Cuckoo Hashing

An Overview of The Algorithm and Its Associated Data Structures

Henok Woldemichael, Siaba Bamba

Department of Mathematics and Computer Science, University of Lethbridge

Computer Science 3620: Data Structures and Algorithms

Professor Daya Gaur

December 8, 2021

Cuckoo Hashing was first introduced by Rasmus Pagh in 2001. Cuckoo Hashing is a simple collision-resolution algorithm that provides for constant-time worst-case lookups and deletions, as well as amortised constant-time insertions. The term "amortised constant time" refers to the fact that while a single invocation of an operation may incur a significant runtime cost (for example, in a hash table, the table is resized to fit an increased number of elements), the average runtime over these invocations is constant. Cuckoo Hashing is named after the cuckoo bird, because the algorithm resembles the cuckoo bird's behaviour of throwing out the eggs of other birds, and leaving its eggs in their nest for them to take care of.

Our application(like most) of Cuckoo Hashing uses two hash tables and a dictionary. The class we created uses a method "cuckoo(dictionary)" which takes in a dictionary and returns the two hash tables with the dictionary values in their respective positions according to the algorithm. The dictionary data type in computer science is a fundamental data structure that allows users to store key-value pairs and look up the appropriate value in the data structure by the appropriate key. Hash tables are one of the most prevalent and fastest implementations of the dictionary data structure. The hash table allows for amortised constant-time lookup, insertion, and deletion operations.

Hash Table:

Search(Relevant Key)

Returns true if the Relevant Key is an index in the table, false otherwise
This function should take an amortized O(1) constant time because no matter the number of elements in the hash table the Relevant Key will be located at a constant rate.

Insertion(Relevant Key)

Adds the item Relevant Key to the hash table if it is not already included.
This function should take an amortized O(1) constant time because no matter the number of elements in the hash table the Relevant Key will be inserted at a constant rate.

Deletion(Relevant Key)

Removes the Relevant Key from the hash table. This function should take an amortized O(1) constant time because no matter the
number of elements in the hash table the Relevant Key will be located at a constant rate.

A hash function also known as a hashing algorithm is a calculation applied to a key to convert it into a proportionately small index number that corresponds to a position in a hash table. This index number is functionally a memory address. In our program we accept dictionary keys that are words, so we convert the words into their corresponding ASCII values by utilizing the order() function provided by the Python 3 programming language adding each letters ASCII value, then modding it by the size of the hash table to calculate it's index.

Collisions:

Because a hash function returns a relatively small number for a large key, two keys may return the same value. Collision occurs when a newly inserted key maps to an existing occupied space in a hash table and must be dealt with using a collision handling mechanism.

The likeliness of collisions is illustrated by the probability theory problem known as The Birthday Paradox which states at least two people in a group of n will have the same birthday. The birthday paradox is that, counterintuitively, in a group of only 23 persons, the likelihood of sharing a birthday approaches 50%.

Closed addressing or Chaining:

A chained hash table is a hash table that resolves collisions by putting all colliding elements into the same bucket. To find an element you hash to the element's bucket and scan for it to see if it is present. You would use a data structure like a linked list in this case to store the elements.

Open addressing:

Open addressing consists of letting elements flow into other spaces in the hash table to Resolve collisions. An example of this is linear probing.

The expected, amortised cost of a lookup with universal hash functions is O(1), but the expected worst-case cost of a lookup in Open Addressing (with linear probing) is O(log n) and in a chained hash table it is O(log n / log log n). The following are two approaches that are utilised to bridge the gap between expected and worst-case expected times.

Hashing with numerous options: Give each element multiple options for where it can live in the hash table

Hashing through relocation:
Allow elements in the hash table to move after they've been placed using relocation hashing

Cuckoo hashing applies both these methods and promises a O(1) worst case look up time.
Hashing with numerous options:

We use two different hash functions to calculate the index. h1(Relevant key), and h2(Relevant Key).

Hashing through relocation:

This is where the fun is. In the case that the hash function h1(Relevant key) produces an index that is already occupied in the hash table, we emulate the cuckoo bird's brash behaviour and delete the element that is occupying the space. We then insert the current Relevant Key, and then use the h2(Relevant Key) index of the deleted element, and insert it into the second hash table.

Cycle of death:
If both hash functions of a key produce indexes that are already occupied, and the displaced keys also do this, then you encounter an infinite loop, or a cycle. In this case you would need to generate two new hash functions and two new hash tables and run the Cuckoo Hashing algorithm on the dictionary of data again, multiple rehashes may be necessary for Cuckoo Hashing to work with large data sets.

The load factor of a hash table is the ratio of the number of elements (n) versus the total number of buckets(b). A bucket is the indexes in a hash table. So the load factor of a table is calculated by dividing n by b or load factor = n/b.

Cuckoo Hashing Insertion:

Even with the possibility of rehashing, insertion is predicted to be O(1) (amortised) with a high probability as long as the number of keys is kept below half of the hash table's capacity or in other words the load factor is below 50%.

Cuckoo Hashing Deletion:

Deletion is O(1) worst-case since constantly only requires inspection of two hash table locations.

Experiments:
Our Cuckoo Hashing program allows for keys that are words, and numbers.

Sample Input:

```
 thisdict = {
  "Mia": "Ford",
  "Sue": "Mustang",
  "yeor": 1964
}
```

Relevant Keys:
“Mia”, Sue, yeor

	“Mia”	“Sue”	“yeor”
h1(Relevant Key)	0	1	1
h2(Relevant Key	1	1	0

```
Sample Output:
['Ford', 1964, None, None, None,None]
['Mustang', None, None, None, None, None]
```

What happened?

Here the hash function simply mods the key by the size of the dictionary. In the previous Sample input the keys Sue and yeor both produce the same index 1, after being passed into the hash function. As you can see, according to the algorithm, first the value “Mustang” which is associated with the dictionary key “Sue” wass placed in the first hash array at position[1], but then it is deleted due to the value 1964 which is associated with the dictionary “yeor” having the same hash index as it. After deleting the value associated with “Sue”, and placing the value associated with “yeor”, the value associated with “Sue” (“Mustang”) is then placed in its respective index in the second hash array, based on its second hash function index at position[0].

We attempted searching, deleting and inserting into the hash arrays, and even though we had dictionaries with ten times the data. All of these operations were constant as expected, and as outlined by the previous explanations.

References

Cuckoo hashing - stanford university. (n.d.). Retrieved December 9, 2021, from https://web.stanford.edu/class/archive/cs/cs166/cs166.1146/lectures/13/Small13.pdf.

Cuckoo hashing - worst case O(1) lookup! GeeksforGeeks. (2021, September 13). Retrieved December 9, 2021, from https://www.geeksforgeeks.org/cuckoo-hashing/.

Cuckoo hashing - stanford university. (n.d.). Retrieved December 9, 2021, from https://web.stanford.edu/class/archive/cs/cs166/cs166.1146/lectures/13/Small13.pdf.