Date: 13/04/2024

ADVANCED DATA STRUCTURES

ASSIGNMENT - 4

Hashing : Spell Checker using Hash Table

Assignment Type of Submission:			
Group **COMP47500 Advanced Data Structures** **Group Assignment- 4**	Yes/No Yes	**List all group members' details:** Student Name: Avantika Sivakumar Student ID: 23205789 Student Name: Ria Rahul Asgekar Student ID: 23203987 Student Name: Jeevan Raju Ravi Govind Raju Student ID: 23207314	**% Contribution Assignment Workload** 33.33% (Code implementation, Result evaluation, video explanation) 33.33% (Code Implementation, material gathered for report, video explanation) 33.33% (Code implementation, material gathered for report, video explanation)

1. Problem Domain Description:

This assignment revolves around efficient storage, retrieval, and spell checking of words using Hash Tables and Linked lists to compare their performance and efficiency. The code implements a hash table data structure to organise words based on their hash values, enabling quick insertion and retrieval operations. We also perform spell check by searching for words in the hash table. If a word is found, it's considered correctly spelled; otherwise, it's flagged as misspelt.
The code implements a hash table data structure for efficient storage and retrieval of words, alongside a linked list for comparison. It measures insertion and search times for both data structures, aiming to evaluate their performance in spell checking applications.

1. Hash Table:

1.1. Overview:
Hash tables make use of key-value pairs where each key is mapped to a unique index in an array. They use a hash function to compute the index for each key, enabling fast insertion, retrieval, and deletion of elements.

1.2. Insertion:
The insertion operation in a hash table involves mapping a key to its corresponding index in the array and storing the key-value pair at that index. This process typically consists of the following steps:

- Hashing: to convert the key into a numerical value, which is used to determine the index in the array.
- Index Calculation- to map the hash value to an index in the array. This index serves as the location where the key-value pair will be stored.
- Collision Handling- to handle collisions that may occur when multiple keys are hashed to the same index.
- Insertion- to store the key-value pair at the computed index in the array.

Since the hash function evenly distributes elements across the array, the time complexity of insertion remains constant regardless of the number of elements in the hash table [6].

1.3. Search:
The search operation in a hash table involves retrieving the value associated with a given key. This process involves the following steps:

- Hashing: Similar to the insertion operation, the key is hashed to compute its index within the hash table's array.
- Bucket Access: Once the index is determined, the corresponding bucket is accessed to retrieve the values stored there.
- Search: Within the bucket, the search algorithm locates the desired value based on its key. This may involve iterating through the elements in the bucket or using a search algorithm optimised for the data structure used to handle collisions.

2. Linked List:

2.1. Overview:
In a linked list, the elements aren't necessarily stored in contiguous locations. Instead, each element contains a reference to the next node in the sequence, forming a chain-like structure.

2.2. Insertion:
The insertion operation in a linked list involves adding a new element to the list at a specified position.

2.3. Search:
The search operation in a linked list involves traversing the list to find a specific element with a given value. This typically requires starting from the head of the list and moving through each node sequentially until either the desired element is found or the end of the list is reached.

In conclusion, for the scope of this assignment, the hash table implementation exhibits notably higher efficiency compared to the linked list. With constant time complexities for both insertion and search operations on average, hash tables offer consistent performance regardless of the dataset size. In contrast, while linked lists provide O(1) insertion at the end, they suffer from linear search time complexity, resulting in slower performance, particularly with larger datasets. Therefore, the hash table emerges as the preferred data structure for applications requiring rapid and scalable word storage and retrieval, ensuring efficient spell checking functionality.

2. Theoretical Foundations of the Data Structure(s) utilised

A hash table relies on a hash function to map keys (such as words) to indices in an array. The hash function takes a key as input and produces a fixed-size hash value. The goal of the hash function is to evenly distribute keys across the array to minimise collisions. A good hash function generates unique hash codes for different keys while avoiding clustering, where multiple keys map to the same index.

2.1. Collision Resolution:
Collisions occur when two or more keys hash to the same index in the array. Hash tables employ collision resolution strategies to handle collisions effectively. Common approaches include:

- Separate Chaining: In separate chaining, each array index contains a linked list (or another data structure) that stores all keys hashing to that index. Colliding keys are appended to the linked list at the corresponding index.

- Open Addressing: In open addressing, collisions are resolved by finding an alternative (or "secondary") position within the array. This may involve linear probing (trying

consecutive array positions), quadratic probing (using a quadratic function to find the next position), or double hashing (applying a second hash function to calculate the next position).

2.2. Load Factor and Rehashing:
The load factor of a hash table represents the ratio of the number of stored elements to the size of the array. A high load factor increases the likelihood of collisions, potentially degrading performance. To maintain efficiency, hash tables often resize dynamically (rehashing) when the load factor exceeds a certain threshold. During rehashing, the size of the array is increased, and all elements are rehashed into the new array. This helps to distribute elements more evenly and reduce the chance of collisions.

3. Applications:

3.1. Hash Tables:

- Databases and Indexing: Hash tables are extensively used in databases and indexing systems to store and retrieve data efficiently. They provide fast access to records based on keys, allowing for quick searches and retrieval of information.

- Symbol Tables in Compilers: Hash tables are used in compilers to implement symbol tables, which store identifiers (e.g., variables, functions) and their associated attributes (e.g., data type, scope). This allows compilers to quickly look up identifiers during parsing and semantic analysis.

- Caching: Hash tables are employed in caching systems to store frequently accessed data, such as web pages, database query results, or computed values. By storing data in a hash table, caching systems can quickly determine if a requested item is already cached, reducing the need to regenerate or fetch it from the original source.

- Distributed Hash Tables: They are decentralised, distributed systems that use hash tables to store and retrieve data across a network of nodes. They are commonly used in peer-to-peer networks, distributed file systems, and decentralised applications to facilitate efficient data sharing and retrieval among network participants.

- Hash-Based Data Structures: Several other data structures, such as hash maps, hash sets, and hash-based priority queues, are built using hash tables as their underlying implementation. These data structures provide efficient storage and retrieval of elements based on keys or values, making them useful in a wide range of applications.

- String Matching and Text Processing: Hash tables are used in string matching algorithms and text processing applications to efficiently search for patterns or substrings within large text corpora. By storing substrings and their positions in a hash table, these algorithms can quickly locate occurrences of patterns in the text.

- Password Authentication and Cryptography: Hash tables are employed in password authentication systems and cryptographic protocols to securely store hashed passwords or cryptographic keys. By using a hash function to map passwords or keys to hash values, sensitive information can be stored in a hashed form, protecting it from unauthorised access.

- Spell Checking and Dictionary Lookup: Hash tables are used in spell checking applications and dictionary lookup systems to store a large collection of words or terms and quickly determine if a given word or term is present in the dictionary. This allows for efficient spell checking and auto correct in text editing software and search engines. The explosion of user-generated content on the internet has led to a vast amount of heterogeneous data, varying greatly in quality due to differences in language usage and expertise. Spell checker applications utilising hashing play a crucial role in improving the quality of this data by efficiently identifying and correcting errors, ensuring more reliable outcomes for data mining and recommender systems. [3]

3.2. Linked Lists:

- Dynamic Memory Allocation: Linked lists are used in memory management systems for dynamic memory allocation. They allow for efficient allocation and deallocation of memory blocks of varying sizes, making them suitable for implementing memory pools and dynamic memory allocation algorithms.

- File Systems: Linked lists are used in file systems to maintain the directory structure and manage file metadata. Each directory entry or file record can be represented as a node in a linked list, with pointers to other directories or files.

- Undo Functionality in Text Editors: Many text editors implement undo functionality using a linked list of text changes. Each edit operation creates a new node in the linked list, allowing users to undo changes by traversing the list backward and applying the reverse of each edit.

- Music and Playlist Management: Linked lists are commonly used in music and playlist management applications to represent playlists. Each song can be represented as a node in a linked list, with pointers to the next and previous songs, enabling efficient traversal and navigation through the playlist.

- Sparse Matrices: Linked lists are used to represent sparse matrices, which contain a large number of zero elements. Each non-zero element is stored as a node in a linked list, with pointers to the next non-zero element in the same row and column.

- Symbol Tables in Compilers: Linked lists are used to implement symbol tables in compilers and interpreters. Each symbol in the program is stored as a node in a linked list, allowing for efficient lookup and management of symbols during compilation or interpretation.

- Graph Algorithms: Linked lists are used in graph algorithms such as depth-first search and breadth-first search to represent adjacency lists. Each vertex in the graph is represented as a node in a linked list, with pointers to adjacent vertices.

- History Management in Web Browsers: Linked lists are used to implement the back and forward navigation functionality in web browsers. Each visited webpage is stored as a node in a linked list, allowing users to navigate backward and forward through their browsing history.

- Job Scheduling in Operating Systems: Linked lists are used in job scheduling algorithms in operating systems. Each process in the system's job queue is represented as a node in a linked list, with pointers to the next process to be executed. This allows for efficient scheduling and management of system resources.

- Simulation of Train or Bus Routes: Linked lists can be used to simulate train or bus routes in transportation systems. Each station along the route is represented as a node in a linked list, with pointers to the next station. This allows for efficient traversal and navigation through the route.

4. Time Complexities:

4.1. Hash Tables:

4.1.1. Insertion: O(1) average, potentially O(n) worst case
In hash tables, insertion typically involves computing the hash value of the key to determine its index in the array and then inserting the key-value pair at that index. Assuming a good hash function and minimal collisions, insertion is O(1) on average. However, in the case of collisions, additional steps may be required for collision resolution, potentially degrading the time complexity to O(n) in the worst case.

4.1.2. Search: O(1) average, potentially O(n) worst case
Searching in hash tables also involves computing the hash value of the key to determine its index in the array. Once the index is determined, the corresponding value can be retrieved directly. Assuming a good hash function and minimal collisions, search is O(1) on average. However, in the worst case, where many keys hash to the same index (resulting in a long collision chain), search time complexity may degrade to O(n).

4.2. Linked Lists:

4.2.1. Insertion: O(1) at the beginning or end, potentially O(n)
In linked lists, insertion can occur at the beginning, end, or a specific position. Insertion at the beginning or end of a linked list involves updating a constant number of pointers and is therefore O(1). However, insertion at a specific position may require traversing the list to find the insertion point, resulting in a time complexity of O(n).

4.2.2. Search: O(n) due to linear traversal

Searching in linked lists involves traversing the list from the head node to locate the target element. Since each node may need to be visited, searching in linked lists has a time complexity of O(n), where n is the number of elements in the list. This is because the entire list may need to be traversed to find the target element.

In summary, hash tables offer efficient insertion and search operations on average, with the time complexity depending on factors such as the quality of the hash function and the presence of collisions. Insertion typically involves computing the hash value of the key and inserting the corresponding key-value pair, while search entails computing the hash value and retrieving the associated value directly. However, the worst-case time complexity may degrade if many keys hash to the same index, resulting in longer collision chains and potentially slower performance. In contrast, linked lists provide constant-time insertion at the beginning or end but may require linear traversal for insertion at specific positions. Similarly, search in linked lists involves traversing the list linearly from the head node, potentially leading to slower performance as all nodes may need to be visited to find the target element.

5. Advantages:

5.1. Hash Tables:

- Fast Average-Case Performance: Hash tables provide fast average-case performance for insertion, deletion, and search operations. With a good hash function and minimal collisions, these operations can be performed in constant time, leading to efficient data retrieval and manipulation.

- Flexible Key-Value Storage: Hash tables allow for flexible storage of key-value pairs, where keys are hashed to determine their storage location in the underlying array. This flexibility enables efficient storage and retrieval of data without the need for manual sorting or organising.

- Dynamic Memory Management: Hash tables dynamically allocate memory to accommodate new elements as needed. Unlike arrays with fixed sizes, hash tables can grow or shrink dynamically based on the number of elements stored, making them suitable for dynamic data storage requirements.

- Efficient Search Operations: Hash tables offer efficient search operations, especially when dealing with large datasets. With a constant-time average complexity for search, hash tables provide fast lookup times, making them ideal for applications requiring quick data retrieval.

- Collision Resolution: Hash tables provide mechanisms for collision resolution, allowing multiple keys to map to the same hash value. Techniques such as separate

chaining or open addressing ensure that collisions are handled efficiently, maintaining the integrity and performance of the hash table.

- Space Efficiency: Hash tables use memory efficiently by allocating space only for the elements stored in the table. This makes them suitable for applications with memory constraints, as they minimise wasted space compared to data structures like arrays or linked lists.

- Versatility: Hash tables can be used in a wide range of applications across various domains, including database systems, caching mechanisms, symbol tables, and associative arrays. Their versatility makes them a valuable tool for solving diverse computational problems efficiently.

5.2. Linked Lists:

- Dynamic Memory Allocation: Unlike arrays, linked lists dynamically allocate memory for elements, allowing for efficient memory utilisation. This dynamic allocation makes linked lists flexible and adaptable to changing data sizes.

- Constant-Time Insertion and Deletion: Insertion and deletion operations at the beginning or end of a linked list are typically O(1) in time complexity, regardless of the size of the list. This makes linked lists ideal for scenarios where frequent insertion and deletion operations are required.

- Flexibility in Size: Linked lists can grow or shrink in size dynamically without the need for resizing operations. This flexibility enables efficient management of data structures that may vary in size over time.

- No Fixed Size Limitation: Unlike arrays, linked lists do not have a fixed size limitation imposed by memory allocation. Linked lists can theoretically grow to accommodate as many elements as available memory allows.

- Efficient Memory Utilisation: Linked lists only use memory proportional to the number of elements they contain, avoiding the need for pre-allocation of memory as in arrays. This efficient memory utilisation is beneficial in scenarios where memory resources are limited or need to be optimised.

- Ease of Insertion and Deletion: Linked lists facilitate easy insertion and deletion of elements at any position within the list. Since only pointers need to be updated, these operations are typically faster and more straightforward compared to arrays, especially for large datasets.

- Support for Dynamic Data Structures: Linked lists serve as the foundation for implementing more complex data structures such as stacks, queues, and graphs. Their dynamic nature and efficient insertion and deletion operations make them well-suited for building dynamic data structures.

- Versatility: Linked lists come in various forms, including singly linked lists, doubly linked lists, and circular linked lists, each offering unique characteristics and advantages. This versatility allows developers to choose the most suitable type of linked list based on specific requirements and use cases.

Essentially, hash tables provide a powerful and efficient solution for storing and retrieving data based on keys, offering fast average case performance, dynamic resizing, effective collision handling, and memory efficiency. Contrarily, linked lists provide flexibility, efficient memory utilisation, and constant-time insertion and deletion operations, making them a valuable data structure for a wide range of applications in computer science and software engineering.

6. History:

Both hash tables and linked lists have evolved over several decades, with contributions from numerous researchers and practitioners in the field of computer science. While hash tables are relatively newer in terms of their practical implementation as a data structure, linked lists have a longer history and have been foundational to the development of modern computing.

6.1. Hash Tables:

- The concept of hashing dates back to the 1950s when computer scientists began exploring techniques for efficient data storage and retrieval.
- One of the earliest known applications of hashing was introduced by Hans Peter Luhn in 1953. He developed a method called "hash coding" to efficiently locate documents in a library catalogue system.
- In the late 1950s and early 1960s, researchers such as Arnold G. Farquhar and Edward A. Feigenbaum further refined hashing techniques for information retrieval systems.
- The first practical implementation of hash tables as a data structure for computer programming is often credited to Gene Amdahl, who introduced them in the IBM System/360 Model 91 in the 1960s. Amdahl's work significantly contributed to the widespread adoption of hash tables in computer science.
- Over the decades, hash tables have become a fundamental data structure in computer science, with various collision resolution techniques and optimizations developed to improve their efficiency and scalability. Today, they are used extensively in database systems, programming languages, networking protocols, and other applications requiring fast data access.

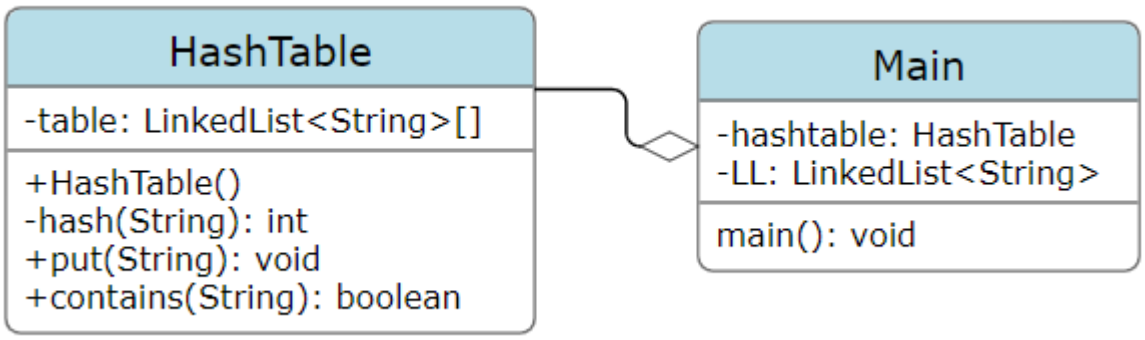
6.2. Linked Lists:

- Linked lists have a long history and can be traced back to the earliest days of computing.
- The concept of linked lists emerged in the 1950s as computer scientists sought efficient ways to manage dynamic data structures.

- One of the earliest documented references to linked lists appears in the work of Allen Newell, Cliff Shaw, and Herbert A. Simon in the late 1950s. They described linked lists as a fundamental data structure for representing and manipulating symbolic expressions in the Logic Theorist program.
- Linked lists gained prominence in the field of computer science in the 1960s and 1970s, with researchers such as Donald Knuth and Niklaus Wirth studying their properties and algorithms.
- In the 1970s and 1980s, linked lists became a cornerstone of computer programming and data structures courses, with textbooks and academic literature extensively covering their implementation and applications.
- Since then, linked lists have remained an essential data structure in computer science and software engineering, used in diverse applications ranging from operating systems and database systems to programming language implementations and algorithm design.

In summary, hash tables and linked lists have left an indelible mark on computer science, transforming the way data is stored, retrieved, and managed. Their impact is felt in virtually every aspect of modern computing, from foundational algorithms to cutting-edge applications, cementing their legacy as timeless pillars of the discipline.

7. Analysis/Design (UML Diagram(s))



- The HashTable class implements the hash table as an array of linked lists, where each element is of type String. It has member functions:
    - constructor
    - Hash function to compute the hash value
    - Put function to add a record to the hash table
    - Contains function to check if an element exists in the hash table
- The Main class implements a dictionary in two ways: as a hash table, and as a linked list. Experiments are defined to test the performance of these in the implementation of a spell checker.

Algorithms:

1. Hashing:
    - Convert the key to lowercase.
    - Get the first character of the lowercase key.
    - Subtract the ASCII value of 'a' from the ASCII value of the first character.
    - Return the result.

2. Adding to the hash table:
    - Compute the hash value for the key using the hash() method.
    - Retrieve the linked list (bucket) at the computed hash value index from the hash table.
    - Add the key to the retrieved bucket.
    - contains(String key):
    - This method checks whether the hash table contains a specific key. It computes the hash value for the key and then checks if the corresponding bucket contains the key.

3. Checking if hash table contains an element:
    - Compute the hash value for the key using the hash() method.
    - Retrieve the linked list (bucket) at the computed hash value index from the hash table.
    - Check if the retrieved bucket contains the key.
    - Return true if the key is found, false otherwise.

8. Time and Space Complexities of Hash Tables and Linked Lists:

8.1. Hash Tables:

8.1.1. Time Complexity:

Average Case: O(1)

- Constant time complexity. Hash tables provide fast average-case performance for insertion, deletion, and search operations, with the time complexity being independent of the number of elements stored in the table.
- This efficiency is achieved by using a hash function to compute the index where each element should be stored, enabling direct access to elements without the need for sequential search.

Worst Case: O(n)

- Linear time complexity. In the worst case, where many keys hash to the same index (resulting in a long collision chain), the performance of hash tables may degrade, requiring sequential search within the collision chain.
- This worst-case scenario occurs infrequently with well-designed hash functions and appropriate table sizes but can impact performance in certain situations.

8.1.2. Space Complexity: O(n)

- The linear space complexity of hash tables depends on the number of elements stored in the table. In the average case, where each element is stored in its own bucket without collisions, the space complexity is proportional to the number of elements (n).
- However, in the worst case, where all elements hash to the same index, additional space may be required to handle collisions, leading to increased memory usage.

8.2. Linked Lists:

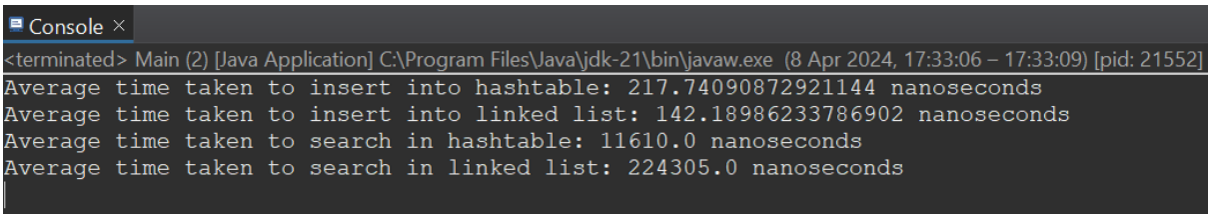
8.2.1. Time Complexity:

- Beginning or End Insertion: O(1)
  Constant time complexity. Insertion at the beginning or end of a linked list involves updating a constant number of pointers and can be performed in constant time.
- Specific Position Insertion: O(n)
  Linear time complexity. Insertion at a specific position in a linked list may require traversing the list to find the insertion point, resulting in time proportional to the length of the list.
- Search: O(n)
  Linear time complexity. Searching in a linked list involves traversing the list linearly from the head node to locate the target element. Since each node may need to be visited, search in linked lists has a time complexity proportional to the length of the list.

8.2.2. Space Complexity: O(n)

- Linear space complexity. The space complexity of linked lists is proportional to the number of elements stored in the list. Each node in the list consumes space for storing the data element and pointers to the next node, contributing to linear memory usage as the list grows in size.

9. RUN:

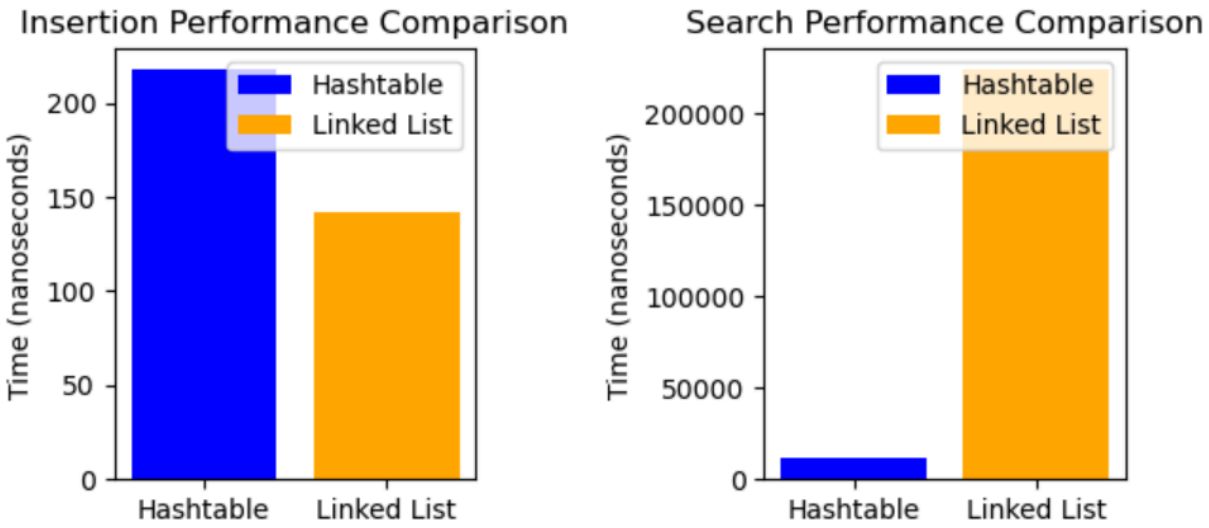


Dataset:

The dictionary was implemented using a dataset with English dictionary words from a to z.
The link of the dataset is given below.
https://www.andrew.cmu.edu/course/15-200/f06/applications/labs/lab2/dict.txt

10. Results:

The words from the dataset are used to insert records into the hash table and a linked list. A list of 100 words is defined, in which 50 are correctly spelled and 50 are misspelt. For each of these words, the spell checker is implemented by checking if the word is in the dictionary. If not, the word is said to be misspelt. Insertion and search times are recorded and averaged for comparison. The times are plotted below for better visualisation.



Observations:

Insertion to hash tables takes slightly longer than to linked lists on average. This is because of the additional time taken to compute the hash value before insertion. On the other hand, for searching, the hash table takes a significantly lesser amount of time than the linked list. This is because for each word, the linked list implementation must search through all the records unless it finds a match. However, in a hash table, based on the computed hash value, the search only needs to be performed for a much smaller subset of the data.

Code Implementation GitHub (link):
https://github.com/RiaAsgekar/COMP47500_Assignment4

Set of Experiments run and results:
https://github.com/RiaAsgekar/COMP47500_Assignment4/blob/main/assignment4/src/Main.java

Video of the Implementation running
Zoom (link & password):
https://ucd-ie.zoom.us/rec/share/OkgtCOItnoBqefgMPZfSqarmIE6W8QFtLFoz7FXyu6RAtpX6M3Lp4wXxGXtN3MoB.uDsyGQUC9JRpNwo3

Passcode: #A+8.UyS

Comments:
Ria gives an overview of our chosen problem domain and an introduction to the topic. Jeevan explains about the implementation of the hash table class and the functions involved. Avantika explains about the implementation of the spell checker using the dataset, the experiments designed, and the results.

References:

1. Tapia-Fernández, Santiago, Daniel García-García, and Pablo García-Hernandez. 2022. Key Concepts, Weakness and Benchmark on Hash Table Data Structures. Algorithms 15, no. 3: 100.
2. Tables, H. Spell Checker.
3. Andrade, G., Teixeira, F., Xavier, C. R., Oliveira, R. S., Rocha, L. C., & Evsukoff, A. G. (2012). Hasch: high performance automatic spell checker for Portuguese texts from the web. Procedia Computer Science, 9, 403-411.
4. Comer, D., & Shen, V. Y. (1982). Hash-bucket search: A fast technique for searching an english spelling dictionary. Software: Practice and Experience, 12(7), 669-682.
5. Randhawa, E. S., & Saroa, E. C. (2014). Study of spell checking techniques and available spell checkers in regional languages: a survey. International Journal For Technological Research In Engineering, 2(3), 148-151.
6. Liu, D., Cui, Z., Xu, S., & Liu, H. (2014, June). An empirical study on the performance of hash table. In 2014 IEEE/ACIS 13th International Conference on Computer and Information Science (ICIS) (pp. 477-484). IEEE.
7. Bellare, M., & Ristenpart, T. (2007, July). Hash functions in the dedicated-key setting: Design choices and MPP transforms. In International Colloquium on Automata, Languages, and Programming (pp. 399-410). Berlin, Heidelberg: Springer Berlin Heidelberg.
8. Ramakrishna, M. V., & Zobel, J. (1997). Performance in practice of string hashing functions. In Database Systems For Advanced Applications' 97 (pp. 215-223).
9. Shuvendu K. Lahiri and Shaz Qadeer. 2006. Verifying properties of well-founded linked lists. SIGPLAN Not. 41, 1 (January 2006), 115–126.
10. https://www.andrew.cmu.edu/course/15-200/f06/applications/labs/lab2/dict.txt
11. https://www.tutorialspoint.com/data_structures_algorithms/hash_data_structure.htm