可迭代位置。

isInternal(p) : 判断位置 p 是否是内部节点 (即它有子节点)。
isExternal(p) : 判断位置 p 是否是外部节点 (即它没有子节点)。
isRoot(p) : 判断位置 p 是否是根节点。

root() : 返回树的根节点的位置。
parent(p) : 返回位置 p 的父节点的位置。
children(p) : 返回位置 p 的子节点 (一个可迭代的列表)。
numChildren(p) : 返回位置 p 的子节点数量。

Tree ADT

- We use **positions** to abstract nodes

- Generic methods:
 - integer **size()**
 - boolean **isEmpty()**
 - Iterator **iterator()**
 - Iterable **positions()**

- Accessor methods:
 - position **root()**
 - position **parent(p)**
 - Iterable **children(p)**
 - Integer **numChildren(p)**

◆ Query methods:

- boolean **isInternal(p)**
- boolean **isExternal(p)**
- boolean **isRoot(p)**

◆ Additional methods may be defined by data structures implementing the Tree ADT

◆ *Why do the assessor methods include **parent(p)** **children(p)**, rather than **ancestors(p)** **descendants(p)**?*

Depth

Let p be a position within tree T .

深度是指在树中某个位置 p 的祖先节点的数量，除了 p 本身。
换句话说，深度就是从根节点到达该节点所经过的边的数量。

The **depth** of p is the number of ancestors of p , other than p itself.

What is the depth of the root of T ?

The depth of the root node is 0 because the root node has no ancestors.
根节点的深度为 0，因为根节点没有祖先。

Recursive Definition of Depth

The depth of p can also be recursively defined as follows:

- If p is the root, then the depth of p is 0.
- Otherwise, the depth of p is one plus the depth of the parent of p .

如果 p 是根节点, 则 p 的深度为 0。
否则, p 的深度是 父节点深度 + 1。

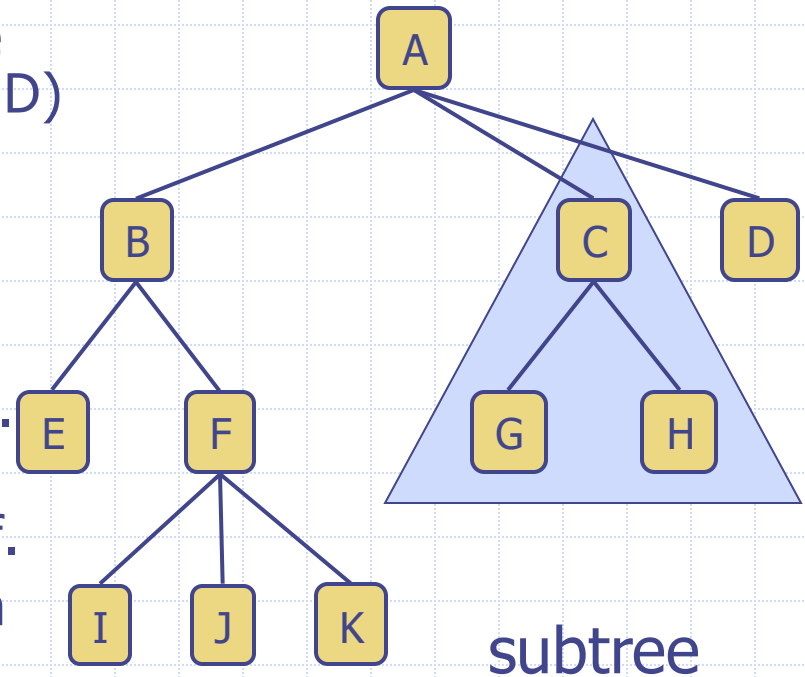
Height

The ***height*** of a tree is the maximum of the depths of its positions.

树的高度 是指树中所有位置（节点）深度的 **最大值**。
简单来说，高度是从根节点到任一叶节点的最长路径。
它是衡量树的“垂直大小”或从根节点到最远叶节点的深度。

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 v: v, parent, grandparent, grand-grandparent, etc.
- ❑ **Descendant** of a node v: v, child, grandchild, grand-grandchild, etc.
- ❑ **Depth** of a node p: number of ancestors of p, other than p itself.
- ❑ **Height** of a tree: maximum depth of any node (3)
- ❑ **Subtree:** tree consisting of a node and its descendants



```
Function height(node):  
    if node is null:  
        return -1  
    left_height = height(node.left)  
    right_height = height(node.right)  
    return 1 + max(left_height, right_height)
```

After class exercise

- ❑ How to calculate the height of a tree?
- ❑ Write down the pseudocode of your algorithm.
- ❑ What is the big-Oh complexity of your algorithm? $O(n)$
- ❑ Read Section 8.1.2 Computing Depth and Height

Recursive Definition of Height

Formally, we define the **height** of a position p in a tree T as follows:

- If p is a leaf, then the height of p is 0.
- Otherwise, the height of p is one more than the maximum of the heights of p 's children.

树高度的递归定义：

如果 p 是叶节点，则位置 p 的高度为0。

否则，位置 p 的高度是其所有子节点高度的最大值再加1。

After class exercise

The following proposition relates our original definition of the height of a tree to the height of the *root* position using this recursive formula.

Why?

在树的高度的递归定义中，如果一个节点是叶子节点，它的高度为0（因为它没有子节点）。任何非叶子节点的高度定义为它的所有子节点高度的最大值加1。因此，树的根节点的高度是由它所有子节点的最大高度决定的，这些子节点最终会到达叶子节点。由于每个叶子节点的深度是0，根节点的高度就是从任何叶子节点到根的最大深度。

Proposition: The height of the root of a nonempty tree T , according to the recursive definition, equals the maximum depth among all leaves of tree T .

Overview of Contents

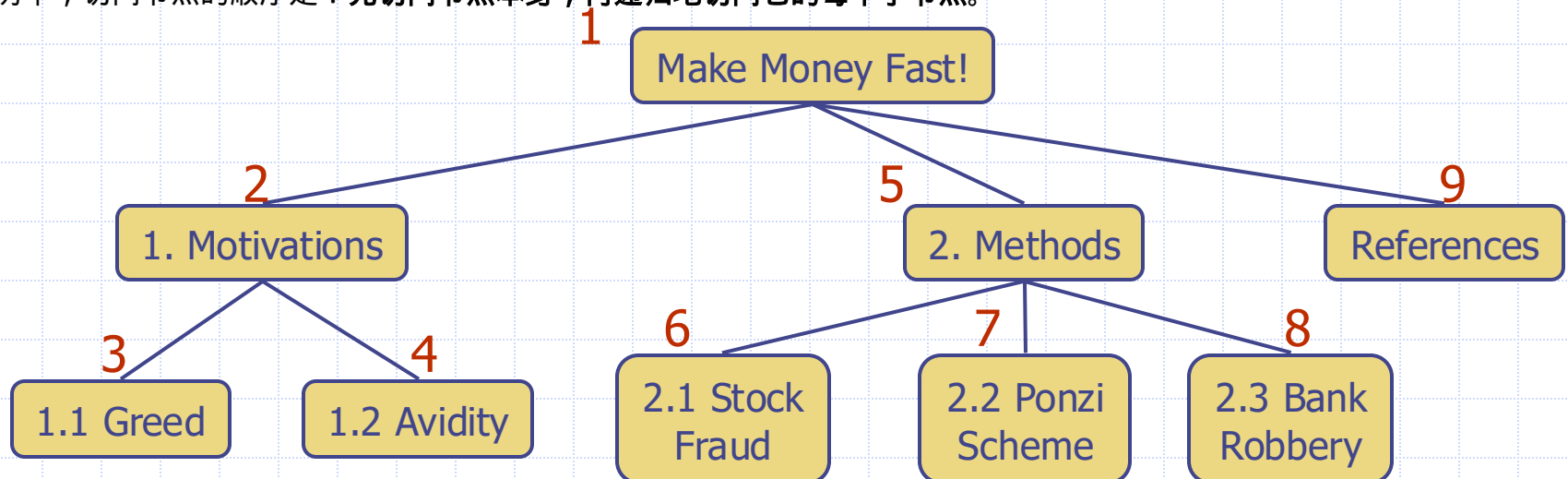
1. Tree definitions and tree ADT
2. **Tree traversal algorithms**
3. Binary tree definitions and binary tree ADT
4. Binary tree traversal algorithms
5. Implementation of tree and binary tree ADTs using concrete data structures

Preorder Traversal 前序遍历

- 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)

在前序遍历中，访问节点的顺序是：先访问节点本身，再递归地访问它的每个子节点。

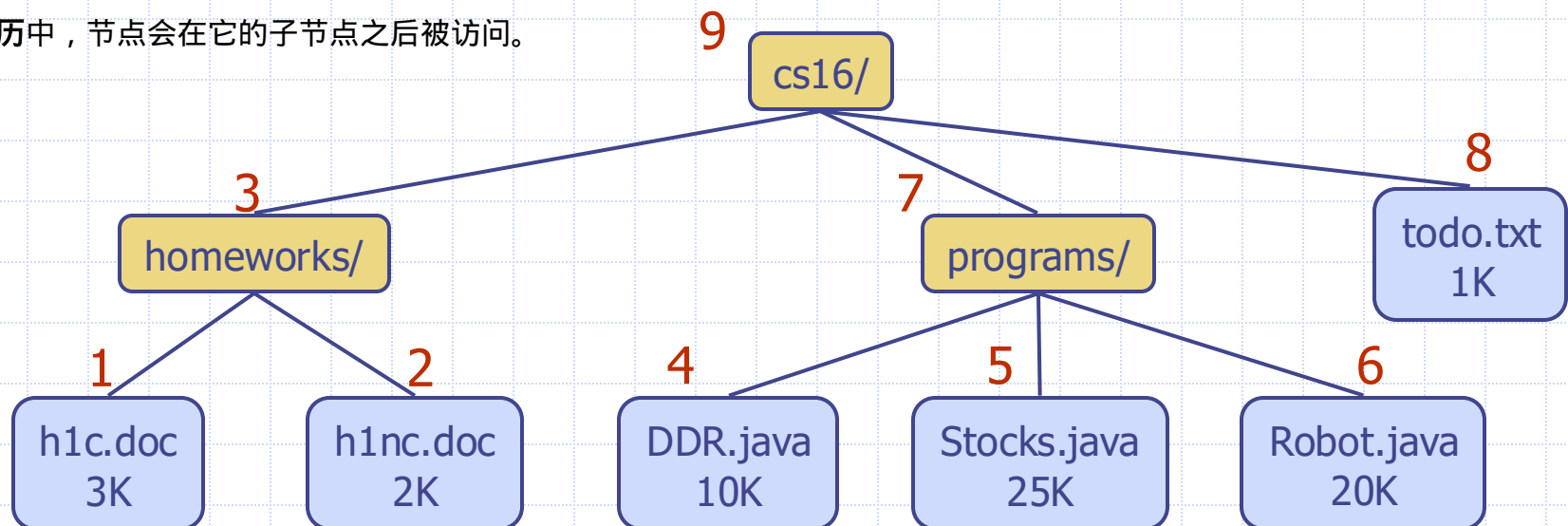


Postorder Traversal 后序遍历

- 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)

在后序遍历中，节点会在它的子节点之后被访问。



Overview of Contents

1. Tree definitions and tree ADT
2. Tree traversal algorithms
3. **Binary tree definitions and binary tree ADT**
4. Binary tree traversal algorithms
5. Implementation of tree and binary tree ADTs using concrete data structures

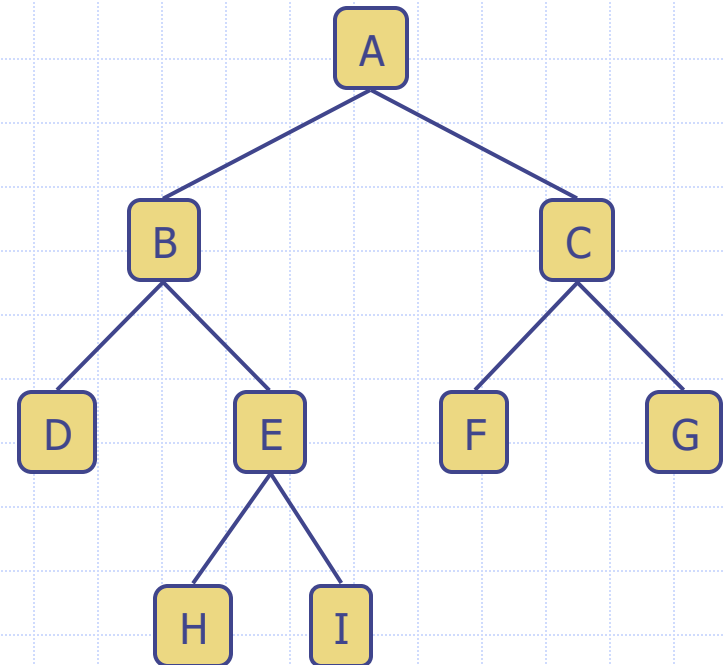
定义：二叉树是具有以下特性的树：每个内部节点最多有两个子节点（对于完全二叉树，每个内部节点恰好有两个子节点）。每个节点的子节点是有顺序的成对存在的。

子节点：内部节点的左子节点和右子节点分别称为左子节点和右子节点。

Binary Trees

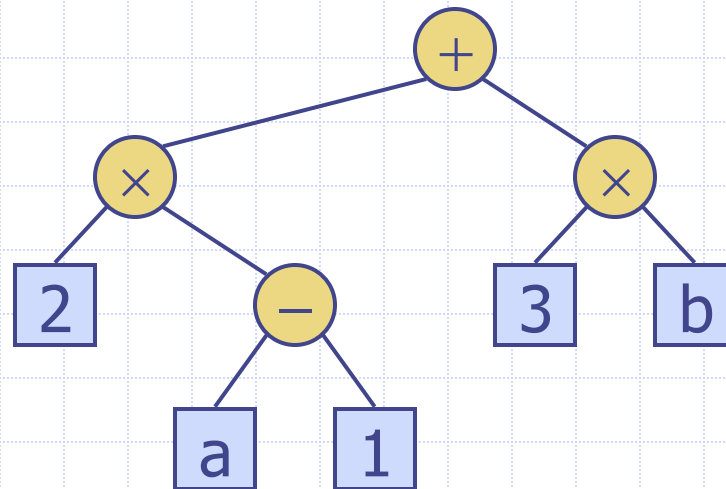
- A *binary tree* is a tree with the following **properties**:
 - Each internal node has **at most two children** (exactly two for **proper** binary trees)
 - The children of a node are **an ordered pair**
- We call the children of an internal node **left child** and **right child**

- Applications:
 - arithmetic expressions
 - decision processes
 - searching



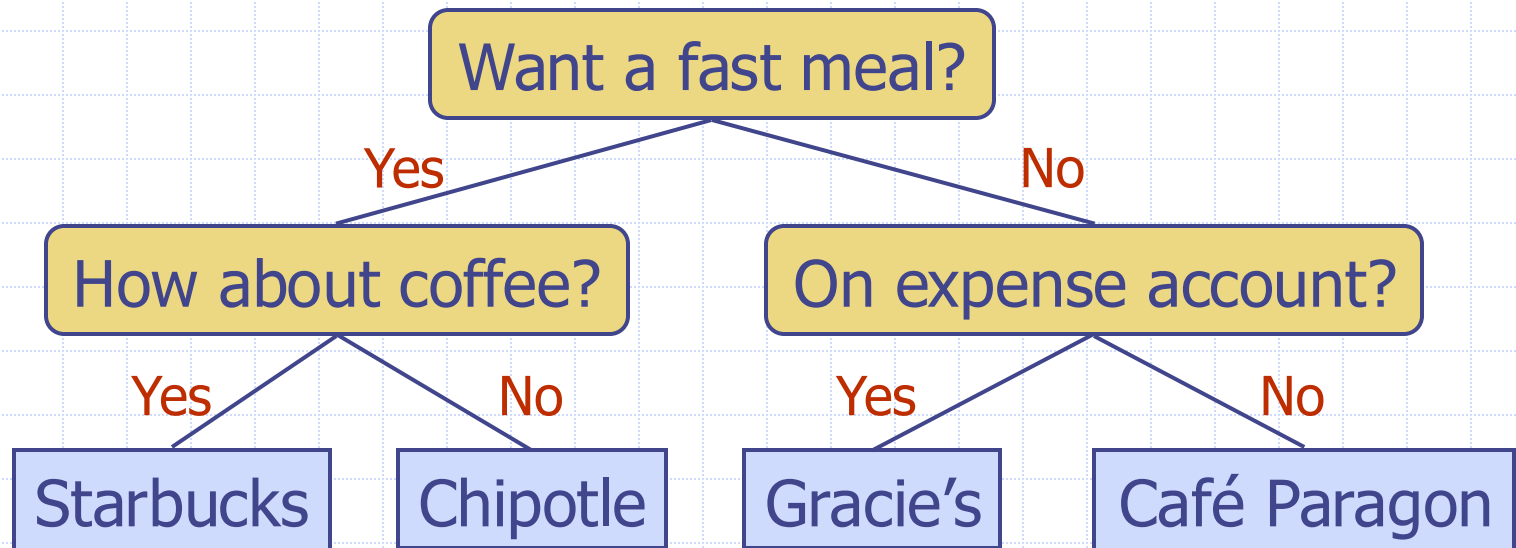
Arithmetic Expression Tree

- Binary tree associated with an arithmetic expression
 - internal nodes: operators $+$ $-$ $*$ $/$
 - external nodes: operands 操作数
- 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



Exercise

Refer to the recursive definition of a tree, write down a *recursive definition* of a binary tree in your own words.

Recursive Definition of a Tree

- A tree can be *empty*, meaning that it does not have any nodes.
- This convention also allows us to define a tree recursively such that
 - a tree T is either empty
 - or consists of a node r , called the root of T , and a (possibly empty) set of *trees* whose roots are the children of r .

树可以为空，这意味着树没有任何节点。

递归定义：

一个树要么是空的。

或者由一个节点组成，称为树的根节点，并且有一组（可能为空）树，这些树的根节点是该节点的子节点。

Recursive Definition of a Binary Tree

- A binary tree T is either
 - empty
 - or consists of a node r , called the root of T , and *two binary trees* (possibly empty) and whose roots are the *left child* and *right child* of r , respectively.

二叉树是以下两种之一：
空树。

由一个节点 r 组成，称为树 T 的根节点，并且有两个二叉树（可能为空），它们的根分别是 r 的左子节点和右子节点。

Recursive Definition of a Binary Tree

A binary tree is either:

- An empty tree.
- A nonempty tree having a root node r , which stores an element, and two binary trees that are respectively the left and right subtrees of r .

We note that one or both of those subtrees can be empty by this definition.

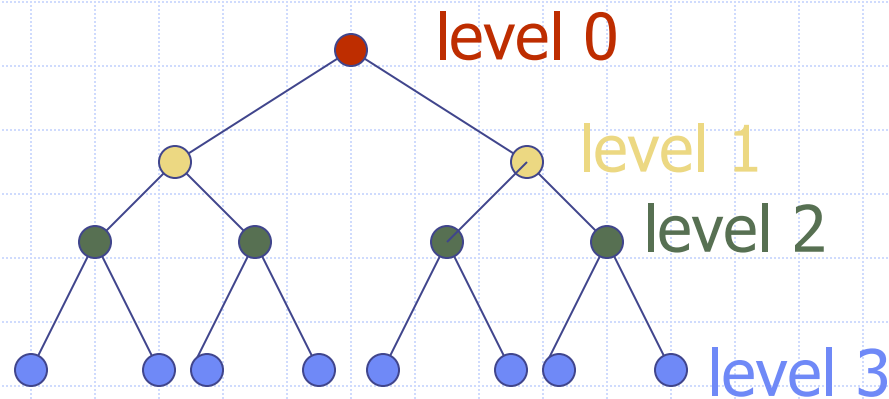
BinaryTree ADT

- The **BinaryTree** ADT extends the Tree ADT, i.e., it *inherits all the methods of the Tree ADT*
- Additional methods:
 - position **left**(p)
 - position **right**(p)
 - position **sibling**(p)
- The above methods return **null** when there is no left, right, or sibling of p, respectively
- Update methods may be defined by data structures implementing the BinaryTree ADT

Properties of Proper Binary Trees

A binary tree is proper if every internal node has exactly 2 children.

Proper
binary tree
of height 3:



Proper Binary Tree : 每个内部节点恰好有两个子节点的二叉树。

示例：高度为 3 的适当二叉树。

在这个树中：

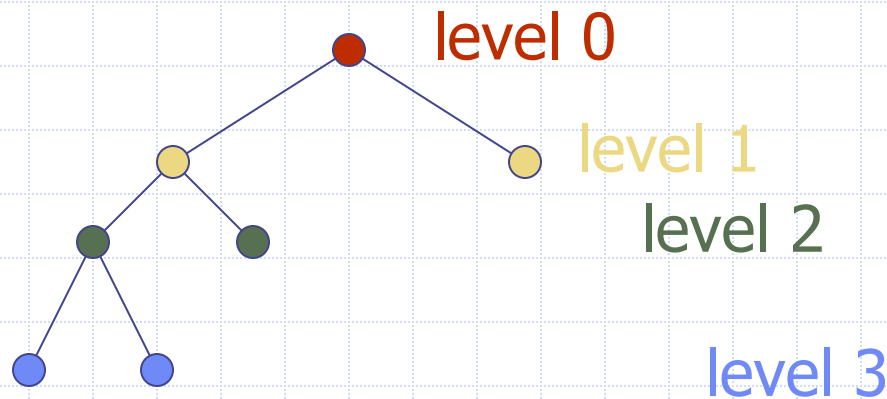
level 0 : 根节点

level 1、level 2、level 3 分别是不同的节点层级，每一层的节点都有两个子节点。

Properties of Proper Binary Trees

A binary tree is proper if every internal node has exactly 2 children.

Proper
binary tree
of height 3:



树的层级：

level 0：根节点。

level 1：根节点的两个子节点。

level 2：每个内部节点の子节点。

level 3：叶节点。

Properties of Proper Binary Trees

A binary tree is proper if every internal node has exactly 2 children.

Let n denote the number of nodes, h denote the height of a proper binary tree T . Then

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

$$\text{Hence } \log(n + 1) - 1 \leq h \leq \frac{n-1}{2}$$

在这个图表中，定义了一个正确的二叉树的性质：
如果每个内部节点都有恰好2个子节点，则该二叉树是正确的。

树的节点数用 n 表示，树的高度用 h 表示。

对于一个正确的二叉树 T ，它满足以下不等式：

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

这意味着二叉树的节点数 n 和高度 h 之间有着明确的关系。

由此得出： $\log(n + 1) - 1 \leq h \leq \frac{n - 1}{2}$

这些公式反映了正确二叉树的结构和它们节点数与高度之间的关系。

How many nodes at level k

- ◆ First, it is useful to find out how many nodes are at a certain level in the proper binary tree
- ◆ Let us count levels from 0. This way level k contains nodes which have depth k .

How many nodes at level k

◆ Claim: level k contains at most 2^k nodes.

How many nodes at level k

- ◆ Claim: level k contains at most 2^k nodes.
- ◆ Proof: by induction on k .

How many nodes at level k

- ◆ Claim: level k contains at most 2^k nodes.
- ◆ Proof: by induction on k .
 - (basis of induction) if $k = 0$, the claim is true:
 $2^0 = 1$, and we only have one node (root) at level 0.

How many nodes at level k

- ◆ Claim: level k contains at most 2^k nodes.
- ◆ Proof: by induction on k.
 - (basis of induction) if $k = 0$, the claim is true:
 $2^0 = 1$, and we only have one node (root) at level 0.
 - (inductive step): suppose the claim is true for $k-1$:
level $k-1$ contains at most 2^{k-1} nodes.

How many nodes at level k

- ◆ Claim: level k contains at most 2^k nodes.
- ◆ Proof: by induction on k.
 - (basis of induction) if $k = 0$, the claim is true:
 $2^0 = 1$, and we only have one node (root) at level 0.
 - (inductive step): suppose the claim is true for $k-1$:
level $k-1$ contains at most 2^{k-1} nodes.
 - We need to prove that then the claim holds for k :
level k holds at most 2^k nodes.

How many nodes at level k

- ◆ Claim: level k contains at most 2^k nodes.
 - ◆ Proof: by induction on k .
 - (basis of induction) if $k = 0$, the claim is true:
 $2^0 = 1$, and we only have one node (root) at level 0.
 - (inductive step): suppose the claim is true for $k-1$:
level $k-1$ contains at most 2^{k-1} nodes.
- Since each node at level $k-1$ has 0 or 2 children,
there are at most twice as many nodes at level k .

How many nodes at level k

◆ Claim: level k contains at most 2^k nodes.

◆ Proof: by induction on k .

- (basis of induction) if $k = 0$, the claim is true:

$2^0 = 1$, and we only have one node (root) at level 0.

- (inductive step): suppose the claim is true for $k-1$:
level $k-1$ contains at most 2^{k-1} nodes.

Since each node at level $k-1$ has 0 or 2 children,
there are at most twice as many nodes at level k .

So, level k contains at most $2 * 2^{k-1} = 2^k$ nodes.

How many nodes in a tree of height h ?

证明过程：

归纳基础：当 $h = 0$ 时，树只有一个节点，即根节点。此时，节点数为 $2^1 - 1 = 1$ ，符合命题。

归纳步骤：假设一个高度为 $h-1$ 的树最多有 $2^h - 1$ 个节点。一个高度为 h 的树比高度为 $h-1$ 的树多出一层，这一层最多有 2^h 个节点。因此，树的总节点数最多为： $2^h - 1 + 2^h = 2 \times 2^h - 1 = 2^{h+1} - 1$ 这样就得出了最终的结论。

Theorem: A proper binary tree of height h contains at most $2^{h+1} - 1$ nodes.

Proof: by induction on h

- (basis of induction): $h=0$. The tree contains at most $2^1 - 1 = 1$ node.
- (inductive step): assume a tree of height $h-1$ contains at most $2^h - 1$ nodes. A tree of depth h has one more level (h) which contains at most 2^h nodes. The total number of nodes in the tree of height h is at most:

$$2^h - 1 + 2^h = 2 \times 2^h - 1 = 2^{h+1} - 1.$$

What is the height of a tree of size n (with n nodes)?

We know that $n \leq 2^{h+1} - 1$.

So $2^{h+1} \geq n + 1$.

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

$h \geq \log_2(n+1) - 1$.

Overview of Contents

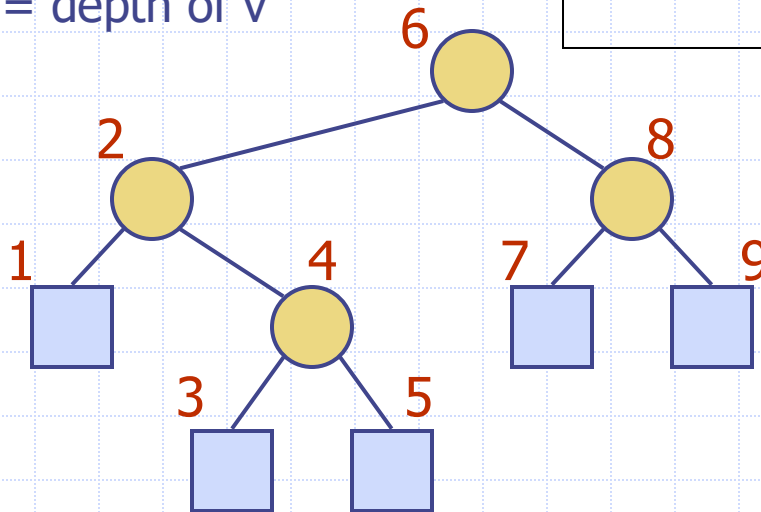
1. Tree definitions and tree ADT
2. Tree traversal algorithms
3. Binary tree definitions and binary tree ADT
4. **Binary tree traversal algorithms**
5. Implementation of tree and binary tree ADTs using concrete data structures

Inorder Traversal

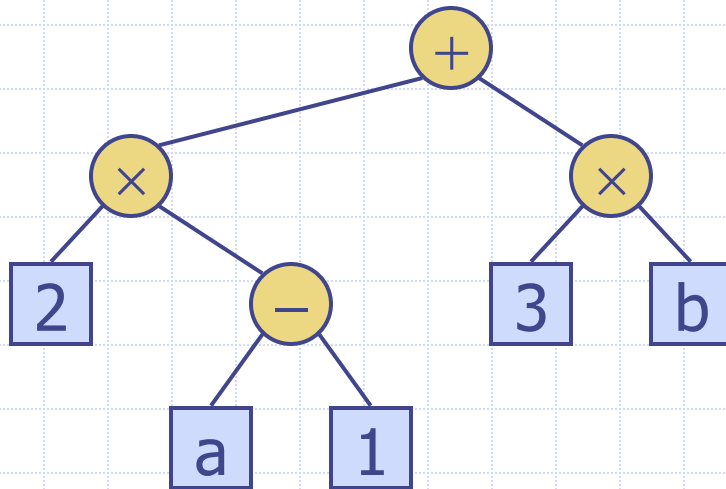
在中序遍历中，一个节点在其左子树之后，右子树之前被访问。

- 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( $v$ )  
  if  $left(v) \neq null$   
    inOrder( $left(v)$ )  
  visit( $v$ )  
  if  $right(v) \neq null$   
    inOrder( $right(v)$ )
```



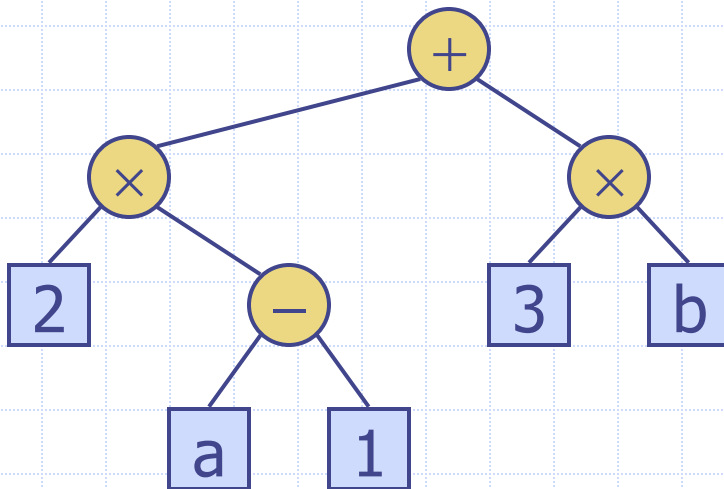
Print Arithmetic Expressions



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

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 *left(v) ≠ null*
 print (“(” ’ ’)

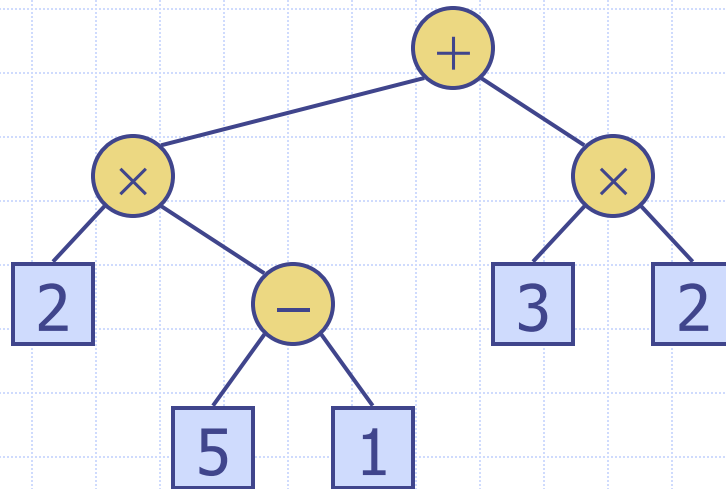
inOrder (*left(v)*)

print (*v.element* ())

if *right(v) ≠ null*
 inOrder (*right(v)*)
 print (“)” ’ ’)

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

Evaluate Arithmetic Expressions



$$\begin{aligned} & 2 \times (5 - 1) + 3 \times 2 \\ &= 2 \times 4 + 6 \\ &= 8 + 6 \\ &= 14 \end{aligned}$$

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* ()

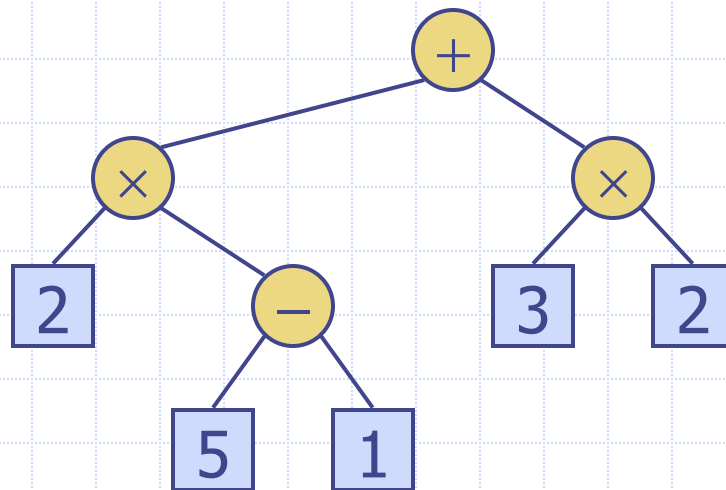
else

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

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

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

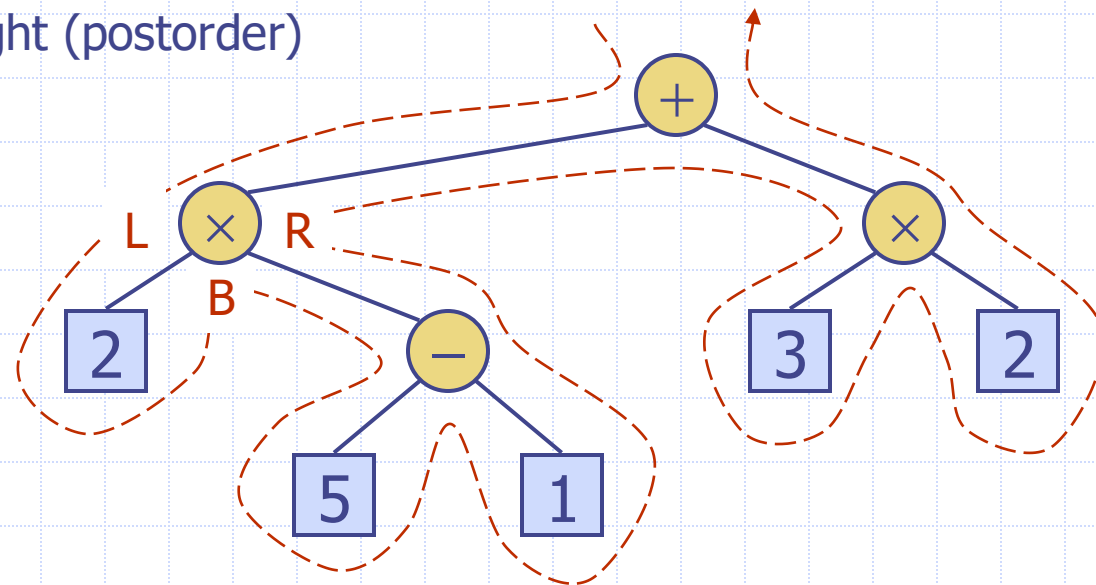
return $x \diamond y$



欧拉遍历 (Euler Tour Traversal)
定义：这是一种二叉树的遍历方式。
包括前序遍历、后序遍历和中序遍历。
遍历方式：遍历整个树，并在以下三个位置访问每个节点：
在左子树时（前序遍历）。
从下方访问节点（中序遍历）。
在右子树时（后序遍历）。
图中显示了一个树，标出了节点访问的路径顺序。

Euler Tour Traversal

- Generic traversal of a binary tree
- Includes 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)



Overview of Contents

1. Tree definitions and tree ADT
2. Tree traversal algorithms
3. Binary tree definitions and binary tree ADT
4. Binary tree traversal algorithms
5. **Implementation of tree and binary tree ADTs using concrete data structures**

节点表示：每个节点由一个对象表示，该对象包含：

元素：节点存储的数据。

父节点：指向父节点的引用。

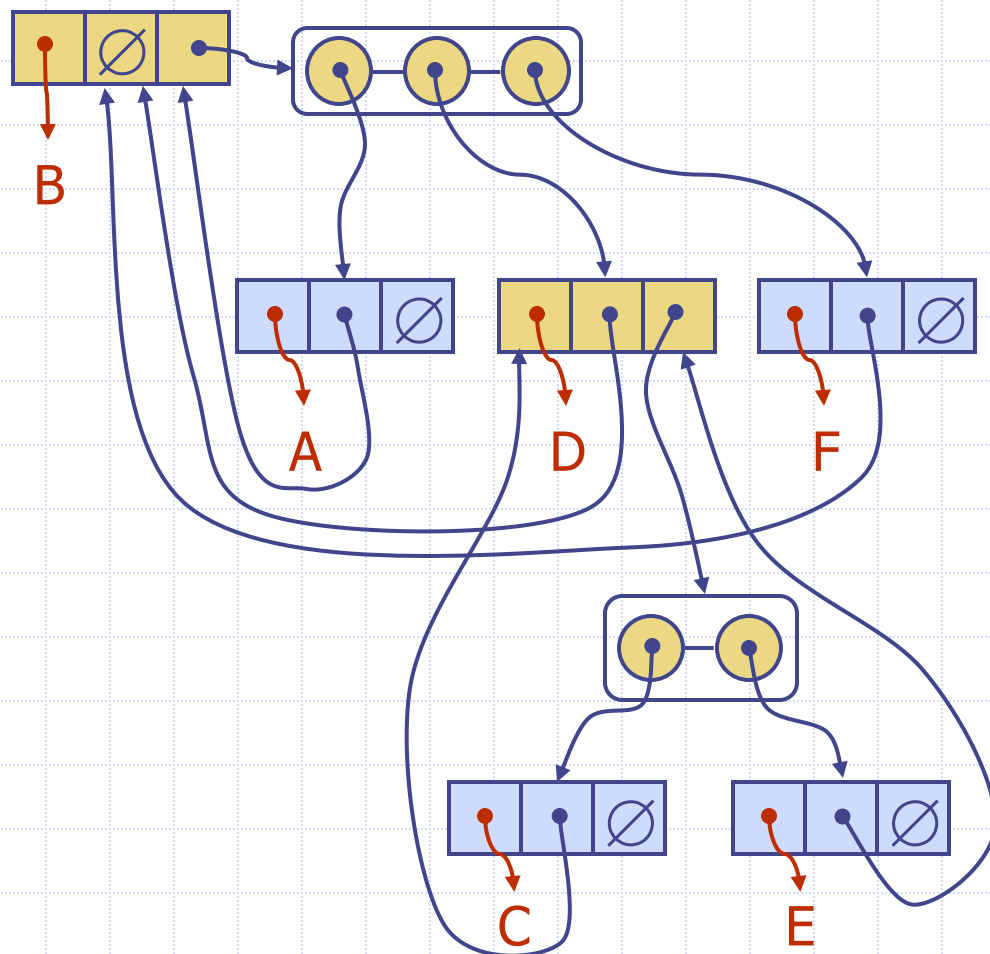
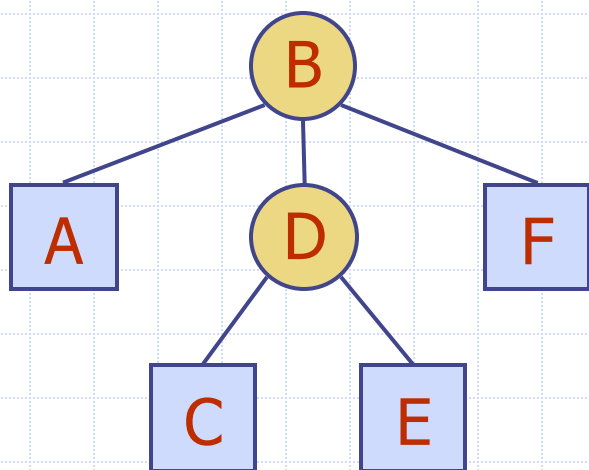
子节点的序列：指向该节点所有子节点的引用。

节点对象实现位置抽象（Position ADT）。

Linked Structure for Trees

右边图示展示了如何通过链接结构表示树。每个节点对象存储自身数据，并通过引用连接到它的父节点和子节点。

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

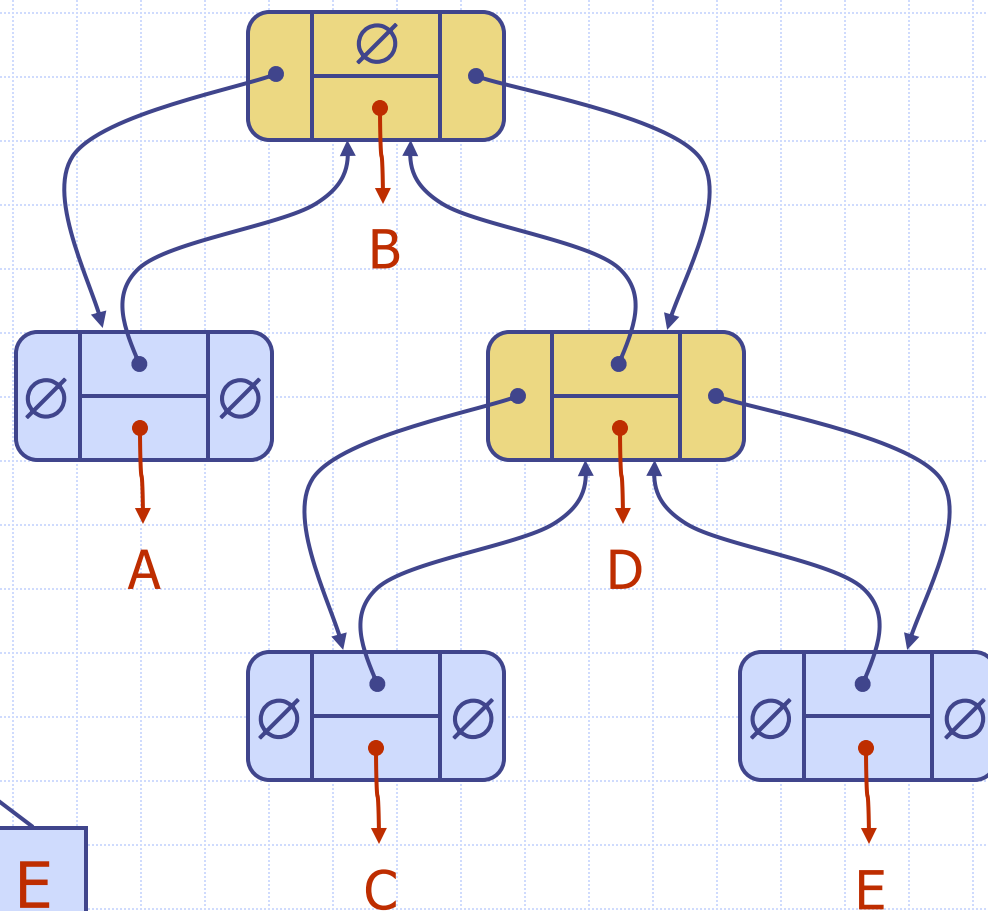
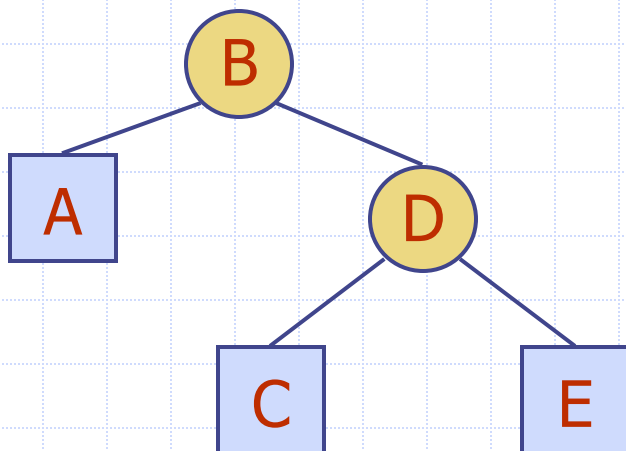


每个节点表示为一个对象，该对象包含：
元素：存储节点的数据。
父节点：指向父节点的引用。
左子节点：指向左子节点的引用。
右子节点：指向右子节点的引用。

Linked Structure for Binary Trees

节点对象实现了位置抽象（Position ADT）。图中展示了如何通过链接结构表示一个二叉树。

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



Array-Based Representation of Binary Trees

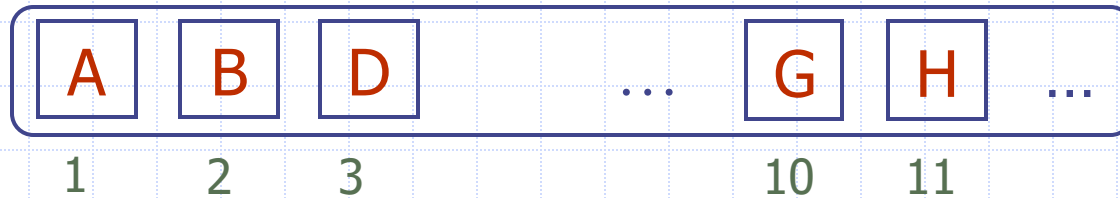
节点存储在数组 A 中，例如节点 A、B、D、G、H 等被依次存储在数组中。
节点 v 存储在 $A[\text{rank}(v)]$ 中。

根节点的 $\text{rank}(\text{root}) = 1$ 。

如果节点是父节点的左子节点， $\text{rank}(\text{node}) = 2 \cdot \text{rank}(\text{parent}(\text{node}))$ 。

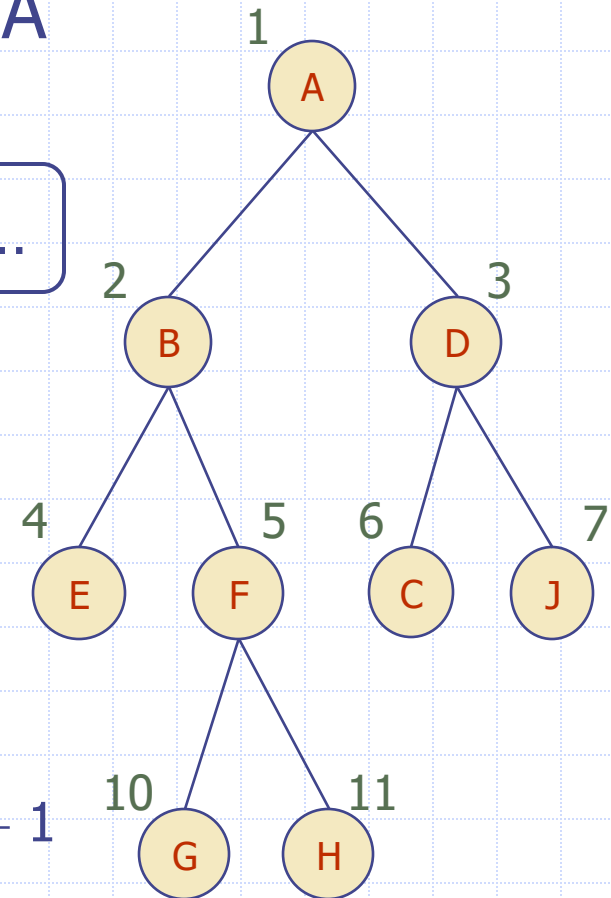
如果节点是父节点的右子节点， $\text{rank}(\text{node}) = 2 \cdot \text{rank}(\text{parent}(\text{node})) + 1$ 。

- Nodes are stored in an array A



Node v is stored at $A[\text{rank}(v)]$

- $\text{rank}(\text{root}) = 1$
- if node is the left child of $\text{parent}(\text{node})$,
 $\text{rank}(\text{node}) = 2 \cdot \text{rank}(\text{parent}(\text{node}))$
- if node is the right child of $\text{parent}(\text{node})$,
 $\text{rank}(\text{node}) = 2 \cdot \text{rank}(\text{parent}(\text{node})) + 1$



Abstract Data Types vs. Concrete Data Structures

Abstract Data Type (ADT)	Concrete Data Structure
Stack	Array
Queue	Singly Linked List
List	Doubly Linked List
Positional List	Linked Structure
Tree	
Binary Tree	

Lab Exercise

- ❑ Implement the tree ADT in Java and analyze the *complexity* of implemented methods.
- ❑ Implement the binary tree ADT in Java and analyze the *complexity* of implemented methods.

斐波那契数列的递归定义：

$F_0 = 0$

$F_1 = 1$

$F_i = F_{i-1} + F_{i-2}$ 对于 $i > 1$

Computing Fibonacci Numbers

- Fibonacci numbers are defined recursively:

$$F_0 = 0$$

$$F_1 = 1$$

$$F_i = F_{i-1} + F_{i-2} \quad \text{for } i > 1.$$

- Design an algorithm for calculating Fibonacci numbers, and analyze its complexity

Algorithm BinaryFib(k):

Input: Nonnegative integer k

Output: The k th Fibonacci number F_k

- How to visualize the process of calculation of Fibonacci Numbers?

Reading

M. T. Goodrich, R. Tamassia and M. H. Goldwasser,
Data Structures and Algorithms in Java, 6th Edition,
2014.

- Chapter 8. Tree Structures