Hellacious Homebrewed Hashing

Jayden Patel

Table of Contents

01

The Assignment

What are we doing?

02

The Problem

What are we solving?

03

My Solution

What did I come up with?

04

The Results

How does it fare?

01

The Assignment

Hash Tables

- The assignment is to create a custom implementation of Python’s dictionary.
- A dictionary stores data in “key, value” pairs, where a given key is used to retrieve a value.
- In the background, the hash of the given key decides in what position the key, value pair is placed.
- Because the hash table knows that the exact hash will always be in that position, it has a retrieval time of O(1).

02

The Problem

Resizing

- One problem with hash tables is what to do when they get full.
- The simple method is to make another, bigger hash table and simply rehash all the items in the bigger table.
- However, this leads to performance concerns, as every item must be rehashed individually and added to the new table the moment the old one gets full.

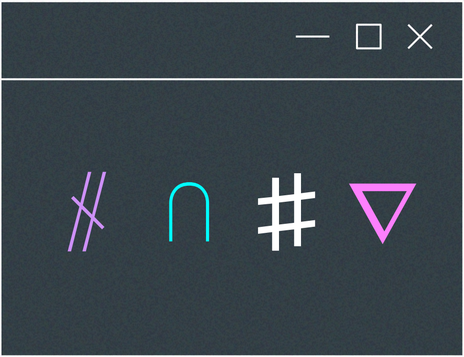
03

The Solution

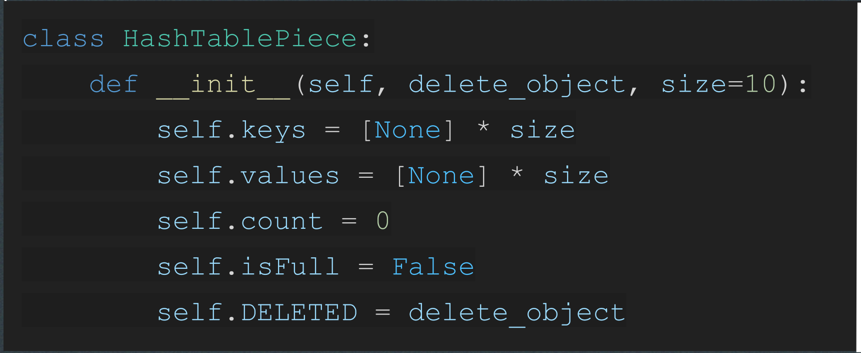
Incremental Resizing

- The other solution is to spread out the item transfer across multiple operations.
- Instead of transferring all at once, the table transfers an item from the smaller table to the bigger one every time a new item is added to the table.
- Once the smaller table is empty, it is replaced with the bigger table, and the cycle continues.
- Once the new table is full, a newer, bigger, better one is created, and the same per operation transfer is used to move everything across.

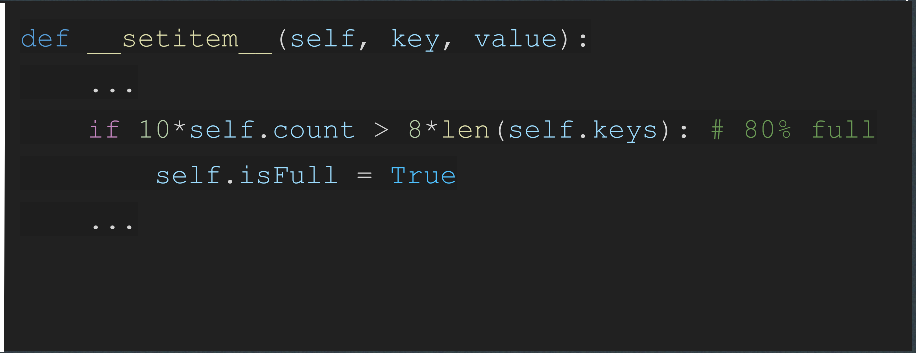
My Implementation



- My implementation uses reduced versions of hash tables.
- Each one is exactly the same as a regular hash table, except that:
  - it does not resize itself
  - it stores a whether or not it has reached capacity (defined as 80% full)
  - it takes in an object to use as a placeholder when an item is deleted instead of defining its own object.



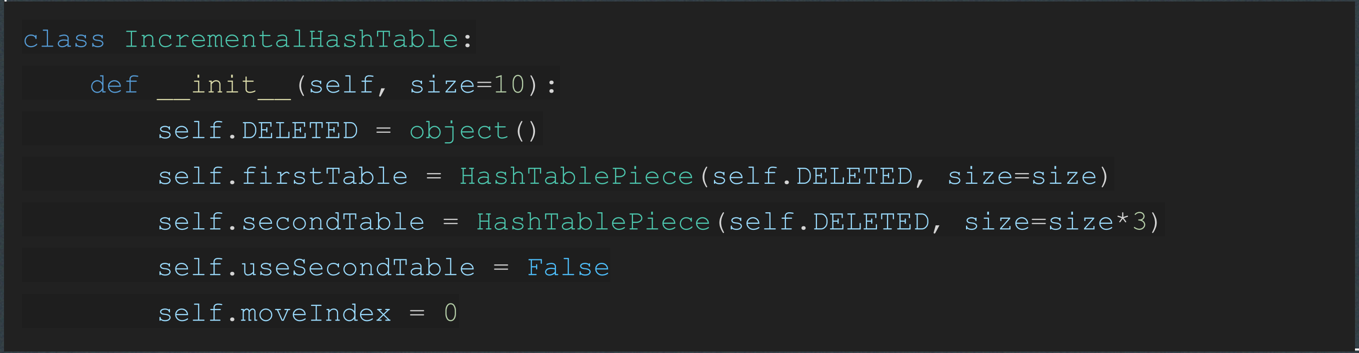
```python
class HashTablePiece:
    def __init__(self, delete_object, size=10):
        self.keys = [None] * size
        self.values = [None] * size
        self.count = 0
        self.isFull = False
        self.DELETED = delete_object
```



```python
def __setitem__(self, key, value):
    ...
    if 10*self.count > 8*len(self.keys): # 80% full
        self.isFull = True
    ...
```

Structure

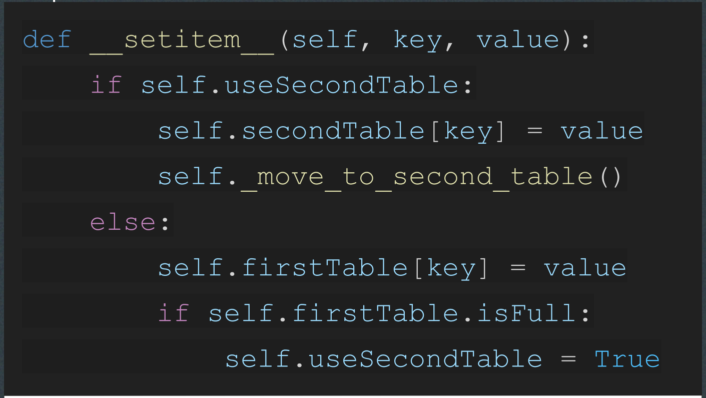
- The master table class contains two of the “reduced” hash tables, as well as a common deleted placeholder object to pass onto the two tables.
    - One of them is three times the size as the other.
- The master table manages the smaller ones and handles switching between the two tables and moving items across them.



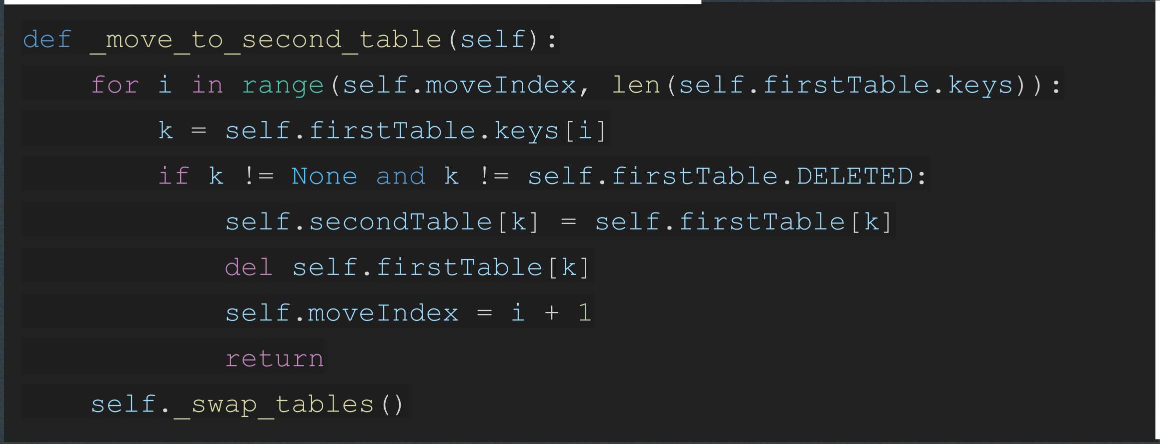
```
class IncrementalHashTable:
    def __init__(self, size=10):
        self.DELETED = object()
        self.firstTable = HashTablePiece(self.DELETED, size=size)
        self.secondTable = HashTablePiece(self.DELETED, size=size*3)
        self.useSecondTable = False
        self.moveIndex = 0
```

Transferring

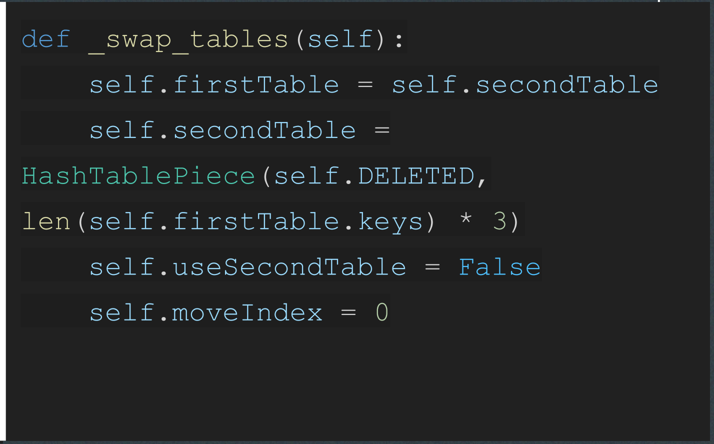
- When items are added to the master table, it stores them in the smaller first table.
- Once it marks itself full, the master table switches to the second table.
  - Then, items are moved over from top to bottom of the first table every time an item is added to the master table (which goes to the second table).



```
def __setitem__(self, key, value):
    if self.useSecondTable:
        self.secondTable[key] = value
        self._move_to_second_table()
    else:
        self.firstTable[key] = value
        if self.firstTable.isFull:
            self.useSecondTable = True
```



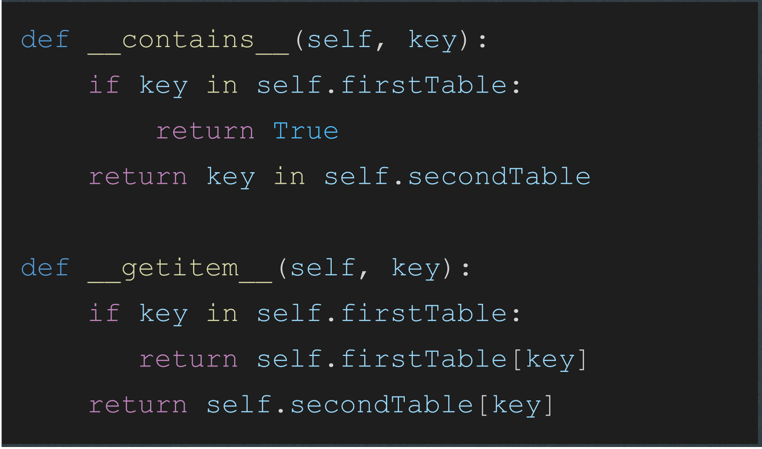
```
def _move_to_second_table(self):
    for i in range(self.moveIndex, len(self.firstTable.keys)):
        k = self.firstTable.keys[i]
        if k != None and k != self.firstTable.DELETED:
            self.secondTable[k] = self.firstTable[k]
            del self.firstTable[k]
            self.moveIndex = i + 1
            return
    self._swap_tables()
```



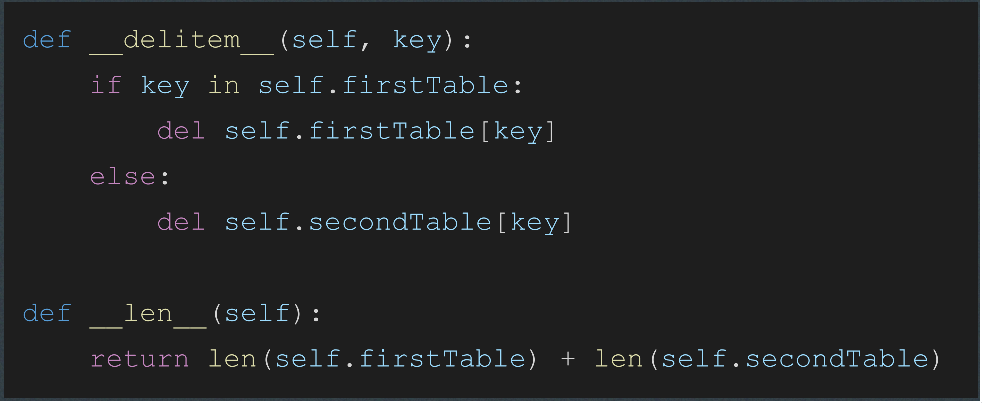
```
def _swap_tables(self):
    self.firstTable = self.secondTable
    self.secondTable =
HashTablePiece(self.DELETED,
len(self.firstTable.keys) * 3)
    self.useSecondTable = False
    self.moveIndex = 0
```

Other Features

- All other features of a hash table were implemented
    - set, contains, get, del, len, str
- The master table handles going between both tables for each function.



```python
def __contains__(self, key):
    if key in self.firstTable:
        return True
    return key in self.secondTable

def __getitem__(self, key):
    if key in self.firstTable:
        return self.firstTable[key]
    return self.secondTable[key]
```

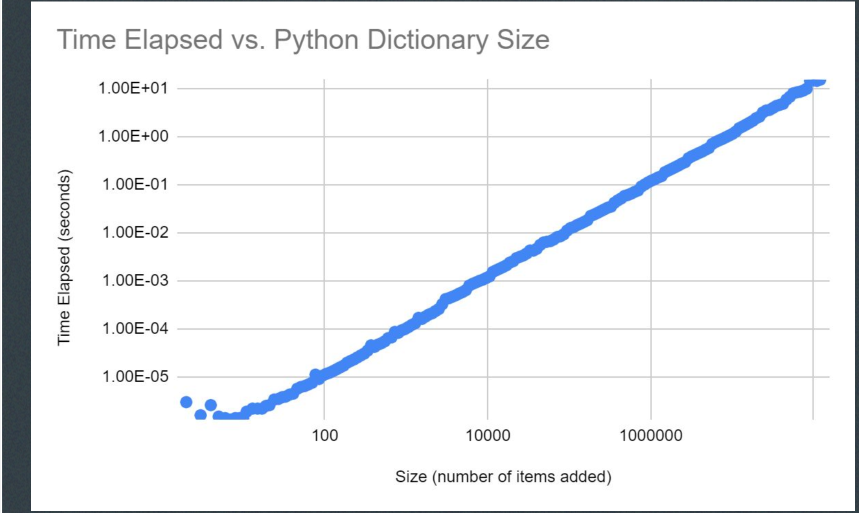


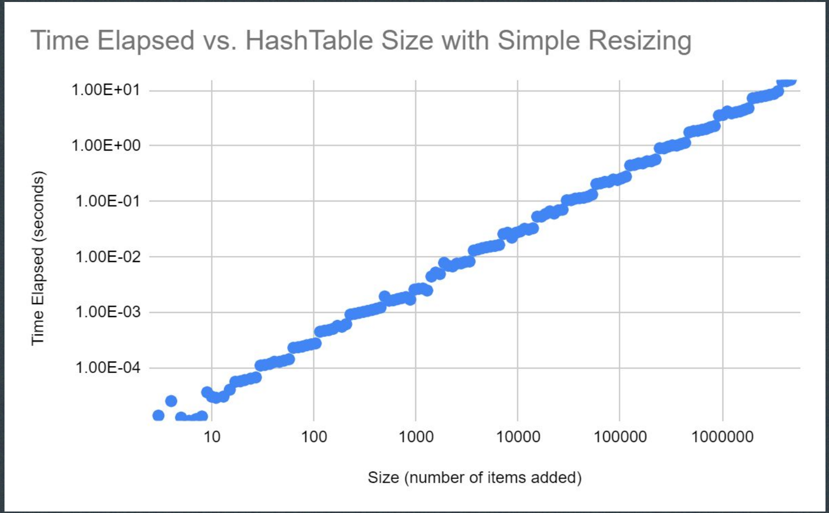
```python
def __delitem__(self, key):
    if key in self.firstTable:
        del self.firstTable[key]
    else:
        del self.secondTable[key]

def __len__(self):
    return len(self.firstTable) + len(self.secondTable)
```

04

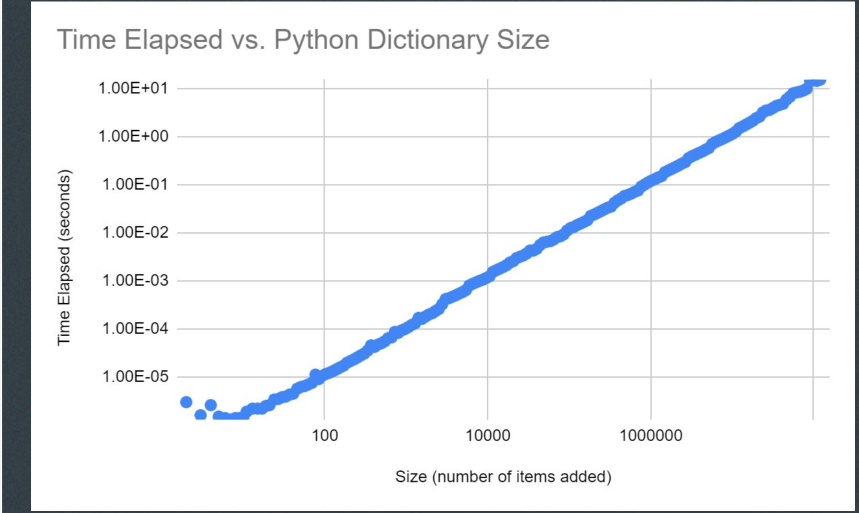
The Results

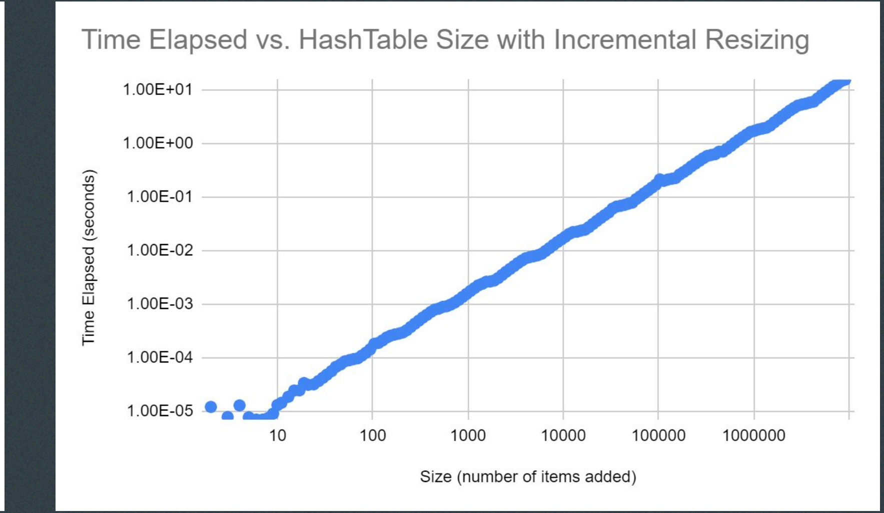
Dictionary vs. Simple Resizing



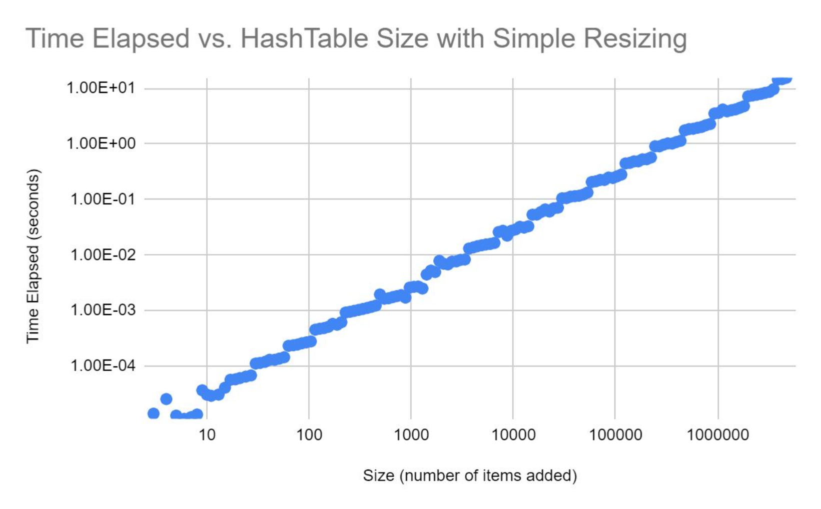


Dictionary vs. Incremental Resizing

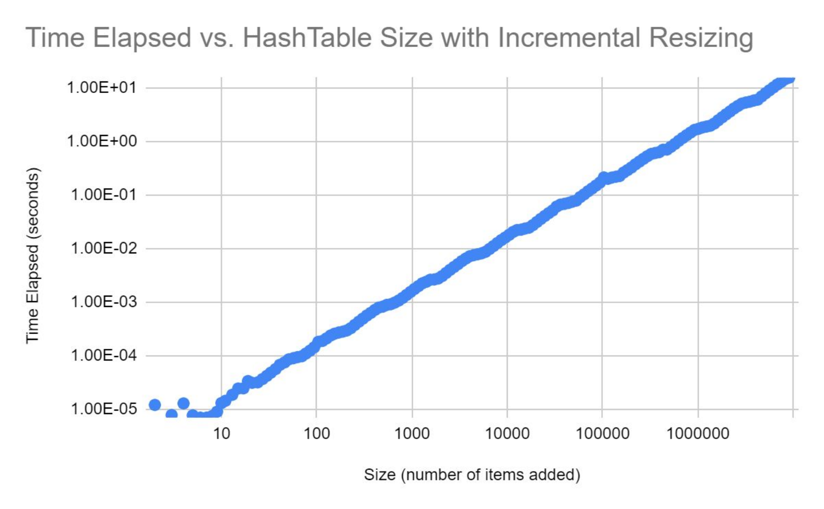




Simple vs. Incremental Resizing



Time Elapsed vs. HashTable Size with Simple Resizing



Time Elapsed vs. HashTable Size with Incremental Resizing

Hellacious Homebrewed Hashing

Jayden Patel