COL106 - Data Structures and Algorithms
Assignment 5

Courtesy: Univ of Washington, https://courses.cs.washington.edu/courses/cse373/13wi/homework/5/spec.pdf

Due: October 31st, 2014

Total Marks: 30

This assignment revisits the maze-solver of assignment 2. Instead of a doubly-linked list, a priority queue will be used.
Please download the required files from the link given on the course webpage. You are required to complete the **TODO** sections in the `Main.cpp` for Section 1 and `PriorityQueueArray.cpp` for Section 2.

Note: A set of data files to use as input is also provided on the website. We do not guarantee that these form an exhaustive test. Perform additional testing of your own before you submit your program. A Makefile is also provided. Use `make` to compile all files. Use `make test` for compiling only the files required for Section 2. The default executable will be `a.out`.

Submission Guidelines

Submit only 2 files: `Solve.txt` (It should have only the solve() definition) and `PriorityQueueArray.cpp` in a zipped format. Name your zipped file as ⟨EntryNumber⟩_⟨FirstName⟩_⟨LastName⟩.zip.

1 Maze Solver (10 Marks)

Input File: maze1.txt

```
10 12
##########
#  S     #
# ### ## #
# #    # #
# #  # # #
# ## #####
# #      #
# # #  # #
##### ####
#   #    #
# #    #E#
##########
```

A Priority Queue is an Abstract Data Type (ADT) that allows fast access to the minimum element in a collection.
You will use a priority queue to implement an algorithm to search for a path to escape from a 2-D maze. We are providing you with a `Maze` class representing the maze with various useful data and operations and `PriorityQueueADT` interface representing the priority queue that you have to use. You will complete the

TODO sections of `Main.cpp` so that it implements the algorithm described by the following pseudo-code:

Pseudocode for solve(Maze, DequeADT⟨ Point ⟩):

```
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 we have already visited L before, or L is a wall, ignore it.
    Else
        Mark location L as being visited.
        If L is the maze's end location, then we have solved the maze.
        Else
            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(S) to the end(E). It is more efficient if it examines paths that move closer to the end first. This can be done by using a priority queue – one that gives the point that is closest to the end point (among the points that it has).

Implementation Details

The method you will implement in this section is named `solve()` in the file `Main.cpp`. It is just one method; you must write it with exactly the header shown, so that it can be called by the provided `main()` within Main.cpp. You may define additional methods in your class if you like, but they must be private. (You probably should not need any data fields to implement this behavior.)

The maze and point classes are the same as in Assignment 2.

If the maze passed is null, you should handle it in an appropriate manner (without changing any other part of the file). 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.

Running and Testing

Run "make" to compile the files. Run "./a.out <absolute-path-to-maze-file> 0" to run the program and solve the maze. You can use the files maze1.txt, maze2.txt and maze3.txt and their respective solutions to test/verify your code. Note that, if your priority queue has two points having the same distance from the end point, then which one it returns when querying the minimum element is undefined. Consequently, different implementations might explore a maze differently. However, if a solution exists, all correct implementations will find it.

2 PriorityQueueArray (15 Marks)

In this part, you will write your own implementation of a priority queue. Specifically, you will use an array to emulate the behavior of a binary min-heap, as discussed in class.

The ordering of the elements is based on the comparator specified in `comparator.h`. Given two elements, this comparator orders them according to their distance from the end point – the one closer to the end point is ordered before the farther one. A priority queue built using this comparator returns the point closest to the end point when top() is called.

Unlike assignment 2. here there is no need to define a templated class. It is sufficient for priority queue to support only "Point" objects.

Implementation Details

You will complete the class in file `PriorityQueueArray` that implements our provided `PriorityQueueADT` interface. It 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.

- `public PriorityQueueArray(Point * destination)`
  In this constructor, you will initialize a newly constructed empty priority queue. Your priority queue must use the comparator in `comparator.h`. Give it a default array capacity of 10.

- `public void push(const Point& val)`
  In this method, you should add the given element value `val` to the priority queue. If the element is null, you should handle it appropriately. (You may need to resize your array to fit the new element.)

- `public boolean empty()`
  In this method, you should return whether your priority queue does not contain any elements.

- `public Point top()`
  In these methods, return the minimum element. If the priority queue does not contain any elements, you should handle it appropriately.

- `public void pop()`
  In these methods, remove the minimum element. If the priority queue does not contain any elements, you should handle it appropriately.

- `public int size()`
  In this method, you should return the number of elements in your priority queue.

NOTE: *You can use the file TestPriorityQueue.cpp to test/verify your code. Run "make test". The expected solution can be found in* `priority_queue_test_correct_solution.txt`*.*

3 Maze Solver Revisited (5 Marks)

Now that you have your own `PriorityQueueArray` class that you have implemented in Section 2, re-run the Maze Solver described in Section 1 using it. Run "make" to compile, followed by "./a.out <absolute-path-to-maze-file> 1" to run the solver using your priority queue implementation.