Hellacious Homebrewed Hashing

CP-3 (Data Structures)

Robert Boerwinkle

Spring 2023

Introduction

Hash tables are a data structure which consists of an unordered collection of key-value pairs. Python has a built-in structure called a dictionary (`dict`) which fills this niche. This project implements a custom hash table in Python and compares it to the built-in.

These structues are special as they have assignment and look-up opperations of time complexity O(1). The stucture consists of a partially empty array in which the index of the element is the hash of the key (mod the length of the array). Some problems do arrise when two keys have the same hash, esspecially when the array is small (so the modulus is smaller and there are fewer total options). In this case, the next empty slot is used. The order in which slots are checked for Python's dictionary is convoluted, but the custom implementation simply checks slots in sequential order.

First Implementation

The crucial components of the structure consist of the table where the actual entries are stored and a hash function. The md5 hash was used. `findDeletes` will be explained in a future section ("Deletion of Items").

```
def _hash(self, x):
        return int(md5(str(x).encode()).hexdigest(),16)%len(self.table)


def _findIndex(self, key, findDeletes=False):
        index = self._hash(key)
        while self.table[index]:
                if self.table[index] is self.deleted:
                        if findDeletes:
                                return index,True
                elif self.table[index][0] == key:
                        return index,True
                index += 1
                index %= len(self.table)
        return index,False
```

The `_findIndex` function was used to set values, get values, delete values, etc. This works well, until the array starts to fill up. This method of choosing new slots is called 'linear probing', and starts to drastically affect performance once the array gets above 80% full.

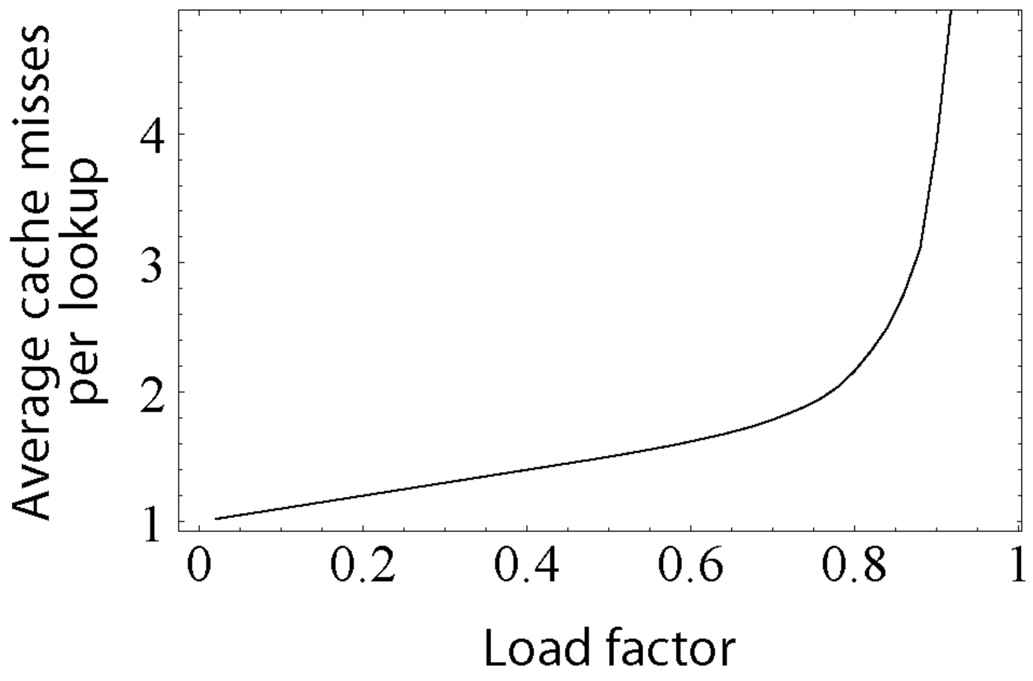

Figure 1: "On modern machines, [cache misses are] a good estimate of actual clock time required", modified from https://commons.wikimedia.org/wiki/File:Hash_table_average_insertion_time.png

The other problem is that there is a hard cap on the number of elements based on the size of the array. The array has to be resized periodically. These resizes were chosen when the array was 70% full. The `ensureSize` function was called whenever an element was added.

```
def ensureSize(self):
        if self.count*10 > len(self.table)*7:
                newTable = HashTable(size=2*len(self.table))
                for item in self.table:
                        if item:
                                newTable[item[0]] = item[1]
                self.table = newTable.table
                self.count = newTable.count
```

Deletion of Items

If deleted items got reverted to `None`, linear probing to find items could conceivably stop short of where the element is supposed to be. A placeholder needs to be used that is looked over when searching for items, but can be re-written and is ignored by any outward-facing interface. A custom object was made to fill this role. The toggle is there to allow for setting items. New items can re-write deleted items, so that opperation would use `findDeletes=True`.

Dynamic Resize

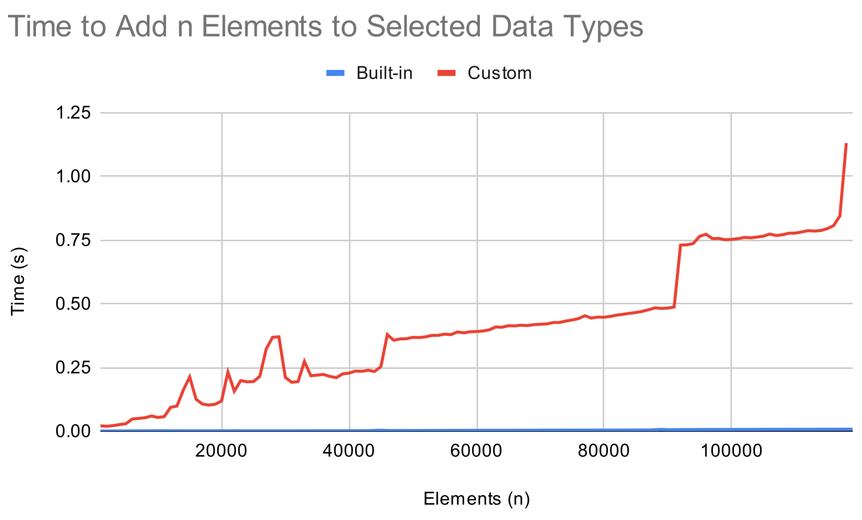

Figure 2: Benchmark of the custom vs. built-in dictionaries

The big jumps occur when the table needs to be resized. This can lead to unpredictable wait times. This cost can be amortized with dynamic resizing. The basic idea is to simply make a new table when the old one fills up. Whenever any value needs to be changed or read, all of the tables are searched.

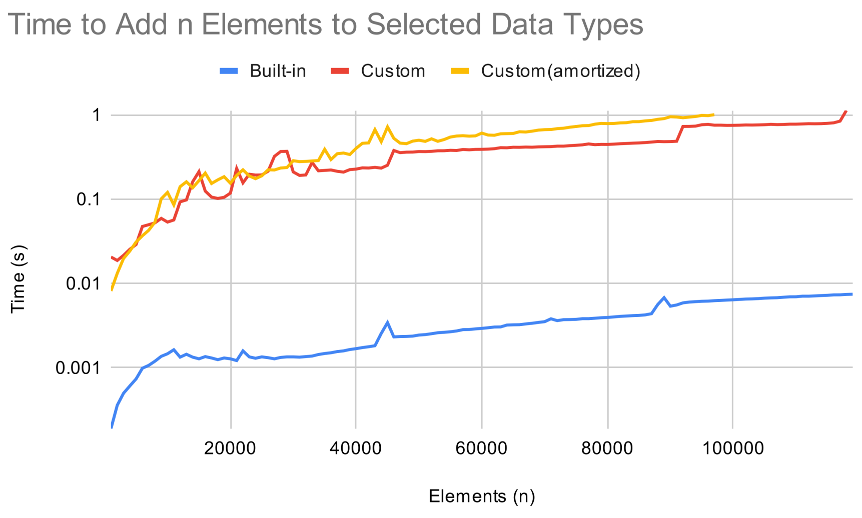

Figure 3: benchmark of the custom vs. custom(amortized) vs. built-in dictionaries (log scale)

Conclusion

Many more optimizations could be made to this implementation. Moving frequently queried entries to the top of the stack of tables is a common improvement. You can also gradually push all old values to the top, removing the empty tables below as you go. None of the implementations could compare to the speed of built-in types, however.