Quest

우아한테크코스 알고리즘 스터디

Week 2. 큐와 덱

이번 주차의 목표

- ✓ 이번 주차에서는 무려 자료구조를 두 종류나 구현해야 합니다. 시간이 촉박할 거에요.
- ✓ 그래서 이번 주의 문제셋은 난이도를 다소 낮춰서 올려 두었습니다.
- ✓ 여러분에게 정해 드리는 목표는 **A, B번 문제의 해결**입니다. 큐와 덱을 구현하실 수 있는 방법을 알 수 있다면 충분하다고 생각합니다.
- ✓ **C, D, E번** 문제는 시간이 남는다면 풀어 보세요.

큐란

- ✓ 큐를 설명하기에 앞서, 지난 주차에 배웠던 스택을 돌아봅시다.
- ✓ 스택의 특징으로는 자료를 꺼내고 넣는 곳이 한 곳(back)이었고, 이에 따라 나중에 들어간 데이터일 수록 일찍 나오는 것이 특징이었습니다.
- ✓ 마지막에 들어간 데이터가 가장 먼저 빠져나가는 스택의 방식을 **후입선출(LIFO, Last In First Out)**이라 부릅니다.

큐란

- ✓ 이번 주차에서 다루는 큐, 덱 또한 스택과 마찬가지로 자료구조의 끝에서만 데이터를 넣고 뺄 수 있는 자료구조입니다.
- ✓ 다만 큐의 경우 데이터가 들어가는 곳은 back으로 같지만, 나오는 곳이 front로 다릅니다. 그림을 참고해 주세요.
- ✓ 가장 먼저 들어간 데이터가 가장 먼저 빠져나가는 큐의 방식을 **선입선출(FIFO, First In First Out)**이라 부릅니다.
- ✓ 큐는 게임 대기열, 차선이 하나인 터널 등에 비유할 수 있겠습니다.

덱이란

- ✓ 덱은 양쪽에서 삽입과 삭제가 가능한 자료구조입니다.
- ✓ 스택의 특징과 큐의 특징이 합쳐진 자료구조이기도 합니다.
- ✓ 덱의 경우 자료를 넣을 수 있는 곳/뺄 수 있는 곳이 각각 두 곳이기 때문에, 각각의 연산을 구별하는 경우가 많습니다. C++의 경우 덱 자료구조를 STL 라이브러리에서 제공하는데, 자료를 넣는 명령어는 그 위치에 따라 `push_back()` 또는 `push_front()` 로 구별하여 사용합니다.

큐와 덱을 구현하기에 앞서...

- ✓ 올바르게 구현된 큐와 덱 모두 양쪽에서 넣고 빼는 연산의 시간복잡도가 모두 O(1)입니다.
- ✓ Python과 C++ 등의 언어에서는 각각 collections와 STL에서 큐와 덱에 관련된 자료구조를 제공합니다.
- ✓ 하지만 JavaScript의 경우 그렇지가 않을 뿐더러, 스택과 같이 JavaScript의 Array를 사용하여 그대로 연산을 구현하는 방법은 사용할 수 없습니다.
- ✓ 왜 그런지는, 다음 페이지에서 알아 봅시다.

큐와 덱을 구현하기에 앞서...

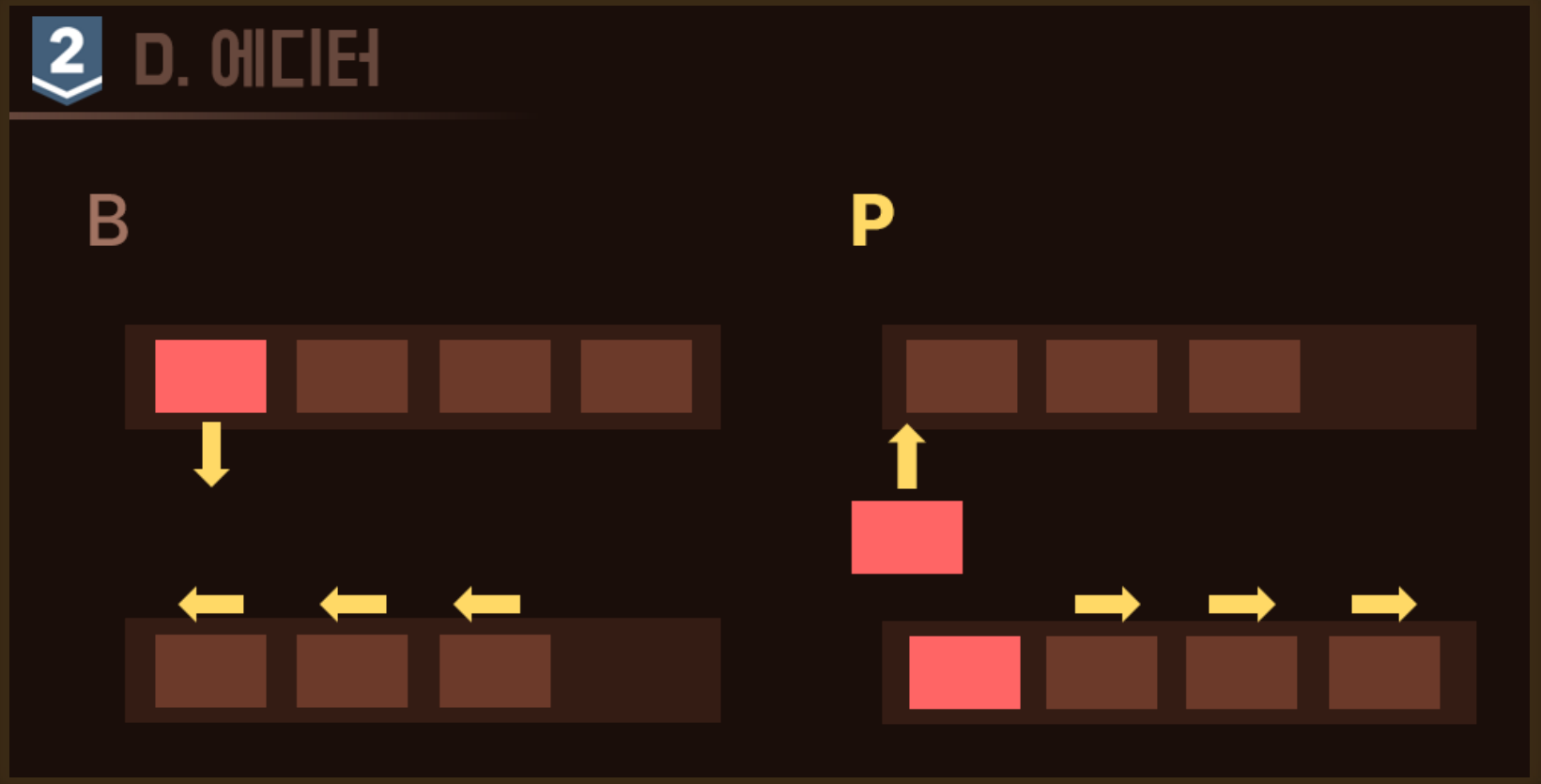
- ✓ 붉은색으로 표시된 부분이 JavaScript의 Array를 그대로 사용할 경우 문제가 일어날 수 있는 연산들입니다. 왜 그럴까요?

큐와 덱을 구현하기에 앞서...

- ✓ Array를 이용하여 구현하였기 때문에, Array의 back(가장 오른쪽)에서 데이터를 넣고 빼는 것은 문제가 되지 않습니다. 해당되는 데이터 하나만을 옮기는 것으로 연산이 가능하기 때문입니다. 시간복잡도는 $O(1)$입니다.
- ✓ 하지만 Array의 front(가장 왼쪽)에서 데이터를 넣고 빼기 위해서는 나머지 데이터들을 모두 옮기는 과정이 동반되기 때문에 **모든 데이터를 옮겨야 하는 최악의 상황이 옵니다.** 이 경우 시간복잡도는 $O(N)$으로, 효율성이 좋지 못 한 방법이 됩니다.
- ✓ 알고리즘에서 `Array.prototype.unshift`와 `Array.prototype.shift`가 기피되는 이유입니다.
- ✓ 입력 데이터의 크기가 충분히 작다면 위의 메서드를 사용하여도 무방하지만, 데이터가 커질 경우에는 정해진 시간을 초과할 수 있으므로, 이를 대비하여 더 효율적인 방법을 알아 두는 것이 좋습니다.

큐와 덱을 구현하기에 앞서...

✓ 저번 주차의 D번 문제에서도 이와 비슷한 그림을 첨부해 드린 적이 있었습니다.



큐와 덱을 구현하기에 앞서...

- ✓ JavaScript로 큐와 덱을 구현할 경우 문제되는 점은, 바로 관련이 없는 데이터들을 모두 옮겨야 한다는 점이었습니다. 데이터를 넣거나 뺐을 때 관련이 없는 데이터들을 이동시키지 않으면서 구현할 수 있는 방법은 없을까요?

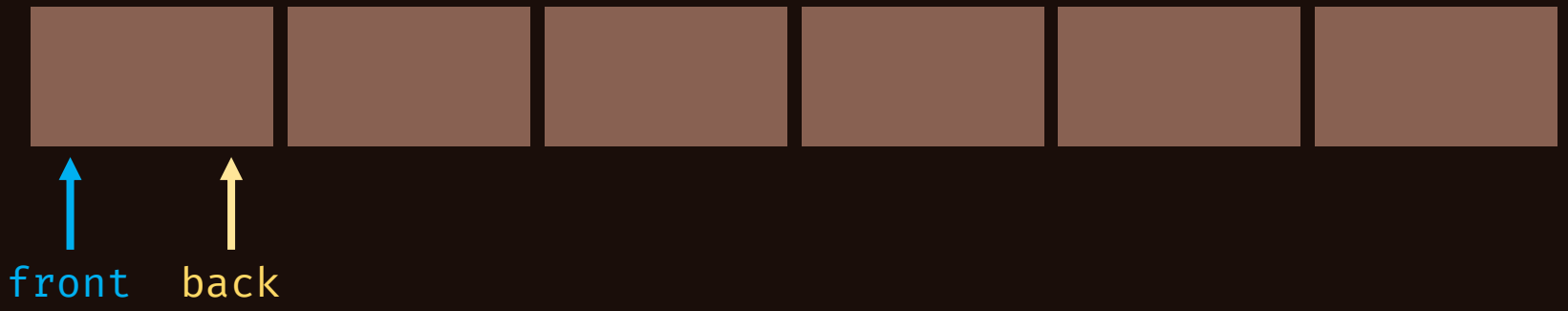
- ✓ 고정 크기의 배열을 이용하여 이를 실현시킬 수 있는 구현 방법을 알려드리겠습니다.

큐의 구현

- ✓ 큐의 구현을 위해, 아래와 같은 고정 크기의 배열을 준비했습니다.
- ✓ 명령어가 문제에서 **N**개 주어진다면, 여러분은 최소한 크기 **N**의 고정 배열을 준비물로 가져오셔야 합니다.
- ✓ 최악의 경우 데이터를 저장하는 연산만 **N**번 주어질 테니까 말이죠!

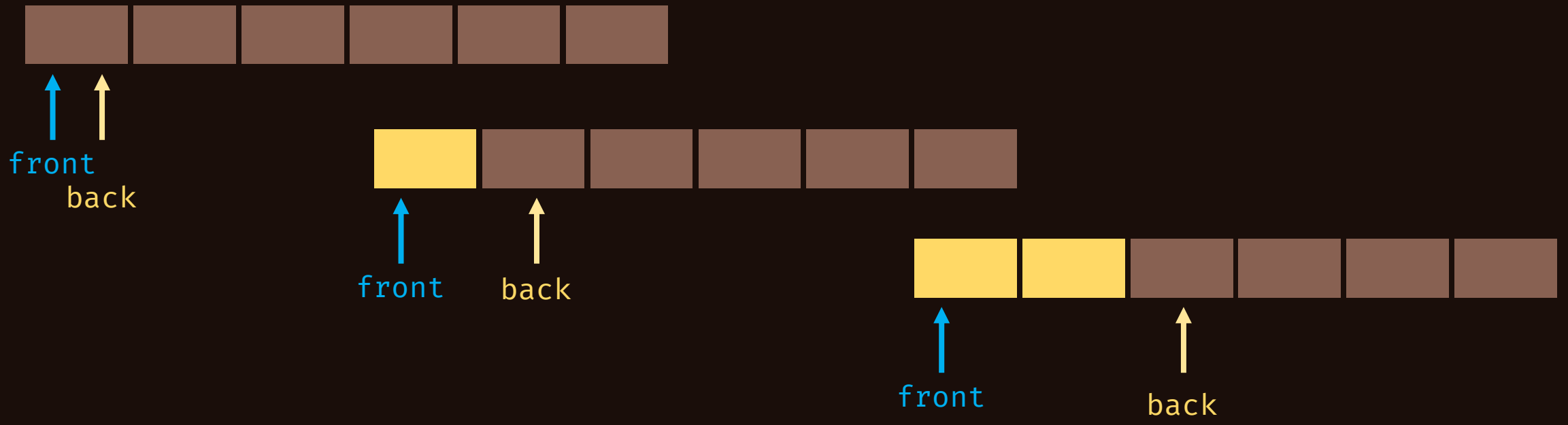
큐의 구현

- ✓ 큐의 back과 front의 위치를 관리할 수 있도록, 두 변수를 별도를 두어 관리할 것입니다.
- ✓ back을 가리키는 변수, front를 가리키는 변수의 초깃값은 모두 `0`입니다. 두 변수의 값의 의미는 아래와 같습니다.
- ✓ back의 경우 이번에 새롭게 **데이터를 넣게 될 경우 어느 인덱스에 넣을 지**를 의미합니다.
- ✓ front의 경우 기존 데이터를 빼낼 경우 **어느 인덱스의 데이터를 뺄 지**를 의미합니다.
- ✓ 처음으로 데이터를 넣을 경우 인덱스 `0`에 데이터를 넣을 것이고, 처음으로 데이터를 뺄 경우 인덱스 `0`에 있는 데이터부터 차례대로 뺄 것입니다. 따라서 두 변수의 초깃값은 `0`입니다.



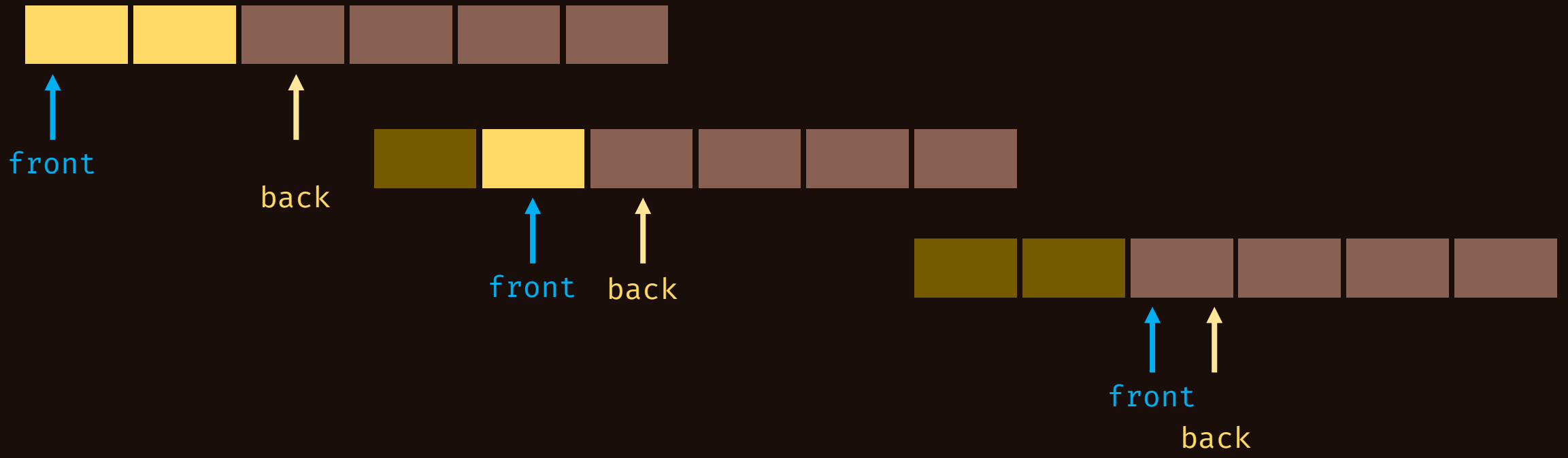
큐의 구현

- ✓ 이제 push 연산이 어떻게 진행되는 지를 알아 봅시다.
- ✓ push 연산이 진행되면, 먼저 back에 해당하는 위치에 데이터를 넣습니다.
- ✓ 그 다음에는 다음 push 연산이 들어올 것에 대비하여 push의 값을 1 증가시켜 주면 됩니다. 오른쪽으로 차례대로 데이터가 쌓이기 때문입니다.



큐의 구현

- ✓ pop 연산도 push 연산과 비슷합니다.
- ✓ pop 연산이 진행되면, 먼저 front에 해당하는 위치의 **데이터를 제거한 셈 치고**, front 값을 1 증가시켜 줍니다.
- ✓ 이렇게 구현하면 데이터를 실제로 제거하지 않으므로 모든 데이터를 이동시켜야 하는 대참사가 발생하지 않습니다.

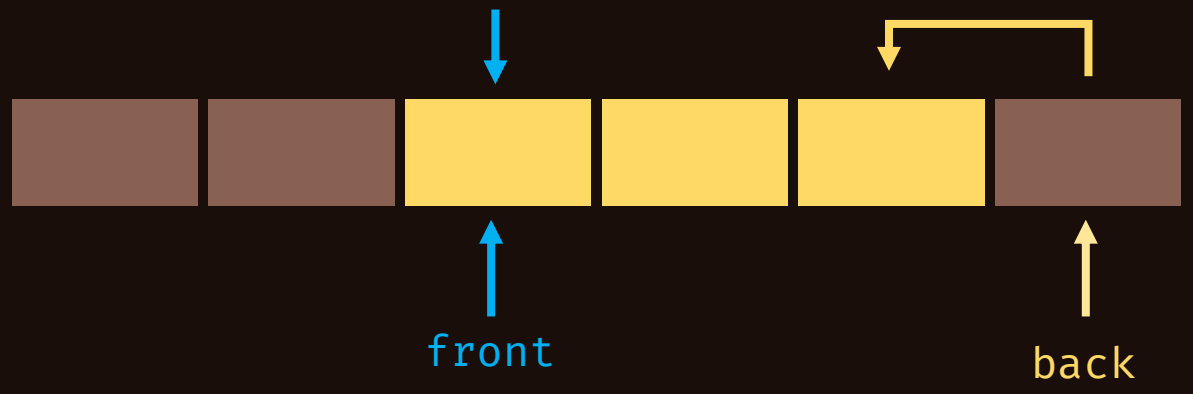


큐의 구현

- ✓ 만약 pop 연산을 진행해야 하는데 front와 back 값이 같은 경우에는 큐가 비어 있음을 의미하므로, front 값을 증가시키는 대신 별도의 예외 처리를 하시면 됩니다. 알고리즘에서는 `-1`을 대신 출력하는 등 문제의 요구 사항을 따르시면 되겠습니다.
- ✓ 비슷하게 접근하면, 현재 큐에 담긴 데이터의 크기는 `back - front` 라는 것 또한 알 수 있게 됩니다.
- ✓ 고정 크기의 배열을 이용하고, front와 back을 의미하는 두 변수를 관리하는 것으로 데이터를 직접 추가/삭제했을 때의 비효율적인 시간복잡도를 개선하는 작업은 큐를 구현함에 있어서 가장 기본적인 방법입니다.
- ✓ 큐를 구현하는 방식은 다양하기 때문에 인터넷에 직접 큐를 구현하는 방법을 검색하셨을 때는 제가 설명한 방법과 다를 수 있습니다.

큐의 구현

- 앞선 방법에서 큐에 데이터를 넣거나 뺄 때는 먼저 해당 위치에 원하는 작업을 수행하고, 그 다음 인덱스를 한 칸 뒤로 옮겼습니다.
- 이를 이용하면 큐의 양끝의 데이터를 출력하여야 할 때 어느 인덱스의 데이터를 출력해야 할 지도 유추할 수 있을 것입니다.



큐의 구현

```
class Queue {
  constructor(size) {
    this.queue = new Array(size);
    this.front = 0;
    this.back = 0;
  }

  push(value) {
    this.queue[this.back] = value;
    this.back += 1;
  }

  pop() {
    if (this.isEmpty()) {
      return -1;
    }

    const popped = this.queue[this.front];
    this.front += 1;

    return popped;
  }

  getSize() {
    return this.back - this.front;
  }

  isEmpty() {
    return this.getSize() === 0 ? 1 : 0;
  }

  getFront() {
    if (this.isEmpty()) {
      return -1;
    }

    return this.queue[this.front];
  }

  getBack() {
    if (this.isEmpty()) {
      return -1;
    }

    return this.queue[this.back - 1];
  }
}
```

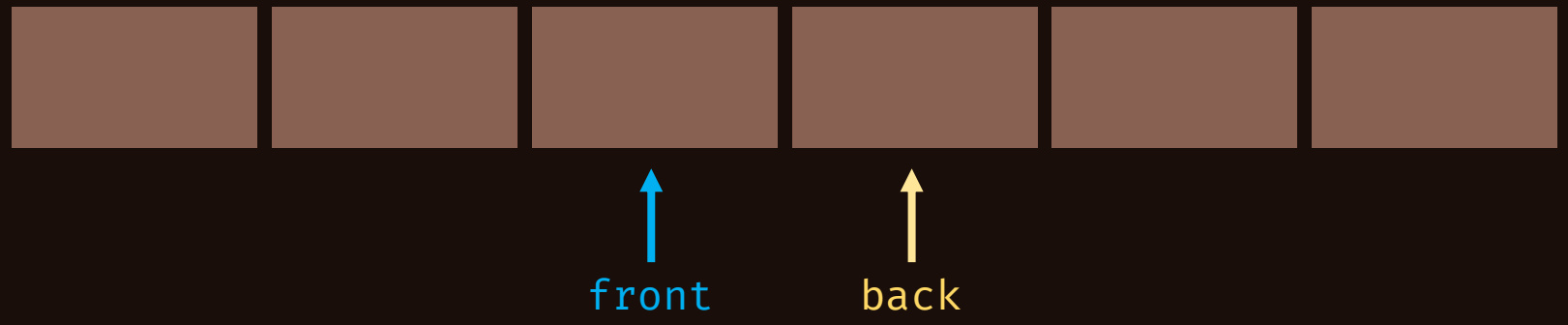
✓ 큐를 구현한 예시 코드입니다.

덱의 구현

- ✓ 이번에는 덱을 구현해 봅시다. 역시 고정 크기의 배열을 사용해 보겠습니다.
- ✓ 참고로 이번 방법은 다소 야매인 점을 알려드립니다. 인터넷에 검색해 보았습니다만 너무 많은 자료들이 JavaScript의 `Array.prototype.unshift`와 `Array.prototype.shift`를 사용했더군요. 어쩔 수 없이 저는 저만의 방법을 찾아야 했습니다.
- ✓ 이 구현 방법에서는 명령어가 문제에서 **N**개 주어진다면, 여러분은 최소한 크기 **2N**의 고정 배열을 준비물로 가져오셔야 합니다.
- ✓ 최악의 경우 데이터를 front에 저장하는 연산만 **N**번 주어지거나, back에 저장하는 연산만 **N**번 주어질 테니까요! 이 부분은 다음 페이지에서 더 자세히 설명하겠습니다.

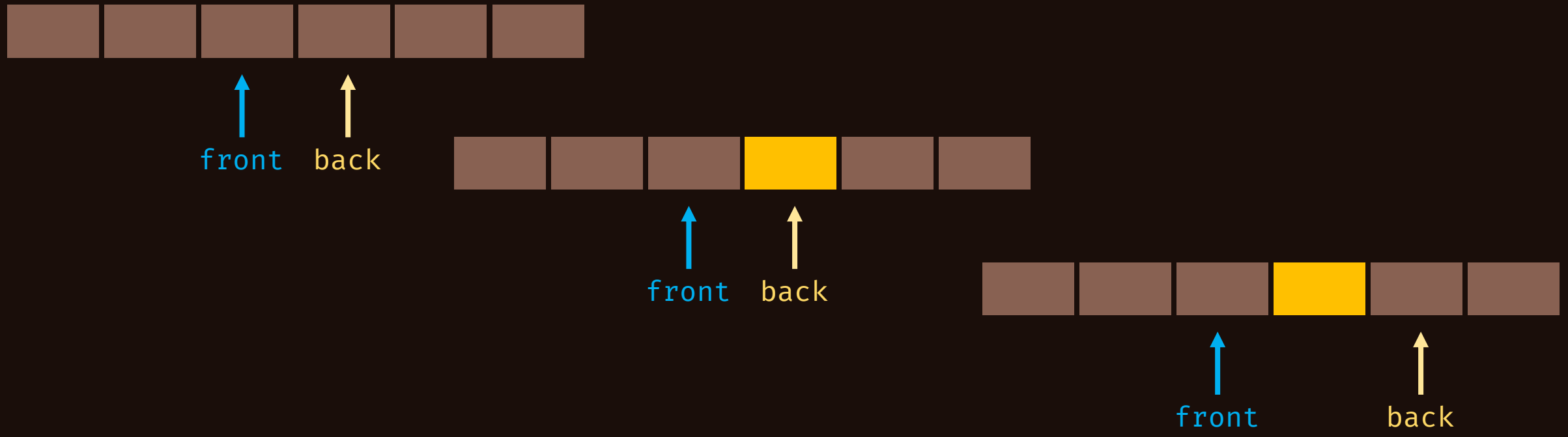
덱의 구현

- ✓ 덱도 큐와 마찬가지로 back, front 두 변수를 두도록 하겠습니다. 다만 이번에는 두 변수가 배열의 왼쪽 끝부분에 위치하는 것이 아닌, 가운데 위치에 위치하게 됩니다.
- ✓ 이는 front에 데이터를 계속해서 넣을 경우 저장되는 데이터들이 점차 왼쪽으로 가게 되고, back에 넣을 경우 저장되는 데이터들이 점차 오른쪽으로 가게 되기 때문입니다. 따라서 큐와는 다르게 공간이 양쪽으로 마련되어야 하며, 이것이 두 변수의 초깃값을 가운데에 두는 이유입니다.
- ✓ 큐와는 다르게 back과 front의 인덱스는 한 칸의 간격을 둡니다. 이는 back에서 값을 추가하는 경우, front에서 값을 추가하는 경우를 서로 겹치지 않게 하기 위함입니다.



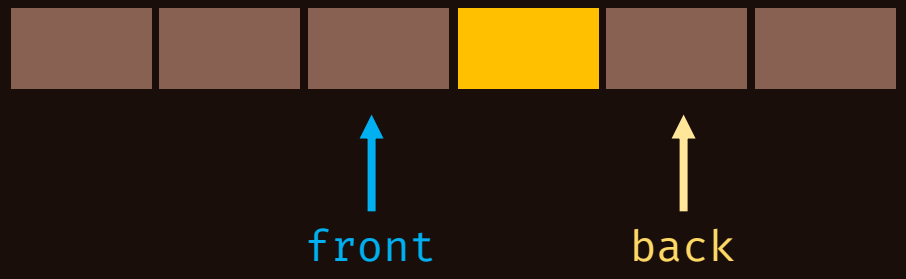
덱의 구현

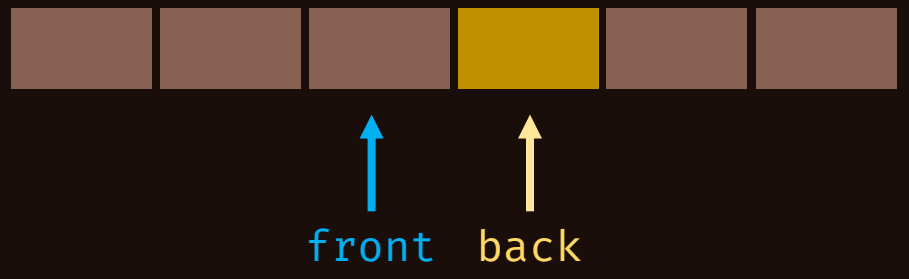
- ✓ push_back 연산이 어떻게 진행되는지 알아 봅시다.
- ✓ push_back 연산을 사용할 경우, 먼저 back이 가리키는 인덱스에 추가하고자 하는 값을 넣은 후 back 값을 1 증가시키면 됩니다.



덱의 구현

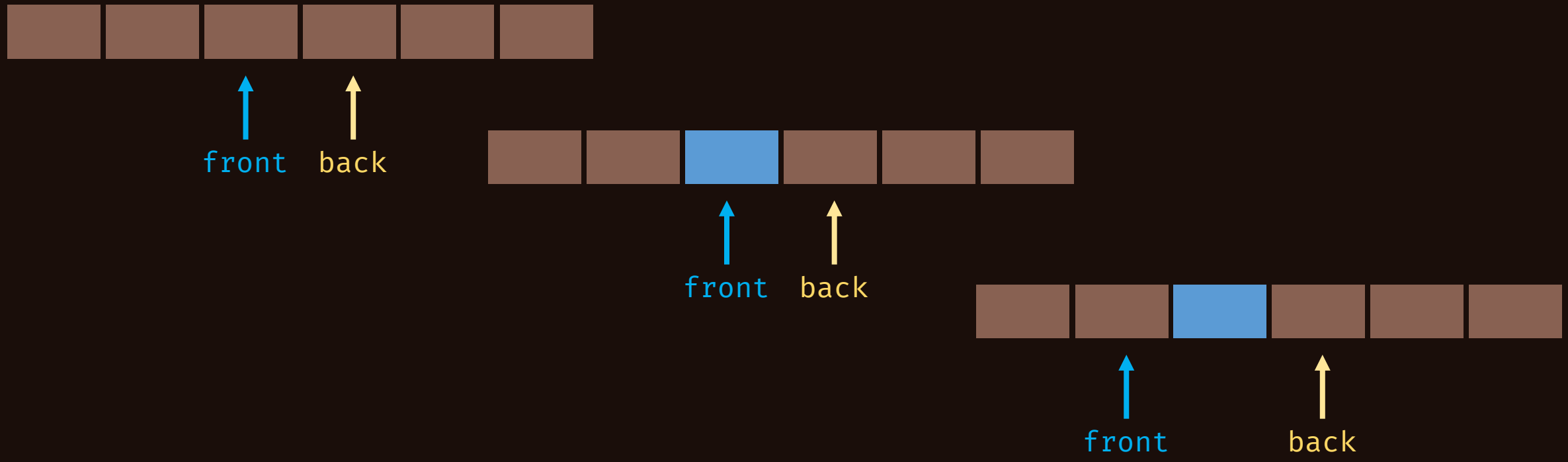
- ✓ pop_back 연산이 어떻게 진행되는지 알아 봅시다.
- ✓ pop_back 연산을 사용할 경우, back 값을 1 감소시키면 끝입니다. 감소시킨 후의 인덱스에 해당하는 값은 지워졌다고 생각해 주세요.





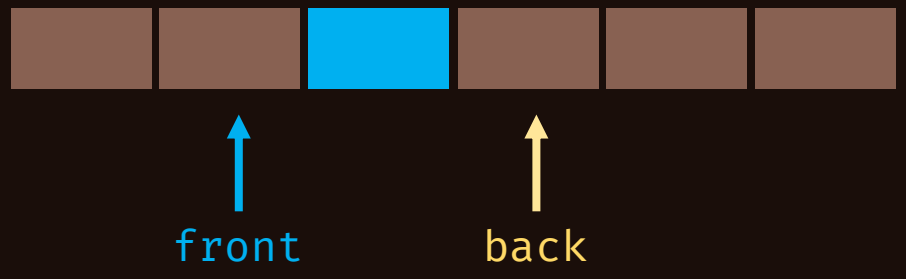
덱의 구현

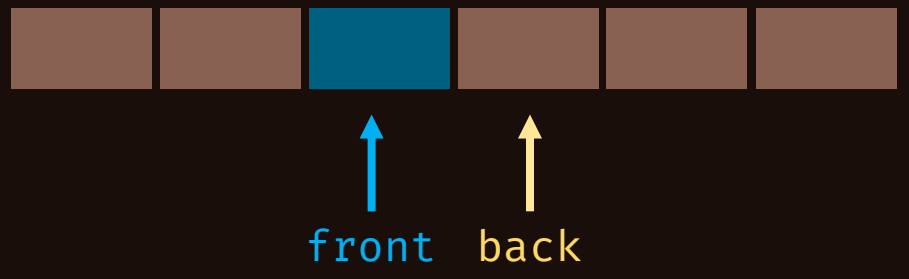
- ✓ push_front, pop_front 연산의 경우도 push_back, pop_back 연산과 비슷합니다. 다만 방향이 다를 뿐이죠. 여기서부터는 복붙이 될 것 같습니다.
- ✓ push_front의 연산을 사용할 경우, 먼저 front가 가리키는 인덱스에 추가하고자 하는 값을 넣은 후 front 값을 1 감소시키면 됩니다.



덱의 구현

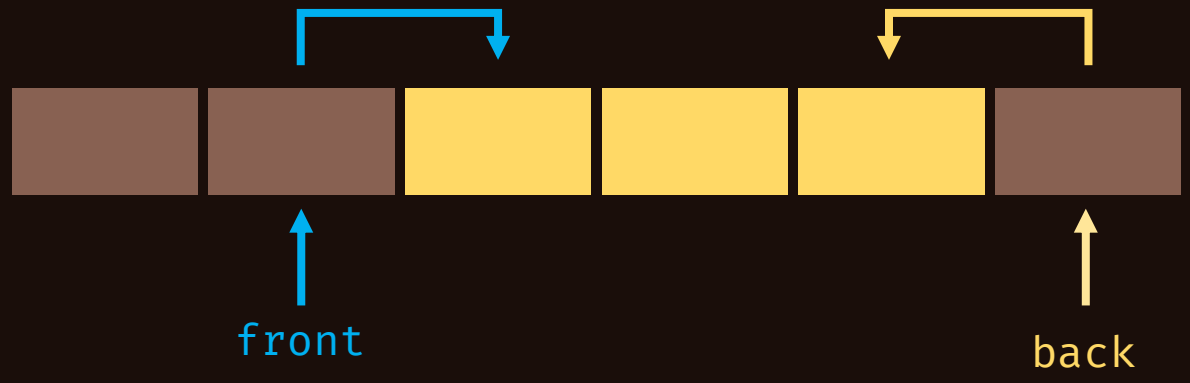
- ✓ pop_front 연산이 어떻게 진행되는지 알아 봅시다.
- ✓ pop_front 연산을 사용할 경우, front 값을 1 증가시키면 끝입니다. 증가시킨 후의 인덱스에 해당하는 값은 지워졌다고 생각해 주세요.





덱의 구현

- ✓ 덱에서 어느 방향으로 값을 추가하든, 인덱스의 위치에 값을 추가한 후 인덱스의 위치를 한 칸 옮기므로, 덱의 왼쪽 끝의 값과 오른쪽 끝의 값을 구하는 방법 역시 유추하실 수 있을 것이라 생각됩니다.



덱의 구현

- ✓ 덱의 데이터의 크기는 `back - front - 1`과 일치합니다.
- ✓ 초기에 덱이 비어 있었을 때, back의 값이 front의 값보다 1 더 크며, 데이터를 새로 추가할 때마다 back과 front의 거리가 1씩 멀어지며, 반대로 데이터를 제거할 때는 거리가 1씩 가까워짐을 생각해 주시면 될 것 같습니다.

덱의 구현

✓ 덱을 구현한 예시 코드입니다.

```
class Deque {
  deque = new Array(2_000_001);
  frontIndex = 1_000_000;
  backIndex = 1_000_001;

  pushFront(item) {
    this.deque[this.frontIndex] = item;
    this.frontIndex -= 1;
  }

  pushBack(item) {
    this.deque[this.backIndex] = item;
    this.backIndex += 1;
  }

  popFront() {
    if (this.getSize() === 0) {
      return -1;
    }

    this.frontIndex += 1;
    return this.deque[this.frontIndex];
  }

  popBack() {
    if (this.getSize() === 0) {
      return -1;
    }

    this.backIndex -= 1;
    return this.deque[this.backIndex];
  }

  getFront() {
    if (this.getSize() === 0) {
      return -1;
    }

    return this.deque[this.frontIndex + 1];
  }

  getBack() {
    if (this.getSize() === 0) {
      return -1;
    }

    return this.deque[this.backIndex - 1];
  }

  getSize() {
    return this.backIndex - this.frontIndex - 1;
  }

  isEmpty() {
    return this.getSize() === 0;
  }
}
```

이제 문제를 풀어볼 시간입니다

- ✓ 이로써 큐와 덱을 직접 구현하는 방법을 알아보았습니다.
- ✓ 처음으로 큐와 덱을 접하셨다면 많이 생소하실 것이라고 생각합니다. 구현에 어려움을 겪으신다면 우선 라이브러리에서 제공해 주는 자료구조를 사용해 보시고, 익숙되시면 직접 구현해 보시는 것도 좋은 방법이라 생각됩니다.
- ✓ 아쉽지만 JavaScript는 이 모든 과정을 외워야만 합니다. 이번 연습에서는 많은 문제를 풀지 않으셔도 좋으니 이 구현 방법에 대해 오랜 시간을 두고 이해하는 시간을 가지시면 좋을 것이라고 생각합니다.

이제 문제를 풀어볼 시간입니다

- ✓ 이제 A번 문제와 B번 문제를 풀어보셔도 좋습니다.
- ✓ 선정해 드린 기본 큐, 덱 문제들은 기존 튜토리얼 문제에 데이터의 크기가 매우 커진 버전의 문제입니다. 따라서, 모든 연산이 $O(1)$의 시간복잡도를 가지도록 구현하셔야 통과하실 수 있습니다.
- ✓ 화이팅입니다! 이해가 안 되는 점이 있으시다면, 꼭 슬랙에 질문을 올려주셨으면 좋겠습니다.
- ✓ 다음 페이지부터는 C번~E번 문제에 대한 풀이를 간략하게 정리하겠습니다.

3 C. 프린터 큐

- ✓ 문제에서 제시한대로 정직하게 구현하면 되는 시뮬레이션 문제입니다.
- ✓ 기존에는 큐에 하나의 원소 (예를 들면 정수) 를 저장했다면, 이번에는 문서의 중요도와 해당 문서가 우리가 찾고자 하는 그 문서인지의 여부를 같이 저장해야 합니다. 예를 들면, `[4, true]` 와 같이 말이죠.
- ✓ 큐의 front에서 원소를 꺼내고, 해당 문서가 인쇄되어야 하는 문서인지 확인한 후, 인쇄되어야 하는 문서이면 꺼낸 원소를 넣지 않고, 인쇄되어야 하는 문서가 아니라면 다시 큐의 back에 넣어주는 연산을 반복하시면 됩니다.
- ✓ 이 과정을 우리가 원하는 문서가 인쇄될 때까지 반복해 주면 됩니다.

3 C. 프린터 큐

- ✓ 큐의 front에서 원소를 꺼냈을 때, 해당 문서가 인쇄되어야 하는 지의 여부를 어떻게 판단해야 하는지는 어떻게 판단할까요?
- ✓ 바로 각 중요도의 문서가 몇 개 있는지를 저장하는 배열을 두면 됩니다. 예를 들어 중요도가 1인 문서가 3개, 중요도가 2인 문서가 0개, 중요도가 3인 문서가 1개일 경우 [0, 3, 0, 1] 과 같이 나타낼 수 있습니다.
- ✓ 문서를 꺼낼 때마다 해당 배열을 처음부터 끝까지 탐색하면서 해당 문서보다 중요도가 높은 문서가 하나라도 있는 지 일일이 찾아보시면 됩니다. 시간이 굉장히 오래 걸릴 것 같아보이지만, 중요도는 아무리 커봐야 9밖에 안 되기 때문에 풀이에 크게 영향을 주지 않습니다.
- ✓ 이해를 돕기 위해 예를 하나 들고, 이를 시뮬레이션 해 보는 것으로 풀이는 마무리 하겠습니다.

3 C. 프린터 큐

입력

```
1
4 2
2 1 2 3
```

풀이에서 설명한 중요도 배열입니다

중요도

1	2	1
1	2	3

문제에서 요구한,
인쇄되어야 하는 문서인지의 여부입니다

queue

2	false	1	false	2	true	3	false

문서의 중요도입니다

인쇄되어야 하는 문서입니다

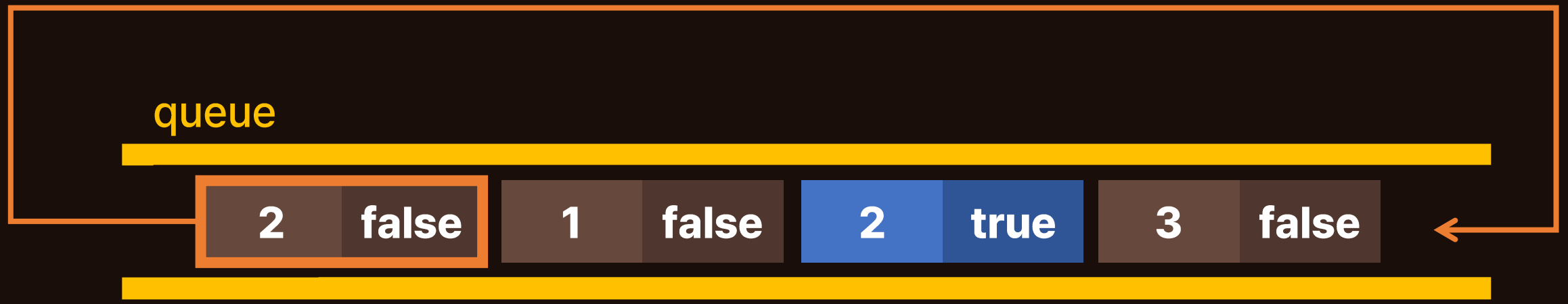
3 C. 프린터 큐

중요도

1	2	1
1	2	3

입력

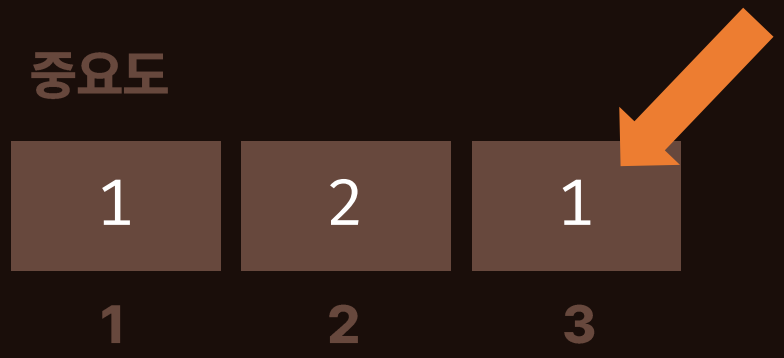
```
1
4 2
2 3 2 1
```



우선순위가 더 높은 문서가 있군요, 인쇄되지 않고 대기열의 맨 뒤로 가겠네요?

3 C. 프린터 큐

중요도



입력

```
1
4 2
2 3 2 1
```

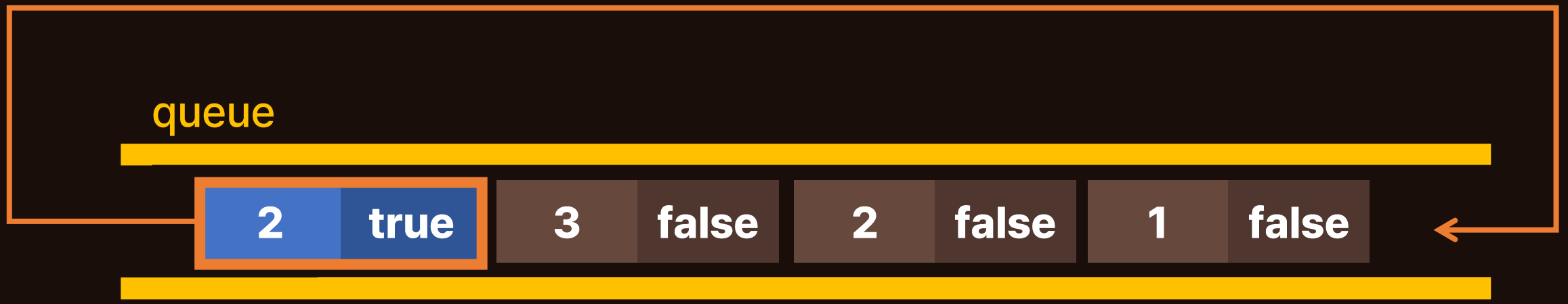
3 C. 프린터 큐

중요도

1	2	1
1	2	3

입력

```
1
4 2
2 3 2 1
```



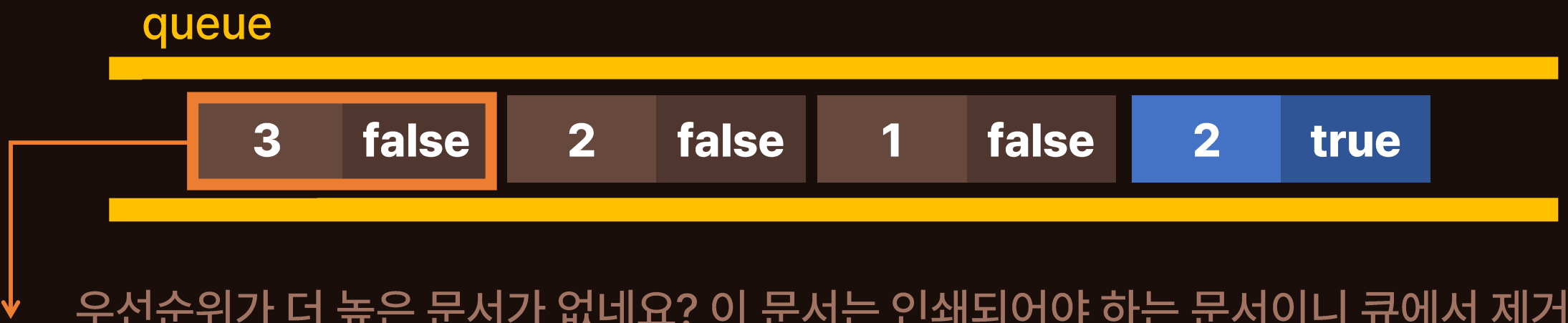
3 C. 프린터 큐

입력

```
1
4 2
2 3 2 1
```

중요도

1	2	1
1	2	3



우선순위가 더 높은 문서가 없네요? 이 문서는 인쇄되어야 하는 문서이니 큐에서 제거해 줍시다. 첫 번째 문서 인쇄 완료!

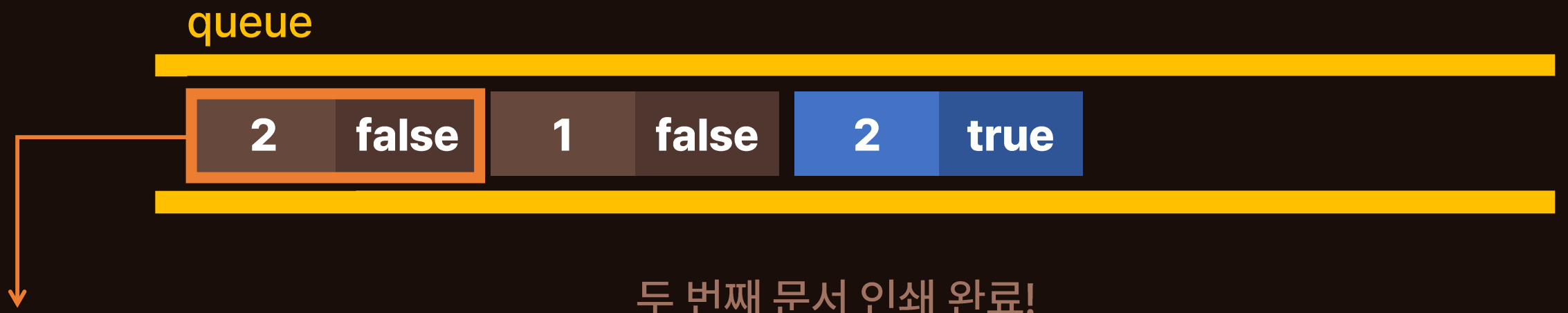
3 C. 프린터 큐

중요도

1	2	0
1	2	3

입력

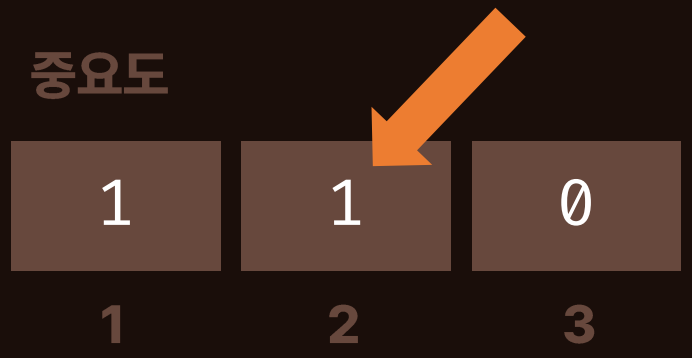
```
1
4 2
2 3 2 1
```



두 번째 문서 인쇄 완료!

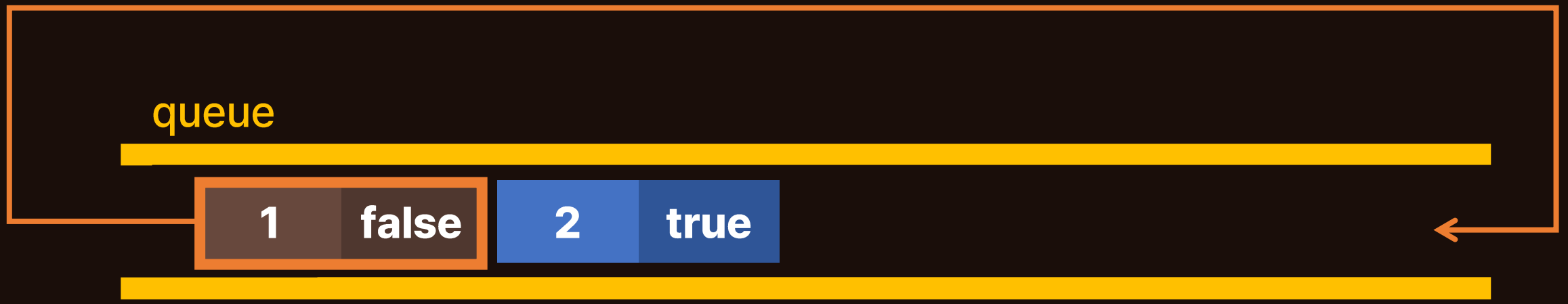
3 C. 프린터 큐

중요도



입력

```
1
4 2
2 3 2 1
```



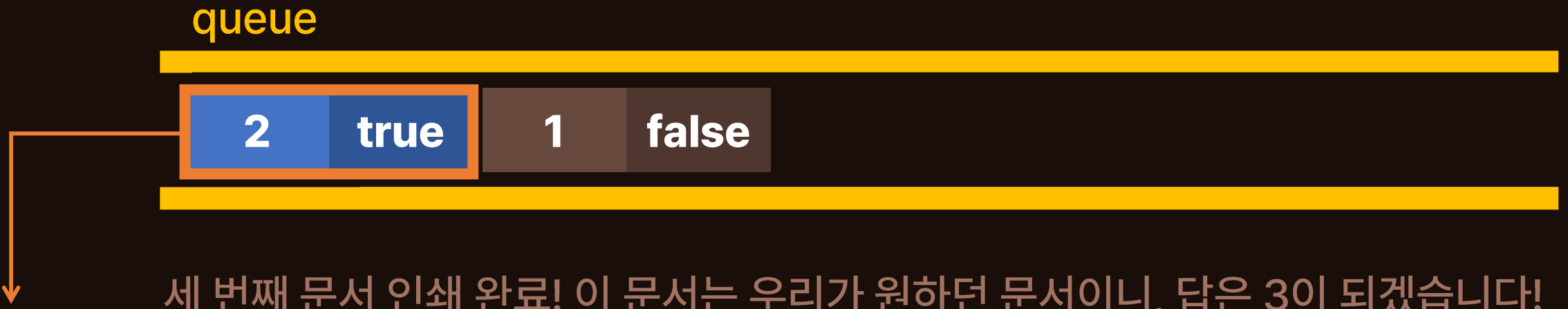
C. 프린터 큐

중요도

1	1	0
1	2	3

입력

```
1
4 2
2 3 2 1
```



세 번째 문서 인쇄 완료! 이 문서는 우리가 원하던 문서이니, 답은 3이 되겠습니다!

3 C. 프린터 큐

- ✓ 이렇게 시뮬레이션을 돌리면 문제를 해결할 수 있습니다.

- ✓ 테스트 케이스 당 시간복잡도는 $O(N^2)$ 입니다. 대기열에 문서가 N개 있고, 항상 최악의 상황을 마주해서 마지막 문서인 경우에만 인쇄되는 경우, 마지막 문서가 인쇄되는 횟수가 N회가 되기 때문입니다. N이 최대 100이니 순식간에 문제를 풀 수 있습니다.

- ✓ 테스트 케이스의 수가 문제에서 주어지지 않은 것은 이 문제가 옛날 문제이기 때문입니다. 보통 이런 경우 테스트 케이스의 수는 시간복잡도에 영향을 주지 않는 선에서 주어집니다. 실제 코딩 테스트에서는 모든 정보가 정확하게 주어지니, 그냥 그러려니 하고 풀어 주셔도 될 것 같습니다!

3 D. 풍선 터뜨리기

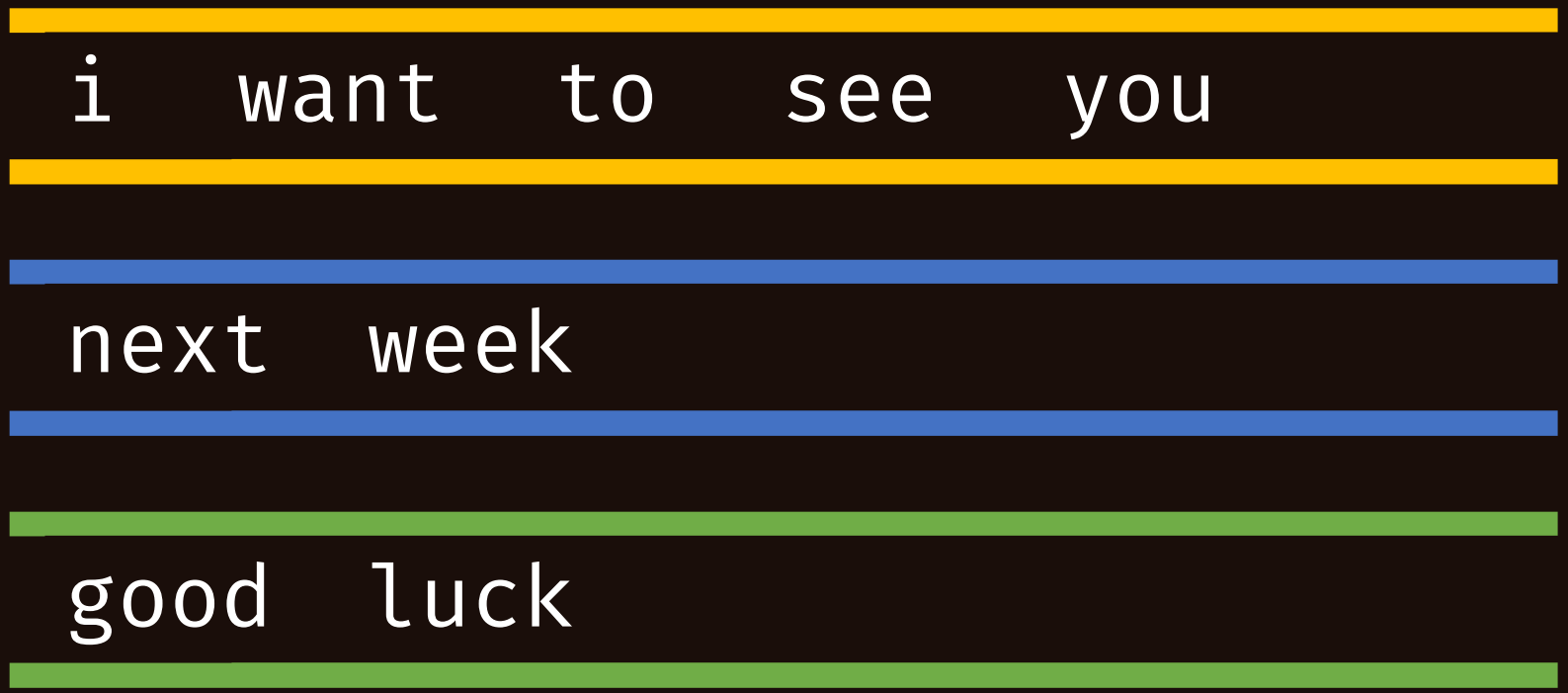
- ✓ 이 문제도 프린터 큐 문제처럼 계속해서 값들을 돌려가며 푸는 문제입니다.
- ✓ 풍선에 적힌 수만큼 돌아가면서 세되, 이미 터진 풍선은 세지 않는 것이 문제의 핵심입니다.
- ✓ 풍선의 수가 양수일 경우와 음수일 경우 세는 방향이 다르므로, 덱을 사용하면 이 문제를 해결할 수 있습니다.
- ✓ 원하는 방향으로 덱에 있는 원소를 돌리면서 풍선을 세면 되고, 터진 풍선은 덱에서 빼내는 방식으로 구현하면 문제를 해결할 수 있습니다. 이전 문제와 비슷하므로 자세한 설명은 생략합니다.
- ✓ 시간복잡도는 $O(N^2)$ 입니다. 최악의 경우 $N = 1\,000$으로, 제한 시간 내에 충분히 문제를 해결할 수 있습니다.

2 E. 앵무새

- ✓ 앵무새 여러 마리가 외치는 단어들이 각 문장의 순서와 일치하는 지를 확인하는 문제입니다.
- ✓ 생각해 보면, 불가능한 문장이라는 것은 앵무새가 한 마리라도 자신에게 주어진 문장을 처음부터 순서대로 말하지 않는다는 것을 의미합니다.
- ✓ 각 앵무새는 정해진 하나의 문장을 순서대로 외치는 것이 예상되니, 각 앵무새가 말하는 문장을 큐로 저장해 두고, 앵무새가 한 단어를 말할 때마다 큐에서 빼는 것은 어떨까요?
- ✓ 이 문제에서는 **같은 단어가 두 번 이상 주어지지 않습니다**. 따라서 어떤 단어를 외쳤을 때, 어떤 앵무새가 외친 단어인지 정확하게 식별할 수 있습니다. 이를 이용해 봅시다.

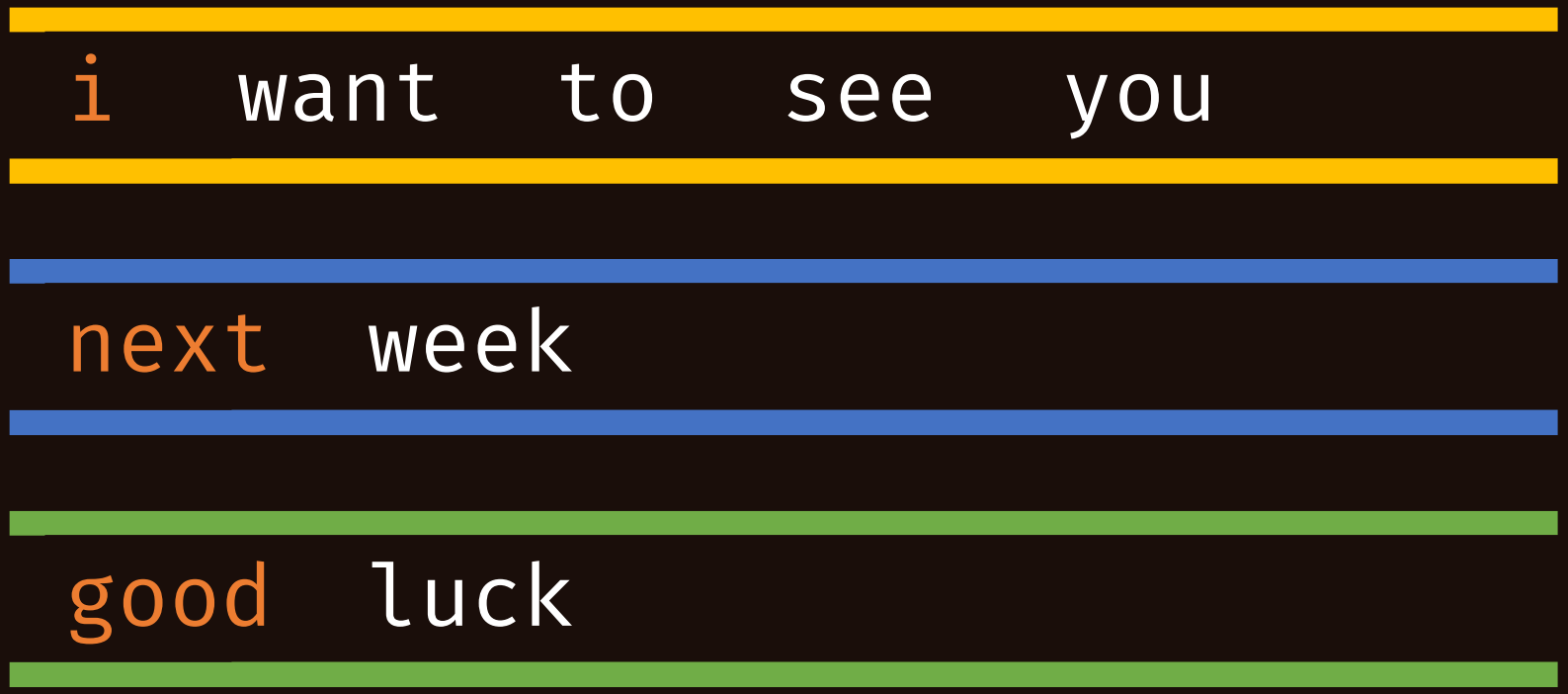
2 E. 앵무새

✓ 이런 식으로, 앵무새의 수만큼 큐를 만들고, 앵무새가 말하게 될 문장을 단어 단위로 넣어 줍시다.



2 E. 앵무새

- ✓ 이제, 여러 앵무새가 말한 것을 섞어놓은 문장이 주어졌을 때, 우리가 보면 되는 부분은 오로지 큐의 맨 앞(front) 뿐입니다.
- ✓ 앵무새가 문장의 순서를 지키면서 이야기한다면 **큐의 맨 앞에 위치한 단어보다 뒤에 있는 단어는 절대 먼저 말하지 않을 것이기 때문입니다.**



2 E. 앵무새

- ✓ 따라서, 섞인 단어마다 모든 큐의 front를 비교해보면서 단어가 큐의 front에 포함되는 지 일일이 모두 확인해 보면 됩니다.
- ✓ 단어를 찾았다면 그 큐에 해당하는 앵무새가 말한 단어임이 확실하므로 그 단어를 pop합니다.
- ✓ 단어를 찾지 못했다면, 그 단어는 큐의 안쪽에 있는 단어가 된다는 뜻이며, 이는 순서가 지켜지지 않았음을 의미합니다. 따라서 이 경우 불가능한 문장임을 알 수 있게 됩니다.
- ✓ 앵무새의 수는 최악의 경우 100 마리, 단어의 개수는 최악의 경우 $10\,000$ 개입니다. 각 단어의 길이는 최악의 경우 32 자입니다. 모든 수를 곱해 연산 횟수를 따져보았을 때 충분히 시간 내에 풀 수 있음을 알 수 있습니다.
- ✓ 이전 주차에서 알고리즘 문제를 풀 때에는 1초에 수행할 수 있는 대략적인 연산의 횟수를 1억~수 억번으로 잡는다고 설명하였습니다.

The End