Project 2AB: KDTree and Extrinsic PQ

Over the next 5 weeks, you are going to building a system very similar to Google Maps.

This system will have four major parts:

- Project 2A (due 10/16): Building a k-d tree.
- Project 2B (due 10/23): Building an ExtrinsicPQ.
- Project 2C (due 11/2): Building an AI for solving problems (AStarSolver).
- Project 2D (due 11/9): Combining all the pieces into a web-browser mapping application called BearMaps.

Projects 2A and 2B are being released at the same time and in the same spec since they are quite similar in structure.

All parts of this project should be completed individually. Please refer to the course policies for our official policy on academic dishonesty.

This part of the project will be as much about testing as it is about implementation. The autograder will be very sparse, and it will be up to you to test your own code! **Grading details for project 2A and 2B will not be released until the amnesty period for submissions is over (December 12th).**

In this document, we'll only discuss proj2ab. More details about the mapping system (proj2c and proj2d) will come later.

Proj2A: K-d Tree

For this part of the project, your primary goal will be to implement the `KDTree` class. This class implements a k-d tree as described in lecture 19.

This part of the project is a warmup to part 2B, where you'll be doing something quite similar, but with much less guidance from us.

If you're not familiar with how k-d trees work, please see Lecture 19 before proceeding. It is absolutely imperative that you do this before starting this assignment.

This will ultimately be useful in your BearMaps application. Specifically, a user will click on a point on the map, and your k-d tree will be used to find the nearest intersection to that click.

As mentioned above, the autograder for this project is minimal. Thus, you'll need to verify the correctness and speed of your code on your own using your own tests. Since this is probably the first time you've ever had to do this in a project, I've created a pseudo-walkthrough video where I give specific recommendations for steps that you might want to take to complete the project. This is ONLY available for project 2a! When you get to project 2b, you'll be entirely on your own.

IMPORTANT NOTE: YOU MAY NOT SHARE ANY CODE ON THIS ASSIGNMENT WITH ANYONE. It is not OK to share code, even if it's very similar to the code shown in the walkthrough video.

This walkthrough also includes links to a series of solutions videos, where in each video, I solve one part of the project (with the code mostly blurred out). These videos cover some, but not all of the project. Even if you solve the project entirely on your own, you still might find these solutions videos useful, so that you can compare your problem solving strategy to mine.

Slides for this pseudo-walkthrough (including links to solution videos) can be found at this link. You are not required to follow these steps or use this walkthrough!

Provided Files

We provide a class called `Point.java` that represents a location with an x and y coordinate. You may not modify this class. There are three useful methods in this class:

- `public double getX()`

- `public double getY()`
- `public static double distance(Point p1, Point p2)`: Note that this function returns the squared Euclidean distance.

We also provide a interface called `PointSet` that represents a set of such points. It has only one method.

- `public Point nearest(double x, double y)`: Returns the point in the set nearest to x, y.

NaivePointSet

As we saw in lab 5, it is often easiest to test the implementation of a complex but fast data structure by first implementing a simple but slow data structure that we can use as a gold standard. Before you create your efficient `KDTree` class, you will first create a naive linear-time solution to solve the problem of finding the closest point to a given coordinate. The goal of creating this class is that you will have a alternative, albeit slower, solution that you can use to easily verify if the result of your k-d tree's `nearest` is correct. Create a class called `NaivePointSet` The declaration on your class should be:

```
public class NaivePointSet implements PointSet {
    ...
}
```

Your `NaivePointSet` should have the following constructor and method:

- `public NaivePointSet(List<Point> points)`: You can assume `points` has at least size 1.
- `public Point nearest(double x, double y)`: Returns the closest point to the inputted coordinates. This should take \(\Theta(N)\) time where \(N\) is the number of points.

Note that your `NaivePointSet` class should be immutable, meaning that you cannot add or or remove points from it. You do not need to do anything special to guarantee this.

Example Usage

```
Point p1 = new Point(1.1, 2.2); // constructs a Point with x =
Point p2 = new Point(3.3, 4.4);
```

Proj2A: K-d Tree

Proj2A Submission & Grading

Proj2B - Extrinsic MinPQ

Project 2A FAQ

Project 2B Submission & Grading

```
    Point p3 = new Point(-2.9, 4.2);

    NaivePointSet nn = new NaivePointSet(List.of(p1, p2, p3));
    Point ret = nn.nearest(3.0, 4.0); // returns p2
    ret.getX(); // evaluates to 3.3
```

Part 2: K-d Tree

Now, create the class `KDTree` class. Your `KDTree` should have the following constructor and method:

- `public KDTree(List<Point> points)`: You can assume `points` has at least size 1.
- `public Point nearest(double x, double y)`: Returns the closest point to the inputted coordinates. This should take \(O(\log N)\) time on average, where \(N\) is the number of points.

As with `NaivePointSet`, your `KDTree` class should be immutable. Also note that while k-d trees can theoretically work for any number of dimensions, your implementation only has to work for the 2-dimensional case, i.e. when our points have only x and y coordinates.

For `nearest`, we recommend that you write a simple version and verify that it works before adding in the various optimizations from the pseudocode. For example, you might start by writing a `nearest` method that simply traverses your entire tree. Then, you might add code so that it always visits the "good" child before the "bad" child. Then after verifying that this works, you might try writing code that prunes the bad side. By doing things this way, you're testing smaller ideas at once.

Testing

There are a number of different ways that you can construct tests for this part of the project, but we will go over our recommended approach.

It's imporant to note that the `Point`s that your `KDTree` and `NaivePointSet`'s return do not need to be exactly the same: instead, you should check that the distance from the goal `Point` to each of the `Point`s that your `KDTree` and `NaivePointSet` finds are equal. If you're using JUnit (which you should be), then you should read the JUnit javadoc for the `assertEquals` method for `double` values. You'll see that you actually need to pass in a third argument which you should think of as a threshold. Use a very small threshold like `0.00000001`.

Basic Sanity Checks

After you are confident in your partial implementation (the constructor or unoptimized `nearest`, for example), in a `main` method, construct the k-d tree from lecture 19, and verify with the Java Visualizer that you are able to construct the tree correctly, and return the right `Point` for a particular `nearest` query. You are likely to gain some confidence in your implementation through this exercise, and also discover some bugs. When you are done with your sanity checks, you can move forward to more rigorous randomized testing.

Randomized Testing

One approach to testing this method would be to create a bunch of hand-curated datasets to pass to the constructor and calls to `nearest` that try to verify various edge cases. To avoid thinking about all possible strange edge cases, we can turn towards techniques other than creating specific, deterministic tests to cover all of the possible errors.

Our suggestion is to use the lab 5 approach of using randomized comparison tests which will allow you to test your code on a large sample of points which should encompass most if not all possible edge cases.

As in lab 5, we recommend generating a large number of random points to be in your tree, as well as a large number of points to query using the `nearest` function. To verify the correctness of the results you should be able to compare the results of your `KDTree`'s `nearest` function to the results to the `NaivePointSet`'s `nearest` function. If we test a large number of queries without error we can be fairly confident in the correctness of our data structure and algorithm.

An issue is that randomized tests are not deterministic. This mean if we run the tests again and again, different points will be generated each time which will make debugging nearly impossible because we do not have any ability to replicate the failure cases. However, randomness in computers is almost never true randomness and is instead generated by pseudorandom number generators (PRNGs). Tests can be made deterministic by seeding the PRNG, where we are essentially initializing the PRNG to start in a specific state such that the random numbers generated will be the same each time. We suggest you use the class `Random` which will allow you to generate random coordinate values as well as provide the seed for the PRNG. More can be found about the specifications of this class in the online documentation.

Please put any tests you write in `KDTreeTest.java`. **You cannot share your tests with other students**, but you are free to discuss testing strategies with them. **Even if your tests are very similar to the ones I write in my video, YOU MAY STILL NOT SHARE YOUR CODE!**

Note: Our autograder will test points that have x and y coordinates whose magnitudes are less than 1000. We recommend your tests also consider points over this range.

Timing Tests

You will also be required to verify the speed of your k-d tree.

It should be possible to build a k-d tree of a million random points in a reasonable amount of time (less than 20 seconds). For example, our `KDTree` consruction timing table is displayed below. Note, this was running on a fairly fast computer, so your times may be longer. If you can hit 1,000,000 points in less than 20 seconds, you're in good shape. Don't worry about worst case inputs (i.e. your code should not gracefully handle a very spindly k-d tree, this requires a much more sophisticated approach, e.g. a k-d-b-tree). Note that "# ops" here refers to the number of points added to the table, and doesn't actually represent the count of any specific method calls.

```
Timing table for Kd-Tree Construction
           N     time (s)        # ops  microsec/op
------------------------------------------------------------
       31250         0.03        31250         1.02
       62500         0.05        62500         0.78
      125000         0.10       125000         0.82
      250000         0.23       250000         0.90
      500000         0.63       500000         1.26
     1000000         1.25      1000000         1.25
     2000000         3.14      2000000         1.57
```

Your `KDTree`'s `nearest` method should also be MUCH faster than your NaivePointSet. For example, our timing table for 1,000,000 queries on a NaivePointSet with N points is given below:

```
Timing table for Naive Nearest
           N      time (s)        # ops  microsec/op
------------------------------------------------------------
         125          0.30      1000000         0.30
         250          0.47      1000000         0.47
         500          0.80      1000000         0.80
        1000          1.66      1000000         1.66
```

And for our `KDTree` below. Note that our `KDTree` was able to handle `PointSet`s that were 1,000 times as large as a a naive implementation.

```
Timing table for Kd-Tree Nearest
           N      time (s)        # ops  microsec/op
------------------------------------------------------------
       31250          0.93      1000000         0.93
       62500          0.86      1000000         0.86
      125000          1.03      1000000         1.03
      250000          1.34      1000000         1.34
      500000          2.09      1000000         2.09
     1000000          2.38      1000000         2.38
```

Note: To use the Stopwatch class you'll need to `import edu.princeton.cs.algs4.Stopwatch;`.

External Resources

If you don't like our presentation of the k-d tree, you are free to reference pseudocode from course slides and other websites (use the @source tag to annotate your sources), but all code you write must be your own. We strongly recommend that you use the approach described in CS 61B, as it is simpler than described elsewhere.

You are welcome to look at source code for existing implementations of a k-d tree, though of course you should not be looking at solutions to this specific CS 61B project. Make sure to cite any sources you use with the @source tag somewhere in your code. **As always you should not use other student's solutions to this project as a source**, e.g. roommate's code, something you found on pastebin, github, etc.

Proj2A Submission & Grading

Required Files for Proj2A

You are required to submit the following files:

- `NaivePointSet.java`: your complete naive solution as described above.
- `KDTree.java` : your complete correct k-d tree implementation as described above.
- `KDTreeTest.java`: any tests you write for your k-d tree implementation. If you do not write any tests, you should still submit this file.

You are also welcome to submit other files.

You will not get results from our full correctness and efficiency tests on your code when you submit on Gradescope.

Grading

NaivePointSet

We will only test your `NaivePointSet` for correctness, not efficiency.

K-d Tree

We will test your `KDTree` for correctness and efficiency. Correctness tests will be very similar to the randomized testing described in the Testing section, where we will compare your `KDTree`'s `nearest` results against our staff solution's results (using the distance metric as described above). We will not be grading these based on efficiency, but please ensure that the method call completes in a reasonable about of time.

Efficiency tests will be similar to the correctness tests and will still require correctness, but they will also see how your solution performs with a very large number of points (hundreds of thousands) and many queries to `nearest` (tens of thousands). Your `KDTree`'s runtime will be compared to our staff `KDTree`'s runtime, and we will assign points based on that. If you implement the k-d tree correctly (similar to how you learn in lecture) and do not have repeated, redundant, or unnecessary function calls, you should be fine for these tests. For reference, our `KDTree`'s runtime while making 10,000 queries with 100,000 points is about 65-85x faster than our `NaivePointSet`'s runtime for the same points and queries. In addition, we will have a test that ensures your constructor is at most 10x slower than our staff `KDTree`'s constructor. This should not be a strict test as our constructor is relatively naive and doesn't do anything fancier than what you learned in class.

K-d Tree Tests

You should also submit your tests, but we will not be testing your tests with an autograder.

Provided Tests

Since we are not releasing results of your submission on our full autograder, we will provide you with some very basic sanity checks to make sure your code will run with our full autograder once we run it. These will include:

- File, API, compilation, style, and dependency checks.
- Two small correctness checks for your `NaivePointSet` and `KDTree` classes' `nearest` functions tested with one query on ten points each.
- One speed check for `KDTree`. We will run your solution on many points and many queries. If your solution takes 15 times as long as the staff solution or faster, you will pass this test. This test will be the exact same as the lowest tier (easiest) of our speed tests, except that it will **not test for correctness**.

These tests are by no means a good indicator of how well you will do on our full suite of tests, so make sure you write your own tests as well. Your Gradescope submission may show that you receive full points for this assignment if you pass all these tests, but **this score is not your final score.**

Point Distribution

The point distribution for this project will be:

- ~5%: `NaivePointSet` correctness.
- ~55%: `KDTree` correctness.
- ~16%: `KDTree` constructor efficiency.
- ~24%: `KdTree` nearest efficiency (only eligible if implementation passes correctness tests).

Proj2B - Extrinsic MinPQ

In project 2B, you will build another data structure. Like project 2A, you will have to verify the timing and correctness of your code yourself, i.e. the autograder won't give you very much information. Unlike 2A, where we told you exactly which algorithm to implement the steps you should take to complete it, in 2B you will have to design your own solution to the problem at hand.

A quick video overview can be found at this link. Note that this video is from Spring 2019, so some small details may be different (e.g. Extrinsic MinPQ was project 2A in spring 2019).

For this part of the project, you will build an implementation of the `ExtrinsicMinPQ` interface. Ultimately, this will be useful for implementing the `AStarSolver` class that will be described in the Proj2C spec.

If you're not familiar with how heaps and priority queues work, please see Lecture 21 before proceeding. It is absolutely imperative that you do this before starting this assignment.

The `ExtrinsicMinPQ` interface quite similar to the `MinPQ` interface that we discussed in lecture 21. The operations are described below:

- `public boolean contains(T item)`: Returns true if the PQ contains the given item.

- `public void add(T item, double priority)`: Adds an item of type `T` with the given priority. If the item already exists, throw an `IllegalArgumentException`. You may assume that `item` is never null.
- `public T getSmallest()`: Returns the item with smallest priority. If no items exist, throw a `NoSuchElementException`.
- `public T removeSmallest()`: Removes and returns the item with smallest priority. If no items exist, throw a `NoSuchElementException`.
- `public int size()`: Returns the number of items.
- `public void changePriority(T item, double priority)`: Sets the priority of the given item to the given value. If the item does not exist, throw a `NoSuchElementException`.

There are four key differences between this abstract data structure and the MinPQ ADT described in lecture 21.

- The priority is extrinsic to the object. That is, rather than relying on some sort of comparison function to decide which item is less than another, we simply assign a priority value using the `add` or `changePriority` functions.
- There is an additional `changePriority` function that allows us to set the extrinsic priority of an item after it has been added.
- There is a `contains` method that returns true if the given item exists in the PQ.
- There may only be one copy of a given item in the priority queue at any time. To be more precise, if you try to add an item that is equal to another according to `.equals`, the PQ should throw an exception.
- If there are 2 items with the same priority, you may break ties arbitrarily.

Provided Files

We have provided a `NaiveMinPQ`, which is a slow but correct implementation of this interface. `contains`, `getSmallest` and `removeSmallest` use built-in `ArrayList` or `Collections` methods to do a brute-force search over the entire PQ. This takes time proportional to the size of the PQ, or `O(n)`. This implementation does not throw the correct exception for the `add` method. This is because this exception would make this class too slow to be usable for comparing runtimes with an optimized implementation.

We have also provided a class called `PrintHeapDemo.java` that can print out an array as a nice heap drawing. You're welcome to adapt this code for your own use. It might be useful for debugging.

ArrayHeapMinPQ

Your job for this part of the project is to create the `ArrayHeapMinPQ` class, which must implement the `ExtrinsicMinPQ` interface.

Your `ArrayHeapMinPQ` is subject to the following rules:

- **One of your instance variables must be a min heap, and this min heap must be represented as either an array or a `java.util.ArrayList`**, as described in lecture 21 (tree representation 3 or 3b). It is OK to have additional private variables. If you choose to use a `java.util.ArrayList`, you'll find the set method useful.
- You may only import from the `java.util` and `org.junit` libraries. In addition, you may also import the `Stopwatch` and `StdRandom` classes from the Princeton libraries. You're not required to import anything.
- Your `getSmallest`, `contains`, `size` and `changePriority` methods must run in `O(log(n))` time. Your `add` and `removeSmallest` must run in `O(log(n))` average time, i.e. they should be logarithmic, except for the rare resize operation. For reference, when making 1000 queries on a heap of size 1,000,000, our solution is about `300x` faster than the naive solution. Iterating over your entire min heap array (or `ArrayList`) for any of these methods will be linear time and thus too slow!
- If using an array representation for your min heap, make sure to increase the array size by a multiplicative factor when resizing,

otherwise `add` will be linear time on average, not log time. If using a heap ordered `ArrayList`, you don't need to worry about this requirement as `ArrayLists` automatically resize their underlying array.

- If using an array representation for your min heap, it should never be more than **approximately** 3/4s empty. If using a heap ordered `ArrayList`, you don't need to worry about this requirement as `ArrayLists` automatically resize their underlying array.
- You may not add additional public methods. Private methods or "package private" methods are fine. See the testing section below for a description of what "package private" means.

Note: We have not discussed how you should implement the `changePriority` method in lecture. You'll have to invent this yourself. You may discuss your approach at a high level with other students (e.g. drawing out diagrams), but you should not share or look at each other's code, nor should you work closely enough that your code may resemble each other's.

Testing Proj2B

As in Proj2A, we will not provide any skeleton tests nor autograder messages (beyond a basic sanity check) for this project. You will be responsible for writing your own tests and ensuring the correctness of your code. **You may not share tests with any other students - please ensure that all code in your possession, including tests, was written by you.**

You should write your tests in a file called `ArrayHeapMinPQTest.java`. This file should be part of the `bearmaps` package.

If you're not sure how to start writing tests, some tips follow.

1. We encourage you to primarily write tests that evaluate correctness based on the outputs of methods that provide output (e.g. `getSmallest` and `removeSmallest`). This is in contrast to trying to directly test the states of instance variables (see tip #3 below). For example, to test `changePriority`, you might use a sequence of `add` operations, a `changePriority` call, and then finally check the

output of a `removeSmallest` call. This is (usually) a better idea than iterating over your private variables to see if `changePriority` is setting some specific instance variable.

2. Write tests for functions in the order that you write them. You might even find it helpful to write the tests first. Since each function may call previously written functions, this helps ensure that you are building a solid foundation.

3. If you want to write tests that require looking at your `private` instance variables, these tests should be part of the `ArrayHeapMinPQ` class itself. For example, if you want to write a test that only calls the `add` method, there's no way to write it in the manner suggested in tip #1 in `ArrayHeapMinPQTest`. Suppose that you want to verify that your array is `[1, 2, 4, 5, 3]` after inserting 5, 4, 3, 2, 1. In this case, a hypothetical `testAdd54321` method would need to be part of the `ArrayHeapMinPQ` class since the `ArrayHeapMinPQTest` should not be able to access the private instance variables. It's debatable whether you should even write such tests, since tests of type #1 should ideally catch any errors. Follow your heart.

4. One annoying issue in Java is testing of private helper methods. For example, suppose you have a `leftChild` method in `ArrayHeapMinPQ` that you'd like to test in `ArrayHeapMinPQTest`. If this method is private, then the test file will be unable to call the method. For this project, if you have helper methods you'd like to test, you can either include the tests in `ArrayHeapMinPQ`, or you should make those helper methods "package private". To do this, simply remove the access modifier completely. That is, rather than saying `public` or `private`, you should add no access modifier at all. This will make this method accessible only to other classes in the package. You can think of package private as a level of access control in between public (anybody can use it) and private (only this class can use it).

5. Don't forget edge cases - consider how the heap could look before inserting or removing items, and ensure that your code handles all

possible cases (for example, sinking a node when its priority is greater than both of its children).

6. Rather than thinking about all possible edge cases, feel free to perform randomized tests by comparing the outputs of your data structure and the provided `NaiveMinPQ`, similar to what you did in Project 2A.

7. To test the runtime of your code, you should use the `System.currentTimeMillis` method or the `edu.princeton.cs.algs4.Stopwatch` class, similarly to what you did in Lab 5 and Project 2A. Keep in mind that calling the `add` method `N` times will take `O(Nlog(N))` *total* time if each `add` call is `O(log(N))` time. Also, keep in mind that if you call the `contains` method `N` times on a data structure of size `N`, you'd again expect *total* runtime to grow as `O(Nlog(N))`. Note: If you attempt to fit a log function using some sort of fancy software, you will probably be disappointed. Timing tests are likely to be too noisy for such fitting tools to work well. If you're expect `O(Nlog(N))` and you're getting something that looks linear, then you're in good shape.

8. To test that your code throws the proper exceptions, you should leverage JUnit's power. Read this post to see how to do that. Note that as soon as your code throws some type of `Exception`, it will not complete the rest of the lines in that testcase. So, if you have multiple cases that should each throw an exception, you should write multiple testcases: one for each case.

External Resources

You are welcome to look at source code for existing priority queue implementations. Make sure to cite any sources you use with the @source tag somewhere in your code. **As always you should not use other student's solutions to this project as a source**, e.g. roommate's code, something you found on pastebin, github, etc.

You might find the MinPQ implementation from the optional textbook to be a helpful reference. However, you *should not copy and paste* `MinPQ.java` into your implementation and then try to figure out how to bend it to match our spec. While it might seem like this will save you time and effort, it will end up generating more work than just building everything yourself from scratch. Many students tried doing something similar with `AList.java` from lecture and `ArrayDeque.java` from Project 1A, and the results were generally very bad.

Project 2A FAQ

Q: How do I create a `Node[]` if my `Node` class stores objects of generic type `T`? If I try `(Node[]) new Object[10]`, I get a `classCastException` at runtime, but if I try `new Node[10]`, I get a generic array creation error at compile time. This is one particularly annoying issue with Java generics. Without going into the details, the easy fix is that you can simply say `new ArrayHeapMinPQ.Node[10]` and it will work. See this stack overflow post for more details.

Q: When I make a test for `changePriority` and test `NaiveMinPQ` against `ArrayHeapMinPQ`, the runtime for `ArrayHeapMinPQ` isn't substantially better. Let's consider how `NaiveMinPQ` stores items. It stores them in a list, left to right. So if I insert items 0-1,000,000, it would have item `0` at the 0th index, `1` at the 1st index, and so on. Now, when it changes the priority of an item, it has to scan this list from left to right looking for this item. If my test were to insert items 0-1,000,000, then change the priority of items 0-1,000, this is actually the best case input for `NaiveMinPQ::changePriority`, since these are the closest items it can find. We recommend changing your test to *randomly* select 1000 items in the heap and change their priority.

Project 2B Submission & Grading

Required Files for Proj2A

Your Project 2B submission must contain the following files:

- `ArrayHeapMinPQ.java`

- `ArrayHeapMinPQTest.java` (should contain tests)

You can also include other files that you create, e.g. `TestNaiveMinPQ.java`.

Grading

ArrayHeapMinPQ

We will test your `ArrayMinHeapPQ` for correctness and efficiency. Correctness tests will be very similar to the randomized testing described in the Randomized Testing section of the Proj2A spec above, where we will compare your `ArrayMinHeapPQ`'s `getSmallest`, `removeSmallest`, `add`, and `changePriority` results against our staff solution's results.

Efficiency tests will be similar to the correctness tests and will still require correctness, but they will also see how your solution performs with a very large number of items (millions) and many queries to `changePriority` (thousands). Your `changePriority` runtime will be compared to the staff solution's `changePriority` runtime, and we will assign points based on that. If you implement `changePriority` correctly (`O(log(n))`) and do not have repeated, redundant, or unnecessary function calls, your runtime should be fine for these tests.

ArrayHeapMinPQTest

You must submit your tests. However, unlike Project 1B, your tests will not be tested with an autograder.

Point Distribution

- ~75%: `ArrayHeapMinPQ` correctness.
- ~12.5%: `ArrayHeapMinPQ` constructor efficiency.
- ~12.5%: `changePriority` efficiency (only eligible if implementation passes correctness tests).