# DYNAMIC VEHICLE REGISTRATION MANAGEMENT SYSTEM WITH EFFICIENT EXPIRATION TRACKING AND LICENSE PLATE SEARCH

Brendan Yeong

Student ID: 005025797

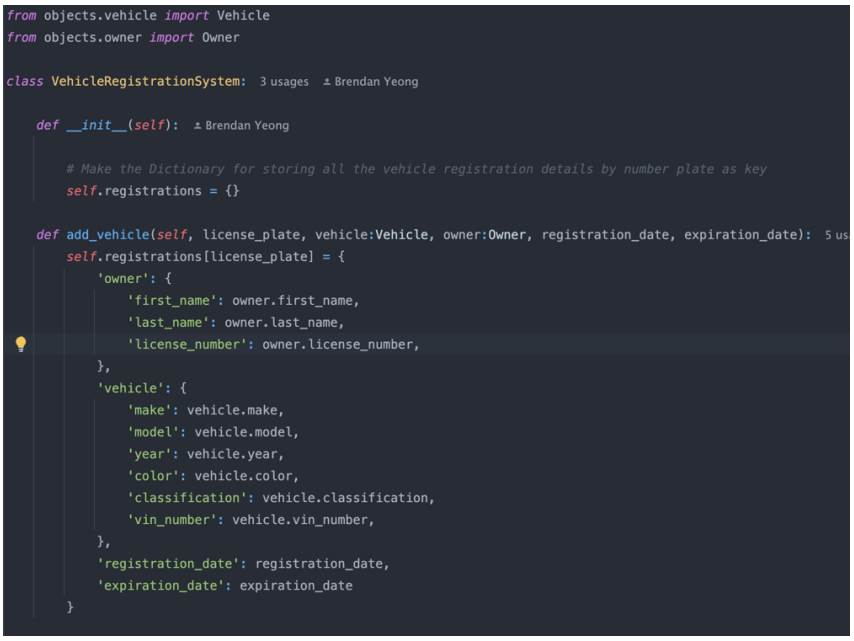
Course: MSCS532

Project Deliverable 2

Deliverable 2:

1. Partial Implementation of Data Structures:

Dictionary for Vehicle registration to store all the main data.



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

class VehicleRegistrationSystem:  3 usages  Brendan Yeong

    def __init__(self):  Brendan Yeong

        # 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):  5 us
        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 ):  5 usages (1 dynamic)
        # 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):  3 usages  Brendan Yeong
        return self.registrations.get(license_plate, None)

    def remove_vehicle(self, license_plate):  3 usages (1 dynamic)  Brendan Yeong
        if license_plate in self.registrations:
            del self.registrations[license_plate]
```

Priority queue for the Expiration dates tracker to identify the next registration that will expire.

```python
import heapq
from datetime import datetime

class ExpirationData:  3 usages  Brendan Yeong *
    def __init__(self):  Brendan Yeong
        # 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):  7 usages (2 dynamic)  Brendan Yeong
        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):  3 usages (1 dynamic)  Brendan Yeong
        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):  4 usages  Brendan Yeong
        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
```

```python
    # Update the expiration date for a given vehicle
    def update_registration(self, license_plate, new_expiration_date):  3 usages (1 dynamic)  new *
        # Step 1: Remove the old registration entry
        removed = self.remove_registration(license_plate)
        if removed:
            # Step 2: Add the new registration with the updated expiration date
            self.add_registration(license_plate, new_expiration_date)
            print(f"Updated expiration date for {license_plate} to {new_expiration_date}.")
        else:
            print(f"Failed to update: License plate {license_plate} not found.")

    # Utility function to check the contents of the heap (for debugging)
    def print_heap(self):  6 usages  Brendan Yeong
        print("Current Expiration Heap:")
        for exp_date, plate in self.expiration_heap:
            print(f"License Plate: {plate}, Expiration Date: {exp_date.strftime('%Y-%m-%d')}")
```

AVL Tree for the Owner based car registration. It can store the owner and the total number of cars that is owned.

```python
class Node:  1 usage  Brendan Yeong *
    def __init__(self, dl_numbers, owner_name, license_plate):  Brendan Yeong *
        self.dl_numbers = dl_numbers  # Driver's License Number (unique key)
        self.owner_name = owner_name
        self.vehicles = [license_plate]  # Store a list of vehicles (license plates) under the owner
        self.left = None
        self.right = None
        self.height = 1

class AVLTree:  3 usages  Brendan Yeong *
    def __init__(self):  Brendan Yeong
        self.root = None

    # Get the height of a node
    def get_height(self, node):  14 usages  Brendan Yeong
        if not node:
            return 0
        return node.height

    # Get the balance factor of a node
    def get_balance(self, node):  6 usages  Brendan Yeong
        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):  6 usages  Brendan Yeong
        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
```

```python
# Left rotate subtree rooted with x
def left_rotate(self, x):  6 usages  Brendan Yeong
    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):  10 usages (2 dynamic)  Brendan Yeong
    # 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
```

```python
    # 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
# Remove a license plate for a specific driver's license
def remove(self, root, dl_number, license_plate):  8 usages (4 dynamic)  new *
    if not root:
        return root

    if dl_number < root.dl_numbers:
        root.left = self.remove(root.left, dl_number, license_plate)
    elif dl_number > root.dl_numbers:
        root.right = self.remove(root.right, dl_number, license_plate)
    else:
        # If we found the driver's license node, remove the vehicle from the list
        if license_plate in root.vehicles:
            root.vehicles.remove(license_plate)

        # If no vehicles remain, we delete the node
        if not root.vehicles:
            if not root.left:
                return root.right
            elif not root.right:
                return root.left

            # If the node has two children, get the inorder successor (smallest in the right subtree)
            temp = self.get_min_value_node(root.right)
            root.dl_numbers = temp.dl_numbers
            root.vehicles = temp.vehicles
            root.right = self.remove(root.right, temp.dl_numbers, license_plate)

    # If the node had children, we need to update its height and balance the tree
    if root is None:
        return root

    root.height = max(self.get_height(root.left), self.get_height(root.right)) + 1

    balance = self.get_balance(root)
```

```
    balance = self.get_balance(root)

    # Balancing
    if balance > 1 and self.get_balance(root.left) >= 0:
        return self.right_rotate(root)

    if balance < -1 and self.get_balance(root.right) <= 0:
        return self.left_rotate(root)

    if balance > 1 and self.get_balance(root.left) < 0:
        root.left = self.left_rotate(root.left)
        return self.right_rotate(root)

    if balance < -1 and self.get_balance(root.right) > 0:
        root.right = self.right_rotate(root.right)
        return self.left_rotate(root)

    return root

# Utility function to get the node with the smallest value in a subtree
def get_min_value_node(self, node):  2 usages  new *
    if node is None or node.left is None:
        return node
    return self.get_min_value_node(node.left)
```

```
# Utility function to get the node with the smallest value in a subtree
def get_min_value_node(self, node):  2 usages  new *
    if node is None or node.left is None:
        return node
    return self.get_min_value_node(node.left)

# Find vehicles by driver's license number
def find_vehicles_by_dl(self, root, dl_numbers):  7 usages (1 dynamic)  Brendan Yeong *
    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:
        return {'owner_name': root.owner_name, 'vehicles': root.vehicles}

# Utility function to print the tree (for debugging)
def pre_order(self, root):  6 usages  Brendan Yeong
    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 vehicle number plate prefix lookups.

```python
class TrieNode:  2 usages  Brendan Yeong
    def __init__(self):  Brendan Yeong
        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


class Trie:  3 usages  Brendan Yeong *
    def __init__(self):  Brendan Yeong
        self.root = TrieNode()

    # Insert a license plate into the Trie
    def insert(self, license_plate):  9 usages (2 dynamic)  Brendan Yeong
        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):  13 usages (1 dynamic)  Brendan Yeong
        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
```

```python
# Delete a license plate from the Trie
def delete(self, license_plate):  4 usages (1 dynamic)  Brendan Yeong *
    # Helper function to recursively delete nodes
    def _delete(node, plate, depth):  Brendan Yeong *
        if depth == len(plate):
            if not node.is_end_of_plate:
                return False  # License plate not present
            node.is_end_of_plate = False  # Unmark the end of the plate
            if plate in node.plates:
                node.plates.remove(plate)  # Remove the plate from the node's list
            return len(node.children) == 0  # If no children, node can be deleted

        char = plate[depth]
        if char not in node.children:
            return False  # Plate not found

        should_delete_current_node = _delete(node.children[char], plate, depth + 1)

        # Remove the license plate from the current node's plates list
        if plate in node.plates:
            node.plates.remove(plate)

        # If the recursive call says to delete the child, do it
        if should_delete_current_node:
            del node.children[char]
            # Return true if the current node has no other children and isn't the end of another plate
            return len(node.children) == 0 and not node.is_end_of_plate

        return False

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

```python
# Utility function to print the Trie (for debugging)
def print_trie(self, node=None, prefix=""):  4 usages  new *
    if node is None:
        node = self.root
    if node.is_end_of_plate:
        print(f"Prefix: {prefix}, Plates: {node.plates}")
    for char, child in node.children.items():
        self.print_trie(child, prefix + char)
```

Demonstration of Key Operations:

To demonstration the key operations of each data structure that is partially implemented, I have created some test cases that I can run, and the results will also be attach below it.

Dictionary for Vehicle registration:

Test Case 1: Adding vehicle registrations

- Adds multiple vehicle registrations for different owners and prints the dictionary to verify the entries.
- Tests adding another vehicle registration for the same owner (LMN456 for John Doe).

Test Case 2: Retrieving vehicle registration by license plate

- Retrieves vehicle registration information using the get_vehicle() method. This includes an edge case where the license plate doesn't exist ("NONEXISTENT").

Test Case 3: Updating specific fields in a vehicle registration

- Tests the update_registration() method to update the owner's first name, the vehicle's color, and the expiration date for various license plates.
- Includes an edge case where an invalid field is provided for update ("vehicle.invalid_field"), which should not modify the registration.

Test Case 4: Removing a vehicle registration by license plate

- Tests the remove_vehicle() method to remove an existing registration (ABC123) and verifies the dictionary after removal.
- Includes an edge case where the system tries to remove a non-existent registration ("NONEXISTENT").

Test Case 5: Edge case - Adding a duplicate license plate

- Tests if adding a registration with a duplicate license plate (XYZ789) correctly overwrites or handles the situation based on the system's logic. The dictionary should reflect the updated registration details.

```
# Create the CarRegistrationSystem object
car_system = VehicleRegistrationSystem()

# Test Case 1: Adding vehicle registrations
print("\n-- Test Case 1: Adding vehicle registrations --")
vehicle_1 = Vehicle( make: "Toyota",  model: "Camry",  year: 2020,  color: "Blue",  classification: "Sedan",  vin_number: "123456789")
owner_1 = Owner( first_name: "John",  last_name: "Doe",  license_number: "DL12345")
car_system.add_vehicle( license_plate: "ABC123", vehicle_1, owner_1,  registration_date: "2023-01-01",  expiration_date: "2024-01-01")

vehicle_2 = Vehicle( make: "Honda",  model: "Civic",  year: 2019,  color: "Red",  classification: "Sedan",  vin_number: "987654321")
owner_2 = Owner( first_name: "Jane",  last_name: "Doe",  license_number: "DL67890")
car_system.add_vehicle( license_plate: "XYZ789", vehicle_2, owner_2,  registration_date: "2022-06-15",  expiration_date: "2023-06-15")

car_system.add_vehicle( license_plate: "LMN456", vehicle_1, owner_1,  registration_date: "2023-04-01",  expiration_date: "2024-04-01")  # Anot
print(car_system.registrations)  # Print the current registrations to verify

# Test Case 2: Retrieving vehicle registration by license plate
print("\n-- Test Case 2: Retrieving vehicle registration by license plate --")
print(f"Registration for 'ABC123': {car_system.get_registrations('ABC123')}")
print(f"Registration for 'XYZ789': {car_system.get_registrations('XYZ789')}")
print(f"Registration for 'NONEXISTENT': {car_system.get_registrations('NONEXISTENT')}")  # Edge case: Non-existent license plate

# Test Case 3: Updating specific fields in a vehicle registration
print("\n-- Test Case 3: Updating specific fields in a vehicle registration --")
car_system.update_registration( license_plate: "ABC123",  field: "owner.first_name",  value: "Alice")  # Update owner first name
car_system.update_registration( license_plate: "XYZ789",  field: "vehicle.color",  value: "Green")  # Update vehicle color
car_system.update_registration( license_plate: "LMN456",  field: "expiration_date",  value: "2025-04-01")  # Update expiration date
car_system.update_registration( license_plate: "XYZ789",  field: "vehicle.invalid_field",  value: "Value")  # Edge case: Invalid field
print(car_system.registrations)  # Verify updates
```

```
# Test Case 4: Removing a vehicle registration by license plate
print("\n-- Test Case 4: Removing a vehicle registration --")
car_system.remove_vehicle("ABC123")  # Remove existing registration
print(car_system.registrations)  # Verify removal
car_system.remove_vehicle("NONEXISTENT")  # Edge case: Try to remove non-existent registration

# Test Case 5: Edge case - Adding a duplicate license plate
print("\n-- Test Case 5: Adding a duplicate license plate --")
car_system.add_vehicle( license_plate: "XYZ789", vehicle_2, owner_2,  registration_date: "2022-06-15",  expiration_date: "2023-06-15")  # Attempt to
print(car_system.registrations)
```

```
-- Test Case 1: Adding vehicle registrations --
{'ABC123': {'owner': {'first_name': 'John', 'last_name': 'Doe', 'license_number': 'DL12345'}, 'vehicle': {'make': 'Toyota', 'model': 'Camry', 'year': 2020, 'color': 'Blue',
 'classification': 'Sedan', 'vin_number': '123456789'}, 'registration_date': '2023-01-01', 'expiration_date': '2024-01-01'}, 'XYZ789': {'owner': {'first_name': 'Jane',
 'last_name': 'Doe', 'license_number': 'DL67890'}, 'vehicle': {'make': 'Honda', 'model': 'Civic', 'year': 2019, 'color': 'Red', 'classification': 'Sedan', 'vin_number':
 '987654321'}, 'registration_date': '2022-06-15', 'expiration_date': '2023-06-15'}, 'LMN456': {'owner': {'first_name': 'John', 'last_name': 'Doe', 'license_number':
 'DL12345'}, 'vehicle': {'make': 'Toyota', 'model': 'Camry', 'year': 2020, 'color': 'Blue', 'classification': 'Sedan', 'vin_number': '123456789'}, 'registration_date':
 '2023-04-01', 'expiration_date': '2024-04-01'}}

-- Test Case 2: Retrieving vehicle registration by license plate --
Registration for 'ABC123': {'owner': {'first_name': 'John', 'last_name': 'Doe', 'license_number': 'DL12345'}, 'vehicle': {'make': 'Toyota', 'model': 'Camry', 'year': 2020,
 'color': 'Blue', 'classification': 'Sedan', 'vin_number': '123456789'}, 'registration_date': '2023-01-01', 'expiration_date': '2024-01-01'}
Registration for 'XYZ789': {'owner': {'first_name': 'Jane', 'last_name': 'Doe', 'license_number': 'DL67890'}, 'vehicle': {'make': 'Honda', 'model': 'Civic', 'year': 2019,
 'color': 'Red', 'classification': 'Sedan', 'vin_number': '987654321'}, 'registration_date': '2022-06-15', 'expiration_date': '2023-06-15'}
Registration for 'NONEXISTENT': None
```

```
-- Test Case 3: Updating specific fields in a vehicle registration --
Updated owner.first_name to Alice
Updated vehicle.color to Green
Updated expiration_date to 2025-04-01
Field 'invalid_field' not found in vehicle.invalid_field
{'ABC123': {'owner': {'first_name': 'Alice', 'last_name': 'Doe', 'license_number': 'DL12345'}, 'vehicle': {'make': 'Toyota', 'model': 'Camry', 'year': 2020, 'color': 'Blue',
 'classification': 'Sedan', 'vin_number': '123456789'}, 'registration_date': '2023-01-01', 'expiration_date': '2024-01-01'}, 'XYZ789': {'owner': {'first_name': 'Jane',
 'last_name': 'Doe', 'license_number': 'DL67890'}, 'vehicle': {'make': 'Honda', 'model': 'Civic', 'year': 2019, 'color': 'Green', 'classification': 'Sedan', 'vin_number':
 '987654321'}, 'registration_date': '2022-06-15', 'expiration_date': '2023-06-15'}, 'LMN456': {'owner': {'first_name': 'John', 'last_name': 'Doe', 'license_number':
 'DL12345'}, 'vehicle': {'make': 'Toyota', 'model': 'Camry', 'year': 2020, 'color': 'Blue', 'classification': 'Sedan', 'vin_number': '123456789'}, 'registration_date':
 '2023-04-01', 'expiration_date': '2025-04-01'}}

-- Test Case 4: Removing a vehicle registration --
{'XYZ789': {'owner': {'first_name': 'Jane', 'last_name': 'Doe', 'license_number': 'DL67890'}, 'vehicle': {'make': 'Honda', 'model': 'Civic', 'year': 2019, 'color': 'Green',
 'classification': 'Sedan', 'vin_number': '987654321'}, 'registration_date': '2022-06-15', 'expiration_date': '2023-06-15'}, 'LMN456': {'owner': {'first_name': 'John',
 'last_name': 'Doe', 'license_number': 'DL12345'}, 'vehicle': {'make': 'Toyota', 'model': 'Camry', 'year': 2020, 'color': 'Blue', 'classification': 'Sedan', 'vin_number':
 '123456789'}, 'registration_date': '2023-04-01', 'expiration_date': '2025-04-01'}}

-- Test Case 5: Adding a duplicate license plate --
{'XYZ789': {'owner': {'first_name': 'Jane', 'last_name': 'Doe', 'license_number': 'DL67890'}, 'vehicle': {'make': 'Honda', 'model': 'Civic', 'year': 2019, 'color': 'Red',
 'classification': 'Sedan', 'vin_number': '987654321'}, 'registration_date': '2022-06-15', 'expiration_date': '2023-06-15'}, 'LMN456': {'owner': {'first_name': 'John',
 'last_name': 'Doe', 'license_number': 'DL12345'}, 'vehicle': {'make': 'Toyota', 'model': 'Camry', 'year': 2020, 'color': 'Blue', 'classification': 'Sedan', 'vin_number':
 '123456789'}, 'registration_date': '2023-04-01', 'expiration_date': '2025-04-01'}}
```

Priority queue for the Expiration dates tracker:

1. Test Case 1: Adding vehicle registrations
    - This test adds several vehicle registrations with different expiration dates to the priority queue. The print_heap() function is used to display the current state of the heap to verify that the heap structure is maintained correctly.
2. Test Case 2: Get the next registration to expire
    - This test checks whether the heap correctly returns the registration with the soonest expiration date using the get_next_expiration() method. The expected behavior is that the earliest expiration date should be at the root of the heap.
3. Test Case 3: Remove the next registration to expire
    - This test removes the next vehicle registration to expire using remove_next_expiration(), and then prints the heap to ensure the heap is still properly ordered and balanced.
4. Test Case 4: Remove a specific registration (valid case)

- This test removes a specific registration from the heap using the remove_registration() method. It tests whether the heap can handle removal by license plate and still maintain its structure after removal.

5. Test Case 5: Try to remove a non-existent registration
    - This tests the behavior when attempting to remove a license plate that does not exist in the heap. The heap should remain unchanged, and an appropriate message should be printed.
6. Test Case 6: Edge Case - Removing from an empty heap
    - This test simulates updating the expiration date for an existing registration where it uses the update_registration(). The heap should have the updated dates, and gets printed to check the values.
7. Test Case 7: Edge Case - Removing from an empty heap
    - This test simulates removing all elements from the heap, eventually leaving it empty. It then verifies the behavior when trying to remove from an empty heap and when calling get_next_expiration() on an empty heap, which should return None.

```
expiration_manager = ExpirationData()

# Test Case 1: Add vehicle registrations with different expiration dates
print("\n-- Test Case 1: Adding vehicle registrations --")
expiration_manager.add_registration( license_plate: "ABC123",  expiration_date: "2024-01-15")
expiration_manager.add_registration( license_plate: "XYZ789",  expiration_date: "2023-12-01")
expiration_manager.add_registration( license_plate: "DEF456",  expiration_date: "2024-03-10")
expiration_manager.add_registration( license_plate: "LMN101",  expiration_date: "2023-11-20")
expiration_manager.print_heap()

# Test Case 2: Get the next registration to expire
print("\n-- Test Case 2: Getting the next registration to expire --")
next_expiration = expiration_manager.get_next_expiration()
if next_expiration:
    print(f"Next to expire: {next_expiration[1]} on {next_expiration[0].strftime('%Y-%m-%d')}")

# Test Case 3: Remove the next registration to expire and verify the heap
print("\n-- Test Case 3: Removing the next registration to expire --")
expiration_manager.remove_next_expiration()
expiration_manager.print_heap()  # Verify the heap after removal

# Test Case 4: Remove a specific registration (valid case)
print("\n-- Test Case 4: Removing a specific registration (DEF456) --")
expiration_manager.remove_registration("DEF456")
expiration_manager.print_heap()  # Verify the heap after removing specific registration

# Test Case 5: Try to remove a non-existent registration
print("\n-- Test Case 5: Trying to remove a non-existent registration (NONEXISTENT) --")
expiration_manager.remove_registration("NONEXISTENT")
```

```
# Test Case 6: Update expiration date for an existing registration
print("\n-- est Case 6: u --")
print("-- Update Registration 'XYZ789' --")
expiration_manager.update_registration( license_plate: "XYZ789",  new_expiration_date: "2024-05-01")
expiration_manager.print_heap()

# Update expiration date for a non-existent registration
print("\n-- Attempt to Update Non-Existent Registration 'NON123' --")
expiration_manager.update_registration( license_plate: "NON123",  new_expiration_date: "2025-01-01")
expiration_manager.print_heap()

# Test Case 7: Edge Case - Remove from an empty heap
print("\n-- Test Case 7: Removing from an empty heap --")
# Emptying the heap by removing all elements
expiration_manager.remove_next_expiration()  # Continue until empty
expiration_manager.remove_next_expiration()  # Remove the last element
expiration_manager.print_heap()  # Heap should now be empty
next_expiration = expiration_manager.get_next_expiration()  # Try to get the next expiration
if next_expiration is None:
    print("Heap is empty, no next expiration.")
```

```
-- Test Case 1: Adding vehicle registrations --
Added ABC123 with expiration date 2024-01-15 to the heap.
Added XYZ789 with expiration date 2023-12-01 to the heap.
Added DEF456 with expiration date 2024-03-10 to the heap.
Added LMN101 with expiration date 2023-11-20 to the heap.
Current Expiration Heap:
License Plate: LMN101, Expiration Date: 2023-11-20
License Plate: XYZ789, Expiration Date: 2023-12-01
License Plate: DEF456, Expiration Date: 2024-03-10
License Plate: ABC123, Expiration Date: 2024-01-15

-- Test Case 2: Getting the next registration to expire --
Next to expire: LMN101 on 2023-11-20

-- Test Case 3: Removing the next registration to expire --
Removed LMN101 with expiration date 2023-11-20 00:00:00 from the heap.
Current Expiration Heap:
License Plate: XYZ789, Expiration Date: 2023-12-01
License Plate: ABC123, Expiration Date: 2024-01-15
License Plate: DEF456, Expiration Date: 2024-03-10

-- Test Case 4: Removing a specific registration (DEF456) --
Removed registration for DEF456.
Current Expiration Heap:
License Plate: XYZ789, Expiration Date: 2023-12-01
License Plate: ABC123, Expiration Date: 2024-01-15
```

```
-- Test Case 5: Trying to remove a non-existent registration (NONEXISTENT) --
License plate NONEXISTENT not found in the heap.

-- est Case 6: Update expiration date for an existing registration --
-- Update Registration 'XYZ789' --
Removed registration for XYZ789.
Added XYZ789 with expiration date 2024-05-01 to the heap.
Updated expiration date for XYZ789 to 2024-05-01.
Current Expiration Heap:
License Plate: ABC123, Expiration Date: 2024-01-15
License Plate: XYZ789, Expiration Date: 2024-05-01

-- Attempt to Update Non-Existent Registration 'NON123' --
License plate NON123 not found in the heap.
Failed to update: License plate NON123 not found.
Current Expiration Heap:
License Plate: ABC123, Expiration Date: 2024-01-15
License Plate: XYZ789, Expiration Date: 2024-05-01

-- Test Case 7: Removing from an empty heap --
Removed ABC123 with expiration date 2024-01-15 00:00:00 from the heap.
Removed XYZ789 with expiration date 2024-05-01 00:00:00 from the heap.
Current Expiration Heap:
Heap is empty, no next expiration.
```

AVL Tree for the Owner based car registration:

1. Test Case 1: Insert vehicles for different owners
    - This tests the insertion of vehicles into the AVL tree for different owners, each having a unique driver's license number (dl_numbers). The pre_order() function prints the structure of the AVL tree after insertion to ensure it's balanced.
2. Test Case 2: Insert multiple vehicles for the same owner (with repeated dl_numbers)
    - This test checks the ability of the AVL tree to handle multiple vehicles for the same owner. The insertion function should append the new license plate to the existing node for the same dl_numbers.
3. Test Case 3: Search for vehicles by driver's license number (dl_numbers)
    - This tests the search functionality by retrieving the list of vehicles for a specific driver's license number. The find_vehicles_by_dl() method is used for this purpose.

4. Test Case 4: Edge case - Search for a non-existent driver's license number
    - This tests the behavior of the AVL tree when trying to search for a driver's license number that doesn't exist in the tree. The search function should return None or an appropriate message.
5. Test Case 5: Removing a license plate
    - This tests removes a license plate according to the driver's license number.
6. Test Case 6: Insert into an empty AVL tree (Edge case)
    - This tests the insertion function when starting with an empty AVL tree. The tree should correctly handle the insertion and maintain its structure.

```
avl_tree = AVLTree()
root = None

# Test Case 1: Insert vehicles for different owners
print("\n-- Test Case 1: Inserting vehicles for multiple owners --")
root = avl_tree.insert(root, dl_numbers: "DL12345", owner_name: "Alice", license_plate: "ABC123")
root = avl_tree.insert(root, dl_numbers: "DL67890", owner_name: "Bob", license_plate: "XYZ789")
root = avl_tree.insert(root, dl_numbers: "DL54321", owner_name: "Charlie", license_plate: "GHI012")
avl_tree.pre_order(root)  # Print the tree structure

# Test Case 2: Insert multiple vehicles for the same owner (with repeated dl_numbers)
print("\n-- Test Case 2: Inserting multiple vehicles for the same owner --")
root = avl_tree.insert(root, dl_numbers: "DL12345", owner_name: "Alice", license_plate: "DEF456")  # Adding another vehicle fo
root = avl_tree.insert(root, dl_numbers: "DL67890", owner_name: "Bob", license_plate: "UVW345")    # Adding another vehicle fo
avl_tree.pre_order(root)  # Print the tree structure

# Test Case 3: Search for vehicles owned by a specific driver (using dl_numbers)
print("\n-- Test Case 3: Searching for vehicles by driver's license number --")
print("Vehicles for DL12345 (Alice):", avl_tree.find_vehicles_by_dl(root, dl_numbers: "DL12345"))
print("Vehicles for DL67890 (Bob):", avl_tree.find_vehicles_by_dl(root, dl_numbers: "DL67890"))
print("Vehicles for DL54321 (Charlie):", avl_tree.find_vehicles_by_dl(root, dl_numbers: "DL54321"))

# Test Case 4: Edge case - Search for a non-existent driver's license number
print("\n-- Test Case 4: Searching for a non-existent driver's license number --")
print("Vehicles for DL99999 (non-existent):", avl_tree.find_vehicles_by_dl(root, dl_numbers: "DL99999"))

# Test Case 5: Remove a license plate
print("\n-- Test Case 5: Remove a license plate --")
print("Vehicles for DL12345 (Alice): remove(DEF456)")
avl_tree.remove(root, dl_number: "DL12345", license_plate: "DEF456")
avl_tree.pre_order(root)  # Print the tree structure

# Test Case 6: Insert into an empty AVL tree (Edge case)
print("\n-- Test Case 6: Inserting into an empty AVL tree --")
empty_tree_root = None
empty_tree_root = avl_tree.insert(empty_tree_root, dl_numbers: "DL11111", owner_name: "Eve", license_plate: "LMN345")
avl_tree.pre_order(empty_tree_root)  # Should only show Eve's data
```

```
-- Test Case 1: Inserting vehicles for multiple owners --
DL Number: DL54321, Owner: Charlie, Vehicles: ['GHI012']
DL Number: DL12345, Owner: Alice, Vehicles: ['ABC123']
DL Number: DL67890, Owner: Bob, Vehicles: ['XYZ789']

-- Test Case 2: Inserting multiple vehicles for the same owner --
DL Number: DL54321, Owner: Charlie, Vehicles: ['GHI012']
DL Number: DL12345, Owner: Alice, Vehicles: ['ABC123', 'DEF456']
DL Number: DL67890, Owner: Bob, Vehicles: ['XYZ789', 'UVW345']

-- Test Case 3: Searching for vehicles by driver's license number --
Vehicles for DL12345 (Alice): {'owner_name': 'Alice', 'vehicles': ['ABC123', 'DEF456']}
Vehicles for DL67890 (Bob): {'owner_name': 'Bob', 'vehicles': ['XYZ789', 'UVW345']}
Vehicles for DL54321 (Charlie): {'owner_name': 'Charlie', 'vehicles': ['GHI012']}

-- Test Case 4: Searching for a non-existent driver's license number --
Vehicles for DL99999 (non-existent): None

-- Test Case 5: Remove a license plate --
Vehicles for DL12345 (Alice): remove(DEF456)
DL Number: DL54321, Owner: Charlie, Vehicles: ['GHI012']
DL Number: DL12345, Owner: Alice, Vehicles: ['ABC123']
DL Number: DL67890, Owner: Bob, Vehicles: ['XYZ789', 'UVW345']

-- Test Case 6: Inserting into an empty AVL tree --
DL Number: DL11111, Owner: Eve, Vehicles: ['LMN345']
```

Trie Data Structure for vehicle number plate prefix lookups:

1. Test Case 1: Insert license plates into the Trie
    - Inserts several license plates with different prefixes into the Trie, ensuring the Trie structure is built correctly. This tests the basic insertion functionality.
2. Test Case 2: Search for license plates by prefix
    - Tests the search_by_prefix() method with various prefixes, including some that match several license plates (e.g., "ABC" and "XYZ") and some that match only one license plate (e.g., "D").
    - An edge case where the prefix doesn't exist in the Trie (e.g., "Z") is also tested.
3. Test Case 3: Edge case - Search for a non-existent prefix
    - Tests how the Trie handles searching for a prefix that does not exist (e.g., "NON"). This ensures that the Trie returns an empty list or None for non-existent prefixes.
4. Test Case 4: Deleting specific license plates

- Tests the delete() method by deleting specific license plates and verifying that the remaining plates are still in the Trie. This tests whether the Trie can correctly remove a plate and maintain its structure.

5. Test Case 5: Edge case - Try to delete a license plate that doesn't exist
    - Tests how the Trie handles an attempt to delete a license plate that doesn't exist (e.g., "NONEXISTENT"). This ensures that the deletion method doesn't break the Trie structure when attempting to remove a non-existent plate.

```
def test_trie():  1 usage  new *
    trie = Trie()

    # Test Case 1: Insert license plates into the Trie
    print("\n-- Test Case 1: Inserting license plates --")
    trie.insert("ABC123")
    trie.insert("ABC456")
    trie.insert("XYZ789")
    trie.insert("DEF123")
    trie.insert("ABX789")
    trie.insert("XYZ101")
    trie.insert("XYZ202")
    print("Inserted license plates.")
    # Print the Trie (for debugging)
    print("\n-- Initial Trie --")
    trie.print_trie()

    # Test Case 2: Search for license plates by prefix
    print("\n-- Test Case 2: Searching for license plates by prefix --")
    print("License plates starting with 'ABC':", trie.search_by_prefix("ABC"))
    print("License plates starting with 'XYZ':", trie.search_by_prefix("XYZ"))
    print("License plates starting with 'A':", trie.search_by_prefix("A"))
    print("License plates starting with 'D':", trie.search_by_prefix("D"))
    print("License plates starting with 'XY':", trie.search_by_prefix("XY"))
    print("License plates starting with 'Z':", trie.search_by_prefix("Z"))  # Edge case: No matching prefix

    # Test Case 3: Edge case - Search for a non-existent prefix
    print("\n-- Test Case 3: Searching for a non-existent prefix --")
    print("License plates starting with 'NON':", trie.search_by_prefix("NON"))  # Edge case: Non-existent prefix
```

```python
# Test Case 4: Deleting specific license plates
print("\n-- Test Case 4: Deleting specific license plates --")
print("-- Deleting 'ABC123' --")
trie.delete("ABC123")
print("Deleted 'ABC123'.")
print("License plates starting with 'ABC123' after deletion:", trie.search_by_prefix("ABC123"))
print("License plates starting with 'ABC' after deletion:", trie.search_by_prefix("ABC"))
print("-- Deleting 'XYZ101' --")
trie.delete("XYZ101")
print("License plates starting with 'XYZ101' after deletion:", trie.search_by_prefix("XYZ101"))
print("License plates starting with 'XYZ' after deletion:", trie.search_by_prefix("XYZ"))
trie.print_trie()

# Test Case 5: Edge case - Try to delete a license plate that doesn't exist
print("\n-- Test Case 5: Deleting a non-existent license plate --")
trie.delete("NONEXISTENT")
print("Tried to delete 'NONEXISTENT'.")
print("License plates starting with 'A' after attempting to delete non-existent plate:", trie.search_by_prefix("A"))
trie.print_trie()
```

```
-- Test Case 1: Inserting license plates --
Inserted license plates.

-- Initial Trie --
Prefix: ABC123, Plates: ['ABC123']
Prefix: ABC456, Plates: ['ABC456']
Prefix: ABX789, Plates: ['ABX789']
Prefix: XYZ789, Plates: ['XYZ789']
Prefix: XYZ101, Plates: ['XYZ101']
Prefix: XYZ202, Plates: ['XYZ202']
Prefix: DEF123, Plates: ['DEF123']

-- Test Case 2: Searching for license plates by prefix --
License plates starting with 'ABC': ['ABC123', 'ABC456']
License plates starting with 'XYZ': ['XYZ789', 'XYZ101', 'XYZ202']
License plates starting with 'A': ['ABC123', 'ABC456', 'ABX789']
License plates starting with 'D': ['DEF123']
License plates starting with 'XY': ['XYZ789', 'XYZ101', 'XYZ202']
License plates starting with 'Z': []

-- Test Case 3: Searching for a non-existent prefix --
License plates starting with 'NON': []
```

```
-- Test Case 4: Deleting specific license plates --
-- Deleting 'ABC123' --
Deleted 'ABC123'.
License plates starting with 'ABC123' after deletion: []
License plates starting with 'ABC' after deletion: ['ABC456']
-- Deleting 'XYZ101' --
License plates starting with 'XYZ101' after deletion: []
License plates starting with 'XYZ' after deletion: ['XYZ789', 'XYZ202']
Prefix: ABC456, Plates: ['ABC456']
Prefix: ABX789, Plates: ['ABX789']
Prefix: XYZ789, Plates: ['XYZ789']
Prefix: XYZ202, Plates: ['XYZ202']
Prefix: DEF123, Plates: ['DEF123']

-- Test Case 5: Deleting a non-existent license plate --
Tried to delete 'NONEXISTENT'.
License plates starting with 'A' after attempting to delete non-existent plate: ['ABC456', 'ABX789']
Prefix: ABC456, Plates: ['ABC456']
Prefix: ABX789, Plates: ['ABX789']
Prefix: XYZ789, Plates: ['XYZ789']
Prefix: XYZ202, Plates: ['XYZ202']
Prefix: DEF123, Plates: ['DEF123']
```

3. Most of my main challenges revolves around seeing how the data structure looks like. To combat these challenges, I simply created utility function such as printing the entire data structures. Fortunately, python 3 has a built-in function for Dictionary structure so I did not have to implement that. But for the priority queue, AVL Tree and Trie data structure I had to implement my own. Here is how they look like:

Priority queue:

```python
    # Utility function to check the contents of the heap (for debugging)
    def print_heap(self):  6 usages  Brendan Yeong
        print("Current Expiration Heap:")
        for exp_date, plate in self.expiration_heap:
            print(f"License Plate: {plate}, Expiration Date: {exp_date.strftime('%Y-%m-%d')}")
```

AVL Tree:

```
# Utility function to print the tree (for debugging)
def pre_order(self, root):  6 usages   Brendan Yeong
    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:

```
# Utility function to print the Trie (for debugging)
def print_trie(self, node=None, prefix=""):  4 usages   new *
    if node is None:
        node = self.root
    if node.is_end_of_plate:
        print(f"Prefix: {prefix}, Plates: {node.plates}")
    for char, child in node.children.items():
        self.print_trie(child, prefix + char)
```

Another Challenge that I faced is that I had to come up with a way to update the priority queue expiration dates. Here are the steps on how I did it.

To update the expiration date for a vehicle in the Priority Queue (Heap), you need to:

1. Remove the old expiration entry: Since heaps (priority queues) do not allow direct modification of elements, you first need to remove the old expiration entry for the vehicle from the heap.
2. Insert the updated expiration entry: After removing the old entry, insert a new one with the updated expiration date.

```
# Update the expiration date for a given vehicle
def update_registration(self, license_plate, new_expiration_date):  3 usages (1 dynamic)   new *
    # Step 1: Remove the old registration entry
    removed = self.remove_registration(license_plate)
    if removed:
        # Step 2: Add the new registration with the updated expiration date
        self.add_registration(license_plate, new_expiration_date)
        print(f"Updated expiration date for {license_plate} to {new_expiration_date}.")
    else:
        print(f"Failed to update: License plate {license_plate} not found.")
```

The next steps to complete the full implementation of your application. I simply did some mockup of the system. Here are the current implementations of the choices that will be given.

1. Add vehicle: We need to have an option for the user to add in new registration vehicles and ensuring all the data structures gets the information added into.
2. Search plate by Prefix: Have a way for the user to search plate by prefix and returns a list of plates.
3. Update the expiration date: We need a way to also update the registration expiration dates.
4. Remove vehicle: We need to be able to also remove any vehicle in the data structure.
5. Display all registration: We need to also have an option to display out all the vehicle registration that is stored in the dictionary.
6. Find vehicles by driver's license: We need an option where we can search a list of vehicles using the owner's driver's license number.
7. Exit: To simply kill or stop the simulation of this system.

```python
if __name__ == '__main__':
    car_system = VehicleRegistrationSystem()
    trie = Trie()
    heap = ExpirationData()
    avl_tree = AVLTree()

    while True:
        choice = main_menu()

        if choice == '1':
            add_vehicle(trie, heap, car_system, avl_tree)
        elif choice == '2':
            search_by_prefix(trie)
        elif choice == '3':
            update_expiration_date(heap, car_system)
        elif choice == '4':
            remove_vehicle(trie, heap, car_system, avl_tree)
        elif choice == '5':
            get_next_expiring_vehicle(heap)
        elif choice == '6':
            display_all_registrations(car_system)
        elif choice == '7':
            find_vehicles_by_license(avl_tree)
        elif choice == '8':
            print("Exiting system...")
            break
        else:
            print("Invalid choice, please try again.")
```

To ensure the user does not enter any wrong format or values here are some steps that I have placed in the add vehicle functions.

Summary of Validations:

1. First Name & Last Name: Must contain only alphabetic characters.
2. Color: Must contain only alphabetic characters.
3. VIN Number: Must contain only numeric characters.
4. Year: Must be a valid year between 1886 and the current year (numeric).
5. Date Fields (registration_date and expiration_date): Must follow the YYYY-MM-DD format.

```
    # Input validation for VIN number (only numeric characters)
    def get_valid_digit(prompt):  new *
        while True:
            vin = input(prompt)
            if vin.isdigit():
                return vin
            else:
                print("Invalid input. Please enter only numeric characters for the VIN number.")

    vin_number = get_valid_digit("Enter vehicle VIN number: ")

    print("\n---- Enter Owner Registration ----")
    first_name = get_valid_alpha( prompt: "Enter owner's first name: ",  field_name: "first name.")
    last_name = get_valid_alpha( prompt: "Enter owner's last name: ",  field_name: "last name.")

    license_number = input("Enter owner's license number: ")

    # Input validation for date format
    def get_valid_date(prompt):  new *
        while True:
            date = input(prompt)
            try:
                # Try to parse the date in YYYY-MM-DD format
                datetime.datetime.strptime(date, "%Y-%m-%d")
                return date
            except ValueError:
                print("Invalid date format. Please enter the date in YYYY-MM-DD format.")

    print("\n---- Registration Details ----")
    registration_date = get_valid_date("Enter registration date (YYYY-MM-DD): ")
    expiration_date = get_valid_date("Enter expiration date (YYYY-MM-DD): ")
```

```
# We need to add all the details when a new vehicle is added.
def add_vehicle(trie, heap, car_system, avl_tree):  2 usages (1 dynamic)  new *
    print("\n---- Add Vehicle Registration ----")
    license_plate = input("Enter license plate: ")
    make = input("Enter vehicle make: ")
    model = input("Enter vehicle model: ")

    # Input validation for vehicle year (only numeric and within a valid range)
    def get_valid_year(prompt):  new *
        while True:
            year = input(prompt)
            if year.isdigit() and 1886 <= int(year) <= datetime.datetime.now().year:  # First car invented in 1886
                return int(year)
            else:
                print(f"Invalid input. Please enter a valid year between 1886 and {datetime.datetime.now().year}.")

    year = get_valid_year("Enter vehicle year: ")

    # Input validation for first name and last name
    def get_valid_alpha(prompt, field_name):  new *
        while True:
            text = input(prompt)
            if text.isalpha():
                return text
            else:
                print("Invalid input. Please enter only alphabetic characters for the " + field_name)

    color = get_valid_alpha( prompt: "Enter vehicle color: ",  field_name: "color.")
    classification = get_valid_alpha( prompt: "Enter vehicle classification: ",  field_name: "classification")
```

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.