CS1027 Assignment 3

Due: March 22, 11:55pm.
Weight: 9%

Purpose

To gain experience with

- The solution of problems using linked lists
- 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 path in the map from the last assignment from the Western Power Company (WPC) to the house of the customer C. However, this time your program is required to find a shortest path, if one exists. Your program **must** use a doubly linked list, as described below.

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

- an initial map cell, where WPC is located,
- a map cell where the house of customer C is situated,
- map cells containing blocks of houses of other customers,
- map cells containing electrical switches. There are 3 types of electrical switches:
    - omni switches. An omni switch located in a cell L can be used to interconnect all the neighboring map cells of L. A cell L has at most 4 neighboring cells that we will denote as the north, south, east, and west neighbors. The omni switch can be used to interconnect all neighbors of L;
    - vertical switches. A vertical switch can be used to connect the north and south neighbors of a map cell;
    - horizontal switches, A horizontal switch can be used to connect the east and west neighbors of a map cell.

The same restrictions as in Assignment 2 must be followed about the order in which electrical switches can be added to a valid path. For example a vertical switch in cell *p* can be added to the path only if the previous cell of the path is the north or south neighbor of *p* and that neighbor is the WPC cell, an omni switch or a vertical switch.

The following figure shows an example of a map in which the shortest path from the WPC cell to the customer cell goes through cells 1, 3, 4, 8, 12, 11 and has length 6. Note that there might be other paths of the same length, like 1, 3, 2, 6, 10, 11; your algorithm just needs to find one of the paths of shortest length. There might also be other, longer paths, like 1, 3, 4, 5, 9, 13, 12, 11which your algorithm will not select.

2. Classes to Implement

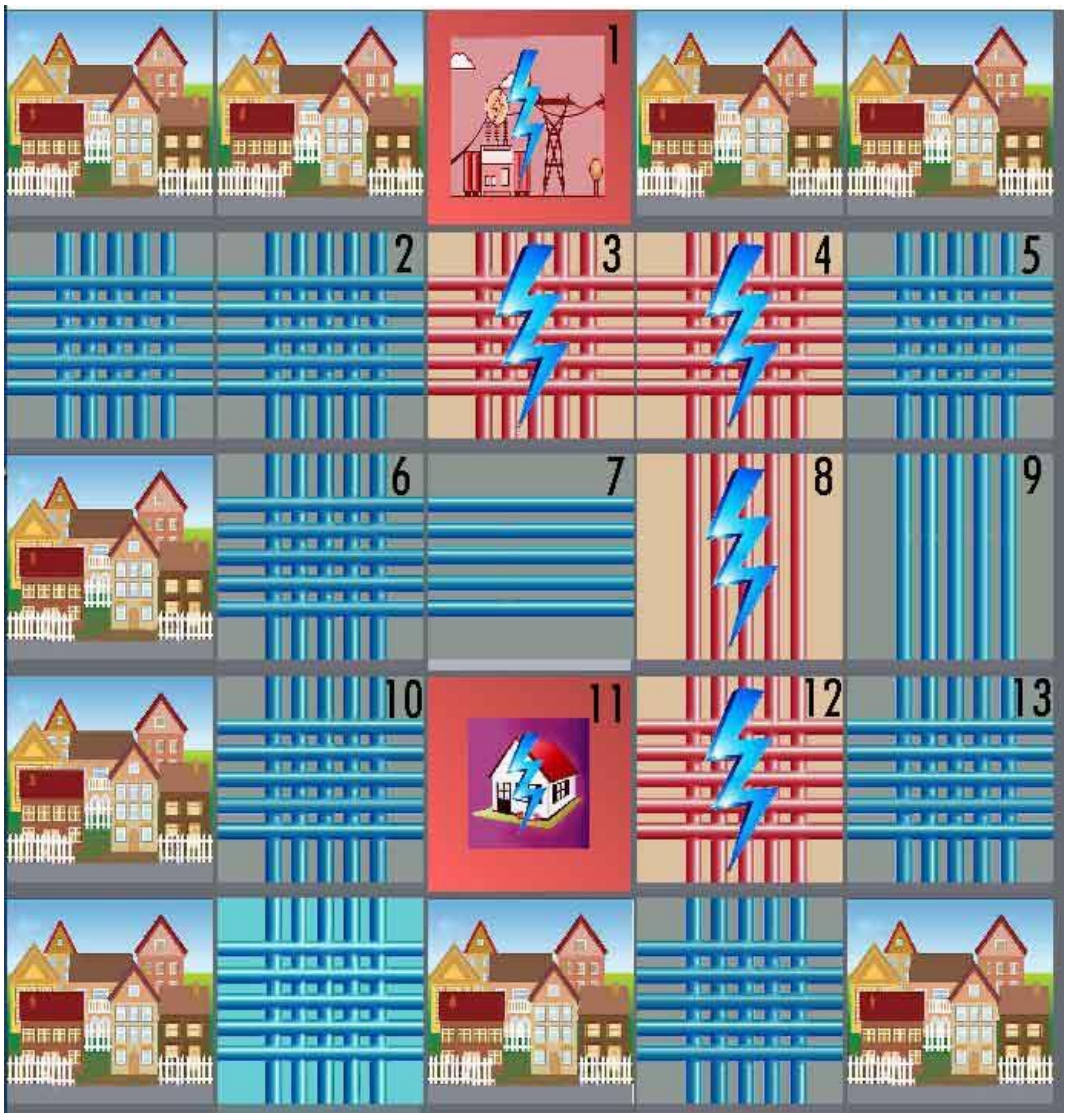
2.2 *DLNode*

This class represents the nodes of a doubly linked list. Each node stores two things: a data item and an associated integer value. This class will be declared as follows:

public class DLNode<T>

This class will have four instance variables:

- *private T dataItem*. A reference to the data item stored in this node.
- *private DLNode<T> next*. A reference to the next node in the liked list.
- *private DLNode<T> previous*. A reference to the previous node in the linked list.
- *private int value*. This is the value of the data item stored in this node.



You need to implement the following methods in this class:

- *public DLNode (T data, int value).* Constructor for the class. Initializes a node storing the given *data* and *value.*
- Setter and getter methods: *getValue, getData, getNext, getPrevious, setData, setNext, setPrevious, setValue.*

2.1 *DLList*

This class implements a doubly linked list in which the nodes are of the class *DLNode*. This class must implement the interface *DLListADT.java* that specifies the public methods in this class. The header for this class will then be

public class DLList<T> implements DLListADT<T>

This class will have three private instance variables:

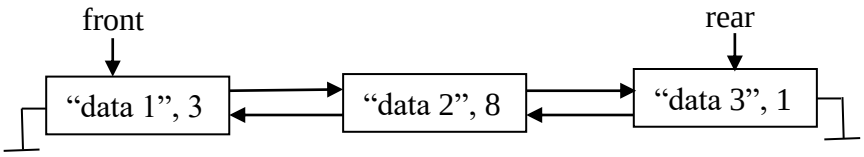
- *private DLNode<T> front*. This is a reference to the first node of the doubly linked list.
- *private DLNode<T> rear*. This is a reference to the last node of the doubly linked list.
- *private int count*. The value of this variable is the number of data items in the linked list.

This class needs to implement the following methods.

- *public DLList()*. Creates an empty list.
- *public void insert (T dataItem, int value)*. Adds a new *DLNode* storing the given *dataItem* and value to the **rear** of the doubly linked list.
- *public int getDataValue(T dataItem) throws InvalidDataItemException*. Returns the integer value associated to the specified *dataItem*. An *InvalidDataItemException* is thrown if the given *dataItem* is not in the list.

  Note that to check whether *dataItem* is in the list you need to scan the linked list using linear search. To check if the data item stored in a node of the linked list is the same as *dataItem* you must use the *equals* method, not the "==" operator.
- *public void changeValue (T dataItem, int newValue) throws InvalidDataItemException*. Changes the value of *dataItem* to *newValue*. An *InvalidDataItemException* is thrown if the given *dataItem* is not in the list.
- *public T getSmallest() throws EmptyListException*. Removes and returns the data item in the list with smallest associated value. If several items in the list have the same associated value, any one of them is returned. An *EmptyListException* is thrown if the list is empty.

  Note that to find the data item with smallest value the entire linked list needs to be scanned using linear search.
- *public boolean isEmpty()*. Returns *true* if the list is empty and it returns *false* otherwise.
- *public int size()*. Returns the number of data items in the list.
- *String toString()*. Returns a *String* representation of the linked list. This method will invoke the *toString* method from each data item in the priority queue and concatenate these strings to produce a string of the form: "List: dataItem1,value1; dataItem2,value2; …".



To make sure you understand the operations that you need to implement, consider the above doubly linked list where each nod stores a String and an integer value. If we invoke *getDataValue* ("data 2"), this method should return the value 8, while invoking *getDataValue*("data 4") should throw an *InvalidDataItemException*. Invoking *changeValue*("data 1", 0) will set the value stored in the first node to 0; if now we *getSmallest*(), this will return "data 1" (as now this data item has the smallest value) and the first node of the list will be removed.

2.3 *ShortestPath*

This class will have an instance variable

Map cityMap;

This variable will reference the object representing the city map where WPC and C are located. This variable must be initialized in the constructor for the class, as described below. You must implement the following methods in this class:

- *public ShortestPath (String filename).* This is the constructor for the class. It receives as input the name of the file containing the description of the city map. In this method you must create an object of the class *Map* (described in Section 5) passing as parameter the given input file; this will display the map on the screen. Some sample input files are also provided in the course's website. Read them if you want to know the format of the input files.
- *public static void main (String[] args).* This method will first create an object of the class *ShortestPath* using the constructor *ShortestPath(args[0])*. When you run the program, you will pass as command line argument the name of the input file. Your *main* method then will try to find a shortest path from the WPC cell to the destination cell C following the switches only in the allowed order (see also Assignment 2). Suggestions on how to look for a shortest path are given in the next section. The code provided to you will show the path selected by the algorithm, so you can visually verify if your program works.
- *private MapCell nextCell(MapCell cell).* The parameter is the current cell. This method returns the first unmarked cell that can be added to the current path. This method is very similar to method *bestCell* from Assignment 2, except that now we do not prefer a customer cell or omni cell over horizontal or vertical cells, instead, the first cell that can be used to continue the path will be returned by the method.

  If there are no unmarked cells adjacent to the current one that can be used to continue the path, this method returns `null`. As in Assignment 2, you will use the methods from the provided *MapCell* class to mark map cells.

  For example, in the figure in page 2, if the current cell is cell 4 and cell 3 is marked, but cells 5 and 8 are not marked, then the method will return cell 5. If the current cell is cell 3, and cells 1 and 4 are marked, then the method will return cell 2. Finally, if the current cell is cell 13, and cells 9 and 12 are marked, this method will return null.

  Hint. Simply modify you *bestCell* method from Assignment 2 so that no preference is given to customer or omni cells.

Your program must print the number of cells in the shortest path from the initial cell to the customer cell, if a path exists. For example, for the map in page 2 the algorithm must print a message indicating that the shortest path has length 6. If there is no path from the initial cell to the destination, your program must print an appropriate message.

If you want, you might add other private methods and/or instance variables to any of the required classes.

Important. Your program must catch any exceptions that might be thrown. Any invocation to a method that throws an exception must be inside a try-catch block. 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 Path

This section describes an algorithm for computing a shortest path. You do not have to implement this algorithm if you do not want to. You are encouraged to design your own algorithm, but the algorithm must use a doubly linked list from class *DLList*.

The algorithm starts at the cell with the WPC and as it traverses the map, it will store in the doubly linked list the map cells that it might visit next. Each node of the linked list will store a map cell and an integer value equal to the distance from that cell to the WPC cell. As the algorithm looks for the path to the customer cell it will keep track of the distance from the current cell to the WPC cell. A description of the algorithm is given below.

- First, create an empty doubly linked list.
- Get the initial cell where the WPC power company is located by using method *getStart()* from class *Map*. Each map cell is represented with an object of the class *MapCell*, described below.
- Insert the initial cell in the linked list with a distance value of zero (as the distance from the initial cell itself is zero). Mark this cell as *in the list* (use method *markInList()* from class *MapCell*).
- ***While*** the list is not empty, ***and*** the customer cell has not been reached, perform the following steps:
    - Remove from the linked list the cell *S* with smallest distance value and mark it as *out of the list* (use method *markOutList* from class *MapCell*).
    - If *S* is the customer cell, then the algorithm exits the while loop.

      Otherwise, consider each one of the neighbouring cells of *S* that is not *null*, is not a block of houses, has not been marked, and can continue the path from *S* (to find these cells you will use your method *nextCell* described above). For each one of these neighbouring cells *C* perform the following steps:
        - Set *D* = 1 + distance from *S* to the initial cell.
        - If the distance between *C* and the initial cell (to find this distance use method *getDistanceToStart* from class *MapCell*) is larger than *D* then:
            - set the distance of *C* to the initial cell to *D* (to do this use method *setDistanceToStart* from class *MapCell*).
            - Set *S* as the predecessor of *C* in the path to the initial cell (use method *setPredecessor* from class *MapCell*); this is necessary to allow the algorithm to reconstruct the path from the initial cell to the destination once the destination has been reached.
        - Set *P* = distance from *C* to the initial cell.
        - If *C* is marked as *in the list* and *P* is less than the distance value stored in the node of the doubly linked list storing *C* (to find this value use method *getDataValue* from class *DLList*) then use the *changeValue* method from class *DLList* to update the distance value of *C* to *P*.
        - If *C* is not marked as *in the list*, then add it to the doubly linked list with distance value equal to *P*. Then mark *C* as *in the list*.

Note. Recall that you are expected to use meaningful names for your variables when implementing this algorithm. So, do not use *C*, *P* and *S* as names for your variables; use more meaningful names.

Consider, for example the map given in page 2. The algorithm starts at cell 1 and t sets the distance from this cell to the initial cell to 0. Then it examines the neighboring cells; since the east and west neighbors are blocks of houses they are ignored. Cell 3 is added to the list with a distance value of 1 and the predecessor of cell 3 is set to cell 1.

In the next iteration of the while loop, the algorithm removes the node storing cell 3 from the linked list. Then two nodes are added to the list containing cells 2 and 4; for each one of these nodes the associated distance value is 2 (as the distance between cells 2 and 1 and between cells 4 and 1 is 2).

Next the algorithm removes from the list the cell with smallest value, say cell 4 (as cells 2 and 4 have the same value). From here the algorithm will examine the neighboring unmarked cells of cell 4, namely 5 and 8. Nodes storing cells 5 and 8 are added to the list, each with associated distance value 3; cell 4 is set as the predecessor of cells 5 and 8. In the next iteration cell 2 will be removed from the list.

The algorithm keeps examining cells in this manner until it reaches the customer cell *C* or it determines that the destination cannot be reached. When the customer cell is found, the code given to you will automatically highlight in red the path that your algorithm has found.

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 carefully the descriptions of the classes.

- **Class *Map***. This class represents the city map. The methods that you might use from this class are the following:
  - *public Map (String inputFile) throws InvalidMapException, IOException, FileNotFoundException.* Reads the input map file and displays the map on the screen. An *InvalidMapException* is thrown when *inputFile* does not contain a valid map. Look at the sample input files to learn their format.
  - *public MapCell getStart().* Returns a *MapCell* object representing the initial cell.
- **Class *MapCell***. This class represents a map cell. Objects of this class are created inside class *Map* when the map file is processed. The methods that you might use from this class are the following:
  - *public MapCell getNeighbour (int i) throws InvalidNeighbourIndexException*. Each map cell has up to four neighbouring cells, indexed from 0 to 3. For each value for the index *i*, from 0 to 3, the method might return either a *MapCell* object representing a cell or *null*. Note that if a cell has fewer than 3 neighbouring cells, these neighbours do not necessarily need to appear at consecutive index values. So, it might be, for example, 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 3.
  - *public boolean* methods: *isOmniSwitch*(), *isVerticalSwitch*(), *isHorizontalSwitch*(), *isBlock*(), *isPowerStation*(), *isCustomer*() return true if ***this*** *MapCell* object represents a cell of type omni switch, vertical switch, horizontal switch, a block of houses, the power station, or the customer house, respectively.
  - *public boolean isMarkedInList*() returns true if ***this*** *MapCell* object represents a cell that has been marked as *in the list*.
  - *public boolean isMarkedOutList*() returns true if ***this*** *MapCell* object represents a cell that has been marked as *out of the list*.
  - *public void markInList*() marks ***this*** *MapCell* object as *in the list*.
  - *public void markOutList*() marks ***this*** *MapCell* object as *out of the list*.

- *public int getDistanceToStart*(). Returns the distance from the cell represented by ***this*** *MapCell* object to the initial cell.
- *public void setDistanceToStart(int dist)*. Sets the distance from the cell represented by ***this*** *MapCell* object to the initial cell to the specified value.
- *public void setPredecessor(MapCell pred).* Sets the predecessor of ***this*** *MapCell* object to the specified value.
- *public Boolean equals(MapCell otherCell).* Returns true if *otherCell* points to ***this*** *MapCell* object; otherwise it returns false.

- **Exception Classes:** *InvalidDataItemException, InvalidMapException, EmptyListException, 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 map on the screen. You are also given several input map 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 map, you might need to move these files to another folder, depending on how your installation of Eclipse has been configured.

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 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 and/or major 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**.
5. Submit all your .java files to OWL. **DO NOT** put the code inline in the textbox. Do not submit a compressed file (.zip, .rar, .gzip, …) with your code. Do not submit your *.class* files. If you do this, and do not attach your *.java* files, your assignment cannot be marked.

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