Task 5

Cache using Hash tables

In computing, a cache is a hardware or software component that stores data so that future requests for that data can be served faster. The data stored in a cache might be the result of an earlier computation or a copy of data stored elsewhere. To be cost-effective and to enable efficient use of data, caches must be relatively small.

For implementing my cache, I have made use of hash table using linear probing. Reason for using this particular hash table is that every new word that would be cached would not take much time to find a NULL bucket for insertion. I made new header and cpp files with some slight modifications to the parameters of insert and search functions. The insert function initially was taking “string” as its argument but since we needed to hash the secrets which are of data type int, so I added another argument to the insert function. The new insert function now takes an int (for secrets) and a string (for corresponding word from dictionary) as its argument. The lookup function now takes an int as its argument instead of the string. The resize and the delete function were removed. Furthermore, a new variable of type int named counter was introduced into the structure of block in the header file. This would be used to keep count of the frequencies of the lookups of each code. The details as to how these functions work now would be explained shortly.

First comes the part where we were supposed to implement this task without using cache. For this purpose, I just did the file reading as we usually do. The secrets file that was provided to us was comma delimited hence I read both the secrets as well as the commas. After reading every secret, I simultaneously opened the dictionary and read the key-value pairs that were provided in them. After reading every key-value pair, I compared the key from dictionary to the secret from the secrets file and if a match was found I simply printed the corresponding word on the screen. This was done for all the secrets file as well as the sample codes provided in the manual which decoded to “THIS COURSE IS LOVE”.

For implementing the cache, the hash I used was bit Hash and the compression function was mad Compression. This choice was generally faster than any other combinations of hash and compression functions. I used the following approach to implement the cache. User is given an option to select any of the three given secret files to decode or to test the codes that we were provided in the manual that decoded to “THIS COURSE IS LOVE”. After selecting the desired secret file, the file is opened, and the codes are read one by one. After reading each code, a lookup operation in the hash table is performed to see if the corresponding word and the secret itself are cached in one of the buckets of the hash table or not. If the word looked up is cached

the word is returned and printed on screen. If the word is not yet cached, the dictionary file is opened, and the file is searched for the corresponding secret and if a match is found the word is displayed onto the screen as well as inserted into the cache.

The insert function first converts the integer secret code into a string and then computes its hash before inserting it into the cache after finding a NULL bucket. Lookup function works in the similar way and searches the hash table until the required secret code is found.

If the cache is full, I made use of least frequency used (LFU) method to store more items inside the cache. This basically works by searching all the entries and comparing the counter that is associated with them. The counter used is incremented every time by 1 whenever lookup on a certain block is performed. Hence the block which would have the least counter would be removed and the new block containing the new word and its secret key will be cached in its place.

For the comparison purposes, I read the first 10,000 secrets from each of the three files as well as from the sample code provided to us in the manual with and without cache implementation and results are as follows:

-	secret1.txt	secret2.txt	secret3.txt	Manual code
Without cache	158.608 seconds	294.243 seconds	465.183 seconds	0.050106 seconds
With cache	109.523 seconds	109.308 seconds	110.631 seconds	0.04776 seconds

The results clearly show that cache implementation made the program run and search for words faster than the one without cache implementation.