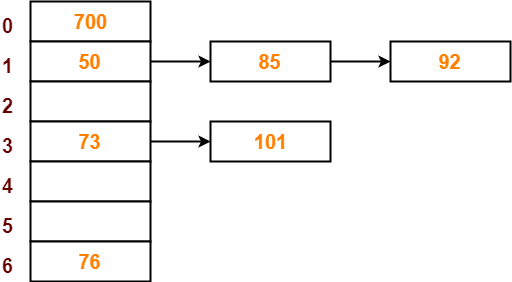
Practical Assignment

BM40A1500 Data Structures and Algorithms

### 1. Implementing the Hash Table

#### 1.1 Structure of the hash table

Present the structure of the hash table.

The hash table uses a linked list structure to achieve open hashing. Initializing the hash table with size n creates n amount of linked lists, where the hashed values will be stored. 

*Figure 1.1a – the structure of the hash table*

The length of each individual linked list is dependent on the overall size of the hash table. As the size of the hash table grows, the amount of keys in a linked list shrinks (assuming that the dataset stays the same).

#### 1.2 Hash function

What hashing function you used and why?

The hash function that is being used is string folding. That goes through the given key x bits at a time. At the start of the hash function (hasher), the given key is turned to a string, this makes it possible to handle strings and integers with the same hashing function. The hash is calculated by summing the value of processed bit values, which are “randomized” using a multiplier variable, the value of the multiplier variable changes while the function is running, to allow maximum spread of hash values. At the end the result is converted to the range 0…M-1(M is the size of the hash table), using the modulo operator.

I chose to use the string folding method, since it seemed to achieve the best spread with good runtimes and fairly easy implementation. With string folding, every linked list in the hash table is filled in a fairly balanced manner, assuming the hash table size is set well. For example with kaikkisanat.txt (93086 words) and hash table size 10 000, each list has around 1 to 10 words, which makes storing and look-up effective. The complexities for different methods are listed below.

|  |  |  |  |
| --- | --- | --- | --- |
| *Action* | *Best case complexity* | *Avg. case complexity* | *Worst case complexity* |
| Searching | O(1) | O(1) | O(n) |
| Insertion | O(1) | O(1) | O(n) |
| Deletion | O(1) | O(1) | O(n) |
| Space complexity | O(n) | O(n) | O(n) |

*Graph 1.2a – complexities for different actions*

Based on testing, the optimal number of bits seems to be 4 to achieve the quickest runtime with hash\_1. Runtime is the total amount taken by the program to initialize hash table of size 10 000, add words from kaikkisanat.txt, and search for a given key from the hash table.

|  |  |  |  |
| --- | --- | --- | --- |
| *# of bits* | *Min. runtime* | *Max. runtime* | *Avg. runtime* |
| 1 bit | 0.2879362106 | **0.3089737892** | 0.2975848198 |
| 2 bits | **0.1701450347** | 0.2212882041 | 0.1915523290 |
| 4 bits | 0.1867020130 | 0.1969845295 | **0.1900324821** |
| 6 bits | 0.1910369396 | 0.1980514526 | 0.1951833248 |

*Graph 1.2b – runtimes with different bit sizes*

#### 1.3 Methods

What methods (including the required) your hash table has? Explain briefly how do they work?

The hash table hash\_1.py includes 9 methods in total, where 4 are “core”-methods (hashed, adder, getter and delete), and other 5 are mainly used for troubleshooting and to understand the program better. Hasher is the method that takes cares of generating the hash values for the given key, input is the key to be hashed, and the output is the hash value in an integer format. Adder-method is used for adding the keys to the hash table, the function is called for every value to be added individually. The method implements a no-duplicate policy, where it checks whether the given value is already present in the hash table. Getter is one of the core-methods, and as the name implies, it takes care of getting a value from the hash table. Based on the given value to be searched for, it calculates its hash (using the hasher), and goes through the linked list that matches the hash value. If the key is found in the linked list, the function returns information regarding it, like its hash value and its location in the hash table/linked-list. The last core function is delete, which can be used to delete a key from the hash table. It works similar to getter, where it calculates the hash and locates the value in a linked list. If the value is found in the linked list, it is deleted using a default python del command.

The other 5 methods can be used to understand the code better. These are mainly to print the hash table out and to add values from a .txt file to the hash table. Methods writeToFile, writeToFile2 and writeOrdered write the hash table to a given .txt file, in a different way. WriteToFile writes the hash table as it is, with linked lists present, writeToFile2 writes each of the hash table values on their own line, and writeOrdered sorts the hash table alphabetically, and writes the values to a file after that.

AddFromFile can be used to easily add values from a .txt file to the hash table. The function goes through each line from the file and adds them to the hash table using the adder function.

PrintTable can be used to print the hash table out in a clear format, where each linked list has its own index. The printTable works best with smaller hash tables, since the output becomes more cluttered as the size of the hash table grows.

### 2. Testing and Analyzing the Hash Table

#### 2.1 Running time analysis of the hash table

* What is the running time of adding a new value in your hash table and why?
* What is the running time of finding a new value in your hash table and why?
* What is the running time of removing a new value in your hash table and why?

Use Θ notation. Consider what factors influence the running time of the methods.

### 3. The Pressure Test

Table 1. Results of the pressure test.

|  |  |
| --- | --- |
| **Step** | **Time (s)** |
| Initializing the hash table |  |
| Adding the words |  |
| Finding the common words |  |
|  |  |

Hash\_3\_1.py runtimes (size 10000) Hash\_3\_2.py runtimes

|  |  |
| --- | --- |
| # | Runtime(s) |
| 1 | 1.66554213 |
| 2 | 1.90644813 |
| 3 | 1.70513701 |
| 4 | 1.71180367 |
| 5 | 1.70722818 |
| 6 | 1.69835520 |
| 7 | 1.72848606 |
| 8 | 1.70621872 |
| 9 | 1.72077060 |
| 10 | 1.85961938 |
| Avg | 1.7409609 |

|  |  |
| --- | --- |
| # | Runtime(s) |
| 1 | 252.58100128 |
| 2 | 251.96795321 |
| 3 | 272.36241317 |
| 4 | 258.65484381 |
| 5 | 254.32136822 |
| 6 | 258.35052371 |
| 7 | 267.83671069 |
| 8 | 268.33103919 |
| 9 | 275.48070908 |
| 10 | 266.20386243 |
| Avg | 262.611986161 |

#### 3.1 Comparison of the data

#### structures

Which data structure was faster in adding the words from the file and why? In which data structure was the search faster and why?

#### 3.2 Further improvements

|  |  |
| --- | --- |
| **Hash table size** | **Average runtime(s)** |
| 100 | 88.741962745 |
| 1k | 8.790079427 |
| 10k | 1.740960908 |
| 100k | 0.909177971 |
| 1M | 1.031345439 |

Are you able to make the program faster?

* Try to change the size of the hash table.
* How well is the data distributed in the hash table?

### List of references

Add here the references and source that you used.

<https://www.gatevidyalay.com/wp-content/uploads/2018/06/Separate-Chaining-Collision-Resolution-Techniques-Step-08.png> - figure 1.1a