SPRING 2003: **COP 3530** DATA STRUCTURES
[PROGRAMMING ASSIGNMENT 2; DUE FEBRUARY 11 IN CLASS.]

Problem Description

Your task is to write a program to compute an approximate solution to the traveling salesperson problem. Given a set of N cities with their x and y coordinates, the goal of a traveling salesperson is to visit all the cities (and return home) while keeping the total distance traveled as small as possible. Exhaustive search can help find an optimal tour for small N. For large N, no one knows an efficient method that can find the shortest possible tour for any given set of points, but 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. Note that the length of a tour segment connecting two cities a and b is given by the following distance formula:

$$\text{distance}(a, b) = \sqrt{(x_a - x_b)^2 + (y_a - y_b)^2}$$

where the coordinates of a are (x_a, y_a) and that of b are (x_b, y_b). Also note that the length of a traveling salesperson tour is equal to the sum of the lengths of its N segments.

Your program should implement the heuristic described below for finding a good traveling salesperson tour (i.e., ordering of the cities). The program will work in N iterations and the tour will be built incrementally. The first three iterations are part of the initialization. The first 3 cities are read in and an initial three-city tour is constructed as going from the first city to the second, then to the third, and then back. The first city is considered as "Home". All subsequent iterations follow the same strategy, until we complete the n-th iteration, when there are no more cities to process. The strategy followed in each iteration can be described as follows: *Inspect the next city p and find its nearest neighbor q among the cities in the current tour. Add city p to the current tour by inserting it immediately after q in the tour.*

More explicitly, in iteration $i + 1$, the program reads in the coordinates of the i-th city and "inserts" it into the "partial" tour that it is maintaining on the the first i cities. It has i different possible locations to "insert" the $i + 1$-st city it reads in – either between the first and second city, or between the second and the third, ..., or between the last and the first. It picks one these possibilities by finding which city is closest, and then inserts this city in that location in the partial tour. Therefore, at the end of this iteration, it now holds a partial tour on i+1 cities.

To implement this heuristic, represent the tour as a doubly-linked list of nodes, one for each city. Each node in the linked list will contain the coordinates of the city and information (pointer) about the next city on the tour. The linked list should be implemented using the `LinkedList` class provided in `java.util`. You may not use your own linked list or extend the implementation of the standard `LinkedList` in any way. The methods contained in the `LinkedList` class are described at the following website:

`http://java.sun.com/products/jdk/1.2/docs/api/java/util/LinkedList.html`

These methods should suffice for any manipulation of the list needed for the program.

The first line of the input file will contain an integer indicating the number of cities N to be processed. The rest of the input file will contain a sequence of coordinates of cities. Each line of the file will contain the data for one city. Each city will be provided as a sequence of 2 floating point numbers (to be stored as `double` precision real numbers) representing the x and y coordinates of the city. As you read in the cities, you programs should number them from 0 through $N - 1$. These numbers identify the cities now. Data files will be made available to you soon on the course

homepage. You should create your own (small) data files to test your program. Your program should output the tour produced by applying the heuristic described above. The tour output by the program should be a list of city numbers. In order to make grading easier, your program should output the "Home" city as the first city on the tour. It should also print out the length of the tour.

What to submit: Run your program with the two data files provided to you: `TenPts.dat` and `HundredPts.dat`, and submit the output produced by your program. Also, submit the source code for your program and the output of `Javadoc`. Your floppy diskette should contain all the requisite `.java, .class, .html, .dat, .out` files that are relevant for the grader to check the program. Make sure that the hard copy you submit is the same as the copy on the floppy.

Visualizing the Tour: If you write your own program to visualize the tour (extra credit, as described below), then you do not need to do this part. If not, your program should produce a second output file called `Tour.out`. This file should only contain N lines of output. On each line you should print out the x and y coordinates of the city separated by a space. Then download the four files `DrawTour.html`, `TSP.class`, `DataFile.class`, and `City.class` (from the course web page) to your directory. Open a DOS command window and at the DOS prompt, type in

```
appletviewer DrawTour.html
```

If you have a file called `Tour.out` with the correct data format, then you should see a window that visualizes the tour output by your program.

Submit a print out of the file `Tour.out` for the two runs. Also, print out the resulting tours as visualized by the program `DrawTour`.

Details

City Class: Declare a class called `City` with private data fields called `x`, `y` and `cityNumber`. Implement constructor(s) to initialize the data fields. Also implement methods `getX(), getY(), getCityNumber()` to access the data fields. Declare a method `double distance(City c)` to return the distance from `this` city to city `c`. Finally implement a method `toString()` to print out information about a city.

Tour Class: Declare a class called `Tour` that **contains** a (private) `LinkedList`. Declare a private data field `double TourLength` to maintain the length of the tour. Also implement a method `double tourLength()` to access the length of the tour. Implement method `int findBestInsertPos(City c)` for finding the best insertion point for a new city `c`. It returns an integer i, which should indicate that the best place to insert the new city `c` is between the i-th city and the $(i+1)$-th city on the current tour. Implement a method called `int insert(City c)` that first calls `findBestInsertPos` and then calls the `add` method from class `LinkedList` to insert city `c` into the tour. It should return the index i where the new city was inserted. Implement a method `updateTourLength(int i)` that updates the length of the tour assuming that a new city has just been inserted after the `i`-th city on the tour. Also implement a method `displayTour()` to print out the tour.

Tour Class (Alternate implementation): Based on an alternate suggestion, here is another way to organize the Tour class. Declare a class called `Tour` that **contains** a (private) `LinkedList`. Declare a private data field `double TourLength` to maintain the length of the tour.

Implement method `insertCity(City c)` that finds the best insertion point for a new city `c`, inserts it at that location, and also updates the value of `TourLength`. Also implement method `computeTour(BufferedReader fIn)` that reads in all the cities from file `fIn`. For each city information it reads in, it creates a new object of type `City`, and then calls method `insertCity` to insert it. Also implement a method `displayTour()` to print out the tour. A separate main program then has to open the input and output files, and call methods `computeTour` and `displayTour`.

How to Start: First write a program that simply reads in the input cities and creates a linked list (consisting of cities in the order in which it is provided in the input) and then prints out the contents of the list to the output file. After this is finished and debugged, try the given traveling salesperson heuristic, where the insertion is done more carefully.

Extensions for the bored

Your program should have appropriate comments describing whatever modifications, additions, and/or improvements you make.

- (Easy) Generalize your program so that it works for points in 3-dimensional space.
- (Moderate) For the 2-dimensional case, get the program to display the tour graphically on the screen as a set of points and line segments.
- (Moderate) Implement the following new heuristic: *Add the next city to the existing "partial" tour by inserting it between two successive cities in the tour at the position where it results in the least possible increase in the tour length.*
- (Hard) Tours that cross each other can never be the best traveling salesperson tour. Based on this idea, implement the following heuristic: *If the edges of a tour cross each other, then "uncross" them appropriately.*