Project 2: Internet Router

Router Simulation based on Route Table as Generic Hash Table

Current version: 04/04/2012

Educational Objectives: After completing this assignment, the student should be able to accomplish the following:

- Describe and explain in detail the concept of hash table
- Implement hash tables as vector of lists
- Define and implement the ADT Table using a private hash table structure
- Define and implement bidirectional iterator class for this implementation of Table
- Explain the concept of Internet Router and Route Table
- Implement a Route Table using the ADT Table (and, in particular, using an implementation using hash tables.

Operational Objectives: Implement the template adaptor classes `HashTable<K,T,H>` and `HashTableIterator<K,T,H>`. Design and implement a class `RouteTable` to be used with the distributed client program `iprouter.cpp` to simulate the operation of an Internet router.

Deliverables: Four files:

```
hashtbl.h    # contains HashTable<> and HashTableIterator<> template classes (including
implementations)
iptable.h    # contains RouteTable class definition and supporting function prototypes
iptable.cpp  # contains RouteTable implementations and supporting function implementations
log.txt      # your project work log
```

Procedural Requirements

1. The official development | testing | assessment environment is `gnu g++` on the `linprog` machines.

2. Create and work within a separate subdirectory `cop4530/proj2`.

3. **Do your own work.** Variations of this project have been used in previous courses. You are not permitted to seek help from former students or their work products. For this and all other projects, it is a violation of course ethics and the student honor code to use, or attempt to use, code from any source other than that explicitly distributed in the course code library, or to give or receive help on this project from anyone other than the course instruction staff. See *Introduction/Work Rules*.

4. Begin by copying the entire directory `LIB/proj2` into your `proj2` directory. Note that there is a subdirectory `testfiles` that needs to be copied. At this point you should see these files in your directory:

   ```
   fhtbl.cpp           # test harness for hash tables
   hasheval.cpp        # hash table analyzer
   rantable.cpp        # random table generator
   hashcalc.cpp        # hash calculator
   hashtbl.start       # starting point for hashtbl.h
   iprouter.cpp        # internet router simulator
   iptable.h.start     # starting point for iptable.h
   iptable.cpp.start   # starting point for iptable.cpp
   makefile.ht         # build hash table project
   makefile.ip         # build ip router project
   proj2submit.sh      # submit script
   testfiles           # directory of test files
   ```

   Then copy these relevant executables:

   ```
   LIB/area51/fhtblKISS.x      # fhtbl.x with KISS hash function
   LIB/area51/fhtblMM.x        # fhtbl.x with MM hash function
   LIB/area51/fhtblSimple.x    # fhtbl.x with Simple hash function / non-prime flag
   LIB/area51/hashevalKISS.x   # hash analysis with KISS
   LIB/area51/hashevalMM.x     # hash analysis with MM
   LIB/area51/hashevalSimple.x # hash analysis with Simple/non-prime
   LIB/area51/rantable.x       # random table generator
   LIB/area51/hashcalc.x       # hash function calculator
   LIB/area51/iprouter.x       # sample iprouter executable
   ```

The executables in area51 are distributed only for your information and experimentation. You will not use these files in your own project, but they will help you understand hashing and hash tables and are very useful in preparing for the final exam. When you have questions about behavior of either hash tables or the router simulation, use these executable to find the answer.

5. You are to define and implement the template classes HashTable<K,T,H> and its associated iterator class HashTableIterator<K,T,H>. In addition you are to design and implement the class RouteTable to be used with the distributed client program iprouter.cpp.

6. File hashtbl.h should contain the definitions and implementations of the template classes HashTable<K,T,H> and HashTableIterator<K,T,H>. Note that a lot of this work is already done in the startup file.

7. File iptable.h should contain the definition of class RouteTable and prototyes of supporting functions.

8. File iptable.cpp should contain implementations of RouteTable methods and supporting functions. Again, the startup files contain all of the boiler plate and some other metods already completed.

9. Two makefiles are supplied: makefile.ht and makefile.ip makefile.ht builds the supporting test infrascrtcure for hash tables, and makefile.ip builds the executable iprouter.x. You can test different targets individually by naming them as an argument to the make command.

10. Submit the assignment using the script proj2submit.sh.

Warning: *Submit scripts do not work on the program and linprog servers. Use shell.cs.fsu.edu to submit assignments. If you do not receive the second confirmation with the contents of your assignment, there has been a malfunction.*

Code Requirements and Specifications - HashTable and HashTableIterator

1. Implement the HashTable<K,T,H> and HashTableIterator<K,T,H> classes as defined in the file LIB/proj2/hashtbl.start using the implementation plan discussed in the lecture notes.

2. The following are the items that have incomplete implementations in hashtbl.start:

```
// ADT Table

template <typename K, typename D, class H>
HashTableIterator<K,D,H> HashTable<K,D,H>::Insert (const K& k, const D& d)
{
}

template <typename K, typename D, class H>
bool HashTable<K,D,H>::Remove (const K& k)
{
}

template <typename K, typename D, class H>
bool HashTable<K,D,H>::Retrieve (const K& k, D& d) const
{
}

template <typename K, typename D, class H>
HashTableIterator<K,D,H> HashTable<K,D,H>::Includes (const K& k) const
{
}

// ADT Associative Array

template <typename K, typename D, class H>
D& HashTable<K,D,H>::Get (const K& key)
{
}

template <typename K, typename D, class H>
void HashTable<K,D,H>::Put (const K& key, const D& data)
{
}

template <typename K, typename D, class H>
D& HashTable<K,D,H>::operator[] (const K& key)
```

```
{
}

// Iterator increment

template <typename K, typename D, class H>
HashTableIterator <K,D,H>& HashTableIterator<K,D,H>::operator ++ ()
{
}
```

Note that our hash table has both the Table API and the Associative Array API.

3. Place `HashTable<K,T,H>` and `HashTableIterator<K,T,H>` in the `fsu` namespace.
4. For the private data storage use a `fsu::Vector < fsu::List < fsu::Entry < KeyType, DataType > > >` object. (This is implied by the definition in `hashtbl.start`.)
5. Note the the class template `Entry<K,T>` is optimized to hold entries in tables. It comes complete with appropriately overloaded operators and a hash function class template. Be sure to familiarize yourself with this class template. (Distributed in file `LIB/tcpp/entry.h`.)
6. Be sure not to change the definition of `HashTable<>` from that distributed in `LIB/proj2/hashtbl.start`.
7. Note that `HashTableIterator` is a ConstIterator type, so that only the const versions of operator * and Retrieve() exist.
8. Place all hash table code, including definitions and implementations for both hash table and hash table iterator, in the file `hashtbl.h`.
9. Thoroughly test your implementation for correct functionality using the provided test client `fthtbl.cpp` and tables from `tests/tables/` and/or tables generated using `rantable.cpp`

IP Addressing

There are two ways to represent ip addresses: the 4-number "dot" notation and the 32-bit (4-byte) "number" notation. The dot notation we store as a `String` object (`typedef ipString`) and the number notation we store as an `unsigned int` object (`typedef ipNumber`). Generally, a router uses `ipNumber` as its internal representation, while externally across the Internet the `ipString` representation is used.

The `ipString` representation consists of four numerical fields separated by (three) dots. For example, `128.186.121.211` is the ip address of a machine in the computer science department. Each numerical field in this address represents a number in the range [0, 255]. (Numbers 256 or greater make an *invalid* `ipString`. We also define the zero address `0.0.0.0` as invalid.) This number in turn denotes an 8-bit (one byte) quantity. The four numerical fields, concatenated, represent 32 bits or 4 bytes. This 4-byte number is the internal `ipNumber` representation. In general, we use hexadecimal (base 16) representation to denote `ipNumber` objects. The `ipNumber` representation of the address above is `10000000101110100111100111010011` (bin) = `80BA79D3` (hex).

The `ipClass` of an ip address is defined in terms of the bits in its `ipNumber` representation. There are three recognized classes of `ipNumbers`: A, B, and C. An `ipNumber` that is not one of these classes is called *bad* and cannot be used. Class A consists of `ipNumbers` beginning (on the left) with bit '`0`'. Class B begins with bits '`10`'. Class C begins with bits '`110`'. All other `ipNumbers` are bad. (Note that bad `ipNumbers` are not the same as invalid `ipStrings`.) A good (i.e., not bad) `ipNumber` contains information in fields called `netID` and `hostID`. The ranges of these fields depend on the class, as shown in the following table (numbering the bits from the left, starting with 1):

```
Class A:          netID = bits 2..8;          hostID = bits   9..32
Class B:          netID = bits 3..16;         hostID = bits  17..32
Class C:          netID = bits 4..24;         hostID = bits  25..32
```

The `netID` and `hostID` are full 32-bit words with the irrelevant bits masked to zero. For example, the address `80BA79D3` is class B with `netID = 00BA0000` and `hostID = 000079D3`. `netID` and `hostID` are used by a router to forward messages. We do not express `netID` and `hostID` in "dot" notation. The following summarizes the calculations:

```
ipString          128.186.121.211
ipNumber  (bin)   1000 0000 1011 1010 0111 1001 1101 0011
```

```
ipNumber  (hex)     8    0    B    A    7    9    D    3    = 80BA79D3
ipClass             class B
netMaskB  (bin)     0011 1111 1111 1111 0000 0000 0000 0000
netMaskB  (hex)     3    F    F    F    0    0    0    0    = 3FFF0000
netID     (bin)     0000 0000 1011 1010 0000 0000 0000 0000
netID     (hex)     0    0    B    A    0    0    0    0    = 00BA0000
hostMaskB (bin)     0000 0000 0000 0000 1111 1111 1111 1111
hostMaskB (hex)     0    0    0    0    F    F    F    F    = 0000FFFF
hostID    (bin)     0000 0000 0000 0000 0111 1001 1101 0011
hostID    (hex)     0    0    0    0    7    9    D    3    = 000079D3
```

Real Routers

Routers are special-purpose computers that monitor Internet traffic, either rejecting or accepting messages. Accepted messages are forwarded and rejected messages are ignored. For example, the Love Building router run by the Computer Science department rejects incoming messages that are not addressed to a machine in the building, and accepts and forwards messages addressed to one of the machines in the building. It also forwards outgoing messages to another router for further routing. Much of the computation has direct hardware support, enabling speed (bandwidth) to be 1 gigabit. The route tables, in contrast, need to be software tables so that the router can be programmed as both internal and external machine configurations and addresses evolve. Hash tables/maps are the only data structure that guarantees fast enough lookup for today's high-bandwidth routers.

Code Requirements and Specifications - RouteTable

1. **RouteTable Types.** You will need the following type definitions:

```
typedef unsigned int ipNumber;
typedef fsu::String  ipString;

enum ipClass
{
  classA, classB, classC, badClass
} ;
```

2. **Output operator**. Overload operator <<() for type `ipClass` using the following prototype:

```
std::ostream& operator << (std::ostream& os, ipClass ipc);
// sends 'A', 'B', 'C', or 'D' to os depending on ipClass value
```

3. **RouteTable Public Interface.** You will need the following public methods in `RouteTable`:

```
void Load        (const char* tblfile);
void Save        (const char* tblfile);
void Insert      (const ipString& destination, const ipString& route);
void Remove      (const ipString& destination);
void Go          (const char* msgfile, const char* logfile);
void Clear       ();
void Dump        (const char* dumpfile);
     RouteTable  (unsigned int sizeEstimate);
     ~RouteTable ();

static ipClass ipInterpret(const ipNumber& address, ipNumber& netID, ipNumber&
hostID);
// returns ipClass and sets netID and hostID of address;
// if address is badClass, netID and hostID are set to 0.

static ipNumber ipS2ipN (const ipString& S);
// converts ipString to ipNumber, checking for syntax and oversize errors
```

`Go()` performs a simulation as described.

`Load()` and `Save()` build/save the route table from/to an external file.

`Insert()` and `Remove()` are similar to the standard table operations, except that they must translate input from ipString to ipNumber prior to accessing the underlying table.

`Go()` and `Dump()` send output to screen when passed a 0 pointer as output file parameter; otherwise all char* parameters are treated as external file names.

Static member functions behave like stand-alone functions with scope limited to the class. These functions should be included in the `RouteTable` class as static methods.

4. **Data Structures.** Use a `HashTable<ipNumber,ipNumber,ipHash>` object as the primary data structure supporting the route table.
   Note : This object will need to be created dynamically in order to set the number of buckets.

5. **Data, Table, and File Format.** `RouteTable` objects use three distinct kinds of files: *table* files, *message* files, and *log* files.

   A *table file* (extension `.tbl`) consists of pairs of `ipNumber` (one pair per line) written in hex notation with `'0'` fill, so that all entries have 8 characters. The `Load(filename)` method of a `RouteTable` object reads data from a table file and inserts the data into its internal table. The `Save(filename)` method writes all data in the internal table to a table file.

   The internal table also uses `ipNumber` representation of addresses.

   The `ipNumber` pairs in a route table are destination and route pairs. The destination is the key, and the route is the data in the table.

   When individual entries are inserted, removed, or looked up (via the `RouteTable` public interface) in the internal table, the external `ipString` representation is used for input.

   The `Go()` method is the method that simulates a router in operational mode. The incoming internet traffic is simulated by a message file. Each line of the message file can be thought of as a "packet" with a destination and a message body. The result of routing this traffic is recorded in a log file.

   `Go()` should read message packets (lines) one at a time from a message file and write the disposition of these message packets to a log file. For each message packet (line of the message file), the router (1) reads the destination address and converts it to ipNumber form; (2) rejects the message packet if the ipClass is bad; (3) looks up the packet destination in the internal table; (4) if not found, rejects the packet; (5) routes the packet using the netID and hostID of the route retrieved from the table; (6) writes the disposition of the message packet to the log file.

   The *message file* (extension `.msg`) consists of two strings per line. The first string of the packet (line) is an ip address in dot notation (intended to be the destination of the message) and the second is a message ID (simulating the body or content of a real message). Each line of this file represents one "message packet" coming in to the router.

   The *log file* (extension `.log`) should contain one line for each message packet in the message file. The log entry for a message should begin with the message ID, followed by the `ipNumber` of the destination (8-digit hex), then the `ipClass` and the routing information (netID and hostID) for the message.

6. **Running a Simulation.** Your `RouteTable` should work with the supplied client program `iprouter.cpp`, which simulates the operation of a router.

Hints

- Start NOW on `HashTable<K,T,H>` and test it. Then start on `RouteTable`.
- A client test program for `HashTable<K,T,H>` is distributed as `tests/fthtbl.cpp`. This will be compiled directly from the course library when you enter the command "`make fthtbl.x`". However, you should copy the source code into your project directory and read the code.
- The directory `tests/tables` contains test table files. A random table file generation program is supplied as `tests/rantable.cpp`.
- You may find the following `typedef` statements (made inside the private portion of RouteTable) very helpful in making your code readable, to yourself and others:

```
typedef fsu::Entry < ipNumber, ipNumber >              EntryType;
typedef fsu::HashTable < ipNumber, ipNumber, ipHash > TableType;
```

- The client program `iprouter.cpp` is distributed in the `proj2` directory, along with a selection of table files.
- You can assume that the table files contain only good `ipNumbers`, since they are all created by a properly operating router.

- You can assume that incoming messages will have valid dot-notation strings as destinations, but you cannot assume that incoming message destination strings denote good ipClasses.
- You cannot assume that user entries are either valid or good.
- To read and write the hex codes correctly, you will need (1) to manipulate the std::ios:: flags hex and uppercase for both input and output streams; (2) to set the fill character using the manipulator std::setfill(); and (3) to use the std::setw() manipulator to set the column width in output.
- The hash function class ipHash may be defined and implemented as follows:

```
// in file iptable.h
class ipHash
{
  public:
    unsigned int operator () (const ipNumber&) const;
} ;

// in file iptable.cpp
#include <hash.h>
unsigned int ipHash::operator () (const ipNumber& ipn) const
{
  return hashfunction::KISS (ipn);
}
```

  You need a hash function object to instantiate a hash table object.

- You need only one private data item in RouteTable: a pointer to a HashTable<> object.
- The following aspects of your solution will be tested:

  1. Completeness and correctness of your HashTable<> design and implementation.
  2. Correct/failsafe file handling
  3. Correct/failsafe table building/saving
  4. Correct/failsafe handling of user input
  5. Correct/failsafe translation of ipStrings to ipNumbers
  6. Correct/failsafe decoding of ipNumbers (class, etc)
  7. Correct/failsafe routing (table lookup)

- Sample executables fthtbl.x and iprouter.x are supplied LIB/area51. Sample tables and a table building program are supplied in LIB/tests/tables. Sample route table and message files are supplied in LIB/proj2.
- **Test Bench:** A test suite and script are supplied in the directory proj2/testfiles. It is not advised that you run this test until you have done your own testing, otherwise you may be overwhelmed. But once you think you have the project complete, do the following: create a directory ~/cop4530/proj2/testbench and copy the script ~cop4530p/spring10/proj2/testfiles/check into that directory. Then change permissions on check to executable. Be sure to read the script first to be sure that there are no name conflicts which could destroy your proj2 files. To be safe, also make a backup copy of all of your critical files (such as the deliverables!) Then execute the script and hold on.