Programming Assignment #3

Due: 2015/10/29 at 11:59pm
Points: 20

Goal

To implement a binary search tree and extend it to a quadtree.

Background

In the last programming assignment, you generated a list of points of interest, sorted by distance to the user's current location. But in most applications, the user will only be interested in points within a certain radius. In this assignment, you'll implement *quadtrees*, which are a simple extension of binary search trees, that make it possible to do this.

Distribution

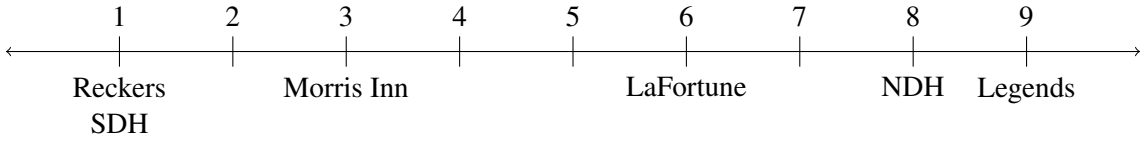
Update your local Git repository (as described at `https://bitbucket.org/CSE-30331-FA15/cse-30331-fa15`) and change to the directory `PG3`. It contains the following files:

`pg3.pdf`	this file
`distance.{cpp,hpp}`	functions for computing distances
`ndfood.txt`	places to eat on campus
`test{1,2}.sh`	test scripts
`test{1,2a,2b}.cpp`	test programs
`measure.c`	tool for measuring performance
`Makefile`	simple makefile

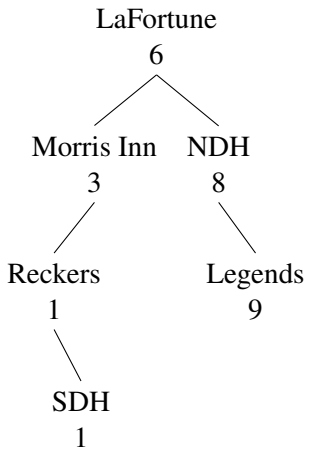
Please copy `osmpoi.txt` from PG2, but don't include it in your submission. (It takes a long time to download all those copies!) Run `make` and ensure that `test1`, `test2a`, and `test2b` are built correctly. Run `test1.sh` and `test2.sh` and note that some tests fail.

Task

Part 1 As a warm-up, let's think about how to do this in a binary search tree. Imagine that the world is one-dimensional, with coordinates from −180 to 180. Let's say the places to eat on campus look like this:



We can insert them into a binary search tree, in which each node has a *name*, a *location*, and two children (*left* and *right*):



Suppose our location is 8 and want all the places within 1 unit (which we write as 8 ± 1). There are a few ways to do this, but consider the following way. We write a function SEARCH that takes a search range (*srange*), a starting node (*node*), and a bounding range (*brange*) that is known to include all the places in the subtree rooted by *node*. The bounding range is written as a pair of endpoints, a *start* and an *end*. Then the pseudocode is as follows:

```
procedure SEARCH(srange, node, brange)
    // Search for locations within srange.
    // node: root of the subtree to search within
    // brange: bounding range of the subtree rooted by node
    if brange overlaps with srange then
        if node.location lies within srange then
            add node.name to list of results
        end if
        SEARCH(srange, node.left, [brange.start, node.pos])
        SEARCH(srange, node.right, [node.pos, brange.end])
    end if
end procedure
SEARCH(8 ± 1, root, [−180, 180])
```

How does this work for our example? The root (LaFortune) is not in the search range. Recurse on the left child (Morris Inn), with bounding range $[-180, 6]$: since this is totally outside the search range, we can skip it. Recurse on the right child (North Dining Hall), with bounding range $[6, 180]$: it is in the search range, so append it to the results. Recurse on NDH's right child (Legends), which is in the search range, so append it to the results.

Program `test1.cpp` reads locations from `ndfood.txt`, using only their longitude and ignoring their latitude. It inserts them into a binary tree (`bintree`) and runs a few tests. Your job is to complete the implementation of `bintree`.

- Implement a binary search tree by filling in the constructor and `insert` in `bintree.hpp`.
- Implement the destructor and copy constructor. The copy assignment operator is implemented for you, but depends on `swap`, which you must fill in.
- Implement `bintree::within_radius` using the algorithm above. Assume that positions and distances are all measured in the same units; the distance between points x_1 and x_2 is just $|x_1 - x_2|$.
- Run `test1.sh` and verify that your program passes all tests.

Part 2 Now, let's move to two dimensions. A *point quadtree* is a tree in which each node has a name, a latitude and longitude (in degrees), and four (pointers to) children: southwest, southeast, northwest, northeast. The southwest subtree contains all the locations southwest of the root, the southeast subtree contains all the locations southeast of the root, and so on.

We want to use a generalization of the above algorithm to search for points within a certain radius. So the search range is a circle specified by a center (degrees latitude and longitude) and a radius (in miles). The SEARCH procedure will again take a bounding range, specified by its southwest corner and northeast corner (degrees latitude and longitude). For simplicity, assume that the international date line is an uncrossable chasm, so the initial bounding range has southwest corner $(-90, -180)$ and northeast corner $(90, 180)$. The pseudocode for the two-dimensional algorithm is the same as before, except that each node has four children.

Program `test2a.cpp` reads locations from `ndfood.txt`, inserts them into a `quadtree` and runs a few tests. Program `test2b.cpp` loads `osmpoi.txt` and tests the performance of a large number of queries. Your job is to complete the implementation of `quadtree`.

- Implement a point quadtree by filling in the the constructor and `insert`.
- Implement the destructor and copy constructor. The copy assignment operator is implemented for you, but depends on `swap`, which you must fill in.
- Implement `quadtree::within_radius` using the appropriate generalization of the algorithm above. As specified in the comments, it should take an argument `p`, the center of the search ball in **degrees** latitude/longitude, and the radius of the ball in **miles**.
  - Function `distance` computes the distance (in miles) between two points (just as in PG2).
  - Function `distance_to_box` computes the minimum distance (in miles) from a point to a box (defined by its southwest and northeast corners). You can use this to check whether the search ball and bounding box overlap.
- Run `test2.sh` and verify that your program passes all tests.

Rubric

binary search tree	
`within_radius` is correct and passes tests:	5
copy constructor and copy assignment pass tests:	2
has no memory errors:	1
quadtree	
`within_radius` is correct and passes tests:	5
copy constructor and copy assignment pass tests:	2
processes 100,000 queries on `osmpoi.txt`	
in under 20 seconds:	1
in under 200 megabytes:	1
has no memory errors:	1
Good style:	1
Good comments:	1
Total:	20

Submission

1. To submit your work, upload it to your repository on Bitbucket:

   (a) `git add` *`filename`* for every file that you created or modified

   (b) `git commit -m` *`message`*

   (c) `git push`

   The last commit made at or before 11:59pm will be the one graded.

2. Double-check your submission by cloning your repository to a new directory, running `make` and running all tests.