DELIVERABLE 3

MSCS 532 Project

Brendan Yeong
Student ID: 005025797

Optimization of Data Structures: Analysis and Optimization Strategies

1. Analyzing Initial Implementation

Let's begin by reviewing the time complexity, space efficiency, and scalability of the initial data structures used in the vehicle registration system:

- Dictionary (for Vehicle Registrations):
    - Time Complexity:
        - Insertion, Deletion, Lookup: $O(1)$ on average.
    - Space Complexity: The space complexity grows linearly with the number of vehicle registrations stored.
- Priority Queue (Heap for Expiration Dates):
    - Time Complexity:
        - Insertion: $O(\log n)$.
        - Get next expiration: $O(1)$.
        - Remove: $O(log\ n)$.
    - Space Complexity: The space complexity is $O(n)$ for storing expiration dates and associated vehicle records.
- AVL Tree (for Driver's License Search):
    - Time Complexity:
        - Insertion, Deletion, Search: $O(log\ n)$, where n is the number of nodes.
    - Space Complexity: Each node in the AVL tree stores a driver's license number, associated vehicles, pointers to left and right children, and height. The space complexity is $O(n)$.
- Trie (for License Plate Prefix Search):

- Time Complexity:
  - Insertion: $O(k)$, where k is the length of the license plate.
  - Search: $O(k)$, where k is the length of the prefix.
  - Deletion: $O(k)$.
- Space Complexity: The space complexity grows linearly with the number of unique nodes (characters in license plates). Common prefixes reduce space, but redundant nodes might still exist.

Identifying Bottlenecks and Inefficiencies

- Trie:
  - Space Inefficiency: The space required for a Trie can grow significantly with the number of license plates, especially with common prefixes resulting in redundant nodes (Goodrich & Tamassia, 2014).
  - Insertion and Deletion Overhead: Inserting or deleting license plates takes linear time based on the plate's length, which might lead to performance degradation as the number of plates grows (Weiss, 2012).

To improve some of the bottlenecks and inefficiencies:

Trie: Path Compression in Trie

Path compression minimizes redundant nodes by combining nodes that form a straight chain in the Trie. This optimization reduces memory consumption and speeds up traversal operations (Weiss, 2012).

- Path Compression:

    - For license plates, the first 3 characters are stored in a traditional Trie node structure.
    - After 3 characters, the remaining part of the license plate is stored in the hash_map_suffix, which reduces the depth of the Trie and makes it more memory efficient. This avoids deep node chains for longer license plates.
- Hybrid Trie-Hash Map:
    - For longer license plates, instead of storing every character individually, the suffix of the plate is stored in a hash map (hash_map_suffix) once the first 3 characters are processed. This hybrid approach reduces memory usage and speeds up both insertions and lookups.
- Search Operation:
    - The search method first navigates through the Trie nodes for the first few characters of the prefix.
    - If the search reaches the hash map level (after the first 3 characters), it searches for matching suffixes in the hash map.

```python
class CompressedTrieNode:  2 usages  Brendan Yeong *
    def __init__(self):  Brendan Yeong *
        self.children = {}  # Child nodes (for each character)
        self.is_end_of_plate = False  # Indicates if a complete license plate ends here
        self.hash_map_suffix = {}  # Hash map for handling long suffixes

class CompressedTrie:  3 usages  Brendan Yeong +1 *
    def __init__(self):  Brendan Yeong *
        self.root = CompressedTrieNode()

    # Insert a license plate into the compressed Trie.
    # The first 3 characters are stored in the Trie, and the remaining are stored in a hash map.
    def insert(self, license_plate):  14 usages (2 dynamic)  Brendan Yeong *
        node = self.root
        i = 0
        while i < len(license_plate):
            if i < 3:  # Use the Trie for the first 3 characters
                current_char = license_plate[i]
                if current_char not in node.children:
                    node.children[current_char] = CompressedTrieNode()
                node = node.children[current_char]
                i += 1
            else:
                # Use the hash map for the remaining part of the license plate
                suffix = license_plate[i:]
                if suffix not in node.hash_map_suffix:
                    node.hash_map_suffix[suffix] = True
                break
        node.is_end_of_plate = i == len(license_plate)  # Set end of plate only if we finished the plate
```

```python
    # Search for license plates starting with the given prefix.
    def search(self, prefix):  14 usages  Brendan Yeong *
        node = self.root
        i = 0
        result = []

        # Traverse the Trie for the first 3 characters
        while i < len(prefix) and i < 3:
            current_char = prefix[i]
            if current_char in node.children:
                node = node.children[current_char]
                i += 1
            else:
                return result  # Prefix not found

        # If the prefix length exceeds 3, check the hash map for suffixes
        if i == 3:
            for suffix in node.hash_map_suffix:
                if suffix.startswith(prefix[3:]):
                    result.append(prefix[:3] + suffix)

        # Perform DFS to collect all license plates starting from this node
        self._collect_plates(node, prefix[:i], result)

        return result
```

Scaling for large datasets:

Since my dictionary implementation is simply and easily maintainable to implement any of more complex dataset is just to modified the add method to add in more attribute:

```
def add_vehicle(self, license_plate, vehicle:Vehicle, owner:Owner, registration_date, expiration_date):  5 usages (1 dynamic)
    self.registrations[license_plate] = {
        'owner': {
            'first_name': owner.first_name,
            'last_name': owner.last_name,
            'license_number': owner.license_number,
        },
        'vehicle': {
            'make': vehicle.make,
            'model': vehicle.model,
            'year': vehicle.year,
            'color': vehicle.color,
            'classification': vehicle.classification,
            'vin_number': vehicle.vin_number,
        },
        'registration_date': registration_date,
        'expiration_date': expiration_date
    }
    # Clear the cache after adding a new vehicle to ensure cache consistency
```

Currently it is already complex with it being able to handle objects and adding correctly and accessing them as well for updates or deletions.

By updating my Trie I was able to make it handle more complex datasets. By optimized Trie with path compression and a hybrid Trie-Hash Map can be further adapted to handle custom license plates such as "ABCDEF" and "ABC1EF", which may contain combinations of letters and numbers in various formats.

```
class CompressedTrieNode:  2 usages  Brendan Yeong *
    def __init__(self):  Brendan Yeong *
        self.children = {}  # Child nodes (for each character)
        self.is_end_of_plate = False  # Indicates if a complete license plate ends here
        self.hash_map_suffix = {}  # Hash map for handling long suffixes
```

1. Handling Custom Plates in Trie Nodes:

- The core functionality of the Trie already supports handling both letters and numbers since it operates on individual characters of a string. License plates like "ABCDEF" and "ABC1EF" would be processed character by character, and since the Trie nodes handle any type of character (whether alphabetic or numeric), they can easily support custom formats.

2. Handling Custom Plates in the Hash Map (Suffixes):
    - The hash map stores the suffix (i.e., the part of the plate after the first 3 characters). Custom plates like "ABC1EF" or "ABCDEF" will have the suffix stored in the hash map. The insertion and search operations will still behave correctly because the Trie will handle the first 3 characters and the hash map will store the rest, regardless of whether the characters are letters or numbers.

Advanced Testing and Validation:

Test Plan Breakdown:

1. Test: Insertion and Search:
    - This test inserts a few plates into the Trie and verifies that they can be correctly searched.
2. Test: Custom Plates:
    - This test checks how the Trie handles custom license plates with mixed alphanumeric characters.
3. Test: Deletion:
    - This ensures that the deletion operation works correctly by inserting a plate, deleting it, and verifying that it's no longer found.

4. Test: Edge Cases:
    - Tests edge cases like inserting an empty string and searching for non-existent plates.
5. Test: Large Dataset:
    - This performance test inserts large number of plates (e.g., 100,000) into the Trie and measures how long it takes. It also performs 100 searches to check how well the Trie scales with large number of plates.
6. Test: Prefix Collision Stress Test:
    - This stress test inserts 1,000 plates with similar prefixes (e.g., "ABC001", "ABC002", etc.) to verify how well the Trie handles deep prefix collisions.
7. Test: Memory usage:
    - This performance test inserts large number of plates (e.g., 100,000) into the Trie and measures much memory usage it takes to check how well the Trie scales with large number of plates.

Stress Testing and Scalability:

- Stress Testing: The test cases for inserting large datasets (100,000 or more plates) and handling deep prefix collisions ensure that the Trie can handle extreme conditions. These tests simulate real-world scenarios where many license plates share similar prefixes or when many plates are managed in the system.
- Scalability: By running tests on progressively larger datasets (from a few plates to millions), you can measure the memory usage and time efficiency of the Trie as the dataset grows. This will validate whether the optimized Trie is scalable for large-scale applications.

Final evaluation and performance analysis:

Comparison:

Let’s assume you’re inserting the following license plates:

- "ABC123"
- "ABC456"
- "XYZ789"
- "ABX789"
- "DEF123"

In a Traditional Trie:

1. The first three characters "ABC" will have separate nodes for A, B, and C.
2. The same is true for "XYZ" and "DEF".
3. For every license plate, each character in the plate creates or traverses an individual node, resulting in a deep tree with many redundant nodes (especially for the plates starting with "ABC").

Space Complexity: Every character in every plate requires a separate node, leading to inefficient memory usage for large datasets.

In the Optimized Trie:

1. For "ABC123" and "ABC456", the first 3 characters are stored in shared nodes (because they are the same), and the remaining part ("123" and "456") is stored in the hash map.
2. For "XYZ789", only the first 3 characters ("XYZ") are stored in the Trie, and "789" is stored in the hash map.

3. This results in far fewer nodes being created for common prefixes, reducing memory usage.

Space Complexity: Only the first few characters are stored as nodes in the Trie, with the remaining suffixes stored in a more efficient data structure (the hash map), reducing the overall memory footprint.

Time Complexity Breakdown and comparison:

- Traditional Trie:
    - Insertion/Deletion: $O(k)$, where k is the length of the license plate.
    - Search: $O(k)$.
- Optimized Trie:
    - Insertion/Deletion:
        - First 3 characters: $O(3)$ (constant time).
        - Suffix in hash map: $O(1)$
    - Search:
        - First 3 characters: $O(3)$
        - Suffix in hash map: $O(1)$.

For long license plates (e.g., 10 or more characters), the optimized Trie saves significant time by offloading most of the string to a hash map after a small, fixed number of Trie node traversals.

This optimization allows for more efficient use of memory and faster operations, especially in real-world applications with large datasets or long identifiers like license plates. Here is the test result after the whole test suite:

Before Optimized

```
 --Test: Large Dataset with 100000 plates --
Time to insert 100000 plates: 0.43172 seconds
Time to perform 100 searches: 0.00010 seconds
Passed: Large Dataset
```

```
-- Test: Memory Usage with 100000 plates --
Time to insert 100000 plates: 0.83 seconds
Current memory usage: 117.61 MB; Peak memory usage: 117.61 MB
Passed: Memory Usage Test
```

After optimized

```
 --Test: Large Dataset with 100000 plates --
Time to insert 100000 plates: 0.11787 seconds
Time to perform 100 searches: 0.00017 seconds
Passed: Large Dataset
```

```
-- Test: Memory Usage with 100000 plates --
Time to insert 100000 plates: 0.35 seconds
Current memory usage: 19.60 MB; Peak memory usage: 19.60 MB
Passed: Memory Usage Test
```

Analyzing Results:

After running the tests, these following metrics:

- Time Complexity:
    - Operations like insertions and searches are faster for long license plates, as the time spent traversing individual nodes is reduced after the first few characters. We can see that after optimization the insertion are significant faster versus the non-optimized. However, the searches perform roughly the same.

- Memory Usage:
    - By compressing paths and using a hash map for suffixes, it significantly reduces the number of nodes required, especially for long license plates or plates with common prefixes. We can see that after optimization memory usage is almost 5 times less than the non-optimized.

Tradeoffs when doing optimization

Time and space complexity: The trade-off here is that the optimization improves time complexity for insertion and search operations by compressing paths and limiting Trie traversal depth, but it slightly increases space complexity due to the use of hash maps. However, the space savings from path compression generally outweigh the hash map overhead, especially for long license plates or large datasets (Cormen et al., 2009).

The **trade-off** between **scalability** and **simplicity**. While the hybrid Trie-Hash Map structure improves the ability to scale for large datasets, it increases the overall complexity of the implementation (Knuth, 1998). The optimization is necessary for high-performance applications, but at the cost of increased maintenance and design complexity.

Another trade-off between memory usage and time efficiency results in improved performance, as compressed paths and hash maps reduce both memory consumption and operation time. However, this introduces some overhead in managing the hybrid structure (Knuth, 1998).

Strengths and limitations of the entire system

The final system integrates several key components, including vehicle registration using a dictionary, a priority queue for managing expiration dates, and a hybrid Trie-Hash Map for license plate management, alongside functions for searching by prefix, updating records, and deleting vehicles. Each of these components contributes to a highly functional and efficient vehicle registration system.

One of the key strengths of the system is the dictionary-based vehicle registration. The dictionary provides constant time $O(1)$ access, insertion, and updates for vehicle records, making it ideal for handling registration data efficiently (Cormen et al., 2009). This ensures that vehicle information such as owner details, vehicle specifications, and license plates can be accessed and modified quickly, even as the dataset grows.

The hybrid Trie-Hash Map structure efficiently handles license plate lookups by compressing common prefixes and using hash maps for the remaining parts of license plates. This reduces the depth of the Trie and speeds up search and insertion operations, especially for large datasets with shared prefixes (Knuth, 1998). Additionally, the system's prefix-based search functionality enables users to quickly retrieve all vehicles matching certain partial license plate entries, which is useful in real-world scenarios like traffic enforcement or insurance databases.

However, the system also has some limitations. While the dictionary-based registration system offers fast lookups, it can consume significant memory as the dataset grows, especially if redundant or stale entries are not properly managed. The priority queue, although efficient, can

become less performant if frequent updates to expiration dates are required. This may introduce bottlenecks in scenarios where vehicles need to frequently renew their registration before expiry (Sedgewick & Wayne, 2011).

The Trie-Hash Map hybrid structure, though scalable and efficient for common operations, adds complexity in managing interactions between the Trie and the hash map. As the hash map grows, it may encounter collisions, leading to occasional performance degradation during lookups (Cormen et al., 2009). Additionally, the system lacks a robust method for garbage collection or record cleanup, which could lead to inefficiencies over time as records accumulate or are modified without removal.

Future improvements

The dictionary-based registration could be optimized with periodic garbage collection to remove stale records and prevent memory bloat.

For the Trie-Hash Map structure, implementing advanced collision resolution techniques like double hashing or open addressing could improve performance in edge cases where hash map collisions occur frequently. Finally, introducing a batch updating or deletion mechanism could help improve performance when managing a large volume of updates or deletions simultaneously.

Reference:

Cormen, T. H., Leiserson, C. E., Rivest, R. L., & Stein, C. (2022). Introduction to Algorithms (4th ed.). Random House Publishing Services. https://reader2.yuzu.com/books/9780262367509

Goodrich, M., & Tamassia, R. (2014). *Data Structures and algorithms in Java, 6th edition*. John Wiley & Sons.

Knuth, D. (1998). *The art of computer programming sorting and searching, volume 3*. Addison Wesley.

Sedgewick, R. (2015). *Algorithms*. Addison-Wesley.

Weiss, M. A. (2014). *Data structures and algorithm analysis in C++*. Pearson.