Programming Assignment 4

Query Me This

Time due: 11:00 PM Wednesday, June 9

For this project, you will build a simple database in memory and process database queries.

Introduction

A simple database *table* is a collection of *records*. Each record has the same number of *fields*; we'll call that the number of *columns* in the table. Each of the columns has a name. For example, we may have a 4-column table whose columns are named customer, product, price, and location. Here are some records we might have in this table:

customer	product	price	location
Patel	12345	42.54	Westwood
O'Reilly	34567	4.99	Westwood
Hoang	12345	30.46	Santa Monica
Patel	67890	142.75	Hollywood

One database operation of interest is the one that finds all records in a table for which one particular field equals a certain value. We make the simplifying assumption for this project that for any particular table, there's only one field for which we want such lookups to be especially fast. We call this field the *key field*. Let's suppose for this example that we designated the customer field as the key field. Then we want the operation "Find all records where the customer is Patel" to quickly produce the records:

> Patel 12345 42.54 Westwood
> Patel 67890 142.75 Hollywood
> ...

Notice that we allow more than one record to have the same key.

Another operation is a more general query capability. We may want to select all records for which, for example, the price is less than 40. This would produce

```
O'Reilly 34567 4.99 Westwood
Hoang 12345 30.46 Santa Monica
...
```

Your Task

You will implement a class Table that will provide the functionality described above.

class Table

The Table class is responsible for holding table records and responding to queries about them. Records **must** be stored in a data structure suitable for fast insertions and lookups by key: This means a tree or hash table. If you choose a tree, you do not have to worry about balancing it; if you choose a hash table, you do not have to worry about changing the number of buckets dynamically. Also, if you choose a hash table, it must not have more than 1000 buckets (to make it easier for us when we test your program on a table with thousands of records). For this project, you must implement the tree or hash table data structure yourself. You must **not** use any STL associative container: no map, multimap, unsorted_map, unsorted_multimap, set, multiset, unsorted_set, or unsorted_multiset. You *are* allowed to use vector, list, stack, and queue.

```
class Table
{
  public:
    Table(std::string keyColumn, const std::vector<std::string>& columns);
    ~Table();
    bool good() const;
    bool insert(const std::string& recordString);
    void find(std::string key, std::vector<std::vector<std::string>>& records) const;
    int select(std::string query, std::vector<std::vector<std::string>>& records) const;

      // We prevent a Table object from being copied or assigned by
      // making the copy constructor and assignment operator unavailable.
    Table(const Table&) = delete;
    Table& operator=(const Table&) = delete;
  private:
    ...
};
```

You must **not** make any deletions, additions, or changes to the *public* interface of this class. You can and will, of course, add private data members and perhaps private member functions to help you with your implementations of the member functions. You must **not** declare any public

data members, nor use any global variables whose values may change during execution (so global *constants* are OK). You may add additional functions that are not members of any class.

Table(std::string keyColumn, const std::vector<std::string>& columns)

Construct an empty Table whose column names are the elements of the vector second parameter. The first parameter is the name of the key field. The parameters might be such that you could not construct a valid table: the columns vector might be empty or contain empty or duplicate strings, or the keyColumn might not be any of the column names. The right way to handle such a problem in modern C++ is to throw an exception, but we haven't discussed exceptions. Instead, for this project you must set the state of the object so that it can be tested using the good member function. For example, a user might write:

```
Table t(constructor argument);
if ( ! t.good() )
   give up, since the constructor failed
```

This isn't a great solution, since a careless user could forget to test for the failure and try to use an invalid Table object, but it's the best we can do without exceptions. To mitigate the problem, each member function should check that it's been called on a valid object, and return harmlessly, if possible, if it hasn't.

bool good() const

Return true if the table was successfully constructed, and false otherwise.

~Table()

The destructor does what is necessary to ensure that a Table releases any resources it holds when its lifetime ends.

bool insert(const std::string& recordString);

Insert a record into the database. The parameter is a string with the fields of the record in a format described below. If the string has the correct number of fields for the table, insert a record with those fields into the table and return true; otherwise, leave the table unchanged and return false. Here is an example:

```
  // Since C++11, vectors can be constructed from initializer lists
vector<string> cols = {
        "customer", "product", "price", "location"
};
Table t("customer", cols);
assert(t.good());
assert(t.insert("Patel 12345 42.54 Westwood"));
```

```
assert(t.insert("O'Reilly 34567     4.99 Westwood   "));
assert(t.insert("   Hoang  12345 30.46 'Santa Monica' "));
assert(t.insert("Patel\t67890\t142.75  \t \t\t  \tHollywood"));
assert( ! t.insert("Figueroa 54321 59.95"));
```

A *delimiter* is a blank, tab, carriage return, or newline character. A field in the recordString is represented by either

- a sequence of one or more non-delimiter characters, not starting with a single quote; or
- a single quote followed by zero or more of the following units followed by a single quote:
    - a character other than a single quote; or
    - two consecutive single quotes (representing one single quote in the actual field value)

Here are some examples:

recordString text ...	**...represents the field value**
Patel	Patel
'Santa Monica'	Santa Monica
chillin'	chillin'
O'Reilly	O'Reilly
'chillin''	chillin'
'O''Reilly'	O'Reilly
'Baba O''Riley - Who''s Next'	Baba O'Riley - Who's Next
''	*the empty string*

In the recordString, the representations of the fields are separated by one or more delimiters, and the recordString starts or ends with zero or more delimiters. One exception: If a field representation surrounded by single quotes is followed by a field representation not surrounded by single quotes, they need not be separated by delimiters, so the two field representations 'da Gama' 98765 could also appear in the string as 'da Gama'98765.

void find(std::string key, std::vector<std::vector<std::string>>& records) const;

Replace the value in the second parameter with a vector containing as many elements as there are records in the table whose key field is a string equal to the value of the first parameter. Each of those elements is one of the matching records, represented as a vector whose elements are the fields of that record. The records placed in the vector don't have to be in any particular order.

For example, for the table t built in the example shown for the insert function, the assertions in the following should succeed:

```
vector<vector<string>> v;
t.find("Patel", v);
assert(v.size() == 2);
vector<vector<string>> expected = {
        { "Patel", "12345", "42.54", "Westwood" },
        { "Patel", "67890", "142.75", "Hollywood" }
};
assert((v[0] == expected[0]  &&  v[1] == expected[1])  ||
     (v[0] == expected[1]  &&  v[1] == expected[0]) );
```

int select(std::string query, std::vector<std::vector<std::string>>& records) const;

The first parameter is a *query* (defined below) that specifies a condition that does or does not hold for a record (e.g., that a field named start date is greater than or equal to 2021-03-29). Replace the value in the second parameter with a vector containing as many elements as there are records in the table that satisfy the query. Each of those elements is one of the matching records, represented as a vector whose elements are the fields of that record. The records placed in the vector don't have to be in any particular order. The function returns 0 if it succeeds without any problems. (The return values are specified later.) For example, for the table t built in the example shown for the insert function, the assertions in the following should succeed:

```
vector<vector<string>> v;
assert(t.select("location < Westwood", v) == 0);
assert(v.size() == 2);
vector<vector<string>> expected = {
        { "Hoang", "12345", "30.46", "Santa Monica" },
        { "Patel", "67890", "142.75", "Hollywood" }
};
assert((v[0] == expected[0]  &&  v[1] == expected[1])  ||
     (v[0] == expected[1]  &&  v[1] == expected[0]) );
```

We define a *query term* as a sequence of three tokens:

- the name of a column in the table, followed by
- an operator, followed by
- a value the field is compared with

The operators are

- <, <=, >, >=, !=, ==, and =. These operators do the indicated string comparison. (The operator = means the same equality comparison as ==.)
- LT, LE, GT, GE, NE, EQ, with the letters in either upper or lower case (so LT, Lt, lT, and lt can be used interchangeably). These operators do numerical comparisons (described

below) and mean "is less than", "is less than or equal to", "is greater than", "is greater than or equal to", "is not equal to", and "is equal to", respectively.

A *query* consists of one query term (but for a bonus opportunity, see the Bonus section below). The tokens in a query follow the same rules as fields in the recordString parameter for the insert member function: the same delimiters, enclosed in single quotes if they contain a delimiter, etc. The tokens that don't require surrounding single quotes mean the same as if they were so surrounded. Here are some examples of queries; the first three mean the same thing:

```
location < Westwood
    location     '<'  'Westwood'
'location' < 'Westwood'
customer != 'O''Reilly'
customer = Patel
product != 12345
```

A record satisfies a query term using a numerical comparison operator (e.g., LT) if the value of the indicated field and the value to the right of the operator are both in the proper form for a number, and the indicated comparison is true for the operands if they are interpreted as numbers in the usual way; otherwise, the record does not satisfy the query term. The proper form for a number is a nonempty string that leads to a successful conversion by std::strtof of the entire string (see Implementation Hints below). Assuming the example table shown for the insert function, each of the following queries is satisfied by at least one record:

```
location = 'Santa Monica'
product == 12345
product != 00012345  12345 and 00012345 are different strings
product EQ 12345
product EQ 00012345  12345 has same numeric value as 00012345
product < 900        initial 1, 3, 6 is less than initial 9
product GT 900       numbers 12345, 34567, 67890 greater than 900
```

If a query term is badly formed, select sets its second parameter to be an empty vector and returns −1. Here are some badly formed queries for the example table:

```
Hollywood = location       Hollywood is not a column name
price is expensive         is is not an operator
location = Santa Monica    four tokens, not three
price GT ''                empty string is not a proper number
price GT 100K              100K is not entirely a number
```

If a query is well formed, then select replaces its second parameter with a vector containing as many elements as there are records in the table that satisfy the query. Each of those elements is one of the satisfying records, represented as a vector whose elements are the fields of that record. The records placed in the vector don't have to be in any particular order. For a query term specifying a field involved in a numerical comparison, any record whose value for that field is not in the proper form for a number is considered to not satisfy the query. The return value for select is the number of such records; this would be 0 if all values for such a field are in proper form for a number. Here's an example:

```
vector<string> cols = { "item name", "price" };
Table t("item name", cols);
assert(t.good());
assert(t.insert("'chocolate bar' 1.69"));
assert(t.insert("coffee 7.99"));
assert(t.insert("hummus 3.49"));
vector<vector<string>> v;
assert(t.select("price LT 5", v) == 0);
assert(v.size() == 2);  // chocolate bar and hummus
assert(t.insert("pretzels 1.W9"));
assert(t.select("price LT H", v) == -1);
assert(v.size() == 0);
assert(t.select("price LT 5", v) == 1);  // 1 because pretzels 1.W9
assert(v.size() == 2);  // chocolate bar and hummus
```

Other Requirements

Your code will go into two files: Table.h, which will contain the declaration of the Table class and must have an appropriate include guard, and Table.cpp, which will contain the implementation of the Table class and any additional functions or classes you might write to help you with the implementation. If you want to have a main routine to test your code, put it in a separate .cpp file that you will not turn in.

You must have an implementation for every function specified by this assignment. If you can't get a function implemented correctly, it must at least compile successfully. For example, if you don't have time to correctly implement Table::select, say, here is an implementation that meets this requirement in that it at least compiles correctly:

```
int select(string query, vector<vector<string>> records) const
{
   return 0;  // not always correct, but at least this compiles
}
```

None of the member functions you write may cause anything to be read from cin or written to cout. If you want to print things out for debugging purposes, write to cerr instead of cout. When we test your program, we will cause everything written to cerr to be discarded; we will never see that output, so you may leave those debugging output statements in your program if you wish.

Your code must compile successfully under both g32 and either Visual C++ or clang++. If your code is linked to a file containing

```
#include "Table.h"
#include <iostream>
#include <cassert>
using namespace std;

int main()
{
   vector<string> cols = { "N", "Z" };
   Table t("Z", cols);
   assert(t.good());
   assert(t.insert("UCLA 90095"));
   assert(t.insert("Caltech 91125"));
   vector<vector<string>> v;
   t.find("90095", v);
   assert(v.size() == 1);
   assert(v[0][0] == "UCLA"  &&  v[0][1] == "90095");
   assert(t.select("Z > 90210", v) == 0);
   assert(v.size() == 1);
   assert(v[0][0] == "Caltech"  &&  v[0][1] == "91125");

   vector<string> uclacols = { "last name", "first name", "major", "UID", "GPA" };
   Table ucla("UID", uclacols);
   assert(ucla.good());
   assert(ucla.insert("Bruin Jose 'COG SCI' 304567890 3.4"));
   assert(ucla.insert("Bruin Josephine 'COM SCI' 605432109 3.8"));
   assert(ucla.insert("Trojan Tommy LOSING 000000000 1.7"));
     // Troy brought a wooden horse full of Greek soldiers inside the
     // city walls and lost the Trojan War.  How strange to look up to
     // gullible losers as role models.
   assert(ucla.select("GPA GE 3.2", v) == 0);
   assert(v.size() == 2);
   assert(v[0][0] == v[1][0]  &&  v[0][1] != v[1][1]);

   cout << "DONE" << endl;
}
```

the linking must succeed. When the resulting executable is run, execution must at least reach the second assert. (Of course, if you've correctly implemented the functions, the program will run to completion.)

During execution, if a client uses the classes as defined, your program must not perform any undefined actions, such as dereferencing a null or uninitialized pointer, or accessing a vector element out of bounds.

Implementation Hints

To help you extract fields from a recordString, we've written this StringParser class. Read it to figure out what it does, use it, and save a lot of work. If you use it, put it in Table.cpp. (It does not have to be nested in the Table class.)

Every STL container has a member function named clear that removes all elements from the container, leaving it empty.

The header <functional> defines a hash class template that you can use to hash strings:

```
string s("Hello there");
unsigned int h = std::hash<std::string>()(s);
```

The hash values produced by this function range from 0 to about 4 billion.

If you include <cstdlib>, then if the first argument to the following function is string that is a number in proper form, the function returns true and sets the second parameter to the number represented by the string; otherwise, it returns false (and the effect on the second argument is not specified).

```
bool stringToDouble(string s, double& d)
{
	char* end;
	d = std::strtof(s.c_str(), &end);
	return end == s.c_str() + s.size()  &&  !s.empty();
}
```

Bonus

For 5 bonus points, correctly implement the following expanded definition of *query*: A *query* is one or more query terms connected by the & operator (meaning AND) or the | operator (meaning inclusive OR). Comparison operators have higher precedence than &, which has higher precedence than |. Parentheses may be used to override the precedence of & over |. Parentheses and the operators must be separated by delimiters from other tokens and are allowed to be surrounded by single quotes. The select function returns −1 if the query is badly formed. Here are some examples of well formed queries:

```
customer = Patel
( customer = Patel ')'
( location == Westwood | location == Hollywood ) '&' price LT 100
price GE 20  &  location != Westwood
```

and here are some badly formed ones:

```
(customer = Patel)      bad unless "(customer" is a column name
product = 12345 location != Westwood
location = ( Westwood | Hollywood )
```

You must modify the Table constructor so that the good function will return false if the user attempted to construct a Table with a column name that was &, |, (, or). Those tokens *are* allowable as the right operand of a comparison operator in a query term; in that context, they are just ordinary string values, not query operators or grouping parentheses.

You are free to use our or your solution to Homework 2 as a basis for your code.

Don't even consider attempting this bonus part until you are sure select works correctly for queries that consist simply of one query term. We will prioritize helping students with issues related to the regular part of this project over issues related to implementing this expanded definition of *query*, since that's just bonus, not essential.

Turn it in

By Tuesday, June 8, there will be a link on the class webpage that will enable you to turn in your source files. You will turn in a zip file containing these two source code files and nothing more:

- Table.h
- Table.cpp

You do not have to turn in a report file.