Evil List

In this homework, you are going to create your own data structure - Evil List. Most data structures are designed to solve tasks quickly and help users, but Evil List is not. Evil list is double linked list and like others it consists of nodes. Each node has unique id, previous and next node objects and data stored in it. Instead of being optimized to be fast, Evil List intentionally tries to make your program slow: Frequently accessed nodes are moved towards the end of the list, so more you need it more time it takes to get it. Furthermore, data can only be accessed using IDs. Every time data is added to the list, Evil List assignes unique ID to the node and returns ID to the user. User should remember and store ID to access same data later.

Implementation

UIDGenerator

As mentioned above, Evil List assigns unique IDs to the nodes. We need to create UIDGenerator class that will be able to generate those IDs for Evil List. Each generated ID should be a unique string.

UIDGenerator class should have following structure:

- **generateRandomNumberString** - A static method that generates a random string of digits. String's length is passed as an argument. Each element of the string should be a randomly chosen digit. Example valid strings generated by this function: '1', '573', '796', '0133', '00', '0238301923019'.
- **decimalToHexavigesimal** - A static method that converts given decimal number to Hexavigesimal. Hexavigesimal is a numeral system using only the English alphabet to represent numbers in base 26. Description of Hexavigesimal system can be found bellow.
- **generate_id** - A class method that generates a unique ID. ID should have the following format: It starts with 3 digits, followed with a dash ('-'), and ends with one or more upper case English letters. First 3 digit part should be generated with *generateRandomNumberString* method. These 3 random digits can not guarantee uniqueness, hence the second part needs to be unique. We should make sure that every time the ID is generated letters following the dash are unique. To achieve this method we should add a class variable **id_seed** to UIDGenerator class. *id_seed_* is a positive integer that can be converted to the sequence of letters using *decimalToHexavigesimal* method. Like any other numerical system, numbers in Hexavigesimal are unique, therefore as long as we provide different *id_seed* every time we will get unique strings. At first *id_seed* should be any 4 digit positive integer (Actually, it could have been any positive integer but 'IAAB' looks fancier than just 'A') and every time a new ID is generated *id_seed* should increase by one.

Hexavigesimal Numbers

Decimal	Hexavigesimal
0	`A`
1	`B`
2	`C`
...	...
24	`Y`
25	`Z`
26	`BA`
27	`BB`

Decimal	Hexavigesimal
28	`BC`
...	...
675	`ZZ`
676	`BAA`
677	`BAB`
...	...

Hexavigesimal is the numeric system with base 26, where instead of digits Latin alphabet is used to write down the number: A for 0, B for 1, C for 2, and so on.

The decimal number can be converted into Hexavigesimal with the following steps:

1. Take the remainder by 26 of the number and convert it into the alphabet.
2. Divide the number by 26 with integer division.
3. Repeat the steps until the number becomes 0.
4. Reverse the resulting sequence.

or:

1. Divide number into the sum of powers of 26. Ex.: 678 = 1 * (26 * 26) + 0 * 26 + 2
2. Take the coefficients and convert them to letters: 0 to A, 1 to B, 2 to C, and so on.
3. Assemble letters into the string

Example: 678 -> 1 * (26 * 26) + 0 * 26 + 2 -> 1, 0, 2 -> B, A, C -> BAC

DLNode

Like ordinary linked lists, Evil List is also a collection of nodes. You should create a **DLNode** class to describe double-linked nodes stored in Evil List. Each node should have the following attributes:

- **id** - Unique ID assigned to each node. ID should be generated using **UIDGenerator.generate_id()** method. ID is assigned to the node when it is created and should not be changed afterward, therefore it should be a hidden attribute.
- **popularity** - Popularity describes how often the node is accessed. It should be an integer that is increased every time the node is accessed. Popularity should be changed only with the **recordAccess** method, to forbid direct access it should be a hidden attribute. *The node is accessed by the user only when it is searched using the **findByID** method.*
- **data** - Data public attribute is used to store information. For simplicity, you can assume that data is an integer. But as we are dealing with a dynamic programming language it can be anything.
- **next** - Each node knows which node comes next. The right neighbor node is stored in the *next* attribute. In linked lists pointer to the next node is the only way to move around. If the node does not have the next node then it is the last node of the list and the *next* attribute will be *None*.
- **previous** - As Evil List is a double linked list, each DLNode object also has a pointer to the previous node which is stored in the public attribute. If the element is the first one it does not have a predecessor and its *previous* attribute will be *None*.

Furthermore, DLNode class should have the following methods:

- **init** - Constructor to initialize object. The constructor should take one argument, data that needs to be stored. *id* attribute should be assigned unique ID generated from UIDGenerator, *popularity* should be 0, *data* attribute should be assigned value passed as an argument, *previous* and *next* attributes should be *None*.
- **recordAccess** - This method should be used to record accessing the object, it should increase popularity by one.
- **getPopularity** - Returns popularity. As *popularity* is a hidden attribute it's better to have a getter function for it.

- **getID** - Returns id. As *id* is a hidden attribute it's better to have a getter function for it.

EvilList

As mentioned above Evil List is a doubly-linked list (DLL). Its structure is similar to a typical DLL. but the behavior is a bit different. Evil List consists of DLNode objects and it has only one attribute - *head*. *head* is *None* when the list is empty, otherwise, it should be DLNode object - first node of the list. *head* can be the start point for traversing the list.

EvilList class should have the following methods:

- **init** - The constructor should initialize the *head* attribute with *None*.
- **put** - This method gets *data* as an argument that should be added to the list. At first DLNode object should be created with the given data. Next, *previous* and *next* attributes should be modified in a way that it becomes part of the list. At last, when the node is added to the list its ID should be returned.
- **removeByID** - This method gets ID as an argument. Find a node with the given ID and remove it from the list. If a node with such ID does not exist do nothing.
- **findByID** - This method gets ID as an argument. Find a node with the given ID and return the data stored in it. If a node with such ID does not exist return *None*. Furthermore, 2 things should happen if the node is found:
  1. Popularity of the accessed node should be increased using the *recordAccess* method.
  2. To make the evil list as not-optimized as possible, we should increase access time for the popular nodes. The farther the node is more time it takes to reach it, so every time the node is accessed we move it one step back (we swap it with its next node) if it is not last already.
- **size** - This method should return a single integer - number of nodes in the list.
- **printList** - This method prints the list into the console. Each node's data should be printed separated by a space.
- **sort** - This method should rearrange nodes in the list in increasing order of popularity. The least popular nodes should be close to the head and the most popular ones would be close to the tail. Using merge sort is suggested, but you are free to implement any sorting algorithm. **NOTE: You are not allowed to use built-in functions to sort the linked list.**

Interactive visualisation for DLL - https://visualgo.net/en/list

Grading

This assignment is worth 10 points in total.

- UIDGenerator
  - generate_id - 1pt
  - generateRandomNumberString - 1pt
  - decimalToHexavigesimal - 1pt
- DLNode - 2pt
- EvilList
  - put - 1pt
  - removeByID - 1pt
  - findByID - 1pt
  - size - 0.5pt
  - printList - 0.5pt
  - sort - 1pt

You can get half points in each subtask if code is almost correct.

If the code does not compile and run, your code **will be graded with 0 points**.