Practical Assignment

BM40A1500 Data Structures and Algorithms

1. Implementing the Hash Table

1.1 Structure of the hash table

Picture 1. The HashTable

```
class HashTable:
    def __init__(self, tableSize):
        self.size = tableSize
        self.table = [None]*tableSize
```

The HashTable is a list of size M, where each slot of the list is first initialised as None, and later in insertion made into a Linked list. The reason why the table is not made of Linked lists in the first place, is to minimise time and memory usage. If we do not access linkedlist a single time, we might've as well not spent the time and memory to create it.

1.2 Hash function

Picture 2. The Hash functionality

```
for i in range(len(key)):
    char = ord(key[i])
    sum += char**2
return sum % self.size
```

My hashing function calculates the sum of all squares of characters as ascii values, modulo size of HashTable. It was by far the simplest solution I came up with, and turned out to be one of the most effective I found (when testing with section 3 data set).

1.3 Methods

My hash table has the required methods (search, insert and delete) as well as method hash for hashing keys.

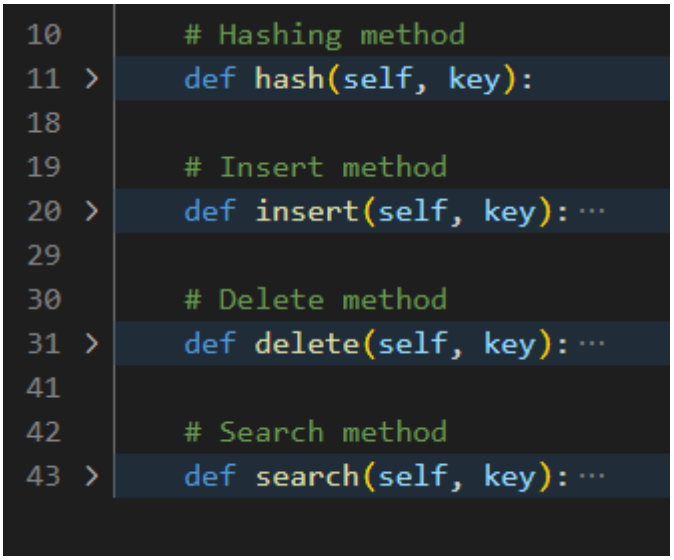
Insert takes a given key, calls the hash method to get an index for the hash table and checks if the slot currently has a Linked list. If there is no list, one is created. Then the key is inserted to the head of the linked list. Appending to the end of the list would take significantly more time when tested with section 3 data set.

Delete takes the key, calls the hash method for index in the hash table, and checks if the slot is empty. If there is no linked list, we have nothing to delete. If we find a list, we search the linked list for the given key. If we have a match, the Node with the matching key value is removed from the linked list. If the search is negative, we have nothing to delete.

Search takes a key, calls the hash method for list index, and checks if the table has a list on that spot. If one is found, it is searched for the given key value. If no list is found or it doesn't contain the key value, 0 is returned. On a match we return 1. This is basically boolean with numbers that is used for hash3_1.py.

Hash method takes a key value, performs the hash function described in 1.2 and returns the hash value.

Picture 3.Used methods



2. Testing and Analysing the Hash Table

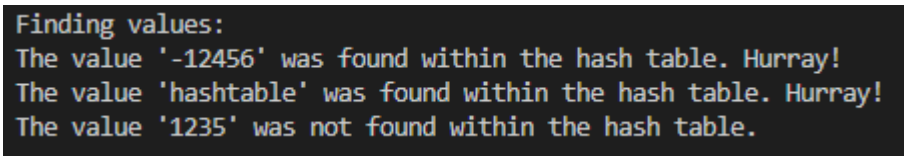
2.1 Running time analysis of the hash table

Picture 4. Adding values to the Hash Table

```
Adding values:

None
None
12

None
None
hashtable -> 12

1234
None
hashtable -> 12

1234
4328989
hashtable -> 12

1234
4328989
BM40A1500 -> hashtable -> 12

1234
-12456 -> 4328989
BM40A1500 -> hashtable -> 12

1234
-12456 -> 4328989
aaaabbbbcccc -> BM40A1500 -> hashtable -> 12
```

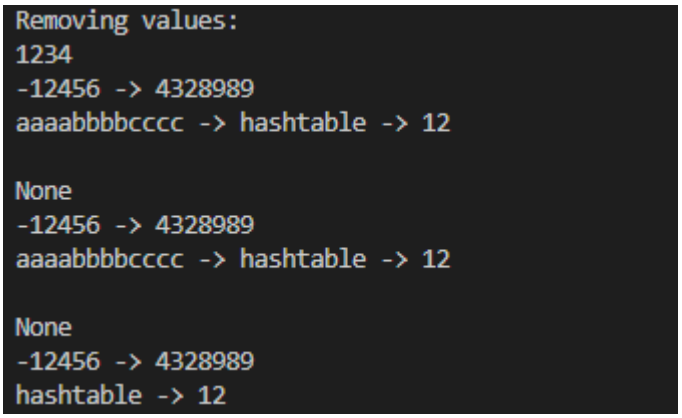
The running time of accessing a Linked list from the hash table is Θ (1), hashing n symbols long 'word' takes Θ(n) times, and inserting value to the start of the linked list takes Θ(1) time. Overall it takes Θ(n) time to insert a n-letter word into the hash table. Here we can notice that the most considerable time delay comes from the hashing function itself. This is because my implemented hash function looks at every symbol in every word. Changing this to ex. every other symbol would cut down the time it takes to insert values, but would cause more collisions and thus increase the time it takes to search them.

Picture 5. Finding values in the Hash Table



Hashing a word of n letters takes Θ(n) time, accessing a Linked list from a table is done in Θ(1) time and searching linked list of size m takes up to Θ(m) time. Therefore the running time of finding values from the Hash Table takes Θ(n*m) time when searching for n-letter word in a m-sized linked list.

Picture 6. Removing values from the Hash Table



Hashing a n-letter word takes Θ(n) time, accessing any given linked list from the hash table takes Θ(1) time and searching m sized linked list takes at most Θ(m) time. Therefore the running time of deleting n-letter value from m-sized linked list is Θ(n*m)

3. The Pressure Test

Table 1. Results of the pressure test.

Step	Time (s)
Initialising the hash table	0.0
Adding the words	2.03
Finding the common words	1.05

3.1 Comparison of the data structures

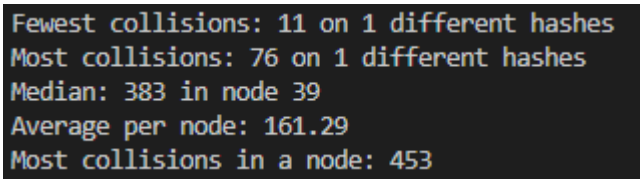
With my code, the clear winner is ArrayList in both adding words and searching them. Adding words to an Array takes Θ(n+m) time and searching them takes additional Θ(n+m) time, where n and m are the size of the lists used. The most potential time loss comes from sorting the lists, which takes up to Θ(n log(n) + m log(m)) time. The lists were not in random order so ordering lists took significantly less time than required. However even with maximum time complexity it is still many times faster than the HashTable structure I implemented.

It could be argued that my Array code doesn't match the given task, because I read both files to lists before searching for matching words. However I don't see a faster way of doing it while not reading the second file to a list without heavy optimization and a plethora of if-statements. The one problem with this approach is that if there isn't enough memory to allocate the whole of both files to lists then the functionality breaks. However the current data size is 1000s of times smaller than the mentioned case would require, and since the point was to find solutions with the given 'big' data set, it's reasonable to argue that memory isn't a notable factor in this assignment.

3.2 Further improvements

After running tests I can conclude that increasing the HashTable size would significantly reduce the running time of the algorithm. For example, on a size 50000 it took half as long to find the common words, and on 100 000 it took almost a third of the original time. Inserting words also went down with the size, but not as drastically. However it should be noted that without further optimising there's a cap on the amount of time we can save. Empty spaces will inevitably start to appear when increasing table size, even for the best hash functions.

Picture 7. Statistics from Hash Table pressure test.



With a table size of 10000 the data is in my eyes rather well distributed. This does not mean that the hashing couldn’t be improved further, but rather that the current one will suffice for the task given.

List of references

No outside material was used in this assignment