You are given the structure of a class that handles basic music playlist operations. Each playlist consists of several tracks. Your task is to complete the implementation of the missing methods.

The class **Track** has **3** properties:

* **name** - the name of the track represented as a non-empty string consisting of no more then **100** ASCII-characters.
* **duration** - an integer representing the duration of the track, in seconds.
* **pausedOn** - the second on which the track was paused the last time it was played.

You must complete the implementation of the following methods:

1. **play(time)** - plays the track starting from moment **pausedOn** for **time** seconds. If the track ends before **time** seconds are left, it starts over.
2. **reset()** - resets the track (sets **pausedOn** to **0**).

The main class **Playlist** contains only the property **trackList** - the list of tracks in the playlist. Note that the order of the tracks in the playlist is important.

You must complete the implementation of the following methods:

1. **addTrack(name, duration)** - adds a track with the given name and duration to the end of the playlist. It is guaranteed that there is no track with the given name when this method is called.
2. **deleteTrack(name)** - deletes the track with the given name. It is guaranteed that there is exactly **1** track with the given name in the playlist when this method is called.
3. **playTrack(name, time)** - plays the track with the given name for the given number of seconds. It is guaranteed that there is exactly **1** track with the given name in the playlist when this method is called.
4. **resetTrack(name)** - resets the track with the given name. **If no name is given, this method resets all the tracks in the playlist**. It is guaranteed that there is exactly **1** track with the given name (if a name is given) in the playlist when this method is called.
5. **moveTrack(name, toIndex)** - moves the track with the given name to the given index. Does not change the order of the other tracks in the playlist. It is guaranteed that there is exactly **1** track with the given name in the playlist when this method is called.

*Please do not edit the rest of the existing methods. However, feel free to add any other new methods to the classes if they will help you to solve the problem*

Example

For **commands = ["add", "play", "get", "add", "get"]**, **names = ["Clair de Lune", "Clair de Lune", "", "Toxicity", ""]**, and **parameters = [303, 603, 0, 283, 0]**, the output should be

**solution(commands, names, parameters) = [**

**"[Track(name = Clair de Lune, duration = 303, pausedOn = 300)]",**

**"[Track(name = Clair de Lune, duration = 303, pausedOn = 300), Track(name = Toxicity, duration = 283, pausedOn = 0)]"**

**]**

Input/Output

* **[execution time limit] 4 seconds (js)**
* **[input] array.string commands**

**commands[i]** is one of the following strings:

* + **add**;
  + **delete**;
  + **play**;
  + **reset**;
  + **move**;
  + **get**.

*Guaranteed constraints:*  
**4 ≤ commands.length ≤ 100**.

* **[input] array.string names**

**names[i]** can have length **0** only for **reset** and **get** commands, in all other cases the name of the track is guaranteed to be a non-empty string.

*Guaranteed constraints:*  
**names.length = commands.length**,  
**0 ≤ names[i].length ≤ 100**.

* **[input] array.integer parameters**

*Guaranteed constraints:*  
**parameters.length = commands.length**,  
**0 ≤ parameters[i] ≤ 1000**.

* **[output] array.string**

**[JavaScript] Syntax Tips**

class Track {

  constructor(name, duration) {

    this.name = name;

    this.duration = duration;

    this.pausedOn = 0;

  }

  play(time) {

    // TODO: implement this method

  }

  reset() {

    // TODO: implement this method

  }

  toString() {

    return `Track(name = ${this.name}, duration = ${this.duration}, pausedOn = ${this.pausedOn})`;

  }

}

class Playlist {

  constructor() {

    this.trackList = [];

  }

  addTrack(name, duration) {

    // TODO: implement this method

  }

  deleteTrack(name) {

    // TODO: implement this method

  }

  playTrack(name, time) {

    // TODO: implement this method

  }

  resetTrack(name) {

    // TODO: implement this method

  }

  trackIndexByName(name) {

    // TODO: implement this method

    return -1;

  }

  moveTrack(name, toIndex) {

    const index = this.trackIndexByName(name);

    if (index === toIndex) {

      return;

    }

    // TODO: complete the implementation of this method

  }

  toString() {

    return '[' + this.trackList.map(track => track.toString()).join(', ') + ']';

  }

}

function solution(commands, names, parameters) {

  const playlist = new Playlist();

  const result = [];

  for (let i = 0; i < commands.length; i++) {

    if (commands[i] === 'add') {

      playlist.addTrack(names[i], parameters[i]);

    } else if (commands[i] === 'delete') {

      playlist.deleteTrack(names[i]);

    } else if (commands[i] === 'play') {

      playlist.playTrack(names[i], parameters[i]);

    } else if (commands[i] === 'reset') {

      if (names[i] === '') {

        playlist.resetTrack();

      } else {

        playlist.resetTrack(names[i]);

      }

    } else if (commands[i] === 'move') {

      playlist.moveTrack(names[i], parameters[i]);

    } else { // commands[i] === 'get'

      result.push(playlist.toString());

    }

  }

  return result;

}

**Input:**

**commands:**

**["add",**

**"play",**

**"get",**

**"add",**

**"get"]**

**names:**

**["Clair de Lune",**

**"Clair de Lune",**

**"",**

**"Toxicity",**

**""]**

**parameters: [303, 603, 0, 283, 0]**

**Expected Output:**

**["[Track(name = Clair de Lune, duration = 303, pausedOn = 300)]",**

**"[Track(name = Clair de Lune, duration = 303, pausedOn = 300), Track(name = Toxicity, duration = 283, pausedOn = 0)]"]**

**Input:**

**commands:**

**["add",**

**"get",**

**"add",**

**"get"]**

**names:**

**["Loveless Love",**

**"",**

**"\"Long Gone(From the Bowlin Green)\"",**

**""]**

**parameters: [1000, 0, 660, 0]**

**Expected Output:**

**["[Track(name = Loveless Love, duration = 1000, pausedOn = 0)]",**

**"[Track(name = Loveless Love, duration = 1000, pausedOn = 0), Track(name = \"Long Gone(From the Bowlin Green)\", duration = 660, pausedOn = 0)]"]**

*Implement the missing code, denoted by ellipses. You may not modify the pre-existing code.*

Usernames consist of an alphabetic name (e.g. **"alex"**) concatenated with some positive number (so **"alex1"**, **"alex2"**, etc. are valid usernames).

Create a class called **Generator** that supports two operations - **create(name)** and **delete(username)**. The former should reserve the first available username and return it, while the latter should release that username back into the pool.

The input for this task is an array of sequential queries, where a query of type **"create <name>"** means a call to **Generator.create(<name>)**, and query of type **"delete <username>"** means a call **Generator.delete(<username>)**. The output should be an array of return values of all **create** calls. There is already a prewritten piece of code that handles the input/output and makes create/delete calls, so you just need to create the **Generator** class that implements them.

**Note** that deleting a non-created username is a valid operation.

Example

For **queries = ["create alex", "create alex", "delete alex1", "create alex", "create john"]**, the output should be  
**solution(queries) = ["alex1", "alex2", "alex1", "john1"]**.  
That's how it should work:

**>> var Generator = new Generator();**

**>> Generator.create('alex');**

**"alex1"**

**>> Generator.create('alex');**

**"alex2"**

**>> Generator.delete('alex1');**

**>> Generator.create('alex');**

**"alex1"**

**>> Generator.create('john');**

**"john1"**

Input/Output

* **[execution time limit] 4 seconds (js)**
* **[input] array.string queries**

An array of queries to the Generator. **queries[i] = "create <name>"** means that you should call **Generator.create(<name>)** and return the reserved username, and **queries[i] = "delete <username>"** means that you should call **Generator.delete(<username>)** and return nothing. It is guaranteed that all server numbers of the deleting queries won't exceed **1000**.

*Guaranteed constraints:*  
**1 ≤ queries.length ≤ 103**.

* **[output] array.string**

An array of answers from the Generator.

**[JavaScript] Syntax Tips**

...

function solution(queries) {

  const generator = new Generator();

  const results = [];

  queries.forEach((query) => {

    const [action, name] = query.split(' ');

    if (action === 'create') {

      results.push(generator.create(name));

    } else if (action === 'delete') {

      generator.delete(name);

    }

  });

  return results;

}

**Input:**

**queries:**

**["create alex",**

**"create alex",**

**"delete alex1",**

**"create alex",**

**"create john"]**

**Expected Output:**

**["alex1",**

**"alex2",**

**"alex1",**

**"john1"]**

When you have free time, you enjoy cracking ciphers. But to crack a cipher you need some ciphered text, and preparing it yourself is way too boring. So, you decide to implement an algorithm that will encode text for you.

After some research you choose the famous *Nihilist cipher*, and here is how it works (please note that this task uses the modified version of the cipher, so it can differ from what you might already know):

* Build a **5 × 5** *Polybius square* for the given **keyword** consisting of distinct letters:
  + first, fill the square with the letters of the keyword row by row.
  + then, continue filling the square with all the letters in alphabetical order except for those that are already in the square.
  + add the final (**26th**) letter to the cell with coordinates **(row, column)**.
* Now each letter can be associated with two integers, the indices of a row and a column of the *Polybius square* containing it. In order to encode a letter, replace it with a concatenation of string representations of these two integers.
* Make the lengths of the given **plaintext** message and the **keyword** equal (if one string is shorter than another one, consider the [cyclic string](keyword://cyclic-string) obtained from it and take its prefix of the length equal to the length of the longer string).
* Finally, encode the letters of both strings obtained on the previous step and add up the values at the same positions. The resulting list of integers is the desired ciphertext.

Given some **plaintext** to encode, a **keyword** and the coordinates of a Polybius square cell which will contain two letters, return the encoded text.

Example

For **plaintext = "codesignal"**, **keyword = "keyword"**, **row = 1**, and **column = 2**, the output should be  
**solution(plaintext, keyword, row, column) = [36, 27, 35, 26, 66, 55, 54, 54, 35, 54]**.

The *Polybius square* appears as follows:

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| k | e/z | y | w | o |
| r | d | a | b | c |
| f | g | h | i | j |
| l | m | n | p | q |
| s | t | u | v | x |

Here are the results of the remaining steps:

|  |  |  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- |
| plaintext | c | o | d | e | f | i | g | h | t | s |
| encoded plaintext | 25 | 15 | 22 | 12 | 31 | 34 | 32 | 33 | 52 | 51 |
| keyword | k | e | y | w | o | r | d | k | e | y |
| encoded keyword | 11 | 12 | 13 | 14 | 15 | 21 | 22 | 11 | 12 | 13 |
| resulting ciphertext | 36 | 27 | 35 | 26 | 46 | 55 | 54 | 44 | 64 | 64 |

Input/Output

* **[execution time limit] 4 seconds (js)**
* **[input] string plaintext**

A string consisting only of lowercase English letters.

*Guaranteed constraints:*  
**1 ≤ plaintext.length ≤ 100**.

* **[input] string keyword**

A string consisting of distinct lowercase English letters.

*Guaranteed constraints:*  
**1 ≤ keyword.length ≤ 26**.

* **[input] integer row**

**1**-based index of the *Polybius square* row which will contain the last remaining letter.

*Guaranteed constraints:*  
**1 ≤ row ≤ 5**.

* **[input] integer column**

**1**-based index of the *Polybius square* column which will contain the last remaining letter.

*Guaranteed constraints:*  
**1 ≤ column ≤ 5**.

* **[output] array.integer**

**[JavaScript] Syntax Tips**

function solution(plaintext, keyword, row, column) {

}

**Input:**

**plaintext: "codesignal"**

**keyword: "keyword"**

**row: 1**

**column: 2**

**Expected Output:**

**[36, 27, 35, 26, 66, 55, 54, 54, 35, 54]**

**Input:**

**plaintext: "nihilist"**

**keyword: "cipher"**

**row: 1**

**column: 1**

**Expected Output:**

**[52, 24, 27, 26, 49, 33, 55, 57]**

**Input:**

**plaintext: "z"**

**keyword: "ab"**

**row: 1**

**column: 1**

**Expected Output:**

**[22, 23]**

One of your new coworkers has submitted code with variable names in *snake case*, where multiword names are separated by underscores (such as **my\_counter**). Your company's convention is to use only *lower camel case*, where multiword variable names are concatenated, capitalizing the first letter of every word except the first (e.g. **myCounter**).

Your team is tasked with taking the source code **src** from your coworker, and returning code with the all the names in snake case converted into lower camel case. More specifically:

* Variable names may start with one or more underscores, and these should be preserved. For example **\_the\_variable** should become **\_theVariable**
* Variable names may end with trailing underscores, and these should be preserved. For example, **the\_variable\_\_** should become **theVariable\_\_**.
* To keep the problem simple, you are not restricted to variable names, but instead should replace all instances of snake case.

Example

For **src = "This is the doc\_string for \_\_secret\_fun"**, the output should be  
**solution(src) = "This is the docString for \_\_secretFun"**.

Input/Output

* **[execution time limit] 4 seconds (js)**
* **[input] string src**

All variables in **src** are in lowercase English letters. It's guaranteed that in the variable names there can be only one **\_** between words.

*Guaranteed constraints:*  
**0 ≤ src.length ≤ 105**.

* **[output] string**

Return the source code **src** with all instances of snake case converted to lower camel case, preserving any leading or trailing underscores.

**[JavaScript] Syntax Tips**

function solution(src) {

}

**Input:**

**src: "This is the doc\_string for \_\_secret\_fun"**

**Expected Output:**

**"This is the docString for \_\_secretFun"**

**nput:**

**src: ""**

**Expected Output:**

**""**

Click the "Run Tests" button to see output and console logs.

**nput:**

**src: "\_one\_variable"**

**Expected Output:**

**"\_oneVariable"**

Click the "Run Tests" button to see output and console logs.

Convert a non-negative integer to its equivalent representation as words in English.

Example

* For **num = 1**, the output should be  
  **solution(num) = "One"**;
* For **num = 123**, the output should be  
  **solution(num) = "One Hundred Twenty Three"**;
* For **num = 12345**, the output should be  
  **solution(num) = "Twelve Thousand Three Hundred Forty Five"**;
* For **num = 1234567**, the output should be  
  **solution(num) = "One Million Two Hundred Thirty Four Thousand Five Hundred Sixty Seven"**.

Input/Output

* **[execution time limit] 4 seconds (js)**
* **[input] integer num**

*Guaranteed constraints:*  
**0 ≤ num ≤ 231 - 1**.

* **[output] string**

**[JavaScript] Syntax Tips**

function solution(num) {

}

**nput:**

**num: 1**

**Expected Output:**

**"One"**

Click the "Run Tests" button to see output and console logs.

**Input:**

**num: 123**

**Expected Output:**

**"One Hundred Twenty Three"**

**Input:**

**num: 12345**

**Expected Output:**

**"Twelve Thousand Three Hundred Forty Five"**

Click the "Run Tests" button to see output and console logs.

You are given an array of integers **memory** consisting of **0**s and **1**s - whether the corresponding memory unit is free or not. **memory[i] = 0** means that the **ith** memory unit is free, and **memory[i] = 1** means it's occupied.

Your task is to perform two types of queries:

* **alloc X**: Find the left-most memory block of **X** consecutive free memory units and mark these units as occupied (ie: find the left-most contiguous subarray of **0**s, and replace them all with **1**s).
  + If there are no blocks with **X** consecutive free units, return **-1**; otherwise return the index of the first position of the allocated block segment and assign an **ID** to every single element in this block, based on an **atomic counter** *(the counter starts at****1****and is incremented on every successful alloc operation)*.
* **erase ID**: If there exists an allocated memory block with element ids equal to **ID**, free all its memory units.
  + Return the length of the deleted memory block. If there is no such **ID** or the block with this **ID** has already been deleted, return **-1**.

The queries are given in the following format:

* **queries** is an array of 2-elements arrays;
* if **queries[i][0] = 0** then this is an **alloc** type query, where **X = queries[i][1]**;
* if **queries[i][0] = 1** then this is an **erase** type query, where **ID = queries[i][1]**.

Return an array containing the results of all the queries.

Example

* For **memory = [0, 1, 0, 0, 0, 1, 1, 0, 0, 0, 1, 0, 0]** and **queries = [[0, 2], [0, 1], [0, 1], [1, 2], [1, 4], [0, 4]]**, the output should be **solution(memory, queries) = [2, 0, 4, 1, -1, -1]**.

Expand to see the example video.

*Note*: If you are not able to see the video, use [this link](https://codesignal.s3.amazonaws.com/uploads/1637066723358/example.mp4) to access it.

* + **[0, 2]** corresponds to **alloc 2**, which allocates a memory block from units **2** to **3**. We also assign **ID = 1** to this segment. After this operation **memory = [0, 1, 1, 1, 0, 1, 1, 0, 0, 0, 1, 0, 0]**. Return the memory block starting index - **2**.
  + **[0, 1]** corresponds to **alloc 1**, which allocates a memory block from units **0** to **0**. After this operation, **memory = [1, 1, 1, 1, 0, 1, 1, 0, 0, 0, 1, 0, 0]**. We also assign **ID = 2** to this segment.
  + **[0, 1]** corresponds to **alloc 1**, which allocates a memory block from units **4** to **4**. After this operation, **memory = [1, 1, 1, 1, 1, 1, 1, 0, 0, 0, 1, 0, 0]**. We also assign **ID = 3** to this segment.
  + **[1, 2]** corresponds to **erase 2**. The range with **ID = 2** is **[0, 0]** (just the **0th** element). After freeing the memory, **memory = [0, 1, 1, 1, 1, 1, 1, 0, 0, 0, 1, 0, 0]**. Return the length of the removed memory block - **1**.
  + **[1, 4]** corresponds to **erase 4**. The current counter is equal to **3** (since there have been 3 successful allocations), so we return **-1** since there is no range with **ID = 4**.
  + **[0, 4]** corresponds to **alloc 4**. There is no memory block of length **4** in which all blocks are free, so return **-1**.

Input/Output

* **[execution time limit] 4 seconds (js)**
* **[input] array.integer memory**

An array of **0**s and **1**s representing bits of memory.

*Guaranteed constraints:*  
**1 ≤ memory.length ≤ 300**.

* **[input] array.array.integer queries**

An array of 2-element arrays representing queries of the type **alloc** or **erase**.

*Guaranteed constraints:*  
**2 ≤ queries.length ≤ 300**,  
**queries[i].length = 2**,  
**0 ≤ queries[i][0] ≤ 1**,  
**If queries[i][0] = 0, then 1 ≤ queries[i][1] ≤ memory.length**,  
**If queries[i][0] = 1, then 1 ≤ queries[i][1] ≤ queries.length - 1**.

* **[output] array.integer**

Return an array in which the **ith** element is equal to the answer of the **ith** query.

**[JavaScript] Syntax Tips**

**nput:**

**memory: [0, 1, 0, 0, 0, 1, 1, 0, 0, 0, 1, 0, 0]**

**queries:**

**[[0,2],**

**[0,1],**

**[0,1],**

**[1,2],**

**[1,4],**

**[0,4]]**

**Expected Output:**

**[2, 0, 4, 1, -1, -1]**

Click the "Run Tests" button to see output and console logs.

**Input:**

**memory: [1]**

**queries:**

**[[0,1],**

**[1,1]]**

**Expected Output:**

**[-1, -1]**

Click the "Run Tests" button to see output and console logs.