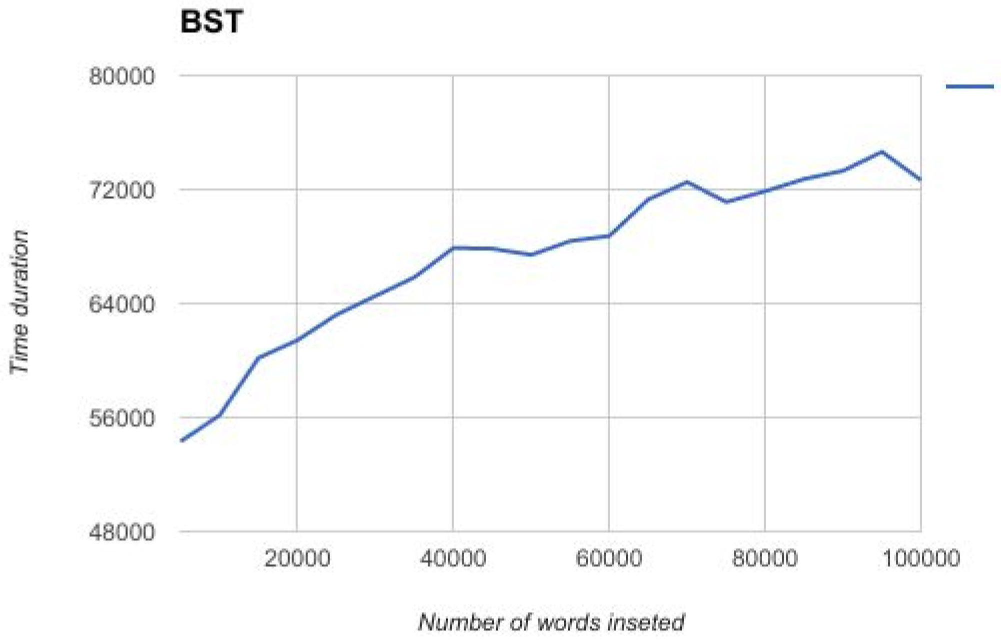
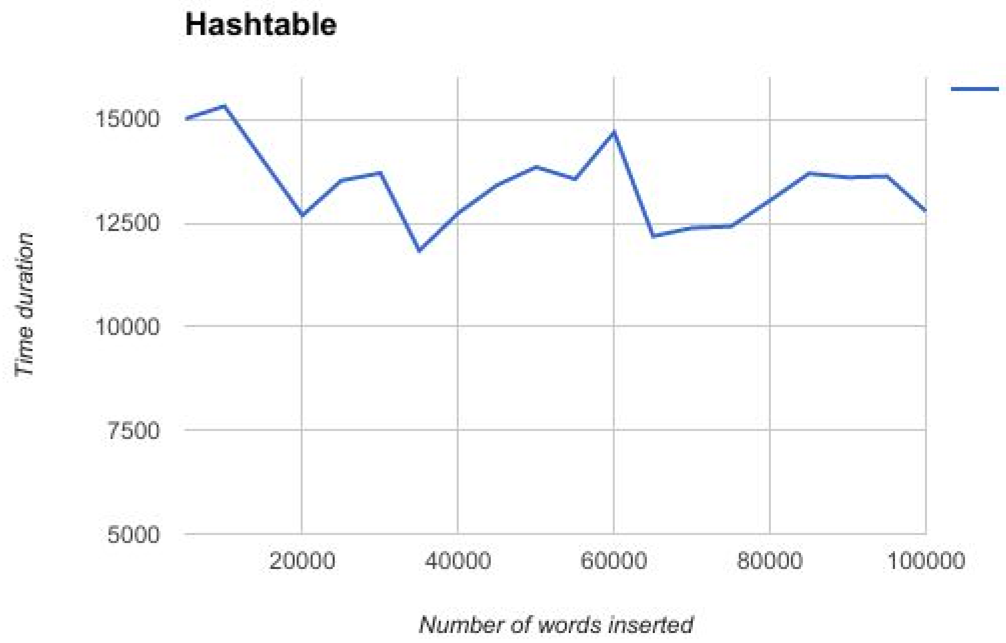
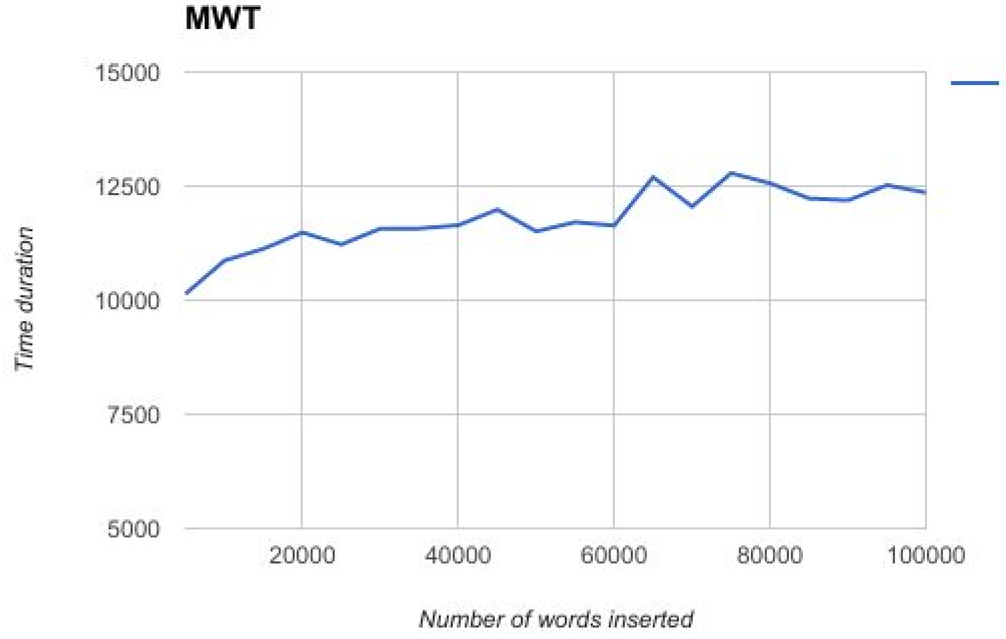
Final Report

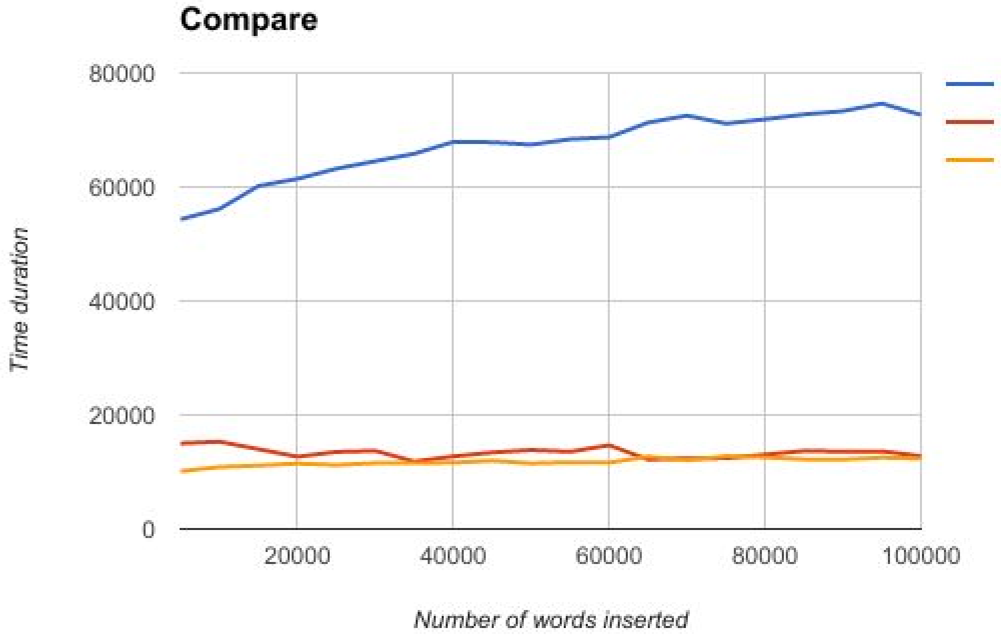
Cheng Qian, PID:A53209561

reason about the running times for your three dictionary files. Include your three graphs (one for each dictionary type) *using the data you collected and reason about the curves you see.*




Hashtable
15000
12500
10000
7500
5000
20000
40000
60000
80000
100000
Time duration
Number of words inserted


MWT
15000
12500
10000
7500
5000
20000
40000
60000
80000
100000
Time duration
Number of words inserted



I run my program on the dictionary “shuffled_freq_dict.txt” and set min_size equal to 5000, and iteratively increment the number of words inserted by 5000 for 20 times. I take 500 times of experiments for every time duration estimation and take the average of them. The graph shows that the runtime of find operation of BST is incremented with the amount of nodes (the number of words inserted), while the runtime for the other two data structures, Hashtable and Muti-way Trie are nearly constant. I will reason these shapes of curves with their analytical runtime more precisely in the question below.

Then, specifically answer the following question:

1. In class we saw that a Hashtable has expected case time to find of O(1), a BST worst case O(logN) and a MWT worst case O(K), where K is the length of the longest string. We didn't look at the running time of the TST, though the book mentions that its average case time to find is O(logN) (and worst case O(N)). Are your empirical results consistent with these analytical running time expectations? If yes, justify how by making reference to your graphs. If not, explain why not and also explain why you think you did not get the results you expected (also referencing your graphs).

Solution to Q1:
Basically consistent with the analytical running time expectation.
The runtime curve of BST is incrementing with the number of insertions, which match with the formula of O(logN), that logarithmic growth with the number of the word in the data structure.
However it has some vibration, which might caused by the running status of CPU.

For the hashtable, the runtime for a find operation in hashtable is constant O(1), more specifically, the length of the word, O(L). So it is steady and barely change with the amount of insertions.
The runtime curve of MWT is steady because the runtime is only related to the largest number of letters in a phrase, O(L). With the increase of the amount of word inserted, the possibility of a long phrase was inserted is increasing. Thus there are some step rises in the curve.

Benchmark Hash function:

1. Describe how each hash function works, and cite the source where you found this information. *Your description of how the hash function works should be in your own words*.

   Sol:
   In my implementation, there are two different hash functions. The first one is the hash function that Java use to hash its string. I found it at
   http://stackoverflow.com/questions/299304/why-does-javas-hashcode-in-string-use-31-as-a-multiplier
   The hashcode in Java String object is computed by the formula s[0]*31^(n-1) + s[1]*31^(n-2) + ... + s[n-1]. The reason why it choose 31, a specific number to be the multiplier is that 31 is an odd prime. Because multiplied by two is same as shifting left. If it were even and the multiplication overflowed, the original bit information will be lost. Also, as mentioned in lecture, the multiplier is better to be a prime. Moreover, we chose 31 because the multiplication can be replaced by a shift and a subtraction for better perfomance as 31 * i == (i << 5) - i.

   The second hash function is djb2. The source where I found it is
   http://stackoverflow.com/questions/7666509/hash-function-for-string
   this algorithm (k=33) was first reported by dan bernstein many years ago in comp.lang.c. another version of this algorithm (now favored by bernstein) uses xor: hash(i) = hash(i - 1) * 33 ^ str[i]; the magic of number 33 (why it works better than many other constants, prime or not) has never been adequately explained.

2. Describe how you verified the correctness of each hash function's implementation. Describe at least 3 test cases you used, what value you expected for each hash function on each test case, and the process you used to verify that the functions gave this desired output.

   Sol:

   1. “tom” with tablesize 2000
      hashfunction 1 1026
      hashfunction 2 1093

      For example, the string ”tom” ‘s hashcode 1 will be (116*31^2+111*31^1+109*31^0)%2000 = 1026. And it is matched with my expected output.

{[(5381*33+116)*33+111]*33+109}/2000 = 1093
For the low-number-of-bit number, it is easy to compute by hand. However, for the long string, I use Matlab as a reliable tool to verify my result.

2. “segmentation fault” with tablesize 2000
   hashfunction 1 950
   hashfunction 2 1963

3. “final report” with tablesize 1000
   hashfunction 1 603
   hashfunction 2 875

3. Run your benchhash program multiple times with different data and include a table that summarizes the results of several runs of each hash function. Format the output nicely--don't just copy and paste from your program's output.

Sol:
Using freq1.txt
Printing the statistics for hashFunction1 with hash table size 2000

#hits	#slots receiving the #hits
0	1215
1	610
2	142
3	27
4	5
5	1

The average number of steps for a successful search for hashfunction 1 would be 1.263
The worst case steps that would be needed to find a word is 5

Printing the statistics for hashFunction2 with hash table size 2000

#hits	slots receiving the #hits
0	1203
1	628
2	138
3	29
4	1
5	1

The average number of steps for a successful search for hashfunction 2 would be 1.241
The worst case steps that would be needed to find a word is 5

Using freq2.txt
Printing the statistics for hashFunction1 with hash table size 4000

#hits	#slots receiving the #hits
0	2416
1	1235

```
2    289
3    54
4    5
5    1
The average number of steps for a successful search for hashfunction 1 would be 1.2455
The worst case steps that would be needed to find a word is 5

Printing the statistics for hashFunction2 with hash table size 4000
#hits    slots receiving the #hits
0    2443
1    1179
2    321
3    49
4    8
5    0
The average number of steps for a successful search for hashfunction 2 would be 1.258
The worst case steps that would be needed to find a word is 5


Using freq3.txt
Printing the statistics for hashFunction1 with hash table size 3000
#hits      #slots receiving the #hits
0          1803
1          941
2          216
3          34
4          5
5          1
The average number of steps for a successful search for hashfunction 1 would be 1.23867
The worst case steps that would be needed to find a word is 5

Printing the statistics for hashFunction2 with hash table size 3000
#hits      slots receiving the #hits
0          1827
1          901
2          230
3          31
4          9
5          2
The average number of steps for a successful search for hashfunction 2 would be 1.26467
The worst case steps that would be needed to find a word is 5
```

4. Comment on which hash function is better, and why, based on your output. Comment on whether this matched your expectation or not.

Based on the output of my program, using dictionary “freq1/2/3.txt” the average number of steps for a successful search for hashfunction 1&2 would be 1.263 and 1.241, 1.2455 and 1.258,1.23867 and 1.26467, respectively. The number of worst case steps are same. Apparently, the second hash function is slightly better than the first one. This matched with my expectation. The reason is as below:

1. The second function has a specific base 5381, which is a prime number. And every time this base was multiplied by 33(another prime number) and add on to ascii value of the letter. This make sure that every letter has a nearly unique contribute to the whole string, which reduce the probability for collision.
2. Why Dan Bernstein(the inventor of this algorithm) chose 33 is still mysterious. But a lot of experiments has shown that this algorithm beat the Murmur hash, FNV variants hashes and many others,
3. However, both of the hash function are pretty good, according to their slightly difference in average time. They are easy to understand and implement. They both ensure that every bit will affect the result, but differ in the way that the bits were engaged in computation. Most important, they are both fast. This is an advantage to become a hash function for practical use. We cannot achieve perfect hashing in practical work because that would sacrifice large amount of computation and storage resources.

5. Read the “Notes for FinalReport.pdf” to make sure you followed all the guidelines.