**Section 1, Task A:**

**Hash Tables**

A hash table is a type of data structure that employs key-value pairs. The key is a handle that is unique to each element (like a primary key in a database). A method to Add/Insert always requires the (key,value) pair to function. Get(key), Update(key,value), and Remove/Delete(key) are other methods that are required to have a working hash table. A Hash function accepts a key as input and tells where to look. All hashing methods use a hash code/hash function denoted by h. If k is a random key, then there will be no collisions, and h(k) is the address of a position in the hash table. An instance of a collision is when there is more than one record of h(k). This cannot happen and as a result a collision will occur [1].

**Collision Resolution**

There are two ways to handle a collision, chaining and open addressing.

**Chaining:** When an element in the table already contains that key, the second element is saved in the linked list which is found within that key. To retrieve the element, the linked list is traversed until the key is found.

**Open Addressing:** On the contrary to chaining, when there is a collision, with open addressing the item is moved to a different location in the table. To search an element, the table needs to be searched to find the key.

**Section 1, Task B:**

The load factor of a hash table is defined as the ratio of the number of elements stored in the table to the number of slots in the table. It is an important factor that affects the performance of a hash table, as it affects the number of collisions that occur and the amount of memory that is required.

The fundamental concept of open addressing in hashing is to utilize the empty spots in the hash table to indicate the end of a search sequence, instead of utilizing memory for links in lists. [3]. In open-addressing collision resolution strategies, the load factor plays a crucial role in determining the performance of the table. There are several open-addressing strategies, but two of the most commonly used are linear probing and quadratic probing. Linear probing works by searching for the next empty slot in the table by incrementing the index of the current slot by one. This method is efficient when the load factor is low, but as the load factor increases, the number of collisions also increases, and the table becomes increasingly cluttered. This results in longer search times and wasted memory.

On the other hand, quadratic probing works by incrementing the index of the current slot by a quadratic function of the number of collisions. This method is more efficient than linear probing when the load factor is high, as it is less likely to result in clustering of elements. However, it requires more memory than linear probing, as it needs to store the quadratic function.

The load factor ˛ for a hash table T with m slots that stores n elements is defined as n/m, which is the average number of elements stored in a chain. The analysis will be based on this factor, which can be less than, equal to, or greater than 1. However, the worst-case performance of using a hash table with chaining is poor, as all n keys could potentially map to the same slot, resulting in a list of length n. This means the worst-case time for searching would be O(n) plus the time to compute the hash function, which is no better than using a single linked list for all elements. Therefore, hash tables are not used for their worst-case performance [2].

In general, open-addressing collision resolution strategies are more memory-efficient than chaining, but they are less efficient when the load factor is high. On the other hand, chaining is more efficient when the load factor is high, but it requires more memory.

In terms of speed, open addressing collision resolution strategies are faster than chaining when the load factor is low, but they become slower as the load factor increases. In contrast, chaining is slower than open addressing when the load factor is low, but it becomes faster as the load factor increases.

In conclusion, the load factor is an important factor that affects the performance of a hash table. The choice of collision resolution strategy depends on the expected load factor and the desired balance between memory efficiency and speed.

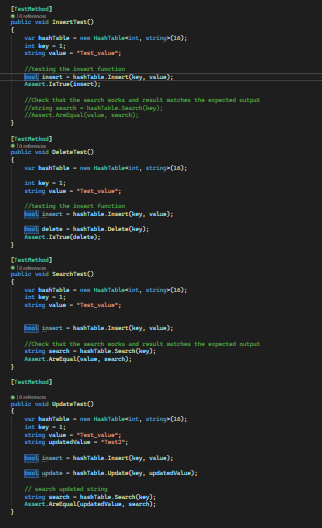
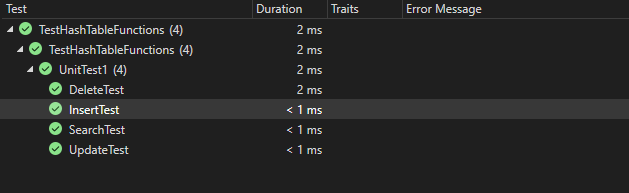
**Bibliography**

[1] W. D. Maurer and T. G. Lewis, “Hash table methods,” *ACM Computing Surveys*, vol. 7, no. 1, pp. 5–19, 1975.

[2] T. H. Cormen, C. E. Leiserson, R. L. Rivest, and C. Stein, Introduction to Algorithms, 2nd ed. The MIT Press, 2001.

[3] R. Sedgewick and K. Wayne, Algorithms, 4th Edition. Addison-Wesley, 2011, p. I–XII, 1-955.

Section 6: Code Screenshot



**Section 7:**

The ABA problem is a problem that can occur in concurrent systems when using compare-and-swap (CAS) operations, such as the Interlocked.CompareExchange method, which was used for section 5 in my implementation. The problem occurs when multiple threads are trying to update a shared resource, and one thread reads a value, a second thread updates the value, and a third thread updates the value back to its original value. The first thread, which still has the original value, then performs a CAS operation using the original value, believing that the value has not been modified, when in fact it has been modified and then restored to its original value.

This can occur when two threads are trying to update the same key-value pair in the hashtable simultaneously, and the second thread updates the value and the third thread updates the value back to its original value. The first thread, which still has the original value, then performs a CAS operation using the original value, believing that the value has not been modified, when in fact it has been modified and then restored to its original value.

There are several solutions that can be applied to solve this problem. One solution is to used a lock to synchronize access to the shared resource, as the lock will ensure that only one thread as a time can access the shared resource, therefore eliminating the possibility of the ABA problem. In my code, one can see that I did a similar implementation on the Delete operation, is everything was within the scope of lock \_lock. Another solution is to use a version number or a timestamp to accompany each value, and have the CAS operation check both the value and the version number or timestamp before making the update. Another solution is to use a double-wide CAS operation, which compares both the address and the value at that address before making the update. This can be done using classes such as the Interlocked.CompareExchange<T> method in C#. Obviously, there are many other options that one could obtain for, these were just a few ideas.