This assignment is due by **10pm on Monday October 2** and is worth 20 points.

1 Goals

The goal of this assignment is to learn about and get some practice with a Linked List data structure. You will also get lots of practice at following sequences of pointers from one object to another.

2 Your Assignment

This is a partner assignment. Unless I tell you otherwise, you will work with the same partner you had for HW4. This will be the last assignment with that partner.

In this assignment we'll be exploring some **approximate** solutions to a well known problem in computer science called **traveling salesperson problem (TSP)**. The idea behind this problem is as follows: you are given n points on a map (so each point has an x-coordinate and a y-coordinate) and you want to find a route to visit all of the points on the map (and arrive back where you started) while keeping the total distance travelled as small as possible.

There are many places in the real world where the TSP comes up: How does UPS decide the routes that it's trucks take? What is the most efficient way for a machine to drill a set a holes in a circuit board? Some important problems in bioinformatics (genome assembly) can be transformed into instances of the TSP. You'll probably see the TSP in future CS classes.

Greedy Heuristics

The traveling salesperson problem is a notoriously difficult *combinatorial optimization* problem, In principle, one can enumerate all possible tours, but, in practice, the number of tours is so staggeringly large (roughly N factorial where N is the number of stops on the tour) that this approach is useless. For large N, no one knows an efficient method that can find the shortest possible tour for any given set of points. However, many methods have been studied that seem to work well in practice, even though they are not guaranteed to produce the best possible tour. Such methods are called *heuristics*. Your main task is to implement the nearest neighbor and smallest increase insertion heuristics for building a tour incrementally. Start with a one-point tour (from the first point back to itself), and iterate the following process until there are no points left.

- **Nearest Neighbor Heuristic:** For each point you add to the tour, add it **immediately after the point to which it is closest**. (If there is more than one point to which it is closest, insert it after the first such point you discover.)
- **Smallest Increase Heuristic:** For each point you add to the tour, add it **to the position where it results in the least possible increase in total tour length**. (If there is more than one place to insert the point, add it to the first such place you discover.)

Your Task

I have provided you with several auxilary classes that will make your job much easier. Download the `HW6.zip` file from Moodle. The only place that you will have to add code is in the file `Tour.java`. Inside this file I have already defined the nested class `Node` for you to use. Also take a look at the code in the `main()` method which you might find useful for testing out your code. Feel free to take a look at the other files if you would like, but you don't need to understand them. You **should not modify** any of the files except for `Tour.java`.

Point Class

You will use the `Point` object extensively in the code you write. You may look at the code in `Point.java`, but all you really need to know to be a USER of this class is the `Point` ADT:

MethodName(Parameters)	Return type	description
`Point(double x, double y)`	`Point`	Creates a `Point` object with (x,y) coordinates.
`toString()`	`String`	Returns a string representation of a `Point` object.
`draw()`	`void`	Draws a point using standard draw
`drawTo(Point b)`	`void`	Draws a line segment from the Point calling the method to Point b.
`distanceTo(Point B)`	`double`	Returns Euclidean* distance between the Point calling the method and Point b.

* Recall, the Euclidean distance between two points (x_1, y_1) and (x_2, y_2) is defined as follows:

$$\sqrt{(x_1 - x_2)^2 + (y_1 - y_2)^2}$$

Tour Class

The `Tour.java` class that you will implement will have the following methods (think of this at the Tour ADT). See a later section of this document for additional notes and comments on all methods.

MethodName(Parameters)	Return type	description
`Tour()`	`Tour`	Creates an empty tour (has no stops).
`toString()`	`String`	Returns a description of the tour as a nicely formatted string.
`draw()`	`void`	Draws the tour
`size()`	`int`	Returns the number of points, or stops in the tour.
`distance()`	`double`	Returns the total distance in the tour.
`insertNearest(Point p)`	`void`	Inserts `p` into the tour using the nearest neighbor heuristic.
`insertSmallest(Point p)`	`void`	Inserts `p` into the tour using the smallest insertion heuristic.

You will implement these functions using a `single linked list` as the underlying data structure. You will start with the code that I have provided you in `Tour.java`. Just to be clear, you MUST implement your own single linked list to receive credit for this assignment; you may not use any similar class provided by Java.

Additional Method Details and Hints

Here is some additional information about the methods I am asking you to implement.

- `Tour()` : Initialize head node of a linked list and any other variables you think will be useful. I've implemented a version of this method for you. Feel free to modify if you decide to add instance variables.
- `String toString()` : Create an empty String, loop through all nodes of the linked list and add a description of each underlying Point (as a String) to the overall String. `Note: You may want to make use of the toString() method from the Point class.`
- `void draw()` : Loop through all of the nodes in the linked list ad draw each `Point` object, followed by a line from that `Point` to the next `Point`. Do this for all nodes except for the last node in the list, for that node draw a line from the last `Point` to the first (or head) `Point`. Make sure to use the `draw()` `drawTo()` methods in the `Point` class.
- `int size()` : Return the size of the linked list (in terms of `Nodes`). In order to be more efficient, make sure to keep track of this value rather than re-computing this value whenever this method is called.
- `double distance()` : Start with a distance of 0. Loop through all nodes of your linked list and compute the distance from each `Point` to the next `Point`, add this distance to your total distance. When you get to the last `Node` in your list, add the distance from this `Point` to the first `Point`. Make sure to use the `distanceTo(Point b)` method from the `Point` class.
- `void insertNearest(Point p)` : Start with some local variable `minDist = Double.MAX_VALUE` that keeps track of the minimum distance found thus far between `Point p` and the nodes in the linked list. Loop through all `Nodes` of the linked list and compute the distance from the underlying `Point` object to `p`. If the distance is less than `minDist` make that distance the new `minDist` and save a reference to the current node. After your loop finishes, insert a new node for `Point p` **after** the saved node that had the minimum distance to `p`.
- `void insertSmallest(Point p)` : This is the most difficult of all the the methods to implement. Make sure your implement and test all other methods first, before starting on this one. Start with some local variable `smallestDist = Double.MAX_VALUE` that keeps track of the minimum total distance of a path through all nodes. Loop through all nodes of the linked list and determine what the new total distance of a tour would be if you inserted `Point p` after each `Point` currently in the tour. You do not need to loop through the entire linked list to compute the new distance for each insertion point, nor would that be a very efficient approach. Instead, you need to use the following equation:

  ```
  newDistance = currentTotalDistance - currentPoint.distanceTo(nextPoint) +
  currentPoint.distanceTo(p) + p.distanceTo(nextPoint)
  ```

If `newDistance` is less than `smallestDist`, update `smallestDist` with this value and save a reference to the current node. After the loop finishes, insert a new node for `Point p` after the saved node associated with `smallestDist`.

- `main()` : I also include some code in this method to help you with debugging of your code. In particular, I suggest you test each function as you write it and that you implement `toString()` first as it will be useful for debugging other methods. You may certainly add and modify code in this `main()` method as you see fit.

Running Code on Input Files

I include with this assignment two programs that will handle reading a bunch of data points from a file (see below for a description of the file format) and calling the appropriate functions from the tour class. These programs are `NearestInsertion.java` and `SmallestInsertion.java`. To run one of these programs on an input file, for example `tsp3.txt`, you will first need to compile it using `javac`, and then run it with the following command:

```
java NearestInsertion < tsp3.txt
```

Note that the "`< tsp3.txt`" part of the line is just telling Java to get its input from the file `tsp3.txt` instead of waiting for the user to type it at the keyboard. If the program just hangs without doing anything, you probably forgot the "`<`" or the "`< tsp3.txt`". Lastly, we will uses these two programs to test your code, so you should make sure that your code works as expected with them.

Input File Format

The input file format will begin with two integers w and h, followed by pairs of real-valued x and y coordinates. All x coordinates will be real numbers between 0 and w; all y coordinates will be real valued numbers between 0 and h. As an example, the file `tsp3.txt` could contain the following data:

```
600 400
532.6531 247.7551
93.0612 393.6735
565.5102 290.0000
10.0000  10.0000
```

The zip file `HW6.zip` includes several example input files, or you can define your own. Below you will find some information about the expected output for these input file to help you with debugging your own code.

Insert Nearest Solution to `tsp10.txt`

```
$ java NearestInsertion < tsp10.txt
Tour distance =  1566.1363051360363
Number of points = 10
(110.0, 225.0)
```

```
(161.0, 280.0)
(157.0, 443.0)
(283.0, 379.0)
(306.0, 360.0)
(325.0, 554.0)
(397.0, 566.0)
(490.0, 285.0)
(552.0, 199.0)
(343.0, 110.0)
```

Insert Smallest Solution to `tsp10.txt`

```
$ java SmallestInsertion < tsp10.txt
Tour distance =  1655.7461857661865
Number of points = 10
(110.0, 225.0)
(283.0, 379.0)
(306.0, 360.0)
(343.0, 110.0)
(552.0, 199.0)
(490.0, 285.0)
(397.0, 566.0)
(325.0, 554.0)
(157.0, 443.0)
(161.0, 280.0)
```

Insert Nearest Solution to `usa13509.txt`

```
$ java NearestInsertion < usa13509.txt
Tour distance =  77449.97941714071
Number of points = 13509
...
```

See Figure 1 for a visual representation of the solution.

Insert Smallest Solution to `usa13509.txt`

```
$ java SmallestInsertion < usa13509.txt
Tour distance =  45074.77692017051
Number of points = 13509
...
```

See Figure 2 for a visual representation of the solution.

Note: It should take less than a minute (probably even faster than that) to run through the example with 13,509 points (unless you are animating the results). If your code is taking much longer, try to narrow down what part of the code is taking the longest. Turning in an efficient solution will be part of what we consider when grading your code.

Figure 1: Insert Nearest Solution to `usa13509.txt`

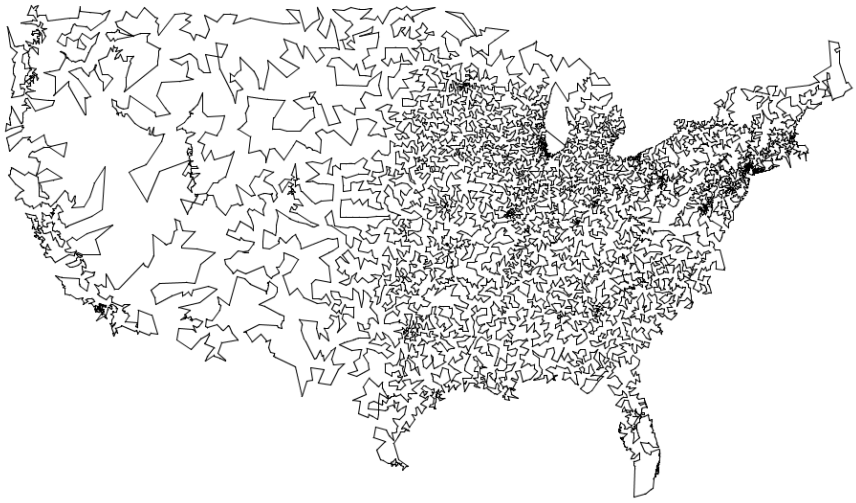

Figure 2: Insert Smallest Solution to `usa13509.txt`

3 Submission and Grading

You'll submit all your files to Moodle as a zipped file. One one partner from each pair needs to submit the assignment. Specifically, put these files into a directory named `[your_last_name_your_partner's_last_name]HW6`, zip this directory, upload it to Moodle. For example, if my partner was Schiller my directory would be named `OesperSchillerHW6` and the resulting file would be `OesperSchillerHW6.zip`.

3.1 Grading

This assignment is worth 20 points. Below is a partial list of the things that we'll look for when evaluating your work.

- Do you implement all of the requested methods as they are described? We'll test this out by running your various methods on different test cases. I suggest you focus on `insertNearest` before attempting `insertSmallest`, which is significantly more difficult. You can still get up to 17 out of 20 points on this assignment if you do everything except for `insertSmallest`.
- How efficient is your code? Are you looping extra times through the list of Nodes?
- Do your classes exhibit good organization and commenting? Don't forget to follow the Style Guidelines on Moodle.

Start early, ask lots of questions, and have fun! Layla, the lab assistants, and the prefect are all here to help you succeed - don't hesitate to ask for help if you're struggling![1]

[1]This assignment modified from `cs.princeton.edu`, COS126 and an assignment by Sherri Goings. Thanks for sharing!