CSE 373, Winter 2013
Homework Assignment #5: The Even-More-Amazing Heap (25 points)
Due Friday, February 15, 2013, 11:30 PM

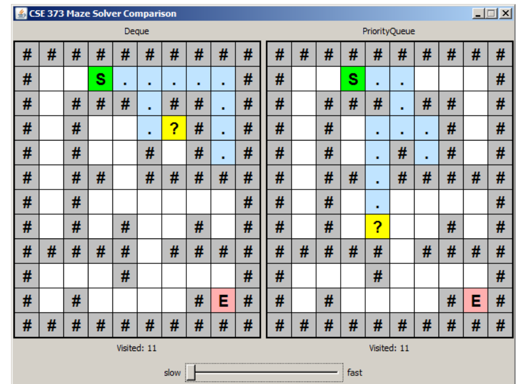
This document and its contents are copyright © University of Washington. All rights reserved.

This program focuses on implementation of a *priority queue* collection using a heap as the internal data structure, as well as practicing writing a comparator to perform external ordering of a collection of objects. Turn in files named `DequeSolver.java`, `PriorityQueueSolver.java`, and `HeapPriorityQueue.java`.

You will also need to download various support Java files and input text files from the Homework section of the course web site; place them in your Java project and add to it. If you are using Eclipse, place .java files in your project's `src/` folder and .txt files in the project root folder (the parent of `src/`). We also provide a 'stub' version of the other files on the web site as a starting point if you like. Some of these files are similar to ones used in your previous Homework 3, but **please re-download all support files**, because all of them have changed from their HW3 versions.

Part A (Maze Solver Comparison) Description:

This program consists of two parts. In the first part, you will revisit the Maze Solver portion of your previous Homework 3. In HW3, you wrote a class to solve a maze using a deque. In Part A of this assignment you will write a new maze solver algorithm that uses a priority queue instead of a deque. You will also grab your old HW3 Maze Solver code and adapt it to be part of this assignment to compare it to the new priority queue version.



A **priority queue** is an ADT for a queue that allows fast access/removal of the minimum element, providing the appearance of a sorted queue. Priority queues are an excellent choice for algorithms involving processing a set of choices or possibilities in some order based on importance or urgency.

We are providing you with essentially the same `Maze` class as in HW3. You will write a class `PriorityQueueSolver` that implements the algorithm described by the following pseudo-code. The pseudo-code is identical to the one in HW3 except as indicated in bold:

```
solve(maze):
   Keep a priority queue of squares to visit, initially empty.
      The priority queue should be ordered by direct distance from the end square,
      with closer squares being 'smaller' or earlier in the queue order.

   Put the maze's start location in the priority queue.
   Repeat the following until the priority queue is empty, or until we solve the maze:
      L := Remove the 'minimum' location from the priority queue.
      If L is the maze's end location, then we have solved the maze.
      If we have already visited L before, or L is a wall, ignore it.
      Otherwise:
         Mark location L as being visited.
         Add unvisited neighbors of L (up 1, down 1, left 1, right 1) to your p.queue.
   If you end up with an empty p.queue and have not found the end, there is no solution.
```

The algorithm examines possible paths, looking for a path from the start to the end. It is more efficient if it examines paths that move closer to the end first. So it takes advantage of the nature of a priority queue by ordering its elements such that neighbors that are closer to the end square will be removed and processed earlier. This is similar to the algorithm we used with our deque in the previous maze solver assignment, but the deque did not fully order the elements in this way; it just gave a crude ordering where closer neighbors were put in the front of the deque and further neighbors were put at the back. This gave the deque a bit of ordering but not a full and accurate ordering as will be found in the priority queue. You will find that the priority queue version of this algorithm generally reaches the end location in much fewer moves than the deque version did (though not always; for certain mazes the deque wins, mostly by accident).

Part A is the shorter and less challenging part of the assignment, intended to get you familiar with using priority queues and showing you a practical use of this collection to solve an interesting problem.

Part A Implementation Details:

We have modified our provided code for this assignment somewhat. We provide you an interface called `Solver` that can be implemented by classes containing algorithms for solving mazes. The `solve` method's behavior is as it was in HW3.

```
public interface Solver {
    public boolean solve(Maze maze);
}
```

The first task for this part is to get your `MazeSolver` code from Homework 3 and rename the class to `DequeSolver` (also rename the file to `DequeSolver.java`) and indicate that it implements the `Solver` interface:

```
public class DequeSolver implements Solver { ...
```

(If your old `MazeSolver` is using your `ArrayDeque` from HW3, for this assignment change it to java.util's versions of `Deque` and `LinkedList` instead, so that you won't need to re-turn-in your HW3 solution code.)

Now you will write a second implementation of `Solver` named `PriorityQueueSolver`. For this one you will use Java's provided `PriorityQueue` class from `java.util`. The methods of `PriorityQueue` are listed on the next page in the description of Part B. In Part B you will write your own implementation of a priority queue.

Your priority queue will be storing maze locations as `java.awt.Point` objects. By default, a priority queue orders its elements by their "natural ordering" as described by the `Comparable` interface and the `compareTo` method of the element type. But `Point` objects are not `Comparable` and have no `compareTo` method. So you must define an external ordering by writing an inner class that implements the `Comparator` interface. You are sorting the points by their direct distance from the maze's end location. So when the comparator is given two points to compare, it should examine each one and compute its distance from the end location, and return an appropriate integer (< 0, 0, or > 0) based on which is closer to the end. `Point` objects have a `distance` method you should use to see how far apart they are from each other.

Here is a brief pseudo-example of using Java's provided `PriorityQueue` class with a comparator:

```
import java.util.*;   // using Java's provided PriorityQueue
...
Comparator<Point> comp = new MyComparatorClass(...);
Queue<Point> example = new PriorityQueue<Point>(999, comp);
example.add(...);
```

The `DequeSolver` and `PriorityQueueSolver` each have just one public method listed below.
You **may define additional methods** if you like, but they must be `private`.

`public boolean solve(Maze maze)`
In this method you will search for a path in the given maze from the start location to the end location using the algorithm described on the previous page, marking each square you visit along the way. If you find a path from the start to the end, you should return `true`; if not, you should return `false`. This method is called by the provided `Main` class, passing you the maze for the input file the user indicates. If the maze passed is `null`, you should throw a `NullPointerException`. If the maze is non-`null`, you may assume that its state is completely valid, though some mazes do not have any successful path from the start to the end location.

Your `PriorityQueueSolver` interacts with our provided `Maze` class. Its methods are listed in the HW3 spec. We also provide a `GraphicalMaze` class with the same methods as `Maze` plus an animated graphical user interface. You can generally re-use your maze code from HW3, only making modifications regarding the data structure being used.

Though the code for `DequeSolver` and `PriorityQueueSolver` will be similar, do not worry about redundancy between the classes. Our intent is that you will copy-paste your `DequeSolver` code into `PriorityQueueSolver` and then modify it to use the priority queue and comparator. You don't need to try to factor out common code between the two files, though if there is redundancy within one file, you should still try to reduce that redundancy, as always.

The majority of the points for this part of the assignment are for the `PriorityQueueSolver`, not the `DequeSolver`.

Part B (HeapPriorityQueue) Description:

Part B is the more substantial part of the assignment where you will implement a priority queue using a heap. Specifically, you will use an array to emulate the behavior of a *binary min-heap*, where the root is stored at index 1, its left/right children at indexes 2/3 respectively, the left-left child at 4, the left-right child at 5, the right-left at 6, etc.

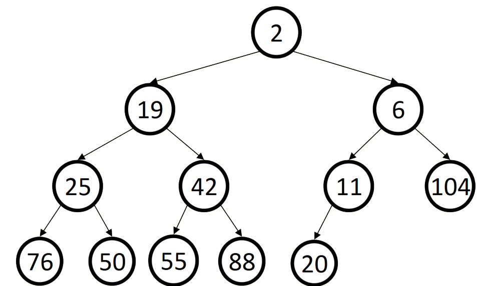
The basic heap operations such as adding and removing elements should be implemented as described in lecture and in the Weiss textbook. Your heap should remain a complete tree with proper vertical ordering after each public method of your class finishes executing.

When a new element is added, it should be placed at the rightmost leaf, then "bubbled up" as necessary to restore heap ordering. When the minimum element is removed, the rightmost leaf is moved into its place, then "bubbled down" as necessary to restore heap ordering.

If an arbitrary element (one other than the min/root) is removed, it should first be "bubbled up" completely to the top/root of the tree, disregarding comparable ordering of the elements; then it should be removed in exactly the way the remove-min operation is performed, swapping the rightmost leaf into the root slot and bubbling it down into its proper place.

The following diagram shows a possible valid state for a heap, as both a tree picture and the corresponding array state:

index	0	1	2	3	4	5	6	7	8	9	10	11	12	13
value	/	2	19	6	25	42	11	104	76	50	55	88	20	...
size	12													...



Part B (HeapPriorityQueue) Implementation Details:

You will write a class named `HeapPriorityQueue<E>` that implements our provided `PriorityQueue<E>` interface. (In the real Java class libraries in `java.util`, the `PriorityQueue` is a class rather than an interface. We will make `PriorityQueue` an interface, and its specific implementation with a heap will be a class implementing that interface.)

Your class has the following methods; you must write them with exactly the header shown, so it can be called by the provided classes. You **may define additional methods** in your class if you like, but they must be `private`. You will also need to declare any private data fields needed by the class in order to implement this behavior. All specified methods must run in O(1) average runtime unless otherwise specified.

Several of these methods were shown in lecture. Such methods are marked with an asterisk (*). You may want to go copy the code from the lecture files, though you may need to modify it so that it fully meets the guidelines in this spec.

`public HeapPriorityQueue()   *`

In this constructor you will initialize your newly constructed empty priority queue. Your queue will order its elements according to their natural ordering, as per `Comparable` / `compareTo`. You should give it a heap array length of 10. The constructors are O(*capacity*) because Java must initialize and clear out the array elements as an array is constructed.

`public HeapPriorityQueue(int capacity, Comparator<E> comparator)`

In this constructor you will initialize your newly constructed empty priority queue that uses the given capacity as its internal heap array size, and that uses the given comparator to order its elements. If the comparator passed is `null`, your queue should order its elements according to their natural ordering, as per `Comparable` / `compareTo`. If the capacity passed is less than 2, you should throw an `IllegalArgumentException`. O(*capacity*).

`public void add(E element)   *`

In this method you should add the given element value to your priority queue. If the element is `null`, you should throw a `NullPointerException`. The average runtime should be O(log *N*). (You should add the element in such a way that your internal heap's ordering properties are maintained. You may need to resize your heap array to fit the new element. Resizing is O(*N*) but if you resize to a multiple of the old size, the average runtime for adding is O(log *N*).)

Note that in the version of `HeapPriorityQueue` shown in class, we assumed that the generic type `E` implemented the `Comparable` interface so that we could compare objects for ordering. But you can't declare your `E` to implement `Comparable` because that would make your priority queue unable to hold elements that lack a natural ordering and are instead ordered by a comparator. So if a Comparator was passed to your constructor, use that to compare elements; but if not, then cast your element to `(Comparable<E>)` momentarily as you want to compare it. The cast will throw a `ClassCastException` if the type `E` is not actually comparable; this is okay and expected. The cast generates a compiler warning that you can turn off via `@SuppressWarnings("unchecked")`.

`public void` **`clear`**`()`

In this method you should remove all elements from your queue. Make sure that your internal array stores only `null` values after a call to `clear` so that the memory previously occupied by the elements can be freed. This method is O(*N*).

`public void` **`contains`**`(E value)`

In this method you should return `true` if the given value is found in your priority queue. If the element is `null`, you should throw a `NullPointerException`. This method does not need to be clever or take advantage of the heap's ordering; you should just do a linear search over the elements. This method is O(*N*).

`public boolean` **`isEmpty`**`()`

In this method you should return whether your queue does not contain any elements. O(1).

`public Iterator<E>` **`iterator`**`()`

In this method you should return an iterator that provides access to the elements of your priority queue in front-to back order. The iterator is described in more detail on the next page.

`public E` **`peek`**`()   *`

In this method you should return the minimum element of your priority queue without modifying the state of the queue. If the queue does not contain any elements, you should throw a `NoSuchElementException`. O(1).

`public E` **`remove`**`()   *`

In this method you should remove and return the minimum element of your priority queue. Remove the element such that your internal heap's ordering properties are maintained. If the queue does not contain any elements, you should throw a `NoSuchElementException`. You should `null` out the now-empty array slot. See the notes above about ordering and casting type `E` to be able to perform comparisons; they also apply here. O(log *N*).

`public void` **`remove`**`(E value)`

In this method you should remove the given value from your priority queue, if it is found in the queue. If it appears in the queue multiple times, you must remove exactly one of the occurrences; which one is unspecified. If the element is `null`, you should throw a `NullPointerException`. This method does not need to be clever or take advantage of the heap's ordering; you should just do a linear search over the elements to find the one to remove. You should `null` out the now-empty array slot. This method is O(*N*).

`public int` **`size`**`()   *`

In this method you should return the number of elements in your priority queue. O(1).

`public String` **`toString`**`()`

In this method you should return a string representation of your priority queue's elements. Though the client might expect the string to list the elements in sorted order, this is inefficient, so you should display them in front-to-back order by their index in your internal heap. Show them in square brackets separated by a comma and space; for example, if the heap contains C, S, and E at array indexes 1-3, you would return `"[C, S, E]"`. For an empty queue return `"[]"` and for a one-element queue return just that one element in brackets such as `"[hello]"`. This method is O(*N*).

Iterator Class:

Your priority queue's `iterator` method should return an iterator object that traverses the elements of the queue. The client might expect the iterator to return the elements in sorted order, but this is inefficient, so you should return them in front-to-back order by their index in your internal array. Implement your iterator as a private inner class.

The iterator must support the `hasNext` and `next` (but not `remove`) operations. You will generally find that this iterator is not as tricky to implement as the one for your deque, because the order to loop over the collection is straightforward.

You may assume that the queue is not modified during iteration. Your iterator class should have the following methods:

`public boolean` **`hasNext`**`()`

Your iterator returns `true` from this method if there are still more elements for the iterator to return. When the client calls this method, it does not visibly modify your iterator's state nor advance its position forward in the queue. O(1).

`public E` **`next`**`()`

In this method your iterator should return the next element from your queue that had not yet been returned by the iterator, and advances your iterator's position to the following element. If your `next` method is called when there is no next element remaining, your iterator should throw a `NoSuchElementException`. O(1).

`public void` **`remove`**`()`

Your iterator does not need to support removal. If this is called, throw an `UnsupportedOperationException`.

Style Guidelines and Grading:

We will grade your program's behavior and output to give you an **External Correctness** score. You can use the course web site's Output Comparison Tool to check your maze solver and check test case output for your priority queue.

We will also grade your program's code quality (**Internal Correctness**). There are many general standards of Java coding style that you should follow, such as naming, indentation, comments, avoiding redundancy, etc. For a list of these standards, please see the **Style Guide** document posted on the class web site. You should follow all of its guidelines as appropriate on this and future assignments.

In Part B, you are implementing a collection of your own. **For this reason, you may not use any of Java's built-in data structures (such as the ones from `java.util`) to implement your `HeapPriorityQueue`.** For example, declaring a private field in your class of type `ArrayList` or `Map` would be unacceptable. A solution that disobeys this restriction will receive a very substantial deduction. Your priority queue's inner heap array should be the only data structure of any kind that is declared by your code.

Your priority queue must be implemented as a binary **heap** stored in an array as described in class to receive substantial credit. Solutions that do not do so will receive substantial deductions for both external and internal correctness. You must implement the heap ordering behavior properly to receive full credit.

You must also match the expected **Big-Oh** demands of each method. Relaxing the Big-Oh could make it easier to implement the functionality, such as if you were to shift elements or copy the array's entire state frequently or other such things; but the whole point of implementing this structure is to do it efficiently. So this will be a grading focus.

Watch out for **redundant code**. If you are performing identical or very similar commands repeatedly, factor out the common code and logic into a helper method, loop, or other facility to remove the redundancy. We strongly recommend that you write some **private helper methods** to help you implement the required public functionality of `HeapPriorityQueue`. Specifically, the operations of bubbling elements up/down the heap and comparing elements for ordering are done repeatedly and should not be redundantly repeated in the code. Our own solution uses 12 helpers.

Do not declare a value as a private field unless it is necessary to retain it as part of the state of your object. `PriorityQueueSolver` needs at most ~0-1 fields. `HeapPriorityQueue` should probably have ~3-4 fields at most.

In your method comment headers make sure to comment what exceptions if any are thrown by each method and why they are thrown. In addition to including comment headers on each class and each method, also make sure to include inline comments next to any complex or tricky code (for example, in the code for the Part A maze solving algorithm).