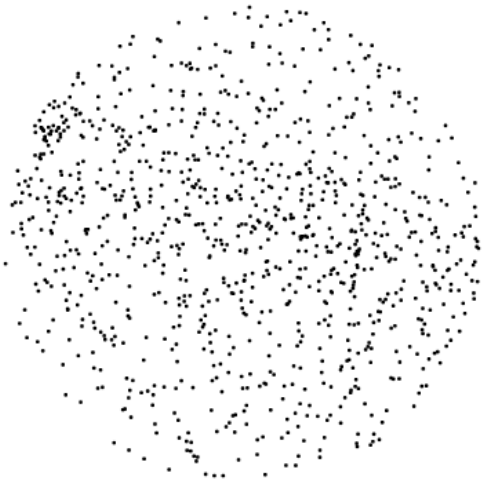
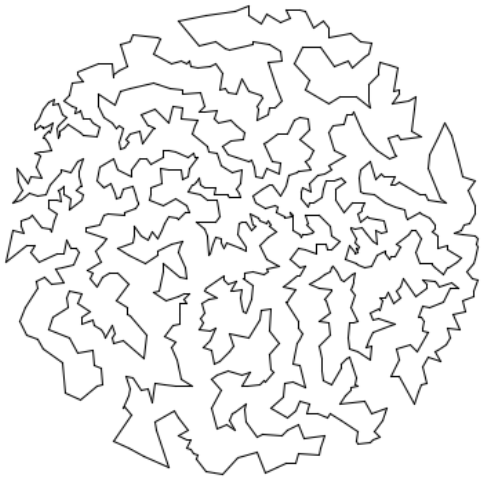
Traveling Salesperson

Given *N* points in the plane, the goal of a traveling salesperson is to visit all of them (and arrive back home) while keeping the total distance traveled as short as possible. In this lab you will use linked structures to implement two greedy heuristics to find good (but not optimal) solutions to the *traveling salesperson problem* (TSP). Read this entire document before beginning, there are hints at the end!



1,000 points



optimal tour

The importance of the TSP algorithm does not arise from an overwhelming demand of salespeople to minimize their travel distance, but rather from a wealth of other applications such as vehicle routing, circuit board drilling, robot control, X-ray crystallography, machine scheduling, and computational biology.

***Greedy heuristics*.** The traveling salesperson problem is a notoriously difficult *combinatorial optimization* problem. In principle, one can enumerate all possible tours and pick the shortest one; in practice, the number of tours is so staggeringly large (roughly *N* factorial) 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:* Read in the next point and add it to the current tour *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:* Read in the next point and add it to the current tour *after* the point where it results in the least possible increase in the tour length. (If there is more than one point, insert it after the first such point you discover.)

The Point data type. You are given a **Point.java** data type that represents a point in the plane, as described by the following API:

```
class Point
-----------------------------------------------------------------------------------------
       Point(double x, double y)     //create the point (x, y)
String toString()                    //return string representation
  void draw()                        //draw point using StdDraw (library file)
  void drawTo(Point that)            //draw a line segment between the two points
double distanceTo(Point that)        //return Euclidean distance between the two points
```

Each Point object can return a string representation of itself, draw itself to standard draw, draw a line segment from itself to another point using standard draw, and calculate the Euclidean distance between itself and another point.

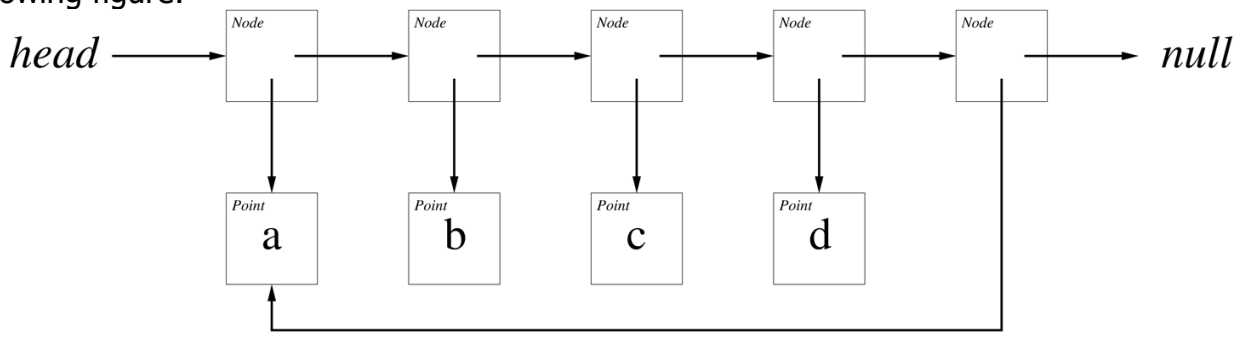
The Tour data type. Your task is to complete the Tour data type that represents the sequence of points visited in a TSP tour. Represent the tour as a *circular linked list* of nodes, one for each point. Each Node will contain a Point and a reference to the next Node in the tour. Within Tour.java, define a nested class Node in the standard way:

```
private class Node {
    Point p;
    Node  next;
}
```

Your Tour data type should implement the following API:

```
class Tour
-----------------------------------------------------------------------------------------
       Tour()                                        //create an empty tour
       Tour(Point a, Point b, Point c, Point d)      //create a 4 point tour a->b->c->d->a
  void show()                                        //print the tour to standard output
  void draw()                                        //draw the tour to standard draw
   int size()                                        //number of points on tour
double distance()                                    //return the total distance of the tour
  void insertNearest(Point p)                        //insert p w/ nearest neighbor heuristic
  void insertSmallest(Point p)                       //insert p w/ smallest increase heuristic
```

Each Tour object should be able to print its constituent points to standard output, draw its points to standard draw, return its number of points, compute its total distance, and insert a new point using either of the two heuristics. The first constructor creates an empty tour; the second constructor creates a 4-point tour and is useful for debugging. Your linked list structure for this case must match the following figure:



***Input and testing*.** The input format will begin with two integers `w` and `h`, followed by pairs of X- and Y-coordinates. All X-coordinates will be real numbers between 0 and `w`; all Y-coordinates will be real numbers between 0 and `h`. The values `w` and `h` are used to set the scale of the GUI window and have already been implemented for you. Multiple test files have been supplied. For example, "tsp1000.txt" contains the following data:

```
tsp1000.txt
775 768
185.0411 457.8824
247.5023 299.4322
701.3532 369.7156
563.2718 442.3282
144.5569 576.4812
535.9311 478.4692
383.8523 458.4757
329.9402 740.9576
...
254.9820 302.2548
```

After implementing Tour.java, use the provided client program NearestInsertion.java to read in the points from file and run the nearest neighbor heuristic; print the resulting tour, its distance, and its number of points to standard output; and draw the resulting tour to standard draw. SmallestInsertion.java works the same way but runs the smallest insertion heuristic.

```
NearestInsertion < tsp1000.txt
Tour distance = 27868.7106
Number of points = 1000
(185.0411, 457.8824)
(198.3921, 464.6812)
(195.8296, 456.6559)
       ...
(264.57, 410.328)
```

```
SmallestInsertion < tsp1000.txt
Tour distance = 17265.6282
Number of points = 1000
(185.0411, 457.8824)
(195.8296, 456.6559)
(193.0671, 450.2405)
         ...
(186.8032, 449.9557)
```

Some hints if you get stuck (you don't have to follow these exactly):

1. Tour.java should include the standard linked list data type Node. Include one instance variable, say `head`, of type Node that is a reference to the "first" node of the circular linked list.

   For debugging purposes, it may help to use the overloaded constructor that accepts four points as arguments, such that you can test this class initially with a smaller list.

2. The method `show` should traverse each Node in the *circular* linked list, starting at `head`, and printing each Point to standard out. This method requires only a few lines of code, but it is important to think about it carefully, because debugging linked-list code is notoriously difficult. Start by just printing out the first Point. With circular linked-lists the last node on the list points back to the first node, so watch out for infinite loops.

   Test your method by writing a `main` function that defines four points, creates a new Tour object using those four points, and calls its `show` method:

```
public static void main(String[] args) {
   //define 4 points forming a square
   Point a = new Point(100.0, 100.0);
   Point b = new Point(500.0, 100.0);
   Point c = new Point(500.0, 500.0);
   Point d = new Point(100.0, 500.0);

   Tour squareTour = new Tour(a, b, c, d);

   squareTour.show();
}
```

   If your method is working properly, you will get the following output for the 4 city problem above.

```
(100.0, 100.0)
(500.0, 100.0)
(500.0, 500.0)
(100.0, 500.0)
```

3. The method `distance` is very similar to `show`, except that you will need to invoke the method `distanceTo` in Point. Add a call to the `distance` method in `main` and print out the distance and size. The 4-point example has a distance of `1600.0`.

4. The method `draw` is also very similar to `show`, except that you will need to invoke the method `drawTo` in Point. You will also need to include the statements:

```
StdDraw.setXscale(0, 600);
StdDraw.setYscale(0, 600);
```

   ...in your `main` method before you call the `draw` method (to initialize the drawing canvas). The four-point example above should produce a square.

5. When attempting to add the points to the tour, it may help to first implement a simpler method named `insertInOrder` to insert a point into the tour after the point that was added last.

It's ok (and expected) that you'll have a special case for the very first point ever inserted, but you shouldn't need any other special cases. The order of the output should match the order of the points in the input file.

6. Implement `insertNearest` before `insertSmallest`. In the `insertNearest` method, determine which node to insert the point `p` after, compute the Euclidean distance between each point in the tour and `p` by traversing the circular linked list. Each time you insert a point, the `size` of the tour should go up by one. Use the current `size` to terminate your loop, when looking for the insertion point.

   As you proceed, store the node containing the closest point *and* its distance to `p`. After you have found the closest node, create a node containing `p`, and insert it *after* the closest node. This involves changing the `next` field of both the newly created node and the closest node.

   As a check, the resulting tour for the 10-city problem has a distance of `1566.1363`. Note that the *optimal* tour has a distance of `1552.9612` so this rule does not, in general, yield the best tour.

7. After doing the nearest insertion heuristic, you should be able write `insertSmallest` by yourself, without any hints. The only difference is that you want to insert the point `p` where it will result in the least possible increase (delta) in the total tour length. As a check, the 10-city tour has a distance of `1655.7462`. In this case, the smallest insertion heuristic actually does worse than the nearest insertion heuristic (although this is not typical).

(Advanced to Over 9000) Shorter path

Implement a better heuristic. For example, observe that any tour with paths that cross can be transformed into a shorter one with no crossing paths: add that improvement to your program. Warning: this is not an easy task to perform. Other options:

Farthest insertion. Just like the smallest increase insertion heuristic described in the assignment, except that the cities need not be inserted in the same order as the input. Start with a tour consisting of the two cities that are farthest apart. Repeat the following:

- Among all cities not in the tour, choose the one that is *farthest* from any city already in the tour.
- Insert it into the tour in the position where it causes the smallest increases in the tour distance.

You will have to store all the unused cities in an appropriate data structure, until they get inserted into the tour. If your code takes a long time, your algorithm probably performs approximately N^3 steps. If you're careful and clever, this can be improved to N^2 steps.

***Node interchange local search*.** Run the original greedy heuristic (or any other heuristic). Then repeat the following:

- Choose a pair of cities.
- Swap the two cities in if this improves the tour. For example, if the original greedy heuristic return a tour of 1-5-6-2-3-4-1, you might consider swapping 5 and 3 to get the tour 1-3-6-2-5-4-1.

Writing a function to swap two nodes in a linked list provides great practice with coding linked lists. Be careful, it can be a little trickier that you might first expect (e.g., make sure your code handles the case when the 2 cities occur consecutively in the original tour).

***Edge exchange local search*.** Run the original greedy heuristic (or any other heuristic). Then repeat the following:

- Choose a pair of edges in the tour, say 1-2 and 3-4.
- Replace them with 1-3 and 2-4 if this improves the tour.

This requires some care, as you will have to reverse the orientation of the links in the original tour between nodes 3 and 2 so that your data structure remains a circular linked list. After performing this heuristic, there will be no crossing edges in the tour, although it need not be optimal.