CSC 250 – Project #4
Fall 2014

Goals: Implement and test a hash table in Java.

Part 1: Project Summary (Due: 5pm December 5th, 2014; Drop dead date: None)

Read in a large data set consisting of part of an Apache Web Server log file. Process and store in a hash table and permit the user to enter queries on the processed data.

Part 2: Background

Each time you request a web page or a component of a page (JavaScript file, cascading style sheet, image, etc.) you are making a single request to attempt to transfer one asset from the web server to your machine via the web browser. A single page that you wish to browse can result in your web browser making 100 requests to the web server for all of the components of the page.

Each request to the server is logged by the server (information about the request written to a log file) for subsequent administrator processing. For example, this information can tell us which pages are being read, what browsers are in use, and much, much more. Log entries can be configured to show a wide variety of information, depending on your needs. Our data for this project comes from the former computer science department web server, spanning 12 days in February 2007. The data file can be accessed via `http://s3.amazonaws.com/depasquale/datasets/access_log`

The lines of data each have the same basic structure as follows. Each item is listed as `<itemname>` and is separated by whitespace (usually a single space character). For some items, there is only a dash character present. We will disregard those fields.

```
<ip> - - [datetime] "<command>" <responsecode> <numbytes> "<referringpage>" "useragent"
```

For example, one such line might look like the following. Note that the line is so long it wraps. There is no new line character following the referring page field ("-"). The only new line character appears following the user agent.

```
74.6.72.231 - - [01/Feb/2007:04:06:37 -0500] "GET /msdn.php HTTP/1.0" 200 14844 "-"
"Mozilla/5.0 (compatible; Yahoo! Slurp; http://help.yahoo.com/help/us/ysearch/slurp)"
```

The following table will further explain the fields to you.

Field name	Description
ip	The Internet Protocol (IP) address of the user making the request.
datetime	The date and time (with the time zone of the server) that the request was received.
command	The command requested by the user's agent (browser). Generally speaking you will see just GET and POST commands, though others are supported. Following the name of the command is the resource (in the case of GET/POST), and the protocol type and version (HTTP/1.0).
responsecode	The status code of the request (was it successful, and error, the page is moved, etc.) For more information, see http://goo.gl/zh6AA
numbytes	The size of the resource returned as a number of bytes, or a '-' if there was a problem fulfilling the request (or no bytes were returned).
referringpage	The page that linked to the page or resource being requested, or the page the user was last viewing prior to the current request.
useragent	The details on the software making the request (web browser name/type, operating system).

Part 3: Processing

For this project, you are to:

1. Download the input file (see above) (the actual download will be executed from the ant build file, so the program can attempt to open a file that it expects to see present) and save it as 'access_log.inp' in the project's directory where the other code is placed. Read the data directly from the input file and extract the following fields:
    a. date and time of the request,
    b. the resource requested,
    c. status code, and
    d. number of bytes
2. Create an **`HTTPLog`** object that encapsulates the storage of a resource name, date and status code.
3. Your object should also store/count:
    a. the sum total number of bytes for the resource across all requests, and
    b. a counter to know how many times the resource was requested.

    Note that the sum total is not the resource size multiplied by the number of times the resource was requested, since request parameters can impact the size of the result returned (see discussion below on resource names).
4. Insert the object into a hash table that has 75 buckets and which utilizes linked chaining to handle collisions. While you will create/define the collection class (**`LinkedChainedHashtable<T>`**), your cells should each reference a **`LinkedList<T>`** object (from the API) to create the chain. Your hash algorithm should use the base resource name including any and all directory names, directory separators and file extensions (discussed further below) as the hash key.

    When processing the input file, you should immediately create an **`HTTPLog`** object if the requested resource you are processing is not already in the hash table. When processing a requested resource that does already exist, you should fetch and update the corresponding **`HTTPLog`** object. Here, updating means updating
    - the number of bytes served for the resource,
    - the most recent date and time of the request for this resource, and
    - the counter of the number of times the resource was requested.

    Do not populate an array of **`HTTPLog`** objects while processing the input file and then insert them into the hashtable when the input file is exhausted. The point of this project is to work with a hashtable; this includes insertions, fetches, and updates on the objects within the table.
5. Once the hash table is loaded, the program should loop continuously allowing the user to enter commands until they enter the **`QUIT`** command to exit the program.
6. The set of valid commands, and associated actions are as follows.
    a. **`DETAIL <resource>`** - lists the details of the requested resource. That is:
        i. the name of the resource,
        ii. the status code of the last request for this resource,
        iii. the last date/time the resource was requested,

iv. the total number of bytes served in requesting this resource (the sum total of the number of bytes over all requests), and

v. the total number of times this resource was requested.

The data printed by the result of executing this command should be properly labeled so that the user can identify the output values. If the element was not served, an informative message is printed to the user. User-input examples:

```
DETAIL /index.php
DETAIL /dir/dir2/filename.jpg
DETAIL /
```

b. **TOPTENSERVED** – lists, in descending order, the top ten most requested resources (based on the counter of how many times a resource was served). Your output should include the resource names and count of the number of times served.

c. **TOPTENSIZE** – lists, in descending order, the top ten largest resources (based on the sum total size served of the resource). Your output should include the resource names and the total sum of size for the resources.

d. **CHAIN** – lists the bucket number, length, and resource names associated with the longest chain in the hash table

e. **QUIT** – quits the program

Part 4:Design

The design of your hashtable shall be of your own. No existing code will be provided or should be used (except for the **LinkedList** class from the API and any exceptions that the **LinkedList** depends upon). Your implementation shall adhere to the following ADT interface, which you shall also provide (code). The interface summary is provided below. Your implementing class should be named **LinkedChainedHashtable<T>** and uses (aggregates) **LinkedList<T>** objects from the API to support chaining the elements in cases of collisions.

Name: **HashTableADT<T>**

Method	Description
`public boolean addElement (T element)`	Adds the specified element to the hashtable. Returns true if the element was successfully added to the table, returns false if there was a collision and the element was not added to the table.
`public T find (T element)`	Locates a matching element in the hashtable that corresponds (matches) the specified element.
`public int size()`	Returns the size of the hashtable (number of elements contained in the table and chains).
`public boolean isEmpty()`	Returns a true value if the hashtable contains no elements, false otherwise.
`public String toString()`	Returns a string representation of this hashtable. (Should contain basic properties of the hashtable [e.g. number of elements, number of buckets, empty state])
`public T[] getElements()`	Returns an array of elements from the entire hashtable.

Exception Handling

All exceptions related to I/O should append an informative message regarding the issue to a file named '*errors.txt*'. Do not propagate any I/O exception out of the location where it occurs and do not let it get to the main method and terminate the program. All catch clauses (I/O and otherwise) should produce an informative message to the errors.txt file (none of the catch clauses shall be without code) and

Comparable

The **HTTPLog** object should implement the **Comparable** interface such that it supports comparing similar objects. The **compareTo** method should be written such that the resource name is the basis of the comparison.

Note that the **find** method of the hashtable should be written to operate on elements of type **T**. Since **T** can be any type of object, you will need to cast objects in the hashtable to be of the **Comparable** type before comparing it to the target. Consider using the **instanceof** operator to check that the cast will occur correctly:

```
Comparable foo;
int result = 0;
if ((data[i] instanceof Comparable) {
      Comparable bar = (Comparable) data[i];
      result = foo.compareTo(bar);
}
return result;
```

Packaging

You should package your code as follows:

File	Package Name
HashtableADT.java	jsjf
LinkedChainedHashtable.java	jsjf
Driver.java	edu.tcnj.csc250
HTTPLog.java	edu.tcnj.csc250
<any other user-defined code that is not supporting the hashtable>	edu.tcnj.csc250

Hashcode and Hashing Function

Your **HTTPLog** object should override the **hashcode** method (that is inherited from the **Object** class) to return an **int** value. To calculate your **int**, sum up the integer values of all of the characters of the resource name for the object, divide by 11, and add 2014.

When applying the hashing function to the object being hashed, use the **hashcode** value provided by the **hashcode** method described above. In terms of a hashing function, create your own implementation of a function that uses at least one folding method followed by the division method to generate the insertion index.

Part 5: Resource Names

When you request a resource via a web request, there are a number of forms it can take. Like calling a method, you can also pass it parameters to modify the behavior of the page that response to the request ('resources' are just information passed back to the user, the resources can vary based on the requested name and/or the parameters it receives).

When using resource names (in the examples below the resource name is bolded to more easily show it) in this program, we will remove the parameters and only use the base name. Base names, however, can take on several forms. For example, assume we have the following entry in the log file:

```
74.6.72.231 - - [01/Feb/2007:04:06:37 -0500] "GET /msdn.php HTTP/1.0" 200 14844 "-"
"Mozilla/5.0 (compatible; Yahoo! Slurp; http://help.yahoo.com/help/us/ysearch/slurp)"
```

The name of this resource is "**/msdn.php**" which is a file located in the web server's root directory.

Other resource requests can be more complex. For example, consider the following line.

```
203.144.144.163 - - [01/Feb/2007:04:06:28 -0500] "GET
/gallery/main.php?g2_view=comment.AddComment&g2_itemId=664&g2_return=http%3A%2F%2Fcs.tcnj
.edu%2Fgallery%2Fv%2Fevents%2Falbum02%2Fcontests%2FprogrammingContest05%2F%3Fg2_page%3D2%
26g2_GALLERYSID%3Db02832e6007d7e6f1eda8897cfea98bf&g2_GALLERYSID=b02832e6007d7e6f1eda8897
cfea98bf&g2_returnName=album HTTP/1.1" 200 11313
"http://cs.tcnj.edu/gallery/v/events/album02/contests/programmingContest05/?g2_page=2"
"Mozilla/5.0 (Windows; U; Windows NT 5.0; en-US; rv:1.0rc2) Gecko/20020510"
```

Here, the request is bolded, but the base name is everything up to (but not including) the "?" that starts the list of parameters. Therefore, our resource's base name is "**/gallery/main.php**".

Parameters, so you know, are listed in name-value pairs and are separated by the "&" sign. Thus "**g2_view**" is the name of a parameter, and "**comment.AddComment**" is the value of that parameter. This is really no different than passing a value to a method, and defining a local variable that has a given name that will be used to store the passed value.

If you see a line such as:

```
202.108.11.106 - - [01/Feb/2007:04:05:28 -0500] "GET / HTTP/1.1" 200 15010 "-"
"Baiduspider+(+http://www.baidu.com/search/spider.htm)"
```

The resource name here is simply "/", but is assumed to translate into "/index.html". So please store this as "/index.html" and assume that the user may search on "/index.html" in the run time commands for this program.

Part 6: Specifics for the Delivery of this Project

The following rules apply to the completion of this project.

1. Create and fully test a small ANT build file (named build.xml). The build file should contain the following targets:
    a. **compile** - compiles the project source code to the same directory in which the original source code file is located (please do not place compiled .class files in a different directory).
    b. **clean** – removes the .class, the META-INF directory and its contents, and the output file from the build area. This target and its tasks should not fail if the file/dir is not present (output file, META-INF, etc.)
    c. **jar** – creates the jar file named <username>.jar for uploading (includes only the build file and source code files, excludes the .class and output/error files)
    d. **authors** – prints the names of the program's authors.
    e. **download** – Downloads the input file to the same directory in which the build file resides.
    f. **run** – executes the program. This target depends on the **authors**, **download**, and **compile** targets.
2. To submit your project, jar up the source code file(s). The task of organizing and creating the jar file for each deliverable will be up to you. You should always check your work, as this file is what I will be

grading. You want to ensure that only the files I request are contained in the jar file. DO NOT INCLUDE THE INPUT FILE OR ANY OTHER FILES AS PART OF THE JAR FILE!

You may submit the project any number of times until the due date. Only the last submission will be downloaded, examined, and graded by me.

Late points, if applicable will be deducted for projects turned in starting 1 minute after the due date (that's why they are called due dates!) Following the drop-dead date for a project, no solutions will be graded for the project. At such a point in time, you will receive a zero grade for the work if you have not uploaded a solution.

3. The class commenting and formatting requirements are in effect for this project. Be sure to refer to that document to review the details of the types of comments we seek.

Part 7: Resources

- The Java API documentation is located at: http://docs.oracle.com/javase/6/docs/api/
- Ant Documentation: http://ant.apache.org/manual/index.html

Part 8: Advice and Disclaimer

Plan your schoolwork and life accordingly. Many students don't adequately plan their work schedules and attempt to finish programs at the last minute. Doing so usually introduces bugs/problems into the solution. Consider how much time you will require to adequately test your solution for boundary cases and bugs. Failure to plan on your part does not constitute an emergency on my part!

I do not give individual extensions for projects, and rarely give class extensions for work. You have nearly a week and a half to complete this project. I suggest you start your design as soon as possible. Consider what you need to build to solve the problem at hand. Programming involves designing a solution, implementing your solution, and testing your solution.

Part 9: Do You Need Help?

Are you stuck? Ask yourself if you have planned the design of your project. Have you asked questions in class? Did you attend the lectures, and labs? Have you come to see me? Like a textbook or web site, I am a resource that I expect you to use throughout the course of the semester. If you are stuck with your project, I strongly suggest you make an appointment to come see me in person. Generally, I can offer some guidance or help to get you back on the path.

When you come to see me, plan on bringing your source code to me on a thumb drive. Do not email me your source code, I do not provide email-based help for coursework. I prefer that you make time to visit me in my office where we can sit down and discuss your source code together. I can ask you questions and you can tell me why you coded something the way you did. We can use my whiteboard to draw large diagrams and talk about good solutions and designs. This is not something that can be done easily through email.