

University of the Cumberland

Assignment 7: Exploring Hash Tables and Their Practical Applications

Student: Suresh Ghimire

UC ID: 005038288

MSCS-532-M20– Algorithms and Data Structures

July 25, 2025

Hash tables are useful data structures that are heavily utilized by software applications for their fast data storage and retrieval functionality. This paper will review some of the key features that help define hash table performance: the hash function, and methods to resolve collisions. We will walk through the certain implications of the hash function and how to remedy collisions, looking specifically at implications for efficiency, memory use, and real world use cases.

1. Hashing function and their impact

Hash functions are critical to the performance of hash tables when utilized correctly. A good hash function should guarantee a uniform distribution of keys across the hash table to maximize performance by minimizing collisions from excess mapping of keys to the same position. Some general characteristics of a good hash function include the following:

- Uniform key distribution
- Minimized collision occurrence
- Computational efficiency
- Deterministic (same input = same output)

Alternatively, if a hash function is poorly constructed, it will likely lead to excessive clustering of keys or collisions, compromising the theoretical performance of the table. For example, as a simple but poor demonstration of a hash function, consider a table that maps, as a hash function, only the ASCII value of the first character of a given key. If many string keys start with the same letter, collisions will be excessive.

To provide a measurable demonstration of demonstrated distributions and collisions with hash functions, we created three separate hash functions in Python and measured the performance of each generated string key when hashed into a 5000 element table. We first generated a list of 10,000 random strings for the experiments:

- Bad Hash :9,938 collisions, 62 unique slots used, down to the simplicity of the logic to only use only the first character.
- Good Hash : 5,669 collisions, with more than 4310 unique slots (used more than 4,300 slots).
- Very Good Hash (SHA-256): Similar performance to built-in hash function but combines cryptographic guarantees at slight cost to computation time.

Following image shows the difference on those 3 different hashing computing metrics

```
PROBLEMS  PORTS  OUTPUT  TERMINAL  DEBUG CONSOLE  Code
PS C:\Users\sures\Desktop\University of the Cumberland\MSCS-532-DSA\MSCS532_Assignment7> python -u "c:\Users\sures\Desktop\Un
he Cumberland\MSCS-532-DSA\MSCS532_Assignment7\hashing.py"
Hash Function  Collisions  Unique Slots Used  Time Taken (s)
0 Bad Hash (First Char)  9938  62  0.001003
1 Good Hash (Built-in)  5669  4331  0.003980
2 Very Good Hash (SHA-256)  5690  4310  0.017664
PS C:\Users\sures\Desktop\University of the Cumberland\MSCS-532-DSA\MSCS532_Assignment7>
```

Python code to calculate the above metric is available on provided github repository:

Balancing Speed and Complexity

There is potential contention between using a complex hash function that is efficient, and a faster but more simplistic function. Although cryptographic hash functions such as SHA-256 provide remarkable uniformity and minimal collision occurrence, they have an overhead for computation. Simpler built-in functions offer a balance between good distribution and speed that still affords higher-performance (e.g., in-memory caches, real-time, etc.) uses.

Observation

Based on the SHA-256, and built-in functions investigated in our Python experiment, the SHA-256 function did slightly better with distribution 4310 unique slots vs. 4331, but took approximately 4x longer to perform. This neatly illustrates almost precisely the typical tradeoffs better performance, and theoretical precision.

2. Open addressing vs separate chaining

There are two types of collision handling strategies for a hash table: open addressing and separation chaining. In open addressing, all elements are stored in an array that is the hash table. If a collision should occur, then the algorithm uses the next available slot (for this case, we will use linear probing). Meanwhile, with separation chaining, there are linked lists kept at each index to cover all entries that share the same hash.

In terms of efficiency and memory efficiencies, open addressing is the most space-efficient because you aren't counting against memory for pointers and linked lists. However, the probing sequences can grow quickly and waste significant time at high load factors. Separation chaining isn't as space efficient, due to the memory usage, however you can look at larger load factors better with more savings.

Real world scenarios

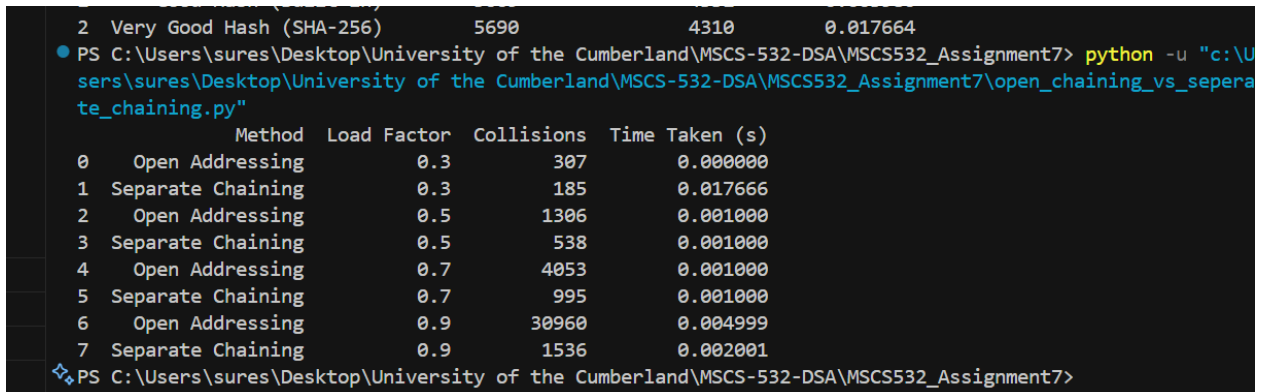
- Open addressing is a really good option for applications where either have low load factors, or are constrained with memory (think DNS caching or small embedded systems).
- Separation chaining is a good option in cases where the hash table will possibly get larger load factors or possibly experience collisions (think Java's HashMap or database indexing). It can handle increases in size and only suffers constant overheads.

Performance in the real world

In order to showcase real world performance, we implemented the two strategies in Python and executed tests against various load factors (0.3, 0.5, 0.7, and 0.9) against a hash table with a length of 5000 and randomly generated string keys. Below are the results of the tests:

- At 0.3 load factor, both types performed quite well (separation chaining returned fewer collisions, 185 vs. 307),
- With a 0.5 load factor, open addressing had more than doubled the collisions (1306 vs. 538).
- With a 0.7 load factor, open addressing had drastically more collisions (4053 vs. 995).
- -At a 0.9 load factor, open addressing had an extremely high number of collisions (30960), while separation chaining stayed consistent (1536).

Follow is the screenshot of the logs from performance testing on open addressing vs separate chaining



```

2 Very Good Hash (SHA-256)          5690          4310          0.017664
● PS C:\Users\sures\Desktop\University of the Cumberland\MSCS-532-DSA\MSCS532_Assignment7> python -u "c:\U
sers\sures\Desktop\University of the Cumberland\MSCS-532-DSA\MSCS532_Assignment7\open_chaining_vs_separa
te_chaining.py"
      Method  Load Factor  Collisions  Time Taken (s)
0   Open Addressing      0.3         307      0.000000
1  Separate Chaining      0.3         185      0.017666
2   Open Addressing      0.5        1306      0.001000
3  Separate Chaining      0.5         538      0.001000
4   Open Addressing      0.7        4053      0.001000
5  Separate Chaining      0.7          995      0.001000
6   Open Addressing      0.9       30960      0.004999
7  Separate Chaining      0.9         1536      0.002001
❖ PS C:\Users\sures\Desktop\University of the Cumberland\MSCS-532-DSA\MSCS532_Assignment7>

```

Full code is available on the provided github repository

This illustrates the tradeoffs really well: open addressing fell off sharply at high load factors while separation chaining handled as you would expect with a negligible extra memory cost.

Open addressing obviously provides a great approach in the problem case of light loads and tight memory budgets, while separation chaining tends to be a consistently better or scalable approach in real-world systems or those with random or dense distributions.

Conclusion

This project emphasized that design of hash function and collision resolution is a significant aspect when optimizing performance with hash tables. Poorly designed hash function creates unnecessary number of collisions and performance degradation, whereas a good hash function supports more collisions (or no collisions at all) and is able to look up an item faster.

In comparing the collision resolution strategies, open addressing performed very well at low load but began to struggle as the table began to fill out while separate chaining held stable even at very high load factors. When designing systems to be efficient, reliable, and scalable, a good combination of hash function and collision resolution strategy is crucial.

References:

Cormen, T. H., Leiserson, C. E., Rivest, R. L., & Stein, C. (2022). Introduction to Algorithms (4th ed.). Random House Publishing Services. <https://reader2.yuzu.com/books/9780262367509>

GeeksforGeeks. (2023, June 7). Open addressing collision handling technique in hashing. <https://www.geeksforgeeks.org/dsa/open-addressing-collision-handling-technique-in-hashing/>

GeeksforGeeks. (2023, June 7). Separate chaining collision handling technique in hashing. <https://www.geeksforgeeks.org/dsa/separate-chaining-collision-handling-technique-in-hashing/>