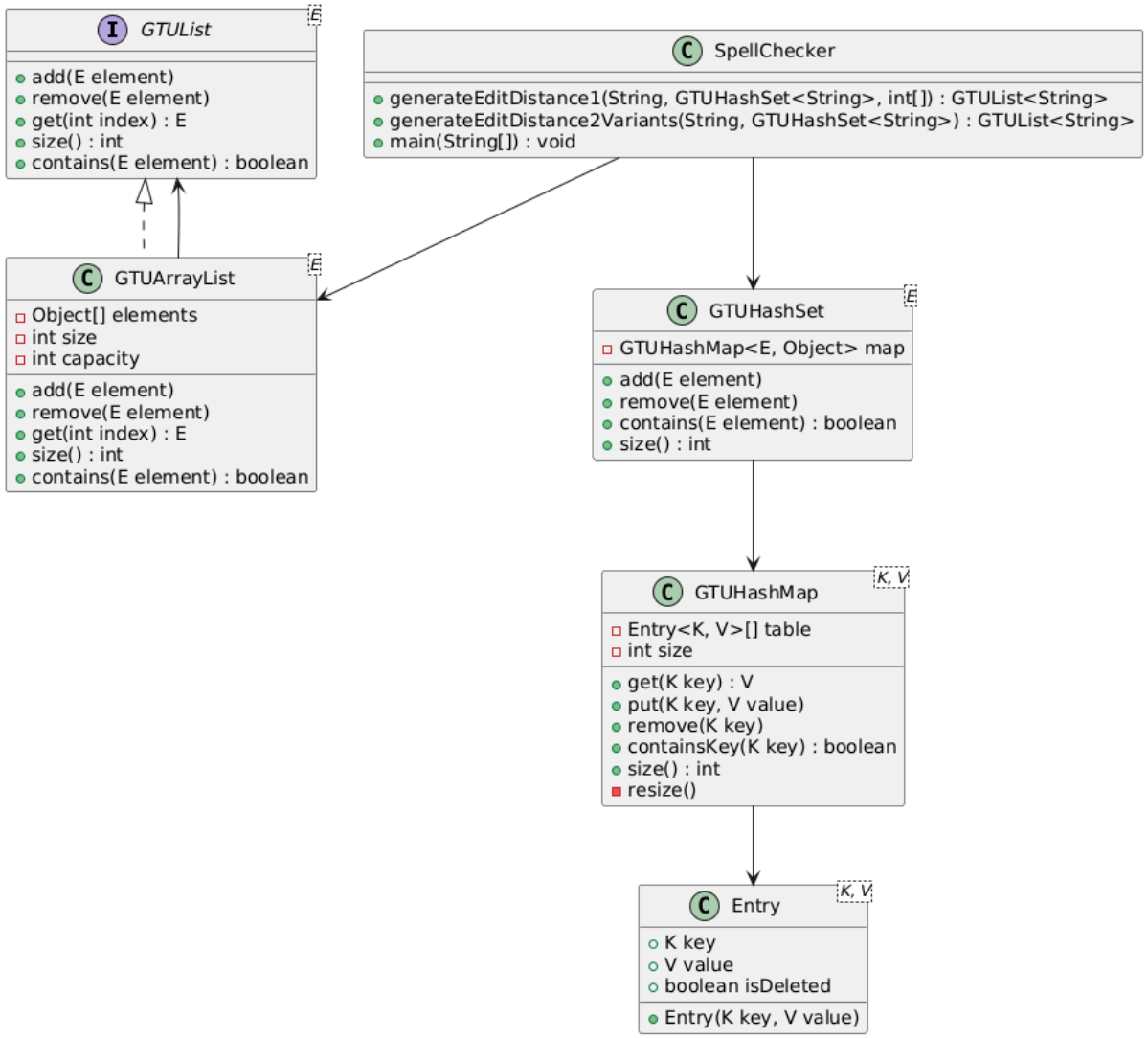
CSE 222 – Homework 6

High-Performance Spell Checker with Custom HashMap and Set Of Report

UML Diagram:



Introduction:

In this project, I developed a spell checker application capable of providing suggestions based on edit distance by implementing high-performance custom versions of `HashMap<K, V>` and `HashSet<E>`. While designing the map and set structures from scratch, I aimed to maintain high performance. To reduce collisions and improve efficiency, I chose to use quadratic probing instead of linear probing for open addressing in the hash map.

Once I completed the design of the custom map, implementing the set structure became significantly easier by internally reusing the map. Since the use of Java Collection classes was strictly prohibited, I also implemented my own `List` and `ArrayList` classes, which were utilized in the spell checker logic.

The main purpose of this project is to provide efficient spelling correction suggestions without scanning the entire dictionary, by generating all possible word variations based on edit distance (≤ 2) through character-level operations such as insertion, deletion, and substitution. These variants are checked against a preloaded dictionary implemented using the custom hash set. Additionally, since the input word must be a valid `String`, type validation was also incorporated.

Throughout this project, I aimed to stay within the principles of object-oriented programming (OOP), while also focusing on memory management and algorithmic efficiency.

METHODOLOGY

Entry<K, V> Class:

The `Entry` class is the main building block of the project. In my hash table implementation, I created a special class called `Entry<K,V>` to represent each key-value pair. This class stores the data and also keeps track of whether it has been deleted or not.

Instead of removing deleted data from the table directly, I mark it as deleted by setting the `isDeleted` flag to `true`. This method is called tombstone deletion. It is important, especially in open addressing systems, to keep searches working correctly. Without this marking, a search might stop too early and miss values that were moved due to collisions.

GTUHashMap <K, V> Class:

The `GTUHashMap` class is a specially developed open addressing hash table that stores the `Entry` objects I created. This structure supports basic operations such as adding, removing, searching for data, and dynamically increasing the table size. To perform these operations, I defined functions like `put`, `get`, `remove`, `containsKey`, and `size`. I also created a helper function called `find`, which calculates the hash of a given key and finds the correct index for it.

When a new key is added, first the `hashCode()` value of the key is taken, and then the index is calculated using the modulo operation with the table length. If the calculated index is already full, I decided to use the quadratic probing method to solve the collision. This method tries new indexes by increasing the steps quadratically, which helps reduce clustering. The `put` function follows this algorithm to either add a new element or update the value of an existing key.

The `get` function uses the `find` function to locate the position of the key in the table and returns the matching value. If the key is not found, it returns `null`. The `remove`

function does not physically delete the data but sets the `isDeleted` flag of the `Entry` object to `true`, which means logical deletion (tombstone). This allows the probing process to continue correctly during future searches and helps find values that were shifted due to collisions.

The `containsKey` function checks whether a given key exists in the table. It works by checking whether the result of the `get` function is `null` or not, and returns a boolean value. The `size` function returns the number of active (non-deleted) elements, showing the current usage of the table.

When the table becomes too full, the `rehash` function is called. In this process, the size of the table is increased. The new size is chosen as the next prime number that is bigger than twice the current size. This helps the data spread more evenly and reduces collisions. During rehashing, all non-deleted entries are placed into the new table at the correct positions. In this way, performance is improved and no data is lost.

With this structure, I saved memory and was able to perform data insertions, deletions, and searches with high performance.

GTUHashSet <K, V> Class:

The `GTUHashSet` class mainly uses the `GTUHashMap` class to store data. Since a set only needs to store keys , I assigned a constant object for each key. This helps save memory and allows the set to work without storing extra values.

All basic set operations are done by calling the functions already defined in `GTUHashMap`. For example, the `add(E element)` function calls `map.put(element,constant object)` in the background. This adds the new element to the set. Similarly, the `remove(E element)` function uses `map.remove(element)` to delete the element.

To check if an element exists in the set, the `contains(E element)` function calls `map.containsKey(element)`. Also, to get the number of elements in the set, the `size()` function works through `map.size()`.

Since the functions I wrote in the map class are also useful here, I reused them by calling them directly in the set class. Instead of writing new functions again, I used the map I had already implemented. This helped me avoid code duplication and follow the modular structure of object-oriented programming.

Edit Distance Calculation:

In this project, I developed two main functions to suggest corrections for incorrect words entered by the user: `generateEditDistance1()` and `generateEditDistance2Variants()`. These functions generate variations of a word with an edit distance of 1 or 2.

The `generateEditDistance1()` function applies three basic character operations: insertion, deletion, and replacement. At each character position, it tries all letters from 'a' to 'z'. In each operation, a new word variation is created using `StringBuilder`. If the new variation has not been generated before, it is added to the list. Also, the number of operations is tracked using a counter (`operationCount[0]`). When this counter reaches 10,000, the function stops to avoid running too many operations and slowing down the system.

The `generateEditDistance2Variants()` function works in a more comprehensive way. First, it generates first-level variations using the `generateEditDistance1()` function. Then, each of these words is passed again to the same function to create second-level variations. In this way, words with an edit distance of 2 are created. During this process, a `GTUHashSet` is used to prevent adding the same word multiple times to the suggestions list. This is important because `GTUHashSet.contains()` works in O(1) time, while `GTUArrayList.contains()` works in O(n). Thanks to this optimization, performance is improved. Also, each generated word is checked to see if it exists in the dictionary, and only valid suggestions are added.

In addition, if the number of operations exceeds the defined limit of `MAX_OPS = 10000`, the system gives a warning and exits the function early. This limitation prevents the system from freezing during testing and ensures that suggestions are generated within a reasonable time. With this structure, when the user enters a misspelled word, the system can quickly and correctly suggest the closest valid words from the dictionary.

Spell Checker Application:

The SpellChecker application is an interactive program that checks if a word entered by the user is correct, and provides suggestions if it is not. When the program starts, it reads the `dictionary.txt` file line by line and stores the words in a `GTUHashSet`.

The input from the user is first trimmed to remove spaces, and then checked to make sure it only contains letters. If the input is valid, the program checks whether the word exists in the dictionary.

If the word is not found, edit distance variations are generated and checked one by one using the dictionary. Suggestions are created based on which variations exist in the set. Additionally, the time taken for each word check is measured using `System.nanoTime()`, and the performance is evaluated.

GTUList – GTUArrayList:

In this project, I was not allowed to use Java's standard collection classes (such as `List` or `ArrayList`), so I had to design my own data structures for list operations.

First, I created an abstract interface called `GTUList<E>` and defined the basic list functions. Then, I implemented this interface with a dynamic array-based class named `GTUArrayList<E>`.

I mainly used these structures to store the suggestion list in the spell checker application.

Result And Discussion

In the end, my application worked as expected. However, I faced several challenges during the development process. At first, the performance was not good enough — the response time was over 100 ms. After reviewing my code, I noticed that some `for` loops were unnecessary. I removed them, which helped improve the performance.

Later, I thought increasing the initial capacity of my `GTUArrayList` and `GTUHashMap` could reduce the number of times rehashing and resizing are triggered and possibly improve performance. I increased those values, but I didn't observe any major improvement.

In early tests, I also noticed that some suggestion words appeared more than once. I solved this by using the `contains()` method to check if the word already existed in the list before adding it. Since `contains()` has O(n) time complexity, I thought I should use the `Set` class's `contains()` instead. So, I created a separate set object and used it to check for duplicates. This helped improve performance.

After doing these, I realized that I was checking the variations generated by `generateEditDistance1()` in the dictionary too early, which caused the system to miss some potentially correct words. So, I removed that check. But this time, the execution time increased significantly.

To fix this, I tried many things. First, I replaced all lists with sets. But since sets don't support indexing, I had to make them iterable. This part was difficult, and even though I tried, I couldn't get a better result — the execution time was still high. So I gave up on that and looked for other ways.

Then, I was using `substring` while generating words. After some research, I saw that `StringBuilder` is more efficient, so I decided to try it. Although it didn't bring the time below 100 ms, it worked faster than `substring`, which was still an improvement.

Even though the time was still high and I didn’t know what else to do, I realized that I wasn’t checking the number of operations. So I updated my code to track operation counts while generating word variations. Since the system is supposed to stop when it reaches the maximum number of operations (as stated in the report), this change helped reduce the total time.Although the system couldn’t suggest all correct words, I think it’s acceptable because it stops after hitting the defined operation limit.

Another problem I faced was a compiler warning about an "unchecked cast." I got this warning when compiling the code.

```
Note: Some input files use unchecked or unsafe operations.
Note: Recompile with -Xlint:unchecked for details.
```

After some research, I learned that Java doesn’t allow creating arrays with generic types. I fixed this problem by adding the annotation `@SuppressWarnings("unchecked")` above the method where the array was created.

Overall, I believe my application gave correct results and met the required performance criteria. Despite the difficulties I faced, I think I achieved a successful result by optimizing the code and solving problems along the way.

When a correct word is entered:

```
sumeyyedalga@Sumeyyes-MacBook-Air datahw6 % java SpellChecker
Enter a word: else
Correct.
```

```
sumeyyedalga@Sumeyyes-MacBook-Air datahw6 % java SpellChecker
Enter a word: mars
Correct.
```

When a invalid word is entered:

```
sumeyyedalga@Sumeyyes-MacBook-Air datahw6 % java SpellChecker
Enter a word: 35
Please enter a valid word.
```

```
sumeyyedalga@Sumeyyes-MacBook-Air datahw6 % java SpellChecker
Enter a word:
Please enter a valid word.
```

When a incorrect word is entered:

```
sumeyyedalga@Sumeyyes-MacBook-Air datahw6 % java SpellChecker
Enter a word: chnking
Incorrect.
Suggestions:
Maximum operations reached.
banking
bonking
bunking
conking
chinking
choking
chunking
dunking
finking
funking
honking
inking
junking
linking
nanking
pinking
ranking
sinking
shaking
tanking
thanking
thinking
Total suggestions: 22
Lookup and suggestion took 77,81 ms
```

```
sumeyyedalga@Sumeyyes-MacBook-Air datahw6 % java SpellChecker
Enter a word: deforeste
Incorrect.
Suggestions:
Maximum operations reached.
deforests
deforest
deforested
Total suggestions: 3
Lookup and suggestion took 74,90 ms
```

Conclusion

In this project, I developed a high-performance spell checker using custom data structures that I built from scratch. By implementing key components such as `GTUHashMap`, `GTUHashSet`, and `GTUArrayList`, I was able to build a modular system without using Java's built-in collection classes.

Throughout the process, I gained hands-on experience with data structure topics like hashing, open addressing, quadratic probing for collision handling, and rehashing. I believe that designing my own data structures helped me improve my skills in algorithm development.

Working on optimizing execution time, removing unnecessary loops, and managing memory helped me understand important details of software development more clearly. One of the most valuable lessons I learned in this project was that it is not enough for code to work correctly, it also needs to work efficiently.

Overall, this project helped me improve my technical skills and increased my awareness in areas such as algorithmic thinking, debugging, and software design.

References

[1] CSE 222 Homework 6 PDF

[2] Koffman & Wolfgang Data Structures Book

[3] Uml Diagram: https://www.plantuml.com/plantuml/dual.html

[4] Edit distance : https://www.geeksforgeeks.org/edit-distance-dp-5/

[5] Prime number: https://www.geeksforgeeks.org/java-prime-number-program/

[6] Word edits: https://www.geeksforgeeks.org/stringbuilder-substring-method-in-java-with-examples/