

Priority Queue; Heapsort; Huffman Code; Graph Traversals

Practice Problems

Practice problems help you get started, if some of the lab and lecture material is not clear. You are not required to do these problems, but make sure you understand how you would solve them. Solving them on paper is a great preparation for the exams.

Finish the implementation of the Priority Queue from the Lab 11.

Pair Programming Assignment

Due Date: 6 April 2011 at 10:00 pm

11.1 Heapsort

Save this work in the directory `Assignment-11-1`

Design the method `heapsort` that consumes an `ArrayList<T>` and a `Comparator<T>` and produces an `ArrayList<T>` sorted in the order defined by the given `Comparator<T>`.

There are two steps to implementing the *heapsort* algorithm, once you have implemented the heap-based priority queue algorithm as described in part 2 of Lab 11:

1. method `heapify`: Insert the given data into your `PriorityQueue`, one item at a time.

2. method `destroyHeap` Remove the data from your `PriorityQueue` and insert them into the resulting `ArrayList<T>`, one at a time.

11.1.1 Extra Credit

Save this work in the directory `Assignment-11-1a`

Design an *in-place* version of the heapsort as follows:

1. **Heapify** Design the method `heapify` that mutates the given `ArrayList<T>` using the given `Comparator<T>` into a priority heap as follows:

   Insert a new **`null`** item at the index 0 so that the index values correspond to the heap node labels.

   Start with the first node that is not a leaf, (at location $k = heapsize/2$ and *downheap* from there.

   Repeat the above step for nodes $k-1, k-2, ...1$.

2. **Build Sorted ArrayList**

   If your priority queue selects the biggest item as the one with the highest priority, you can just move each item (when removed from your heap) the end of the `ArrayList<T>`, but remember to reduce the *size* of the heap area, since the sorted elements will be at the end of your list. Once you remove the last item you will have a list ordered in ascending order (with the unused item at index 0, which can be removed the heap has been destroyed).

   Design the method `buildSorted` that mutates the given `ArrayList<T>` that represents a heap (with a dummy item at index 0) and the given `Comparator<T>` to define the ordering, as follows:

   - Swap the last item in the heap with the first one.
   - Decrease the size of the heap by one.
   - Downheap from the root node (at index 1)
   - Repeat the three steps until the heapsize is equal to 1

3. **HeapSort**

   Design the method `heapsort` that mutates the given `ArrayList<T>` using the given `Comparator<T>` into sorted order by invoking the two methods defined above, and finally removing the unused item at the index 0.

11.2 Huffman Code

Save this work in the directory `Assignment-11-2`

We will now use the solution to the second part of the lab on a practical problem.

1. Start by completing the *Huffman Code* problem from Lab 11.
2. Modify the `computHisto` method so that it consumes an `Iterator<String>`. (This makes it possible to supply the data from a larger collection, such as an entire English dictionary.)
3. Design the method `encodeString` that consumes a `String` (such as a line of text) and a `KeyTree` for the encoding, and produces a `String` that is the encoding of the letters in the given `String` based on the code that the `KeyTree` represents.

   Note: Think carefully before you decide which class each of these methods should be defined.

11.3 Graph Traversal Algorithms

11.3.1 Graph Algorithms for US states

Start by checking in all the work you have done for the graph algorithms last week. Use a directory `Assignment-10-1`

The Background

In the first part of this assignment (last week) you have designed three variants of the *graph traversal* algorithms: the *Breadth-First Search*, the *Depth-First Search*, and the *Shortest Path Search*. The three algorithms should have been using the same code, except for the implementation of the *to-do* data structure, as follows:

- Depth-First Search: uses a `Stack` to record the `ToDo` information
- Breadth-First Search: uses a `Queue` to record the `ToDo` information
- Shortest Path Search: uses a `Priority Queue` to record the `ToDo` information

Our goal now is to turn your program into a library that can be used by anyone to find routing in an arbitrary graph.

The Graph Traversal Library

Download the files in **GraphLibrary.zip**. Create a project `Assignment-11-3` and import all files into it. You should have the following files:

- `USmap.java`: a representation of the 48 continental US states
- `City.java`: a representation of a state capital of the 48 US states
- `State.java`: a representation of one node in the graph that represents the US map
- `EuroGraph.java`: a representation of the map of central European states
- `Node.java`: an interface that represents a node in a graph. The three methods provide all information the graph algorithms need to implement the traversals.
- `Graph.java`: an abstract class that is used by the *graph traversal* algorithms. It includes a representation of a graph, a method to initialize the graph data, a method for extracting the list of neighbors of any node, a method for computing the distance between two nodes in the graph, and a method to provide a `String` representation of the graph.

  The two classes `USmap` and `EuroGraph` **`extend`** this class and show you two ways of designing a new graph yourself.
- `ToDo.java`: an interface that provides all methods that the *graph traversal* algorithms need to use the *to-do* information.

  Last week you should have designed three classes that implement this interface: `ToDoQueue` for *Breadth-First Search*, `ToDoStack` for *Depth-First Search*, and `ToDoPriorityQueue` for *Shortest Path Search*.
- `GraphAlgorithms.java`: This class provides a skeleton for the library class you need to implement. **The main part of the assignment is to complete the design of this class.** A more detailed description is given below.
- `GraphAlgoView.java`: A class that generates a GUI dialog for interactively running several algorithms. For each run you can select the origin, destination, and the algorithm/search variant.

The class is initialized with an instance of a `Graph` and an instance of the `GraphAlgorithms` class. The buttons that select the algorithms invoke the methods defined in the `GraphAlgorithms` class: `findRouteBFS`, `findRouteDFS`, or `findRouteSPS`.

- `jpt.jar` is library used by `GraphAlgoView` to generate the GUI components and handle the user interactions. You will learn more about it in the lab on April 12th.

11.3.2 `ToDo` Classes

Complete the implementation of the three classes that implement the `ToDo` interface: `ToDoQueue`, `ToDoStack`, and `ToDoPriorityQueue`.

11.3.3 `GraphAlgorithms` class

Complete the design of the `GraphAlgorithms` class.

The class `GraphAlgorithms` contains three fields needed by all the graph taversal algorithms: `graphData`, `path`, and `todolist`. It also implements three methods with headers (and obvious purposes):

```
ArrayList<FromTo> findRouteBFS(String origin, String destination)
ArrayList<FromTo> findRouteDFS(String origin, String destination)
ArrayList<FromTo> findRouteSPS(String origin, String destination)
```

Each method invokes the following method passing a new `todolist` appropriate for the selected algorithm:

```
/** Produce the path from the given origin to the
 *    given destination, using the given 'to do' list */
ArrayList<FromTo> runAlgo(String origin, String destination, ToDo todolist){
   this.todolist = todolist;
   this.path = new Path();
   todolist.add(new FromTo("", origin, 0), this.path);
   return this.buildPath(destination);
}
```

Your task is to supply the missing method that implements the graph traversal algorithms you have worked on last week, and any helper methods you may need.

```
ArrayList<FromTo> buildPath(String destination){ ... }
```

You will also need `toString` methods to display the route directions as `String`. You do not need to display the graph of the map or animate the routing.

11.3.4 Additional Examples

Create a class that represents a new graph with at least 8 nodes and 20 edges. Add tests that use this graph with your `GraphAlgorithms` class.

You should be able to swap with a friend/classmate and use each other's graphs interchangeably.

11.3.5 Testing

Remember, designing tests for every part of your program will make your life much easier. You will know what it should do (especially if you write the purpose statements carefully), what it actually does, and how it can be used in further design. If that is not enough motivation, you can also get points for the good test design, or lose points if the tests are not sufficient.

Add several tests that use your new graph.