IE 172 Laboratory 5: Hashing Out Hash Tables

Dr. T.K. Ralphs

Due March 10, 2016

1 Laboratory Description and Procedures

1.1 Learning Objectives

You should be able to do the following after completing this laboratory.

1. Understand the use of hash tables.
2. Understand how to implement a hash table using both chaining and open addressing.
3. Understand the time-space tradeoff.
4. Develop an appreciation for the dramatic changes in performance that can result from subtle changes in implementation.
5. Develop an ability to analyze the tradeoffs between various implementations of the same data structure.
6. Understand when to use the various implementations of hash tables.

1.2 Key Words

You should be able to define the following key words after completing this laboratory.

1. Hash table
2. Open addressing
3. Chaining
4. Time-space tradeoff

1.3 Scenario

Dictionaries are an extremely important part of the Python language. Not only is the use of dictionaries in user code more common than in other languages, but the Python language itself uses dictionaries extensively to store and pass information about the program being executed and the internal state of the interpreter. For example, Python stores the arguments to functions as a dictionary. Even classes themselves are stored as dictionaries, so that the values of attributes can be looked up quickly. Dictionaries in Python are implemented to be extremely efficient across a wide range of use cases, some of them quite different than the usual ones. For example, Python dictionaries are meant to be efficient even if the keys are sequential integers. Typically, one would

use a list in this case, but because dictionaries are used so frequently in Python, it is necessary to anticipate the use of a dictionary instead.

In this lab, we will experiment with the use of dictionaries in a variety of use cases. The scenario you can envision initially is the following. You are a linguist doing analysis of various classic books. You have these books stored in plain text form and you want do things like determine how frequently certain words appear, how many unique words there are, and other such things. For this purpose, you decide to use a hash table. This is the scenario we'll actually enact in this lab.

1.4 Design and Analysis

For hash tables, the running time is dominated by comparison operations, so in this lab, we will study both the number of comparisons needed to perform various operations on a hash table and the actual running time for insertion and lookup, using several implementations. Hash tables illustrate several of the principles that are central to our study of algorithms, so we will put some effort into analyzing them.

The first principle that we will study in this lab is that of the time-space tradeoff. Hash tables present a perfect illustration of this important principle. One of the parameters of a hash table is its size. The size of the table, along with the expected number of elements to be stored, determine the table's *load factor*, which is just the ratio of these two numbers. No matter what the underlying implementation, the load factor of the table is positively correlated with the number of comparisons required for basic operations. In other words, the smaller the load factor, the fewer comparisons are required. If there were no limit on the size of the table, the load factor could always be made small and all operations could be implemented in constant time. In reality, however, we must decide how much memory needs to be dedicated to storage of the table in order to achieve desired performance. This is the time-space tradeoff.

Although this tradeoff exists for all implementations of hash tables, the way in which the performance varies as a function of the load factor (denoted α from here on) differs significantly from one implementation to the next. For instance, the number of comparisons for a search miss varies linearly with α in a chaining implementation, whereas it is inversely proportional to $1 - \alpha$ in a linear probing implementation.

Performance also differs significantly with other factors. One of the most important factors to consider is how often items will have to be deleted from the table. Certain implementations are only appropriate if very few deletions will occur. We must also consider how accurately we can estimate the number of items to be inserted into the table.

In this lab, we will be comparing several different implementations of hash tables in order to determine how they perform in various situations. The two main choices in implementing a hash table are the hash function and the method of resolving collisions. We will use collision resolution by either *chaining* or *open addressing* and the hash functions as described in Lectures 9 and 10. In terms of the time-space tradeoff, it is unclear which method is superior. Chaining is efficient with load factors bigger than one, so we can expect all table slots to be used. However, we must store an extra pointer with each item. With open addressing, we must have a load factor below one, which means we are forced to reserve memory locations for storing pointers that will not actually be used. To analyze this tradeoff, we must determine the performance for load factors at which the two algorithms require the same amount of memory. As in previous labs, we will perform both empirical and theoretical studies to determine these tradeoffs.

The analysis is quite different in the presence of deletions. In this case, the particular method of open addressing is important. For instance, with linear probing, deletion can be handled efficiently, whereas with double hashing, it is more difficult. In the analysis section of this lab, we will explore

these tradeoffs as well.

1.5 Program Specifications

You have been provided with a full implementation of a hash table class implemented using both chaining and linear probing. You have also been given a client program that will generate random sequences of strings to insert into the table in order to test the performance of the two implementations. The comments in the code should help you understand how to use it. Your job will be to modify the implementations in various ways, as described in the Programming and Analysis section below in order to explore the various tradeoffs discussed above.

1.5.1 Algorithms

The algorithms to be implemented in this lab are various hash functions and methods of resolving collisions supporting the operations on a hash table specified by the interface in `hashTable.h`.

1.5.2 Data Structures

The basic data structure required for this laboratory is a hash table.

2 Laboratory Test Files

The main files for this laboratory are `dictionary_oa.py` (the open addressing implementation), `dictionary_chain.py` (the chaining implementation). In addition, you should choose a favorite classic book that is out of copyright and find a plain text copy of it on-line. Project Gutenberg is a good place to browse. The book should have at least 10K words.

3 Laboratory Assignments

3.1 Programming and Analysis (30 points)

1. (10 points) Count the number of occurences of each word in the text by inserting them into a dictionary. The value for each key should be the number of occurences. Be sure you strip punctuation and other characters away before insertion so that you truly get just the words themselves. Also, make sure you ignore case. Sort the list of words by frequency of occurence in the text and print out the sorted list or words along with their frequencies.

2. (10 points) Count the number of comparisons required to insert all the words in your text into the dictionary for both open addressing and chaining. Make a graph showing the number of comparisons required as a function of the load factor for each implementation. What does this say about the time-space tradeoff for each?

3. (10 points) Make a similar graph of the time it takes to insert all the words as a function of load factor. What does this say about the time-space tradeoff for each?

3.2 Follow-up Questions (50 points)

1. (10 points) The main loop of the insertion sort procedure uses a linear search to scan through the sorted subarray $A[0 : j-1]$ to find the insertion point. Since the subarray is already sorted,

can we improve the worst-case running time by using binary search to find the insertion point? If so, what would be the improved worst-case running time?

2. (10 points) Would using binary search to find the insertion point in insertion sort be a good idea in practice? Why or why not? Take into account the kinds of lists on which you would typically expect to use insertion sort.

3. Consider the following scheme for sorting a list of $n - bit$ (the numbers have n digits when written in binary, though some leading digits may be zero).

   - We first insert the items into a hash table with chaining, with the hash function being the first k bits of the number (this is not a good hash function in general, but serves a purpose here).
   - Next, we sort the individual lists of items in each slot in the hash table with insertion sort.
   - Finally, we just concatenate all of these sorted lists together.

   (a) (10 points) Why does this algorithm sort correctly?

   (b) (10 points) What is the worst case running time of this algorithm and with what kinds of inputs is it realized?

   (c) (10 points) Assuming the numbers are completely random, what running time would you expect in practice? Explain.