Hashing with Chaining

Analysis of Expected Search, Insert, and Delete Times under Simple Uniform Hashing

In the context of hash tables, simple uniform hashing assumes that each key is equally likely to be hashed to any of the available slots (buckets), leading to an even distribution of keys across the hash table. For a hash table with n elements and m slots (buckets), the load factor alpha α is defined as the ratio of the number of elements to the number of slots (Cormen et al., 2009). When using chaining for collision resolution, each bucket contains a linked list (or chain) of elements that map to the same hash value.

Expected Time

- **Insert Operation**
    - In the best case, when there are no collisions, the insert operation takes constant time O(1) (Goodrich et al., 2011).
    - On average, under the assumption of uniform hashing, the expected time to insert an element into the hash table is O(1), since the key is simply appended to the appropriate chain (Cormen et al., 2009).
- **Search Operation**
    - The expected search time is proportional to the length of the chain, which is O(1+α). If alpha is small search time will be close to O(1). However, if alpha is large, search time becomes O(α), as longer chains need to be traversed.
- **Delete Operation**:
    - Like search, the expected time to delete a key depends on the length of the chain in the relevant bucket, with the time complexity being O(1+α).

Impact of Load Factor on Performance

The load factor α significantly affects the performance of the hash table. As alpha increases, the chains become longer, slowing down search and delete operations. Insert operations remain efficient even with high α since new elements are simply appended to chains.

When alpha is less than one, most buckets contain only one element, making operations O(1) time. Chains with greater length (alpha > 1) result in O(α) search and delete times.

Strategies for Maintaining a Low Load Factor and Minimizing Collisions

To maintain efficient hash table operations, it is important to control the load factor and minimize collisions. Here are some strategies:

1. Dynamic Resizing (Rehashing)

- Dynamic resizing involves increasing the size of the hash table when the load factor exceeds a certain threshold (e.g., 0.75). When rehashing, all keys are redistributed into a larger table, ensuring shorter chains and faster operations (Cormen et al., 2009).

2. Use of a Universal Hash Function

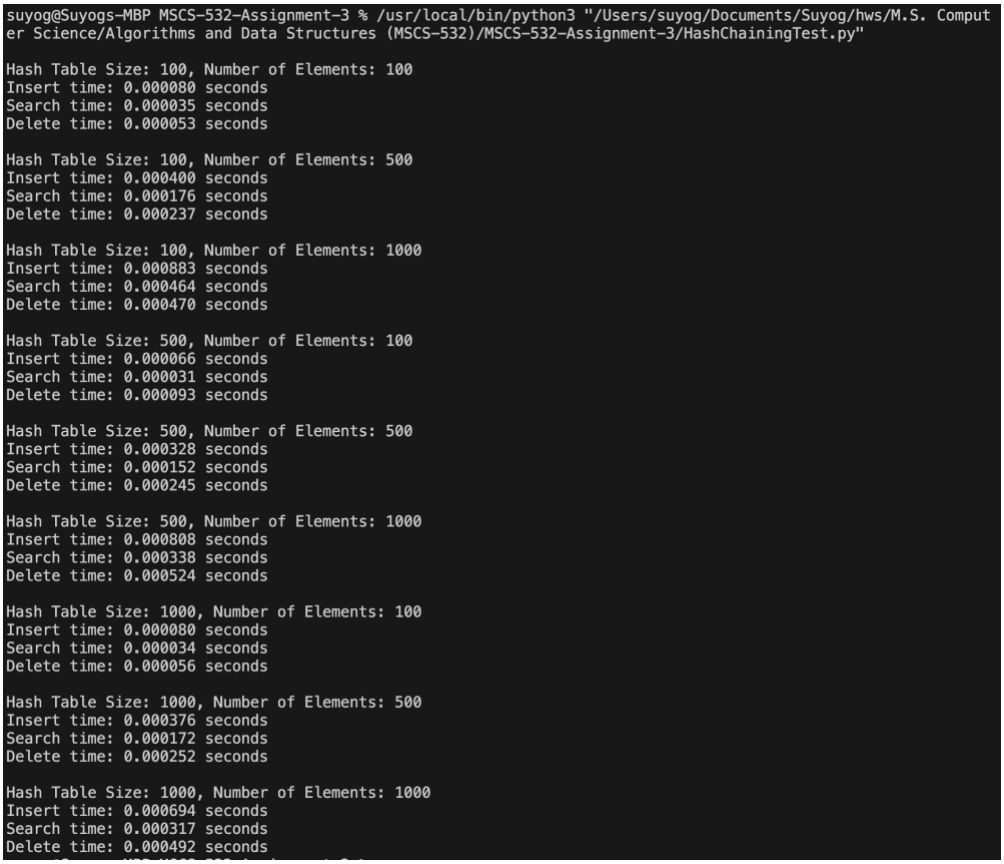
- A universal hash function helps distribute keys uniformly across buckets, minimizing collisions which ensures uniform distribution and reduces clustering.

3. Load Factor Threshold

- A load factor threshold (between 0.7 and 0.9) help maintaining performance. When the number of elements exceeds this threshold, the hash table is resized to reduce the load factor (Goodrich et al., 2011).

Performance Testing

To test the execution time for insert, search, and delete timeit python library was used. Below is the screenshot of the test carried out.



Findings of the Test

Insert Operations: As expected, the insert time grows as the number of elements increases, but the increase is minimal, indicating that the hash function distributes the keys efficiently, minimizing collisions. For example:

- Inserting 1000 elements into a table of size 100 took 0.000883seconds.
- Inserting the same number into a larger table (size 1000) reduced the time to 0.000694 seconds due to a reduced load factor and fewer collisions.

Search Operation: The search time also scales well. As the number of elements increases, the search times remain relatively low, and the increase is small. This shows that the chains in the hash table are generally short, likely due to a good distribution of keys. For example:

- Searching 1000 elements in a table of size 100 took 0.000464 seconds.

- Searching the same number in a table of size 1000 reduced the time to 0.000317 seconds.

Delete Operation: The delete time is similar to search, as both involve finding the key first. The delete times grow in line with search times, which confirms that the hash table efficiently handles collisions even as the number of elements grows. For example:

- Deleting 1000 elements from a table of size 100 took 0.00047 seconds
- Deleting from a larger table of size 1000 reduced the time to 0.000492 seconds

Effect of Load Factor

With 1000 elements in a table of size 100, the insert time is 0.000883 seconds, but in a table of size 1000, the insert time is reduced to 0.000694 seconds. This demonstrates the effect of a lower load factor on reducing collision chances and improving efficiency.

Conclusion

- The hash table performs very efficiently in terms of insert, search, and delete operations, even when the number of elements increases.
- The load factor significantly affects performance, with lower load factors (larger table sizes) leading to better performance.
- The time complexity for the operations remains close to O(1) as expected from a well-implemented hash table with chaining for collision resolution.

References

Cormen, T. H., Leiserson, C. E., Rivest, R. L., & Stein, C. (2009). *Introduction to algorithms* (3rd ed.). MIT Press.

Goodrich, M. T., Tamassia, R., & Goldwasser, M. H. (2011). *Data structures and algorithms in Java* (6th ed.). Wiley.