Practical Assignment

BM40A1500 Data Structures and Algorithms

1. Implementing the Hash Table

1.1 Structure of the hash table

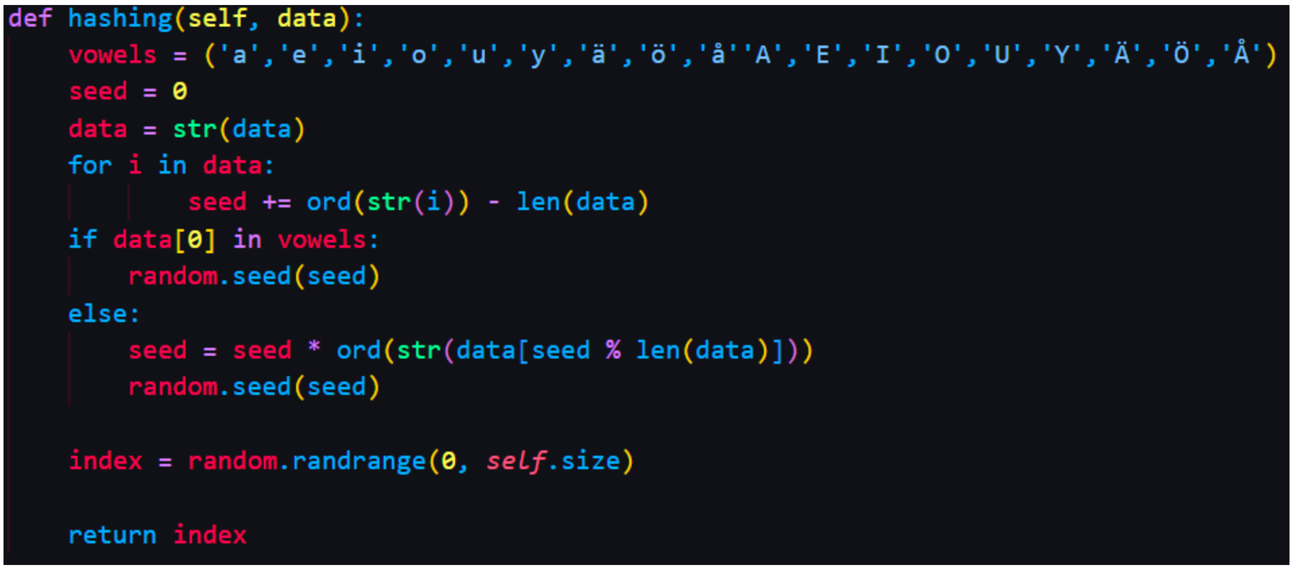
The hash table is made from class, which stores linked lists that hold the data.

1.2 Hash function

I created my own hash function because I wanted to see how fast I could make it.

The hash function takes advantage of Python's random library to generate the hash value. First, the method creates a seed by summing the ASCII values of each character of the string or integer. Then it checks if the first character of the data is a vowel. If it is, the seed is used as it is. Otherwise, the seed is multiplied by the ASCII value of one of the characters in the string. The character is decided by dividing the seed by the length of the data and using the remainder as the index.

Then the seed is used to generate a random number between 0 and the size of the hash table. This way the hashing always produces the same values with randomness, resulting in an efficient hash table.



```python
def hashing(self, data):
    vowels = ('a','e','i','o','u','y','ä','ö','å''A','E','I','O','U','Y','Ä','Ö','Å')
    seed = 0
    data = str(data)
    for i in data:
            seed += ord(str(i)) - len(data)
    if data[0] in vowels:
        random.seed(seed)
    else:
        seed = seed * ord(str(data[seed % len(data)]))
        random.seed(seed)

    index = random.randrange(0, self.size)

    return index
```

Picture 1. Described hash function in Python.

For example, let’s say we have a hash table with a size of four and the user wants to insert a string called “easy”. The seed’s value is calculated from the ASCII sum, which would be 101 + 97 + 115 + 121 = 434. Since the first character is a vowel, the seed is used in the random range function to generate a number between 0 and 3.

1.3 Methods

The hash table and linked list classes have five methods:

Init: is called when creating the table. It takes size as its parameter and then creates a list and fills it with empty linked lists.

Hashing: uses the described method to decide where data is stored and found.

Print: shows the structure of the hash table and what’s inside the linked lists by printing the index of the hash table and then the contents of the linked list inside it.

Search: tries to find the wanted element from the linked list by iterating through it until the element is found or it reaches the end of the list. It returns true if it is found, otherwise false.

Insert: is responsible for inserting typed strings or integers into one of the linked lists by iterating over it until an empty slot is found.

Delete: tries to find the wanted data with the hash function. If it’s the last element of the linked list, remove it. Otherwise, replace the current node with the next node.

2. Testing and Analyzing the Hash Table

2.1 Running time analysis of the hash table

Since search, insert and delete methods all use the same principle of looping through the linked list, they all have a time complexity of O(n), where n is the number of elements in the linked list. It’s O(n) because the most time-consuming part is a single while loop. The worst-case scenario is O(n) as well, but the best-case scenario is O(1) if the first element or slot of the linked list is the wanted one.

3. The Pressure Test

Step	**Time (s)**
Initializing the hash table	0.001997
Adding the words	8.308600
Finding the common words	3.587765

Table 1. Results of the pressure test for the hash table with a size of 5000.

Step	**Time (s)**
Initializing the hash table	0.002000
Adding the words	6.989999
Finding the common words	2.675999

Table 2. Results of the pressure test for the hash table with a size of 10000.

Step	**Time (s)**
Initializing the hash table	0.004000
Adding the words	6.730998
Finding the common words	2.357999

Table 3. Results of the pressure test for the hash table with a size of 20000.

Step	**Time (s)**
Initializing the hash table	0
Adding the words	0.047999
Finding the common words	717.6504

Table 4. Results of the pressure test for the list.

3.1 Comparison of the data structures

Although adding the words to the hash table with a size of 10000 took approximately 146 times longer than a simple list, the hash table found common words from the two text files about 268 times faster. Overall, the hash table was about 708 seconds faster than a normal list. This can be explained by the fact that the hash table didn’t have to iterate one list repeatedly. It had 10000 linked lists, and it always knew in which linked list the searched word should be because of the hash function.

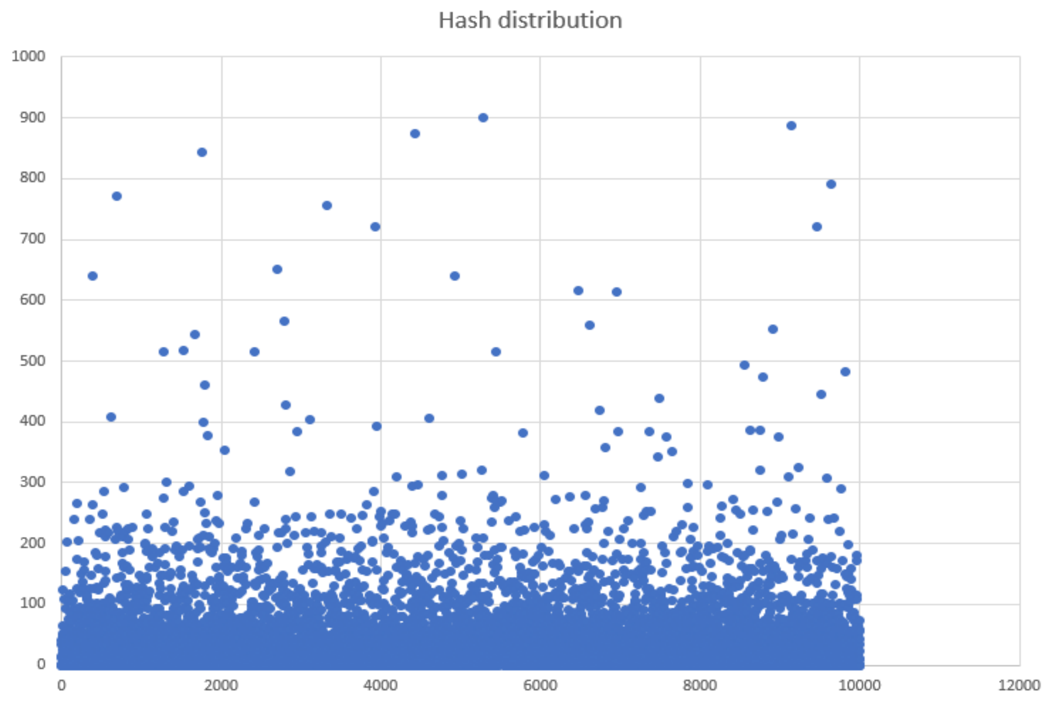
The reason why adding the words to the normal list was much faster, is because the program didn’t have to create the hash table or do any hashing in the first place. Inserting into a linked list is also slower since it must be looped until the last node is found.

3.2 Further improvements

By halving the size of the hash table, it took 23% longer to compute the task. But when the hash table size was doubled, the program was 6% faster.

As we can see from picture 2, the data has distributed quite close to each other excluding a few outliers. So, improvements could be made, but the distribution is still quite even.

I tried to tweak the hash function by changing the operators, but when the reading of the words got faster, comparing the words got slower. Or it became slower in both cases.



Picture 2. Distribution of the hashed data.