EECS 214/395--Data Structures and Data Management
Fall 2014
Programming Assignment No.2—PR Quadtrees
Due: Nov. 11, 2014 midnight

Background

For this project you will build a simple Geographic Information System for storing point data. The focus is organizing city records into a database for fast search. For each city you will store the name and its location (X and Y coordinates). Searches can be by either name or location. Thus, your project will actually implement two trees: you will implement a Binary Search Tree to support searches by name, and you will implement a variation on the PR Quadtree to support searches by position.

Consider what happens if you store the city records using a linked list. The cost of key operations (insert, remove, search) using this representation would be Θ(n) where n is the number of cities. For a few records this is acceptable, but for a large number of records this becomes too slow. Some other representation is needed that makes these operations much faster. In this project, you will implement a variation on one such representation, known as a PR Quadtree. A binary search tree gives O(log *n*) performance for insert, remove, and search operations (if you ignore the possibility that it is unbalanced). This would allow you to insert and remove cities, and locate them by name. However, the BST does not help when doing a search by coordinate. In particular, the primary search operation that we are concerned with in this project is a form of range query called "regionsearch." In a regionsearch, we want to find all cities that are within a certain distance of a query point.

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 does nothing to help regionsearch. The problem is that the BST only works well for one--dimensional keys, while a coordinate is a two--dimensional key. To solve these problems, you will implement a variant of the PR Quadtree. The PR Quadtree is one of many hierarchical data structures commonly used to store data such as city coordinates. It allows for efficient insertion, removal, and regionsearch queries. You should pay particular attention to the discussion regarding parameter passing in recursive functions.

Invocation and I/O Files:

Your program must be named PRprog, and should be inovoked as: PRprog <filename>
1
where <filename> is a commandline argument for the name of the command file. Your program should write to standard output (System.out). The program should terminate when it reads the end of file mark from the input file. The input for this project will consist of a series of commands (some with associated parameters, separated by spaces), one command for each line. Commands are free format in that an arbitrary number of additional spaces may be interspersed between parameters. The input file may also contain blank lines, which your program should ignore. Each input command should result in meaningful feedback in terms of an output message. 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. Note that a *name* is defined to be a string containing only upper and lower case letters and the underscore character.

insert x y name

A city at coordinate (*x*, *y)* with name *name* is entered into the database. That means you will store the city record once in the BST, and once in the PR Quadtree. Spatially, the database should be viewed as a square whose origin
is in the upper left (NorthWest) corner at position (0, 0). The world is 214 by 214 units wide, and so x and y are integers in the range 0 to 214 −1. The x - coordinate increases to the right, the *y*-coordinate increases going down. Note that it is an error to insert two cities with identical coordinates, but not an error to insert two cities with identical names.

remove x y

The city with coordinate (*x*, *y)* is removed from the database (if it exists). That means it must be removed from both the PR Quadtree and the BST. Be sure to print the name and coordinates of the city that is removed.

remove name

Acity with name *name* is removed from the database (if any exists). That means it must be removed from both the PR Quadtree and the BST. Be sure to print the name and coordinates of the city that is removed.

find name

Print the name and coordinates from all city records with name *name*. You would search the BST for this information.

search x y radius

All cities within *radius* distance from location (*x*, *y)* are listed. A city that is exactly *radius* distance from the query point should be listed. x and y are (signed) integers with absolute value less than 214. *radius* is a non-negative integer.

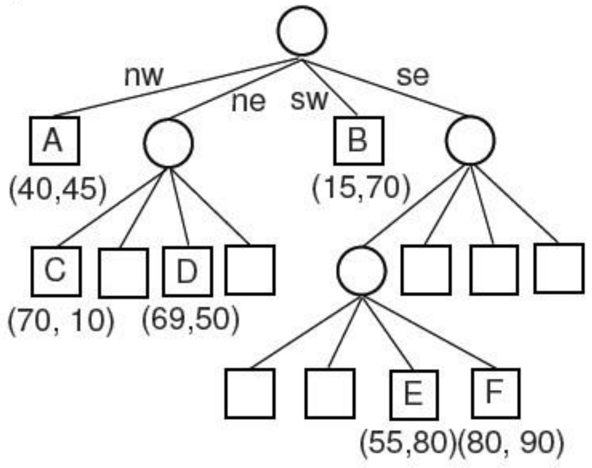
debug

Print a listing of the PR Quadtree nodes in preorder. PR Quadtree children will appear in the order NW, NE, SW, SE. The node listing should appear as a single string (without internal newline characters or spaces) as follows:

- For an internal node, print “I”.
- For an empty leaf node, print “E”.
- For a leaf node containing one or more data points, for each data point print X,Y,NAME where X is the x-coordinate, Y is the *y*-coordinate, and NAME is the city name. After all of the city records for that node have been printed, print a “bar” or “pipe” symbol (the | character).

The tree corresponding to the following Figure would be printed as:

nw se
(70, 10) (69,50)
(55,80)(80, 90)
ne sw
A
C D
B
E F
(40,45) (15,70)



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 Evanston
3
insert 500 140 Skokie
insert 910 510 Chicago
debug { print coords, name for 3 cities }
remove 500 140 { its there to remove }
search 901 501 5 { print info for one city }
makenull { reinitialize }

Implementation suggestions:

- It is recommended that you access tree nodes ONLY through set and get methods in the node classes (this holds true for the PR QuadTree and the BST)

- All operations that traverse or descend the BST or the PR Quadtree structures should be implemented recursively.

- You should use inheritance to implement PR Quadtree nodes. You should have an abstract base class with separate subclasses for the internal nodes and leaf node. A PR quadtree internal node should store references to its children. Leaf nodes should store a (reference to a) city record object.
- For the BST you should deal with the case of allowing multiple city names. You may want to consider using a BST with data stored only at the external nodes.
- The PR Quadtree should be able to handle more than just hard-coded access to "city" records. In other words you should be able to search for different types of objects in a certain region.

Requirements

Environment:

Make sure your program is able to work on T-lab machine. (Note the version of compiler: g++ (GCC) 4.4.7 20120313 (Red Hat 4.4.7-3))

Program:

Your code should be well commented and explain what you are doing. Make sure your runs correctly, efficiently.

Writing:

Give a brief and clear explanation (**less than 1 page)** about your design and thought.

Submission:

The format of your writing is required to be in .pdf format

A Makefile is required to automatically compile your code into executable file.

Everything you do for this project should be included in a folder (e.g. named as project). Compress it in tar.gz format named by your "FirstName_LastName". Submit it to blackboard.

Late policy:

Programs will be accepted for at most two days later with a penalty of 10% off for each day. After submission, if want to modify and submit it, you may upload it again via blackboard.

Good Luck!