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

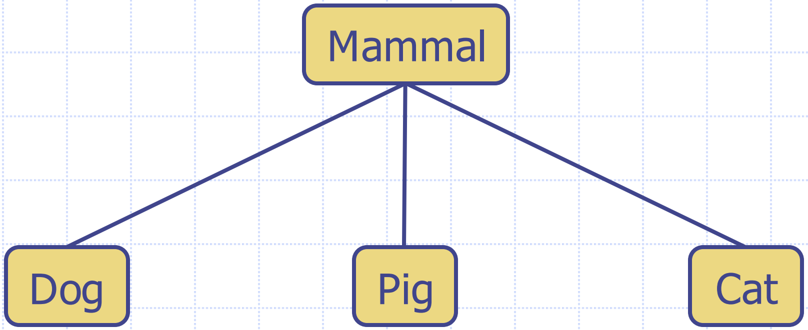
Abstract Data Type (ADT)	Concrete Data Structure
Stack	Array
Queue	Singly Linked List
List	Doubly Linked List
Positional List	

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

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

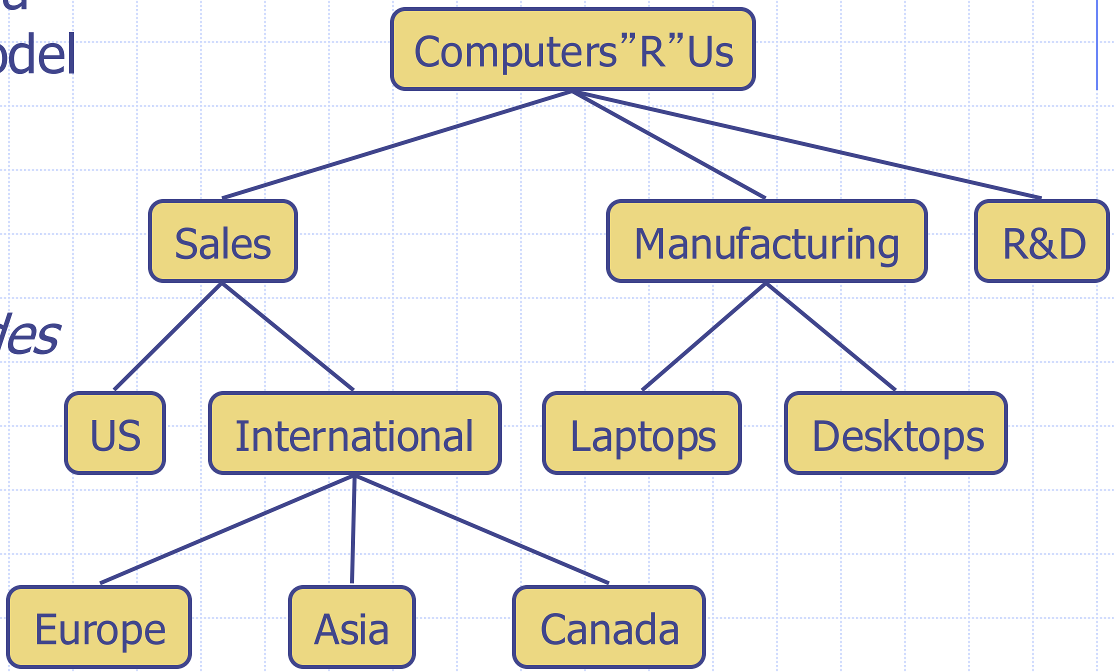
1. Tree definitions and tree ADT
2. Tree traversal algorithms
3. Binary tree definitions and binary tree ADT
4. Binary tree traversal algorithms
5. Implementation of tree and binary tree ADTs using concrete data structures

Overview of Contents

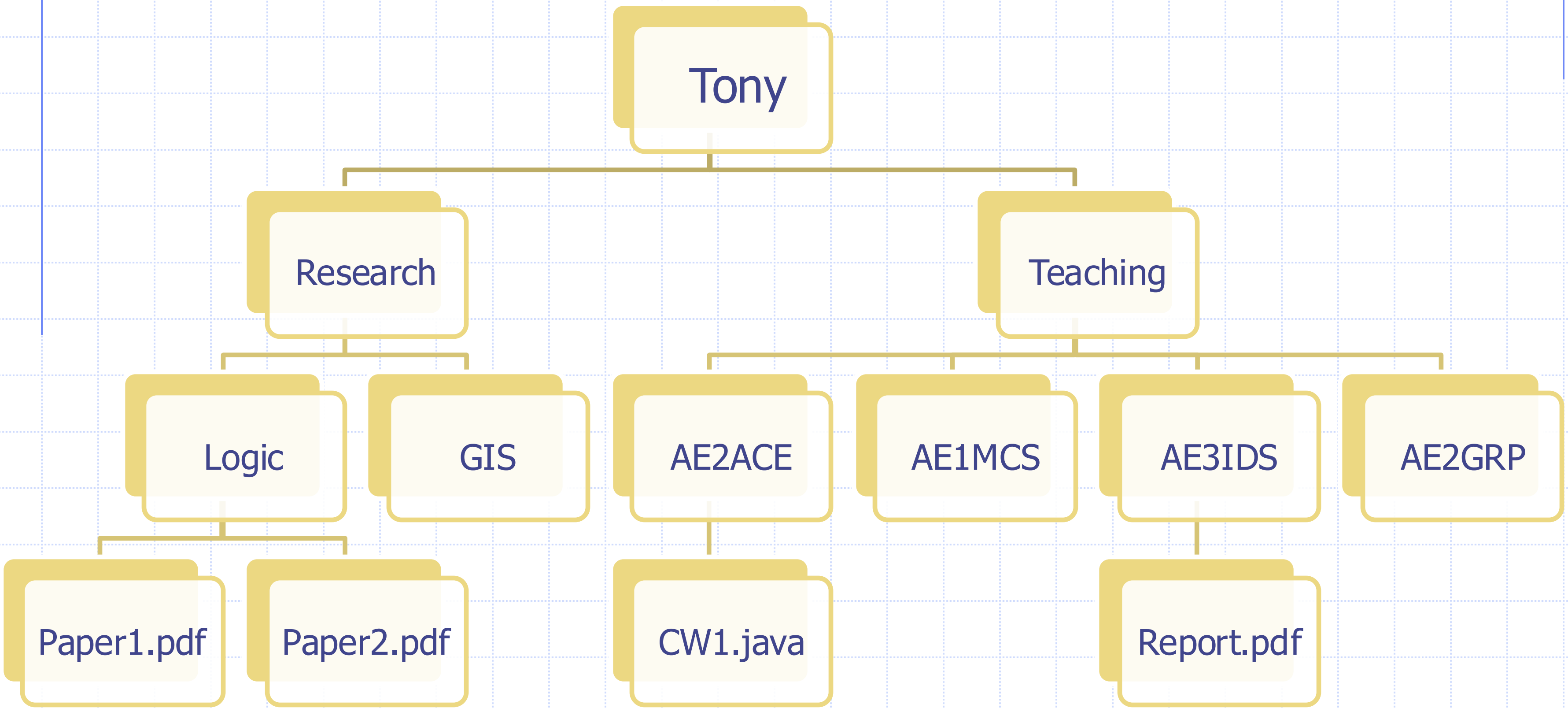
1. **Tree definitions and tree ADT**
2. Tree traversal algorithms
3. Binary tree definitions and binary tree ADT
4. Binary tree traversal algorithms
5. Implementation of tree and binary tree ADTs using concrete data structures

What is a Tree

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

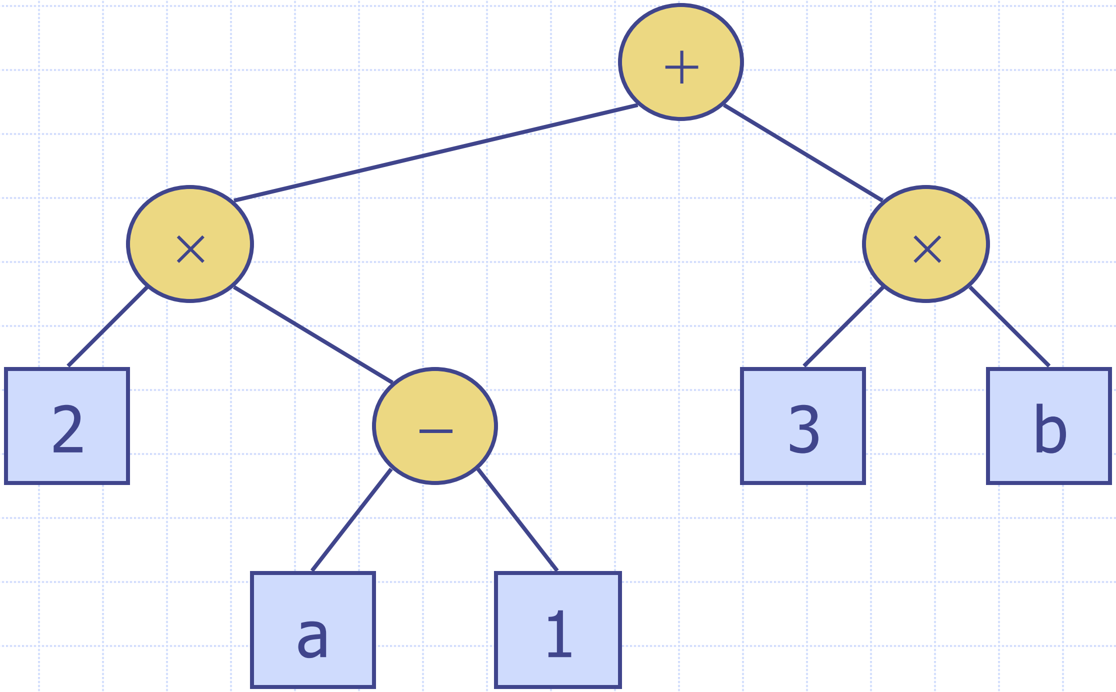


File System


Tony
Research
Teaching
Logic
GIS
AE2ACE
AE1MCS
AE3IDS
AE2GRP
Paper1.pdf
Paper2.pdf
CW1.java
Report.pdf

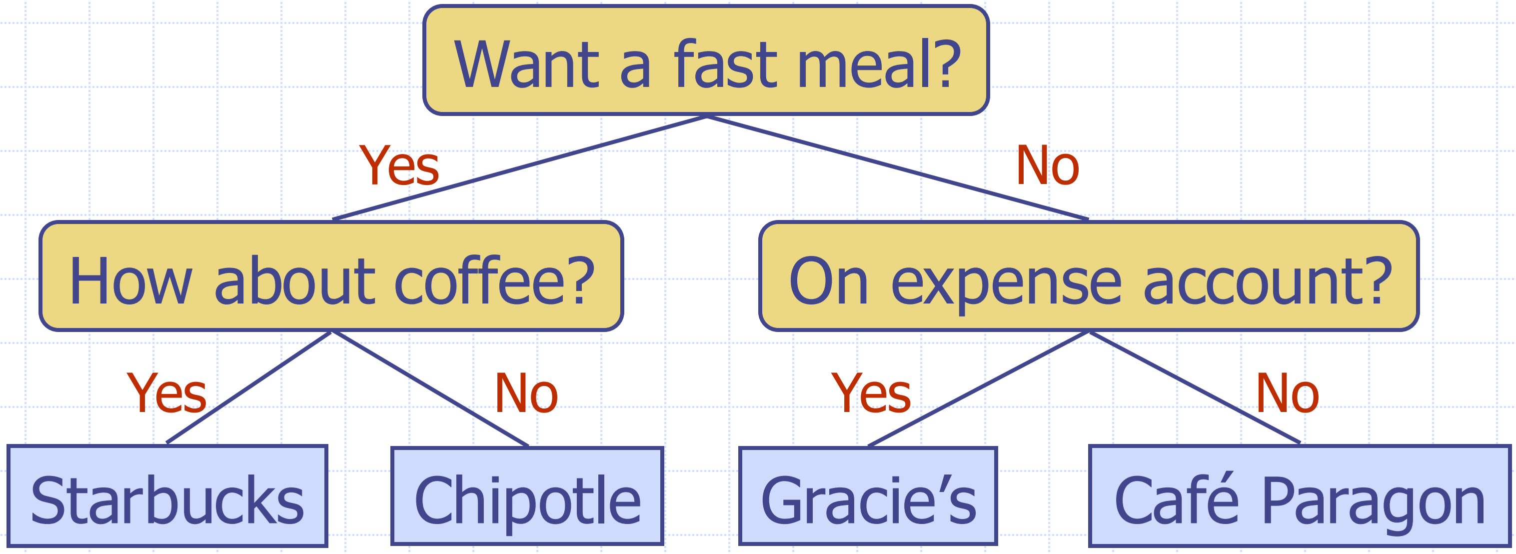
Arithmetic Expression Tree

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



Decision Tree

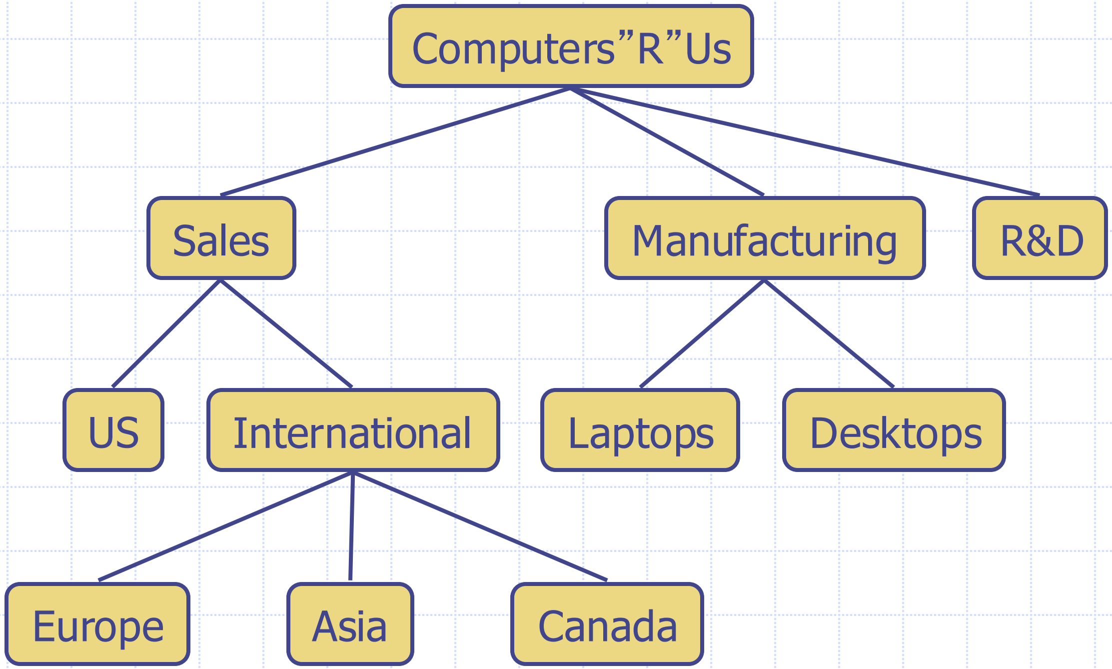
- Example: dining decision



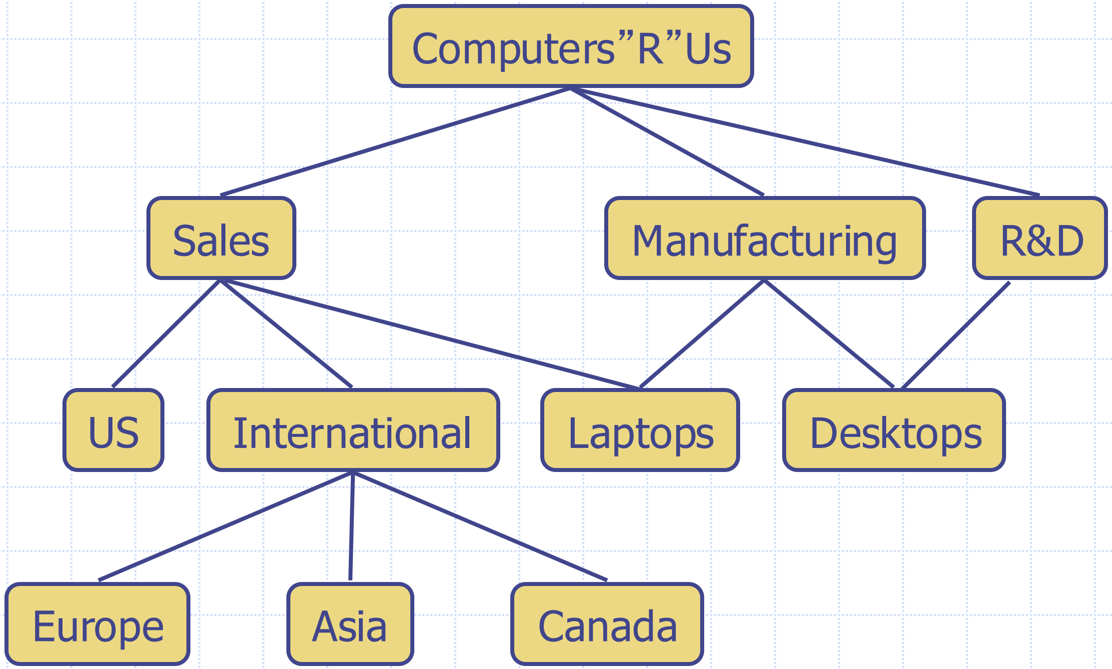
What is a tree?

- 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?

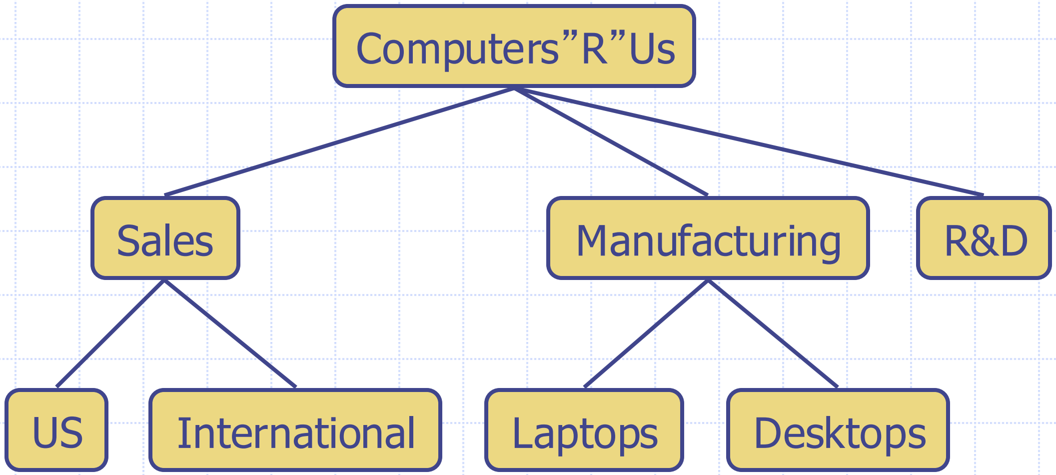
Is this a tree?



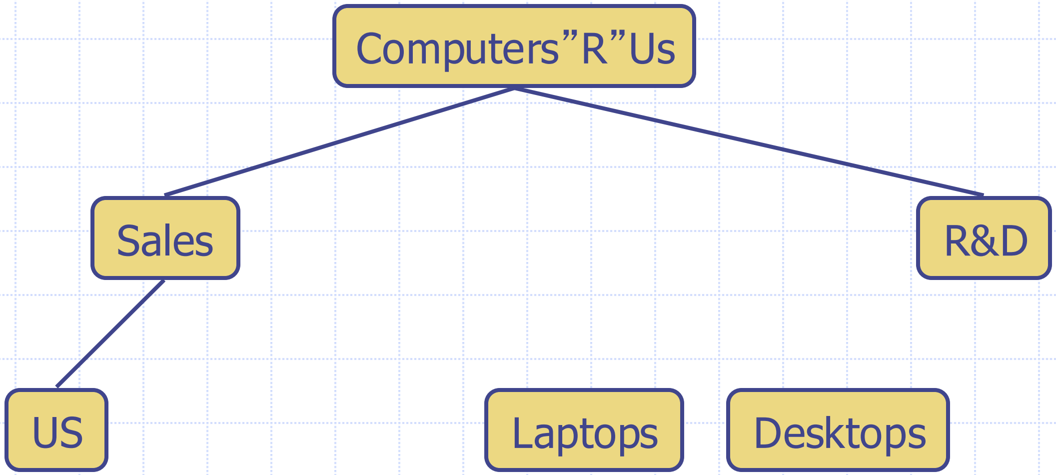
Is this a tree?



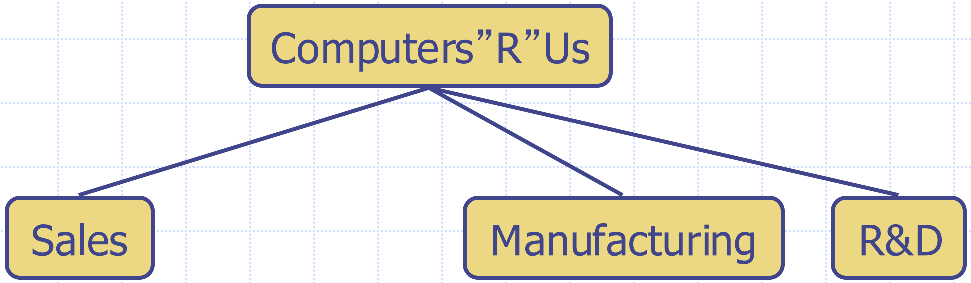
Is this a tree?



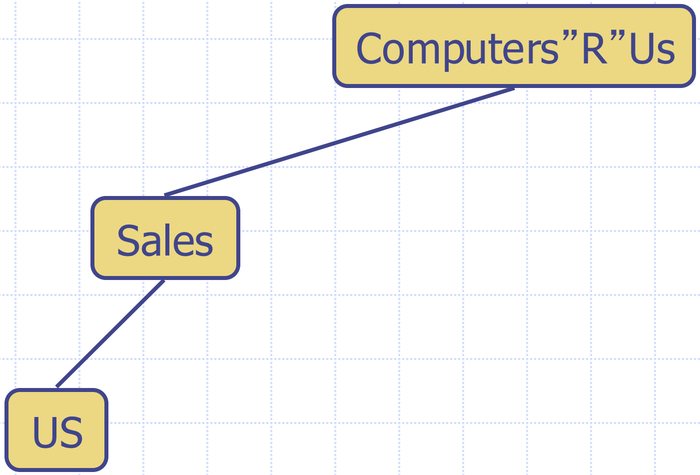
Is this a tree?



Is this a tree?



Is this a tree?



Is this a tree?

What is a tree?

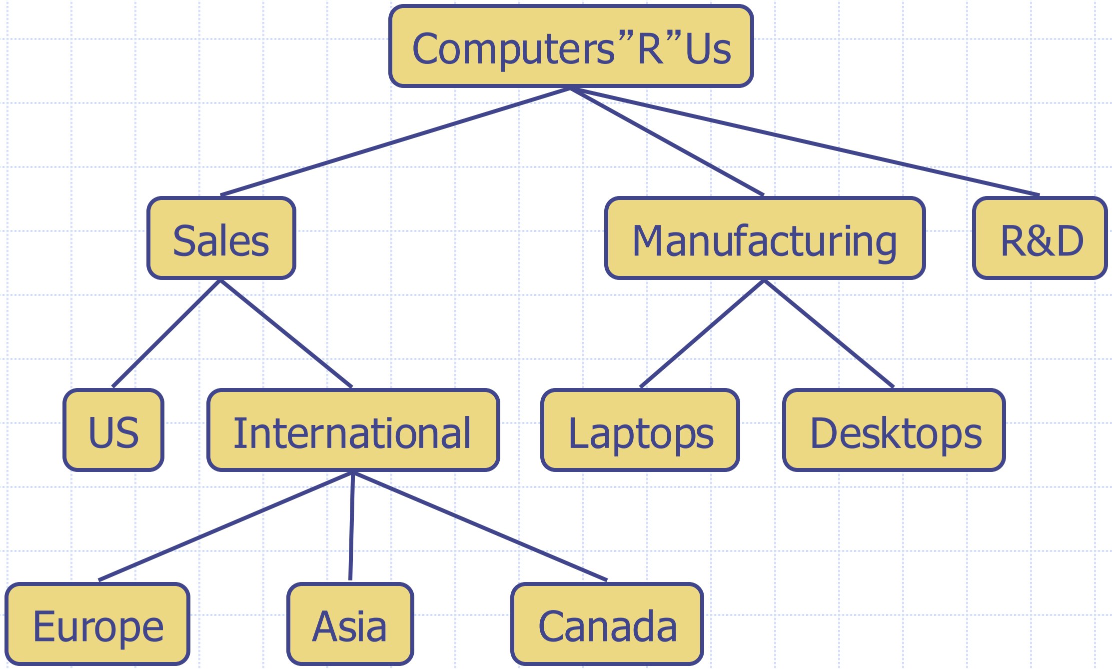
- A tree can be defined regarding to properties of its nodes.
- What property does a node in a tree have?
- Is there any special node in a tree?
- Write down a definition of a tree in your own words.

Tree

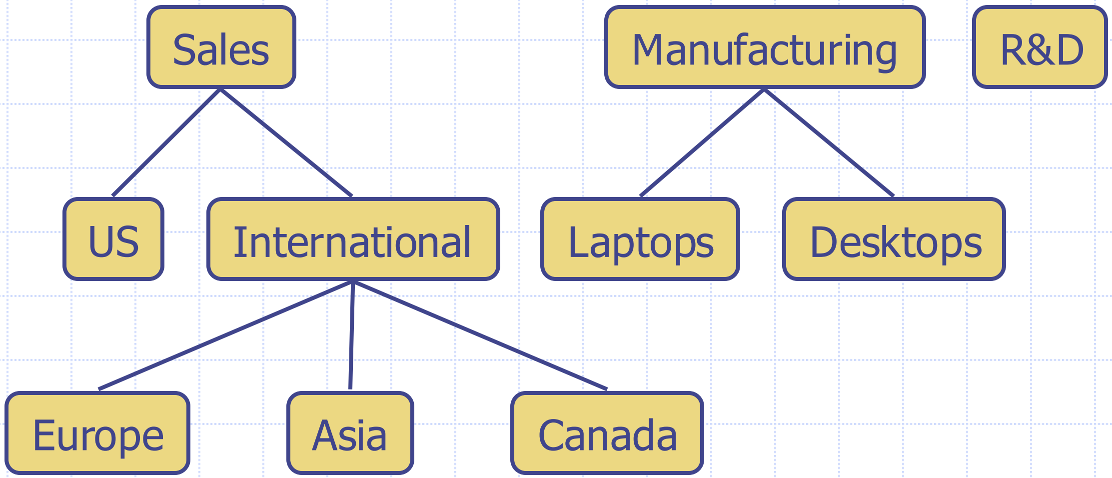
A ***tree*** *T* is defined as a set of ***nodes*** storing elements such that the nodes have a ***parent-child*** relationship that satisfies the following properties:

- 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*.

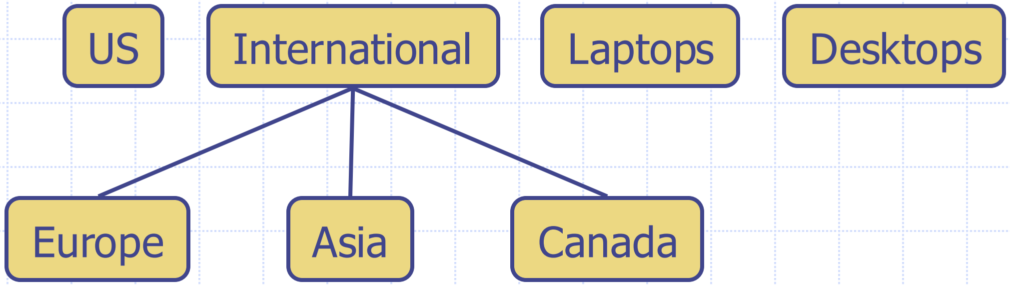
Is this a tree?



Are they trees?



Are they trees?



Are they trees?

What is a tree?

- 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*.

Other Node Relationships

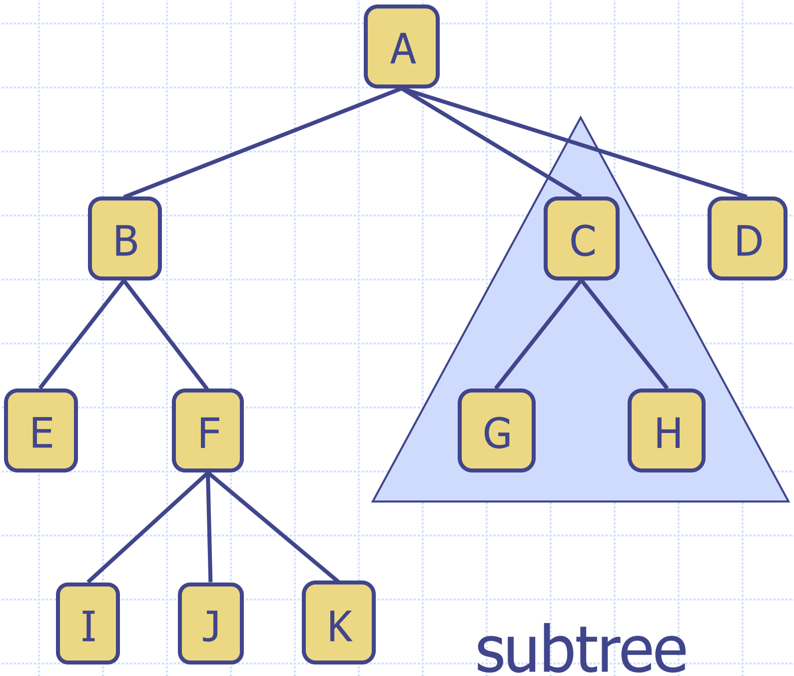
- Two nodes that are children of the same parent are ***siblings***.
- A node *v* is ***internal*** if it has one or more children.
- A node *v* is ***external*** if *v* has no children.
- External nodes are also known as ***leaves***.

Other Node Relationships

- A node u is an ***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 ***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).

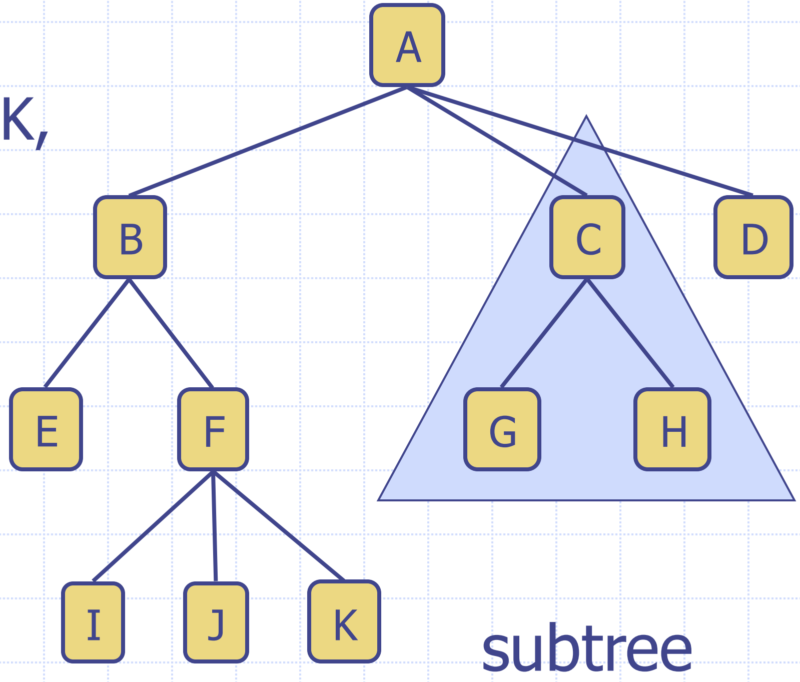
Tree Terminology

- Root: node without parent (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



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



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.

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

What is the depth of the root of T?

Recursive Definition of Depth

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

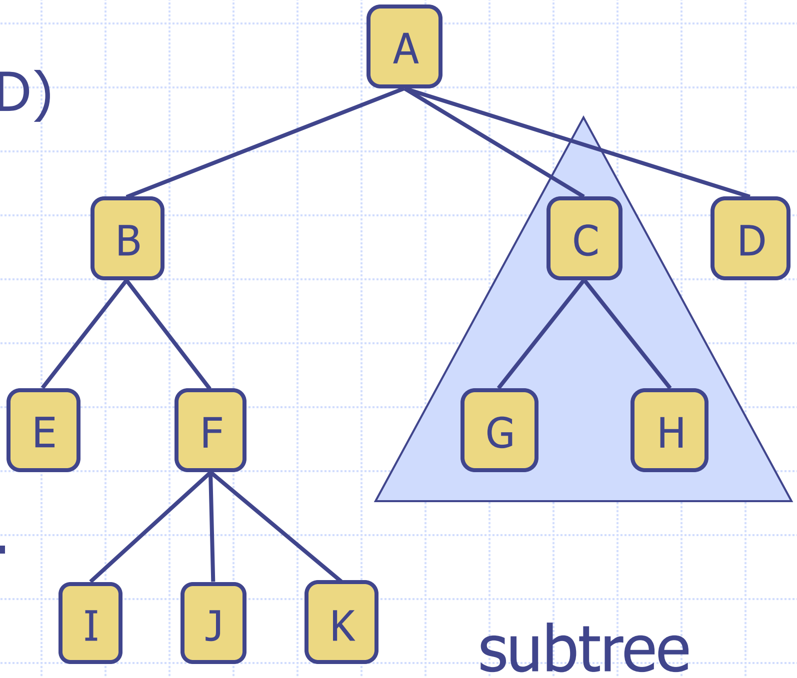
- If *p* is the root, then the depth of *p* is 0.
- Otherwise, the depth of *p* is one plus the depth of the parent of *p*.

Height

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

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



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?

- 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.

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?***

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.

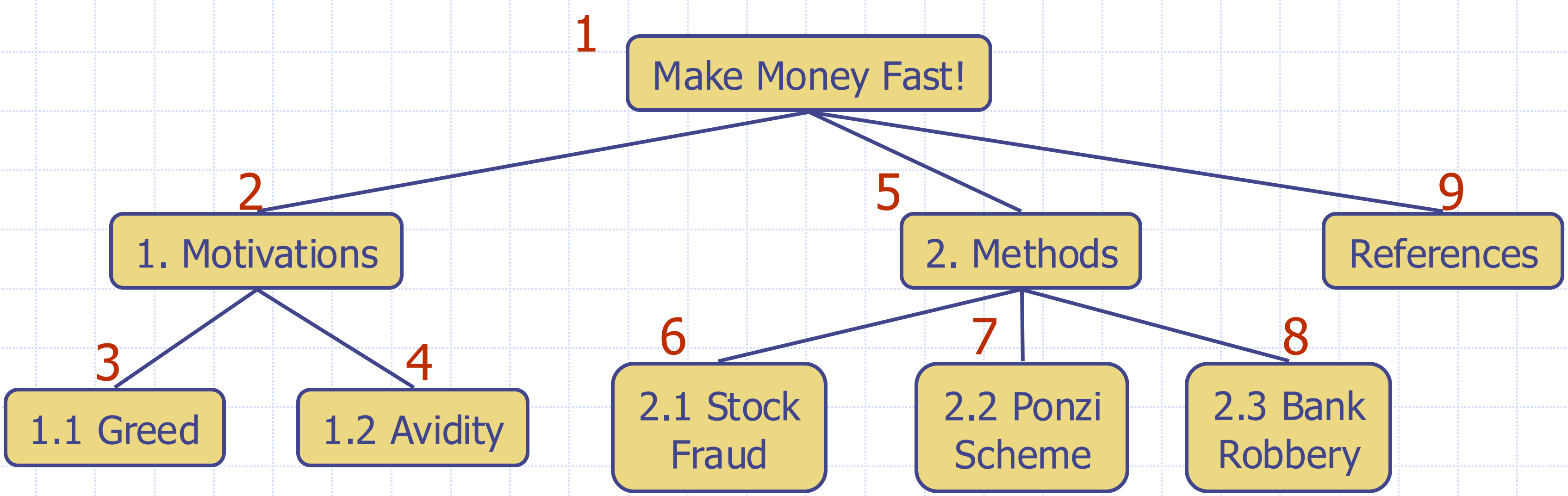
Overview of Contents

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

- 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

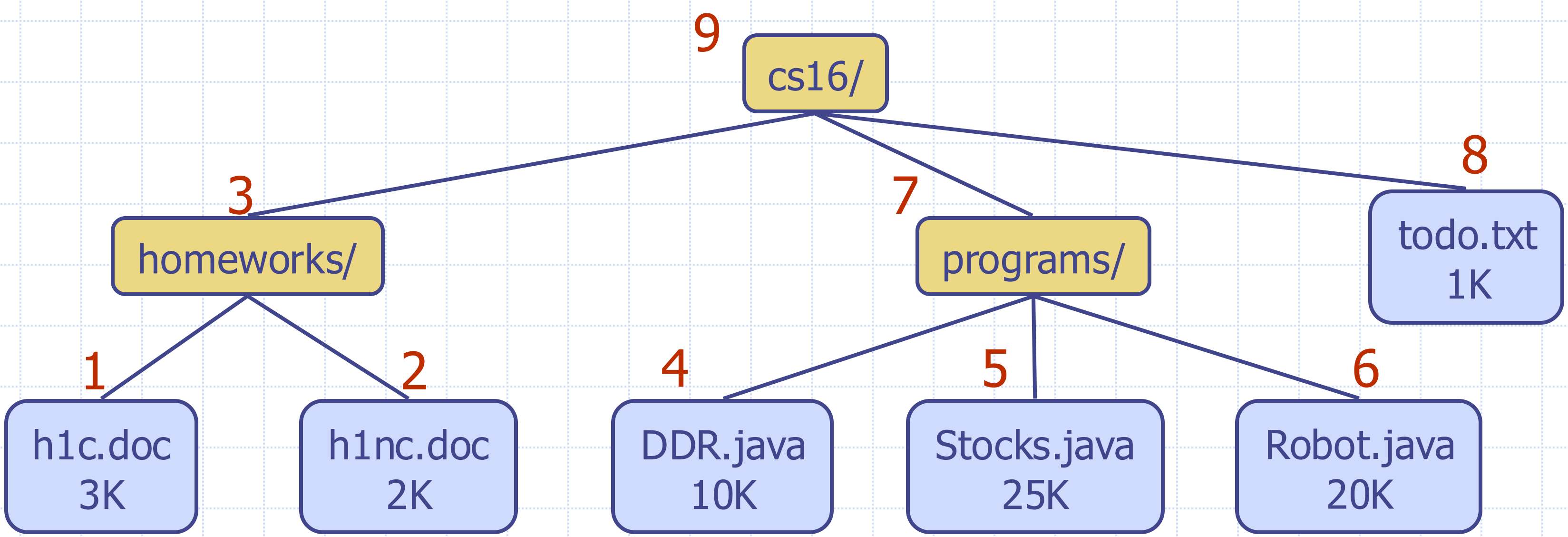
Algorithm ***preOrder***(*v*)
  visit(*v*)
  for each child *w* of *v*
    preorder (*w*)



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)
```



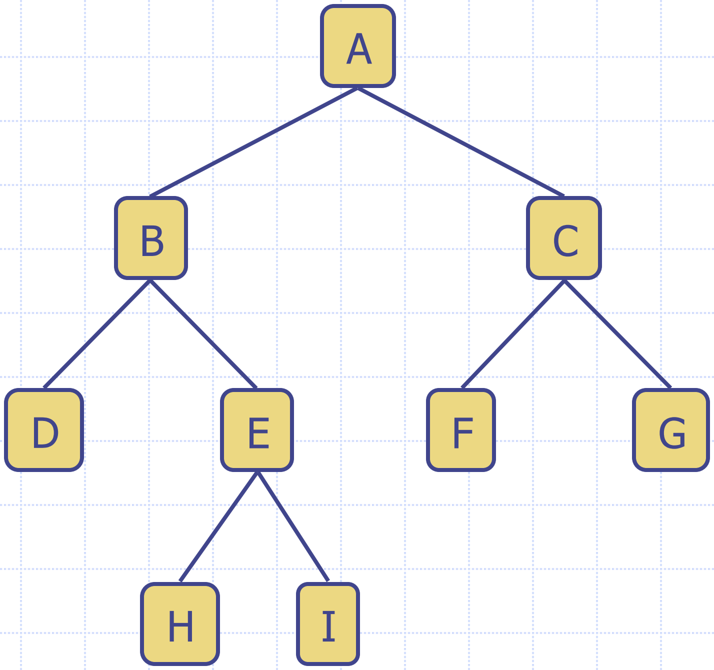
Overview of Contents

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

Binary Trees

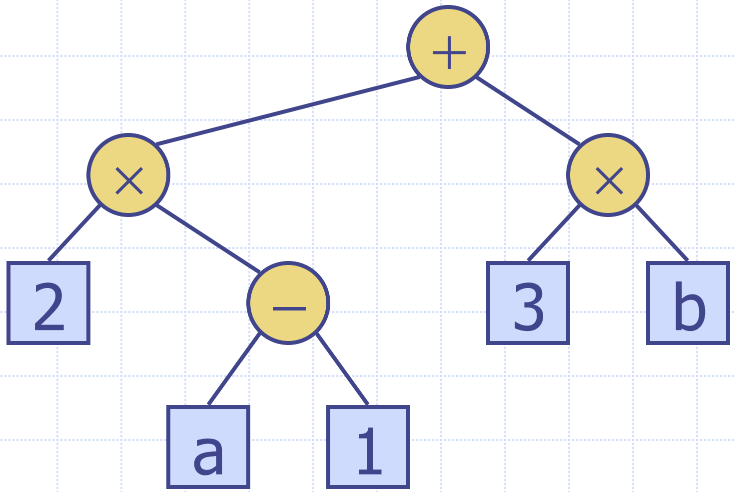
- A *binary* tree is a tree with the following properties:
  - Each internal node has at most two children (exactly two for proper binary trees)
  - The children of a node are an ordered pair
- We call the children of an internal node left child and right child

- Applications:
  - arithmetic expressions
  - decision processes
  - searching



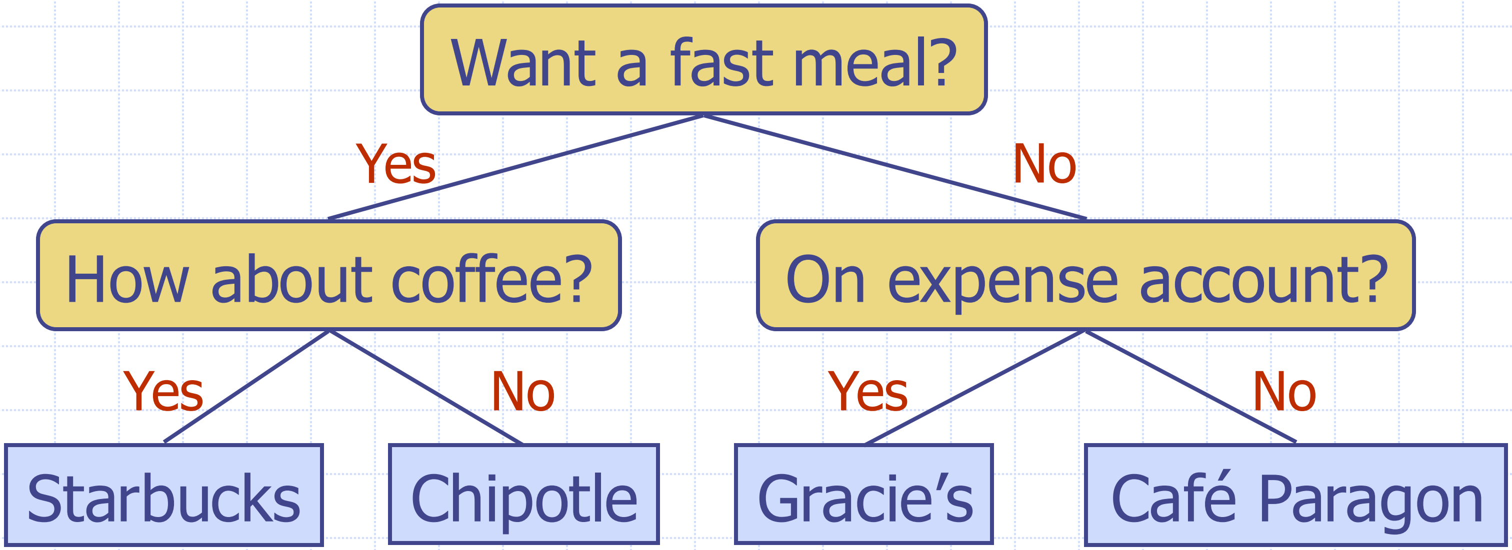
Arithmetic Expression Tree

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

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.

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.

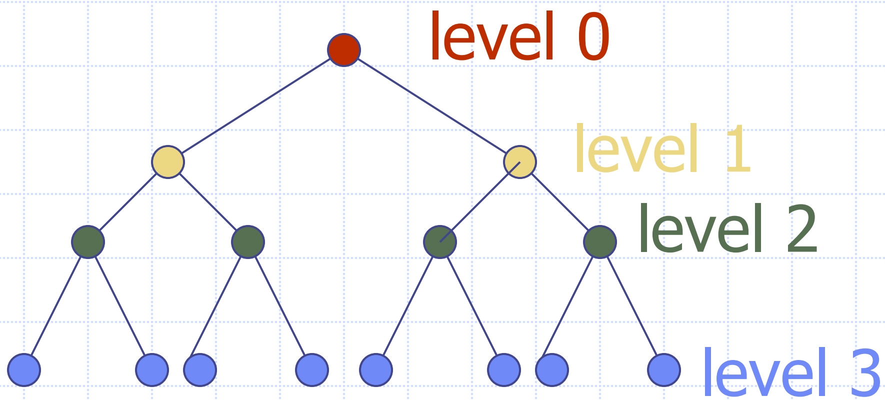
BinaryTree ADT

- 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.

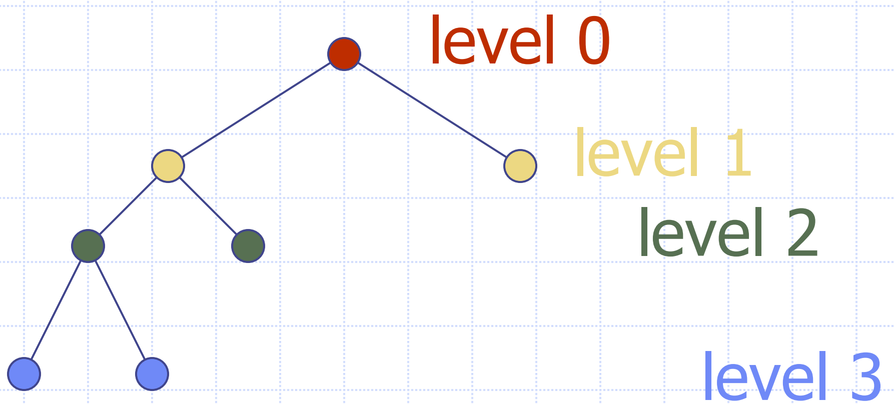
Proper binary tree of height 3:



Properties of Proper Binary Trees

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

Proper binary tree of height 3:



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

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

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

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?

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:

  $2^h - 1 + 2^h = 2 * 2^h - 1 = 2^{h+1} - 1.$

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

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

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

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

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

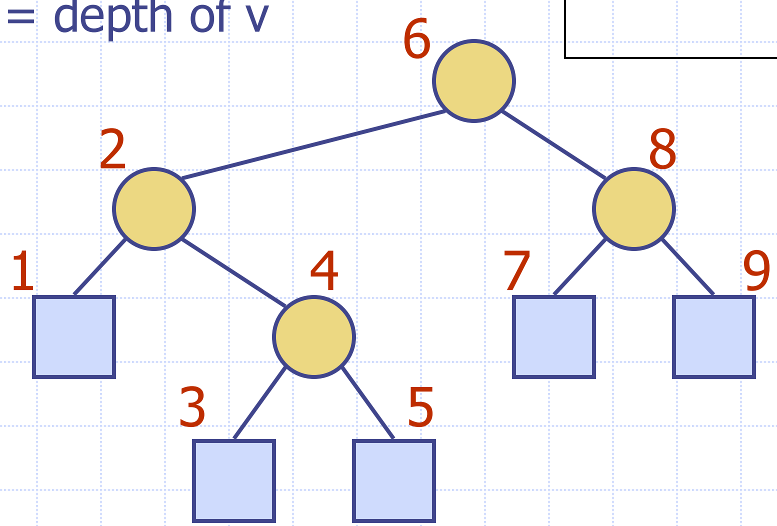
Overview of Contents

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

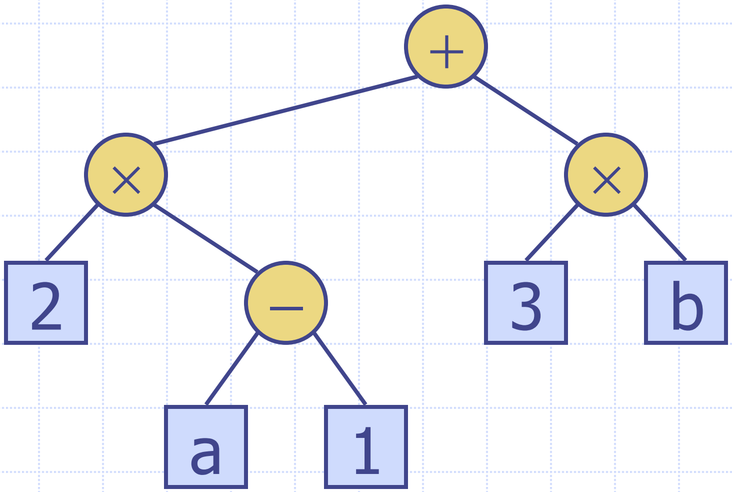
Inorder Traversal

- 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*) ≠ **null**
        inOrder (*left* (*v*))
    visit(*v*)
    if *right*(*v*) ≠ **null**
        inOrder (*right* (*v*))

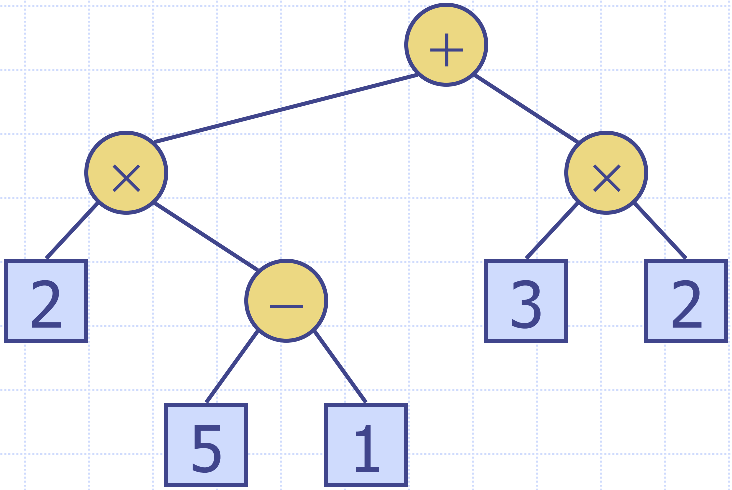


Print Arithmetic Expressions



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

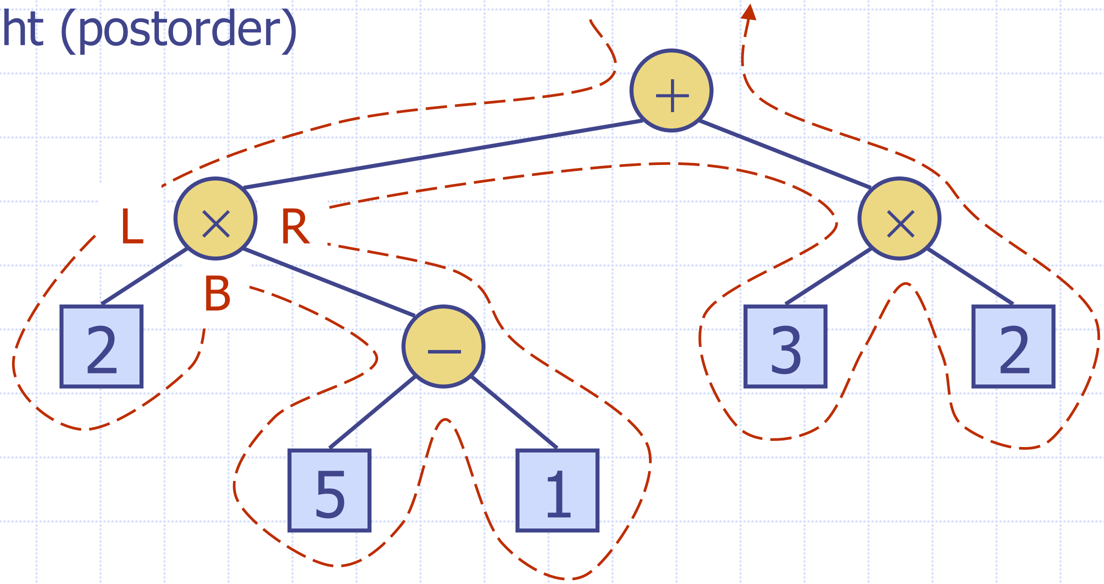
Evaluate Arithmetic Expressions



$$2 \times (5 - 1) + 3 \times 2$$
$$= 2 \times 4 + 6$$
$$= 8 + 6$$
$$= 14$$

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)

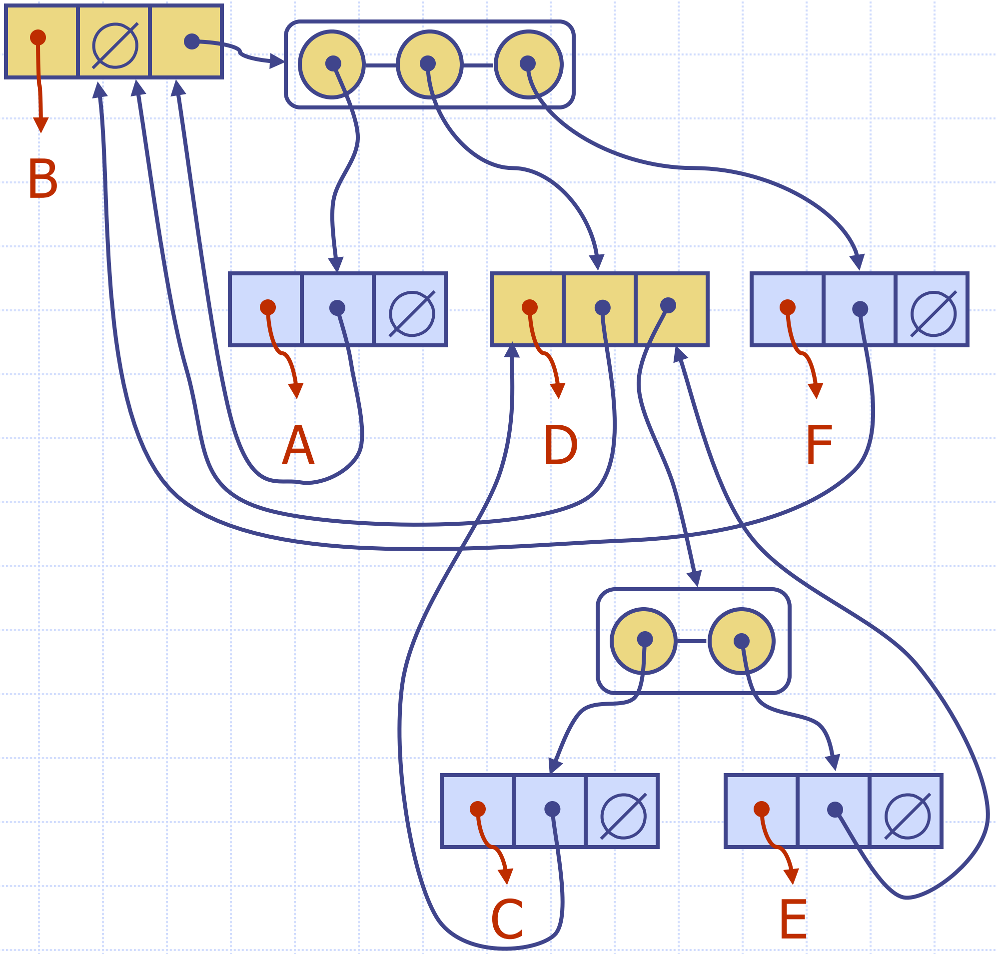


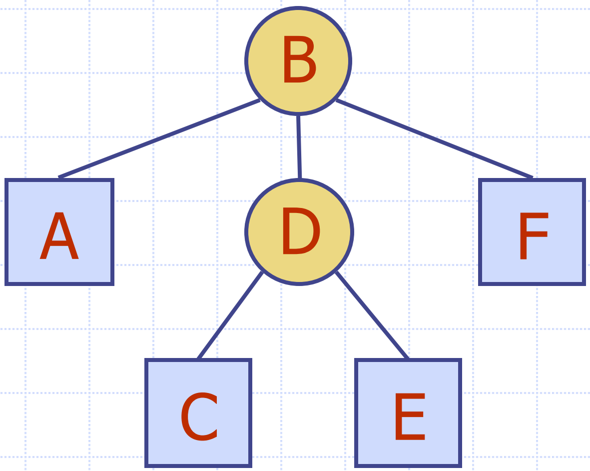
Overview of Contents

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**

Linked Structure for Trees

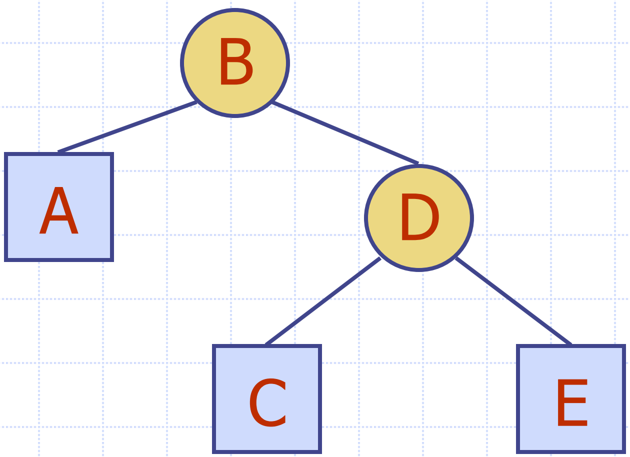
- A node is represented by an object storing
  - Element
  - Parent node
  - Sequence of children nodes
- Node objects implement the Position ADT

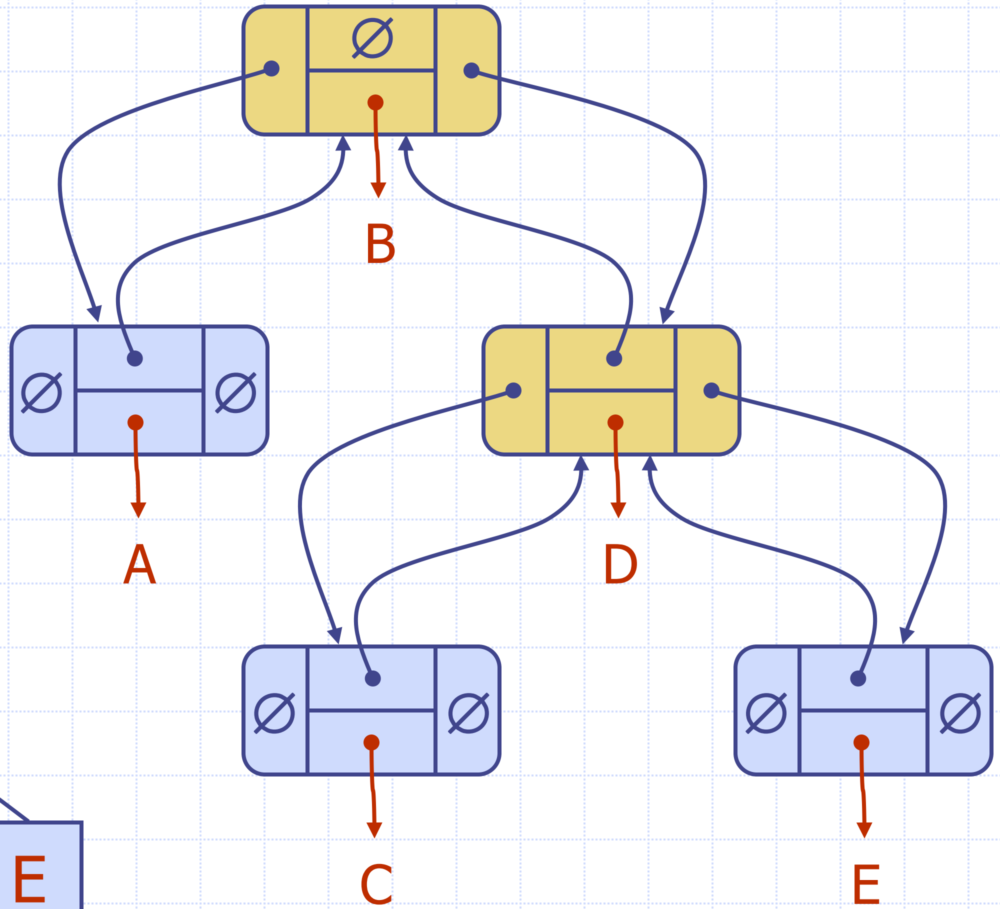




Linked Structure for Binary Trees

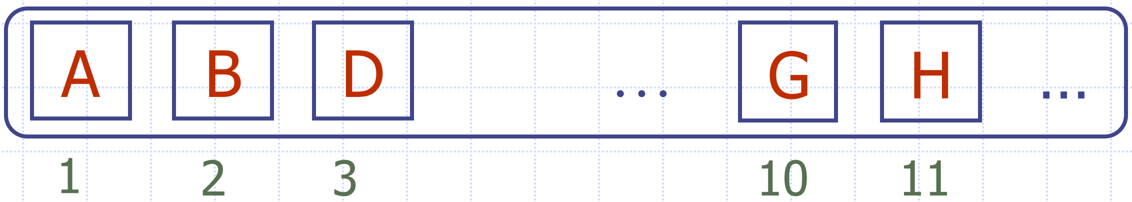
- A node is represented by an object storing
  - Element
  - Parent node
  - Left child node
  - Right child node
- Node objects implement the Position ADT

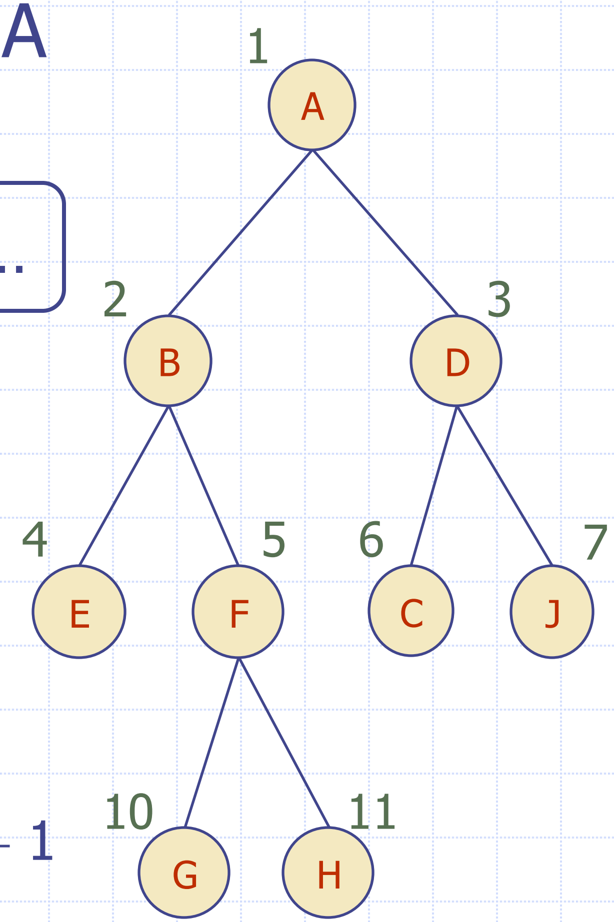




Array-Based Representation of Binary Trees

- 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

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.

Computing Fibonacci Numbers

- Fibonacci numbers are defined recursively:

  $F_0 = 0$

  $F_1 = 1$

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

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

  Algorithm BinaryFib(k):

  Input: Nonnegative integer k

  Output: The kth 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**