Library Digitalization

COL106 Assignment 4

October 2024

1 Background

The IITD library, which houses thousands of rare books, journals and magazines, is undergoing a major transformation. With the increasing demand for digital access to their resources, the library is working on a project to digitize its entire collection. This effort is not just about scanning books into digital format, but also about making the content understandable and searchable, thereby enhancing the user experience and accessibility of the library's resources.

The library has a huge dictionary of all English words. However, since it is so big, it is not very usable. As part of the library's digitization efforts, they wish to send a **compressed dictionary** along with each book to the user, which contains only the words present in that book. Also, they wish to provide relevant books to the readers by doing a **keyword search** and return relevant books containing that keyword.

2 Modelling

To accomplish both of these tasks, the library hires 4 IITD students desperate for a CV point - a 2^{nd} year rookie dev Musk, and three 4^{th} year (slightly) experienced devs Jobs, Gates and Bezos. Their job is to develop a program that can analyse the text of each book and find and count the distinct words used in it.

Each of them believes that a particular method is best. The 2^{nd} year Musk believes a simple sorting algorithm should do the trick, while the 4^{th} year Devs believe Hashing to be a superior method; however, they still disagree on the best collision handling method.

2.1 2^{nd} Year - Musk (Mergesort)

In the first method, we will sort the words using **mergesort**. After sorting, we can traverse through the sorted list, extract unique words, and return the list of distinct words.

2.2 4^{th} Years (Hashing)

In this method, we will maintain a set of words in the form of a **Hash Table**. The method to handle collisions with be different for each dev as follows:

- Jobs uses **Chaining**
- Gates uses **Linear Probing**
- Bezos uses **Double Hashing**

The hash tables will also have dynamic resizing as described in the following section.

2.3 Hash Function

For the purpose of this assignment, we will use **Polynomyal Accumulation Hash** with a simple **mod table_size** compression function. For this, we will need a parameter `z` (will be given) and `table_size` (known at the time of hashing). Then, we assign a number to each latin letter: $p(a) : 0, p(b) : 1 \cdots p(z) : 25, p(A) : 26, p(B) : 27 \cdots p(Z) : 51$. The hash function for a string $s = a_0 a_2 \cdots a_n$ is:

$$h(s) = p(a_0) + p(a_1)z + p(a_2)z^2 \cdots p(a_n)z^n$$

This will then be compressed, to get slot number $k = h(s)$ mod table_size. **Your hash function should run in time linear in the length of string**.

2.4 The Second Double Hash Function

For the step size hash function for double hashing, we will use a hash function similar to Polynomial accumulation with a parameter z (different from the first hash) and a compression function with parameter c2, where c2 will be a given prime. The total hash function that we'll use is:

$$h2(s) = c2 - ((p(a_0) + p(a_1)z + p(a_2)z^2 \cdots p(a_n)z^n) \bmod \text{c2})$$

3 Requirements

This assignment is divided in two parts:

- **Part 1:** we implement the methods related to HashTable and the `DigitalLibrary` methods to implement the required functionalities of the library
- **Part 2:** Then, we will try to optimize the space utilized by our Hash Table using Dynamic Resizing (explained in 3.2)

3.1 Part 1: Static Sized Hash Tables

3.1.1 Hash Table Classes

You are expected to make two classes `HashSet` and `HashMap`, with support for two operations - `insert()` and `find()` as described below.

- `HashSet` would be a Hash Table with entries as `key`.
- `HashMap` would be a Hash Table with entries as `(key, value)`. Note that only `key` is hashed to find the slot for the pair.

Due to the similarities in the two classes, they have been inherited from a base class `HashTable`. You are free to decide whether you want to define some logic in the base class or code all logic in the child classes themselves, however you are encouraged to make a more modular program by writing common code in the base class itself.

- `__init__(self, collision_type, params)`: Initialize the hash table with of the given `collision_type` using the given params. `params` will contain the necessary parameters for the hash function and table size. To be precise, these arguments can take values:

  - `collision_type`:
    `"chaining"`: Use chaining for collision handling and add the new entry at the end of list
    `"linear"`: Use linear probing for collision handling
    `"double"`: Use double hashing for collision handling

  - `params`: Slightly different for different `collision_type`. `z` is the parameter for polynomial accumulation hash, and `table_size` is the table size to start with.
    `"chaining"`: `params = (z, table_size)`
    `"linear"`: `params = (z, table_size)`
    `"double"`: `params = (z1, z2, c2, table_size)`

    For double hashing, `h1` will use `z1` as parameter for hash and table size as parameter for compression function (mod `table_size`), while `h2` (step size for probing) will use `z2` as parameter for hash and `c2` as parameter for compression function (mod `c2`).

  Note that `table_size` here is just the initial size, and may change later in the problem (as discussed in 3.2).

- `insert(self, x)`: Insert a new value in the table, if it does not already exist. `x` will be:

  - `x = key` for `HashSet`

  - `x = (key, value)` for `HashMap`

- `find(self, key)`: Find entry corresponding to `key` in the hash table, and return whether found:
  - Return `True` or `False` for `HashSet`
  - Return `value` or `None` for `HashMap`
- `get_slot(self, key)`: Return the slot index for the given key.
- `get_load(self)`: Return the load factor α:

$$\alpha = \frac{\text{Number of elements in the table}}{\text{Total number of slots in the table}}$$

- `__str__(self)`: Return the contents of the HashTable in a string format. This function will be used to print the table, i.e., when you write `print(ht)`, it will print the string returned by this function similar to a "`print(ht.__str__())`" call.

  The expected format is as follows:
  - For a `HashSet`, an element will just be printed as `key`, while for for a `HashMap`, an entry will be printed as `(key, value)`, eg. `(Hash, 4)`. Notice the lack of `""` in printing any string.
  - Each table slot will be separated by a |
  - For Chaining, entries in the same slot must be separated by a `;`
  - For an empty slot, print $\langle EMPTY \rangle$

  Examples: Consider a `HashMap` and `HashSet` of COL106 assignments, assuming Stack and AVL hash to the same slot
  Chaining HashMap: (Stack, 1) ; (AVL, 2) | $\langle EMPTY \rangle$ | (Heap, 3) | (Hash, 4)
  Chaining HashSet: Stack ; AVL | $\langle EMPTY \rangle$ | Heap | Hash
  Probing HashMap: (Stack, 1) | (AVL, 2) | $\langle EMPTY \rangle$ | (Heap, 3) | (Hash, 4)
  Probing HashSet: Stack | AVL | $\langle EMPTY \rangle$ | Heap | Hash

  Please pay attention to the printing format, otherwise your code may not be autograded correctly. You may ask on Piazza if there is any ambiguity.

3.1.2 DigitalLibrary Class

The `DigitalLibrary` class is our main aim of this assignment. It has two different implementation, for the 2^{nd} year Musk (`MuskLibrary`) and for the 4^{th} years (`JGBLibrary`). The `DigitalLibrary` class is an Abstract Class, only intended to define the methods to be supported by the child classes.

An instance of either of the child classes will be given multiple books, identified by their titles, and the corresponding text. You should implement the logic of each dev as described in section 2. Your implemented `HashTable` should come in handy for the 4^{th} years

Common Functions

- `distinct_words(self, book_title)`: Return the list of distinct words present in the book with the given title. **For `MuskLibrary` the words should be in lexicographically sorted order, while for `JGBLibrary` it should be in order as it appears in your Hash Table**.

- `count_distinct_words(self, book_title)`: Return the number of distinct words in the book with the given title.

- `search_keyword(self, keyword)`: Return a list of all book titles which contain the given keyword.

- `print_books(self)`: Print the book titles along with the distict words in a format similar to the `HashTable` prints.

  For example, if we had two books Hamlet and Macbeth in Jobs method (chaining):
  Hamlet: the; a | mother | son
  Macbeth: the; a | lover | death

  For Musk's method:
  Hamlet: a | mother | son | the
  Macbeth: a | death | lover | the

  Once again, **For `MuskLibrary` the words should be in lexicographically sorted order, while for `JGBLibrary` it should be in order as it appears in your Hash Table**.

Methods Different for the 2 Classes

The two child classes differ in how the books will be given to them. For `MuskLibrary`, you will be given all the books and all the texts together in `__init__` method, while for `JGBLibrary`, each book will be given one by one:

- `MuskLibrary.__init__(self, book_titles, texts)`: All the books in the library are given. `book_titles[i]` corresponds to `texts[i]`. `texts[i]` will be a list of words.

- `JGBLibrary.__init__(self, name, params)`: We are given the name of the person who's strategy is to be implemented (Jobs, Gates or Bezos) and the corresponding params that must be used to make the hash tables (refer to section 3.1 `__init__` for details).

- `JGBLibrary.add_book(self, book_title, text)`: Add a new book to the library's collection with given title and words given as a list (`text`)

3.2 Part 2: Dynamically Sized Hash Tables

One of the major reasons of using Hash Tables instead of simple buckets is space efficiency. Therefore, in this part, we will attempt to keep the space utilized by our Hash Tables to be as efficient as possible, while not sacrificing running time.

If we simply allocate a table for a big size at the start, if the number of inserts is too small in comparison to this size, we would have wasted the memory. Instead, we will start with a table with a small size, and **dynamically resize** the table whenever it is considerably full.

The **load factor(α)** of a hash table can be expressed as:

$$\alpha = \frac{\text{Number of elements in the table}}{\text{Total number of slots in the table}}$$

If the load factor exceeds 50% ($\alpha \geq 50\%$), we allocate a new table with size as a prime that is just over double the old size and rehash all existing elements into this new table (getting this new size is already implemented as `get_new_size` function). For rehashing all elements, we use the following scheme:

- **Linear Probing and Double Hashing:** Iterate the old table, one element at a time, and rehash each element into the new table
- **Chaining:** Iterate the old table, and for each slot (containing a list of elements), iterate through the elements from start to end of the list, and insert them one by one into the new table.

We then discard the old table.

For the implementation, we have created `DynamicHashSet` and `DynamicHashMap` classes that inherit from `HashSet` and `HashMap` respectively. These classes inherit all functions from your implementation in Part 1, except the `insert` function, which now rehashes if load is high. You only need to implement the `rehash` function, and the `insert` function logic has already been written.

- `rehash(self)`: You will need to implement the rehash function in class `DynamicHashSet` and `DynamicHashMap`.
- `insert` function has already been implemented for Dynamic classes, and you need not change it.
- **For rehashing, follow the scheme mentioned above.**
- **Use the `get_next_size()` function to get the new size of the resized hash table**

4 Clarifications

4.1 Regarding Hash Table

- For `HashMap`, you can assume every `key` will have a unique `value`.

- Initialization of a HashTable instance should take time **O(table_size)**.
- `insert` method should run in **amortized O(1)** time.
- `find` method should run in **O(1)** time.
- `get_slot` should run in time **O**(|**key**|), the length of key.
- Functions will only be called with a `HashSet` or `HashMap` instance, and not with a `HashTable` instance. `HashTable` class is only there for ease of coding.

4.2 Regarding DigitalLibrary

- `MuskLibrary` class should not use `HashTable` in any way.
- The expected running time for `MuskLibrary` is:
  - `__init__` with k books each with $\leq$ W words, should run in **O(kW log W)**.
  - `distinct_words` should run in time **O(D + log k)**, where D is the number of distinct words in the book with the given title and k is the number of books.
  - `count_distinct_words` should run in time **O(log k)**.
  - `search_keyword` should run in time **O(k log D)** where D is the maximum number of distinct words in any book.
  - `print_books` should run in time **O(kD)**
- The expected running time for `JGBLibrary` is:
  - `__init__` should run in time **O(table_size)**.
  - `add_book` should run in time **O(W)**, where W is the number of words in the given book.
  - `distinct_words` should run in time **O(D)**, where W is the number of words in the book with the given title and k is the number of books.
  - `count_distinct_words` should run in time **O(1)**.
  - `search_keyword` should run in time **O(k)** according to the same notation.
  - `print_books` should run in time **O(kD)**

We disregard dependancy on word length here, since we expect small enough words in books. However, due to the linear requirement of `get_slot` function, the running time for any `DigitalLibrary` function should have atmost linear dependancy in the length of words.

For any hash based function, the given time complexity requirements are expected time complexity, while for mergesort impelementation it is the worst case time complexity.

4.3 General Clarifications

- As usual, you are free to define new methods in any of the classes, however you must keep the signature of all methods mentioned in the previous section unchanged.
- You are not allowed to use the inbuilt set or dictionary, and are expected to use your own implementation instead.
- You must not import any Python library in any of the files, and must implement the functionalities from scratch.
- **Keep yourself updated with the Piazza post for Assignment 4, as there may be new clarifications that we may have missed while writing the assignment.**

5 Submission

1. You are expected to complete this assignment in the next 2 weeks during your lab hours. (Deadline : 29^{th} October 11:59 PM)
2. The starter code for this assignment is available on Moodle_New.
3. The submission on Moodle_New should contain the following files:
   (a) `dynamic_hash_table.py`
   (b) `hash_table.py`
   (c) `library.py`
   (d) `main.py` (This file is for debugging purposes only)
   (e) `prime_generator.py` (You should NOT MODIFY this file)
4. Late submission policy is available on the course website.