CS1027: Foundations of Computer Science II
Assignment 2
Due: February 28, 11:55pm.

Purpose

To gain experience with

- The solution of problems through the use of stacks
- The design of algorithms in pseudocode and their implementation in Java.
- Handling exceptions

1. Introduction

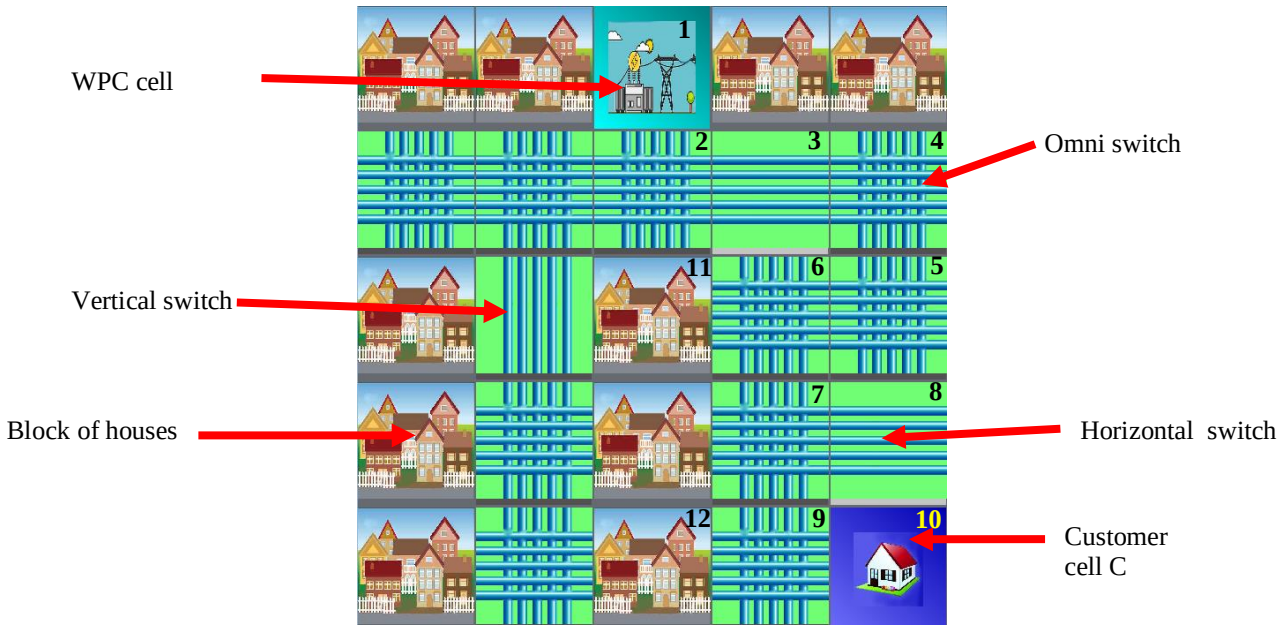
The Western Power Company (WPC) has an electrical grid that allows it to deliver electricity to its customers. The grid consists of a set of electrical switches and cables. A new customer C wants to know whether WPC can provide electricity to its property. For this assignment, you will design and implement a program in Java to determine whether the electrical grid of WPC can deliver power to the new customer C: The program needs to determine whether the set of electrical switches of WPC's grid can be used to form a path from WPC (where the electricity is generated) to C's house.

You are given a map of the city, which is divided into rectangular cells to simplify the task of computing the required path. There are different types of map cells:

- 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 following figure shows an example of a map divided into cells. Each map cell has up to 4 neighboring cells indexed from 0 to 3. Given a cell, the north neighboring cell has index 0 and the remaining neighboring cells are indexed in clockwise order. For example, in the figure the neighboring cells of cell 6 are indexed from 0 to 3 as follows: Neighbor with index 0 is cell 3, neighbor with index 1 is cell 5, neighbor with index 2 is cell 7, and neighbor with index 3 is cell 11. Note that some cells have fewer than 4 neighbors and the indices of these neighbors might not be consecutive numbers; for example, cell 9 in the figure has 3 neighbors indexed 0, 1, and 3.

A path from the WPC cell (cell number 1 in the figure) to C's house (cell number 10) is the following: 1, 2, 3, 4, 5, 6, 7, 9, 10. Note that a path cannot go from cell 3 to cell 6, because the horizontal switch in cell 3 only connects cells 2 and 4. Similarly from cell 5 a path cannot go to cell 8; also from cell 8 it is not possible to go to cell 10. Since cell 11 does not contain a switch, such a cell cannot be part of a path from WPC to C.



1.1 Valid Paths

When looking for a path the program must satisfy the following conditions:

- The path can go from the WPC cell or from an omni switch cell to the following **neighboring** cells:
    - the customer cell
    - an omni switch cell
    - the north cell or the south cell, if such a cell is a vertical switch
    - the east cell or the west cell, if such a cell is a horizontal switch.

- The path can go from a vertical switch cell to the following neighboring cells:
    - The north cell or the south cell, if such a cell is either the customer cell C, an omni switch cell or a vertical switch cell
- The path can go from a horizontal switch cell to the following neighboring cells:
    - The east cell or the west cell, if such a cell is either the customer cell C, am omni switch cell, or a horizontal switch cell.

If while looking for a path the program finds that from the current cell there are several choices as to which adjacent cell to use to continue the path, your program must select the next cell for the path in the following manner:

- the program prefers the customer cell over the other cells;
- if there is no customer cell adjacent to the current cell, then the program must prefer the omni switch cell over the other cells;
- if there is no omni switch cell the program chooses the smallest indexed cell that satisfies the conditions described above.

2. Classes to Implement

A description of the classes that you need to implement in this assignment is given below. You can implement more classes, if you want. You **cannot** use any static instance variables. You **cannot** use java's provided *Stack* class or any of the other java classes from the java library that implements collections. The data structure that you must use for this assignment is an array, as described in Section 2.1.

2.1 ArrayStack.java

This class implements a stack using an array. The header of this class must be this:

public class ArrayStack<T> implements ArrayStackADT<T>

You can download *ArrayStackADT.java* from the course's website. This class will have the following private instance variables:

- *private T[] ArrayStack*. This array will store the data items of the stack.
- *private int top*. This variable stores the position of the last of data item in the stack. In the constructor this variable must be initialized to -1, this means that the stack is empty. Note that this is different from the way in which the variable *top* is used in the lecture notes.

This class needs to provide the following public methods.

- *public ArrayStack()*. Creates an empty stack. The default initial capacity of the array used to store the items of the stack is 20.
- *public ArrayStack(int initialCapacity).* Creates an empty stack using an array of length equal to the value of the parameter.
- *public void push (T dataItem)*. Adds *dataItem* to the top of the stack. If the array storing the data items is full, you will increase its capacity as follows:
    - If the capacity of the array is smaller than 100, then the capacity of the array will be increased by a factor of 2.
    - Otherwise, the capacity of the array will increase by 50. So, if, for example, the size of the array is 100 and the array is full, when a new item is added the size of the array will increase to 150.
- *public T pop*() *throws EmptyStackException*. Removes and returns the data item at the top of the stack. An *EmptyStackException* is thrown if the stack is empty. If after removing a data item from the stack the number of data items remaining is smaller than one third of the length of the array you need to shrink the size of the array by half; to do this create a new array of size equal to half of the size of the original array and copy the data items there.
  For example, if the stack is stored in an array of size 100 and it contains 34 data items, after performing a *pop* operation the stack will contain only 33 data items. Since 33 < 100/3 then the size of the array will be reduced to 50.
  When creating an *EmptyStackException* an appropriate *String* message must be passed as parameter.
- *public T peek*() *throws EmptyStackException*. Returns the data item at the top of the stack **without** removing it. An *EmptyStackException* is thrown if the stack is empty.
- *public boolean isEmpty*(). Returns true if the stack is empty and it returns false otherwise.
- *public int size*(). Returns the number of data items in the stack.
- *public String toString*(). Returns a String representation of the stack of the form:

  "`Stack: elem1, elem2, …`", where `elem`*i* is a String representation of the *i*-th element of the stack. If, for example, the stack is stored in an array called *s*, then elem1 is *s*[0].*toString*(), elem2 is *s*[1].*toString*(), and so.

You can implement other methods in this class, if you want to, but they must be declared as private.

2.2 FindConnection.java

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 FindConnection (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 *FindConnection* using the constructor *FindConnection(args[0])*. When you run the program, you will pass as command line argument the name of the input file (see Section 4). Your *main* method then will try to find a path from the WPC cell to the destination cell C according to the restrictions specified above. The algorithm that looks for a path from the initial cell to the destination **must use a stack and it cannot be recursive**. Suggestions on how to look for this path are given in the next section. The code provided to you will show the path selected by the algorithm as it tries to reach the customer cell C, so you can visually verify if your program works.
- *private MapCell bestCell(MapCell cell).* The parameter is the current cell. This method returns the best cell to continue the path from the current one, as specified in Section 1.1. If several unmarked cells (details about marked and unmarked cells are given in Sections 3 and 5) are adjacent to the current one and can be selected as part of the path (as described in Section 1.1), then this method must return one of them in the following order:
    - the customer cell
    - an omni switch cell (if the conditions of Section 1.1 are satisfied). If there are several possible omni switch cells satisfying the conditions of Section 1.1, then the one with smallest index is returned. Read the description of the class *MapCell* below to learn how to get the index of a neighboring cell
    - a vertical or horizontal switch cell with smallest index that satisfies the conditions stated in Section 1.1.

  If there are no unmarked cells adjacent to the current one that can be used to continue the path, this method returns `null`.

Your program must catch any exceptions that might be thrown. For each exception caught an appropriate message must be printed. The message must explain what caused the exception to be thrown.

You can write more methods in this class, if you want to, but they must be declared as private.

2.3 EmptyStackException

This is the class of exceptions that will be thrown by methods *pop* and *peek* when invoked on an empty stack.

3. Algorithm for Computing a Path

Below is a description of an algorithm for looking for a path from the WPC cell to the destination cell C. Make sure you understand the algorithm before you implement it. You do not need to implement this algorithm. You are encouraged to design your own algorithm, but it must use a stack to keep track of which cells have been processed and it cannot be recursive. You are encouraged to first write a detailed algorithm in pseudocode before implementing it in Java.

- Create an empty stack.
- Get the starting WPC cell using the methods of the supplied class *Map* (description of this class is given below).
- Push the starting cell into the stack and mark the cell as *inStack*. You will use methods of the class *MapCell* to mark a cell.
- ***While*** the stack is not empty *and* the destination has not been found perform the following steps:
    - Peek at the top of the stack to get the current cell.
    - If the current cell is the destination, then the algorithm exits the loop.
    - Otherwise, find the best unmarked neighbouring cell (use method *nextCell* from class *FindConnection* to do this). If this cell exists, push it into the stack and then mark it as i*nStack*; otherwise, since there are no unmarked neighbouring cells that can be added to the path pop the top cell from the stack and mark it as *outOfStack*.

Your program must print a message indicating whether the destination was reached or not. If a path was found the algorithm must also print the number of cells in the path from the initial cell to the destination. Notice that your algorithm does not need to find the shortest path from the starting cell to the destination.

4. Command Line Arguments

Your program **must** read the name of the input map file from the command line. You can run the program with the following command:

```
java FindConnection name_of_map_file
```

where `name_of_map_file` is the name of the file containing the city map. You can use the following code to verify that the program was invoked with the correct number of arguments:

```
public class FindConnection {
        public static void main (String[] args) {
           if(args.length < 1) {
                System.out.println("You must provide the name of the input file");
                System.exit(0);
           }
           String mapFileName = args[0];
                ...
```

To get Eclipse to supply a command line argument to your program open the "Run -> Run Configurations..." menu item. Make it sure that the "Java Application->FindConnection" is the active selection on the left-hand side. Select the "Arguments" tab. Enter the name of the file for the map in the "Program arguments" text box.

5. Classes Provided

You can download from the course's webpage several java classes that allow your program to display the map on the screen. You are encouraged to study the given code to you learn how it works. Below is a description of some of these classes.

5.1 Class *Map*.java

This class represents the map of the city including the location of the WPC cell and the C cell. The methods that you might use from this class are the following:

- *public Map (String inputFile) throws InvalidMapException, FileNotFoundException, IOException.* This method reads the input file and displays the map on the screen. An *InvalidMapException* is thrown if the *inputFile* has the wrong format.
- *public MapCell getStart().* Returns a *MapCell* object representing the cell where the Western Power Company is located.

5.2 Class *MapCell*.java

This class represents the cells of the map. Objects of this class are created inside the class *Map* when its constructor reads the map file. The methods that you might use form this class are the following:

- *public MapCell neighbourCell (int i) throws InvalidNeighbourIndexException*. As explained above, each cell of the map has up to four neighbouring cells, indexed from 0 to 3. This method returns either a *MapCell* object representing the *i*-th neighbor of the current cell or *null* if such a neighbor does not exist. Remember that if a cell has fewer than 4 neighbouring cells, these neighbours do not necessarily need to appear at consecutive index values. So, it might be, for example, that *this.getNeighbour*(2) is null, but *this.getNeighbour*(*i*) for all other values of *i* are not null.
  An *InvalidNeighbourIndexException* exception must be thrown if the value of the parameter *i* is negative or larger than 3.
- *public boolean* methods: *isBlock*(), *isOmniSwitch*(), *isVerticalSwitch*(), *isHorizontalSwitch*(), *isPowerStationl(), isCustomer*(), return true if *this MapCell* object represents a cell corresponding to a block of houses, an omni switch, a vertical switch, a horizontal switch, the initial cell where the WPC is located, or the destination cell C where the customer house is, respectively.
- *public boolean isMarked*() returns true if *this MapCell* object represents a cell that has been marked as *inStack* or *outOfStack.*
- *public void markInStack*() marks *this MapCell* object as *inStack.*
- *public void markOutStack*() marks *this MapCell* object as *outOfStack.*

5.3 Other Classes Provided

ArrayStackADT.java, CellColors.java, CellComponent.java, InvalidNeighbourIndexException.java, CellLayout.java, InvalidMapException.java, IllegalArgumentException.java.

6. Image Files and Sample Input Files Provided

You are given several image files that are used by the provided java code to display the different kinds of map cells 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 folder where the src folder is. ***Do not*** put them inside the src folder as Eclipse will not find them there. If your program does not display the map correctly on your monitor, you might need to move these files to another folder, depending on how your installation of Eclipse has been configured.

7. Submission

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 submit your *.java* files your program cannot be marked. **Do not** submit a compressed file with your java classes (.zip, .rar, .gzip, …). Do not put a "package" command at the top of your java classes.

9. 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 the 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.

10. 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