Gebze Technical University

CSE 222 – Homework 6

High-Performance Spell Checker with Custom HashMap and Set

Project Report

Osmancan Bozali – 220104004011

----

1. Introduction

This report is the detailed explanation of a high-performance spell checker program which is built using custom (own implemented) HashMap and Set data structures. The project consists of designing and implementing two fundemantal data structures without using built in data structures libraries (no java.util): a hash map (GTUHashMap) and a set (GTUHashSet). Also, a simple list (GTUBasicList) with limited operations is implemented extra due to special needs. These custom data structures are used for creating a high-performance spell checker capable of suggesting corrections for misspelled words by manipulating the input within edit distance <= 2.

The main requirements of this project are:

- Developing a custom HashMap which must use open addressing and linear/quadratic probing for collusion resolution (In this implementation quadratic probing is implemented to get bonus points).
- Developing a custom Set which must use the custom HashMap data structure to store elements.

- Developing a spell checker program that uses the custom data structures (Set) to store a dictionary and generate suggestions for correcting misspelled words under 100ms.

In the following parts of this report, both custom data structures and the spell checker logic are explained in detail as well as design choices and their reasons. Also some tests and results are added for demonstration.

2. Methodology

2.1 HashMap Implementation

The GTUHashMap<K, V> class is implemented as a generic key-value storage structure. Open addressing and quadratic probing is used for implementation rather than chaining as requested in the homework pdf. Open addressing with linear/quadratic probing approach is better than chaining because it provides better cache locality since all elements are stored in a single array. Also open addressing with probing does not use pointers which makes it use less memory than chaining.

2.1.1 Data Structure Design

The HashMap internally uses an array of Entry<K, V> objects, where each entry stores a key, a value and a deletion flag. The deletion flag is used for implementing the tombstone mechanism required. Also the HashMap has two integer variables, size for tracking the active elements in the HashMap and occupiedSlots for tracking both active and deactive (tombstone) elements. There are 3 constants which are initial capacity of the HashMap, maximum capacity of the HashMap and load factor (which is explained below).

2.1.2 Hashing and Index Calculation

To calculate the index for a key, Java's built in hashCode() method is used. To make sure that no negative values for index created, the most significant bit of the hashCode() function's output is reverted to 0, if it is 1. This technique is selected over taking the absolute value because bitwise operations are faster than taking absolute value since there might be condition checking while taking the absolute value. Also modulo arithmetic is applied to fit the index value within the array bounds.

For collusion resolution, quadratic probing is chosen over linear probing to get extra points, also to benefit from the advantage of spreading out the probe locations in quadratic probing. The following formula is used for generating the probe sequence:

$h(k, i) = (h(k) + i^2) \bmod m$

Where h(k) is the initial hash value of key k, i is the probe number and m is the array size.

2.1.3 Dynamic Resizing

To maintain the performance of the HashMap as it grows, a dynamic resizing mechanism is implemented. This mechanism monitors the load factor (which is the ratio of occupied slots to array size), triggers resizing when the load factor exceeds the determined threshold value (which is set to 0.75 in this implementation), calculates the new capacity which is the next prime number doubling the current capacity (This logic is also implemented for bonus points, also using prime numbers for the table size helps distribute hash values more evenly across the table helping to reduce collusions), checks whether the overflow of the maximum capacity is occurred, recreates the array with new capacity and rehashes all existing entries in the new table. Those are implemented with the methods of isPrime(int n), findNextPrime(int n) and resize().

2.1.4 Core Operations

put(K key, V value): This function adds a key-value pair to the HashMap. It returns IllegalArgumentException if key is null. It checks the load factor to determine whether a resizing is needed and calls resize() according to it. After that it calculates starting index and initializes prob number (which is used for calculation of probing). In a for loop which can iterate size times in order to guarantee each slot is checked for insertion (since probing gives another index at each iteration). It also initializes a firstTombstoneIndex variable which is tracking the first deleted entry encountered while probing in order to use it for putting to provide a more efficient putting strategy (reusing the space of deleted entries makes the implementation more efficient and space-friendly). While probing if an empty slot is found and there is no previous tombstones encountered, new key-value pair is added to this index and size is incremented. If there is firstTombstoneIndex, new key-value pair is added to firstTombstoneIndex. If there is an active slot encountered, it checks whether the keys are matching. If so, it updates the value, if it is not it moves on with probing. The occupiedSlots is incremented only if inserting into a previously unoccupied slot.

get(K key): This function returns the value of a key if it is exists in the HashMap. It returns IllegalArgumentException if parameter key is null. It initializes the initialIndex for searching by probing quadratically. If key is found and it is an active entry, it returns the value. If the key corresponds to an inactive entry, it continues searching with probing. If it encounters a null entry (which means empty) it returns null since the key cannot be found by further probing and it does not exist in the HashMap.

remove(K key): This function removes the key-value pair from HashMap by setting the entries isDeleted flag to true. It returns IllegalArgumentException if parameter key is null. It initializes the initialIndex for searching by probing quadratically. If empty slot is encountered function returns since the key cannot be found by further probing and it does not exist in the HashMap. If an occupied slot is found, it checks whether the keys match and the slot is active. If keys match and slot is active, it sets the slot inactive (making it tombstone). If not, it continues searching until all array is searched.

containsKey(K key): This function searches the HashMap and checks whether the parameter key exists in the HashMap. If exists, it returns true; otherwise it returns false. It returns IllegalArgumentException if parameter key is null. It initializes the initialIndex for searching by probing quadratically. If empty slot is encountered function returns false since the key cannot be found by further probing and it does not exist in the HashMap. If an occupied slot is found, it checks whether the keys match and the slot is active. If keys match and slot is active, returns true. Otherwise it keeps searching until all array is searched. It is implemented just like the remove function, their searching logic is the exact same.

size(): This function returns the size of the HashMap. Size represents the active element count in the HashMap.

getKeys(): This function is implemented for a reason similar to iterator. Instead of implementing iterator, this function is implemented. It returns a list of active elements in HashMap by traversing in the inner array with for loop. It returns the list as

GTUBasicList type which is a custom basic list implementation explained in below sections. This is an extra function, it is used in Spell Checker program.

2.2 HashSet Implementation

The GTUHashSet<E> class is implemented as a wrapper around the GtuHashMap<K, V> using the custom HashMap's logic and mechanisms to provide set functionality.

2.2.1 Data Structure Design

The HashSet internally uses a GTUHashMap<E, Object>, with a single placeholder object as the value for all entries.

2.2.2 Core Operations

add(E element): This function adds an element to the set by calling the HashMap's put function. Since for same key the put function updates the value, single existance of an element is ensured by the HashMap.

remove(E element): This function removes an element from the set by calling the HashMap's remove function.

contains(E element): This function checks if an element exists in the set by calling the HashMaps's containsKey function. It return true if set contains the element, false otherwise.

size(): This function returns the number of the elements in the set by calling the size function of the HashMap.

getAllElements(): This function is implemented for a reason similar to iterator. Instead of implementing iterator, this function is implemented. It returns a list of elements in set by calling the getKeys function of HashMap. It returns the list as GTUBasicList type which is a custom basic list implementation explained in below sections.

2.3 BasicList Implementation (Extra Data Structure)

The GTUBasicList<E> class is implemented as an alternative to ArrayList since java.util libraries are forbidden. Since this is not in the main requirements of the homework, only needed functionality is implemented. That's why it is named as Basic List, it just provides a way of using dynamically resizing array which is used in Spell Checker implementation.

2.3.1 Data Structure Design

The BasicList uses an array of Object type, and a size variable for tracking its size. Also, there is an initial capacity constant which determines the initial capacity of array.

2.3.2 Core Operations

add(E element): This function adds an element to the list. It checks whether the capacity is reached, if so it calls reallocate function to increase the capacity of the list (reallocate function is explained below). It adds the element to list and increases the size variable.

get(int index): This function returns the element in the parameter index. It checks whether the index is in bounds, if not throws IndexOutOfBoundsException.

size(): This function returns the number of the elements in the list.

reallocate(): This function doubles the capacity of the inner object array by creating a new array with double capacity and transferring the elements one by one using a for loop. It provides the resizing functionality to the list and the main reason of implementing a custom list.

toString(): This function returns the string representation of the elements by adding them to a string builder in a format of "[element1, element2, element3, ..., elementN]" and calling the string builder's toString function. It is used for printing the suggestions in the Spell Checker program.

2.4 Spell Checker Implementation

The SpellChecker class implements a high-performance word validation and suggestion system using the custom data structures that explained above. It generates word variations within <=2 letter edit distance and checks each generated variant whether it exists in the dictionary. After it's generation and search, it prints the generated valid suggestions as well as the time taken for lookup and suggestion.

2.4.1 Data Structure Design

The SpellChecker uses a GTUHashSet to store dictionary words, Also there are 2 constants, first one representing the alphabet letters and other one is the threshold word length for switching to the alternative pruning strategy (both strategies and reasons are explained and discussed below).

2.4.2 Core Operations

loadDict(String filePath): Loads a dictionary file into a GTUHashSet<String>. This function reads the file line by line, trims each word in case of whitespaces, converts it to lowercase (in given dictionary file each letter is lowercase except É, this operation is done for double checking), and adds it to the dictionary set.

getSuggestions(String misspelledWord): This function generates potential corrections for a misspelled word based on edit distance. Returns a list of valid dictionary words that are within edit distance <= 2 of the misspelled word. Also the decision of which pruning method will be used is implemented in this function. (See Key Algorithms and Logic section to learn more about pruning methods implemented) It calls the processEdits function and uses the firstOrderEdits set to create and check second order edits by sending the first order variants (edit distance = 1) to processEdits again. Finally it returns the foundSuggestions set which is retrivied from the processEdits functin and contains valid suggestions by calling the getAllElements function of GTUHashSet to return the suggestions in a list format.

processEdits(String wordToEdit, ...): This function generates all possible edit distance 1 variants for a given word by applying four types of edit operations: deletions, transpositions, replacements, and insertions. For each variant generated, calls processVariant function to determine if it is a valid word and should be added to the found suggestions set. The types of the edit operations are explained in Key Algorithms and Logic section.

processVariant(String variant, ...): This function checks each generated word variant to determine whether it should be added to the found suggestions set (if it's a valid dictionary word) and whether it should be added to the first-order edits collection for further processing (used in alternative pruning, explained in the next section).

main(String[] args): This is the main function of the Spell Checker program. It provides a command-line interface for the spell checker. It initializes the spell checker with a dictionary file, sets up a infinite loop to accept user input (it breaks if the input is EXIT all uppercase), processes each input word, and displays results and time taken for lookup and suggestions. It also catches the errors occurred and prints the error messages if any occurs.

2.4.3 Key Algorithms and Logic

Edit Distance Calculation: The spell checker uses edit distance <=2 for generating suggestions. Edit distance is a measure of how many operations (insertions, deletions, substitutions, or transpositions[extra operation, not in pdf, added for more powerfull suggestion generation]) are needed to transform one word into another. In this implementation there is a 2 step approach:

1. Generate all words with edit distance 1 (ED1) from the misspelled word
2. Generate all words with edit distance 1 from each ED1 word, resulting in words with edit distance 2 (ED2) from the original word

This approach is chosen rather than directly calculating all possible ED2 words because it prevents redundant code and allows for using more than one pruning strategy.

Edit Operations:

1. **Insertions**: Insert a new character at any position
2. **Deletions**: Remove one character from the word
3. **Replacements**: Replace one character with another
4. **Transpositions**: Swap two adjacent characters (extra added for better suggestion generation, most misspelling occurs from this reason such as reason → reason etc.)

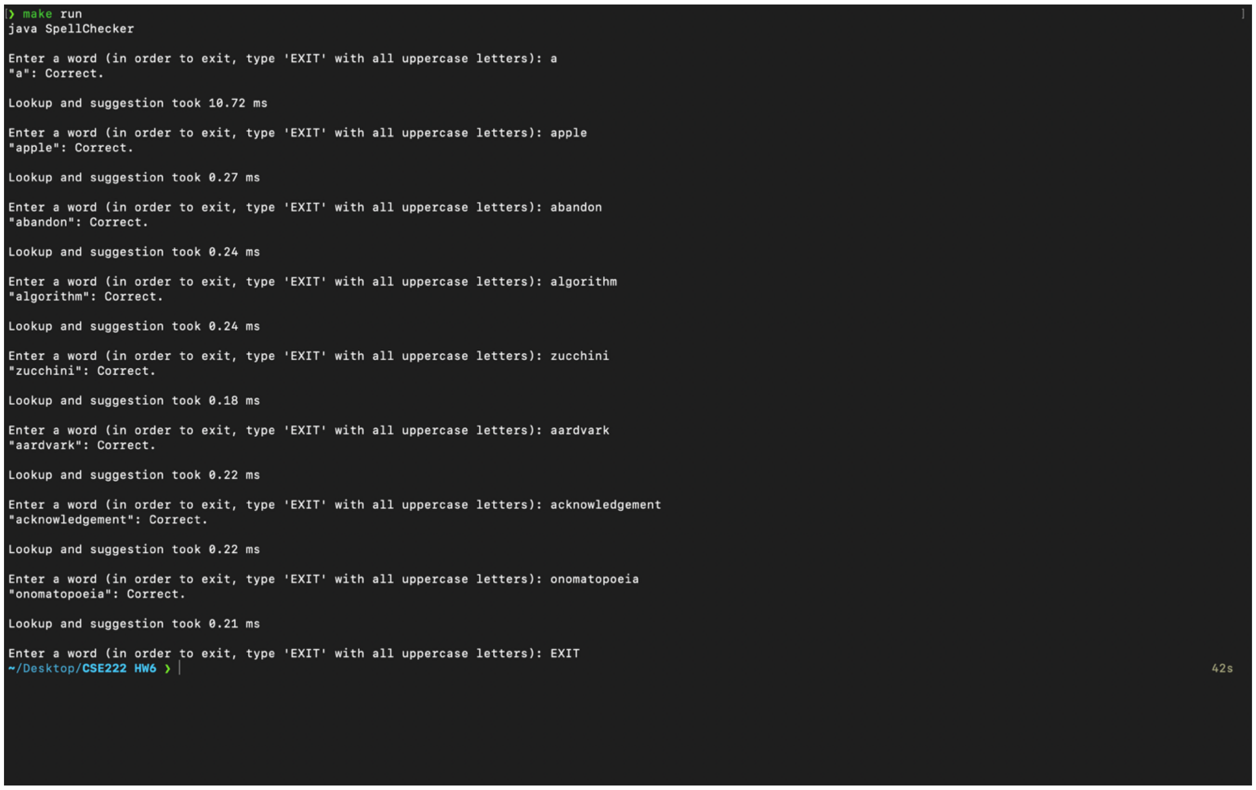
Pruning Strategies: In order to improve the performance of the spell checker and <100ms constraint, two pruning strategies are implemented. Alternative pruning strategy is used for words longer then a threshold (which is 20, it should be arranged according to the computers specifications) by reducing the ED1 elements used for ED2. This approach is sacrificing some of the potential suggestions for long words, but it keeps the lookup and suggestion time under 100ms as requested in the pdf. In the live demonistration of the homework, this part will be explained better by me.

1. **Basic Pruning**: Skip processing if a second-level edit operation results in the original misspelled word, otherwise process all edit distance variants (both ED1 and ED2) by checking whether they are existing in the dictionary. This is the requested approach in the homework and it works well on my computer up to 20 word length words.
2. **Alternative Pruning for Long Words**: For words longer than the threshold only first-order variants that are valid dictionary words are added first order edits set in order the reduce the processed word count. After a certain word length, the variation count rises rapidly which leads to a performance >100ms. This may not be the expected strategy in the pdf, it is implemented due to time constraint. If the threshold value is determined correctly for the system that is running that program, it does not exceed 100ms while lookup and suggestion while sacrificing some of the potential suggestion may found if basic pruning used.

Memory Optimization: The implementation uses a single StringBuilder for all edit operations, which is reused throughout the process rather than creating new string objects for each variant. At first implementation it was not like this, it is added later to optimize the suggestion process. Due to tests made during development phase by me, it speeds up the suggestion process successfully.

3. Results And Discussion

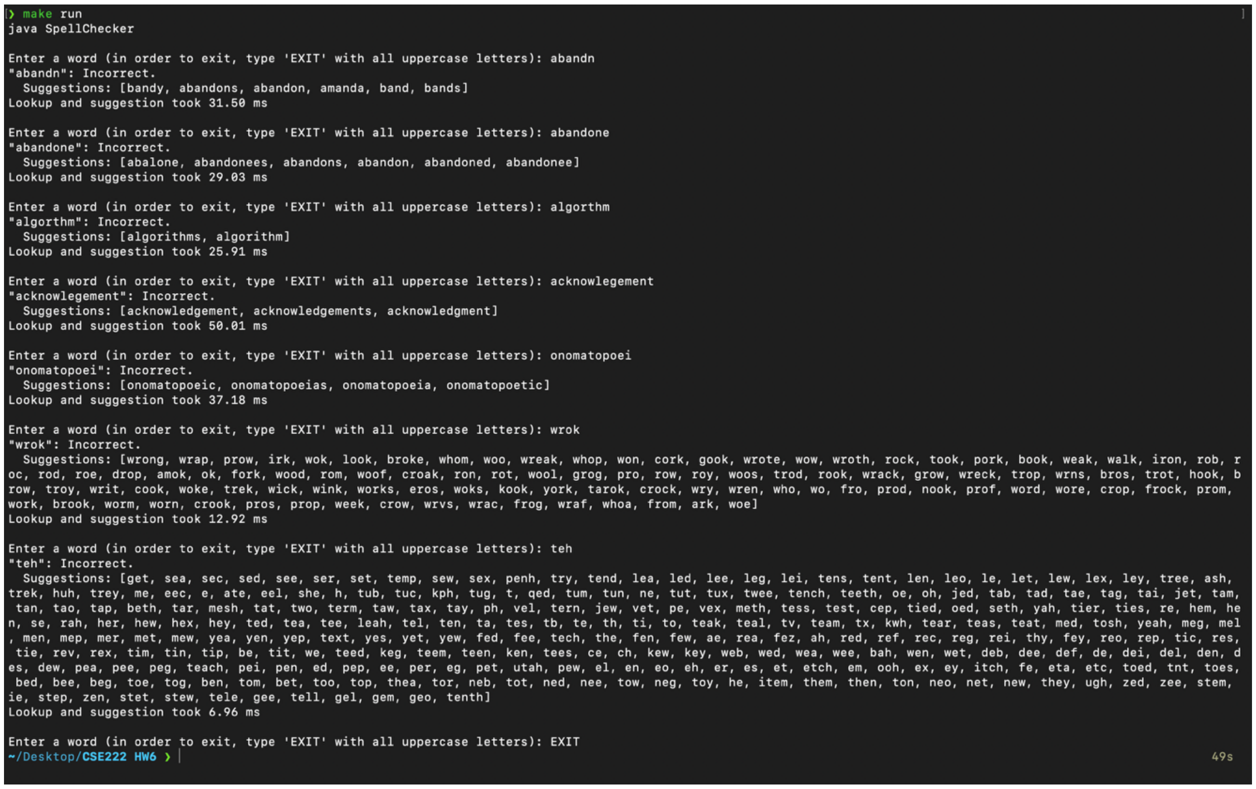
Correct Words: This test is for checking the spell checker's ability to correctly identify words that exist in the loaded dictionary, including various word lengths. The expected result for all inputs in this category is the program printing a "Correct" message for the input word under 100ms.



```
❯ make run
java SpellChecker

Enter a word (in order to exit, type 'EXIT' with all uppercase letters): a
"a": Correct.

Lookup and suggestion took 10.72 ms

Enter a word (in order to exit, type 'EXIT' with all uppercase letters): apple
"apple": Correct.

Lookup and suggestion took 0.27 ms

Enter a word (in order to exit, type 'EXIT' with all uppercase letters): abandon
"abandon": Correct.

Lookup and suggestion took 0.24 ms

Enter a word (in order to exit, type 'EXIT' with all uppercase letters): algorithm
"algorithm": Correct.

Lookup and suggestion took 0.24 ms

Enter a word (in order to exit, type 'EXIT' with all uppercase letters): zucchini
"zucchini": Correct.

Lookup and suggestion took 0.18 ms

Enter a word (in order to exit, type 'EXIT' with all uppercase letters): aardvark
"aardvark": Correct.

Lookup and suggestion took 0.22 ms

Enter a word (in order to exit, type 'EXIT' with all uppercase letters): acknowledgement
"acknowledgement": Correct.

Lookup and suggestion took 0.22 ms

Enter a word (in order to exit, type 'EXIT' with all uppercase letters): onomatopoeia
"onomatopoeia": Correct.

Lookup and suggestion took 0.21 ms

Enter a word (in order to exit, type 'EXIT' with all uppercase letters): EXIT
~/Desktop/CSE222 HW6 ❯                                                                        42s
```

Actual results match with the expected result.

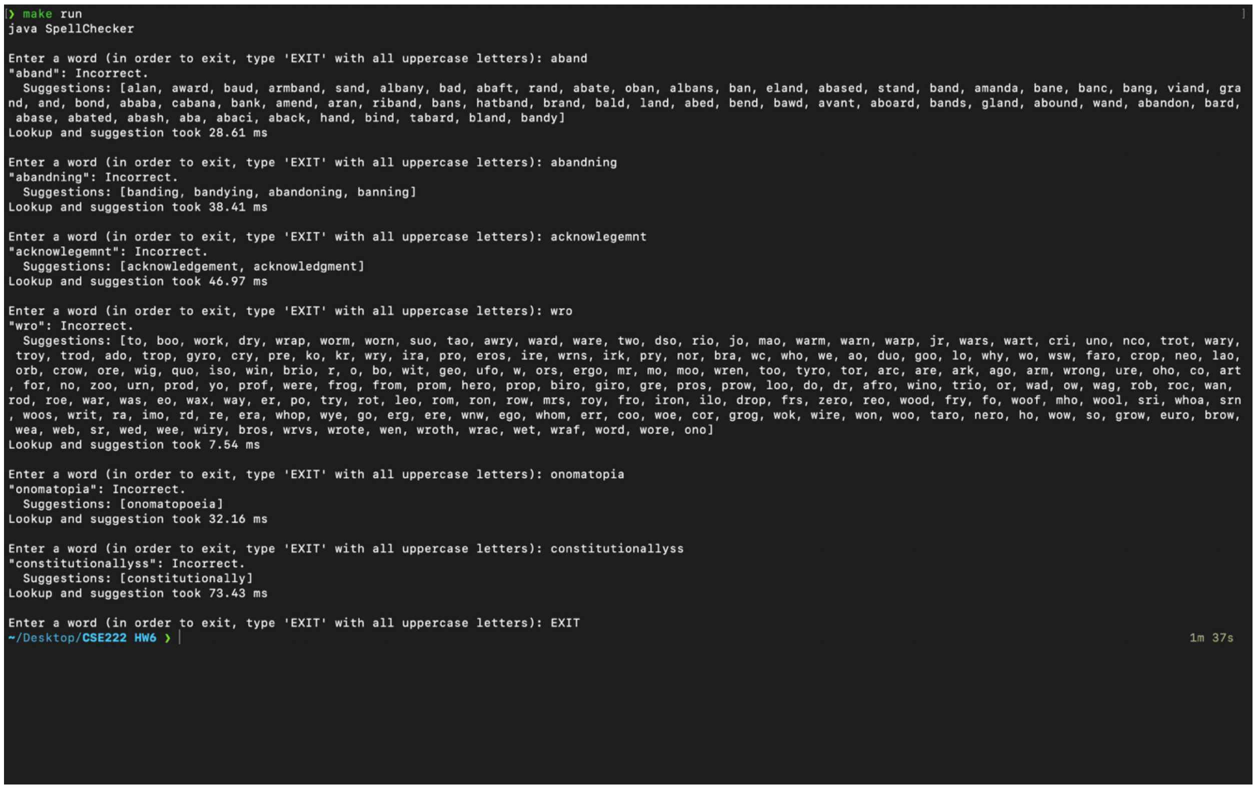
Incorrect Words within Edit Distance 1: This test is for checking the spell checker's ability to identify words that are one edit away from a word in the dictionary and to suggest those correct dictionary words. The expected result is the program printing an "Incorrect" message for the input word, followed by a list of suggestions that includes the correct dictionary word(s) reachable by a single edit, all within the 100ms time limit.



```
❯ make run
java SpellChecker

Enter a word (in order to exit, type 'EXIT' with all uppercase letters): abandn
"abandn": Incorrect.
  Suggestions: [bandy, abandons, abandon, amanda, band, bands]
Lookup and suggestion took 31.50 ms

Enter a word (in order to exit, type 'EXIT' with all uppercase letters): abandone
"abandone": Incorrect.
  Suggestions: [abalone, abandonees, abandons, abandon, abandoned, abandonee]
Lookup and suggestion took 29.03 ms

Enter a word (in order to exit, type 'EXIT' with all uppercase letters): algorthm
"algorthm": Incorrect.
  Suggestions: [algorithms, algorithm]
Lookup and suggestion took 25.91 ms

Enter a word (in order to exit, type 'EXIT' with all uppercase letters): acknowlegement
"acknowlegement": Incorrect.
  Suggestions: [acknowledgement, acknowledgements, acknowledgment]
Lookup and suggestion took 50.01 ms

Enter a word (in order to exit, type 'EXIT' with all uppercase letters): onomatopoei
"onomatopoei": Incorrect.
  Suggestions: [onomatopoeic, onomatopoeias, onomatopoeia, onomatopoetic]
Lookup and suggestion took 37.18 ms

Enter a word (in order to exit, type 'EXIT' with all uppercase letters): wrok
"wrok": Incorrect.
  Suggestions: [wrong, wrap, prow, irk, wok, look, broke, whom, woo, wreak, whop, won, cork, gook, wrote, wow, wroth, rock, took, pork, book, weak, walk, iron, rob, r
oc, rod, roe, drop, amok, ok, fork, wood, rom, woof, croak, ron, rot, wool, grog, pro, row, roy, woos, trod, rook, wrack, grow, wreck, trop, wrns, bros, trot, hook, b
row, troy, writ, cook, woke, trek, wick, wink, works, eros, woks, kook, york, tarok, crock, wry, wren, who, wo, fro, prod, nook, prof, word, wore, crop, frock, prom,
work, brook, worm, worn, crook, pros, prop, week, crow, wrvs, wrac, frog, wraf, whoa, from, ark, woe]
Lookup and suggestion took 12.92 ms

Enter a word (in order to exit, type 'EXIT' with all uppercase letters): teh
"teh": Incorrect.
  Suggestions: [get, sea, sec, sed, see, ser, set, temp, sew, sex, penh, try, tend, lea, led, lee, leg, lei, tens, tent, len, leo, le, let, lew, lex, ley, tree, ash,
trek, huh, trey, me, eec, e, ate, eel, she, h, tub, tuc, kph, tug, t, qed, tum, tun, ne, tut, tux, twee, tench, teeth, oe, oh, jed, tab, tad, tae, tag, tai, jet, tam,
 tan, tao, tap, beth, tar, mesh, tat, two, term, taw, tax, tay, ph, vel, tern, jew, vet, pe, vex, meth, tess, test, cep, tied, oed, seth, yah, tier, ties, re, hem, he
n, se, rah, her, hew, hex, hey, ted, tea, tee, leah, tel, ten, ta, tes, tb, te, th, ti, to, teak, teal, tv, team, tx, kwh, tear, teas, teat, med, tosh, yeah, meg, mel
, men, mep, mer, met, mew, yea, yen, yep, text, yes, yet, yew, fed, fee, tech, the, fen, few, ae, rea, fez, ah, red, ref, rec, reg, rei, thy, fey, reo, rep, tic, res,
 tie, rev, rex, tim, tin, tip, be, tit, we, teed, keg, teem, teen, ken, tees, ce, ch, kew, key, web, wed, wea, wee, bah, wen, wet, deb, dee, def, de, dei, del, den, d
es, dew, pea, pee, peg, teach, pei, pen, ed, pep, ee, per, eg, pet, utah, pew, el, en, eo, eh, er, es, et, etch, em, ooh, ex, ey, itch, fe, eta, etc, toed, tnt, toes,
 bed, bee, beg, toe, tog, ben, tom, bet, too, top, thea, tor, neb, tot, ned, nee, tow, neg, toy, he, item, them, then, ton, neo, net, new, they, ugh, zed, zee, stem,
ie, step, zen, stet, stew, tele, gee, tell, gel, gem, geo, tenth]
Lookup and suggestion took 6.96 ms

Enter a word (in order to exit, type 'EXIT' with all uppercase letters): EXIT
~/Desktop/CSE222 HW6 ❯ |                                                                                                                                         49s
```

Actual results match with the expected result.

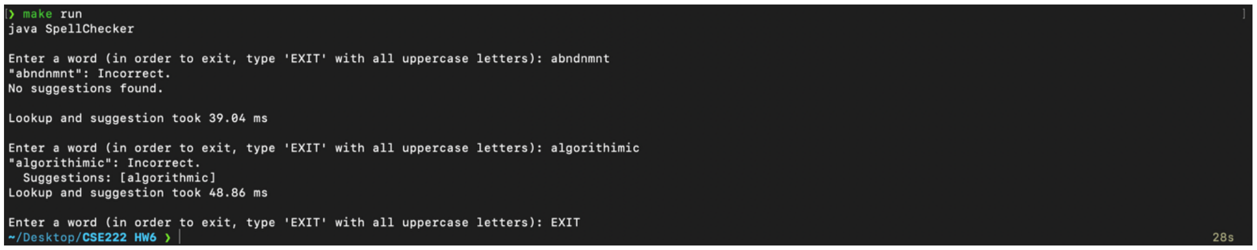
Incorrect Words within Edit Distance 2: This test is for checking the spell checker's ability to identify words that are two edits away from a word in the dictionary and to suggest those correct dictionary words. The expected result is the program printing an "Incorrect" message for the input word, followed by a list of suggestions that includes the correct dictionary word(s) reachable by two edits, all within the 100ms time limit.



```
❯ make run
java SpellChecker

Enter a word (in order to exit, type 'EXIT' with all uppercase letters): aband
"aband": Incorrect.
  Suggestions: [alan, award, baud, armband, sand, albany, bad, abaft, rand, abate, oban, albans, ban, eland, abased, stand, band, amanda, bane, banc, bang, viand, gra
nd, and, bond, ababa, cabana, bank, amend, aran, riband, bans, hatband, brand, bald, land, abed, bend, bawd, avant, aboard, bands, gland, abound, wand, abandon, bard,
 abase, abated, abash, aba, abaci, aback, hand, bind, tabard, bland, bandy]
Lookup and suggestion took 28.61 ms

Enter a word (in order to exit, type 'EXIT' with all uppercase letters): abandning
"abandning": Incorrect.
  Suggestions: [banding, bandying, abandoning, banning]
Lookup and suggestion took 38.41 ms

Enter a word (in order to exit, type 'EXIT' with all uppercase letters): acknowlegemnt
"acknowlegemnt": Incorrect.
  Suggestions: [acknowledgement, acknowledgment]
Lookup and suggestion took 46.97 ms

Enter a word (in order to exit, type 'EXIT' with all uppercase letters): wro
"wro": Incorrect.
  Suggestions: [to, boo, work, dry, wrap, worm, worn, suo, tao, awry, ward, ware, two, dso, rio, jo, mao, warm, warn, warp, jr, wars, wart, cri, uno, nco, trot, wary,
 troy, trod, ado, trop, gyro, cry, pre, ko, kr, wry, ira, pro, eros, ire, wrns, irk, pry, nor, bra, wc, who, we, ao, duo, goo, lo, why, wo, wsw, faro, crop, neo, lao,
 orb, crow, ore, wig, quo, iso, win, brio, r, o, bo, wit, geo, ufo, w, ors, ergo, mr, mo, moo, wren, too, tyro, tor, arc, are, ark, ago, arm, wrong, ure, oho, co, art
, for, no, zoo, urn, prod, yo, prof, were, frog, from, prom, hero, prop, biro, giro, gre, pros, prow, loo, do, dr, afro, wino, trio, or, wad, ow, wag, rob, roc, wan,
rod, roe, war, was, eo, wax, way, er, po, try, rot, leo, rom, ron, row, mrs, roy, fro, iron, ilo, drop, frs, zero, reo, wood, fry, fo, woof, mho, wool, sri, whoa, srn
, woos, writ, ra, imo, rd, re, era, whop, wye, go, erg, ere, wnw, ego, whom, err, coo, woe, cor, grog, wok, wire, won, woo, taro, nero, ho, wow, so, grow, euro, brow,
 wea, web, sr, wed, wee, wiry, bros, wrvs, wrote, wen, wroth, wrac, wet, wraf, word, wore, ono]
Lookup and suggestion took 7.54 ms

Enter a word (in order to exit, type 'EXIT' with all uppercase letters): onomatopia
"onomatopia": Incorrect.
  Suggestions: [onomatopoeia]
Lookup and suggestion took 32.16 ms

Enter a word (in order to exit, type 'EXIT' with all uppercase letters): constitutionallyss
"constitutionallyss": Incorrect.
  Suggestions: [constitutionally]
Lookup and suggestion took 73.43 ms

Enter a word (in order to exit, type 'EXIT' with all uppercase letters): EXIT
~/Desktop/CSE222 HW6 ❯                                                                                                                                        1m 37s
```

Actual results match with the expected result.

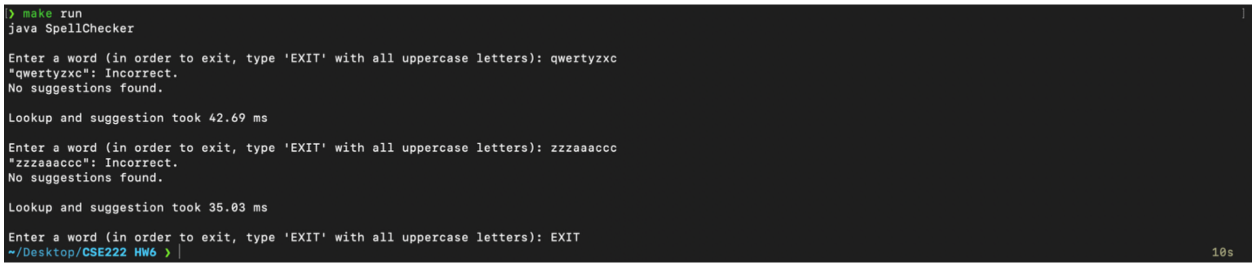
Incorrect Words within Edit Distance > 2: This test is for checking the spell checker's ability to handle words that are significantly different (more than two edits away) from any word in the dictionary. The expected result is the program printing an "Incorrect" message, and it maybe produce very few or no suggestions, showing that it doesn't find close matches beyond the edit distance 2, within the 100ms limit.



```
❯ make run
java SpellChecker

Enter a word (in order to exit, type 'EXIT' with all uppercase letters): abndnmnt
"abndnmnt": Incorrect.
No suggestions found.

Lookup and suggestion took 39.04 ms

Enter a word (in order to exit, type 'EXIT' with all uppercase letters): algorithimic
"algorithimic": Incorrect.
  Suggestions: [algorithmic]
Lookup and suggestion took 48.86 ms

Enter a word (in order to exit, type 'EXIT' with all uppercase letters): EXIT
~/Desktop/CSE222 HW6 ❯                                                                                                                                        28s
```

Actual results match with the expected result.

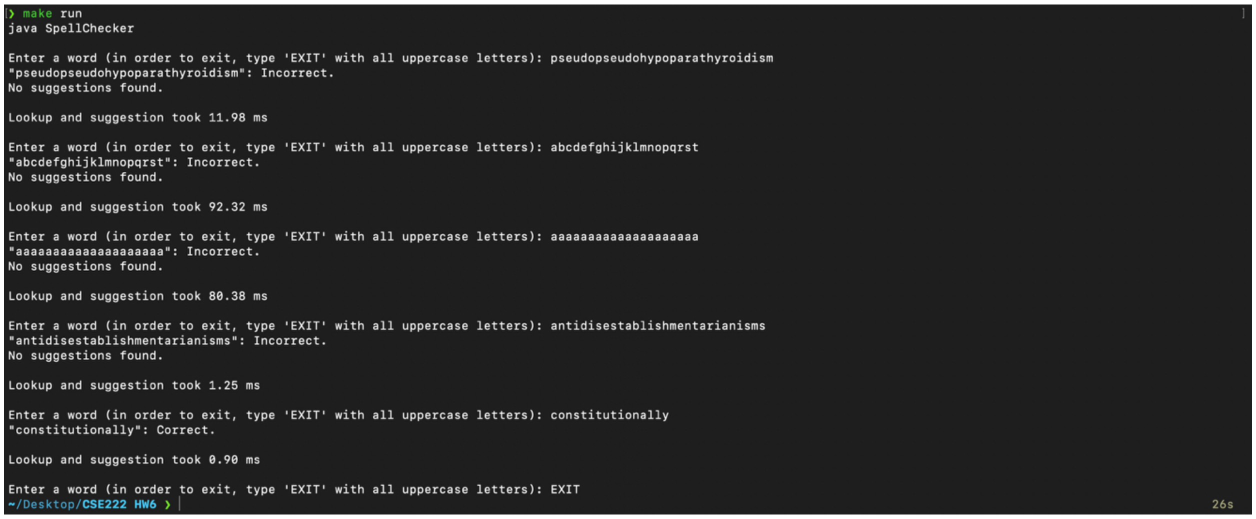
No Suggestions Expected: This test is for checking the spell checker with random character inputs that are unlikely to have any valid dictionary words within an edit distance of 2. The expected result is the program printing an "Incorrect" message followed by a "No suggestions found" message, within the 100ms limit.



```
❯ make run
java SpellChecker

Enter a word (in order to exit, type 'EXIT' with all uppercase letters): qwertyzxc
"qwertyzxc": Incorrect.
No suggestions found.

Lookup and suggestion took 42.69 ms

Enter a word (in order to exit, type 'EXIT' with all uppercase letters): zzzaaaccc
"zzzaaaccc": Incorrect.
No suggestions found.

Lookup and suggestion took 35.03 ms

Enter a word (in order to exit, type 'EXIT' with all uppercase letters): EXIT
~/Desktop/CSE222 HW6 ❯                                                                                10s
```

Actual results match with the expected result.

Too Long Words: This test uses very long words (both correct and incorrect) to test the efficiency of the dictionary lookups and the suggestion generation process for long words. The expected result is the program should complete its check and suggestion generation within the 100ms time limit, with the alternative probing for words larger than 20 characters.



```
❯ make run
java SpellChecker

Enter a word (in order to exit, type 'EXIT' with all uppercase letters): pseudopseudohypoparathyroidism
"pseudopseudohypoparathyroidism": Incorrect.
No suggestions found.

Lookup and suggestion took 11.98 ms

Enter a word (in order to exit, type 'EXIT' with all uppercase letters): abcdefghijklmnopqrst
"abcdefghijklmnopqrst": Incorrect.
No suggestions found.

Lookup and suggestion took 92.32 ms

Enter a word (in order to exit, type 'EXIT' with all uppercase letters): aaaaaaaaaaaaaaaaaaaa
"aaaaaaaaaaaaaaaaaaaa": Incorrect.
No suggestions found.

Lookup and suggestion took 80.38 ms

Enter a word (in order to exit, type 'EXIT' with all uppercase letters): antidisestablishmentarianisms
"antidisestablishmentarianisms": Incorrect.
No suggestions found.

Lookup and suggestion took 1.25 ms

Enter a word (in order to exit, type 'EXIT' with all uppercase letters): constitutionally
"constitutionally": Correct.

Lookup and suggestion took 0.90 ms

Enter a word (in order to exit, type 'EXIT' with all uppercase letters): EXIT
~/Desktop/CSE222 HW6 ❯                                                                                26s
```

Actual results match with the expected result.

4. Conclusion

This project is a successfull implementation of a high-performance spell checker program built with custom HashMap and HashSet data structures. The implementation of GTUHashMap and GTUHashSet was challenging but made me understand the internal workings of hash tables and the critical role of collision resolution strategies. By implementing quadratic probing instead of linear probing resulted in more efficient hash table operations. The tombstone deletion mechanism is forced me to implement a better put mechanism in order to use the space efficiently.

The spell checker program successfully showed the usage of custom data structures implemented. It also pushed me through my limits while optimizing the logic and implementing it.

In conclusion, this project is a successful implemetation of the homework requirements. The experience I have gained through implementing custom data structures and optimizing the algorithms was very beneficial. To be honest, it was fun even for a homework (which I do not like doing). After finishing this homework, I can say that now I now a lot more about hash based data structures, the importance of optimizing algorithms and how complex is a single spell checker is. I admire hashing and the logic behind it.

5. References

1. https://www.geeksforgeeks.org/introduction-to-hashing-2/
2. https://www.geeksforgeeks.org/internal-working-of-hashmap-java/
3. https://www.geeksforgeeks.org/open-addressing-collision-handling-technique-in-hashing/
4. https://stackoverflow.com/questions/17386138/quadratic-probing-over-linear-probing
5. https://www.youtube.com/watch?v=KyUTuwz_b7Q&list=WL&index=2