

[previous](#) [up](#) [next](#)

## Exercises

You will need the Node class from lecture!

### New tree methods

- Write a `count(self)` method to the Node class which counts the number of nodes in the tree!
- Write a `sum(self)` method which returns the sum of elements in the tree!
- Write a `height(self)` method which returns the height (depth) of the tree!
- Write an `is_list(self)` method which returns True if the tree is a long path without a junction. False otherwise. The tree is a list if all of the nodes have at most one branches.

## Calculator

We will improve the expression tree from the lecture. Make an empty Node class first!

- Write its constructor with one parameter: a string containing the mathematical expression to calculate. Let's suppose for now that there are only two operations: + and \* and there are no parenthesis, neither negative numbers.
  - ◆ If the string is a number, then store that number in `self.data` as a float number.
  - ◆ If not then search for an operation in it. Cut the string at a + character (if found any) and store the operator in `self.data`. Also set `self.left` to the recursive result on the first part of the string (before the operation) and set the `self.right` to the recursive result on the second part (after the operation).
  - ◆ Make the same with \* operation if there was no +
- Write a `calculate` method for this class
- Implement other operations such as - / and power: ^
- Write a `__repr__` method which prints the expression
  - ◆ This was already implemented in the lecture just make it a method.
  - ◆ Implement a so called polish normal form

You can even handle parenthesis. First remember the parenthesis function from the previous lab!

- If the input string is entirely inside a parenthesis then erase the outermost parenthesis
- Find the lowest precedence operation outside a parenthesis
- Use recursion!

[previous](#) [up](#) [next](#)