COMP202: Exercise 2 - Binary Trees

Odysseas Stavrou 2018030199
May 2020 – Technical University of Crete – ECE

Used Software:

- Visual Studio Code with Python Extension
- Kite
- Pycharm Ultimate Edition (Student License)
- Python Sphinx

Resources:

- Study material from the lab staff
- Python Docs

Classes:

- BST_Array.py:
  - User created
- BST.py:
  - Created Using parts from the already provided Bst Code in Java and added some modifications of my own, plus some more measurements.

Files:

- Create_test_files.py:
  - Creates a specific number of files with random numbers inside, using multiple processes (10 in such case)
  - Used for creating more test files for better results
- Main.py:
  - Initializes each class, runs the test cases and exports the data (graphs, excel sheet, etc.)

Usage:

chmod +x main.py && ./main.py – On linux
python main.py – On windows

Requirements:

If you'd like to generate new data, please install the requirements (pip install -r requirements.txt). The script will check if the requirements are satisfied and prompt you if you'd like to install them. Analytically the required packages are:

- numpy
- matplotlib
- pandas
- pyopenxl

Runtime Info:

The program will launch and run tests for test_1000000 (1.000.000 elements). It will create 3 independent processes, each of them responsible to read the file, populate its specific data structure with the numbers and gather the required data. Afterwards, it exports them into a text file and if the dependencies are

satisfied it exports an excel sheet as well. Lastly if the user has specified the generation of new data (for all the files) then the program will map all the test files into 3 processes for each of the data structures one after the other. (meaning 3 processes to handle all the files for the Array based BST and after all the files have finished, 3 processes will launch for the Dynamically allocated BST, etc.) In all cases all zombie processes are collected by the parent process, either manually using **Process.join()** or from the **Pool()** Manager. Runtime duration ~ 20 sec for simple data and ~ 60 – 70 sec for full data.

Design Flow:

For each part of the exercise I split the implementation to smaller more feasible tasks so I could keep a clear mind and work more productively. Firstly, I implemented the Array Based BST and checked its functionality. After that I modified the specific functions to return me useful information utilizing the awesome feature of python to return multiple Types and Values from a single function. The same logic was applied for the Dynamically Allocated BST and sorted Array.

Python's GIL (Global Interpreter Lock), which allows you to run only a single thread at a time, forced me to switch my design model from multithreading to multiprocessing, because there is no real gain from running 2 threads one after the other and running 2 functions one after the other, it's basically the same, no Parallel Execution. Because of the Multiple Processes It means no shared data between the Processes (unless with the use of Queues) so I had to test every data structure implementation discretely. I Pass the file into the function and the function generates the results. This is the only thing I have done that goes against the "rules" of the exercise. For the sorted array, I should have traversed the tree in-order and fill the array as such. But because of the multiple processes I would have needed to synchronise the parent and child process to transfer that array between them, which I do not have the required knowledge or even worse, use a 2nd Queue.

The main.py program is quite long, but the length of the program is mostly formatting the output and the plot functions for the graphs.

I have a lot of duplicate code in my program mainly because I like to do things discretely and don't like to pass Objects around. However, this is probably bad thinking and makes for more unnecessary code. For Example, I could write a function that takes as a parameter the object to run tests on rather that writing 3 different functions with each of them having their own object.

Documentation:

All documentation was generated using Sphinx, open index.html inside docs directory