Assignment 2

Report

|  |  |
| --- | --- |
| School: | University of Aberdeen &  South China Normal University |
| Major: | Computer Science |
| Name: | Ruofan Liao |
| Student Number: | 50079732 |
| Date: | May 25th, 2021 |

Contents

**[1. Introduction 3](#_Toc104454222)**

[1.1 File structure 3](#_Toc104454223)

[1.2 Function list 3](#_Toc104454224)

**[2. Basic design for the program 4](#_Toc104454225)**

[2.1 Description of the program 4](#_Toc104454226)

[2.2 Comment about the implementation of all the six tasks 4](#_Toc104454227)

2.2.1 Task 1 ………..………………………………………………………………………………………………………..4

2.2.2 Task 2 ………..………………………………………………………………………………………………………..4

2.2.3 Task 3 ..………………………………………………………………………………………………………………..5

2.2.4 Task 4 ..………………………………………………………………………………………………………………..5

2.2.5 Task 5 …..……………………………………………………………………………………………………………..6

2.2.6 Task 6 ..………………………………………………………………………………………………………………..7

**[3. Description of the algorithmic choices made for the application 8](#_Toc104454228)**

[3.1 Justification of selecting Data Structures for the program 8](#_Toc104454229)

3.1.1 Stack .….………………..…………………………………………………………………………………………….8

3.1.2 Binary Tree .………………………………………………….……………………………………………………..8

[3.2 Description and justification of the Classes used for the program 9](#_Toc104454230)

3.2.1 BiTreeNode ..……………………………………………………………………………………………………….9

3.2.2 TestException & TestExpression …………………………………………..………………………………9

[3.3 Snippets of the unit tests, and description and comments of the tests output 9](#_Toc104454231)

**[4. Reference list 10](#_Toc104454232)**

# Introduction

## 1.1 File structure

├── README.md                   // help

├── tree\_demo

│   └── default\_read.pkl               // the default document for reading the tree

│   └── default\_save.pkl               // the default document for saving the tree

├── assignment.py               // core code for turning an equation into a tree

├── main.py                     // platform for user

└── testunit.py                 // for testing whether the functions work as expected

## 1.2 Function list

|  |  |
| --- | --- |
| **Functions** | **Descriptions** |
| suffix | // Receives an infix expression, turns it into suffix expression and returns it |
| build\_tree | // Receives a suffix expression, turns it into tree structure and returns the root node |
| print\_inorder | // Using inorder traversal to visualize the tree |
| match\_bracket | // Checking whether the brackets in equation are matched |
| check\_operands | // Checking whether the number of operands in equation are correct |
| check\_valid | // Checking whether all the input symbols are valid |
| test | // Controls the three functions above simultaneously |
| save\_tree | // Save the root node of the tree into a document |
| read\_tree | // Read the root node of the tree from a document and print it out by print\_inorder() |
| main\_function | // Controls all the functions except save\_tree() and read\_tree() |

# 2. Basic design for the program

## 2.1 Description of the program

This program implements the conversion of mathematical expressions into binary trees and print the trees and calculation results out. In addition, it can save the trees to the specified path and read the trees out and visualize it next time you run the program.

## 2.2 Comment about the implementation of all the six tasks

**2.2.1 Task 1**

The first task is relatively easy, just receive an input mathematical expression, if it is valid, print the calculation result out.

**2.2.2 Task 2**

Task two is much harder than task one, I have tried a variety of ways to turn the expression into binary tree. Fortunately, I remember that I have learnt an algorithm called Inverse Polish algorithm which can turn an infix expression into suffix expression, and this inspired me a lot. I first turn the expression into suffix expression and then traverse the list of expression. Once I found an operator, I pop it out and turn it into a root node of tree and insert it back to the same place in the list. Then I pop out the two elements before it to be the left child and right child of root node respectively. Loop this process until the list has only one element, and this element is exactly the root node of the whole tree. According to the task, I use inorder traversal to visualize the tree recursively (Figure 1). Actually, I have also written another way to print the tree, although it still has some tiny bugs, it can print most of the trees in a more beautiful way (Figure 2).

文本

描述已自动生成

(Figure 1)

图形用户界面, 文本

描述已自动生成

(Figure 2)

**2.2.3 Task 3**

As for task three, I used a built-in module in python called pickle, which is what we have learnt in class. I originally decided to save the whole tree like a string or a list, but later I found maybe saving the root node of the tree could be a wiser choice. Thus, I have written two functions, one is for saving the tree through a specified path into a .pkl document and the other one is for reading the tree out and visualize it (Figure 3).

电脑屏幕的截图

描述已自动生成

(Figure 3)

**2.2.4 Task 4**

Task four is to report the invalid expression and give out the reason. First, I use a stack structure to detect whether the brackets are mismatched. Then I wrote a function. This function traverse the expression, push all operators and “(“ into a stack, every time it found a “)”, it pop out the top element until “(“. In this popping cycle, I create a temporary list, if the element is operator, it appends it into the list. At the end of this process, check whether the length of the list, if its length is zero, it returns “operator missing”, and if its length is greater than one, it will return “wrong number of operands”. After looping this process, it will check whether the stack is empty and whether all elements have been traversed, if not, it will return “brackets mismatched". The following chart (Figure 4) is an example of how the process works.

表格

低可信度描述已自动生成

(Figure 4)

I have also written some other reports of invalid expressions. For instance, ”invalid symbol was entered”, “one number missing”, “two numbers missing” (Figure 5, Figure 6).

图形用户界面, 文本

描述已自动生成

(Figure 5)

文本

描述已自动生成

(Figure 6)

**2.2.5 Task 5**

The fifth task is to write some unit tests for the code. By using the in-built module “unittest”, I tested whether my functions using for handling the exceptions are working exactly as expected. The detail of how it works is in Chapter 3.3

**2.2.6 Task 6**

The last one is task six, it has four requirements. To meet those requirements, firstly, I use the Camel Case and Under Score Case to name the class and function respectively. For example the “BiTreeNode” class and the “TestException” class, the “build\_tree()” function. In addition, I have written a lot of comments in my program, which makes the code more readable. All the code in my program has its own meaning, they are maintainable and understandable. I have also written a README.md in my document (Figure 7).

表格

中度可信度描述已自动生成

(Figure 7)

# 3. Description of the algorithmic choices made for the application

## 3.1 Justification of selecting Data Structures for the program

**3.1.1 Stack**

Stack is a linear data structure which follows a particular order in which the operations are performed, so this data structure makes every step of operation on it clearer

Since it has such good properties, I used Stack Structure to implement the Inverse Polish Algorithm, to collect all the operators and brackets (Figure 8). I also use Stack when checking whether the brackets of the input expression are matched, this structure helped me a lot.

文本

描述已自动生成

(Figure 8)

**3.1.2 Binary Tree**

This Data Structure is required in tasks, and it was used to store the input equations (Figure 9). It can show the relation between each node and make the whole structure clearer and more understandable.

文本

描述已自动生成

(Figure 9)

## 3.2 Description and justification of the Classes used for the program

**3.2.1 BITreeNode**

Just as the description above, it was a data structure I wrote for storing equations.

**3.2.2 TestException & TestExpression**

This is a class use for unit test. It contains numbers of functions using to test whether my program is working as expected.

## 3.3 Snippets of unit tests, and description and comments of the tests output

The unit tests can be roughly divided into two parts. TestException (Figure 10) and TestExpression (Figure 11). One is mainly for testing the functions in Task 4 and the other one is for Task 2.

文本

描述已自动生成

(Figure 10)

文本

描述已自动生成

(Figure 11)

I first wrote some correct equation and called the functions that I want to test, I asserted that what they return should be equal to True. Next, I wrote some equation with mistakes and find those reasons why they are invalid, then I asserted the functions with those equations will return those mistakes. I have also written an equation and its suffix expression by myself, then I called the suffix() function and asserted its return is equal to the suffix expression that I just wrote.

**4. Reference list**

1. <http://c.biancheng.net/view/5736.html>

I have learnt some knowledges about the in-built module pickle through this website. For example the usages of dump() and load().

2. <https://blog.csdn.net/unbelieveme/article/details/122798791>

This article written by author “落花的世界” has inspired me the idea and method to print the tree in another way. That is my function “print\_tree()”, though this is not required in tasks.