Assignment:

The projects for this semester will build upon each other to form a scheduling system for training sessions. Eventually, it will be possible to enter records for training sessions, delete them, and search by keyword, time, cost, or location. To get some idea of what the final goal will be, see http://www.findaseminar.com/. Of course, your implementation will focus on the relevant data structures to support (a simplified version of) such a system, without the graphical user interface. And your implementation will support true spatial queries, which the FindaSeminar site does not!

For this project, you will be building the spatial indexing component to support queries by location. Your implementation will be fairly “vanilla” in that you will be implementing a spatial data structure without a lot of bells and whistles directed at the eventual training scheduling system. The context for this project is different as well: a database of city records.

As you work on this project, remember that you will have different partners for every project this semester. You will be reusing much of the code from this project in later project, but with different people. And while you will reuse the k-d tree later, that will be in the context of the training scheduling system, not the city GIS of this project. Therefore, flexibility, clarity, and good documentation will be important to your future survival in this class!

The k-d tree:

A binary search tree gives \(O(\log n)\) performance for insert, delete, and search operations (if you ignore the possibility that it is unbalanced). This would allow you to insert and delete cities, and locate them by name. However, the BST does not help when doing a coordinate search. You could combine the \((x, y)\) coordinates into a single key and store cities using this key in a second BST. That would allow search by coordinate, but would not allow for efficient range queries – searching for cities within a given distance of a point. The problem is that the BST only works well for one-dimensional keys, while a coordinate is a two-dimensional key.

The k-d tree (see Section 13.3.1 of the textbook, pp. 436-441) is one of many hierarchical data structures commonly used to store data such as city coordinates. It allows for efficient insertion, deletion and search queries.

Input and Output

The name of your executable must be \(p1\). There will be no input parameters to the program. Your program will read from standard input (\(\text{stdin}\)) and write to standard output (\(\text{stdout}\)). The input for this project will consist of a series of commands (some with associated parameters, separated by spaces), one command for each line. No command line will require more than 80 characters. Commands are free format in that an arbitrary number of additional spaces may be interspersed between parameters, and blank lines may appear between commands. You do not need to check for syntax errors in the command lines (although you do need to check for logical errors such as duplicate insertions or deletions of non-existent cities).
Each input command should result in meaningful feedback in terms of an output message. Each input command should be echo’ed to the output. In addition, some indication of success or error should be reported. Some of the command specifications below indicate particular additional information that is to be output.

Commands and their syntax are as follows.

**insert** $x$ $y$ *name*

A city at coordinate $(x, y)$ with name *name* is entered into the database. $x$ and $y$ are integers in the range 0 to 1023. A *name* must start with a letter, and may contain letters (upper or lower case), digits, and the underscore character. Names are case sensitive, so **new_York** is not the same as **New_York**. It is an error to insert two cities with identical coordinates, but **not** an error to insert two cities with identical names.

**delete** $x$ $y$

The city with coordinate $(x, y)$ is deleted from the database (if it exists). If no city exists with these coordinates, it should be so reported.

**delete** *name*

The city with name *name* is deleted from the database (if it exists). If two or more cities have this name, then **all** such cities must be removed. If no city exists with this name, it should be so reported.

**info** $x$ $y$

Display the name of the city at coordinate $(x, y)$ if it exists.

**info** *name*

Display the coordinates of all cities with name *name* if any exist.

**search** $x$ $y$ *radius*

All cities within *radius* distance from location $(x, y)$ are listed. You should also output a count of the number of k-d tree nodes looked at during the search. $x$ and $y$ are integers with absolute value less than 16384; *radius* is a non-negative integer less than 16384.

**dump**

The BST and the k-d tree are each listed in preorder. All city records are listed for each tree (so that means each city will be listed twice). Records should be printed one per line, and appropriate indentation should be used so that the structure of the tree can be deduced from the listing.

**makenull**

Initialize the database to be empty.

**Example:** Note: in this example, statements enclosed in {} are comments to help you under the example; comments do NOT appear in the data file!
Implementation:

You must maintain two tree structures to support access to the database. A BST will store
the cities indexed by name. A k-d tree will store the cities indexed by (X, Y) coordinate. We
recommend that each tree store nodes whose data field is a pointer to a city record. Thus, each city
has a single city record pointed to by a node in each tree. You may store parent pointers in one or
both trees if you feel that parent pointers will make programming easier. Nodes deleted from the
trees, as well as the city records, are to be placed on a freelist.

Insert, delete and makenull operations affect both the BST and the k-d tree. The list
operation should perform a preorder traversal of both the BST and the k-d tree. First, traverse
the BST, listing all the cities in the order found. Then, traverse the k-d tree, again listing the cities
in the order found. Info with a name parameter should search the BST. Info with coordinate
parameters should search the k-d tree. The search command should search the k-d tree. Search
should also output a count of the number of k-d treenodes visited.

Programming Standards:

You must conform to good programming/documentation standards, as described in the Ele-
ments of Programming Style. Some specifics:

- You must include a header comment, preceding main(), specifying the compiler and operating
  system used and the date completed.
- Your header comment must describe what your program does; don’t just plagiarize language
  from this spec.
- You must include a comment explaining the purpose of every variable or named constant you
  use in your program.
- You must use meaningful identifier names that suggest the meaning or purpose of the constant,
  variable, function, etc.
- Always use named constants or enumerated types instead of literal constants in the code.
- Precede every major block of your code with a comment explaining its purpose. You don’t
  have to describe how it works unless you do something so sneaky it deserves special recogni-
  tion.
- You must use indentation and blank lines to make control structures more readable.
- Precede each function and/or class method with a header comment describing what the
  function does, the logical significance of each parameter (if any), and pre- and post-conditions.
- Decompose your design logically, identifying which components should be objects and what operations should be encapsulated for each.

Neither the GTAs nor the instructors will help any student debug an implementation unless it is properly documented and exhibits good programming style. Be sure to begin your internal documentation right from the start.

You may only use code you have written, either specifically for this project or for earlier programs, or code taken from the textbook. Note that the textbook code is not designed for the specific purpose of this assignment, and is therefore likely to require modification. It may, however, provide a useful starting point. You may not use code from STL, MFC, or a similar library in your program.

Testing:

A sample data file will be posted to the website to help you test your program. This is not the data file that will be used in grading your program. The test data provided to you will attempt to exercise the various syntactic elements of the command specifications. It makes no effort to be comprehensive in terms of testing the data structures required by the program. Thus, while the test data provided should be useful, you should also do testing on your own test data to ensure that your program works correctly.

Deliverables:

When structuring the source files of your project (be it in Eclipse as a “Managed C++ Project,” or in another environment), use a flat directory structure; that is, your source files will all be contained in the project root. Any subdirectories in the project will be ignored. You submission must include either an Eclipse project file, or else a standard Makefile. The GTAs will compile your program with whichever you provide.

If submitting through Eclipse, the format of the submitted archive will be managed for you. If you choose not to develop in Eclipse, you will submit either a ZIP-compressed archive (compatible with Windows ZIP tools or the Unix zip command) or else a tar’ed and gzip’ed archive. Either way, your archive should contain all of the source code for the project, along with any “make” or “project” files necessary to compile the code. If you need to explain any pertinent information to aid the TA in the grading of your project, you may include an optional “readme” file in your submitted archive.

You will submit your project through the automated Web-CAT server. Links to the Web-CAT client and instructions for those students who are not developing in Eclipse are posted at the CS2606 website. If you make multiple submissions, only your last submission will be evaluated. Note that only one member of the pair will make a submission. Whatever is the final submission from either of the pair members is what we will grade.

In addition to the project submission, you are also required to submit a schedule and weekly updates to that schedule. You will find the schedule template at the course website. The initial schedule is due Monday, August 28 by 11pm. You must submit updates on Friday, September 1 by 11pm and Friday, September 8 by 11pm. You must also submit (as part of your final submission package) a final schedule with the “elapsed” column filled in indicating the total hours that you spent to complete all aspects of the project. You won’t receive direct credit for submitting the schedule as required, but each instance of failing to submit scheduling information as required will lose five points for you and your partner from the project grade.
Finally, you and your partner are required to spend a 2-hour block of time in the McBryde 116/118 labs during the TAs posted office hours for each designated “lab week.” During this time, you are expected to work together in “pair programming” mode, where one of you is “driving” at the keyboard, typing, while the other is “navigating.” There are two designated lab weeks for this project, 8/28-9/1 and 9/4-9/8. While you do not directly receive points for attending lab week sessions, each failure to do so will lose you and your partner five points from the project grade.