CS-202
Data Structures

Assignment 3 Topics: Hashing

Due Date: March 24th, 11:55 pm

Learning Outcomes

In this assignment, you will:

- Practice implementation of chaining and linear probing.
- Perform various operation on Hash tables
- Practice real-life implementation of Hash tables

Pre-Requisites

For this assignment, you will require:

- Thorough understanding of Chaining
- Thorough understanding of Linear Probing
- Recommended to use Linux Machine

Structure

This assignment has three parts. Part 1 is on Hash function and Chaining. Part 2 is on Linear probing. Part 3 is a real-world application of Hash tables.

The deadline for this assignment is strict. No extension will be provided. You can use your grace days if you are unable to finish your assignment on time.

Refer to the end of this manual for the Grading Scheme and instructions for automated testing.

Plagiarism Policy

The course policy about plagiarism is as follows:

1. Students must not share the actual program code with other students.
2. Students must be prepared to explain any program code they submit.
3. Students cannot copy code from the Internet.
4. Students must indicate any assistance they received.
5. All submissions are subject to automated plagiarism detection. Students are strongly advised that any act of plagiarism will be reported to the Disciplinary Committee.

Best of Luck! ☺

PART 1: Hash Function and Chaining

Bitwise Hash:

In this task you will be implementing a bitwise hash function as shown below. Write your implementation in **Task1_A.py**

```
Initialize bitwise_hash = 0
For every $s_i$ in str
bitwise_hash ^= (bitwise_hash << 5) + (bitwise_hash >> 2) + $s_i$
```

NOTE: $\mathbf{s_i}$ Here is the ascii value of the character.

Required Functions:

Write the implementation for the following functions as described here:

get_hashcode(value)

- Takes a string and produces a corresponding hash key based on the bitwise hash operation described earlier.

div_compression(hash_code, table_size)

- Takes a hash key and compresses it to fit within the range bounded by the size of the hash table.

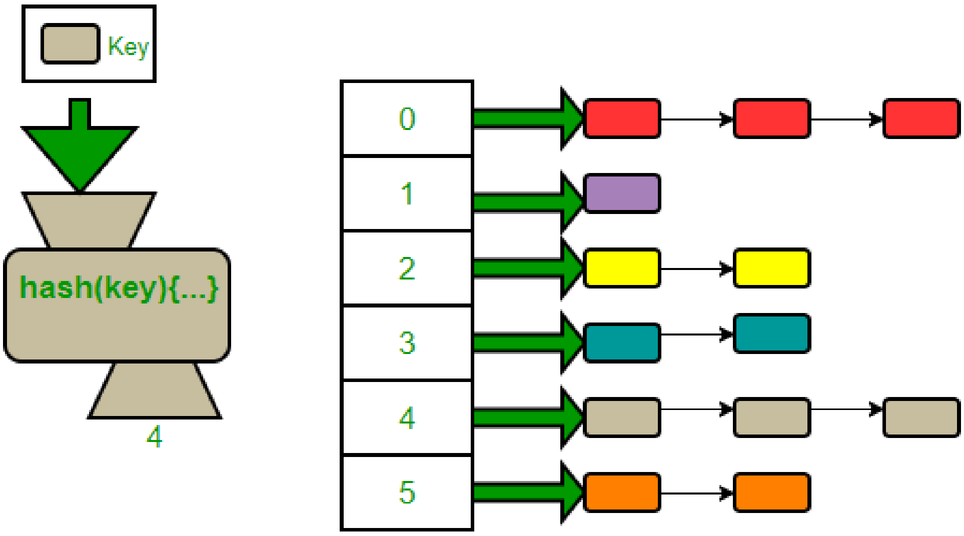
Chaining:

In this task you will be implementing a hash table of a fixed size that uses chaining to resolve any collisions. Write your implementation in **Task1_B.py**. In order to implement the chaining aspect, you will need to use the provided **LinkedList.py** implementation. Your implementations should be time efficient. The testing file checks that the total time to run all the functions does not exceed **17 seconds**.

Member Functions:

Write the implementation for the following functions as described here:

- **get_hash** (self, value)
  - Returns a hash key corresponding to the given string. Make use of the functions you wrote in **Task1_A.py** to implement this method.
- **insert_word** (self, value)
  - Inserts the given word into the hash table.
- **lookup_word** (self, value)
  - Finds the given word in the hash table and returns the value. **False** is returned if the word does not exist.
- **delete_word** (self, value)
  - Deletes the given word from the hash table if it exists.



PART 2: Linear Probing

In this task you will be implementing a hash table using open addressing with linear probing. Write your implementation in **Task2.py**. Your implementations should be time efficient. The testing file checks that the total time to run all the functions does not exceed **40 seconds**.

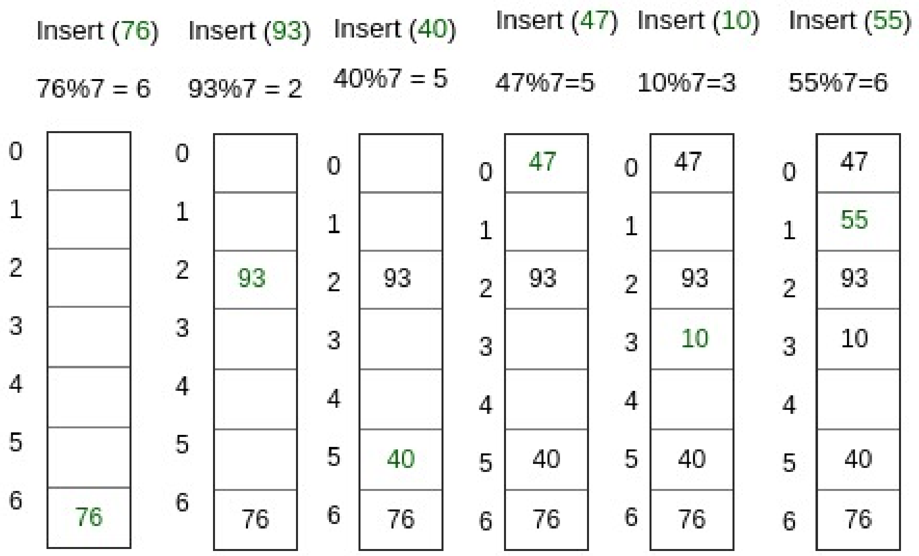
Member Functions:

Write the implementation for the following functions as described here:

NOTE: self.honeyword should be declared as None for this part.

- **get_hash** (self, value)
  - Returns a hash key corresponding to the given string. Make use of the functions you wrote in **Task1_A.py** to implement this method.
- **insert_word** (self, value)
  - Inserts the given word (NOT its hash) into the hash table.
- **resize_table** (self)
  - Resize the hash table once the load factor becomes too large. Call this function in insert_word().
  - The load factor and resize factor will influence the time it takes to pass the test in the required time so experiment with different value and choose the most suitable ones.
- **lookup_word** (self, value)
  - Finds the given word in the hash table and returns the value. **None** is returned if the word does not exist.
- **delete_word** (self, value)
  - Deletes the given word from the hash table if it exists.
  - Hint: Reset the value of key and value appropriately

Linear Probing Example



PART 3: Password Authentication System

In this task you will be making a password authentication system. Passwords are stored as their hashes to avoid attackers to hack and obtain the real passwords. Even if attackers get access to these hashed passwords, cryptographic hashes are hard to decode. However, many users use common or weak passwords. Attackers keep log of hashed output of these common passwords and other dictionary words. Then they use this to compare against the password hash they retrieved. *(yes, it's a high time to change your password if it is 123456789)*

Breaches are inevitable. A greater challenge is to detect these breaches. One way is to use *honeywords*. The key idea is to store multiple hashed passwords for each user. As usual, users create a single password and use it to login. User is unaware that additional honeywords are stored with their account

What happens after a data breach? Attacker dumps the user/password database but the attacker doesn't know which passwords are honeywords. Attacker cracks all passwords and uses them to login to accounts. If the attacker logs-in with a honeyword, the honey-server raises an alert!

NOW LET'S MAKE THIS HONEY SERVER

Our honey-server consist of a Dictionary which has the hashes of the passwords and honeywords corresponding to the username and a Hash-Table which stores which stores hash of all the passwords and honeywords along with a honeyword boolean flag.
We assume that our Hash-Table cannot be breached.

Write your implementation in **Task3.py**.

You have two variables:

self.password_table = Hash-Table which stores hash of all the passwords and honeywords.
self.honeyword is a Boolean flag which will be set to True if the word entered is a honeyword else false. (self.honeyword is declared as the variable in part 2 in TableItem class)

self.all_users = Dictionary which will store usernames as keys and list of hashes of the true password and honeywords as values.

Member Functions:

Write the implementation for the following functions as described here:

NOTE: All passwords and honeywords are unique for all the users.

store_password (self, username, password_tuple):

- password_tuple is a tuple consisting of 1 true password and 2 honeywords. The last index of tuple consists of the *index* at which the true password is stored in the tuple.
- Store the username and hashes of true password and honeywords in the self.all_users dictionary. (Don't store the index of the true password)

- Also store hashes of password and honeywords in the self.password_table. Set the value of self.honeyword to True if the word entered is honeyword and False otherwise.

find_password (self, password):

- Find the password and return a message according:
    - If the password is not found in self.password_table return the tuple (“login_failed”)
    - If the password is true password return the tuple (username, “successful_login”)
    - If the password is honeyword return the tuple (username, “hack_alert”)

      Where username is the corresponding user to the true password or honeyword entered.

update_password (self, old_password, new_password):

- Update the password and return a message according:
    - If the password is not found in self.password_table return the tuple (“login_failed”)
    - If the password is true password, replace the old password with the new password in both self.password_table and self.all_users and return the tuple (username, “successful_password_update”)
    - If the password is honeyword return the tuple (username, “hack_alert”)

      Where username is the corresponding user to the true password or honeyword entered.

Grading Scheme

Each file except the Task1_A.py has its own testing file.
To test your implementation, run this file on the terminal in the same folder as your other files.

Part 1: 30 marks
Part 2: 40 marks
Part 3: 30 marks

Note: You can create an object of the particular class within each part and use the functions that you have written in order to test your implementation. However, please make sure to comment out this portion before you run your tests as it can mix up with the test cases.

If you need any help, please do not resort to unfair means, and feel free to reach out to any of the teaching staff. All of us are here to help you learn and grow as budding computer scientists, and above all, as responsible professionals.

Best of luck! All of you are fully capable of attempting this assignment, and we are sure you will do great.