CS1027

ASSIGNMENT 3

Computer Science Fundamentals II

This assignment is due on Friday, July 16, 2021 by 11:55pm. See the bottom for submission details.

Learning Outcomes

To gain experience with

- Implementing a stack with an array data structure
- Using stack methods to solve a problem
- Algorithm design

Introduction

In this assignment, you are working on a program that searches for a path from the houses of CS students to Middlesex College so they can study. There are various map files and each map includes one student's house as the starting point and Middlesex College as the destination.

The students have limited energy (10, by default) so the program must omit paths that are too long. Walking from one cell to the next results in the student losing 1 energy, so they can walk up to 10 cells before they get too weary. However, the students can eat donuts to re-gain 3 energy as they walk so that their journey can be longer before they get tired.

In the current time, we have to be careful to avoid COVID-19 while out in public places. Some of these maps contain one or more Covid cells which are dangerous and must be completely avoided. The students cannot go within 1 space of those cells or else they will risk contracting the virus. This program should omit any paths in which the students steps onto a cell that is orthogonally adjacent to a Covid cell.

You are given several maps that contain various elements as listed below. The map is divided into rectangular cells to simplify the task of computing the required path. There are different types of map cells:

- The starting point (yellow house),
- The destination point (Middlesex College),
- Donut cells, in which the student gains energy (these are treated the same as cross paths in terms of walking access),
- Covid cells, which are dangerous and thus the student should not even walk on a cell adjacent to a Covid cell,
- Map cells indicating walls where the student cannot walk,
- Map cells containing pathways. There are 3 types of pathways:

- Cross paths. A cross path located in cell L can be used to interconnect all the neighbouring map cells of L. A cell L has at most 4 neighbouring cells that we denote as the north, south, east, and west neighbours. The cross path can be used to interconnect all neighbours of L;
- Vertical paths. A vertical path can be used to connect the north and south neighbours of a map cell; and
- Horizontal paths. A horizontal path can be used to connect the east and west neighbours of a map cell.

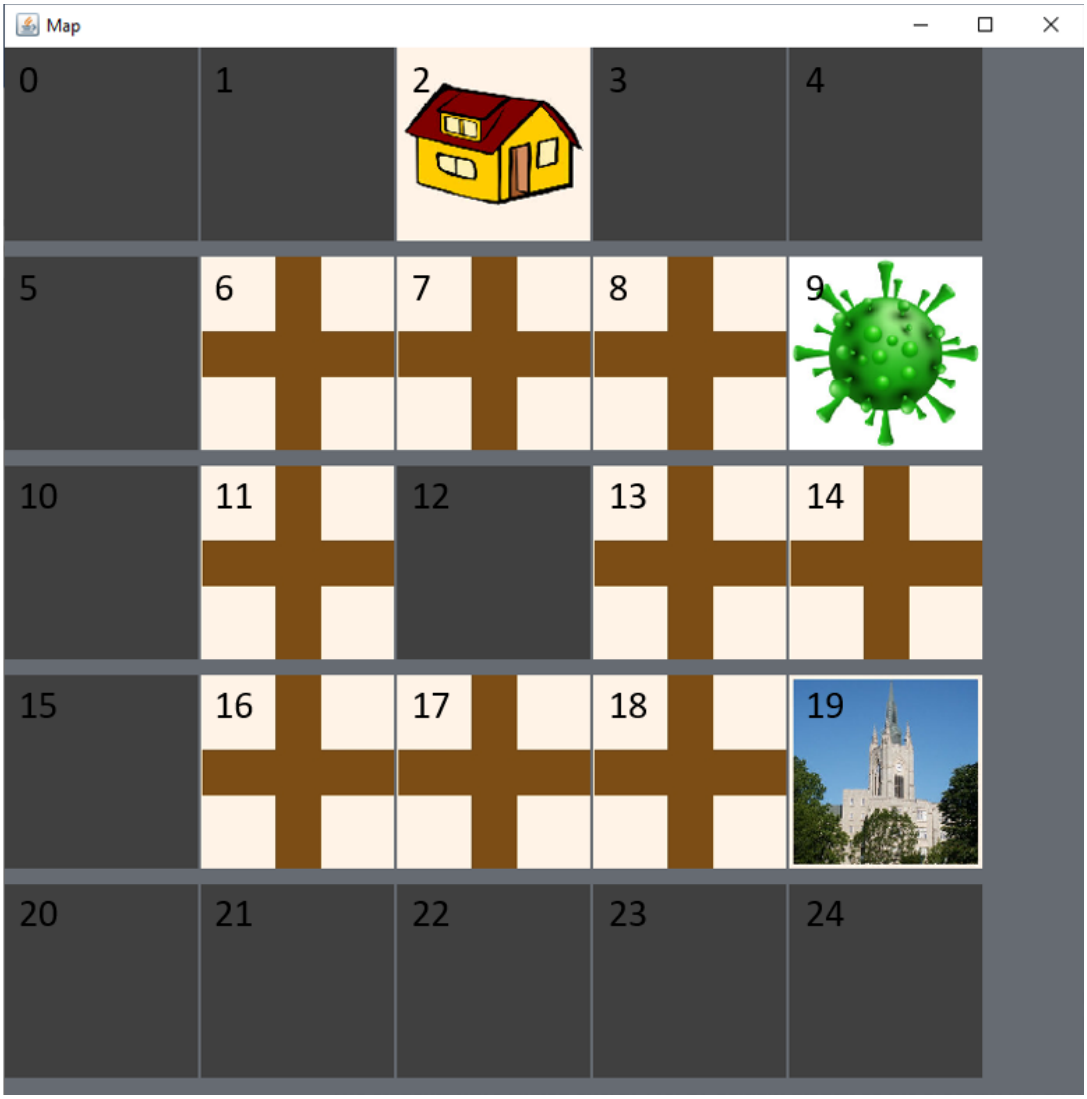


Figure 1. An example of a map that includes the starting point in cell 2, the destination in cell 19, a Covid cell in cell 9, crosspaths in cells 6, 7, 8, 11, 13, 14, 16, 17, and 18, and walls in all the remaining cells.

Each map cell has up to 4 neighbouring cells indexed from 0 to 3. Given a cell, the north neighbouring cell has index 0 and the remaining neighbouring cells are indexed in clockwise order. For example, in Figure 1 the neighbouring cells of cell 8 are indexed from 0 to 3 as follows: neighbour with index 0 is cell 3, neighbour with index 1 is cell 9, neighbour with index 2 is cell 13, and neighbour with index 3 is cell 7.

Some cells (i.e. those on corners or along edges of a map) have fewer than 4 neighbours and the indices of these neighbours might not be consecutive numbers; for example, cell 4 in Figure 1 has two neighbours indexed 2 and 3 (no 0 or 1 neighbours).

A valid path from the starting point (cell 2 in the figure) to the destination (cell 19) is the following: 2, 7, 6, 11, 16, 17, 18, 19. Technically there is also a path: 2, 7, 8, 13, 18, 19; however this path is invalid because it requires the student to step on cell 8 which is adjacent to the Covid in cell 9. Thus the former is the **only** valid path to the destination in this map.

This program has a visual component so that you can watch the path being followed on the map. Most of the cells start with a beige background/border and they change to yellow when pushed on the stack and then to blue when popped off the stack. You may adjust the timing to watch the algorithm run quicker or slower by changing the timeDelay value in the provided Map.java class (do not change anything else in the Map class or other provided classes).

Valid Paths

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

- The path cannot include any cells that are orthogonally adjacent to a Covid cell.
- The path can go from the start point, the destination, a cross path, or a donut cell to the following neighbouring cells:
    - The start point,
    - The destination,
    - A cross path cell,
    - A donut cell,
    - The north cell or the south cell, if such a cell is a vertical path, or
    - The east cell or the west cell, if such a cell is a horizontal path.
- The path can go from a vertical path cell to the following neighbouring cells:
    - The north cell or the south cell, if such a cell is either the start point, the destination, a cross path cell, a donut cell, or a vertical path cell.
- The path can go from a horizontal path cell to the following neighbouring cells:
    - The east cell or the west cell, if such a cell is either the start point, the destination, a cross path cell, a donut cell, or a horizontal path cell.

Provided files

The following is a list of Java files provided to you for this assignment. Please do not alter these files in any way.

- CellColours.java
- CellComponent.java
- CellLayout.java
- IllegalArgumentException.java
- InvalidMapException.java
- InvalidNeighbourIndexException.java
- Map.java
- MapCell.java
- StackADT.java
- StackException.java
- TestStack.java
- TestPath.java

Classes to implement

For this assignment, you must implement 2 Java classes: *ArrayStack* and *StartSearch*. Follow the guidelines for each one below.

In both of these classes, you can implement more private (helper) methods, if you want to, but you may **not** implement more public methods. You may **not** add instance variables other than the ones specified below nor change the variable types or accessibility (i.e. making a variable public when it should be private). Penalties will be applied if you implement additional instance variables or change the variable types or modifiers from what is described here.

ArrayStack.java

This class represents a Stack implemented with an array data structure. It must work for the generic type T and must implement StackADT<T>. This implementation differs from the one shown to you in lecture and the one you used in Lab 5. For this one, the first element will be added to the end of the array and then new items will be added to the left of existing items. For example, if we created an array with 5 slots, the default value of top would be 4 so that the first element would be stored in the cell at index 4. The next item should be added into cell 3, then cell 2, and so on.

The class must have the following *private* variables:

- array (T[ ])

- top (int)

The class must have the following *public* methods:

- public ArrayStack() [constructor]
  - Initialize the array with 10 slots and top = 9
- public ArrayStack(int) [constructor]
  - Initialize the array with N slots and top = N-1, where N is the input parameter
- public void push(T)
  - If the stack is full, add 5 new spaces to the stack and ensure all the elements are stored in the correct slots
  - Add the element in the input parameter to the top of the stack and update top
- public T pop() throws StackException
  - Throw a StackException if the stack is empty
  - Remove and return the element from the top of the stack and update top
- public T peek() throws StackException
  - Throw a StackException if the stack is empty
  - Return (without removing) the element from the top of the stack and update top
- public boolean isEmpty()
  - Return true if the stack contains 0 elements; otherwise return false
- public int size()
  - Return the number of elements on the stack
- public int getLength() [not part of StackADT.java but still required]
  - Return the number of slots in the array
- public int getTop() [not part of StackADT.java but still required]
  - Return the top index
- public String toString()
  - If the stack is empty, return "The stack is empty."
  - Otherwise, build a string that starts with "Stack: " and then contains each of the stack's elements in order from the top (most recent) to the bottom (earliest) with a comma and space (,) between each of the elements. The last one should end with a period (.) instead.

StartSearch.java

This class is the main heart of the program in which we are writing the algorithm that performs a search on a given map file. This program must use an ArrayStack to keep track of the MapCells that have been visited or attempted to be visited. See the Algorithm Pseudocode section for more information on how this algorithm should be implemented.

The class must have the following *private* variable:

- map (Map)

The class must have the following *public* methods:

- public StartSearch(String) [constructor]
  - Initialize the map object with the given String variable
  - Wrap the above line in try-catch statements to satisfy the compiler
- public MapCell bestCell(MapCell)
  - The parameter is the current cell
  - This method must determine the best (unmarked) cell from the current cell to continue the path, following the rules described near the top of this document.
  - Include try-catch statements that handle an InvalidNeighbourIndexException that could be thrown from the getNeighbour() method.
  - Return null if none of the neighbours are valid options, or if the current cell is orthogonally adjacent to a Covid cell (regardless of what is on the cell)
  - If several unmarked cells are adjacent to the current one and can be selected as part of the path, then this method must return the preferred one based on the following order (from most preferred to least preferred):
    - Destination cell
    - Donut cell
    - Cross path cell
    - Horizontal or vertical cell
  - For the cell priorities listed above, if there are multiple cells of the same top priority, the one with the lowest neighbour index should be chosen first
  - For example, if a cell is surrounded by (0) cross path, (1) donut cell, (2) horizontal cell, and (3) cross path; the best cell would be the donut cell. Now, if that donut cell is already marked in or out of the stack, then it's not a valid option so the next best option is the cross path at index 0 (it is preferred over the cross path at index 3 because the index is lower).
- public String findPath()
  - Run the algorithm that searches for a valid path from the start point to the destination while following all the rules and restrictions. See the Algorithm Pseudocode section for more information on this method.
  - Use the bestCell() method described above to help with this algorithm
  - Return the string of actions that contains the entire sequence of visited cells separated with hyphens and suffixed with the energy level (see below for more information about the expected format of this string).
- public static void main (String[])
  - This is the method that starts the program. See the Command Line Arguments section below for info on setting up the String[] args parameter for this method.
  - If one argument is given, use it to initialize a StartSearch object with that map name. Call the findPath method on the StartSearch object.

Algorithm Pseudocode

Below is a description for an algorithm that will look for a path from the starting point (yellow house) to the destination (Middlesex College). It will be helpful to understand the algorithm deeply before attempting to implement it. Implement the algorithm as described to make sure your code passes the test cases.

You must use a stack to keep track of which cells are in the path, and it cannot be recursive. Writing your algorithm in pseudocode will make coding it in Java easier and is helpful to show to the TAs and the instructors if you need help.

- Create and initialize a stack.
- Create an actionString variable.
- Create a status flag to indicate whether or not the destination has been found.
- Get the start cell using the methods of the supplied class Map.
- Push the starting cell into the stack and mark the cell as inStack (use methods of the class MapCell to mark a cell)
- Create a variable for the energy level and set it to 10 at the start.
- While the stack is not empty and the destination has not been reached, perform the following steps:
    - Peek at the top of the stack to get the current cell.
    - Find the next unmarked neighbouring cell (use method nextCell from class StartSearch to do this).
    - Update the actionString to contain the cell being visited.
    - If such a next cell exists and the energy level is above 0:
        - Check if the next cell is the destination. If so, set the status flag to true.
        - Check if the next cell is a donut cell. If so, increase the energy level by 3.
        - Push the neighbouring cell into the stack and mark it as inStack.
        - Decrease the energy level by 1 because of this single movement.
    - Otherwise, since there are no unmarked neighbouring cells that can be added to the path, perform the following steps:
        - Pop the top cell from the stack and mark it as out of stack.
        - If that top cell is a donut cell, decrease the energy level by 3 (because we are undoing a donut consumption).
        - If that top cell is anything other than the start point, increase the energy level by 1 (because we are undoing a single movement).
- While the stack is not empty, perform the following:
    - Pop the top cell from the stack and mark it as out of stack.

Your program must return the actionString that contains a sequence of all the cells that were visited, even if they ended up being backtracked. Include a hyphen between each pair of cell IDs and then append the energy level at the end of the string in the format "E#" where # is the energy level after finishing the path finding algorithm. For example, a final sequence (for a hypothetical map) could be: "1-6-11-12-17-22-21-22-23-24-E2".

Command Line Arguments

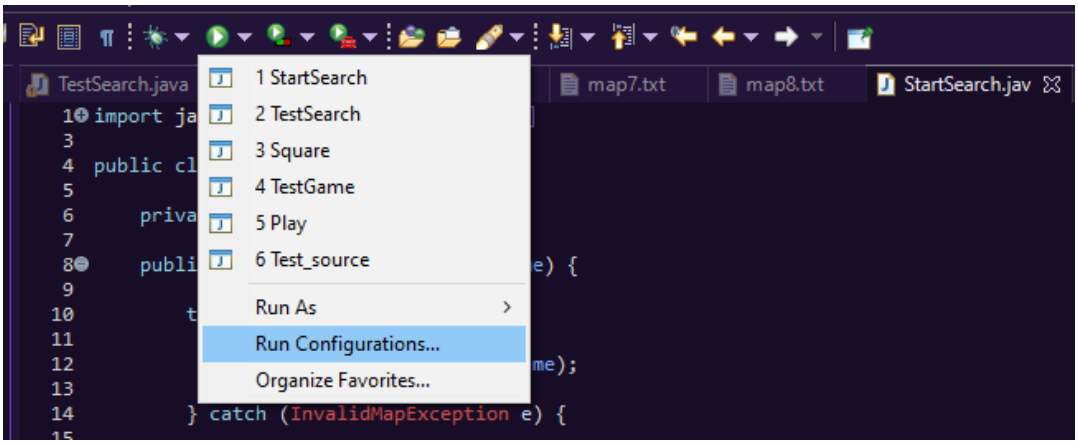
Your StartSearch program must read the name of the input map file from the command line. You can run the program with the following command:

java StartSearch nameOfMapFile

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

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

You can access run configurations in Eclipse as follows:



In the text box for "Program arguments" you can type your argument in, i.e. map3.txt

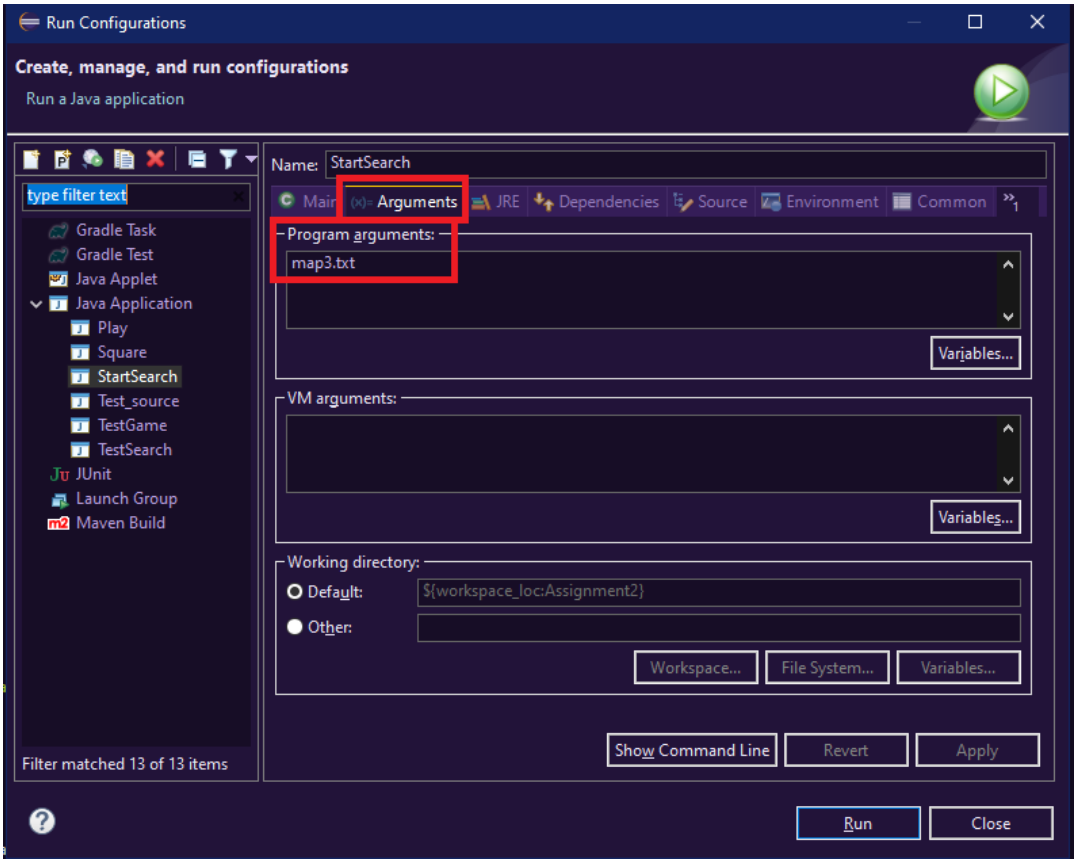


Image Files and Sample Input Files Provided

- You are given several image files that are used by the provided Java code to display the map 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 root folder (not inside bin or src)
- 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 or the project was created.

Marking notes

Marking categories

- Functional specifications
  - Does the program behave according to specifications?

    - Does it produce the correct output and pass all tests?
    - Are the class implemented properly?
    - Are you using appropriate data structures (if applicable)?
    - Does the code run properly on Gradescope (even if it runs on Eclipse, it is up to you to ensure it works on Gradescope to get the test marks)
- Non-functional specifications
    - Are there comments throughout the code (Javadocs or other comments)?
    - Are the variables and methods given appropriate, meaningful names?
    - Is the code clean and readable with proper indenting and white-space?
    - Is the code consistent regarding formatting and naming conventions?
- Penalties
    - Lateness: 10% per day
    - Submission error (i.e. missing files, too many files, etc.): 5%
    - "package" line at the top of a file: 5%
    - Compile or run-time error on Gradescope
    - Failure to follow instructions, (i.e. changing variable types, etc.)

Remember you must do all the work on your own. Do not copy or even look at the work of another student. All submitted code will be run through cheating-detection software.

Submission (due Friday, July 16, 2021 at 11:55pm ET)

Assignments must be submitted to Gradescope, not on OWL. If you are new to this platform, see these instructions on submitting on Gradescope.

Rules

- Please only submit the files specified below. Do not attach other files even if they were part of the assignment.
- Do not upload the .class files! Penalties will be applied for this.
- Submit the assignment on time. Late submissions will receive a penalty of 10% per day.
- Forgetting to submit is not a valid excuse for submitting late.
- Submissions must be done through Gradescope and the test marks come directly from there. If your code runs on Eclipse but not on Gradescope, you will NOT get the marks! Make sure it works on Gradescope to get these marks.
- Assignment files are NOT to be emailed to the instructor(s) or TA(s). They will not be marked if sent by email.
- You may re-submit code if your previous submission was not complete or correct, however, re-submissions after the regular assignment deadline will receive a penalty.

ASSIGNMENT 3

Files to submit

- ArrayStack.java
- StartSearch.java

Grading Criteria

- Total Marks: [20]
- Functional Specifications:
    - [3] ArrayStack.java
    - [3] StartSearch.java
    - [10] Passing Tests
- Non-Functional Specifications:
    - [1.5] Meaningful variable names, private instance variables
    - [0.5] Code readability and indentation
    - [2] Code comments