Practical Assignment

BM40A1500 Data Structures and Algorithms

1. Implementing the Hash Table

1.1 Structure of the hash table

The hash table consists of a list of linked lists. The linked lists have their own search, insert and remove operations. Each linked list also has empty head and tail nodes to make list modifications easier. The “hash table”-class initializes the linked lists and stores the hash table. The hash function is also stored in the “hash table”-class. All list modifications are done by the “linked list”-class.

1.2 Hash function

The built-in Python hash() function is used to return the hash value of a given key. This hash value is then used in combination with the modulo-operator to find a slot for the key in the hash table.

1.3 Methods

The hash table includes a findIndex()-function that returns the location of a given key in the linked list. This is used in the remove()-function. FindIndex()-function iterates over a list and returns the locations of the key if it is found. The function returns –1 if key is not found.

2. Testing and Analyzing the Hash Table

2.1 Running time analysis of the hash table

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

When working with linked lists we cannot instantly jump to the end of the list. Because each node only knows where the next node is, we must traverse the entire list from the beginning one node at a time. If the length of our linked list is *n*, on average we must traverse *n/2* nodes before reaching the end, where we can insert the new node. The running time for traversing the list is **Θ(n)**. The insertion itself only requires us to link the new node with the last node in the list. This can be done in **Θ(1).**

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

Again, finding values requires us to traverse the list from the beginning. The value can be found only if we happen to run into it while traversing. The only way to confirm that a value is not in the list is to go through every element. For a list of *n* elements, the running time is **Θ(n)**.

3. What is the running time of removing a new value in your hash table and why?

Similary to adding a value to the linked list, removing an element only requires **Θ(1)**. The removal is done by assigning the previous element's next node to jump over the node we want to remove (for example, removing the node after "current", **current.next = current.next.next**). Traversing the list is required here too, unless we remove the first node. Running time for traversal **Θ(n)**.

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	0.02
Adding the words	3.10
Finding the common words	1.66

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?

Adding the words to a linear array was significantly faster in comparison to the hash table. This only took around 0.17 seconds on average. Appending to a linear array has constant time complexity **O(1)**, while inserting values into the linked list has time complexity of **O(n)** due to the list traversal that is required.

Searching in a linked list and a linear array both have a time complexity of **O(n)**. Still, the difference in runtimes is massive. On average, the linear array search took 320 seconds, almost 200 times longer than searching in the hash table. However, the difference is easily explainable. There are 370105 words inserted into the hash table and array. The hash table is split into 10000 different linked lists. If we assume that our algorithm is able to split the values perfectly among all the linked lists, each of the lists would contain 37 items. In the worst case, we must traverse through all 37 items on the list. With the array, we must traverse through 370105 items in the worst case. Assuming that none of the values we search for are in our hash table or array, the worst-case traversal must be repeated 93086 times. That would make 3 444 182 checked items for the hash table and 34 451 594 030 for the array. 10000 times more checks are required in the array search, which is also the number of linked lists used in the hash table for this test.

3.2 Further improvements

Are you able to make the program faster?

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

The performance of the hash table search is based on the number of nodes that need to be traversed. By increasing the size of the hash table, we can increase the performance of the search function. On average the hash table has all its linked lists between lengths 20 and 60. Considering the mathematical average of 37 items in each list for size 10000 hash table, the deviations are unlikely to cause any serious performance issues. Still, the worst-case running time would be 1.5 times longer than that of an optimally distributed hash table.

One possible solution to improve the performance of the hash table is to replace the linked lists with binary trees. This way the time complexity of our search would be reduced to **O(log n)**.

List of references

https://pankaj-rai.medium.com/know-your-data-structure-linked-list-4b00fcfbda93