Dynamic Vehicle Registration Management System with Efficient Expiration Tracking and License Plate Search

Name: Brendan Yeong

Student ID: 005025797

Course: MSCS532

Project Phase 1 Deliverable 1

1. Application context: Vehicle Registration System

A vehicle registration system that is responsible for keeping track of vehicle registrations, owner details (first name, last name), and associated information such as vehicle details, registration statuses (active, expired, pending, suspended, etc.), and expiration dates(dd/mm/yyyy). This system must be able to handle the following dynamic changes:

- vehicle registration records (including adding, updating, retrieve and removing vehicles).
- Owner details (each car is associated with an owner who can be queried by name or ID).
- Registration status(active, expired, pending, suspended, etc.) and expiration dates for registration that will expire .

The system should be able to handle searches, modifications, and queries (e.g., finding cars by owner, searching for expired registrations, etc.) efficiently.

2. Design the data structure

For this dynamic vehicle registration system, I have decided these data structures that could be a good fit for managing the types of data to be stored and handle dynamic changes. Let's define a few data structure needed for these features, vehicle registration records, expiration dates queries, finding vehicle by owner, vehicle plate queries.

- Hash Table (Dictionary) for vehicle details:
    - We need to store each vehicle uniquely with the number plate and its associating details such as registration details (make, model, color, year, classification, vin number), and details such as owner name, registration date, and expiration date. A hash table allows $O(1)$ average time complexity for it inserts, search, updates, and remove operations. This is important for when we

are going to be managing thousands to millions of registrations. Using the license plate number as a unique key, the Hash Table offers fast access to vehicle details and it is space-efficient

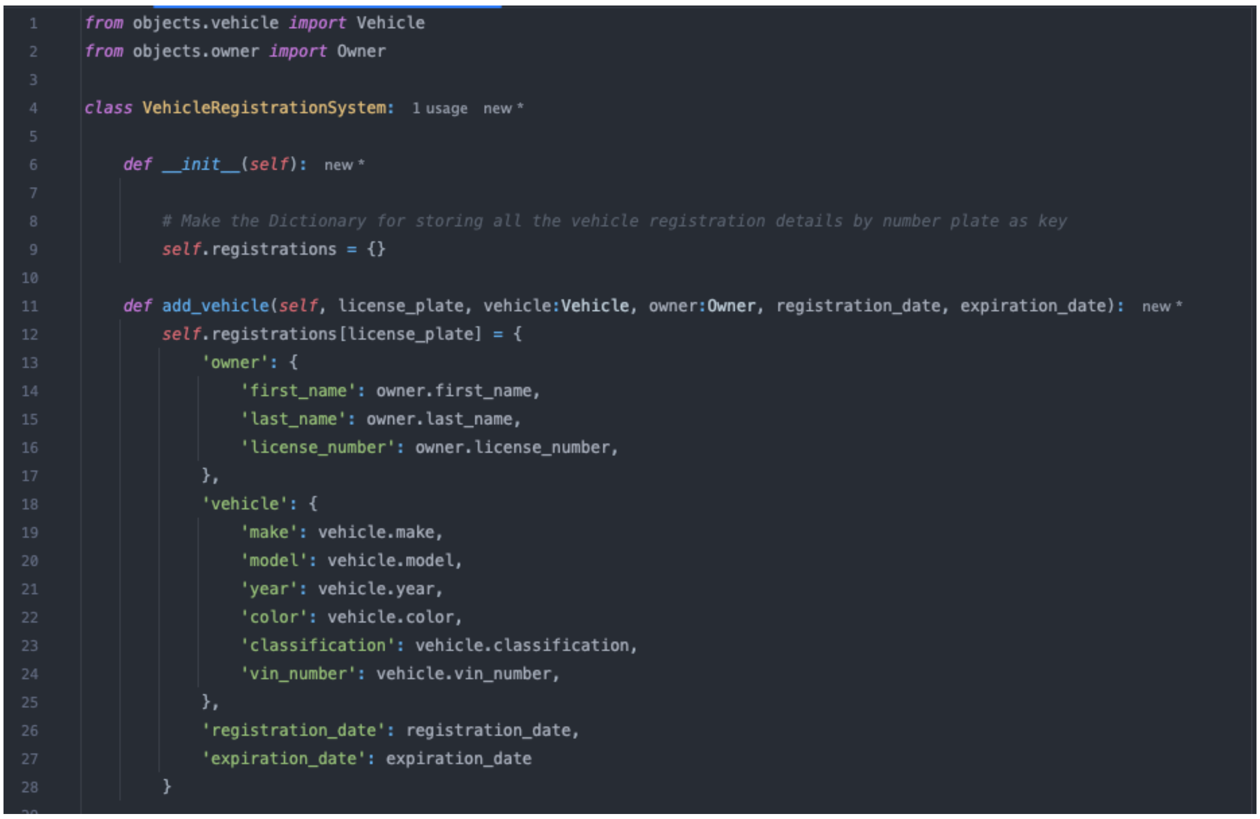
- Heap (Priority Queue) for expiration dates:
    - We can use a priority queue to keep track of registration expiration dates to quickly identify the next registration that will expire. With priority queue the run-time for insertion and deletion are $O(\log n)$ and accessing to the top element is $O(1)$, making it easy to lookup the nearest expiration date with the license plate efficiently. Using heap in this case is very efficient because we can quickly access and update the expiring registrations without needing to iterate over all entries.
- Balance BST (AVL) for owner-based queries:
    - We need a feature to that allows searches to find all the vehicle associated by a specific owner (Driver's License number). An AVL, binary search tree with a balance condition ensures the operation of insertion, deletion, and search run-time is $O(\log n)$. This means that it is efficient for owner-base queries. Another advantage in doing this implementation is for each time a new owner is entered, it ensures that the height is balanced and this guarantees that the tree will remain logarithmic in height making it consistently efficient with all the operations.
- Trie Data Structure for Plate prefix queries:
    - Last feature that I would like to include is a plate prefix lookup feature that can return a list of plates from the partial lookup. For this feature we can use Trie data structure as it allows a autocomplete feature. A Trie uses a time complexity of $O(m)$ where m is the length of the prefix. The only drawback is

it uses a lot of memory but it allows users to search up vehicle license plate numbers partially.

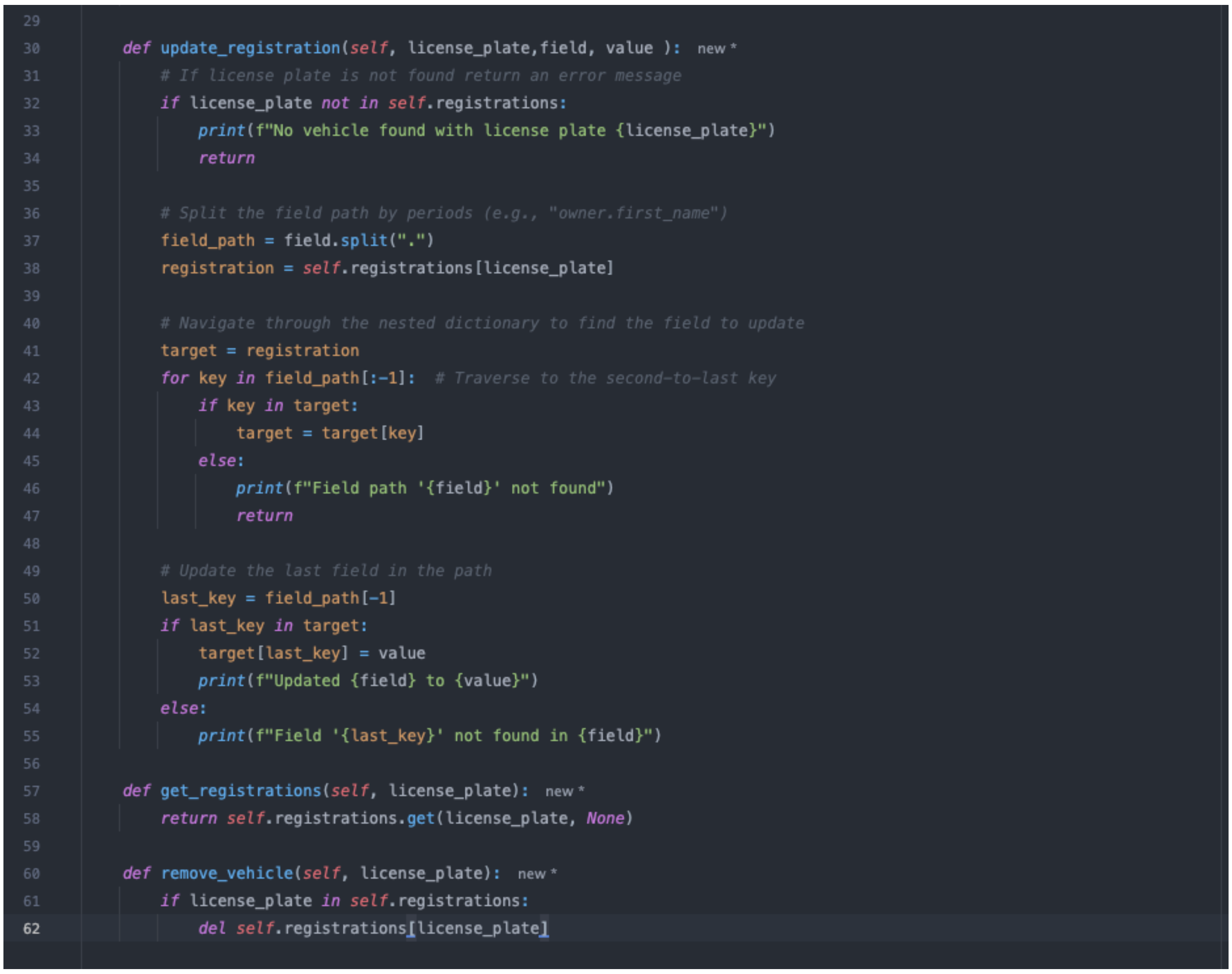
3. Implement the Data Structures in Python:

Here are the code snippets for all the implementations. For more details head to the GitHub link, https://github.com/CookiesBot3/MSCS532_Project

Hash Table (Dictionary) for vehicle details:

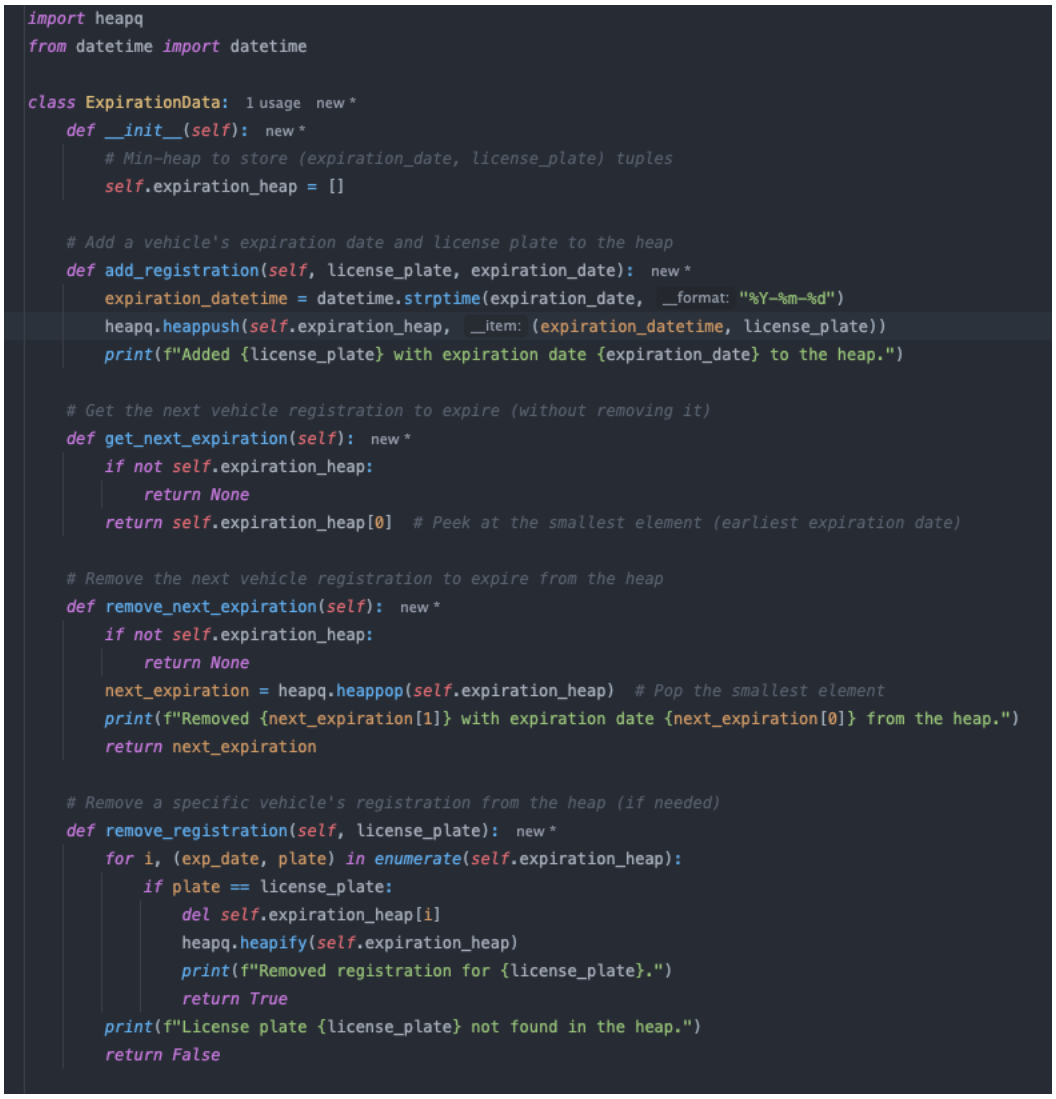


```python
from objects.vehicle import Vehicle
from objects.owner import Owner

class VehicleRegistrationSystem:  1 usage  new *

    def __init__(self):  new *

        # Make the Dictionary for storing all the vehicle registration details by number plate as key
        self.registrations = {}

    def add_vehicle(self, license_plate, vehicle:Vehicle, owner:Owner, registration_date, expiration_date):  new *
        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
        }
```



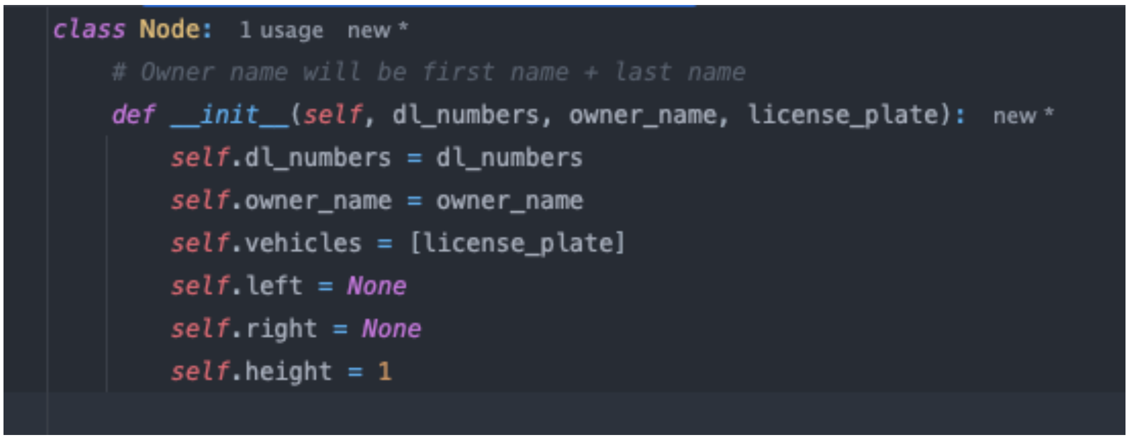
```python

    def update_registration(self, license_plate,field, value ):  new *
        # If license plate is not found return an error message
        if license_plate not in self.registrations:
            print(f"No vehicle found with license plate {license_plate}")
            return

        # Split the field path by periods (e.g., "owner.first_name")
        field_path = field.split(".")
        registration = self.registrations[license_plate]

        # Navigate through the nested dictionary to find the field to update
        target = registration
        for key in field_path[:-1]:  # Traverse to the second-to-last key
            if key in target:
                target = target[key]
            else:
                print(f"Field path '{field}' not found")
                return

        # Update the last field in the path
        last_key = field_path[-1]
        if last_key in target:
            target[last_key] = value
            print(f"Updated {field} to {value}")
        else:
            print(f"Field '{last_key}' not found in {field}")

    def get_registrations(self, license_plate):  new *
        return self.registrations.get(license_plate, None)

    def remove_vehicle(self, license_plate):  new *
        if license_plate in self.registrations:
            del self.registrations[license_plate]
```

Heap (Priority Queue) for expiration dates:

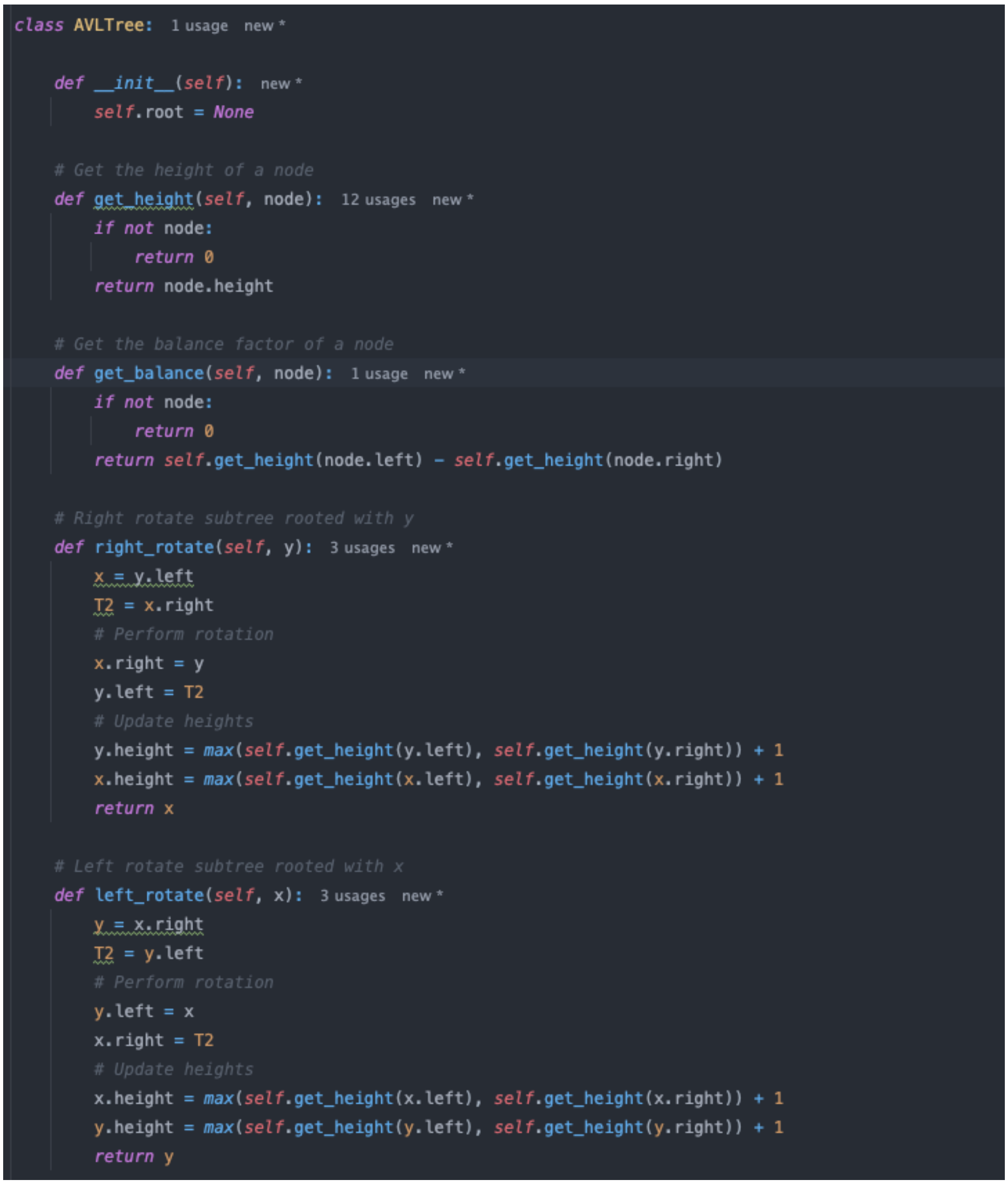


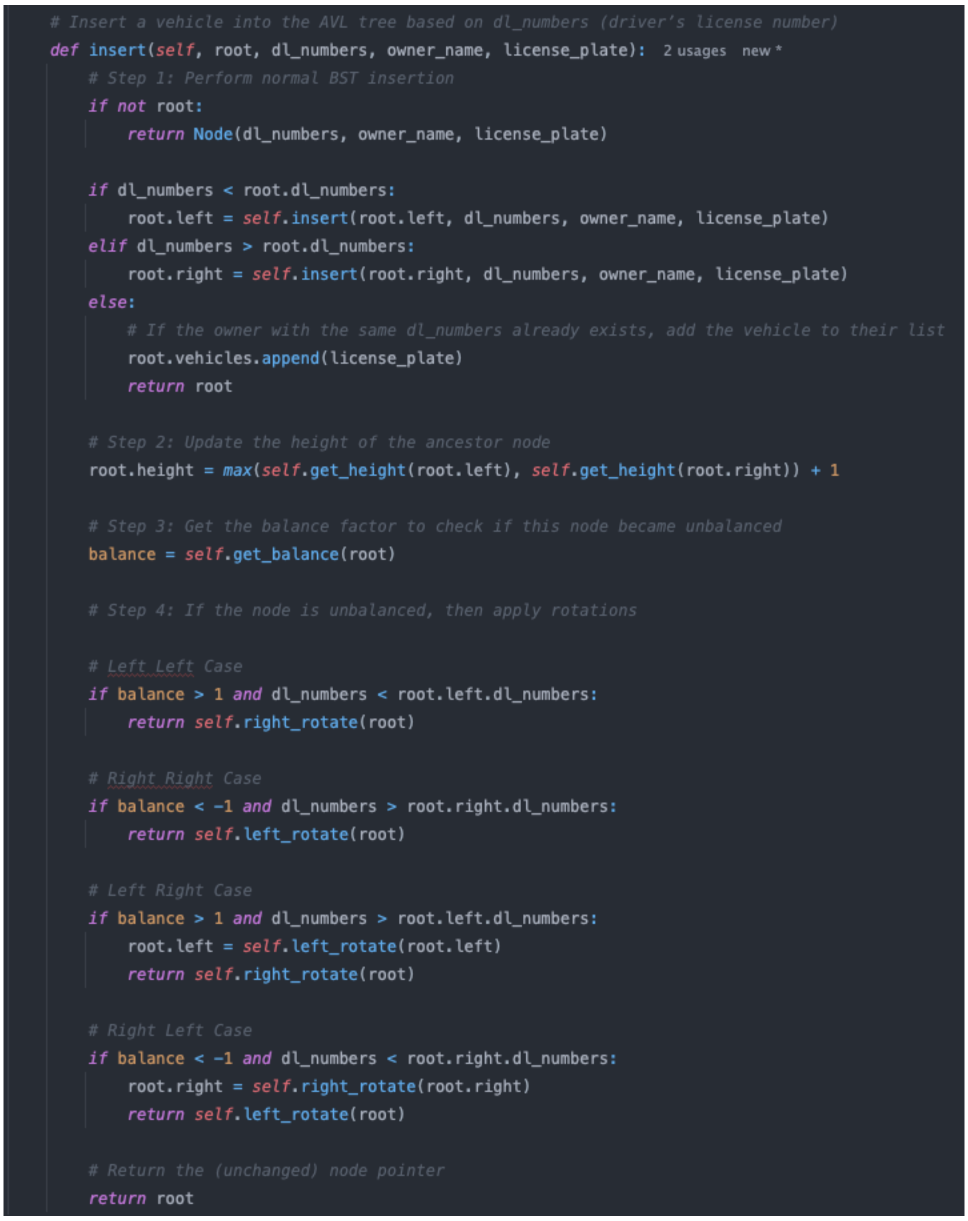
```python
import heapq
from datetime import datetime

class ExpirationData:  1 usage  new *
    def __init__(self):  new *
        # Min-heap to store (expiration_date, license_plate) tuples
        self.expiration_heap = []

    # Add a vehicle's expiration date and license plate to the heap
    def add_registration(self, license_plate, expiration_date):  new *
        expiration_datetime = datetime.strptime(expiration_date,  __format: "%Y-%m-%d")
        heapq.heappush(self.expiration_heap,  __item: (expiration_datetime, license_plate))
        print(f"Added {license_plate} with expiration date {expiration_date} to the heap.")

    # Get the next vehicle registration to expire (without removing it)
    def get_next_expiration(self):  new *
        if not self.expiration_heap:
            return None
        return self.expiration_heap[0]  # Peek at the smallest element (earliest expiration date)

    # Remove the next vehicle registration to expire from the heap
    def remove_next_expiration(self):  new *
        if not self.expiration_heap:
            return None
        next_expiration = heapq.heappop(self.expiration_heap)  # Pop the smallest element
        print(f"Removed {next_expiration[1]} with expiration date {next_expiration[0]} from the heap.")
        return next_expiration

    # Remove a specific vehicle's registration from the heap (if needed)
    def remove_registration(self, license_plate):  new *
        for i, (exp_date, plate) in enumerate(self.expiration_heap):
            if plate == license_plate:
                del self.expiration_heap[i]
                heapq.heapify(self.expiration_heap)
                print(f"Removed registration for {license_plate}.")
                return True
        print(f"License plate {license_plate} not found in the heap.")
        return False
```

Balance BST (AVL) for owner-based queries:

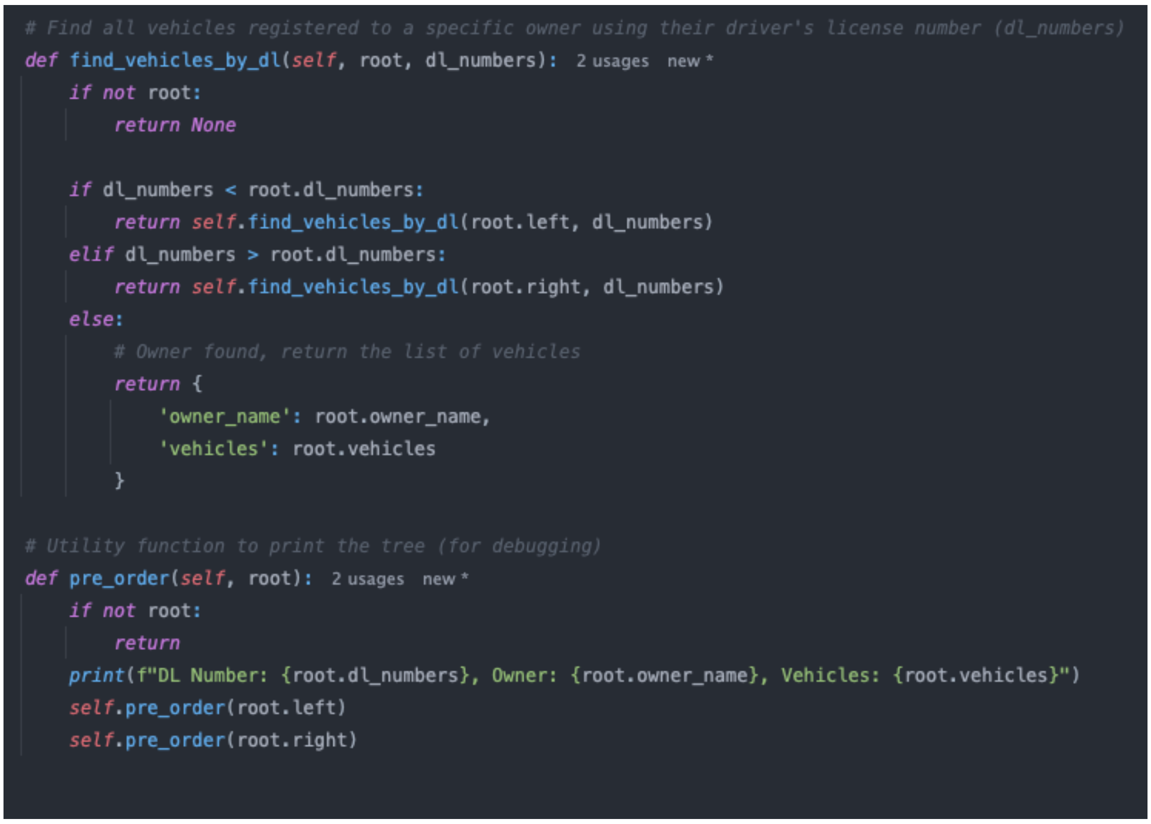


```python
class Node:  1 usage  new *
    # Owner name will be first name + last name
    def __init__(self, dl_numbers, owner_name, license_plate):  new *
        self.dl_numbers = dl_numbers
        self.owner_name = owner_name
        self.vehicles = [license_plate]
        self.left = None
        self.right = None
        self.height = 1
```


```python
class AVLTree:  1 usage  new *

    def __init__(self):  new *
        self.root = None

    # Get the height of a node
    def get_height(self, node):  12 usages  new *
        if not node:
            return 0
        return node.height

    # Get the balance factor of a node
    def get_balance(self, node):  1 usage  new *
        if not node:
            return 0
        return self.get_height(node.left) - self.get_height(node.right)

    # Right rotate subtree rooted with y
    def right_rotate(self, y):  3 usages  new *
        x = y.left
        T2 = x.right
        # Perform rotation
        x.right = y
        y.left = T2
        # Update heights
        y.height = max(self.get_height(y.left), self.get_height(y.right)) + 1
        x.height = max(self.get_height(x.left), self.get_height(x.right)) + 1
        return x

    # Left rotate subtree rooted with x
    def left_rotate(self, x):  3 usages  new *
        y = x.right
        T2 = y.left
        # Perform rotation
        y.left = x
        x.right = T2
        # Update heights
        x.height = max(self.get_height(x.left), self.get_height(x.right)) + 1
        y.height = max(self.get_height(y.left), self.get_height(y.right)) + 1
        return y
```

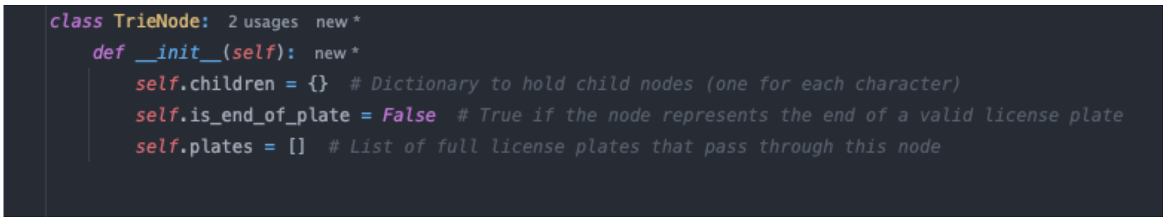


```
# Insert a vehicle into the AVL tree based on dl_numbers (driver's license number)
def insert(self, root, dl_numbers, owner_name, license_plate):  2 usages  new *
    # Step 1: Perform normal BST insertion
    if not root:
        return Node(dl_numbers, owner_name, license_plate)

    if dl_numbers < root.dl_numbers:
        root.left = self.insert(root.left, dl_numbers, owner_name, license_plate)
    elif dl_numbers > root.dl_numbers:
        root.right = self.insert(root.right, dl_numbers, owner_name, license_plate)
    else:
        # If the owner with the same dl_numbers already exists, add the vehicle to their list
        root.vehicles.append(license_plate)
        return root

    # Step 2: Update the height of the ancestor node
    root.height = max(self.get_height(root.left), self.get_height(root.right)) + 1

    # Step 3: Get the balance factor to check if this node became unbalanced
    balance = self.get_balance(root)

    # Step 4: If the node is unbalanced, then apply rotations

    # Left Left Case
    if balance > 1 and dl_numbers < root.left.dl_numbers:
        return self.right_rotate(root)

    # Right Right Case
    if balance < -1 and dl_numbers > root.right.dl_numbers:
        return self.left_rotate(root)

    # Left Right Case
    if balance > 1 and dl_numbers > root.left.dl_numbers:
        root.left = self.left_rotate(root.left)
        return self.right_rotate(root)

    # Right Left Case
    if balance < -1 and dl_numbers < root.right.dl_numbers:
        root.right = self.right_rotate(root.right)
        return self.left_rotate(root)

    # Return the (unchanged) node pointer
    return root
```

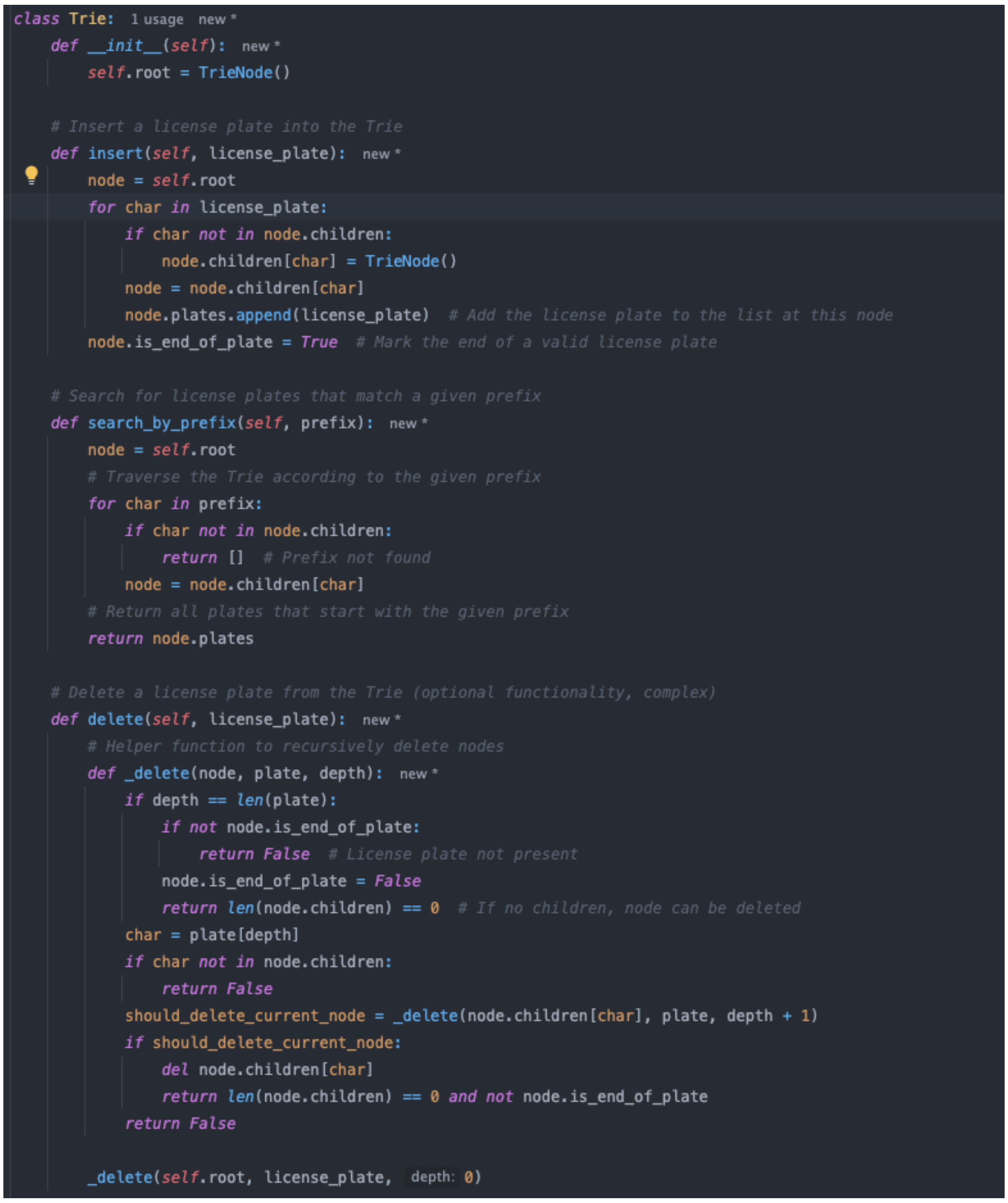


```python
# Find all vehicles registered to a specific owner using their driver's license number (dl_numbers)
def find_vehicles_by_dl(self, root, dl_numbers):  2 usages  new *
    if not root:
        return None

    if dl_numbers < root.dl_numbers:
        return self.find_vehicles_by_dl(root.left, dl_numbers)
    elif dl_numbers > root.dl_numbers:
        return self.find_vehicles_by_dl(root.right, dl_numbers)
    else:
        # Owner found, return the list of vehicles
        return {
            'owner_name': root.owner_name,
            'vehicles': root.vehicles
        }

# Utility function to print the tree (for debugging)
def pre_order(self, root):  2 usages  new *
    if not root:
        return
    print(f"DL Number: {root.dl_numbers}, Owner: {root.owner_name}, Vehicles: {root.vehicles}")
    self.pre_order(root.left)
    self.pre_order(root.right)
```

Trie Data Structure for Plate prefix queries:



```python
class TrieNode:  2 usages  new *
    def __init__(self):  new *
        self.children = {}  # Dictionary to hold child nodes (one for each character)
        self.is_end_of_plate = False  # True if the node represents the end of a valid license plate
        self.plates = []  # List of full license plates that pass through this node
```



```python
class Trie:  1 usage  new *
    def __init__(self):  new *
        self.root = TrieNode()

    # Insert a license plate into the Trie
    def insert(self, license_plate):  new *
        node = self.root
        for char in license_plate:
            if char not in node.children:
                node.children[char] = TrieNode()
            node = node.children[char]
            node.plates.append(license_plate)  # Add the license plate to the list at this node
        node.is_end_of_plate = True  # Mark the end of a valid license plate

    # Search for license plates that match a given prefix
    def search_by_prefix(self, prefix):  new *
        node = self.root
        # Traverse the Trie according to the given prefix
        for char in prefix:
            if char not in node.children:
                return []  # Prefix not found
            node = node.children[char]
        # Return all plates that start with the given prefix
        return node.plates

    # Delete a license plate from the Trie (optional functionality, complex)
    def delete(self, license_plate):  new *
        # Helper function to recursively delete nodes
        def _delete(node, plate, depth):  new *
            if depth == len(plate):
                if not node.is_end_of_plate:
                    return False  # License plate not present
                node.is_end_of_plate = False
                return len(node.children) == 0  # If no children, node can be deleted
            char = plate[depth]
            if char not in node.children:
                return False
            should_delete_current_node = _delete(node.children[char], plate, depth + 1)
            if should_delete_current_node:
                del node.children[char]
                return len(node.children) == 0 and not node.is_end_of_plate
            return False

        _delete(self.root, license_plate,  depth: 0)
```

Challenges and limitations

One of the limitations, I see in the long term is managing data synchronizations. As the system grows, managing data consistency and synchronization with multiple users accessing and updating data could lead to some complications and that could be a challenge. A layer of concurrency control or transaction handling is definitely needed.

Although Trie data structure is one of the main data structure for search results it does consume a lot of memory, especially when they are many unique license plates. Optimizations such as making it into a compressed tries can potentially mitigate this issue but it adds a layer of complexity to the implementations.

Another is the AVL tree. Even though it is very efficient in doing it’s basic operations it is very hard to implement and maintain overtime when new data or requirement is added. For a future large-scale application, a more vigorous testing is needed to prevent performance bottlenecks or potential errors occurs in the system.

Reference

Goodrich, M. T., & Tamassia, R. (2014). *Data Structures and algorithms in Java Michael T. Goodrich; Roberto Tamassia*. John Wiley.

Knuth, D. E. (1998). *The art of computer programming. volume 3*. Addison-Wesley.

Weiss, M. A. (2012). *Data structures and algorithm analysis in Java*. Addison-Wesley.