

Writeup for fifth project of  
CMSC 420: “Data Structures”  
Section 0201, Fall 2018

**Theme: KD-Trees & PR-QuadTrees**

**Handout date:** Sunday, 11-18-2018

**On-time deadline:** **12-12-2018, 11:59pm**

**Late deadline (30% penalty):** **12-14-2018, 11:59pm**

**Contents**

<b>1</b>	<b>Overview</b>	<b>2</b>
<b>2</b>	<b>Getting started</b>	<b>2</b>
<b>3</b>	<b>What you need to implement</b>	<b>2</b>
<b>4</b>	<b>Code base</b>	<b>2</b>
4.1	Top level . . . . .	2
4.2	KD-Tree and Bounded Priority Queue . . . . .	4
4.3	PR-QuadTree . . . . .	5

# 1 Overview

In this project you will implement **KD-Trees** and **P-R (Point-Region) QuadTrees**. For the former, you will also implement **spatial queries** (**range**, **nearest neighbor**). 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 with as well as the classes that you need to implement.

Speaking of...

## 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 classes:

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

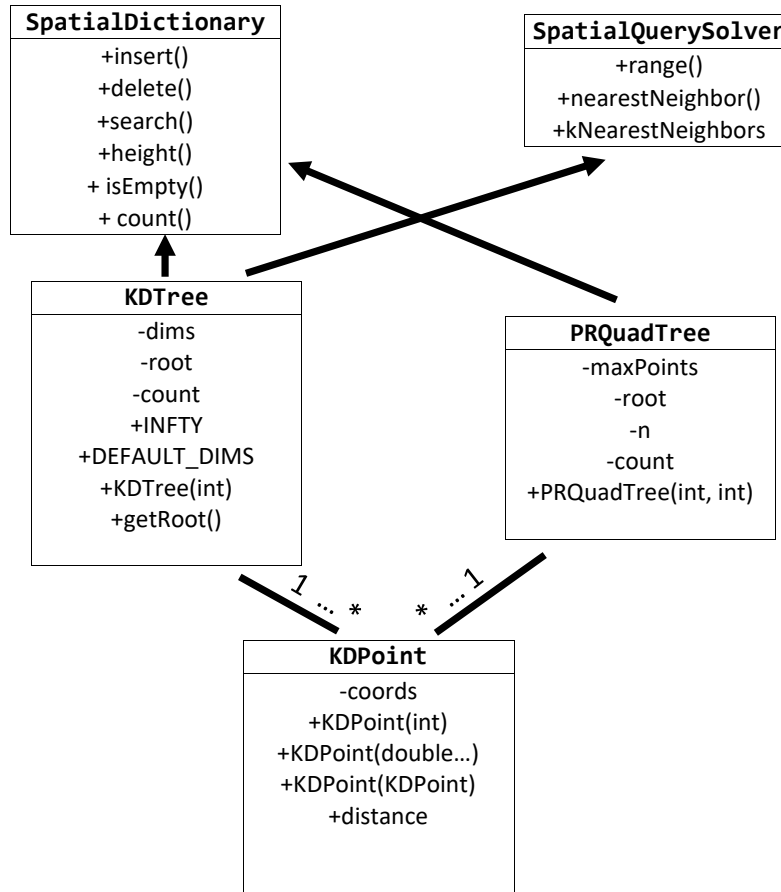
The classes come with sufficient documentation that you will be able to find under the directory `doc`, so that you can have a full view of the functionality exposed by the class' `public` interface. You are given the skeletons of the above classes, as well as various other classes and interfaces, further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

One additional point about the implementation of **KDPoints**: as you can see by browsing either the documentation or the source code of the class **KDPoint**, the data type used to hold the coordinates of a  $d$ -dimensional point is **double**. This can cause some problems with comparing **doubles** up to an arbitrary degree of precision.<sup>1</sup> To have an idea of how you can alleviate this problem of comparing **doubles** in your unit tests, please feel free to consult the file `kdpoint.KDPointTests.java`, which is a **jUnit4** test suite for **KDPoints** that we include just for your own benefit.

**KDTree** implements both **SpatialDictionary** and **SpatialQuerySolver**, while **PRQuadTree**

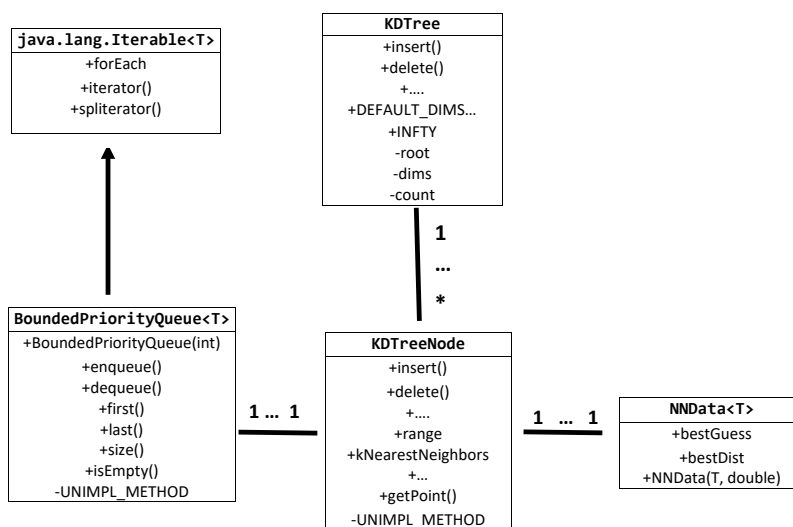
<sup>1</sup>To alleviate these problems, **ava** has introduced the type **BigDecimal**, which is an immutable floating-point number type that allows for arbitrary precision.

only implements `SpatialDictionary`. This has been **deliberate**, so that we can make the project **easier** for you. However, it also shows you one of the benefits of *Object-Oriented Design*. If you are ever interested in extending the functionality of `PRQuadTree` to allow for spatial queries (range, 1 or  $m$  nearest neighbors), all you need to do is **extend** that interface (in addition to `SpatialDictionary`). Your IDE will then prompt you for an implementation of those methods, with their **exact signatures**, and even copy over the JavaDocs so that you don't need to re-write them! This is an example of how you should be thinking later on in your careers, if you end up having to do a lot of object-oriented software development: if you see that you have many classes that share a **good portion** of a **common public interface**, then maybe you should package all of their declarations into an actual **Java interface**, instead of copying - and - pasting declarations of methods! Those methods might be in the hundreds! Not to mention that if the actual *implementation* is entirely identical, then the **interface** can be made into an **abstract class**, and the same implementation can be re-used! For example, methods such as `height()`, `count()`, `isEmpty()` tend to be implemented in **exactly the same way line-by-line** in **many** tree-based data structures. Perhaps if we had a common interface for all those structures called `Tree`, we would be able to include all the common information there and not waste time on re-writing stub methods!

**The less code you have to write for any given task, the better.** For one, you reduce the likelihood of **inserting bugs** in the code base. You also avoid going through the trouble of having to **generate new documentation**, which you can **also** introduce bugs in! Finally, you also make **compilation** faster: you avoid compiling and linking new sources with your existing ones.

## 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’ source code, you would note that all the work

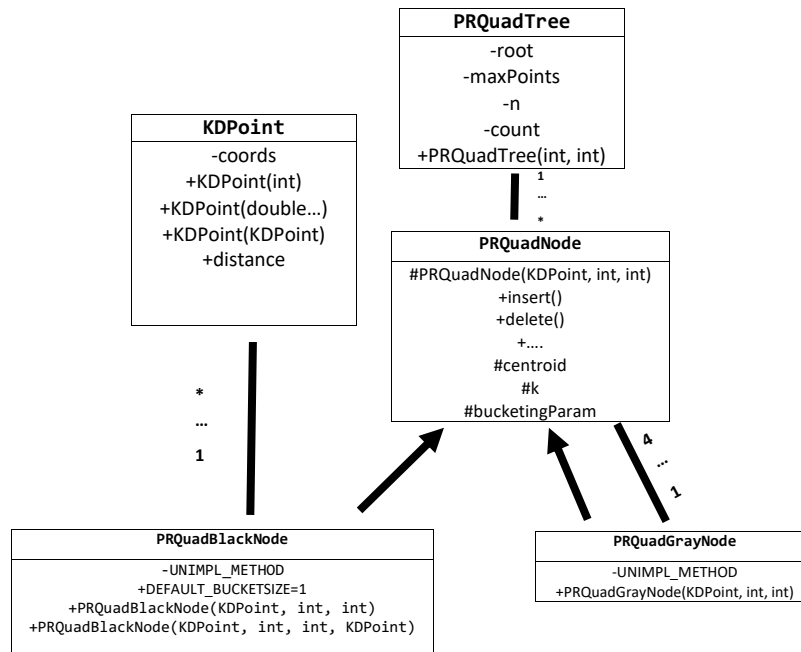
you have to do is now part of the class `KDTreeNode`, located inside the package `nodes`. Section 4.3 will shed some light on why we structure the code you have to submit in this manner.

`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 `Priority Queue` from project 1 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.



**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!

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`. 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).

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 your **public** methods!
- 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<sup>2</sup> 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.

---

<sup>2</sup>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.