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.

We will employ public data obtained from the Geographic Names Information System (geonames.usgs.gov). Each data file consists of a list of ASCII records, each describing a single geographic feature with many attributes.

More precisely, each GIS record may contain the following fields in the indicated order (all are mandatory unless indicated otherwise):

Figure 1: Geographic Data Fields

Significance	**Type/Format**	**Comments**
Feature ID number (FID)	non-negative integer	unique identifier for this geographic feature
State alphabetic code	two-characters	US postal code abbreviation
Feature name	string	standard name of feature
Feature type	string	descriptive classification of feature
County name	string	county in which feature occurs
State number code	non-negative integer	numeric code for state
County number code	non-negative integer	numeric code for county
Primary latitude (DMS)	DDMMSS['N' | 'S']	feature latitude in DMS format or UNKNOWN
Primary longitude (DMS)	DDDMMSS['E' | 'W']	feature longitude in DMS format or UNKNOWN
Primary latitude (dec deg)	decimal number	feature latitude in decimal format or UNKNOWN
Primary longitude (dec deg)	decimal number	feature longitude in decimal format or UNKNOWN
Source latitude (DMS)	DDMMSS['N' | 'S']	latitude of feature source in DMS format, optional
Source longitude (DMS)	DDDMMSS['E' | 'W']	longitude of feature source in DMS format, optional
Source latitude (dec deg)	decimal number	latitude of feature source in decimal format, optional
Source longitude (dec deg)	decimal number	longitude of feature source in decimal format, optional
Feature elevation	integer	altitude above/below sea level, optional
Est. feature population	non-negative integer	estimated population of feature, optional
Federal status	string	???, optional
Cell	string	???

In the GIS record file, each record will occur on a single line, and the fields will be separated by pipe ('|') symbols. Some sample records are shown below.

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.

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.

```
1495182|VA|Acre of Rocks|summit|Montgomery|51|121|371636N|0801608W|37.27667|-80.26889||||2270|||McDonalds Mill
1674451|VA|Agnew Hall|building|Montgomery|51|121|371329N|0802528W|37.22472|-80.42444|||||||Blacksburg
1481269|VA|Aiken Dam Hollow|valley|Montgomery|51|121|372056N|0801741W|37.34889|-80.29472|371932N|0801752W|37.32556|-80.29778|||McDonalds Mill
1674452|VA|Airport Acres (subdivision)|ppl|Montgomery|51|121|371238N|0802427W|37.21056|-80.4075||||2125|||Blacksburg
1462389|VA|Alburn High School|school|Montgomery|51|121|370341N|0802633W|37.06139|-80.4425|||||||Riner
1674453|VA|Alleghany Addition (subdivision)|ppl|Montgomery|51|121|370744N|0802350W|37.12889|-80.39722||||2120|||Blacksburg
1462398|VA|Alleghany Church|church|Montgomery|51|121|370726N|0801610W|37.12389|-80.26944|||||||Pilot
1462399|VA|Alleghany Church|church|Montgomery|51|121|370925N|0801519W|37.15694|-80.25528|||||||Ironto
1481276|VA|Alleghany Church|church|Montgomery|51|121|370930N|0802507W|37.15833|-80.41861|||||||Blacksburg
1674454|VA|Alleghany Heights (subdivision)|ppl|Montgomery|51|121|371509N|0802447W|37.2525|-80.41306||||2220|||Newport
1674455|VA|Alleghany School (historical)|school|Montgomery|51|121|370812N|0801519W|37.13667|-80.25528|||||||Ironto
1674456|VA|Alleghany School (historical)|school|Montgomery|51|121|370921N|0802430W|37.15583|-80.40833|||||||Blacksburg
1674457|VA|Alleghany Spring School (historical)|school|Montgomery|51|121|370706N|0801602W|37.11833|-80.26722|||||||Pilot
1462400|VA|Alleghany Springs|ppl|Montgomery|51|121|370741N|0801555W|37.12806|-80.26528||||1399|||Ironto
1481287|VA|Allen Hollow|valley|Montgomery|51|121|372001N|0801939W|37.33361|-80.3275|371849N|0801938W|37.31361|-80.32722|||McDonalds Mill
1462408|VA|Allen Hollow|valley|Montgomery|51|121|370849N|0801614W|37.14694|-80.27056|370937N|0801636W|37.16028|-80.27667|||Ironto
1674458|VA|Altoona School (historical)|school|Montgomery|51|121|370425N|0801805W|37.07361|-80.30139|||||||Pilot
1674459|VA|Ambler Johnston Hall|building|Montgomery|51|121|371323N|0802517W|37.22306|-80.42139|||||||Blacksburg
1481314|VA|Anderson Cemetery|cemetery|Montgomery|51|121|371519N|0802058W|37.25528|-80.34944|||||||McDonalds Mill
1674460|VA|Apperson Park (subdivision)|ppl|Montgomery|51|121|371425N|0802413W|37.24028|-80.40361||||2220|||Blacksburg
1674461|VA|Apple Acres (subdivision)|ppl|Montgomery|51|121|370923N|0802508W|37.15639|-80.41889||||2130|||Blacksburg
1674462|VA|Asbury Methodist Church|church|Montgomery|51|121|370800N|0802436W|37.13333|-80.41|||||||Blacksburg
1674463|VA|Atkinson Acres (subdivision)|ppl|Montgomery|51|121|370712N|0802436W|37.12|-80.41||||2060|||Riner
1481363|VA|Austin Hollow|valley|Montgomery|51|121|371621N|0801822W|37.2725|-80.30611|371617N|0801657W|37.27139|-80.2825|||McDonalds Mill
1462695|VA|Bain Chapel|church|Montgomery|51|121|370536N|0803150W|37.09333|-80.53056||||2116|||Radford South
1674464|VA|Bangs (historical)|ppl|Montgomery|51|121|370828N|0802407W|37.14111|-80.40194||||2029|||Blacksburg
1674465|VA|Barringer Hall|building|Montgomery|51|121|371334N|0802502W|37.22611|-80.41722|||||||Blacksburg
1481476|VA|Barringer Mountain|summit|Montgomery|51|121|370817N|0802809W|37.13806|-80.46917||||2342|||Blacksburg
1477097|VA|Basham|ppl|Montgomery|51|121|370210N|0802034W|37.03611|-80.34278||||2430|||Pilot
1481600|VA|Beeks School|school|Montgomery|51|121|371249N|0802423W|37.21361|-80.40639|||||||Blacksburg
```

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:

- Retrieving GIS records matching given geographic coordinates
- Retrieving GIS records matching a given feature name and state.
- Retrieving 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>
```

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. If the command script file is not found the program should write an error message to the console and exit.

Data and File Structures:

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 will not treat geographic coordinates as a primary key.

The GIS records will be indexed by the FeatureName and State (abbreviation), using a hash table for the physical organization. See the following section for details of the hash table requirements. The hash table entries will store a FeatureName and State 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.

The GIS records will also be indexed by geographic coordinate, using 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.

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.

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 object can contain references to an unlimited number of different GIS records.

When importing GIS records (see below), your program will make one complete pass through the specified GIS record file. Aside from where specific data structures are required, you may use any suitable Java library containers you like.

Each index object should have the ability to supply a nicely-formatted representation of itself as a `String` object, or to write a nicely-formatted display of itself to an output stream.

There will be a buffer pool serving as a front-end for the database file. Retrieving a GIS record from the database file must be managed through the buffer pool. During an import operation, records will be written to the database file by bypassing the buffer pool; the buffer pool would not have much effect on performance during imports.

Note: your implementation will not simply store the complete GIS records in memory. The file on disk is only place that complete records will exist, aside from a small number of temporary objects created when servicing search requests.

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. A display that does not clearly indicate the structure of the tree, and the relationships between its nodes, is not acceptable.

Hash Table Details:

The hash table must use a contiguous physical structure (array). The initial size of the table will be 1019, and the table will resize itself automatically. The precise logic for resizing is up to you; if you want to match my output logs, increase the size of the table to the next value in the list below when the table becomes 70% full:

1019, 2027, 4079, 8123, 16267, 32503, 65011, 130027, 260111, 520279, 1040387, 2080763, 4161539, 832315 , 16646323

Your table will use some form of quadratic probing to resolve collisions. Given the extremely low probability that quadratic probing will fail to find an empty slot in a reasonable number of steps, there will be no mandatory fall-back strategy. However, you must ensure than every record is actually inserted into the table. The table sizes given above are all primes of the form 4k + 3, and if you use the quadratic function (n^2 + n)/2 to compute the probe slots, an empty slot will always be found.

You will use the `elfhash()` function from the course notes, and apply it to the concatenation of the FeatureName and State (abbreviation) field of the data records.

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

Buffer Pool Details:

The buffer pool for the database file should be capable of buffering up to 20 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.

It is up to you to decide whether your buffer pool stores interpreted or raw data.

You must be able to display the contents of the buffer pool, listed from MRU to LRU entry, in a readable manner.

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 a specific location (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 will usually contain many points).

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 may use floating-point values or integer values to represent region boundaries when computing region boundaries during splitting and quadtree traversals. If you use integers, be careful not to unintentionally create "gaps" between regions.

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 minor 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.

Finally, do not interpret longitude/latitude values sloppily. In particular, do not represent a value given in DMS format, like 1214322W, as the integer -1214322. (The negative sign comes from the fact that it's west longitude, and that's fine.) Some students have encountered problems in the past when using that approach. Instead, convert the given value to total seconds; in this case that would be -438202 seconds.

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. Each non-comment line of the command file will specify one of the commands described below. Each line consists of a sequence of tokens, which will be separated by single tab characters. A newline character 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 following commands must be supported:

`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 likely 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.

`import<tab><GIS record file>`

Add all the 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 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.

`what_is_at<tab><geographic coordinate>`

Log the number of matching records, and all the data fields, properly labeled, in every GIS record in the database file that matches the given `<geographic coordinate>`.

`what_is<tab><FeatureName><tab><State abbreviation>`

Log the number of matching records, and all the data fields in the unique GIS record in the database file that matches the given `<FeatureName>` and `<State abbreviation>`.

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

Log the number of matching records, and all the data fields in the GIS records in the database file whose coordinates fall within the closed rectangle with the specified height and width, centered at the `<geographic coordinate>`. The half-height and half-width are specified as seconds.

`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 it would be useful to include information like key values and file offsets where appropriate.

`quit<tab>`

Terminate program execution.

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.

Sample command scripts 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 the Curator, 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.

The remainder of the log file output should come directly from your processing of the command file. You are required to echo 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 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.

Administrative Issues:

Submission:

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

For this assignment, you must submit a zip file containing all the Java source code files for your implementation (i.e., `.java` files). Submit only the Java source files. Do not submit Java bytecode (class) files. Submit nothing else.

In order to correct submission errors and late-breaking implementation errors, you will be allowed up to five submissions for this assignment. You may choose which one will be evaluated at your demo.

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 will determine a substantial portion of your grade for this assignment. Pay attention to the discussion and solution for the related design assignment, Homework 3, and make changes as needed. It is possible you may lose points during the project evaluation for poor design choices, even if you have already lost points for the same poor decisions on Homework 3. On the other hand, if you lose points on Homework 3, but your implementation of this project indicates that you have corrected those decisions in your final design, we will rebate part of the deductions you received on Homework 3.

Remember that your implementation will be tested in the specified environment. If you use a different development platform, it is entirely your responsibility to make sure your implementation works correctly in the lab.

Note that the evaluation of your project will depend substantially on the quality of your code and documentation. See the Programming Standards page on the course website for specific requirements that should be observed in this course.

Pedagogic points:

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

- implementation of a hash table generic in Java
- implementation of a bucket PR quadtree generic in Java
- implementation of a buffer pool in Java
- implementation of complementary internal and leaf node types to conserve memory, using an appropriate hierarchy of types
- 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
- 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 on the Submitting Assignments page of the course website.