COSC 2306
Data Programming

Trees

A tree-like organization

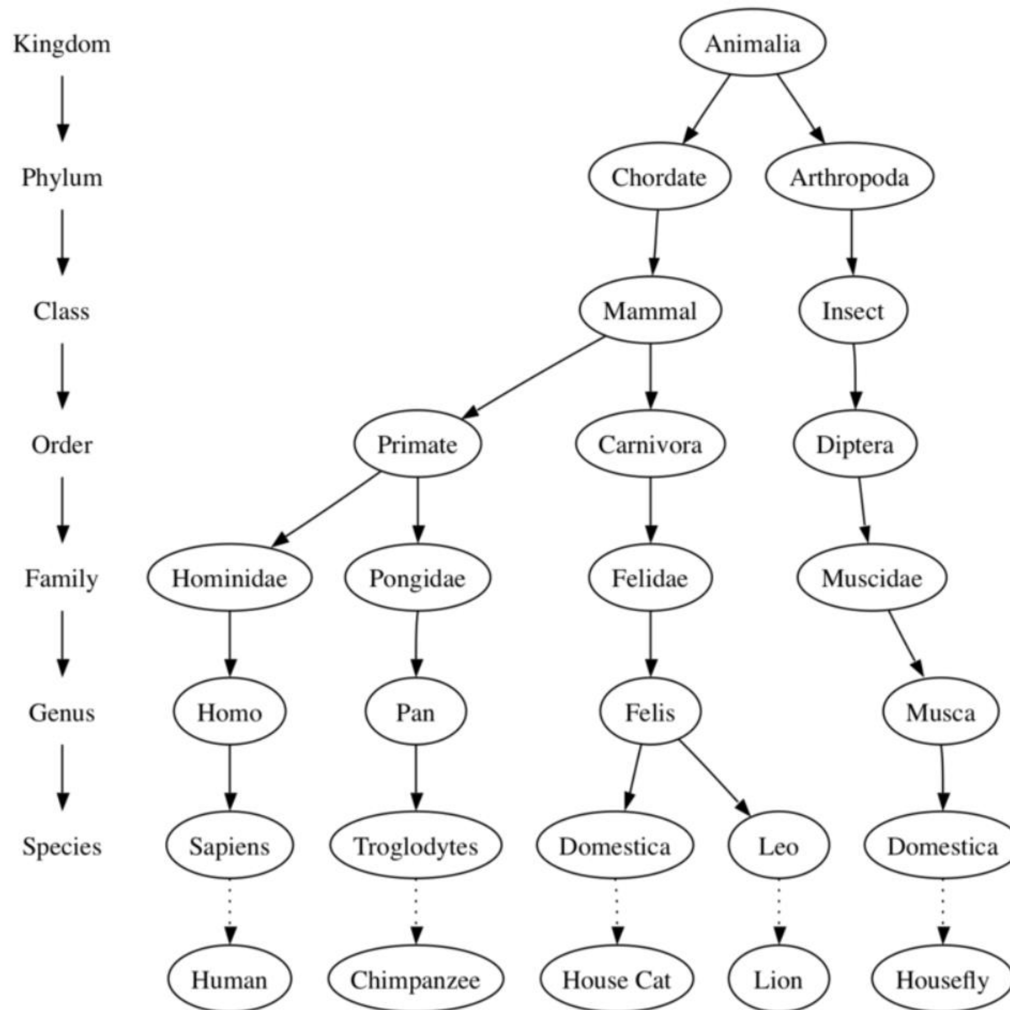


Figure 1: Taxonomy of Some Common Animals Shown as a Tree

A tree-like organization

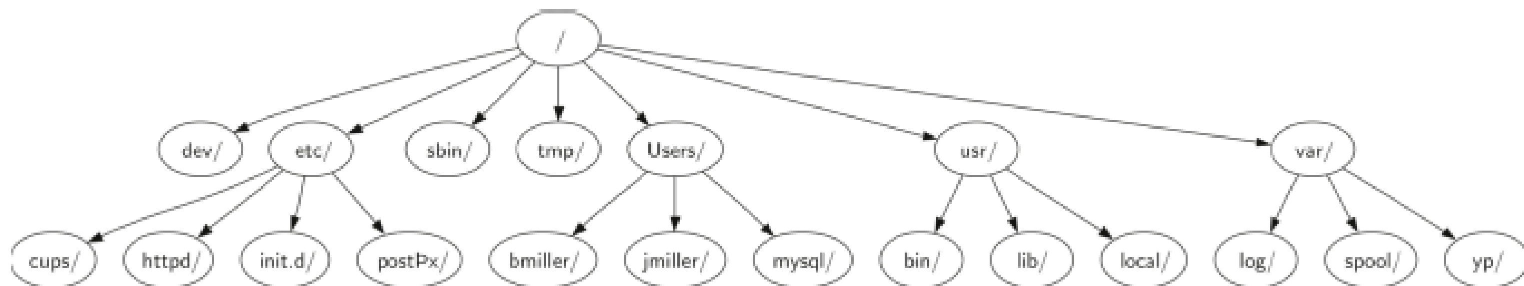
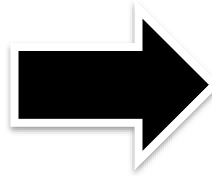
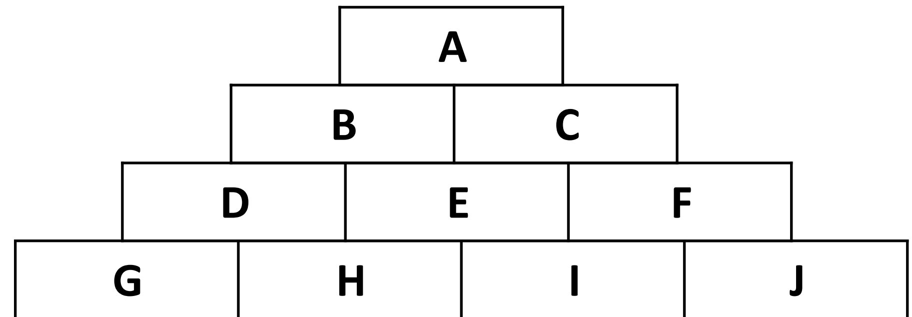


Figure 6.2: A Small Part of the Unix File System Hierarchy

Organizing data

A
B
C
D
E
F
G
H
I
J



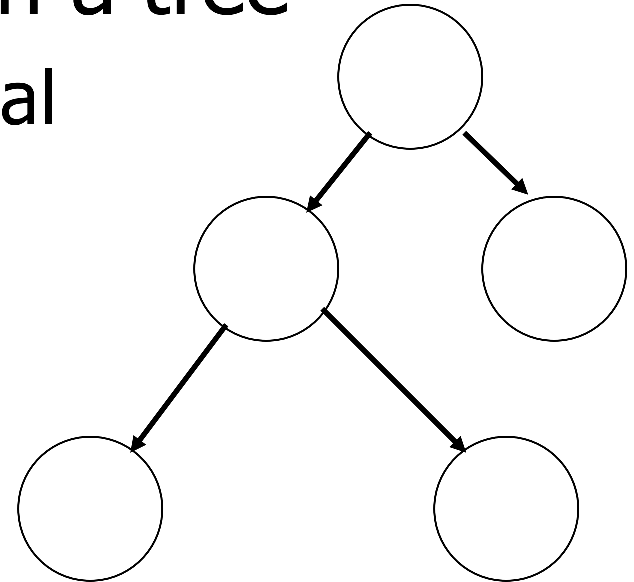


Application of trees

- 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

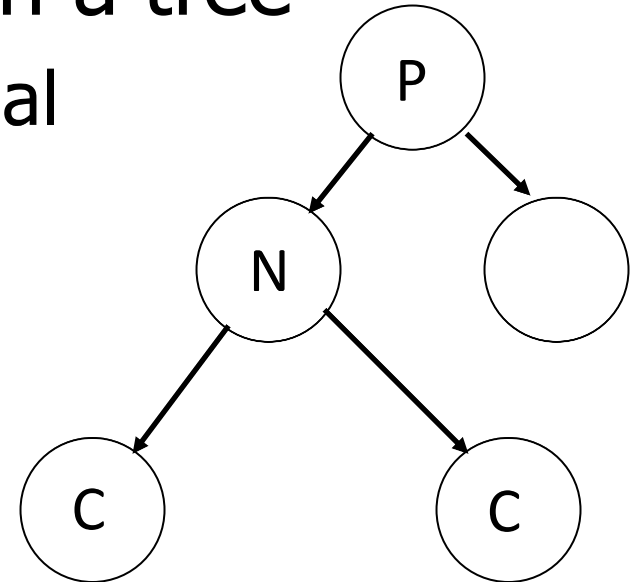
Tree node

- 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



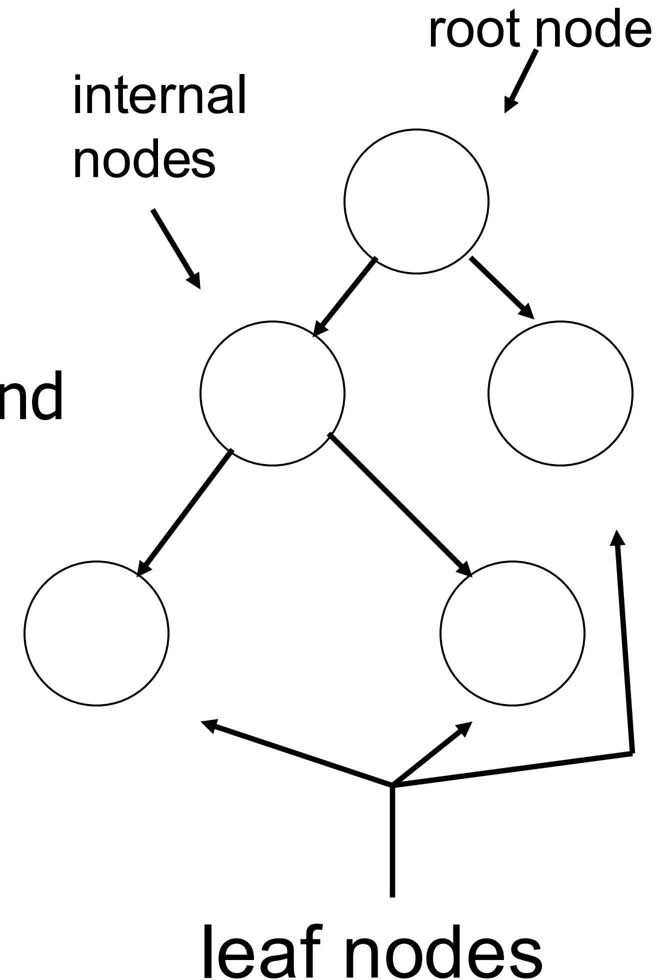
Tree node

- 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



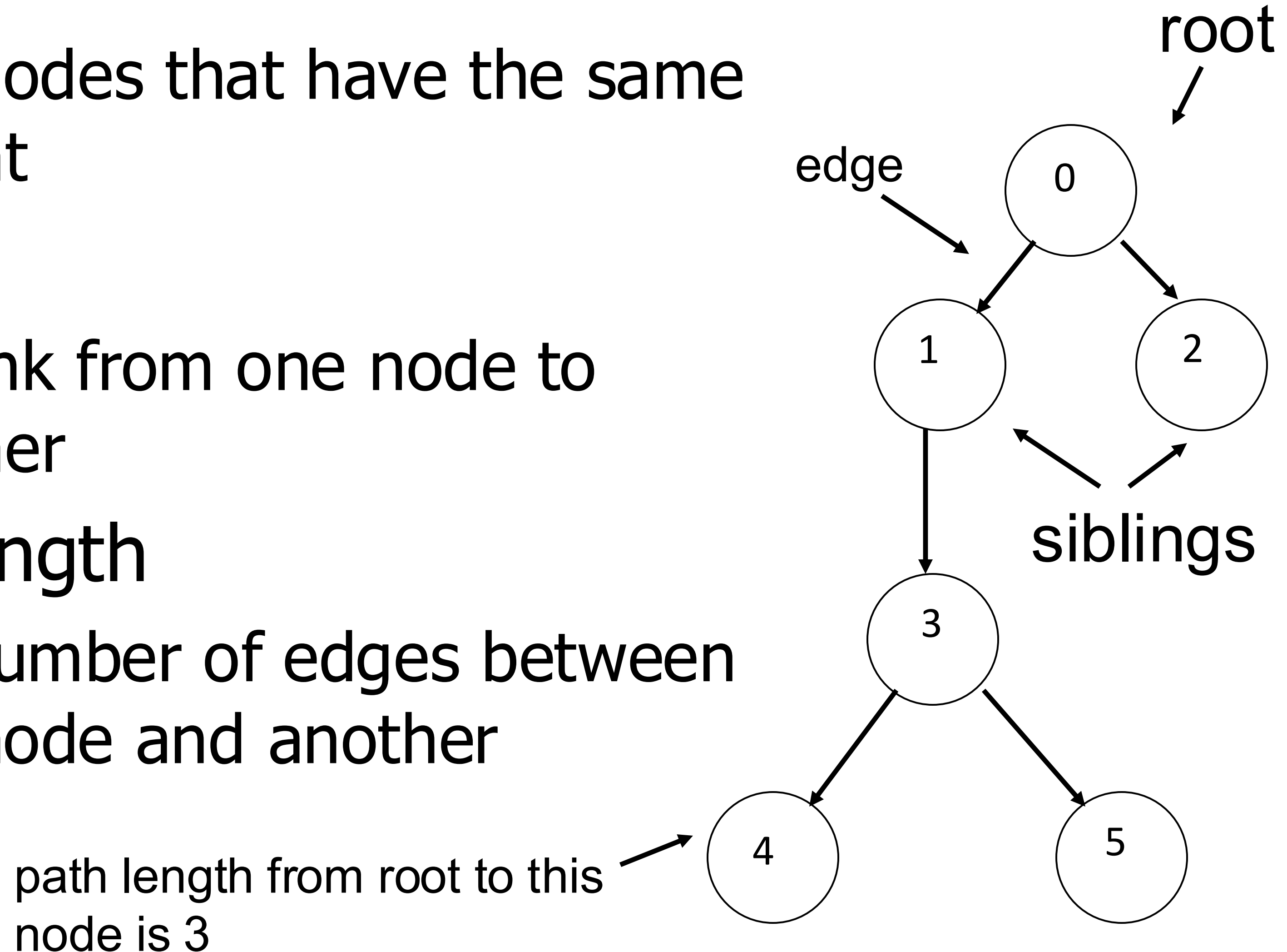
Tree node

- 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



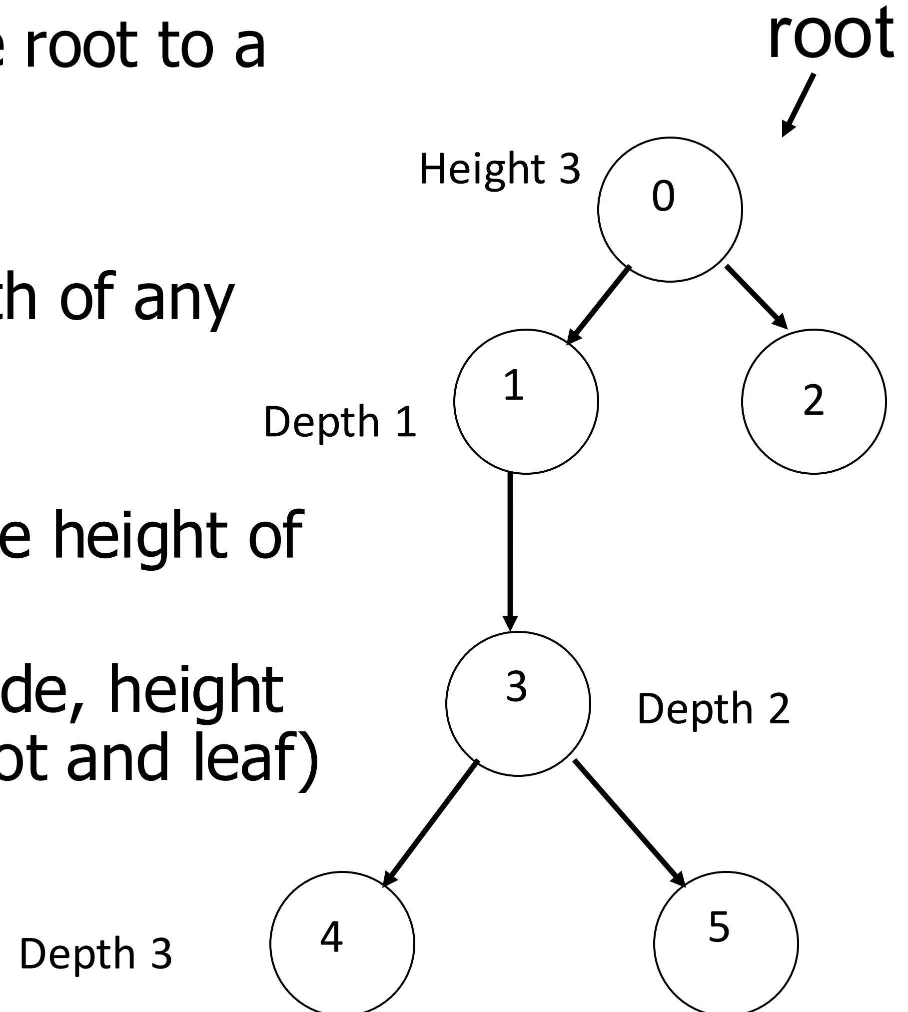
Tree vocabularies

- 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



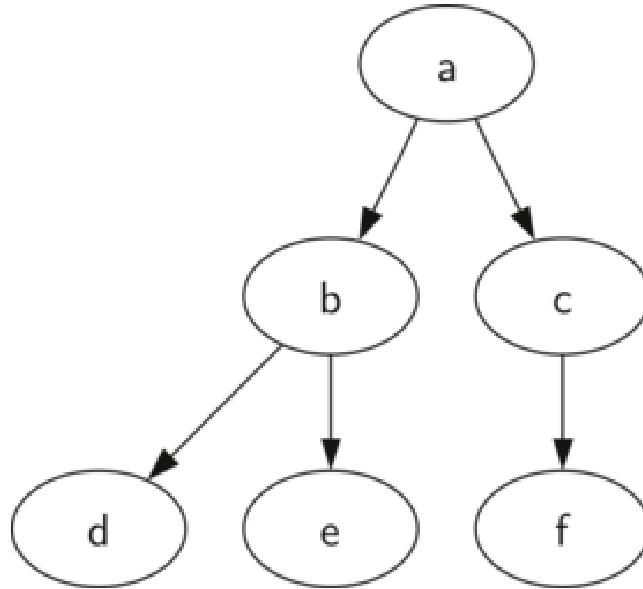
Tree vocabularies

- 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

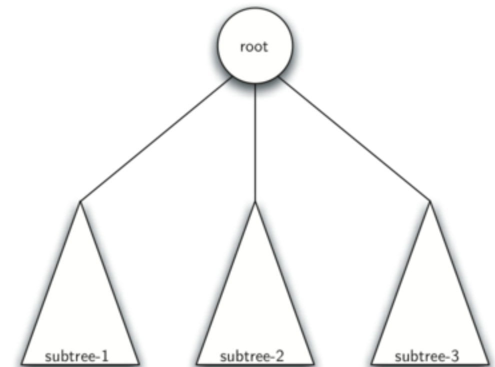
Subtree: a set of nodes and edges of a parent and all descendants of that parent (e.g., b, d, e)



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 either empty (no node) or consist of a root and 0 or more subtrees
  - *Recursive definition*
  - *Subtree is a tree*
  - *Example shows at least 4 nodes*



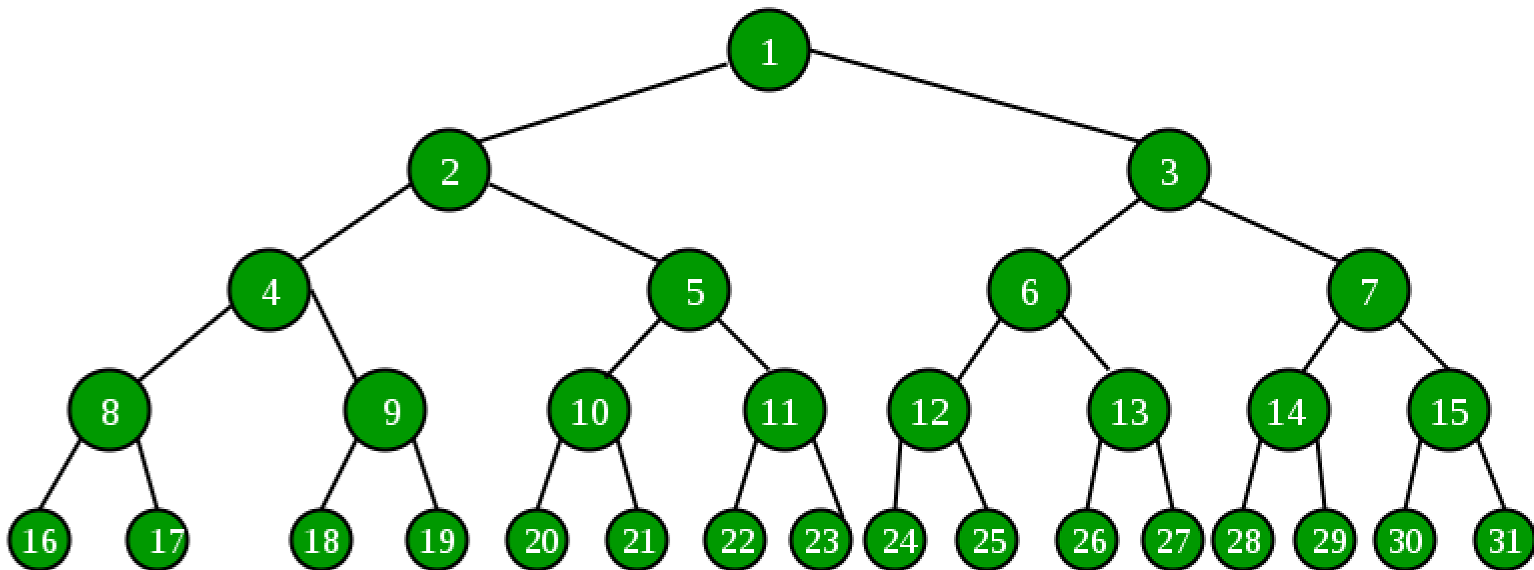
Binary Tree

- 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*

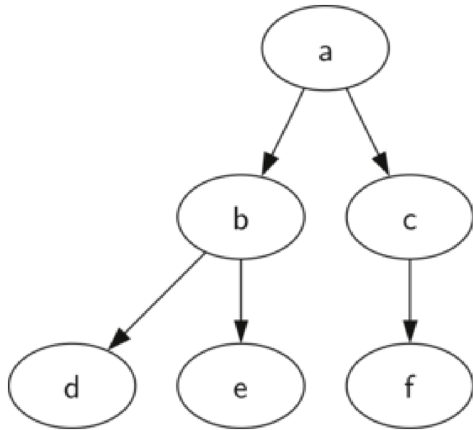
- Another way:
  - *T* has a special node called the root node
  - *T* has two sets of nodes, L_T and R_T, called the left subtree and right subtree of *T*, respectively
  - L_T and R_T are binary trees

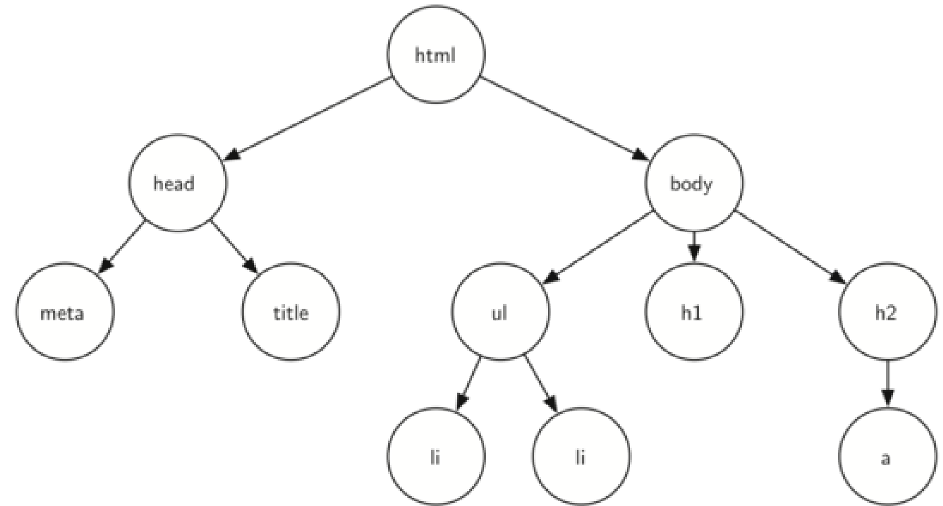
Perfect Binary Tree

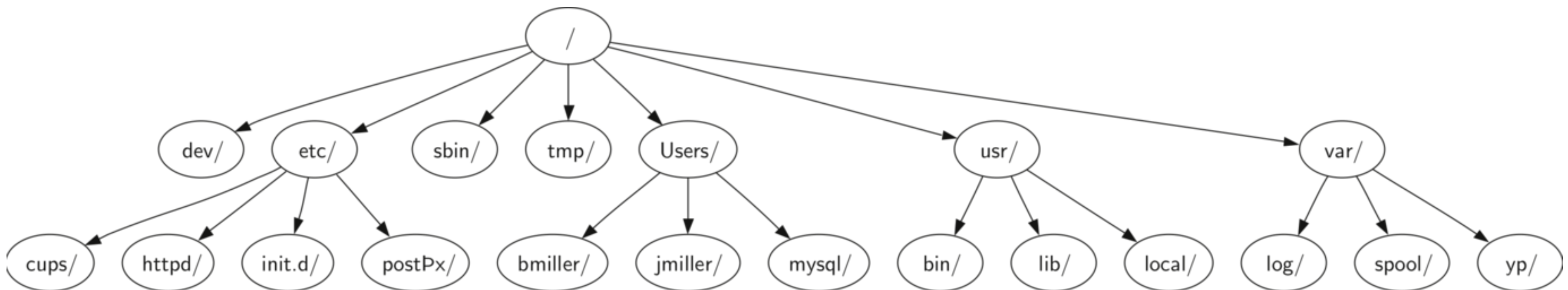
- Perfect Binary tree
  - All leaf nodes on the same depth
  - All non-leaf nodes have 2 children.



Binary Tree

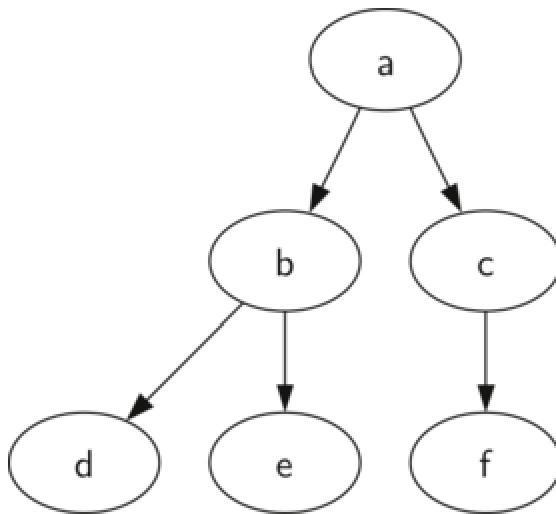






List of lists representation

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



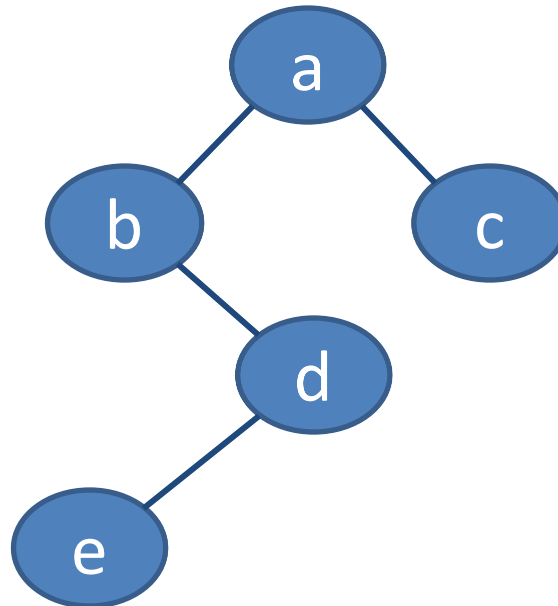

```
my_tree = ['a', #root
['b', #left subtree
['d' [], []],
['e' [], []] ],
['c', #right subtree
['f' [], []],
[] ]
]
```

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

Exercise

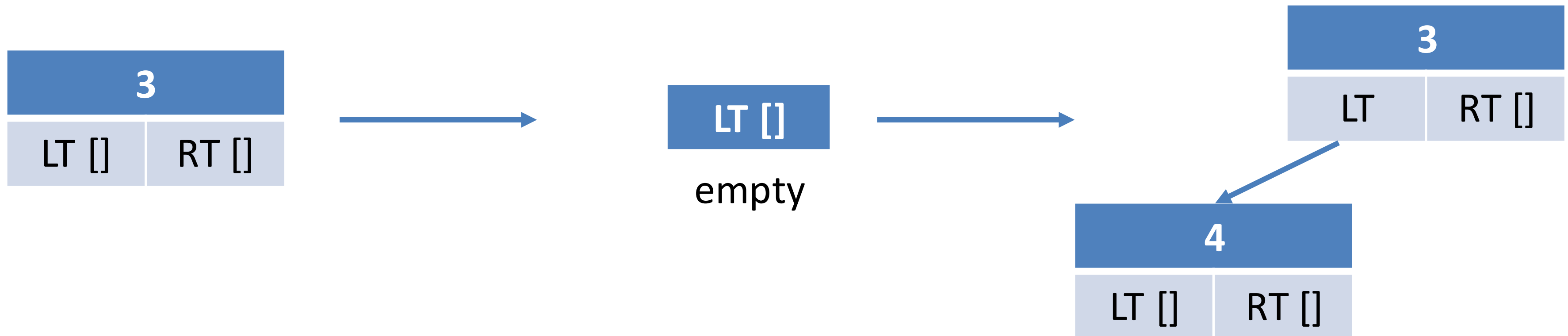
Draw the following tree:

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

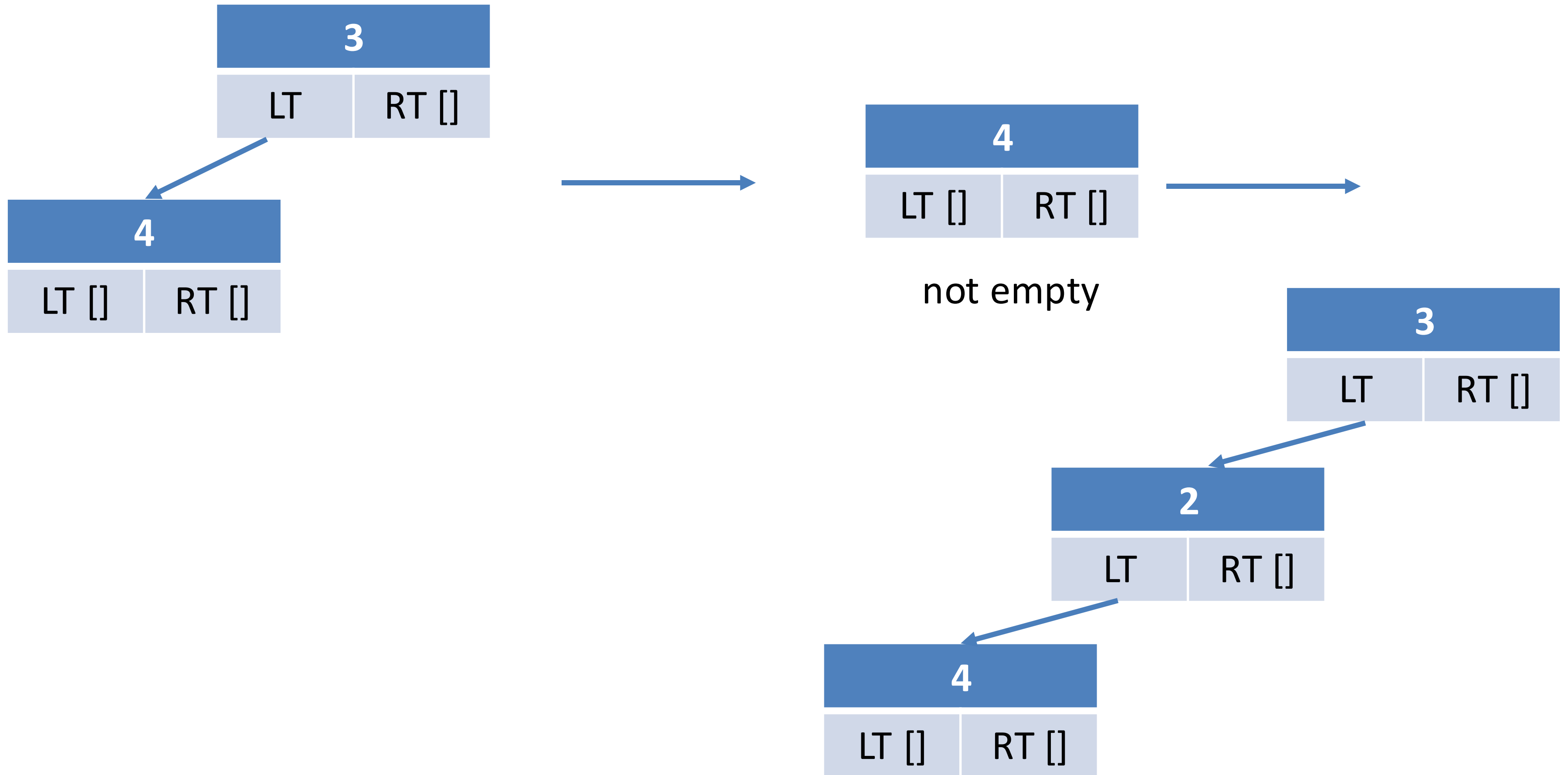


Insertion

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

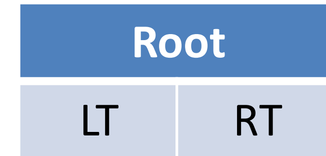


```
r = BinaryTree(3)
insertLeft(r,4)
insertLeft(r,2)
```



Insertion code

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



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

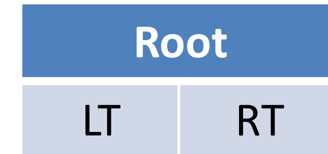
```
def insertLeft(root,newBranch)
```

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, [], []]
```

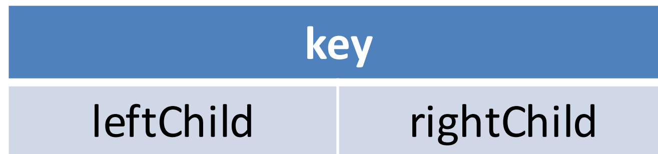


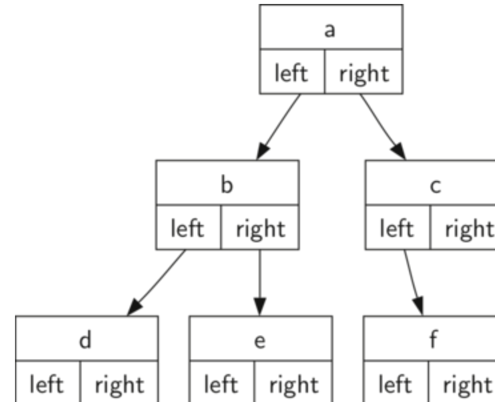
Write an insertion function to insert new branch as a node's left child

```
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
```

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

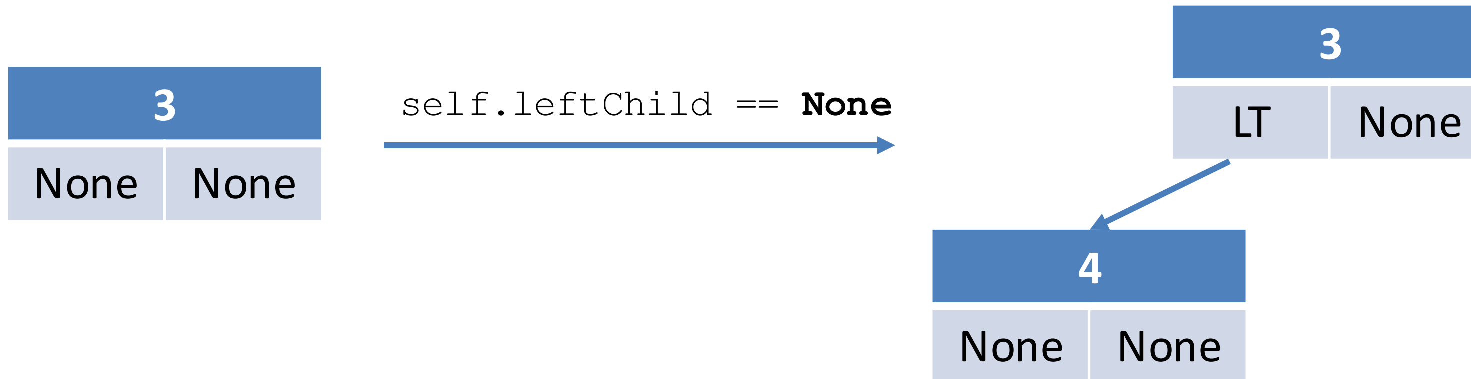
Case 1: no existing right child – add a node directly

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
```

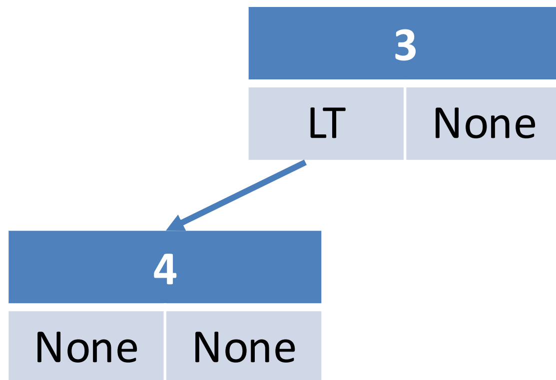
Insertion example

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



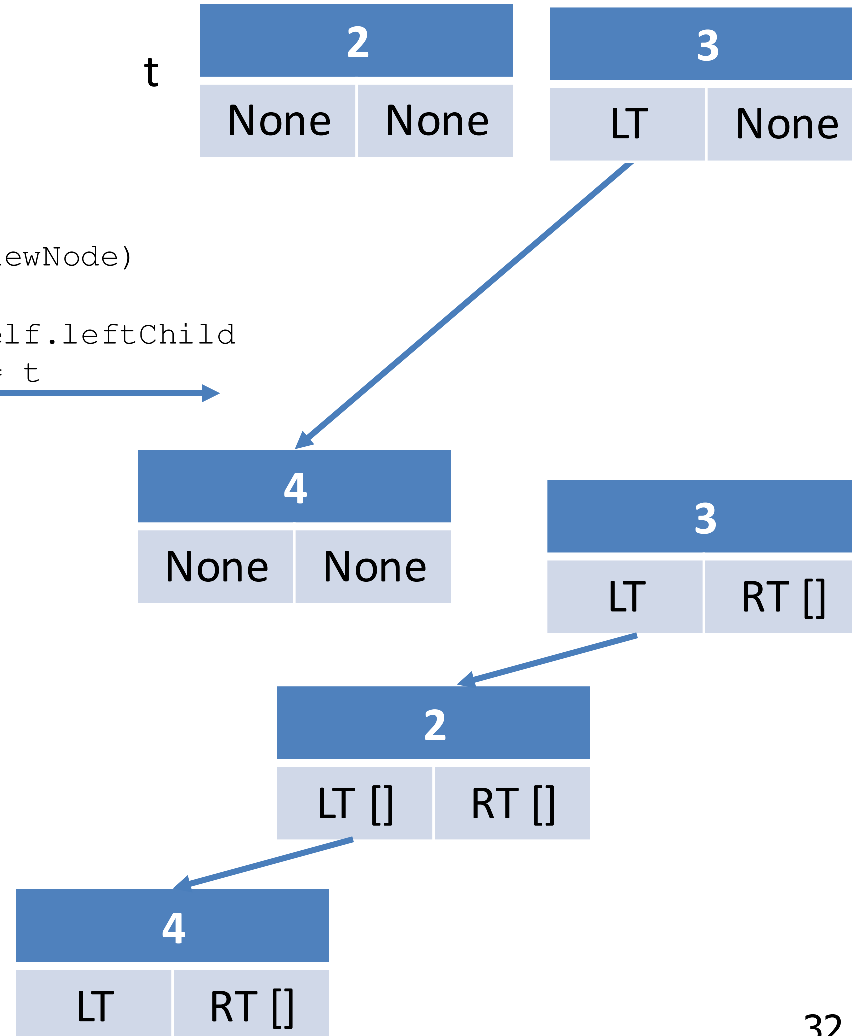
Insertion example

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




```
t = BinaryTree(newNode)

t.leftChild = self.leftChild
self.leftChild = 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 (introduce next)

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

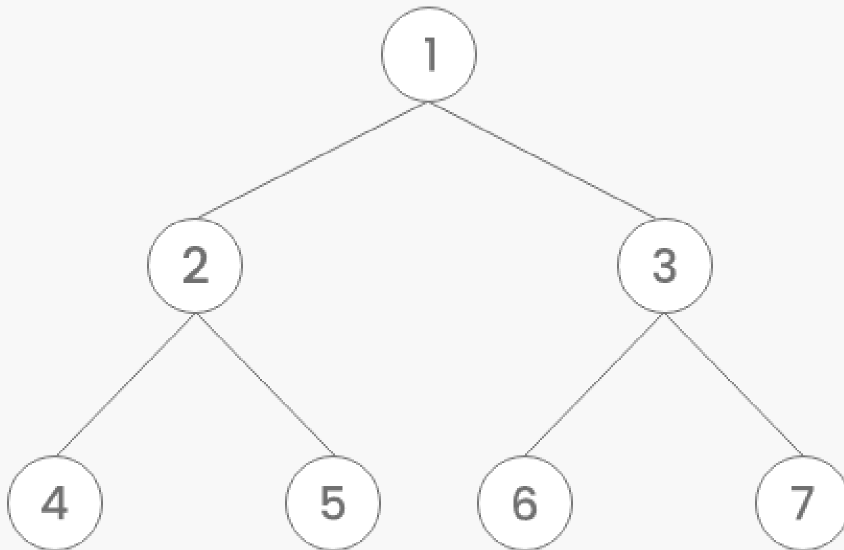
Most natural (e.g., reading a book), create a tree

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

Check leaves first, delete a tree

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())
```