**COMP0005 Group Coursework | Academic Year 2024-2025 | Group Number: 14**

# Overview of Experimental Framework

## Framework Design/Architecture

The framework provides several data generators, all which use a common interface and can be readily interchanged in the tests.

The generators are pseudorandom, i.e. they are seeded and can be used deterministically to collect repeats for the same generated data. This aids in identifying performance spikes related to random background overheads, and implementation overheads (which occur with every repeat of the same data).

They have also been designed with a high level of parameterisation, which allows a great range of potential when generating.

The most basic functioning generator is the Uniform generator, which allows for the parameters for the range of string lengths generated, seed to use, and alphabet to generate strings with.

The Uniform generator just provides the generated strings in the order they are generated, however, there are further variations which allow for sorted (ascending and descending), sloped (ascending to a peak then descending, and vice-versa), and diverging (data which alternates between increasing and decreasing). The orders are relative to sorting the strings, and as such how the strings would be inserted as keys into the tree types. The sorted generators have an initial overhead when generating the first string, which is later considered by the experimental methods. Additionally, the different generator types produce the same sets of strings for the same seeds provided to them, just ordered as described.

The ExperimentalFramework class provides a suite of static experimental methods. These include timing the nth insertion, cumulative timing for inserting n items (as it proves difficult to reliably filter outliers from the extremely sensitive tens of microseconds it takes to plot a single insertion’s time operation), and the time taken to search for a key after the nth key has been inserted. These experimentation methods provide the ability to exchange the set of trees being tested, the data generators to be used for data to insert (and when necessary, search for), the number of times to repeat each experiment for each structure, and the number of keys to be tested over. Within these tests, due to the nature of the cumulative timer recording the time to iterate a for loop, the data generator for the provided generator is initialised before the for loop, which removes any overheads found in the sorted generators from being considered, but still allowing the full range of data to be tested.

Additionally, there is a static method provided by the framework class which plots the data. All that is required for this is passing the data directly from the experimentation methods into the plotting method, along with some formatting parameters for the type of test and the trees used for labelling and titling the graphs automatically. There has also been the option to ‘prune’ outlying data, with respect to the single point’s repeats, so as not to remove outliers which represent constantly occurring spikes due to the algorithmic implementation overheads (which is the point of the experiments), but prunes the spikes due to performance overheads of the device being ran on.

Extensive docstrings are provided in the submitted code with further explanations and details about parameters and methods, along with ample use code to demonstrate the features.

## Hardware/Software Setup for Experimentation Device

# Performance Results

It is important to note some restrictions in the variety of results we are able to collect and present in such a small space. Justifications will now be described on why certain variety has not been shown, and how it is accounted for.

Different seeds: the graphs produced are all using the same insertion generator seed of 666, when testing whether varying seeds had effect on the relative and comparative data, little to no difference was seen in the several random seeds compared. Hence, this is a factor which has been omitted from the experiment.

Duplicate strings: while there is a chance of generating duplicate strings, the string length and alphabet sizes have chosen such that duplicates are extremely unlikely, and would have little effect on the data.

Parameters for string generations: the main property to be tested is the string’s ordering, as that is what effects the insertion and search times the most, so the generators have been set to use strings which are easily ordered (higher variance in positional characters, and low duplicates).

Search string generator: to hit the worst case search times more frequently, a different generator has been used for generating the strings to search, it has been chosen such that the strings generated can be in any ordered position between the already inserted strings, with a low chance of duplicates of already inserted strings and not being concentrated at the extremes of inserted strings. This allows all branches in the trees to be search consistently and thoroughly.

## Two-Three Tree

## B Tree

## Left-Leaning Red-Black Tree

*(Section 2 should be about two / two-and-a-half pages)*

# Comparative Assessment

*(Section 3 should be about one / one-and-a-half pages)*

# Team Contributions

|  |  |  |  |
| --- | --- | --- | --- |
| Name | Portico ID | Key Contributions | Work Share |
| Ioan Thomas |  | Experimental framework, data generator, relevant report sections. | 25% |
| Sameer Kurbanov |  | LLRB BST implementation and report section. And plotting for experimental framework. | 25% |
| Ivan Vargin |  | 2-3 Tree implementation and report section. | 25% |
| Mathias Tidball Gerez |  | B-Tree implementation and report section. | 25% |