Organized Truck

In this homework, you will implement an iterable class called `OrganizedTruck` that stores a truck of rotated sorted boxes and provides $O(\log n)$ time search operations. Here, we assume that the boxes are sorted by their volume and the boxes with the same volume are equal. Also, an `OrganizedTruck` may contain repeating elements.

Rotated sorted truck

We call a `Truck` rotated sorted if the rotation is applied on a sorted sequence of boxes. For example, the following code shows a rotated sorted truck.

```
L = [Box(5,1,1), Box(3,2,1), Box(1,1,1), Box(2,1,1), Box(1,3,1), Box(2,1,2)]
t = Truck(L)
```

The volume of boxes in `L` are 5, 6, 1, 2, 3, 4 and it is a sorted sequence rotated for 2 steps to the right.

A truck can store boxes with the same volume but different dimensions. Here, we consider the boxes with the same volume to be equal. A truck may contain repeating elements. However, all the repeating elements in a truck should be adjacent. For example, the following truck is a valid rotated sorted truck with repetition.

```
L = [Box(4,1,1), Box(1,2,2), Box(1,1,1), Box(2,1,1), Box(1,3,1), Box(3,1,1)]
t = Truck(L)
```

The following code shows an **invalid** rotated sorted truck with repetition, since `Box(4,1,1)` and `Box(1,2,2)` are the same but not adjacent.

```
L = [Box(1,2,2), Box(1,1,1), Box(2,1,1), Box(1,3,1), Box(3,1,1), Box(4,1,1)]
t = Truck(L)
```

Organized truck

An `OrganizedTruck` has the following ADT.

- `__init__(self, truck)` - receive a rotated sorted truck and store it internally.
- `__iter__(self)` - return an iterator that iterates over the boxes in sorted order.

- `__getitem__(self, index)` - return the box at position `index`.
- `__len__(self)` - return the number of boxes in the truck.
- `min(self)` - return a box with the minimum volume.
- `max(self)` - return a box with the maximum volume.
- `search(self, box)` - return all boxes in the truck that have the same volume as `box`.

Start by writing a class called `OrganizedTruck` and store it in a file called `organizedtruck.py`. Then, implement `__init__` using the above ADT. Note that `OrganizedTruck` **does not extend** `Truck` and it only stores a given `Truck` internally.

`__getitem__` and `__len__`

These methods enables us to change the look of `OrganizedTruck` to other iterable objects such as `list`. In a `list`, we use squared brackets to retrieve an element at a certain position and `len` function to access the number of items in that list.

```
L = ['a', 'b', 'c', 'd', 'e']
print(L[2])
print(len(L))
```

```
c
5
```

Implement `__getitem__` and `__len__` method of `OrganizedTruck` to have similar functionalities.

Making `OrganizedTruck` iterable

Any object with a method called `__iter__` that returns an **iterator** is called **iterable**. The responsibility of an **iterator** is to provide access to the items in a collection one at a time. For example, a list is an iterable object. When you use a list in a `for` loop, it creates a **new iterator** to traverse the list and each iterator has its own state. The iterator is a distinct object from the list itself. That's why we can have multiple iterators for the same collection as in the following.

```
L = ['a', 'b']
for ch1 in L:
  for ch2 in L:
    print(ch1 + ch2)
```

```
aa
ab
ba
bb
```

To make an object iterable, we need to implement the `__iter__` method. An iterator can be implemented in different ways. In this homework, we use **generators** to implement `__iter__`. Generators allow us to implement iterators succinctly. A generator is very similar to a function, but uses the `yield` keyword instead of `return`. When Python reaches a `yield` statement, it is as if it saves the current states of all variables and suspends the execution of the generator and returns to the caller. When the next item is requested (such as by a `for` statement), the execution will resume after the `yield` statement with the previously stored states. Here is a simple example.

```
def squares(n):
  for i in range(n):
    yield i ** 2

for i in squares(4):
  print(i)
```

```
0
1
4
9
```

Implement `__iter__` for `OrganizedTruck` using generators. In this method, you should create an iterator providing access to `boxes` in sorted order from the smallest box to the largest one. For example, for the following rotated sorted truck, it should start from the left most `Box(1,1,1)` and finish at `Box(1,1,4)`

```
L = [Box(1,1,3), Box(2,2,1), Box(1,1,4), Box(1,1,1), Box(1,1,1), Box(1,2,1), Box(1,1,2)]
t = Truck(L)
```

Your `__iter__` method should run in $O(n)$ time.

Search operations

In this section, you write `min` and `max` methods to report the smallest and the largest boxes in a rotated sorted truck. If there are multiple minimum or maximum, report one. These methods should run in $O(\log n)$ time and their implementation is similar to the binary search algorithm.

Then, write a method called `search` that takes a `Box` as an argument and returns all the boxes in the rotated sorted truck with the same volume. This method should be executed in $O(\log n)$ time.

Summary

For this homework you should implement `OrganizedTruck` class with the described ADT and store it in `organizedtruck.py`. Then you need to submit `organizedtruck.py`, `boxarrangement.py`, and `intfunction.py` files to Mimir. Do not modify `intfunction.py` file.