IE 170 Laboratory 6: Hash Tables

Dr. T.K. Ralphs

Due March 13, 2006

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

You are an employee in the IT department of the telephone service provider Phones'R'Us. As part of the new enhanced caller ID service that Phones'R'Us is planning to roll out, they must implement an extremely fast reverse lookup procedure by which one can look up the name, address, and other miscellaneous information of any of any subscriber using just their phone number. This reverse lookup database will be stored on small local servers in each switching office, so it is not practical to simply place the information in an array that has one entry for each possible phone number. It is also not possible to predict what additional exchanges will be used in the future. The phone numbers themselves are not assigned randomly, but the patterns used for assignment are not predictable ahead of time. Therefore, it is decided that a hash table is needed to store the customer information for extremely fast reverse lookup. Because of the unpredictability of the telecom market, it is not known whether phone numbers will need to be deleted on a regular basis. This depends on the competitiveness of the company in a tumultuous market. You have therefore been asked to do a preliminary study of the various implementations of hash tables to determine the best course of action under various scenarios.

1.4 Design and Analysis

For hash tables, the running time is dominated by comparison operations, so in this lab, we will study the number of comparisons needed to perform various operations on a hash table 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 able.

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. In this lab, we will use modular hash functions with collision resolution by either *chaining* or *open addressing*, as described in Lectures 11 and 12. 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 insertions and deletions in order to test the performance of the two implementations. The comments in the code should help you understand who 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 files for this laboratory are in the zip archive `Lab6.zip` available on the course Web site. The archive will unpack into a directory called `Lab6` with a `shell` subdirectory. The shell directory contains the files

1. `main.cpp`: the client program used for testing.
2. `hashTable.h`: the hash table interface.
3. `hashOpen.cpp`: the implementation file for linear probing.
4. `hashChain.cpp`: the implementation file for chaining.
5. `hashItem.h`: A class for storing the items.

Note that ordinarily, we would have a different item class for each implementation (possibly derived from a single base class using C++ inheritance) in order to take advantage of the different requirements of chaining and open addressing. However, to keep things simple, we are using only one class here that contains all the fields required for either implementation.

3 Laboratory Assignments

3.1 Programming and Analysis (60 points)

1. (10 points) Compare the performance of chaining to linear probing when performing only insertions and searches. Keep the number of items at 1000 and vary the table size to achieve different load factors. For each implementation, create a graph of load factor versus average time for search hits and misses and display the theoretical average time on the same graph. Use load factors between .5 and 1 for linear probing and between 1 and 10 for chaining. What are your observations?

2. (10 points) Determine approximately how much smaller the load factor needs to be in linear probing to achieve the same performance as in chaining. At a similar level of performance, which one uses more memory (just take into account the pointers, as described above)?

3. (10 points) Compare chaining to linear probing when performing insertions, searches, and deletions. For each implementation, create a graph of load factor versus average time for search hits and misses, with the load factors in the same range as above. What are your observations?

4. (15 points) Try to improve upon the performance of linear probing with deletion by implementing deletion with rehashing. Chart the performance of each method with load factors in the same range as above on the same graph. Please paste the relevant snippets of code into your write-up and explain the changes that had to be made to implement your methods. What are your observations?

5. (15 points) Please paste the relevant snippets of code into your write-up and explain the changes that had to be made to implement your methods. Try to improve the performance of linear probing by implementing double hashing (this can be done with only a few lines of code). Chart the performance of each method with load factors in the same range as above on the same graph. Please paste the relevant snippets of code into your write-up and explain the changes that had to be made to implement your methods. What are your observations? Note that you cannot use deletion with rehashing with insertion by double hashing.

3.2 Follow-up Questions (40 points)

1. (10 points) 10-1

2. (10 points) 10.4-3

3. (10 points) CLRS 11.2-2. Do the same for linear probing and analyze the time for search hits and misses with each implementation on this small example.

4. (10 points) CLRS 11.2-3