INDIAN STATISTICAL INSTITUTE

MTech(CS) I year 2022-2023

Subject: Computing Laboratory

Assignment 3

Due date: 09:00AM, December 24, 2022 Total: $20 \times 3 = 60$ marks

Q1. You are given a text file containing various details about n distinct cars, e.g., their manufacturer, model name, year of manufacture, capacity of their fuel tanks (in litres), mileage (number of kilometres that the car can travel on average per litre of fuel), price, rate of toxic emissions (mcg per litre of exhaust gas emitted). Write a program to print all the details of the cars that are within the **best** k (a given number that is much smaller than n) in terms of **BOTH** mileage and emissions.

NOTE: **Higher** mileage and **lower** emissions are regarded as "better".

Input format: Your program should take 2 command line arguments: the name of the text file, and k, respectively, as shown below.

```
$ ./a.out car-data.txt 5
```

The first line of the text file will specify n. Each of the following n lines will contain 7 space-separated fields corresponding to the information mentioned above, as shown in the following fictitious example. **You may assume that the manufacturer and model names will not contain spaces.**

```
100
# Manufacturer Model Year Capacity Mileage Price Emissions
Maruti         Alto  2010 30.1     15.3    2.2   13.8
Mahindra       Beto  2021 44.0     16.1    8.1    4.8
...
```

This line is only provided for explanatory purposes. It will not appear in the input file.

Output format: Your program should print to stdout information about all the qualifying cars. Each line should be in the same format that is used in the input file.

Q2. Write a program to compute the symmetric difference and Dice Coefficient[1] for two sets of integers. Each set will be given to you in the form of a balanced Binary Search Tree (BST). For full credit, the additional memory used by your program should be no more than logarithmic in the cardinality of the sets (but it is better to submit a working program that is memory inefficient, than a buggy/non-working program that tries to be memory efficient).

Input format: Your program should take 2 text file names as command line arguments, as shown below.

```
$ ./a.out set1.txt set2.txt
```

The files will contain the two BSTs in the "usual" format that we have been using: the first line of each file will specify the number of nodes in the corresponding BST; the remaining lines will each correspond to a node, and contain 4 fields: the index of the node, the integer stored in the node, and the indices of the left and right child nodes. You may assume that the indices will start from 0 and increase by 1.

[1]Strictly speaking, your program should print the Sørensen index; see the Wikipedia article for Sørensen–Dice coefficient.

Output format: Your program should print two lines to stdout. The first line should list the integers contained in the symmetric difference of `set1` and `set2` in ascending order; the integers should be space separated. The second line should contain the Dice Coefficient printed to 4 decimal places.

Q3. Your task in this problem is to implement and measure the performance of a *Bloom filter*, a data structure used for inexact searching in sets. Suppose S is a set stored using a Bloom filter. In response to a search for an element x, the Bloom filter returns one of two answers: "x is not in S, and I am sure of that", or "I think x is in S, but I could be mistaken" (this is called a *false positive* error when x is not actually in S). At the end of this question, there is an example from `https://en.wikipedia.org/wiki/Bloom_filter` of a situation where inexact searching using Bloom filters is useful.

How a Bloom filter works[2]. An empty Bloom filter is a bit array A of m bits, all set to 0. There are also k different hash functions, say h_1, h_2, ..., h_k, each of which maps an element of S to one of the m array positions in a uniformly random manner.

To add an element x to the Bloom filter, the bits $A[h_1(x)], A[h_2(x)], \ldots, A[h_k(x)]$ are each set to 1. To query for an element, say y, i.e., to test whether $y \in S$, we check $A[h_1(y)], A[h_2(y)], \ldots, A[h_k(y)]$. If any of these bits is 0, then $y \notin S$ (if $y \in S$, then all the bits would have been set to 1 when y was inserted). If all are 1, then the Bloom filter returns "y may be in S". If these bits were set by chance to 1 during the insertion of other elements, this would be a false positive result. Intuitively, if m and k are large, the chances of a false positive are small; as more and more elements are inserted into the Bloom filter, the chance of a false positive increases.

Note that the space required by a Bloom filter to store a set of n items remains fixed at m, while the space required by the exact search structures such as Balanced Search Trees (BSTs) usually grow linearly. On the other hand, the probability of a false positive error for a Bloom filter grows with n, whereas it is always zero for BSTs.

Problem statement. The objective of this question is to study the false positive rate vs. space-efficiency tradeoff for Bloom filters.

(a) Given a file containing a list of non-negative integers, possibly with repetitions, insert the integers in an AVL tree. For each element, report whether it was actually inserted (print INSERTED), or if it was already contained in the tree (print DUPLICATE). Measure the time taken to process the sequence; also, compute the total number of nodes in your final balanced search tree, and thus, estimate the total storage space required (in bytes) for the tree.

(b) Repeat the above exercise with the same sequence but use the Bloom filter defined below. Note that your output would sometimes be wrong, i.e., your program would print DUPLICATE for some integers that are actually appearing for the first time. As before, measure the time taken to process the sequence; also, compute the false positive error rate per cent.

Bloom filter definition. For a fixed m and k, to insert a non-negative integer x in the Bloom filter, call `srand(x)`; then compute `rand() % m` k times to get k values. Set the bits in these k positions to 1 in the Bloom filter.

[2] `https://en.wikipedia.org/wiki/Bloom_filter`

NOTE: You may use a byte instead of a bit to store 0 and 1, but you will get full credit only if you store the 0 and 1 at the bit level.

Input format: ~~The name of the text file containing the list of integers, and the values of m and k will be provided as three command-line arguments, in that order.~~

Your program should take the following 4 command-line arguments in order: the names of two text files (say `insert.txt` and `search.txt`), each containing a list of integers, and the values of m and k.

Output format: Your program should print to stdout a sequence of INSERTED or DUPLICATE (one word per line), corresponding to the integers in `insert.txt`. It should also print to stderr the number of bytes and the time taken to process the sequence using an AVL tree, m, k, and the time taken to process the sequence using a Bloom filter. Finally, your program should print to stderr, the proportion of false positives reported by the Bloom filter when searching for the entries present in `search.txt`. In other words, if there are u entries in `search.txt` for which the Bloom filter responds with DUPLICATE, and this response is a false positive in v out of the u cases, your program should print v/u correct to 6 decimal places as the last output in stderr.

Example: A Web server can use a Bloom filter to determine whether a Web object (a page, an image, etc.) has been requested earlier. If it has been requested at least once earlier, only then it is stored in a cache.[3] If a cached object is requested again (i.e., thrice or more in all), it can be served quickly from the cache; otherwise, the object is served more slowly from the hard disk where it is stored. The reason for this strategy is that a very large proportion of Web objects are requested only once; there is no benefit to storing such objects in the cache.

One way to implement this strategy would involve storing the identifiers of the requested objects in an exact search structure, such as a balanced search tree (BST). Since Web servers service requests for a very large number of objects, such a BST would be large. In response to the question: "Has object x been requested before?", a BST would always correctly reply YES or NO.

Instead, if the identifiers were stored in a Bloom filter, the amount of space needed would be *much* less. Of course, this space-saving would come at a cost. In response to the above question, a NO would be guaranteed to be a correct answer, but a YES would sometimes be mistaken, i.e., occasionally, an object being requested for the first time would appear to have been requested earlier, and would end up being cached unnecessarily.

[3] A *cache* is a faster, but much smaller, storage space that is intended to store a frequently accessed subset of the complete data stored in a larger, but much slower, storage medium.