Ahmed-Zayne El Husseini - 24418699 / Paolo Lamesa - 24824815

Advanced Algorithms - Assignment 3

Comparison of deterministic and probabilistic cuckoo hashing

1 The problem of collisions in hash tables

This section provides a brief overview of hash tables, the problem of collisions and some common collision resolution methods apart from cuckoo hashing.

1.1 Overview of hash tables

Hash tables are the most common way that unordered dictionaries are implemented in general modern computing. They're a useful data structure that allow developers to store and retrieve any hash-able data type in amortised constant time using a relatively small chunk of memory.

Hash tables consist of an array of size *m*, where *m* is in the order of the expected number of values we wish to store.
Let *S* denote the set of all possible values (or keys) we wish to store. A hash function maps each key in *S* to one of the *m* positions (or buckets) in the array, so on average each bucket corresponds to |*S*| / *m* possible keys.

When data is to be stored or retrieved from the hash table, the hash function is applied to the key, and the result tells us at what position in the array we should store or look for the data. This allows us to store and retrieve any key from *S* in O(1) time without allocating an array of size |*S*|, with the limitation that we can only store a maximum of *m* values.

Important to the functionality of a hash table is its load factor, commonly notated as *α*, which is derived by n/m, where *n* is the number of keys stored in the table. This value gives us an idea of how many keys on average we should expect to be cast to any given bucket when storing *n* keys.

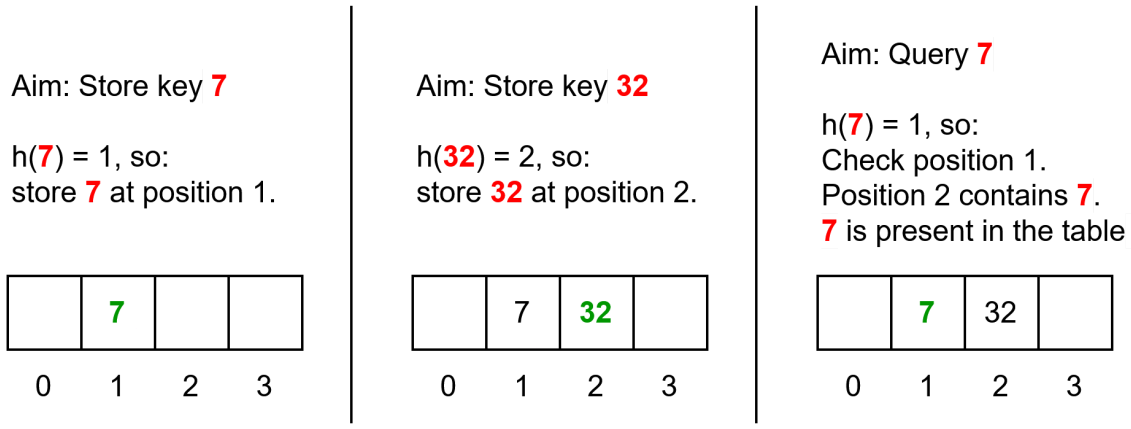

Figure 1: Illustration of basic storage and retrieval in a hash table

1.2 Collisions

The obvious limitation of the above data structure is the inability to store multiple values with the same hash in the table. For a small-scale example, let us have a hash table of size 11, in which we want to be able to store keys in set S={0,1,…,99}. The hash we use could be h(k) = k % 11, where k is the key. If we try to store 34 and 45 in this table, they will both be hashed to bucket 1, as seen in Figure 2. The phenomenon where we have multiple values we need to store at one address is called a collision.

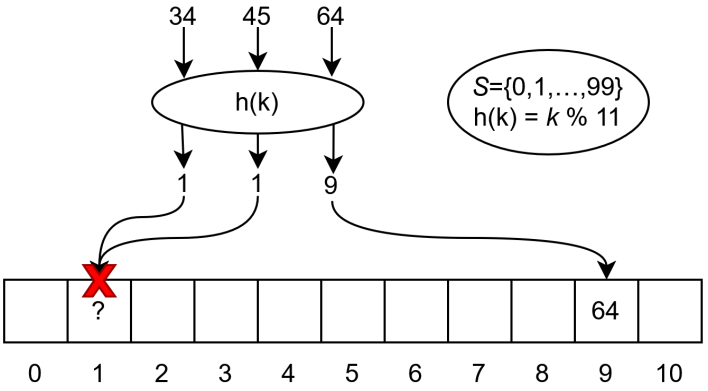

Figure 2: Example of collision when trying to store keys 34, 45 and 64 in a hash table

1.3 Common collision resolution methods

Described below are the two most common methods for handling collisions in hash tables.

1.3.1 Separate chaining

Separate chaining is the most common and simple collision resolution method. It appears in common hash table implementations such as C++'s std::unordered_set and Java's HashSet.

Separate chaining maintains a separate structure (typically a linked list) for every bucket. When a value is queried, its value is hashed to find the bucket, then the bucket's list is iterated through to determine if the value is present. Similarly, to insert a key, it is appended to the bucket's list, as shown in Figure 3.

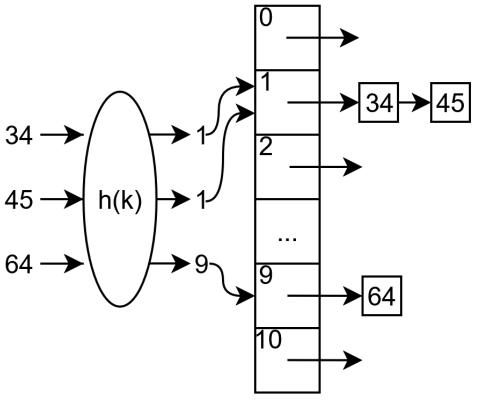

Figure 3: Illustration of separate-chaining method

Technically, this method's retrieval/insertion/deletion worst case time complexity is O(n), as if all keys in the table have the same hash, they will all be appended to one list, resulting in the same search and insertion time complexity as a linked list. In this case, insertion is also O(n) as we still need to traverse the list to confirm the key isn't already in there.

However, Cormen et al. show the average time complexity for these functions is θ(1), provided the table size m is proportional to n (i.e, m = Θ(n)). This is because the expected time complexity is Θ(1+α), as on average we will need to iterate through α elements on a given operation. So by keeping m = Θ(n), we keep α = O(1), thus keeping Θ(1+α) = Θ(1).

1.3.2 Open addressing with probing

Open addressing is an alternative to separate chaining which doesn't require any secondary data structures. Tables using this collision resolution method store keys normally as described in Section 1.1 until a collision occurs, at which point "probing" occurs. Probing methods include linear and quadratic probing, of which only the former will be described, for the sake of simplicity.

Linear probing is simply the act of travelling down the list of buckets until an empty bucket is found. For example, if a key k with a hash of 1 is inserted, but bucket 1's address is already filled, the program will check to see if bucket 2 is empty. If so, it will place k there, and if not, it will continue to bucket 3, and so on.

Similarly, when a value is queried, the program will look at h(k), and if k is not there, it will descend the list until it finds k or an empty bucket. In the latter case, the empty space tells the program that k is not in the table. Insertion and retrieval operations are illustrated in Figure 4.

When a value is erased, a special marker often called a "tombstone" takes its place, which tells the prober that there's no value but it should keep searching.

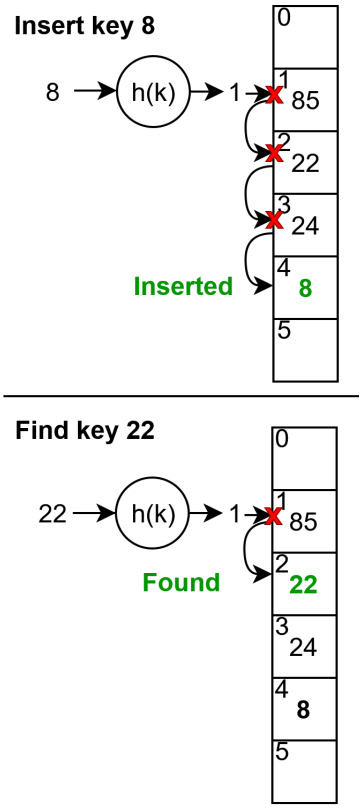

Figure 4: Illustration of insertion and retrieval with linear probing

Like separate chaining, this method has a worst case time complexity for all operations in the order O(n), as it can effectively devolve to an array search if every key has the same hash. However, Cormen et al. show that the expected number of probes needed to find or insert a key is at most 1/(1 − α), meaning if the load factor is below 0.7, for example, we can expect to make ~3.33 probes on average. Therefore, assuming a load factor less than about 0.7 and a uniform hash, insertion/retrieval have an average time complexity of Θ(1).

Open addressing takes less memory than separate chaining due to the lack of the separate data structures, but performance degrades rapidly once the load factor exceeds about 0.7. Meanwhile, separate chaining performance only degrades linearly and the load factor can exceed 1 without failure. Also, many deletions on an open-adressed table can lead to lots of spaces taken by tombstones, increasing load factor. This means if keys are being erased frequently, rehashes may be needed more often than they would in separate chaining.

2 Cuckoo hashing and universal hashing

In this section, the cuckoo hashing scheme will be justified as a solution for the problem of hash table collisions. The mechanism of the cuckoo hashing scheme will be outlined, and the application of universal hashing in cuckoo hash tables will be detailed. Cuckoo

2.1 Cuckoo hashing

Cuckoo hashing is another method that aims to utilise the benefits of open address hashing, whilst attempting to avoid its drawbacks such as performance degeneration to O(n) lookups, and frequent element collision. Introduced by Pagh and Rodler (2004), Cuckoo hashing differs from other hashing methods by introducing a second hash function which maps its output to a position in a second table. This gives an inserted key two possible buckets to occupy. Cuckoo hashing is inspired from the behaviour of some species of newly hatched Cuckoo chicks, who infamously are known to push other eggs or young out of the nest. It approaches hashing in a way that analogously imitates this behaviour through its eviction process.

In a traditional Cuckoo hash setup, two evenly-sized tables are maintained each with its own respective hash function to map elements. When a key is inserted into the structure, it is indexed into one of two candidate positions by its hash function. If the hashed key value maps to a vacant bucket in the table, the value is inserted and will occupy the position. If instead it maps to an occupied position in the table, then it will evict and replace the value that occupies the bucket from the table. The evicted value will then proceed to be mapped to the second table provided it also has a vacant position. If not, the original evicted value will evict the occupied value from the second table and take its position.

This process will repeatedly alternate between the two tables and continue until either a vacant spot is found for the evicted value, or when a predefined amount of evictions is reached. If the predefined limit is reached, the structure will then trigger a rehash. Similarly to most methods of rehashing, a cuckoo hashtable will resize both arrays to a larger value, and reinsert its elements accordingly.

Cuckoo hashing prides itself on providing constant time lookups as it primarily relies on an array data structure. Table lookups will only need to check at most two positions per key, which reduces the risk of clustering and long probe sequences which are common in traditional open addressing schemes. Because each key only has two possible positions and the table grows dynamically when necessary, cuckoo hashing guarantees O(1) expected lookup, insertion, and deletion times under typical conditions. As Cuckoo hash tables will need to track the position of n variables spread across two evenly sized arrays, it has a space complexity of O(n).

Despite its advantages, Cuckoo hashing does have some drawbacks to consider before implementation. First, if poorly selected hash functions will almost certainly trigger multiple eviction chains, which can degrade insertion performance. Second, excessive evictions can cause frequent rehashing, a costly procedure with O(n) time complexity. Finally, to maintain its desired O(1) lookup performance, Cuckoo tables also take in a maximum capacity of roughly 50% with collision likelihood significantly increasing above 50% (Pagh, R., & Rodler, F. F., 2004)

2.2 Universal hashing

A disadvantage of hash tables, especially ones that use common and fast hash functions such as modulo hashes, is the problem of key sets that produce an uneven distribution of hash results. For a simple example, if we are using h(k) = k % 13, and the data set is *S*={0,13, 26, 39…} or *S*={0,4, 17, 30…}, all members of set S will produce the same hash, which is catastrophic for hash table performance. In practice, real-world hash functions use additional mixing steps to reduce such simple patterns, but it remains true that deterministic hashes can still produce pathological collisions for certain key sets.

Even if all members of the set don't follow an adversarial pattern, there is a possibility of patterns in user inputted sets only being hashed to utilize a fraction of the total buckets. Skewed distributions of keys can also happen to map unevenly to the bucket list given an unfortunate hash. Finally, an adversary can intentionally degrade hash table performance if they know the hash function, by inputting many values that all hash to the same bucket.

Carter and Wegman (1979) introduce the concept of a "universal" class (or family) of hash functions to ameliorate this issue. They define a universal hash family H as a set of hashes h where for $x \neq y \in U$:

$$Pr[h(x) = h(y)] \leq \frac{1}{m}$$

In plain english, the universal family is a set for which, when we randomly take a hash from the set, and apply it to any unequal x and y, the probability of there being a collision is less than or equal to $\frac{1}{m}$, where m is the number of possible hash values.

As applied to the hash table, the idea is that instead of using one set hash function for every hash table we make, we randomly select a hash function from a universal set each time we create or re-hash the table. This way, the likelihood of the hash being adversarial to the key set used is low, and even if it is adversarial, the table can be rehashed with another hash from the universal family if we see performance degrading. The result is, as Carter and Wegman put it, "the average performance of the program on any input will be comparable to the performance of a single function constructed with knowledge of the input".

Carter and Wegman also provided the following construction as an example of a universal hash family. Where p is a prime that is $\geq |U|$, a is a randomly chosen integer from 1 to $p - 1$ and b is a randomly chosen integer from 0 to $p - 1$, a hash $h \in H$ can be defined as:

$$h_{a,b}(k) = ((ak + b) \bmod p) \bmod m$$

In theory, implementing this constructor to choose hash functions in a cuckoo hash table does not increase time complexity over that of the base cuckoo hash, because the time taken to randomly pick a and b is constant, and so is the time to actually perform the hash's operations. Given a reasonably sized universe, the prime $p \geq |U|$ can usually be hardcoded, avoiding the need to spend computing time on finding an appropriately sized prime number.

3 Complexity of the provided solutions

Provided with this report are a deterministic and non-deterministic implementation of the cuckoo hash table. The deterministic implementation uses one fixed hash for each bucket array, while the non-deterministic solution picks its hashes from a universal family.

3.1 Deterministic cuckoo hash

3.1.1 Insertion

The insertion method in the provided cuckoo hashing solution operates at O(1) amortized complexity. In regular insertion method calls, operations will typically occur in O(1) insertion due to array insertions being implemented in this manner due to their random access nature. However, due to the nature of cuckoo hash tables having the potential for multiple collisions to occur and the requirement of a rehashing process, the performance per insertion can sometimes be degraded to O(n). Although individual insertions can occasionally be expensive due to evictions and rehashing, the cost is amortized over multiple insertions, ensuring that the average runtime per insertion remains close to O(1).

3.1.2 Contains / Find

The contains or find method in the provided solution only requires two constant time operations to be performed. This is because, for a given key, the method only needs to compute two hash functions and check the corresponding positions in the underlying arrays. Hashing allows for a direct map for the key into the table which bypasses the requirement to iteratively search through the array for the vacant spot. Furthermore, arrays store its elements contiguously in memory which allows for random access, therefore, accessing any position is a constant-time operation at the hardware level due to predictable memory addresses and CPU cache efficiency. By combining hashing with array storage, the contains method can determine whether an element exists in the table with just two simple lookups, making it an extremely efficient operation.

3.1.3 Erase

The erase method in the provided cuckoo hashing solution runs at a complexity of worst-case O(1). This efficiency was achieved as the underlying data structure that was used were arrays, combined with the standard library's optional type to represent stored elements. As mentioned in the contains and find method, an array provides a O(1) lookup due to the way they are stored contiguously in memory. While traditional deletions in arrays require O(n) operations to shift all elements, this implementation's utilisation of std::optional allowed for the eradication of this process as deleted components only needed to be marked empty to indicate bucket vacancy.

3.1.4 Rehash

The rehash solution operates at a worst-case complexity of O(n), where n is the number of elements in the table. The method only triggers when the table reaches either a predefined load factor, or a predefined total number of collisions. By using a predefined array of prime sizes, which roughly doubles with each step, the implementation avoids calculating the next

table size dynamically. Although insertion is a O(1) operation, the bottleneck of this method comes from the repeated reinsertion of n elements into an array leading to its O(n) complexity, with this cost being amortized over multiple insertions.

3.1.5 Hashing

The provided solutions' hashing method was designed to run in O(1) time complexity as it computes a numeric index using simple arithmetic operations such as multiplication, addition, and modulo. On modern CPUs, these operations are performed extremely fast, with multiplication and addition instructions being performed in constant-time effortlessly. While modulo operations require slightly more overhead, they are still efficiently handled in modern CPUs with near constant-time performance. With the arithmetic operations being kept simple, this hashing method achieves a good balance between uniform key distribution and high performance, allowing elements to be inserted, searched, or deleted with minimal computational cost.

3.1.6 Space Complexity

The cuckoo hash table requires O(n) space to store n elements, with minor additional overhead from std::optional wrappers and temporary arrays during rehashing.

3.2 Universal hash cuckoo hashing

The provided non-deterministic variant of the cuckoo tree draws its hashes from a universal family whose hashes are constructed by Carter and Wegmans' (1979) construction as seen in Section 2.2. This solution just overrides the constructor, the hashes and the rehash functions of the base cuckoo hash, meaning the insertion, find and erase functions have the same time complexity.

The effectiveness of rehashing has been formally motivated through the famous balls-and-bins problem as discussed by Flemming (2018). In this setting, the largest number of keys mapped to any single bucket is expected to be $O(\log(n) / \log \log (n))$, ensuring that no bucket becomes overly congested. This rationale provided the motivation to implement universal hashing in cuckoo hashtables as it maintains low collision probability and supports the O(1) expected insertion and lookup times.

The rehash implementation in this solution simply randomly re-picks the hashes and then runs the rehash as in the deterministic solution. As explained in Section 2.2, the time complexity to pick the hashes is constant, so the non-deterministic rehash operation has the same time complexity as the deterministic one.

Similarly, the time complexity to run the hash on the keys is also constant as explained in Section 2.2. Meanwhile, only a constant amount of extra memory is required for this implementation.

So overall, implementing universal hashing on the cuckoo hash table had no incurrence on time or storage complexity.

4 How tests were used to assure algorithmic correctness

4.1 Deterministic Cuckoo hashing

To ensure the correctness and reliability of the deterministic Cuckoo Hash implementation, a comprehensive suite of automated tests was developed using Google Test (GTest). These tests target key operations of the hash table, including insertion, deletion, load factor management, hash function behavior, and collision handling whilst ensuring that the hash table maintains all its invariants.

4.1.1 Hash function validation

A singular test was used to confirm whether the hash function would distribute keys effectively. Insertions were made with a set of sequential keys from 0 - 999 to simulate a dense data input, as they often represent a worst-case scenario for naive hash functions. Testing with these values ensured that the hashes could handle structured input without clustering. Distribution was accounted for by tracking the number of keys that were assigned to each bucket for both hash functions and comparing the buck load against the average expected load calculated by $total\ keys/count\ size$. This approach ensures that no single bucket becomes overly populated with keys.

4.1.2 Insertion Tests

A wide suite of insertion tests was created to ensure that all aspects of insertion were thoroughly validated, providing a sound grounding for algorithmic correctness. Basic insertion behaviours such as single-element, negative number, and zero insertions were utilised to confirm that the data structure was able to handle varying input types.

Tests were also created to ensure that the invariants of a cuckoo hash table were upheld, these included duplicate insertions, which verified that the table maintained uniqueness of keys, and collision handling scenarios, which confirmed that the displacement mechanism correctly resolved conflicts without losing or overwriting existing elements. Furthermore, rehashing tests after insertion for both load factor thresholds and maximum evictions ensured that correct load factor and eviction invariants were upheld.

To ensure complete algorithmic functionality, insertion outcomes were compared with the standard library's unordered_set implementation, as it confirmed that the behaviour of the solution mimicked a rigorously tested standard. Furthermore, hand-verified bucket placements were calculated using the given hash functions and were implemented to ensure that both hash function and displacement logic operated as intended. Large-scale randomised stress tests were employed to ensure that invariants were upheld through a combination of cases and confirmed that the implementation remained effective.

4.1.3 Erase Tests

Similarly to insertion testing, erase tests were created to validate the accuracy of the data structure in removing elements and ensuring that the core invariant's of a cuckoo hashing were not violated. Basic erasure scenarios mimicked insertion with single-element removal and attempts to delete non-existent keys. These tests verified that the table correctly updated its size and empty status with no errors.

Tests were also for multiple-element deletions to verify that displaced elements and collisions did not obstruct deletion logic. Similarly deletions verified their structure when they were contrasted with the standard library's unordered set implementation, ensuring that erase operation logic behaved consistently. Furthermore, large-scale stress tests of randomly generated numbers confirmed that the structure performed technically sound whilst under high load.

4.1.4 Basic Functionality tests

A set of functionality tests were implemented to ensure that the core requirements of cuckoo hashing behaved consistently and provided a grounded foundation of correctly behaving structures before evaluating more complex scenarios. Basic functionality such as valid size increment and decrement functionality were compared with unordered_set to ensure it replicated modelled behaviour, hand-written load factor calculations before and after rehashing confirmed that the table maintained its invariants after an insertion and determined that breaking invariants would trigger a response to have them maintained. Finally both hash functions were tested with large integer samples to confirm that they both produced valid bucket indices so that no array indexing errors would occur during insertion or lookup scenarios.

4.2 Universal hash

4.2.1 Random cuckoo table functionality test

The basic functionality tests were run with the universally hashed cuckoo table to ensure that the non-deterministic implementation functionally worlds the same as the same as the deterministic one.

4.2.2 Universal hash function properties

Recall from Section 2.2 the condition for a universal family:

$$Pr[h(x) = h(y)] \leq \frac{1}{m}$$

The universal hash function properties tests ensure that the universal hash family is correctly implemented by testing if it adheres to the above condition. This is done in two different ways:

- **Testing the average pairwise collision rate for one hash in the universal family:** For any hash in the family, expect any two random x and y will collide on one in every m runs. To test this, pick a single hash h from H, then calculate empirical probability of collision over a million runs with random x and y values. If it is within a reasonable range $1/m$, we can mark the test as passed.
- **Test the universality of the family:** Select a few fixed pairs x and y, then test over many h ∈ H (a million runs), that the empirical probability that each pair collides for any random hash h is close to $1/m$.

4.2.3 Comparing performance of deterministic and probabilistic implementations

Two tests were run to test the comparative behaviour of the deterministic and probabilistic cuckoo hash tables. The first test takes the hash from the deterministic implementation and one from the probabilistic one to demonstrate that for a truly random dataset, there's no real advantage of using a universal hash family over a fixed hash function. This is seen when feeding the same keys into both hashes, as they on average produce the same collision rate.

The second test creates a deterministic and a probabilistic hash table, then inserts a set of 25 keys with an interval of 1109 (the size of each array) into each. The test demonstrates that the deterministic table rehashes 12 times, while the universally hashed table only has to rehash once to resize, showing universal hashes are not vulnerable to patterns like deterministic hashes are.

5 Insights

When designing the final solution, certain design choices were required to be made and justified, as they would have a direct impact on both performance and reliability. This section is dedicated to the insights and reasoning behind these design decisions, explaining why specific parameters were chosen and how they influence the behavior of the cuckoo hash table

5.1 Table Sizes

When selecting the table size of the hash table, there are two canonical approaches that can be justified. Firstly you can increase the sizes starting at 2^0 and further increasing each rehash by the next power of 2. Alternatively you can choose prime numbers for the table size and increase each rehash to the next prime roughly double the current size. Both of these approaches provide benefits and drawbacks to their implementation.

By increasing the sizes by powers of 2, you receive the benefit of having consistently growing, uncapped resize of the arrays as well as having better compatibility with bitmask hashing which is computationally less expensive than modulo hashing. However, the draw back is that with size increase, if the array sizes are not occupied as much as possible by keys, and frequent rehashes occur due to max collisions or low load factor, the growth of the table size is $O(2^n)$ and can be potentially very wasteful.

Contrastingly, choosing prime numbers for table sizes provides better key distribution, particularly when paired with modulo-based hash functions. It reduces the likelihood of patterns in key-values that cause repeated collisions, therefore, improving insertion success rate. However, the computational cost of modulo functions is worse than bitmask hashing as well as dynamically finding the next prime number. This can be minimized by having a static array of primes to increment to during rehash but this loses the ability to dynamically increase the capacity to a certain limit.

5.2 Bitmask Hashing vs. Modulo Hashing

Similarly with Table sizes, two canonical approaches are regarded as valid when determining the hashing functions of a hashset. Bitmasking provides a cheap, quick, and efficient way of mapping keys to bucket indices when the table size is a power of 2. By using bitwise AND operations with a mask of size - 1, the computation avoids the more expensive division involved in modulo operations, allowing for constant-time index calculation with minimal CPU overhead.

The drawback with bitmask hashing is that it does not provide the same algorithmic guarantee as modulo based hashing in randomised universal hash families of reducing collisions. Furthermore, bitmasking hashing only provides a valid solution when paired with a table size that is a power of 2 and only has a limited application domain, therefore, giving the design option of sacrificing space complexity for slightly better hashing speed.

Contrastingly, modulo hashing is a more versatile solution that is algorithmically more rigorous and soundly tested to work in a wide domain of applications. By modulating the key with the table capacity, it provides a guaranteed indexing to any chosen table size. This works best with prime-numbered table sizes, as they help reduce clustering and the likelihood of patterns in the keys causing repeated collisions.

While modulo hashing is slightly more computationally expensive than bitmasking due to the division operation, it offers better uniformity of key distribution and is compatible with both power-of-2 and non-power-of-2 table sizes. Furthermore, it provides a guarantee of reducing collisions by 1/m after rehash for universal hashing families. This makes modulo hashing a safer and more general-purpose choice, especially when the table size cannot be constrained to powers of 2 or when using universal hash functions to minimize collisions.

6 References

Carter, J. L. & Wegman, M. N. (1979). Universal Classes of Hash Functions. Journal of Computer and System Sciences, 18(2), 143-154. doi:10.1016/0022-0000(79)90044-8.

Chen, C. (n.d.). *An overview of Cuckoo hashing* [Lecture notes]. Stanford University. Retrieved November 9, 2025, from https://cs.stanford.edu/~rishig/courses/ref/l13a.pdf

Cormen, T., Leiserson, C., Rivest, R., & Stein, C. (2009). Introduction to Algorithms (Third). Mit Press. https://www.cs.mcgill.ca/~akroit/math/compsci/Cormen%20Introduction%20to%20Algorithms.pdf

Fleming, N. (2018, May 17). *Cuckoo hashing and Cuckoo filters* [Technical report]. University of Toronto. https://www.cs.toronto.edu/~noahfleming/CuckooHashing.pdf

Pagh, R., & Rodler, F. F. (2004). Cuckoo hashing. *Journal of Algorithms, 51*(2), 122-144. https://doi.org/10.1016/j.jalgor.2003.12.002