Project 3

Cuckoo Hashing

Anthony Lupica

arl127@uakron.edu

Abstract

Cuckoo hashing is a popular option when it is necessary to perform exact-match inserts, searches, and deletes in a worst-case O(1). It is particularly useful when the client does not have information regarding the quality of keys that are inserted. If the worst-case input is unavoidable—input which results in indices that produce tight clustering in the hash table—it may be a good choice to look into the cuckoo technique. Tight clustering would mean increased frequency of collisions and the use of potentially costly resolution policies, which is where cuckoo shines.

Introduction

Our goal in this report is to look in depth at cuckoo hashing, analyzing some of its potential difficulties, and verify its efficiency. We aim to empirically demonstrate constant time inserts, searches, and deletes. Also, we will test varying inputs to measure if *secondary clustering* can trigger longer eviction cycles, and thus longer insert times. Finally, we will

compare to an alternate hashing method, two-choice, and attempt to prove that cuckoo, comparatively, is less effected by input size.

Discussion

Hypothesis: cuckoo hashing will show constant time operations, including inserts which occasionally run long due to near half load factors, and requiring a rehash. Also, secondary clustering will affect performance initially, but all effects will dissipate once several rehash operations have occurred.

Hashing is the structure of choice for *exact-match queries*, as it has typically stellar time-complexity. In the average case the big three operations of insert, search, and delete are O(1). Still, in the worst case, many implementation techniques for open addressing hash tables are O(N) for these operations.[1] This occurs when a *probe sequence* is required to exhaust all elements along its path in the table before it arrives (or finds that it can't) at the requested record. Is it possible to achieve hash table inserts, searches, and deletes in a worst case O(1) complexity as well?

The goal of any implementation of cuckoo hashing is, exactly this, to try and ensure O(1) time for inserts, searches, and deletes in the worst case.[2] Cuckoo's ability to do this when implemented well is what makes it such an attractive option. It's predicated on the idea that new inserts into the table will "evict" any records that cause a collision from the hash spot to make room for a new record. To do this it maintains two arrays, each of size M elements—

call them table 1 and table 2—and two distinct hash functions, h1() and h2(), which are responsible for mapping inputs to their like-numbered table.[2] h1() computes the "home position" for records, that is, their primary, and preferred, location found in table 1. Table 2, and h2(), may be seen then as backup in case a record needs to move out of its home position.

As with any hashing technique, choosing hash functions that can guarantee an even distribution of mappings to hash slots is paramount to the efficacy of the structure. Cuckoo's claim to fame, however, is not it's use of well-thought hash functions, but how it manages these two tables in tandem when a collision occurs—it's *collision resolution policy*. Unlike implementations that feature a probe sequence (a good one will eventually hit all indices of the table), records in a cuckoo hash table are only ever in one of two spots: the index found by h1() for table 1, or the index found by h2() for table 2. For search and delete, one must only check these two spots to unequivocally say whether the record exists. Collisions will happen for insertions, but these are handled by "evicting" records from table 1 to table 2, or table 2 back to table 1, and inserting new records in their place. Records are technically only ever directly inserted into table 1, and all other "jostling" of records qualifies as the result of evictions. Once collisions threaten to become too frequent, we rehash with a larger table to maintain constant time inserts.

For tables 1 and 2, hash functions h1() and h2(), and keys input in order A through C, the following is a depiction of a possible result…

	table 1	table 2
0	A	B
1	C	
2		

	h1()	h2()
A:	0	2
B:	1	0
C:	1	2

Figure 1

The right table shows the keys to be inserted, and the result of calling h1() and h2() on each one. A is first inserted in table 1 at index 0, as h1() takes precedence over h2(). B is then inserted in table 1 at index 1. However, h1() computed the same hash slot for C that it did for B—a collision. To resolve the issue, B is evicted to its secondary hash slot in table 2 at index 0, and C takes ownership of the slot. All inserted records can be found at either their index found by h1(), or their index found by h2().

Some issues may arise with cuckoo hashing, such as a cycle with a (possibly) endless loop of evictions. In the above example, it's clear to see how such an occurrence may arise if you imagine a record, D, were already in table 2 at index 0 when B was evicted to that slot. In that case, B would still be able to enter but would displace D back to table 1. While this would be only two evictions, assuming D doesn't go on to evict a record on its path to a new home, these cycles can become troublingly long in some circumstances. In this case, there

may be no “resting state” where all evicted records have safely been reseated at a new index. An insert of a record that causes an eviction cycle is surely not O(1), so naturally an implementer of a cuckoo hash table would be best to devise some method of minimizing and preventing long eviction cycles. It’s possible that any given input could spur a cascading eviction avalanche. Our only defense against this is to use some method of detecting that a cycle is happening, or often we establish a condition where if an eviction cycle of at least $\log_2 N$ (with N being the number of total records among both tables) length occurs, we rehash the tables with about double the table size.

A stipulation of the constant time goal of cuckoo hashing is, of course, that no insert can truly be constant time in the worst case if we aim to create a structure that conveniently resizes on the fly. This is due to the memory allocation of new underlying structures, and costly operations to move data from the old memory to the new memory. In the case of Cuckoo, and any hashing implementation that supports load factor balancing, this is more specifically due to the rehash operation. In our case, rehash is called to approximately double the size of the table. Not only is this convenient for the user, but necessary to maintain the time efficiency of the structure.

The load factor α is defined as…

$\alpha = N/M$; where N is the number of records in the table, and M is the size of the table.

To minimize the frequency of collisions, and resultantly minimize the likelihood of an eviction cycle, we would like our tables to have load factors of less than half the size of the table at all times.[2] When this condition triggers a call to rehash, we are effectively trading space efficiency for time efficiency, as there will always be more space in the tables than the user is allowed to populate, and they can only ever be less than half full. It's a worthy tradeoff in the name of constant time. We say insert() is *amortized* and can think of it as O(1), because calls to the *definitely not O(1)* rehash become more and more infrequent as the size of the tables grow. Delete here does not have the same concerns that a delete using a probe sequence would have. Using a probe sequence, we need to make sure that deleted records do not disrupt the flow of the probe sequence for future probing. In cuckoo, delete need only look in two places, no probe sequence required.

My Implementation

Some implementation details of my particular cuckoo hash are in order. I opted for table sizes that begin at eleven, and upon a rehash, are doubled and rounded up to the nearest prime number. Only prime numbers are used, as applying modulus over prime table sizes contributes to a more even distribution of map index locations. When making a hash table, you have the choice of which type of keys and values you wish to store, and also the association held between them. I went with std:string keys, and integer values. The table

anticipates that the user will insert keys as a name, and values that represent some association with the name as a corresponding year. Thus, I enforce that values be in four-digit form to strengthen the class's coupling to a name-year pair. The choice is up to the user as to what name-year association to store. This can be anything, such as a student and their graduation year, pets and their year of adoption, or the example that I favored in my testing, celebrities and their birth years. The name-year pairs are stored internally in one of two dynamic arrays (table 1 or table 2) as struct nodes. The one operation that is comparatively inefficient in my design is display(), it always has to visit every struct node within both tables, even if they contain no records. This is because we are not simply looking to verify that a single record exists, and the user does not provide a key to be hashed, we need to loop through to manually find records. Furthermore, because the tables are only ever less than half full, display() iterates through many records that are "out-of-bounds" to the user. My version of a delete reorients all records to again be biased towards table 1, which it does by recalling an overloaded insert helper on all records.

A rehash occurs under two potential circumstances: like previously described, if either table becomes half full, or upon an eviction cycle of length greater than or equal to $\log_2 N$. Rehashes are relatively frequent with minimal inserts but become sparse as the structure becomes fuller. To illustrate the amortization of inserts, this is the result of running inserts in a for-loop, where the condition variable, i, for iteration is initialized to 33, and updates to

126. This range is chosen to represent characters ‘!’ through ‘~’ in ASCII. In the loop body, the ASCII code of i is converted to a string and inserted into the hash object…

```
tableSize increases for inserts in a loop: 11 11 11 11 11 23 23 23 23 23 47 47 47 47 47 47 47 47 47 47
 47 97 97 97 97 97 97 97 97 97 97 97 97 97 97 97 97 97 97 97 97 97 97 97 97 197 197 197 197 197 197 1
97 197 197 197 197 197 197 197 197 197 197 197 197 197 197 197 197 197 197 197 197 197 197 197 197 1
97 197 197 197 197 197 197 197 197 197 197

| " : 1000 || D : 1000 || f : 1000 || ) : 1000 || K : 1000 || m : 1000 || 0 : 1000 || R : 1000 || t : 1000 |
| 7 : 1000 || Y : 1000 || { : 1000 || > : 1000 || ` : 1000 || # : 1000 || E : 1000 || g : 1000 || * : 1000 |
| L : 1000 || n : 1000 || 1 : 1000 || S : 1000 || u : 1000 || 8 : 1000 || Z : 1000 || | : 1000 || ? : 1000 |
| a : 1000 || $ : 1000 || F : 1000 || h : 1000 || + : 1000 || M : 1000 || o : 1000 || 2 : 1000 || T : 1000 |
| v : 1000 || 9 : 1000 || [ : 1000 || } : 1000 || @ : 1000 || b : 1000 || % : 1000 || G : 1000 || i : 1000 |
| , : 1000 || N : 1000 || p : 1000 || 3 : 1000 || U : 1000 || w : 1000 || : : 1000 || \ : 1000 || A : 1000 |
| c : 1000 || & : 1000 || H : 1000 || j : 1000 || - : 1000 || O : 1000 || q : 1000 || 4 : 1000 || V : 1000 |
| x : 1000 || ; : 1000 || ] : 1000 || B : 1000 || d : 1000 || ' : 1000 || I : 1000 || k : 1000 || . : 1000 |
| P : 1000 || r : 1000 || 5 : 1000 || W : 1000 || y : 1000 || < : 1000 || ^ : 1000 || ! : 1000 || C : 1000 |
| e : 1000 || ( : 1000 || J : 1000 || l : 1000 || / : 1000 || Q : 1000 || s : 1000 || 6 : 1000 || X : 1000 |
| z : 1000 || = : 1000 || _ : 1000 |
```

Figure 2

The table size is output to the screen after each input, and the entire hash table is displayed following all insertions, 93 in total. An arbitrary year of 1000 is used for all inserts.

The frequency of calls to rehash to maintain the load factor begins high, but eventually they become a rarity.

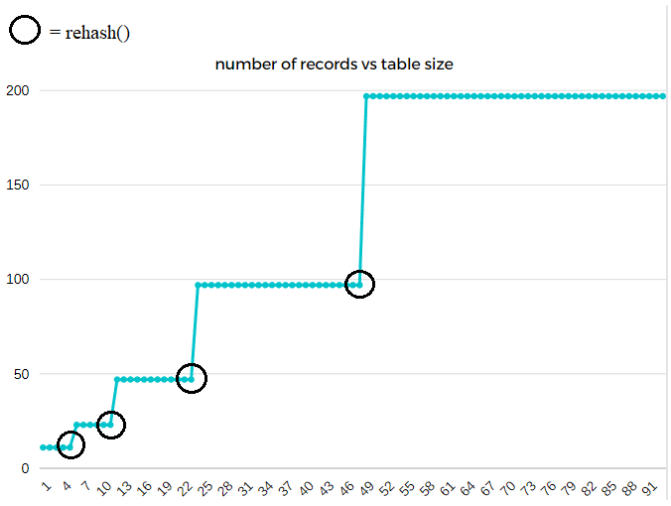


Figure 3

A chart pitting the number of records against the size of the tables in use shows the density of calls to rehash() thinning out over time. The prevalence of rehash() calls follows a log(N) degradation.

Because cuckoo hashing does not technically use a probe sequence, of which we could use to check that all hash slots have been visited, it becomes more important to place restrictions on and closely monitor how many evictions have taken place per insert. Seen here, my implementation does not allow the load factor to encroach on or greatly affect the length of eviction cycles as a function of the record count.

Testing Methodology

Insert, search, and delete is timed for each input in an iteration loop of length thirty.

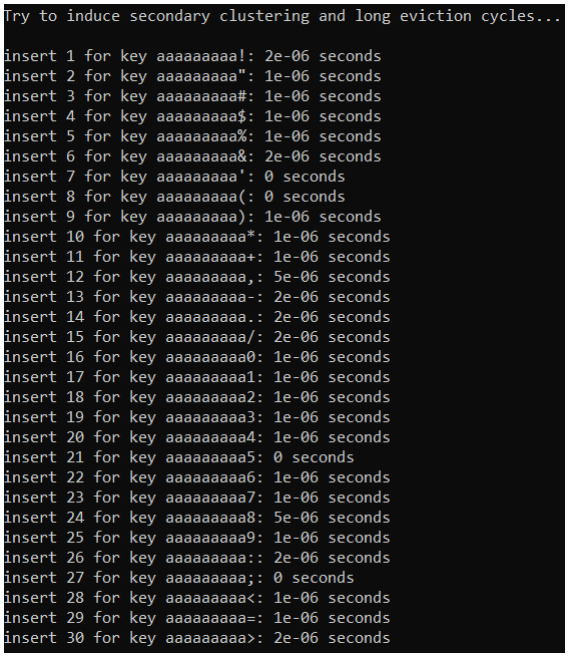
This is done both for inputs attempting to provide a wide range of characters, and for inputs attempting to provide very little, in order to trigger long eviction cycles.

```
Try to induce secondary clustering and long eviction cycles...

| aaaaaaaaa; : 1000 || aaaaaaaaa* : 1000 || aaaaaaaaa. : 1000 || aaaaaaaaa2 : 1000 || aaaaaaaaa! : 1000 || aaaaaaaaa6 : 1000 |
| aaaaaaaaa% : 1000 || aaaaaaaaa: : 1000 || aaaaaaaaa) : 1000 || aaaaaaaaa> : 1000 || aaaaaaaaa- : 1000 || aaaaaaaaa1 : 1000 |
| aaaaaaaaa5 : 1000 || aaaaaaaaa$ : 1000 || aaaaaaaaa9 : 1000 || aaaaaaaaa( : 1000 || aaaaaaaaa= : 1000 || aaaaaaaaa, : 1000 |
| aaaaaaaaa0 : 1000 || aaaaaaaaa4 : 1000 || aaaaaaaaa# : 1000 || aaaaaaaaa8 : 1000 || aaaaaaaaa' : 1000 || aaaaaaaaa< : 1000 |
| aaaaaaaaa+ : 1000 || aaaaaaaaa/ : 1000 || aaaaaaaaa3 : 1000 || aaaaaaaaa" : 1000 || aaaaaaaaa7 : 1000 || aaaaaaaaa& : 1000 |


Provide a list of celebrities with wide character variation...

| Tony Leung : 1962 || Alanis Morissette : 1974 || Zendaya : 1996 || Jay-Z : 1969 || Jason Mraz : 1977 || Winona Ryder : 1971
|| Drake : 1986 || Ice Cube : 1969 || Michael Fassbender : 1977 || Dr. Phil : 1950 || Tom Holland : 1996 || Dax Shepard : 1975
 || Julian Casablancas : 1978 || Javier Bardem : 1969 || Jake Gyllenhaal : 1980 || Joaquin Phoenix : 1974 || Björk : 1965 || E
llen DeGeneres : 1958 || Robert Herjavec : 1962 || Cardi B : 1992 || Idris Elba : 1972 || Taylor Swift : 1989 || Zach Galifian
akis : 1969 || Matthew McConaughey : 1969 || George Washington : 1732 || Kyrie Irving : 1992 || Alicia Keys : 1981 || Hugh Lau
rie : 1959 || Adele : 1988 || Henry VIII : 1491 |
```

Figure 4

Shows the two input data sets. The first consists of strings which are more likely to hash to identical slots. The second provides the intended variation of keys to ensure minimal cycling.

Analysis



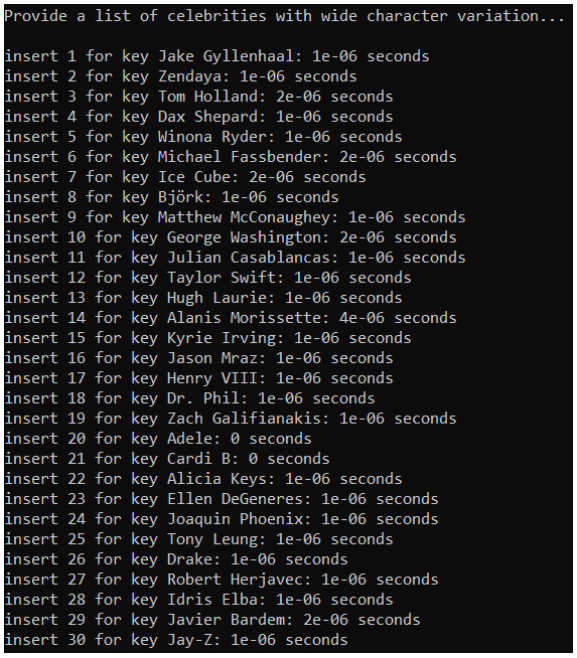


Figure 5

The closely related keys inserted in the left image result in what appears to be a less stable time complexity. This is due to inserts frequently requiring additional reseating of displaced records, which in turn results in additional calls to rehash. The hash functions used will likely compute indices very near each other for those keys. On the right, we see the records uniformly seem to insert in 0.000001 seconds or less, regardless of the number of inserts. Occasionally, of course, we do see a spike in runtime, and this is caused by the rehash operation, which seen here becomes more and more infrequent. This version also normalizes much sooner than the left insert loop. Notably, when both

insert loops are extended by orders of 10 (100, 1000, 10000…) records, runtimes stay within range of a few decimal places. They never scale along with the size of the structure; relatively unaffected by the number of records.

Now we look at search and delete…

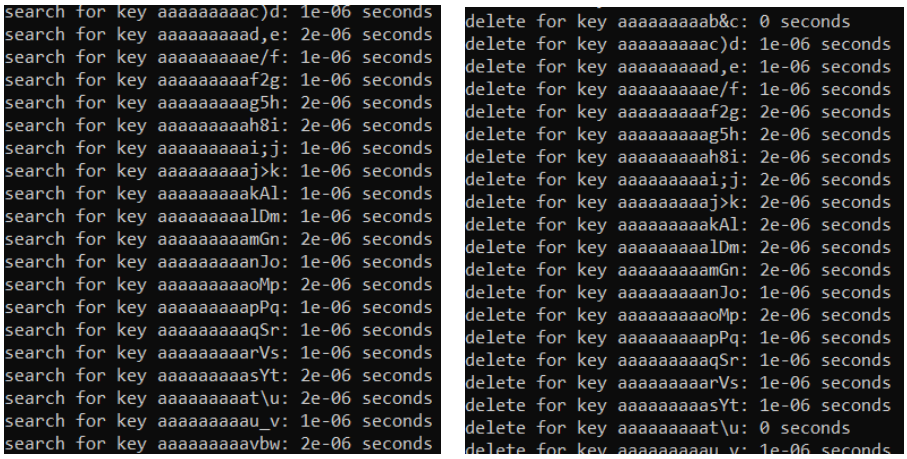


Figure 6

This data was found by inserting 100000 unique keys, and then subsequently running search and delete on those records. As the output was very large, this is just the final portion of it, showing that search and delete times are unaffected by insert sizes.

Insert, search, and delete are shown to run constant time in the worst and in the ideal case.

Now, let's compare to a two-choice hashing implementation, provided by Andy Mee. Two-choice is quite similar to cuckoo; they both play off the idea of "the power of two choices." Cuckoo provides the choice of two tables, and two-choice provides the choice of two hash

functions. The one which produces the least collisions will be used. From there, Andy utilized double hashing for a probing sequence. Invariably, Andy's structure always inserted and deleted in constant time. The search operation, however, ran slightly longer in time than cuckoo's search, while still being constant time. I believe the reason Andy's two-choice ran faster on average is because he opted to use integer keys and did not process values alongside them. My implementation had to iterate through string keys to compute a hash index on each insert, and move string objects upon evictions, as opposed to lightweight primitives, operations which likely forgo some efficiency. Two-choice gives two potential spots at a time for a new insert, using different hash functions. This, coinciding with the double hashing probe sequence, likely cuts down on secondary clustering, which allows search and delete to reliably be found not to far from their home positions. The fact that there is always a second position being checked nearly cuts the time complexity in half. Both two-choice and cuckoo are ultimately good options to alleviate clustering.

Conclusion

Cuckoo hashing is not affected by load size for the insert, search, and delete operations due to allowing evictions between two tables up to some eviction limit constant. Rehash, while O(N) itself, is amortized and does not greatly affect performance. I was also able to produce some distinguishable differences in run time stability of cuckoo when insert keys are tightly clumped and contribute to the mapping of similar hash indexes.

References

1) Iyer, J. V. (2021, December 6). *Time and space complexity of hash table operations*. OpenGenus IQ: Computing Expertise & Legacy. Retrieved April 24, 2022, from https://iq.opengenus.org/time-complexity-of-hash-table/

2) Unit 2 DS09 - Searching. Demo "*Hashing Part 4 – perfect and cuckoo_default".*

3) Shaffer, C. A. (2011). *Data Structures & Algorithm Analysis in C++*. Dover Publications.

4) DS09 – Searching. U2 ds09 Searching slides.