Geographic Information System

Geographic information systems organize information pertaining to geographic features and provide various kinds of access to the information. A geographic feature may possess many attributes (see below). In particular, a geographic feature has a specific location. There are a number of ways to specify location. For this project, we will use latitude and longitude, which will allow us to deal with geographic features at any location on Earth. A reasonably detailed tutorial on latitude and longitude can be found in the Wikipedia at en.wikipedia.org/wiki/Latitude and en.wikipedia.org/wiki/Longitude.

The GIS record files were obtained from the website for the USGS Board on Geographic Names (www.usgs.gov/core-science-systems/ngp/board-on-geographic-names/download-gnis-data). The file begins with a descriptive header line, followed by a sequence of GIS records, one per line, which contain the following fields in the indicated order:

Figure 1: Geographic Data Record Format

Name	Type	Length/ Decimals	Short Description
Feature ID	Integer	10	Permanent, unique feature record identifier and official feature name
Feature Name	String	120	
Feature Class	String	50	See Figure 3 later in this specification
State Alpha	String	2	The unique two letter alphabetic code and the unique two number code for a US State
State Numeric	String	2	
County Name	String	100	The name and unique three number code for a county or county equivalent
County Numeric	String	3	
Primary Latitude DMS	String	7	The official feature location *DMS-degrees/minutes/seconds* *DEC-decimal degrees.* *Note: Records showing "Unknown" and zeros for the latitude and longitude DMS and decimal fields, respectively, indicate that the coordinates of the feature are unknown. They are recorded in the database as zeros to satisfy the format requirements of a numerical data type. They are not errors and do not reference the actual geographic coordinates at 0 latitude, 0 longitude.*
Primary Longitude DMS	String	8	
Primary Latitude DEC	Real Number	11/7	
Primary Longitude DEC	Real Number	12/7	
Source Latitude DMS	String	7	Source coordinates of linear feature only (Class = Stream, Valley, Arroyo) DMS-degrees/minutes/seconds DEC-decimal degrees. *Note: Records showing "Unknown" and zeros for the latitude and longitude DMS and decimal fields, respectively, indicate that the coordinates of the feature are unknown. They are recorded in the database as zeros to satisfy the format requirements of a numerical data type. They are not errors and do not reference the actual geographic coordinates at 0 latitude, 0 longitude.*
Source Longitude DMS	String	8	
Source Latitude DEC	Real Number	11/7	
Source Longitude DEC	Real Number	12/7	

Elevation (meters)	Integer	5	Elevation in meters above (-below) sea level of the surface at the primary coordinates
Elevation (feet)	Integer	6	Elevation in feet above (-below) sea level of the surface at the primary coordinates
Map Name	String	100	Name of USGS base series topographic map containing the primary coordinates.
Date Created	String		The date the feature was initially committed to the database.
Date Edited	String		The date any attribute of an existing feature was last edited.

Notes:

- See https://geonames.usgs.gov/domestic/states_fileformat.htm for the full field descriptions.
- The type specifications used here have been modified from the source (URL above) to better reflect the realities of your programming environment.
- Latitude and longitude may be expressed in DMS (degrees/minutes/seconds, 0820830W) format, or DEC (real number, -82.1417975) format. In DMS format, latitude will always be expressed using 6 digits followed by a single character specifying the hemisphere, and longitude will always be expressed using 7 digits followed by a hemisphere designator.
- Although some fields are mandatory, some may be omitted altogether. Best practice is to treat every field as if it may be left unspecified. Certain fields are necessary in order to index a record: the feature name and the primary latitude and primary longitude. If a record omits any of those fields, you may discard the record, or index it as far as possible.

In the GIS record file, each record will occur on a single line, and the fields will be separated by pipe (`'|'`) symbols. Empty fields will be indicated by a pair of pipe symbols with no characters between them. See the posted `VA_Monterey.txt` file for many examples.

GIS record files are guaranteed to conform to this syntax, so there is no explicit requirement that you validate the files. On the other hand, some error-checking during parsing may help you detect errors in your parsing logic.

The file can be thought of as a sequence of bytes, each at a unique offset from the beginning of the file, just like the cells of an array. So, each GIS record begins at a unique offset from the beginning of the file.

Line Termination

Each line of a text file ends with a particular marker (known as the line terminator). In MS-DOS/Windows file systems, the line terminator is a sequence of two ASCII characters (CR + LF, `0X0D0A`). In Unix systems, the line terminator is a single ASCII character (LF). Other systems may use other line termination conventions.

Why should you care? Which line termination is used has an effect on the file offsets for all but the first record in the data file. As long as we're all testing with files that use the same line termination, we should all get the same file offsets. But if you change the file format (of the posted data files) to use different line termination, you will get different file offsets than are shown in the posted log files. Most good text editors will tell you what line termination is used in an opened file, and also let you change the line termination scheme.

All that being said, this project is not auto-graded, and the grading of correctness will depend on whether you report the correct search results, not on the file offsets you report.

Figure 2: Sample Geographic Data Records

Note that some record fields are optional, and that when there is no given value for a field, there are still delimiter symbols for it.

Also, some of the lines are "wrapped" to fit into the text box; lines are never "wrapped" in the actual data files.

```
FEATURE_ID|FEATURE_NAME|FEATURE_CLASS|STATE_ALPHA|STATE_NUMERIC|COUNTY_NAME|COUNTY_NUMERIC|PRIMARY_LAT_DMS|PRIM_LONG_DMS|PRIM_LAT_DEC|PRIM_LON
G_DEC|SOURCE_LAT_DMS|SOURCE_LONG_DMS|SOURCE_LAT_DEC|SOURCE_LONG_DEC|ELEV_IN_M|ELEV_IN_FT|MAP_NAME|DATE_CREATED|DATE_EDITED
1479116|Monterey Elementary School|School|VA|51|Roanoke (city)|770|371906N|0795608W|37.3183753|-
79.9355857|||||323|1060|Roanoke|09/28/1979|09/15/2010
1481345|Asbury Church|Church|VA|51|Highland|091|382607N|0793312W|38.4353981|-79.5533807|||||818|2684|Monterey|09/28/1979|
1481852|Blue Grass|Populated Place|VA|51|Highland|091|383000N|0793259W|38.5001188|-79.5497702|||||777|2549|Monterey|09/28/1979|
1481878|Bluegrass Valley|Valley|VA|51|Highland|091|382953N|0793222W|38.4981745|-79.539492|382601N|0793800W|38.4337309|-
79.6333833|759|2490|Monterey|09/28/1979|
1482110|Buck Hill|Summit|VA|51|Highland|091|381902N|0793358W|38.3173452|-79.5661577|||||1003|3291|Monterey SE|09/28/1979|
1482176|Burners Run|Stream|VA|51|Highland|091|382509N|0793409W|38.4192873|-79.5692144|382531N|0793538W|38.4252778|-
79.5938889|848|2782|Monterey|09/28/1979|
1482324|Mount Carlyle|Summit|VA|51|Highland|091|381556N|0793353W|38.2656799|-79.5647682|||||698|2290|Monterey SE|09/28/1979|
1482434|Central Church|Church|VA|51|Highland|091|382953N|0793323W|38.4981744|-79.5564371|||||773|2536|Monterey|09/28/1979|
1482557|Claylick Hollow|Valley|VA|51|Highland|091|381613N|0793238W|38.2704021|-79.5439343|381733N|0793324W|38.2925|-
79.5566667|573|1880|Monterey SE|09/28/1979|
1482785|Crab Run|Stream|VA|51|Highland|091|381707N|0793144W|38.2854018|-79.528934|381903N|0793415W|38.3175|-79.5708333|579|1900|Monterey
SE|09/28/1979|
1482950|Davis Run|Stream|VA|51|Highland|091|381824N|0793053W|38.3067903|-79.5147671|382057N|0793505W|38.3491667|-79.5847222|601|1972|Monterey
SE|09/28/1979|
1483281|Elk Run|Stream|VA|51|Highland|091|382936N|0793153W|38.4934524|-79.5314362|383121N|0793056W|38.5226185|-
79.5156027|757|2484|Monterey|09/28/1979|
1483492|Forks of Waters|Locale|VA|51|Highland|091|382856N|0793031W|38.4823417|-79.5086575|||||705|2313|Monterey|09/28/1979|
1483527|Frank Run|Stream|VA|51|Highland|091|382953N|0793310W|38.4981744|-79.5528258|383304N|0793341W|38.5512285|-
79.5614381|780|2559|Monterey|09/28/1979|
1483647|Ginseng Mountain|Summit|VA|51|Highland|091|382850N|0793139W|38.480675|-79.527547|||||978|3209|Monterey|09/28/1979|
1483860|Gulf Mountain|Summit|VA|51|Highland|091|382940N|0793103W|38.4945636|-79.5175468|||||1006|3300|Monterey|09/28/1979|
1483916|Hamilton Chapel|Church|VA|51|Highland|091|381740N|0793707W|38.2945677|-79.6186591|||||823|2700|Monterey SE|09/28/1979|
1484097|Highland High School|School|VA|51|Highland|091|382426N|0793444W|38.4071387|-79.5789333|||||879|2884|Monterey|09/28/1979|09/15/2010
1484099|Highland Wildlife Management Area|Park|VA|51|Highland|091|381905N|0793439W|38.3181785|-79.577547|||||954|3130|Monterey SE|09/28/1979|
. . .
```

Assignment

You will implement a system that indexes and provides search features for a file of GIS records, as described above.

Your system will build and maintain several in-memory index data structures to support these operations:

- Importing new GIS records into the database file
- Retrieving data for all GIS records matching given geographic coordinates
- Retrieving data for all GIS records matching a given feature name and state
- Retrieving data for all GIS records that fall within a given (rectangular) geographic region
- Displaying the in-memory indices in a human-readable manner

You will implement a single software system in Java to perform all system functions.

Program Invocation

The program will take the names of three files from the command line, like this:

```
java GIS <database file name> <command script file name> <log file name>
```

Note that this implies your main class must be named `GIS`, and must be in the default Java package.

The database file should be created as an empty file; note that the specified database file may already exist, in which case the existing file should be truncated or deleted and recreated. If the command script file is not found the program should write an error message to the console and exit. The log file should be rewritten every time the program is run, so if the file already exists it should be truncated or deleted and recreated.

System Overview

The system will create and maintain a GIS database file that contains all the records that are imported as the program runs. The GIS database file will be empty initially. All the indexing of records will be done relative to this file.

There is no guarantee that the GIS record file will not contain two or more distinct records that have the same geographic coordinates. In fact, this is natural since the coordinates are expressed in the usual DMS system. So, we cannot treat geographic coordinates as a primary (unique) key.

The GIS records will be indexed by the Feature Name and State (abbreviation) fields. This *name index* will support finding offsets of GIS records that match a given feature name and state abbreviation.

The GIS records will also be indexed by geographic coordinate. This *coordinate index* will support finding offsets of GIS records that match a given primary latitude and primary longitude.

The system will include a *buffer pool*, as a front end for the GIS database file, to improve search speed. See the discussion of the buffer pool below for detailed requirements. When performing searches, retrieving a GIS record from the database file must be managed through the buffer pool. During an import operation, when records are written to the database file, the buffer pool will be bypassed, since the buffer pool would not improve performance during imports.

When searches are performed, complete GIS records will be retrieved from the GIS database file that your program maintains. The only complete GIS records that are stored in memory at any time are those that have just been retrieved to satisfy the current search, or individual GIS records created while importing data or GIS records stored in the buffer pool.

Aside from where specific data structures are required, you may use any suitable Java library containers you like.

Each index should have the ability to write a nicely-formatted display of itself to an output stream.

Name Index Internals

The *name index* will use a hash table for its physical organization. Each hash table entry will store a feature name and state abbreviation (separately or concatenated, as you like) and the file offset(s) of the matching record(s). Since each GIS record occupies one line in the file, it is a trivial matter to locate and read a record given nothing but the file offset at which the record begins.

Your table will use chaining to resolve collisions, with a contiguous physical structure (physical array) to store the chains. The initial size of the table will be 1024, and the table will resize itself automatically, by doubling its size whenever the table becomes 70% full. Obviously, you should base this on a slight modification of the hash table you implemented for J3.

You will use the same hash function that was supplied for J3, and apply it to a concatenation of the feature name and state abbreviation field of the data records. Precisely how you form the concatenation is up to you.

You must be able to display the contents of the hash table in a readable manner.

Coordinate Index Internals

The coordinate index will use a *bucket* PR quadtree for the physical organization. In a bucket PR quadtree, each leaf stores up to K data objects (for some fixed value of K). Upon insertion, if the added value would fall into a leaf that is already full, then the region corresponding to the leaf will be partitioned into quadrants and the K+1 data objects will be inserted into those quadrants as appropriate. As is the case with the regular PR quadtree, this may lead to a sequence of partitioning steps, extending the relevant branch of the quadtree by multiple levels. In this project, K will probably equal 4, but I reserve the right to specify a different bucket size with little notice, so this should be easy to modify.

The index entries held in the quadtree will store a geographic coordinate and a collection of the file offsets of the matching GIS records in the database file. Obviously, this is derived directly from your solution to J2.

Note: do not confuse the bucket size with any limit on the number of GIS records that may be associated with a single geographic coordinate. A quadtree node can contain index objects for up to K different geographic coordinates. Each such index object can contain references to an unlimited number of different GIS records.

The PR quadtree implementation should follow good design practices, and its interface should be somewhat similar to that of the BST. You are expected to implement different types for the leaf and internal nodes, with appropriate data membership for each, and an abstract base type from which they are both derived. Of course, these were all requirements for the related minor project.

You must be able to display the PR quadtree in a readable manner. PR quadtree display code is given in the course notes. The display must clearly indicate the structure of the tree, the relationships between its nodes, and the data objects in the leaf nodes.

Buffer Pool Details

The buffer pool for the database file should be capable of buffering up to 15 records, and will use LRU replacement. You may use any structure you like to organize the pool slots; however, since the pool will have to deal with record replacements, some structures will be more efficient (and simpler) to use. You may use any classes from the Java library you think are appropriate.

It is up to you to decide whether your buffer pool stores interpreted or raw data; i.e., whether the buffer pool stores GIS record objects or just strings.

You must be able to display the contents of the buffer pool, listed from MRU to LRU entry, in a readable manner. The order in which you retrieve records when servicing a multi-match search is not specified, so such searches may result in different orderings of the records within the buffer pool. That is OK.

A Note on Coordinates and Spatial Regions

It is important to remember that there are fundamental differences between the notion that a geographic feature has specific coordinates (which may be thought of as a point) and the notion that each node of the PR quadtree corresponds to a particular sub-region of the coordinate space (which may contain many geographic features).

In this assignment, coordinates of geographic features are specified as latitude/longitude pairs, and the minimum resolution is one second of arc. Thus, you may think of the geographic coordinates as being specified by a pair of integer values.

On the other hand, the boundaries of the sub-regions are determined by performing arithmetic operations, including division, starting with the values that define the boundaries of the "world". Unless the dimensions of the world happen to be powers of 2, this can quickly lead to regions whose boundaries cannot be expressed exactly as integer values. You should use floating-point values to represent region boundaries, when carrying out splitting operations and quadtree traversals.

Your implementation should view the boundary between regions as belonging to one of those regions. The choice of a particular rule for handling this situation is left to you. The specification for the PR quadtree project describes how I made that decision, but there is absolutely no requirement that you follow the same approach.

When carrying out a region search, you must determine whether the search region overlaps with the region corresponding to a subtree node before descending into that subtree. The Java libraries include a Rectangle class which could be (too) useful. You may make use of the Rectangle class, but you will be penalized 10% if your submitted solution makes use of any of the following Rectangle methods: contains(), intersection(), and intersects(). Note though, that it is acceptable to make use of those methods during development, but you must implement your own versions of them in your final submission.

Other System Elements

There should be an overall controller that validates the command line arguments and manages the initialization of the various system components. The controller should hand off execution to a command processor that manages retrieving commands from the script file, and making the necessary calls to other components in order to carry out those commands.

Naturally, there should be a data type that models a GIS record.

There may well be additional system elements, whether data types or data structures, or system components that are not mentioned here. The fact no additional elements are explicitly identified here does not imply that you will not be expected to analyze the design issues carefully, and to perhaps include such elements.

Aside from the command-line interface, there are no specific requirements for interfaces of any of the classes that will make up your GIS; it is up to you to analyze the specification and come up with an appropriate set of classes, and to design their interfaces to facilitate the necessary interactions. It is probably worth pointing out that an index (e.g., a geographic coordinate index) should not simply be a naked container object (e.g, quadtree); if that's not clear to you, think more carefully about what sort of interface would be appropriate for an index, as opposed to a container.

Command File

The execution of the program will be driven by a script file. Lines beginning with a semicolon character (`';'`) are comments and should be ignored. Blank lines are possible. Each line in the command file consists of a sequence of tokens, which will be separated by single tab characters. A line terminator will immediately follow the final token on each line. The command file is guaranteed to conform to this specification, so you do not need to worry about error-checking when reading it.

The first non-comment line will specify the world boundaries to be used:

`world<tab><westLong><tab><eastLong><tab><southLat><tab><northLat>`

> This will be the first command in the file, and will occur once. It specifies the boundaries of the coordinate space to be modeled. The four parameters will be longitude and latitudes expressed in DMS format, representing the vertical and horizontal boundaries of the coordinate space.
>
> It is certainly possible that the GIS record file will contain records for features that lie outside the specified coordinate space. Such records should be ignored; i.e., they will not be indexed.

Each subsequent non-comment line of the command file will specify one of the commands described below. One command is used to load records into your database from external files:

`import<tab><GIS record file>`

Add all the valid GIS records in the specified file to the database file. This means that the records will be appended to the existing database file, and that those records will be indexed in the manner described earlier. When the import is completed, log the number of entries added to each index, and the longest probe sequence that was needed when inserting to the hash table. (A valid record is one that lies within the specified world boundaries.)

Another command requires producing a human-friendly display of the contents of an index structure:

`debug<tab>[ quad | hash | pool ]`

Log the contents of the specified index structure in a fashion that makes the internal structure and contents of the index clear. It is not necessary to be overly verbose here, but you should include information like key values and file offsets where appropriate.

Another simply terminates execution, which is handy if you want to process only part of a command file:

`quit<tab>`

Terminate program execution.

The other commands involve searches of the indexed records:

`what_is_at<tab><geographic coordinate>`

For every GIS record in the database file that matches the given `<geographic coordinate>`, log the offset at which the record was found, and the feature name, county name, and state abbreviation. Do not log any other data from the matching records.

`what_is<tab><feature name><tab><state abbreviation>`

For every GIS record in the database file that matches the given `<feature name>` and `<state abbreviation>`, log the offset at which the record was found, and the county name, the primary latitude, and the primary longitude. Do not log any other data from the matching records.

`what_is_in<tab><geographic coordinate><tab><half-height><tab><half-width>`

For every GIS record in the database file whose coordinates fall within the closed rectangle with the specified height and width, centered at the `<geographic coordinate>`, log the offset at which the record was found, and the feature name, the state name, and the primary latitude and primary longitude. Do not log any other data from the matching records. The half-height and half-width are specified as seconds.

If a `<geographic coordinate>` is specified for a command, it will be expressed as a pair of latitude/longitude values, expressed in the same DMS format that is used in the GIS record files.

For all the commands, if a search results in displaying information about multiple records, you may display that data in any order that is natural to your design.

Sample command scripts, and corresponding log files, will be provided on the website. As a general rule, every command should result in some output. In particular, a descriptive message should be logged if a search yields no matching records.

Log File Description

Since this assignment will be graded by TAs, rather than script automation, the format of the output is left up to you. Of course, your output should be clear, concise, well labeled, and correct. You should begin the log with a few lines identifying yourself, and listing the names of the input files that are being used. No one will object if you use the posted log files as a model for formatting your own output, but you are certainly not expected to match the formatting of those exactly.

The remainder of the log file output should come directly from your processing of the command file. You are required to echo each comment line, and each command that you process to the log file so that it's easy to determine which command each section of your output corresponds to. Each command (except for `"world"`) should be numbered, starting with 1, and the output from each command should be well formatted, and delimited from the output resulting from processing other commands. A complete sample log will be posted shortly on the course website.

A zip file is posted, containing scripts, GIS data files, and reference log files you can use in testing. The grading will be done using files in exactly the same format (perhaps even including some of the posted files).

Administrative Issues

Submission

You will submit this assignment to the Curator System (read the *Student Guide*), where it will be archived for grading by a course TA.

For this assignment, you must submit a single zip file containing:

- all the Java source code files for your implementation (i.e., `.java` files); do not submit Java bytecode (class) files
- a plain text file, named `readme.txt`, containing the following information:
  - any special instructions that might help the TA in compiling your Java code from the command line
  - names of the files and line numbers where the TA can find the following elements:
    - your hash table implementation and the data type stored in the hash table
    - your PR quadtree implementation and the data type stored in the PR quadtree
    - your buffer pool implementation
    - your design for representing a GIS record
    - your feature name index
    - your location index

If you use packages in your implementation (and that's good practice), your zip file must include the correct directory structure for those packages, and your `GIS.java` file must be in the top directory when the tar file is unpacked. That's easy to verify by running "`unzip -l`" on the tar file you are planning to submit.

In order to correct submission errors and late-breaking implementation errors, you will be allowed up to five submissions for this assignment. By default, we will grade the last submission you make.

The Student Guide and link to the submission client can be found at: `http://www.cs.vt.edu/curator/`

Evaluation

The quality of the OO design in your solution will be evaluated; you are expected to identify and implement a sound collection of classes, whether the specification mentions them or not. The quality of your internal documentation will be evaluated. Finally, the correctness of your solution will be evaluated by executing your solution on a collection of test data files. Be sure to test your solution with all of the data sets that will be posted, since we will use a variety of data sets, including at least one very large data one (perhaps hundreds of thousands of records) in our evaluation.

The quality of your design and internal documentation will determine approximately 30% of your grade for this assignment. Pay attention to the design discussion in class for J1.

We will compile and test your solution on CentOS 8, using version 11.0.8 of the JDK. The choice of operating system should not matter with a Java program (write once, run everywhere, right?), but you should compile and test your solution on rlogin before submitting it.

Late submissions will be penalized at a rate of 10% per day or fraction thereof. There will also be a final deadline, after which we will not accept submissions.

Pedagogic Points

The goals of this assignment include, but are not limited to:

- implementation of a hash table generic in Java (as in J3)
- implementation of a bucket PR quadtree generic in Java (as in J2)
- implementation of a buffer pool in Java
- design of appropriate index classes to wrap the containers
- design of appropriate data objects to store in each index
- understanding how to navigate a file in Java
- understanding how to parse delimited strings in Java (based on your solutions to J1 and J3)
- creation of a sensible OO design for the overall system, including the identification of a number of useful classes not explicitly named in this specification
- implementation of such an OO design into a working system
- incremental testing of the basic components of the system in isolation
- satisfaction when the entire system comes together in good working order

Pledge

Each of your program submissions must be pledged to conform to the Honor Code requirements for this course. Specifically, you **must** include the pledge statement provided below. The pledge statement should be in the file that contains your main class (which must be named `GIS.java`).

```
//    On my honor:
//
//    - I have not discussed the Java language code in my program with
//      anyone other than my instructor or the teaching assistants
//      assigned to this course.
//
//    - I have not used Java language code obtained from another student,
//      or any other unauthorized source, including the Internet, either
//      modified or unmodified.
//
//    - If any Java language code or documentation used in my program
//      was obtained from another source, such as a text book or course
//      notes, that has been clearly noted with a proper citation in
//      the comments of my program.
//
//    - I have not designed this program in such a way as to defeat or
//      interfere with the normal operation of the grading code.
//
//    <Student's Name>
//    <Student's VT email PID>
```

We reserve the option of assigning a score of zero to any submission that is undocumented or does not contain this statement.

Change Log

Version	Date	Page	Change(s)
6.00	Mar 8		Base document
6.10	Mar 11	5	Changes to the hash table requirements to bring it in line with J3; other minor changes
6.20	April 5	8	Changes to the specification for the file to be submitted