Trees

Linear Data Structures

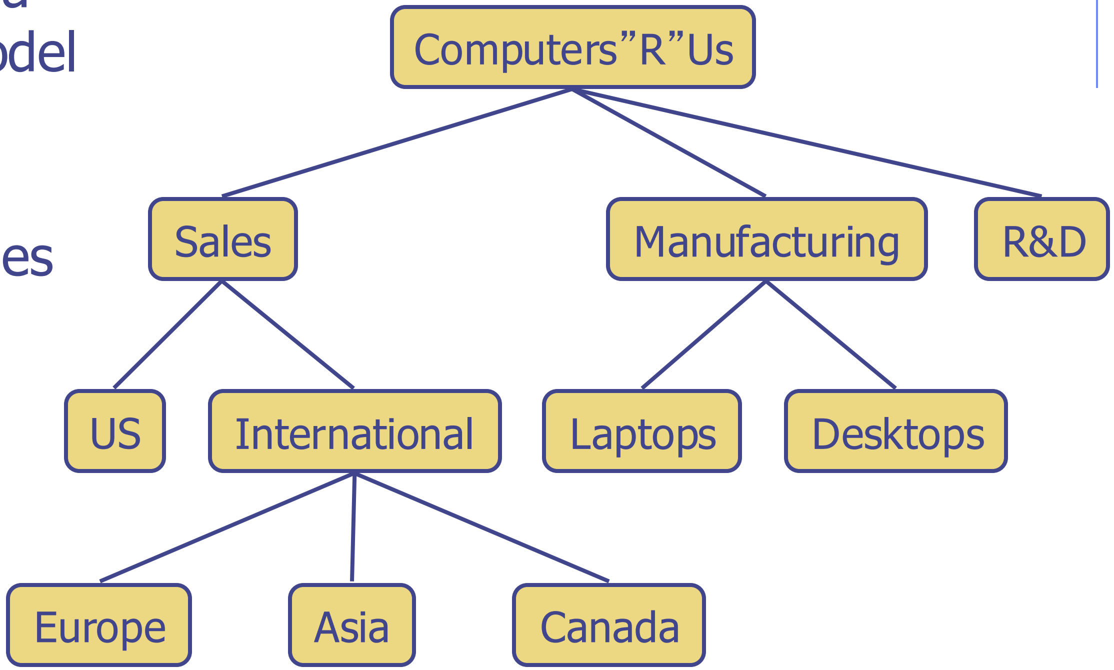
- Data structures we have studied so far:
  - Arrays
  - Stacks, queues, and deques
  - Singly-linked lists and doubly-linked lists
- Common property

  Their visual representation forms a straight line
- Trees are nonlinear data structures

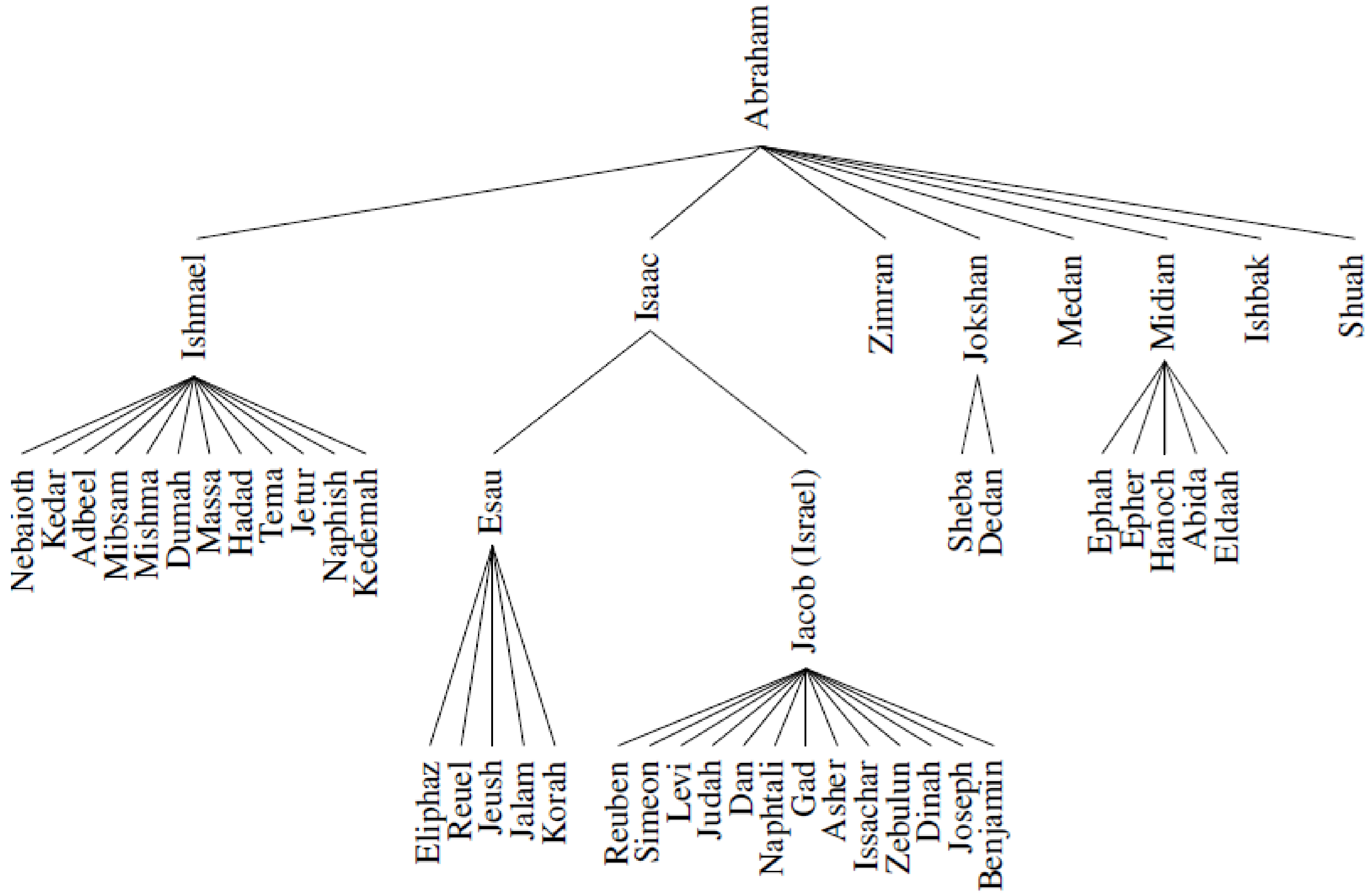
  Binary trees: one of the simplest nonlinear data structures

What is a Tree

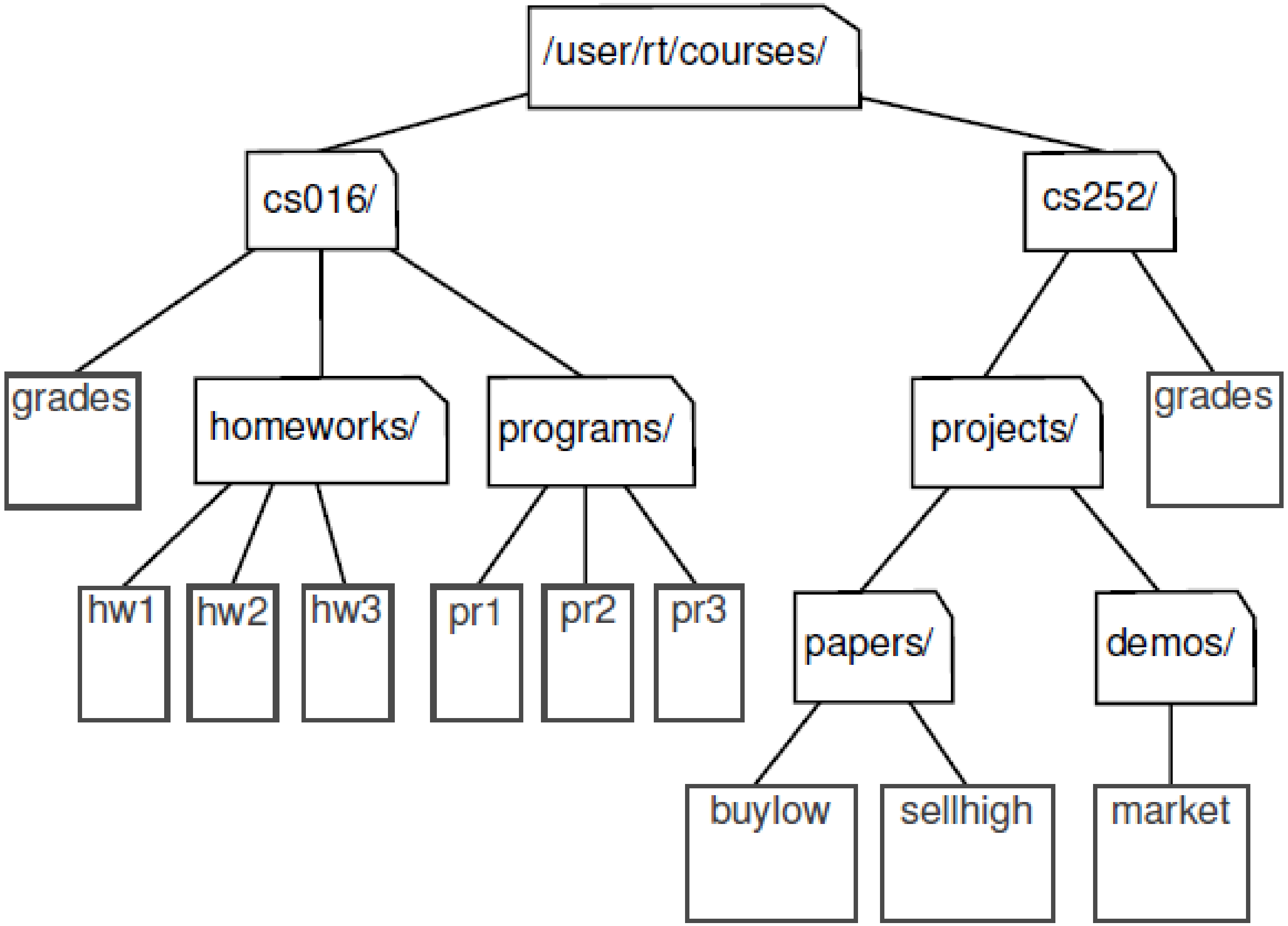
- In computer science, a tree is an abstract model of a hierarchical structure
- A tree consists of nodes with a parent-child relation
- Applications:
  - Organization charts
  - File systems
  - Programming environments
  - Games



Family Tree

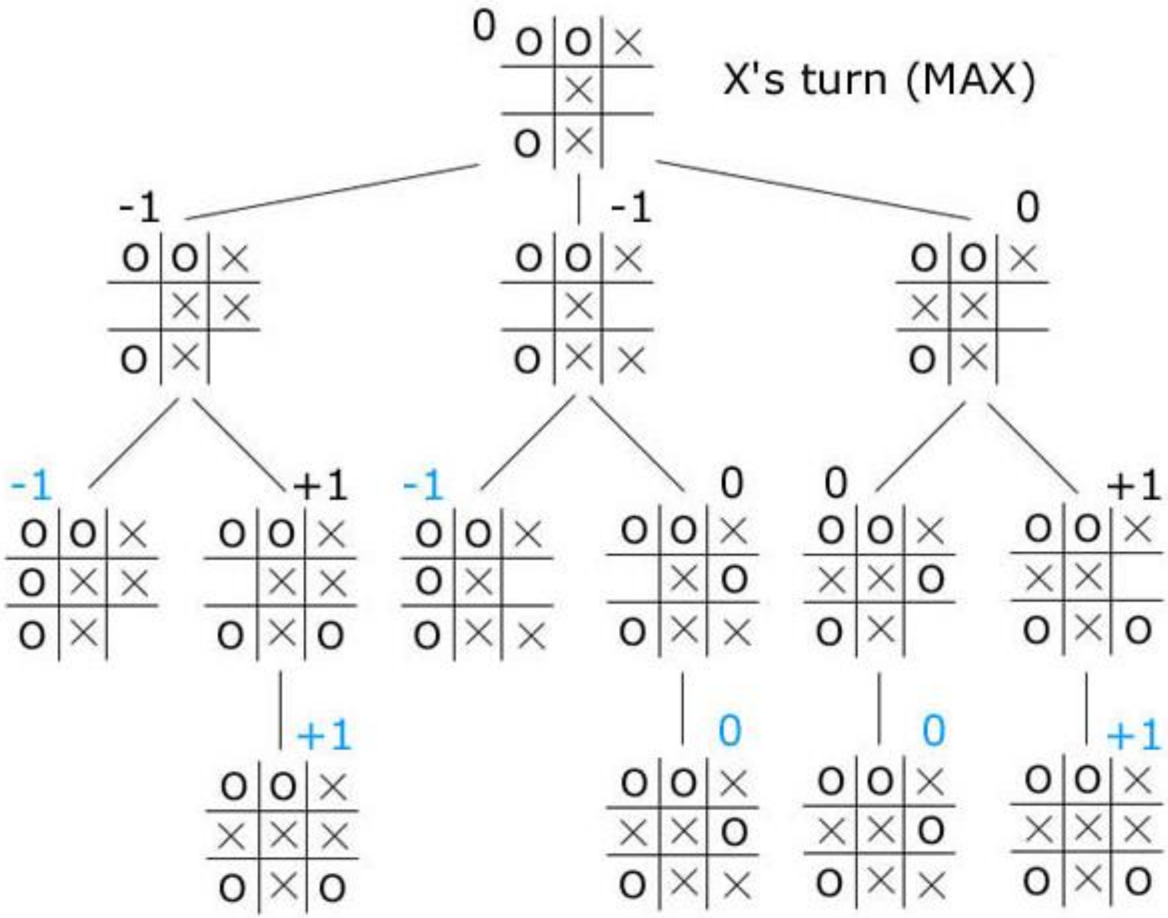

Abraham
Ishmael
Isaac
Zimran
Jokshan
Medan
Midian
Ishbak
Shuah
Nebaioth
Kedar
Adbeel
Mibsam
Mishma
Dumah
Massa
Hadad
Tema
Jetur
Naphish
Kedemah
Esau
Jacob (Israel)
Sheba
Dedan
Ephah
Epher
Hanoch
Abida
Eldaah
Eliphaz
Reuel
Jeush
Jalam
Korah
Reuben
Simeon
Levi
Judah
Dan
Naphtali
Gad
Asher
Issachar
Zebulun
Dinah
Joseph
Benjamin

Computer File Systems



Game tree

- Lookahead possible moves in multiple turns between 2 opponents.
- Select the most favorable move for X

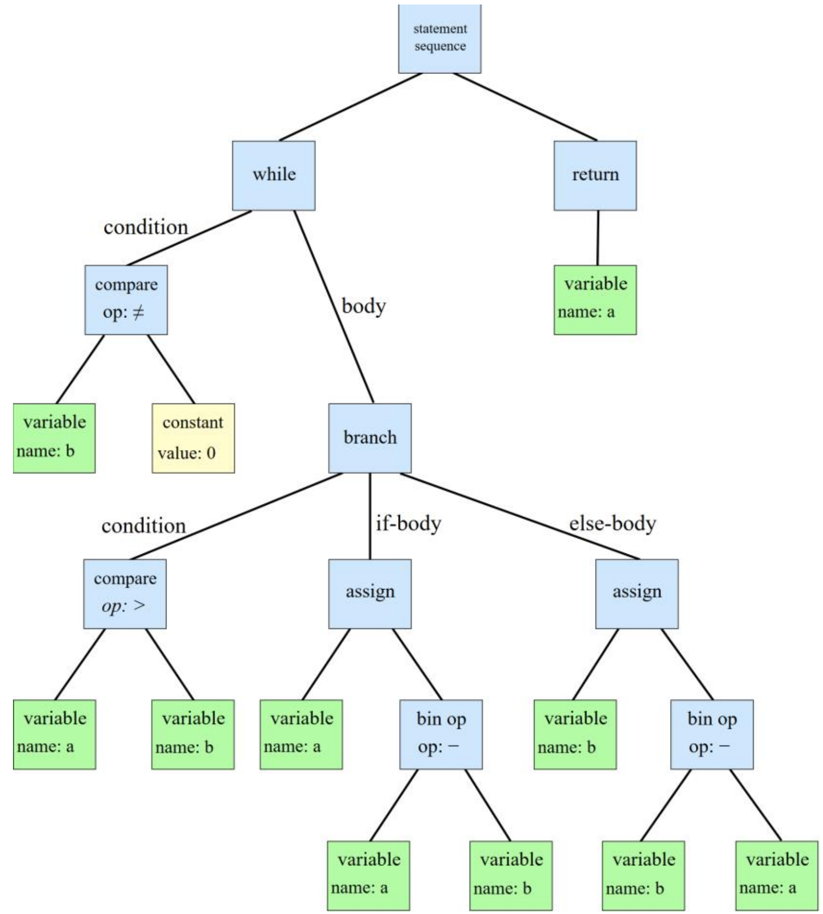


Abstract Syntax Tree

```python
import ast

source_code = """
while b != 0:
    if a > b:
        a = a - b
    else:
        b = b - a
return a
"""

tree = ast.parse(source_code)

# Dump the AST as a string
print(ast.dump(tree, indent=3))
```

--NORMAL--

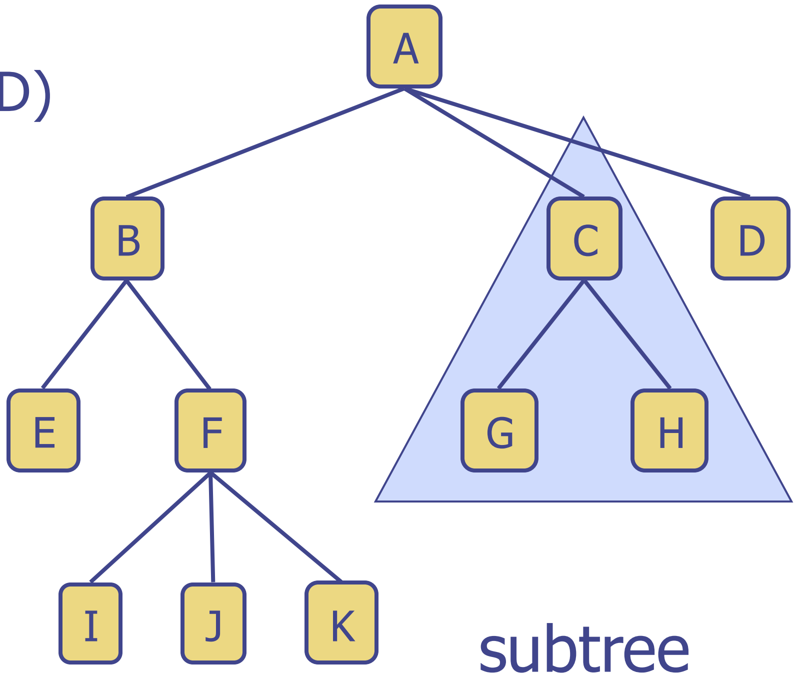
```
Module(
   body=[
      While(
         test=Compare(
            left=Name(id='b', ctx=Load()),
            ops=[
               NotEq()],
            comparators=[
               Constant(value=0)]),
         body=[
            If(
               test=Compare(
                  left=Name(id='a', ctx=Load()),
                  ops=[
                     Gt()],
                  comparators=[
                     Name(id='b', ctx=Load())]),
               body=[
                  Assign(
                     targets=[
                        Name(id='a', ctx=Store())],
                     value=BinOp(
                        left=Name(id='a', ctx=Load()),
                        op=Sub(),
                        right=Name(id='b', ctx=Load())))],
               orelse=[
                  Assign(
                     targets=[
                        Name(id='b', ctx=Store())],
                     value=BinOp(
                        left=Name(id='b', ctx=Load()),
                        op=Sub(),
                        right=Name(id='a', ctx=Load())))])],
         orelse=[]),
      Return(
         value=Name(id='a', ctx=Load()))],
   type_ignores=[])
```



https://upload.wikimedia.org/wikipedia/commons/c/c7/Abstract_syntax_tree_for_Euclidean_algorithm.svg

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

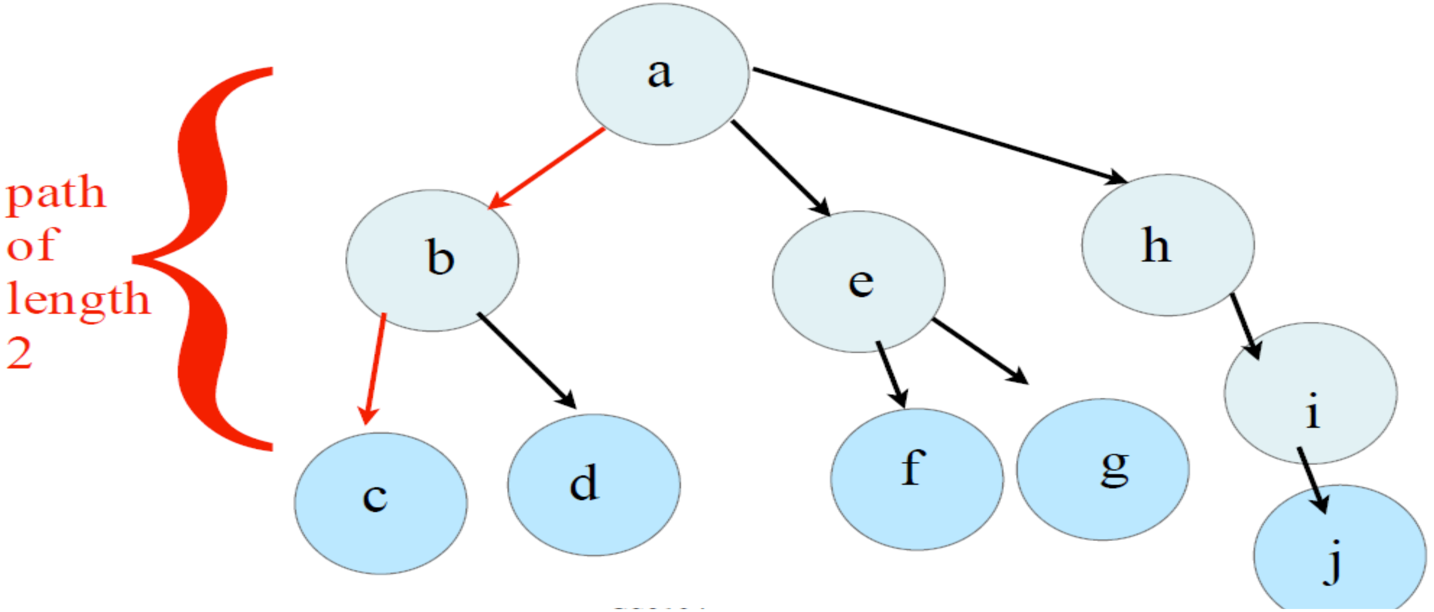


Path between two nodes

- What is the definition of the length of a path between two nodes?

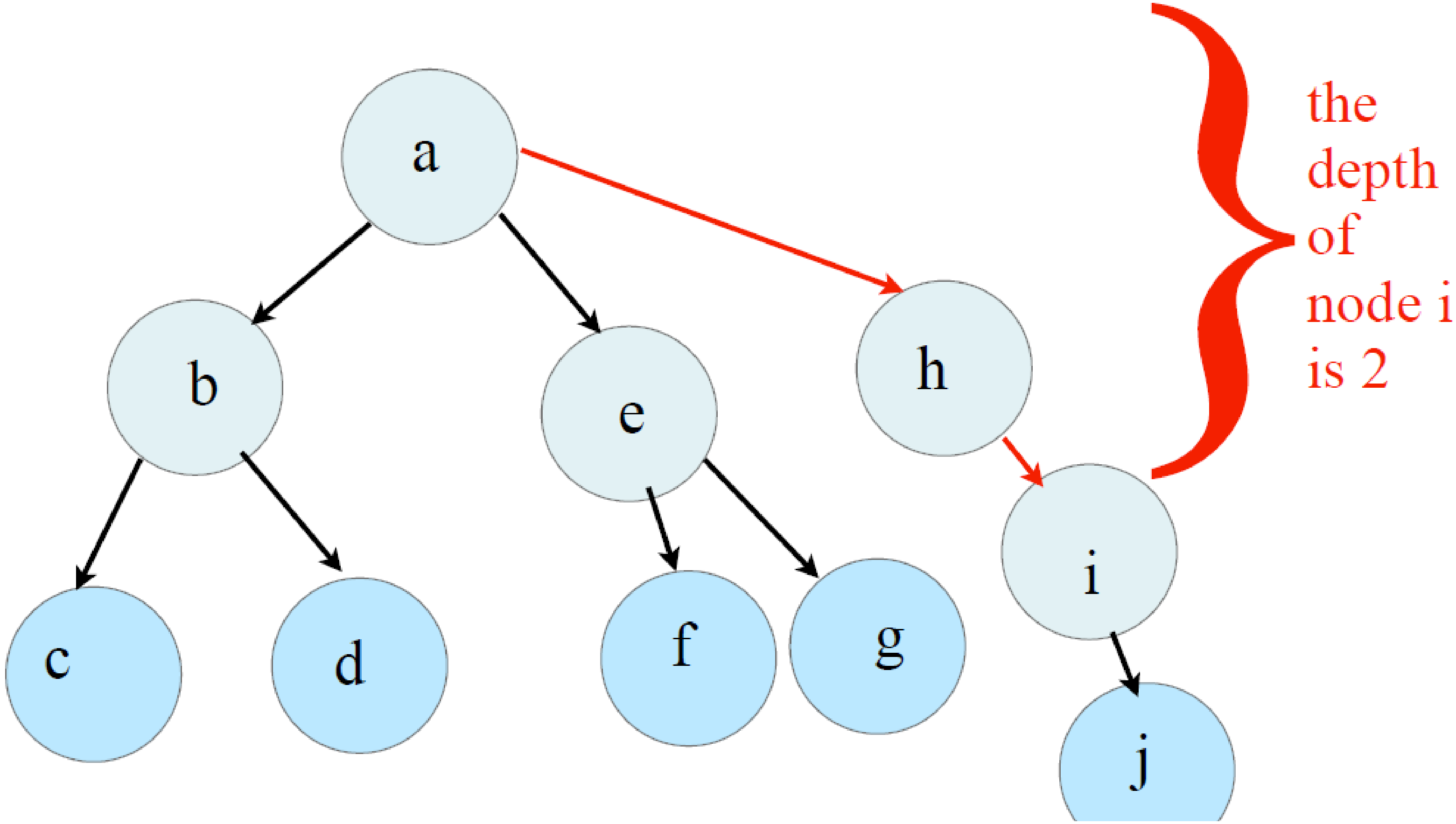
There is one unique path from the root to any node in the tree.

The number of edges along the path is the path length.



Depth of a Tree

The *depth* of a node is the number of edges from the root to the node



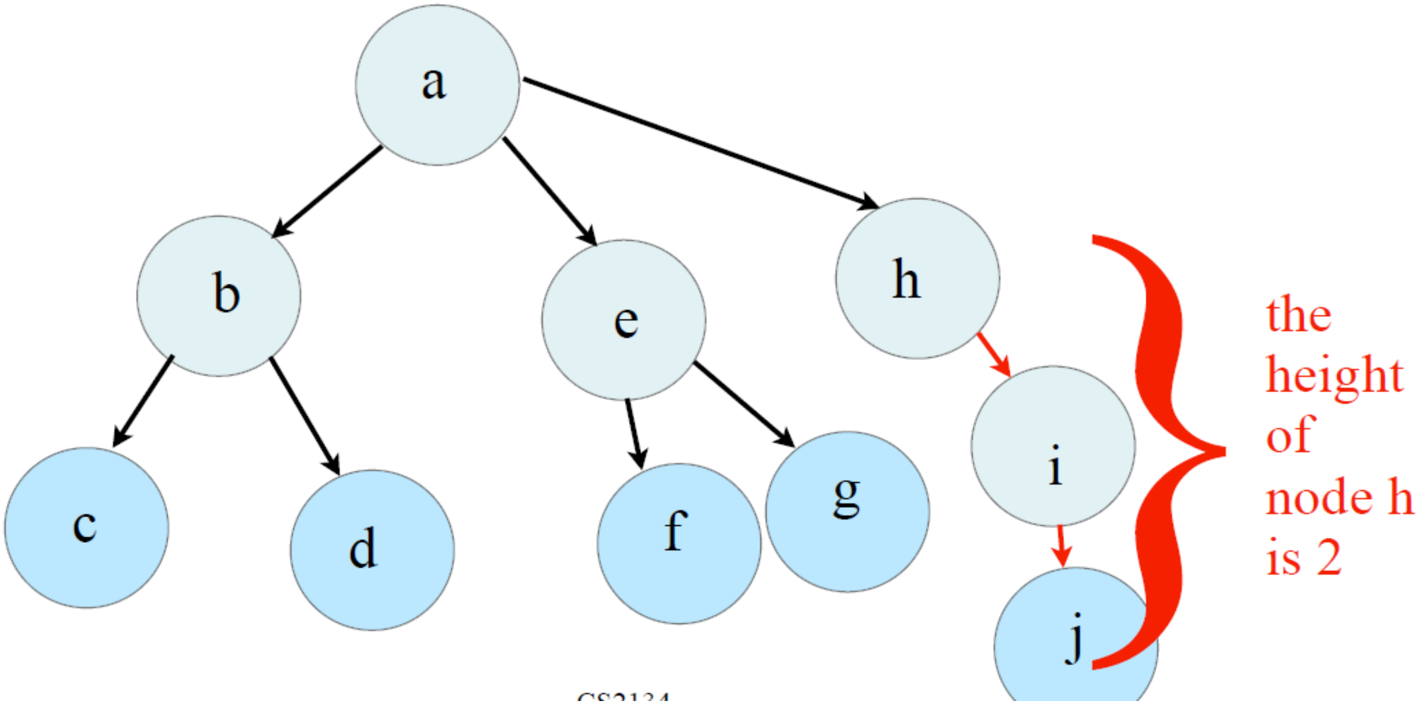
Recursive computation

- Base case: At root node, depth is 0

```
def depth(self, p):
  """Return the number of levels separating Position p from the root."""
  if self.is_root(p):
    return 0
  else:
    return 1 + self.depth(self.parent(p))
```

Height of a tree

The *height* of a node is the number of edges from the node to the deepest leaf.

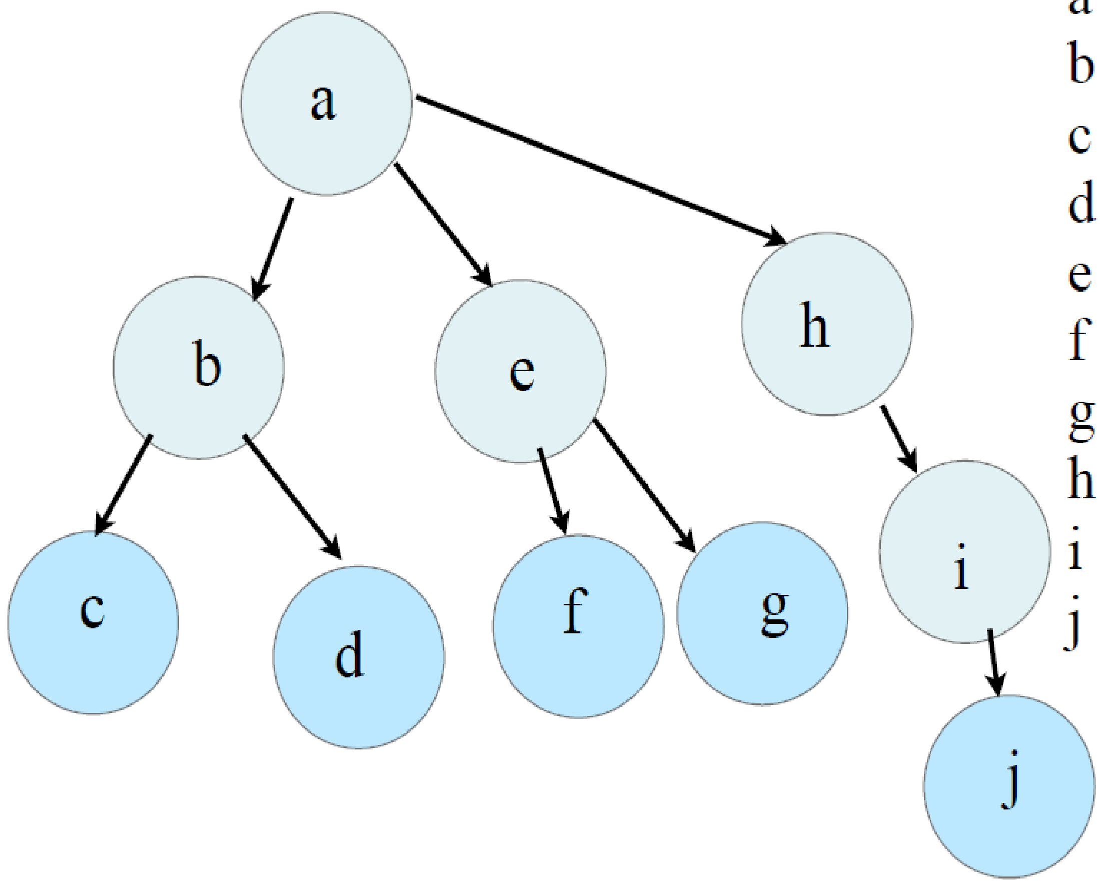


Height of a tree

- The ***height*** of a position *p* in a tree *T* is also defined recursively:
  - 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.
- The ***height*** of a nonempty tree *T* is the height of the root of *T*.

```
def _height2(self, p):                  # time is linear in size of subtree
  """Return the height of the subtree rooted at Position p."""
  if self.is_leaf(p):
    return 0
  else:
    return 1 + max(self._height2(c) for c in self.children(p))
```

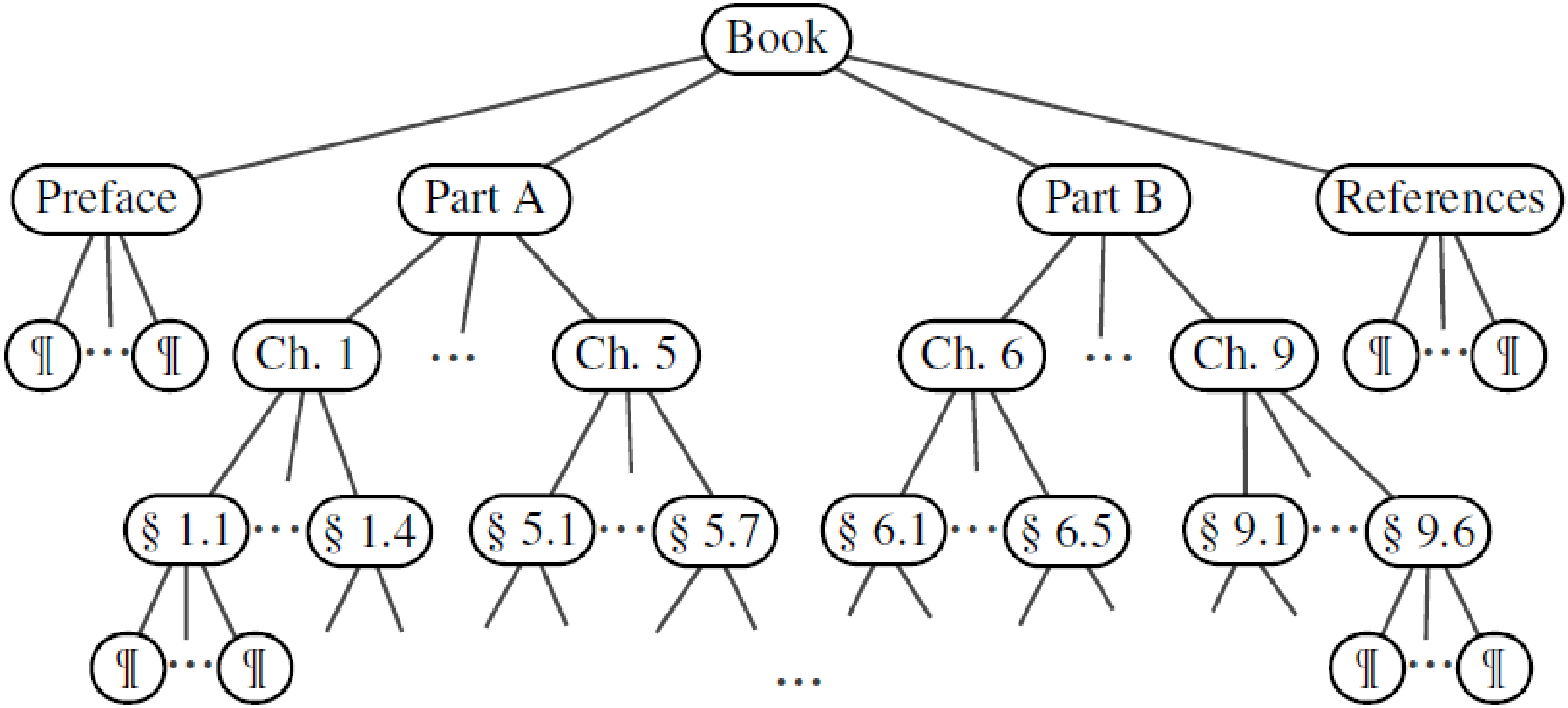
What are the height and depth of nodes a, b and c?



Node	Height	Depth
a	3	0
b	1	1
c	0	2
d	0	2
e	1	1
f	0	2
g	0	2
h	2	1
i	1	2
j	0	3

Ordered Trees

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



Question?

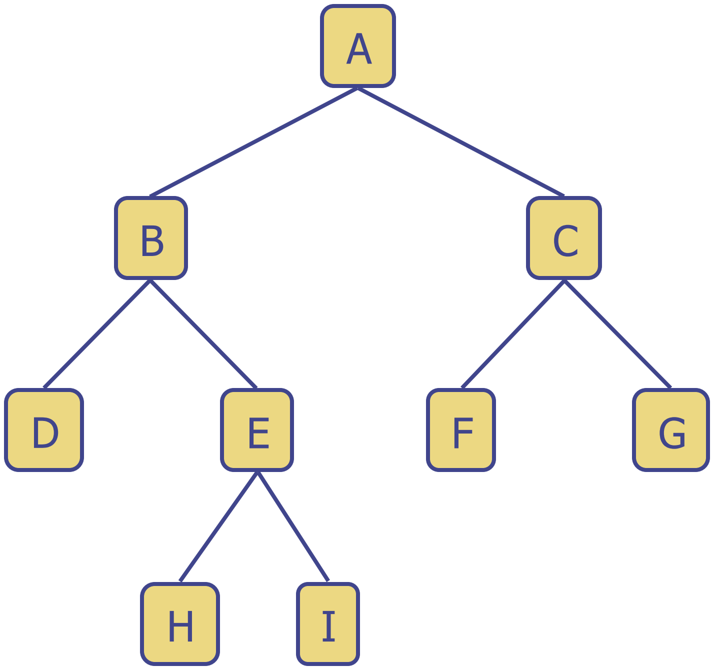
- If a tree has n nodes how many edges does it have?

Answer: n-1 edges

Binary Trees

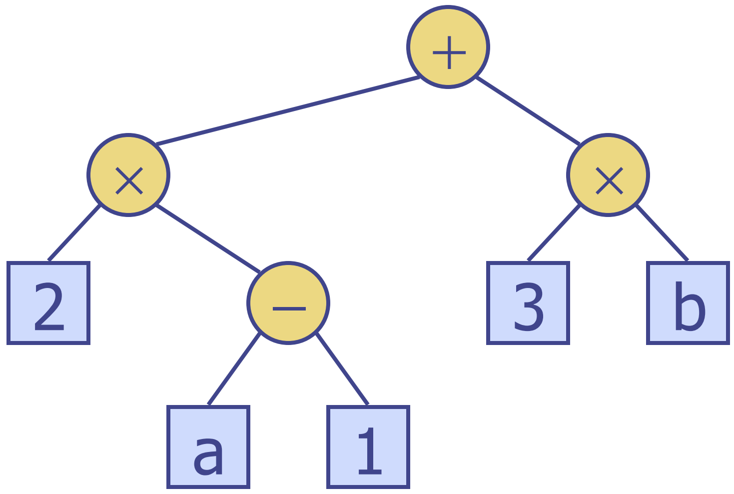
- A binary tree is a tree with the following properties:
  - Each internal node has at most two children (exactly two for proper/full binary trees)
  - The children of a node are an ordered pair
- We call the children of an internal node left child and right child
- Alternative recursive definition: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



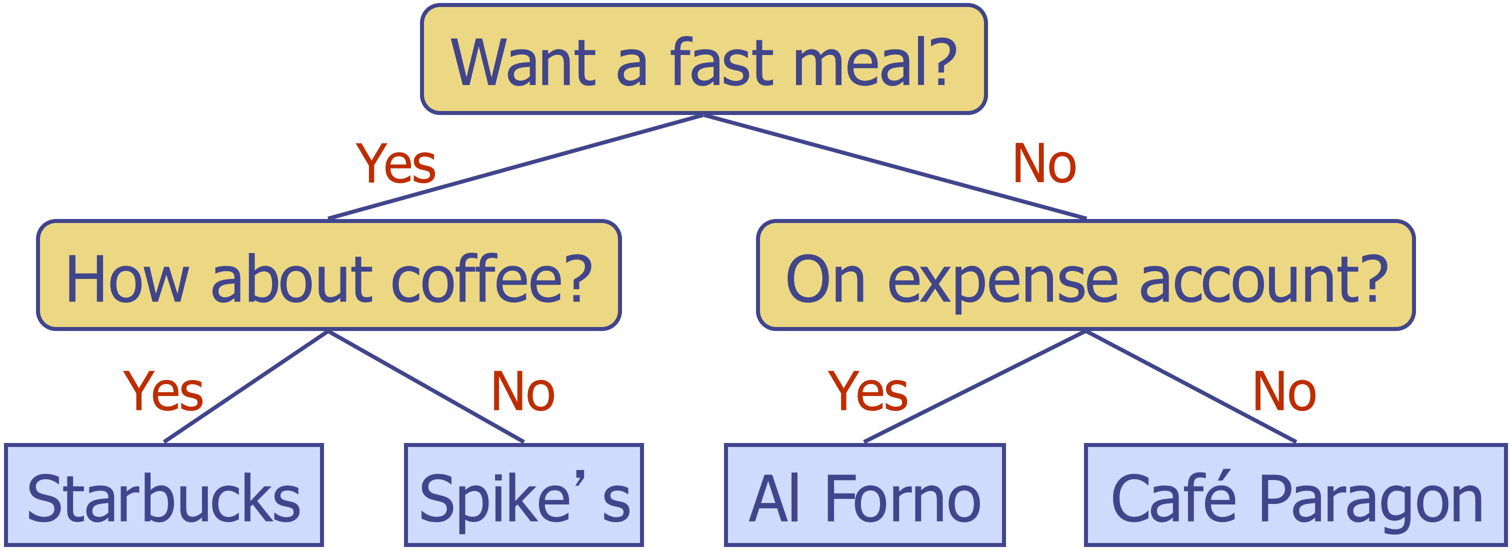
Arithmetic Expression Tree

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

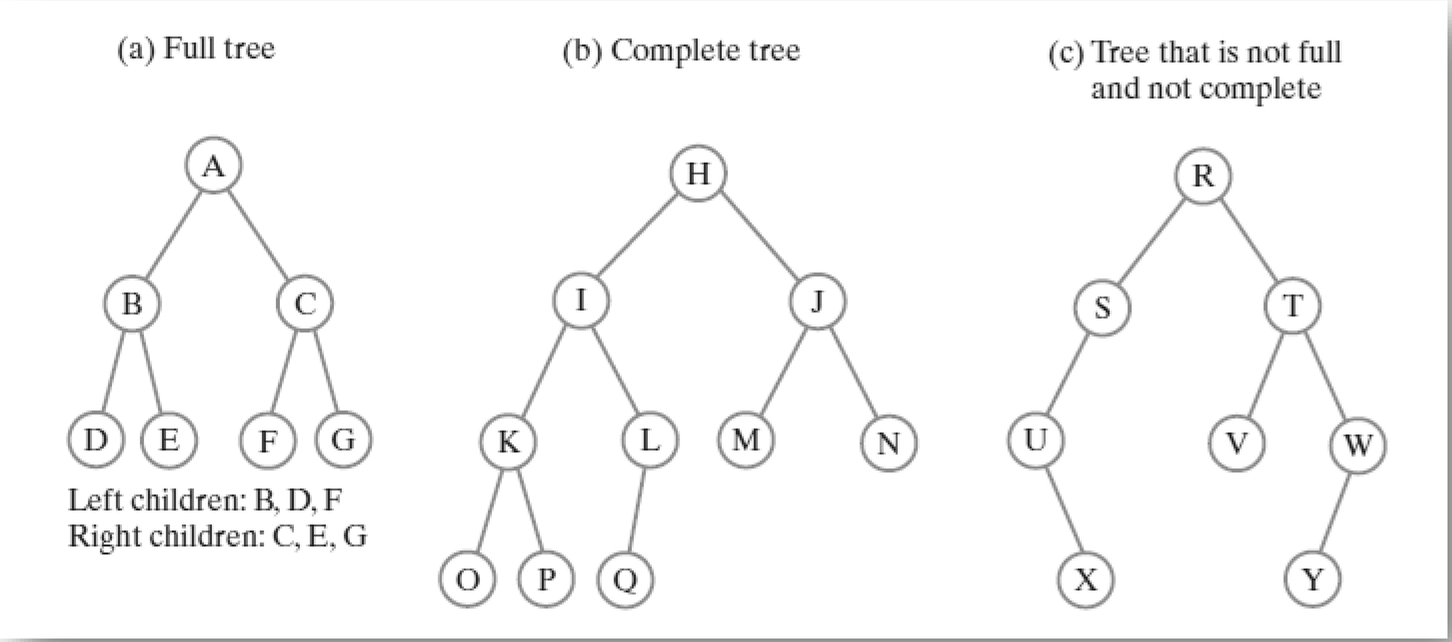


Decision Tree

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

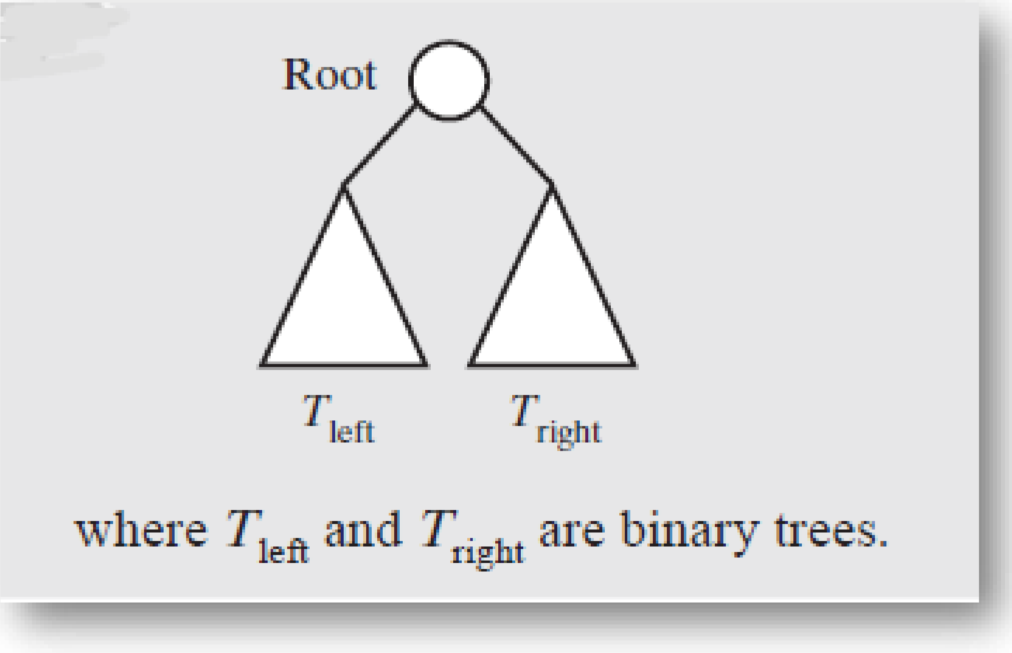


Binary Trees



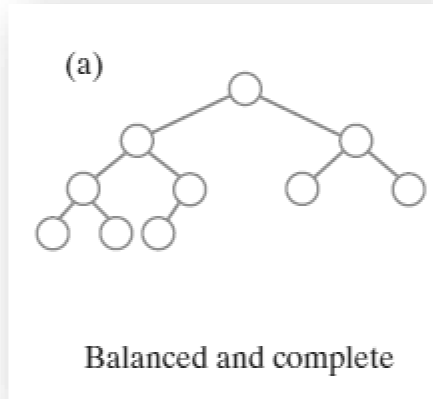
Three binary trees

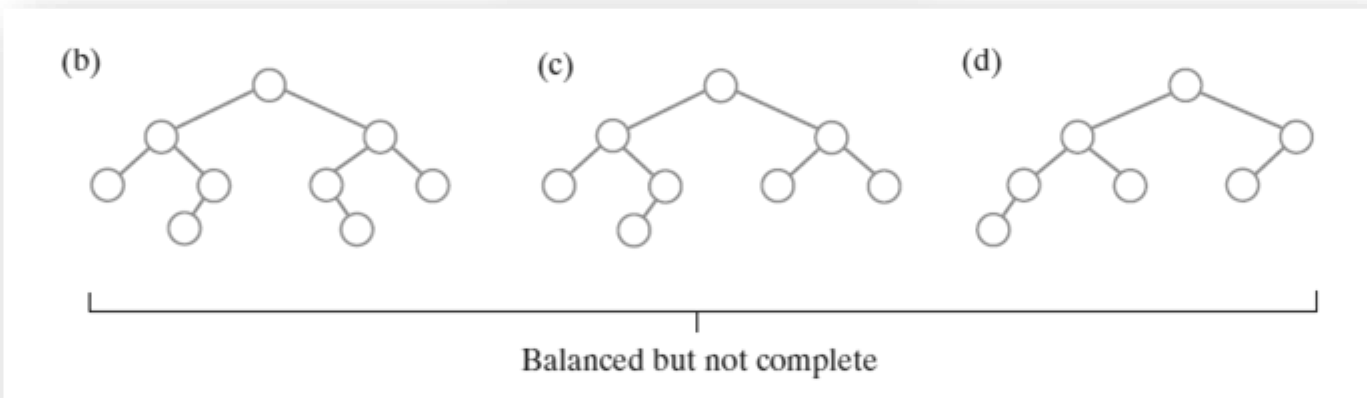
Binary Trees



A binary tree is empty or has the above form

Binary Trees





Some binary trees that are height balanced

A **balanced binary tree** is a **binary tree structure** in which the left and right subtrees of every node differ in height by no more than 1.

Properties of Binary Trees

- Notation

  n number of nodes

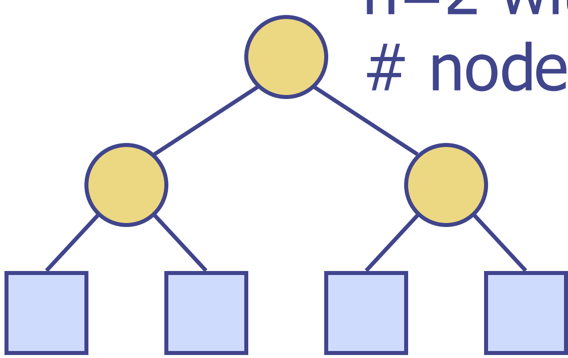
  n_e number of external nodes

  n_i number of internal nodes

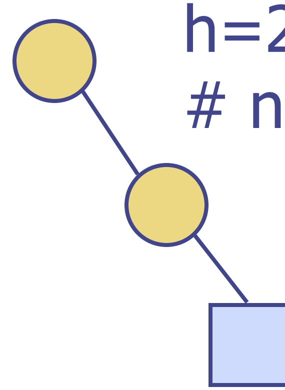
  h height

Properties:

1. $h+1 \le n \le 2^{h+1}-1$
2. $1 \le n_E \le 2^h$
3. $h \le n_I \le 2^h - 1$
4. $\log(n+1)-1 \le h \le n-1$



h=2 with max # nodes



h=2 with min # nodes

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.

- Notation

  n number of nodes

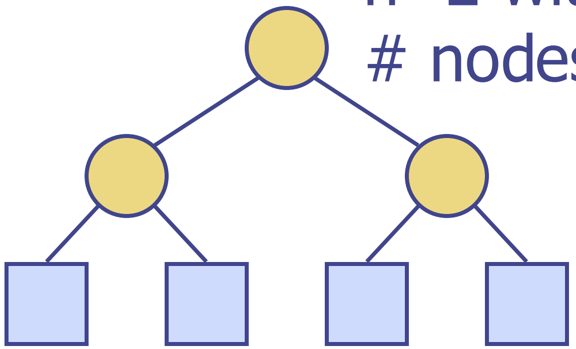
  n_e number of external nodes

  n_i number of internal nodes

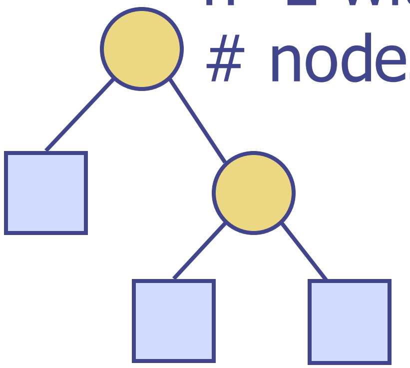
  h height

1. $2h+1 \le n \le 2^{h+1}-1$
2. $h+1 \le n_E \le 2^h$
3. $h \le n_I \le 2^h - 1$
4. $\log(n+1)-1 \le h \le (n-1)/2$

h=2 with max # nodes



h=2 with min # nodes



Traversals of A Tree

- Definition
  - Visit, or process, each data item exactly once
  - Visit can be delayed
    - Traversal can pass through a node without processing it
- Order of the visits is not unique
- First consider traversals of a binary tree
  - Somewhat easier to understand

Tree Traversals (Binary Tree)

- **pre-order**
  - –visit root
  - –traverse left subtree
  - –traverse right subtree
- **post-order**
  - –traverse left subtree
  - –traverse right subtree
  - –visit root
- **in-order**
  - –traverse left subtree
  - –visit root
  - –traverse right subtree

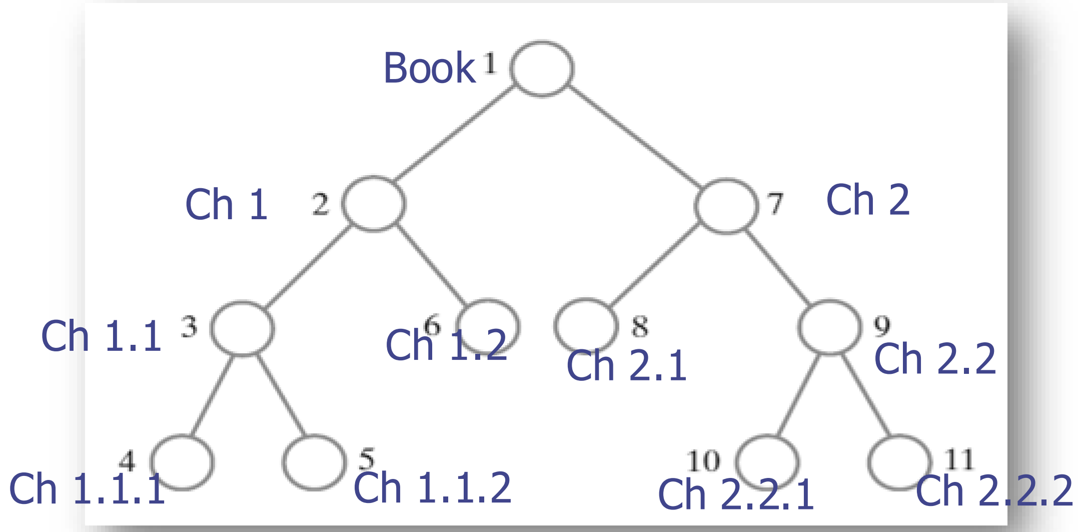
- **level-order (**
  - –visit root
  - – traverse level 1 nodes
  - – traverse level 2 nodes
  - – -----
  - – -----
  - – -----
  - – traverse last level nodes

* For Proper tree, level order is same as **Breadth First Traversal**

Preorder Traversal (any tree)

- 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

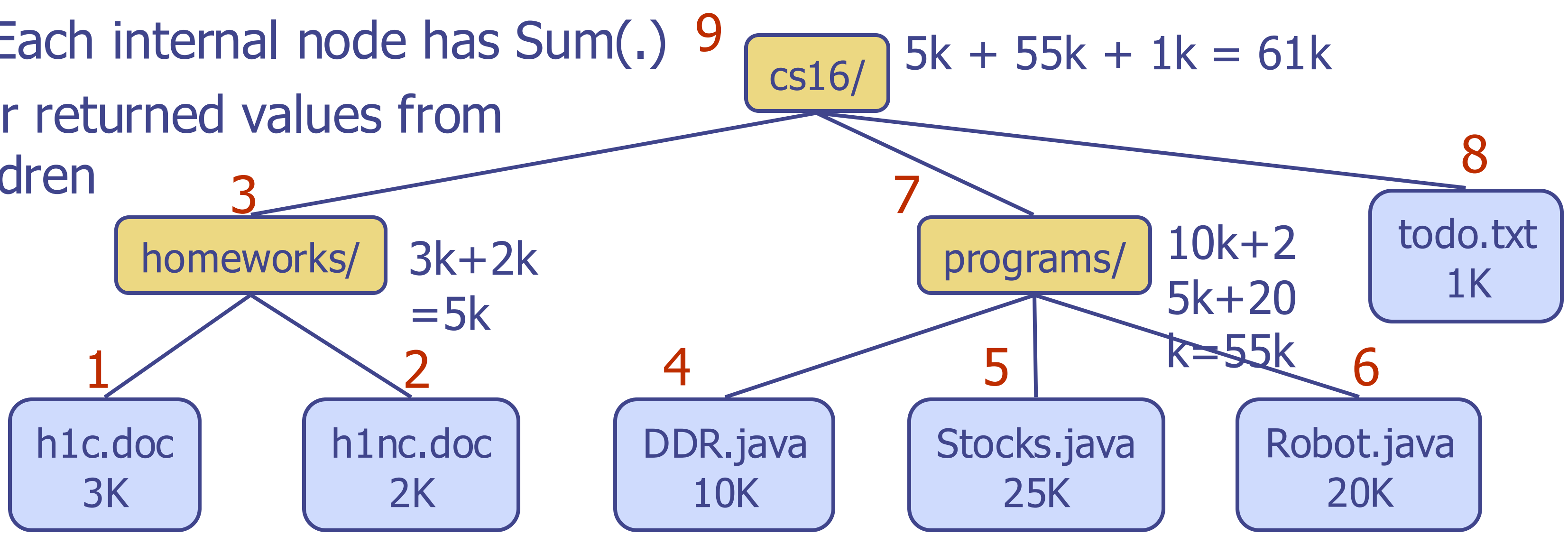
Algorithm *preOrder*(v)
  visit(v)
  for each child w of v
    preorder (w)



Postorder Traversal (any tree)

- In a postorder traversal, a node is visited after its descendants
- Application: compute space used by files in a directory and its subdirectories
- Each internal node has Sum(.) over returned values from children

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



Post-order sum operations

```
def sum_directory_space(node):
  if node.is_leaf():
    # we hit a file item. So return the file size
    return node.value
  # now we are in an internal node
  left_size, right_size = 0, 0
  if node.left:
    # Process the left sub-tree (Recursion here)
    left_size = sum_directory_space(node.left)
  if node.right:
    # Process the right sub-tree (Recursion here)
    right_size = sum_directory_space(node.right)
  return left_size + right_size
```

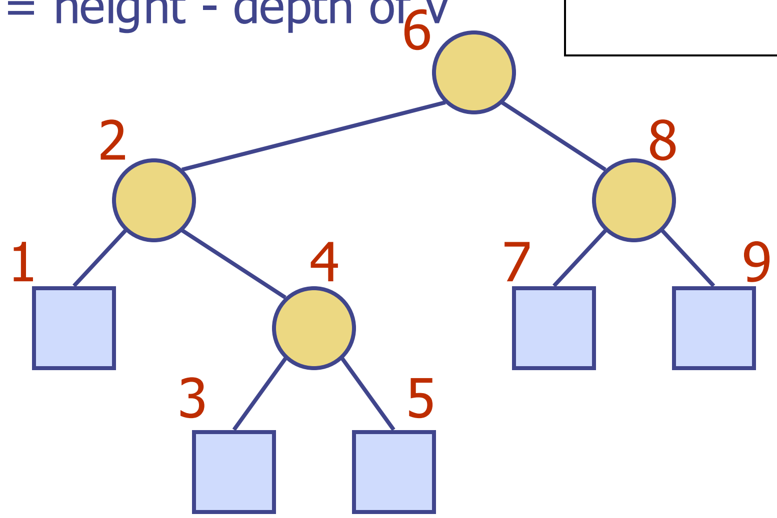
Post-order:

A node finishes processing after all of its child nodes finished processing

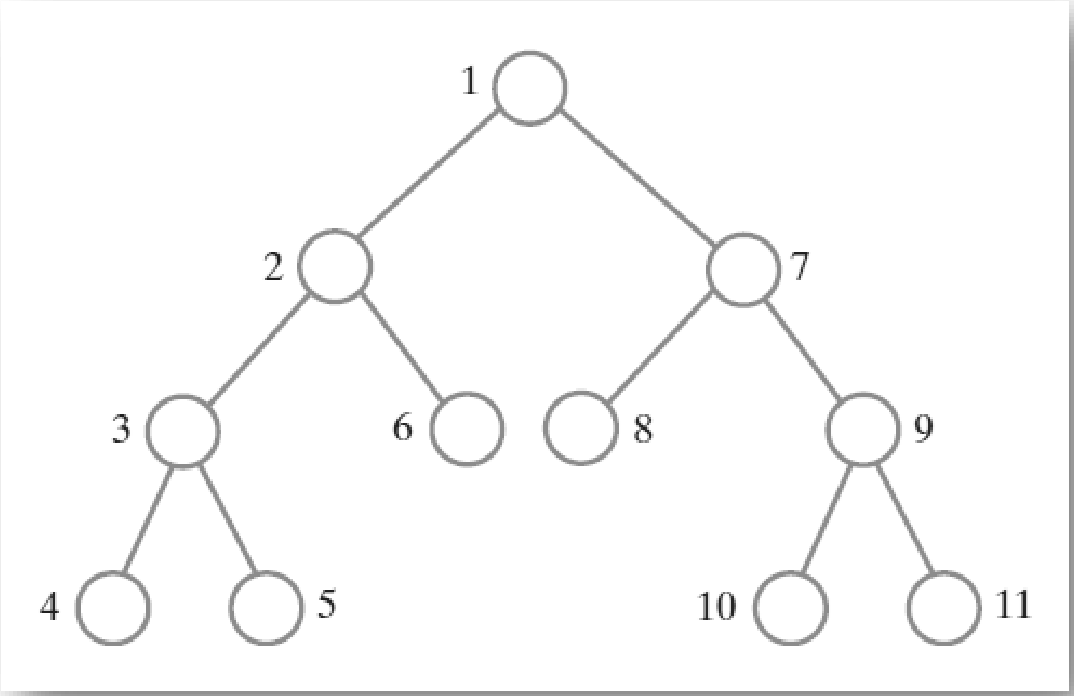
Inorder Traversal (binary tree)

- 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) = height - depth of v

```
Algorithm inOrder(v)
    if v has a left child
        inOrder (left (v))
    visit(v)
    if v has a right child
        inOrder (right (v))
```

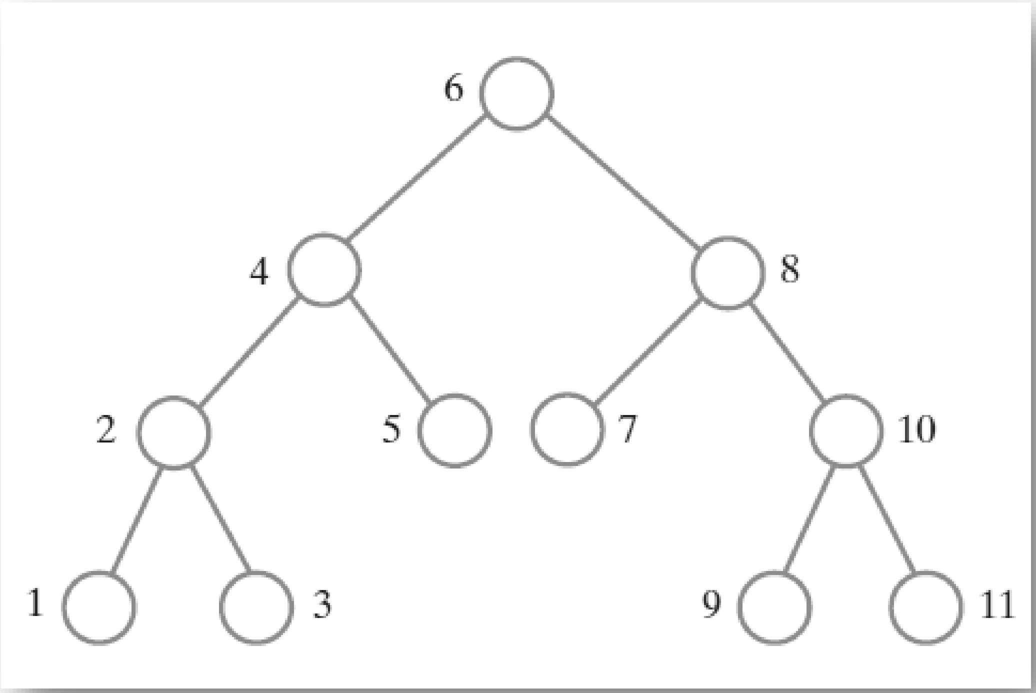


Traversals of a Binary Tree



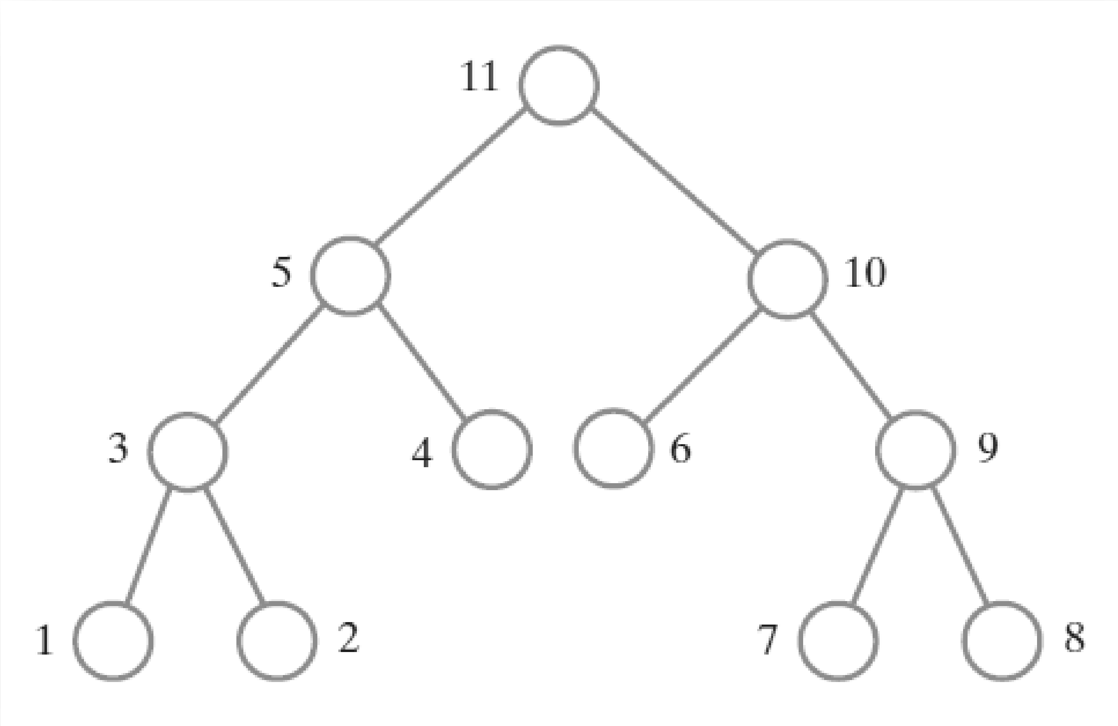
The visitation order of a preorder traversal

Traversals of a Binary Tree



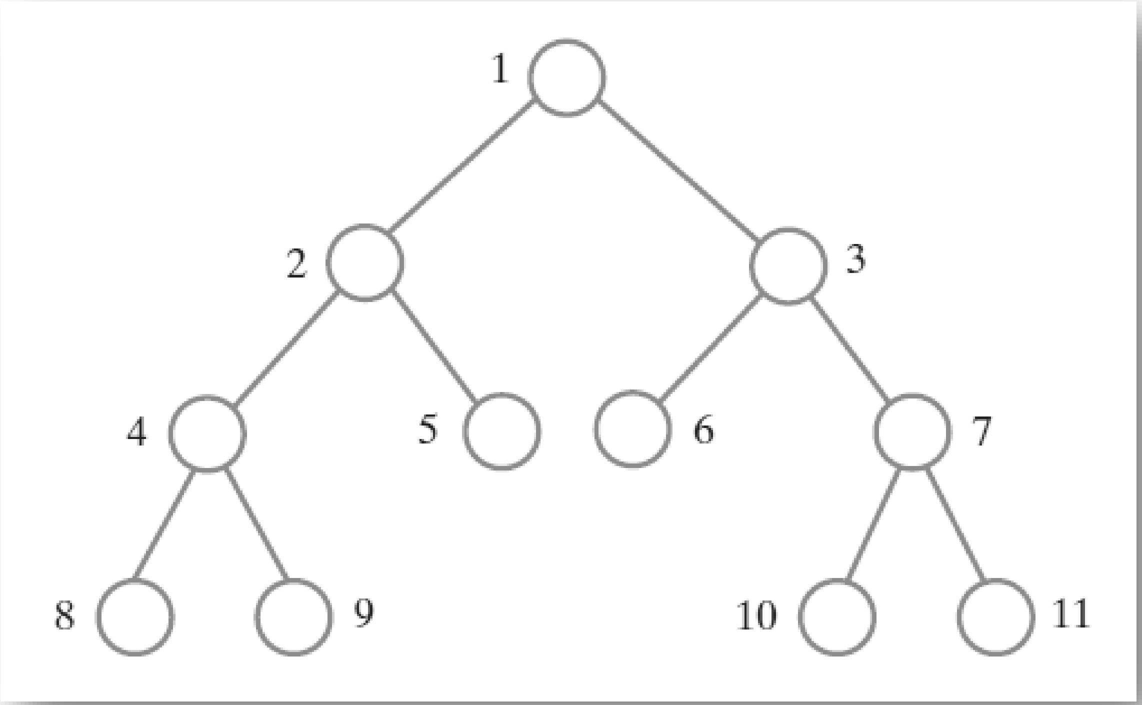
The visitation order of an inorder traversal

Traversals of a Binary Tree



The visitation order of a postorder traversal

Traversals of a Binary Tree



The visitation order of a level-order traversal

Practice
Pre-order, Post-Order, In-order

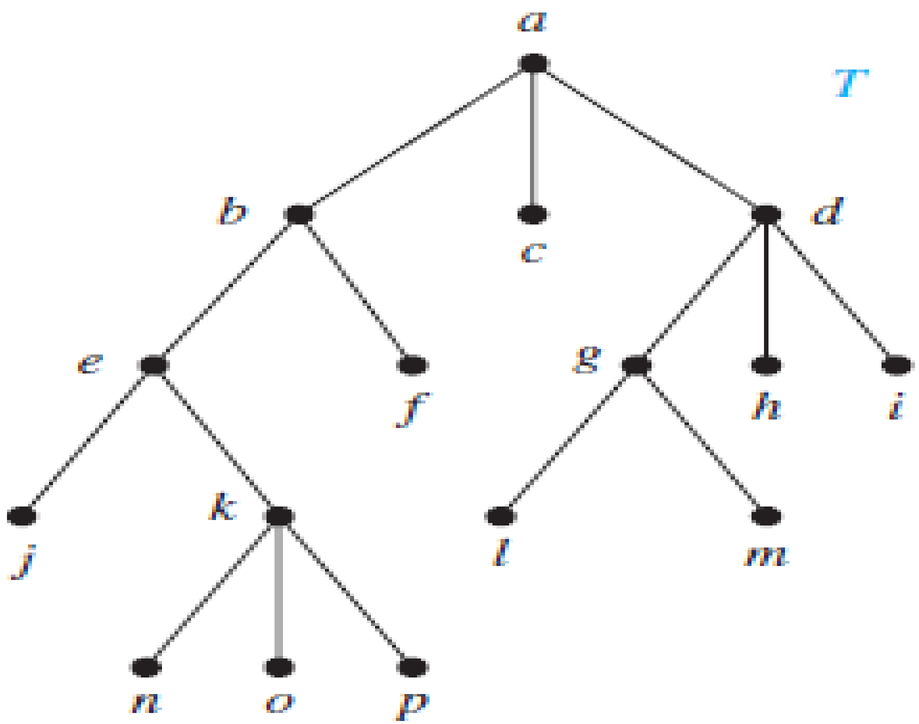


FIGURE 3 The Ordered Rooted Tree T.

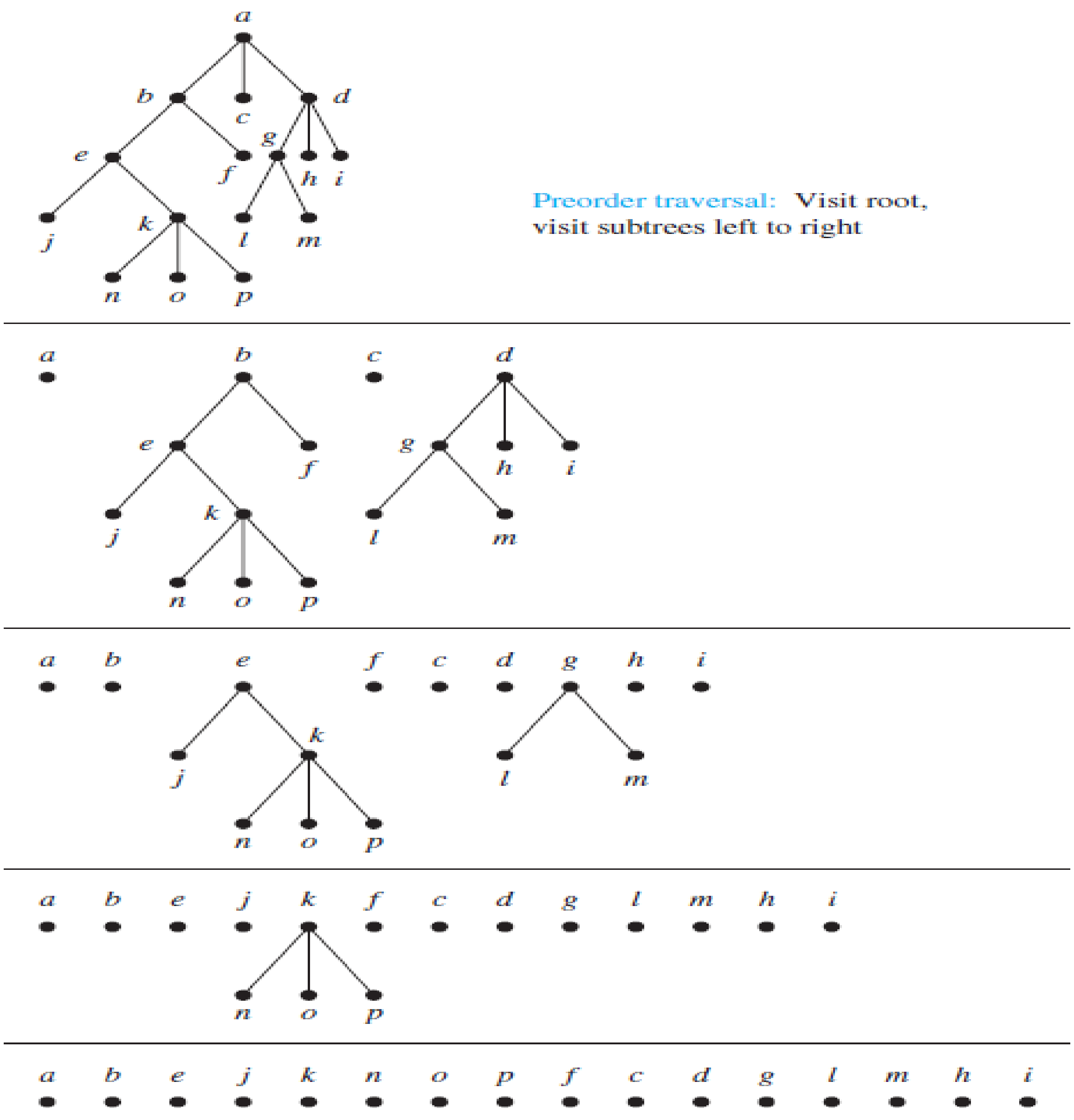


FIGURE 4 **The Preorder Traversal of *T*.**

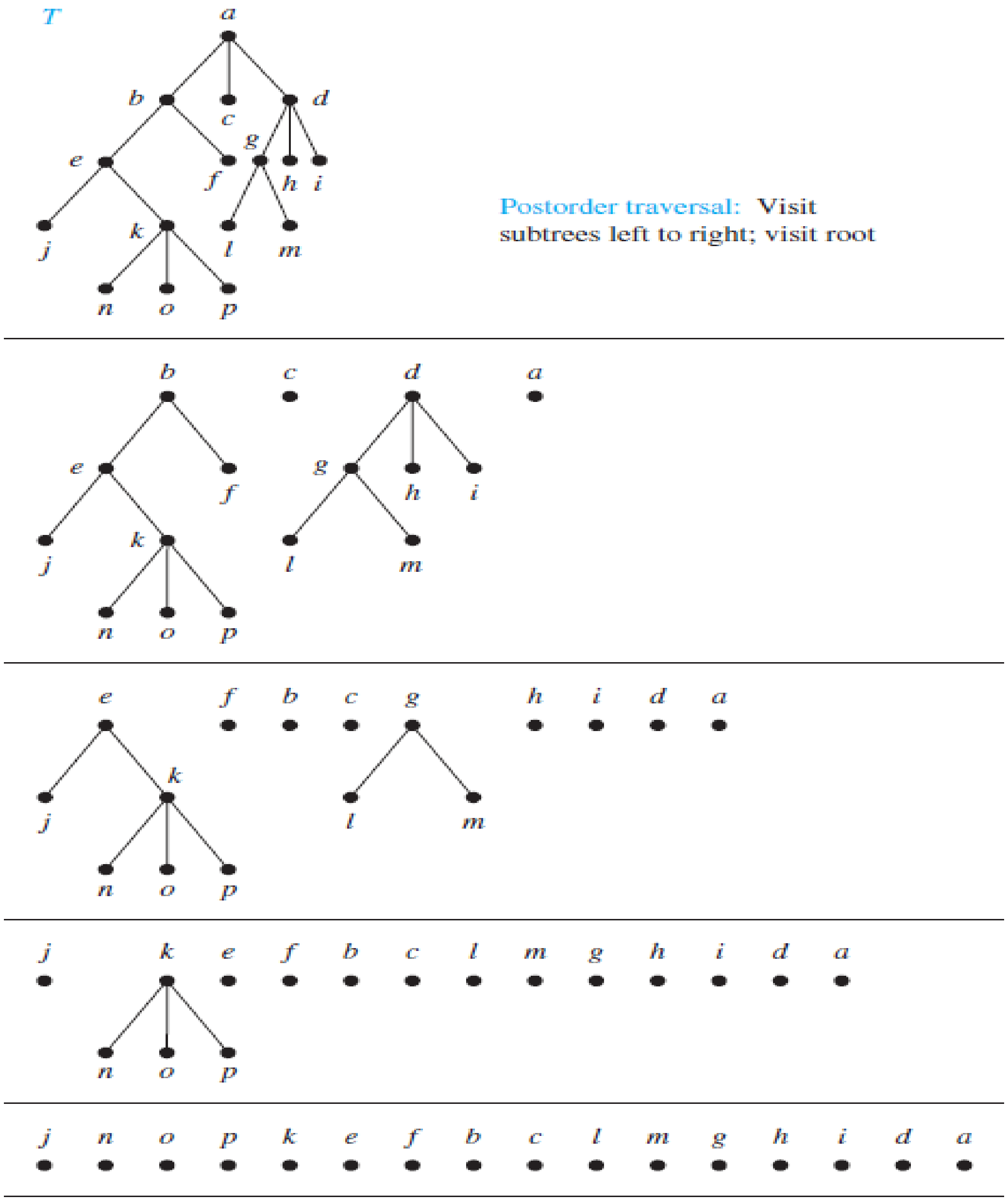


FIGURE 8 The Postorder Traversal of T.

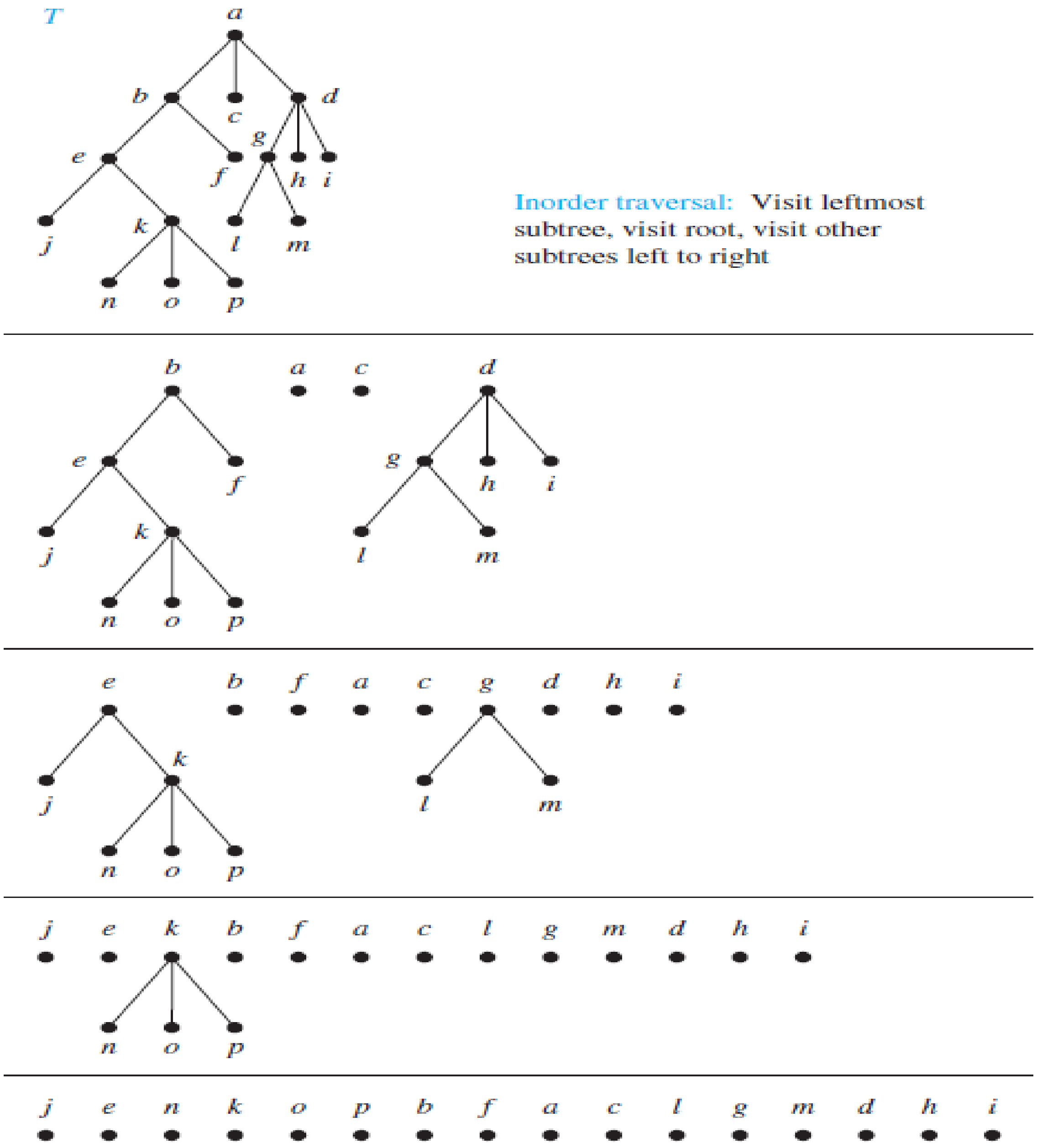
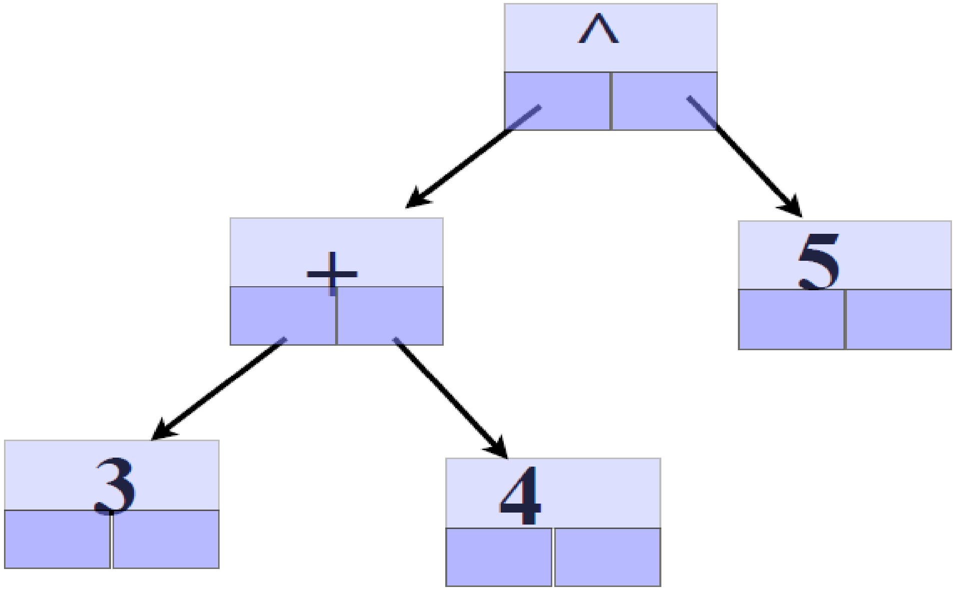


FIGURE 6 The Inorder Traversal of T.

Expression Tree

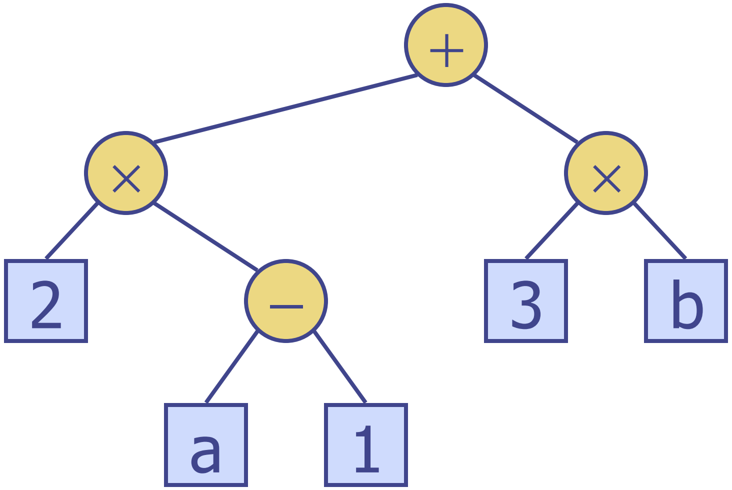
$(3+4)^\wedge 5$



Pre-order Traversal: ?
Post-order Traversal: ?
Level-order Traversal: ?

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



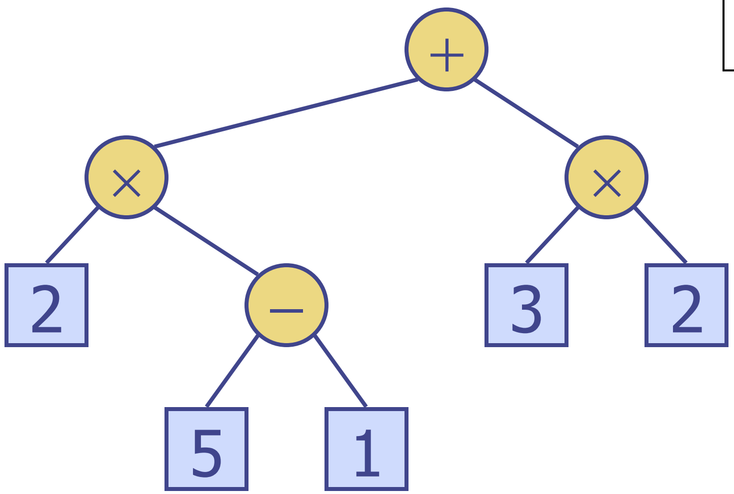
Algorithm *printExpression(v)*
  if *v* **has a left child**
    print(“(’ ’)
    inOrder (*left*(*v*))
  print(*v.element* ())
  if *v* **has a right child**
    inOrder (*right*(*v*))
    print (“)’ ’)

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

Evaluate Arithmetic Expressions

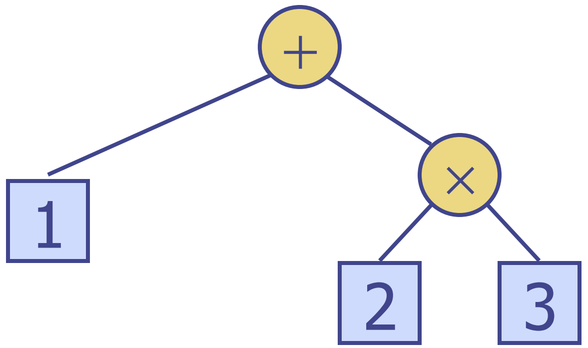
- Specialization of a postorder traversal
  - recursive method returning the value of a subtree
  - when visiting an internal node, combine the values of the subtrees

Algorithm *evalExpr*(v)
    if *is_leaf* (v)
        return *v.element* ()
    else
        $x \leftarrow$ *evalExpr*(*left* (v))
        $y \leftarrow$ *evalExpr*(*right* (v))
        $\Diamond \leftarrow$ operator stored at v
        return $x \Diamond y$



Simple Binary Tree (w/o parent)

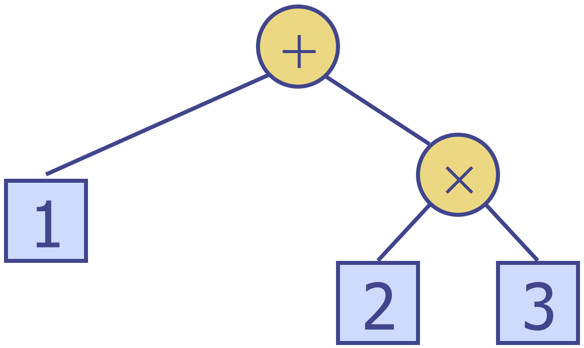
```
class TreeWithoutParent:
    def __init__(self,element,left=None,right=None):
        self._element = element
        self._left = left
        self._right = right
    def __str__(self):
        return str(self._element)
```




```
tree = TreeWithoutParent('+', TreeWithoutParent(1), TreeWithoutParent('*', TreeWithoutParent(2), TreeWithoutParent(3)))
```

Simple Binary Tree (w/o parent)

```
def printTreePreOrder(tree):
    if tree == None:
        return
    print(tree._element,end=" ")
    printTreePreOrder(tree._left)
    printTreePreOrder(tree._right)
```

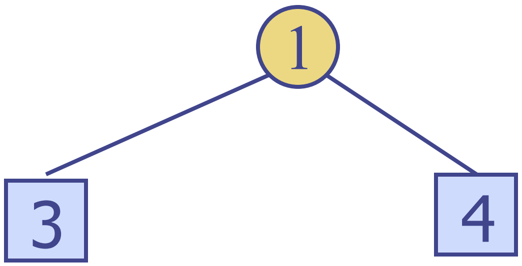



```
def printTreePostOrder(tree):
    if tree == None:
        return
    printTreePostOrder(tree._left)
    printTreePostOrder(tree._right)
    print(tree._element,end=" ")
```

```
def printTreeInOrder(tree):
    if tree == None:
        return
    printTreeInOrder(tree._left)
    print(tree._element,end=" ")
    printTreeInOrder(tree._right)
```

Simple Binary Tree (with parent)

```
class TreeWithParent:
    def __init__(self,element, parent= None, left=None, right=None):
        self._element = element
        self._parent = parent
        self._left = left
        self._right = right
    def __str__(self):
        return str(self._element)
left = TreeWithParent(3)
right = TreeWithParent(4)
tree = TreeWithParent(1,None,left,right)
left._parent = tree
right._parent = tree
```



In-class exercise time

- Download simple_Tree_without_parent_in_class_student.py from Brightspace.
- Create an expression tree for 3*2 + 5-2
- Complete the PreOrderTraversal(tree), PostOrderTraversal(tree) & InOrderTraversal(tree) functions.
- Upload your solution to Gradescope

Tree ADT

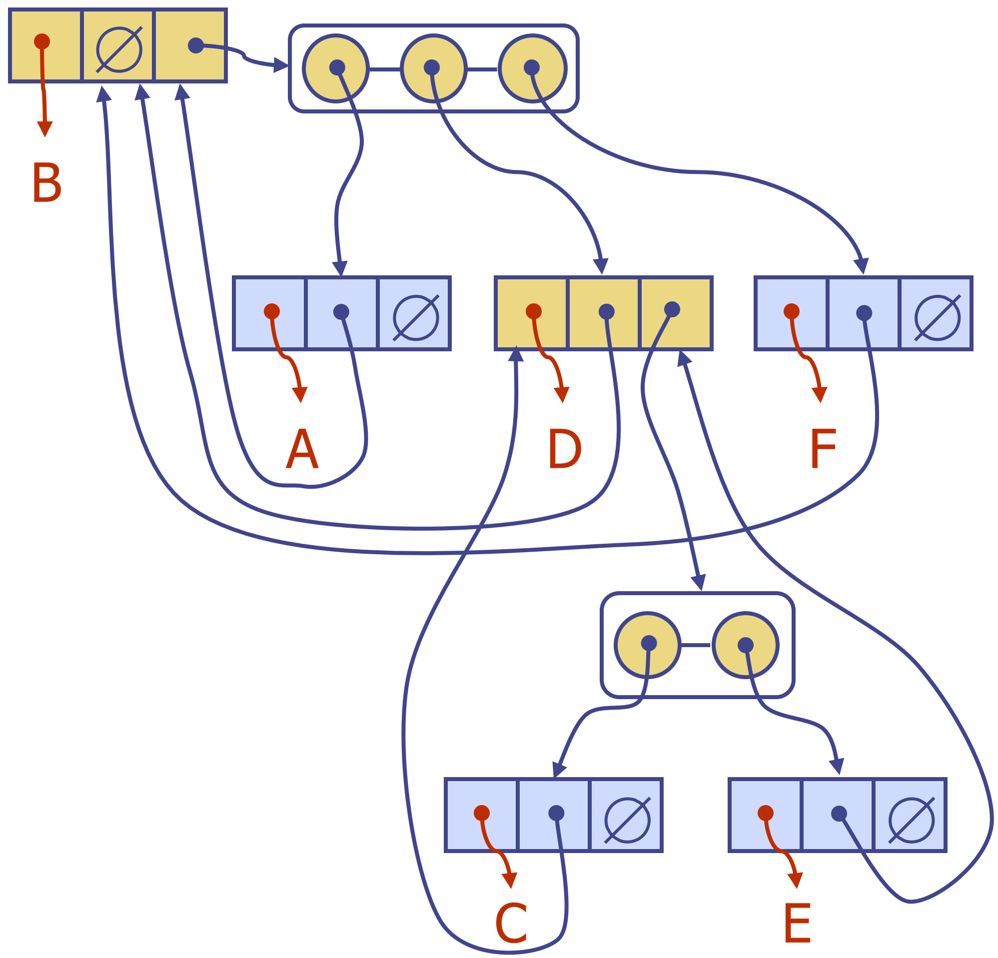
- We use abstract nodes
- Generic methods:
  - Integer len()
  - Boolean is_empty()
  - Iterator nodes()
  - Iterator iter()
- Accessor methods:
  - node root()
  - node parent(node)
  - Iterator children(node)
  - Integer num_children(node)
- Query methods:
  - Boolean is_leaf(node)
  - Boolean is_root(node)
- Update method:
  - element replace (p, o)
- Additional update methods may be defined by data structures implementing the Tree ADT

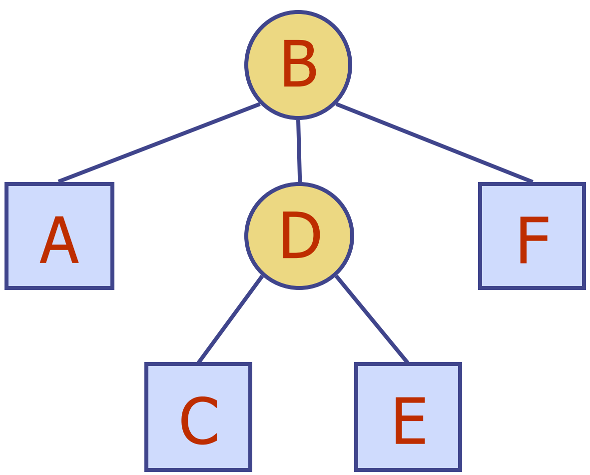
BinaryTree ADT

- The BinaryTree ADT extends the Tree ADT, i.e., it inherits all the methods of the Tree ADT
- Additional methods:
  - node left(node)
  - node right(node)
  - node sibling(node)
- Update methods may be defined by data structures implementing the BinaryTree ADT

Linked Structure for Trees

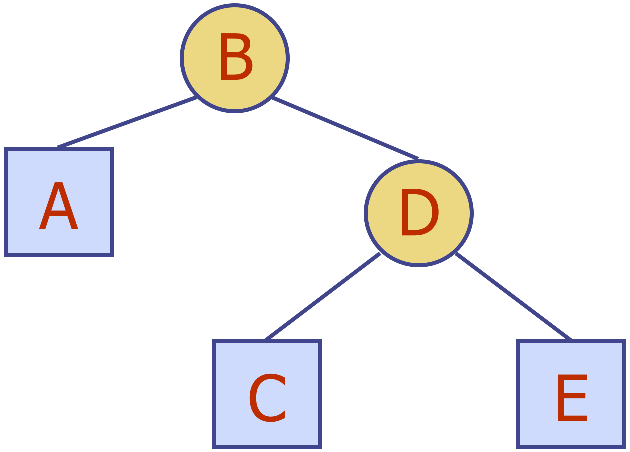
- A node is represented by an object storing
  - Element
  - Parent node
  - Sequence of children nodes

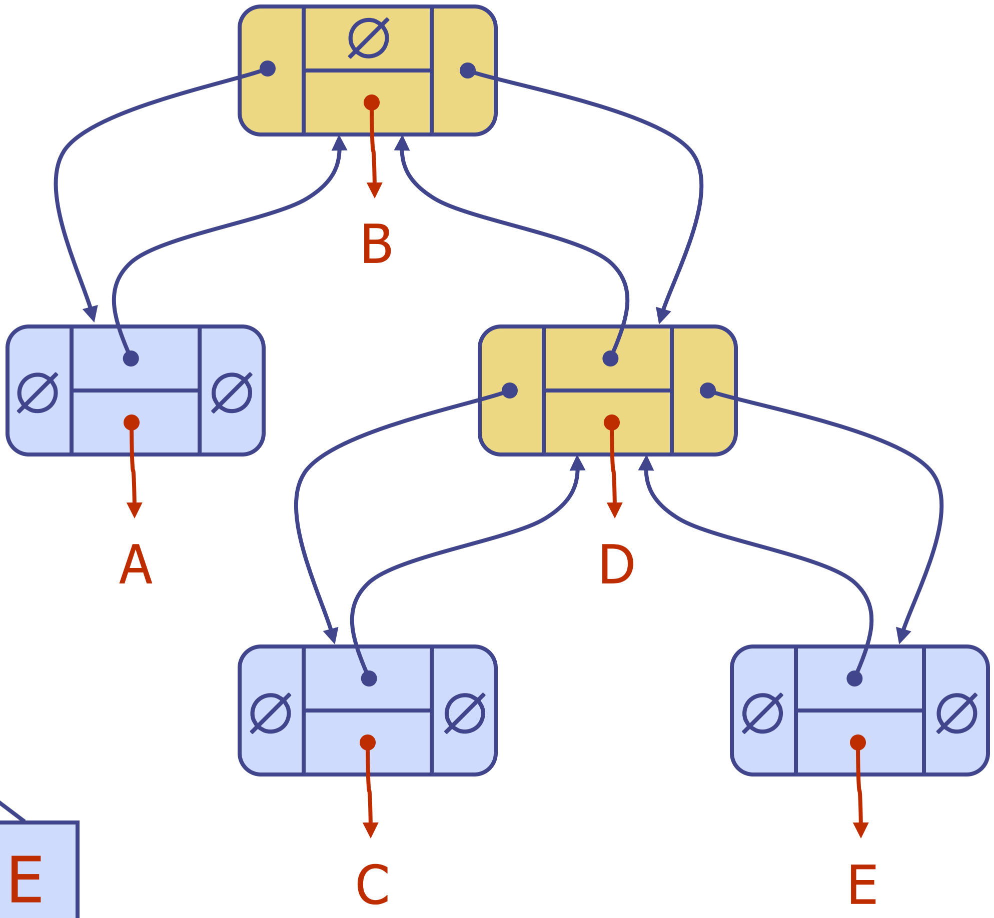




Linked Structure for Binary Trees

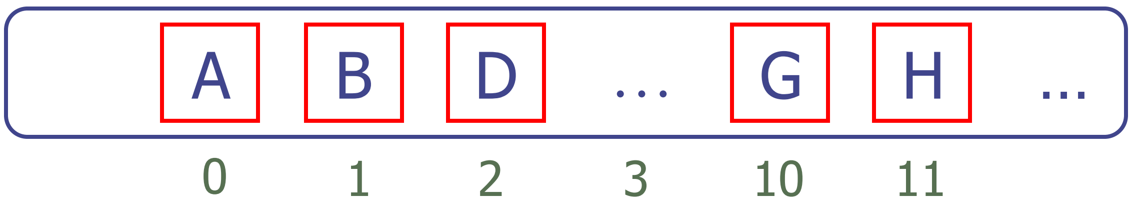
- A node is represented by an object storing
  - Element
  - Parent node
  - Left child node
  - Right child node



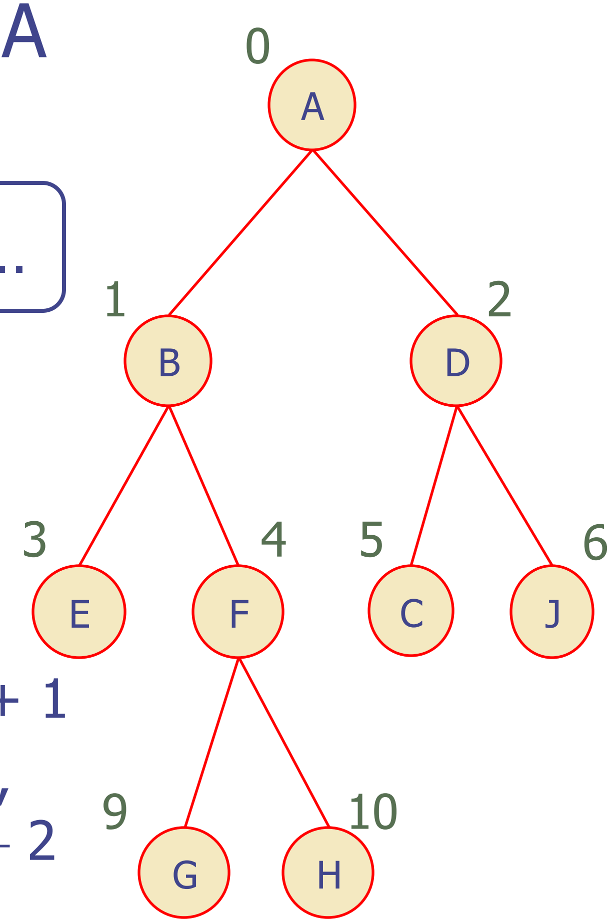


Array-Based Representation of Binary Trees

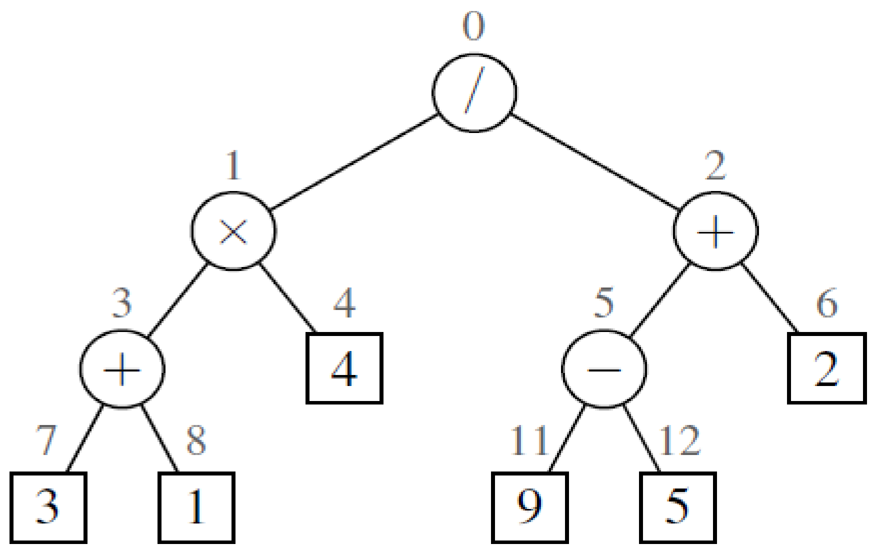
- Nodes are stored in an array A



- 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



Example of using Lists to present Binary Tree



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