Assignment 3

DUE DATE: FRIDAY JUNE 26, 2020 AT 11:59 PM

Notes:

- Submit ONE .java file per question. Each .java file must contain ALL of the source code for a question. See the Programming Standards document for how to store several classes in the same file.
- Each filename must have the format `<your last name><your first name>A3Q2.java` (e.g. `SmithJohnA3Q2.java`). Please use your name as shown in UM Learn.
- Do not submit any output. Your code will be run during marking.
- Your program must compile, run, and end normally (not crash or get stuck in an infinite loop) to receive any marks. See the Assignment Information file for some tips on how to make sure your code runs for the markers.
- Assignments must follow the Programming Standards posted in UM Learn.
- Assignment submissions are only accepted via UM Learn. Submissions by email will not be accepted.
- You may submit your assignment multiple times, but only the most recent version will be marked.
- Assignments become late immediately after the posted due date and time. Late assignments will be accepted up to 49 hours after that time, at a penalty of 2% per hour or portion thereof. After 49 hours, an assignment is worth 0 marks and will no longer be accepted.
- The time of the last submission controls the late penalty for the entire assignment.
- These assignments are your chance to learn the material for the tests. *Code your assignments independently*. We use software to compare all submitted assignments to each other, and pursue academic dishonestly vigorously.

Question 1: Find a Path Through a Maze [25 marks]

This question can be done after Week 6 of the course.

In this question you will read a maze from a file, and attempt to find a path through the maze twice. One attempt will use a stack to store the positions visited in the maze, and the other attempt will use a queue to store the positions visited in the maze. (Later in the term we will talk about depth-first and breadth-first searches. That is essentially what you will be doing here.)

Maze Representation

Each position in the maze will be one of the following:

- a path (an open space you can walk on), represented by **.** (period)
- a wall (a blocked space that you can not step on), represented by **#** (hash mark)
- a starting point, represented by **S**

- a finish line, represented by **F**

If a path has been found and the maze is displayed, the path from start to finish should be marked with ***** (asterisk).

The input file will contain one maze. **You can assume that the input file does not contain any errors.** The first line in the file will contain the number of rows, followed by a space, followed by the number of columns. The rest of the file will contain the text representation of the maze, using the symbols listed above (`.` and # and S and F). Please see the sample input below.

Approach to Path Finding

You will implement two searches, one using a stack and one using a queue. For each search algorithm, you will maintain the data structure (stack/queue) containing the Positions left to explore, while updating the Maze object. Think carefully about keeping track of which positions in the maze have been visited, and where they have been visited from.

The following pseudocode describes both searches:

```
Add start position to data structure
Mark start position as visited
while (data structure is not empty):
   current = remove position from data structure
   if (current is the finish):
      exit search
   for (each neighbour of current that is an unvisited path):
      Mark neighbour as visited
      Record neighbour as visited from current
      Add neighbour to the data structure
```

When this algorithm is complete, `current` will be the finish position if the search found a path, but will be some other position if the search failed. If the search succeeded, you can reconstruct the path by checking from which position the finish was visited, and then checking from which position that position was visited, and so on, until you reach the start.

Your solution must include the following classes. **It is strongly suggested that you write one class at a time, and test each thoroughly before moving on to the next class.** You will have a very difficult time debugging your stack and queue classes if you try to do so while solving a maze.

Position Class

The **Position** class represents one location in the maze (i.e. a particular row and column). It stores information about that position, including whether it is a path or a wall, and whether it is the start/origin or finish/destination. The fields in one Position object should be:

- The row number.
- The column number.
- A type that identifies the type of square (start, finish, path, wall). Something similar to `enum SquareType{START, FINISH, PATH, WALL;}` is appropriate. (See the section below on enumerated types.)
- A boolean to indicate whether this position has been visited (so that a search does not get stuck in a cycle).
- A reference to the previous position on the path (will be null if this position is not on the path).

The Position class should have a **constructor** that accepts a row, column, and type for a location. The constructor should set the other fields to default values.

You will also need methods that return String representations of the position, to use when displaying the maze or listing the path through the maze. One method should return the appropriate symbol (e.g. "#" for a wall), and the other should return the coordinates in the format `(2, 3)`.

Stack & Queue Classes

You will need a Stack that stores Positions, and a Queue that stores Positions. Either the stack or queue should be implemented using **an array implementation**, and the other should be implemented using **a linked list implementation**. The Stack class should have the standard push, pop, top, and isEmpty methods. The Queue class should have the standard enqueue, dequeue, front, and isEmpty methods. You may also find toString/print methods handy for debugging (if you write them, leave them in the code that you submit).

The constructors for the Stack and Queue should accept a size. Estimate for the number of Positions that might be stored by looking at the size of the maze.

Maze Class

The Maze will consist of a two-dimensional array of Positions (or, more specifically, references to Positions).

The **Maze constructor** should read the input file, create a 2D array of the appropriate size, and then fill the array.

The search methods should be in this class. One method will use a stack to track the progress through the maze, and the other will use a queue. They should be otherwise similar.

You will find that you need a number of short methods in this class. For example, where should the search start? Finish? You will need methods that return the start/finish Positions for the maze.

Because you are going to try two different searches for a path through the maze, write a `resetMaze()` method, that will reset all Positions in the maze to unvisited.

Application (Main) Class

Your application class will create a Maze (reading from a file, where the filename is typed by the user). It will attempt to solve the maze using a stack, and print the result (the maze and the list of positions along the path from start to finish, or a message that no path could be found). The maze will be reset, and then the application class will attempt to solve the maze using a queue, and print that result.

Sample Input 1

```
4 7
#######
#...#S#
#F#...#
#######
```

Sample Output 1

```
Please enter the input file name (.txt only):
sample1.txt
Processing sample1.txt...
The initial maze is:
#######
#...#S#
#F#...#
#######
The path found using a stack is:
#######
#***#S#
#F#***#
#######
```

```
Path from start to finish: (1, 5) (2, 5) (2, 4) (2, 3) (1, 3) (1, 2)
(1, 1) (2, 1)
The path found using a queue is:
#######
#***#S#
#F#***#
#######
Path from start to finish: (1, 5) (2, 5) (2, 4) (2, 3) (1, 3) (1, 2)
(1, 1) (2, 1)
Processing terminated normally.
```

Sample Input 2

```
10 12
############
#.#........#
#.#.######.#
#.#....#...#
#.###.F#.#.#
#...####.#.#
#.#.#..#.#.#
#.#.#.##.#.#
#S#......#.#
############
```

Enumerated Types

An enumerated type is basically a type that says any variable of that type is only allowed to have one of a predefined set of values. They are useful when you want to limit the possible values for something.

You define the type outside of any classes so that it applies globally.

If you haven't seen enumerated types before, here are a couple of links.

https://www.w3schools.in/java-tutorial/enumeration/
https://docs.oracle.com/javase/tutorial/java/javaOO/enum.html

Once you have defined the type, you can use it in the maze to test the type of the current square. For example, if your enumerated type is

```
enum SquareType{
    START,
    FINISH,
    PATH,
    WALL;
}
```

and a Position has a field

```
public SquareType typeOfSquare;
```

then you would set the type using statements such as

```
typeOfSquare = SquareType.PATH;
```

and can then test the type with statements such as

```
if (current.typeOfSquare == SquareType.PATH){
   ...do processing for a path...
}
else if (current.typeOfSquare == SquareType.WALL){
   ...do processing for a wall...
}
```

(where current is a reference to a Position object).

Enumerated types in Java can have their own methods. You may write a toString method for this Type if you choose, but it is not required.

Question 2: Dictionary Implementations [25 marks]

This question requires material from Week 7.

In this question you will create three implementations of a dictionary, and will compare the time to fill and search the dictionaries. The first dictionary implementation will use an ordered array to store the contents of the dictionary. The second and third implementations will use a hash table to store the contents of the dictionary. The second dictionary will use open addressing, with double hashing to resolve collisions. The third will use separate chaining. Details specific to each table are listed in separate sections below.

The Application (Main) Class

- The application class is provided for you. The only change you should make is to rename this class to include your name.
- Look at how the dictionaries are created, filled, and searched in this class. For each dictionary, you must implement methods with names and parameters such that they work with this class.
- `GreatExpectations.txt` (from `http://www.gutenberg.org/ebooks/1400`) will be used to build the dictionaries, with the goal of each dictionary storing all words that appear in this file.
    - Place a copy of this file in your working directory.
    - The application class builds the dictionaries by processing this file one line at a time, and building a dictionary as it goes. Each line is split into tokens, and each token is added to the dictionary. **The dictionary insert methods should prevent duplicates – see below.** Punctuation is stripped and words are converted to lower case. Anything left is treated as a word. Due to the nature of the input file, not everything added to the dictionary will be a real word.
- Notice that initially an empty dictionary of each type is created **by passing an initial size of 100 to each constructor**. Hash table sizes should be prime numbers and the constructors should adjust the initial size appropriately – see the sections below for details.
- The output will include the size of each dictionary. These should all be equal because the input file is processed in an identical manner for each dictionary.
- `A3Q2TestWords.txt` contains a list of words that will be searched for in the dictionaries.
    - Place a copy of this file in your working directory.
    - The output includes the number of words found. That number should be the same for each dictionary.

The Dictionary Classes

Your dictionary classes **must** be named `DictionaryOrdered`, `DictionaryOpen`, and `DictionaryChain`. **Each dictionary class must have the following public methods.** Use additional (private) helper methods as appropriate.

- A constructor that accepts an integer indicating the initial size of the dictionary. That is, `public DictionaryOrdered(int size)` for the dictionary using an ordered array, `public DictionaryOpen(int size)` for the dictionary using open addressing, and `public DictionaryChain(int size)` for the dictionary using separate chaining. See the instructions for each dictionary below re: initial size.
- `public int getSize()`: Return the number of words in the dictionary.
- `public void insert(String newWord)`: Insert the given word into the dictionary. If a word already exists in the dictionary **do not** add it. All entries in the dictionary should be unique.
- `public boolean search(String wordToFind)`: Return true if the word is in the dictionary, false otherwise.

Keep the implementation details hidden from the user of the dictionary classes. The main class accesses the dictionary contents ONLY via the above public methods.

It is common when working in a team on a large software projects to specify the public methods that a class will have, and the task that each method will accomplish. It is then possible for different programmers to work on different classes and the classes to work together as expected when all programmers have finished. You must follow this procedure and implement the dictionary classes and methods as specified. Helper methods should be private, and unreachable from your main class.

Details Specific to the Dictionary using an Ordered Array

- The constructor for the dictionary class accepts an initial size from the user. This should be the size of the initial array.
- When inserting words, if the dictionary is full, double the size of the array.
- The search method should use a (**non**-recursive) binary search to search the ordered array. Use a non-recursive search to avoid timing the overhead associated with recursive calls.

Details Specific to the Dictionary using Open Addressing

- The constructor for the dictionary class accepts an initial size from the user. However, the hash table size should always be a prime number. Find the first prime number larger than the requested array size, and use that prime number as the size of your hash table.
    - To find the prime number, start with the requested array size, test whether that number is prime, if not increment by 1 and test again, until you find a number that is prime.
    - To test if a given number (n) is prime:

```
starting with j=2, and as long as j*j <= n
      if n % j == 0, n is not prime
```

- Use Horner's method (with a = 27) as your primary hash function.
    - To convert characters to integers, since we are dealing with lowercase letters, cast a char to an int and subtract 96 (ASCII for a is 97, meaning that a would end up as 1, b as 2, c as 3, etc.).
    - If there happen to be any non-alphabetical characters left in the "words", ignore them. That is, when computing the hash value, only include characters with ASCII values from 97 (a) to 122 (z).
- You will resolve collisions using double hashing. For the secondary hash function, use `stepSize = constant – (sum_of_characters % constant)`. As above, subtract 96 to map a to z into the range 1 to 26, and include only letters in the sum of characters. The constant should be prime and smaller than the array size. In this assignment, use `constant = 41`.
- When inserting words, if the hash table is more than **60%** full, enlarge the array and rehash all the words currently in the dictionary, transferring the contents to the larger array. To determine the size of the new array, find the first prime number larger than double the current array size.

Details Specific to the Dictionary using Separate Chaining

- You will need a linked list class and will store a linked list at each location in your hash array. Your linked lists do not need to be ordered.
- Similar to the open addressing implementation, find the first prime number larger than the requested array size, and use that prime number as the size of the hash array.
- Use Horner's method (as above) as your hash function, where the sum of characters again uses the method of subtracting 96 from the char value to obtain a number between 1 and 26 for each letter.
- Enlarge the hash table when the load factor exceeds **2**. That is, enlarge the hash array when the number of words stored in the table is more than twice the size of the hash array. (Recall that the hash array is the array of chains/linked lists.)

Additional Notes

- Create your own test methods to test your dictionary classes as you build them. It is suggested that you also create your own small test input. Only try running with the full input file once you are confident that the dictionary classes are in good shape.
- Notice how much faster it is to fill a hash table than an ordered array.
- Place the provided application class, the 3 dictionary classes, and any supporting classes (e.g. linked list) in a single .java file for submission.

[Programming Standards are worth 8 marks]