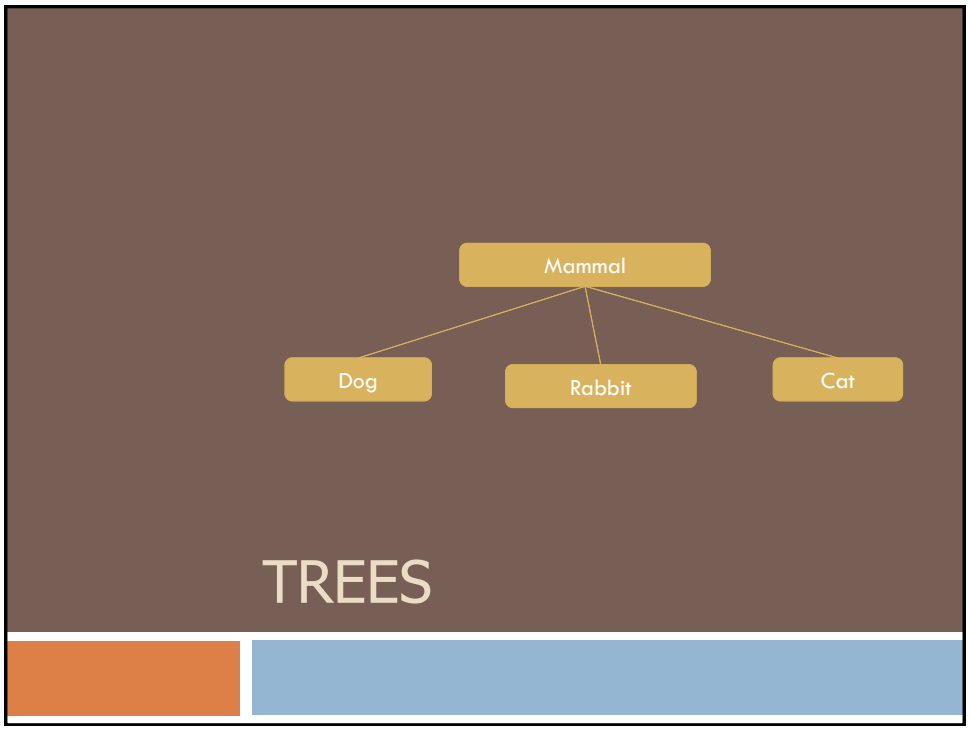
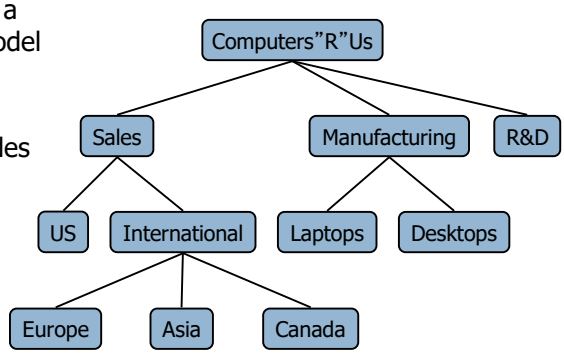


What is a Tree

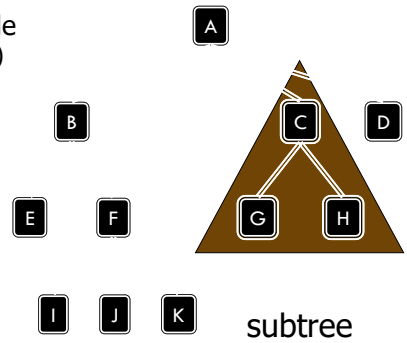
- In computer science, a tree is an abstract model of a hierarchical structure
- A tree consists of nodes with a **parent-child** relation
- Applications:
  - Organization charts
  - File systems
  - Programming environments



Tree Terminology

- **Root:** node without parent (A)
- **Internal node:** node with at least one child (A, B, C, F)
- **External node (a.k.a. leaf):** node without children (E, I, J, K, G, H, D)
- **Ancestors of a node:** parent, grandparent, grand-grandparent, etc.
- **Depth of a node:** number of ancestors
- **Height of a tree:** maximum depth of any node (3)
- **Descendant of a node:** child, grandchild, grand-grandchild, etc.

- Subtree: tree consisting of a node and its descendants

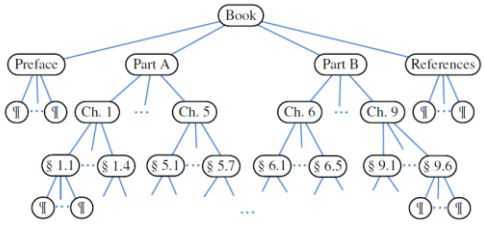


subtree

Tree Terminology

- **Sibling:** Two nodes that are children of the same parent.
- E***dge of tree T:*** *is a pair of nodes (u,v) such that u is the parent of v, or vice* versa.
- A ***path of T:*** *is a sequence of nodes such that any two consecutive nodes in* the sequence form an edge. (Ex: C,A,B,F,K)

- **Ordered Trees:** A tree is *ordered if there is a meaningful linear order among the children of each* node.



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 update methods may be defined by data structures implementing the Tree ADT

Java Interface

Methods for a Tree interface:

```
1  /** An interface for a tree where nodes can have an arbitrary number of children. */
2  public interface Tree<E> extends Iterable<E> {
3    Position<E> root( );
4    Position<E> parent(Position<E> p) throws IllegalArgumentException;
5    Iterable<Position<E>> children(Position<E> p)
6                                        throws IllegalArgumentException;
7    int numChildren(Position<E> p) throws IllegalArgumentException;
8    boolean isInternal(Position<E> p) throws IllegalArgumentException;
9    boolean isExternal(Position<E> p) throws IllegalArgumentException;
10   boolean isRoot(Position<E> p) throws IllegalArgumentException;
11   int size( );
12   boolean isEmpty( );
13   Iterator<E> iterator( );
14   Iterable<Position<E>> positions( );
15 }
```

Computing Depth

- Let *p be a position within tree T. The* ***depth of p is the number of ancestors of*** *p, other than p itself.*
- The depth of *p can 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.*
- The running time of depth(*p*) *for position p is* $O(d_p + 1)$, *where dp denotes* the depth of *p in the tree.*

```
/** Returns the number of levels separating Position p
from the root. */
public int depth(Position<E> p) {
if (isRoot(p))
    return 0;
else
    return 1 + depth(parent(p));
}
```

Computing Height

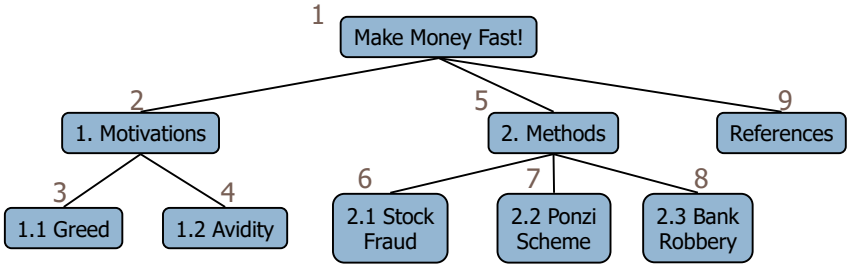
- The *height of a tree to be equal to the maximum of the depths of* its positions (or zero, if the tree is empty).

```
/** Returns the height of the tree. */
private int heightBad( ) { // works, but quadratic worst-case time
    int h = 0;
    for (Position<E> p : positions( ))
        if (isExternal(p)) // only consider leaf positions
            h = Math.max(h, depth(p));
    return h;
}
```

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

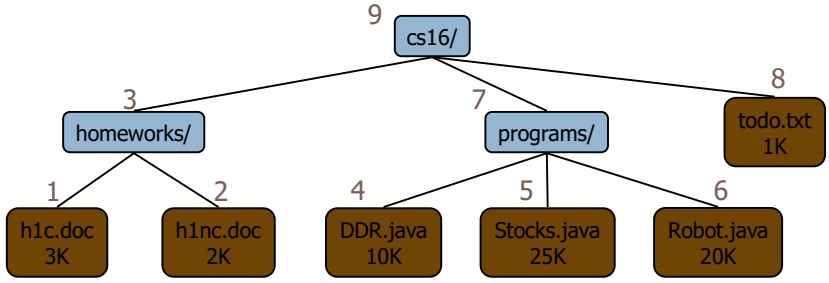
Algorithm *preOrder*(*v*)
  visit(*v*)
  for each child *w* of *v*
    preorder (*w*)



Postorder Traversal

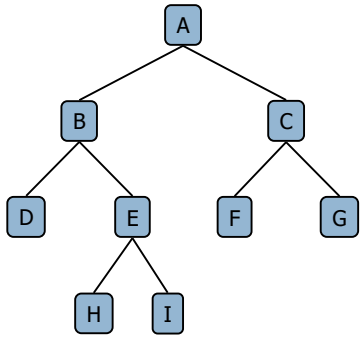
- In a postorder traversal, a node is visited after its descendants
- Application: compute space used by files in a directory and its subdirectories

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



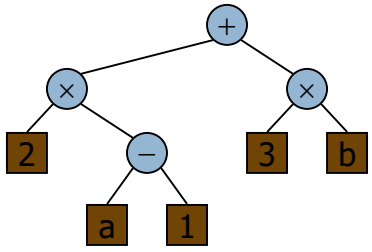
Binary Trees

- 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
- Alternative recursive definition: a binary tree is either
  - a tree consisting of a single node, or
  - a tree whose root has an ordered pair of children, each of which is a binary tree
- Applications:
  - arithmetic expressions
  - decision processes
  - searching



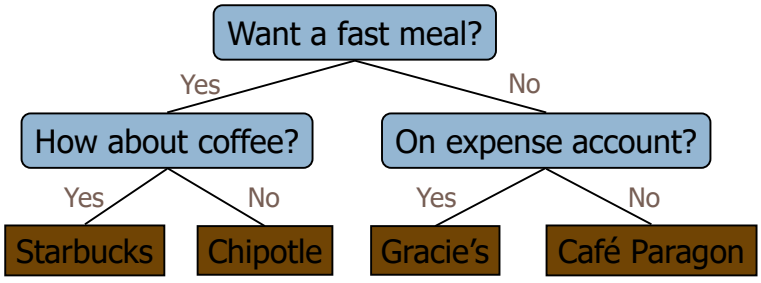
Arithmetic Expression Tree

- 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

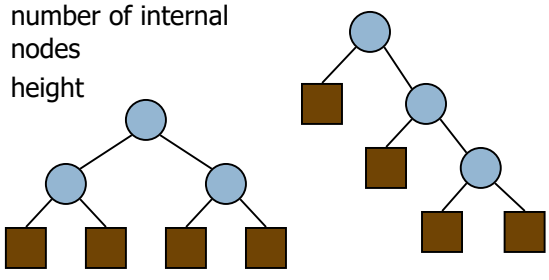
- Binary tree associated with a decision process
  - internal nodes: questions with yes/no answer
  - external nodes: decisions
- Example: dining decision



Properties of Proper Binary Trees

A full **binary tree** (sometimes **proper binary tree** or 2-**tree**) is a **tree** in which every node other than the leaves has two children. A complete **binary tree** is a **binary tree** in which every level, except possibly the last, is completely filled, and all nodes are as far left as possible.

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



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

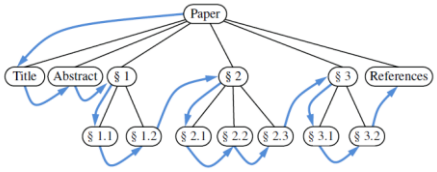
BinaryTree ADT

- 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

Three Traversal Algorithms

- A ***traversal of a tree T is a systematic way of accessing, or "visiting," all the positions*** of *T*.
- *The specific action associated with the "visit" of a position p depends* on the application of this traversal, and could involve anything from incrementing a counter to performing some complex computation for *p*.

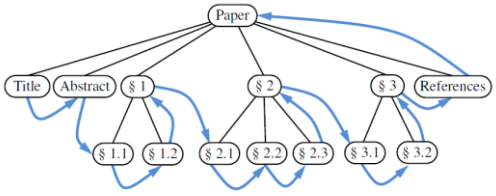
Preorder Traversal



- In a ***preorder traversal of a tree T, the root of T is visited first and then the subtrees*** rooted at its children are traversed recursively. If the tree is ordered, then the subtrees are traversed according to the order of the children.

Algorithm preorder(*p*):
perform the "visit" action for position *p* *{ this happens before any recursion }*
for each child *c* *in children(p) do*
preorder(*c*) *{ recursively traverse the subtree rooted at c }*

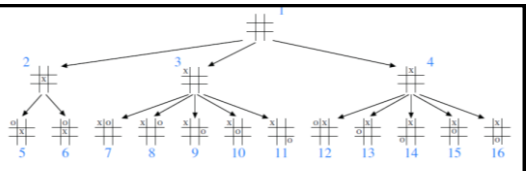
Postorder



- Another important tree traversal algorithm is the ***postorder traversal. In some*** sense, this algorithm can be viewed as the opposite of the preorder traversal, because it recursively traverses the subtrees rooted at the children of the root first, and then visits the root (hence, the name "postorder").

Algorithm postorder(*p*):
for each child *c* *in children(p) do*
postorder(*c*) *{ recursively traverse the subtree rooted at c }*
perform the "visit" action for position *p* *{ this happens after any recursion }*

Breadth-First Tree Traversal



Partial game tree for Tic-Tac-Toe when ignoring symmetries; annotations denote the order in which positions are visited in a breadth-first tree traversal.

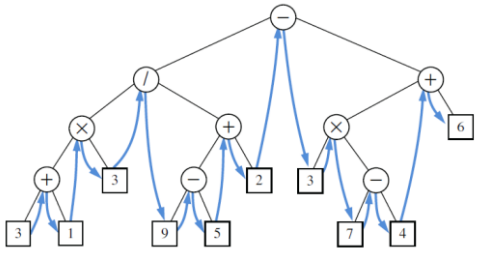
- Although the preorder and postorder traversals are common ways of visiting the positions of a tree, another approach is to traverse a tree so that we visit all the positions at depth *d* before we visit the positions at depth *d*+1. Such an algorithm is known as a *breadth-first traversal*.

Algorithm breadthfirst( ):
Initialize queue *Q to contain root( )*
while *Q not empty do*
p = Q.dequeue( ) { p is the oldest entry in the queue }
perform the "visit" action for position *p*
for each child *c in children(p) do*
Q.enqueue(c) { add p's children to the end of the queue for later visits }

Inorder Traversal

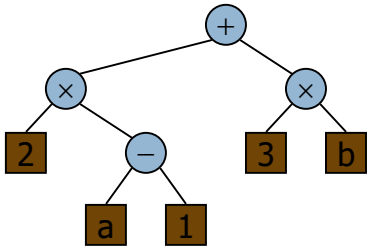
- In an inorder traversal a node is visited after its left subtree and before its right subtree
- Application: draw a binary tree
  - x(v) = inorder rank of v
  - y(v) = depth of v

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



Print Arithmetic Expressions

- Specialization of an inorder traversal
  - print operand or operator when visiting node
  - print "(" before traversing left subtree
  - print ")" after traversing right subtree

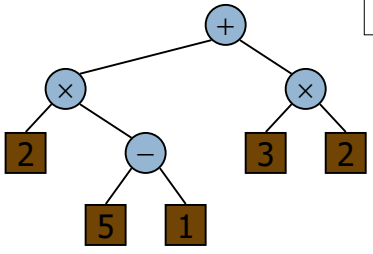


Algorithm *printExpression*(*v*)
  if *left* (*v*) ≠ **null**
    print("(")
    inOrder (*left*(*v*))
  print(*v.element* ())
  if *right*(*v*) ≠ **null**
    inOrder (*right*(*v*))
    print (")")

((2 × (a − 1)) + (3 × b))

Evaluate Arithmetic Expressions

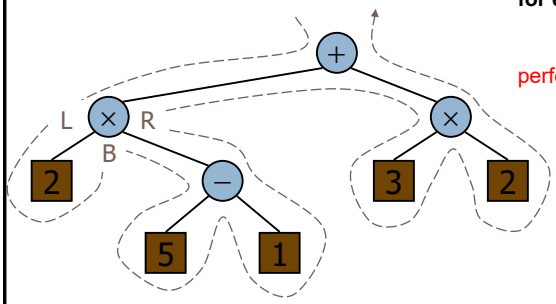
- Specialization of a postorder traversal
  - recursive method returning the value of a subtree
  - when visiting an internal node, combine the values of the subtrees



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

Euler Tour Traversal

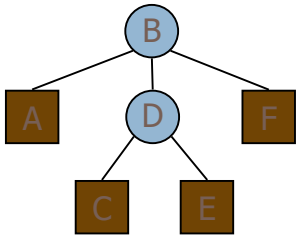
- Generic traversal of a binary tree
- Includes a special cases the preorder, postorder and inorder traversals
- Walk around the tree and visit each node three times:
  - on the left (preorder)
  - from below (inorder)
  - on the right (postorder)

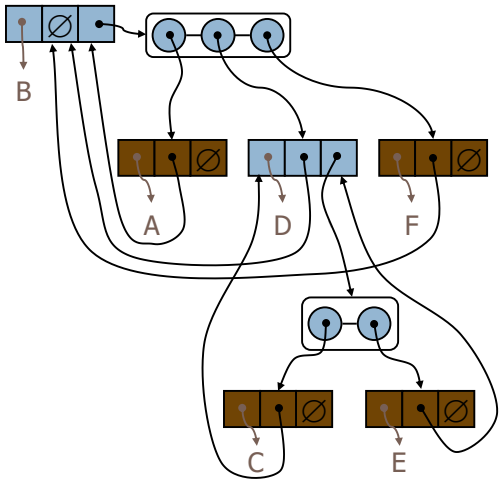


Algorithm eulerTour(*T, p*):
perform the "pre visit" action for position *p*
for each child *c* in *T.children(p)* do
eulerTour(*T, c*) *{ recursively tour the subtree rooted at c }*
perform the "post visit" action for position *p*

Linked Structure for Trees

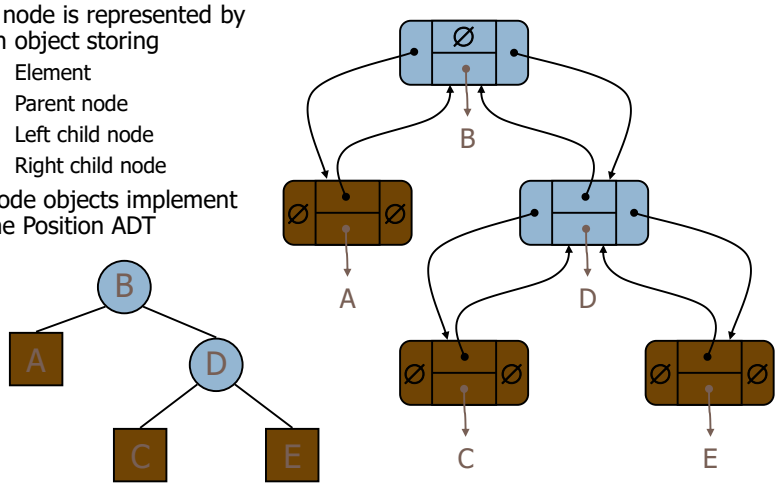
- A node is represented by an object storing
  - Element
  - Parent node
  - Sequence of children nodes
- Node objects implement the Position ADT





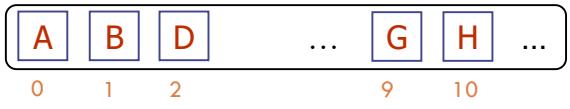
Linked Structure for Binary Trees

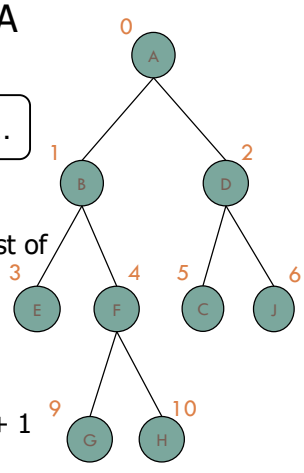
- A node is represented by an object storing
  - Element
  - Parent node
  - Left child node
  - Right child node
- Node objects implement the Position ADT



Array-Based Representation of Binary Trees

- Nodes are stored in an array A





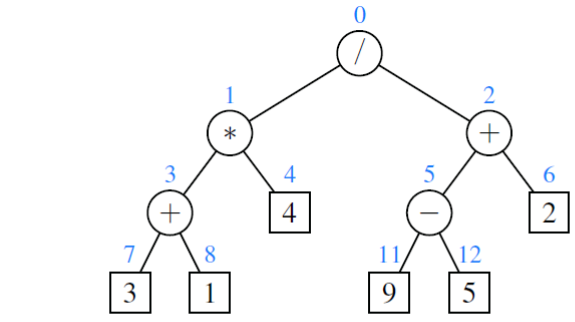
The rank of a node is its position in a sorted list of the nodes.

Node v is stored at A[rank(v)]

- rank(root) = 0
- if node is the left child of parent(node),
  rank(node) = 2 · rank(parent(node)) + 1
- if node is the right child of parent(node),
  rank(node) = 2· rank(parent(node)) + 2

Ex. Binary Tree Array Representation

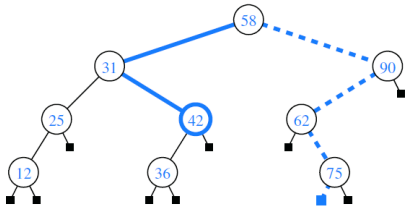
Representation of a binary tree by means of an array.



/	*	+	+	4	−	2	3	1			9	5		
0	1	2	3	4	5	6	7	8	9	10	11	12	13	14

Binary Search Trees

- An important application of the inorder traversal algorithm arises when we store an ordered sequence of elements in a binary tree, defining a structure we call a ***binary search tree.***
- *Let S be a set whose unique elements have an order relation. For example, S could be a set of integers. A binary search tree for S is a proper binary tree T such that, for each internal position p of T:*
  - Position *p stores an element of S, denoted as e(p).*
  - Elements stored in the left subtree of *p (if any) are less than e(p).*
  - Elements stored in the right subtree of *p (if any) are greater than e(p).*



A binary search tree storing integers. The solid path is traversed when searching (successfully) for 42. The dashed path is traversed when searching (unsuccessfully) for 70.

Hw.

- Two ordered trees *T′ and T″ are said to be* ***isomorphic if one of the following*** holds:
- Both *T′ and T″ are empty.*
- Both *T′ and T″ consist of a single node*
- The roots of *T′ and T″ have the same number k ≥ 1 of subtrees, and the i th such subtree of T′ is isomorphic to the i th such subtree of T″ for i =* 1, . . . ,*k.*
- Design and implement an algorithm that tests whether two given ordered trees are isomorphic. What is the running time of your algorithm?