CSC148H Week 8

Sadia Sharmin

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:
  - Function calls in programs (stack)
  - A lineup in a bank (queue)
  - Event-handling based on timestamps (priority queue)

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 try 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 end and we are back up one level?
- Other examples:
  - Structure of an HTML document
  - Structure of a Python program

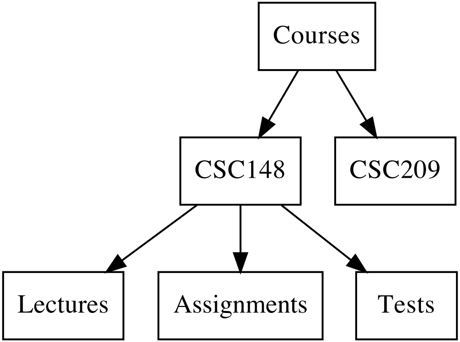
Tree Definition

- A tree has one node called the **root**. This is at the very top of the tree structure.
- The root can have a set of nodes that are connected to it (each of these nodes is a **child** of the root node).
- Each of these nodes can have their **own children**.
- Nodes at the bottom of the tree structure have **no children**. These nodes are called **leaves**.
- These nodes often also have a **value or label**.

Tree Definition

- A tree has a set of nodes (often with values or labels), and directed edges that connect nodes
- One node is the **root**
- Every node besides the root has exactly one parent

Sample Tree



- How many nodes are there? What are they?
- How many edges are there? What are they?
- What is the root?

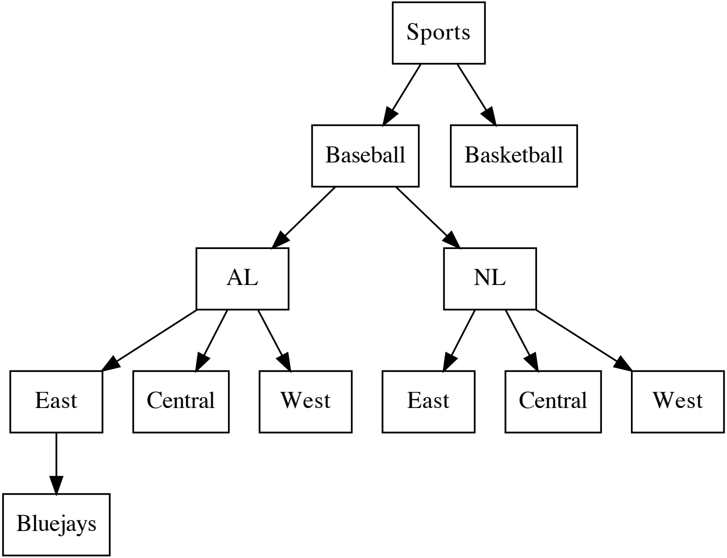
Tree Terminology

- 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: node that has no children (i.e. no outgoing edges)
- Internal node: nonleaf node
- Path: sequence of nodes $n_1, n_2, \ldots, n_k$, where there is an edge from n_1 to n_2, n_2 to n_3, etc.
- Descendant: node n is a descendant of some other node p if there is a path from p to n
- 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 and the edges among them

Tree Terminology...

- Branching Factor: maximum number of children of any node
- 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
- Length of a path: number of edges on a path
- Tree height: maximum of all node levels

Another Sample Tree



What is the height of the tree? Branching factor? Depth of Baseball? Length of path from Sports to AL?

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.

Representing Binary Trees

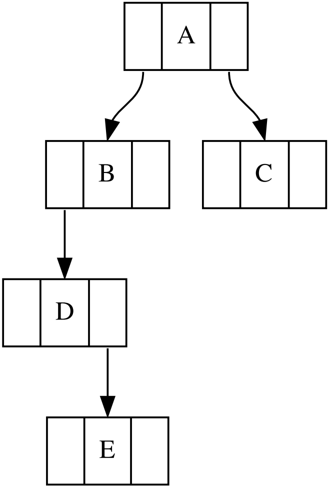
We will represent binary trees in our programs in two ways:

- List of lists, or
- Nodes and references

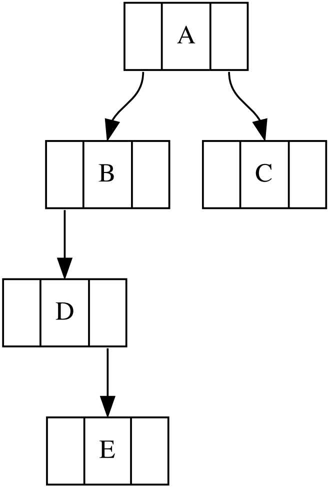
Representation 1: List of Lists

- 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`

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

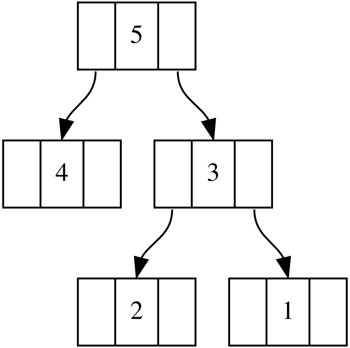
What is the tree represented by this list?

```
[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]

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])
```

List of Lists: Methods

Let's write some new methods.

- `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 the binary tree contains a given value

Representation 2: Nodes and References

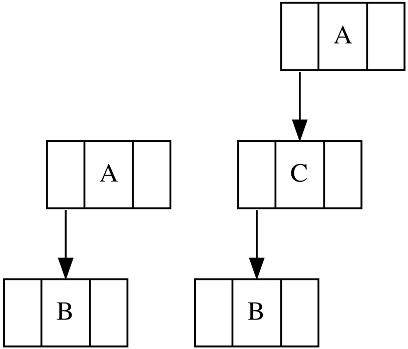
- Since the left and right children of a node are each roots of (sub)trees, we can model a tree as a recursive data structure
- Our tree objects will have attributes for the root value, left child and right child

```
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



(a) Original Tree

(b) Original Tree after inserting node C

Figure: Inserting a Node Using Code on Previous Slide

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, change, etc.)
- Lists have two obvious traversals: left to right, and right to left
- What are some ways we can systematically visit each node in a tree?

Visit each level one-by-one

- We can start at the root
- Then look at all of the root's immediate children
- Then look at all the children of the first child, and then the second child, then third, and so on
- And repeat this through the whole tree
- This algorithm is called **breadth-first search**

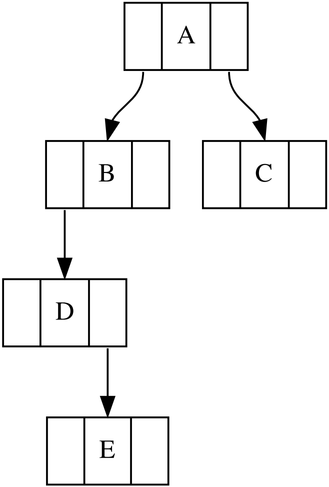
Depth-first search

- We can start at the root
- Then look at one of the root's immediate children
- Then look at one of this child's children
- Then one of that child's children and so on, until we reach a leaf. Then we go back to the last parent and repeat for the next child.
- Basically, we go as deep as we can into the tree on one child before moving on to the next sibling
- This search algorithm is called **depth-first search**

DFS Tree Traversals...

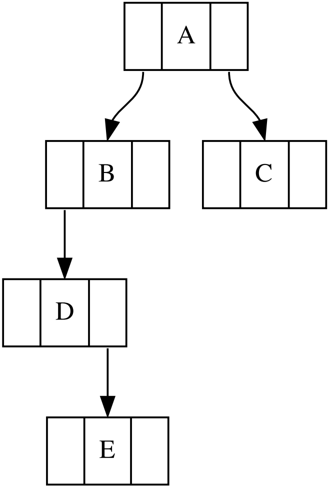
- Preorder: Visit the root node, do a preorder traversal of the left subtree, and do a preorder traversal of the right subtree
- Inorder: Do an inorder traversal of the left subtree, visit the root node, and then do an inorder traversal of the right subtree
- Postorder: do a postorder traversal of the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

Example: Tree Traversals



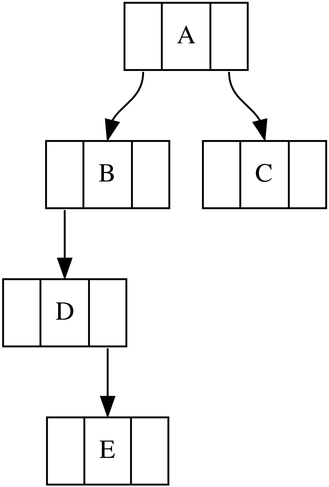
- Preorder:

Example: Tree Traversals



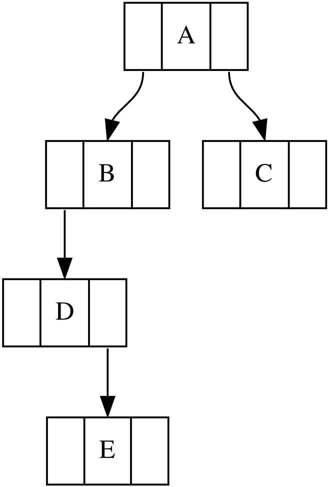
- Preorder: A, B, D, E, C
- Inorder:

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

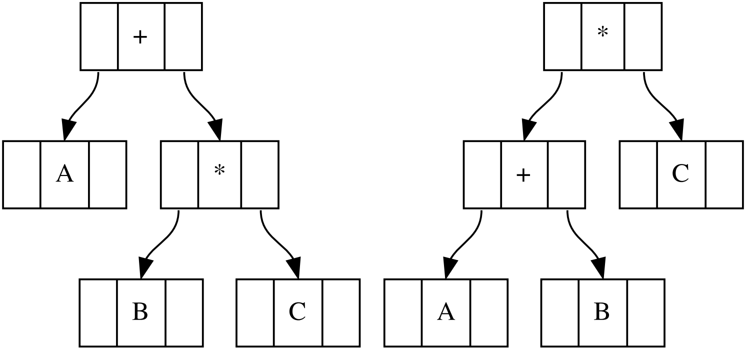
Traversal Code

Recursion helps us write concise traversal code directly from its definition:

```
from nr import BinaryTree

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

Expression Trees



(a) Tree for $A + (B * C)$ (b) Tree for $(A + B) * C$

Figure: Two expression trees forcing different orders of operation

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 my_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]]]]
```