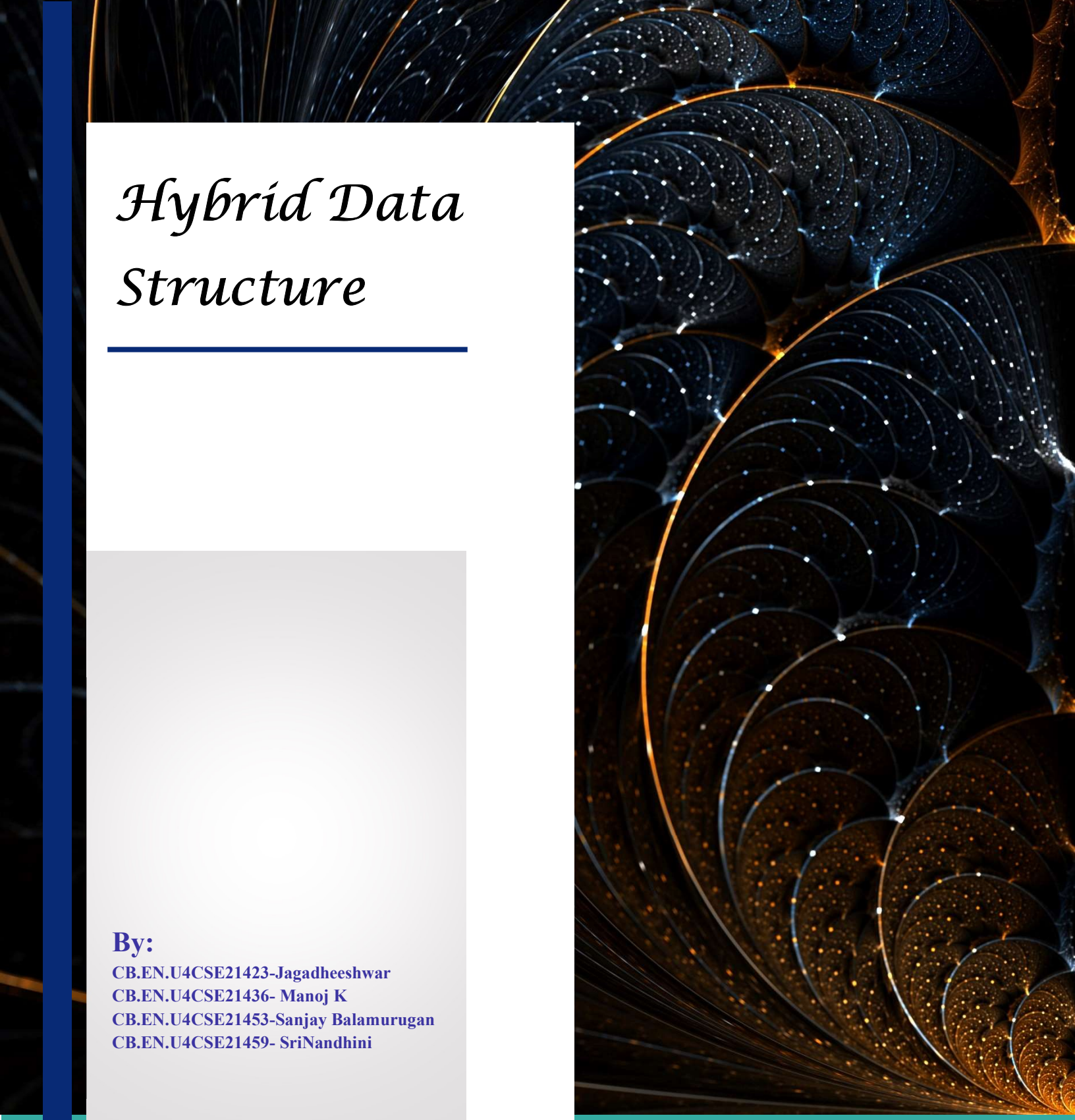
Hybrid Data Structure

By:

CB.EN.U4CSE21423-Jagadheeshwar
CB.EN.U4CSE21436- Manoj K
CB.EN.U4CSE21453-Sanjay Balamurugan
CB.EN.U4CSE21459- SriNandhini



AMRITA
VISHWA VIDYAPEETHAM
DEEMED TO BE UNIVERSITY

Hybrid Data Structure

Introduction

A hybrid data structure is a combination of two or more different data structures. It aims to leverage the strengths of each individual data structure to optimize performance and meet specific requirements. By combining different data structures, a hybrid data structure can provide efficient storage, retrieval, and manipulation of data.

Hybrid data structures are important because they offer a balance between different operations such as insertion, deletion, and searching. They can enhance efficiency by utilizing the best features of multiple data structures. For example, a hybrid data structure might use an array for fast random access and a linked list for efficient insertions and deletions. This flexibility allows for improved performance and scalability in various applications, such as databases, search engines, and computational algorithms.

Overall, hybrid data structures are valuable tools for solving complex problems that require a combination of data manipulation operations. They enable developers to design efficient algorithms and optimize resource utilization, leading to improved performance and enhanced user experiences.

The hybrid data structure we are implementing in this report is a Hashed Linked List. It combines the concepts of hashing and linked lists to provide efficient storage and retrieval of data. The hash function determines the index in the array where each element is stored, while the linked list handles collisions, allowing multiple elements to be stored at the same index. This hybrid approach optimizes both search and insert operations, making it suitable for applications with dynamic data sets and varying access patterns.

OVERVIEW

Linked List

Linked List is a linear data structure, in which elements are not stored at a contiguous location, rather they are linked using pointers. Linked List forms a series of connected nodes, where each node stores the data and the address of the next node.

Node Structure: A node in a linked list typically consists of two components:

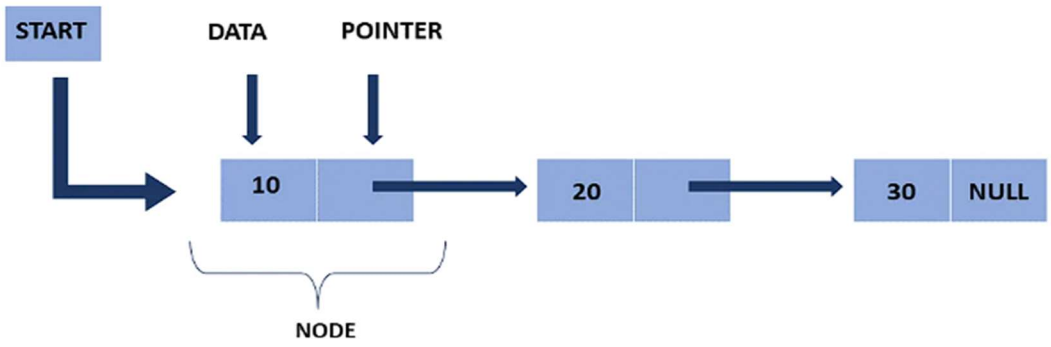
Data: It holds the actual value or data associated with the node.

Next Pointer: It stores the memory address (reference) of the next node in the sequence.

Head and Tail: The linked list is accessed through the head node, which points to the first node in the list. The last node in the list points to NULL or nullptr, indicating the end of the list. This node is known as the tail node.

Why do we need Linked List?

- Dynamic Data structure: The size of memory can be allocated or de-allocated at run time based on the operation insertion or deletion.
- Ease of Insertion/Deletion: The insertion and deletion of elements are simpler than arrays since no elements need to be shifted after insertion and deletion, Just the address needed to be updated.
- Efficient Memory Utilization: As we know Linked List is a dynamic data structure the size increases or decreases as per the requirement so this avoids the wastage of memory.
- Implementation: Various advanced data structures can be implemented using a linked list like a stack, queue, graph, hash maps, etc.



What is hashing?

Hashing is the process of transforming any given key or a string of characters into another value. This is usually represented by a shorter, fixed-length value or key that represents and makes it easier to find or employ the original string.

The most popular use for hashing is the implementation of hash tables. A hash table stores key and value pairs in a list that is accessible through its index. Because key and value pairs are unlimited, the hash function will map the keys to the table size. A hash value then becomes the index for a specific element.

A hash function generates new values according to a mathematical hashing algorithm, known as a hash value or simply a hash. To prevent the conversion of hash back into the original key, a good hash always uses a one-way hashing algorithm.

Hashing is relevant to -- but not limited to -- data indexing and retrieval, digital signatures, cybersecurity and cryptography.

What is hashing used for?

Data retrieval

Hashing uses functions or algorithms to map object data to a representative integer value. A hash can then be used to narrow down searches when locating these items on that object data map.

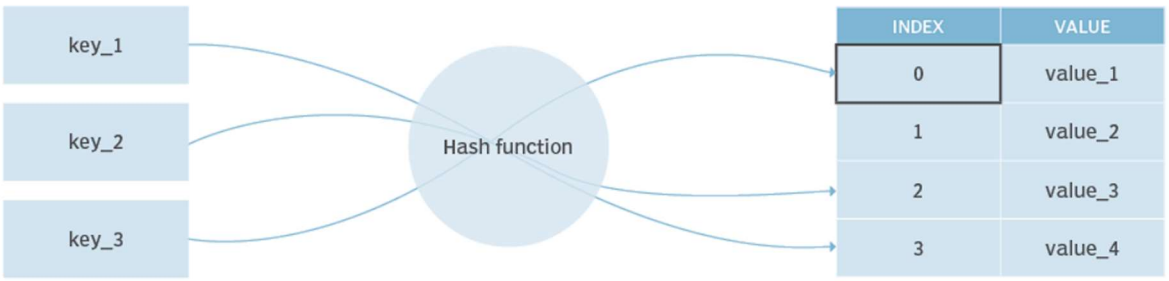
For example, in hash tables, developers store data -- perhaps a customer record -- in the form of key and value pairs. The key helps identify the data and operates as an input to the hashing function, while the hash code or the integer is then mapped to a fixed size.

Hash tables support functions that include the following:

insert (key, value)

get (key)

delete (key)



Hashed linked list

Implementation

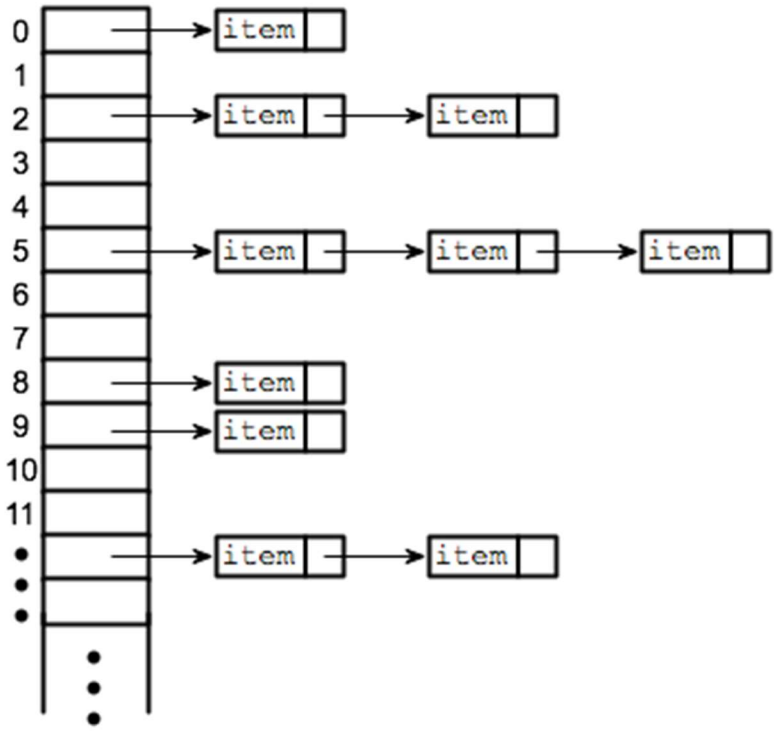
Implementation of Hashed Linked List:

HashedLinkedList (GitHubLink)

We have implemented a hashed linked list using the multiplication method for hashing. This data structure combines the advantages of both a linked list and a hash table.
To begin, we defined a Node class that represents a node in the linked list. Each node has a key attribute to store the value and a next attribute to point to the next node in the list.

Next, we implemented the HashedLinkedList class, which is responsible for managing the hashed linked list. In the constructor (__init__ method), we initialized an array called arr to store our linked list elements. The size of the array is specified by the size parameter passed to the constructor. We also generated a random value x using the formula 1/random.random() to be used in the hash function.

The hash function takes a value (val), the length of the array (arrlen), and the random value (x) as inputs. It employs the multiplication method for hashing. If the input value is of type string, we calculate the sum of the ASCII values of its characters. Otherwise, we use the value as is. Then, we multiply the sum by x, take the fractional part, multiply it by arrlen, and convert it to an integer. The resulting value is the hashed index for the input.



In the insert method, we insert an item into the linked list. We calculate the hash index for the item using the hash function. If the slot at the calculated index is empty (None), we create a new node with the item as the key and place it in that slot. If the slot is not empty, we traverse the linked list until we find an empty slot and insert the new node there.

The delete method is used to remove an item from the linked list. It calculates the hash index for the item, then iterates through the linked list at that index. If the item is found, we adjust the previous node's next pointer to skip the current node, effectively removing it from the linked list.
The traverse method allows us to retrieve all the items from the linked list. It iterates through each slot in the array and traverses the linked list at each slot, appending the keys to a list. Finally, it returns the list of items.
To visualize the contents of the hashed linked list, we implemented the print_list method. It displays the index number and the items in each slot of the array, along with the linked list representation. This method helps us understand the structure of the hashed linked list.
In the example usage, we create an instance of the HashedLinkedList class with a size of 10. Then we insert several items into the hashed linked list using the insert method. After that, we call the print_list method to display the contents of the hashed linked list. Finally, we call the traverse method to retrieve all the items from the list and print them.
Overall, our hashed linked list implementation provides us with the ability to efficiently insert, delete, traverse, and print the contents of the linked list. The multiplication method for hashing ensures a good distribution of items across the available slots in the array, leading to efficient retrieval and deletion operations.

Benefits of Using Hashed Linked List

Efficient Search and Retrieval: The primary advantage of a hashed linked list is its ability to provide efficient search and retrieval operations. By using a hash function to map keys to specific indices in the underlying array, we can quickly locate the desired elements without having to iterate through the entire list. This makes hashed linked lists ideal for applications that require fast access to specific elements.

1. Dynamic Size: Unlike traditional arrays, which have a fixed size, hashed linked lists can dynamically resize and adjust their capacity based on the number of elements stored. This dynamic resizing allows for efficient memory utilization and eliminates the need to allocate a large contiguous block of memory upfront.

2. Collision Handling: Hash functions may produce the same index for different keys, resulting in collisions. Hashed linked lists provide a means to handle collisions by using a linked list at each index. When a collision occurs, the new element is simply appended to the linked list at the corresponding index. This chaining approach ensures that all elements can be stored without loss of data.

3. Flexibility: Hashed linked lists offer flexibility in terms of the types of data that can be stored. They can accommodate a wide range of data types, including integers, strings, objects, and more. Additionally, the size of the underlying array can be adjusted as needed, allowing for scalability in managing larger datasets.

4. Memory Efficiency: Hashed linked lists consume memory efficiently, especially when the number of elements is significantly smaller than the total number of available indices. Unlike arrays, which reserve memory for the entire range of indices, hashed linked lists only allocate memory for the occupied slots and the necessary linked list nodes.

5. Easy Insertion and Deletion: Insertion and deletion operations in a hashed linked list are relatively straightforward and efficient. Inserting a new element involves calculating the hash index and appending the element to the corresponding linked list. Deleting an element requires locating the element in the linked list and adjusting the pointers to remove it. Both operations have an average time complexity of O(1), making them efficient for most practical use cases.

6. Overall, hashed linked lists combine the benefits of constant-time search and retrieval with the flexibility of linked lists, making them a suitable data structure for scenarios that prioritize fast element access and dynamic resizing.

Some Practical Applications of Hashed Linked List

1. Hash Tables: Hash tables are one of the most common applications of hashed linked lists. A hash table uses a hashed linked list to store key-value pairs. The hash function maps keys to indices in the underlying array, and each index contains a linked list of key-value nodes. This data structure allows for efficient key-based lookup, insertion, and deletion operations.

2. Caches: Hashed linked lists are used in caching mechanisms to store frequently accessed data. The cache can be implemented as a fixed-size hash table with a hashed linked list at each index. This allows for fast cache lookups and efficient replacement of least-recently-used items.

3. Symbol Tables: Symbol tables, used in programming languages and compilers, store symbols (e.g., variable names, function names) and their associated information (e.g., memory locations, data types). A hashed linked list can be employed to implement a symbol table, enabling quick symbol lookup and efficient management of symbol information.

4. Databases: Hashed linked lists can be utilized in database systems for indexing and searching data efficiently. Hash indexes use a hashed linked list to store the keys and pointers to the corresponding data records. This enables fast data retrieval based on key values.

5. Spell Checkers: In spell-checking applications, a hashed linked list can be employed to store a dictionary of words. The hash function maps words to indices in the linked list, and each index contains a chain of words with the same hash value. This allows for efficient word lookups and suggestions during spell-checking operations.

6. File Systems: Hashed linked lists can be used in file systems to manage file directories efficiently. Each directory entry can be stored as a node in a hashed linked list, with the hash function mapping filenames to indices. This enables fast file lookup and retrieval in large directory structures.

7. Caching Web Content: Hashed linked lists can be utilized in web caching systems to store web content such as web pages, images, and resources. The hash function maps the URLs to indices in the linked list, allowing for efficient retrieval of cached content based on URL keys.

Performance Analysis:

The time and space complexities of each function in the provided hashed linked list implementation are as follows:

hash function:

Time complexity: The time complexity of this function depends on the length of the input string (val). If the string has a length of n, the function performs a loop over the characters in the string, resulting in O(n) time complexity.
Space complexity: The space complexity of this function is O(1) because it only uses a constant amount of space to store intermediate variables.

Node class __init__ method:

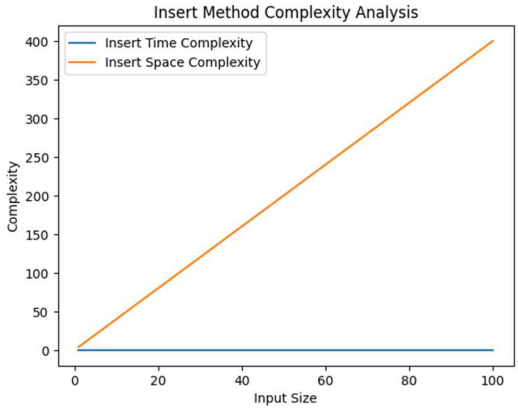
Time complexity: This method has a time complexity of O(1) because it performs a constant number of operations regardless of the input size.
Space complexity: The space complexity of this method is O(1) as it creates only two attributes (key and next) for each node.

HashedLinkedList class __init__ method:

Time complexity: This method has a time complexity of O(size) because it iterates over the size variable to initialize the arr list.
Space complexity: The space complexity of this method is O(size) as it creates an array (arr) of size size to store the linked list elements.

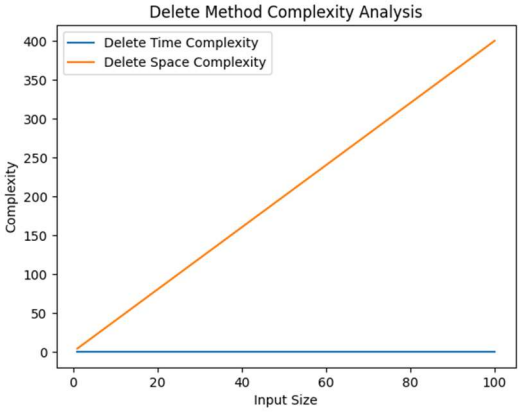
insert method:

Time complexity: In the worst case, when there are no collisions, the time complexity of the insert method is O(1) because it directly inserts the element at the calculated index. However, in the case of collisions, where elements need to be appended to the linked list at the index, the time complexity can increase up to O(n), where n is the number of collisions at the index.
Space complexity: The space complexity of this method is O(1) because it only uses a constant amount of space to store the new node.



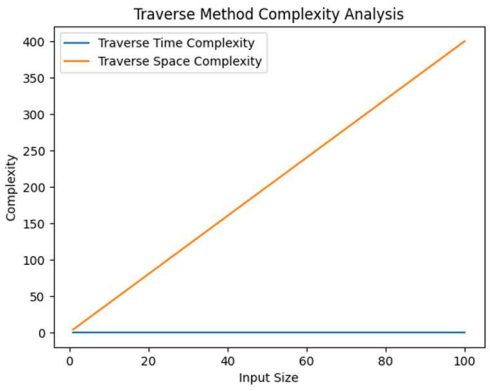
delete method:

Time complexity: The time complexity of the delete method is O(1) on average because it directly finds the element at the calculated index and removes it. In the worst case, when there are collisions and the element is at the end of the linked list, the time complexity can increase up to O(n), where n is the length of the linked list at the index.
Space complexity: The space complexity of this method is O(1) because it only uses a constant amount of space to store variables for node traversal and deletion.



traverse method:

Time complexity: The time complexity of the traverse method is O(size + n), where size is the size of the array and n is the total number of elements in the linked lists. It iterates over each index in the array and each node in the linked list to retrieve the elements.
Space complexity: The space complexity of this method is O(n) because it creates a list (items) to store the elements from the linked lists.



Code: [TimeSpaceComplexityPlotCode](#) (GitHUB)

Implementation Using Arrays

Implementation GitHUB link:
HashMapUsingArray

Analysis of Space and Time complexities:

insert function:

Time Complexity: O(1) on average. Since the hashing function distributes the keys uniformly across the buckets, the average time complexity for inserting a key is constant.
Space Complexity: O(1). The space required for storing the key is constant as it depends on the number of elements inserted, not the size of the hashmap.

delete function:

Time Complexity: O(k), where k is the number of keys stored in the bucket where the key is hashed. In the worst case, when all the keys are hashed to the same bucket, the time complexity becomes O(n), where n is the total number of keys inserted.
Space Complexity: O(1). The space required is constant as it depends on the number of elements inserted, not the size of the hashmap.

traverse function:

Time Complexity: O(n), where n is the total number of keys inserted. The function iterates over all the buckets and keys to collect and return all the keys.
Space Complexity: O(n). The space required is proportional to the number of keys stored in the hashmap as all the keys are collected and stored in the items list.

In general, the insert and delete operations have a time complexity of O(1) on average, making them efficient for most cases. However, in the worst case, the time complexity can become O(n) when all the keys are hashed to the same bucket. The traverse operation has a time complexity of O(n) as it needs to iterate over all the keys. The space complexity for all functions is proportional to the number of keys stored in the hashmap.

Comprison of HashedLinkedList and HashedArray

1. Time Complexity:
    - HashedLinkedList:
        - Insertion: O(1) on average, as the hash function distributes the keys uniformly across the linked list buckets, resulting in constant time insertion.

        - Deletion: O(1) on average, as it only requires traversing the linked list at the hashed index.
        - Traversal: O(n), where n is the total number of elements, as it needs to traverse all the elements in the linked list.
    - HashMap:
        - Insertion: O(1) on average, as the hash function distributes the keys uniformly across the hashmap buckets, resulting in constant time insertion.
        - Deletion: O(k) in the worst case, where k is the number of keys stored in the bucket where the key is hashed. In the average case, it is O(1).
        - Traversal: O(n), where n is the total number of keys, as it needs to iterate over all the keys in the hashmap.
2. Space Complexity:
    - HashedLinkedList:
        - Space complexity is dependent on the number of elements inserted, as each element requires a node object and some additional pointers. Therefore, the space complexity is O(n), where n is the total number of elements.
    - HashMap:
        - Space complexity is also dependent on the number of elements inserted. However, the space required for each element is smaller compared to the HashedLinkedList, as the keys are stored directly in the hashmap buckets without the need for additional node objects. Therefore, the space complexity is typically lower than O(n), depending on the distribution of keys among buckets.

In summary, both implementations offer efficient average case time complexity for insertion and deletion operations, with O(1) for the HashedLinkedList and O(1) or O(k) for the HashMap, depending on the collision resolution strategy. However, the HashMap implementation may provide a more optimized space complexity compared to the HashedLinkedList, depending on the distribution of keys among buckets.

Conclusion:

In terms of time complexity, both implementations have O(1) average case complexity for insertion and deletion. However, the Hashed Linked List implementation may have increased time complexity in the case of collisions, while the implementation using arrays maintains O(1) complexity for these operations.

In terms of space complexity, both implementations have similar characteristics. They both use a constant amount of space for individual nodes or keys. The Hashed Linked List implementation requires additional space for the array and linked lists, while the implementation using arrays requires space proportional to the number of keys stored.

Overall, the choice between the two implementations depends on the specific requirements and trade-offs of the application. The Hashed Linked List may be more suitable when handling collisions is important and when the number of elements varies dynamically. The implementation using arrays may be preferred when a simpler structure is sufficient and when memory efficiency is a concern.

References:

https://www.simplilearn.com/tutorials/data-structure-tutorial/linked-list-in-data-structure#:~:text=A%20linked%20list%20is%20the,reference%20to%20the%20next%20node.

https://www.geeksforgeeks.org/what-is-linked-list//

https://www.techtarget.com/searchdatamanagement/definition/hashing#:~:text=Hashing%20is%20the%20process%20of,the%20implementation%20of%20hash%20tables.