Hybrid Data Structure

Hashed Linked List

By:
CB.EN.U4CSE21423-Jagadheeshwar
CB.EN.U4CSE21436- Manoj Kumaran S
CB.EN.U4CSE21453-Sanjay Balamurugan
CB.EN.U4CSE21459- SriNandhini

AMRITA
VISHWA VIDYAPEETHAM
DEEMED TO BE UNIVERSITY

Introduction

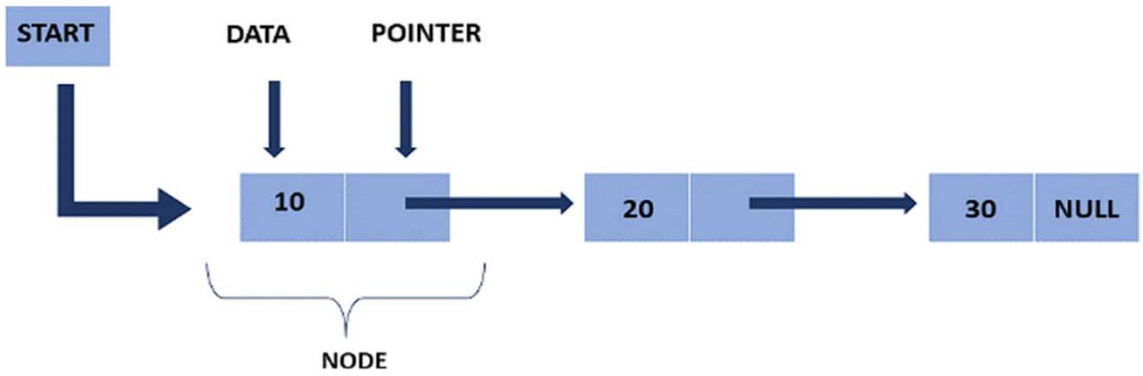
A hybrid data structure is a combination of two or more different data structures to create a single structure that benefits from the strengths of each individual structure. By combining different data structures, hybrid data structures aim to achieve improved performance, efficiency, and functionality for specific use cases.

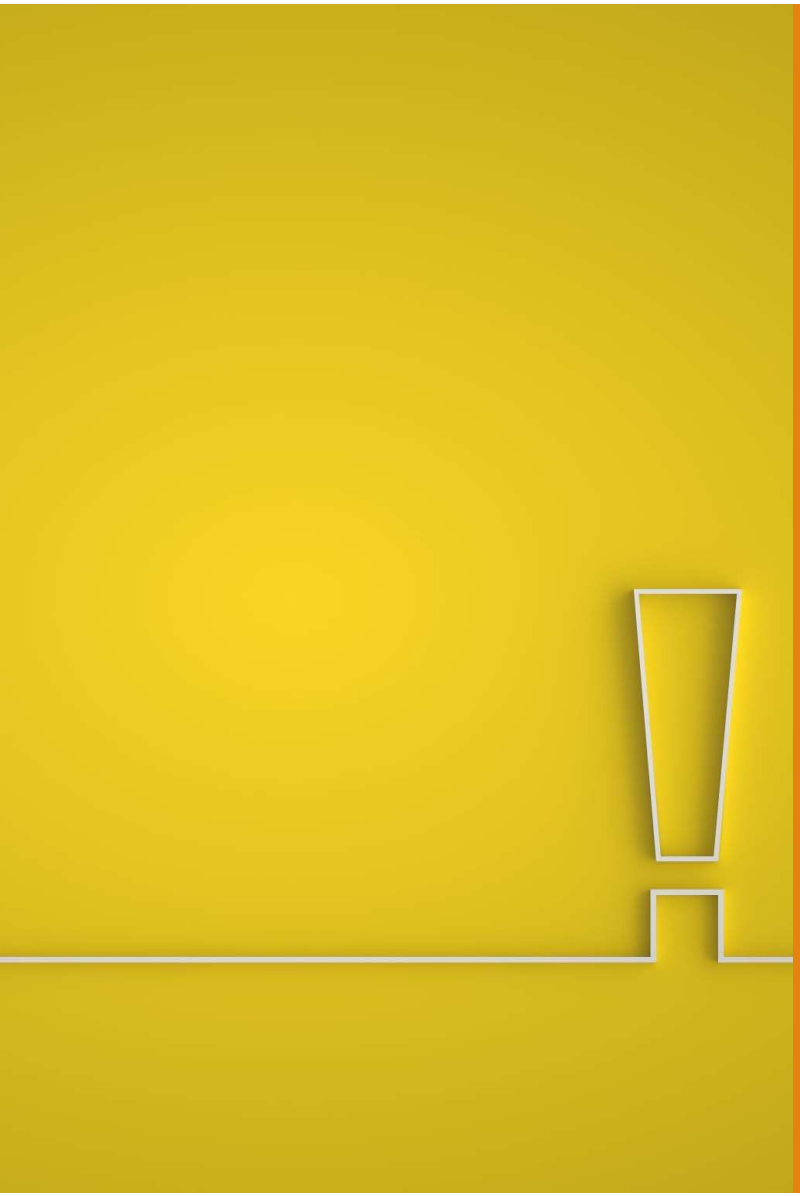
Hybrid data structures often arise when existing data structures do not fully meet the requirements of a particular problem or when a combination of structures can provide better performance characteristics. They can be designed to optimize different operations such as insertion, deletion, search, or traversal based on the specific needs of the application.

Linked List

A linked list is a linear data structure in which each element is a separate object. Each element, called a node, contains data and a reference (or pointer) to the next node in the list.

Linked lists are a dynamic data structure, which means that they can grow and shrink as needed. This makes them a good choice for applications where the size of the data set is not known in advance, or where the data set is frequently changing.





Hashing

Hashing is a technique for converting data into a smaller, fixed-length value. This value is called a hash code or simply a hash.

Hashes are used for a variety of purposes, including:

Data storage: Hashes can be used to store data in a more efficient way. For example, a hash table is a data structure that uses hashing to store data in a way that allows for fast lookups.

Data security: Hashes can be used to protect data from unauthorized access. For example, passwords are often hashed before they are stored, so that even if an attacker gains access to the database, they will not be able to read the passwords.

Data integrity: Hashes can be used to verify the integrity of data. For example, a checksum is a hash value that is calculated for a piece of data. If the data is changed, the checksum will also change. This can be used to detect errors or malicious tampering with data.

```
PS D:\Amrita\Sem_4\DSA\Python> & C:/Users/sanja/AppData/Local/Programs/Python/Python310
------------------------------------------------
Index 0: -> 25  ->  4  ->  2  ->  None
Index 1: -> 8  ->  6  ->  None
Index 2: -> 10  ->  12  ->  None
Index 3: -> 16  ->  None
Index 4: -> 20  ->  18  ->  None
Index 5: -> 1  ->  3  ->  22  ->  None
Index 6: -> 5  ->  7  ->  None
Index 7: -> 9  ->  11  ->  None
Index 8: -> 15  ->  13  ->  None
Index 9: -> 17  ->  None
------------------------------------------------
[25, 4, 2, 8, 6, 10, 12, 20, 18, 1, 3, 22, 5, 7, 9, 11, 15, 13, 17]
```

Output

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.

- Collision Handling
- Flexibility
- Dynamic Size
- Memory Efficiency
- Easy Insertion and Deletion

Implementation

The implementation of a hashed linked list using the multiplication method for hashing combines the benefits of linked lists and hash tables.

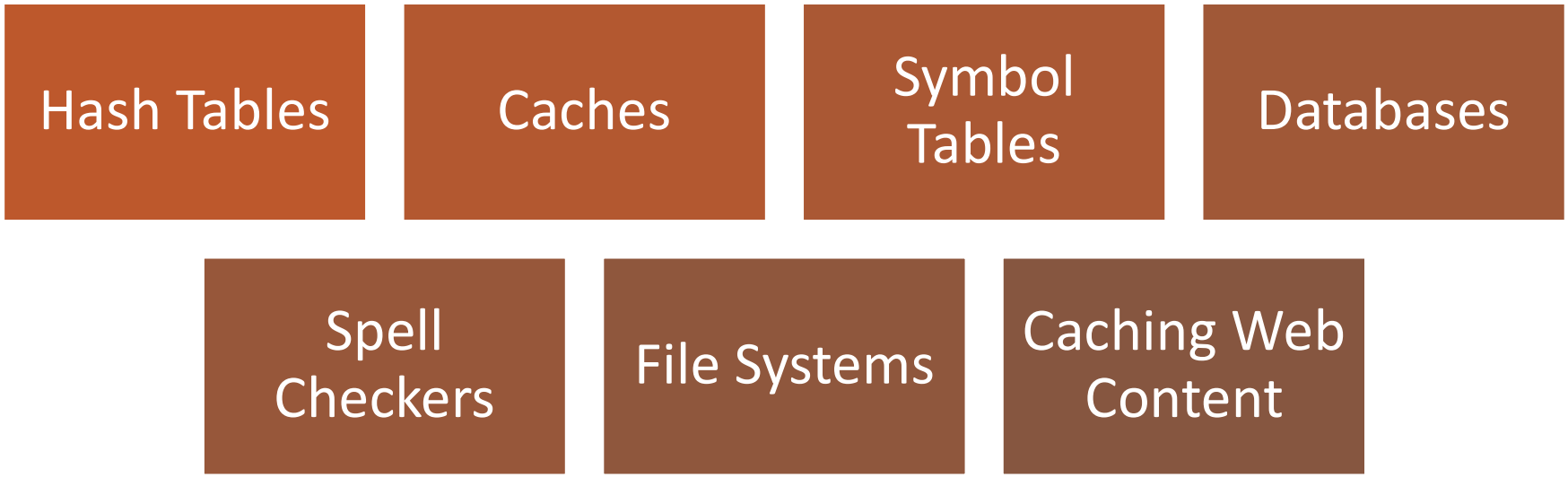
It utilizes a Node class to represent nodes with key attributes and a next pointer.

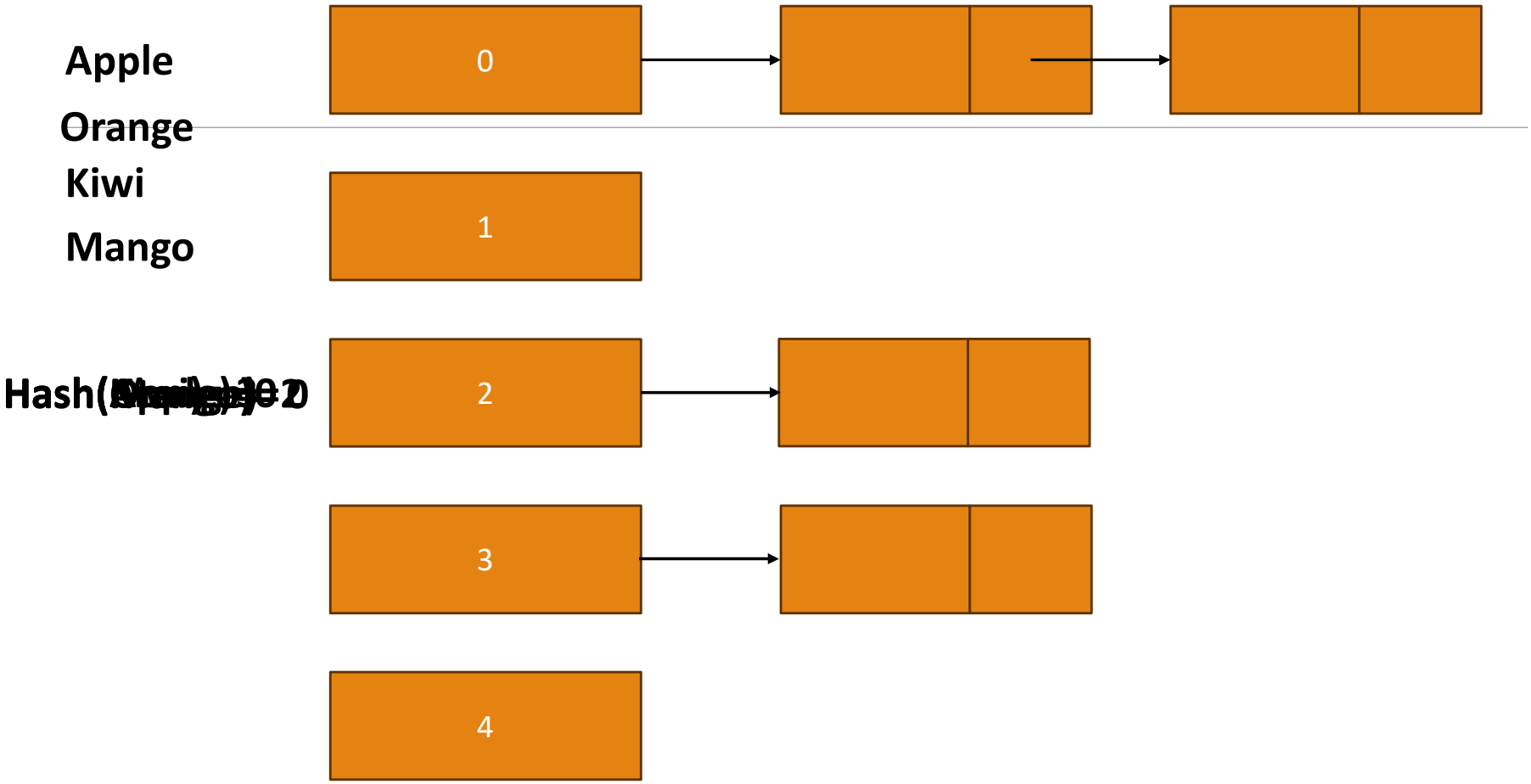
The HashedLinkedList class manages the structure, using a hashed index calculated by the multiplication method.

The data structure enables efficient insertion, deletion, traversal, and retrieval operations. The approach provides balanced item distribution, minimizes collisions, and ensures effective memory utilization.

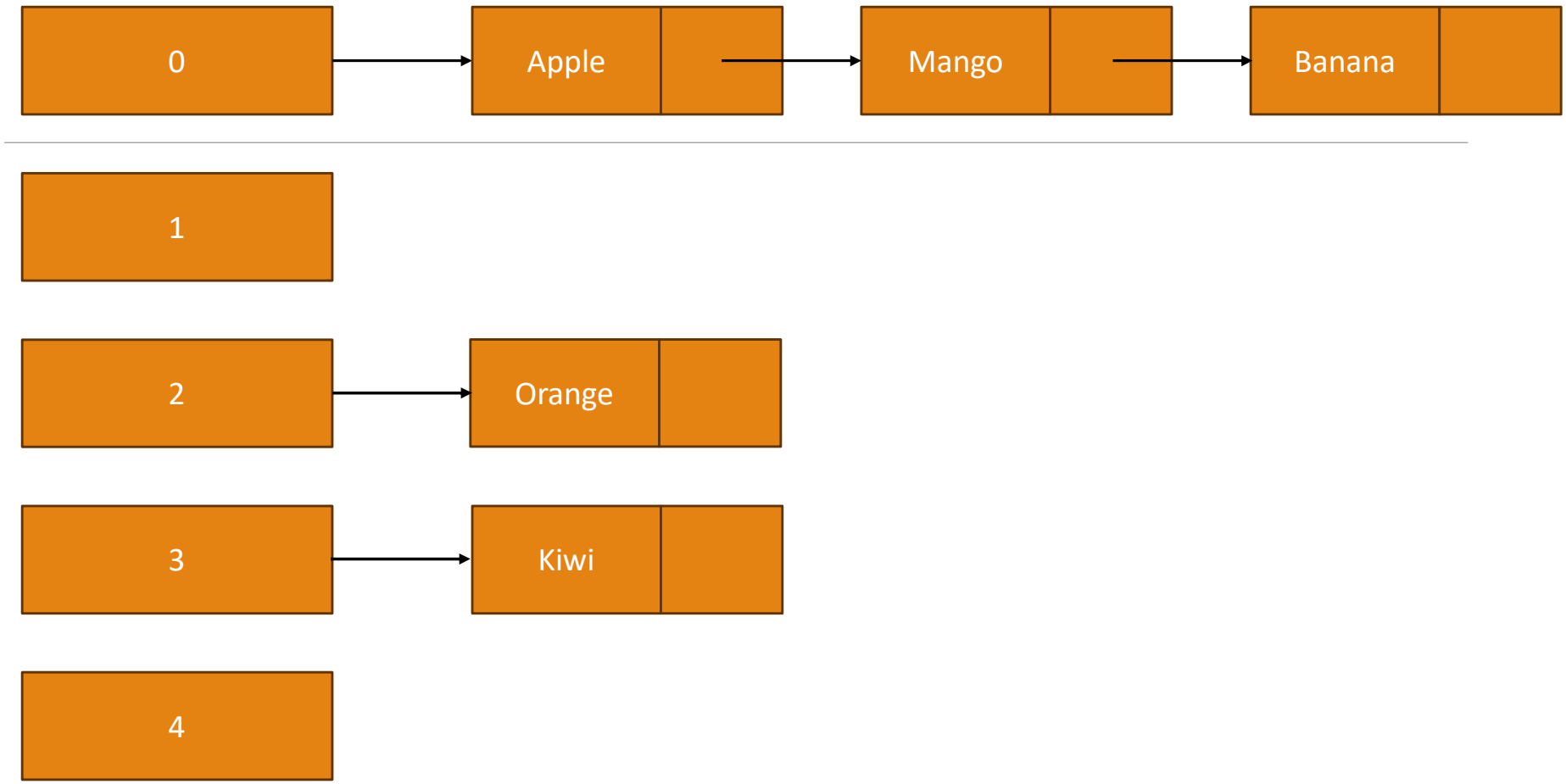
Overall, the implementation offers a robust and efficient data structure for managing data with improved performance characteristics.

Some Practical Applications of Hashed Linked List




Apple
Orange
Kiwi
Mango
0
1
2
3
4

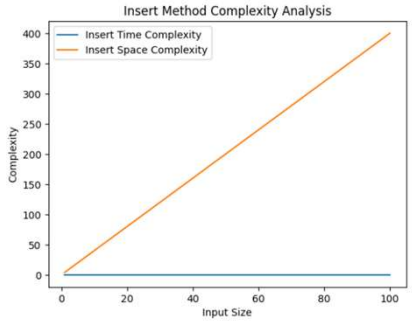
Delete(Mango)

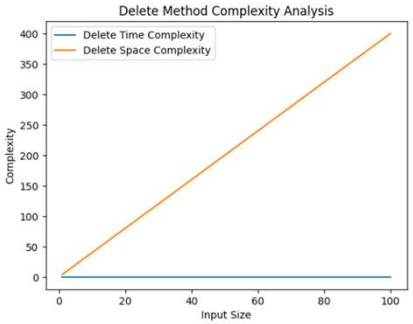


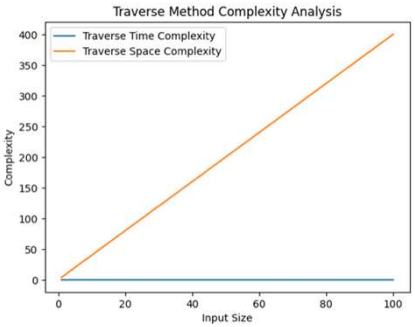
Performance Analysis

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

	Time Complexity	Space Complexity
hash function:	O(N)	O(1)
Node class __init__ method	O(1)	O(1)
HashedLinkedList class __init__ method	O(size)	O(size)







Performance Analysis

	Time Complexity	Space Complexity
Insert Method	O(1)	O(1)
Delete Method	O(1)	O(1)
Traverse method	O(size + n)	O(n)

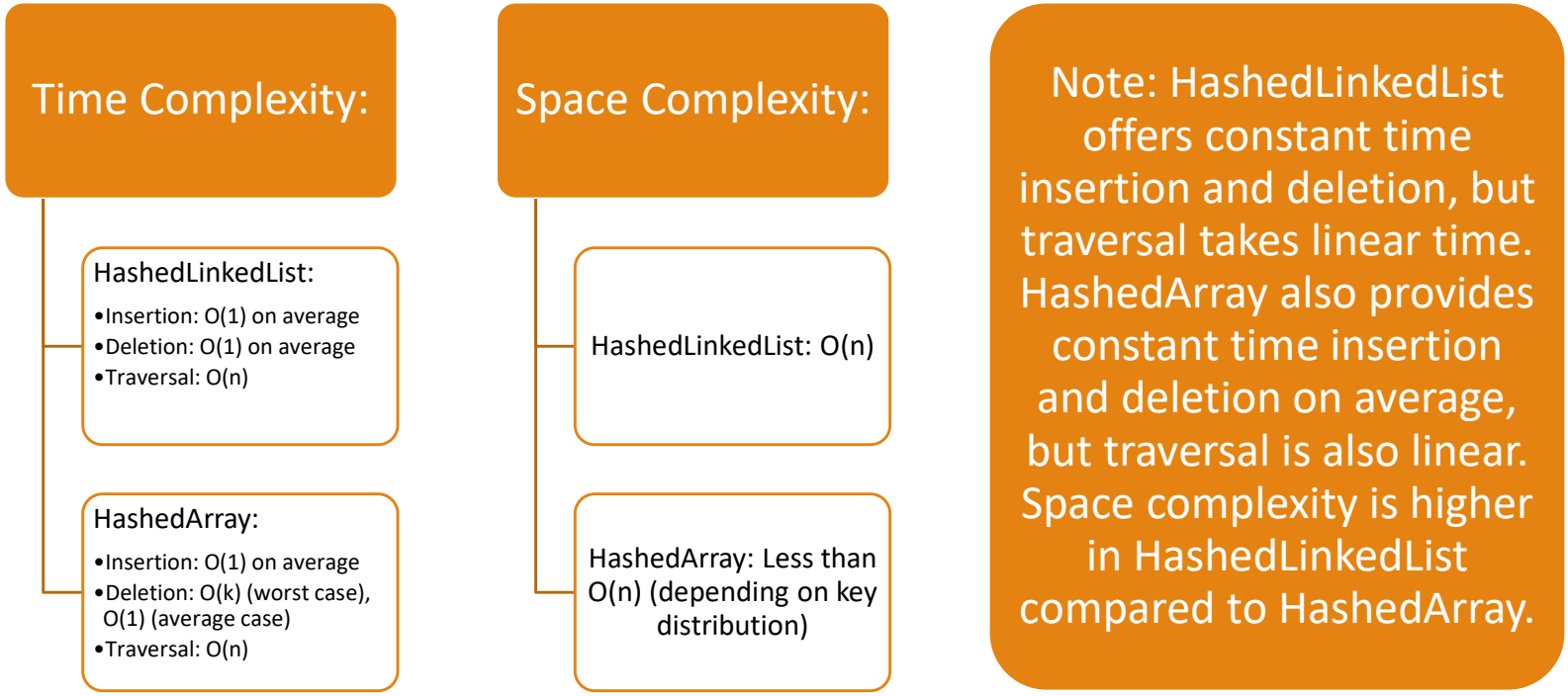
Analysis of Space and Time complexities:

	Time Complexity	Space Complexity
Insert Method	O(1)	O(1)
Delete Method	O(k)	O(1)
Traverse method	O(n)	O(n)

Comprison of HashedLinkedList and HashedArray

	Time Complexity			Space Complexity
	Insertion	Deletion	traversal	
HashedLinkedList:	O(1)	O(1)	O(n)	O(n)
HashMap:	O(1)	O(k)	O(n)	O(n)

Comparison of HashedLinkedList and HashedArray



Both HashedLinkedList and HashedArray offer efficient insertion and deletion operations with average-case constant time complexity.

HashedLinkedList has a linear time complexity for traversal, while HashedArray has linear traversal as well.

HashedLinkedList has a space complexity of O(n) due to the node objects, while HashedArray may have a lower space complexity depending on key distribution.

Considerations:

- Use HashedLinkedList when constant time insertion and deletion are crucial, and frequent traversals are not required.
- Use HashedArray when constant time insertion and deletion are important, and efficient space utilization is desired.

Note: Choose the data structure based on specific requirements, such as time complexity priorities (insertion, deletion, traversal), space limitations, and key distribution characteristics.

Conclusion

- Handling Collisions: Hashed Linked List efficiently handles collisions, while Hashed Array may require additional strategies.
- Dynamic Element Count: Hashed Linked List is more flexible in handling varying numbers of elements.
- Memory Efficiency: Hashed Array may be preferred when memory efficiency is a concern.

Use Cases:

- Hashed Linked List: Suitable for collision-prone scenarios, dynamic element count, and flexibility.
- Hashed Array: Suitable for simpler structures and memory efficiency requirements.

Note: Both implementations offer O(1) average case time complexity for insertion and deletion. The choice depends on collision handling, dynamic element count, and memory efficiency considerations.

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.