Tests for C++ Assignment #2

Pair B3408
Renault Benoit, Espeute Clément

1 Early Visual Tests

During the prototyping phase of the BinaryTree, we quickly realised that we needed to have a tool that allowed us to display a graphical representation of the BinaryTree. More precisely, we needed that so we could easily see if the tree was balanced with a quick glance. Because of our lack of knowledge of the C++ libraries, we chose to create an external tool with a framework we already knew how to use.

We created a Lua application that used the LÖVE2d framework. It's normally intended as a set of libraries for 2d videogame making, so it had all the tools we needed to display graphical info on the screen.

In order to pass the information from the application to the utility we created, we made a serializing method for the BinaryTree to turn it into a Lua table (an associative array capable of storing many types of data, including other tables). The serialization was done recursively on the BinaryTree's Nodes, and can be found in the `Node::Serialize()` method.

Once the data was serialized, it was just a matter of loading that data into the Lua program and display it on the screen. The displayed data shows ID of the Node's sensor, and the height of the Node subtree (to see if the Node height was correctly computed).

Using the tool

If you wish to try the tool out, you'll need to install the `love` package on a Gnu/Linux machine, and then run the following command : `love utils/dispTree/` from the project root. The data needed to display the tree should already be in the right file, but if you want to export that data again, all you have to do is add the `#define DEBUG` and `#define SERIALIZE` in the Main.cpp, and then run : `make runSave` in the source folder. That should export all the needed data in the `utils/dispTree/data.lua` file. If you don't want to install love, there is a screenshot of the program in the `utils/dispTree/` folder.

2 <TestEngine> Class

The <TestEngine> class contains static functions meant to test the different core classes of the application. These were used really often during the development to ensure modifications to the data structures did not cause any unforeseen behaviour.

Especially, because the visualisation tool wasn't meant to find small imperfections in the tree, we had to implement some automatics tests in order to see if the tree's balance met all the requirements of a self-balancing binary search tree (abb. "AVL tree"). The first two tests below are designed to find those small flaws.

You may want to get more specific error messages. If so please set (`#define DEFAULT_VERBOSE`) to true.

To execute these run in the sources folder after updating the Main.h: `make run`

2.0.1 IterationTest_1

Checks if the Iteration method visited every single Node in the tree exactly once. A boolean array keeps in memory which Nodes have been visited or not and the binary tree is filled with empty Sensors (that

is to say, with all Stats array elements equal to 0). As we iterate through the tree, the boolean array is updated. The function returns a true boolean value if all the Node where visited once (if the entire boolean array has its elements set to true). If an error is found, a message will appear in the console either warning that a Node hasn't been visited at the end, or if a Node has been visited twice.

2.0.2 BalanceTest_1

Checks if the BinaryTree is balanced after a long series of random insertions. It creates and fills a BinaryTree with random sensors ID, and then iterates through all the Nodes, checking the balance of every Node and prompting an error if a balance of 2 or more is found (indicating that we don't fill the AVL tree requirements). Again, the function returns true if the tests are successful, and false otherwise.

2.0.3 SearchTest_1

Almost a copy of the Iteration Test. It just creates the Nodes in the tree and checks if the Search function of the Node class works with all the IDs. Returns true if the test is successful and false otherwise.

2.0.4 SensorTest_1

This test checks if the basic functionnalities of Sensor are working. Simply put, it creates a single Sensor object, adds events to it, and displays statistics about the sensor. It returns a boolean value of true if the test is successful, false else.

2.0.5 PerformanceSearch and PerformanceInsert

These two functions are almost identical. They sole purpose is to test the performance of the BinaryTree. For that, each function initializes the tree with 1500 Sensors, then performs random Search OR Insert function calls on the tree (depending of which function is called). They are used in conjonction with the macro `MESURE_TIME(func,name)`

2.0.6 `MESURE_TIME` (in Utils.h)

Useful macro that allows to measure the time that a function *func* takes to execute, and display it in seconds. Used for some performance tests. *name* should be a String containing the function name that is to be displayed in the output message.

2.1 External tests

Those tests were created in order to see if our application was properly functioning according to the inputs/outputs described in the assignement. We used the `test.sh` file provided with some in and out files also provided in order to see if the basic functions were properly working (that is, passing them into the program, and automatically checking the results it provided). We also used a 20 000 000 instructions long file provided by another group that allowed us to see if our program was performing well under the worst case scenario for the TP.

We also wrote two other pairs of i/o files, the first being viciousTests, which tests incorrect function calls like asking for statistics with an empty array to see if the program doesn't crash in front of invalid user commands.

Finally, we needed to test the optional instruction `OPT`, for which no test files were provided. For that, we wrote a simple lua script : (`generateValues.lua` in the `tests/` folder) that generated the testOTP.in and testOPT.out files for this test. The program chooses a random minute between 2 given hours as the "optimal" path. It then writes ADD commands filling all the minutes between the 2 hours with "R" traffic except for the chosen minute where the traffic is "V" into the .in file. Multiples Sensors are created in the same way, but with the "V" traffic put so the chosen path will be fully composed of "V" traffic states. Finally, the script writes in the .out file the expected output.

To test these run in the sources folder :

```
make && cd ../tests && ./test.sh ../sources/out/executable.out
```