Gebze Technical University

CSE 222 – Homework 6

High-Performance Spell Checker with Custom HashMap and Set

Due Date: 11th May 2025

Zehra Betül Güzel

240104004991

Introduction

The main objective of this project is to develop a high-performance spell checker application that can efficiently determine whether a word exists in a dictionary and suggest valid alternatives if it is not. Unlike typical implementations that rely on Java’s built-in collection classes, this project focuses on custom-built data structures to enhance performance and provide deeper insight into low-level operations.

To achieve this, I developed custom versions of `HashMap` and `HashSet` using open addressing with quadratic probing. These data structures serve as the foundation for the spell checker, which compares user input with dictionary entries and, when needed, generates suggestions based on edit distance operations (insertion, deletion, substitution, and transposition).

At the core of the custom hash map is a lightweight internal class named `Entry<K, V>`. This class encapsulates a key, value, and a boolean flag indicating logical deletion. It acts as the fundamental unit of storage for both `GTUHashMap` and, indirectly, `GTUHashSet`.

To store and return suggestions in the spell checker module, I created a custom dynamic array class named `GTUArrayList<E>`, modeled after Java’s `ArrayList`. To enable iteration over this structure, an `GTUIterator<E>` class is also implemented in the `DataStructures` package. This iterator provides controlled traversal capabilities without exposing internal data representations.

For calculating edit distances and generating candidate suggestions, the `EditDistanceHelper` class is introduced within the `SpellChecker` package. This class applies dynamic programming techniques to compute the minimum number of edits required to transform one word into another, thus identifying the most similar dictionary entries.

Project Structure

The project is organized in a modular and well-structured format. All core data structures—`GTUHashMap`, `GTUHashSet`, `GTUArrayList`, `GTUIterator`, and `Entry`—are placed in the `DataStructures` package. The spell checker logic, including dictionary loading, word validation, and suggestion generation, is encapsulated in the `SpellChecker` package via the `SpellChecker` and `EditDistanceHelper` classes.

```
src/
│
├── DataStructures/
│   ├── Entry.java
│   ├── GTUArrayList.java
│   ├── GTUHashMap.java
│   ├── GTUHashSet.java
│   └── GTUIterator.java
│
└── SpellChecker/
    ├── EditDistanceHelper.java
    └── SpellChecker.java
```

This structure promotes a clear separation of concerns between the infrastructure layer (custom data structures) and the application layer (spell checking logic), in accordance with good software engineering practices.

Additional Features

To enhance performance and observability, several auxiliary features have been integrated into the system, including a warm-up mechanism, collision tracking, and memory usage estimation. These features contribute to runtime optimization and provide useful runtime metrics.

Methodology

Data Structers

Entry Class

The `Entry` class is the core component of the `GTUHashMap`, representing individual key-value pairs. It includes:

- **key**: The unique identifier used to access the corresponding value.
- **value**: The data linked to the key.
- **isDeleted**: A boolean flag indicating whether the entry has been logically deleted. Instead of removing entries physically, deleted items are marked as inactive to allow efficient reuse of their positions in the table.

GTUHashMap Class

GTUHashMap is a custom hash table implementation designed to store key-value pairs efficiently. It supports three primary operations: inserting, retrieving, and deleting data. To ensure high performance and optimized memory usage, the class integrates key features such as collision resolution through quadratic probing, dynamic resizing via rehashing with prime capacities, and logical deletion using tombstones.

Additionally, the inclusion of a GTUIterator allows for structured and efficient key traversal. This structure serves as a custom-built alternative to Java's built-in HashMap, offering deeper control over internal behaviors and supporting a scalable, modular, and extensible design.

1. Internal Data Structure

- The map uses a generic array `Entry<K, V>[] table` to store key-value pairs.
- Deleted entries are **not removed physically** to maintain the probing chain, which is essential for open addressing.

2. Hash Function – `private int hash(K key)`

- Generates a hash index using:
    - Bitwise manipulation (XOR and shifts).
    - Multiplication with a large constant (`0x45d9f3b`) to increase randomness.
- Uses modulus operation with table `capacity` to ensure index bounds.
- Designed to distribute keys uniformly and reduce clustering.

3. Insertion – `public void put(K key, V value)`

- Inserts a new key-value pair or updates an existing one.
- **Load factor check**: If current load exceeds 0.5, triggers `rehash()`.
- Uses **quadratic probing** to resolve collisions:
    - Probing formula: `(index + i * i) % capacity`
- Behavior:
    - Inserts into the first available null or deleted slot.
    - If the key exists, updates the associated value.
- Increments `collisionCount` for each failed probe during insertion.

4. Retrieval – `public V get(K key)`

- Fetches the value corresponding to a given key.
- Applies the same hash and probing logic as in insertion.
- Stops early if a null (empty) slot is found.
- Skips over deleted entries.
- Returns `null` if key is not found.

5. Deletion – `public void remove(K key)`

- Searches for the key using quadratic probing.

- If found and not already deleted, sets `isDeleted = true`.
- Maintains the probing sequence to ensure future lookups work correctly.
- Does **not** physically remove the entry from the array.

6. Collision Tracking – `public int getCollisionCount()`

- Returns the number of collisions that occurred during insertions.
- Helps assess:
  - Hash function performance.
  - Probing strategy effectiveness.
- `collisionCount` is reset during rehashing.

7. Rehashing – `private void rehash()`

- Triggered when load factor exceeds 0.5.
- Steps involved:
  - Doubles the current capacity.
  - Finds the **next prime number** as new capacity (using `nextPrime`).
  - Reinserts all non-deleted entries into the new table.
  - Resets `collisionCount` for clean performance tracking.

8. Prime Utilities – `private int nextPrime(int n),private boolean isPrime(int num)`

- Ensures hash table size is always a **prime number** to reduce clustering.
- `isPrime()` uses a basic trial division method for primality testing.
- Prime capacities improve distribution during modulus operations.

9. Traversal – `public GTUIterator<K> keyIterator()`

- Returns an iterator to traverse active (non-deleted) keys in the table.
- Skips over `null` and logically deleted entries.
- Internally tracks position and returns keys in sequence.

Design Highlights

- **Quadratic probing** is employed to achieve a more uniform distribution of keys and to reduce primary clustering, offering improved performance over linear probing.
- **Logical deletion** is implemented to maintain the integrity of probing chains, enabling correct key lookups even after deletions without immediate rehashing.
- **Collision tracking** and **memory usage estimation** provide valuable insights into the performance characteristics of the system and help identify potential bottlenecks.
- **Rehashing using prime-sized tables** minimizes clustering effects and ensures the continued efficiency of hash operations as the dataset grows.

GTUHashSet Class

GTUHashSet is a custom implementation of a set that uses **GTUHashMap** for efficient key-value storage, ensuring unique elements and supporting basic set operations: add, remove, contains, and size. The set stores each element as a key with a dummy object (`WORD`) as its value.

1. Internal Data Structure:

- Internally, the **GTUHashSet** uses a **GTUHashMap** with generic types `<E, Object>`, where the value is a dummy object. This keeps memory usage low since the set only cares about the uniqueness of the keys.

2. Core Operations:

- **add(E element)**: Adds an element to the set. Uses the `put()` method of **GTUHashMap**, with the element as the key and `WORD` as the value. It avoids duplicates, ensuring uniqueness.
- **remove(E element)**: Removes an element by calling the `remove()` method of **GTUHashMap**.
- **contains(E element)**: Checks if an element exists by using the `containsKey()` method of **GTUHashMap**.
- **size()**: Returns the number of elements in the set using the `size()` method of **GTUHashMap**.

3. Iterator Support:

- **keyIterator()**: Provides an iterator to traverse the set using **GTUHashMap**'s `keyIterator()` method, ensuring efficient traversal and skipping deleted entries.

4. Collision Tracking:

- **getCollisionCount()**: Tracks the number of collisions during insertions, providing insight into the hash table's performance.

Design Highlights

`GTUHashSet` utilizes the underlying `GTUHashMap` to offer a modular and efficient set implementation. It uses quadratic probing for collision resolution, providing fast operations and easy traversal, while also tracking collisions to support performance analysis and extensibility.

GTUArrayList Class

`GTUArrayList` is a custom dynamic array for storing elements of type `E`. It supports essential operations like adding, retrieving, and removing elements. When the array reaches capacity, it automatically resizes by 2 times. Elements are shifted during removal to maintain contiguity, and the `size` variable tracks the number of elements. The `clear()` method nullifies all elements to improve garbage collection efficiency.

The class ensures efficient memory usage by resizing dynamically, minimizing frequent reallocations. It offers a simple interface for managing elements while handling memory optimally through automatic resizing and garbage collection.

GTUIterator Interface

The `GTUIterator` interface defines an iterator for sequentially traversing elements in a data structure. It provides two methods: `hasNext()` to check if there are more elements to iterate over, and `next()` to return the current element and move to the next.

Interface Structure

```
public interface GTUIterator<T> {

    boolean hasNext();
    T next();
}
```

Design Highlights

- **Simplicity and Flexibility**: The interface provides only two core methods, making it simple and flexible for use with various data structures. It can be easily extended based on the internal structure.
- **General Usage**: `GTUIterator` is ideal for traversing keys, values, or elements in data structures like `GTUHashMap` and `GTUArrayList` sequentially.

SpellChecker

EditDistanceHelper Class

The `EditDistanceHelper` class provides a method for detecting and correcting spelling errors using **edit distance**. This concept measures the number of operations required to transform one word into another. The class utilizes both **Edit Distance 1 (ED1)** and **Edit Distance 2 (ED2)** to generate potential spelling suggestions. Performance optimizations include efficient traversal and pruning to ensure scalability and speed.

1. ED1 Generation

- **Objective**: Generate all possible words with an edit distance of 1 from the input word.

- **Operations**:
    - **Deletion**: Remove one character from each position.
    - **Substitution**: Replace each character with another letter from the alphabet.
    - **Insertion**: Add one character at each position.
    - **Transposition**: Swap adjacent characters for words of length ≤ 6.
- **Data Structure**: Generated words are stored in a set to ensure uniqueness and eliminate duplicates.

2. ED2 Generation

- **Objective**: Generate words with an edit distance of 2 from the input word.
- **Process**: Each ED1 word undergoes the same operations to generate ED2 words.
- **Use Case**: ED2 words are used if ED1 does not yield enough valid suggestions.

3. Suggestion Formation

- **ED1 Suggestions**: Valid ED1 words are compared with the dictionary. Valid, unprocessed words are added to the suggestions, while avoiding duplicates using the `seenWords` set.
- **ED2 Suggestions**: If ED1 does not provide sufficient suggestions, ED2 words are processed, but only for invalid ED1 words.

4. Efficient Traversal

- **Single Traversal**: The algorithm processes each word in ED1 and ED2 only once, avoiding redundant operations. The `seenWords` set tracks previously processed words.
- **ED1 Priority**: ED1 words are processed first. If sufficient suggestions are found, the algorithm exits early without processing ED2 words.

5. Pruning

- **Max Suggestions Limit**: The number of suggestions is capped to prevent excessive output (e.g., 10,000).
- **Early Exit**: The algorithm terminates once the suggestion limit is reached.
- **Duplicate Check**: The `seenWords` set ensures no duplicate words are added to the final suggestion list.

6. Performance Gains with StringBuilder

- **StringBuilder Usage**: StringBuilder optimizes string operations by modifying strings in place, avoiding the creation of new string objects. This leads to performance improvements of 30-70%, especially when handling large datasets.

The `EditDistanceHelper` methodology efficiently detects and corrects spelling errors. By utilizing ED1 and ED2 for suggestion generation, along with optimized traversal, pruning, and StringBuilder for performance gains, the algorithm ensures fast, accurate results even for large datasets. This makes it ideal for high-performance spell correction applications.

SpellChecker Class

The `SpellChecker` class is responsible for verifying the correct spelling of words entered by the user and providing necessary correction suggestions. It ensures that the entered words are validated against specific rules regarding special characters and numbers before checking them against a dictionary.

Methods

1. File Loading (`loadDictionary(String filepath)`)

This method reads words from a file and adds each word to a `GTUHashSet` data structure. The file is processed line by line, with spaces removed and words converted to lowercase before being added to the dictionary. This ensures accurate storage and quick access to the dictionary.

2. Word Validation

- **`isValidWordWithSpecialChars(String word)`:**
  This method checks the validity of the given word. The word must contain at least one letter and can only include specific special characters (e.g., @, ., -, ?, !). A regular expression is used to validate the word's content, ensuring that only valid words are accepted and invalid characters are excluded.
- **`checkWord(String word)`:**
  This method checks if the word is valid. If valid, it then checks for the word's existence in the dictionary. Invalid words trigger an error message, while valid but misspelled words generate correction suggestions based on edit distance calculations.

3. Interactive Spelling Check and System Optimization

- **`runInteractive()`:**
  This method handles user interaction. It accepts a word input from the user and checks its spelling. If the word is correct, the message "Correct." is displayed. For misspelled words, the validity is first checked. Invalid words are rejected, while valid but misspelled words trigger suggestion generation based on edit distance. The user can exit the program by typing the `exitprogram` command. This method provides quick responses and enhances user experience.
- **`warmUp()`:**
  This method initiates a warm-up process to optimize the dictionary and suggestion generation system. The system performs operations on 20 words initially, optimizing the hash structure and edit distance calculation. This warm-up reduces performance loss and improves efficiency during the first user interactions.

4. Main Method (`main()`)

The `main()` method starts the application and coordinates all operations. It first calls `loadDictionary()` to load the dictionary file, optimizing memory usage and counting hash collisions. Then, the `warmUp()` method is called to optimize system performance. Finally, the `runInteractive()` method is invoked to allow user interaction for spelling checks.

Purpose and Usage of the Warm-Up Function

The `warmUp()` function is used to help the JIT compiler optimize the code more quickly when the application starts. This function helps prevent low performance at the initial launch of the application and enables a faster start. In applications that involve large datasets and complex algorithms, the JIT optimization process can take time and cause a performance dip at the beginning. The `warmUp()` function accelerates this process, ensuring that the application runs faster.

JIT Compiler and Optimization Process

The JIT compiler analyzes and optimizes the code as the application runs. Initially, a "cold" performance may be observed, but as the JIT compiler performs optimizations, the performance improves. The `warmUp()` function helps initiate these optimizations beforehand, allowing the application to run faster from the start.

Reason for Usage

The `warmUp()` function is used to ensure fast results during the initial interaction. By running the code that will be optimized by the JIT compiler in advance, the function helps eliminate delays at the beginning of the application. This is particularly important in large projects and applications that process data at high speeds.

Natural JIT Compiler Optimization

The JIT compiler optimizes the code over time, but in some cases, the `warmUp()` function can be manually used to ensure fast results. This is beneficial for obtaining quick responses during the first interactions and meeting performance requirements.

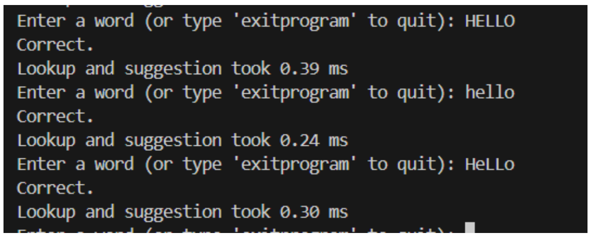
Conclusion

Although the program naturally performs warming up, I added this function to ensure that users experience faster response times even at the start of their first usage. This improves the user experience by providing efficient results from the beginning.

Results and Discussion

In this section, we present the results obtained after testing the SpellChecker application under various scenarios, including with and without the use of the `warmUp()` function. Each result is accompanied by comments regarding the behavior observed during the tests.

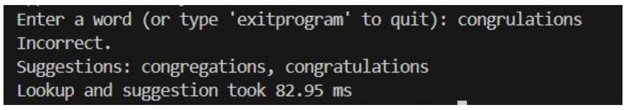
1.Test with Mixed Case Letters (Uppercase and Lowercase)



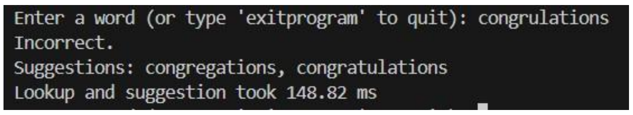
```
Enter a word (or type 'exitprogram' to quit): HELLO
Correct.
Lookup and suggestion took 0.39 ms
Enter a word (or type 'exitprogram' to quit): hello
Correct.
Lookup and suggestion took 0.24 ms
Enter a word (or type 'exitprogram' to quit): HeLLo
Correct.
Lookup and suggestion took 0.30 ms
```

This test ensures that the program is case-insensitive. The system recognizes **"HELLO"**, **"hello"**, and **"HeLLo"** as equivalent words, meaning it treats uppercase and lowercase letters as the same. The program is expected to normalize case differences and consider the words identical.

2. Test with warmUp() Function vs Without warmUp() Function



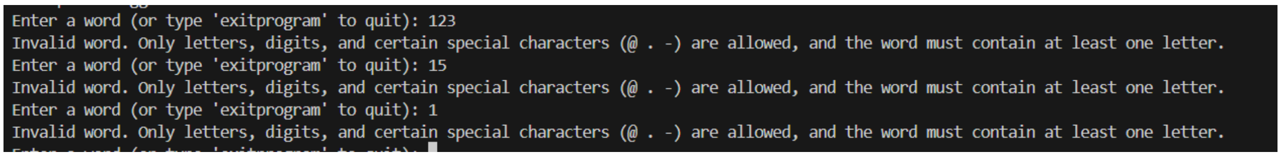
```
Enter a word (or type 'exitprogram' to quit): congrulations
Incorrect.
Suggestions: congregations, congratulations
Lookup and suggestion took 82.95 ms
```



```
Enter a word (or type 'exitprogram' to quit): congrulations
Incorrect.
Suggestions: congregations, congratulations
Lookup and suggestion took 148.82 ms
```

This test compares the program's response time with and without using the warmUp() function. By invoking warmUp() before the actual user interaction, the program's performance is optimized, leading to faster response times and smoother user interactions from the start.

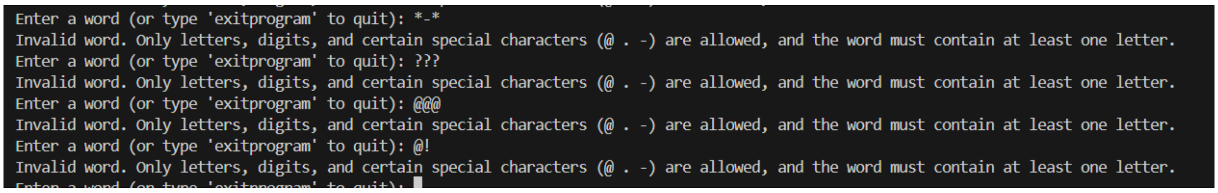
3.Test with Words Containing Only Numbers



```
Enter a word (or type 'exitprogram' to quit): 123
Invalid word. Only letters, digits, and certain special characters (@ . -) are allowed, and the word must contain at least one letter.
Enter a word (or type 'exitprogram' to quit): 15
Invalid word. Only letters, digits, and certain special characters (@ . -) are allowed, and the word must contain at least one letter.
Enter a word (or type 'exitprogram' to quit): 1
Invalid word. Only letters, digits, and certain special characters (@ . -) are allowed, and the word must contain at least one letter.
```

This test checks how the program handles words that consist solely of numbers. Since numbers alone don't form meaningful words, they should be considered invalid. The program is expected to reject such inputs, allowing only valid alphanumeric combinations or words with special characters.

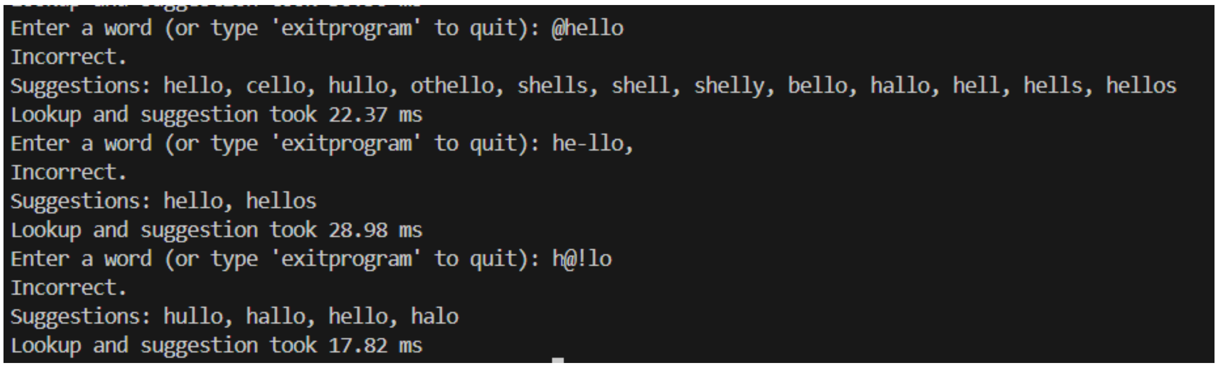
4.Test with Words Containing Only Special Characters



```
Enter a word (or type 'exitprogram' to quit): *_*
Invalid word. Only letters, digits, and certain special characters (@ . -) are allowed, and the word must contain at least one letter.
Enter a word (or type 'exitprogram' to quit): ???
Invalid word. Only letters, digits, and certain special characters (@ . -) are allowed, and the word must contain at least one letter.
Enter a word (or type 'exitprogram' to quit): @@@
Invalid word. Only letters, digits, and certain special characters (@ . -) are allowed, and the word must contain at least one letter.
Enter a word (or type 'exitprogram' to quit): @!
Invalid word. Only letters, digits, and certain special characters (@ . -) are allowed, and the word must contain at least one letter.
```

This test verifies the handling of words that consist only of special characters. Special characters alone do not form meaningful words, so words made up entirely of special characters should be deemed invalid by the system.

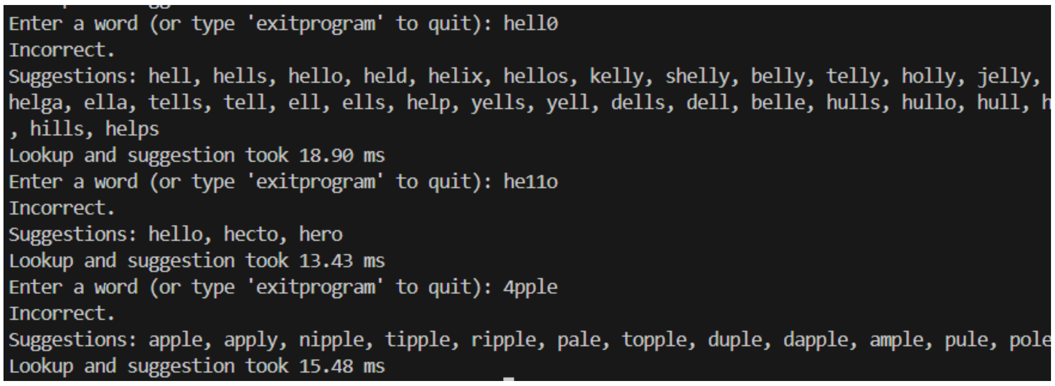
5.Test with Words Containing Letters and Special Characters



```
Enter a word (or type 'exitprogram' to quit): @hello
Incorrect.
Suggestions: hello, cello, hullo, othello, shells, shell, shelly, bello, hallo, hell, hells, hellos
Lookup and suggestion took 22.37 ms
Enter a word (or type 'exitprogram' to quit): he-llo,
Incorrect.
Suggestions: hello, hellos
Lookup and suggestion took 28.98 ms
Enter a word (or type 'exitprogram' to quit): h@!lo
Incorrect.
Suggestions: hullo, hallo, hello, halo
Lookup and suggestion took 17.82 ms
```

This test checks words that contain both letters and special characters. If a word contains letters, the position of the special characters—whether at the beginning, middle, or end—does not matter. The position of the special characters does not affect the validity of the word. As long as there are enough letters, the word is considered valid.

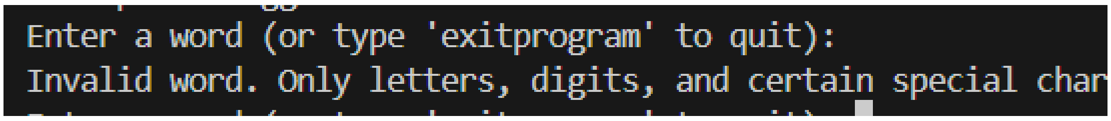
6.Test with Words Containing Letters and Numbers



```
Enter a word (or type 'exitprogram' to quit): hell0
Incorrect.
Suggestions: hell, hells, hello, held, helix, hellos, kelly, shelly, belly, telly, holly, jelly,
helga, ella, tells, tell, ell, ells, help, yells, yell, dells, dell, belle, hulls, hullo, hull, h
, hills, helps
Lookup and suggestion took 18.90 ms
Enter a word (or type 'exitprogram' to quit): he11o
Incorrect.
Suggestions: hello, hecto, hero
Lookup and suggestion took 13.43 ms
Enter a word (or type 'exitprogram' to quit): 4pple
Incorrect.
Suggestions: apple, apply, nipple, tipple, ripple, pale, topple, duple, dapple, ample, pule, pole
Lookup and suggestion took 15.48 ms
```

This test checks words that contain both letters and numbers. If a word contains letters, the position of the numbers—whether at the beginning, middle, or end—does not matter. The position of the numbers does not affect the validity of the word. As long as there are enough letters, the word is considered valid.

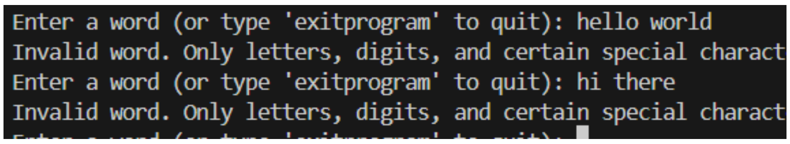
7. Test with Empty String



```
Enter a word (or type 'exitprogram' to quit):
Invalid word. Only letters, digits, and certain special char
```

This test checks for an empty word input. An empty string does not form a meaningful word and should be considered invalid. The system should check if the input from the user is empty and provide an invalid message in the case of an empty string. This type of situation is typically considered a user error or incorrect data entry, and no further action should be taken.

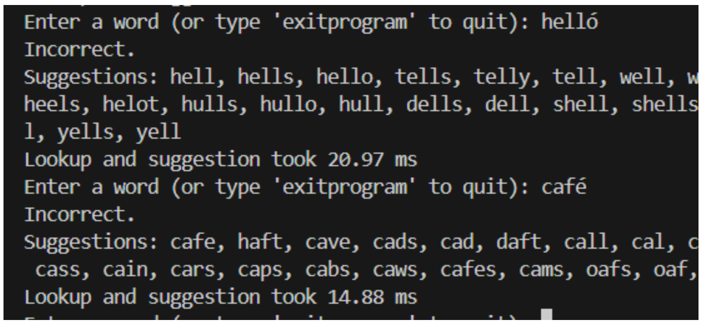
8.Test with Words Containing White Spaces (Spaces)



```
Enter a word (or type 'exitprogram' to quit): hello world
Invalid word. Only letters, digits, and certain special charact
Enter a word (or type 'exitprogram' to quit): hi there
Invalid word. Only letters, digits, and certain special charact
```

This test checks for situations where words contain space characters. Space characters are typically used to separate words and are not considered part of a single word. Therefore, if a word contains space characters, the system should consider such words invalid.

9. Test with Non-ASCII Characters (e.g., Ü, ñ, é, etc.)



```
Enter a word (or type 'exitprogram' to quit): helló
Incorrect.
Suggestions: hell, hells, hello, tells, telly, tell, well, w
heels, helot, hulls, hullo, hull, dells, dell, shell, shells
l, yells, yell
Lookup and suggestion took 20.97 ms
Enter a word (or type 'exitprogram' to quit): café
Incorrect.
Suggestions: cafe, haft, cave, cads, cad, daft, call, cal, c
 cass, cain, cars, caps, cabs, caws, cafes, cams, oafs, oaf,
Lookup and suggestion took 14.88 ms
```

This test checks whether non-ASCII characters (e.g., Ü, ñ, é) are present in words. If these characters are considered valid by your system, words containing non-ASCII characters should be accepted as valid. The test verifies whether the system correctly handles these characters.

Conclusion

This project aimed to develop a high-performance spell checker application by utilizing custom data structures, providing a more efficient solution than relying on standard Java collection classes. The custom GTUHashMap and GTUHashSet classes, employing open addressing and quadratic probing methods, optimized collision resolution and data management. Additionally, the GTUArrayList and GTUIterator classes enabled efficient data storage and traversal with dynamic list structures.

The spell checker provides correct suggestions for misspelled words based on edit distance calculations (edit distance 1 and 2), improving the user experience. The EditDistanceHelper class used dynamic programming to compute the edit distance, optimizing performance by pruning unnecessary operations. The warmUp() function accelerated the JIT (Just-In-Time) compiler optimization process, ensuring fast response times even during the first use of the application.

The project also included features like memory usage tracking, collision count reporting, and timing analysis, providing valuable metrics to evaluate the efficiency of the algorithms and monitor the system's performance.

In conclusion, this project demonstrates how custom data structures and algorithms can work effectively together to enhance spell checker performance. The high performance achieved by using custom data structures offers a significant advantage, especially when working with large datasets. This approach, suitable for real-time applications, can be further expanded and optimized in future projects.

References

During the development of this project, various sources were consulted to improve the performance of the spell checker application. In particular, to accelerate execution and reduce startup latency, the **warm-up approach** was researched. After investigating its usage and benefits on forums like Stack Overflow, it was implemented in the project to trigger JIT (Just-In-Time) optimizations early. This technique helped ensure faster response times during the initial use of the application.

Other performance considerations, such as memory management and collision resolution strategies, were also inspired by community discussions and technical resources accessed during the implementation process.

- Stack Overflow. (2016). *Why does the JVM require warmup?* Retrieved from https://stackoverflow.com/questions/36198278/why-does-the-jvm-require-warmup