CSCI 2300 — Data Structures and Algorithms
Project 3 — Jog Phone Company

March 7, 2000

Submission Deadline: Friday, March 31 at 11:59:59 pm

Many applications require fast storage and retrieval of data of some type `T` based on *keys* of some other type `Key`. If the set of possible keys is a reasonably small range of integers then the table could simply be an array indexed by the keys. Often though the number of possible keys is much too large for an array, as for example when the keys are people's names or social security numbers. (Since social security numbers are 9 digits long, the array would have to have room for one billion entries.) So instead we must store both keys and their associated data values, in an abstract data type called a `table` or `dictionary`. When we need to retrieve the value associated with a given key k, we have to search for k among the stored keys. We want to arrange the table so that such searches can be done quickly, and we assume that it's also required that inserting or deleting (key, data) pairs must be efficient operations also. The first solution we might think of is an array or vector of (key, data) pairs, sorted based on the keys, so that keys can be looked up quickly using a binary search. (Why is a vector to be preferred over a plain array?) In order to keep the table sorted, a new entry must be made at the correct place in the table (we can't just put it at the beginning or end). In an array, such insertions take linear time (time proportional to the size of the table), which can be very slow for a large table. Thus, using an array or vector is satisfactory only if the number of insertions and deletions is very small compared to the number of lookup operations.

The STL library provides *sorted associative containers*, which are a solution to this problem. With these containers, insertion, deletion, and lookup are all fast operations—they work in time proportional to the *logarithm* of the size of the container. These containers are implemented with *balanced binary search trees*, which, as we've seen in Chapter 4 of the textbook, does support these operations with a logarithmic bound on their computing time. In this project, you are to write a couple of short programs that construct tables of (key, data) pairs. One program should use the STL associative containers, `map`. However, it's possible to do better than logarithmic time for insertion, deletion, and lookup, on average, by using a *hash table*, which is another type of associative container. Hash tables have average case *constant* time bounds on insertion, deletion, and lookup. Their disadvantage is that their worst case behavior can be very bad compared to balanced binary search trees—linear time instead of logarithmic. Achieving the optimal constant time behavior requires careful choice of the hash function and other parameters. In this project you will experiment with these factors and make some measurements of the performance of your programs using timing functions that are provided in another of the standard C/C++ libraries, `time.h`.

An application of associative containers

The application to be programmed and timed is as follows. The Jog Phone Company keeps track of phone connections in a table. The key of an entry in the table is the calling phone number and the associated data is the called number and the time the call started.

Insertions and deletions in this table are made in response to *call events*. A call event is either a connection or a disconnection.

The program operates in either a verbose mode, in which it outputs a commentary on the events and its actions, or a silent mode, in which it only prints overall statistics at the end of its operation.

In verbose mode, the program's response to a connection event is either "*n2* is busy" or "At time *t* connected call from *n1* to *n2*", where *n1* is the calling number, *n2* is the called number and *t* is the time at which the event was processed. The "busy" response occurs if there is already a connection involving the called number. Otherwise, in either mode, an entry is made in the table with *n1* as the key and (*n2, t*) as the associated data, and another entry with *n2* as the key and (*n1, t*) as the associated data.

The response to a disconnection event with number *n* is "At time *t* disconnected call between *n1* and *n2*, duration: *d*" where *n1* or *n2* is *n*, *t* is the time at which the event was processed, and *d* is the difference between *t* and the start time of the connection. There is no action taken if there is no call in progress with *n1* or *n2* equal to *n* (i.e., the event is ignored). But if there is such a call in progress, it is removed from the table by deleting both of the entries involving phone number *n*.

Also a discrete event simulation

The program you are to write (each version) performs a simulation of the phone company's operation in handling a specified number of call events and entering them or deleting them from the table. The type of simulation it should do is called a *discrete event simulation*. (See also Weiss, Section 6.4.3.) Such simulations are an important application of priority queues. A discrete event simulation consists of processing of events, such as the connection and disconnection events of the phone company. Events, including the times at which they occur, might be generated randomly. These times do not have to be exact time-of-day times; instead we can just use multiples of some quantum unit, called a tick. Similarly, times between events can also be generated randomly. A simple form of simulation would be to start the clock at zero ticks, then advance it one tick at a time, checking to see if there is an event. This is a *discrete time-driven simulation*. The problem with a discrete time-driven simulation is that it can waste a lot of computing time just advancing the clock tick by tick to the next event. The problem is worse when the events occur infrequently compared to the granularity of the clock. A better way: after completing each event, advance the clock instantly to the time for the next event. This is called a discrete event-driven simulation. Since we need to find the nearest event in the future among those waiting to be processed, a priority queue is the natural data structure to use to hold the events.

Program parameters and output

The program should take four command line parameters. The first parameter *N* indicates the number of events to generate, the second parameter *p* is a probability that determines the rate at which connection events occur, the third parameter is a seed for the random number generator, and the fourth is a flag that controls whether the program operates in verbose or silent mode. The value `log` means verbose mode, while `nolog` means silent mode.

A random connection event is generated by using a random number generator to generate two 7-digit numbers and a start time for the call. This information is packaged in the event and it is placed in the event queue, which is set up as a priority queue in which earliest (smallest) times have the highest priority. When the connection event is later removed from the queue, a disconnection event is constructed for it with its time set to the current time plus a randomly generated duration, and this event is entered in the queue.

After the prescribed number of events has been processed, the program should display (on `cout`) a line that tells the number of calls completed and the average duration of the completed calls and the average number of active connections (calls that are still in progress are ignored in this summary). Finally, it should display the amount of time taken to process all of the events. This (real, not simulated) time can be computed using the function `clock` from `time.h`:

```
clock_t start, duration;
...
start = clock();
... code to be timed
duration = (double)(clock() - start)/CLOCKS_PER_SEC;
cout << "Time used: " << duration << " seconds\n";
```

The type `clock_t` and the factor `CLOCKS_PER_SEC` are also defined in `time.h`.

Two versions required: maps versus hash tables

Write two versions of the program, one that keeps the connected calls in an STL `map`, and the other that keeps them in an hash table that you implement. Represent the phone numbers as strings (C++ strings, using the `<string>` header), and the times or time differences as `float`s. Thus the `map` should be declared as

```
map<string, pair<string, float> >
```

That is, it associates a `pair` consisting of a `string` and a `float` with each key of type `string`, and the ordering of the keys is computed using the usual `<` operator on `string`s. In the `map`, each entry is of the form `pair(n1, pair(n2, t))`.

For the hash table version, you must also implement a hash table class. In Sample hash table[1] there is already a hash table class, implemented using separate chaining. (It is similar to the hash table class in the Chapter 5 online notes, but the interface is extended and modified some to make it more similar to the the STL map interface.) You must implement your hash table with *exactly the same public interface*, but using *open addressing with quadratic probing* instead of separate chaining. (See Section 5.4.2 in the textbook, where some of the implementation is given.)

As mentioned, the public interface of this hash table class is designed to be very similar to that of the STL map class—more precisely, it is very similar to the partial specialization

```
template <class ElementType> map<string, ElementType>
```

i.e., the keys must be strings. This similarity means that very few changes to your phone company simulation program should be required to change it from the map version to the hash table version. For simplicity, the given hash table class omits some features of the map class; for example, it defines `iterator` and `const_iterator` types but doesn't provide them with all of the usual iterator operations. This isn't a problem for the phone company simulation, because it doesn't need full-blown iterators. Note that GNU C++ does provide a set of hashed associative containers (hash_set, hash_multiset, hash_map, hash_multimap) that have full-blown iterators and are as fully compatible with the sorted associated containers as possible. (These containers were actually developed by the generic programming project headed by Alexander Stepanov at Silicon Graphics; they are not part of the C++ standard library but are part of SGI's version of STL. For purposes of this project you are not permitted to use any of SGI's hash tables. In fact, they don't meet the project specification, because they are implemented with separate chaining.)

[1]http://www.cs.rpi.edu/courses/spring00/dsa/projdir/project3/hash.h

Performance comparisons

Since insertions and deletions in the connections table are a large part of the computation in the the phone company simulation, it is a good benchmark for comparing the performance of hashed associative containers versus sorted associative containers (hash tables versus balanced binary search trees). The hash table version of your program should be somewhat faster than the map version. Is it? How much faster (or slower)? Include in the README file you submit with your project a brief report on the performance results you obtained. You should report the times for the two versions on several (silent) runs with at least the values 1000, 10000, 20000, 40000, 80000 for the number of connections.

Additional notes

- To get a better understanding of what the Job Phone Company simulation is supposed to do, look at some Sample Runs[2]
- You can also find two compiled versions of the program (both using a map) in Programs[3] called `jog` (SUN version) or `jog.exe` (Windows PC version). Try them out with various command-line parameters to give you still more information about what is expected.
- Some additional notes and advice are available in Notes[4]

Submission dates and guidelines

The submission deadline is Friday, March 31 at 11:59:59 pm. Projects submitted later than this will NOT be accepted for credit.

This project will require a number of files, so submission will require some extra steps. Make sure that every file that you submit has your name, section meeting time, and instructor name in a comment at the top.

Copy your files to RCS

Copy all of the files that you will submit to RCS. This will include a readme file, make file, and any file with a `.cpp` suffix or a `.h` suffix, but *not*, I repeat *not* files with a `.exe`, `.dsp`, `.dsw` `.ncb` or `.opt` suffix.

Your submission must include a text file called `readme.txt` This file must include your name, section meeting time, and instructor name. In addition, it **must** include a list of all the inplementation files and any special instructions that the TA will need to know in order to run your program.

Example:

```
Snoop Doggie Dog
Section 8: Thursday, 6 PM
Eric Breimer

hash.h
main.cpp

To run the program type
jog.exe N p seed [nolog | log]
```

[2]http://www.cs.rpi.edu/courses/spring00/dsa/projdir/project3/examples.html
[3]http://www.cs.rpi.edu/courses/spring00/dsa/projdir/project3/programs/
[4]http://www.cs.rpi.edu/courses/spring00/dsa/projdir/project3/notes.html

```
where N is the number of events
p is the probability of that event occurs in one unit of time
seed is the random number seed
and the last argument is log if you want a full output log
or nolog if you just want summary results
```

Create a tar file

Login to a Unix system and do the following:

Create a `tar` file. Tar is an acronym for Tape Archive. To create a tar file, use the tar command with two flags `c` (for create) and `f` (for file), followed by the file name of the tar file (it should have a `.tar` suffix) followed by the names of all of the files that you want to send. For example the following command:

`tar cf temp.tar myfile.h myfile.cpp readme.txt`

creates a tar file called `temp.tar` which contains three files, `myfile.h, myfile.cpp` and `readme.txt`.

You will need to check to make sure that the tar file is correct. The best way to do this is to copy the tar file to another directory and run the following command: `tar xf temp.tar` After running this command, the files `myfile,cpp, myfile.h` and `myfile.rc` will be in the new directory. Here are the unix commands to do this:

```
mkdir newdir
cp temp.tar newdir/temp.tar
cd newdir
tar xf temp.tar
```

The first line creates a new directory called `newdir`, the second copies the tar file to the new directory, the third line makes `newdir` the current directory, and the fourth line extracts the files.

Copy the tar file

The submission directory is `/dept/cs/cs230/project3/`. In the submission directory are sub-directories for each section. Copy your tar file to the proper sub-directory. For example, if you are in section 8, copy your tar file to `/dept/cs/cs230/project3/section8`. The name of your tar file must be *login*`.tar`, where *login* is your RCS login. For example, if your login name is `snoopd`, your tar file must have the name `snoopd.tar`. If you want to update your submission, the name of your tar file must be *login*`-2.tar`. Keep incrementing the number every time you want to update your file. Therefor, the name of your third update should be *login*`-3.tar` and your fourth update should be *login*`-4.tar`.

Use the unix copy command `cp` to copy your submission. For example:

`cp temp.tar /dept/cs/cs230/project3/section8/snoopd.tar`

Please, make sure you copy your tar file into the subdirectory for your section and verify that the name of your tar file is your RCS login. **You will not receive credit for your submission unless you follow the submission guidelines.**

Check list

- Make sure the name of your tar file matches your RCS user id.
- Make sure you submit to the proper section.
- Make sure your tar file includes all .h and .cpp files.
- Make sure all your files are text format before you use tar.

Grading criteria

- 25% **Hash Table Class Design and Effort:** The hash table must use the given interface and must use quadratic probing. Although your code may not be completely correct and may not compile, you can still earn the full 25%. You must alter the find, insert, erase, CopyTable, ExpandTable, DeleteTable, and AllocateTable so the the Hash Table efficiently implements quadratic probing.

- 25% **Discrete Event Simulation Class Design and Effort:** Your code must attempt to implement a realistic discrete event simulation. Although your code may not be completely correct and may not compile, you can still earn the full 25%. Your event simulator must attempt to generate N connection events according to probability p (the probability that a connection event is generated in one unit of time). The duration of the calls should be between 0 and 2000 seconds and only valid disconnection events should be generated.

- 35% **Hash and Map Comparison** Your program must have two versions. One version maintains the connection table using a map and the other version uses a hash table with quadratic probing. You should report the running times for the two versions on several runs with at least the values 1000, 10000, 20000, 40000, 80000 for the number of connections. Your program should verify what you've reported. If you summary information is realistic and consistent, and your hash table implementation outperforms the map implementation, then you will get 35%. Otherwise, you should explain in your report why you could not achieve these results. If your hash table does not outperform the map and you don't offer a reasonable explanation, then you will get 30%. If you were not able to implement the quadratic probing hash table, you should still report the data for the map and you will still recieve 20%.

- 10% **Compilation** Even if your program does not produce correct or realistic summary information, you can still earn up to 10% for successful compilation. You can not get credit for compilation unless you've made a valid attempt to implement all the components (quadratic probing hash table, discrete event simulation, and summary calculations).

- 5% **Coding Guidelines:** You must make a valid effort to follow the coding guidelines whenever appropriate. Use your best judgement to determine which guidelines should be followed. If you are in doubt, explain (in your comments) your choice to follow or not to follow certain guidelines. You can not get credit for the coding guidelines unless you've made a valid attempt to implement all the components (quadratic probing hash table, discrete event simulation, and summary calculations).

Academic integrity

Students may discuss the class and algorithm design. All code must be implemented by the student alone. Sharing of code among students is expressly forbidden and will be detected by code comparison tools. Projects from all eight sections will be graded together by the TAs.