**CMSC204**

**Assignment #4**

**Assignment Description**

Most data is stored in databases, for ready access and organization. Our course data is backed up by IT in databases which makes our data easy to access and use.

Write a program that creates a database of courses. It will either read from a file of courses or allow the user to add one course at a time. Upload the initial files and your working files in the repository in GitHub you created in Lab 1, in a directory named Assignment4. Take a screenshot of your repo and post the assignment to the Assignment 4 dropbox.

**Concepts tested by this assignment**

Hash Table,

Link List,

hash code, buckets/chaining,

exception handling,

read/write files using FileChooser

**Classes**

**Data Element -** *CourseDBElement*

*CourseDBElement* implements Comparable interface and consists of five attributes: the Course ID (a String), the CRN (an int), the number of credits (an int), the room number (a String), and the instructor name (a String). Normally the *CourseDBElement* will be an object consisting of these five attributes and is referred to as a CDE.

**Data Structure -** *CourseDBStructure*

Referred as CDS, Implements the *CourseDBStructureInterface* that is provided.

You will be implementing a hash table with buckets. Each bucket will be an array of linked lists of *CourseDBElements*. Each *CourseDBElement* object will have a hash code that is calculate based on the CRN, since the CRN is unique for courses. Note that the CRN is an int, and the tests require the hashcode of a string, so you will need to coerce it to a *String* and take the hash code of the resulting string. The *add* method of *CourseDBStructure* will take a *CourseDBElement object* and add it to the data structure based on the calculated hashcode. If a linked list at the relevant hash code doesn’t exist (the bucket is empty), create a LinkedList with the first element being the *CourseDBElement* object and add it to the HashTable. If the LinkedList already exists, add the *CourseDBElement* object to the existing list.

Two constructors for the *CourseDBStructure* will be required, one that takes in an integer that is the estimated number of courses, the other is used for testing purposes. The comments in the *CourseDBStructureInterface* (provided) should help you figure out how to set the length of the hash table.

**Note**: In hash table structure with buckets the *load factor* can be larger than one and represents the average number of elements stored in each list, assuming that the hash function distributes elements uniformly over all positions. For this assignment use a *load factor* of **1.5**.

This class has two constructors:

1. A constructor that takes in an integer **n** which represents the estimated number of

courses and determines the size of the hash table by finding a 4K+3 prime just greater than **n /loading factor**.

Example: if n is 500 courses, then 500/1.5 = 333, The next 4K+3 prime over 333 is 347. So, you would set the table a length to 347.

1. A Constructor for testing purposes. This constructor will take a string “Testing” and an int for the hashtable size. This is used only for testing.

**Data Manager -** *CourseDBManager*

Implements the *CourseDBManagerInterface* that is provided.

The data manager allows the user to read the courses from a file or to enter the data by hand and uses an Alert to print out the database elements. The input is read from a file or read from the textfields and is added to the data structure through the add method. The add method uses the *CDS* ‘s *add* method. The *CourseDBManager* is also referred to as a CDM.

**Exception Classes**

IOException – created and thrown when user selects an input file that cannot be read or attempting to retrieve a CDE that does not exist in the DB.

**GUI Driver (provided)**

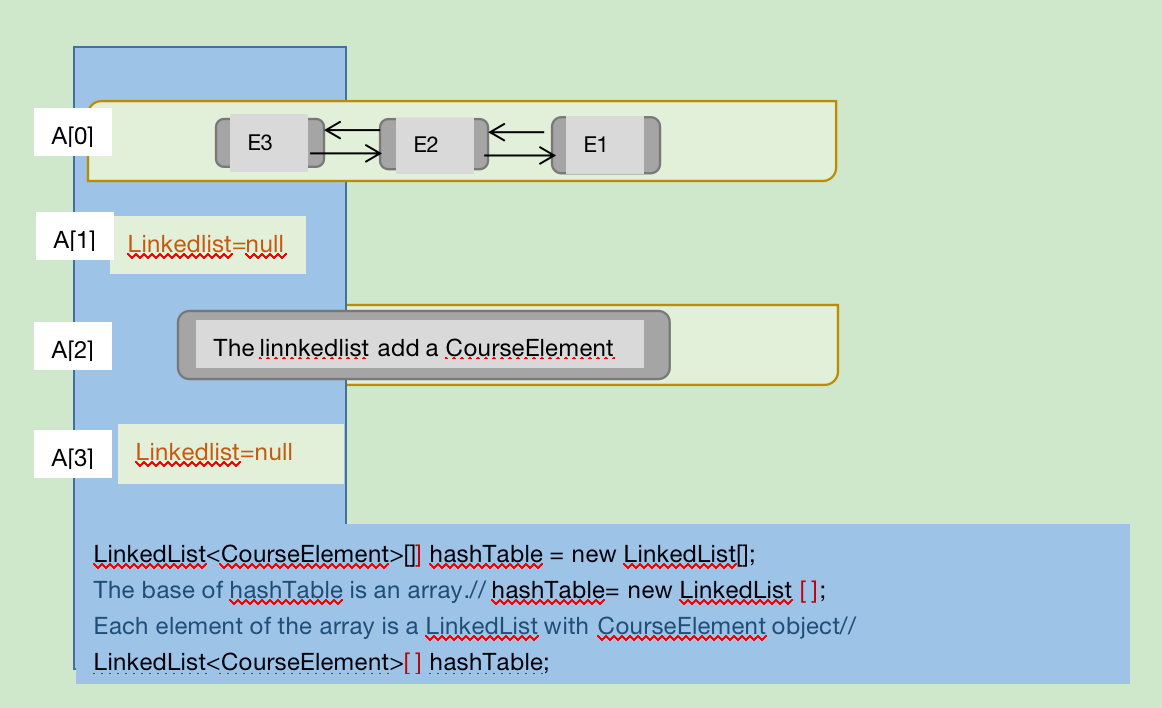
* User will only create a course database once they have entered an input file or entered one or more sets of attributes.
* Buttons and textfields are grayed out if they are not relevant at the current time.
* A *FileChooser* is used to select the input and output files.
* Inform the user if there is an error with the input file.
* Use exception handling for the validity of the files.
* A way is provided for the user to “clear” the text fields.
* A way is provided for the user to select a CRN and retrieve the corresponding course.

**Testing**

Create a JUnit Test - CourseDBManager\_STUDENT\_Test.

**Assignment Details**

There will be two ways to create a course database. The first requires a document to be read from an input file. The second reads the input from five textfields. Once there is data in the database, the GetCourse button will be enabled to allow you to see the data. See the example below.



**Examples**

Example of creating a Course database from an input file

A screenshot of a cell phone

Description automatically generated

Select the input file button and navigate to the file.

A screenshot of a cell phone

Description automatically generated

Use ShowDB button to display the CDEs that were read:

A screenshot of a cell phone

Description automatically generated

Example of Creating a CDE from text fields. First select the other radio button, then fill in the textfields, then select the “Add to DB” button.

A screenshot of a cell phone

Description automatically generated

Select the Show DB button to see the resulting CDEs.

A screenshot of a cell phone

Description automatically generated

Example of retrieving a CDE. First select the *GetCourse* button. Three components at the bottom of the screen will become enabled. Then fill in the CRN and select the FindCourse button to find the applicable course from the database.

A screenshot of a cell phone

Description automatically generated

The general design is shown here to guide you in formulating your own design:

A screenshot of a cell phone

Description automatically generated

**Deliverables**

**Deliverables / Submissions:**

Design: UML class diagram with algorithm (pseudo-code) for methods

Implementation: Submit a compressed file containing the follow (see below): The Java application (it must compile and run correctly); Javadoc files in a directory; a write-up as specified below. Be sure to review the provided project rubric to understand project expectations. The write-up will include:

* UML diagram
* In three or more paragraphs, highlights of your learning experience

**Deliverable format:** The above deliverables will be packaged as follows. Two compressed files in the following formats:

* LastNameFirstName\_Assignment4\_Complete.zip, a compressed file in the zip format, with the following:
  + Write up (Word document) - reflection paragraphs
  + GitHub repository with java files from Assignment 4.
  + Final Design: UML Diagram - latest version (Word or image)
  + doc (directory) – Javadoc
    - * subdirectories (as is)
      * File1.html (example)
      * File2.html (example)
  + src (directory) 
    - * File1.java (example)
      * File2.java (example)
* LastNameFirstName\_Assignment4\_Moss.zip, a compressed file containing one or more Java files:
  + File1.java (example)
  + File2.java (example)

This folder should contain Java source files only

ONLY SUBMIT THE FILES THAT YOU IMPLEMENTED OR UPDATED FOR this ASSIGNMENT.  YOU DO NOT NEED TO SUBMIT THE FILES THAT ARE ALREADY PROVIDED FOR YOU.