All-at-Once Hash Tables & Incremental Hash Tables

by

Brent Orlina

Computer Programming III

Nicholas Seward

January 30, 2024

All-at-Once Implementation

I implemented the All-at-Once Hash Table dynamic resizing by creating a two new lists with twice the size of the original key and value lists.

Then, it iterates through the original key list. For each key in the key list, it rehashes the key and adds it to the new key list. The corresponding value also transferred to the corresponding spot in the new value list.

Then, it sets the new key and value list as the “original” key and value list.

```
def _upsize(self):
    new_length = len(self.key_table)*2
    nkey_table = [None]*new_length
    nvalue_table = [None]*new_length

    for index in range(len(self.key_table)):
        k = self.key_table[index]
        v = self.value_table[index]
        if k is None or k == self.DELETED:
            continue

        nindex = self._get_index(k, nkey_table)
        nkey_table[nindex] = k
        nvalue_table[nindex] = v

    self.key_table = nkey_table
    self.value_table = nvalue_table
```

Incremental Implementation

I implemented Incremental Hash Table dynamic resizing by creating a “cleaning table.” Once the hash table needs to resize, the original key list becomes the cleaning table and the original key list gets replaced by an empty list with double the size.

It sets the “cleaning index”, self.cindex, to an index with a key in the cleaning table.

Then it calls the `_clean()` function to transfer the first key + value in the cleaning index of the cleaning table to the new key and value list. The `_clean()` function is called every time a new element is added to the table.

```
def _upsize(self):
    new_length = len(self.key_table)*2
    self.key_table, self.ckey_table = [None]*new_length, self.key_table
    self.value_table, self.cvalue_table = [None]*new_length, self.value_table
    self.cindex = 0

    while self.ckey_table[self.cindex] is None or self.ckey_table[self.cindex] == self.DELETED:
        self.cindex += 1

    self._clean()
```

`_clean()` Function

I implemented the `_clean()` function by rehashing the key in the current cleaning index and adds it to the main key list. The corresponding value in the cleaning table is also added into the main value list.

The key in the cleaning table of the current cleaning index is replaced by a `self.DELETED` so that other functions such as lookups still work properly.

Then it advances the cleaning index until it reaches an index with an element that has a key.

```
def _clean(self):
    if self.cindex >= len(self.ckey_table):
            return

    k = self.ckey_table[self.cindex]
    index = self._get_index(k, self.key_table)

    self.key_table[index] = k
    self.value_table[index] = self.cvalue_table[self.cindex]

    self.ckey_table[self.cindex] = self.DELETED
    self.cvalue_table[self.cindex] = None

    while (self.cindex < len(self.ckey_table) and
          (self.ckey_table[self.cindex] is None or
           self.ckey_table[self.cindex] == self.DELETED)):
        self.cindex += 1
```

Making Other Functions Work

Making other functions work mostly consisted of making sure to check the cleaning table since it won't be empty most of the time.

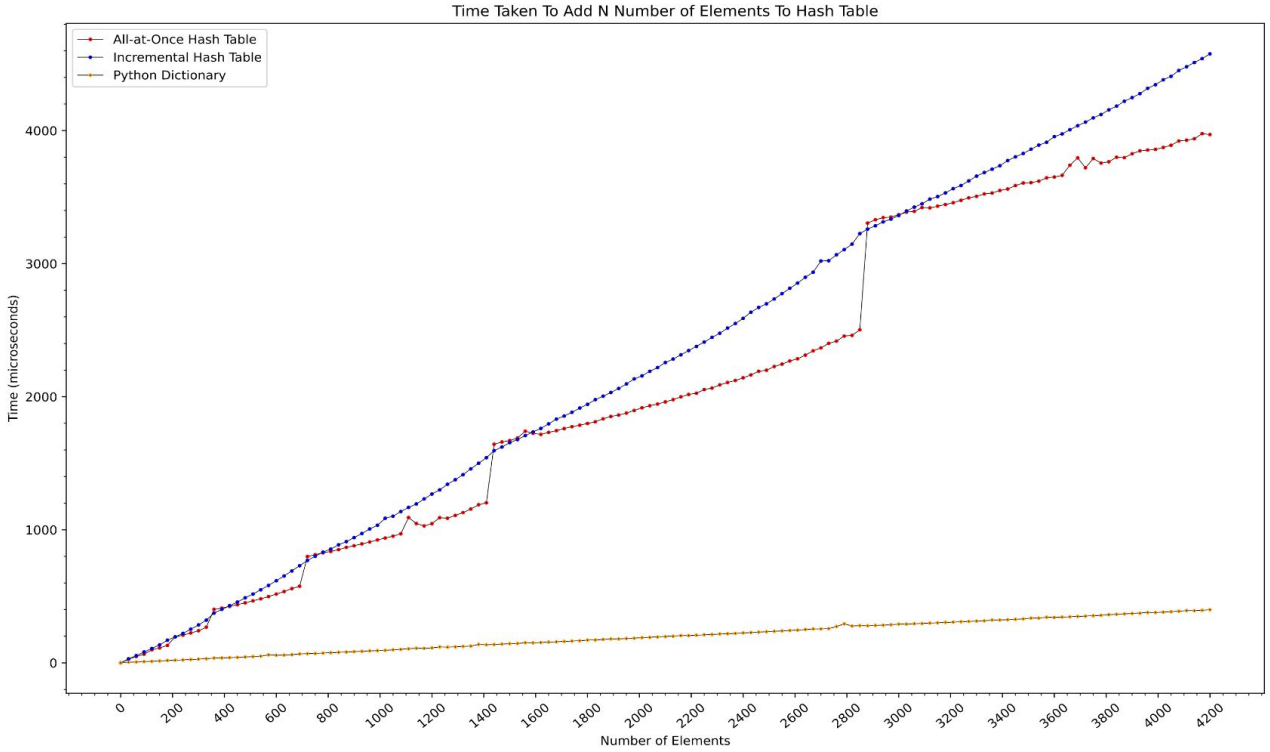
For `__contains__()`, we just had to check if either the main key list or the cleaning table contained the requested key.

For `__getitem__()`, we just had to check if the given key matches the key in either the main key list or the cleaning table. Then we return the value or raise an error if the key doesn't match.

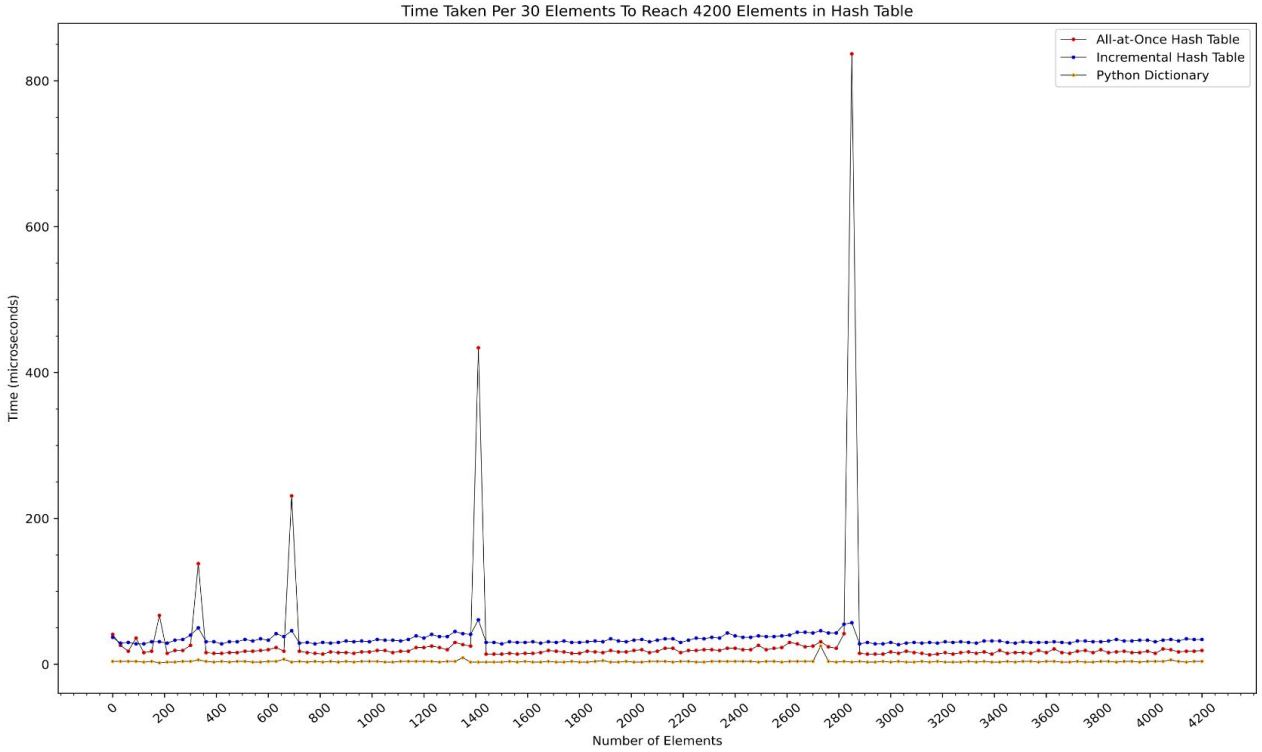
```
  def __contains__(self, k):
-     index = self._get_index(k, self.key_table)
+     oindex = self._get_index(k, self.key_table)
+     cindex = self._get_index(k, self.ckey_table)

-     return self.key_table[index] == k
+     return (self.key_table[oindex] == k
+          or self.ckey_table[cindex] == k)

  def __getitem__(self, k):
      index = self._get_index(k, self.key_table]
      if self.key_table[index] == k:
          return self.value_table[index]

+     index = self._get_index(k, self.ckey_table)
+     if self.ckey_table[index] == k:
+         return self.cvalue_table[index]

      raise KeyError(f'The key "{k}" of type
{type(k).__name__} is not in the table.')
```

Diff between __contains__() and __getitem__() implementations All-at-once and Incremental hash tables.

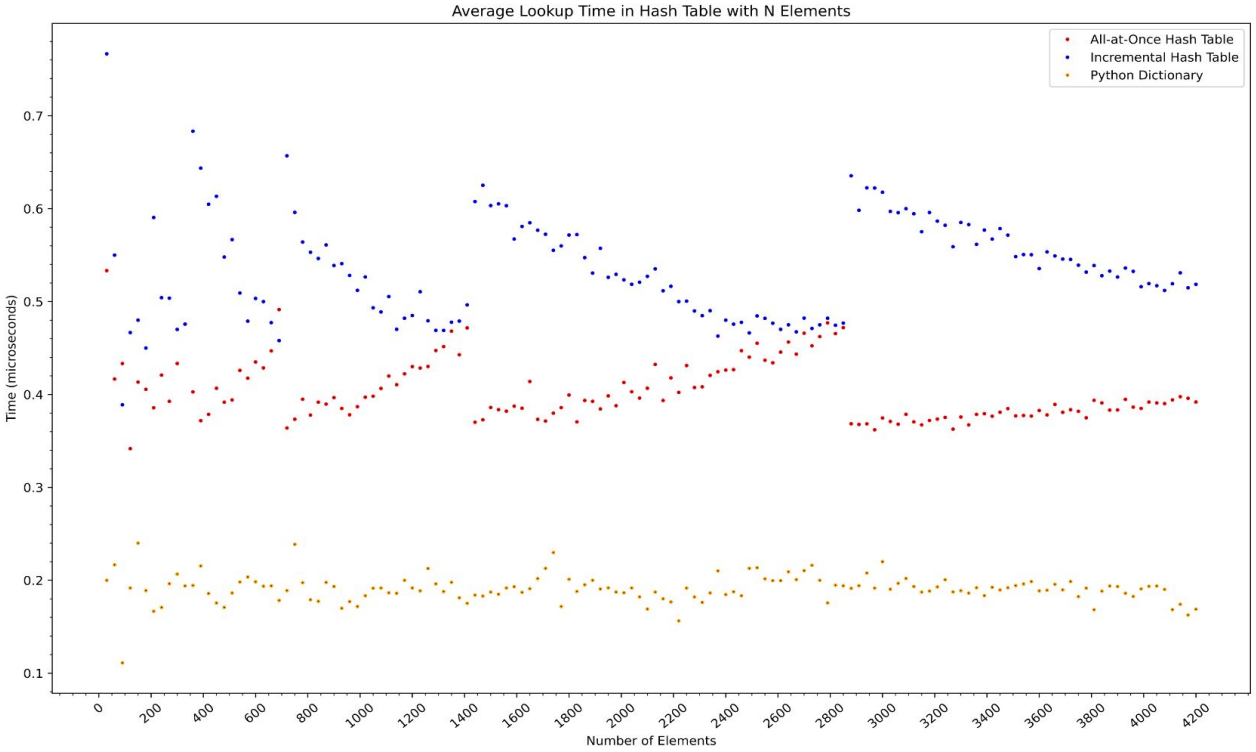
Benchmarks


Time Taken To Add N Number of Elements To Hash Table
All-at-Once Hash Table
Incremental Hash Table
Python Dictionary
Time (microseconds)
0
1000
2000
3000
4000
Number of Elements
0
200
400
600
800
1000
1200
1400
1600
1800
2000
2200
2400
2600
2800
3000
3200
3400
3600
3800
4000
4200

Benchmarks


Time Taken Per 30 Elements To Reach 4200 Elements in Hash Table
All-at-Once Hash Table
Incremental Hash Table
Python Dictionary
Time (microseconds)
0
200
400
600
800
Number of Elements
0
200
400
600
800
1000
1200
1400
1600
1800
2000
2200
2400
2600
2800
3000
3200
3400
3600
3800
4000
4200

Benchmarks


Average Lookup Time in Hash Table with N Elements
All-at-Once Hash Table
Incremental Hash Table
Python Dictionary
Time (microseconds)
0.1
0.2
0.3
0.4
0.5
0.6
0.7
0
200
400
600
800
1000
1200
1400
1600
1800
2000
2200
2400
2600
2800
3000
3200
3400
3600
3800
4000
4200
Number of Elements