from entry import Entry

class OpenAddrHashDict(object):
    EMPTY = None  # class variables shared by all objects of the class
    DELETED = True

    def __init__(self, capacity = 8, hashFunction = hash,
                 linear = True):
        self._table = [OpenAddrHashDict.EMPTY] * capacity
        self._size = 0
        self._hash = hashFunction
        self._homeIndex = -1
        self._actualIndex = -1
        self._linear = linear
        self._probeCount = 0

    def __getitem__(self, key):
        """ """ Returns the value associated with key or returns None if key does not exist. """ """
        if key in self:
            return self._table[self._actualIndex].getValue()
        else:
            return None

    def __delitem__(self, key):
        """ """ Removes the entry associated with key. """ """
        if key in self:
            self._table[self._actualIndex] = OpenAddrHashDict.DELETED
            self._size -= 1

    def __setitem__(self, key, value):
        """ """ Inserts an entry with key/value if key does not exist or replaces the existing value with value if key exists. """ """
        entry = Entry(key, value)
        if key in self:
            self._table[self._actualIndex] = entry
        else:
            self._table[self._actualIndex] = entry
            self._size += 1

    def __contains__(self, key):
        """ """ Return True if key is in the dictionary; return False otherwise """ """
        entry = Entry(key, None)
        self._probeCount = 0
        # Get the home index
        self._homeIndex = abs(self._hash(key)) % len(self._table)
        rehashAttempt = 0
        index = self._homeIndex
        # Stop searching when an empty cell is encountered
        while rehashAttempt < len(self._table):
            if self._table[index] == OpenAddrHashDict.EMPTY:
                self._actualIndex = index
                return False  # An empty cell is found, so key not found
            elif self._table[index] == entry:
                self._actualIndex = index
                return True
            # Calculate the index and wrap around to first position if necessary
            rehashAttempt += 1
            if self._linear:
                index = (self._homeIndex + rehashAttempt) % len(self._table)
            else:  # Quadratic probing
                index = (self._homeIndex + (rehashAttempt ** 2 + rehashAttempt) // 2) % len(self._table)
        return False  # tried all the slots in the hash table and did not find key

    def __len__(self):
        return self._size

    def __str__(self):
        resultStr = "{""""for item in self._table:
            if not item in (OpenAddrHashDict.EMPTY, OpenAddrHashDict.DELETED):
                resultStr += """" + str(item)
        return resultStr + """"}

    def __iter__(self):
        """ """ Itterates over the keys of the dictionary """ """

a) Complete the __iter__ method.
Data Structures

Lecture 16

Name:_____________________

2. All simple sorts consist of two nested loops where:
   - The **outer loop** keeps track of the dividing line between the sorted and unsorted part with the sorted part growing by one in size each iteration of the outer loop.
   - The **inner loop**'s job is to do the work to extend the sorted part's size by one.

Initially, the sorted part is typically empty. The simple sorts differ in how their inner loops perform their job.

*Selection sort* is an example of a simple sort. Selection sort’s inner loop scans the unsorted part of the list to find the maximum item. The maximum item in the unsorted part is then exchanged with the last unsorted item to extend the sorted part by one item.

At the **start of the first iteration** of the outer loop, initial list is completely unsorted:

```
Unsorted Part | Empty Sorted Part
-------------|------------------
| 0 1 2 3 4 5 6 7 8 |
```

```
myList: [25 35 20 40 90 60 10 50 45]
```

The inner loop scans the unsorted part and determines that the index of the maximum item, `maxIndex = 4`.

```
Unsorted Part | Sorted Part
--------------|-------------
| 0 1 2 3 4 5 6 7 8 |
```

```
myList: [25 35 20 40 90 60 10 50 45]
```

```
maxIndex = 4 lastUnsortedIndex = 8
```

After the inner loop (but still inside the outer loop), the item at `maxIndex` is exchanged with the item at `lastUnsortedIndex`. Thus, extending the Sorted Part of the list by one item.

```
Unsorted Part | Sorted Part
--------------|-------------
| 0 1 2 3 4 5 6 7 8 |
```

```
myList: [25 35 20 40 45 60 10 50 90]
```

```
maxIndex = 4 lastUnsortedIndex = 8
```

a) Write the code for the outer loop

```python
for lastUnsortedIndex in range(len(myList)-1, 0, -1):
```

b) Write the code for the inner loop to scan the unsorted part of the list to determine the index of the maximum item

```python
maxIndex = 0
for testIndex in range(1, lastUnsortedIndex, 1):
    if myList[testIndex] > myList[maxIndex]:
        maxIndex = testIndex
```

c) Write the code to exchange the list items at positions `maxIndex` and `lastUnsortedIndex`.

```python
temp = myList[maxIndex]
myList[maxIndex] = myList[lastUnsortedIndex]
myList[lastUnsortedIndex] = temp
```

d) What is the big-oh notation for selection sort?

**Basic Ops:** compare items \( (O(n^2)) \) compare

**Moves:** \( (n-1) \times 3 \)
3. **Bubble sort** is another example of a simple sort. Bubble sort's inner loop scans the unsorted part of the list comparing adjacent items. If it finds adjacent items out of order, then it exchanges them. This causes the largest item to "bubble" up to the "top" of the unsorted part of the list.

At the start of the first iteration of the outer loop, initial list is completely unsorted:

```
+-----------------+---------------+
| Unsorted Part   | Empty Sorted Part |
+-----------------+---------------+
myList: 25 35 20 40 90 60 10 50 45
```

The inner loop scans the unsorted part by comparing adjacent items and exchanging them if out of order.

```
+-----------------+----------------+------------------+
| Unsorted Part   | Sorted Part     | lastUnsortedIndex = 8 |
+-----------------+----------------+------------------+
myList: 25 35 20 40 90 60 10 50 45
```

```
+-----------------+----------------+------------------+
| Unsorted Part   | Sorted Part     | lastUnsortedIndex = 8 |
+-----------------+----------------+------------------+
myList: 25 35 20 40 90 60 10 50 45
```

```
+-----------------+----------------+------------------+
| Unsorted Part   | Sorted Part     | lastUnsortedIndex = 8 |
+-----------------+----------------+------------------+
myList: 25 35 20 40 90 60 10 50 45
```

```
+-----------------+----------------+------------------+
| Unsorted Part   | Sorted Part     | lastUnsortedIndex = 8 |
+-----------------+----------------+------------------+
myList: 25 35 20 40 90 60 10 50 45
```

```
+-----------------+----------------+------------------+
| Unsorted Part   | Sorted Part     | lastUnsortedIndex = 8 |
+-----------------+----------------+------------------+
myList: 25 35 20 40 90 60 10 50 45
```

After the inner loop (but still inside the outer loop), there is nothing to do since the exchanges occurred inside the inner loop.

a) What would be the worst-case big-oh of bubble sort? \( O(n^2) \) comparisons, \( O(n^2) \) moves.

b) What would be true if we scanned the unsorted part and didn't need to do any exchanges? **Unsorted part (and whole list) is sorted.**
Selection Sort

N items in list

(n-1) compares
(n-2) compares
(n-3) compares
\vdots
N

Last unsorted index

0
1
2
\vdots
N

N+N+\cdots+N = N \times \frac{n(n-1)}{2}

\frac{(n-1)}{2} \text{ pairs} = \frac{n^2}{2} = \frac{n}{2}

O(n^2) compares
Bubble sort - worst case descending order initial
best case - ascending order

# compares # moves
(n-1) 3(n-1)
(n-2) 3(n-2)
(n-3) 3(n-3)

2 3(2)
+ 1 3(1)

\[ \frac{n(n-1)}{2} \quad \frac{3n(n-1)}{2} \]

\[ O(n^2) \quad O(n^2) \]

best case \[ O(n^2) \quad O(1) \]
Bubble sort code

```python
for lastUnsortedIndex in range(len(myList)-1, 0, -1):
    didSwap = False
    for testIndex in range(0, lastUnsortedIndex, 1):
        if myList[testIndex] > myList[testIndex+1]:
            temp = myList[testIndex]
            myList[testIndex] = myList[testIndex+1]
            myList[testIndex+1] = temp
            didSwap = True
        if not didSwap:
            return
```