CSC148H Week 6

February 9, 2015

Announcements:

- Term test 1 this Friday at 17:00 in IB110

Motivating Trees

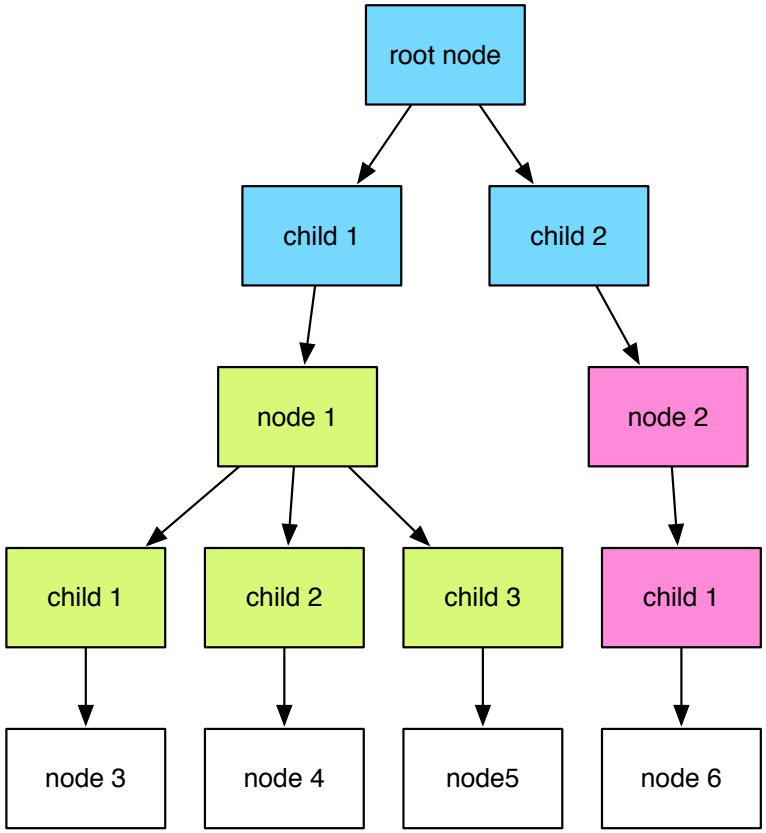
- A data structure is a way of organizing data
- Stacks, queues, and lists are all linear structures
- They are linear in the sense that data is ordered (i.e. first piece of data is followed by second is followed by third ...)
- This makes sense for many applications:
  - A lineup in a bank (queues)
  - Function calls in programs (stacks)

Motivating Trees...

- It doesn't make sense to organize certain types of data into a linear structure
- Consider directories in a file system. They have a natural hierarchical structure that is difficult to represent linearly
- If we want to use a list, we might imagine storing the root directory at the first position and its subdirectories and files to its right
- But how would we know when the files of a subdirectory ends and we are back up one level?
- Other examples:
  - Structure of an HTML document
  - Structure of a Python program

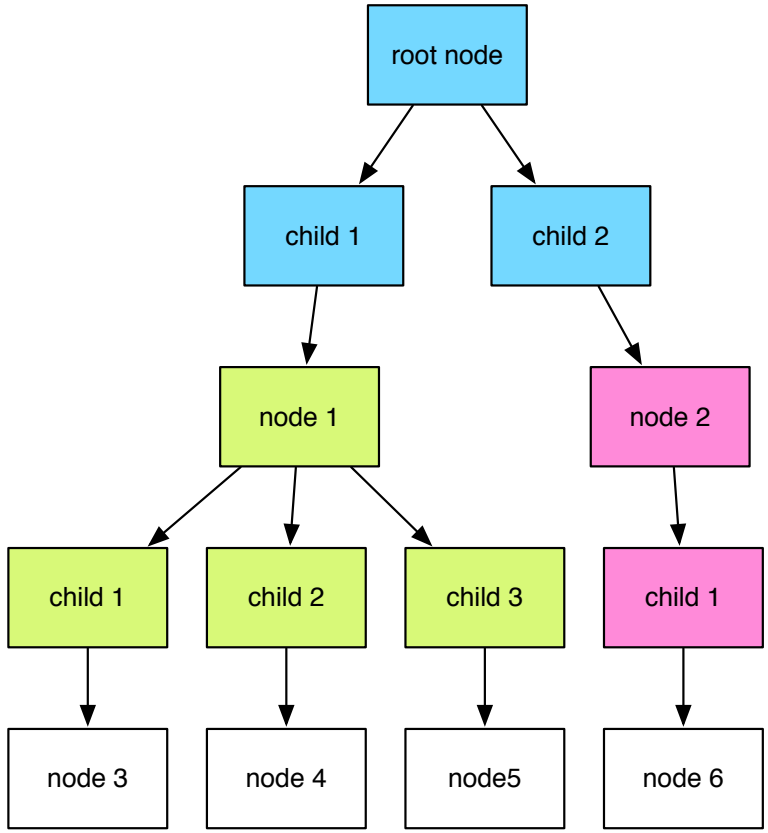
Tree Definition

- Every tree T, is either **empty** or **non-empty**
- A tree stores elements with nodes that have *parent-child* relationship where:
  - If T is non-empty, it has a special node, called the **root** of T, that has no parent.
  - Each node v of T different from the root has a unique parent node w; every node with parent w is a child of w.



Tree Definition

- A tree has a set of nodes, and directed **edges** that connect nodes.
- An **edge** connects two nodes to show that there is a relationship between them.
- Every node besides the root has exactly one **parent**.

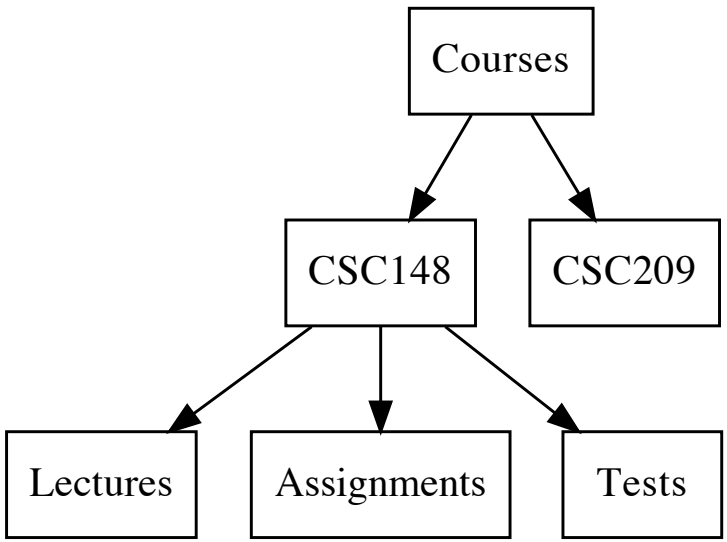


Tree Definition

According to our definition, a tree can be empty, meaning that it does not have any nodes. This allows us to define a tree recursively such that a tree T is either empty or consists of a node r, called the root of T, and a (possibly empty) set of subtrees whose roots are the children of r.

```
class EmptyValue:
    pass

class Tree:

    def __init__(self, root=EmptyValue):
        '''(Tree, object) -> NoneType'''
        self.root = root
        self.subtrees = []

    def is_empty(self):
        '''(Tree) -> bool'''
        return self.root is EmptyValue
```

Sample Tree

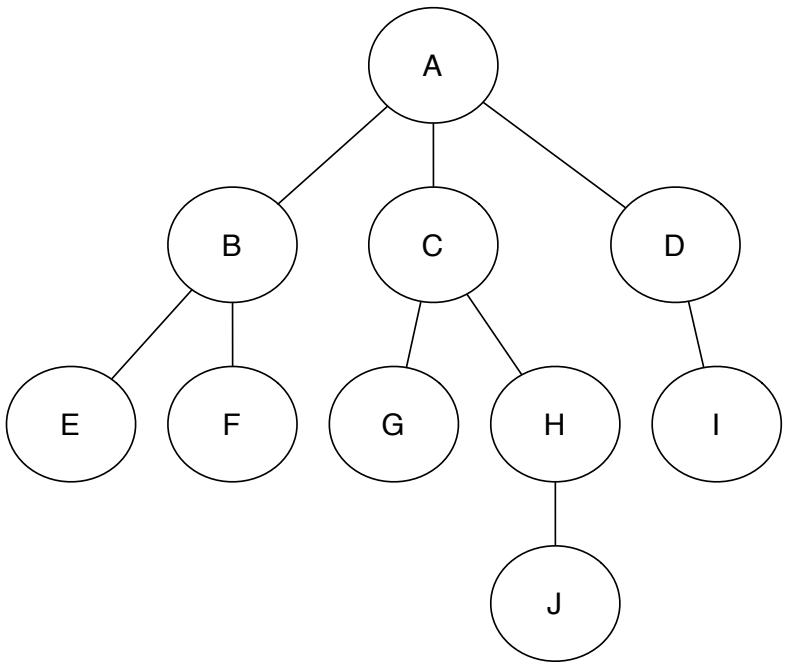


- Which is the root node?
- How many nodes are there?
- How many edges are there?

Tree Terminology

- Parent: A node is the parent of all nodes to which it has outgoing edges
- Siblings: Set of nodes that share a common parent
- Leaf: A node that has no children (i.e. no outgoing edges)
- Internal node: A nonleaf node
- Path: sequence of nodes $n_1, n_2, \ldots, n_k$, where there is an edge from n_i to n_{i+1}
- Subtree: A subtree of tree T is a tree whose root node r is a node in T, and which consists of all the descendants of r in T and the edges among them

Sample Tree

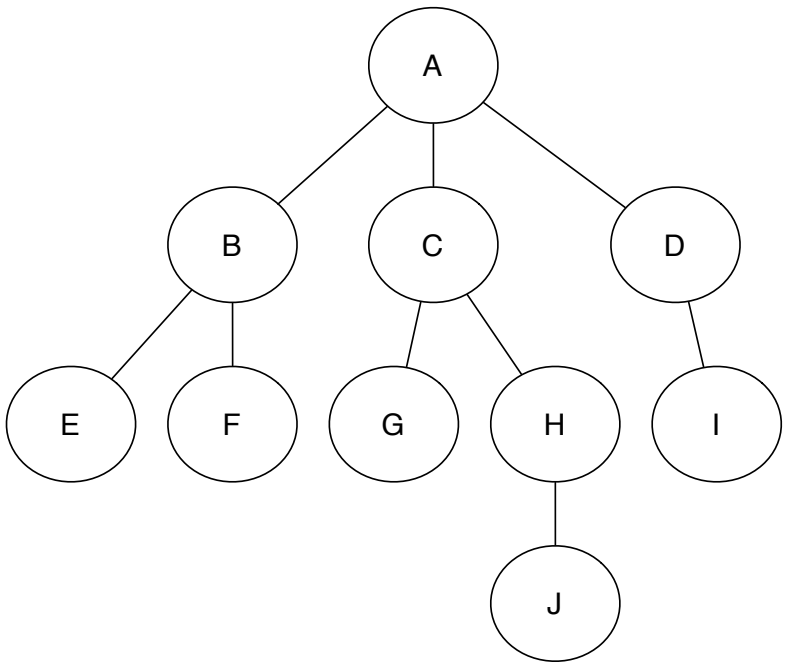


- What are the children of node A?
- How many leaves are there? What are they?
- How many internal nodes are there? What are they?

More Tree Terminology

- Height: The length of the longest path from its root node r to one of its leaves
- Size: Number of nodes in a tree T
- Ancestor: Parents of a node p
- Descendant: A node n is a descendant of some other node p if there is a path from p to n

Sample Tree



- What is the height of the tree?
- What is the size of the tree?
- What is the size of an empty tree?

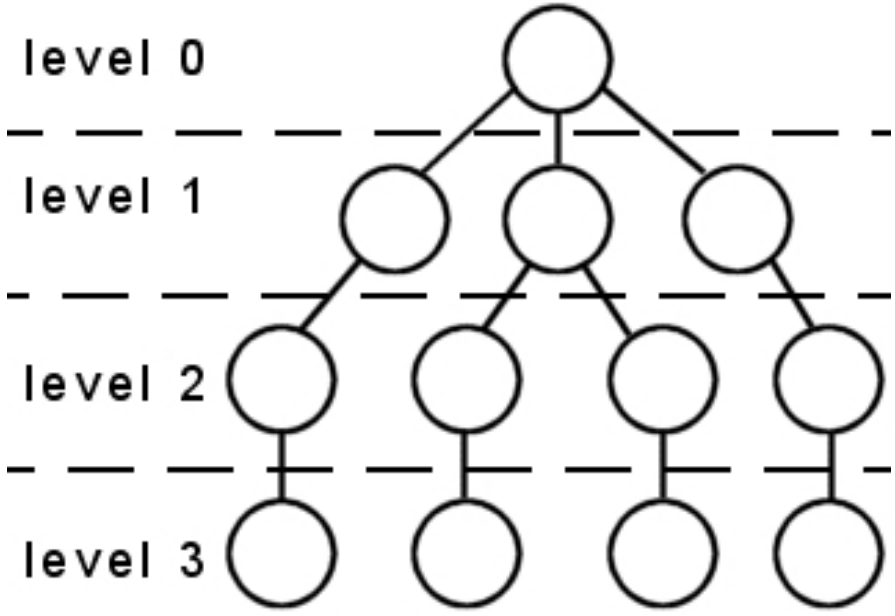
Trees and Recursion

- Size of a non-empty tree is the sum of the sizes of its subtrees, plus 1 for the root;
- Size of an empty tree is 0
- How can we compute the size of a tree with this information?

```
def size(self):
    if self.is_empty():
        return 0
    else:
        size = 1

        for subtree in subtrees:
            size += subtree.size()
        return size
```

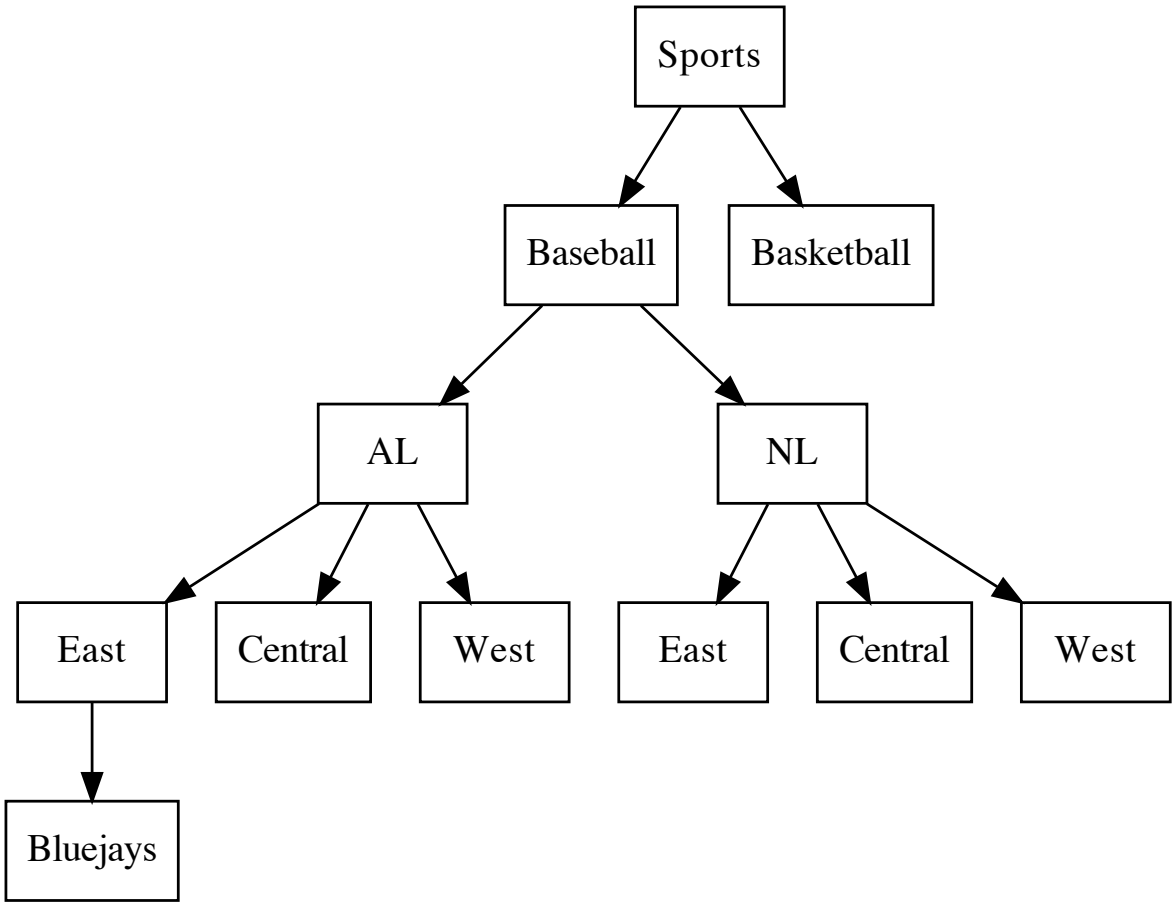
Tree Terminology...

- Degree: Number of direct children
- Branching factor: The maximum number of children of any node (maximum degree of all nodes) in the tree
- Level (Depth): The level (or depth) of node n is the number of edges on the path from the root node to n. The level of the root is 0.



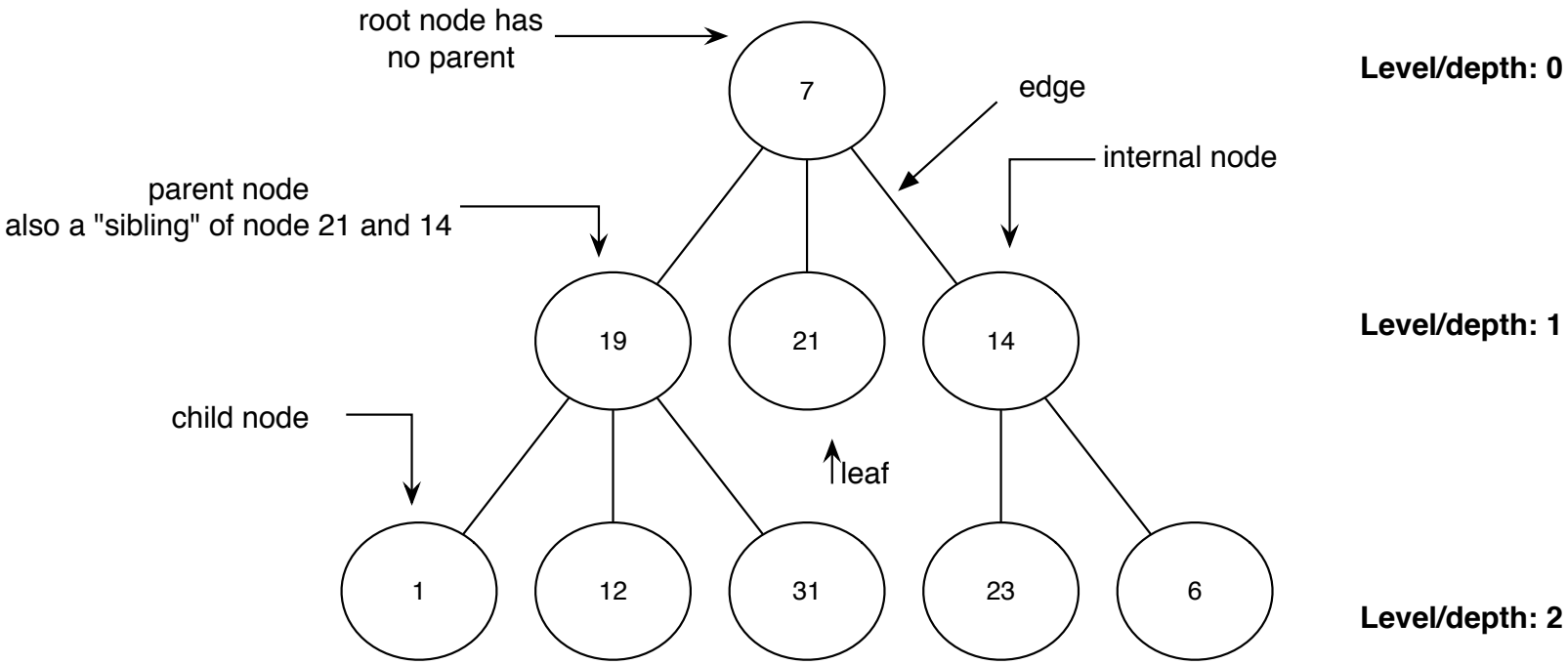
- Path: Sequence of nodes connected by edges, where no node is repeated
- Path length: The number of of edges connecting a sequence nodes in a path

Another Sample Tree



What is the height of the tree? Levels? Branching factor? Depth of baseball? Length of path from sport to AL?

Piecing The Tree Together



Degree
Degree of "19" and 7" is 3
Degree of "14" is 2
Degree of "1" is 0
(leaf nodes have a degree of 0)

Path Sequence and Length
Path sequence: "1", "19", "7", "21"
Path length: 3 (number of nodes minus 1)

Sequence "19", "32" is not a path since they are not connected

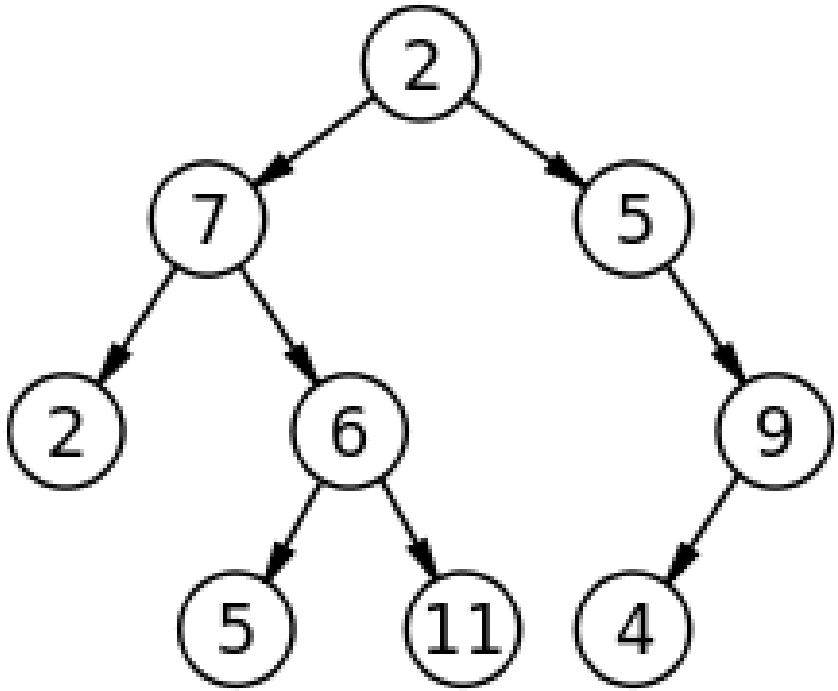
Other
Tree height: 2

Tree branching factor: 3

Common Operations on Trees

- Traverse a tree: visit the nodes in some order and apply some operation to each node
- Insert a new node
- Remove a node
- Attach a subtree at a node
- Remove a subtree

We will focus on binary trees; trees where each node has at most two children.



Tree Traversals

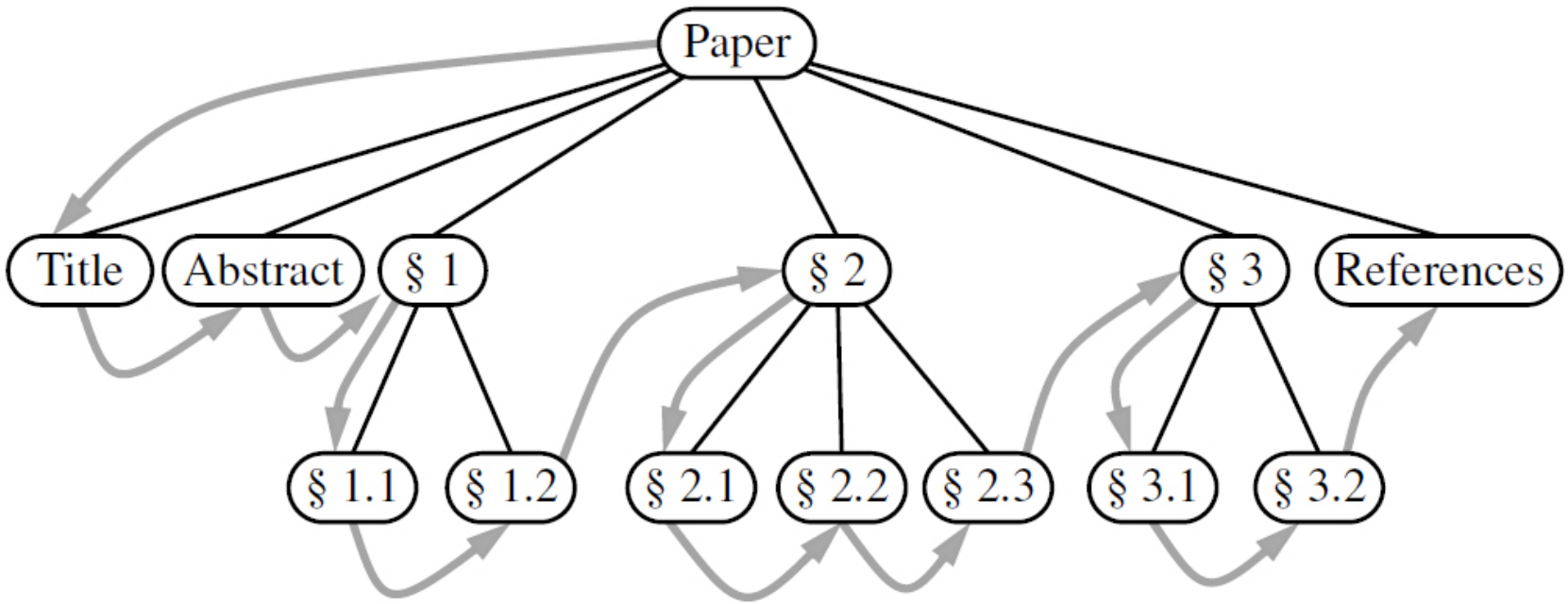
- **Traversal**: accessing each element of a structure
- We say we have visited an element when we have done something with it (e.g. print, compare, etc.)
- Lists have two obvious traversals: left to right, right to left
- Binary trees give us more ways to systematically visit each node
  - Preorder
  - Inorder
  - Postorder

Tree Traversals...

- Preorder: Visit the root node, do a preorder traversal of the left subtree, and do a preorder traversal of the right subtree
- Inorder: Do an inorder traversal of the left subtree, visit the root node, and then do an inorder traversal of the right subtree
- Postorder: do a postorder traversal of the left subtree, do a postorder traversal of the right subtree, and visit the root node

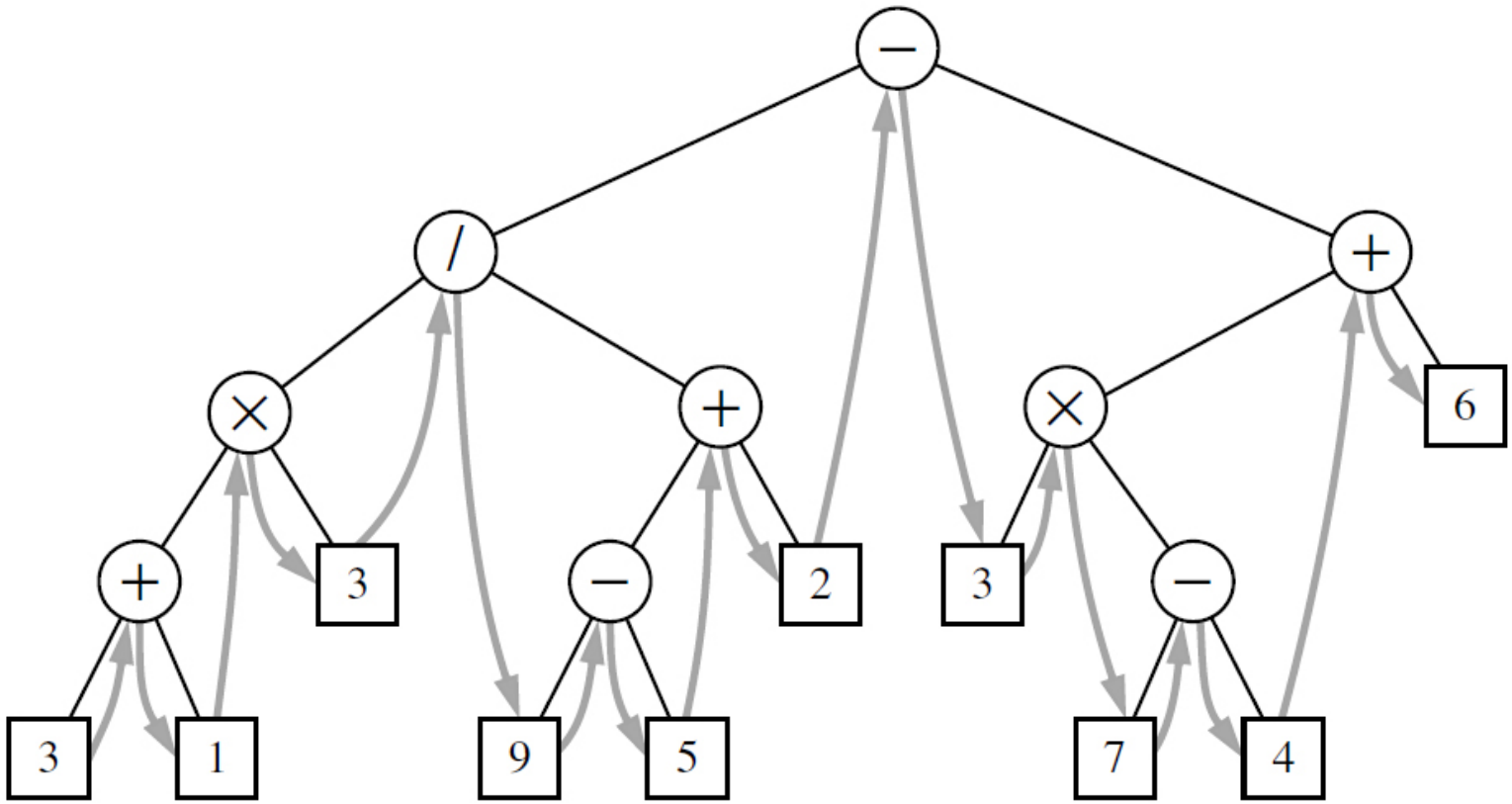
Tree Traversals...Preorder

- Preorder: Visit the root node, do a preorder traversal of the left subtree, and do a preorder traversal of the right subtree



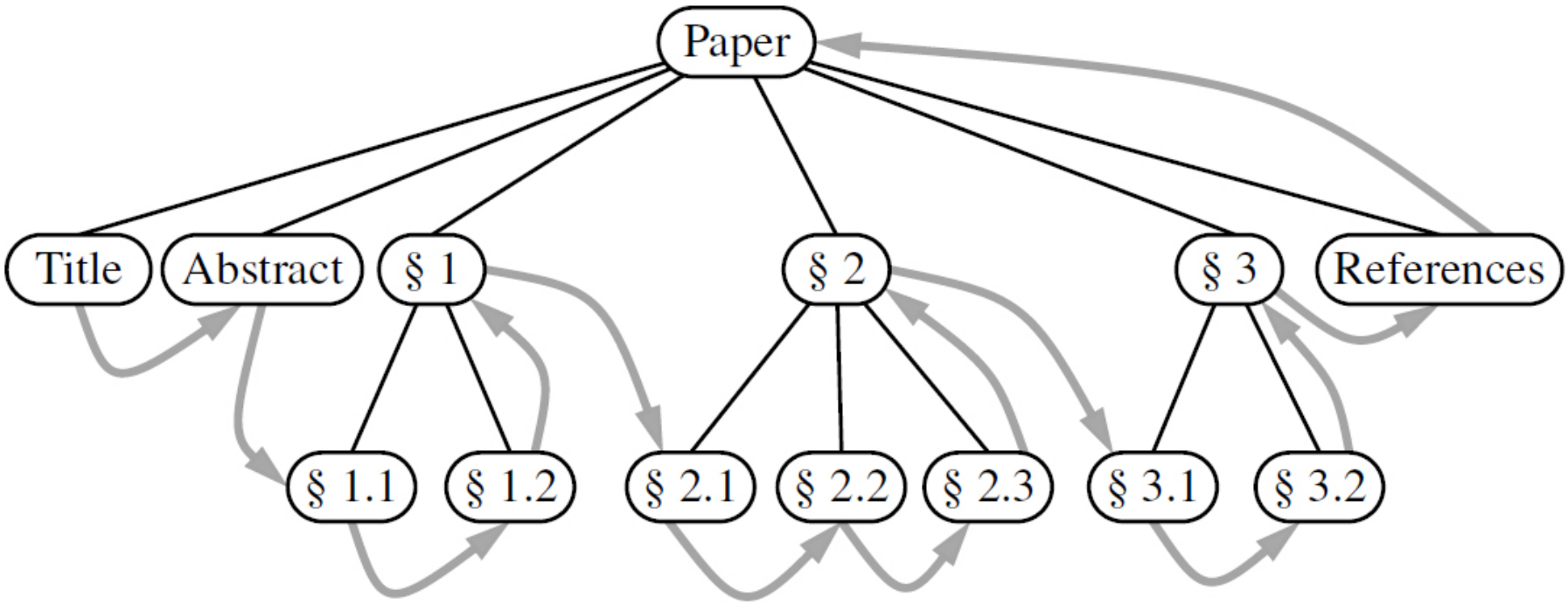
Tree Traversals...Inorder

- Inorder: Do an inorder traversal of the left subtree, visit the root node, and then do an inorder traversal of the right subtree



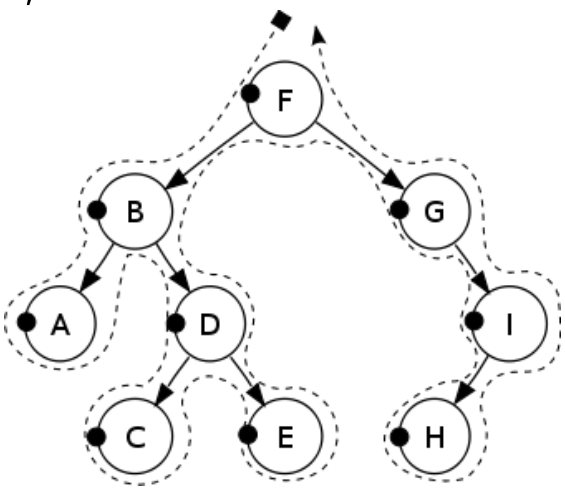
Tree Traversals...Postorder

- Postorder: do a postorder traversal of the left subtree, do a postorder traversal of the right subtree, and visit the root node

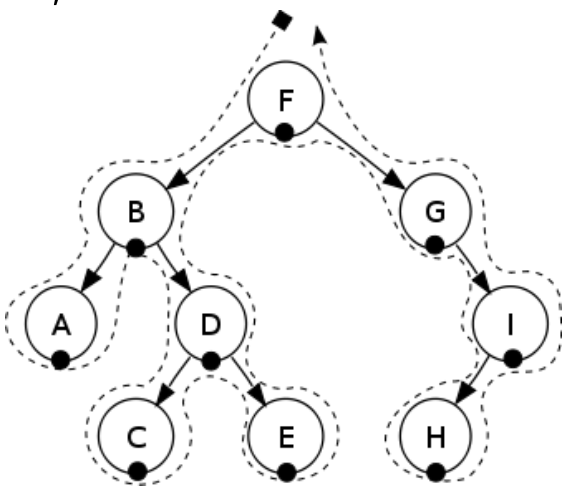


Tree Traversals...

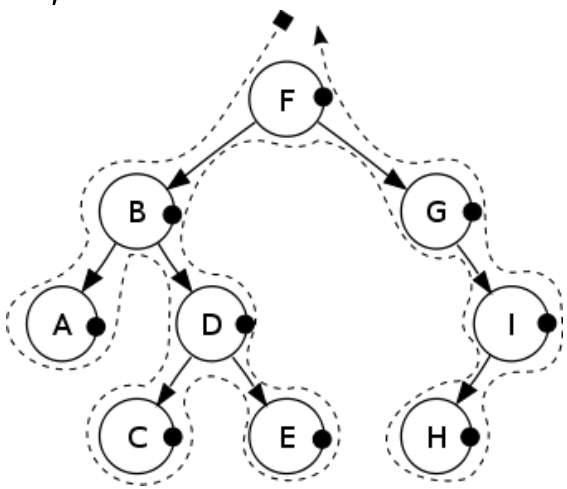
Preorder:
F, B, A, D, C, E, G, I, H



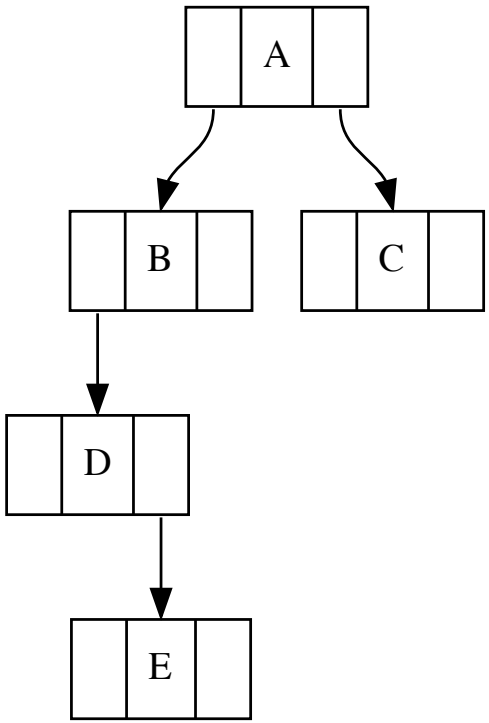
Inorder:
A, B, C, D, E, F, G, H, I



Postorder:
A, C, E, D, B, H, I, G, F

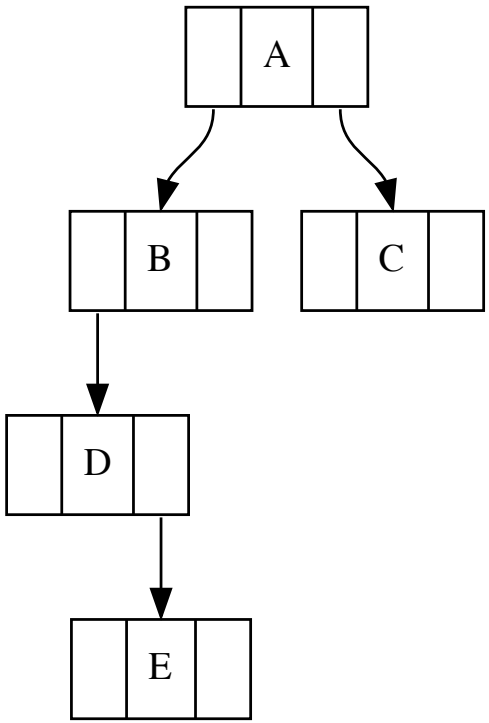


Example: Tree Traversals



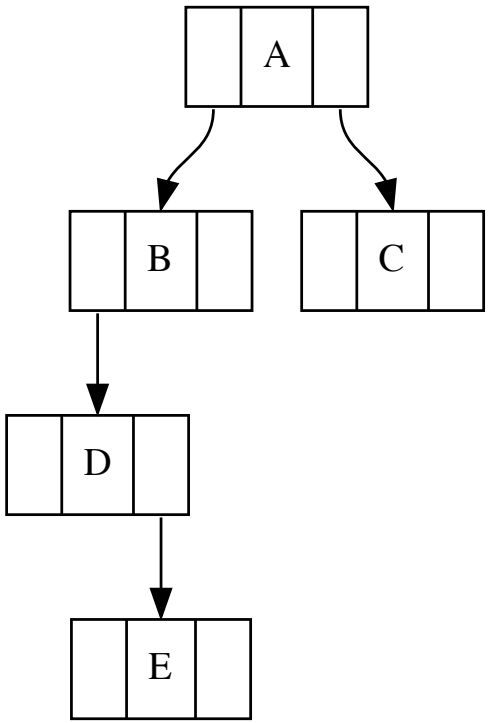
- Preorder:

Example: Tree Traversals



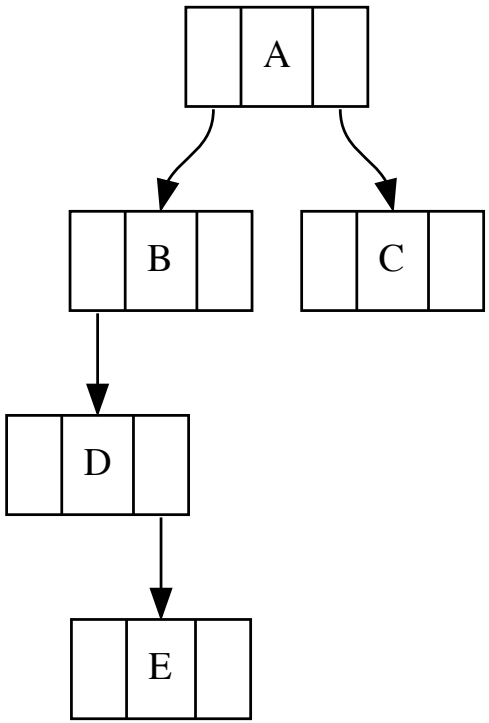
- Preorder: A, B, D, E, C
- Inorder:

Example: Tree Traversals



- Preorder: A, B, D, E, C
- Inorder: D, E, B, A, C
- Postorder:

Example: Tree Traversals



- Preorder: A, B, D, E, C
- Inorder: D, E, B, A, C
- Postorder: E, D, B, C, A

Representing Binary Trees

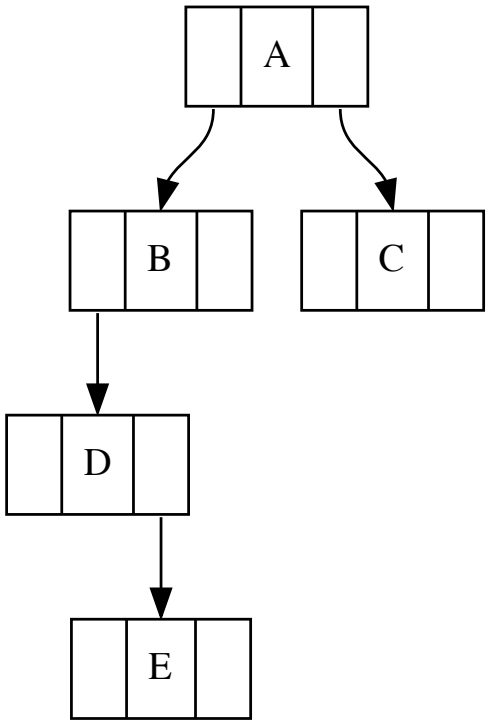
We will represent binary trees in our programs in two ways:

- List of lists, or
- Nodes and references

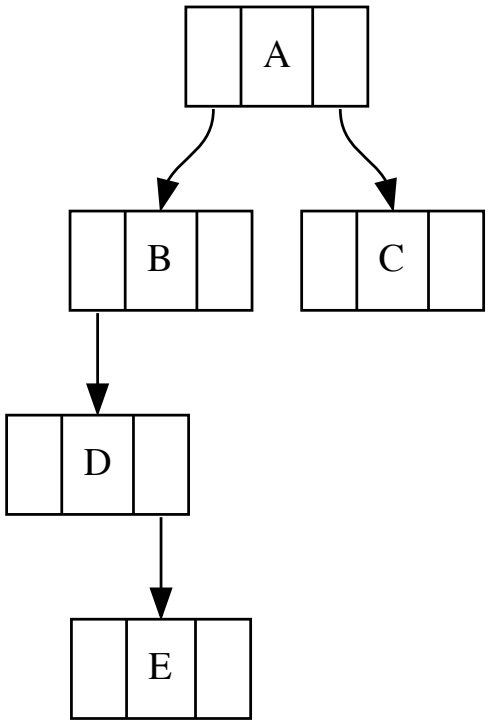
Representation 1: List of Lists

- First element of the list contains the label of the root node
- Second element is the list that represents the left subtree, or `None`
- Third element is the list that represents the right subtree, or `None`
- `[root, left subtree, right subtree]`

Convert to List of Lists



Convert to List of Lists




```
['A',
['B', ['D', None, ['E', None, None]], None],
['C', None, None]]
```

Converting to a Tree

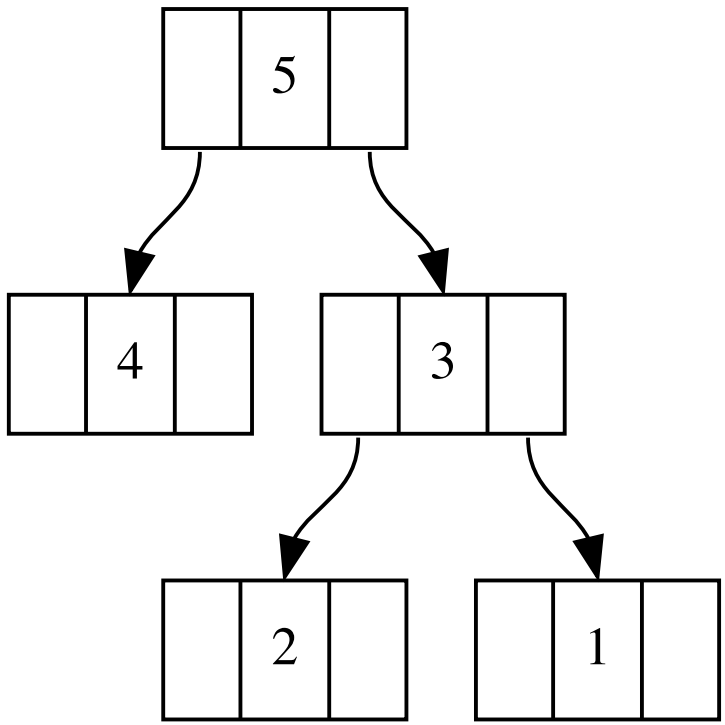
What is the tree represented by this list?

```
[5, [4, None, None],
[3, [2, None, None], [1, None, None]]]
```

Converting to a Tree

What is the tree represented by this list?

```
[5, [4, None, None],
[3, [2, None, None], [1, None, None]]]
```



Implementation of List of Lists

```
# A binary tree (BT) is None or a list of three elements

def binary_tree(value):
  '''(value) -> BT
  Create BT with value as root and no children.
  '''
  return [value, None, None]
```

List of Lists: Methods

Let's write some new methods.

- `insert_left`: add a value as the lef child
- `insert_right`: add a value as the right child
- `preorder`: return a list of node values in preorder
- `inorder`: return a list of node values in inorder
- `postorder`: return a list of node values in postorder
- `contains`: return `True` iff a binary tree contains a value

Implementation of List of Lists

```
def insert_left(bt, value):
  '''(BT, value) -> NoneType
  Insert value as the left node of the root of bt.
  '''
  if not bt:
    raise ValueError('cannot insert into empty tree')
  left_branch = bt.pop(1)
  if not left_branch:
    bt.insert(1, [value, None, None])
  else:
    bt.insert(1, [value, left_branch, None])


def insert_right(bt, value):
  if not bt:
    raise ValueError('cannot insert into empty tree')
  right_branch = bt.pop(2)
  if not right_branch:
    bt.insert(2, [value, None, None])
  else:
    bt.insert(2, [value, None, right_branch])
```

Implementation of List of Lists

```
def preorder(bt):
  '''(BT) -> list
  Return elements of bt in preorder.

  >>> preorder([5, [4, None, None], [3, None, None]]
  [5, 4, 3]
  '''
  if not bt:
    return []
  return [bt[0]] + preorder(bt[1]) + preorder(bt[2])
```

Implementation of List of Lists

```
def contains(bt, value):
  '''(bt, value) -> bool
  Return True iff value is contained in bt.

  >>> contains([5, [4, None, None], [3, [2, None, None], None], 2)
  True
  >>> contains([10, None, None], 5)
  False
  '''
  if not bt:
    return False
  return bt[0] == value or contains(bt[1], value) \
    or contains(bt[2], value)

def contains(bt, value):
  return value in preorder(bt)
```

Representation 2: Nodes and References

- Since the left and right children of a node are each roots of (sub)trees, we can model a tree as a recursive data structure
- Our tree objects will have attributes for the root value, left child and right child

Representation 2: Nodes and References

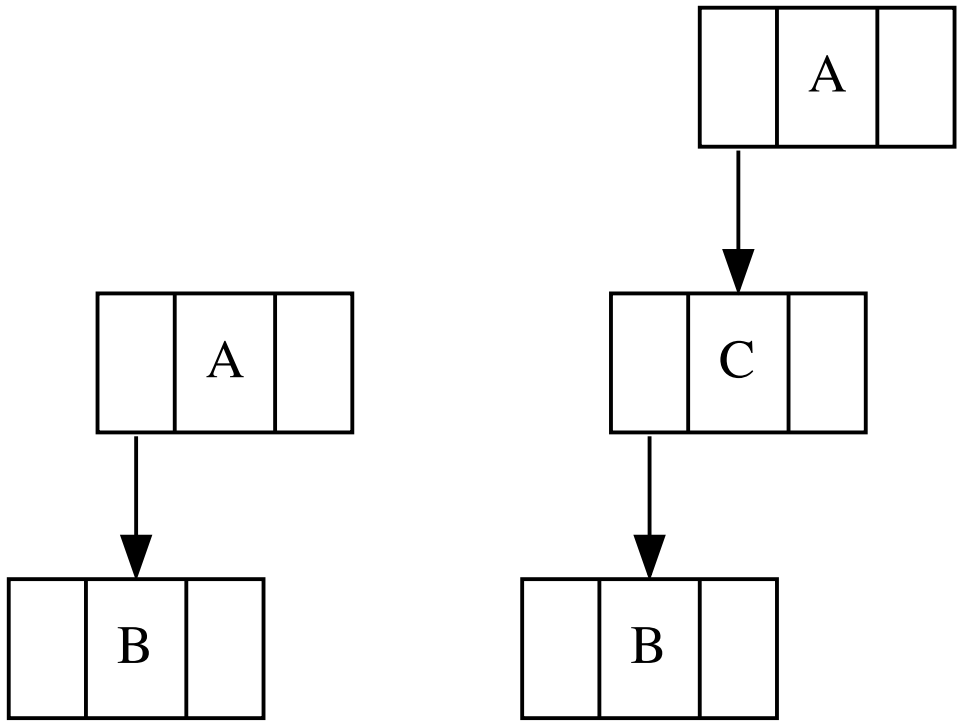
```
class BinaryTree:

  def __init__(self, value):
    self.key = value
    self.left = None
    self.right = None
```

Representation 2: Nodes and References

```
class BinaryTree:

  def __init__(self, value):
    self.key = value
    self.left = None
    self.right = None

  def insert_left(self, value):
    if not self.left:
      self.left = BinaryTree(value)
    else:
      t = BinaryTree(value)
      t.left = self.left
      self.left = t
```

Example of Node Insertion

Inserting a Node Using Code on Previous Slide



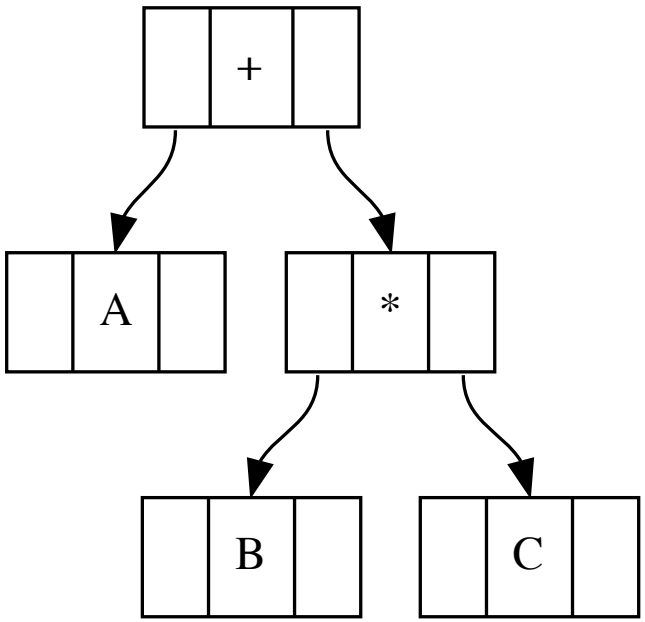
Traversal Code

As with the list of lists representation, we can write traversals recursively.

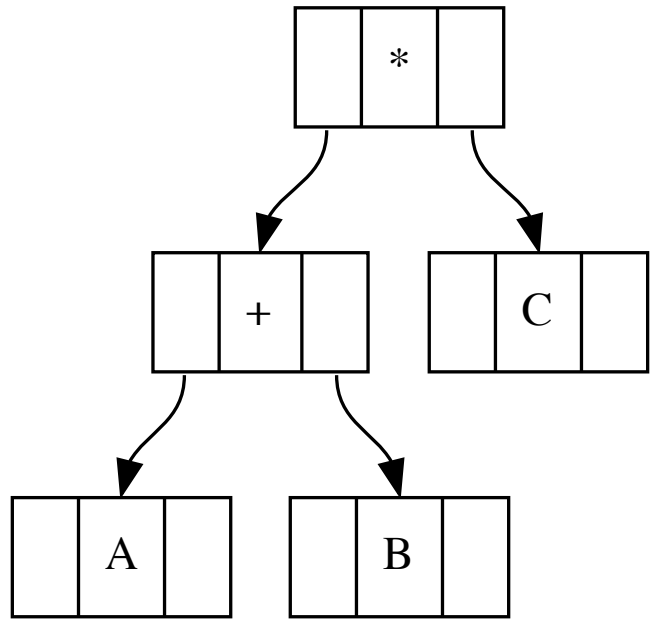
```
from nr import BinaryTree

def preorder(t):
  '''(BinaryTree) -> list'''
  if not t:
    return []
  return [t.key] + preorder(t.left) + preorder(t.right)
```

Expression Trees

Two expression trees forcing different orders of operation



Tree for $A + (B * C)$



Tree for $(A + B) * C$

Uses of Expression Trees

A slightly modified inorder traversal gives us a fully parenthesized expression corresponding to the tree's order of operations.

```
from nr import BinaryTree

def expr(tree):
  if not tree:
    return ''
  s = '(' + expr(tree.left)
  s = s + str(tree.key)
  s = s + expr(tree.right)+')'
  return s
```

Uses of Expression Trees...

- A postorder evaluation gives us a way to evaluate the expression in an expression tree
- Postorder will recursively calculate the value for the left subtree, then the value for the right subtree
- The parent of these two subtrees will be an operator that we can apply to the above results to yield the result for the entire tree
- . . . an exercise for you!

Mystery Traversal Code

- We're going to look at code that uses a queue to do a tree traversal
- It works on the list of lists representation of trees that we've been using
- Two questions
  - What does it do on a sample tree?
  - What does it do in general?

Mystery Traversal Code...

```
from queue import Queue

def some_order(t):
  if t:
    q = Queue()
    q.enqueue(t)

    while not q.is_empty():
      t = q.dequeue()
      print(t[0])
      if t[1]:
        q.enqueue(t[1])
      if t[2]:
        q.enqueue(t[2])

treelist = [
 'a',
 ['b', ['c', ['d', None, None], None], None],
 ['e', None, ['f', None, ['g', ['h', None, None], None]]]]
```

That Sample Tree

```
[
'a',
['b', ['c', ['d', None, None], None], None],
['e', None, ['f', None, ['g', ['h', None, None], None]]]]
```