Assignment 1 COSC2123

Student Id-s3820112

Empirical Analysis Report

Data Generation

The mechanism that I have utilized for the generation is such that it takes care of all three scenarios namely “Growing Dictionary”, “Shrinking Dictionary” and “Static Dictionary”. In order to achieve this, I closely followed the pattern used in the skeleton code for implementation of the analysis. In this part B of the assignment, I have chosen to use the Sample200k.txt dataset to give me a large sample size to work upon. Parameters settings involved were basic and are displayed in figure below. For the run time calculation I have used sum()/len() function to help me analyze the average run times of the different algorithms.

Size Distribution

I created dictionaries of various sizes so that we can use those to perform the analysis. The size distribution that I have created for the analysis is as follows

- 100
- 500
- 1000
- 5000
- 10000
- 50000
- 100000
- 200000

Here we can see that the size is increasing in a non-linear fashion. I intend to perform operations on these sizes and test them on our program to analyze the effects

Data Curation

To create these dictionaries, I used the following steps

- Read the data from the dataset file
- Create an array of Word Frequencies sourced from the dataset file

```python
def read_data():
    # read from data file to populate the initial set of points
    data_filename = "sampleData200k.txt"

    words_frequencies_from_file = []

    try:
        data_file = open(data_filename, 'r')
        for line in data_file:
            values = line.split()
            word = values[0]
            frequency = int(values[1])
            # each line contains a word and its frequency
            word_frequency = WordFrequency(word, frequency)
            words_frequencies_from_file.append(word_frequency)
        data_file.close()
        return words_frequencies_from_file

    except FileNotFoundError as e:
        print("Data file doesn't exist.")
```

- Use splices of the array to create a dictionary

```python
def create_dict(dict_type: str, word_frequency, sizes):
    # initialise search agent
    all_dicts = []
    agent: BaseDictionary = None
    for i in range(len(sizes)):
        if dict_type == 'list':
            agent = ListDictionary()
        elif dict_type == 'hashtable':
            agent = HashTableDictionary()
        elif dict_type == 'tst':
            agent = TernarySearchTreeDictionary()
        all_dicts.append(agent)
    for itr in range(len(sizes)):
        all_dicts[itr].build_dictionary(word_frequency[:sizes[itr]])
```

- These dictionaries are stored in an array and returned so that an analysis can be performed

Analysis

To perform the analysis, I wrote the script. What the script does is that:

- It creates an array of dictionaries of various sizes.
- Then perform an operation on each size.
- Measures the time stamp before and after the operations
- I do multiple runs for the operation and average the result.

I have used sum()/len() method instead of numpy.average() as I have selected a large dataset with 200k elements and wanted to start the timestamp just before the operation and preferred sum()/len() method for it to average it out for the most accurate result not just the fastest.

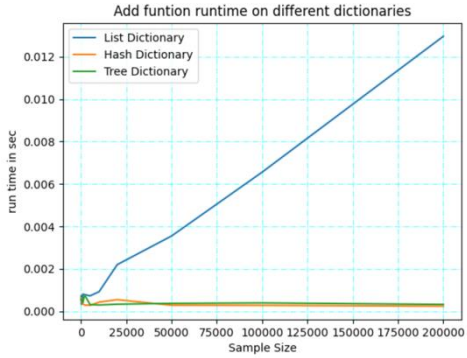
Analysis Operations

Here are 4 operations we are analyzing the algorithms on- Add, Autocomplete, Search, Delete.

We will check the performance of these operations on increasing the sample size of dataset used in the operation.

Add

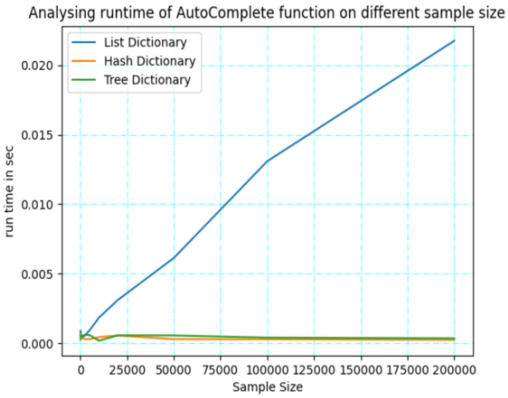
```
analysis_file.write("---------- ADD ----------\n")
hash_dicts=create_dict("hashtable", word_frequency, sizes)
for itr, dic in enumerate(hash_dicts):
    analysis_file.write("-------\n")
    analysis_file.write("-------\n")
    add_times = []
    # Addition to structure
    add_times.append(execute_command("A booming1 123456",   dic))
    add_times.append(execute_command("A loblolly1 123456",  dic))
    add_times.append(execute_command("A tvpg1 123456",      dic))
    add_times.append(execute_command("A ohhlacom1 123456",  dic))
    # Status
    analysis_file.write(f"Dictionary Length: {sizes[itr]}\nCommand: Add a word\nAverageTime:{sum(add_times)/len(add_times)}\n")
```



As evident from this graph add operation on each dictionary produces different runtimes in the increase of the sample space. List dictionary performs worst with steep growth in the runtime with the increase of sample size, Hash and Tree dictionaries however doesn't get affected much, other than the small jerk observed in the starting the average run time 0.0002-0.0006 sec. If observed carefully the hash dictionary performs slightly better than tree dictionary in the large datasets.

Auto Complete

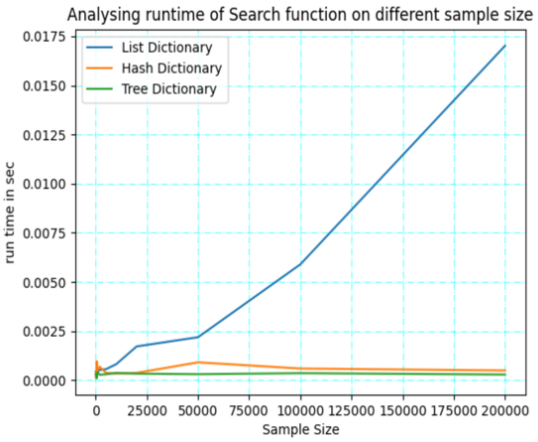
```
analysis_file.write("---------- ACP ----------\n")
for itr, dic in enumerate(hash_dicts):
    analysis_file.write("-------\n")
    analysis_file.write("-------\n")
    ser_times = []
    ser_times.append(execute_command("ACP boom",  dic))
    ser_times.append(execute_command("ACP lobl", dic))
    ser_times.append(execute_command("ACP tvp",     dic))
    ser_times.append(execute_command("ACP ohh", dic))
    # Status
    analysis_file.write(f"Dictionary Length: {sizes[itr]}\nCommand: Search a word\nAverageTime:{sum(ser_times)/len(ser_times)}\n")
```



As we can see in this graph list dictionary shows steep growth in runtime exceeding 0.02 sec with sample size of 200k. Hash and Tree dictionary remains somewhere constant with the little disturbance in the start with less sample size. Analyzing between hash and tree dictionary here also hash dictionary preforms slightly better with less runtime at 200k sample size.

Search

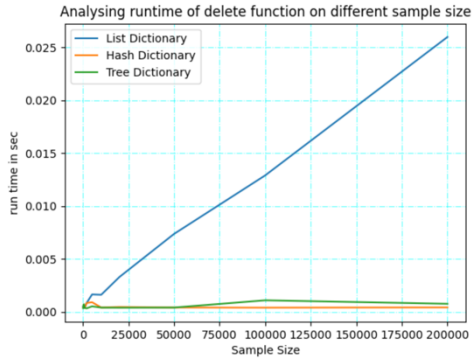
```
analysis_file.write("---------- SER ----------\n")
for itr, dic in enumerate(hash_dicts):
    analysis_file.write("-------\n")
    analysis_file.write("-------\n")
    acp_times = []
    acp_times.append(execute_command("S booming1",  dic))
    acp_times.append(execute_command("S loblolly1", dic))
    acp_times.append(execute_command("S tvpg1",     dic))
    acp_times.append(execute_command("S ohhlacom1", dic))
    # Status
    analysis_file.write(f"Dictionary Length: {sizes[itr]}\nCommand: AC a word\nAverageTime:{sum(acp_times)/len(acp_times)}\n")
```



If we look at the search function plot graph of all the dictionary, the list dictionary here again displays steep significant growth and shows direct propositional relationship with the sample size. The Hash and Tree dictionary however show a very less variation with the increase in the sample data used. On careful observation the tree dictionary performs better in search function than hash dictionary here.

Delete

```
analysis_file.write("---------- DEL ----------\n")
for itr, dic in enumerate(hash_dicts):
    analysis_file.write("-------\n")
    analysis_file.write("-------\n")
    del_times = []
    del_times.append(execute_command("D booming1",  dic))
    del_times.append(execute_command("D loblolly1", dic))
    del_times.append(execute_command("D tvpg1",     dic))
    del_times.append(execute_command("D ohhlacom1", dic))
    # Status
    analysis_file.write(f"Dictionary Length: {sizes[itr]}\nCommand: Delete a word\nAverageTime:{sum(del_times)/len(del_times)}\n")
```



In the final delete function shows no difference with list dictionary being the most time consuming with the increase in sample space. Between hash and Tree dictionary however, hash dictionary here has less runtime at large sample spaces than tree dictionary

Result Formatting/Conclusion

The results for the analysis are saved in analysis.txt in the shared source code.

So, the report here concludes that list dictionary requires most of the time when we approach large datasets. The hash and Tree dictionaries don't show any remarkable difference with the increase of sample sizes but in most cases like with Add, Autocomplete and Delete functions the hash dictionary requires less time to operate than tree dictionary. The tree dictionary performs only better than hash in the case scenario of search function.