Assignment 2

ECSE 250 Fall 2024

posted: Oct. 24, 2024
due: Nov. 10th, 2024

Learning Objectives

In this assignment, you will get some experience working with linked lists and stacks. You will implement a data structure for a fun application. Starting with this assignment we will start to focus also on the efficiency of your algorithms. You will learn to look at code with a more critical eye, without only focusing on the correctness of your methods.

General Instructions

For this assignment need to download the files provided. Your task is to complete and submit the following files:

`Caterpillar.java`

- **Do not change any of the starter code that is given to you. Add code only where instructed, namely in the "ADD YOUR CODE HERE" block.** You may add `private` helper methods to the class you have to submit, but you are not allowed to modify any other class.
- Please make sure that the file you submit is part of a package called `assignment2`.
- You are NOT allowed to use any class other than those that have already been imported for you. **Any failure to comply with these rules will give you an automatic 0.**
- Do NOT start testing your code only after you are done writing the entire assignment. It will be extremely hard to debug your program otherwise. If you need help debugging, feel free to reach out to the teaching staff. When doing so, make sure to mention what is the bug you are trying to fix, what have you tried to do to fix it, and where have you isolated the error to be.
- Read through the description of all of the methods before starting to code. This will give you an idea of the general assignment and might help you decide on how to better organize your code (e.g. what could be a good `private` helper method to add?)

Submission Instructions

- Submissions will be accepted up to 3 days late. Note that submitting one minute late is the same as submitting 3 days late. We will deduct points for any student who has to resubmit after the due date (i.e. late) irrespective of the reason, be it the wrong file submitted, the wrong file format submitted or any other reason. We will not accept any submission after the 3-day grace period. This policy will hold regardless of whether or not the student can provide proof that the assignment was indeed "done" on time.
- Your first late penalty is waved **automatically** at the end of the semester. There is no need to make an extra request.
- Don't worry if you realize that you made a mistake after you submitted: you can submit multiple times but **only the latest submission will be evaluated**. We encourage you to submit a first version a few days before the deadline (computer crashes do happen and Ed Lessons may be overloaded during rush hours).
- **Do not submit any other files, especially .class files and the tester files. Any failure to comply with these rules will give you an automatic 0.**
- Whenever you submit your files to Ed, **you will see the results**. Your assignment is not submitted correctly if you do not see the results. **If your assignment is not submitted correctly, you will get an automatic 0. If your submission does not compile on ED, you will get an automatic 0.**
- The assignment shall be graded automatically on ED. **Requests to evaluate the assignment manually shall not be entertained, and it might result in your final marks being lower than the results from the auto-tests.** Please make sure that you follow the instructions closely or your code may fail to pass the automatic tests.
- Next week, a mini-tester will also be posted. We encourage you to modify and expand it. ***You are welcome to share your tester code with other students on Ed.*** Try to identify tricky cases. Do **not** hand in your tester code.
- **Failure to comply with any of these rules will be penalized**. If anything is unclear, it is up to you to clarify it by asking either directly a TA during office hours or on Ed.
- In addition to the required methods, you are free to add as many other **private** methods and classes as you want (granted that all of the classes and methods listed below have been made). Unless otherwise specified, no other non-private fields or methods are allowed.

The Very Hungry Caterpillar

Once upon a time, in a lush colorful meadow nestled between the rolling hills of MunchGill Valley, there lived a peculiar caterpillar named Gus. Now, Gus was no ordinary caterpillar. He had an insatiable appetite and an adventurous spirit that was as boundless as its stomach was bottomless.

Figure 1: Gus. An AI generated image from Canva's Magic Media tool, created from the prompt 'Gus, a colorful caterpillar with a big appetite and an adventurous spirit'.

Gus's favorite pastime was munching on delectable delicacies, and he had a particular fondness for trying new foods. One fine day, as he slithered along the meadow, he stumbled upon a magical patch of food that seemed to have an endless variety of treats - from succulent strawberries to Swiss cheese, and even shimmering stardust-infused snacks!

This enchanted patch of food, however, came with a twist. Each time Gus indulged in a new treat, he magically transformed - he'd grow longer, shrink down, or hilariously flip himself upside down. It didn't matter though; Gus loved the adventure of it all and kept munching away.

But Gus wasn't alone. Little did he know, the wise Professor from the MunchGill University of Magic had been watching its antics with great interest. She noticed how Gus's body would change with every bite, just like a data structure growing with each new element. Intrigued, she gave her brightest students a challenge: create a game that mirrors Gus's magical transformations using the power of singly linked lists.

The students, eager to prove their skills, were thrilled by the task. They knew that, like Gus, they'd need to navigate some tricky twists and turns. They would need to use their knowledge of linked lists to represent Gus's ever-changing body, ensuring that each segment of its wiggly form reflected its latest delicious bite.

With wands (or rather, keyboards) in hand, the students set out to turn Gus's gluttonous journey into a whimsical and hilarious game. As they coded, they knew they weren't just crafting a program—they were bringing Gus's unpredictable world to life, one bite at a time. What's more, they began to realize that this journey, much like Gus's, would teach them lessons beyond just data structures—lessons of efficiency, logic, and, of course, how to have fun with their code.

And so, the fate of Gus's gluttonous escapades was left to these budding computer scientists. Armed with their knowledge of singly linked lists, they embarked on their coding adventure, determined to craft a game worthy of Gus's boundless appetite.

Starter code

Completed class

You will need to understand how to use the following classes, but you should not modify them.

- `assignment2` package
  - A `EvolutionStage.java`, representing 4 different stages of the caterpillar with java enumeration. An enumeration is a special data type that represents a fixed set of constants. All you need to know for this assignment is that you can compare enumeration values using `==` and `!=`. For instance, `EvolutionStage stage == EvolutionStage.FEEDING_STAGE` is a valid boolean expression.
  - A `GameColors.java`, representing 5 different colours that the caterpillar can have.
  - A `Position.java`, representing the position, defined by the x and y coordinates, in a 2-dimensional world.
  - A `MyStack.java` implements a stack with Java generic types.
- `assignment2.food` package – This package contains all the classes representing the possible food items the caterpillar might encounter while slithering along the meadow. You can look inside these files to see what can be useful to you, but you should not modify them!

The Caterpillar class

You need to modify the `Caterpillar.java` in order to complete this assignment. A caterpillar will be represented as a singly linked list of `Segment`s.

A nested class `Segment` is in `Caterpillar.java`, A `Segment` stores the following information:

- A `Position`, representing where this segment is (see `Position.java`). Throughout this pdf, we will represent such objects using the ordered pair `(x, y)`.
- A `Color`, representing the color of this specific segment. The colors of the segments will mostly depend on what the caterpillar has eaten in order to have gained those segments.
- A `Segment`, representing the next segment in the body of the caterpillar. If this is the last body segment, then this field will be `null`.

Each caterpillar is defined by the following fields:

- `Segment head` : a reference to the head of the caterpillar
- `Segment tail` : a reference to the last segment in the caterpillar's body
- `int length` : the number of segments in the caterpillar's body
- `EvolutionStage stage` : the stage in which the caterpillar finds itself. `EvolutionStage` is an enumeration.
- `MyStack<Position> positionsPreviouslyOccupied` : all the positions previously occupied by the caterpillar with the most recent one on the top of the stack (this will become more

clear once you read the description of the methods below). We will represent the stack as a list of elements between square bracket: the element at the top of the stack will be the right most one, while the element at the bottom of the stack will be the left most.

- `int goal` : the number of segments the caterpillar aspires to in order to be able to become a wonderful butterfly.
- `int turnsNeededToDigest` : the number indicating how many turns are left before the caterpillar is ready to take its next bite (see the method definitions below for more information).
- `static Random randNumGenerator` : To use it, you must follow the following guidelines:
  - Do **NOT** assign a new reference to this variable. In fact, you should never in your code create a new `Random` object.
  - If asked to generate a random number you must use `randNumGenerator`.
  - You should generate random numbers only when it is strictly necessary.

  If in your code you fail to respect the guidelines above, your methods will probably not behave how we expect them to, leading to some tests possibly failing.

Each caterpillar is defined by the following methods:

- `toString()`: it has been implemented to make debugging a little easier for you. The method returns a string representing a caterpillar. The string contains the positions of each of the segments. The right most pair of coordinates represents the position of the head, while the left most represents the position of the tail. The color of the segment is used to color the text. So, for example, if we have a caterpillar with a red head in position `(3,1)`, a second yellow segment in position `(2,1)`, and its last blue segment in position `(2,2)`, then `toString()` will return the following string:

  (2,2) (2,1) (3,1)

- `move()`: the method that you need to implement in order to move the caterpillar.
- `eat()`: this method has been overloaded for various food. You will need to provide the implementation. The methods are listed in the order of difficulty, but you don't need to implement them in that order.

Note that, all the fields of the `Caterpillar` class as well as the nested class `Segment` have been left `public` just to make testing easier on our side. As discussed in class, in general, you would want to keep as much as you can private and instead provide `public` getters/setters when needed.

Step 1: Growing the Basics

In this section, you'll start by laying down the groundwork for your caterpillar, Gus. The tasks here will focus on implementing the foundational methods that allow Gus to grow, move, and interact with its surroundings. These basic operations are essential for understanding how linked lists function in Java, and they will provide you with the building blocks to handle more complex tasks later on. Take your time understanding how to implement this part carefully!

- `Caterpillar(Position p, Color c, int goal)` : creates a caterpillar with one single segment of the given color, in the given position. The input `goal` is used to set the number of segments the caterpillar needs to acquire before becoming a butterfly. Whenever a caterpillar is first created, it finds itself in a feeding stage, ready to devour every delicacy within its reach.

  This method runs in $O(1)$.

- `getSegmentColor(Position p)` : returns the color of the segment in the given position. If the caterpillar, does not have a segment placed upon the given position, then the method returns `null`.

  This method runs in $O(n)$, where n is the number of segments.

- `eat(Fruit f)` : this foundational food group is the basis of growth for our wiggly friend. With each fruit bite, the caterpillar grows longer by getting a new segment added matching the color of the fruit ingested. The new segment should be added at the tail of the caterpillar, and its position should be the most recent position previously occupied by the caterpillar. Make sure to update all relevant fields to represent this growth.

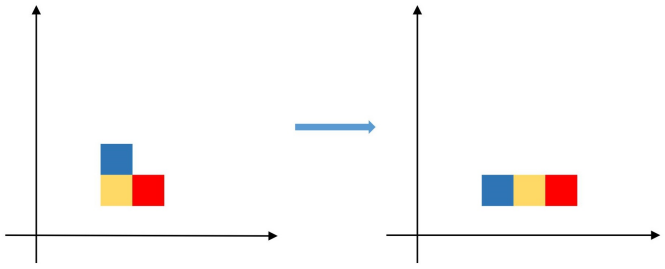
  This method runs in $O(1)$.

- `move(Position p)` : if possible the caterpillar moves its head (and all its body) to the input position. If the input position is out of reach, i.e. if when seen as a point on a Cartesian plane it is not orthogonally connected to the head's position, then an `IllegalArgumentException` should be raised. If the position is within reach, but it is already occupied by a segment of the caterpillar's body, then moving will generate a collision leading to the caterpillar being in an `ENTANGLED` stage (unfortunately, caterpillars do not recover from this stage and the game will end.). If on the other hand, the position is reachable and moving to it would not lead to a collision, then the body of the caterpillar should be updated accordingly: all the positions in the segments should be updated to represent the caterpillar moving forward to the input position, while the colors remain the same. For example, below is the representation of a caterpillar before and after the move to the position `(4,1)`:

  (2,2) **(2,1)** **(3,1)**

  (2,1) **(3,1)** **(4,1)**

  Or, using the representation with the Cartesian plane, this is how the caterpillar would change:



The method should update the stack keeping track of the previously occupied positions accordingly (e.g., in the example above, the top of the stack would have become `(2,2)`).

This method runs in $O(n)$, where n is the number of segments.

Safe Assumptions

Throughout the assignment you can always assume the following:

- The caterpillar will always *move* before attempting to *eat*.
- The caterpillar will *eat* only when it is in a feeding stage.
- The methods will be tested only on valid inputs: `null` is not a valid input for any of the methods!
- The goal will always be initialized with a value greater than 1.
- None of the methods will be called if the caterpillar is in a butterfly or entangled stage.

Step 2: Tricky Treats

Now that Gus is up and running, it's time to deal with the more peculiar foods he encounters on its journey. In this section, you'll implement methods to handle the quirky consequences of Gus eating pickles, lollipops, and ice cream. Each food brings its own challenge, requiring you to manipulate Gus's body in more complex ways.

- `eat(Pickle p)` : the sourness of the pickle makes the caterpillar retrace its steps. This method updates its segments accordingly. Note that the position where the pickle was found should not be added to the stack of previously occupied positions.

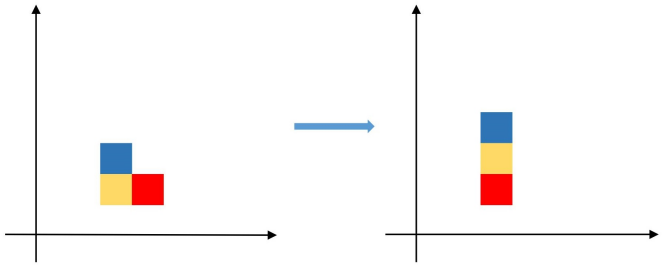
  For instance, suppose we have the following caterpillar:

  (2,2) (2,1) **(3,1)**

  And suppose that the most recent position previously occupied by this caterpillar is `(2,3)`. Then, after eating a pickle the caterpillar will have the following segments:

  (2,3) (2,2) **(2,1)**

  It might be easier to picture those segments in a Cartesian plane. If you do so, this is how the caterpillar would change in the example above:



This method runs in $O(n)$, where n is the number of segments.

- `eat(Lollipop lolly)` : this swirl of colors makes our caterpillar longing for a playful makeover. Shuffle all the caterpillar's colors!

  There are different ways of doing this, but for this assignment you will need to implement the method using the Fisher–Yates shuffle algorithm. The algorithm runs in $O(n)$ using $O(n)$ space, where n is the number of segments. To perform a shuffle of the colors follow the steps:

  – Copy all the colors inside an array

  – Shuffle the array using the following algorithm:

  ```
  for i from n-1 to 1 do
      j <-- random integer such that 0 <= j <= i
      swap a[j] and a[i]
  ```

  To generate a random integer use the `Random` object stored in the class field called `randNumGenerator`.

  – Use the array to update the colors in the all of the segments.

- `eat(IceCream gelato)` : the caterpillar did not expect this deliciously looking, creamy treat to be sooo cold! In shock, its body does a full reversal, with its tail becoming the new head. Its (new) head turns `GameColors.BLUE` like an icicle, but all other segments retain their original colors. At this point, the caterpillar loses track of where it has been before, and the stack of previously occupied positions is now empty.

  This method runs in $O(n)$, where n is the number of segments.

Step 3: Gus's Greatest Challenges

In this final section, you'll face Gus's toughest challenges yet. These tasks are designed to test your understanding of how linked lists work, especially when dealing with growth and transformation. From managing how Gus digests a cake and grows by multiple segments to handling the cheesy delight that makes him shrink, you'll need to think critically and design your solutions with efficiency in mind. This section will push you to integrate all the skills you've learned so far, tackling complex operations with precision.

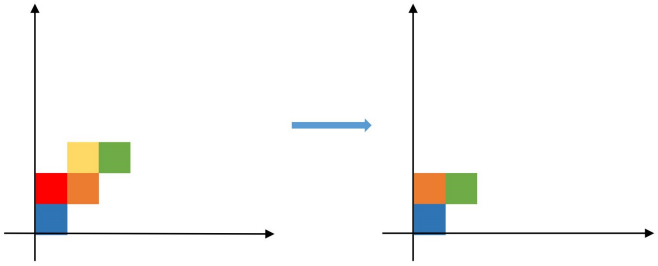
- `eat(SwissCheese cheese)` : who can resist the cheesy delight of those perfectly shaped eyes![1] But wait...by indulging on those cheesy holes the caterpillar body is getting "holey" too. This method shrinks the caterpillar who loses every other segment of the its body. This means that the segments left will have the colors of every other segment of the original body. But be careful, we do not want a caterpillar in pieces! The segments should still appear in positions that are adjacent to one another, specifically the head will remain in the original position and only the first half (rounding up) of the segments' positions will be maintained. For example, consider the following caterpillar:

  (2,2) (1,2) (1,1) (0,1) (0,0)

  After eating Swiss cheese, the caterpillar will become as follows:

  (1,1) (0,1) (0,0)

  Or, using a Cartesian plane, this is how the change to the caterpillar can be represented:

  Pay attention when updating the stack of positions previously occupied by the caterpillar. In the example above, if the stack was originally empty, after eating the cheese the stack will contain the positions `[(2,2), (1,2)]`, with `(1,2)` being at the top of the stack.

  This method runs in $O(n)$, where n is the number of segments.

- `eat(Cake cake)` : the holy grail of all treats! Here's something that will make the caterpillar truly grow. When eating a cake, the caterpillar enters its `GROWING_STAGE`. Its body will grow by as many segments as the energy provided by the cake. These segments will have a random color and will be added at the tail of the caterpillar's body. Be careful though, this growth might not take place entirely in this method! In fact, the caterpillar will grow by a number of segments that is equivalent to the minimum between the energy provided and the number of previously occupied positions that are still available to the caterpillar. For example, consider the following caterpillar:

  (0,1) (0,0) (1,0)

  and assume the cake eaten provides an energy equal to `3` while the stack of previously occupied positions contains `[(2,0), (1,0), (1,1)]` (the right most element being at the top of the stack). From eating the cake, the caterpillar should grow by 3 segments, but it is not possible to do so in this method. Only `(1,1)` can be used for growth by the caterpillar, since `(1,0)` has already been occupied by another caterpillar's segment.

[1] Yes, holes in cheese like Swiss Cheese are actually called 'eyes'!

If the cake's energy cannot be all consumed by the caterpillar in this method, the "left over" is then stored in the field `turnsNeededToDigest`, as the caterpillar will need that many turns in order to fully digest the cake.

There are two more things you should keep in mind: it is possible for the caterpillar to reach its butterfly stage while growing in this method. If this is the case, its stage should be updated and the method should terminate right away. It is also possible for the caterpillar to fully consume the energy provided by the cake right here in this method. If this is the case, and the butterfly stage has not been reached, then the feeding stage should be resumed.

Note: to generate a random color generate a random index that you can then use to select a color from the array `GameColors.SEGMENT_COLORS`.

This method runs in $O(n * m)$, where n is the number of segments and m the energy provided by the cake.

- `move(Position p)` : Time to go back to one of the first methods you implemented and add some functionality to it. If the caterpillar is still digesting a cake it had previously eaten, a segment with a random color should be added at the tail of its body ensuring its ever-growing journey continues. This in turn should decrease the number of turns left for its digestion, and might update its evolution stage to `BUTTERFLY` (if the goal was reached). If the caterpillar is in a growing stage, but its digestion is fully completed, then the `FEEDING_STAGE` should be resumed since caterpillar is now ready to venture out for more treats.

  This method runs in $O(n)$, where n is the number of segments.

A small example

Consider the following snippet of code:

```
Position startingPoint = new Position(3, 2);
Caterpillar gus = new Caterpillar(startingPoint, GameColors.GREEN, 10);
System.out.println("1) Gus: " + gus);
System.out.println("Stack of previously occupied positions: " + gus.positionsPreviouslyOccupied);

gus.move(new Position(3,1));
gus.eat(new Fruit(GameColors.RED));
gus.move(new Position(2,1));
gus.move(new Position(1,1));
gus.eat(new Fruit(GameColors.YELLOW));
System.out.println("\n2) Gus: " + gus);
System.out.println("Stack of previously occupied positions: " + gus.positionsPreviouslyOccupied);

gus.move(new Position(1,2));
gus.eat(new IceCream());
System.out.println("\n3) Gus: " + gus);
System.out.println("Stack of previously occupied positions: " + gus.positionsPreviouslyOccupied);

gus.move(new Position(3,1));
gus.move(new Position(3,2));
gus.eat(new Fruit(GameColors.ORANGE));
System.out.println("\n4) Gus: " + gus);
System.out.println("Stack of previously occupied positions: " + gus.positionsPreviouslyOccupied);

gus.move(new Position(2,2));
gus.eat(new SwissCheese());
System.out.println("\n5) Gus: " + gus);
System.out.println("Stack of previously occupied positions: " + gus.positionsPreviouslyOccupied);

gus.move(new Position(2, 3));
gus.eat(new Cake(4));
System.out.println("\n6) Gus: " + gus);
System.out.println("Stack of previously occupied positions: " + gus.positionsPreviouslyOccupied);
```

When executed the following will be displayed:

```
1) Gus: (3,2)
Stack of previously occupied positions: []

2) Gus: (3,1) (2,1) (1,1)
Stack of previously occupied positions: [(3,2)]

3) Gus: (1,2) (1,1) (2,1)
Stack of previously occupied positions: []

4) Gus: (1,1) (2,1) (3,1) (3,2)
Stack of previously occupied positions: [(1,2)]

5) Gus: (3,2) (2,2)
Stack of previously occupied positions: [(1,2), (1,1), (2,1), (3,1)]

6) Gus: (1,1) (2,1) (3,1) (3,2) (2,2) (2,3)
Stack of previously occupied positions: [(1,2)]
```