BLG 335E – Analysis of Algorithm I, Fall 2017

Project 3 Report

Assignment Date: 17 Nov 2017 Friday

Due Date: 01 Dec 2017 Friday– 23:59

Kadir Emre Oto

150140032

Code Analysis

In this assignment, we were expected to implement the hashing and probing functions for our dictionary approach, two data structures, dictionary (hash table) and linked list. I used a pair<key, value> array for dictionary implementation and a pair<key, value> vector for linked list implementation. Because a vector can be used as a linked list, all consecutive elements of vector are linked each other, and for this assignment vector runs faster.

All codes were written in **kod.cpp** file, and compilation and running commands are given below. Additionally, all input files ("ds-set-input.txt" and "ds-set-lookup.txt") should be in the same directory with the program. After the execution, output files ("output-dict.txt" and "output-list.txt") will be appeared in the same directory.

Compilation Command: g++ kod.cpp -o kod -O2 -std=c++11
Running Command: ./kod

a. (10 Points) Provide two graphs or tables (for the block insertion and block lookup procedures) comparing the runtime on your dictionary and list implementations.

Running Times	Block Insertion	Block Lookup
Dictionary	0.06 s	0.10 s
List	0.05 s	3.99 s

b. (10 Points) Briefly analyze the computational performances of your data structure implementations in relation to the runtimes you measured. Is there a noticeable difference between the two data structures for either procedure? Are the runtimes you measured in line with the corresponding average-case time complexities?

According to the results (Figure 1), runtimes for insertion operation are almost the same, but there is a noticeable difference between hash table and linked list implementations for lookup operation. The results show us hash map runs 40 times faster than the linked list for looking operation. The average complexities for hash table and linked list implementation for looking up is O(1) and O(n) respectively. So the runtimes you measured are in line with the corresponding average-case time complexities.

```
Emre:Homework3 KEO$ cc
    DICTIONARY
    Insertion finished after 0.06 seconds

    Average number of collisions (first 1,000)    | 0.002
    Average number of collisions (first 10,000)   | 0.036
    Average number of collisions (first 100,000)  | 0.800
    Average number of collisions (overall)        | 9.969

    Lookup finished after 0.10 seconds

    LIST
    Insertion finished after 0.05 seconds
    Lookup finished after 3.99 seconds
```

(Figure 1. Output of the program)

c. (10 Points) How does the average number of collisions differ as more items are inserted into your dictionary? Does it stay roughly the same, or increase linearly, or exponentially? Briefly discuss the reason for this observed behavior of your hash table implementation.

The average number of collisions increases exponentially because number of empty locations in array are decreases, and probability of finding free place for the elements that will be inserted reduces exponentially at every insertion operation. Our hashing function uses the golden ratio for distribution the numbers more naturally, and the quadratic probing function prevents the bucket clustering.

d. (10 Points) What is the worst case for looking up a key in your dictionary? What is the corresponding worst-case time complexity in O notation? In the worst case, which part of the lookup process becomes slow enough to dominate the complexity?

If our hashing function produces the same values continuously for all keys, probing function will call n times for the added nth element. So probing function will be the slowest process for the worst case scenario.