COMP1819
Algorithms and Data Structures

Lecture 06: Trees

Dr. Tuan Vuong

23/02/2021

Content

- Review Lab 05
- General Trees
- Binary Trees
- Implementing Trees
- Tree Traversal Algorithms
- Reinforcement

Lab 05

1. Improve Bubble Sort

The given code (lecture/github) always runs O(n^2) time even if the array is sorted. It can be optimized by stopping the algorithm if inner loop didn't cause any swap.

Examples:

Input	Output
`1 3 10 18`	`Bubble sort function stops after 1 pass.`

Hints

- Code for these searches given in the lecture slides

2. Compare Bubble sort, Selection sort and Insertion sort

Create a large array, compare the time complexity of three sort algorithms with these cases:

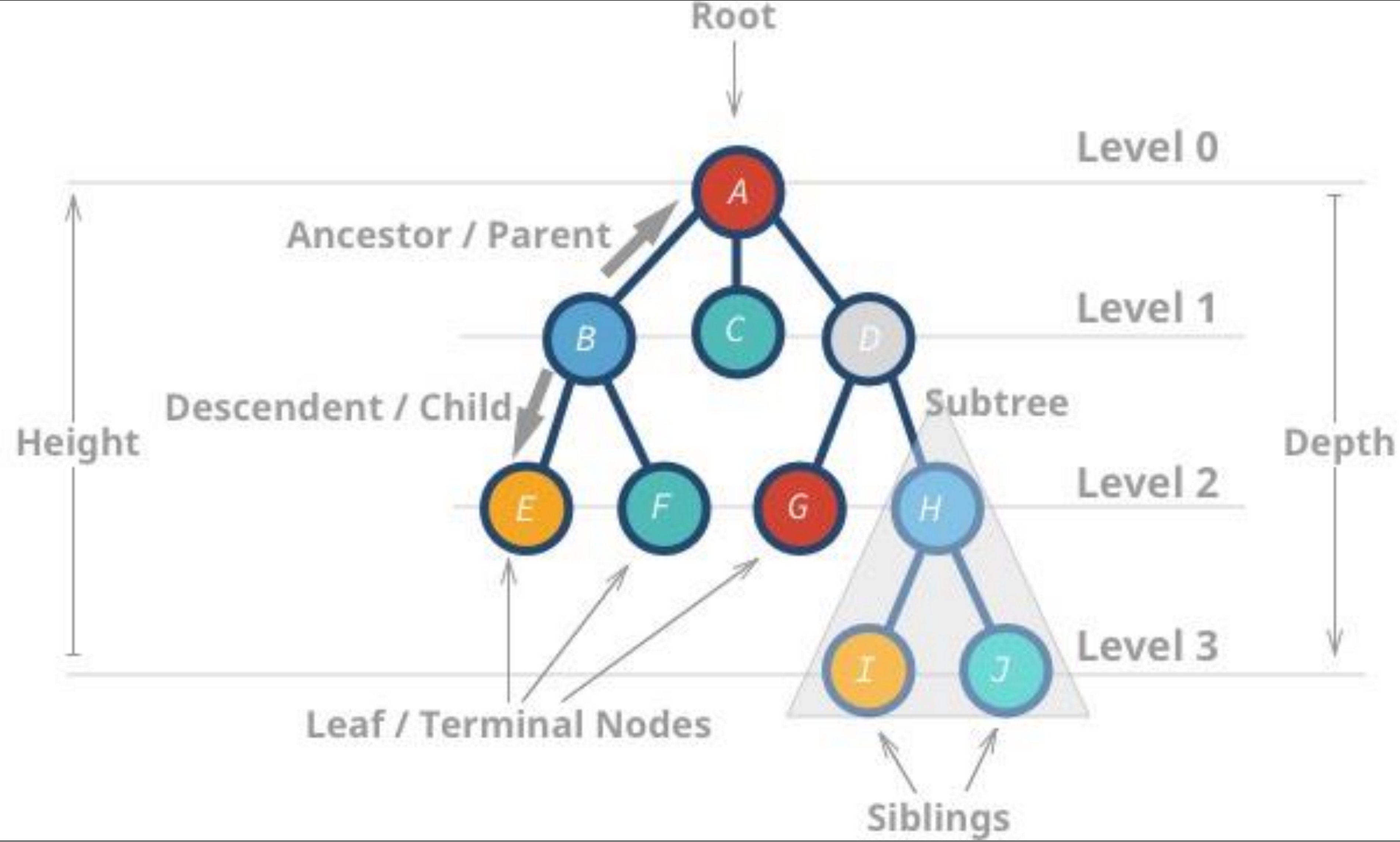
- Sorted list (ascending order)
- Sorted list (descending order)
- Random list

Which one is faster?

Today

[Tree link]

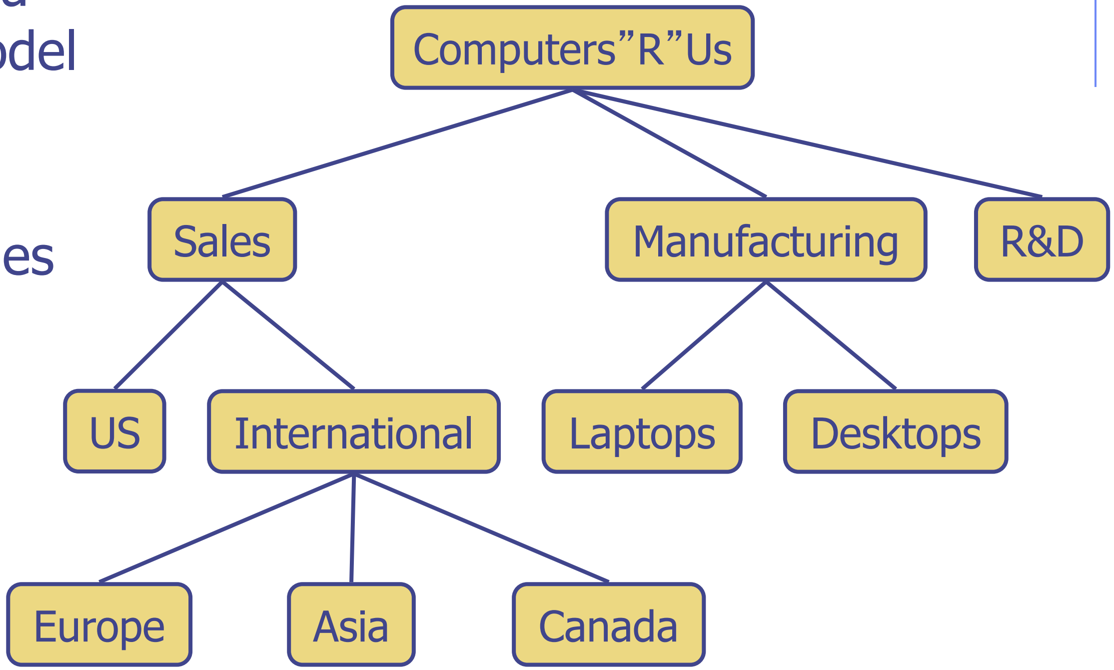
This one



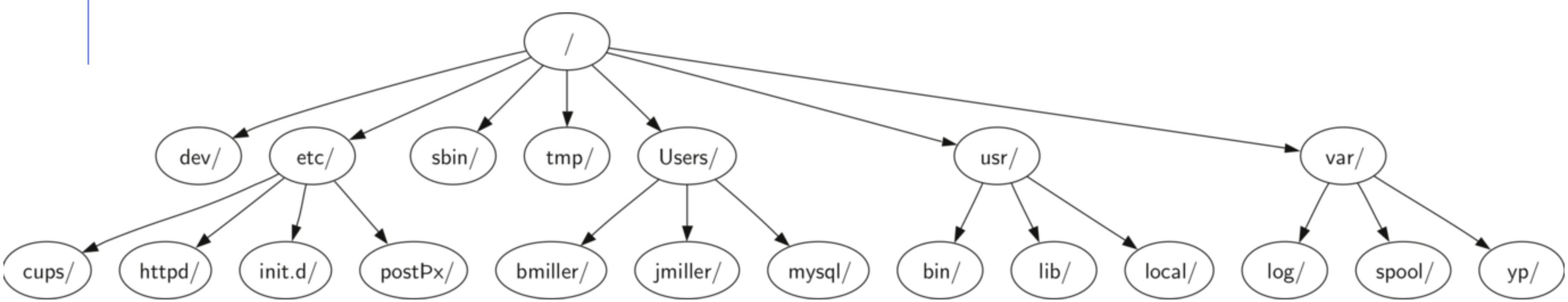
Root
Level 0
A
Ancestor / Parent
B
C
D
Level 1
Descendent / Child
Subtree
Height
Depth
E
F
G
H
Level 2
I
J
Level 3
Leaf / Terminal Nodes
Siblings

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

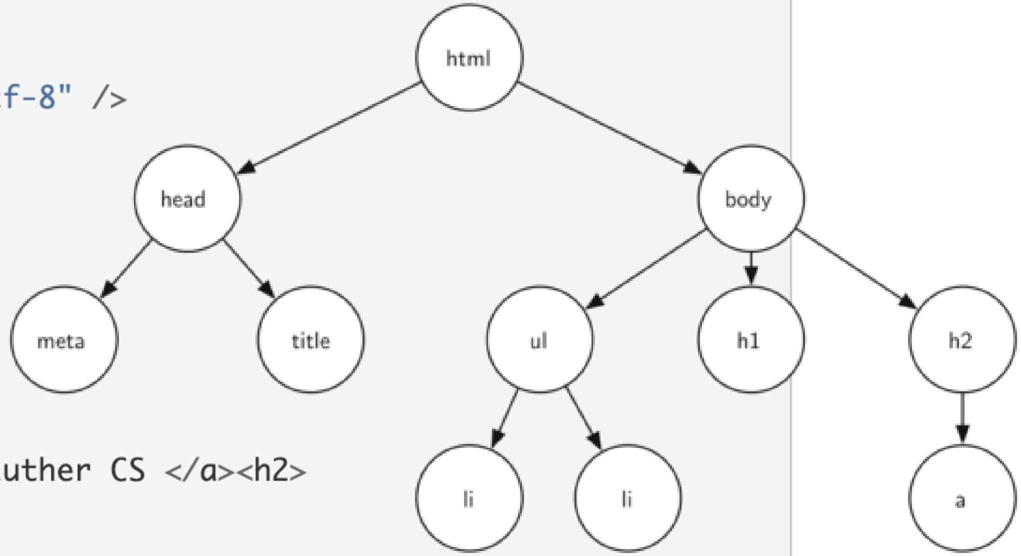


Example: Unix File System Hierarchy



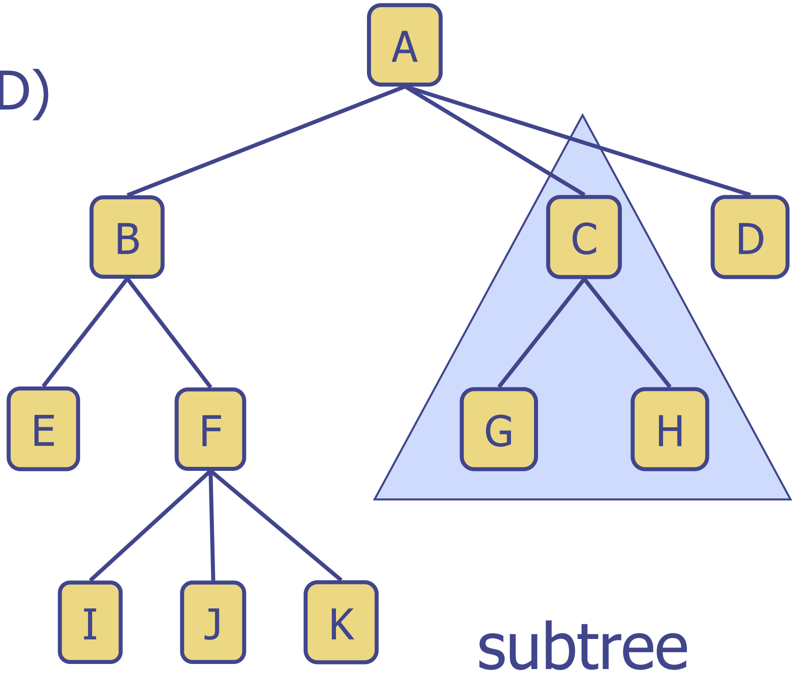
Example: HTML webpage

```
<html xmlns="http://www.w3.org/1999/xhtml"
      xml:lang="en" lang="en">
<head>
    <meta http-equiv="Content-Type"
          content="text/html; charset=utf-8" />
    <title>simple</title>
</head>
<body>
<h1>A simple web page</h1>
<ul>
    <li>List item one</li>
    <li>List item two</li>
</ul>
<h2><a href="http://www.cs.luther.edu">Luther CS </a><h2>
</body>
</html>
```

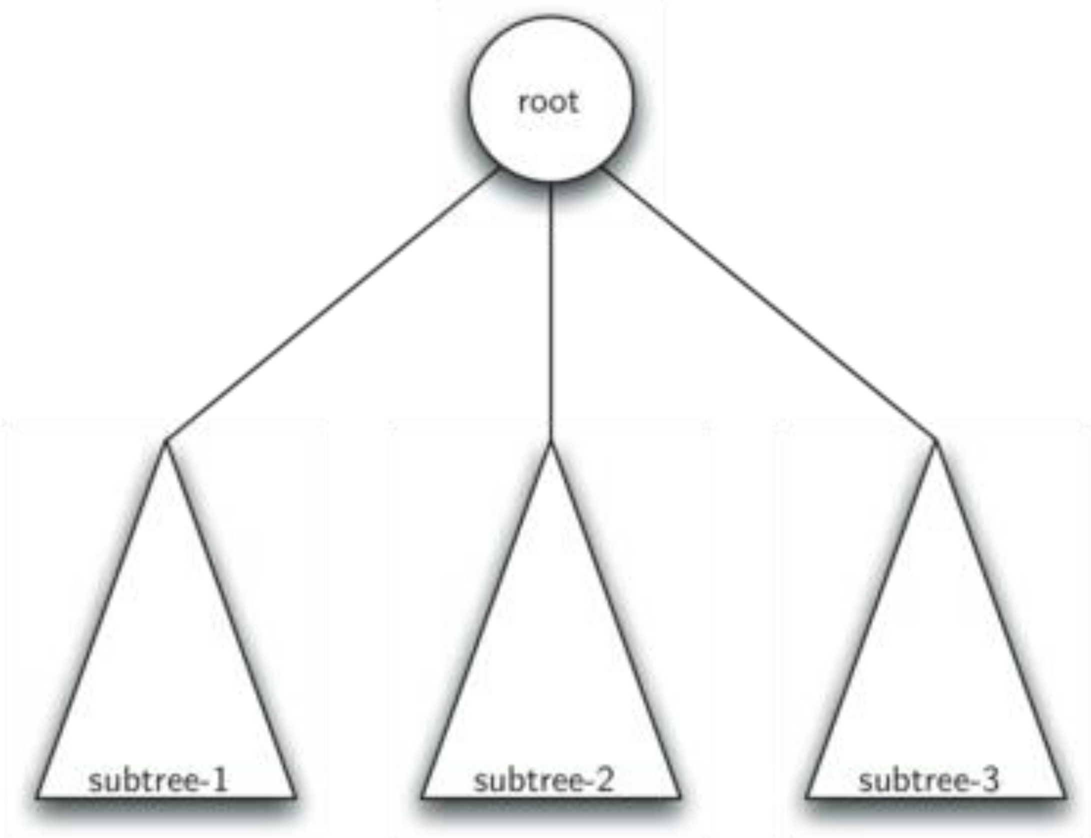


Tree Terminology

- 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



Recursive definition



A tree is either empty or consists of a root and zero or more subtrees, each of which is also a tree. The root of each subtree is connected to the root of the parent tree by an edge.

Tree ADT

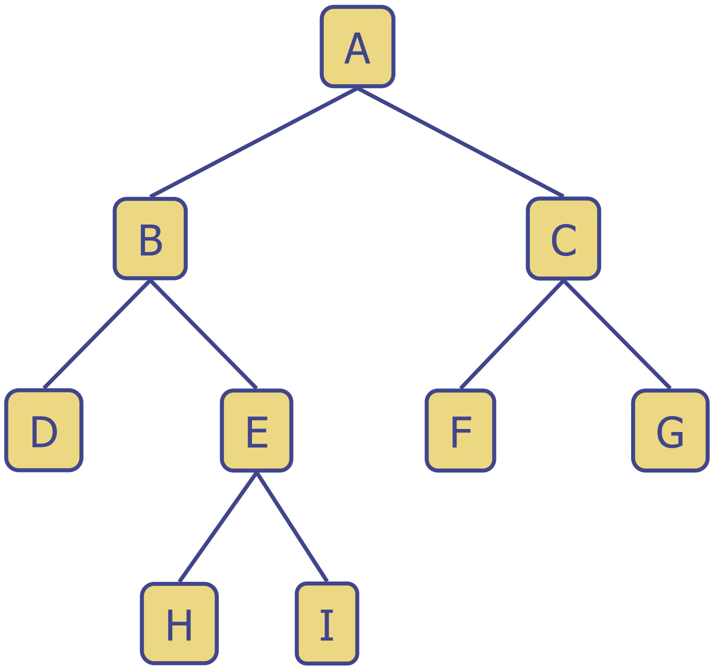
- We use positions to abstract nodes
- Generic methods:
  - Integer len()
  - Boolean is_empty()
  - Iterator positions()
  - Iterator iter()
- Accessor methods:
  - position root()
  - position parent(p)
  - Iterator children(p)
  - Integer num_children(p)

- Query methods:
  - Boolean is_leaf(p)
  - Boolean is_root(p)
- Update method:
  - element replace (p, o)
- Additional update methods may be defined by data structures implementing the Tree ADT

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

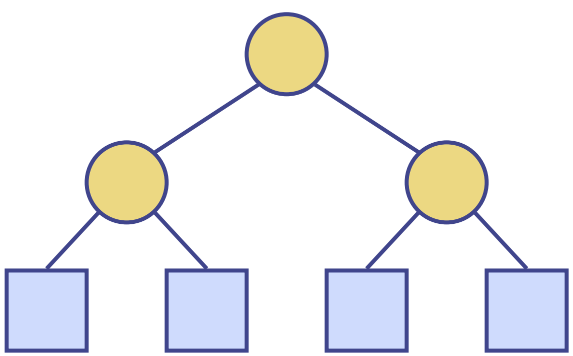
- Applications:
  - arithmetic expressions
  - decision processes
  - searching

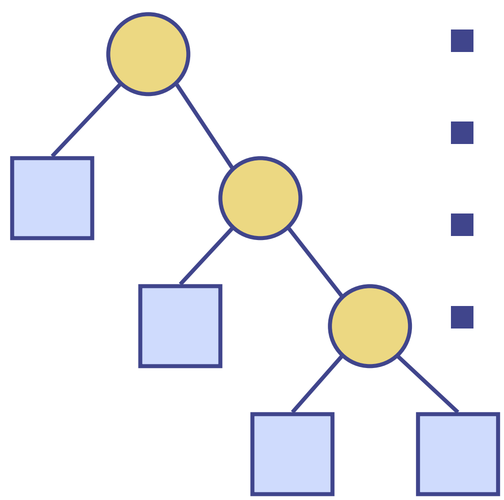


Properties of Proper Binary Trees

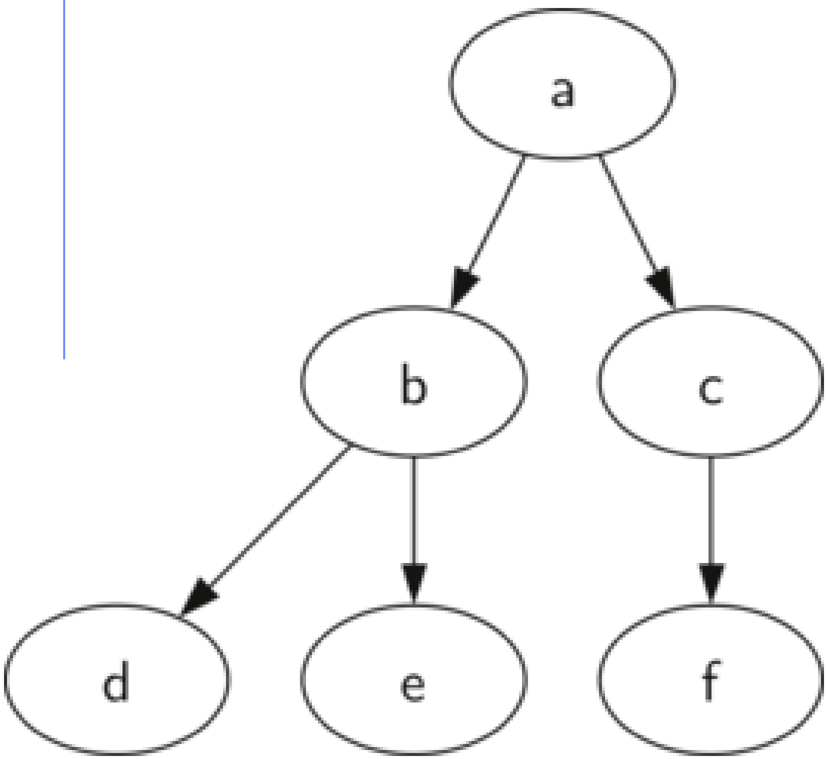
- Notation
  - n number of nodes
  - e number of external nodes
  - i number of internal nodes
  - h height

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





Tree by a list of lists




```
myTree = ['a',   #root
      ['b',  #left subtree
       ['d', [], []],
       ['e', [], []] ],
      ['c',  #right subtree
       ['f', [], []],
       [] ]
     ]
```

Example in github (myTree.py)

```python
'''
        This is an example of a binary tree data structure created with
        python lists as the underlying data structure.
'''

def BinaryTree(r):
        return [r, [], []]

def insertLeft(root,newBranch):
        t = root.pop(1)
        if len(t) > 1:
                root.insert(1,[newBranch,t,[]])
        else:
                root.insert(1,[newBranch, [], []])
        return root

def insertRight(root,newBranch):
        t = root.pop(2)
        if len(t) > 1:
                root.insert(2,[newBranch,[],t])
        else:
                root.insert(2,[newBranch,[],[]])
        return root

def getRootVal(root):
        return root[0]

def setRootVal(root,newVal):
        root[0] = newVal

def getLeftChild(root):
        return root[1]

def getRightChild(root):
        return root[2]


r = BinaryTree(3)
insertLeft(r,4)
insertLeft(r,5)
insertRight(r,6)
insertRight(r,7)
l = getLeftChild(r)
print(l)
```

insertLeft: first obtain the (possibly empty) list that corresponds to the current left child. Then add the new left child, installing the old left child as the left child of the new one.

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)
- Update methods may be defined by data structures implementing the BinaryTree ADT

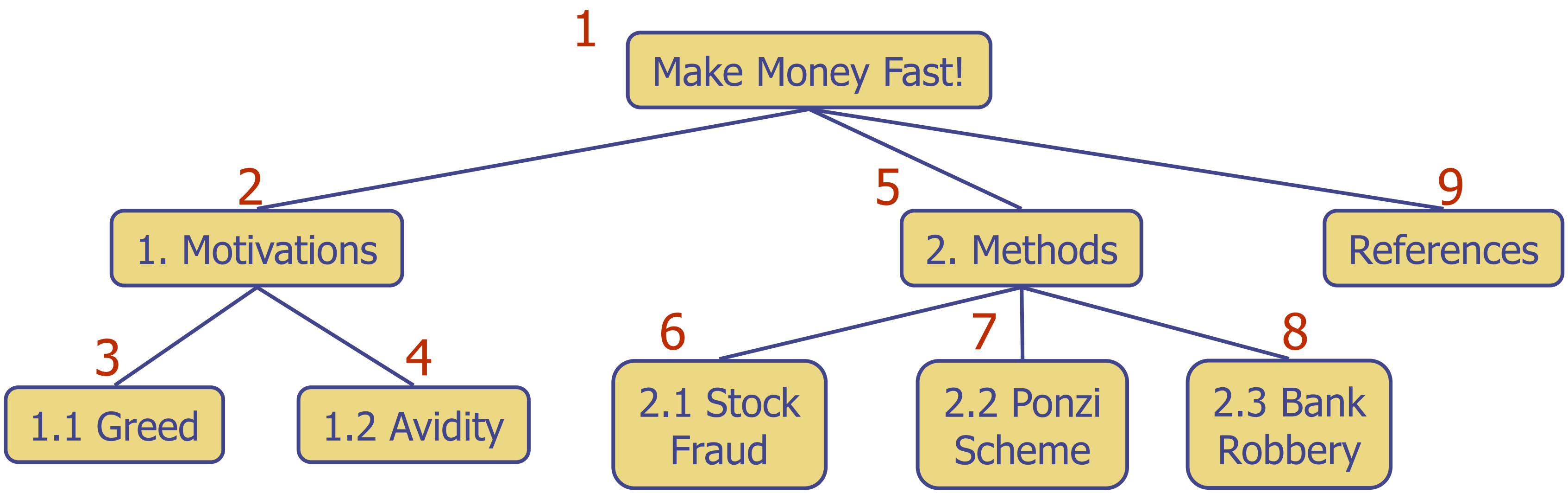
ADT Binary Tree Class

```
class BinaryTree:
    def __init__(self,rootObj):
        self.key = rootObj
        self.leftChild = None
        self.rightChild = None

    def insertLeft(self,newNode):
        if self.leftChild == None:
            self.leftChild = BinaryTree(newNode)
        else:
            t = BinaryTree(newNode)
            t.leftChild = self.leftChild
            self.leftChild = t

    def insertRight(self,newNode):
        if self.rightChild == None:
            self.rightChild = BinaryTree(newNode
        else:
            t = BinaryTree(newNode)
            t.rightChild = self.rightChild
            self.rightChild = t

    def getRightChild(self):
        return self.rightChild

    def getLeftChild(self):
        return self.leftChild

    def setRootVal(self,obj):
        self.key =obj

    def getRootVal(self):
        return self.key
```

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

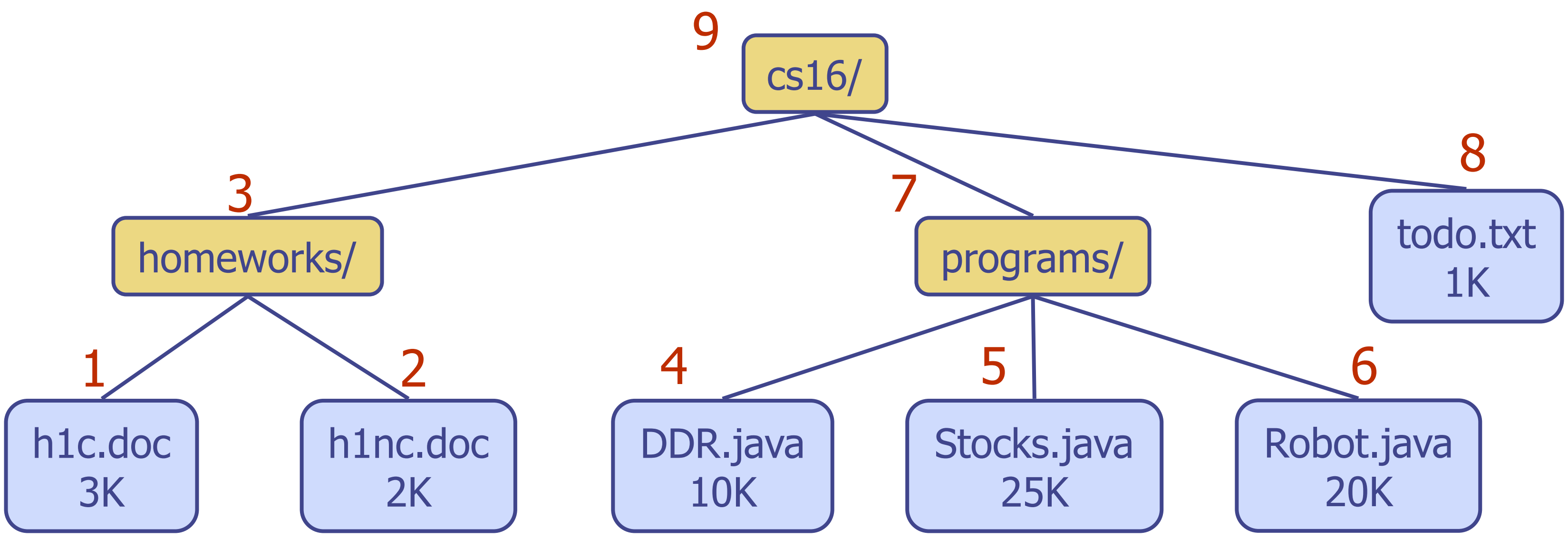
Algorithm *preOrder*(v)
    visit(v)
    for each child w of v
        preorder (w)



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)
```

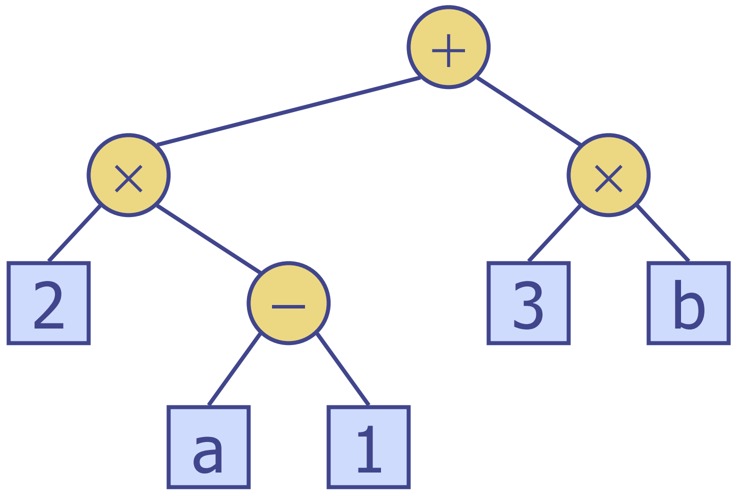


Preoder & Postorder for Binary Tree

```
def preorder(self):
        print(self.key)
        if self.leftChild:
                self.leftChild.preorder()
        if self.rightChild:
                self.rightChild.preorder()

def postorder(self):
        if self.leftChild:
                self.leftChild.postorder()
        if self.rightChild:
                self.rightChild.postorder()
        print(self.key)
```

Arithmetic Expression Tree

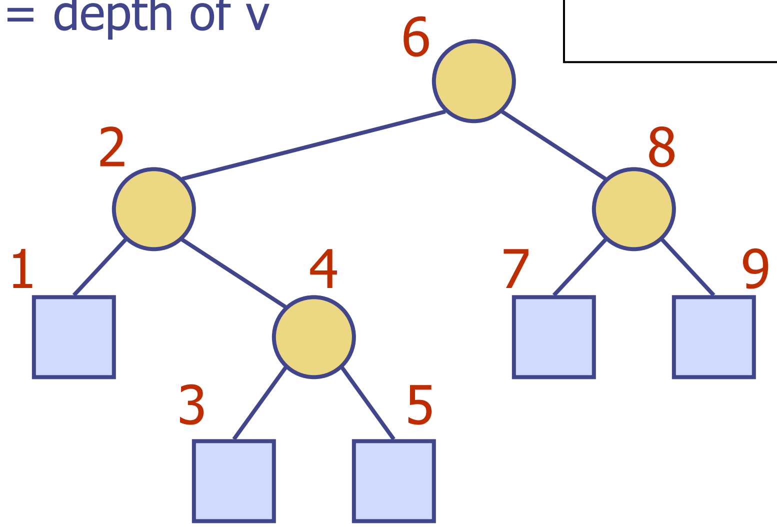
- Binary tree associated with an arithmetic expression
  - internal nodes: operators
  - external nodes: operands
- Example: arithmetic expression tree for the expression $(2 \times (a - 1) + (3 \times b))$



Inorder Traversal

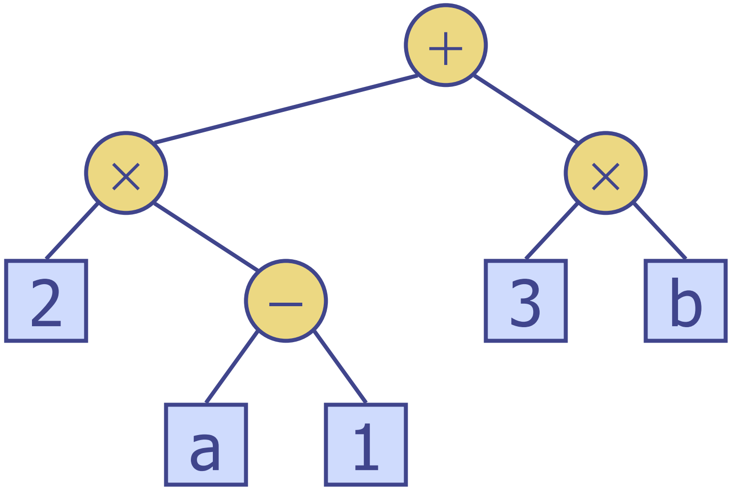
- In an inorder traversal a node is visited after its left subtree and before its right subtree
- Application: draw a binary tree
  - x(v) = inorder rank of v
  - y(v) = depth of v

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

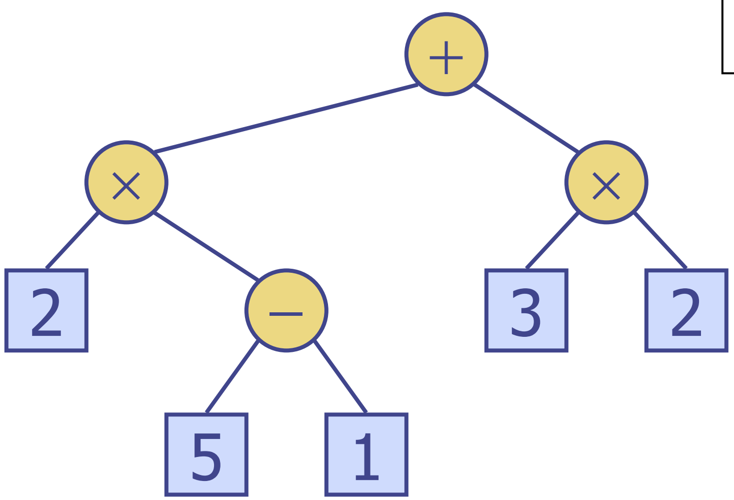


Algorithm *printExpression*(*v*)
  if *v* **has a left child**
    print(“(’ ’)
    inOrder (*left*(*v*))
  print(*v.element* ())
  if *v* **has a right child**
    inOrder (*right*(*v*))
    print (“)’ ’)

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

Analysis of Algorithms

- What is the best/worst/average case? When?
- Big-O performance

Reinforcement

Discussion 2

Draw the tree structure resulting from the following set of tree function calls:

```
>>> r = BinaryTree(3)
>>> insertLeft(r,4)
[3, [4, [], []], []]
>>> insertLeft(r,5)
[3, [5, [4, [], []], []], []]
>>> insertRight(r,6)
[3, [5, [4, [], []], []], [6, [], []]]
>>> insertRight(r,7)
[3, [5, [4, [], []], []], [7, [], [6, [], []]]]
>>> setRootVal(r,9)
>>> insertLeft(r,11)
[9, [11, [5, [4, [], []], []], []], [7, [], [6, [], []]]]
```

Self check:

```
x = BinaryTree('a')
insertLeft(x,'b')
insertRight(x,'c')
insertRight(getRightChild(x),'d')
insertLeft(getRightChild(getRightChild(x)),'e')
```

Which of the answers is the correct representation of the tree?

- A. ['a', ['b', [], []], ['c', [], ['d', [], []]]]
- B. ['a', ['c', [], ['d', ['e', [], []], []]], ['b', [], []]]
- C. ['a', ['b', [], []], ['c', [], ['d', ['e', [], []], []]]]
- D. ['a', ['b', [], ['d', ['e', [], []], []]], ['c', [], []]]

Using binary tree data structure created with python lists.

Discussion

Consider the following list of integers: [1,2,3,4,5,6,7,8,9,10]. Show the binary search tree resulting from inserting the integers in the list.

Discussion

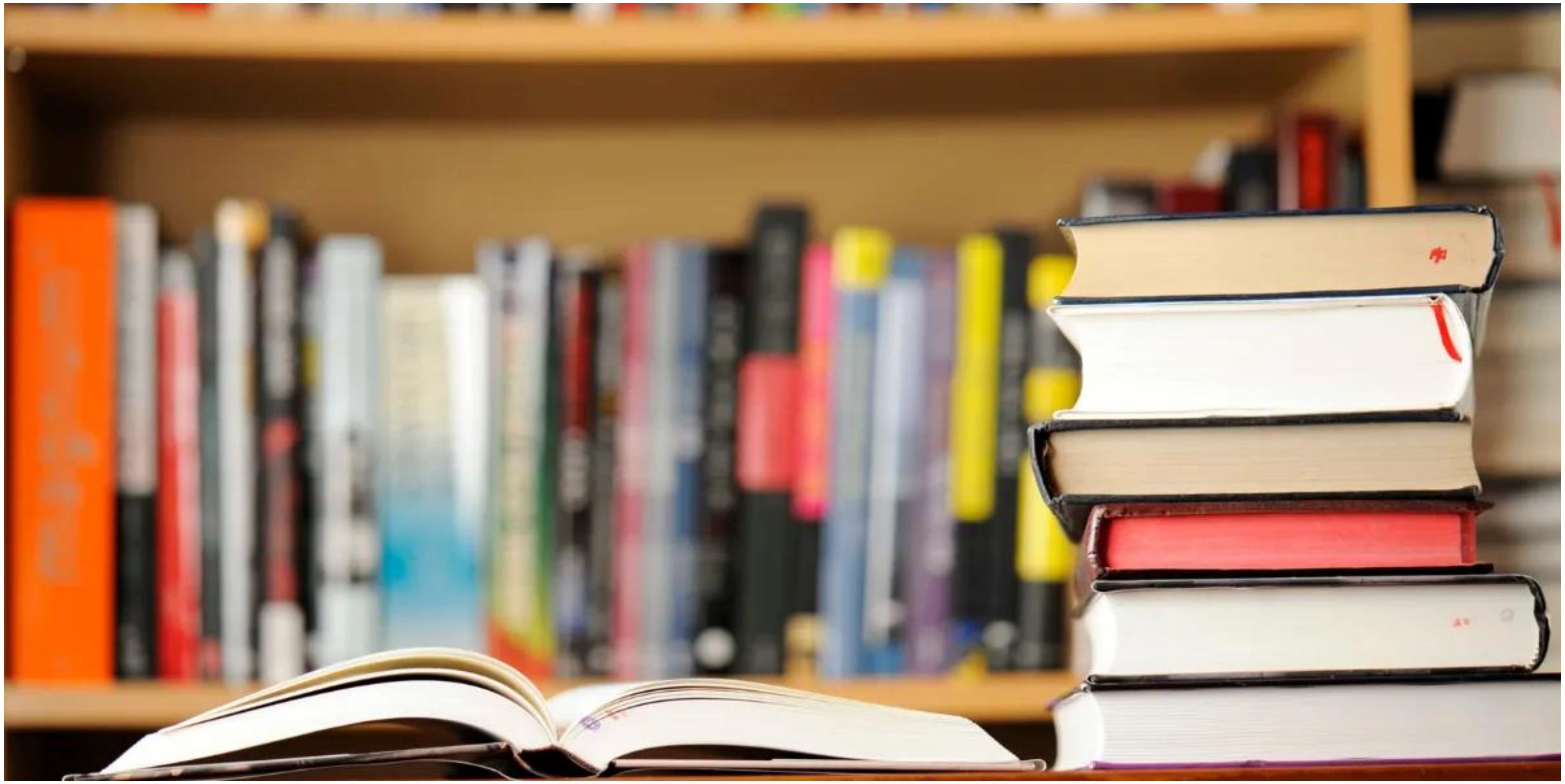
Trace the algorithm for creating an expression tree for the expression $(4*8)/6-3(4*8)/6-3$.

Quick overview

- Trees
- Binary Tree with Python list
- Binary Tree with ADT class
- Binary Tree for parsing and evaluating

Extra reading

- Binary Search Tree (BTS)
- Balanced BTS



Next week