Lara Croft : Treasure Hunter

Overview

A treasure hunter named Lara Croft is developing a GPS-based software system that will help her crawl through narrow underground tunnels in search of treasure buried by an ancient civilization known as the Rightangulars. This civilization only ever walked in one of the four cardinal directions: north, south, east or west. She has a lot of money from previous expeditions and has hired you to write a test system where she enters a map into the software and the software gives her a path to follow to extract all of the treasure and finally exit the underground complex.

Algorithm

Lara Croft is smart. She knows how to compute paths through tunnels. For example, the following algorithm will find a path from a start location to a grid cell that is of type "whatToFind" (either a treasure or an exit presumably). This algorithm assumes that a Location remembers the direction from which it was entered and the Location from which it was entered.

```
algorithm List<Direction> path(Map t, Location start, LinearStructure C, whatToFind)
C.add( start )
List<Direction> result is a new list
while(C is not empty)
   Location current = C.remove();
   if(current is whatToFind ) then
         return the path taken to get to the current
   else if(current is traversable and current has not been visited)
        mark current as visited
        C.add(Location to the east of current);
        C.add(Location to the south of current);
        C.add(Location to the west of current);
        C.add(Location to the north of current);

return null // no path from start to a cell labeled as whatToFind
```

To find a path from the start to all treasures and, finally some exit, the following algorithm should be used.

```
algorithm List<Direction> path(Map t, Location start, LinearStructure C)
  List<Direction> path = new list
  int found = 0 and boolean exited = false;
  Location beginning = new Location( start.row, start.col );
  while( found < treasures.size() && !exited )
    beginning = new Location( beginning.row, beginning.col, null, null );
    whatToFind is TREASURE if more treasure to find, otherwise EXIT
    Location end = path( beginning, collection, whatToFind );
    if( end == null ) return new ArrayList<Direction>();
    else addToPath( end, path );
    if whatToFind was TREASURE then
      remove found treasure from map and increment found
    otherwise
      exited = true

return path
```

Data Structures and Design

You must write a number of classes that will collaborate to solve this problem. The first two classes are named LinearStack (following the LIFO principle) and LinearQueue (following the FIFO principle), both of which implement the LinearStructure interface which is given below. Both classes should use an array-based implementation.

```
public interface LinearStructure<E> {
  public int size();
  public boolean add(E e);
  public E remove() throws NoSuchElementException;
  public boolean isEmpty();
  public E peek() throws NoSuchElementException;
  public void clear();
  public Iterator<E> iterator();
}
```

Note that you must not use any of Java's built-in data structures within the LinearStack or LinearQueue classes. Also note that the iterator for each class must return the elements of the LinearStructure in the order that they would be naturally removed. Of course, iterating over the elements does not remove any element.

You must also complete a Map class that represents a map. A Map is a two-dimensional grid of elements. A Location object is used to ask questions about the map such as “is this a start location’ or ‘is this an exit location’. The Map class is informally described below.

```
public class Map {
  public Map(File f) throws IOException {…}
  public int getWidth() {…}
  public int getHeight() {…}
  public boolean isExit(Location pt) {…}
  public boolean isStart(Location pt) {…}
  public boolean isTreasure(Location pt) {…}
  public boolean isWall(Location pt) {…}
  public List<Location> getTreasures() {…}
  public List<Location> getExits() {…}
  public Location getStart() {…}
  public List<Direction> path(LinearStructure c);
}
```

Map and Direction Details

A map is a rectangular grid. Each cell of the grid is marked as WALL, TUNNEL, TREASURE, START or EXIT. Lara can enter any grid that is not a WALL. Although she is very clever, she cannot, of course, pass through a WALL. Any cell that is off-of-the-grid is also considered to be a WALL. Lara will only walk straight north or south or east or west on this expedition in honor of the Rightangulars. A path is then given as a list of directions such that each direction is NORTH, SOUTH, EAST or WEST.

Map File

The map will be given as a text file. The first line contains two integer numbers: the height H of the map followed by the width W. Each line that follows describes one row of the underground complex. There are exactly W characters on each of these lines. The only characters in these rows are

- '#' : WALL
- '!' : TREASURE

- '_' : TUNNEL
- '$" : START
- '^' : EXIT

There will be exactly one START. There will be at least one EXIT and at least one TREASURE.

Program

Write a program named Treasure. The program will accept the name of a map-file and either the word STACK or QUEUE as input. The program will then read the map file, construct a path, and print the result. If the word STACK is given as the 2nd argument, the linear structure of the path-finding algorithm is a stack. If the word QUEUE is given as the 2nd argument, the linear structure of the path-finding algorithm is a queue.

If there is no path that allows Lara to obtain all treasures and exit, print NO PATH. Otherwise, print a comma-separated list of DIRECTIONS. Provide reasonable error messages if any error occurs. Your program must never crash.