Homework #14
due Monday, December 16, 10:00 PM

In this assignment you will implement a `PriorityQueue`, which will be used to implement uniform cost search on a weighted graph.

1 Concerning Uniform Cost Search

Previously, we saw that depth-first search can find a path between two nodes when one exists, and breadth-first search can can find the shortest path. Here we explore the idea of finding a lowest-cost path.

In this assignment, we will augment our graph to store weights along with each edge. So, an edge from `n1` to `n2` migh thave a weight of `3`. We will represent these weights in a 2d-array adjacency matrix, where a value of `-1` indicates no edge, and a positive integer represent an edge with that weight. The total cost of a path equals the sum of the weights of all edges on that path.

If we wish to find the lowest-cost path, we can't use an ordinary `Stack` or `Queue` for our worklist. Instead we need to two additional things. First, store the total path cost along with the edge in the worklist. To achieve this, we implement a class called `ProfileLink`. Second, when we want to visit another node, retrieve the edge with with the lowest total cost from the worklist. To achieve this, we will design a class that implements the `PriorityQueue` ADT.

Uniform cost search is already implemented for you in `WeightedProfileGraph.java`. Make sure you understand it.

2 Concerning the `PriorityQueue`

The ADT for the `PriorityQueue` is quite simple. Its `add` method adds a `ProfileLink` to the `PriorityQueue`, or throws an `IllegalArgumentException` if null. The `remove` method removes and returns the `ProfileLink` with the lowest `cost`, or throws a `NoSuchElementException` if the queue is empty.

To implement this ADT, we will use a minheap data structure. It is an array-based ternary minheap. The `n` elements will be stored in the first `n` indices of the array, such that each element is larger than its implied parent. This means that the `ProfileLink`s stored at indices `1`, `2`, and `3` must have a larger `cost` than the one stored at index `0`, and so on. Adding involves placing an element at the end, and then "bubbling up." Removing involves moving the last element to the first position and then "bubbling down." See the textbook pages 521-527, as well as 649-659 (3rd ed: pp. 511-517, 633-643) and lecture notes for more information.

3 What you need to do

You need to finish the implementation of `PriorityQueue.java`, including the invariant and all public methods.

4 Files

The repository includes the following files:

src/TestInvariant.java Tests your invariant.

src/TestPriorityQueue.java Tests your ADT.

src/TestProfileGraph.java Tests the uniform cost search.

src/TestEfficiency.java Tests your ADT's efficiency.

src/edu/uwm/cs351/PriorityQueue.java The main ADT to work on.

src/edu/uwm/cs351/WeightedProfileGraph.java The weighted graph.

src/edu/uwm/cs351/util/Profile.java The profile class.

src/edu/uwm/cs351/util/ProfileLink.java The data type stored in your ADT.