

# ContactBookPro: Data Structures Final Project

December 13, 2008

Mike Sullivan  
Natalie Dehen  
Alex Bess

## 0 Abstract

Our project is an interactive contact book. It uses Qt4 to implement the user interface. It loads a text file into the main interface, from which users can add, search for, and delete entries at will. The main purpose of the project was to demonstrate the differences between a hash table and a multimap. Our goal was to create two different working contact books, one using a multimap and one using a hash table, and to compare the different runtimes. We also ended up writing our own hash function to overload the existing one and try to see if we could improve performance even further.

## 1 Keywords

ContactBook

Main class used for declaring all the widgets.

ContactBookPro

Our name for our contact book application

GUI

Graphical user interface.

Hash Table

Data structure used to associate keys with values. This is an unordered associative container. For one of our contact books we employed a hash table to store and look up the contacts.

Map

Data structure used to associate keys with values. This is an ordered associative container

Multimap

Modified Map, in which key may be associated with multiple values. For one of our contact books we implemented a multimap to store and look up the contacts.

QHash

Hash table class provided by Qt.

QMultiHash

This is the variation of QHash we used for our project.

QLabel

Widget that displays text or an image: we used it to label the other widgets displayed on the user interface.

QLineEdit

Widget that creates a single editable line of text, used in our application to add contact names or delete contact values.

QListWidget

Provides an item-based list widget, which we used to display contacts upon adding or searching.

QMultiMap

Multimap class provided by Qt.

QPushButton

Widget that provides a command button. Our command buttons included add, search, delete, and cancel.

Qt

The widget toolkit we used to implement our contact books.

**QTextEdit**  
Widget that provides a single-page text editor. We used this widget for entering the contact information of a new entry.

**STL**  
The standard template library.

**uint QHash(const QString &key)**  
Hash function provided by Qt. We overloaded this in our hash implementation.

**Widget**  
An element of a GUI that displays information changeable by the user. We primarily used the QLineEdit widget in our contact books.

**Widget Toolkit**  
Set of widgets used for developing GUI programs.

## **2 Introduction**

Our project is an interactive contact book. It uses Qt4 to implement the user interface. We learned about Qt from Sam Banina (Reference 1). Our project loads a text file into the main interface, from which users can add, search for, and delete entries at will. The main purpose of the project was to demonstrate the differences between a hash table and a multimap. Our goal was to create two different working contact books, one using a multimap and one using a hash table, and to compare the different runtimes. We also ended up writing our own hash function to overload the existing one and try to see if we could improve performance even further.

## **3 ContactBookPro**

### **3.0 Functionality**

The ContactBookPro has very simple functionality. There is a search function, and add function, and a delete function. The delete function will delete a single phone number (entered by the user) from a given name, or will delete the entire entry if only one phone number is present. The search function works as an exact-match search. The add function is straightforward. Enter a name and phone number, and it will insert the new contact into the database.

The reason the functions in this project are so simplistic is that the main point of our project was the comparison of the efficiencies of two different data structures. Because we used a multimap (ordered) and a hash table (unordered) there would be differences in the running times of the various functions.

### **3.1 Multimap**

To create our first contact book, the main data structure we implemented was the multimap function provided by Qt. We found this on trolltech.com (Reference 2).

### **3.2 Hash Table**

To create our second address book, the main data structure we implemented was the hash function provided by Qt. (Found at Reference 2.)

### 3.3 Overloaded Hash Function

We also decided to implement our own hash function to use with QHash. We wanted to see if we could improve performance by overloading the QHash default function. We researched common hash functions (see Reference 3), and found that a common way to hash strings is to use the ASCII or Unicode values of each character in the key. One method consists of doing the following for each character: raise 26 to a power (corresponding to the position of the character in the string) and multiply by the Unicode value for the character. The hash value is obtained by adding all of these terms and taking the sum modulus a large number. We implemented a hash function of this form.

We experimented with the function in order to obtain the best performance. This involved determining the optimal base to take to a power, and an optimal large number to use for the modulus operation. We found that, in the case of our application, taking 26 to a power provides no improvement over taking 2 to a power. Therefore we decided that our function would use 2 as the base. Likewise, we experimented with which large number is best. We found that as you increase the value, performance increases up to a certain point. Performance peaked at about 10,000, so we decided to use this number. Our final function takes  $2^i$  for each character, where  $i$  is the character's position in the string, and multiplies it by the character's Unicode value. Then it adds the values obtained for each character, and calculates the sum mod 10,000. See Appendix 8.1 for the code.

## 4 Comparisons

In order to compare the performance of the hash table versus the multimap, we used a very large database of contact information (over 3 million entries). This was necessary in order to obtain a detectable difference in execution time.

We loaded all of the contact information from a file into the multimap or hash table. This operation uses the insert function. To obtain the performance difference for the insert function, we timed the file loading process. The multimap implementation took approximately 9.5 seconds on average to insert all 3 million contacts. The hash table implementation, on the other hand, took an average of 5 seconds to do the same task.

The delete operation dropped from approximately 0.1 seconds for the multimap to approximately 0.3 seconds for the hash table. This is a speedup of 70%!

The search operation did not produce a noticeable difference when executed once (the same issue we had with insert). We added code that would search for a list of elements in the Contact Book immediately after all of the contacts were inserted. This code allowed us to detect a difference in execution time. The multimap took approximately 0.15 seconds, to perform the multiple search, while the hash table took approximately 0.05 seconds. This is a speedup of 66%!

Finally, we decided to implement our own hash function. We wanted to see if we could improve performance by overloading the QHash default function. The details of our hash function are discussed above. Replacing the default hash function with our own hash function resulted in a decrease in performance. We expected this because the default function used is one that is optimized for string hashing. In order to compare performance, we used a smaller file size to accommodate our new hash function's much slower execution. For the smaller file, the implementation using our new hash function took 0.4 seconds to insert all of the elements, and 1.55 seconds to do a multiple search. This is much worse than the 0.07 and 0.05 seconds, respectively, for the default hash function.

## **5 Challenges**

We began our project by creating a text-based user interface using an STL multimap. We faced challenges identifying the correct functions to use in order to obtain the desired outcome. Once we studied the behavior of an STL multimap, we were able to implement the desired functionality. However, when we attempted to transfer our code to a Qt interface, we encountered a problem. We were unable to link the multimap data structure to the Qt widgets. At this point we decided to abandon our STL-based code, and instead use the Qt data structure QMap. QMap had different functions than the STL multimap, but we were able to replace the STL functions we used with equivalent QMap functions.

Since the goal of our project was to compare and contrast the performance of multimaps versus hash tables, we decided to use QHash as our hash table implementation. This way, we could directly compare performance because QHash and QMap are implemented with very few differences.

Another challenge we faced was that none of us had experience with Qt. We spent a lot of time familiarizing ourselves with Qt and the relevant functionality for our Contact Book. Along with this, we encountered some difficulty working with both Qt classes and STL classes. We were able to solve most of these problems by using Qt class member functions which converted data to STL datatypes.

Finally, timing each implementation of our Contact Book was a challenge in itself. We ended up writing test code to call functions repetitively. This was necessary because individually, each function executes too quickly to time.

## **6 Summary**

The purpose of this project was twofold: to create an application utilizing at least one data structure covered in class, and then to compare the merits of using another data structure for the same application. The application we chose to implement was a basic contact book. We chose this because it could utilize a database that could be converted into both data structures that we used, and because it could feature several functions that could test and compare the speed of each structure.

The data structures we chose to implement were a multimap and a hash table. We chose these two because a multimap is ordered and a hash table is unordered, and also because there are existing templates for each. The large database that we encoded into each data structure allowed for non-trivial time comparisons for each function.

We found that because multimaps are ordered and hash tables are unordered, the hash table performed better. This is because it is faster and easier to randomly search the unordered hash table than to go through the ordered multimap. See Appendix 8.2 for the algorithmic complexity comparisons.

## 7 The Future

To improve on this project, many features can be modified or added. The main improvement to make is to eliminate the need for an exact-match search. This could be accomplished by implementing a more complex search algorithm. It would also be very interesting to try to create our own hash function that matches or exceeds the performance of the default hash function. Other goals include improving the user-friendliness of the entire application overall, and the ability to modify the actual text file used in the database.

## 8 Appendices

### 8.1 Overloaded Hash Function

```
uint qHash(const QString &key)
{
    const QChar *ptr = key.begin(); // Initialize pointer to first char of key
    uint sum=0;
    int i = 1;

    while(i<=key.size())           // Run for each character in key
    {
        sum+=((2^i)*ptr->unicode()); // Add next value
        i++;                          // Increment i
    }
    sum = sum % 10000;              // Take sum modulus 10,000
    return sum;                     // Return the hash value
}
```

### 8.2 Algorithmic Complexity

	Key lookup		Insertion	
	Average	Worst case	Average	Worst case
QMap<Key, T>	$O(\log n)$	$O(\log n)$	$O(\log n)$	$O(\log n)$
QHash<Key, T>	Amort. $O(1)$	$O(n)$	Amort. $O(1)$	$O(n)$
QSet<Key>	Amort. $O(1)$	$O(n)$	Amort. $O(1)$	$O(n)$

## 9 References

1. Sam Banina's lecture on Qt
2. <http://trolltech.com/>
3. <http://www.cs.sunysb.edu/~skiena/214/lectures/lect21/lect21.html>

## 10 Biographies

### 10.1 Natalie Dehen – Junior – CS



Natalie is a playful, fun-loving person, much like your cocker spaniel. She's almost as loyal and hardworking as one, too, and thus eons better than any kind of cat. She has been known to chase squirrels on occasion, but because she is not *quite* as good as your cocker spaniel, she has never caught one. This is probably a good thing.

### 10.2 Mike Sullivan – Junior – CS



Mike is known for his soul-searching grasp of human nature and his mean sandwich-making skills. Should you ever encounter him in his natural habitat, be wary: he seems harmless, but is actually rather dangerous. He routinely swings around heavy blunt instruments and threatens people with pirate-like roars. Strategies for halting his rampages include asking for a sandwich and offering to play sousaphone-hero in duel mode.

### 10.3 Alex Bess – Senior – CS



No one knows what happened to Alex that year he spent in the wilderness, but we know for sure that he came out a changed man. Small clues lead to larger mysteries, such as his newfound interest in Computer Science, his trombone-playing skills, and his sudden fear of all things Playmobil. These mysteries will likely never be solved.