Semi-Optional Project 2: Building Your Own Hash Table

Due Whenever

This optional assignment will have you implementing in python a dictionary which works identically to the dictionary in python, right down to the use of brackets and notation. For this assignment, you have a choice: you can either use chaining or quadratic probing. My guidance here may be biased in favor of chaining, as that is what I chose myself. This one is highly recommended - it is a great exercise in object oriented programming using python, and will acquaint you with a lot of python's cooler features.

(1) (Optional: Only needed if using chaining) Make a Node class with no methods, and just three attributes:

- self.item: This will be the key, value pair, i.e. the data encapsulated by the node.
- self.next, and self.prev. These last two will be pointers that are meant to be assigned to more nodes.

Your constructor should initialize all of these attributes to None. The actual assignment will be handled by your LinkedList class! This class should have three attributes:

- self.first: The first node in the list. Initialize to None in the constructor.
- self.last: Self explanatory.
- self.list_size: Also self explanatory. Initialize to 0.

Your linked list class should also have three methods:

- add_node(self, node, side = 'front'): Should do what you think it should. A call to this function should also be able to input side = 'back' as a keyword argument, in which case the added node is attached to the back of the list instead of the front.
- delete_node(self, index): Deletes the indexed node. So if a user calls delete_node(5), it should work it's way to node 6 and rewire surrounding nodes so as to effectively remove this node from the program, so that garbage collection removes it later.
- __str__(self): This is a special dunder method which is called whenever the user types print(object). Your linked list should return the item stored in the node,

printed in the way that python would print them if you were to print a dictionary, i.e. key: value, separated by commas. You might consider adding an __str__ method to the Node class as well, in order to help with this.

Now for the hash table.

(2) You are going to first make a hash table with no dynamic scaling, and then implement the scaling as a separate class in order to compare the runtime difference. For the no-scaling version, here is the laundry list of attributes and methods:

Attributes

- self.table: The list! If chaining, then your constructor should initialize it by filling it with some number of empty linked lists. If you're using probing, you will probably want to start this at a power of two (see lecture notes) and fill it with the entry None rather than linked lists. For chaining, I started table size at 10.
- self.m: A counter to keep track of the table size. Not strictly necessary, but ends up helpful. It should be the same size as the table, always (except possible durring the table scaling process).
- self.n: A counter to keep track of how many items are currently in the table. This one is *very* necessary, especially if you are chaining.
- self.load_factor: This should always just be self.n/self.m. It is what you will monitor in order to know when to scale the table and preserve performance.

Methods

- self.__hash(self, key): Your hash function. You are welcome to use any hash function you want or even a universal set of hash functions chosen from randomly. I just used the division method and that is fine too. Don't forget to do the prehashing first: Use python's built in hash() function for this.
- __getitem__(self, key) and __setitem__(self, key, value): *These* are where the action will be. These dunder methods allow you to treat your object in code as you would any built-in indexable item in python. For example, if I defined

```
def __setitem__(self, key, value):
    print(f'Depositing {value} into the void')
```

And then created an instance of my object and wrote this:

```
object = Object()
object[-1] = 'hello'
```

Then when I ran the program it would print 'Depositing hello into the void.' This is how numpy arrays can work the way they do.

Your __setitem__ and __getitem__ methods should do all of the necessary work in storing items in their appropriate places in the hash table and retrieving them, keeping in mind that theoretically speaking, __setitem__ is the *insert* dictionary

operation and __getitem__ is the *search* operation. My guidance here will be minimal. **Your hash table needs to behave identically to python's own built-in dictionary object from a front-end standpoint!** Theoretically, you will need to make sure that the insert operation properly overrides existing keys, and that the search operation returns the proper KeyError when a search fails. This is also where you will implement your probing if you are going with an open address hash table. Don't forget to update m, n and the load factors!

- __delitem__(self, key): Again, this is what defines when you can say del object[key] and have the right thing happen. With python dictionaries, I can type del D[key] and have the item be removed from the dictionary. This is what I expect from your hash table as well.
- __str__(self, key): Should print a string which looks identical to what would happen if you called print on a python dictionary. Your str method for linked lists should be helpful here.

After you have your table working properly, you're almost done.

(3) Copy paste your hash table class and rename the old one HashTable_NoScaling. Then modify the regular hash table to implement dynamic table scaling. This will require one final private method, called __table_double(self) (it doesn't have to be double, but that's what I went with. Probers should be careful here!)

This function will be pretty involved. You will need to save the old table as a local scope temporary variable, then clear the entire existing table, initialize a new table which is larger, and then copy everything from the old table over, rehashing everything one at a time.

(4) Make a generator function called make_junk_items which creates a bunch of random key-value pairs, in which the keys are strings of random lengths between 1 and 20, and the values are whatever you want them to be. Then use this in order to test your hash tables: Create a list of 5000 junk items. Then start choosing random items from that list and inserting them into both kinds of dictionary, one at a time. As you do this, whenever n is a multiple of 10, stop and perform 100 random searches for random keys (you can go with keys in the list, i.e. successful searches, to avoid dealing with exceptions) in the two dictionaries, timing how long each search takes. Then store the average of those 100 searches in arrays, and plot the two performances. You should see the scaling hash table performing in constant time, with the non-scaling version performing in a high variance linear time. Attach a plot to your write-up.