Kesong Xie

Advanced Data Structure, CSE 100

PA2

Running Time Analysis

Ternary Search Tree:

min_size: 10000

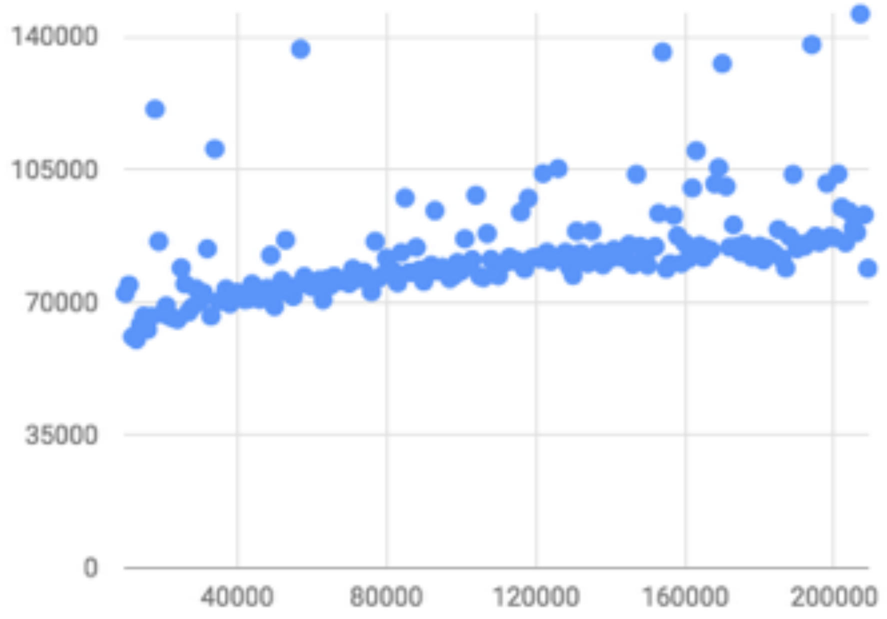
step_size: 1000

num_iterations: 200

dictfile: shuffled_freq_dict.txt

average for each find: 10

The graph below is the plot for finding 100 elements:



From the graph, we can tell that the find function running time is O(logN) for ternary search tree, because with the size of the elements grows, the graph above presents a logarithmic shape.

DictionaryHashTable:

min_size: 10000

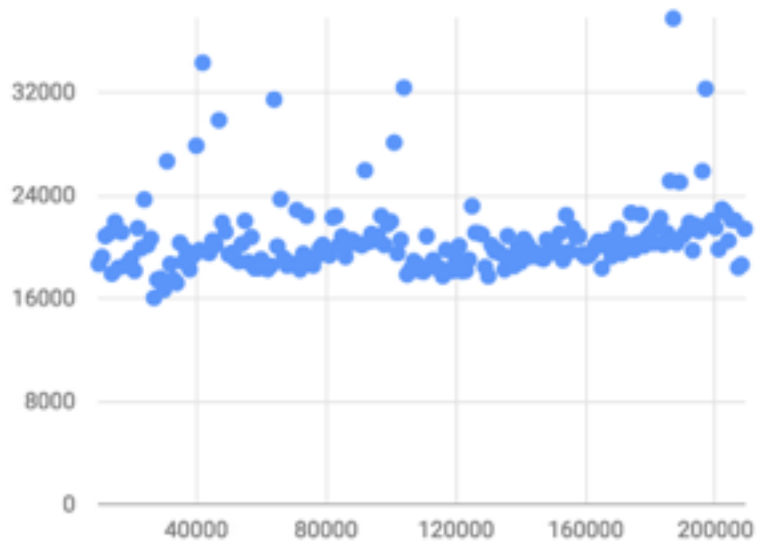
step_size: 1000

num_iterations: 200

dictfile: shuffled_freq_dict.txt

average for each find: 10

The graph below is the plot for finding 100 elements:



From the graph, we can tell that the find function running time is O(1) for Dictionary with HashTable, because with the size of the elements grows, the graph above presents a horizontal line.

DictionaryBST:

min_size: 10000

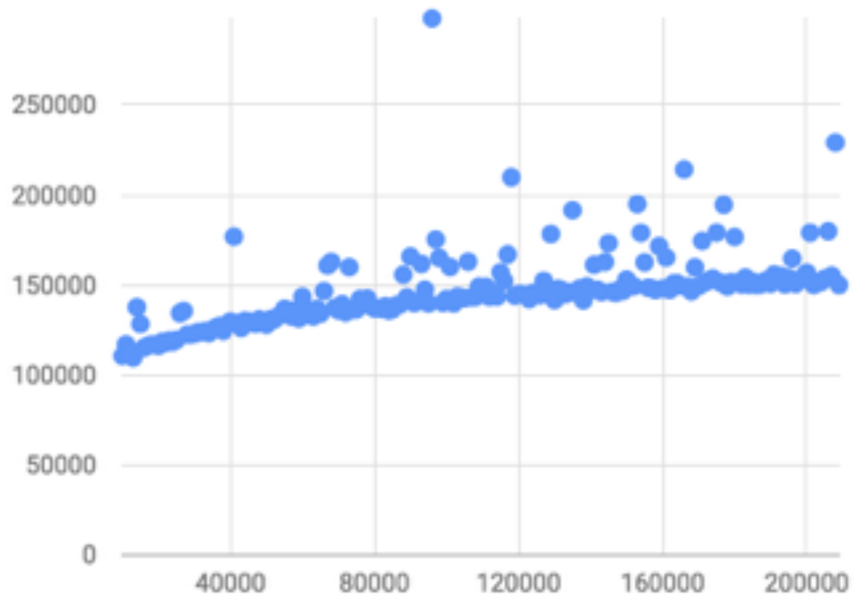
step_size: 1000

num_iterations: 200

dictfile: shuffled_freq_dict.txt

average for each find: 10

The graph below is the plot for finding 100 elements:



From the graph, we can tell that the find function running time is O(logN) for Dictionary with BST, because with the size of the elements grows, the graph above presents a logarithmic shape

Performance of different hash functions for strings

Hash Function One:

Reference

http://stackoverflow.com/questions/2624192/good-hash-function-for-strings

Source Code

```
unsigned int hashOne(std::string key, unsigned int tableSize){
    int hash = 7;
    for (int i = 0; i < key.length(); i++) {
        hash = hash * 31 + key[i];
    }
    return hash % tableSize;
}
```

How it works

The hashOne function hash the string key by iterating through the entire string, and for each iteration, it computes the hash value by summing the sub-hash using the previous hash value times 31 plus the ASCII code for the character in that given iteration. Notice the both number 7 and 31 are prime number for improve the uniqueness of the hash value for any given character in the string. After it exists the loop, the hashOne function simply modulo the tableSize to guarantee that the hash value returns from the hashOne will fall between 0 to tableSize - 1

Calculate test case:

Test case strings used(Table Size 2000):
“hel”, “ehl”, “leh”

Input string: “**hel**”
expected output: 1720
hash = (7 * 31 + 104 + (7 * 31 + 104)*31 + 101 + ((7 * 31 + 104)*31 + 101)*31 + 108)%2000 = 1720

Input string: “**ehl**”

expected output: 930

hash = (7 * 31 + 101 + (7 * 31 + 101)*31 + 104 + ((7 * 31 + 101)*31 + 104)*31 + 108)%2000 = 930

Input string: “**leh**”

expected output: 1560

hash = (7 * 31 + 108 + (7 * 31 + 108)*31 + 101 + ((7 * 31 + 108)*31 + 101)*31 + 104)%2000 = 1560

Hash Function One Performance Report

Run 1(Table size 20000):

#hits	#slots receiving that #hits
0	12066
1	6148
2	1541
3	213
4	29
5	3

The average number of steps for a successful search would be: 1.2384
The worst case steps that would be needed to find a word is: 5

Run 2(Table size 20000):

#hits	#slots receiving that #hits
0	12079
1	6123
2	1552
3	214
4	29
5	3

The average number of steps for a successful search would be: 1.2398
The worst case steps that would be needed to find a word is: 5

Hash Function Two:

Reference

https://stepik.org/lesson/Hash-Functions-31712/step/3?course=Data-Structures-(CSE-100)&unit=11896

Source Code

```
unsigned int hashTwo(std::string key, unsigned int tableSize){
    unsigned int val = 0;
    for(int i = 0; i < key.length(); i++) {
        val += (unsigned int)(key[i]); // cast each character of key
to unsigned int
    }
    return val % tableSize;
}
```

How it works

The hashTwo function hash the string key by iterating through the entire string,, it computes the hash value by summing the ASCII code for each characters in the string and modulo the tableSize to guarantee that the hash value returns from the hashOne will fall between 0 to tableSize - 1

Calculate test case:

Test case strings used(Table Size 2000):

“hel”, “ehl”, “leh”

Input string: “**hel**”

expected output: 313

hash = (104 + 101 + 108) % 2000 = 313

Input string: “**ehl**”

expected output: 313

hash = (101 + 104 + 108) % 2000 = 313

Input string: “**leh**”

expected output: 313

hash = (108 + 101 + 104) % 2000 = 313

Hash Function Two Performance Report

Run 1(Table size 20000):

#hits	#slots receiving that #hits
0	17768
1	499
2	316
3	251
4	211

5	173
6	189
7	182
8	131
9	92
10	76
11	46
12	36
13	11
14	9
15	5
16	4
17	1

The average number of steps for a successful search would be: 3.8846
The worst case steps that would be needed to find a word is: 17

Run 2(Table size 20000):

#hits	#slots receiving that #hits
0	17774
1	493
2	319
3	245
4	214
5	174

6	185
7	183
8	130
9	94
10	80
11	43
12	36
13	10
14	10
15	5
16	4
17	1

The average number of steps for a successful search would be: 3.8892
The worst case steps that would be needed to find a word is: 17

Comments and conclusion:

Overall, the hashOne function performs better than hashTwo function, from which we can tell that there are about 12000 remained empty(out of 20000) for hashing 10000 words into the table using hashOne function, while there are almost 18000 remained empty(out of 20000) when using hashTwo function. From the average number of steps for a successful search, the hashOne is about 1.2 steps, while hashTwo requires about 3.9 steps. Further, on the worst case for finding a word, the hashOne requires 5 step, while the hashTwo function requires 17 steps.