Word Auto-Completion

Prateek Kumar Singh

DESIGN OF EVALUATION:

INPUT GENERATION:

<u>For Scenario 1 & Scenario 2:</u>

We used the random module of python to generate a list of random words from the 200 thousand unique words available in sampleData200k.txt.

```
import random

random_words = random.sample(total_words,10)
random_freq = random.sample(range(500),10)

input_fileADD = open('input_search.in', 'w')

for i in range(len(random_words)):
    input_fileADD.write('A '+random_words[i]+' '+str(random_freq[i])+'\n')

input_fileDELETE = open('input_delete.in', 'w')
for i in range(len(random_words)):
    input_fileDELETE.write('D '+random_words[i]+'\n')

input_fileADD.close()
input_fileDELETE.close()
```

A random sample is taken every time when the dictionary is grown for Growing Dictionary (scenario 1) or when it is reduced for Shrinking Dictionary (scenario 2). We did this to ensure that the algorithm is introduced to new words to make observationsmore dependable and there is no bias.

The inputs are the same for all the data structures for a particular dictionary size so that the data structures can be tested without the likelihood of the worst-case or the best case which may be possible when we change the inputs to test a different data structure. For instance, the inputs used to test the time complexity for list implementation of a 1000 words dictionary, the same inputs will be used to test hash or TST of the same dictionary size for a fair comparison.

For Scenario 3:

For the search operations, we used the random module to randomly select 5 words from every static dictionary i.e. Small, Medium, Large dictionaries. To make the search observations dependable, we purposely used 5 words that belong to the dataset and 5 words that are not in the dictionary. Since the dictionaries are distinct and do not have words in common, we used the 5 random words which are present in other dictionaries. For example, in the case of a small dictionary, we randomly took 5 words from the small dictionary and for the other 5 words, we usedthe words present in either a Medium dictionary or Large Dictionary. This code aboveis used to generate 5 random words from each size of the dictionary and then use theircombination as discussed above for analysis.

```
# Program to get 5 random words from a dictionary
import random

data_file = open('SMALL.txt', 'r')
total_words = []
for line in data_file:
    values = line.split()
    word = values[0]
    total_words.append(word)
data_file.close()

random_words = random.sample(total_words,5)
input_file = open('input_search.in', 'w')
for i in range(len(random_words)):
    input_file.write('S '+random_words[i]+'\n')
input_file.close()
```

For autocomplete operation, we used the string module of python along with the random module to generate a combination of random letters. From these random letters, we test the autocomplete function and record their running times for different data structures.

```
import random
import string
random.seed(10)
letters = string.ascii_lowercase

input_file = open('input_auto.in', 'w')
for i in range(10):
    rand_letters = random.choices(letters,k=2)
    input_file.write('AC '+rand_letters[0]+rand_letters[1]+'\n')

input_file.close()
```

DATA GENERATION:

The data is taken by subsetting the provided sampleData200k.txt into 4 parts.

```
i = 1
for line in data_file:

    if i <= 50:
        output_file1.write(line)

    if i<=1000:
        output_file2.write(line)

    if i<=10000:
        output_file3.write(line)

    if i<=50000:
        output_file4.write(line)

    i += 1
```

For scenario 1 i.e. Growing Dictionaries, We used 4 Datasets of 50,1000,10000, and 50000 words approximately and then performed Add operations for each one with random inputs and observed the time complexities for the different data structures.

We wrote the python script on the right to generate these datasets. The thing to note here is that the dataset is growing and it also contains

the words from the previous dataset as well and hence called the growing dictionary.

For scenario 2 i.e. Shrinking Dictionaries, we did the reverse of what we did for growingdictionaries. We started with the 50 thousand word dictionary and went on till the 50 word dictionary to perform Delete operations with random inputs and observe the timecomplexities for the different data structures.

For scenario 3, As per the specification, we created Small, Medium, and Large dictionaries with 1,000, 50,000, and 149,000 words respectively.

The dictionaries are distinct from each other. We chose the split of 1,000, 50,000, and 149,000 words, to make a considerable difference between the running times so that the analysis can be made easily for different data structures.

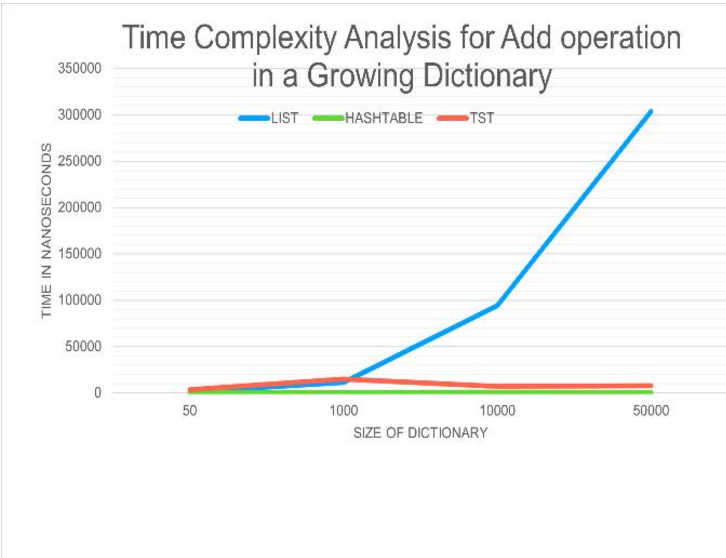
```
i = 1
for line in data_file:
    if i <= 1000:
        output_file1.write(line)

    elif i > 1000 and i<=51000:
        output_file2.write(line)

    else:
        output_file3.write(line)

    i += 1
```

<u>The Method used for measuring Time:</u>

From the time module of python, we used the method time_ns(), to calculate the running times of algorithms in nanoseconds. We used the unit nanoseconds becausesome of the operations that were below 0 seconds were not calculated by the compiler.The running time is calculated for 10 operations in every scenario i.e. for scenario 1, scenario 2, and scenario 3, and the time taken for all the operations was averaged and then recorded.

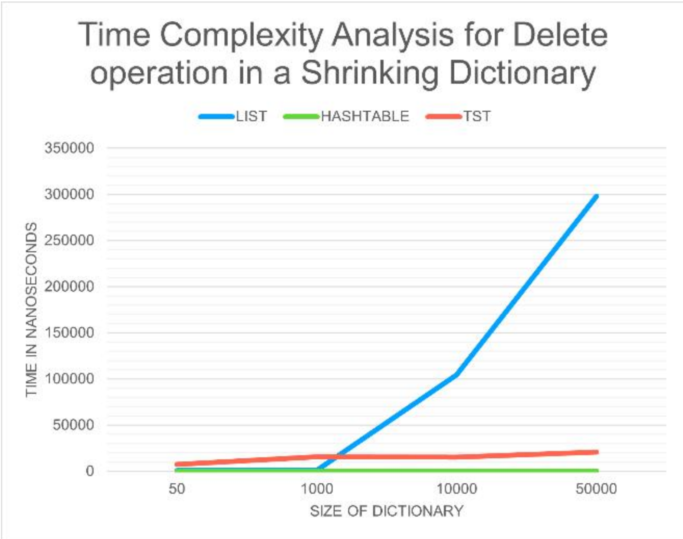
Evaluation and Analysis of Results:

Following are our testing of these different approaches, we can graph our results to see the differences in the time in nanoseconds based on the size of our test data.

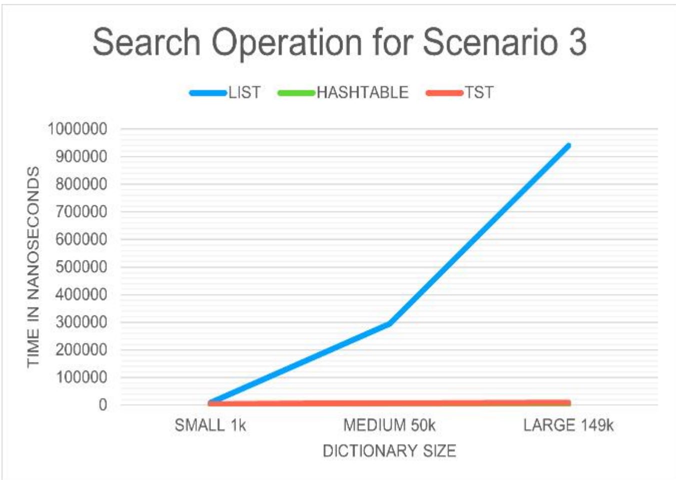


First is the result data for add operation in the growing dictionary, from the graph, we can see that from a size of 50 to 1000, the difference in times are minimal, but as thesize increases further, to 10000 and to 50000, the list implementation stands out as amajor increase in the time taken. While the hash table implementation maintains a relatively similar time throughout all the sizes.

List or array approaches have a theoretical average time complexity of O(n), whilst the hash table has a theoretical average time complexity of O(1). The test data that is generated follows the exact theoretical time complexities of these approaches within the margin of error.

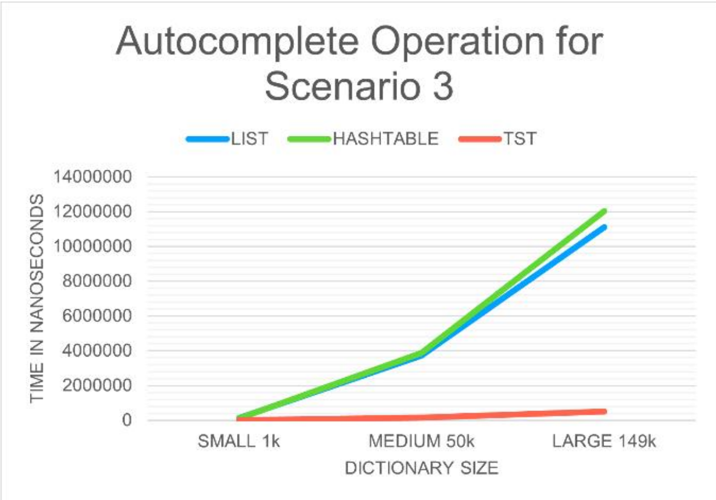


Next, is the graph showing the result for the time taken reported for the delete operation for these approaches. This graph is like the result seen in the add operation, with the list approach drastically increasing as the size of the dictionary increases past the size of 1000. The hash table approach also still maintains a straight line across theboard of different sizes. In these approaches, the average theoretical time complexityis the same as the add operation, and since the result data of the delete operation hasthe same result as the add operation, within margins of error of sub-optimized code, the results seen align themselves with their respectively average time complexity.



The next operation that we will look at is the search operation using our Scenario 3 to test their performance with respect to broad differences in the dictionary size ranging from one thousand to 149 thousand. Throughout all the sizes, the hash table approach is seen again with little to no difference in time taken. Compared to its average time complexity which is O(1), the result aligns itself very well with what is expected.

Regarding the list approach, the expected average time complexity for search operation is O(n), which is exactly what can be seen from the result data from the graph.



Lastly is the autocomplete operation, for this both the list and hash table approaches show similar results with the hash table being marginally higher, which may be due tomargins of error in coding efficiency.

Given the analysis of the different operations of add, delete, search and autocomplete above, when comparing list and hash tables. Hash table is highly recommended instead of lists for the add, delete and search. Hash tables are better in this use case,especially for a dictionary implementation like this, since each word has a unique keywhich makes for faster and easier when trying to look up existing words using the index. Whereas lists are better suited if memory size is a concern.

APPENDIX:

Table 1

	LIST (time in ns)		HASH (time in ns)		TST (time in ns)	
ADD_SIZE						
50	1000		500		3600	
1000	11300		500		15000	
10000	94200		600		7000	
50000	303500		300		7700	
DELETE_SIZE						
50	900		300		7400	
1000	1280		400		15700	
10000	104400		300		15200	
50000	298000		500		20700	
SEARCH_SIZE						
SMALL 1k	8500		300		3700	
MEDIUM 50k	294700		500		7000	
LARGE 149k	940800		1000		7900	
AUTO_SIZE						
SMALL 1k	130900		108100		6100	
MEDIUM 50k	3723900		3865600		164300	
LARGE 149k	11103300		12040900		503500	