Programming assignment 4: General trees

Part 1 (50%): Design a tree representing a directory structure in a computer.
Implement a program that allows the user to traverse such a tree, at any given time being in a current directory. The user can then give commands that affect that current directory or the current location itself.
The following commands can be given:

- ***mkdir <name>***: Makes a new node in the tree, with that name, under the current node. Displays an error message if the directory already has a subdirectory with that name.
- ***cd <name>***: moves the current location (current node) to its child with the name <name>. If the current node has no child node with that name a message is displayed (already written in the base code).
  If <name> is '..' the current location is instead moved to its parent.
  If the current location is the **root** the command 'cd ..' will exit the program.
- ***ls***: prints the names of all the children (only the children, not the entire subtree) of the current node, each in its own line. ***The names should be ordered alphabetically*** (*The order that <, >, ==, etc. give on a string is adequate*).
- ***rm*** <name>: Removes a subdirectory with that name from the current directory. Displays an error message if the directory has no subdirectory with that name.

It is not necessary to use the code given in DirectoryTreeBase.zip, but all the success and error messages are correctly written there, so please use those. It is not necessary to use recursion but a recursive solution may make some things simpler, for example going "up" in the tree ('cd ..'). It is OK to use full functionality of python lists to supplement your program, but the overall structure must be a tree, implemented with your own nodes.
Bonus 5%: Implement the solution without using any built in python structures (not even a list with array limitations). This could be done by implementing the children/siblings in an encapsulating linked list or implementing the sibling list as links within the TreeNode itself. Basically full marks for implementing the entire solution with nodes.

In this program you can add any operations, classes, parameters and variables. Nothing has to fit with a ADT or class specification except the input and output of the program itself. You can add any parameters to __init__ operations and any other operations. You can add Exception classes and use them, raise, try and except, within your program structure.

Part 2 (50%): Implement a prefix parser using a binary tree
Read all about prefix statements and statement trees in the original class assignments on recursion and in the warm-up class exercise on trees.
In this assignment you will not only parse the statement and give a solution. Instead the program will build a tree. It is then possible to print the contents of the tree, get the value of the root, simplify the tree and solve for an unknown value in the tree.

In some of the statements you will see a string value (x) instead of a number. Your statement tree should be able to have leaves that are both numbers and unknown values:
PREFIX: - + - 7 x + 2 3 4
INFIX: (((7 - x) + (2 + 3)) - 4)
POSTFIX: 7 x - 2 3 + + 4 -

Finish implementing the class PrefixParseTree. It has the following operations:

- ***load_statement_string(statement)***: It takes in a prefix statement as a string, parses the string (you can use the tokenizer class from the recursive prefix parser) and builds a statement tree.
- ***set_format(out_format)***: It takes in an enumerator value that decides how the tree should be printed. It will be printed in this format until the format is changed. The default format (if this operation has not been called) should be ***OutputFormat.PREFIX***.
    - The possible values are:
        - ***OutputFormat.PREFIX***
        - ***OutputFormat.INFIX***
        - ***OutputFormat.POSTFIX***
- ***__str__()***: Returns a string with the statement from the tree. The output format desides how it is formatted.
    - ***OutputFormat.PREFIX***
        - The original prefix statement: + 3 + * 4 7 2
    - ***OutputFormat.INFIX***
        - Infix statement with parentheses for disambiguation: (3 + ((4 * 7) + 2))
    - ***OutputFormat.POSTFIX***
        - Postfix statement: 3 4 7 * 2 + +
- ***root_value()***: This operation returns the value of the tree, basically the value of the whole statement.
  If there is a division by zero raise DivisionByZero()
  If there is an unknown value in the tree raise UnknownInTree()
- *More on next page...*

- ***simplify_tree()***: This operation calculates the full values of any subtrees that do not have an unknown and combines that value into a single node.
    - If a division by zero occurs, the division operator node should be left in the tree, but its children simplified as much as possible.
    - Look at specific cases, particularly cases with an unknown and cases with division by zero. In the expected_out.txt file.
        - These are all easy to search for in the text file

  Examples:
    - "+ 3 4" simply becomes "7" so what is originally three nodes, "+" with a left node "3" and a right node "4" becomes a single node "7" with no left or right children.
    - PREFIX: - + - x 5 + 6 3 4
      SIMPLIFIED: - + - x 5 9 4
    - PREFIX: - + - 8 5 + 6 3 x
      SIMPLIFIED: - 12 x
    - PREFIX: / - - 3 / 4 / - 8 9 0 9 / / + * 9 5 * 0 5 5 8
      SIMPLIFIED: / - - 3 / 4 / -1 0 9 1.125
- ***solve_tree(root_value)***:
    - If the tree has an unknown value you can not calculate the root value. Instead we take the root value as a parameter and return the value of the unknown (x) that would give that root value.
    - *This only needs to work for statements with + and - (except for bonus), and it only needs to solve for a tree with one instance of one unknown.*
    - PREFIX: - + - 8 5 + 6 3 x
      The value of x if the root_value is 7 is: 5
    - PREFIX: + x 6
      The value of x if the root_value is 13 is: 7

Bonus 5% For 100% correct output in PrefixParseTree (in teacher's tests)