CS 1027B: Computer Science Fundamentals II

Online Section (650)

Summer 2023

Assignment 3 – National Park Bike Paths

Due Mon Jul 24 at 23:55pm ET

Learning Outcomes

In this assignment, you will gain hands-on experience with:

- implementing doubly linked lists,
- implementing an extended stack abstract data type (ADT) using doubly linked lists, and
- finding a path in a map using a stack.

Introduction

Figure 1: Pyramid Falls in British Columbia, Canada

The management of a National Park wants to improve the accessibility for visitors by building bike paths by creating a new National Park around the current Pyramid Creek Falls Provincial Park in British Columbia (which is a beautiful area, if you ever get the chance to go see it). The park consists of a vast area with various terrains, including forests, hills, and lakes. The goal is to create a network of bike paths that connect different natural treasures within the park, such as scenic viewpoints, picnic areas, and historical sites. To achieve this, you need to develop a program that finds the optimal paths for the bike paths while considering the park's topography and safety.

Let's model the inside of the park space as interconnected hexagonal chambers. Many of the chambers have dangerous wildlife or natural hazards in them, so it is very important that the bike paths only go through through chambers where enough light enters so the park visitors can bike through them safely. There are several types of chambers in the park:

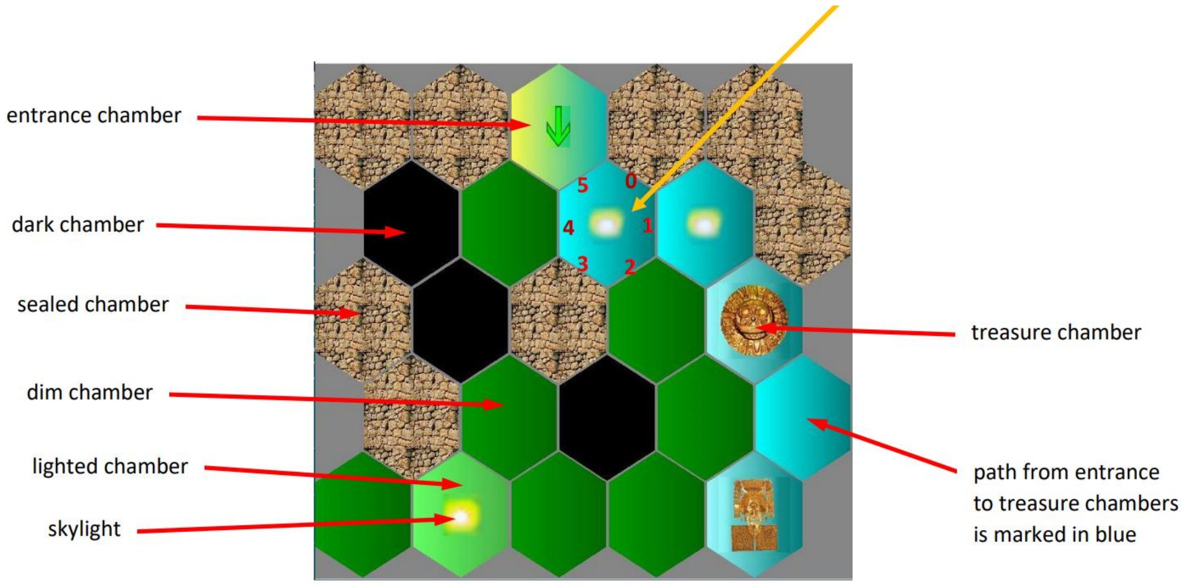
- **Sealed chambers**: chambers where the cyclists cannot enter.
- **Lighted chambers**: chambers that have a skylight, a hole in the ceiling where sunlight enters. The cyclists can safely bike through these chambers as there is enough light in them to spot the natural hazards.
- **Dim chambers**: these are chambers that are adjacent to a lighted chamber. Enough light enters these chambers to make it safe to bike through them.
- **Dark chambers**: these are chambers where there is no light, so the cyclysts should not enter them as they would not be safe. A dark chamber is one that has no adjacent lighted chambers.
- **Entrance chamber**: this is the chamber through which the cyclist can enter the park. The entrance chamber is lighted.
- **Treasure chambers**: these are chambers that hold invaluable natural relics that will leave the bikers or hikers with lifetimes of memories. Treasure chambers are lighted.

For this assignment you need to write a program that finds a path from the entrance chamber to all the treasure chambers that does not include any sealed or dark chambers. As the path is being computed, **the following constraints must be satisfied**:

- if the program has a choice between moving into a treasure chamber or a chamber without a treasure your program **must** prefer the treasure chamber
- if the program has a choice between moving to a lighted chamber or moving to a dim chamber the program **must** prefer the lighted chamber.

The following figure shows an example of the possible chambers of the park. The entrance chamber has an arrow inside of it, lighted chambers have a yellow skylight in the middle, dim chambers do not have a skylight, dark chambers are black, sealed chambers are filled with rocks, and treasure chambers have a treasure in them. The path from the entrance to all the treasure chambers is marked in blue. Note that the blue path in the figure includes two lighted chambers (near the entrance) and one dim chamber (between the two treasure chambers; you recognize the dim chamber because it does not have a skylight).

Each chamber has up to 6 neighbors indexed from 0 to 5. The neighbor with index 0 is in the upper right side of a chamber. The remaining neighboring chambers are increasingly indexed in clockwise direction. The figure shows the indices for the neighbors of a chamber adjacent to the entrance.



Classes to implement

For this assignment, you must implement two Java classes: *DLStack.java* and *PathFinder.java*. Follow the guidelines for each one below.

In these two 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 from what is described here.

DLStack.java

This class represents an extended stack ADT implemented using a doubly linked list. An example of a stack is shown in the figure below; note the position of the top of the stack. The nodes of the doubly linked list are of the provided class *DoubleLinkedNode*; this is the same class you used in Lab 4. You are also provided with a java interface *DLStackADT.java* file that specifies the operations of this ADT. Hence, your implementation will declare class *DLStack* as follows:

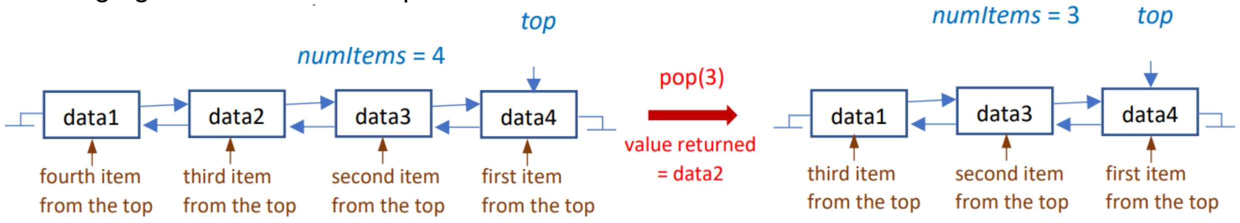
```
public class DLStack implements DLStackADT
```

The class must have the following **private** instance variables:

- `DoubleLinkedNode top`: This a reference to the node at the top of the stack.
- `int numItems`: The value of this variable is the number of data items stored in the stack.

The class must have the following **public** methods:

- `public DLStack()`: Creates an empty stack, so top must be set to null and *numItems* to zero.
- `public void push(T dataItem)`: Adds the given *dataItem* to the top of the stack.
- `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.
- `public T pop(int k) throws InvalidItemException`: Removes and returns the *k*-th data item from the top of the stack. So if *k = 1* the method must remove the data item at the top of the stack, if *k = 2* the method must remove the second item from the top of the stack, and so on. An *InvalidItemException* is thrown if the value of *k* is larger than the number of data items stored in the stack or if *k* is less than or equal to zero. The following figure illustrates this operation.



- `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 DoubleLinkedNode getTop()`: Returns top.
- `pubic toString()`: Returns a string of the form *"[data1 data2 … datan]"*, where *$data_1$* is the data item at the top of the stack, and *$data_n$* is the data item at the bottom of the stack.

PathFinder.java

This class contains the code needed to compute a path from the entrance of the park to all the treasure chambers. This class must have the following **private** instance variable:

- `Map pyramidMap`: This is a reference to an object of the provided class Map that represents the chambers of the Pyramid Falls National Park.

This class must have the following **public** methods:

- `public PathFinder(String fileName)`: This is the constructor for the class. It receives as its input the name of a file which contains a description of the chambers of the park. In the constructor you need to create an object of the type Map (see description of this class below) and pass the name of the input file to the constructor of class Map; you need to store the created Map object in instance variable *pyramidMap*. Creating an object of the class Map will display the chambers of the park on the screen. Some sample inputs are provided for you; please read them if you are interested to know the format of the input files. The names of these files are of the form map#.txt.

- `public DLStack path()`: This method finds a path from the entrance to all the treasure chambers that can be reached by satisfying the constraints specified in the introduction (read method *bestChamber* below that explains the constraints in more detail). The chambers along the computed path must be stored in a stack of the class *DLStack*. If there is no path from the entrance to any of the treasure chambers, then the method must return an empty stack. Some suggestions as to how to compute this path are given in the next section. Your program will show the chambers selected by the algorithm as it tries to reach all the treasure chambers, so you can visually check how the program works (read description of the provided class Chamber below).
- `public Map getMap()`: Returns the value of pyramidMap.
- `public boolean isDim(Chamber currentChamber)`: Returns true if *currentChamber* is dim and returns false otherwise. *currentChamber* is dim if: it is not null, it is not sealed, it is not lighted, and one of its neighboring chambers is lighted. Read the description of class *Chamber* below to learn how to check whether a chamber is sealed or lighted.
- `public Chamber bestChamber(Chamber currentChamber)`: Selects the best chamber to move to from *currentChamber* according to these restrictions specified in the introduction:
    - from *currentChamber* your program will prefer to move to an adjacent unmarked treasure chamber, if any: so, in this case the method will return the neighboring unmarked treasure chamber with smallest index. Read the description of class Chamber to learn how to check whether a chamber is unmarked or contains a treasure
    - otherwise, from *currentChamber* your program will prefer to move to an unmarked lighted chamber, if any: so, in this case the method will return the neighboring unmarked lighted chamber with smallest index
    - otherwise, from *currentChamber* your program will move to an unmarked, dim chamber: in this case the method will return the neighboring unmarked dim chamber with the smallest index
    - if there is no unmarked treasure, lighted or dim chamber, the method must return the value null.

Algorithm for Exploring the Park

Here is a description in pseudocode of an algorithm for trying to find a bike path from the entrance to the treasure chambers. Make sure you understand the algorithm before you implement it. You do not have to use this algorithm if you do not want to. You are strongly encouraged to design your own algorithm, but your algorithm must use a stack of the class *DLStack* to keep track of the chambers that have been visited.

- Create an empty stack.
- Get the starting chamber and number *N* of treasure chambers from class *Map*, described below.
- Push the starting chamber into the stack and mark the chamber as pushed. Read below the description of the provided class *Chamber* to learn how to mark a chamber.
- Now, <u>while</u> the stack is not empty perform the following steps:
    - Peek at the top of the stack to get the current chamber.
    - If the current chamber is a treasure chamber and the number of treasure chambers that have been already found is equal to *N*, then exit the while loop.
    - Find the best neighboring chamber *c* to move to using method *bestChamber* from class *PathFinder*. If *c* is not null, push it into the stack and then mark it as pushed; otherwise pop the top chamber from the stack and mark it as popped.
- After the while loop terminates return the stack

Notice that your algorithm does not need to find the shortest path from the starting chamber to the exit. You **CANNOT** use class Stack from the Java libraries.

Provided Files

You are given several java classes that allow your program to display the chambers on the screen. You are encouraged to study the given code, so you learn how it works. Below is a description of some of these classes that you will need to use in your code. Other java classes, sample input files, and image files are also provided.

<u>Map.java</u>

This class represents the chambers of the park. The methods that you will use from this class are the following:

- `public Map (String inputFile) throws InvalidMapCharacterException, FileNotFoundException, IOException`*:* This method reads the input file and displays the park chambers on the screen. An *InvalidMapCharacterException* is thrown if the *inputFile* contains an invalid character. Look at the sample input files to learn which characters are allowed.

- `public Chamber getEntrance()`: Returns a *Chamber* object representing the entrance to the park.

- `public int getNumTreasures()`: Returns the number of chambers that contain treasures and that your program must reach.

Chamber.java

This class represents the park chambers. Objects of this class are created inside class *Map* when the input file is read. The methods that you might use from this class are the following:

- `public Chamber getNeighbour (int i) throws InvalidNeighbourIndexException`: Each park chamber has up to six neighbouring chambers, indexed from 0 to 5. For each value of *i*, from 0 to 5, the method might return either a *Chamber* object representing a chamber or *null*. Note that if a chamber has fewer than 6 neighboring chambers, these neighbors 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* is thrown if the value of the parameter *i* is negative or larger than 5.
- public `boolean` methods: `isSealed()`, `isLighted()`, `isTreasure()` return true if **this** *Chamber* object represents a chamber that is sealed, lighted, or it contains treasure, respectively.
- `public boolean isMarked()`: Returns true if **this** *Chamber* object represents a chamber that has been marked as pushed or popped.
- `public void markPushed()`: Marks **this** *Chamber* object as pushed.
- `public void markPopped()`: Marks **this** *Chamber* object as popped.

Sample Input Files and Running the Program

1. You are given several input files (map1.txt – map5.txt) that you can use to test your code. The **main method is in the provided class Pyramid.java.**

- To run the program from a terminal, put all provided files and your java classes in the same directory:

  From that directory compile the code by typing

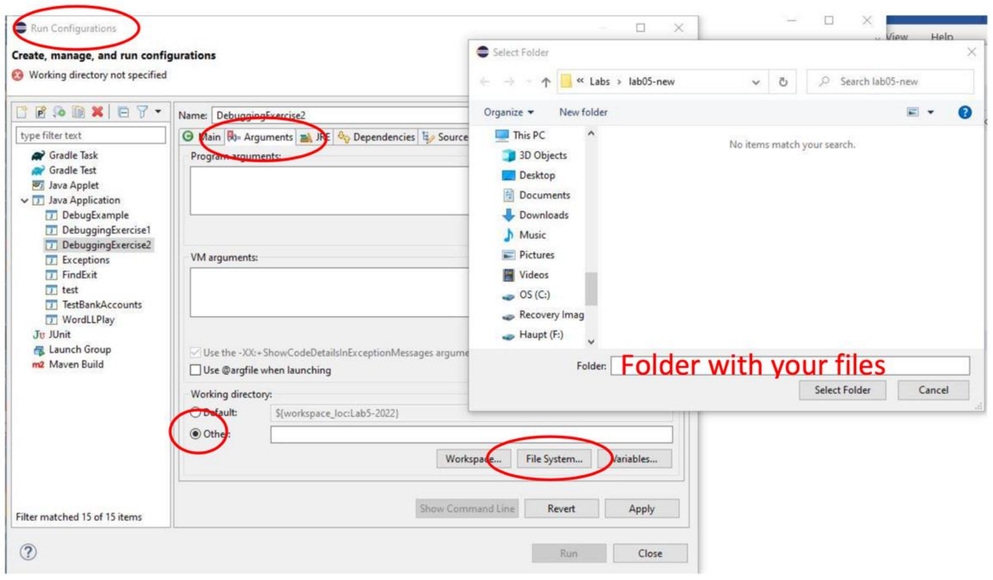
  - `javac *.java`

  Then, run the program by typing

  - `java Pyramid inputFile`

  where *inputFile* is one of the provided sample input files: map1.txt, map2.txt, ...

- To run the program from Eclipse, first put all the provided files and your java classes in the same directory. Then in Eclipse select Run→Run Configurations; in the window that opens select Arguments and under “Working directory” select “Other”. Click on “File System” and choose the folder where you have placed all your files.



After having configured Eclipse, to run the program, select class Pyramid.java in the Package Explorer, then select Run→Arguments and in the Program Arguments box enter the name of the input file (map1.txt, map2.txt, ...). Then click Run.

If you use another IDE you will have to read its documentation to figure out where to put the provided files and your java code. If you cannot figure that out, please run the program from a terminal.

2. You can use class TestStackMap.java to test your implementation of the extended stack and some of the methods of class PathFinder.java.

Marking Notes:

Functional Specifications

- Does the program behave according to specifications?
- Does it produce the correct output and pass all tests?
- Are the classes implemented properly?
- Will 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). Note that Gradescope will not provide you any results of the autograder when you are submitting, except a general confirmation that the file was accepted.
- You are expected to perform additional testing (create your own test harness class to do this) to ensure that your code works for other scenarios as well. We are providing you with some tests but we may use additional tests that you haven't seen before for marking.
- Does the program produces compilation or run-time errors on Gradescope?
- Does the program fail to follow the instructions (i.e. changing variable types, etc.)

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?
- Submission errors (i.e. missing files, too many files, etc.) will receive a minimum penalty of 5%
- Including a “package” line at the top of a file will receive a minimum penalty of 5%
- **150 Word Summary**
  - Students are required to provide a 150-word explanation within the JavaDoc of their ***PathFinder*** class, elaborating their approach towards designing the code of the entire program, any challenges faced and the steps taken for testing the solution. The explanation should be precise (5), logical (5), and demonstrate clear understanding of the code functionality (10).

Remember you must do all the work on your own. Do not copy or even look at the work of another student, and please develop your code yourself rather than relying on automated tools. All submitted code will be run through similarity-detection software.
Assignments must be submitted to Gradescope, not on OWL.

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. Submissions CANNOT be more than 2-days late.
- Forgetting to submit is not a valid excuse for submitting late.
- Submissions must be done through Gradescope. 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.
- You are expected to perform additional testing (create your own test harness class to do this) to ensure that your code works in other scenarios.
- 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.
- We are providing you with some tests, but we will use additional tests that you haven't seen before for marking.

Files to Submit

  - *DLStack.java*
  - *PathFinder.java*

Grading Criteria

Total Marks: [100%]

- Functional Specifications:
  - [50%] Passing Auto-Graded Tests
- Non-Functional Specifications:
  - [10%] Meaningful variable names, private instance variables
  - [10%] Code readability and indentation
  - [10%] Code comments and JavaDoc
  - [20%] 150 Word Summary of assignment logic in PathFinder.java

Good luck, and thanks for your help making scenery like this more accessible!