CENG310 Programming Assignment 3: Uncovering Celestial Messages using Hash Tables

Instructor: Çiğdem Avcı
Teaching Assistant: Ardan Yılmaz

Due Date: 20.05.2024, 23.55

Abstract

This assignment is to engage you in the practical application of hash tables to decode messages hidden within celestial patterns. By implementing a hashing function and collision handling strategy, you will unlock secrets of the cosmos, simulating communication with future humans.

1 Introduction

In an era where the boundaries of space and time are being redefined, we stand on the brink of uncovering messages from future civilizations encoded within the celestial texts. These messages, believed to be from future iterations of humanity or other forms of advanced life, are hidden within complex patterns of cosmic phenomena. The challenge lies not only in detecting these messages but in deciphering their content, a task that

necessitates a blend of astrophysics, cryptography, and data science. Hash tables emerge as a pivotal tool in this quest, offering a way to efficiently map, store, and retrieve these celestial codes. Their application in this context is not just a matter of convenience but of necessity, as the sheer volume and complexity of cosmic data demand a structure that can handle dynamic and rapid access requirements.

2 Objective

Our objective is to harness the power of hash tables to create a decoding system capable of translating the celestial patterns into comprehensible messages to establish a foundation for interstellar communication. The use of hash tables is motivated by their efficiency in handling large datasets and their ability to resolve conflicts—a metaphor for the challenges we face in interpreting the myriad signals that fill our universe.

3 Defining the Hash Functions

To adapt to the dynamic nature of celestial message decoding, we will be using the following hash functions. For collision handling, we will employ double hashing, wherein the secondary hash function comes in handy.

3.1 Primary Hash Function

The first hash function, designed for initial insertion, is given by:

$$\text{hash}_1(key) = \left(\sum_{i=1}^{n} \text{ord}(key[i]) \times i^2 \right) \mod (tableSize)$$

3.2 Secondary Hash Function

This is to be used for collision handling within the hash table, ie, when the determined slot by the primary hash function is filled, use the secondary one to get the index.

$$\text{hash}_2(key) = 1 + \left(\sum_{i=1}^{n} \text{ord}(key[i]) \times i \right) \mod (tableSize - 1)$$

Note: ord function returns the number representing the unicode code of a specified character.

4 Tasks

For this assignment, you will undertake a series of tasks designed to simulate the decoding of celestial messages using a hash table. The tasks will guide you through the process of building, managing, and utilizing the hash table, ensuring a comprehensive understanding of this data structure and its application in a real-world scenario.

4.1 The Input File for Hash Table Construction

The provided input file, 'CelestialMessagesInitialization.dat', contains the necessary data for initializing and managing the hash table. This **includes configurations** such as the hash **table's size** and **the threshold load factor, followed by a series of operations for inserting and deleting celestial messages.** Please note that Insert, Delete, and Mixed operations are provided in different sections to allow for partial credit. **The parser is already provided for your convenience to parse the hash table**. Below is the structure of the input file:

```
# CelestialMessagesInitialization.dat
# Hash Table Configuration
TableSize: 10
LoadFactorThreshold: 0.75

# Operations
# Format: OperationType, "Key", "Value"
# Insertion (I) and Deletion (D)
# Examples:
# I, "ConstellationName", "DecodedMessage"
# D, "ConstellationName"

# Insert
I, "Pleiades", "Welcome to the universe!"
I, "Orion", "Seek the stars."
D, "Pleiades"
I, "Andromeda", "Beyond the Milky Way."

# Delete
D, "Orion"
D, "Cassiopeia"
D, "Libra"
D, "Vega"
D, "Draco"

# Mixed
I, "Norma", "Standard of the skies."
D, "Leo"
I, "Crux", "Southern cross."
D, "Taurus"
I, "Carina", "Keel of the ship."
I, "Canopus", "Bright guide of the southern skies."
D, "Serpens"
I, "Pollux", "Bright twin in the winter sky."
D, "Hercules"
I, "Castor", "Other half of the Gemini twins."
D, "Hydra"
I, "Procyon", "Little dog following Orion."
```

4.2 Hash Table Maintenance

1. **Inserting Elements:** Insert new elements into the hash table using the primary hash function. Collisions are managed using the secondary hash function. Please refer to the **Collision Handling with Double Hashing** part for detailed explanation. Also, once the **load factor**, the ratio of the number of stored entries to the table size, **exceeds the pre-set threshold** specified in the initialization file, the **table size is doubled** to maintain efficient data retrieval and storage. This is followed by **rehashing all the remaining elements**.

2. **Deleting Elements:** Deletions in a hash table using open addressing can disrupt the integrity of the probe sequence, potentially making subsequent elements inaccessible. To address this issue, we introduce the concept of *tombstones*:

   - **Tombstones** are markers used to indicate that an element has been deleted from the hash table but the slot is still part of an active probe sequence. This mechanism allows us to maintain the integrity of the probe sequence after deletions, ensuring that all elements inserted after the deleted element remain accessible.
   - **Problem Addressed:** Without tombstones, deleting an element could leave an empty slot, potentially causing search operations to terminate prematurely when they encounter this slot. This can result in missing elements that were placed in subsequent slots due to collision resolution. Consider a hash table scenario: addition of element e_1 is followed by another element e_2 that maps to the same slot. If e_1 is removed after e_2 is added, a search for e_2 would fail because the slot where e_2 should be is now empty. Normally, the search algorithm does not continue to resolve collisions through open addressing since the slot appears available; however, it should. To address this issue, we mark the deleted spot with a placeholder: "TOMBSTONE."

3. **Collision Handling with Double Hashing:** Double hashing uses two hash functions to resolve collisions efficiently:

   (a) Compute the initial index using the primary hash function $H_1(key)$. If this slot is empty or marked with a tombstone, insert the key.

   (b) If a collision occurs (the slot is occupied), calculate a new index using $(H_1(key) + i \cdot H_2(key)) \mod table\_size$, where:

   - $H_1(key)$ is the result of the primary hash function.
   - $H_2(key)$ provides the step size from the secondary hash function.
   - i increments from 1 for each probe.
   - $table\_size$ is the hash table's capacity.

   Repeat this step until an empty slot or a slot marked with a tombstone is found for the insertion.

4. **Retrieving Data: During search, when a tombstone is encountered, the algorithm does not terminate** as it would for an empty slot. Instead, it continues to the next slot in the probe sequence until it either finds the target element or encounters an absolutely empty slot, indicating the element is not in the table.

5. **Resizing the Hash Table:** The size of the hash table is dynamically adjusted based on the load factor to maintain operational efficiency. **When the load factor exceeds the predetermined upper threshold, the table size is doubled.** Each resizing operation necessitates **rehashing** the existing items. During this process, slots previously marked as **tombstones are disregarded**, treated as if they were empty. The **rehashing respects the original order** of active entries in the hash table, ie, it rehashes the items as they are ordered within the hash table.

Message Decoding

This section outlines the process of decoding messages encoded within celestial patterns using direct hash table lookups.

1. **Input an encoded message:** a sequence of celestial patterns separated by delimiters **|**, *****, **-**.
2. For each celestial pattern in the encoded message:
   (a) Search for the pattern in the hash table to find its corresponding decoded text.
   (b) If the pattern is found, append the decoded text to the message structure being constructed.
   (c) If the pattern is not found, append the placeholder "**[UNKNOWN]**" to indicate an undecodable pattern.
3. Concatenate the separate segments to form the complete decoded message, which reflects both the effectiveness and limitations of using hash tables for data retrieval.

Testing Your Implementation

A `tester.ipynb` Jupyter Notebook is provided to assist you with both the initialization of the system and the execution of operations using the `CelestialCommunicationSystem` class. This notebook includes a series of cells corresponding to each operation you need to test within your implementation.

Running Evaluation Cells

Each cell in the notebook is tailored to test specific operations such as insertions, deletions, and a mix of both to allow for comprehensive evaluation and partial credit. The testing sequence is structured as follows:

- **Insertion Tests:** After executing a series of insertion operations, your hash table will be compared against the expected hash table specifically for these insertions, allowing you to receive partial credit based on the accuracy of these operations alone.
- **Deletion Tests:** Similarly, deletion operations are tested separately to assess how well your implementation handles the removal of elements, with its own dedicated scoring.
- **Mixed Operations Tests:** A combination of insertions and deletions is used to evaluate the robustness and accuracy of your hash table under dynamic conditions.

Decoding and Search Testing

Provided an input message, your decoded message will be tested against the expected outcome. Please note that this inherently tests the search function in your hash table implementation.

Note: The **grade** calculated by this notebook is **provisional** and is based on the operations tested within this notebook. It is unlikely to cover all possible scenarios or test cases. The **final assessment may differ**, and thus, the grade obtained through this notebook should be viewed as indicative and not final. Thus, you are encouraged to **write your own test cases in addition to those provided.**

Note: Be cautious when re-running cells as each execution updates your provisional grade based on the current state of the hash table, which may be misleading.

Note: The overall provisional grade proposed by the tester will be out of 90. The remaining 10 points will be obtained through a `README.MD` file, which should **document your operations, usage instructions, and suggestions for improving the hash functions.** Please refer to the Grading Criteria section for detailed information.

Classes Overview

This section provides an overview of the classes you will interact with or implement as part of this assignment. The provided and to-be-implemented classes form an integrated system for managing celestial messages using a hash table.

CelestialCommunicationSystem

The `CelestialCommunicationSystem` class acts as the **central processor for the entire message decoding operation**, responsible for initializing and managing operations within the `CelestialHash` hash table. It utilizes the `Parser` class to read and interpret initialization data from an input file, initializing the hash table with parameters such as `tableSize` and `loadFactorThreshold`.

Methods of CelestialCommunicationSystem

- `__init__(self, input_file)`: Initializes the system by reading metadata from the specified input file and preparing operations lists.
- `handle_Insert(self)`: Processes insert operations by adding messages to the hash table with respective keys.
- `handle_Delete(self)`: Manages delete operations by removing specified keys from the hash table.
- `handle_Mixed(self)`: Executes mixed operations that involve either inserting or deleting messages based on the type of each operation.
- `handle_Search(self, key)`: Searches for and retrieves messages from the hash table using the specified key.

- `getHashTable(self)`: Serializes the state of the hash table for inspection or storage.
- `handle_Decode(self, encodedMessage)`: Decodes an encoded message using the `CelestialMessageDecoder`.

This class orchestrates the parsing of input data, manages hash table operations (insertions, deletions, and mixed operations), and coordinates message decoding through the `CelestialMessageDecoder`. This setup ensures that each component functions seamlessly within the system, demonstrating how student implementations will be utilized in the broader context of the entire system. This class is **already provided** in the homework text for your convenience.

Parser

The `Parser` class, which is already provided, is tailored to read the initialization and operations data from the specified file. This class is responsible for system initialization and hash table creation within the `CelestialCommunicationSystem`.

CelestialHash

The `CelestialHash` class is responsible for managing the hash table's core operations. These operations include inserting new elements, deleting elements, and retrieving stored messages based on given celestial patterns. Additionally, this class dynamically resizes the hash table based on its load factor.

- **To Implement:**
  - `insert(self, key, message)`: Insert elements using the primary hash function, handle collisions with the secondary hash function, and resize the table when needed.
  - `delete(self, key)`: Remove elements and mark their locations with 'TOMBSTONE' to maintain the integrity of probing sequences.
  - `search(self, key)`: Retrieve messages by correctly navigating through valid entries and tombstones.

CelestialMessageDecoder

After the hash table is constructed and populated with celestial patterns and their corresponding messages, the `CelestialMessageDecoder` class is responsible for decoding encrypted messages. It achieves this by looking up each pattern in the `CelestialHash` table and assembling the decoded message.

- **To Implement:**
  - `decode_message(self, encodedMessage)`: Implement the logic to decode messages according to the specified algorithm, ensuring accurate retrieval and assembly of the decoded text.

Grading Criteria

The grading for this assignment totals 100 points, distributed as follows:

1. **Building the Hash Table (75 points)** Each operation is tested separately to enable partial credit.
    - **Insertions:** 25 points
    - **Deletions:** 25 points
    - **Mixed Operations:** 25 points
2. **Decoding the Message (15 points)**: Correctly decoding messages tests your hash table's search functionality.
3. **README.MD File (10 points)**: Provide explanations of each class and **discuss possible improvements to the hash functions**.

Submission

Submissions are through ODTUCLASS. Please package your files into a zip archive with the naming format `<student_id>.zip` including the following files:

- `CelestialCommunicationSystem.py`
- `CelestialHash.py`
- `CelestialMessageDecoder.py`
- `Parser.py`
- `README.MD`

Ensure that all files are correctly named and contain the necessary implementations as specified in the assignment guidelines. The README file should provide clear and concise documentation of your project. **Ensure your implementation works correctly with the tester.ipynb shared.**

Plagiarism Policy

For any questions or discussions, if they do not contain specific code snippets etc., please use the discussion thread on `ODTUClass` so that everyone can benefit and be kept up-to-date. In such cases where you need to share specific code/pseudo code (anything not abstract regarding your solution), please do not hesitate to reach out via e-mail. You are encouraged to answer each other's questions and discuss among yourselves provided you do not share any code/pseudocode etc. Such actions of sharing are a direct violation of academic integrity and will be subject to disciplinary action. This also holds for online/AI sources, including Google, ChatGPT, and Copilot. **Make use of them responsibly**- refrain from generating the code you are asked to implement. Remember that we also have have access to these tools, making it easier to detect such cases. **In short, please do not resort to practices violating academic integrity - we are here to help.**

5 ACKNOWLEDGEMENTS

- The overall story of `Uncovering Celestial Messages` was created using `GPT-4`.
- The visual on the cover page is created using `DALL·E 3`.