CSDS 233 Assignment 4 Due: Nov 6, 2020 at 11:59pm

Special office hours: Monday, November 2, 7:00pm - 9:00pm through Zoom.
(Meeting ID: 947 5986 4631, passcode: CSDS 233)

Guidelines:

- This assignment contains 2 components, written and programming. The total grade is 100 points, divided equally between 2 parts.
- You can write/type your solutions for written exercises. Please submit them in a single pdf file and label it accordingly (ex: W4_CaseID_Last name). Show your work in a proper manner, as unreadable solutions will not be graded.
- For the programming exercise, generate a single zip file containing all .java files needed for the assignment and label it accordingly (ex: P4_CaseID_Last name). Feel free to include sufficient code comments to increase code readability.
- My contact information: mqp4@case.edu. Feel free to email me if you have questions about the assignment, need encouragement or a pat on your shoulder.

Written exercises (50 pts):

1. **(15 pts)** The keys 12, 18, 13, 2, 3, 23, 5 and 15 are inserted into an initially empty hash table of length 10 with hash function h(x) = x mod 10. Show the resulting hash table if the keys are inserted using:
   a. Open addressing and linear probing.
   b. Open addressing and quadratic probing.
   c. Separate chaining with linked lists.

2. **(20 pts)** Consider the following hashing scheme using double hashing with $h_1(x) = x \bmod 5$ and $h_2(x) = x \bmod 10$. Starting with an empty hash table of size 11:
   a. Show the resulting hash table after performing the following operations: add(70), add(71), add(72), add(32), add(28), add(35), add(36), add(37), add(56), add(74).
   b. Now perform the following operations: delete(72), delete(56). Explain how you search for the elements to be deleted in the hash table.
   c. In the lecture, Prof. Ayday mentioned that for open addressing with double hashing, if the table size is prime, a new item can always be inserted. Explain why this is true. *(Hint: Consider the situation where the size of the hash table is 10, and indices 0 and 5 of the table are filled. If we want to insert x to the table where $h_1(x) = 0$, $h_2(x) = 5$, i.e. we start searching for an available index from index 0 with an increment of 5, will we ever find an index to insert x?)*

3. **(15 pts)** Consider two sets of integers, $S = \{s_1, s_2, \ldots, s_m\}$ and $T = \{t_1, t_2, \ldots, t_n\}$, $m \leq n$. Describe an algorithm of O(n) runtime complexity that uses a hash table of size m to test whether S is a subset of T. You may assume the existence of a suitable hash function that satisfies the assumption of simple uniform hashing.

Programming exercise (50 pts):

Write a method called wordCount that takes a string as an input and prints out all the words encountered in that input, along with their number of occurrences. Use a **hash table** with **separate chaining** to implement the method.

Assumptions for simplicity:

- The method is not case-sensitive, meaning that “CSDS” and “csds” are the same.
- A word is defined to be a string between 2 non-alphabetical characters, which include but not limited to space, punctuations, `'\t′`, `'\n′`, hyphens, underscores, parentheses, etc.

General procedure:

- **Split** the input string into a list of strings based on non-alphabetical characters.. To do this, you can use the method `String.split(“\\P{Alpha}+”)`
- For each word, **search** if it is already in the hash table or not. If it is not, add a new entry with an initial frequency of 1. If it is, update the frequency.
- If a new entry is added, **check** if the table needs to be expanded.
- After scanning the entire list of words, loop through the hash table and **print** out the list of words and their frequencies in any order you like.

Additional instructions:

- **Use of Java built-in HashTable or any libraries is prohibited**, and your work will not be graded if you do so. You should be able to build the hash table, along with desired methods, without using any Java or third-party libraries.
- For the hash function, Java has a `hashCode` function on strings that you can use. For a string `str`, its hash code `h` would be `h = Math.abs(str.hashCode()) % tableSize`. Feel free to come up with your own hash function if you wish to.
- `tableSize` should be adaptive to the situation. If you set `tableSize` to be sufficiently large so it won't need to be expanded and rehashed, **you will be penalized.**
- Keep track of the load factor to determine when to expand and rehash your table. For example, if the load factor exceeds a predefined threshold, you can double the size of the hash table. Keep in mind that while expanding, the hash code of strings might change, so **a helper method is needed for rehashing**.
- Your method should cover most, if not all, cases, as it will be tested with various inputs.

Grading criteria:

- 20 pts: Correctly implement hashing, table expansion, and rehashing.
- 10 pts: Entry and hash table data structures are designed in a proper manner.
- 10 pts: Hash table is printed correctly.
- 10 pts: Code is bug-free and clean (high readability, proper helper methods used, no redundancy, etc.)