Hellacious Homebrew Hashing

Daniel Nkunga

No cool slides this time because I'm lazy ;-;

What are Hash Tables

Hash tables are data structures that store key-value pairs where each key is mapped to a unique value through a hash function. They are used to efficiently retrieve and store data because they provide constant-time average-case complexity for insertions, deletions, and lookups. Hash tables are commonly employed in computer science and programming to implement associative arrays, database indexing, caches, and sets, offering a fast and effective way to manage large amounts of data by quickly locating values associated with given keys. Their performance advantage comes from the ability to quickly compute the storage location (or bucket) for each key, allowing for rapid access and modification of data.

Why Incremental Resizing

Incremental resizing is crucial for hash tables because it allows the table to dynamically adjust its size as the number of elements (key-value pairs) increases or decreases. When a hash table reaches a certain capacity threshold (typically determined by a load factor), it triggers a resize operation to either increase or decrease its size.

Increasing the size (rehashing) prevents the table from becoming too full, which could lead to performance degradation due to increased collisions. This ensures that the hash table maintains an optimal load factor for efficient operations.

My Implementation

Initialization and Insertion

Initialization - The table is created to have 16 empty elements where every element is a None value. The size can be dynamically set.

Insertion - Insertion takes the extra parameters fo “key” and “value’ ; The “value” is the actual readable information stored. The “key” is a generated location for the information in the hash table set by the __get_hash_index function. It checks that if a location does not contain a value, it will automatically update that value. If there is something at that value (collision) the table will add that value to the end of the table. At the end it also tells the table to resize itself if after an insertion it is more than halfway full.

```
def __init__(self, size=16):
        self.num_elements = 0
        self.data = [None] * size
        self.size = size
        self.resize_count = 0
        self.total_resize_time = 0

def insert(self, key, value):
        hash_data = (key, value)
        hash_index = self.__get_hash_index(key)
        if self.data[hash_index] is None:
            self.data[hash_index] = [hash_data]
        else:
self.data[hash_index].append(hash_data)
        self.num_elements += 1
        if self.num_elements > self.size * 0.5:
            self.__resize()
```

Resize

Resize - The resize function starts by creating a new, empty table double the size of the previous table. It then copies down all the previous information from the past table into the new table.

```
def __resize(self):
        start_time = time.time()
        new_size = self.size * 2
        new_data = [None] * new_size
        for bucket in self.data:
            if bucket is not None:
                for (k, v) in bucket:
                    new_hash_index = self.__get_hash_index(k)
                    if new_data[new_hash_index] is None:
                        new_data[new_hash_index] = [(k, v)]
                    else:
                        new_data[new_hash_index].append((k, v))
        self.data = new_data
        self.size = new_size
        self.resize_count += 1
        elapsed_time = time.time() - start_time
        self.total_resize_time += elapsed_time
        print(f"{self.size}, {elapsed_time:.10f}")
```

Resizing Benchmarking

All resizing was done by doubling the size of the table when it reached 50% of full capacity. Further test should determine the optimal time to resize and how much the table should resize by.