COSC 2306
Data Programming

Trees

Grades

- Another 1.25 bonus credits for assignment
- Already gave 5 free credits to curve the quiz
- 2 bonus credits for Course Evaluation
- We have a total Bonus Credits of **13+**: 2 for Course Evaluation, 1 for Course Survey, 5 for Quizzes + curving for the first 3 quizzes, 5+ for Assignment, unlimited in-class activity credits
- This means even if you lost **23** points, you could still get A if you get all the bonus credits (not even counting curving and in-class activity)
- Some examples:
  - Case 1: even if you did not attend a single class (get 0 for quiz) but get all bonus, you may still get A
  - Case 2: even if you only get 1/3 of quiz questions answered correctly and lose several points in other parts, you still have a good chance to get A
  - Case 3: even if you only get half quiz questions answered correctly, and you lose half points in exam, lose almost half points in homework, but get all bonus, you may still get C or even C+

Exam 2 (notice: no make-up)

- Date: **Dec 3** during the class time
- Duration: **60 mins** (but will give you **full class time**)
- Questions: multiple choices, coding
- Location: testing center seems challenging, may still in classroom
- Open-booked, but **only paper-based materials** such as notes/slides/books/documents are allowed
- Same as the quiz, except the laptop running Respondus, no access to any electronic devices/internet/AI, no discussion/communication in any kind
- You need to sit at least 3 feet from your neighboring students during the exam.
- If you have questions or issues, please only talk to the person supervising the exam -- never talk to other students.

A tree-like organization

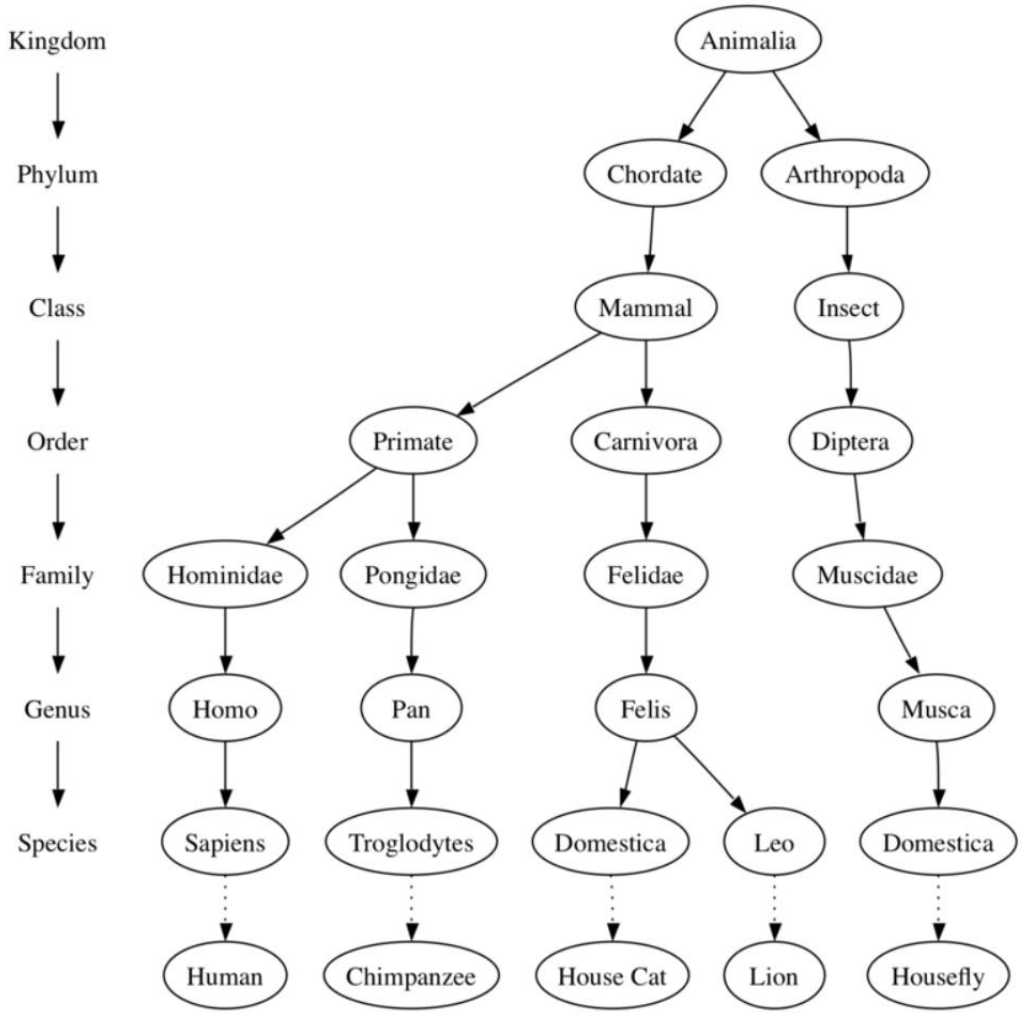


Figure 1: Taxonomy of Some Common Animals Shown as a Tree

A tree-like organization

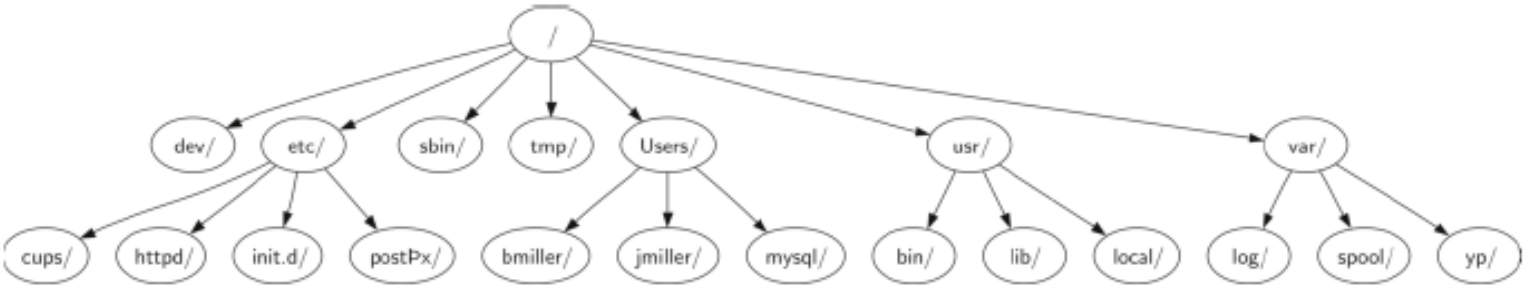
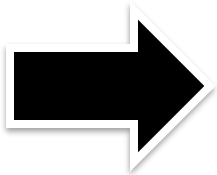
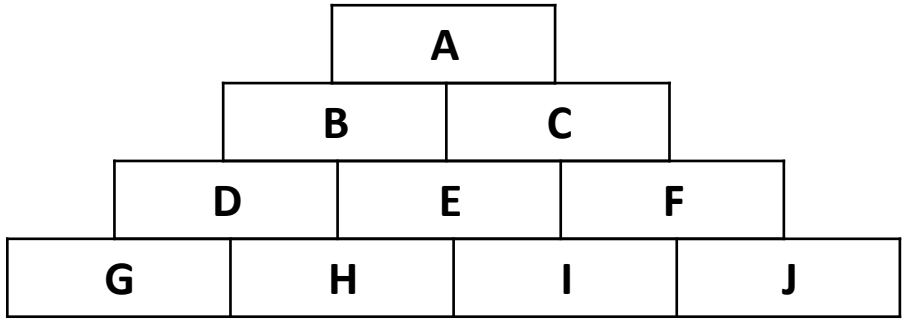


Figure 6.2: A Small Part of the Unix File System Hierarchy

Organizing data

A
B
C
D
E
F
G
H
I
J



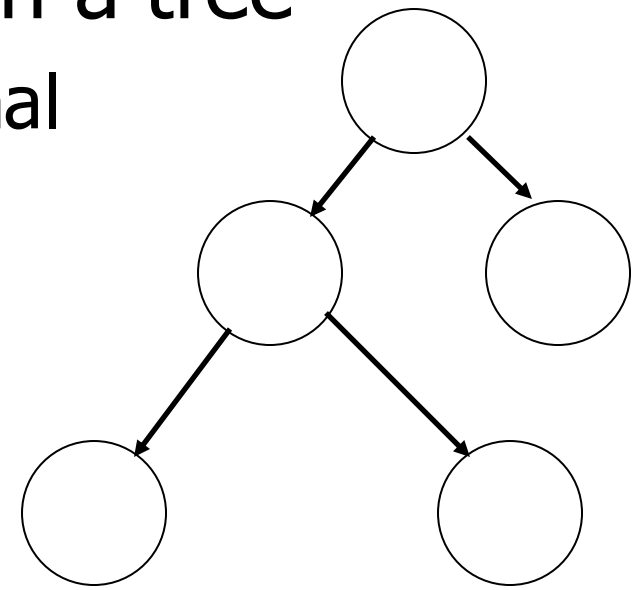


Application of trees

- Database system: Binary Search Tree
- File system: B-tree B+-tree
- Encoding and Compression: Huffman Tree
- Data structures: Red-black Tree
  - C++ set and map (dict) implementation
- Machine Learning: Decision Tree
- Data Mining: K-D Tree

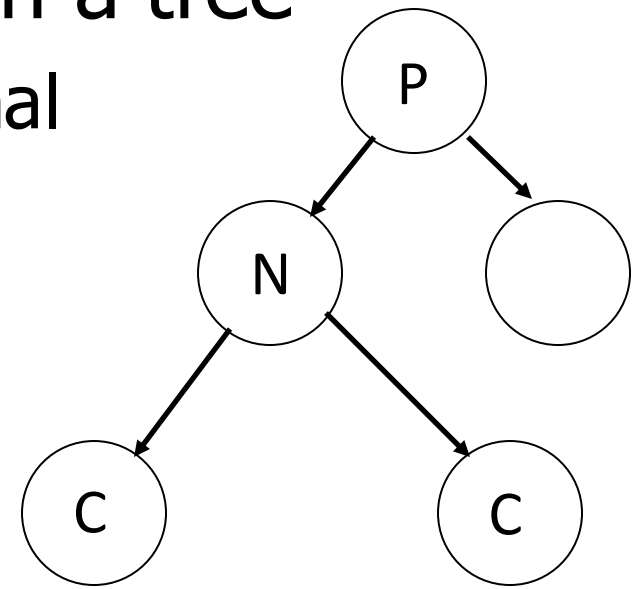
Tree node

- The basic component of tree
- Connected to each other in a tree
  - Connection is one-directional
- For each node N:
  - Children nodes
    - The nodes N connect to
  - Parent node
    - The node connect to N



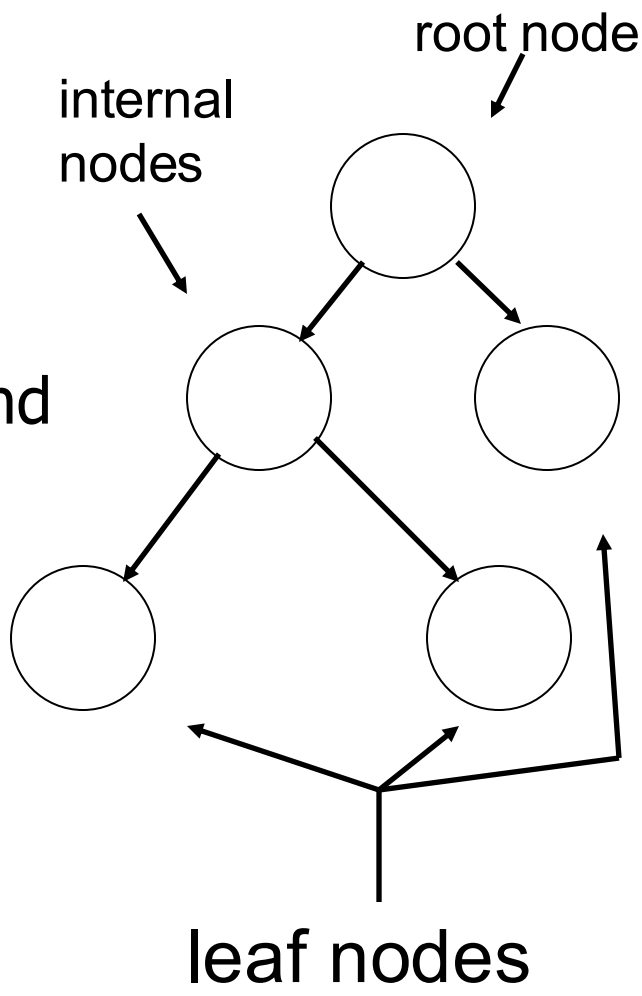
Tree node

- The basic component of tree
- Connected to each other in a tree
  - Connection is one-directional
- For each node N:
  - Children nodes
    - The nodes N connect to
  - Parent node
    - The node connect to N



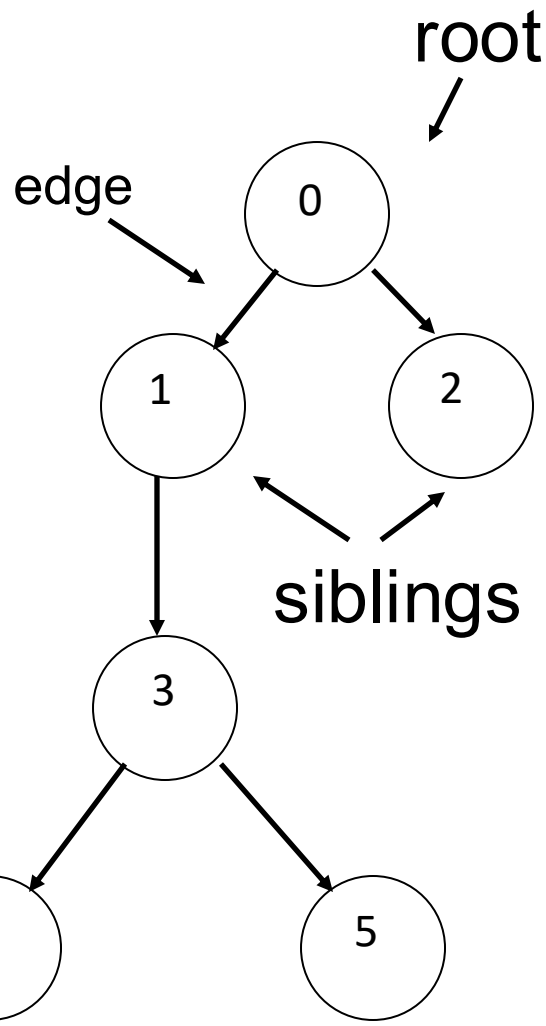
Tree node

- 3 node types:
  - Root node
    - Only 1 in each tree
  - Internal node
    - Node that has both parent and children
  - Leaf node
    - Node that has no children



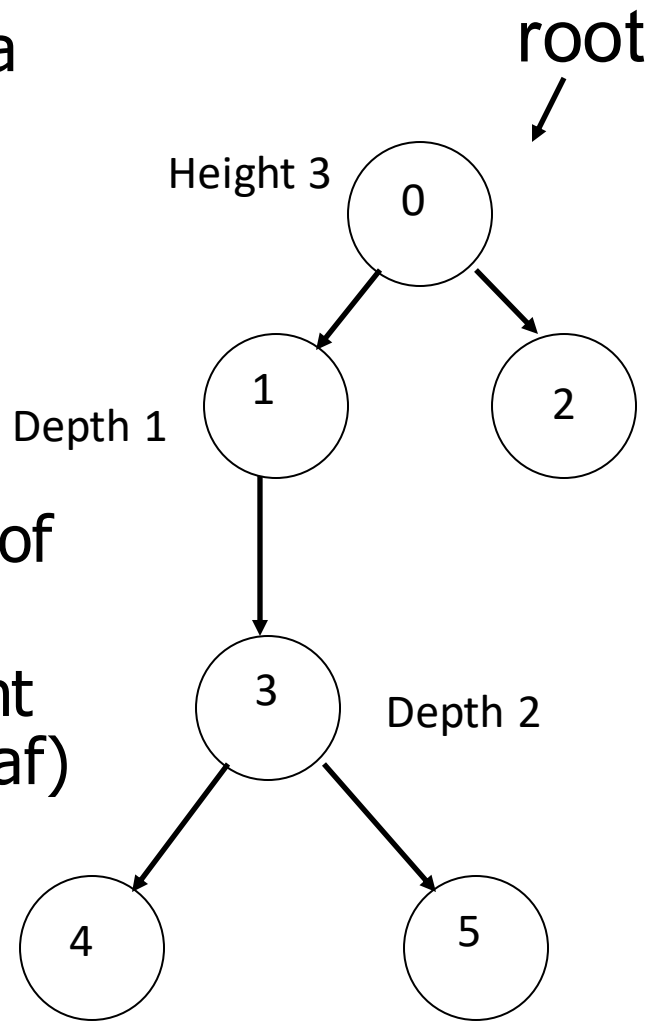
Tree vocabularies

- Siblings
  - two nodes that have the same parent
- Edge
  - the link from one node to another
- Path length
  - the number of edges between one node and another



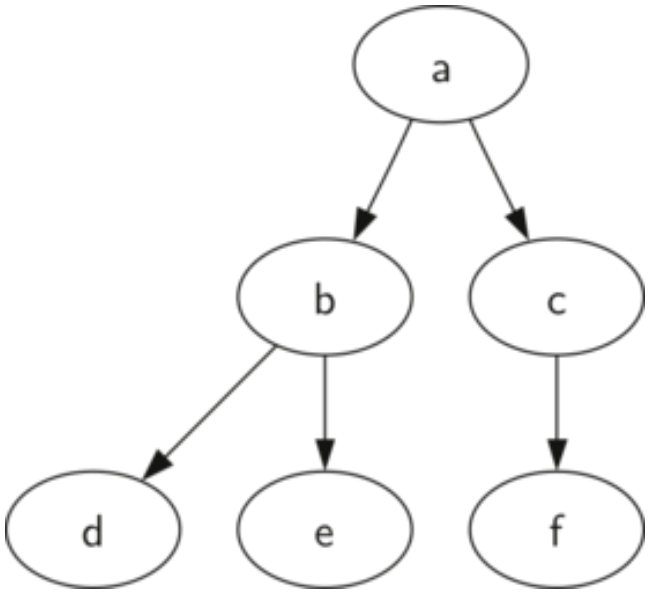
Tree vocabularies

- Depth
  - the path length from the root to a node
- Height
  - the maximum path length of any leaf from this node
  - a leaf has a height of 0
  - the height of a tree is the height of the root of that tree
  - if a tree has only one node, height is 0 (the node is both root and leaf)



Subtree

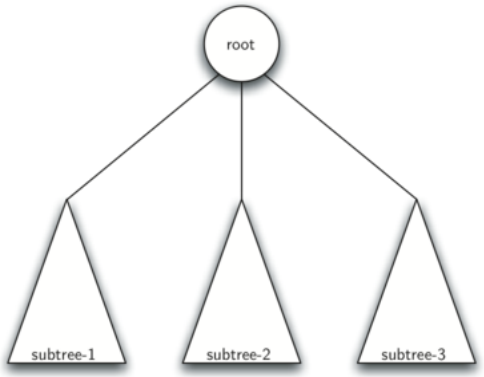
Subtree: a set of nodes and edges of a parent and all descendants of that parent (e.g., b, d, e)



Definition

- Definition1: a tree, *T*, is consists a set of nodes and edges that connect pairs of nodes
  - *T* has a special node called the root node
  - *Every node except the root is connected by an edge from exactly one other node (its parent)*
  - *A unique path from root to each node*

- Definition2: a tree, *T*, is consist of a root and 0 or more subtrees
  - *Recursive definition*
  - *Subtree is a tree*
  - *Example shows at least 4 nodes*



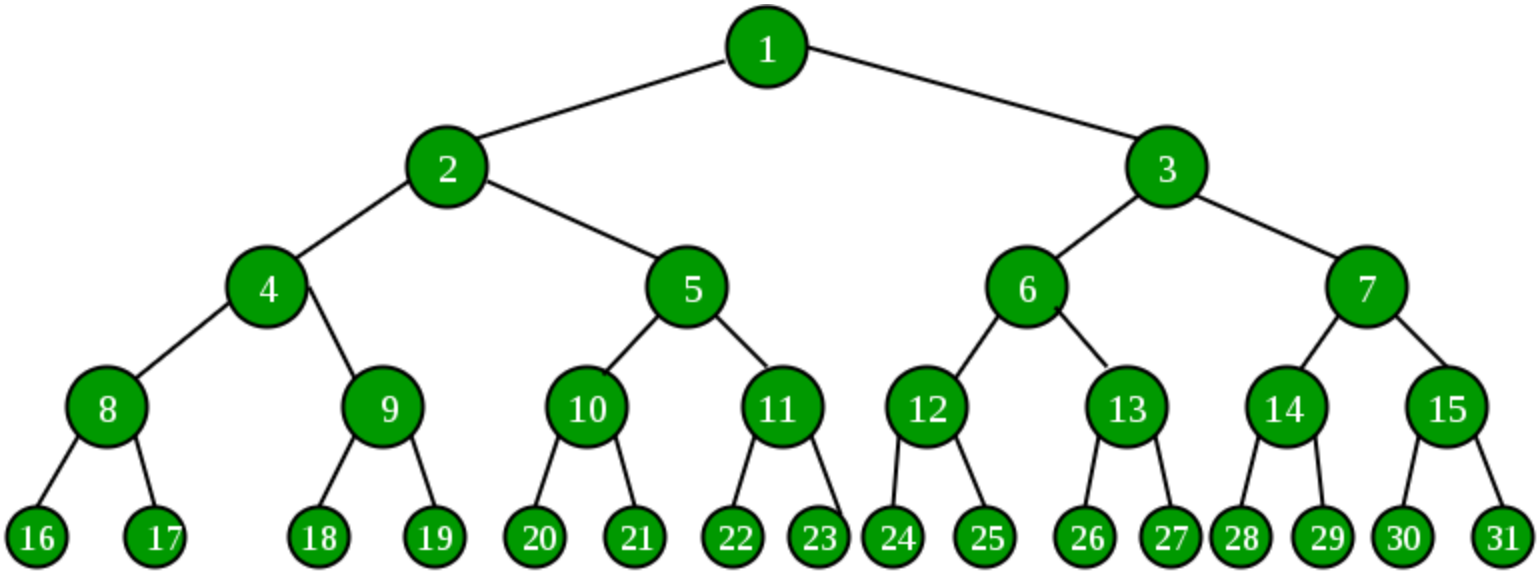
Binary Tree

- Definition: a binary tree is a tree + one property
  - *T* has a special node called the root node
  - *Every node except the root is connected by an edge from exactly one other node (its parent)*
  - *A unique path from root to each node*
  - *Each node has a max of 2 children*

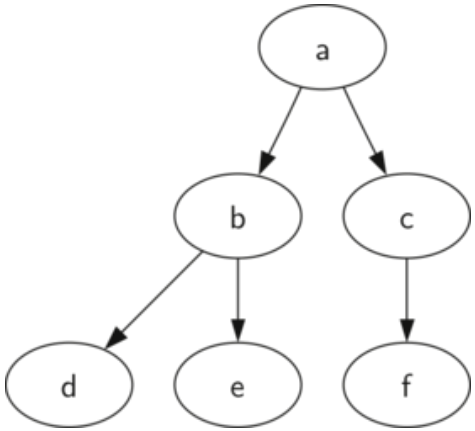
- Another way:
  - *T* has a special node called the root node
  - *T* has two sets of nodes, L_T and R_T, called the left subtree and right subtree of *T*, respectively
  - L_T and R_T are binary trees

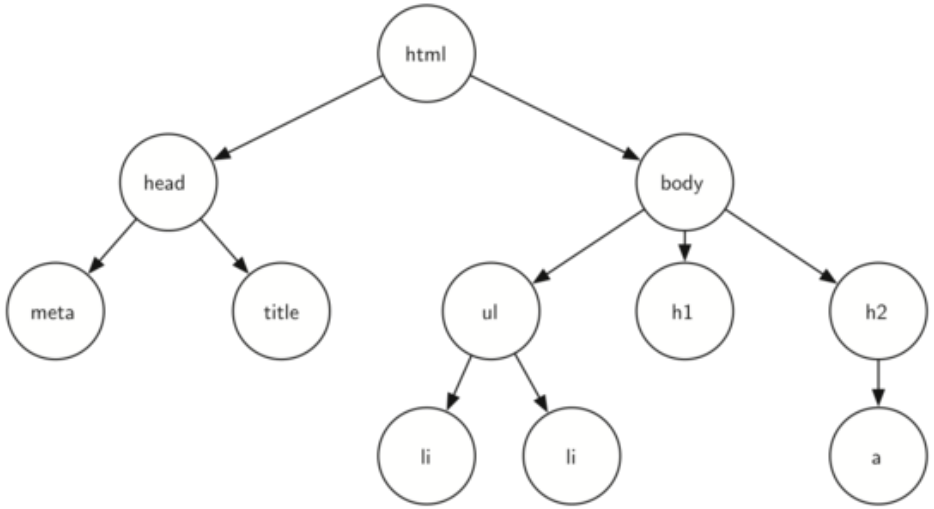
Perfect Binary Tree

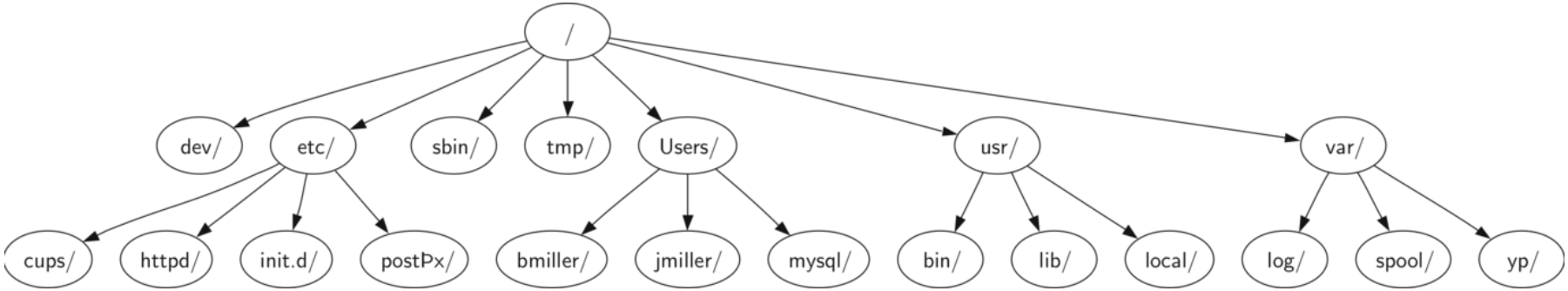
- Perfect Binary tree
  - All leaf nodes on the same depth
  - All non-leaf nodes have 2 children.



Binary Tree

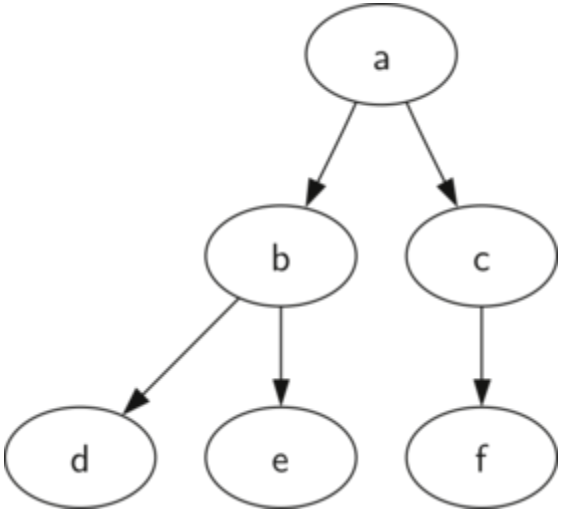

a
b
c
d
e
f


html
head
body
meta
title
ul
h1
h2
li
li
a


/
dev/
etc/
sbin/
tmp/
Users/
usr/
var/
cups/
httpd/
init.d/
postPx/
bmiller/
jmiller/
mysql/
bin/
lib/
local/
log/
spool/
yp/

List of lists representation

Root node: first element of the list
Left subtree: the second element of the list
Right subtree: the third element of the list



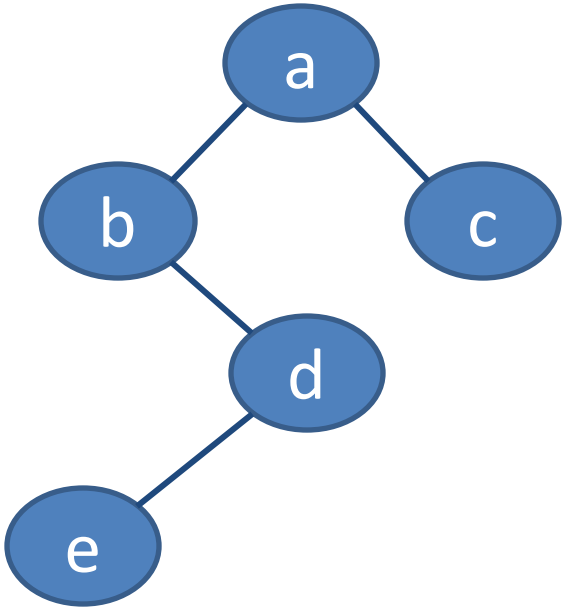

```
my_tree = ['a', #root
['b', #left subtree
['d' [], []],
['e' [], []] ],
['c', #right subtree
['f' [], []],
[] ]
]
```

Access root: my_tree[0]
Access left subtree of root: my_tree[1]
Access right subtree of root: my_tree[2]

Exercise

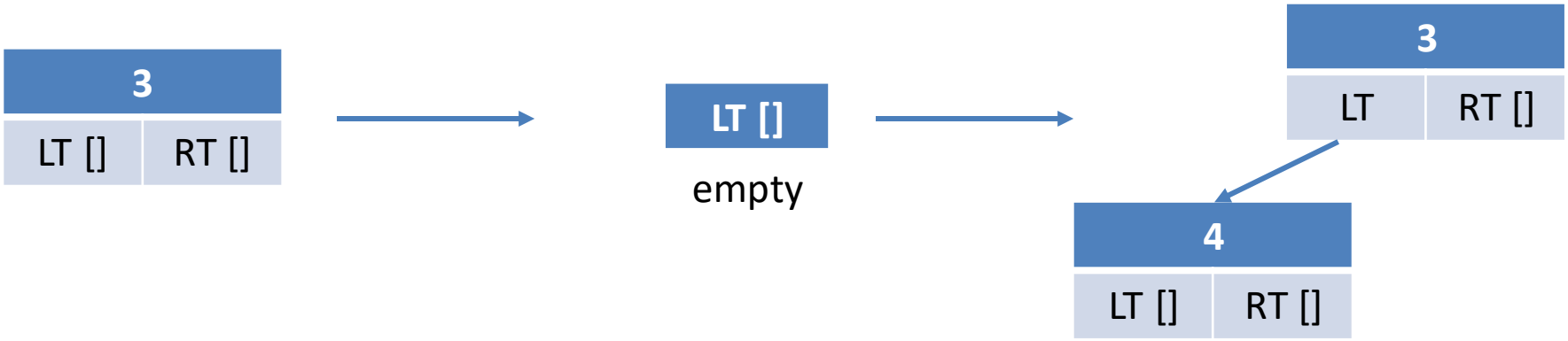
Draw the following tree:

['a', ['b', [], ['d', ['e', [], []], []]], ['c', [], []]]

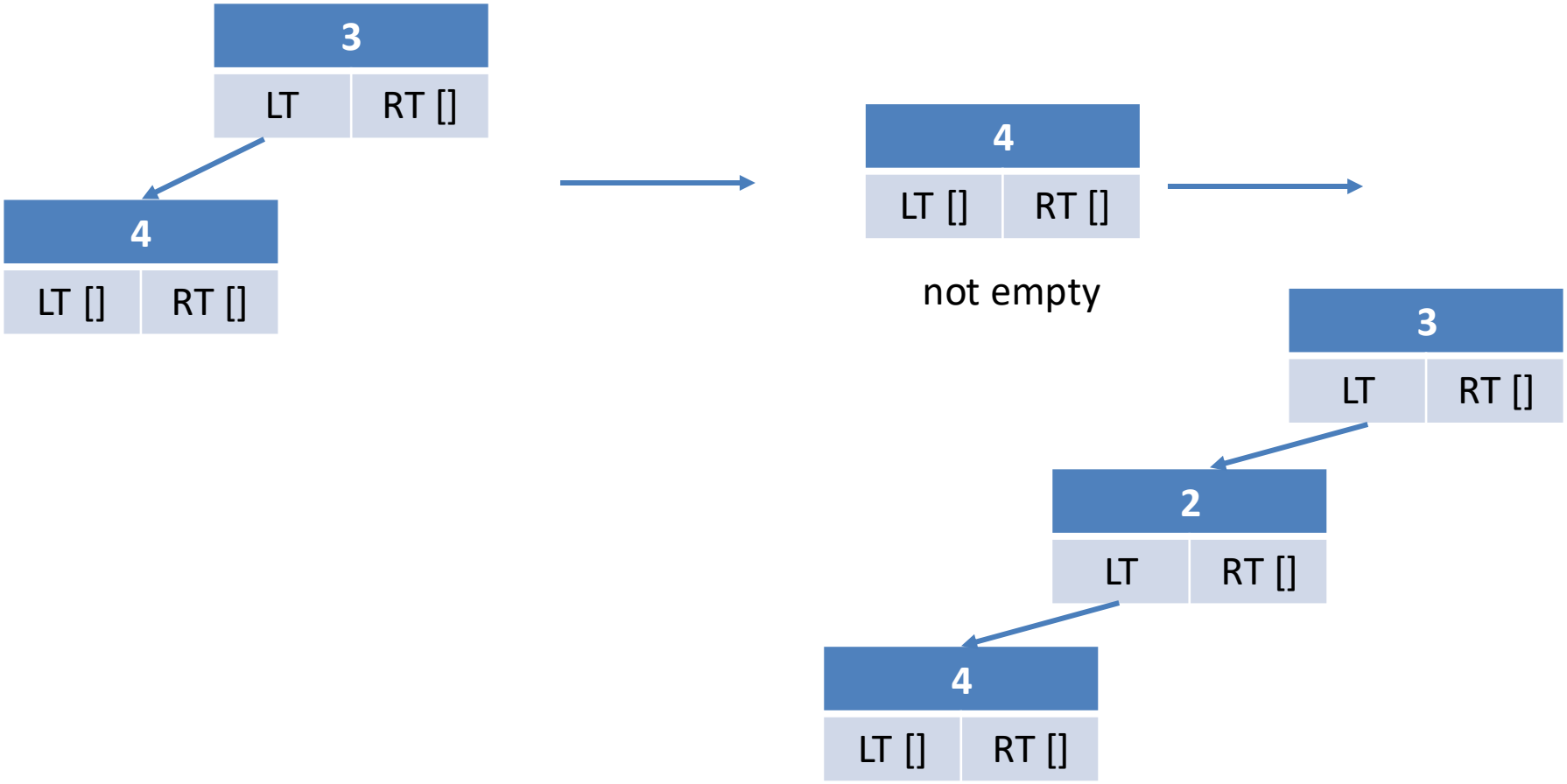


Insertion

```
r = BinaryTree(3)
insertLeft(r,4) #insert 4 as the
left child of r(3)
```

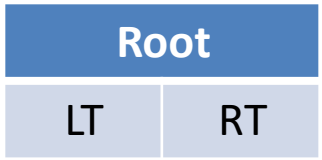


```
r = BinaryTree(3)
insertLeft(r,4)
insertLeft(r,2)
```



Insertion code

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



Write an insertion function to insert new branch as a node's left child

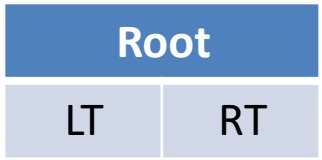
```
def insertLeft(root,newBranch)
```

Idea: check whether LT is empty:

- If not empty: push existing child down one level
- If empty: insert directly

List of lists representation code

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

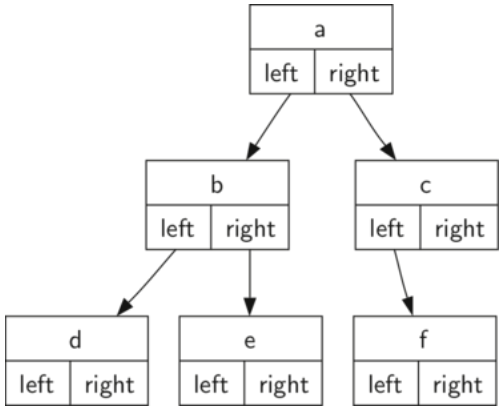


Write an insertion function to insert new branch as a node's left child

```
def insertLeft(root,newBranch):
    t = root.pop(1) #remove left child
    if len(t) > 1:  #LT is not empty: push existing child down one level
      root.insert(1,[newBranch,t,[]])
    else: #LT is empty: insert directly
      root.insert(1,[newBranch, [], []])
    return root
```

Binary Tree class

Object-oriented programing



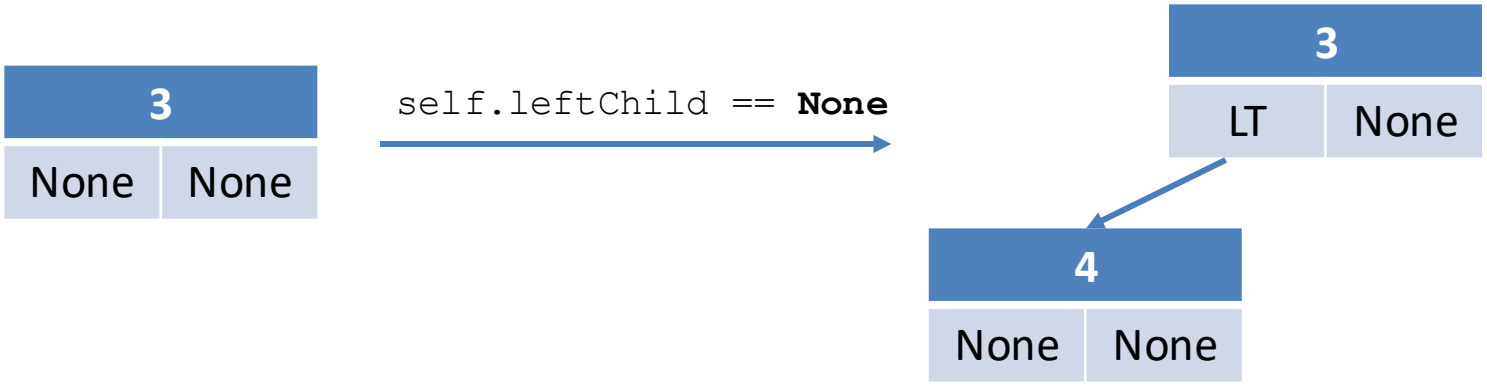

```
class BinaryTree:
    def __init__(self,rootObj):
        self.key = rootObj
        self.leftChild = None
        self.rightChild = None
```

Binary Tree class

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

        def getRootVal(self):
                return self.key

        def getLeftChild(self):
                return self.leftChild

        def getRightChild(self):
                return self.rightChild
```

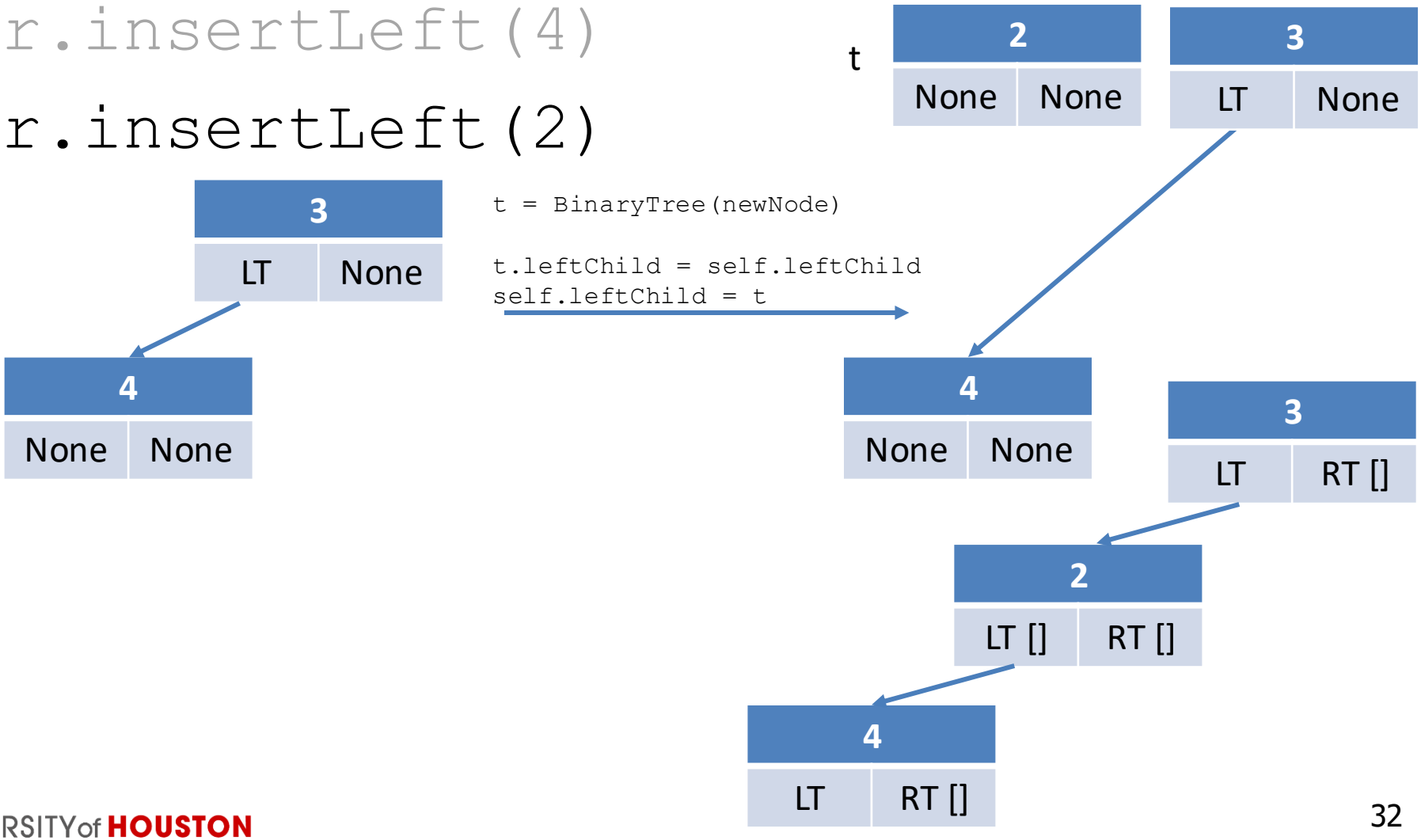
Insertion example

```
r = BinaryTree(3)
r.insertLeft(4)
r.insertLeft(2)
```



Insertion example

```
r = BinaryTree(3)
r.insertLeft(4)
r.insertLeft(2)
```



Insertion method code

Write an insertion method **`def`** `insertLeft(self,newNode)` to insert the new node as a node's left child

Case 1: no existing left child – add a node directly

Case 2: with an existing left child – push existing child down one level

Insertion method code

Case 1: no existing left child – add a node directly

Case 2: with an existing left child – push existing child down one level

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

Insertion method code

Case 1: no existing right child – add a node directly

Case 2: with an existing right child – push existing child down one level

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

Binary trees and recursion

- Binary trees are big fans of recursion!
- That's because all nodes look the same
- Access is typically done through reference (first reference being the root)

Binary Tree Traversal

- Inorder traversal
  - Traverse the left subtree
  - Visit the node
  - Traverse the right subtree

Gives non-decreasing order in binary search trees (left to right)

- Preorder traversal
  - Visit the node
  - Traverse the left subtree
  - Traverse the right subtree

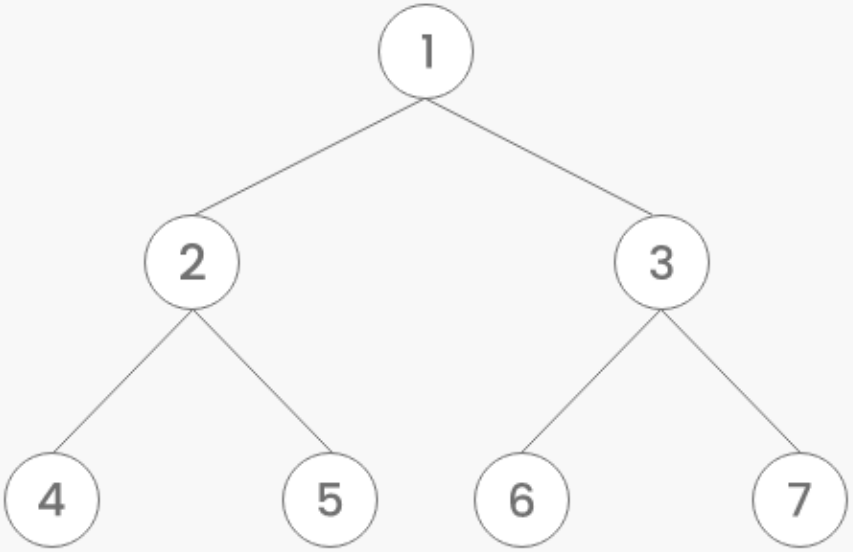
Most natural (e.g., reading a book), create a tree (kind of top down)

- Postorder traversal
  - Traverse the left subtree
  - Traverse the right subtree
  - Visit the node

Check leaves first, delete a tree (kind of bottom up)

Binary Tree Traversal

Tree Traversal Techniques



Inorder Traversal

4	2	5	1	6	3	7

Preorder Traversal

1	2	4	5	3	6	7

Postorder Traversal

4	5	2	6	7	3	1

https://www.geeksforgeeks.org/tree-traversals-inorder-preorder-and-postorder/

Binary Tree Traversal

- Each traversal algorithm: recursive
- Listing of nodes
  - **Inorder sequence**
  - Preorder sequence
  - Postorder sequence

Write inorder sequence function def `inorder(tree)`

Traverse the left subtree
Visit the node
Traverse the right subtree

Binary Tree Traversal

- Each traversal algorithm: recursive
- Listing of nodes
  - **Inorder sequence**
  - Preorder sequence
  - Postorder sequence

Traverse the left subtree
Visit the node
Traverse the right subtree

```
def inorder(tree):
    if tree:
        inorder(tree.getLeftChild())
        print(tree.getRootVal())
        inorder(tree.getRightChild())
```

Binary Tree Traversal

- Each traversal algorithm: recursive
- Listing of nodes
  - Inorder sequence
  - **Preorder sequence**
  - Postorder sequence

Write preorder sequence function def `preorder(tree)`

Visit the node
Traverse the left subtree
Traverse the right subtree

Binary Tree Traversal

- Each traversal algorithm: recursive
- Listing of nodes
  - Inorder sequence
  - **Preorder sequence**
  - Postorder sequence

Visit the node
Traverse the left subtree
Traverse the right subtree

```
def preorder(tree):
    if tree:
      print(tree.getRootVal())
      preorder(tree.getLeftChild())
      preorder(tree.getRightChild())
```

Binary Tree Traversal

- Each traversal algorithm: recursive
- Listing of nodes
  - Inorder sequence
  - Preorder sequence
  - **Postorder sequence**

Write postorder sequence function def `postorder(tree)`

Traverse the left subtree
Traverse the right subtree
Visit the node

Binary Tree Traversal

- Each traversal algorithm: recursive
- Listing of nodes
  - Inorder sequence
  - Preorder sequence
  - **Postorder sequence**

Traverse the left subtree
Traverse the right subtree
Visit the node

```
def postorder(tree):
    if tree:
      postorder(tree.getLeftChild())
      postorder(tree.getRightChild())
      print(tree.getRootVal())
```