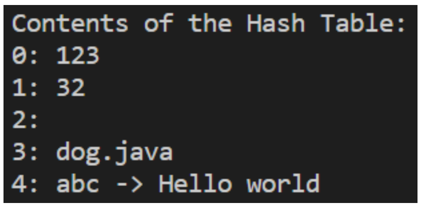
Practical Assignment

BM40A1500 Data Structures and Algorithms

1. Implementing the Hash Table

1.1 Structure of the hash table

Figure 1 shows an example structure of a hash table, size of 5. Each non-empty slot has their keys stored in a linked list. The slot where a new value is stored is based on the results of the hash function, and then it will be inserted as the last item of the slot's linked list. When we insert a new value to an empty slot, we will create a linked list to that slot and insert the value as the first key of the list. Also, when deleting the only key of a slot, we will delete the slot's linked list. This means that there are no empty linked lists in this structure.



```
Contents of the Hash Table:
0: 123
1: 32
2:
3: dog.java
4: abc -> Hello world
```

Figure 1: Structure of the Hash table

1.2 Hash function

I used string folding as the hash function. Sum based on string folding can be calculated using following formula: $\sum_{i=0}^{j-1} f(s_i) \cdot 256^{i \bmod 4}$, where j is the number of characters in the input, s_i is a character of the input and f is a function which calculates the ASCII-values, known as ord() in Python. After calculating the sum, we will calculate the remainder based on the size of the hash table. The sums based on string folding are long integer values, and therefore they work well with large hash tables.

1.3 Methods

Class Openhash contains following methods:

- __init__(self, S): Sets up the hashing table, size of input S.
- stringfolding(self, data): Calculates hash value of the input data using string folding.
- insert(self, data): Stores the given input data in hash table; Calls the method stringfolding to get the location of the linked list in the hash table, where we want to store the input data, and then calls the LinkedList's method append (see class LinkedList append(self,data)) to store the data in the slot's linked list.
- delete(self, data): Deletes the given input data from the hash table; Calls the method stringfolding to get the location of the linked list in hash table where the input data is stored (if it is stored in hash table), then calls LinkedList's method searchindex (see class Linked List searchindex(self,value)) to get the location (index) of the data in the linked list. If the

data is stored in the linked list, it will call the LinkedList's method delete (see class LinkedList delete(self,index)) to delete the value from the linked list.

- print(self): Prints the contents of the hash table; Goes through every slot of the table and calls the LinkedList's method print for each slot to print every key of the table located in linked lists.
- search(self, value): Searches the hash table if the input value is stored in the hash table; Calls the method stringfolding to get the location of the linked list in hash table where the input data is stored (if it is stored in hash table), then calls LinkedList's method searchindex (see class Linked List searchindex(self,value)) to get the location (index) of the data in the linked list. Based on the result of the searchindex, boolean value (True/False) is returned.

Class LinkedList contains following methods:

- append(self, data): Inserts input data as the last key of the linked list. The method also checks that the input data is not already stored in the linked list, so it doesn't allow duplicates.
- print(self): Prints each key of the linked list.
- searchindex(self, value): Searches location of the input value from the linked list. Returns location as index. If the value is not stored in the linked list, method returns -1.
- delete(self, index): Based on input index, method deletes the wanted key from linked list.

2. Testing and Analyzing the Hash Table

2.1 Running time analysis of the hash table

- The running time of adding a new value:
  - Let n be number of characters in the input, S the size of the hash table and m the number of inputs. Running time of the string folding function is $\Theta(n)$. Running time of accessing the slot based on result from string folding is $\Theta(1)$. When inserting a value to a slot that has no previously inserted keys, we will create a linked list and insert the value to the list, so running time is constant. When we insert a new value to a linked list which has previously inserted keys, it can contain $1 \dots m$ keys, depending on how the keys are distributed in the table. We must check each value of the linked list, to avoid duplicates. When $m \leq S$, each linked list contains ~1 key on average and therefore the running time of inserting a new value to linked list is $\Theta(1)$. When $m > S$, which is a much more typical situation, each linked list of the table contains $\frac{m}{S}$ keys on average and therefore the running time of the append-function is $\Theta(\frac{m}{S})$. Since the latter situation is much more relevant, the running time of adding a new value is $\Theta(n + \frac{m}{S})$.
- The running time of finding a value:
  - Let n be number of characters in the input, S the size of the hash table and m the number of inputs. Running time of the string folding function is $\Theta(n)$. Running time of accessing the slot based on result from string folding is $\Theta(1)$. Using the searchindex-function we will find out if the searched value is stored in the list. A linked list can

contain 1 ... m keys, depending on how the keys are distributed in the table. When $m > S$, a linked list contains an average of $\frac{m}{S}$ keys. Comparing keys (nodes) with the searched value, done in searchindex-function, has a running time of $\Theta(\frac{m}{S})$. Therefore, the running time of the finding a value is $\Theta(n + \frac{m}{S})$.

- The running time of removing a value:
    - Let n be number of characters in the input, S the size of the hash table and m the number of inputs. Running time of the string folding function is $\Theta(n)$. The running time of accessing the slot based on result from string folding is $\Theta(1)$. The running time of the searchindex-function is $\Theta(\frac{m}{S})$. Based on the result from the searchindex-function, we go through a linked lists values until we reach the value to be deleted. On the average case, index value is somewhere between 0 ... $\frac{m}{S}$, so the remove-function has a running time of $\Theta(\frac{m}{S})$. Therefore, the running time of removing a value is $\Theta(n + \frac{m}{S})$.

3. The Pressure Test

Table 1. Results of the pressure test.

Step	**Time (s)**
Initializing the hash table	$1.800 \cdot 10^{-5}$
Adding the words	3.818
Finding the common words	1.716

3.1 Comparison of the data structures

Linear array was faster than hash table in adding words because the running time with inserting keys using linear array is only $\Theta(1)$ compared to the running time using hash table $\Theta(n + \frac{m}{S})$. However, hash table was faster with searching because the running time of the search using hash table is $\Theta(n + \frac{m}{S})$ which is much less compared to running time of search using array $\Theta(m)$, note that size of the hash table was $S = 10\ 000$ in the pressure test. The benefit of the open hashing is that we can search the desired value from a smaller set of keys, compared to arrays where we must search the value among all the keys.

3.2 Further improvements

Changing size of the hash table:

- Changing size of the table from 10 000 to 15 000 increased the initializing time of the table by $0.6 \cdot 10^{-5}$ seconds but decreased both inserting time by 0.54 seconds and search time by 0.43 seconds, so overall time improvement was approximately 0.96 seconds.

Figure 2 shows how the number of keys is distributed between slots with the hash table size of 10 000. According to figure 2, the linked lists contain 17-60 keys/list, and we can see that frequencies of the small and large lists are quite small compared to frequencies of the lists which sizes are closer to 37.

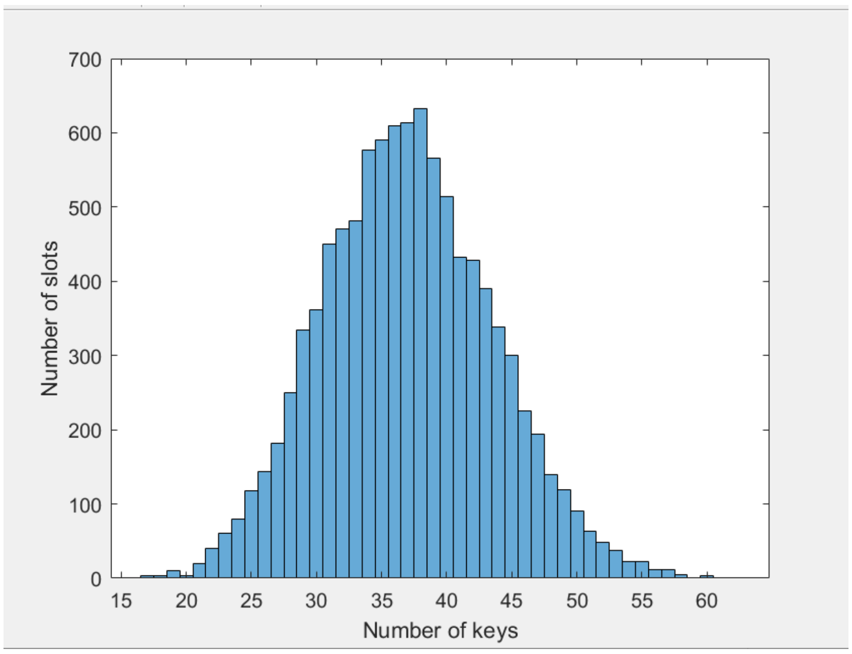


Figure 2

List of references

Antti Laaksonen, Tietorakenteet ja Algoritmit 2021