Practical Assignment

BM40A1500 Data Structures and Algorithms

1. Implementing the Hash Table

1.1 Structure of the hash table

My hash table contains a list which contains *n* linked lists when *n* is the size of the hash table. It means that I have an own linked list for each slot of my hash table. Each value added to the hash table has a key which is a number from 0 to *n*-1. It defines the slot of the value in the hash table.

Class ‘Node’ is for each node of each linked list. Class ‘LinkedList’ is for the *n* linked lists. Class ‘HashTable’ contains the hash table itself.

1.2 Hash function

I used string folding as my hashing function. I started from it as it was mentioned in the task description. It worked very well, and I couldn’t find any better solutions, so I decided to use it. When using string folding, the words in the task split evenly between the slots which is important when using big data.

The hashing function takes a new value as an input parameter and modifies it to a string. Then, it handles the string four bytes at a time and modifies them to integers using the ascii values of the string. Then, we sum these integers and finally get the key value with modulus operator so that we get values from 0 to *n*-1.

1.3 Methods

The method *hashValue* is for calculating the key for each value in the hash table. It takes the value as an input parameter and modifies it to string. Then the method calculates and returns the key for the value using the formula for string folding.

The method *insert* is for adding new values to the hash table and it takes the value as an input parameter. It calls the *hashValue* method to find a key for the input value. Then it appends a value to the correct slot defined by key by calling the *append* method in class ‘LinkedList’.

The method *delete* is for deleting a certain value from the hash table. It takes the value as an input parameter. Then it calls the *hashValue* method to get a key to the value. After that it calls the *search* method in class 'LinkedList' which checks if the value is in the correct linked list in the hash table. The list is defined by the key. If the *search* method returns 'True', the value is in the hash table so it calls *delete* method in class 'LinkedList' which then deletes the value from the linked list (and the hash table).

The method *search* tells if a certain value is in the hash table. First, it calls the *hashValue* method to get a key for the value. Then it calls a method *search* in class 'LinkedList', which goes through a correct linked lists and returns a Boolean value 'True' if a value is in the list and False if the value is not in the list. The method *search* in class 'HashTable' returns the same Boolean value.

The method *print* prints the hash table. It goes through each slot of the hash table in a for loop and prints the contents of each slot (= each linked list) with the help of the *print* method in class 'LinkedList'.

2. Testing and Analyzing the Hash Table

2.1 Running time analysis of the hash table

When a new value is added, removed, or searched for, my program first calculates the key value for a new value. If s is the length of the new value (as a string), the complexity for calculating the key value is $\Theta(s)$, because there's one for loop from 0 to s needed. After the calculation the program goes through a linked list. The complexity in this part is $\Theta(n)$ when adding, when n is the number of values in linked list, because we go through all the values and add to the end of the list. When searching for a value or removing, we might not need to go through all the values in linked list (so n is smaller). We could say that the complexity of these operations is $\Theta(s+n)$.

However, if we consider the time complexity of the whole program, I would still say that it's $\Theta(1)$. The key value calculation and adding / removing / searching with a linked list are very fast operations when the data splits evenly to the linked lists. Even with large hash table sizes the number of items in one linked list is small.

3. The Pressure Test

Table 1. Results of the pressure test.

Step	**Time (s), hash table**	**Time (s), list**
Initializing the hash table / list	0,014	-
Adding the words	5,937	0,215
Finding the common words	1,782	566,000

3.1 Comparison of the data structures

The normal Python list is faster in adding the words from the file because it just appends every single word to the same list. The hash table first calculates the key value for each word and then appends it to the correct linked lists. It's obvious that the calculating of the key values takes some time when we have that many words, so the hash table is a bit lower.

It's also good to note that the hash table still checks if the value already is already in the table, but the normal list doesn't. However, there aren't any duplicates in these files.

Hash table is much faster than list in searching for the words. That is because when searching for a value, the hash table first calculates the key for the value and then it can search for the value from the correct list. The words are divided into 10 000 linked lists so when it's only needed to check one of them, the searching is fast. When using normal list, all the words are in one list so there are much more words to go through so the search is much slower.

3.2 Further improvements

I couldn't make any further improvements to my hash table. I tried to optimize the code as well as I could since the beginning of the assignment, and I didn't find anything that I could do to make it faster. I tried to modify especially the methods which search for a specific value in the table but concluded that the first one was the best I could do.

When using the asked size 10 000, the words distribute very evenly to the different slots. However, the program runs faster when the size of the hash table is bigger. That is because when the hash table is bigger, there are more linked lists in it which means that when searching for a word there are always less words to go through. For example, if the size of the hash table is the number of words (370 105), it took 2,056 seconds to add the words to the table and 0,333 seconds to find the common words. The initializing of the hash table takes more time the bigger the hash table is, in this case it took 1,143 seconds. When using a hash table that big, the words don't distribute well and there are lots of slots without any words in them. A more reasonable solution is for example the size 20 000, when the initializing (0,033 s), adding the words (4,468 s) and searching for the common words (1,408 s) are very fast operations and the words are still very evenly distributed.

List of references

https://pynative.com/python-get-execution-time-of-program/

BM40A1500 Data Structures and Algorithms, the course material