Task description: Open Hashing

1. Implementing the Hash Table

Create your own hash table that uses open hashing in Python. Each slot of the hash table contains a linked structure where the data (keys) are stored. The hash system must include search, insert, and delete operations. The hash table must be able to store both integer (*int*) and string (*str*) values. That means that also all the operations/methods (search, insert, and delete) need to work with both data types. You can decide or design the hash function by yourself.

Consider following things when you are creating the hash table:

- The size of the hash table is fixed. That means that after initializing the size of the table must stay the same.
- You must implement the linked structure where the data is stored by yourself.
- Choose your hashing function wisely because it must work efficiently with very large hash tables. A Good start is the string folding. Be as creative you want but be prepared to explain how it works!
- Document your code!

Save the code of your new data structure as **hash_1.py**

Answer to the following essay questions:

1. Present the structure of your hash table.
2. What hashing function did you choose and why?
3. What (including required) methods your hash table has and explain briefly how do they work?

2. Testing and Analysing the Hash Table

Create a Python program: **hash_2.py**:

1. Create a new hash table of size **3.** Add items **12, 'hashtable', 1234, 4328989, 'BM40A1500', -12456, 'aaaabbbbcccc'** to the hash table. Present the structure of the hash table each time when a new value is added.
2. Now try to find values **-12456, 'hashtable', 1235**. Print out the results.
3. Remove values **'BM40A1500', 1234, 'aaaabbbbcccc'**. Present the final structure of the hash table.

Answer to the following essay questions:

1. What is the running time of adding a new value in your hash table and why?
2. What is the running time of finding a new value in your hash table and why?

3. 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

Let's put the hash table in a real use. The text file *words_alpha.txt* (source: https://github.com/dwyl/english-words/) contains **370105** English (and not so English) words. The text file *kaikkisanat.txt* (source: https://github.com/hugovk/everyfinnishword) contains **93086** Finnish words. Your task is to find all words from *kaikkisanat.txt* that are also in *words_alpha.txt* (exact matches).

Create a new Python file **hash_3_1.py**:

1. Create a new hash table of size **10000**.
2. Read all words from *words_alpha.txt* and store them to your hash table.
3. While reading words from *kaikkisanat.txt* check how many of them can you find from the hash table and print out the final result.

Measure the runtime for each step. Tabulate the results as follows:

Process	Time (s)
Initializing the hash table	
Adding the words	
Finding the common words	

How does your hash table stand against a linear array? Repeat the previous test, but this time store the words from *words_alpha.txt* to a *list* instead of the hash table. Save your code as **hash_3_2.py**

Answer to the following essay questions:

1. Which data structure was faster in adding the words from the file and why?
2. In which data structure was the search faster and why?
3. Are you able to make the test program in **hash_3_1.py** faster (even slight improvements)?
    - Try to change the size of the hash table.
    - How well is the data distributed in the hash table?

Provide your answers to all the essay questions in a single PDF file.