Writeup for fifth project of CMSC 420: "Data Structures" Section 0101, Fall 2019

Theme: KD-Trees & PR-QuadTrees

Handout date: Monday, 11-25-2019
On-time deadline: **12-11-2019, 11:59pm**
Late deadline (30% penalty): **12-13-2019, 11:59pm**

Contents

1 Overview

In this project you will implement **KD-Trees** and **P-R (Point-Region) QuadTrees**. You will have to implement **both** dictionary **and** spatial query functionality. You will be tested against unit tests hosted on the department's submit server.

Half of this project is the study of spatial data structures and half the practice of Object-Oriented Programming primitives, in particular, *Inheritance* and *Polymorphism*. You will need to spend some time studying the provided documentation and code structure, in order to understand how the various components are pieced together.

2 Getting started

All you need to do to get started is run a `git pull` from your working Git directory. This will update your project files with a subpackage called `projects.spatial`, which contains all the code we provide you, the skeleton code of the classes that you need to implement as well as a copy of this writeup. Documentation of all the provided class is under `doc`, as usual.

3 What you need to implement

Everything you need to get started is available in our common Git repository. You will need to fill in the implementation of the following 4 (four) classes:

- `knnutils.BoundedPriorityQueue`
- `nodes.KDTreeNode`
- `nodes.PRQuadBlackNode`
- `nodes.PRQuadGrayNode`

The classes come with sufficient documentation that you will be able to find under the directory `doc`, so that you can have a full view of the functionality exposed by the classes' `public` interface. You are given the skeletons of the above classes, as well as various other classes and interfaces, further described in Section 4, which follows.

4 Code base

4.1 Top level

This project can essentially be divided into two "mini-projects". It doesn't matter which you will implement first and which one you will implement second, so we will just **arbitrarily** decide to call the **KD-Tree** part of the project as the **first** part, and the **PR-QuadTree** part of the project as its **second** part.

In this project, we supply you with a lot of code to use to build your own. Figure 1 provides a bird's eye-view of the project.

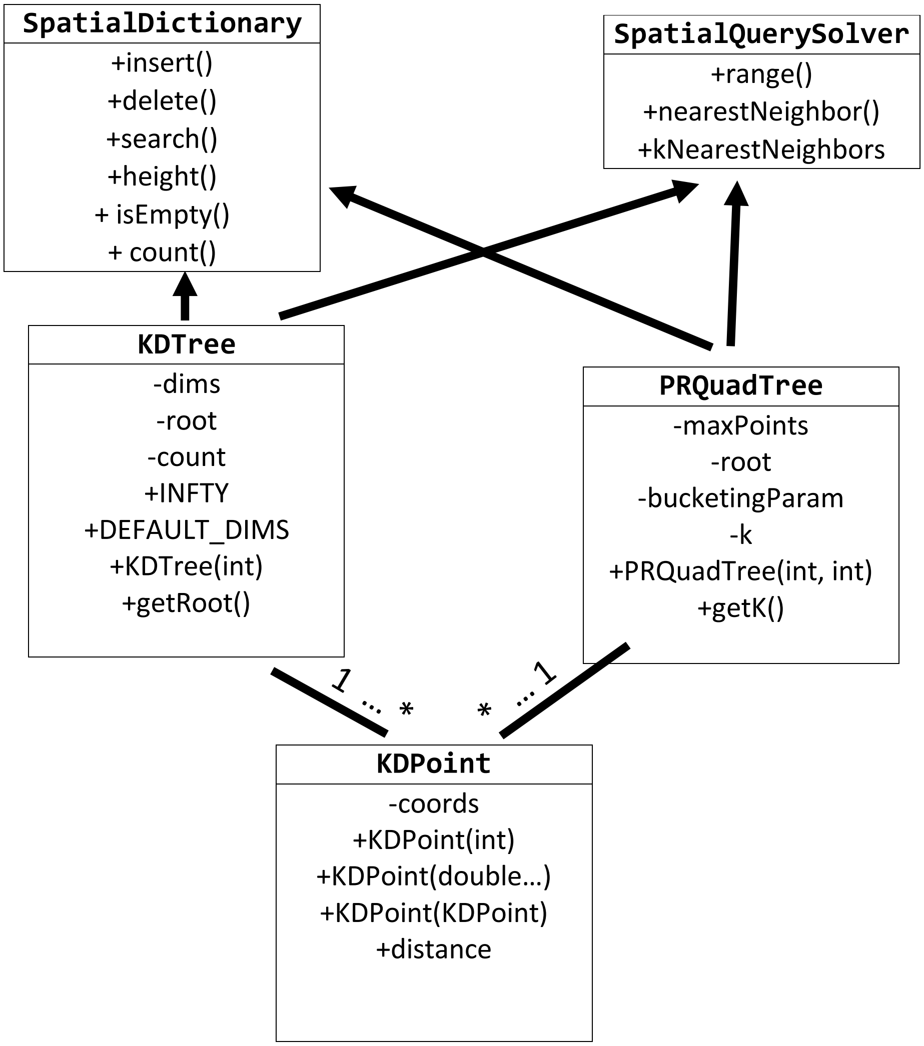

Figure 1: A UML diagram describing the behavior and basic dependencies of the classes `KDTree` and `PRQuadTree`. Simple lines reflect one-to-many (1 - $*$) "has - a" relations, while arrows show "is-a" relations (derived class, implemented interface, etc).

Both KD-Trees and PR-QuadTrees are **multi-dimensional indices**. Since they are multi-dimensional indices, the first thing they need to know is the **nature of**

the keys that they will store. The type of key stored is defined by the class `kdpoint.KDPoint`. Since, in theory, a point can have any real number in its dimensions, the inner representation of `KDPoints` will be an array of `BigDecimal` instances. `BigDecimals` are essentially extremely precise `doubles` which can be safely compared to each other (with `equals()` and `compareTo()`) without the problems that arise from comparing primitive `doubles`.

Instances of `KDPoint` will appear in virtually **all of the methods** that you will have to implement! You should study `KDPoint` **extensively** to understand how it works. In particular, notice that, **since the internal buffer of `KDPoints` is exposed to the outside world for convenience, `KDPoints` are mutable objects!** This means that you should **always** make **deep copies** of `KDPoint` instances when you have to! **No aliasing!** A copy constructor for `KDPoints` is provided for you. You should also have a look at how extensive the testing suite for `KDPoint` is, despite the apparent "simplicity" of the class. You might get some ideas for testing your own classes by looking at `KDPointTests.java`.

4.2 KD-Tree and Bounded Priority Queue

Figure 2 shows the structure of the first part of the project:

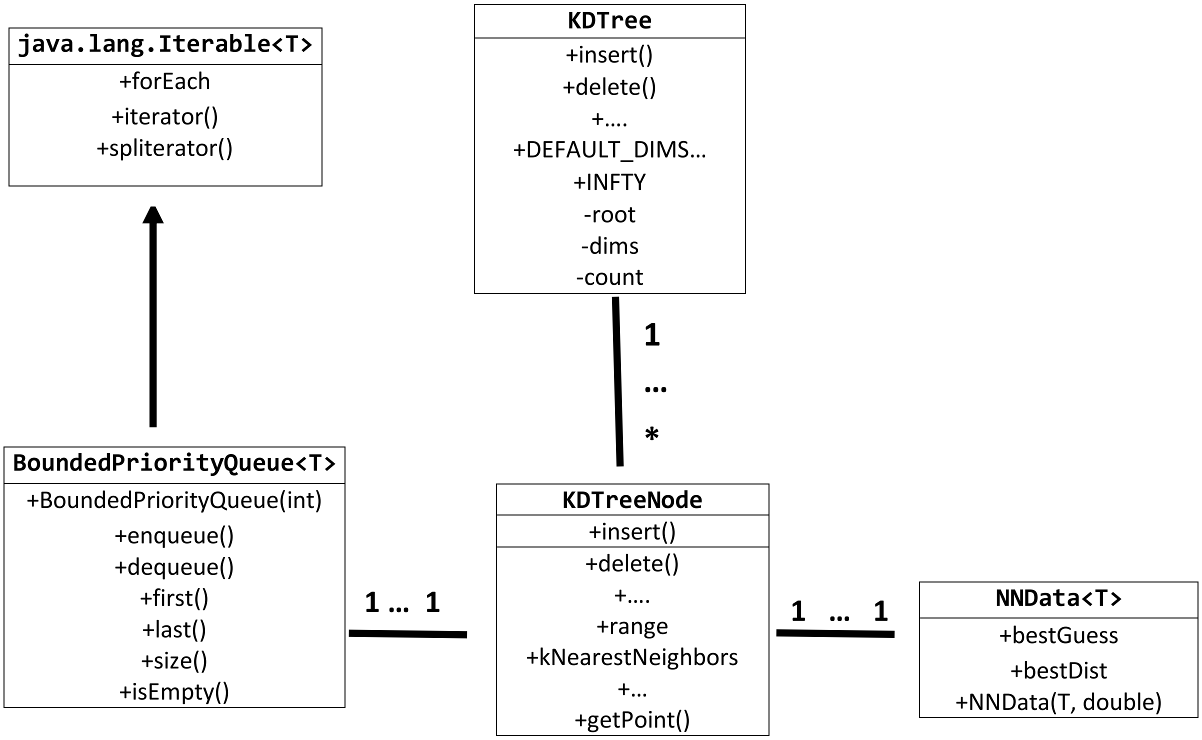

Figure 2: Structure of the first part of the project, dedicated to the implementation of KD-Trees and "plug-ins" that allow for spatial queries.

Interestingly, the implementation of the class `KDTree` is **provided** for you! However, this is **not much respite**: If you were to browse the class's source code, you would note that all the work you have to do is now part of the class `KDTreeNode`, located inside the package `nodes`. Section 4.3 will shed some light on why we structure the code you have to submit in this manner.

`KDTreeNode` uses `BoundedPriorityQueue` **only** for the implementation of m-nearest neighbor queries, with $m \geq 2$. For 1-nearest neighbor, it uses the simple `struct` - like class `NNData`. These types are declared as parameters of the relevant methods of the class `KDTree` **and** `KDTreeNode`, so your project **won't compile** against our tests if you don't have them **exactly where they are in the code tree**!

For your implementation of `BoundedPriorityQueue`, you are given **complete implementational freedom**. That is to say, if you wanted to use your own Priority Queue and adjust it to the semantics of a priority queue bounded above, you can do this. If you want to extend Java's built-in Priority Queue to do what you want it to, that is **also** fine with us. You do not even have to adhere to a **particular implementation** of a Priority Queue: you can use a min-heap, an array of lists made up with elements of the same priority, sorted by insertion order, anything really, as long as the implementation is **correct** and provides **elementary** efficiency for `first(), dequeue(), enqueue(), last()`. By *"elementary efficiency"*, we essentially mean: *"Don't make it so bad that the submit server will hang."* Doing this would be a *major achievement* in its own right, and we trust that we don't need to define it any further. However, the semantics of `BoundedPriorityQueue` are different from that of a classic Priority Queue in that it limits the number of "best neighbors" that a given `KDPoint` can have. Consult the lecture slides and recordings for more information on what we mean by that.

4.3 PR-QuadTree

Figure 3 contains a UML diagram that shows the structure of the second part of the project, which concerns the data structure known as a **"Point-Region" (P-R) QuadTree** .

As with the class `KD-Tree`, the "central" class `PRQuadTree` has been given to you! However, **all** the work that you need to do is in the *derived* classes `PRQuadGrayNode` and `PRQuadBlackNode`. Since **any node** in the PR-QuadTree can **dynamically change status between black, gray and white node**, we need to deal with those nodes *polymorphically*: i.e, we want to call insertion and deletion routines that do the **right thing given the runtime class of the `PRQuadNode` instance for which they are called!** For this reason, we provide you with the `abstract` class `PRQuadNode`, which provides the **common interface** that we expect from **any** given node in a PR-QuadTree to implement. In detail, every node should have *some* way to insert and delete a KDPoint, query the node about the height of the subtree rooted at the node, ask for the number of KDPoints stored either in the node itself (if it's black) or **anywhere** in the (sub)tree **rooted** at the node (if it's gray).

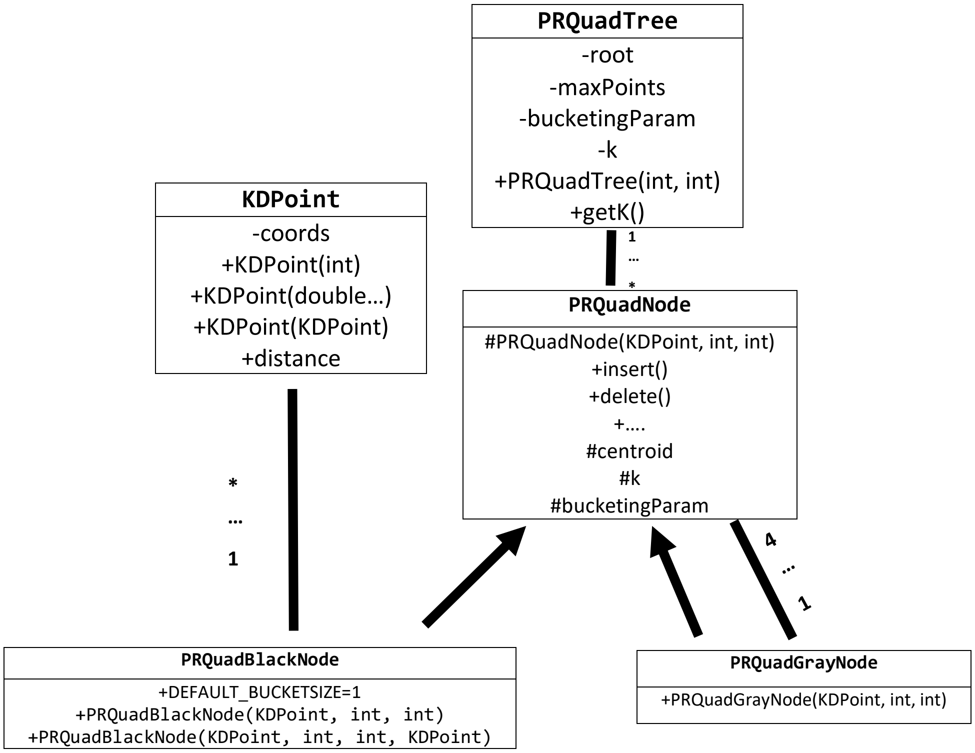

Figure 3: Structure of the second part of the project, dedicated to the polymorphic implementation of a PR-QuadTree. Note that PRQuadGrayNode instances are **both** PRQuadNode instances and **contain** up to 4 PRQuadNode instances! So we have mixed *is-a* **and** *has-a* relationships here.

Some points of interest:

- You might wonder why there is no class PRQuadWhiteNode in the provided code base. This is because such a class would be **fundamentally useless**, since white nodes **don't really do or contain anything**! This means that one can model them adequately (and cheaply!) using a null reference. As always, you yourselves **absolutely can** use a separate class called, e.g, PRQuadWhiteNode for your own purposes (debugging or otherwise), if you want to. Remember: you should **not** alter the **existing** code base in **any** way, but **adding your own functionality** is **always** fine. The submit server unit test suites only care about what they can call, and what they can call is the public methods we need you to implement!

- The base class `PRQuadNode` is made into an `abstract class` instead of an `interface` because it contains a `protected` data field of type `KDPoint`, which is called `centroid`, and another `int` field called `k`. Refer to their documentation[1] to understand what they are useful for. In the `Java` programming language, one cannot have data fields in `interfaces`. Every one of our nodes, irrespective of color, has a **geometrical interpretation**, since it models a certain quadrant of our space. As you think about the project, you might wonder why we have `centroid` and `k` as data types in the base class, instead of just in `PRQuadGrayNode`. This you might think because, at least initially, **it doesn't seem as if a `PRQuadBlackNode` actually uses any geometrical information;** it just stores `KDPoints`. **We will not answer this question in the writeup**; it is for you to think about.

4.4 Goodies

- Some students shared on Piazza that they were having trouble with `Archiver` generating a very big archive. In several of these cases, it was detected that the student had a very large chain of "recursive" directory structures within their project. In order to help detect if this is the case, we have included, under `src`, the script `AdvancedArchiver.java`. Since it is in beta, we include `Archiver.java` which we have found to be working in all cases where the directory structure of the project is intact.

- Check the methods `KDTree.treeDescription()` and `PRQuadTree.treeDescription()` for some visualizations of your trees that you can generate. Those visualizations follow, more or less, the style of this website that was discussed in Piazza.

4.5 Important note about implementation of spatial queries

It is **possible** to pass the relevant tests on the submit server by implementing range and k-NN queries *naively*. That is to say, you can pass a range query test by performing a Depth-First Traversal of the tree, adding all points to a list, and then filtering out the points that do **not** satisfy the query. For nearest neighbors, you can perform a *sorting* of the list based on distance to the anchor point and only keep the k first elements.

[1] Yes, you can generate JavaDocs for fields too, and even `private` or `protected` fields or methods. All modern IDEs have ways to toggle whether documentation of such "access-restricted" elements of a given class will appear in the documentation. We provide you with JavaDocs for **everything** in this project.

We want to clarify that solving the spatial queries for either data structure in this naive manner is **not** an acceptable method of implementation. The entire point of the heuristic and branch - and - bound approaches that we follow in class is that **huge swaths of a given KD-Tree/PR-QuadTree can be avoided**. We want you to show us that you can **implement these algorithms**, that you can finesse your way around the recursive calls, make the appropriate checks, disambiguate between Manhattan and Euclidean distances. Everybody can do a DFS of a tree and an insertion into a list; a CS420 student *can* and *should* do better. We will be **manually** inspecting your submission to make sure you follow these algorithms closely. Submissions that do **not** follow these algorithms and instead perform a naive search of the tree will **not** get **any** credit from the submit server tests that test spatial queries.

5 Hints

- Don't spend too much time reviewing `BigDecimal`. Just make sure you understand that you compare them using `equals()`, `compareTo()`, and you can always retrieve an underlying `double` value by applying the "getter" method `doubleValue()`. You do **not** need to understand how `BigDecimal` ensures immeasurably higher accuracy than a primitive `double` in this project.

- For the KD-Tree component:

  - For deletions, consider the routine `findMin()` that we did in class. How can you use it to cover the special cases of deletion?
  - Consider the theory behind range and NN-Queries **before** you attempt to implement them. How do we behave in the descending / greedy phase? How do we behave in the ascending / pruning phase? What about the edge cases? How do we behave?
  - You can choose to leave the implementation of `BoundedPriorityQueue` implementation for after you implement `kNearestNeighbors()` if you wish, but realize that proper k-NN functionality depends on proper BPQ functionality...

- For the PR-QuadTree component:

  - There are three types of students in this class. Those who study the parameter `PRQuadTreeNode.k` **well**, those who don't, and others.
  - By the time you read this, it might have already been mentioned in lecture, but it's definitely worth repeating here. In the backtracking

phase of the range and nearest neighbor queries in a PR-QuadTree, while you **should** only recurse to quadrants which **intersect** with the given range (or current *worst* range, for k-NN), you should **not** be prioritizing the quadrants by the area of intersection; instead, keep it simple and loop through only the **valid** quadrants in **Z-order**. Finding the area of intersection is a hard numerical problem which we will not solve in this project.

- Speaking of intersection of a quadrant with a given range, check the `protected` method `PRQuadNode.doesQuadIntersectAnchorRange()`. It might be useful to you.