1. How good is the provided `stringHash` function -- are we really getting constant time operations with our hashmap? What is the big O of the insert and remove operations with this hash function for our stock market use case?

   The original stringHash function is quite simple and fast, as it only depends on the length of the key. However, it's not very effective in distributing keys evenly across the buckets, especially for our stock market use case where company abbreviations may have similar lengths. This could lead to many collisions and degrade the performance of the hashmap. The time complexity of insert and remove operations with this hash function is O(1) in the best case, but it could degrade to O(n) in the worst case if many keys hash to the same bucket.

2. Write 1-2 paragraphs describing the findings from your research about simple hashmap hash functions, which new hash function you decided to use or create, and why you think it is a good choice for our use case. Make sure you talk about the big O in the worst case for your hash function, and that you describe how the hash function works in C. Be sure to describe all the syntax features that are being used in your hash function.

   The djb2 algorithm, which I used in my yourHash function, is a much better choice for our use case. It's a well-known string hashing algorithm that uses bit shifting and addition to create a hash value, which tends to produce a more even distribution of hash values for a wider range of keys. This reduces the likelihood of collisions and improves the performance of the hashmap. The time complexity of insert and remove operations with this hash function is still O(1) in the average case, but even in the worst case, it's likely to be much better than the original stringHash function.

   The yourHash function works by initializing a hash value to 5381, then for each character in the key, it shifts the hash value 5 bits to the left (which is equivalent to multiplying by 32), adds the original hash value (effectively multiplying the hash by 33). Bit shifts is on many CPUs a faster way to perform this operation. and then adds the ASCII value of the character. The final hash value is the modulo of the number of buckets, which ensures that the hash value fits within the range of valid bucket indices. The function uses a while loop to iterate over the characters in the key, and the *myKey++ syntax is a compact way of accessing the current character and then incrementing the pointer.

3. In addition, write 1 paragraph describing why you think the unit test that you've written shows a good worst case for your hash function.

   The unit test I have written tests the worst-case scenario for the hash function, where most keys hash to the same value. This is a good worst case to test because it forces the hashmap to handle the maximum possible number of collisions, which can reveal any issues with the collision resolution strategy.

4. What is "open addressing" in regards to hash maps and hash tables?

   Open addressing is a method of collision resolution in hash maps and hash tables. When a collision occurs, instead of storing multiple items in the same bucket (as in chaining), open addressing finds another bucket to store the collided item. This is typically done using a process called probing, where the hashmap checks the next bucket(s) until it finds an empty one. Common probing methods include linear probing, quadratic probing, and double hashing.

Reference

Stanis, Filip. "DJB2 Hash." The Art in Code, https://theartincode.stanis.me/008-djb2/. Accessed 12 Apr. 2024.