**CS1027: Assignment 3**

**Due: March 19, 11:55pm.**

**Weight: 9%**

**Purpose**

To gain experience with

* The solution of problems through the use of priority queues
* The design of algorithms in pseudocode and their implementation in Java.

**1. Task**

In this assignment you will design and implement a program in Java to find a way for the warrior from the last assignment to exit the dungeons of the Bee Queen. However, this time your program is required to find the shortest possible path from the initial chamber to the exit, if one exists. Your algorithm **must** use a priority queue, as described below. To make things simpler for you, for this assignment you will not need to consider cacti or lava chambers (except for the bonus part as described below).

For this assignment, then, there will be 4 types of chambers (these are the same as in Assignment 2):

* Empty chambers. The warrior can safely walk across one of these chambers to move to an adjacent one.
* Dragon chambers. Each one of these chambers is a dragon lair. The warrior cannot go through these chambers or through any chamber adjacent to a dragon chamber. The warrior can enter a dragon chamber or a chamber adjacent to a dragon chamber, but when he realizes that he is in one of these chambers he has to leave it right away.
* Wall chambers. The warrior cannot enter them.
* Exit chamber. This allows the warrior to exit the dungeon. The exit chamber is an empty chamber.

Initially the warrior is positioned in an empty chamber. The following figure shows an example of a dungeon. The shortest path from the initial chamber to the exit goes through chambers 1, 2, 5, 7, 9, and 11 and has length 6. Note that there might be other paths of the same length, like 1, 2, 5, 6, 9, 11; your algorithm just needs to find one of the paths of shortest length. There might also be other longer paths, like 1, 2, 3, 5, 7, 9, 11 which your algorithm will not select. Paths that go through dragon chambers or through chambers adjacent to dragon chambers must not be selected by your algorithm.

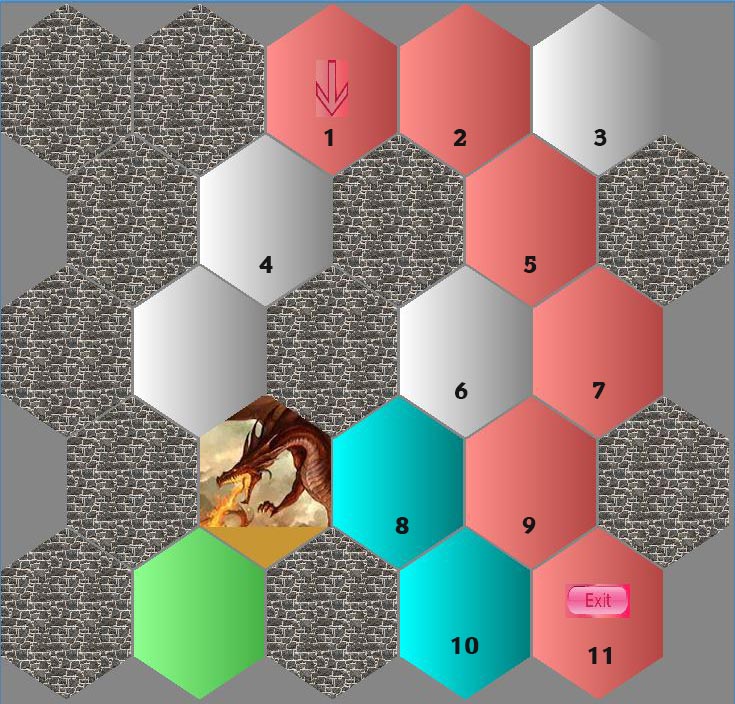
**2. Classes to Implement**

**2.1 DLinkedPriorityQueue**

A priority queue is an ADT that stores a collection of data items, where each item has a priority. For this assignment the priorities of the data items are going to be values of type *double*. The operations provided by this ADT are the following: add, removeMin, updatePriority, isEmpty, size, and toString. The operations are described below. You are provided with a Java interface for the priority queue ADT called *PriorityQueueADT.java*. Your class *DLinkedPriorityQueue.java* will implement all the methods in the *PriorityQueueADT.java* interface and it will store the data items of the priority queue in a **doubly linked list**. The header for this class will then be

*public class DLinkedPriorityQueue<T> implements PriorityQueueADT<T>*

This class will have three private instance variables:



*d*

*d*

Wall chamber

Exit

Initial chamber

* *private DPriorityNode<T> front*. This is a reference to the first node of the doubly linked list.
* *private DPriorityNode<T> rear*. This is a reference to the last node of the doubly linked list.
* *private int count*. The value of this variable is the number of data items in the priority queue.

This class needs to provide the following methods.

* *public DLinkedPriorityQueue()*. Creates an empty priority queue.
* *public void add (T element, double priority)*. Adds to the priority queue the given element with its associated priority.
* *public void updatePriority (T element, double newPriority) throws InvalidElementException*. Changes the priority of the given element to the new value. Notice that depending on how you implement the priority queue, you may need to update where the node storing the given element is within the doubly linked list. An *InvalidElementException* is thrown in the given element is not in the priority queue.
* *public T removeMin() throws EmptyPriorityQueueException*. Removes and returns the data item in the priority queue with smallest priority. If several items in the priority queue have the same smallest priority, any one of them is returned. An *EmptyPriorityQueueException* is thrown if the priority queue is empty.
* *public boolean isEmpty()*. Returns *true* if the priority queue is empty and it returns *false* otherwise.
* *public int size().* Returns the number of data items in the priority queue.
* *String toString().* Returns a *String* representation of the priority queue. The priority queue will store data items of the type *Hexagon*, described below. This method will just invoke the *toString* method from each data item in the priority queue and concatenate these strings.

When storing the data items in the priority queue you have two choices:

1. Keep the data items sorted by priority. In this case the data item with smallest priority will be at the front of the linked list. When inserting a new data item into the priority queue, it needs to be inserted in its proper position within the sorted linked list. Similarly, when updating the priority of a data item, the node storing such a data item might need to be moved so that the linked list remains sorted after the priority of the data item has been updated. Or
2. keep the data items unsorted. In this case when performing the *removeMin* operation, the list needs to be scanned to look for a data item with smallest priority. When adding a new data item, it should be inserted at the rear of the linked list.

Both choices are OK. Depending on which choice you make some operations will be more complicated to implement and other will be simpler.

**2.2 DPriorityNode**

This class represents the nodes of the doubly linked list used to implement the priority queue. This class will be declared as follows:

*public class DPriorityNode<T>*

This class will have 4 instance variables:

* *private T element*. A reference to the data item stored in this node.
* *private DPriorityNode<T> next*. A reference to the next node in the liked list.
* *private DPriorityNode<T> prev*. A reference to the previous node in the linked list.
* *private double priority*. This is the priority of the data item stored in this node.

You need to implement the following methods in this class:

* *public DPriorityNode (T data, double prio).* Creates a node storing the given data item and priority.
* *public DPriorityNode().* Creates an empty node, with null data and zero priority.
* Setter and getter methods: *getPriority, getElement, getNext, getPrev, setElement, setNext, setPrev, setPriority.*

**2.3 FindShortestPath**

This class will implement the algorithm to compute a shortest path from the initial chamber to the exit. A description of this algorithm is given below. This class will contain the main method:

*public static void main (String*[] *args)*

This method will first create an object of the class *Dungeon* passing *args*[0] as the parameter to the constructor of class *Dungeon*; *args*[0] should store the name of the file containing the description of the dungeon. See the notes on command line arguments in Assignment 2. As in the previous assignment, when your program creates an object of the class *Dungeon*, the dungeon will be displayed on the screen. Description of the class *Dungeon* is given below.

Your program must print the number of chambers in the path from the initial chamber to the exit. For example, for the dungeon of the above figure the algorithm must print a message indicating that the path has length 6. If there is no path from the initial chamber to the exit, your program must print an appropriate message. You might add any private methods and instance variables that you want in this class.

Your program must catch any exceptions that might be thrown. For each exception thrown an appropriate message must be printed. The message must explain what caused the exception to be thrown instead of just a generic message saying that an exception was thrown**.**

**3. Algorithm for Finding a Shortest Way Out of the Dungeon**

The algorithm described here is called the *A\* algorithm*. Make sure you understand the algorithm before you implement it. There are online tutorials explaining this algorithm, in case you want to learn more about it. Use the Internet to find some of these tutorials. The algorithm will start at the initial chamber and as it traverses the dungeon, it will keep in the priority queue the chambers that it might visit next. Each chamber has a priority equal to the distance from the chamber to the initial chamber plus an estimation of the distance from the chamber to the exit.

As the algorithm traverses the chambers it will compute the distance from each visited chamber to the initial one (see algorithm below to learn how to do this). Each object of the class *Hexagon* representing a chamber has an instance variable where it stores the distance to the initial chamber. This value will be set when the warrior arrives at the chamber represented by the *Hexagon* object; note that this value is not known before the warrior arrives at the chamber.

Since the distance to the exit from a given chamber is not known, the algorithm will estimate it as the *Euclidean* *distance* between the current chamber and the exit (See example below. If this is confusing to you, do not worry too much; you will be provided with code for computing the Euclidean distance from a chamber to the exit). Because the algorithm visits chambers in increasing order of priority, it will always try to move to the chamber that seems to be closest to the exit. Each object of the class *Hexagon* has an instance variable that will point to the predecessor chamber in the shortest path from the initial chamber to the current chamber.

Consider, for example the dungeon given in the above figure. The warrior will start at chamber 1 and he will set the distance to the initial chamber to 0. Then it will examine neighboring chambers 2 and 4. For each of these chambers the distance to the initial chamber will be set to 1 and the predecessor of each one of these chambers will be set to chamber 1. Since the Euclidean distance between two chambers is the length of the straight line connecting the centers of the chambers, the warrior will assign the same priority, 1 + *d*, to chambers 2 and 4, where *d* is the Euclidean distance between chambers 2 and 11 or between chambers 4 and 11 (which should be the same, see figure).

Next the warrior will examine the chamber with the smallest priority, say 2 (as chambers 2 and 4 have the same priority). From here it will examine the neighboring chambers to chamber 2, namely 3 and 5. The distances from 3 and 5 to the initial chamber are set to 2 (distance from current chamber, chamber 2, to the initial chamber –which was *d* = 1– plus the additional step to get to chambers 3 or 5); chamber 2 is set as the predecessor of chambers 3 and 5. The priority of chamber 5 will be set to a lower value than the priority of chamber 3 as the Euclidean distance from 5 to the exit is smaller than the distance from chamber 3 to the exit. The warrior keeps examining chambers, avoiding those containing a dragon and those adjacent to a dragon, until he reaches the exit or determines that the exit cannot be reached.

A more detailed description of the algorithm in pseudocode is given below:

* First, create an empty priority queue.
* Get the starting chamber from class *Dungeon*, described below. Each chamber is represented with an object of class *Hexagon*, also described below.
* Add the starting chamber to the priority queue with a priority of zero. Mark the chamber as *enqueued* (see methods of class *Hexagon* to see how to do this).
* Now, ***while*** the priority queue is not empty,***and***the exit has not been found perform the following steps:
  + Remove the chamber with smallest priority from the priority queue and mark it as *dequeued* (see class *Hexagon*).
  + If the current chamber is the exit, then the algorithm exits the loop.
  + If the current chamber has a dragon in it or if any of the neighbouring chambers has a dragon, then this means that this chamber cannot be part of the solution; go back to the ***while*** loop to try a different path.
  + Otherwise, consider each one of the neighbouring chambers to the current one that are not of type *wall* and have not been marked as *dequeued*. For each neighbouring chamber, let’s call it *neighbour,* of the *current* chamber do as follows:
    - Let *D* be equal to 1 + distance from *current* to the initial chamber.
    - If distance of *neighbour* to initial chamber is larger than *D* then set the distance of *neighbour* to the initial chamber to *D*. Note that if the previous condition is true, it means that the previous value stored in *neighbour* as the distance from it to the initial chamber was incorrect.

You also need to set *current* as the predecessor of *neighbour*. See class *Hexagon* to learn how to set the predecessor of a chamber. The predecessor values will allow the algorithm to reconstruct the path from the entrance to the exit, once the exit has been reached.

* + - If *neighbour* is marked as *enqueued* and the distance from *neighbour* to the initial chamber was modified in the previous step, then invoke the *updatePriority* method to update the priority that *neighbour* has in the priority queue. The new priority of *neighbour* will be the distance from *neighbour* to the initial chamber plus the distance from *neighbour* to the exit. The distance to the exit is computed by method *getDistanceToExit* from the *Hexagon* class described below. The distance to the initial chamber is obtained by invoking method *getDistanceToStart* from the *Hexagon* class.
    - Otherwise, if *neighbour* is not marked as *enqueued*, then add it to the queue with priority equal to its distance to the initial chamber plus its distance to the exit. Finally mark *neighbour* as *enqueued*.

**4. Classes Provided**

You are given several java classes. Some of these classes are the same as those for the previous assignment, but some of them have been modified. Read the below descriptions of the classes carefully to see which classes are different. Full documentation for the code provided is in the course’s website.

* **Class *Dungeon***. This class represents the dungeon. The methods that you might use from this class are the following:
  + *public Dungeon (String inputFile) throws InvalidDungeonCharacterException, IOException, FileNotFoundException.* Reads the input file and displays the dungeon on the screen. An *InvalidDungeonCharacterException* is thrown when the *inputFile* contains an invalid character. Look at the sample input files to learn which characters are allowed.
  + *public Hexagon getStart().* Returns a *Hexagon* object representing the starting chamber.
  + *public int numChambers().* Returns the number of chambers in this dungeon. You might need this method only if you decide to implement the optional part for the bonus marks, as explained below.
* **Class *Hexagon***. This class represents a chamber of the dungeon. Objects of this class are created inside class *Dungeon* when the dungeon file is read. The methods that you might use from this class are the following:
  + *public Hexagon getNeighbour (int i) throws InvalidNeighbourIndexException*. Each chamber of the dungeon has up to six neighbouring chambers, indexed from 0 to 5. For each value for the index *i*, from 0 to 5, the method might return either a *Hexagon* object representing a chamber or *null*. Note that if a chamber has fewer than 6 neighbouring chambers, these neighbours do not necessarily need to appear at consecutive index values. So, it might be that *this.getNeighbour*(0) and *this.getNeighbour*(3*)* are null, but *this.getNeighbour*(*i*) for all other values of *i* are not null.

An *InvalidNeighbourIndexException* exception is thrown if the value of the parameter *i* is negative or larger than 5.

* + *public boolean* methods: *isDragon*(), *isEmpty*(), *isWall*(), *isExit*(), return true if *this* *Hexagon* object represents a chamber of type dragon, empty, wall, or exit, respectively. There is also method *isLava*() if you want to implement the bonus part.
  + *public boolean isMarkedEnqueued*() returns true if ***this*** *Hexagon* object represents a chamber that has been marked as *enqueued*.
  + *public boolean isMarkedDequeued*() returns true if ***this*** *Hexagon* object represents a chamber that has been marked as *dequeued*.
  + *public void markEnqueued*() marks ***this*** *Hexagon* object as *enqueued*.
  + *public void markDequeued*() marks ***this*** *Hexagon* object as *dequeued*.
  + *public int getDistanceToExit(Dungeon d).* Returns the Euclidean distance from the chamber represented by ***this*** *Hexagon* object to the exit. The parameter is the *Dungeon* object containing ***this*** *Hexagon* object.
  + *public int getDistanceToStart*()*.* Returns the distance from the chamber represented by ***this*** *Hexagon* object to the initial chamber.
  + *public void setDistanceToStart(int dist)*. Sets the distance from the chamber represented by ***this*** *Hexagon* object to the initial chamber to the specified value.
  + *public void setPredecessor(Hexagon pred).* Sets the predecessor chamber of ***this*** *Hexagon* object in the shortest path to the initial chamber to the specified value.
  + *public Boolean equals(Hexagon otherChamber).* Returns true if the specified value points to ***this*** *Hexagon* object; otherwise it returns false.
* **Exception Classes: *InvalidDungeonCharacterException, EmptyPriorityQueueException, InvalidElementException, InvalidNeighbourIndexException.***

**5. Image Files and Sample Input Files Provided**

You are given several image files that are used by the provided java code to display the dungeon on the screen. You are also given several input dungeon files that you can use to test your program. In Eclipse put all these files inside your project file in the same directory where the default package and the JRE System Library are. **Do not** put them in the src folder as Eclipse will not find them there. If your program does not display the dungeon correctly on your monitor, you might need to move these files to another folder, depending on how your installation of Eclipse has been configured.

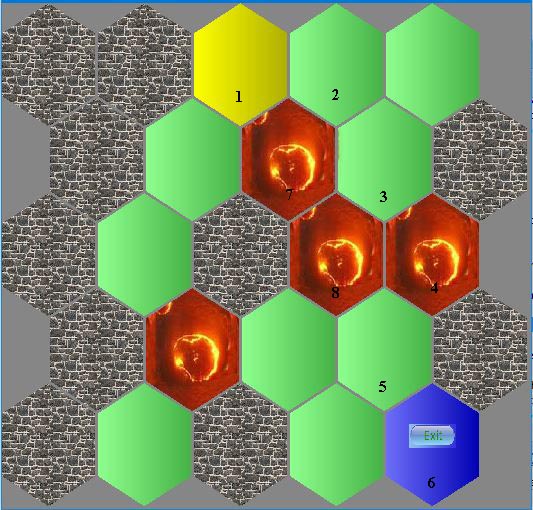
**6. Extra Credit (5% Bonus)**

For the extra credit your algorithm must find shortest path from the initial chamber to the exit when the dungeon can also contain lava chambers:

* If there is a way to go from the initial chamber to the exit that uses only empty chambers, then your algorithm must find a shortest path that goes only through empty chambers from the initial chamber to the exit.
* Otherwise, your algorithm must find a path form the initial chamber to the destination that uses the **least** number of lava chambers. If the minimum number of lava chambers in such a path is, say, *k* then the path computed by the algorithm must use the smallest number of empty chambers among all paths from the initial chamber to the destination that use *k* lava chambers.

None of the above paths can go through a chamber that is adjacent to a chamber hosting a dragon.

For example, for the dungeon shown in the following figure, every path to the exit needs to go through at least one lava chamber. There are several paths that use one lava chamber; two of these paths have 5 empty chambers and the others have either 6, 7, or 8 empty chambers. Hence, the algorithm must select a path with one lava chamber and 5 empty chambers, for example the path that goes through chambers 1, 2, 3, 4, 5, and 6. Note that the path 1, 7, 8, 5, 6 cannot be selected by your algorithm even though it has only 5 chambers because it includes 2 lava chambers.



exit

Initial chamber

**7. Non-functional Specifications**

1. Assignments are to be done individually and must be your own work. Software will be used to detect cheating.
2. Include comments in your code in javadoc format. Add javadoc comments at the beginning of your classes indicating who the author of the code is and a giving a brief description of the class. Add javadoc comments to methods and instance variables. Read information about javadoc in the second lab for this course.
3. Add comments to explain the meaning of potentially confusing parts of your code.
4. Use Java coding conventions and good programming techniques. **Read the notes about comments, coding conventions and good programming techniques in the first assignment**.

Submit all your .java files to OWL. **DO NOT** put the code inline in the textbox. **Do not submit your *.class* files. If you do this, and do not attach your *.java* files, you will receive a mark of zero!**

**8. What You Will Be Marked On**

1. Functional specifications:

* Does the program behave according to specifications?
* Does it run with the test input files provided?
* Are your classes created properly?
* Are you using appropriate data structures?
* Is the output according to specifications?

1. Non-functional specifications: as described above
2. Assignment submission: via OWL assignment submission.