CS 3345

Programming Assignments #2 & #3

University of Texas at Dallas — *October 1, 2024*
Lecturer: Dr. Andrew Nemec — *Due: October 18, 2024*

Introduction.

The laws of probability indicate that if you left some monkeys in a room with some typewriters for long enough, eventually they would reproduce Shakespeare. This assignment asks you to write a program that tries to speed up the process. Instead of producing an output text in which each character is chosen independently and totally at random, the program tries to "learn" from an input text about which sequences of characters are likely to occur in English.

This assignment is intended to give you experience with implementing the Map ADT (entries are (key, value) pairs, keys must be unique, unordered) with both a binary search tree and a hash table.

Simple Case.

First, suppose you want to implement a simple version of this program that just mimics the distribution of the 26 lower-case letters of the English alphabet and the space character in the input text.

- Scan through the input text, keeping track of how many times each character occurs. During this process, you will update a `CharDistribution` object using a function called `occurs(c)` that will increment a counter for character `c` every time `c` occurs. So a `CharDistribution` object is implemented with a vector of 27 counters.
- After the distribution has been initialized in this way, you can obtain characters from the `CharDistribution` object using a function `getRandomChar()`. This function will return a character chosen randomly with the appropriate frequency. For example, if the input text has 100 characters and the letter `"a"` appears 20 times, then the probability that the function returns `"a"` should be .2 (i.e., 20%).
- Here is one way to implement `getRandomChar()`: choose a number at random between 1 and the size of the input text, such that each number has equal probability of being chosen. Then scan through the vector of counters, keeping a cumulative sum of character occurrences. When the sum exceeds the random number, return the associated character. (You may use a different method if you prefer.)
- To generate the output text for the simple case, call `getRandomChar()` the required number of times, after initializing the `CharDistribution` object on the input text.

General Case.

The simple case described above generates a series of characters in which each individual character occurs with the same frequency as it does in the input text. However, the

combinations of characters will not reflect the input text (i.e., will not be plausible English). To capture the frequency with which different character combinations occur, we need to alter this approach.

- When scanning the input text, you must create several different `CharDistribution` objects. The number of `CharDistribution` objects depends on the window size w provided as input to your program. First, suppose $w = 1$. Then we will need 27 different character distributions: the first one records the frequency with which different characters follow `"a"` in the input text; the second one records the frequency with which different characters follow `"b"` in the input text; etc.
- In general, when the window size is w, potentially 27^w different character distributions are needed: one that records the frequency with which different characters follow `"aaa...a"`; one that records the frequency with which different characters follow `"aaa...b"`; etc.
- Although 27^w is huge even for relatively small values of w, almost all of these distributions are not needed! For instance, it is extremely unlikely that `"aaa...a"` will occur in the input text, and if it does not occur, then the distribution of characters following `"aaa...a"` is not needed.
- We will keep the different distribution objects in a map data structure, where the key for each distribution object is the window that it corresponds to (e.g., `"aaa"`) and the value is the `CharDistribution` object itself.
- After scanning the input text, creating the different `CharDistribution` objects and storing them in a map, the output text is generated as follows:

```
1: the first w characters of the output are the first w characters of the input text
2: while output has not reached desired length do
3:     let the window be the last w characters that have been generated
4:     use the window as the key to search the map and obtain a character distribution
5:     call the getRandomChar function for that distribution to get the next character to be generated
6: end while
```

Map ADT.

The Map ADT usually has the following operations defined for an instance M:

- SIZE( ): Return the number of elements in M.
- EMPTY( ): Return true if M is empty and false otherwise.
- FIND(k): If M contains an element $e = (k, v)$ with a key k, then return a pointer or reference to that element, otherwise return a null pointer or reference.
- INSERT(k, v): If M does not have an element e with key equal to k, then add element e to M, otherwise replace the value of element e with v; return a pointer or reference to the inserted or modified element.

- REMOVE(k): Remove from M the element e with key equal to k; an error condition occurs if M has no such element.

You will create two different implementations of the Map ADT, each containing, at a minimum, the operations listed above. You can modify the arguments of the operations to suit your needs.

- Implement a map data structure using a binary search tree. Compare the keys lexicographically. **Implementing as a balanced binary search tree will be 20 extra points.**
- Implement a map data structure using a hash table with separate chaining. You'll have to experiment to find a good hash function and table size.

Instructions.

Implement your map data structures as described in the sections **General Case** and **Map ADT** located above.

You will turn in a zip file containing two programs: one implementing a map using a binary search tree and the other using a hash table. Each main program should do the following:

- Use the file `merchant.txt` as the input; it is Shakespeare's *The Merchant of Venice* with only lower case letters and spaces.
- Prompt the user for a positive integer, the window size, which, as described above, affects how the program calculates its output text.
- Prompt the user for a positive integer, the length, in characters, of the output text.
- Write to a text file an output text of the specified length, as described in the section **General Case**.
- One version of the program will use the binary search tree implementation for the map and a second version will use the hash table implementation for the map. The only changes allowed between the two versions of `main` is the choice of map implementation. That is, you should have proper data abstraction, where the implementation of the map is irrelevant to the user of the map.
- Use good style and make sure to comment your code well. If you discuss the problem with someone, write their name in your comments. **If you implemented a balanced binary tree, make sure to say so in a prominent location in the comments.**
- Turn in your program using binary search trees as a one source code file titled `FILIPA2.java` or `FILIPA2.cpp`, where `FI` is the first two letters of your first (given) name and `LI` is the first two letters of your last (family) name, e.g., Andrew Nemec $\rightarrow$ `ANNEPA2.cpp`. Turn in the hash table program as one source code file similarly titled `FILIPA3.java` or `FILIPA3.cpp`.

You may base your code on that in the textbook, but otherwise all code should be your own. **Do not use premade data structures!**

Adapted with permission from Dr. Jennifer Welch, Texas A&M University.