Maps and Python

When we store information in a Map, our data consists of two things, a key and a value. Suppose we're wanting to use our BST from last week. We have two choices, we could either modify our BST Nodes to store both a key and a value like this:

```
class Node:
  def __init__(self, key, value):
    self.key=key
    self.value=value
    self.left=None
    self.right=None
```

or, we could keep our Nodes as they were, and make a single class called KVPair which can be stored as the data field:

```
class KVPair:
  def __init__(self, key, value):
    self.key = key
    self.value = value
```

Now, we can keep our BST Nodes as they were, and just add KVPairs as data. Which is better? Keeping your BST general and applicable for lots of purposes, without extra fields hard-coded in, is a nice thing, especially for if you're passing your code off to someone else or coming back to it later after a few weeks away. For this lab, we're going to make this KVPair object, and build a map that uses it to store key-value pairs.

Operator Overloading

Some of you may not be happy with the above paragraphs, and might be saying, "but my BST (and other data structures we have talked about) depends upon using the < and > signs on data, and those won't work with my new, invented class!" And you'd be right. But, it turns out that if you have your own class, you can define those operators to do what you want. For example:

```
class KVPair:
  def __init__(self, key, value):
    self.key = key
    self.value = value
  def __lt__(self, other):
    if self.key < other.key:
      return True
    return False
```

Now, given two KVPairs `pair1` and `pair2`, you can run `pair1<pair2`, and the right thing will happen! Cool! So, now you can use your BST as written, without changing a thing. Overloading the less-than operator even lets you sort a list full of your objects using the `.sort()` command.

There are a lot of operators you can overload! (https://docs.python.org/3.4/library/operator.html#mapping-operators-to-functions)

The Assignment

You're going to create two classes, `KVPair` and `SortedArrayMap`. `SortedArrayMap` should keep its KVPairs in a list, sorted by key. Your KVPairs **(10%)**have to:

- Compare by keys, meaning `pair1<pair2` should work, as should >, <=, and >=, based on the keys,
- Compare for equality, meaning `pair1==pair2` should work, based on the equality of the keys.
- Compare for inequality, meaning `pair1!=pair2` should work. (NB: If you overload ==, you should always also overload !=, or else you get weird behavior.)

Your map has to:

- **(5%)**Have an `__init__` function that initialized the map with an empty array.
- **(15%)** Insert key-value pairs. The way this is normally done in Python is by using the [] operator, meaning for some key `k` and some value `v`, `aMap[k]=v` adds that pair to the map. This involves overloading the `__setitem__(self, k, v)` method.

  This method should start by turning k and v into a single KVPair before inserting that single object into the Map.

  You may assume each insert has a new, unique key.
- **(15%)** Given a key, get the value. This is again done using the [] operator: `v=aMap[k]`. When retrieving an item from a map, Python calls the `__getitem__(self,k)` method on your class, which you should overload.

  You may assume a competant user, who will only insert valid keys. If this makes you uncomfortable, as it probably should, the correct behavior is to raise an Exception (a KeyError, to be precise) to notify the user.
- **(15%)** Tell you if a key appears in the Map using `someKey in aMap`. The "in" operator is overloaded by writing a `__contains__(self, k)` function in your class.

Sorting your list: the easiest way to do this is to just keep your list sorted from the point that you start building it. Each time you want to insert a new KVPair, put it in the right location in the list so that the list remains sorted (think AddInOrder from your recursion lab). . Once you find the right location, you can use the insert (https://docs.python.org/3/tutorial/datastructures.html) function that Python provides for lists, no need to do it manually at this point.

All methods should run the fastest we can make them run. That means that `__getitem__` and `__contains__` should make use of binary search to find the right location in the list! Hint: look back at this homework (class.html?hw=bigohw) for an example of binary search.

You DO NOT need to use binary search for `__setitem__`. This is because inserting a new KVPair into the array is already $O(n)$ so binary search would not save you any time. However, you should not use `sorted()` or `list.sort()` to do `__setitem__` because those functions run in $O(n \log n)$ time (we will get to that later in the class).

Particulars

Put your classes into a file called `maps.py`.

To be explicit, code like the following (which is in no way an exhaustive testing suite) should work:

```
from maps import *

pair1=KVPair(160000,'Aardvark, Aaron')
pair2=KVPair(169998,'Zebra, Zeke')
print(pair1<pair2) #prints True
print(pair2<pair1) #prints False
print(pair2==pair1) #prints False

arrMap=SortedArrayMap()
arrMap[160000]='Aardvark, Aaron'
print(arrMap[160000]) #prints 'Aardvark, Aaron'
print(160000 in arrMap) #prints True
print(169998 in arrMap) #prints False
```

Part Two **(40%)**

As we know from SY110, every network device has a 6-byte address that it uses for link layer communication called a Media Access Control (MAC) address. This address is unique to each device (laptop, cell phone, smart watch, etc.) and is assigned by the manufacturer when it is created. To make sure that two manufacturers don't give the same address to different devices, the set of all MAC addresses is divided up and assigned to manufacturers. The first 3 bytes of the MAC address, called an (https://en.wikipedia.org/wiki/Organizationally_unique_identifier)Organizationally Unique Identifier (OUI) identify which manufacturer created that device.

For instance, the MAC address of my computer is `24:f0:94:ea:61:6e`. The OUI is the first 3 bytes, `24:f0:94`. That OUI belongs to Apple, indicating that my computer is manufactured by Apple.

The OUI system was created to prevent accidental address collisions between manufacturers, but it has important implications for cyber security and privacy. Every time a device communicates on a network it uses its MAC address to identify itself. Since every MAC address contains an OUI, that means that it is easy to determine the manufacturer of every device on a network. This is great if you are a hacker because you can quickly determine which devices to target based on known exploits for different operating systems, manufacturers, models, etc. In fact, we have an active research group (https://www.cmand.org/furiousmac/) at USNA working on automatically determining information from devices based on their MAC addresses and other passive wireless signals.

Your task for this lab is to write a program `ouilookup.py` that reads in this file (labs/ouis.txt) which contains a list of all OUIs and the manufacturers assigned to them and creates a SortedArrayMap where the keys are OUIs and the values are the manufacturer assigned to that OUI. The OUI file contains two strings for each manufacturer, a short and long description. You should use the first (short) one.

Your program should take a single command line argument specifying a network capture file (PCAP) like this one (labs/pcap.txt). This is WiFi network data that was captured by Professor Mayberry's research group at Reagan National Airport that contains many devices. Each line represents a single packet with a sequence number, length, source address, and destination address separated by commas (this type of file is usually called a CSV). You should use your OUI map to lookup the manufacturer of the source and destination and print them to the screen. If the OUI doesn't exist in the list, you should print "Unknown". A MAC address of `ff:ff:ff:ff:ff:ff` is a special address that means the frame is broadcasting to any recipient that is listening, so for these you should indicate "Broadcast". For instance, the top of that file looks like this:

```
1,389,28:80:a2:0e:bb:f5,ba:bd:3a:54:0f:d9
2,232,2c:33:11:cf:bd:40,ff:ff:ff:ff:ff:ff
6,225,2c:33:11:cf:bd:40,ba:bd:3a:54:0f:d9
```

The OUI `28:80:a2` corresponds to the manufacturer `NovatelW`, the OUI `ba:bd:3a` is unknown, the OUI `2c:33:11` corresponds to `Cisco`, etc. Therefore the output of your program should be:

```
NovatelW Unknown
Cisco Broadcast
Cisco Unknown
```

Here is a (labs/pcapShort.txt) shorter version of the PCAP file you can use for initial testing.

Note: The hex digits in the OUI file are in uppercase while the ones in the PCAP are in lowercase. Make sure to either convert the OUIs to lowercase before inserting them into your map or convert to uppsercase when you check addresses from the PCAP.

Unknown manufacturers

You might be wondering why there are so many Unknown MAC addresses in this network capture. Recently, manufacturers have realized that broadcasting MAC addresses is a privacy vulnerability and some have implemented features where they rotate a device's MAC address randomly to try to hide from observers. These random addresses fall outside of the range of allocated OUIs and so will show up as Unknown. The network traffic captured above was used in a research project where we assessed (https://petsymposium.org/2017/papers/issue4/paper82-2017-4-source.pdf) the effectiveness of these randomization methods.

Grade Breakdown

- 10%: KVPair
- 50%: SortedArrayMap
- 40%: OUI Lookup

Submit as Lab7.