k-d tree Project

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.

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 (`stdin`) and write to standard output (`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!

```
insert 900 500  Blacksburg
    insert   500   140   Roanoke
 insert 910  510  New_York
dump                        { print coords, name for 3 cities }
  delete  500 140            { its there to delete }
search 901  501  5           { print info for one city }
 info 500 140                { it shouldn't be there }
info  900  500               { print coordinates and name }
makenull                     { reinitialize }
```

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.