lab : Used Car Lot

In this lab, you'll have the opportunity to practice:

- Defining classes in Python
- Overloading the `==`, `<`, and `>` operators in a Python class
- Implementing and applying the Binary Search Tree (BST) data structure
- Writing functions that ensure Car objects are in sorted order
- Testing your functionality with pytest

Note: This lab will be broken down into two separate labs. It is important that you start this lab early so you can utilize our lab sections / office hours to seek assistance / ask clarifying questions during the weekdays before the deadline if needed as the first part will be used in the second part!

Introduction

The goal for this lab is to write a program that will manage cars for a used car dealership. All cars have a make, model, year, and price, which can be used to determine the value of cars in relation to each other. All cars will be managed by a Binary Search Tree (BST) where the nodes are sorted by make, then model in lexicographical order. Within each make/model node, the `Car` objects will be added to a Python List based on insertion order.

In order to manage the cars for this lab, you will define `Car`, `CarInventoryNode` and `CarInventory` classes that organize the Cars in a BST data structure. Cars with the same make/model will be located in the same node, and appended to a list based on insertion order.

You will also write pytests in `testFile.py` illustrating your behavior works correctly. This lab writeup will provide some test cases for clarity, but the Gradescope autograder will run different tests shown here.

Instructions

You will need to create four files:

- `Car.py` - Defines a Car class. This class will assume all Cars have a `make`(str), `model`(str), `year`(int), and a `price`(int).
- `CarInventoryNode.py` - Defines a BST Node class containing all fields for a BST Node and a Python List collection of Car objects.
- `CarInventory.py` - Defines a CarInventory (BST) class that is an ordered collection of a Dealership's Cars. You can adapt the BST implementation shown in the textbook supporting the specifications in this lab.
- `testFile.py` - This file will contain your pytest functions that tests the overall correctness of your class definitions.

There will be no starter code for this assignment, but rather class descriptions and required methods are defined in the specification below.

You should organize your lab work in its own directory. This way all files for a lab are located in a single folder. Also, this will be easy to import various files into your code using the `import / from` technique shown in lecture.

Car.py

The `Car.py` file will contain the definition of a `Car` class. The `Car` class will hold information about the cars (`make`, `model`, `year`, and `price`). We will define the `Car` attributes as follows:

- `make` - string value representing the brand of the car (eg. `Nissan`, `Tesla`, `Toyota`, `Ferrari`). Your program must store this attribute in uppercase characters
- `model` - string value representing the model of the car (eg. `Electra`, `Model3`, `Prius`, `Portofino`). Your program must store this attribute in uppercase characters
- `year` - integer value representing the year of the car (eg. 2010, 2000, 1963)
- `price` - integer value representing the price value of the car (eg. 20000, 30000, 25000)

You will write a constructor that allows the user to construct a `Car` object by passing in values for the `make`, `model`, `year`, and `price`.

- `__init__(self, make, model, year, price)`

In addition to the constructor of the `Car` class, the following methods are required to be implemented:

- `__gt__(self, rhs)` - comparator operator that allows us to check if a `Car` object is greater than another `Car` object. `Car` objects are first organized by the lexicographical/alphabetical order of their `make` attribute. If the `make` attribute is the same, then they'll be determined by the lexicographical/alphabetical order of their `model` attribute. If the `model` attribute is the same, then they are organized by the year (from least-to-greatest). If the year is the same, then the they are organized by their price (from least-to-greatest). For example, if both `Car2` and `Car1` have the same `make` and `model`, then `Car2 > Car1` if `Car2` is newer; if `Car2` and `Car1` have the same `year` as well, then `Car2 > Car1` if `Car2` is more expensive.
- `__lt__(self, rhs)` - comparator operator that allows us to check that a `Car` object is less than another `Car` object via the operation `Car1 < Car2` according to the specifications above.
- `__eq__(self, rhs)` - comparator operator that allows us to check that a `Car` object is equivalent to another `Car` (both cars have the same `make`, `model`, `year`, and `price`) via the operation `Car1 == Car2`.

- `__str__(self)` - overload method that returns the details of a car via the operation `str(Car1)`. The string representation should fit the format: `"Make: [make], Model: [model], Year: [year], Price: $[price]"`.

For example:

```
c = Car("Honda", "CRV", 2007, 8000)
print(c)
```

Output:

```
Make: HONDA, Model: CRV, Year: 2007, Price: $8000
```

CarInventoryNode.py

The `CarInventoryNode.py` file will contain the definition of a `CarInventoryNode` class, which is the node for a BST.

The `CarInventoryNode` class should have the following attributes:

- `make` - string value that represents the make of the car; just like in the `Car` class, your program must store this attribute in uppercase characters.
- `model` - string value that represents the model of the car; just like in the `Car` class, your program must store this attribute in uppercase characters.
- `cars` - a Python List that contains `Car` objects that should have the same `make` and `model`. `Car` objects will be organized in insertion order (most recently inserted `Car` will exist at the end of the Python List).
- `parent` - a reference to the parent of a `CarInventoryNode`, `None` if it has no parent (it is the root).
- `left` - a reference to the left child of a `CarInventoryNode`, `None` if it has no left child.
- `right` - a reference to the right child of a `CarInventoryNode`, `None` if it has no right child.

The `CarInventoryNode` class should define the following methods:

- `__init__(self, car)` - the constructor for the `CarInventoryNode` will take in a `Car` object, and initialize all attributes in the `CarInventoryNode`. The constructor should also append the parameter `Car` object from the parameter into the list attribute `cars`.

In addition to the construction of the BST in this class, the following methods are required to be implemented (note, some methods may be helpful for next week's lab):

- `getMake(self)` - returns a string containing the `make` of a `CarInventoryNode`.
- `getModel(self)` - returns a string containing the `model` of a `CarInventoryNode`.
- `getParent(self)` - returns the `parent` Node of the `CarInventoryNode`. If the parent does not exist, return `None`.
- `setParent(self, parent)` - sets the `parent` Node of the `CarInventoryNode`.
- `getLeft(self)` - returns the left child of the `CarInventoryNode`. If the left child does not exist, return `None`.
- `setLeft(self, left)` - sets the left child of the `CarInventoryNode`.
- `getRight(self)` - returns the right child of the `CarInventoryNode`. If the right child does not exist, return `None`.
- `setRight(self, right)` - sets the right child of the `CarInventoryNode`.
- `__str__(self)` - overload the string operator to allow us to get the details of all cars in the `CarInventoryNode` (eg, `str(CarNode1)`). The string representation should contain all the `Car` objects in this `CarInventoryNode` in insertion order. Make use of the `Car` class' `__str__` method, and separate each car with a newline character (`\n`) (including the last `Car` object in the cars Python List).

For example:

```
car1 = Car("Dodge", "dart", 2015, 6000)
car2 = Car("dodge", "DaRt", 2003, 5000)
carNode = CarInventoryNode(car1)
carNode.cars.append(car2)
print(carNode)
```

Output: (note the extra newline)

```
Make: DODGE, Model: DART, Year: 2015, Price: $6000
Make: DODGE, Model: DART, Year: 2003, Price: $5000

```

Some tips for the implementation:

`CarInventoryNode`s are ordered by `make` and `model` attributes. You *can* implement comparators (`<`, `==`, `>`) for `CarInventoryNode` to help with navigating through the `CarInventory`, but this is not required.

CarInventory.py

The `CarInventory.py` file will contain the definition of a `CarInventory` class. This will manage the `CarInventoryNode`s and keep track of all the cars a dealership has. The `CarInventory` is implemented as a BST. The `CarInventory` will create and manage `CarInventoryNode` objects based on a car's `make` and `model` attributes. When storing `Car` objects in the `CarInventory`, `Car` objects with the same `make` and `model` will be appended to a Python List based on insertion order within the `CarInventoryNode` object.

- `__init__(self)` - the constructor for the `CarInventory` will simply initialize the empty BST. You should have an attribute called `root` to represent the root node of the `CarInventory` BST.

In addition to the construction of the BST in this class, the following methods are required to be implemented:

- `addCar(self, car)` - adds a `Car` object to the BST. If a `CarInventoryNode` with the same `make` and `model` exists, then append the car to the end of its car list.
- `doesCarExist(self, car)` - searches for a `Car` object in the `CarInventory` by traversing to the appropriate `CarInventoryNode` (if it exists), and checking the `cars` Python List to see if any `Car` object is equal to the parameter `car`. This method returns `True` if it does, and returns `False` otherwise (i.e. no `Car` object with the same `make`, `model`, `year`, and `price` exists).
- `inOrder(self)` - returns a string with the in-order traversal of the BST. Printing the in-order traversal should help check that the cars are in the correct order in the BST
- `preOrder(self)` - returns a string with the pre-order traversal of the BST. BSTs with the same structure should always have the same pre-order traversal, so this can be used to verify that everything was inserted correctly
- `postOrder(self)` - returns a string with the post-order traversal of the BST.
- `getBestCar(self, make, model)` - returns the `Car` with the newest year - and if multiple, then highest price - given the make and model. If the make and model doesn't exist, then return None.
- `getWorstCar(self, make, model)` - returns the car with the oldest year - and if multiple, then lowest price - given the make and model. If the make and model doesn't exist, then return None.
- `getTotalInventoryPrice(self)` - returns an integer the total price of all the cars in the dealership.

Given an example BST:

```
bst = CarInventory()

car1 = Car("Nissan", "Leaf", 2018, 18000)
car2 = Car("Tesla", "Model3", 2018, 50000)
car3 = Car("Mercedes", "Sprinter", 2022, 40000)
car4 = Car("Mercedes", "Sprinter", 2014, 25000)
car5 = Car("Ford", "Ranger", 2021, 25000)

bst.addCar(car1)
bst.addCar(car2)
bst.addCar(car3)
bst.addCar(car4)
bst.addCar(car5)
```

An example of the traversal functions is given below:

```
assert bst.getBestCar("Nissan", "Leaf") == car1
assert bst.getBestCar("Mercedes", "Sprinter") == car3
assert bst.getBestCar("Honda", "Accord") == None

assert bst.getWorstCar("Nissan", "Leaf") == car1
assert bst.getWorstCar("Mercedes", "Sprinter") == car4
assert bst.getBestCar("Honda", "Accord") == None

assert bst.getTotalInventoryPrice() == 158000
```

An example of the `inOrder()` string format is given below:

```
assert bst.inOrder() == \
"""\
Make: FORD, Model: RANGER, Year: 2021, Price: $25000
Make: MERCEDES, Model: SPRINTER, Year: 2022, Price: $40000
Make: MERCEDES, Model: SPRINTER, Year: 2014, Price: $25000
Make: NISSAN, Model: LEAF, Year: 2018, Price: $18000
Make: TESLA, Model: MODEL3, Year: 2018, Price: $50000
"""
```

An example of the `preOrder()` string format is given below:

```
assert bst.preOrder() == \
"""\
Make: NISSAN, Model: LEAF, Year: 2018, Price: $18000
Make: MERCEDES, Model: SPRINTER, Year: 2022, Price: $40000
Make: MERCEDES, Model: SPRINTER, Year: 2014, Price: $25000
Make: FORD, Model: RANGER, Year: 2021, Price: $25000
Make: TESLA, Model: MODEL3, Year: 2018, Price: $50000
"""
```

An example of the `postOrder()` string format is given below:

```
assert bst.postOrder() == \
"""\
Make: FORD, Model: RANGER, Year: 2021, Price: $25000
Make: MERCEDES, Model: SPRINTER, Year: 2022, Price: $40000
Make: MERCEDES, Model: SPRINTER, Year: 2014, Price: $25000
Make: TESLA, Model: MODEL3, Year: 2018, Price: $50000
Make: NISSAN, Model: LEAF, Year: 2018, Price: $18000
"""
```

Other than the required methods, feel free to implement any helper methods that you think are useful in your implementation (Gradescope will test the required methods). The automated tests will test your implementation of the required methods by creating a `CarInventory` containing various `CarInventoryNode`s containing `Car`s with different `make`, `model`, `year`, and `price` attributes. The `addCar()` method will be run, with `doesCarExist()`, `getBestCar()`, `getWorstCar()`, `inOrder()`, `preOrder()`, and `postOrder()`, etc. being used to verify that the `CarInventory` is fully functional. You should write similar tests to confirm your BST is working properly.

testFile.py

This file should test all of your classes and required methods using pytest. Think of various scenarios and edge cases when testing your code according to the given descriptions. You should test every class' method functionality (except for getters / setters). Even though Gradescope will not use this file when running automated tests (there are separate tests defined for this), it is important to provide this file with various test cases (testing is important!!).

A note about Gradescope tests: Gradescope will run many edge cases, so it's important to thoroughly test as many different cases you can think of. Gradescope will also use your functions to correctly check the state of your Cars and CarInventory with many scenarios. In order to test if everything is in the correct state, these tests use your CarInventory's `preOrder` / `inOrder` / `postOrder` traversals and `addCar` methods, as well as getting the string representation of your Cars (using your `__str__` overloaded method in `Car`) to run other tests such as CarInventory's `doesCarExist`, `getBestCar`, `getWorstCar`, `getTotalAssets`, etc.

It is important to ensure your `preOrder` / `inOrder` / `postOrder` traversals, Car's `__str__` method, and `CarInventory`'s `addCar` methods work correctly first or else many of the other tests may not pass.

Of course, feel free to reach out / post questions on Piazza as they come up!

How to best approach this lab

This lab contains a lot of implementation details, and different parts of the lab intertwine and depend on each other. Here are some suggestions on the order of approaching the lab:

1. Implement the `Car` class, and especially double check that your comparators are working
2. Implement the `CarInventoryNode` class. You should go through and test your `__str__` overloading before moving on
3. Start with `addCar`, and then the BST traversal methods. You should test to see if your `addCar` is working by inserting several Cars into the BST, and using the traversals to verify your results; is a new `CarInventoryNode` being created if the Car didn't exist in the BST, and if the node did already exist, is the Car being inserted to the list of cars in the node?
4. Once you've made sure your `addCar` is working, you can then move on to `doesCarExist`, `getBestCar` and `getWorstCar`.
5. Testing is extremely important to help debug any issues you may experience. Be sure to write thorough tests with various edge cases to make sure your program works as expected.

Submission

Once you're done with writing your class definitions and tests, submit the following files to Gradescope's Lab08 assignment:

- `Car.py`
- `CarInventoryNode.py`
- `CarInventory.py`
- `testFile.py`

There will be various unit tests Gradescope will run to ensure your code is working correctly based on the specifications given in this lab.

If the tests don't pass, you may get some error message that may or may not be obvious at this point. Don't worry - if the tests didn't pass, take a minute to think about what may have caused the error. If your tests didn't pass and you're still not sure why you're getting the error, feel free to ask your TAs or Learning Assistants.

lab : Used Car Lot - Part Two

In this lab, you'll have the opportunity to practice:

- Modifying classes in Python
- Further implementing Binary Search Tree (BST) data structure supporting removal functionality
- Testing your functionality with pytest

Note: This lab will be dependent on your previous lab. Certain tests from the previous lab will be autograded in this week's lab. It is important that you start this lab early so you can utilize our office hours to seek assistance / ask clarifying questions during the weekdays before the deadline if needed!

Introduction

The goal for this lab is to take your existing Used Car Lot program in Lab08 that will manage cars for a second-hand car dealership, and support removing Cars from the lot. As a reminder, all Cars have a `make`, `model`, `year`, and `price`, which can be used to determine the value of cars in relation to each other. All Cars will be managed with a Binary Search Tree (BST) where the BST nodes are sorted by `make`, then `model`.

In order to remove the cars for this lab, you will define a `removeCar` method in the `CarInventory` class that will remove Cars with the same `make`/`model`/`year`/`price` from a `CarInventoryNode`'s cars list. After removing a Car and no cars exist in the `CarInventoryNode`'s cars list, you will then need to remove the node from the BST while preserving the BST property.

You will also write pytests in `testFile.py` illustrating your behavior works correctly. This lab writeup will provide some test cases for clarity, but the Gradescope autograder will run different tests shown here. It's important to thoroughly test your program with various cases!

Instructions

You will need to copy over all your files from Lab08 and modify two files:

- `CarInventory.py` - Defines a CarInventory (BST) class that is an ordered collection of a Dealership's Cars. You will be adding to your existing `CarInventory` class.
- `testFile.py` - This file will contain your pytest functions that tests the overall correctness of your class definitions.

Your starter code for this assignment will be your program from Lab08, and you'll have to add the additional specifications defined below.

You should organize your lab work in its own directory. This way all files for a lab are located in a single folder. Also, this will be easy to import various files into your code using the `import / from` technique shown in lecture.

CarInventory.py

The `CarInventory.py` file will contain the definition of a `CarInventory` class. This will keep track of the cars a dealership has, implemented as a BST. The `CarInventory` will create `CarInventoryNode` objects using `Car` objects based on their `make` and `model`. `Car` objects with the same make and model will be appended to a list based on insertion order within the `CarInventoryNode` object. For further specifications regarding existing requirements, reference the Lab08 page.

In addition to the methods created before, the following methods are required to be implemented:

- `getSuccessor(self, make, model)` - attempts to finds the `CarInventoryNode` with the `make` and `model`, and returns the CarInventoryNode with the next greatest value (using the same heirarchy of `make`, then `model`). Returns `None` if there is no `CarInventoryNode` with the specified `make` and `model`, or if the `CarInventoryNode` is the maximum and has no successor. **Note, this includes the successor of any `CarInventoryNode` in the BST if it exists, not just the successor used for BST maintenance.**
- `removeCar(self, make, model, year, price)` - attempts to find the `Car` with the specified `make`, `model`, `year`, and `price`, and removes it the `CarInventoryNode`'s cars list. If the list is empty after removing the `Car`, remove the `CarInventoryNode` from the BST entirely. Returns `True` if the `Car` was successfully removed, and `False` if the `Car` is not present in the `CarInventory`. If there are duplicate cars within a `CarInventoryNode`'s car list that matches the specifications, you will just remove the first matching `Car` object in the cars list.

A note if you have implemented `CarInventoryNode` comparators: If you have implemented `CarInventoryNode` comparators in last week's lab, in your `__eq__` comparator overload, before you check for the `make` and the `model`, you should check if the right-hand-side is `None`. If it is `None`, you should return `False`. This is because of a quirk about how Python handles comparators between overloaded comparators and `None`.

Examples

Given an example BST:

```
bst = CarInventory()

car1 = Car("Mazda", "CX-5", 2022, 25000)
car2 = Car("Tesla", "Model3", 2018, 50000)
car3 = Car("BMW", "X5", 2022, 60000)
car4 = Car("BMW", "X5", 2020, 58000)
car5 = Car("Audi", "A3", 2021, 25000)

bst.addCar(car1)
bst.addCar(car2)
bst.addCar(car3)
bst.addCar(car4)
bst.addCar(car5)

#                                 Mazda,CX-5,[Car(Mazda,CX-5,2022,25000)]
#                                /                                       \
#           BMW,X5,[Car(BMW,X5,2022,60000),Car(BMW,X5,2020,58000)]    Tesla,Model3,[Car(Tesla, Model3,2018,50000)]
#                    /
#  Audi,A3,[Car(Audi,A3,2021,25000)]
```

InOrder Traversal

Using the `CarInventory` after the `addCar` methods above, an example of the `inOrder()` string format for removal is given below after removing the following Car:

```
bst.removeCar("BMW", "X5", 2020, 58000)

#                                 Mazda,CX-5,[Car(Mazda,CX-5,2022,25000)]
#                                /                                       \
#           BMW,X5,[Car(BMW,X5,2022,60000)]    Tesla,Model3,[Car(Tesla,Model3,2018,50000)]
#                    /
#  Audi,A3,[Car(Audi,A3,2021,25000)]

assert bst.inOrder() == \
"""\
Make: AUDI, Model: A3, Year: 2021, Price: $25000
Make: BMW, Model: X5, Year: 2022, Price: $60000
Make: MAZDA, Model: CX-5, Year: 2022, Price: $25000
Make: TESLA, Model: MODEL3, Year: 2018, Price: $50000
"""
```

and if we then remove the following car, the `CarInventoryNode` will be removed from the BST. The `CarInventory` and `inOrder()` string format is given below in this case:

```
bst.removeCar("BMW", "X5", 2022, 60000)

#                                 Mazda,CX-5,[Car(Mazda,CX-5,2022,25000)]
#                                /                                       \
#           Audi,A3,[Car(Audi,A3,2021,25000)]    Tesla,Model3,[Car(Tesla,Model3,2018,50000)]


assert bst.inOrder() == \
"""\
Make: AUDI, Model: A3, Year: 2021, Price: $25000
Make: MAZDA, Model: CX-5, Year: 2022, Price: $25000
Make: TESLA, Model: MODEL3, Year: 2018, Price: $50000
"""
```

PreOrder Traversal

Using the `CarInventory` after the `addCar` methods above, an example of the `preOrder()` string format is given below after removing the following Cars:

```
bst.removeCar("BMW", "X5", 2020, 58000)

#                                 Mazda,CX-5,[Car(Mazda,CX-5,2022,25000)]
#                                /                                       \
#           BMW,X5,[Car(BMW,X5,2022,60000)]    Tesla,Model3,[Car(Tesla,Model3,2018,50000)]
#                    /
#  Audi,A3,[Car(Audi,A3,2021,25000)]

assert bst.preOrder() == \
"""\
Make: MAZDA, Model: CX-5, Year: 2022, Price: $25000
Make: BMW, Model: X5, Year: 2022, Price: $60000
Make: AUDI, Model: A3, Year: 2021, Price: $25000
Make: TESLA, Model: MODEL3, Year: 2018, Price: $50000
"""

bst.removeCar("BMW", "X5", 2022, 60000)

#                                 Mazda,CX-5,[Car(Mazda,CX-5,2022,25000)]
#                                /                                       \
#           Audi,A3,[Car(Audi,A3,2021,25000)]    Tesla,Model3,[Car(Tesla,Model3,2018,50000)]


assert bst.preOrder() == \
"""\
Make: MAZDA, Model: CX-5, Year: 2022, Price: $25000
Make: AUDI, Model: A3, Year: 2021, Price: $25000
Make: TESLA, Model: MODEL3, Year: 2018, Price: $50000
"""
```

PostOrder Traversal

Using the `CarInventory` after the `addCar` methods above, an example of the `postOrder()` string format is given below after removing the following Cars:

```
bst.removeCar("BMW", "X5", 2020, 58000)

#                                 Mazda,CX-5,[Car(Mazda,CX-5,2022,25000)]
#                                /                                       \
#           BMW,X5,[Car(BMW,X5,2022,60000)]    Tesla,Model3,[Car(Tesla,Model3,2018,50000)]
#                    /
#  Audi,A3,[Car(Audi,A3,2021,25000)]

assert bst.postOrder() == \
"""\
Make: AUDI, Model: A3, Year: 2021, Price: $25000
Make: BMW, Model: X5, Year: 2022, Price: $60000
Make: TESLA, Model: MODEL3, Year: 2018, Price: $50000
Make: MAZDA, Model: CX-5, Year: 2022, Price: $25000
"""

bst.removeCar("BMW", "X5", 2022, 60000)

#                                 Mazda,CX-5,[Car(Mazda,CX-5,2022,25000)]
#                                /                                       \
#           Audi,A3,[Car(Audi,A3,2021,25000)]    Tesla,Model3,[Car(Tesla,Model3,2018,50000)]


assert bst.postOrder() == \
"""\
Make: AUDI, Model: A3, Year: 2021, Price: $25000
Make: TESLA, Model: MODEL3, Year: 2018, Price: $50000
Make: MAZDA, Model: CX-5, Year: 2022, Price: $25000
"""
```

These are just a few simple examples illustrating the functionality of removing a Car from the `CarInventory` cars list, and removing the `CarInventoryNode` from the `CarInventory`. Gradescope will thoroughly test various cases. As always, it's important to thoroughly test your own code with various possible cases.

Other than the required methods, feel free to implement any helper methods that you think are useful in your implementation. The automated tests will test only your implementation of the required methods and certain methods from last week by creating a `CarInventory` containing various `Cars` with different `make`, `model`, `year`, and `price` attributes. The `removeCar()` and `addCar()` methods will be run, with `getCar()`, `getSuccessor()`, `inOrder()`, `preOrder()`, and `postOrder()` being used to verify that the `CarInventory` is fully functional. You should be sure that Lab08 is working correctly, and write tests to confirm your program for this lab is working properly.

testFile.py

This file should test all of the new methods in `CarInventory.py` using pytest. Think of and create your own various scenarios and edge cases when testing your code according to the given descriptions. For the `getSuccessor` method, your tests should test the general case and the case used for BST maintenance. For the `removeCar` method, you tests should cover Cases 1, 2, and 3 (as discussed in lecture) at a minimum, as well as only removing a Car from the `CarInventoryNode`'s cars list without removing the `CarInventoryNode`. Even though Gradescope will not use this file when running automated tests (Gradescope will use other tests), it is important to provide this file with various test cases (testing is important!!).

A note about Gradescope tests: Gradescope will use your functions to correctly check the state of your `Car`s and `CarInventory` with many scenarios. In order to test if everything is in the correct state, these tests use your CarInventory's `preOrder` / `inOrder` / `postOrder` traversals and `addCar` methods, as well as getting the string representation of your `Car`s and `CarInventoryNode`s to run other tests. It is important to ensure your `preOrder` / `inOrder` / `postOrder` traversals, your various string representations, and `CarInventory`'s `addCar` methods work correctly first or else many of the other

tests may not pass.

Of course, feel free to reach out / post questions on Piazza as they come up!

Submission

Once you're done with writing your class definitions and tests, submit the following files to Gradescope's Lab09 assignment:

- `Car.py`
- `CarInventoryNode.py`
- `CarInventory.py`
- `testFile.py`

There will be various unit tests Gradescope will run to ensure your code is working correctly based on the specifications given in this lab.

If the tests don't pass, you may get some error message that may or may not be obvious at this point. Don't worry - take a minute to think about what may have caused the error. Try writing more test cases to see if you're able to reproduce the problem. If you're still not sure why you're getting the error, feel free to ask your TAs or Learning Assistants.

My work below:

Car.py

```python
class Car:

    def __init__(self, make, model, year, price):
        self.make = make.upper()
        self.model = model.upper()
        self.year = year
        self.price = price

    def __eq__(self, rhs):
        if self.make == rhs.make.upper():
            if self.model == rhs.model.upper():
                if self.year == rhs.year:
                    if self.price == rhs.price:
                        return True
        return False

    def __gt__(self, rhs):
        if self.make > rhs.make.upper():
            return True
        elif self.make < rhs.make.upper():
            return False

        if self.model > rhs.model.upper():
            return True
        elif self.model < rhs.model.upper():
            return False

        if self.year > rhs.year:
            return True
        elif self.year < rhs.year:
            return False

        if self.price > rhs.price:
            return True
        elif self.price < rhs.price:
            return False

        return False

    def __str__(self):
        return f"Make: {self.make}, Model: {self.model}, Year: {self.year}, Price: ${self.price}"
```

CarInventoryNode.py

```python
from Car import Car

class CarInventoryNode:

    def __init__(self, car):
        self.make = car.make
        self.model = car.model
        self.cars = [car]
        self.parent = None
        self.left = None
        self.right = None

    def getMake(self):
        return self.make

    def getModel(self):
        return self.model

    def getParent(self):
        return self.parent

    def getLeft(self):
        return self.left

    def getRight(self):
        return self.right

    def setParent(self, parent):
        self.parent = parent

    def setLeft(self, left):
        self.left = left

    def setRight(self, right):
        self.right = right

    def __str__(self):
        result = ""
        for i in self.cars:
            result += str(i) + "\n"
        return result

    def isLeft(self):
        return self.parent and self.parent.left == self

    def isRight(self):
        return self.parent and self.parent.right == self

    def isRoot(self):
        return not self.parent

    def isLeaf(self):
        return not (self.right or self.left)

    def hasAny(self):
        return self.right or self.left

    def hasBoth(self):
        return self.right and self.left

    def replaceNodeData(self, node):
        self.make = node.make
        self.model = node.model
        self.cars = node.cars
        if self.left == node or self.right == node:
            self.left = node.left
            self.right = node.right
        if self.getLeft():
            self.left.parent = self
        if self.getRight():
            self.right.parent = self

    def spliceOut(self):
        # Case 1: Node is leaf
        if self.isLeaf():
            if self.isLeft():
                self.parent.left = None
            else:
                self.parent.right = None

        # Case 2: Node have one child
        elif self.hasAny():
            if self.getRight():
                if self.isLeft():
                    self.parent.left = self.right
                else:
                    self.parent.right = self.right
                self.right.parent = self.parent
```

CarInventory.py

```python
from CarInventoryNode import *

class CarInventory:

    def __init__(self):
        self.root = None

    # add Car class to CarInventory
    def addCar(self, car):
        if self.root:
            self._add(car, self.root)
        else:
            self.root = CarInventoryNode(car)

    def _add(self, currentCar, currentNode):
        if currentCar.make == currentNode.make:
            if currentCar.model == currentNode.model:
                currentNode.cars.append(currentCar)
            elif currentCar.model < currentNode.model:
                self._addLeft(currentCar, currentNode)
            else: # currentCar.model > currentNode.model
                self._addRight(currentCar, currentNode)
        elif currentCar.make < currentNode.make:
            self._addLeft(currentCar, currentNode)
        else: # currentCar.make > currentNode.make
            self._addRight(currentCar, currentNode)

    def _addLeft(self, currentCar, currentNode):
        if currentNode.getLeft():
            self._add(currentCar, currentNode.left)
        else:
            currentNode.left = CarInventoryNode(currentCar)
            currentNode.left.parent = currentNode

    def _addRight(self, currentCar, currentNode):
        if currentNode.getRight():
            self._ add(currentCar, currentNode.right)
        else:
            currentNode.right = CarInventoryNode(currentCar)
            currentNode.right.parent = currentNode

    # find Car class in CarInventory
    def doesCarExist(self, car):
        if self.root:
            node = self._get(car.make, car.model, self.root)
            if node:
                for i in node.cars:
                    if car.year == i.year and car.price == i.price:
                        return True
                return False
            else:
                return False
        else:
            return False

    def _get(self, make, model, currentNode):
        # not exist
        if not currentNode:
            return None
        # found, return this Node
        elif make == currentNode.make and model == currentNode.model:
            return currentNode

        # continue traversal inventory
        if make < currentNode.make:
            return self._get(make, model, currentNode.left)
        elif make > currentNode.make:
            return self._get(make, model, currentNode.right)
        else: # make == currentNode.make
            if model < currentNode.model:
                return self._get(make, model, currentNode.left)
            elif model > currentNode.model:
                return self._get(make, model, currentNode.right)

    def inOrder(self):
        return self._inOrder(self.root)

    def  inOrder(self, currentNode):
        result = ""
        if currentNode:
            result += self._inOrder(currentNode.left)
            result += str(currentNode)
            result += self._inOrder(currentNode.right)
        return result

    def preOrder(self):
        return self._preOrder(self.root)

    def _preOrder(self, currentNode):
        result = ""
        if currentNode:
            result += str(currentNode)
            result += self._preOrder(currentNode.left)
            result += self._preOrder(currentNode.right)
        return result

    def postOrder(self):
        return self._postOrder(self.root)

    def _postOrder(self, currentNode):
        result = ""
        if currentNode:
            result += self._postOrder(currentNode.left)
            result += self._postOrder(currentNode.right)
            result += str(currentNode)
        return result

    # find best car by make and model
    def getBestCar(self, make, model):
        make = make.upper()
        model = model.upper()

        if self.root:
            node = self._get(make, model, self.root)
            if node:
```

```
                bestCar = node.cars[0]
                for i in node.cars:
                    if i > bestCar:
                        bestCar = i
                return bestCar
        return None

    # find worst car by make and model
    def getWorstCar(self, make, model):
        make = make.upper()
        model = model.upper()

        if self.root:
            node = self._get(make, model, self.root)
            if node:
                WorstCar = node.cars[0]
                for i in node.cars:
                    if i < WorstCar:
                        WorstCar = i
                return WorstCar
        return None

    # calculate total inventory value
    def getTotalInventoryPrice(self):
        return self._total(self.root)

    def _total(self, currentNode):
        total = 0
        if currentNode:
            total += self._total(currentNode.left)
            for i in currentNode.cars:
                total += i.price
            total += self._total(currentNode.right)
        return total

    # remove car by make, model, year, and price
    def removeCar(self, make, model, year, price):
        make = make.upper()
        model = model.upper()

        # No InventoryNode
        if self.root != None:
            nodeToRemove = self._get(make, model, self.root)
            if nodeToRemove:
                for i in nodeToRemove.cars:
                    if year == i.year and price == i.price:
                        nodeToRemove.cars.remove(i)

                        if len(nodeToRemove.cars) == 0:
                            # Only one InventoryNode
                            self._remove(nodeToRemove)
                        return True
        return False

    def _remove(self, currentNode):
        # Case 1: Node to remove is leaf
        if currentNode.isLeaf():
            if currentNode.isLeft():
                currentNode.parent.left = None
            elif currentNode.isRight():
                currentNode.parent.right = None
            else: # Node is root
                self.root = None

        # Case 3: Node to remove has both children
        elif currentNode.hasBoth():
            succ = self.getSuccessor(currentNode.make, currentNode.model)
            succ.spliceOut()
            currentNode.replaceNodeData(succ)

        # Case 2: Node to remove has one child
        else:
            if currentNode.getLeft():
                currentNode.replaceNodeData(currentNode.left)
            else:
                currentNode.replaceNodeData(currentNode.right)

    # find successor if needed
    def getSuccessor(self, make, model):
        make = make.upper()
        model = model.upper()
        nodeToRemove = self._get(make, model, self.root)
        succ = None

        if nodeToRemove:
            if nodeToRemove.getRight():
                succ = nodeToRemove.right
                while succ.getLeft():
                    succ = succ.left
            else:
                succ = nodeToRemove
                while succ:
                    if succ.cars[0] > nodeToRemove.cars[0]:
                        return succ
                    succ = succ.parent
                succ = None
        return succ
```

testFile.py

```
from CarInventory import *

def test_Car():
    a1 = Car("Dodge", "dart", 2025, 30000)
    a2 = Car("Ford", "Ranger", 2021, 25000)
    a3 = Car("Nissan", "Leaf", 2018, 18000)
    a4 = Car("Nissan", "Leaf", 2020, 18000)

    assert (a1 == a2) == False
    assert (a4 == a4) == True

    assert (a4 < a3) == False
    assert (a2 < a3) == True

    assert (a3 > a2) == True
    assert (a1 > a4) == False

def test_CarInventoryNode():
    b1 = Car("Dodge", "dart", 2025, 30000)
    b2 = Car("Ford", "Ranger", 2021, 25000)
    carNode = CarInventoryNode(b2)

    assert str(carNode) == \
    "Make: FORD, Model: RANGER, Year: 2021, Price: $25000\n"

    carNode.cars.append(b1)
    assert carNode.__str__() == \
    "Make: FORD, Model: RANGER, Year: 2021, Price: $25000\n\
Make: DODGE, Model: DART, Year: 2025, Price: $30000\n"

def test_CarInventory():
    c1 = Car("Dodge", "dart", 2025, 30000)
    c2 = Car("Nissan", "Leaf", 2018, 18000)
    c3 = Car("Nissan", "Leaf", 2020, 18000)
    c4 = Car("Tesla", "Model3", 2021, 50000)
    c5 = Car("Ford", "Ranger", 2021, 25000)

    d1 = CarInventory()
    d2 = CarInventory()

    d1.addCar(c1)
    d1.addCar(c2)

    assert d1.addCar(c3) == None
    assert d1.addCar(c5) == None

    assert d1.doesCarExist(c1) == True
    assert d1.doesCarExist(c4) == False

    assert d1.inOrder() == """\
Make: DODGE, Model: DART, Year: 2025, Price: $30000
Make: FORD, Model: RANGER, Year: 2021, Price: $25000
Make: NISSAN, Model: LEAF, Year: 2018, Price: $18000
Make: NISSAN, Model: LEAF, Year: 2020, Price: $18000
"""
    assert d2.inOrder() == ""

    assert d1.preOrder() == """\
Make: DODGE, Model: DART, Year: 2025, Price: $30000
Make: NISSAN, Model: LEAF, Year: 2018, Price: $18000
Make: NISSAN, Model: LEAF, Year: 2020, Price: $18000
Make: FORD, Model: RANGER, Year: 2021, Price: $25000
"""
    assert d2.preOrder() == ""

    assert d1.postOrder() == """\
Make: FORD, Model: RANGER, Year: 2021, Price: $25000
Make: NISSAN, Model: LEAF, Year: 2018, Price: $18000
Make: NISSAN, Model: LEAF, Year: 2020, Price: $18000
Make: DODGE, Model: DART, Year: 2025, Price: $30000
"""
    assert d2.postOrder() == ""

    assert d1.getBestCar("NISSAN", "LEAF") == c3
    assert d2.getBestCar("Tesla", "Model3") == None

    assert d1.getWorstCar("DODGE", "RANGER") == None
    assert d1.getWorstCar("FORD", "RANGER") == c5

    assert d1.getTotalInventoryPrice() == 91000
    assert d2.getTotalInventoryPrice() == 0


    d3 = CarInventory()
    car1 = Car("Mazda", "CX-5", 2022, 25000)
    car2 = Car("Tesla", "Model3", 2018, 50000)
    car3 = Car("Benz", "E300", 2022, 60000)
    car4 = Car("Benz", "E300", 2020, 58000)
    car5 = Car("Audi", "A3", 2021, 25000)

    d3.addCar(car1)
    d3.addCar(car2)
    d3.addCar(car3)
    d3.addCar(car4)
    d3.addCar(car5)

    assert d3.getSuccessor("Mazda", "CX-5") == d3.root.right

    assert d3.removeCar("Mazda", "CX-5", 2022, 25000) == True
    assert d3.removeCar("Tesla", "Model3", 2018, 50000) == True

    assert d3.getSuccessor("Benz", "E300") == None

    assert d3.preOrder() == "\
Make: BENZ, Model: E300, Year: 2022, Price: $60000\n\
Make: BENZ, Model: E300, Year: 2020, Price: $58000\n\
Make: AUDI, Model: A3, Year: 2021, Price: $25000\n"

    assert d3.removeCar("BENZ", "E300", 2022, 60000) == True
    assert d3.removeCar("Audi", "A3", 2021, 25000) == True

    assert d3.getSuccessor("Mazda", "CX-5") == None

    assert d3.inOrder() == "Make: BENZ, Model: E300, Year: 2020, Price: $58000\n"

```

```
116     assert d3.removeCar("Tesla", "Model3", 2018, 50000) == False
117     assert d3.removeCar("BENZ", "E300", 2020, 58000) == True
118 
119     assert d3.getSuccessor("Tesla", "Model3") == None
120 
121     assert d3.postOrder() == ""
```

pytest result:

```
PS D:\VSCode\My Code> python -m pytest "d:/VSCode/My Code/CS 9/lab09/testFile.py"
========================================= test session starts =========================================
platform win32 -- Python 3.10.8, pytest-7.3.1, pluggy-1.0.0
rootdir: D:\VSCode\My Code
collected 3 items

CS 9\lab09\testFile.py ...                                                                    [100%]

========================================== 3 passed in 0.01s ==========================================
```