

An Analysis of Different Types of Hashing

By Nicole Brink and Aditya Mittal

Introduction

Hash tables are a fundamental data structure in computer science designed to achieve constant time, $O(1)$, time complexity for insertion, deletion, and search operations. At the core of hashing is a hash function, which maps keys to specific indices in an array. However, because the keys are not known beforehand, and the number of possible keys exceeds the array size, multiple keys can map to the same index. This is known as a collision. These collisions need to be resolved in some way, as two keys cannot occupy the same place in memory.

This project outlines four such collision resolution strategies, namely separate chaining, linear probing, quadratic probing, and double hashing. For each of these strategies, their theoretical performance for various operations is discussed.

1. Separate Chaining

Instead of storing a value at the memory address, it stores a linked list (or a vector). When a collision occurs, that value is simply appended to the list at that index.

a) Theoretical performance:

$$\text{Load factor } (\lambda) = N/M$$

Where N is the number of keys, and M is the table size.

The load factor (λ) refers to the average number of records there are per linked list. Ideally, this value should be close to 1 for separate chaining to obtain $O(1)$ time complexity for inserts, searches, and deletes.

b) Successful Searches: $1 + \lambda/2$

On average, one goes through half of the list before finding the correct key.

c) Unsuccessful Searches: λ

One needs to go through the entire list to confirm the key does not exist.

2. Linear Probing

When a collision occurs, the algorithm checks the next available slot sequentially until it finds a location to store the key.

The formula for this hash function is $hi(x) = (h(x) + i) \% M$
where i is the index, and M is the table size.

- a) Successful Search: $\frac{1}{2} (1 + 1/(1 - \lambda))$
- b) Unsuccessful Search: $\frac{1}{2} (1 + 1/((1 - \lambda)^2))$

However, one big disadvantage of this method is that it is susceptible to data clustering. More specifically, continuous blocks of occupied slots develop, which can drastically increase search times.

3. Quadratic Probing

To reduce clustering, this method searches for slots further away from the collision using a quadratic polynomial ($1^2, 2^2, 3^2 \dots$).

The formula for this hash function is $hi(x) = (h(x) + i^2) \% M$
where i is the index, and M is the table size.

Quadratic probing does not have a successful and an unsuccessful search complexity like the chaining and linear probing methods. It should, however, in theory, be better than that of the linear probe. This said, quadratic probing may also run into problems with data clustering, as keys hashing to the same initial position follow the same probe path.

4. Double Hashing

This method uses a second, independent hash function to determine the step size upon collision.

The formula for the hash function is $hi(x) = (h(x) + i (h2(x))) \% M$
Where $h(x)$ is the first hash function, $h2(x)$ is the second hash function, i is the index, and M is the table size.

For our experiment, we used $h2(x) = R - (x \% R)$

Where R is some prime number smaller than the table size, M . It is crucial that the second hash function never returns 0, and it should be relatively prime to the table size.

Implementation

The code for this project was decided to be made in C++ due to its low-level manipulation capabilities and support for the Standard Template Library. Firstly, a basic hash table was made. A table size was decided to be a large prime number, and a structure was made for integer key-value pairs. We wrote the hash function soon after, along with functions for inserting, searching, and deleting pairs in the table. The code initially only focused on linear probing, but support for quadratic probing, double hashing, and separate chaining was soon added. The functions also keep track of relevant metrics like the number of collisions and attempts. An enum was designed to conveniently switch between the different algorithm types, along with a function to actually change to the desired algorithm. We designed the tests to perform the load factor and search tests. Functions for generating data randomly, running the tests, and saving the results to a CSV file were designed. Finally, a main method was written to run the actual program.

The insert function for a table using open addressing inserts a key-value pair by computing the index using a hash function. If the key already exists, replace the value. If another key exists at the index, resolve the collision using the selected algorithm. Finally, insert a pair if an empty index is found, or return false if the number of attempts exceeds the table's size. The insert function for separate chaining is similar, except it only requires the pair to be pushed to the end of the vector at the computed index (if the key is unique). The search functions are similar to the respective insert functions, but instead of inserting a pair, they take a key as an argument and return the pair if found. The hashing and collision resolution strategies are the same.

Evaluation

To test the real-world performance of the four hashing strategies against their theoretical performance, two tests were implemented:

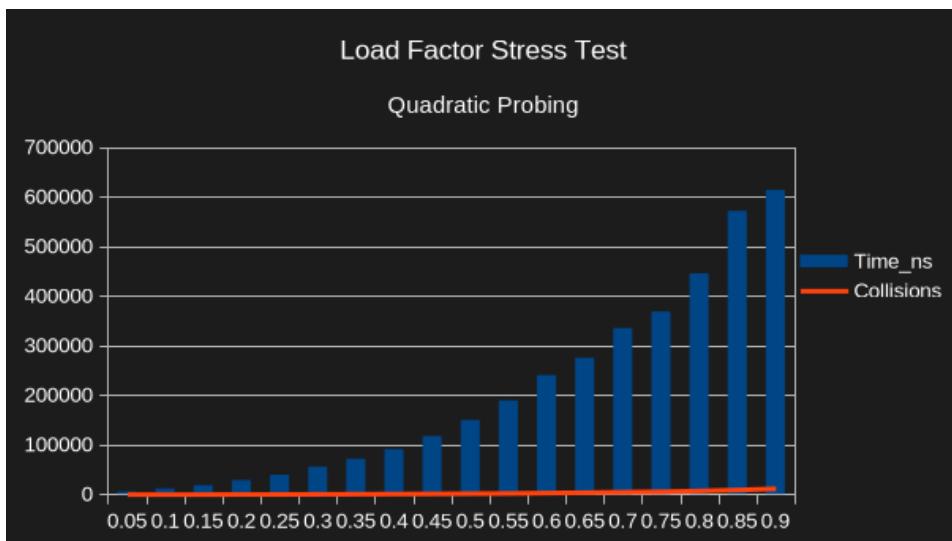
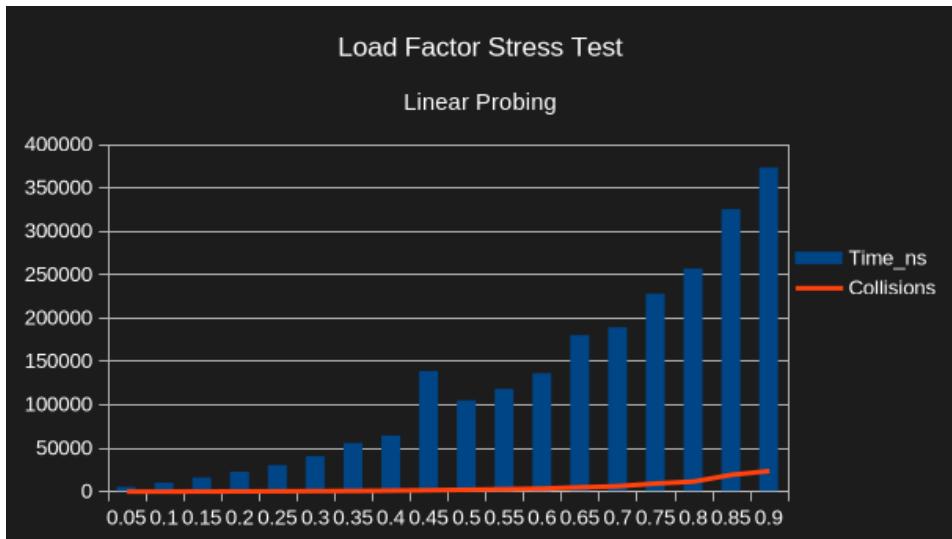
Experiment A: Load Factor Stress Test

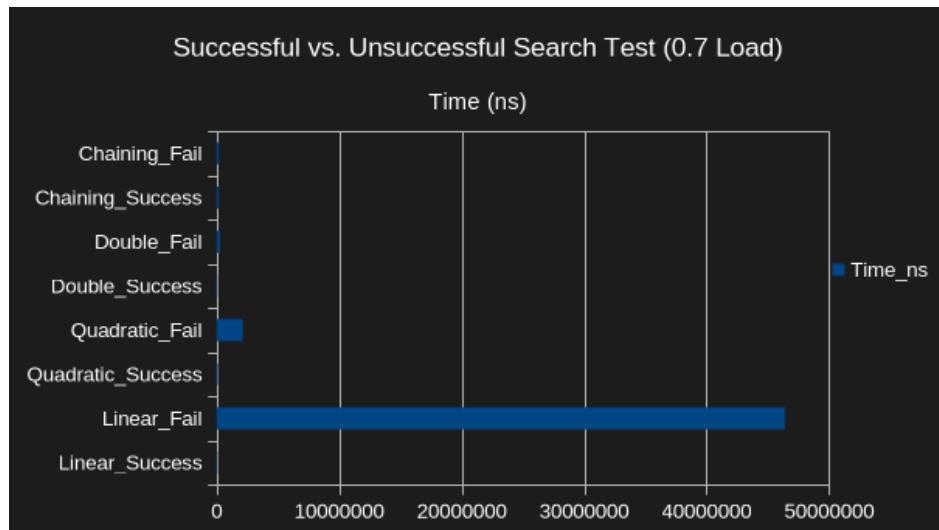
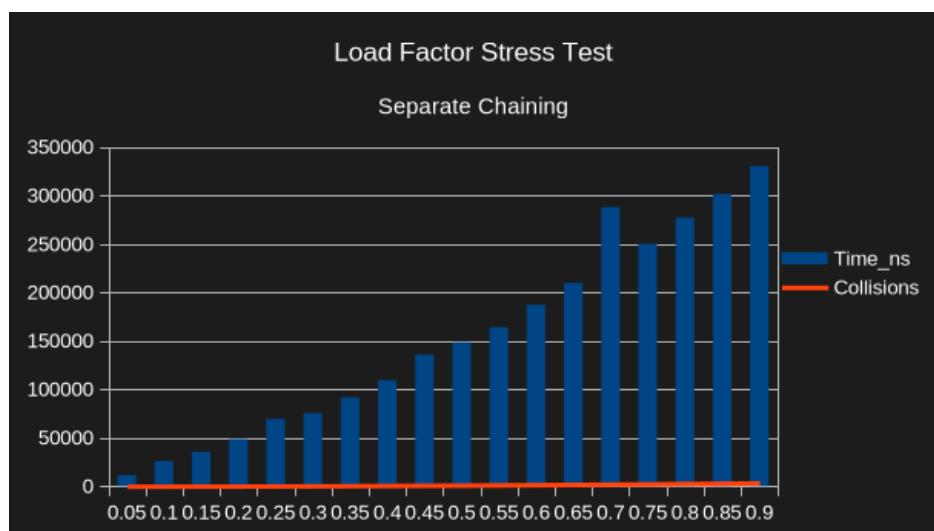
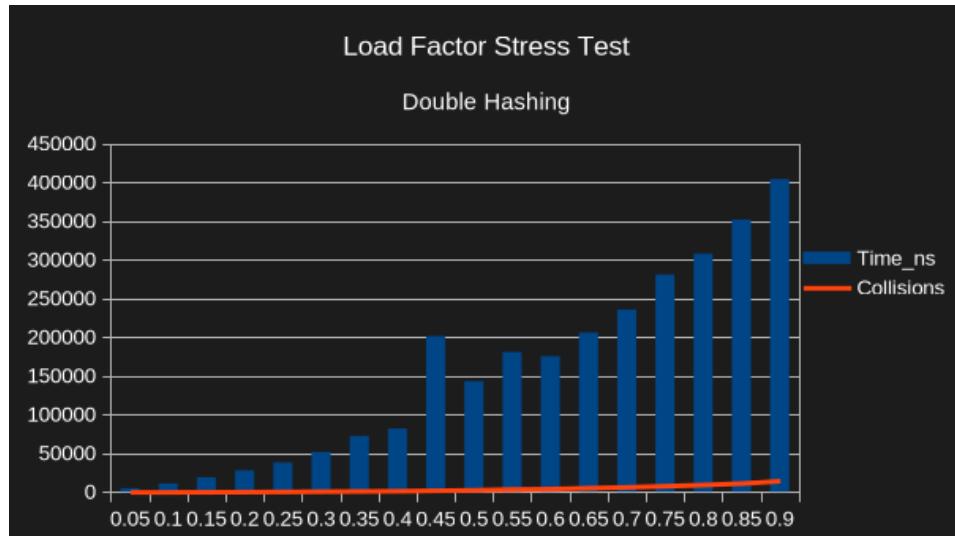
- Methodology: The table size was fixed at 10,007. We inserted random integers into the table to reach load factors ranging from 0.05 (5%) to 0.95 (95%) in 5% increments.
- Metric: We measured the time required to search for every key currently in the table (Successful Search).
- Goal: To observe how performance degrades as the table fills up, particularly for Linear Probing.

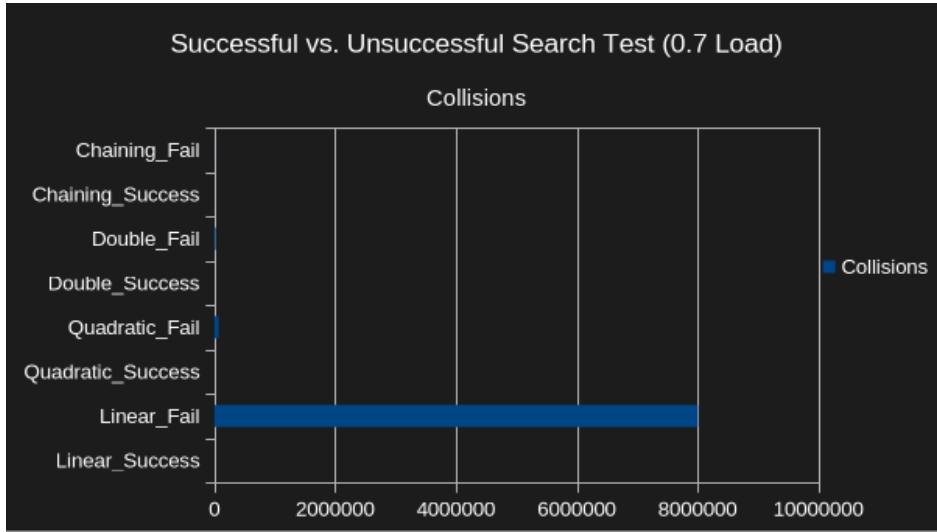
Experiment B: Successful vs. Unsuccessful Search

- Methodology: The table was filled to a fixed load factor of 0.70 (70%).
- Data Distribution: To test strict success/failure scenarios, we generated keys using a patterned approach:
 - o Inserted Keys: Multiples of 2 (0, 2, 4, 6...).
 - o Search Keys (Failure): Odd numbers (1, 3, 5, 7...), which were guaranteed not to be in the table.
- Goal: To compare the cost of confirming an item exists vs. confirming it does not.

The results were as follows:







Analysis

Load Factor Stress Test:

When examining the differences between theoretical predictions and observed results, our Load Factor Stress Test matched the Big-O expectation. Theoretically, Open Addressing methods like Linear Probing should degrade as the load factor (λ) approaches 1. Our results confirmed this: Linear Probing performed well up to $\lambda = 0.75$, but at $\lambda = 0.95$, the collision count went up exponentially. In contrast, separate chaining seems to take more time than open addressing algorithms for lower load factors, but performs equally well (or even better) for large load factors. This method maintained near-constant time performance even at a 95% load, which aligns with the theoretical prediction that its performance degrades linearly rather than exponentially.

Successful vs. Unsuccessful Search Test:

Data distribution played a massive role in the strange behaviours we observed in the Search Test. A key observation is that all the hashing algorithms have exactly 0 collisions during a successful search. This is because in this test, only pairs with even-numbered keys are being inserted into the hash table. First, all the even slots fill up, after which the remaining even numbers wrap around and start filling the odd slots. All input keys involved in the successful search test are even, and those in the unsuccessful search are odd. Hence, all search algorithms manage to return the key (if

present) on their first try with no collision. As a result, we accidentally created a perfect hash function without realising it, which is why every successful test shows 0 collisions.

We found a significant data clustering effect where Linear Probing failed catastrophically during the unsuccessful search test, recording over 8 million collisions compared to Quadratic Probing's 70,000. This was caused by the patterned data (multiples of 2) creating long contiguous blocks in the table, forcing the Linear Probing method to traverse thousands of slots to confirm a key was missing.

It is also important to note that hardware limitations and overhead related to how the code was implemented could have influenced the wall clock time. Even at lower load times (such as $\lambda < 0.5$), the linear probing outperformed the separate chaining, even when their collision counts are similar. This is likely due to cache locality - linear probing accesses contiguous memory addresses (index, then index +1 ...). This can allow the CPU to pre-fetch the data it needs from cache. In contrast, separate chaining relies more on following pointers to scattered locations (node \rightarrow next), which can cause cache misses, thereby increasing its wall clock time. However, as observed in our high-load tests, the overhead associated with repeatedly resolving collisions eventually outweighs these advantages.

Conclusion

This project showed that there is a distinct trade-off between Open Addressing and Separate Chaining. Our experimental results confirmed that while Linear Probing worked well at lower load factors due to lower cache locality, it is a lot more unstable at high load factors, and it is quite sensitive to data distribution. This was most evident in our search test, where the data clustering caused over 8 million collisions in the Linear Probing test to determine that keys were missing. In contrast, Quadratic Probing and Double Hashing handled the same scenario with significantly less overhead.

Ultimately, Separate Chaining was found to be the most consistent and robust strategy. It handled load factors up to 95% without the exponential degradation seen with the Open Addressing techniques, which makes it a good, safe choice as a hash function.

Future research could involve redoing the Search test to have a more realistic data distribution instead of creating a perfect hash function. In addition to this, other hashing strategies can be tested, such as Cuckoo or Extendible Hashing.

Appendix - Results

Strategy	Load_Factor	Time_ns	Collisions
Linear	0.05	4629	12
Linear	0.1	9488	54
Linear	0.15	15460	123
Linear	0.2	21982	213
Linear	0.25	29776	358
Linear	0.3	39976	577
Linear	0.35	55195	745
Linear	0.4	63882	1109
Linear	0.45	138204	1625
Linear	0.5	104379	2132
Linear	0.55	117574	2795
Linear	0.6	135629	3543
Linear	0.65	179553	4948
Linear	0.7	188540	6286
Linear	0.75	227424	9363
Linear	0.8	256510	11596
Linear	0.85	324840	19420
Linear	0.9	372992	23771

Strategy	Load_Factor	Time_ns	Collisions
Quadratic	0.05	4759	12
Quadratic	0.1	10900	55
Quadratic	0.15	17623	116
Quadratic	0.2	27723	219
Quadratic	0.25	38393	341
Quadratic	0.3	54925	559
Quadratic	0.35	70945	769
Quadratic	0.4	90483	1039
Quadratic	0.45	116653	1412
Quadratic	0.5	148925	1858
Quadratic	0.55	188600	2348
Quadratic	0.6	239768	3145
Quadratic	0.65	274624	3902
Quadratic	0.7	334739	4804
Quadratic	0.75	368132	5748
Quadratic	0.8	445270	7268
Quadratic	0.85	571390	9460
Quadratic	0.9	613681	11782

Strategy	Load_Factor	Time_ns	Collisions
Double	0.05	4619	13
Double	0.1	10681	64
Double	0.15	18806	171
Double	0.2	27673	349
Double	0.25	37842	574
Double	0.3	51338	887
Double	0.35	72058	1245
Double	0.4	81626	1606
Double	0.45	201104	2189
Double	0.5	142702	2754
Double	0.55	180615	3723
Double	0.6	175084	4205
Double	0.65	205753	5484
Double	0.7	235419	6605
Double	0.75	280616	8059
Double	0.8	307648	9524
Double	0.85	351701	11470
Double	0.9	404062	14509

Strategy	Load_Factor	Time_ns	Collisions
Chaining	0.05	11271	13
Chaining	0.1	25709	38
Chaining	0.15	35267	111
Chaining	0.2	48432	177
Chaining	0.25	69332	297
Chaining	0.3	75244	390
Chaining	0.35	91354	499
Chaining	0.4	109148	723
Chaining	0.45	135569	905
Chaining	0.5	148273	1075
Chaining	0.55	163943	1322
Chaining	0.6	187037	1593
Chaining	0.65	209460	1824
Chaining	0.7	288010	2056
Chaining	0.75	249927	2388
Chaining	0.8	277118	2776
Chaining	0.85	301565	3097
Chaining	0.9	330321	3405

Strategy	Time_ns	Collisions
Linear_Success	63021	0
Linear_Fail	46402566	8006001
Quadratic_Success	91745	0
Quadratic_Fail	2134213	71725
Double_Success	54975	0
Double_Fail	266609	23038
Chaining_Success	172198	0
Chaining_Fail	189823	4001