

Cuckoo Hashing vs Open Addressing (Linear Probing)

41052 Advanced Algorithms

Ilias Vasiliou
Craig Jones

Assessment Task 3 - Group Programming
Assignment



Contents

1 Problem Definition	2
1.1 Assumptions and restrictions.	2
2 Algorithm Outlines	3
2.1 Open Addressing with Linear Probing	3
2.2 Cuckoo Hashing with Tabulation Hashing	3
3 Theoretical & Implemented Complexity	4
3.1 Open Addressing	4
3.1.1 Theoretical	4
3.1.2 Implemented Complexity and Practical Considerations	4
3.2 Cuckoo Hashing	4
3.2.1 Theoretical	4
3.2.2 Implemented Complexity and Practical Considerations	5
4 Testing & Correctness	5
4.0.1 Open Addressing Tests	5
4.1 Cuckoo Hashing Tests	6
5 Results Evaluation & Conclusion	6
5.1 Empirical comparison	7

1 Problem Definition

The aim of this work was to design, implement, and evaluate two dictionary-style data structures that support insertion, membership queries, and deletion in expected constant time. The target interface was a key-value map for integer keys (for the open-addressing version) and a set-like structure for 64-bit unsigned integers (for the cuckoo version). The problem is non-trivial because high performance must be maintained even when the table becomes moderately full and when deletions occur. In addition, the implementation must deal with clustering in open addressing, cycles in cuckoo hashing, and the need to resize/rehash when the current configuration no longer guarantees constant-time access.

1.1 Assumptions and restrictions.

- Keys for the open-addressing hash map are 32-bit signed integers; values are 32-bit signed integers.
- Keys for the cuckoo table are 64-bit unsigned integers; the value is implicit (membership).
- A load factor of about 0.7 was considered an acceptable upper bound for open addressing; beyond this the table is proactively rehashed to preserve probe lengths close to constant.
- For cuckoo hashing, a fixed maximum number of displacements (“kicks”) per insertion was chosen; failure to place an item within that budget triggers a rehash.
- The hash functions must be fast and must distribute keys well. For the map, the standard integer hash provided by the language runtime was sufficient; for the cuckoo table, tabulation hashing was used to obtain two independent-looking hash values over 64-bit items.
- Within these assumptions, the problem reduces to: store and retrieve many integer items so that all basic operations run in $O(1)$ expected time and remain robust under insertions, deletions, and growth.

2 Algorithm Outlines

2.1 Open Addressing with Linear Probing

The first structure is a classic open-addressed hash table. Keys are mapped to an array slot by hashing and then, if the slot is not suitable, by probing linearly through the array (index + 1, wrapping) until a suitable slot is found. Each slot carries a small state: EMPTY (never used), OCCUPIED (stores a key–value pair), or DELETED (tombstone). Insertions start at the home slot, remember the first tombstone and finally write the new key–value in either that tombstone or in the first empty slot. Deletions do not shift later elements; instead, a tombstone is written so that subsequent searches can continue past that position. The table monitors the effective load factor $+ \text{tombstones}/\text{capacity}$; if it exceeds 0.7, the table is rehashed into an array twice as large and all live entries are re-inserted. Lookup starts from the home slot and stops only when an EMPTY slot is seen or the key is found. This design emphasises simplicity, locality of reference, and amortised-constant rehashing.

2.2 Cuckoo Hashing with Tabulation Hashing

The second structure varies significantly, utilising Tabulation Hashing as our selected method to create a low probability for collisions. Our Tabulation Hash functions via splitting the key into 8 blocks, and XOR Operations for low probability for collisions. Two Tables of values for XOR operations are created, allowing us to create two hash functions, by taking a relative value of our XOR to the size of our Cuckoo Table. Items are attempted to be inserted into each of the two tables using the two distinct hash functions and consequent locations, otherwise kicking out items in a random selection of the two locations. This process may repeat until a pre-determined number of times before triggering a rehash. Lookups are performed using the pre-determined hash functions allowing two constant time operations to find an item.

3 Theoretical & Implemented Complexity

3.1 Open Addressing

3.1.1 Theoretical

The theoretical model of a hash table with open addressing and linear probing has the following complexity:

- Successful search: expected $O(1)$ when the load factor $\alpha < 1$ and clustering is moderate
- Unsuccessful search: expected $O(1/(1 - \alpha))$ probes, with $\alpha \leq 0.7$ this is still a small constant.
- Insertion: same as unsuccessful search, hence expected $O(1)$ as long as α is bounded away from 1.
- Deletion: $O(1)$ as it only marks a tombstone,
- Rehash: $O(n)$, but rehashing only happens after a geometric increase in capacity, so the amortised cost per operation stays $O(1)$.

3.1.2 Implemented Complexity and Practical Considerations

The implementation counts tombstones and includes them in the load-factor calculation. This is important as if the tombstones were ignored, the table could become effectively full even though the live element count is low. By rehashing when $(\text{live} + \text{tombstones}) / \text{capacity} > 0.7$, probe lengths are kept short, and deletions do not slowly degrade performance. Moreover, insertions detect when the probed slot already holds the key and then updates the value in place instead of inserting a duplicate.

This preserves the dictionary semantics and ensures that insertion remains $O(1)$ even in the presence of repeats. A standard integer hash was considered sufficient because keys are integers and the table size is always adapted to the data. The cost of hashing is therefore $O(1)$.

3.2 Cuckoo Hashing

3.2.1 Theoretical

The theoretical complexity of a Cuckoo Hash Table is as follows.

- Search : $O(1)$ due to two constant time operations to check both hash locations.

- Insertion : Expected time is $O(1)$ due to the two constant time hash location checks, but if a cycle occurs, or the max number of kicks is reached, a rehash is reached causing $O(n)$ insertion time. Consequently, we can conclude it is amortized $O(1)$.
- Deletion : $O(1)$ due to constant time find operation, we can simply replace the found item with a sentinel value.
- Rehash : $O(n)$ where n is representative of all current items within both tables.

3.2.2 Implemented Complexity and Practical Considerations

An important consideration was made regarding the presence of zeroes within the table. As they are used for sentinel values, they can not be inserted into the table. This was chosen as an intentional design choice, as otherwise additional memory overhead is required to accommodate the position of all items.

4 Testing & Correctness

Correctness was demonstrated through a structured unit-test suite built with GTEST. The goal of the suite was to exercise every transition of the open-addressing table and cuckoo hashing table and to confirm that the more intricate behaviours are implemented as intended.

4.0.1 Open Addressing Tests

The suite begins with a construction test to confirm that a freshly created map does not report spurious membership. It then checks the core dictionary behaviour: inserting a new key returns success, the key becomes visible to contains, and at returns the stored value. A separate test targets updates by checking that inserting the same key again must not be treated as a new insertion, and the value must be overwritten in place. This is important to maintain map semantics rather than multimap semantics. Further, deletion is tested in two ways. Firstly, checking that deleting an existing key removes it from the table and returns success. Secondly, testing that deleting the same key again fails cleanly, which shows the table is not leaving "ghost" entries around.

Because this implementation uses tombstones, it is essential to show that later insertions and lookups can traverse past deleted slots. A targeted test

deletes an early key and then inserts a different key that should reuse or probe through that tombstone. The fact that the new key can be inserted and later found shows that the search logic correctly distinguishes EMPTY (stop) from DELETED (keep probing).

To validate the amortised $O(1)$ growth path, the suite inserts a few hundred consecutive integers which is enough to push the effective load factor past the 0.7 threshold and trigger a rehash. After the rehash, the test re-queries every key and checks that `contains` returns true and `at` returns the exact value originally stored. This shows that the rehash routine correctly reinserts all occupied slots into the new table and that no data is lost during resizing,. The same test then deletes a key after rehash and verifies that lookups fail and that `at` throws, ensuring that deletion still works on the resized structure.

4.1 Cuckoo Hashing Tests

A number of tests are provided in the suite to ensure correctness. We first check for basic functionality of both insertion and deletion, ensuring the table correctly can confirm such operations with `contains`. We confirm that zero can not be entered into the table as is covered in the practical limitations section, and check that duplicates are not able to be inserted. Multiple insertions are performed to ensure all insertions are found without triggering a rehash prematurely, and removal is checked to ensure it functions as intended. We validate the rehash which is important, checking that it occurs when several elements are inserted, and also that all elements are present after the rehash.

5 Results Evaluation & Conclusion

This report presented two hash-table - open addressing with linear probing and cuckoo hashing - and showed through unit tests and benchmarking that both meet the assignment requirement of supporting insertion, membership and deletion expected constant time. The open-addressing design emphasised simplicity, tight cache locality and a conservative load-factor policy that counts tombstones. The cuckoo design by contrast used two tabulation-based hash functions and bounded kick-out insertion to guarantee that a successful lookup needs at most two probes, and it relied on occasional full rehashes to escape bad configurations.

5.1 Empirical comparison

The benchmark on 10^5 random keys showed a clear difference in where each structure is fastest: open addressing inserted much faster ($\approx 2.7ms$ vs $\approx 70.7ms$) and was slightly faster to lookup ($\approx 1.35ms$ vs $\approx 3.06ms$) but slower to erase ($\approx 2.46ms$ vs $\approx 1.34ms$). Open addressing clearly dominated on insertion (over an order of magnitude faster) and was also faster on lookup. The one operation where cuckoo hashing was better in this benchmark was erase. This is largely due to the fact that its deletion is just "check two places and clear", whereas open addressing must preserve the probe chain with tombstones, which makes deletes costlier, especially after many inserts/deletes. The numbers therefore match the designs: open addressing is the better choice for general map-style workloads that insert a lot and query a lot; cuckoo hashing is attractive when you can afford expensive inserts but want simple, bounded-time queries and deletes.

Structure	Insert (ms)	Lookup (ms)	Erase (ms)
Open addressing	2.68525	1.35808	2.46421
Cuckoo hashing	70.7477	3.05767	1.3405

Table 1: Performance comparison of open addressing and cuckoo hashing for the same workload.