IbraFSG™ 3 - Week 8; Trees and Recursive Data Structures

Ibrahim Chehab

UTM RGASC

March 6, 2024

Table of Contents

Welcome back to IbraFSGs™

- Welcome back to IbraFSGs™! Hello to new people and welcome back to tenured members.
- This week we will be discussing *Trees and Recurisve Data Structures*.
- Trees are a very important data structure in Computer Science, and are used in many applications. They will play a key role in the remainder of the course, and will be a main character in A2
- There are several types of trees:
  - Binary Trees
  - Terenary Trees
  - Binary Search Trees (BSTs)
  - AVL Trees [Also known as a balanced BST]
  - Abstract Syntax Trees (ASTs) [also known as Expression Trees]
  - Heaps
  - Normal Trees
- CSC148 Focuses on Binary Trees, BSTs, ASTs, and normal Trees

A Recap of the UltraSheet™

- An *UltraSheet™* is a "cheat sheet" that you compile for **yourself** to review course materials
  - Sharing UltraSheets™ is **counter-productive** and **will not help you learn the material**
  - However, reviewing content in a group and simultaneously updating your UltraSheets™ is a good idea
- It acts like your own personalized textbook chapter
  - It allows you to **regurgitate all the course information in a contigous, organized manner** and helps you **find gaps in your knowledge**
- UltraSheets™ help with type 1 and 2 questions
  - Can you remember what type 1 and 2 questions are?
  - Can you remember what a *type 3* question is? :troll:

Key Terms

Required Key Terms: The following key terms are required for this week's content. You should be able to define and explain these terms in your UltraSheets™:

- **Tree**
- **Parent, Child, and Ancestor**
- **Leaf**
- **Root**
- **Subtree**
- **Branching Factor**
- **Height**

Key Terms (Cont'd)

Required Key Terms: The following key terms are required for this week's content. You should be able to define and explain these terms in your UltraSheets™:

- **Tree**
- **Parent, Child, and Ancestor**
- **Leaf** *Hint: These are typically your base-cases in recursive functions*
- **Root**
- **Subtree**
- **Branching Factor**
- **Height**

Key Terms (Cont'd)

Required Key Terms: The following key terms are required for this week's content. You should be able to define and explain these terms in your UltraSheets™:

- **Tree**
- **Parent, Child, and Ancestor**
- **Leaf** *Hint: These are typically your base-cases in recursive functions*
- **Root** *Hint: Think about how this can be applied recursively when writing recursive functions*
- **Subtree**
- **Branching Factor**
- **Height**

Key Terms (Cont'd)

Required Key Terms: The following key terms are required for this week's content. You should be able to define and explain these terms in your UltraSheets™:

- **Tree**
- **Parent, Child, and Ancestor**
- **Leaf** *Hint: These are typically your base-cases in recursive functions*
- **Root** *Hint: Think about how this can be applied recursively when writing recursive functions*
- **Subtree** *Hint: Relate this to nested lists. Hint #2: This term fits in very nicely with my recursion analogy from 2 weeks ago*
- **Branching Factor**
- **Height**

Key Terms (Cont'd)

Reccomended Key Terms: The following key terms are recommended for this week's content. Most are out of the scope of the course, but are fun for extra learning:

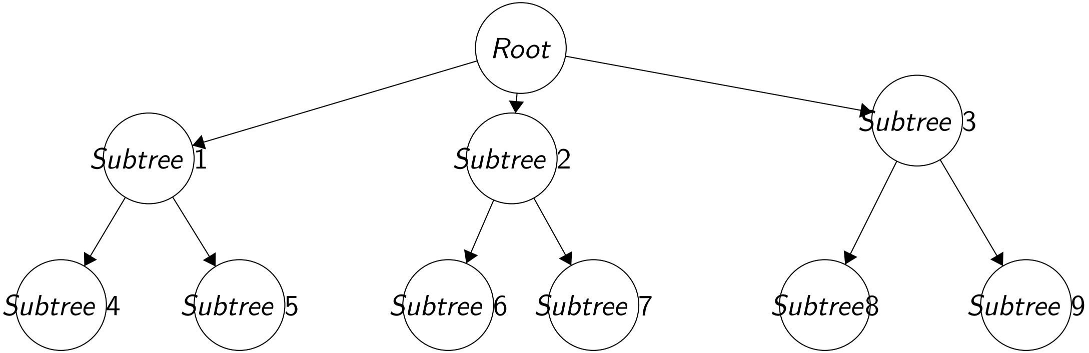
- **Graph:** A family of data structures that are used to represent relationships between objects.
- **Balancing Factor:** Useful in CSC263 when talking about *AVL Trees* - Which are self-balancing BSTs
- **Cycle:** Similar to a tree, however is not strictly linearly connected.

A note about today's FSG

1. If today's FSG feels "all over the place"; that's because it is. Trees are a very broad topic, and we're only scratching the surface today.
2. I can't go over *everything* today since you're still learning about trees in CSC148
3. Next week, we'll have more detailed and focused examples, after you've had more time to learn about trees in CSC148

Base Cases, Recursions, and Trees

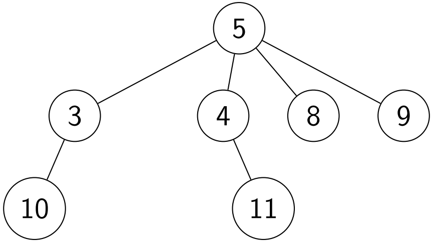
Recall my recursion analogy from two FSGs ago:
A recursive function aims to do the least amount of work in the current recursive call before delegating the rest of the work to the next recursive call
How can we apply this to trees?
More generally, what is the *Minimum amount of work*, and what is the *Work delegated*?



Hint: Think about how the term Subtree can be applied recursively, and think about the context of a subtree

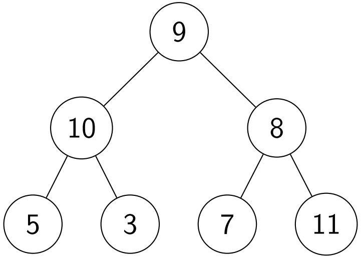
Practice Problem I: TraversalOverflow

Dipal's computer was hit by a solar flare while he was working on his FSG Homework. As a result, all his trees were converted into their respective list representations. Unfortunately, he doesn't know what kind of traversals were used! He needs your help to match up the list representations with their respective trees.



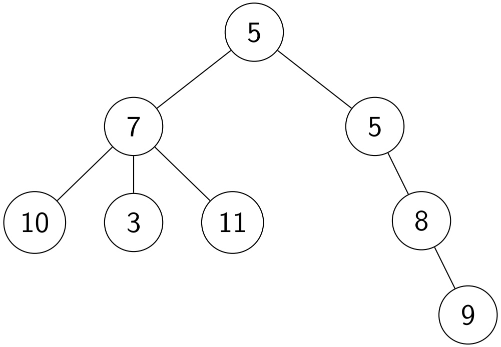
Which of the following traversals apply to this tree?
1. `5, 3, 10, 7, 11, 8, 9.` **2.** `5, 11, 10, 9, 8, 4, 3.` **3.** `10, 3, 11, 7, 8, 9, 5.` **4.** `10, 3, 11, 4, 8, 9, 5.`

Practice Problem I: TraversalOverflow



Which of the following traversals apply to this tree?
1. `5, 3, 10, 7, 11, 8, 9.` **2.** `9, 10, 8, 5, 3, 7, 11` **3.** `10, 3, 11, 7, 8, 9, 5.` **4.** `10, 3, 11, 4, 8, 9, 5.` **Bonus!** What type of tree is this? *This isn't a question you can answer this week, I'm just curious who read ahead*

Practice Problem I: TraversalOverflow



Which of the following traversals apply to this tree?

1. `5, 3, 10, 7, 11, 8, 9.` **2.** `9, 10, 8, 5, 3, 7, 11` **3.** `10, 3, 11, 7, 8, 9, 5.` **4.** `10, 3, 11, 4, 8, 9, 5.`

TraversalOverflow Debrief

What did you notice with the preoder and postorder traversals of each tree?

- Were they unique?
- Were they bijective?
- Was it possible to determine the tree from the traversal, or did you need to reference the tree first?

What does this tell you about the uniqueness of the pre-order and post-order traversals?

- Are they unique?
- Can you accurately reconstruct the tree from the traversal? Why or why not?
- Is there a bijection between the traversal and the tree?

Keep this in the back of your head for next week's FSG

Practice Problem II: To Traverse! Or Not To Traverse!

Bingleton is working on a method that will help him determine whether a given tree is a *Cycle* or not. *Note: A cycle is a tree that has a node that is both a parent and a child of itself. That is, a Node that has a child that's further up the tree.*

```python
class TreeNode:
    """
    Class representing a node in a tree
    """
    # Implementation omitted; Assume it's a standard TreeNode class

    def is_cycle(self) -> bool:
        """
        Method which determines whether the tree is a cycle or not
        """
        # TODO: Implement this method
```

Hint 1: Consider the type of traversal you're going to use carefully. Sometimes recursion isn't the best option. Hint 2: If you choose to use recursion, consider using a helper function

A final challenge...

As per usual, you're getting homework this week!

RESTRICTIONS:

- ...None!

But no, that doesn't make the question any easier :P

Chicago Med's File Problem: Dr. Choi is facing a challenging problem with organizing his medical files. His computer's hard drive contains a complex arrangement of directories and files, forming a tree-like hierarchical structure. However, he's noticed that there are numerous duplicate entries scattered throughout this structure, leading to unnecessary clutter and confusion.
To address this issue, Dr. Choi needs assistance in developing a script that will efficiently remove all duplicate file entries from his directory tree. Your task is to implement a method that takes the root node of this file system tree as input and eliminates all duplicate file entries, ensuring that each unique file is retained only once. You can assume the first entry of the file you find is the one to be kept.

A final challenge...

```
from __future__ import annotations

class FileNode:
    """
    Class representing a file in a file system

    === Representation Invariants ===
    - file_size >= 0
    len(file_name) > 0
    if default program is None, file_type is also None

    """
    file_name: str
    file_size: float
    file_type: str
    default_program: str

    def __init__(self, file_name: str, file_size: float, file_type:
   str = None, default_program: str = None):
        self.file_name = file_name
```