Due: Sep. 27 (09/27), 13:00

Overview

This assignment consists of two parts, implementing a circular linked list and implementing a basic polygon class.

General Notes

- *Read this homework guideline carefully.* If you do not follow the guidelines, you may receive a 0 regardless of whether your code works or not.
- Do not use any IDEs (Eclipse, IntelliJ IDEA, etc.)
  - We recommend Sublime Text (Linux/Mac/Windows), Notepad++ (Windows), or TextWrangler (Mac).
  - IDEs often create a "package" of your code, which breaks the autograder.
  - **If you know how to fix the package problem**, you can use any IDE you want. However, we will not answer any questions related to this problem since we have already recommended a solution.
- Do not change any method or class signatures. If your code changes any class or method names or signatures, you will receive an automatic 0.
- Make sure your code compiles. Non-compiling code will automatically receive a 0. If your code does not compile, it may be better to just comment out the incorrect code and return a dummy value (something like null or -1) so the rest can compile.
- To ensure that your code will be accepted by the autograder, you should submit your code on LearnUS, download it again, recompile it and check the provided test suite. This way, you know the file you are submitting is correct.

1 Linked List

There are many variations on the basic linked list that are used frequently in practice. Two common ones are doubly linked lists (where each node has a pointer to a *next* and *previous* node) and circularly linked lists (where the *tail* of the linked list points to the *head*).

You will implement a circular, doubly linked list. This implementation will only have a *head* pointer. Again, you will implement several common operations related to circular linked lists. The following are the methods you have to implement in `CircularLinkedList.java`.

1. `public CircularLinkedList()`: the constructor method.
2. `public int size()`: returns the number of elements in the linked list.
3. `public boolean isEmpty()`: returns `True` if there are no elements in the linked list. Otherwise returns `False`.
4. `public T getHead()`: returns the element stored in the node pointed by the *head*.
5. `public void rotate(Direction direction)`: rotates the linked list so that the *head* points to the node in the given direction. `Direction` is an `Enum` with two constants, `TO_NEXT` and `TO_PREVIOUS`.
6. `public void insert(T element)`: inserts `element` before the *head*. That is, the new node becomes the *head*'s *previous* node.
7. `public T delete()`: deletes the element pointed by the *head* and returns it. If the list is empty, then return `null`. If the list is nonempty after deletion, the new *head* should be the *next* node of the original *head*.
8. `public boolean find(T target)`: finds the first matching element and let *head* point that element. If no match exists, the element pointed by the *head* should stay the same. Returns `True` if the search succeeded, otherwise returns `False`. You can check whether two items are equal using the `.equals()` method.

2 Polygons

One common use of circularly linked lists is representing the boundary of a 2-dimensional polygon. Since the boundary of a polygon can be represented by a circular, ordered list of vertices, we can simply use our previously implemented circular linked list with a custom `Point` class (which we have implemented for you) to handle the creation and manipulation of the polygon.

You will implement a basic polygon built on top of a circular linked list. You do not have to implement many of the methods required by the circular linked list sections (since you already did it in the circularly linked list implementation), but you will have to implement a special algorithm, *point-in-polygon*.

2.1 Point in Polygon

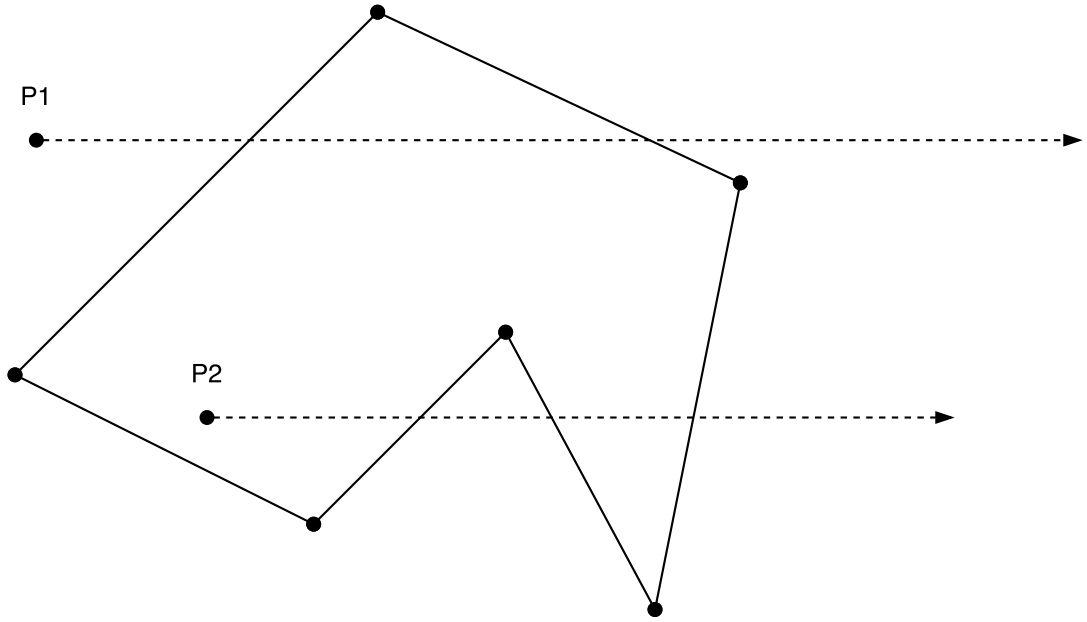
Given a polygon P, and a point p, one important question is to determine whether or not p lies inside P. A popular algorithm for this problem is known as the *Ray-Casting* algorithm. The algorithm is as follows:

Data: Polygon P; Point p
Result: Whether or not p is inside P
r = a ray starting at p extending to ∞
$count = 0$
for *edge* $e \in P$ **do**
  if r *intersects* e **then**
    $count = count + 1$
  end
end
if *count is odd* **then**
  return `True`
else
  return `False`
end

Algorithm 1: Ray-Casting Algorithm

Here, an edge is defined as two sequential points in the polygon boundary. Below you can see a demonstration of the algorithm. The ray extending from point $P1$, which is outside the polygon, crosses two edges while the ray

extending from $P2$, which is inside, crosses 3.



While in practice there are some edge cases that you should consider for this algorithm (if the point to be tested is a point on the boundary of the polygon, if the polygon has 3 points but they are all colinear, etc), we will not be testing them in this assignment. All tested polygons will have non-zero area (if they have at least 3 points) and will not self-intersect.

For this algorithm any ray will work in general, but in this assignment you should use a ray of slope 0 (meaning it extends to infinity towards the right of the point). If you use this slope, it will guarantee that none of the tests have a case where the ray intersects the polygon at a point where two edges meet. If you use another slope, you will have to consider this edge case, which can be difficult.

One final note is that if the polygon contains fewer than 3 points, you should always return `False` for this algorithm.

General Directions

- Write your name and student ID number in the comment at the top of the files in src/main directory.
- Implement all of the required methods.
- You should not import anything that is not already included in the file.
- Pay careful attention to the required return types and edge cases.
- All the codes we provided can be found in src/base directory. If you are unsure what a class/method exactly does, please refer to the code.

Submission Procedure

Submit the files in the src/main directory, excluding Main.java, as a zip file. You *must* make a zip file for submission using the Gradle build tool (refer to Compiling section).

For this assignment, you should submit only the following three files:

- CircularLinkedList.java
- Polygon.java
- your_student_id_number.txt

You must rename 2023xxxxxx.txt to your actual student ID number. Inside that text file, you must include the following text. Please be sure to write all the following text including the last period.

In completing this assignment, I pledge that I have not given nor received any unauthorized assistance.

If this file is missing, you will get a 0 on the assignment. It should be named *exactly* your student id, with no other text. For example, *2023123456.txt* is correct while something like *2023123456_pa1.txt* will receive 0.

Compiling

You can test your Java code using the following command:

```
% ./gradlew -q runTestRunner
```

You can also make a zip file for submission using the following command. The zip file named with your student id (the name of the .txt file) will lie in the "build" directory:

```
% ./gradlew -q zipSubmission
```

We provide an empty Main class for testing using standard input/output:

```
% ./gradlew -q --console=plain runMain
```

Since this file (src/main/Main.java) is not for submission, you can use any package in the file.

On Windows, use `gradlew.bat` instead of `./gradlew`.

Testing

We have provided a small test suite (src/test) for you to check your code. You can test your code by compiling and running the tester program.

Note that the test suite we will use to grade your code will be much more rigorous than the one provided here (and not necessarily a superset of the provided tests). You should consider making your own test cases to check your code more thoroughly.