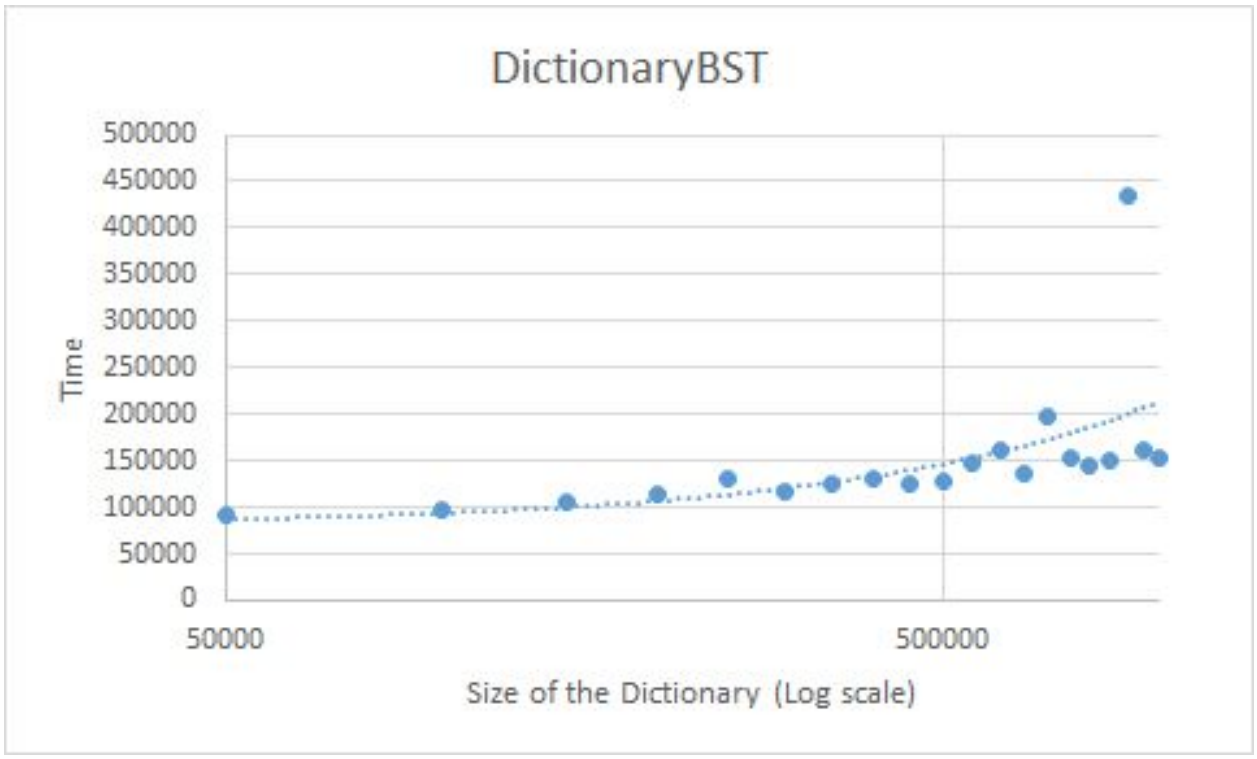
PA2 Final Report
Dion Chung
Bhanu Renukuntla
10/24/2016

BenchDict Write-Up

Here are the graphs of Time Vs Dictionary size for various dictionaries. Time is the average time taken to perform find operation the dictionary (averaged over 5000 iterations).

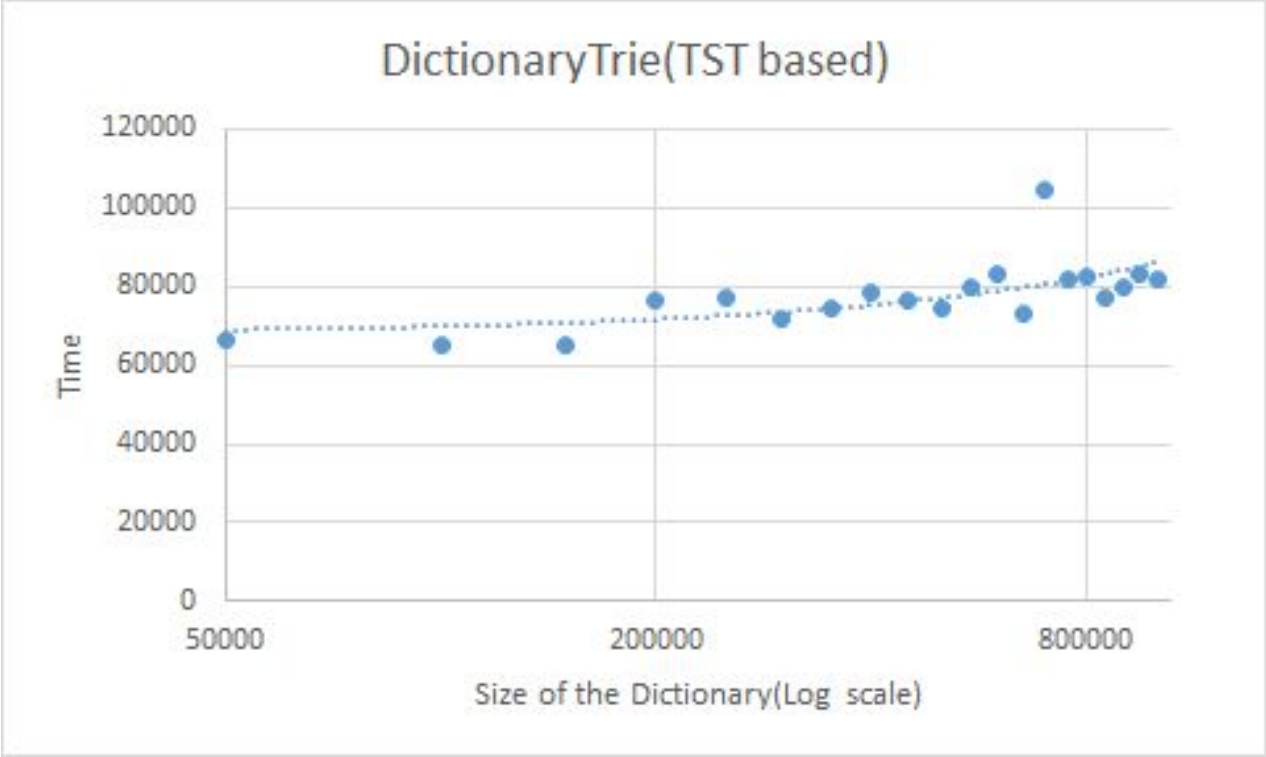
- We expect that time for DictionaryBST and DictionaryTrie (TST based) to be linear in log(N). But here is it slightly curved up because of the outlier.
- We expect time for DictionaryHashtable to be constant but it is slightly increasing.

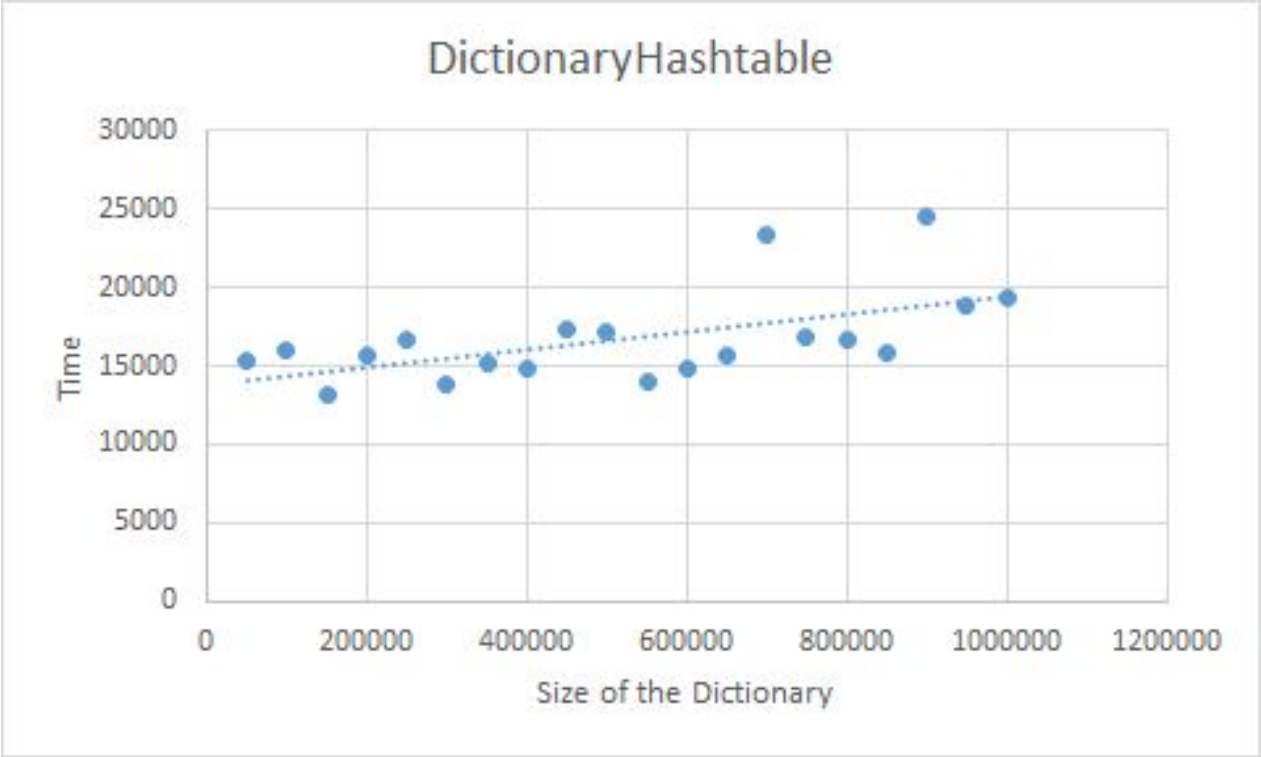
These results are quite surprising and we are not really sure why there is an upwards trend. We guess that this has to do with the min_size and step_size values chosen. Here min_size = 50,000 and step_size = 50,000. This could also be because of the machine. May be at higher values of N the machine is bit slower as compared to the lower values of N



In our other experiments, when step_size was less compared to min_size the results seemed random and the upward trend wasn't there. High values of min_size and step_size were chosen to bring out the asymptotic behaviour of time with N.

- We also note that since TST's worst case and average case values differ and hence we notice that TST has higher variance in values as compared to BST. (excluding the outlier)





BenchHash Write-Up

4.4.1 - First hashing algorithm comes from Virgina Tech's hashing tutorial (http://research.cs.vt.edu/AVresearch/hashing/strings.php). At the core, it is a character by character conversion to ASCII character code and summation. Building off that, the characters are processed in 4 character chunks and that each character scaled by a factor of 256. Characters are processed in chunks of 4 to better differentiate character order. As an example calculation, let f() take a character and return the ASCII code, given a seven character (c) string, the hash value is as follows:

$$f(c_1) * 1 + f(c_2) * 256 + f(c_3) * 256^2 + f(c_4) * 256^3 + f(c_5) * 1 + f(c_6) * 256 + f(c_7) * 256^2$$

The second hash function comes from York University in Toronto, originally written by Dan Bernstein (http://www.cse.yorku.ca/~oz/hash.html). This function is a modified geometric series in that each subsequent character ASCII value is modified by the previous value. The beginning hash value is 5381 and the magic number 33 is used as the ratio which proves to be better than other numbers, even primes. Let f() take a character and return the ASCII code, given a seven character (c) string, the hash value is as follows:

$((((((5381 * 33 + f(c_1)) * 33 + f(c_2)) * 33 + f(c_3)) * 33 + f(c_4)) * 33 + f(c_5)) * 33 + f(c_6)) * 33 + f(c_7)$

4.4.2 - We verified the logic by performing the hashing by hand to arrive at the following:

String	Hash Function 1	Hash Function 2
aaa	6381921	193485928
car	7496035	193488123
abcde	1684234950	210706217108

Calculations for “abcde” are shown here for demonstration.

Let $H_1()$ be the first hash function and f() convert a character to ASCII character code, then,

$$H_1(\text{“abcde”}) = f(\text{‘a’}) * 1 + f(\text{‘b’}) * 256 + f(\text{‘c’}) * 256^2 + f(\text{‘d’}) * 256^3 + f(\text{‘e’}) * 1$$
$$H_1(\text{“abcde”}) = 97 * 1 + 98 * 256 + 99 * 256^2 + 100 * 256^3 + 101 * 1$$
$$H_1(\text{“abcde”}) = 97 + 25088 + 6488064 + 1677721600 + 101$$
$$H_1(\text{“abcde”}) = 1684234950$$

Let $H_2()$ be the second hash function and f() convert a character to ASCII character code, then,

$$H_2(\text{“abcde”}) = ((((5381 * 33 + f(\text{‘a’})) * 33 + f(\text{‘b’})) * 33 + f(\text{‘c’})) * 33 + f(\text{‘d’})) * 33 + f(\text{‘e’})$$
$$H_2(\text{“abcde”}) = ((((5381 * 33 + 97) * 33 + 98) * 33 + 99) * 33 + 100) * 33 + 101$$
$$H_2(\text{“abcde”}) = ((((177670) * 33 + 98) * 33 + 99) * 33 + 100) * 33 + 101$$
$$H_2(\text{“abcde”}) = (((5863208) * 33 + 99) * 33 + 100) * 33 + 101$$
$$H_2(\text{“abcde”}) = ((193485963) * 33 + 100) * 33 + 101$$
$$H_2(\text{“abcde”}) = (6385036879) * 33 + 101$$
$$H_2(\text{“abcde”}) = 210706217108$$

By printing the hash values in the function, it was easy to verify correctness. With both functions, the hash value is modded (%) by the size of the hash table to get the index.

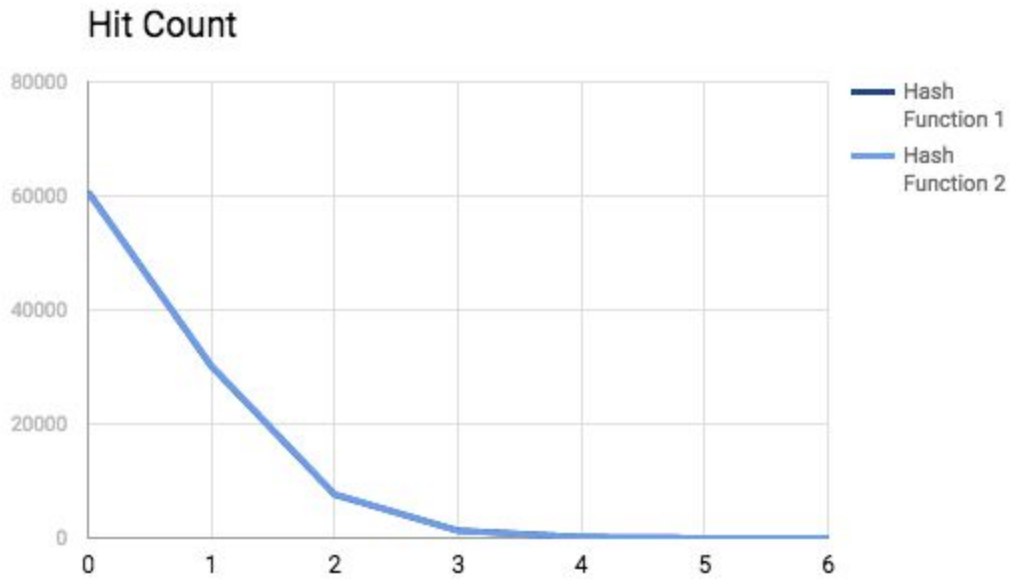
4.4.3 - Results for table size 100000 from shuffled dictionary

Hits	Hash Function 1	Hash Function 2
0	60663	60641
1	30282	30402
2	7627	7465

3	1261	1317
4	155	160
5	11	13
6	1	2
Average steps in search	1.2493	1.25072
Max steps in search	6	6

Results for table size 100000 from sorted dictionary

Hits	Hash Function 1	Hash Function 2
0	60760	60659
1	30157	30253
2	7608	7695
3	1294	1230
4	163	151
5	15	9
6	3	3
Average steps in search	1.25326	1.24852
Max steps in search	6	6



Overall both functions trend similarly, in fact in successive runs with larger sets, the performance for the two functions are comparable.

Average steps to search results with insertions from shuffled dictionary

Table Size	Hash Function 1	Hash Function 2
1000	1.254	1.23
100000	1.2493	1.25072
200000	1.25336	1.24724
500000	1.24812	1.24909
1000000	1.25319	1.25034

On the other hand the ordering of the words and by extension the variability of the characters in the words, we see that randomized words tends to be more performant.

Average steps to search results with different insertion source for Hash Function 1

Table Size	Shuffled Dictionary	Sorted Dictionary
1000	1.254	1.188

100000	1.2493	1.25326
200000	1.25336	1.26522
500000	1.24812	1.25325
1000000	1.25319	1.2678

4.4.4 - Both hash functions perform similarly with neither being consistently better than the other, particularly when measured by the average steps to perform a successful search. We had expected that the second hash function, the one that uses a geometric series to perform better as it was not a modified summation of ASCII code values that is the trivial example used in class. With the choice of 33 as the ratio, function 2 also seemed to have more research baked in. In the end this was not the case and both hash functions perform equally well.