CSCI 230 — Data Structures and Algorithms
Project 3 — Jog Phone Company — Additional Notes

March 17, 1999

In programming the Jog Phone Company simulation and hash table implementation, here are some other things to watch out for:

1. **Searching a map with the find member function.** In the above example note that in the connections table the entries appear with their keys in increasing order. Sorted order is maintained by an STL `map` so that searches can be conducted with a binary search algorithm, which is much faster than linear search. (The time required is proportional to $\log n$ rather than n.) The main function for searching a `map` is the `find` member function, which takes a key k to be searched for as its argument and returns an iterator referring to the entry with that key, if there is one, or the iterator returned by the `end` member function if there is no entry with key equal to k, as in the following code

   ```
   typedef map<double, pair<double, float> > table_type;
   ...
   table_type connections;
   table_type::iterator i;

   double number = ....

   i = connections.find(number);
   if (i != connections.end())
     ... action to take if the number was found
   else
     ... action to take if the number wasn't found
   ```

   In case `(i != connections.end())`, it is valid to use `*i` to retrieve the key and its associated value, which are stored as a `pair`.

2. **Inserting into a map using array-like bracket notation.** There are several ways in which new entries can be inserted in a `map`. One of the easiest is to use the same bracket subscript notation as you would with an array, as in the following code (which assumes the same `typedef` and declarations as in the previous part).

   ```
   string number1;
   string number2;
   float start_time;

   ... code to give values to number1, number2, start_time

   connections[number1] = pair<double, float>(number2, start_time);
   ```

   This means store the value created by the `pair<string, float>` constructor in the table so that it can be found using key `number1`. On the left hand side of the assignment, it looks like you are treating `connections` as an array; this works because the `map` class overloads the definition of `operator[]`.

3. **The map insert member function.** Another way of inserting a new entry into a map is with the `map` class's *insert* member function. This function takes a `pair` consisting of a key and its associated value. It returns a `pair` consisting of an iterator and a `bool`. The `bool` value is `true` if the pair was successfully inserted in the table, `false` if it wasn't (which would be the case if there was already an entry in the table with the given key). If the insertion was successful, the iterator refers to the new entry; otherwise it refers to the existing entry with the given key. Here is some sample code (again assuming the same declarations as in part 1).

```
typedef pair<double, float> associated_type;
typedef pair<const double, associated_type> entry_type;

typedef pair<table_type::iterator, bool> pib;

pib p = connections.insert(
          entry_type(number1, associated_type(number_2, start_time)));
if (p.second)
  ... action to take if the insertion was successful,
  ... using p.first to refer to the entry with key == number1
else
  ... action to take if there was already an entry with key == number1,
  ... using p.first to refer to that entry
```

Thus `insert` allows you to check if a key is in the table and, if not, put a new entry with that key into the table in one step—that is, with one search of the table. This is faster than using `find` and then `operator[]`, which requires searching the `map` twice, but either way is acceptable for this assignment.

4. **Using drand48.** For randomly generating phone numbers and simulated times you'll need to use a pseudo-random number generator from the C++ library. Unfortunately the only random number generator required to be in all C++ libraries is `rand`, which is not a very good generator. If you are using the `g++` compiler, a better random number generator is `drand48`. The `rand` function has associated with it another function called `srand` (and `drand48` similarly has `srand48`) which is called to "seed" the random number generator; i.e., to initialize it. The initial seed value determines the entire sequence of values generated, so one can repeat an experiment by reusing the same seed, or one can ensure that a different sequence of values will be generated by using a different seed.