# Project 3 Documentation:

# Binary Trees and Recursion

Vincent Testagrossa

University of Maryland Global Campus

CMSC 350: Data Structures and Analysis

Dr. Jim Huskins

June 26th, 2022

# Contents

## [Problem Statement](#_Problem_Statement_1)

## [UML Class Diagram](#_UML_Class_Diagram)

## [Testing](#_Testing)

## [Reflection](#_Reflection)

# Problem Statement

Write a program that allows the user to enter a binary tree in a parenthesized prefix format and then allows it to display the following information or be categorized in the following ways:

1. Whether the binary tree is balanced (absolute difference between left and right subtrees is at most 1)

2. Whether the binary tree is full (maximum number of nodes for the number of levels)

3. Whether the binary tree is proper (each node has 2 or 0 children)

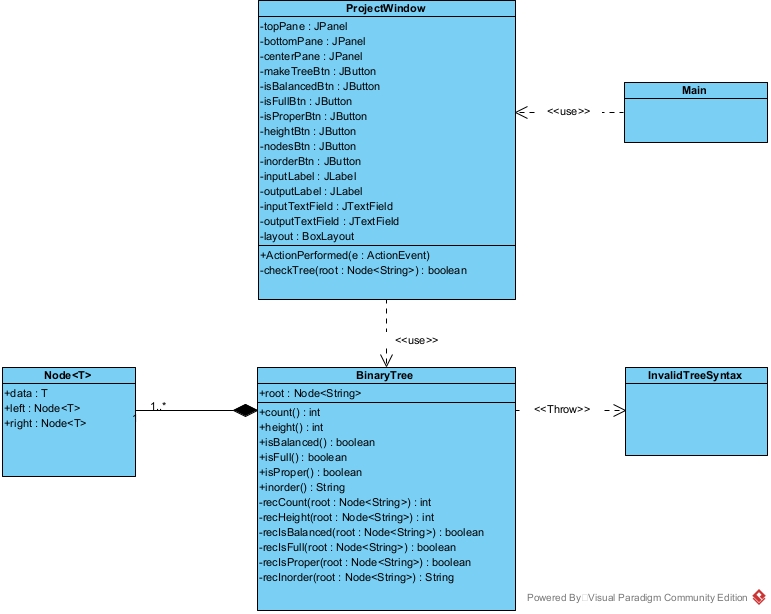
5. Displays the height of the tree (max level of all nodes starting with 0 as the first)

6. Displays the number of nodes in a tree

7. Displays a fully parenthesized inorder traversal of the tree.

Each of the preceding functions is a public method called by a button press in a GUI on the main class. The public methods belong to *BinaryTree* and call recursive methods to perform the required task. *InvalidTreeFormat* is a handled exception thrown by the constructor of *BinaryTree* if the format is incorrect. The *GUI* will handle the exception and display the message in a JOptionPane.

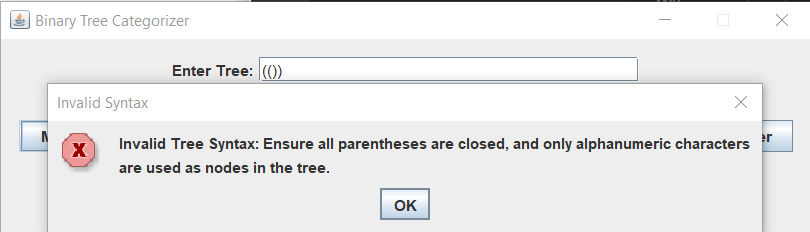
# UML Class Diagram



# Testing

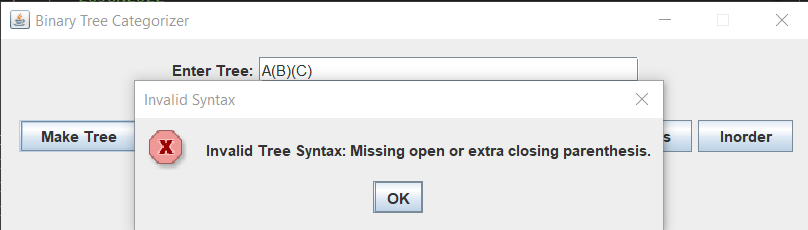
## Testing InvalidTreeSyntax:

Input: (())

Output: 

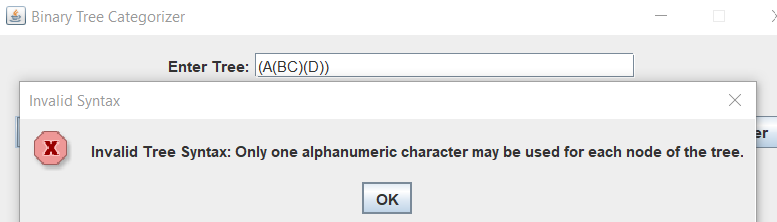
Input: Test of missing open and close parens A(B)(C)

Output:



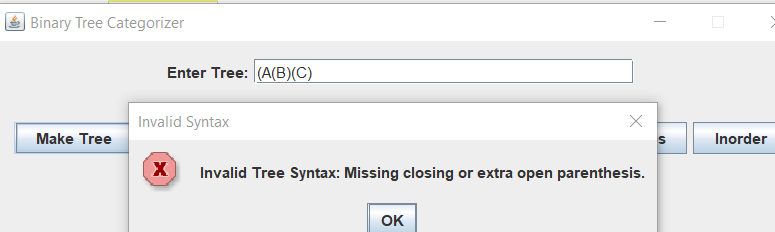
Input: Test of extra character (A(BC)(D))

Output:



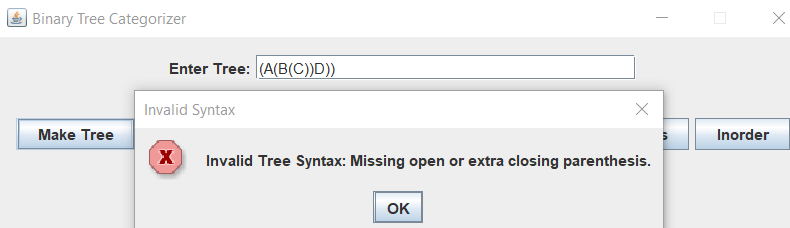
Input: Test of missing external parens (A(B)(C)

Output:



Input: Test of missing internal parens (A(B(C))D))

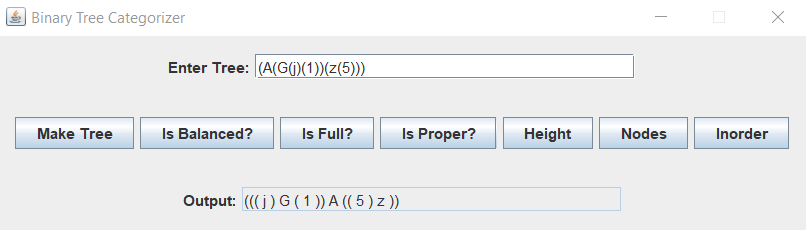
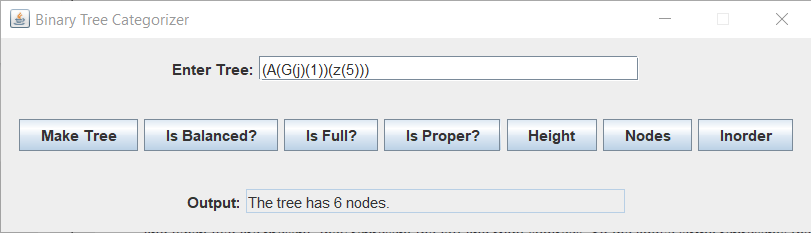
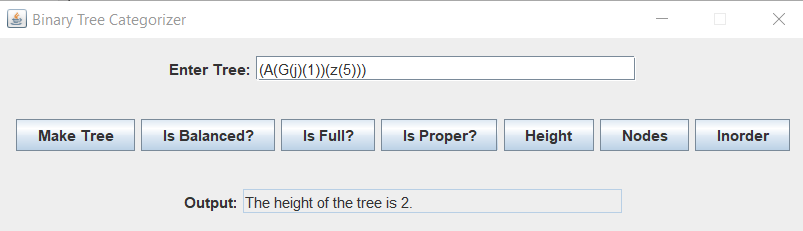
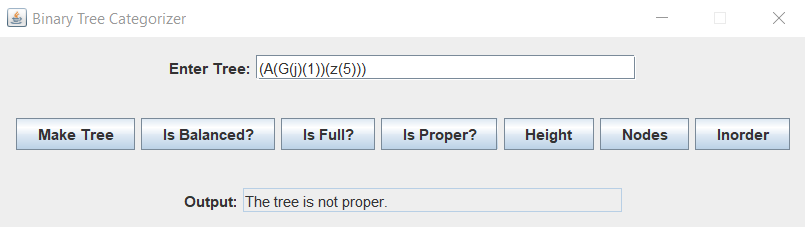
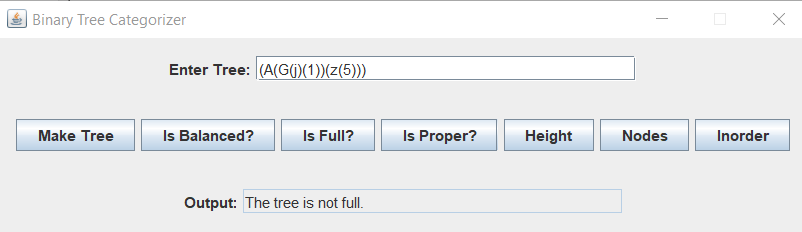
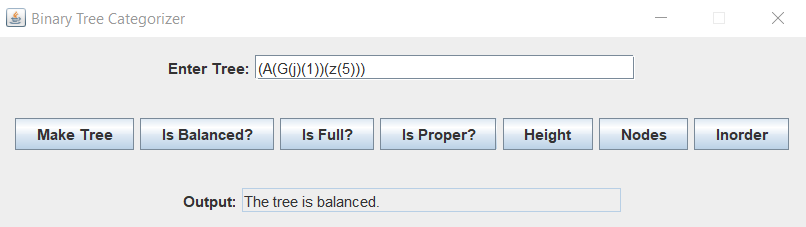
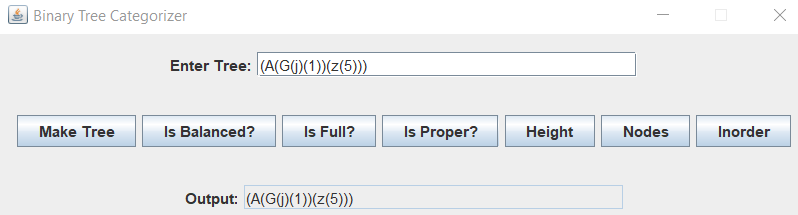
Output:



## Testing Valid Formats:

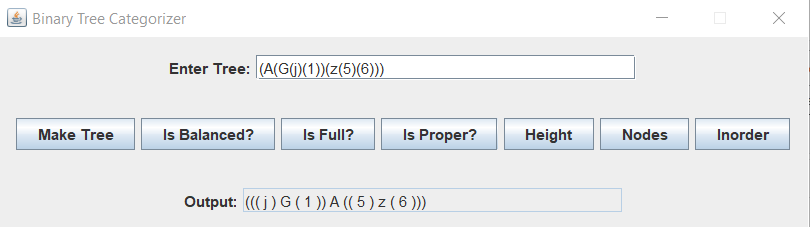
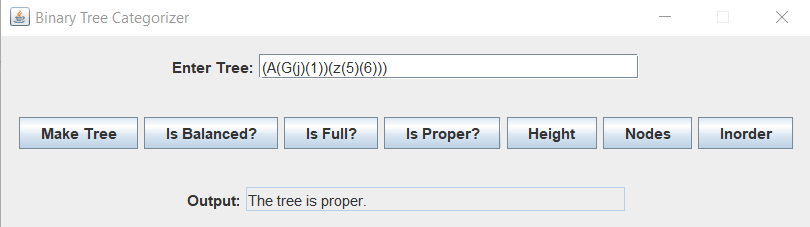
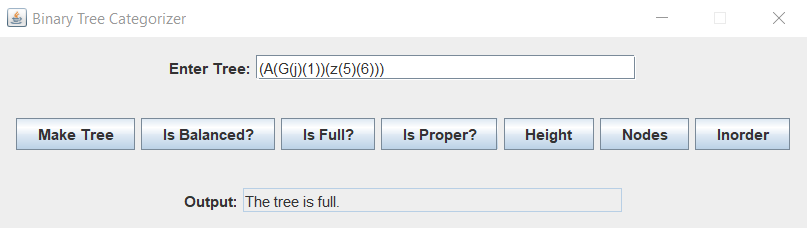
Input: (A(G(j)(1))(z(5)))

Output:



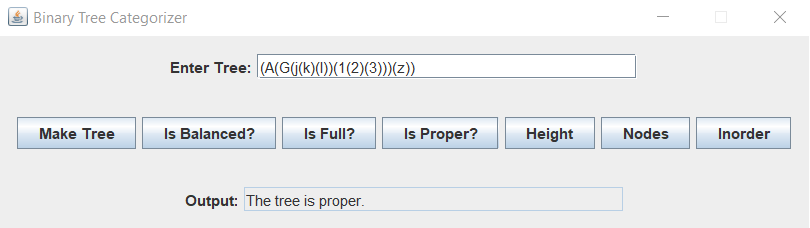
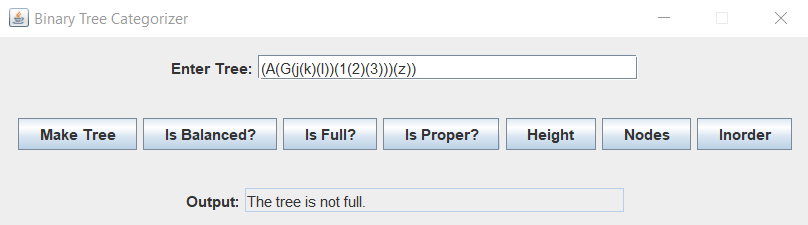
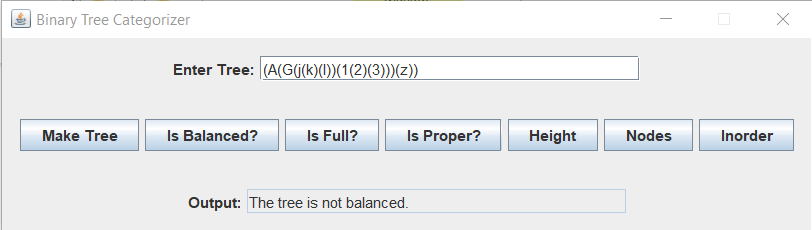
Input: Full tree (A(G(j)(1))(z(5)(6)))

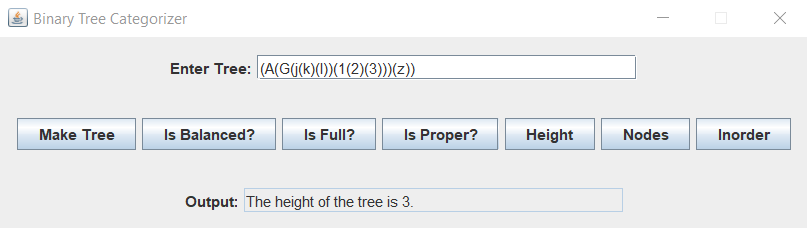
Output: Tree is full and proper.



Input: Proper tree that isn’t full, or balanced (A(G(j(k)(l))(1(2)(3)))(z))

Output:





# Reflection

The tree itself wasn’t too difficult to create. It’s not much different from a linked list, internally. The harder part is thinking recursively and completing each task using recursion. I’m glad I got the practice because I definitely needed it. The biggest hurdle I had was probably in the constructor. I was trying to find a balance between keeping the variable scope contained and having more flexibility with design but was very set against using external counters for my methods, which made things a bit tougher than absolutely necessary. Parsing the initial string was also a challenge because of the parentheses surrounding the whole statement. I decided to drop the parentheses initially after performing a check for them (to ensure formatting), but I was overthinking it a little bit, as later on I used the findIndex method in a way that could have been adapted for that purpose, and made things easier for me in the long run.