CS1027: Foundations of Computer Science II
Assignment 2
Due: February 27, 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

For this assignment you will be given a map and you are required to write a program that finds a path from a starting location to a destination, if such a path exists. The starting location is the Middlesex College building, and the destination is your house. The map is divided into rectangular cells to simplify the task of computing the required path. There are different types of map cells:

- a starting map cell, where the Middlesex College building is located,
- a destination map cell where your house is situated,
- map cells containing blocks of houses or parks, where cars cannot drive
- map cells containing roads, where cars can drive. There are several types of road cells:
    - road intersections; up to four different roads can converge into a road intersection: a road coming from the north, a road coming from the south, a road coming from the east, and a road coming from the west. A car coming into an intersection can turn in any direction where there is a road
    - north road, south road, east road, and west road cells; these are roads that can be used to travel north, south, east, or west, respectively. All these roads are one-way roads; the direction in which they can be travelled is indicated in the road cell.

Figure 1 shows an example of a map divided into cells.

1.1 Cell Indexing

Each map cell has up to 4 neighboring cells indexed from 0 to 3. Given a cell, the north neighboring cell has index 0, the east neighboring cell has index 1, the south neighbor has index 2, and the west neighbor has index 3.

For example, consider the cell marked as 6 in Figure 1. The neighboring cells of this cell 6 are indexed from 0 to 3 as follows: The neighbor cell with index 0 is the cell marked as 3, the neighbor cell with index 1 is the cell marked as 7, the neighbor cell with index 2 is the cell marked as 9, and the neighbor cell with index 3 is the cell marked as 5.

Note that some cells have fewer than four neighbors and the indices of these neighbors might not be consecutive numbers. For example, the cell marked as 11 Figure 1 has three neighbors indexed 0, 1, and 3. The cell marked as 12 has only two neighbors indexed 0 and 3.

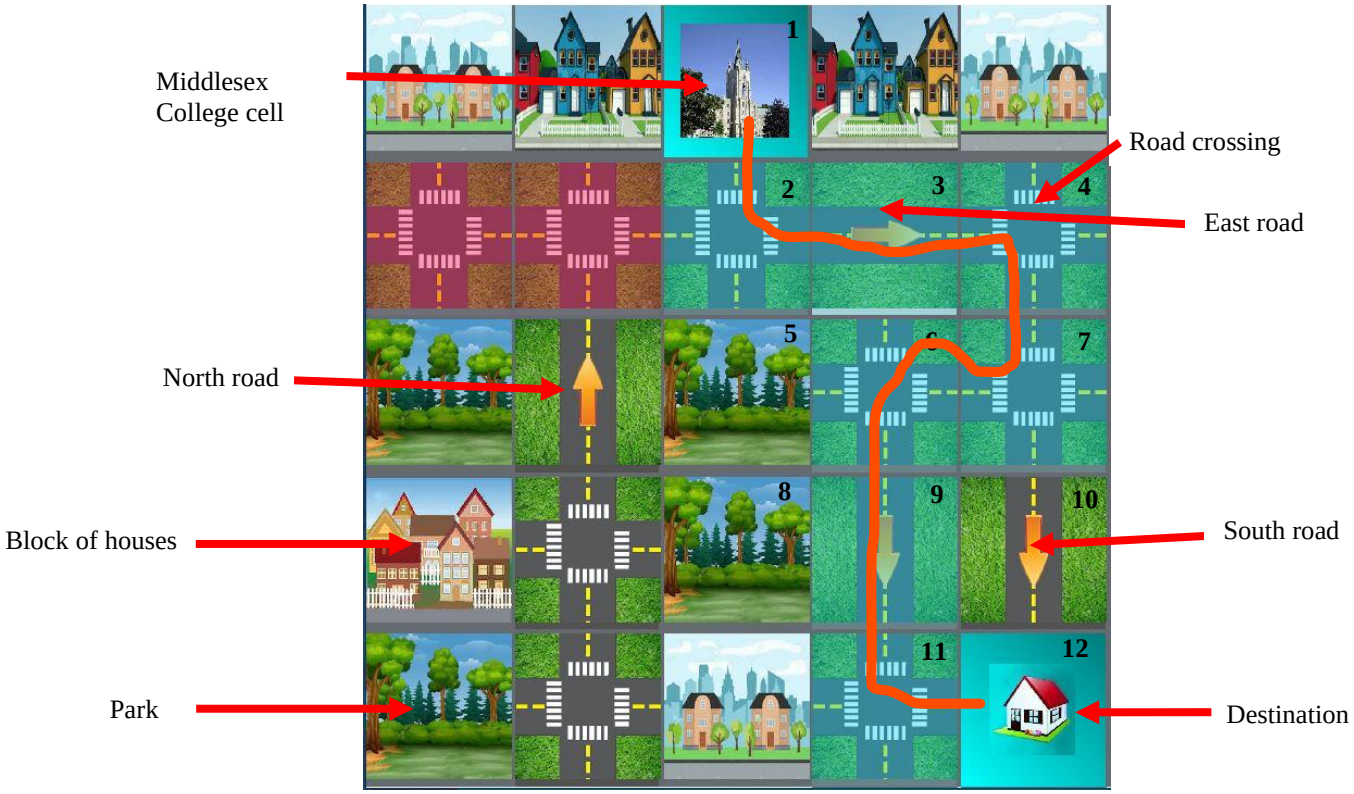

Figure 1.

1.2 Valid Paths

A path from the Middlesex College cell (cell marked as number 1 in Figure 1) to the destination (cell marked as number 12) goes through the following cells: 1, 2, 3, 4, 7, 6, 9, 11, 12. Note that a path cannot go from cell 3 to cell 6 because the east road in cell 3 only connects cells 2 and 4.

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

- For a cell *C*, which is either the starting cell, where the Middlesex College building is located, or a road intersection cell, a valid path can go from *C* to the following **neighboring** cells:
    - the destination cell, where your house is located
    - another road intersection cell
    - a north road cell if such a cell is the neighboring cell of *C* with index 0, i.e. if the north road cell is the neighboring cell located north of *C*
    - a south road cell if such a cell is the neighboring cell located south of *C*
    - an east road cell if it is the neighboring cell located east of *C*
    - a west road cell if it is the neighboring cell located west of *C*.
- A valid path can go from a north road cell *C* to the neighboring cell *C'* located north of *C* if *C'* is either a north road, a road intersection, or the destination cell.
- A valid path can go from a south road cell *C* to the neighboring cell *C'* located south of *C* if *C'* is either a south road, a road intersection, or the destination cell.
- A valid path can go from an east road cell *C* to the neighboring cell *C'* located east of *C* if *C'* is either an east road, a road intersection, or the destination cell.
- A valid path can go from a west road cell *C* to the neighboring cell *C'* located west of *C* if *C'* is either a west road, a road intersection, or the destination cell.

These rules are not as complicated as they seem. Look at Figure 2; the red arrows show which adjacent cells can be part of a path and the red crosses show adjacent cells that cannot belong to a path.

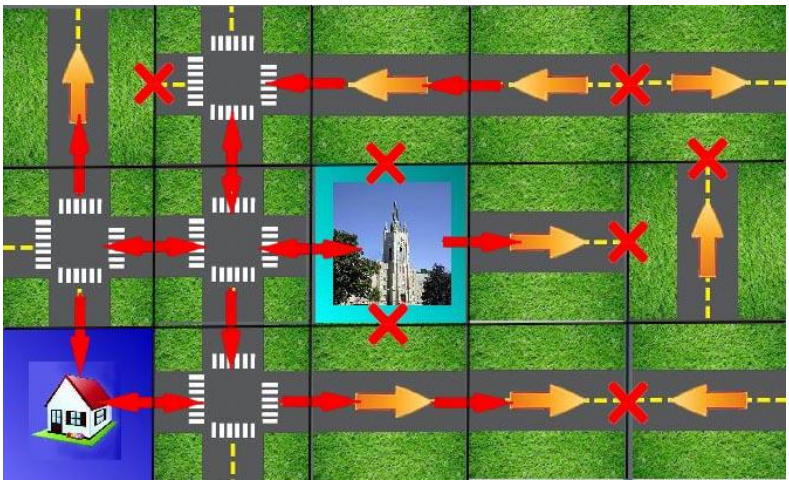

Figure 2.

2. Classes that You Need 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 implement collections. The data structure that you must use for this assignment is an array, as described in the next section.

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 must have the following private instance variables:

- *private T[] stack.* This array will store the data items of the stack.
- *private int top*. This variable stores the **position of the last 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.
- *private int initialCapacity.* This is the initial size of the array *stack.*
- *private int sizeIncrease.* When array *stack* is full and a new item must be added to it, the size of the array will increase by this amount (see description of method *push* below).
- *private int sizeDecrease.* If after removing a data item from array *stack* the number of data items in it is less than one fourth of the size of the array, **and** the size of the array is larger than *initialCapacity*, then the size of the array will decrease by this amount (see description of method *pop* below).

This class needs to provide the following public methods.

- *public ArrayStack(int initialCap, int sizeInc, int sizeDec).* Creates an empty stack using an array of length equal to the value of the first parameter. The values of the parameters will be stored in *initialCapacity*, *sizeIncrease* and *sizeDecrease*, respectively.

- *public void push (T dataItem)*. Adds *dataItem* to the top of the stack and updates the value of *top*. If array *stack* is full, before adding the new *dataItem* you will increase its capacity by the value of *sizeIncrease*. So, if for example, the size of the array is 100, the array is full, and the value of *sizeIncrease* is 50, then the new array will have size 150.
- *public T pop*() *throws EmptyStackException*. Removes and returns the data item at the top of the stack and updates the value of *top*. 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 fourth of the length of the array and the length of the array is larger than *initalCapacity*, you need to reduce the size of the array by *sizeDecrease*; to do this create a new array of size equal to the size of the original array minus *sizeDecrease* and copy the data items there.
  For example, if the stack is stored in an array of size 100 and it contains 25 data items, after performing a *pop* operation the stack will contain only 24 data items. Since 24 < 100/4 then the size of the array will be reduced to 100 - *sizeDecrease*. If, say *sizeDecrease* = 50, then the new array will have size 50.
  If the stack is empty when the *pop()* operation is invoked, an *EmptyStackException* will be created and an appropriate *String* message must be passed as parameter (see description of class *EmptyStackException* below).
- *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 int length().* Returns the length or capacity of the array *stack.*
- *public String toString*(). Returns a String representation of the stack of the form:

  `"Stack: elem1, elem2, …"`

  For example, if the stack contains two data items `"data 1"` and `"data 2"`, and `"data 2"` is at the top of the stack, then method *toString* should return the string

  `"Stack: data 1, data 2".`

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

2.2 EmptyStackException

This is the class of exceptions that will be thrown by methods *pop* and *peek* when invoked on an empty stack. The constructor for this class must be of the form

public EmptyStackException(String message)

When an object of this class is created an appropriate message must be passed as parameter explaining why the exception was thrown.

2.3 Path.java

This class will have an instance variable

Map cityMap;

This variable references the object representing the city map where your program will try to find a path from the starting cell to the destination cell. Class *Map*, described below, is given to you. You must implement the following methods in this class:

- *public Path (Map theMap).* This is the constructor for the class. It receives as input a reference to a *Map* object representing the city map. In this method you must initialize instance variable *cityMap*: *cityMap = theMap.*
- *public void findPath().* This method will look for a path from the starting cell to the destination cell. If a path is found, this method must print a message indicating how many cells there were in the path. If no path was found, this method must print an appropriate message.
  This method **cannot be recursive**, you must use a stack to look for a path from the starting cell to the destination. This method will start at the starting cell, it will look at the neighboring cells and it will select the best cell to go to (to select such a cell method *nextCell* described below must be used); from this cell the best cell will be selected and so on until the destination is reached or the program discovers that there is no path to the destination. A stack must be used to keep track of the cells that the algorithm has visited. Every time a cell is visited, it must be marked so the algorithm does not visit it again (as allowing a cell to be re-visited could cause the algorithm to get trapped in an infinite loop). Read the description of class *MapCell* below to see how to mark cells as visited.
  You are encouraged to design your own algorithm to look for a path from the starting cell to the destination. Read the description of the classes *Map* and *MapCell* below to see how to select the starting cell and the destination cell. If you cannot figure out how to find a path, please read the description of an algorithm given in the next section.
- *private MapCell nextCell(MapCell cell).* The parameter of this method is the current map cell. This method returns the best map cell to continue the path from the current one; read Section 1.2 to see which cells can be used to continue a path from the current cell. If several un-marked cells are adjacent to the current one and can be used to continue a valid path, then this method must return one of them in the following order:
  - the destination cell
  - a road intersection cell. If there are several adjacent road intersection cells satisfying the conditions of Section 1.2, 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 north, east, south, or west road cell with smallest index that satisfies the conditions stated in Section 1.2.

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

3. An Algorithm for Computing a Path

Below is a description of an algorithm for looking for a path from the starting cell to the destination cell C. You can use this algorithm if you want to. If you decide to use this algorithm, we encourage you to first write detailed pseudocode for before you implement it in Java.

- Create an empty stack. You decide on its initial size, *sizeIncrease* and *sizeDecrease*. This stack will store objects of the class *MapCell*.
- Get the starting 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* using method *markInStack*() of the class *MapCell*.

- ***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 *Path* 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* (see class *MapCell* to see how to do this).

As indicated above, 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 starting 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

The *main* method is in class *Map*. The program will read the name of the input map file from the command line. From the console you can run the program as follows:

```
java Map name_of_map_file
```

where `name_of_map_file` is the name of the file containing the city map. To run the program from Eclipse you need to indicate what the name of the input file is. To do this, in Eclipse select "Run → Run Configurations...". Make it sure that "Java Application → Map" 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. For more details check the tutorial posted in the course's website:

http://www.csd.uwo.ca/courses/CS1027b/passingParameters.html

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 starting cell and the destination 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 starting cell where the Middlesex College Building 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 getNeighbour (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 neighbouring cells 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* is thrown if the value of the parameter *i* is negative or larger than 3.

- *public boolean* methods: *isBlock*(), *isIntersection*(), *isNorthRoad*(), *isEastRoad*(), *isSouthRoad(), isWestRoad(), isStart(), isDestination*(), return true if *this MapCell* object represents a cell corresponding to a block of houses or a park (where a car cannot drive), an road intersection, a north road, an east road, a south road, a west road, the staring cell where the Middlesex College Building is located, or the destination cell *C* where your 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