CSCI-1200 Data Structures — Spring 2019
Homework 11 — Big 'O' Notation & Performance

In this lab we will carry out a series of tests on the five fundamental data structures in the Standard Template Library we have studied this semester: `vector`, `list`, binary search tree (`set` / `map`), `priority_queue`, and hash table (`unordered_set` / `unordered_map`). We will hypothesize and evaluate the relative performance of these data structures and solidify our understanding of data structure and algorithm complexity analysis.

For each data structure you will predict and measure the runtime for four different moderately compute-intensive operations: *sorting*, *removing duplicates* (without changing the overall order), *removing duplicates* (allowed to change the overall order), and *finding the mode* (most frequently occurring element). You will perform these tests on STL `string` objects that can be read from a file or constructed from random sequences of `char`s. **Important:** Keep in mind that you should consider efficient implementations and may want to consider the strategies in Lectures 13.1-13.3 and 13.8 .

Order Notation Table

First, consider the table of order notations in `order_table.txt`. You should replace the letter "X" with the appropriate **captial** letter below for each cell. Do not change any other formatting including spacing. The Submitty gradeable will have a test case called "Table Format" that will give 1 point if we can parse your table, and 0 points otherwise. You may want to revisit this table once you have done your implementation to correct any mistakes. These will be graded on what an efficient implementation's order notation would be, even if that's more efficient than the solution you code.

"Rules" for comparison: For each operation, analyze the cost of a function that takes in two arguments: an unordered, read-only c-style array storing the input and a second c-style array where you will write the output. The function should read through the input array only once (and must do so without skipping input: i.e. it must read index 0, then index 1, etc.) and construct and use one instance of the specified STL data structure to compute the output.

The letters are:

- **A** - O(1)
- **B** - O($log\ n$)
- **C** - O(n)
- **D** - O($n\ log\ n$)
- **E** - O(n^2)
- **F** - O(n^3)
- **X** - "Not feasible/sensible"

Additionally the following rules apply:

- There are between 0-3 cells with "A"
- There are between 0-5 cells with "B"
- There are between 0-5 cells with "C"
- There are between 8-15 cells with "D"
- There are between 0-5 cells with "E"
- There are between 0-3 cells with "F"
- There are exactly 2 cells with "X"

Performance Analysis of Vectors

We have implemented the 4 operations for the `vector` datatype and included them in `performance.cpp` . The data structure, operation, source of the input, size & length of input (for randomly generated input), and output file are specified on the command line. Here are a few sample command lines:

```
a.exe vector mode in.txt out.txt
a.exe vector sort random 10000 5 out.txt
a.exe vector remove_dups_same_order random 10000 2 out.txt
```

The first example reads the file named `in.txt`, uses a vector to find the most frequently occurring value (implemented by first sorting the data), and then outputs that string (the mode) to a file named `out.txt`. The second example will generate 10,000 random strings of length 5, use a vector to sort them, and then output the result to the file named `out.txt`. Similarly, the command line option `remove_dups_same_order` will remove the duplicate elements from the input (without otherwise changing the order), while the command line option `remove_dups_any_order` will remove the duplicate elements from the input but can change the ordering.

Compile and test the provided code on a variety of tests of different sizes for the `vector` datatype for each of the 4 operations. The provided code uses the `clock()` function to measure the processing time of the computation. The resolution accuracy of the timing mechanism is system and hardware dependent and may be in seconds, milliseconds, or something else. Make sure you use large enough inputs so that your running time for the largest test is about 5 seconds (to ensure the measurement isn't just noise). The program reports the time to load, process, and save the data. Record the results in a table like this:

Sorting random 5 letter `strings` using STL `vector`

# of strings	load time (sec)	operation time (sec)	output time (sec)
10000	0.023	0.031	0.089
20000	0.043	0.067	0.172
50000	0.115	0.180	0.445
100000	0.226	0.402	0.918

As the dataset grows, does your predicted big 'O' notation match the raw performance numbers? For example, in the table above, the time to both load and output is linear, in the # of strings, that is O(n).

$$\begin{aligned} \textit{load time}(n) &= k_{load} * n \\ \textit{output time}(n) &= k_{output} * n \end{aligned}$$

We can estimate the coefficients from the collected numbers: $k_{load} \approx$ 2.2 x 10^{-6} sec and $k_{output} \approx$ 9.0 x 10^{-6} sec. The running time for sorting with the STL `vector` sorting algorithm is O($n \log_2 n$):

$$\textit{operation time}(n) = k_{operation} * n \log_2 n$$

with coefficient $k_{operation} \approx$ 2.3 x 10^{-7} sec. Of course these constants will be different on different operating systems, different compilers, and different hardware! These constants will allow us to compare data structures / algorithms with the same big 'O' notation.

Performance Analysis of Other Code

Finish the implementation of the remaining data structures by filling in the incomplete functions in `perfomance_bst.cpp`, `perfomance_hash_table.cpp`, `perfomance_linked_list.cpp`, and `perfomance_priority_queue.cpp` and debug your code before proceeding.

In the provided code base, two fixed length arrays of string objects are used to load and output the data. Thus, the load & output times should be similar for all datatypes. The data structure specified on the command line is the only additional data structure that is "allowed" in the implementation of the operation. You should carefully consider the most efficient way (minimize the running time) to use the data structure to complete the operation. *Make sure you debug the output of your implementation by comparing it to the output from the vector datatype!*

Do the same tests you did for the `vector` on the remaining data structures, keeping in mind that the input sizes may need to be different to achieve a 5 second runtime. Put your findings into a PDF file called `report.pdf` which contains both the table of timing measurements and the input sizes you used for all 4 operations and all 5 data types. Make sure to label each table clearly. You can write this up in a word/spreadsheet processor such as Microsoft Word/Excel, Pages/Sheets, or LibreOffice and then "Export to PDF" or "Save As PDF" depending on the software. If you are unsure how to do this step, we recommend you ask a friend (without sharing your document!) or ask in lab. We will not accept formats other than PDF files. You can check on Submitty to see if your report is viewable.

Submission

Submit the four implementation files, the tables with runtimes, the simple README, and the order notation table:

- `perfomance_bst.cpp`
- `perfomance_hash_table.cpp`
- `perfomance_linked_list.cpp`
- `perfomance_priority_queue.cpp`
- `report.pdf`
- `order_table.txt`
- `README.txt`