

8월 21일 (토)

1. 백준 알고리즘 문제풀이 - 큐 10845번

큐 성공 ☆

시간 제한	메모리 제한	제출	정답	맞은 사람	정답 비율
0.5 초 (추가 시간 없음)	256 MB	57018	27085	20840	49.099%

문제

정수를 저장하는 큐를 구현한 다음, 입력으로 주어지는 명령을 처리하는 프로그램을 작성하시오.

명령은 총 여섯 가지이다.

- push X: 정수 X를 큐에 넣는 연산이다.
- pop: 큐에서 가장 앞에 있는 정수를 빼고, 그 수를 출력한다. 만약 큐에 들어있는 정수가 없는 경우에는 -1을 출력한다.
- size: 큐에 들어있는 정수의 개수를 출력한다.
- empty: 큐가 비어있으면 1, 아니면 0을 출력한다.
- front: 큐의 가장 앞에 있는 정수를 출력한다. 만약 큐에 들어있는 정수가 없는 경우에는 -1을 출력한다.
- back: 큐의 가장 뒤에 있는 정수를 출력한다. 만약 큐에 들어있는 정수가 없는 경우에는 -1을 출력한다.

입력

첫째 줄에 주어지는 명령의 수 N ($1 \leq N \leq 10,000$)이 주어진다. 둘째 줄부터 N개의 줄에는 명령이 하나씩 주어진다. 주어지는 정수는 1보다 크거나 같고, 100,000보다 작거나 같다. 문제에 나와있지 않은 명령이 주어지는 경우는 없다.

출력

출력해야하는 명령이 주어질 때마다, 한 줄에 하나씩 출력한다.

출력

출력해야하는 명령이 주어질 때마다, 한 줄에 하나씩 출력한다.

예제 입력 1 복사

```
15
push 1
push 2
front
back
size
empty
pop
pop
pop
size
empty
pop
push 3
empty
front
```

예제 출력 1 복사

```
1
2
2
0
1
2
-1
0
1
-1
0
3
```

출처

- 문제를 만든 사람: baekjoon
- 문제의 오답을 찾은 사람: compro0317

```

6  from sys import stdin
7
8  n = int(input())
9  queue = []
10 result = []
11
12 for _ in range(n):
13
14     command = stdin.readline().strip() #strip() 공이 안써도 됐다.
15
16     if command[0:4] == 'push': # 애초에 입력할 때 split()를 사용한다면 슬라이스로 나누지 않아도 됐다.
17         queue.append(command[5:])
18
19     elif command == 'pop':
20         if len(queue) != 0:
21             result.append(queue[0])
22             del queue[0]
23         else:
24             result.append(-1)
25             continue
26
27     elif command == 'size':
28         result.append(len(queue))
29
30     elif command == 'empty':
31         if len(queue) != 0:
32             result.append(0)
33         else:
34             result.append(1)
35
36     elif command == "front":
37         if len(queue) != 0:
38             result.append(int(queue[0]))
39         else:
40             result.append(-1)
41
42
43     elif command == "back":
44         if len(queue) != 0:
45             result.append(int(queue[-1]))
46         else:
47             result.append(-1)
48
49 #답 출력
50 for i in result: # 답 출력시 그냥 print()로 해도 된다

```

- 입력 커맨드에 띄어쓰기를 사용한 커맨드가 있다면 `split()` 을 사용하자 !
- 파이썬에는 리스트의 맨 앞 인덱스를 추가, 제거하는 `unshift` , `shift` 메서드가 없다.
다음 아래 방법을 이용하자.

👍

0

👎

✓

1. s.pop([i])

s의 i 번째 아이템을 제거하고 i 번째 아이템을 return합니다

```
>>> l = [0, 1, 2, 3, 4]
>>> l.pop(0)
0
>>> l
[1, 2, 3, 4]
```

[코드 실행하기 >](#)

2. del s[i:j]

s[i:j] = [] 과 같습니다

```
>>> l = [0, 1, 2, 3, 4]
>>> del l[0]
>>> l
[1, 2, 3, 4]
```

[코드 실행하기 >](#)


3. slicer 이용

l[1:] 은 iterable l의 두 번째 원소부터 끝까지를 복사해 return해줍니다

```
>>> l = [0, 1, 2, 3, 4]
>>> l = l[1:]
>>> l
[1, 2, 3, 4]
```

[코드 실행하기 >](#)

[편집 요청](#)

바바 40 points

2016-02-04 17:01:53에 작성됨

2. Codecademy - TDD Fundamentals - Write Good Tests with Mocha

Lesson 1. Automate and Organize Tests

- Introduction
- Install Mocha 1

8월 21일 (토)

4

- Install Mocha 2
- describe and it blocks
- assert
- Setup, Exercise, and Verify
- Teardown
- Hooks
- Review

• Introduction

개발에 있어서 테스트는 필수적이다. 모든 기능을 수동적으로 테스트하는 대신에, 테스트 프레임워크를 이용하여 테스트를 자동화할 수 있다. 이번 레슨에서는 Mocha 테스트 프레임워크를 사용해 볼 것이다.

이번에 배울 것

- ⇒ 기본적인 Mocha test suite 써보기
- ⇒ 코드의 기대값을 확인하는 노드의 assert.ok 메서드 사용해보기
- ⇒ expressive testing suite 를 생성하기 위한 테스트의 4단계를 이해하고 적용하기
- ⇒ 좋은 테스트의 특성을 참고하여 테스트 퀄리티 평가하기

• Install Mocha 1

테스트를 작성하기 전에, 자바스크립트 프로젝트 셋팅과 Mocha 설치를 위해 Node.js 와 npm 을 사용해야한다.

- ⇒ 노드는 자바스크립트를 터미널에서 실행시킬 수 있게한다.

⇒ npm은 웹으로부터 패키지를 다운로드할 수 있도록 하고, 자바스크립트 프로젝트 안에서 그들을 관리할 수 있도록 하는 Node 툴이다.

⇒ Mocha 는 자바스크립트 코드를 테스트하기 위해 사용되는 여러 패키지들 중 하나이다.

```
$ npm init
```

```
$ npm install mocha -D
```

```
project
├_ node_modules
├___ .bin
├___ mocha
├___ ...
└_ package.json
```

일단 npm install package 를 하면 모든 종속 패키지를 node_modules 폴더에서 찾아볼 수 있을 것이다. 위는 새로운 mocha 디렉토리가 포함된 모습이다.

- **Install Mocha 2**

Mocha 를 종속모듈로 설치한 뒤, 이를 실행할 수 있는 두 가지 방법이 있다.

⇒ 첫 번째 방법은 지루한 방법으로 node_modules 로부터 직접 부르는 것이다.

```
$ ./node_modules/mocha/bin/mocha
```

⇒ 두 번째 방법은 권고하는 방법으로 package.json 파일에 스크립트를 추가하는 것이다.
package.json 파일내 scripts 객체에 아래와 같은 값을 추가해라.

```
"scripts": {  
  "test": "mocha"  
}
```

이렇게 추가하고 나면 너는 아래 커맨드를 사용하여 Mocha 를 실행할 수 있다.

```
$ npm test
```

이로써 자동화 테스트를 할 수 있는 여건을 마련한 것이다.

실습 예제

```
× package.json

1  ▼ {
2    "name": "learn-mocha-learn-mocha-install-mocha-ii",
3    "version": "1.0.0",
4    "description": "",
5    "main": "index.js",
6    ▼ "scripts": {
7      "test": "echo \"Error: no test specified\" && exit 1"
8      "test": "mocha"
9    },
10   "author": "",
11   "license": "ISC",
12   ▼ "devDependencies": {
13     "mocha": "^4.0.1"
14   }
15 }
16
```

- **describe and it blocks (describe, it 함수)**

모카에서 describe 함수를 사용해서 테스트를 그룹화한다. 그리고 it 함수를 사용하여 테스트를 정의한다. 이 두 함수는 test suite 를 완벽하고, 유지보수가 가능하며, 표현력있도록 만들어 준다.

예시로 .max 메서드를 가진 Math 객체를 테스트해보자.


```
describe('Math', () => {
  describe('.max', () => {
    it('returns the argument with the highest value', () => {
      // Your test goes here
    });
    it('returns -Infinity when no arguments are provided', ()
=> {
      // Your test goes here
    });
  });
});
```

describe 와 it 함수는 설명문자열, 콜백함수 이 두 가지 파라미터를 받는다. 이것은 test suite 를 독립시키고, 유지보수가가능하게 하며, 표현력있게 한다. it 문 안에 테스트 코드를 입력하는 것이다.

- **assert**

테스트를 작성하기 위해 우리는 Node.js에서 제공하는 assert.ok 메서드를 사용할 수 있다. 프로그래밍에서 테스트는 기대값과 실제값을 비교한다.

```
const a = 1 + 2;
```

다음 값을 테스트해본다면, a는 3이 되도록 기대된다. 바닐라 자바스크립트에서 조건문을 작성하여 결과값과 기대값을 비교하여 에러를 낼 수 있다. 우리가 사용할 assert.ok 도 이러한 기능을 갖는데, 이 메서드 또한 값을 비교하고 에러를 던져준다.

```
const assert = require('assert');
```

노드의 하나의 모듈로써 이와 같이 import 해줘야한다.

```
assert.ok(a === 3);
```

이와 같이 사용하면된다. 이러한 경우 `assert.ok` 는 `True` 를 출력하며 에러를 던지지 않는다. 이 메서드가 `false` 를 출력할 땐, `AssertionError` 를 던지며, 모카는 테스트가 실패했다는 문구를 출력하며, 에러 메시지를 콘솔안에 로그한다.

실습 예제

```
× index_test.js

1 // import assert here
2 const assert = require('assert');
3
4 ▼ describe('+', () => {
5   ▼ it('returns the sum of its arguments', () => {
6     // Write assertion here
7     assert.ok(3 + 4 === 8);
8   });
9 });
10
```

```
+ | × bash | ↗  
$ npm test  
  
> learn-mocha-intro-start@1.0.0 test /home/ccuser/workspace/  
learn-mocha-learn-mocha-assert  
> mocha test/**/*.test.js  
  
+  
  1) returns the sum of its arguments  
  
0 passing (7ms)  
1 failing  
  
1) + returns the sum of its arguments:  
  
    AssertionError: false == true  
      + expected - actual  
  
      -false  
      +true  
  
      at Context.it (test/index_test.js:7:12)  
  
npm ERR! Test failed.  See above for more details.  
$
```

값을 7로 다시 고치고 npm test 를 실행하면 아래와 같은 결과를 터미널에 출력한다.

```
$ npm test

> learn-mocha-intro-start@1.0.0 test /home/ccuser/workspace/learn-mocha-learn-mocha-assert
> mocha test/**/*.test.js

+
  ✓ returns the sum of its arguments

1 passing (5ms)
```

- **Setup, Exercise, and Verify**

이번 실습에서 테스트를 setup, exercise, verify 이 3단계로 나눠볼 것이다. 뚜렷하고 잘 정의된 스텝의 분리는 테스트를 더 믿음직하고, 유지보수가 가능하게 하며, 표현력 있게 한다.

⇒ Setup : 객체, 변수를 만들고 너의 테스트가 의존하는 조건을 세팅한다.

⇒ Exercise : 너가 테스트하려는 기능을 실행한다.

⇒ Verify : 실행단계의 결과에 대해서 너의 기대값을 체크해라. 이 때 assert 라이브러리를 사용할 수 있다.

실습예제

```
× index_test.js

1  const assert = require('assert');
2
3  // Naive approach
4  ▼ describe('.pop', () => {
5    ▼ it('returns the last element in the array [naive]', () => {
6      assert.ok(pop(['padawan', 'knight']) === 'knight');
7    });
8  });
9
10 // 3 phase approach
11 ▼ describe('.pop', () => {
12   ▼ it('returns the last element in the array [3phase]', () => {
13     // Setup
14     const knightString = 'knight';
15     const jediPath = ['padawan', knightString];
16
17     // Exercise
18     const popped = pop(jediPath);
19
20     // Verify
21     assert.ok(popped === knightString);
22   });
23 });
24
25
```

```

$ npm test

> learn-mocha-intro-start@1.0.0 test /home/ccuser/workspace/learn-mocha-learn-mocha-setup-exercise-verify
> mocha test/**/*.test.js

    .pop
      1) returns the last element in the array [naive]

    .pop
      2) returns the last element in the array [3phase]

0 passing (6ms)
2 failing

1) .pop returns the last element in the array [naive]:
   ReferenceError: pop is not defined
       at Context.it (test/index_test.js:6:15)

2) .pop returns the last element in the array [3phase]:
   ReferenceError: pop is not defined
       at Context.it (test/index_test.js:18:20)

npm ERR! Test failed.  See above for more details.
$ 

```

pop() 함수를 잘못 써서 에러가 났다. 다음과 같이 pop함수를 바꿔주고 다시 테스트를 실행하면 아래와 같이 된다.

```

1  const assert = require('assert');
2
3  // Naive approach
4  ▼ describe('.pop', () => {
5    ▼ it('returns the last element in the array [naive]', () => {
6      assert.ok(['padawan', 'knight'].pop() === 'knight');
7    });
8  });
9
10 // 3 phase approach
11 ▼ describe('.pop', () => {
12   ▼ it('returns the last element in the array [3phase]', () => {
13     // Setup
14     const knightString = 'knight';
15     const jediPath = ['padawan', knightString];
16
17     // Exercise
18     const popped = jediPath.pop();
19
20     // Verify
21     assert.ok(popped === knightString);
22   });
23 });
24
25

```

```
+ | × bash
$ npm test

> learn-mocha-intro-start@1.0.0 test /home/ccuser/workspace
learn-mocha-learn-mocha-setup-exercise-verify
> mocha test/**/*.test.js

    .pop
      ✓ returns the last element in the array [naive]

    .pop
      ✓ returns the last element in the array [3phase]

  2 passing (5ms)

$
```

어느 둘 중 테스트 접근법이 읽고 수정하기가 더 쉽나? 바로 3단계 접근법이 더 쉽다.

- Teardown (분해)

몇몇 테스트는 teardown 이라고 불리는 4단계를 요구한다. 이 스텝은 테스트를 더욱 더 독립되도록 만든다.

- teardown - 테스트동안 바뀌어지는 어떠한 조건들을 리셋한다.

하나의 테스트는 그것의 다른 테스트에 영향을 끼칠 수 있는 환경을 변화시킬 수 있다. teardown 단계는 다음 테스트가 실행되기 전에 환경을 리셋하기 위해 사용된다.

몇몇 공통된 환경변화는 다음과 같다.

⇒ 파일과 디렉토리의 구조를 변경하는 것

⇒ 파일에 대한 읽기, 쓰기 허가를 변경하는 것

⇒ 데이터베이스 안에 레코드를 수정하는 것

몇몇 케이스(이전 연습과 같은)에서는 teardown 단계가 필요하지않다. 왜냐하면 리셋할 조건이 없기 때문이다.

실습 예제

```

1  const assert = require('assert');
2  const fs = require('fs');
3
4  ▼ describe('appendFileSync', () => {
5      it('writes a string to text file at given path name', () =>
6          ▼ {
7              // Setup
8              const path = './message.txt';
9              const str = 'Hello Node.js';
10
11             // Exercise: write to file
12             fs.appendFileSync(path, str);
13
14             // Verify: compare file contents to string
15             const contents = fs.readFileSync(path);
16             assert.ok(contents.toString() === str);
17
18             // Teardown: delete path
19
20         });
21     });
22
23

```

이번 연습에서는 Node의 파일시스템 라이브러리인 fs 와 assert 모듈을 사용한다.

fs.appendFileSync(path, str) 은 다음 기능을 갖는다.

⇒ 만약 path에 파일이 존재하지 않는다면, 파일을 생성하고 그 안에 string 을 추가한다.

⇒ 만약 path에 파일이 존재하면, 파일의 끝에 string 을 추가한다.

아직 4단계 teardown 을 추가하지 않았다.

```
$ npm test

> learn-mocha-intro-start@1.0.0 test /home/ccuser/workspace/learn-mocha-automate-organize-tests-teardown
> mocha test/**/*.test.js

appendFileSync
  ✓ writes a string to text file at given path name

1 passing (6ms)

$
```

```
project
├─ node_modules
├─ test
├─ index_test.js
├─ message.txt
└─ package.json
```

새로운 message.txt 가 디렉토리에 생성되었다.

이후 한 번더 npm test 를 실행하면 터미널에 다음과 같이 출력한다.

```

$ npm test

> learn-mocha-intro-start@1.0.0 test /home/ccuser/workspace/learn-mocha-automate-organize-tests-teardown
> mocha test/**/*.test.js

appendFileSync
  1) writes a string to text file at given path name

0 passing (8ms)
1 failing

1) appendFileSync writes a string to text file at given path name:

    AssertionError: false == true
      + expected - actual

      -false
      +true

      at Context.it (test/index_test.js:16:12)

npm ERR! Test failed.  See above for more details.
$ 

```

다음과 같이 에러가 뜬다. 왜냐하면 새로운 파일의 생성으로 인해 환경이 변화했기 때문이다.

```
npm ERR! Test failed.  See above for more details.  
$ rm message.txt  
$
```

먼저 터미널에서 생성된 message.txt 를 삭제해주자.

```
▼ describe('appendFileSync', () => {  
  it('writes a string to text file at given path name', () =>  
    ▼ {  
  
      // Setup  
      const path = './message.txt';  
      const str = 'Hello Node.js';  
  
      // Exercise: write to file  
      fs.appendFileSync(path, str);  
  
      // Verify: compare file contents to string  
      const contents = fs.readFileSync(path);  
      assert.ok(contents.toString() === str);  
  
      // Teardown: delete path  
      fs.unlinkSync(path);  
    });  
  });  
});
```

teardown 스텝 안에 다음 과 같은 커맨드를 입력해주자. 이 메서드는 테스트가 끝나기 전에 path에 있는 파일을 삭제할 것이다.

```
$ npm test

> learn-mocha-intro-start@1.0.0 test /home/ccuser/workspace/learn-mocha-automate-organize-tests-teardown
> mocha test/**/*.test.js

  appendFileSync
    ✓ writes a string to text file at given path name

  1 passing (6ms)

$ npm test

> learn-mocha-intro-start@1.0.0 test /home/ccuser/workspace/learn-mocha-automate-organize-tests-teardown
> mocha test/**/*.test.js

  appendFileSync
    ✓ writes a string to text file at given path name

  1 passing (6ms)

$
```

이후 테스트를 두 번 이상 실행하게 되어도 에러가 뜨지 않는 것을 확인할 수 있다.

- Hooks

it 블록 안에 teardown 을 사용하는 것은 테스트를 고립되도록 만들 수 있지만, 의존성을 가지진 못했다.

만약 시스템이 teardown에 도달하기 전에 에러를 마주친다면, 4단계인 teardown 을 실행하지 못할 것이다. 그로 인해 환경 초기화가 불가능 할 수 있다는 것이다. 이전 예시에서 에러는 파일이 생성되고나서 삭제되기 전에 일어날 수 있다. 그렇게 되면 파일은 환경리셋을 하기 위해 삭제되지 않고 그대로 유지되며, 테스트는 false 를 출력해낼 것이다.

이러한 문제를 해결하기 위해 Mocha 는 Hook 기능을 제공한다. hook 은 특정한 이벤트가 일어날 때 실행되는 하나의 코드이다. 혹은 setup, teardown 단계에서의 조건들을 설정하거나 리셋하기 위해 사용되어 진다. 이러한 hook 코드는 it 블록이 아닌 describe 블록 안에 쓰여진다.

```
describe('example', () => {  
  
  afterEach(() => {  
    // teardown goes here  
  });  
  
  it('.sample', () => {  
    // test goes here  
  });  
});
```

이 예시에서는 afterEach()는 각 it 블록이 실행되고나서 불려지게 된다.

모카 라이브러리에는 before(), beforeEach(), after()과 같은 다른 hook들이 있다.

실습 예제

```
× index_test.js

1  const assert = require('assert');
2  const fs = require('fs');
3
4  ▼ describe('appendFileSync', () => {
5      const path = './message.txt';
6
7      it('writes a string to text file at given path name', () =>
8  ▼ {
9      // Setup
10     const str = 'Hello Node.js';
11
12     // Exercise: write to file
13     fs.appendFileSync(path, str);
14
15     // Verify: compare file contents to string
16     const contents = fs.readFileSync(path);
17     assert.ok(contents.toString() === str);
18
19     // Teardown: delete path
20     fs.unlinkSync(path);
21
22     });
23 });
24
25
```



```
+ | × bash | ↗
$ npm test

> learn-mocha-intro-start@1.0.0 test /home/ccuser/workspace/learn-mocha-automate-organize-tests-hooks
> mocha test/**/*.test.js

appendFileSync
  1) writes a string to text file at given path name

0 passing (7ms)
1 failing

1) appendFileSync writes a string to text file at given path name:
  TypeError: fs.readileSync is not a function
    at Context.it (test/index_test.js:16:25)

npm ERR! Test failed.  See above for more details.
$
```

위 코드를 test 하면 이러한 결과를 낸다. 16번째 줄에 TypeError 로 오타가 발견되었다. 이 결과 teardown 단계가 실행되지 않았고 Exercise 단계에서 생성된 파일이 삭제 되지 않고 그대로 유지되고 있다. rm message.txt 로 해당 파일을 삭제하자.

```

4 ▼ describe('appendFileSync', () => {
5     const path = './message.txt';
6
7     ▼ afterEach(() => {
8         fs.unlinkSync(path);
9     });
10
11     it('writes a string to text file at given path name', () =>
12     ▼ {

```

파일 삭제 후 describe 함수 안에 afterEach() 함수를 넣어주고 블록안에 teardown 단계에 실행되는 코드를 넣어준다.

이렇게 되면 테스트 중간에 오류가 나더라도 teardown 단계까지 접근할 수 있다. 즉, 테스트 중간에 생성된 파일이 오류가 있더라도 제거되어 환경을 초기화시킬 수 있다는 것이다.

• Review

- ⇒ npm을 활용한 Mocha 설치를 배웠다.
- ⇒ describe() 와 it() 을 사용하여 테스트를 구성하는 법을 배웠다.
- ⇒ 테스트 4단계로 너의 테스트가 독립적이며, 표현력이 깃들 수 있도록 하는 방법을 배웠다.
- ⇒ Hook 을 통해 테스트가 믿을만하게 만들 수 있다.
- ⇒ assert.ok() 를 사용하여 단언을 작성한다.