




# Quest

우아한테크코스 알고리즘 스터디

Week 1. 스택

# 스택이란

- ✓ 팀 내의 알고리즘 동아리에서 설명한 적이 있으니 설명은 복붙 날먹을 사용하겠습니다.
- ✓ 스택을 구현하기 위해 **배열, 리스트** 등을 사용할 수 있습니다.








요술토끼 🐼 오후 8:23

3주차의 주제는 《스택》입니다.  
이번 주부터 자료구조를 다루게 됩니다.

- 자료구조를 설명해 주시는 분들의 설명은 다양합니다. 저는 "데이터를 처리하는 방식" 이라고 생각합니다.
- 현실의 여러 문제에 따라 "적절한 방식" 을 고른다면 보다 데이터를 효율적으로 처리할 수 있게 됩니다. 어떤 상황에서는 배열이 유리할 것이고, 어떤 상황에서는 스택이, 아니면 또다른 자료구조를 사용하는 것이 유리한 상황이 올 것입니다...
- 알고리즘 문제에서는 어떤 적절한 자료구조를 선택하느냐에 따라 시간/공간복잡도에 영향을 줄 수 있기에 상당히 중요합니다.

"스택" 도 이러한 자료구조 중 하나입니다.

- 스택은 한쪽에서만 데이터를 넣고 뺄 수 있는 자료구조입니다. 그림을 참고하시면 이해가 되실 거라고 생각합니다.
- 이렇게 구조 자체는 단순한 자료구조이지만 발상이 신기하게도 정말 다양합니다.



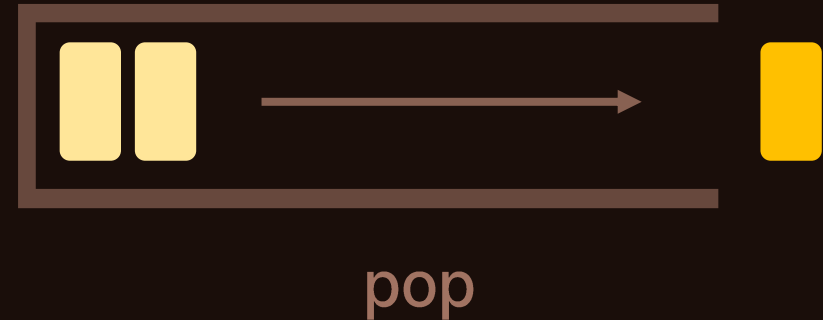
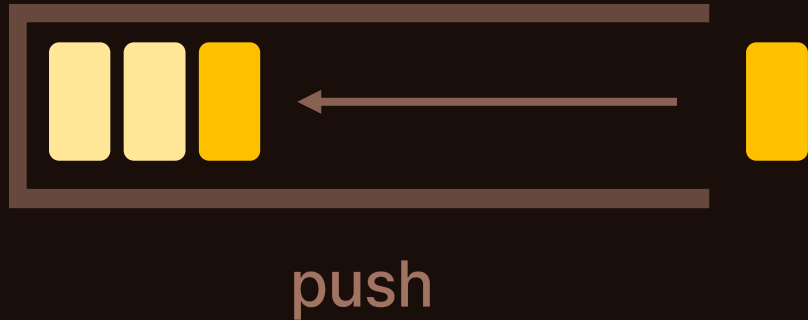
# 스택의 연산

---

- ✓ 마침 연습셋의 첫 번째 문제가 기본적인 스택을 구현하는 문제이니, 스택에서 지원하는 기본적인 연산을 배우면서 문제를 풀어 봅시다.
- ✓ 첫 번째 문제에서 언급되는 연산은 **push, pop, size, empty, top**입니다.

# 스택의 연산

- ✓ 스택에는 가장 바깥쪽에 데이터를 넣을 수 있는 **push** 연산과 빼낼 수 있는 **pop** 연산이 있습니다.
- ✓ 모두  **$O(1)$** 의 시간복잡도로 수행할 수 있습니다. 넣는/빼는 데이터에만 영향을 주기 때문입니다.
- ✓ 스택은 기본적으로 데이터의 중간에서의 push/pop은 지원하지 않습니다.



# 스택의 연산

- ✓ 스택에는 가장 마지막에 삽입된 (=바깥쪽에 위치하는) 데이터를 확인할 수 있는 top이라는 연산이 있습니다.
- ✓ 또한, 스택의 크기를 확인할 수 있는 size, 스택이 비어있는지의 여부를 알 수 있는 empty도 있습니다.
- ✓ 설명한 세 연산 모두  **$O(1)$** 의 시간복잡도로 수행할 수 있습니다.
- ✓ 어떤 명령을 지원하는지는 스택을 지원하는 언어마다 다를 수 있습니다. 저는 문제셋의 첫 번째 문제를 기준으로 언급된 명령들을 설명했습니다.



# 스택의 구현

- ✓ 스택을 본격적으로 구현해 보는 시간도 가져야 하지만, 사실 지금까지 설명한 연산들을 보면 아시겠지만, 모두 JavaScript, C++, Python을 비롯한 언어에서 기본적으로 지원하는 연산들이거나, 어렵지 않게 구현할 수 있는 연산들입니다.
- ✓ 특히 JavaScript의 Array, Python의 list, C++의 <vector>, <stack>, <deque> 등을 사용할 경우 이러한 메서드들의 대부분이 기본적으로 지원됩니다.
- ✓ 따라서, 리스트를 사용하여 구현하신다면 실제로 구현하게 되는 부분은 많지 않을 것으로 예상됩니다. 따라서 방법은 따로 적어 두지 않겠습니다.
- ✓ 그렇지만, 어떻게 직접 구현해야 하는 지 궁금하신 분들을 위해, JavaScript 언어를 이용해 고정된 크기의 배열에서 스택을 구현하는 클래스를 짜 보겠습니다.

# 스택의 구현

✓ 고정 크기의 배열을 이용하여, 최대한 자바스크립트에서 지원하는 메서드들을 사용하지 않고 구현해 보았습니다.

```
class Stack {
  constructor(stackSize) {
    this.stack = new Array(stackSize);
    this.topIndex = -1;
  }

  push(item) {
    this.topIndex += 1;
    this.stack[this.topIndex] = item;
  }

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

    const poppedItem = this.stack[this.topIndex];
    this.topIndex -= 1;

    return poppedItem;
  }
}
```

```
size() {
  return this.topIndex + 1;
}

empty() {
  return this.topIndex === -1;
}

top() {
  if (this.topIndex === -1) {
    return -1;
  }

  return this.stack[this.topIndex];
}
```

# 본격적인 문제풀이 시작

- ✓ 첫 번째 문제를 풀면서 스택이 무엇인지, 어떤 기능을 지원하는지 기본적으로 알아 보았습니다. 이 다음 페이지부터는 B번 문제부터 스택을 이용하여 문제를 어떻게 풀 수 있는지 전략들을 소개하고자 합니다.





- ✓ VPS의 여부를 판단하는 문제입니다.
- ✓ 괄호의 모양이 바르게 구성되었다는 것은, 모든 ( 와 모든 ) 가 서로 짝지어졌음을 의미합니다.
- ✓ ) 가 등장했을 경우, ) 는 직전에 등장한 ( 와만 짝을 짓게 됩니다. 이에 따라 우리는 ( 가 있는지를 판단하는 것뿐만 아니라, 한 쌍의 괄호가 짝지어졌을 경우 짝지어진 괄호를 제외했을 때에도 이전에 등장했던 ( 들의 정보를 기억해야 합니다.
- ✓ 따라서 스택을 사용해 등장했던 ( 들의 정보를 저장하고, 필요할 경우 가장 최근에 등장했던 ( 부터 꺼냄으로써 우리가 원하는 목표를 달성할 수 있습니다.
- ✓ 다음 페이지에서 본격적인 알고리즘을 알아보시다.



## B. 괄호

- ✓ 처음에, 스택은 비어 있습니다. 이 스택은 등장하는 ( 를 저장하고 관리하기 위해 사용할 것입니다.
- ✓ 괄호 문자열을 차례대로 돌면서 각 문자가 ( 인 경우와 ) 인 경우에 대해 각각 처리를 해 봅시다.

stack





## B. 괄호

- ✓ ( 가 등장할 때마다, 스택에 ( 를 저장합니다.
- ✓ 이 작업은 짝지어야 하는 ( 를 기억해 두고, 이후 ) 가 등장할 때마다 짝을 지을 수 있도록 하기 위한 작업입니다.

stack

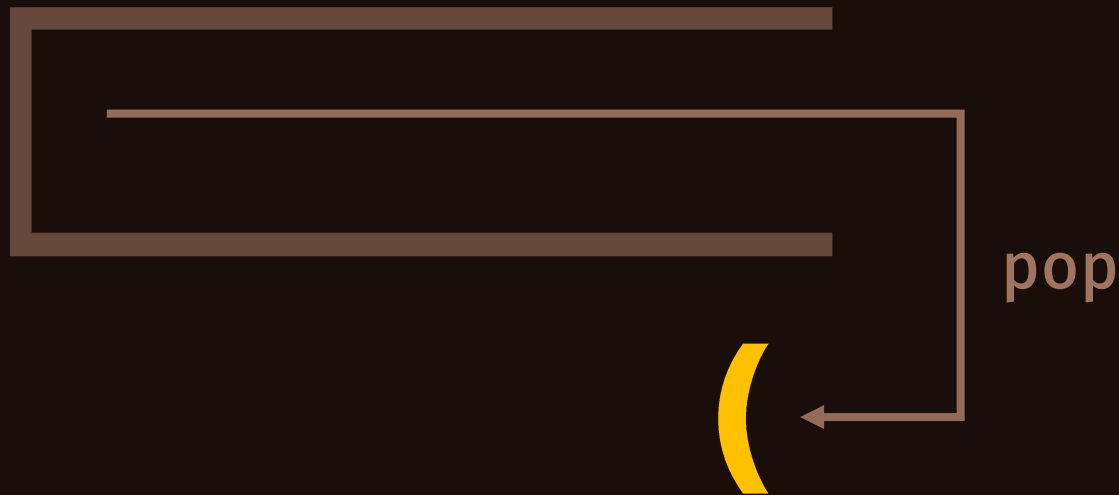


- ✓ ) 가 등장했을 경우에는 먼저 스택을 확인해 보아야 합니다.
- ✓ ( 가 스택의 top에 저장되어 있다면, 지금 등장한 ) 와 스택의 top에 있던 ( 가 서로 짝짓는 상황이 됩니다. 짝을 지었으므로 ( 를 pop해 스택에서 제거해 줍시다.

stack



stack



- ✓ ) 가 등장했을 때, 스택이 비어 있었다면 ) 와 짝지을 ( 가 없음을 의미합니다. 이후 ( 가 더 등장하더라도, 이미 등장했던 ) 와 짝을 지을 수는 없겠죠.
- ✓ 따라서 이 상황이 한 번이라도 발생했을 경우 해당 괄호 문자열은 VPS가 아니게 됩니다.

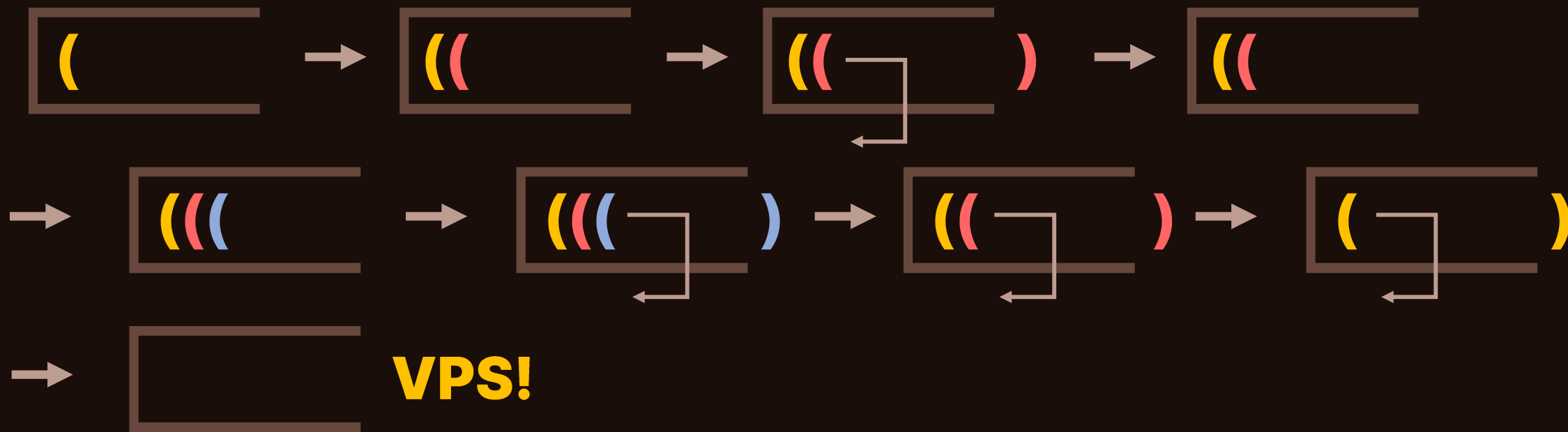
- ✓ 이와 같은 과정으로, 괄호 문자열의 처음부터 끝까지 괄호를 하나씩 넣어보면서 작업을 진행합니다.
- ✓ 모든 작업을 마친 후, 스택이 비어 있는지 확인합니다.
- ✓ 주어진 괄호 문자열이 올바른 괄호 문자열이라면 모든 ( 는 모든 ) 와 짝지어졌을 것이므로, 스택은 비어 있어야 합니다.
- ✓ 그렇지 않다면, 짝지을 수 없는 ( 가 있음을 의미하므로, 해당 괄호 문자열은 VPS가 아니게 됩니다.
- ✓ VPS를 판단하기 위해서는 최악의 경우 괄호 문자열의 처음부터 끝까지 확인해야 하므로 시간복잡도는  $O(n)$  입니다.

## 4

## B. 괄호

✓ 이해를 돕기 위해 몇 가지 예시를 제시하고자 합니다.

((()(( )))

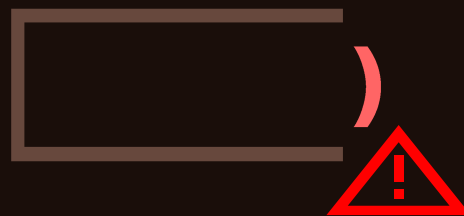
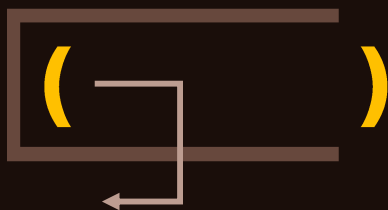


## 4

## B. 괄호

✓ 이해를 돕기 위해 몇 가지 예시를 제시하고자 합니다.

**( ) ) (**

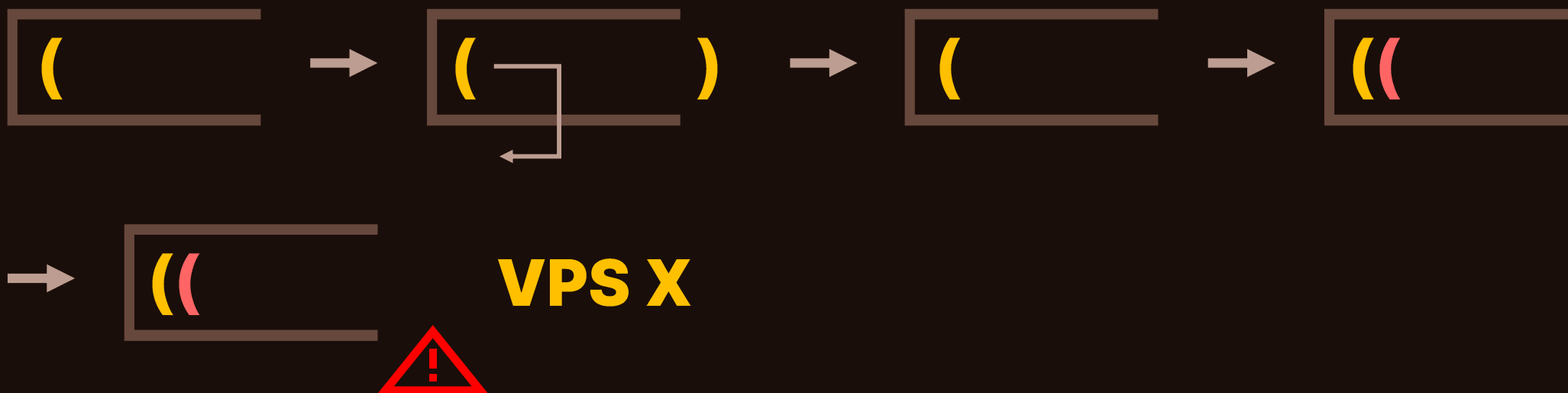


**VPS X**



✓ 이해를 돕기 위해 몇 가지 예시를 제시하고자 합니다.

) ( (





```
const isVPS = (brackets) => {  
  const stack = [];  
  
  for (let i = 0; i < brackets.length; i++) {  
    if (brackets[i] === '(') {  
      stack.push(brackets[i]);  
      continue;  
    }  
  
    if (stack.length === 0) {  
      return false;  
    }  
  
    stack.pop();  
  }  
  
  return stack.length === 0;  
}
```



## C. 균형잡힌 세상

- ✓ 이전 문제인 괄호 문제와 비슷한 문제입니다.
- ✓ 하지만 괄호 외에 다른 문자열들이 주어져 추가적인 처리가 필요하고, 사용되는 괄호도 **() []** 로 두 종류가 되었습니다.

## 4

# C. 균형잡힌 세상

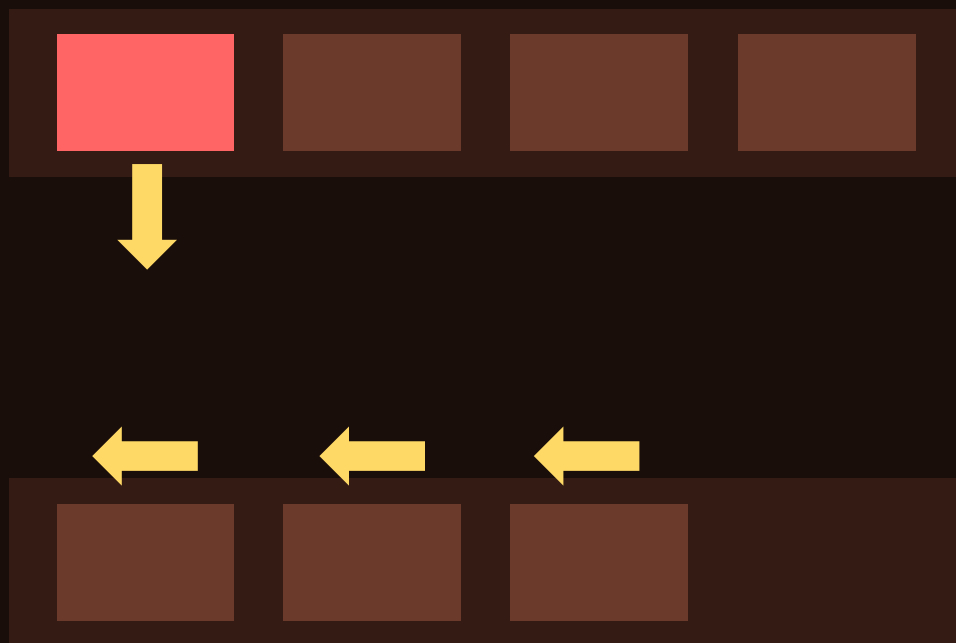
- ✓ 더 어려워졌지만, VPS를 판단하는 조건에서 크게 벗어나지 않았습니다. 단 괄호는 두 종류가 되었기 때문에 추가적인 조건이 필요합니다.
- ✓ 이 문제에서는 스택의 top이 ( 이거나, [ 일 것입니다. 새로운 닫는 괄호가 들어왔을 때, 스택의 top을 확인해 보는 작업이 추가로 필요합니다.
- ✓ 괄호가 ([, ]) 와 같이 짝지어지는 경우를 고민해 보시면 좋을 것 같습니다.



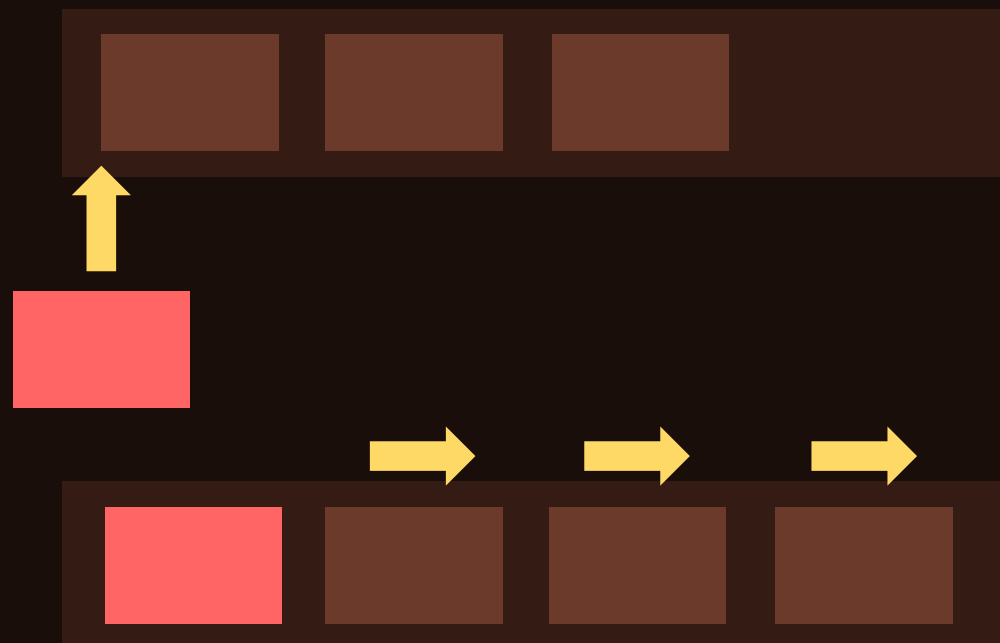
- ✓ 스택을 이용한 아이디어를 생각해 보기 전, 먼저 무식하고 간단한 방법부터 생각해 봅시다.
- ✓ 커서의 위치는 변수로 관리하고, 커서에 있는 문자를 삭제하거나 삽입할 때, 배열의 중간 부분에서 데이터를 끼워넣거나 빼는 방법을 사용할 수 있어 보입니다. 문자열은 배열에 넣어 관리합니다.
- ✓ 이 방법을 사용했을 때, 시간 복잡도는 어떻게 될까요?

- ✓ **L, R**: 단순히 커서의 위치를 저장하는 변수의 값을 1 더하거나 빼면 됩니다. 시간복잡도는  $O(1)$ 로, 즉시 수행할 수 있습니다.
- ✓ **B**: 어떤 값을 빼야 하는 지 인덱스에 접근하는 것 자체는  $O(1)$ 로, 즉시 수행할 수 있습니다. 그러나 값을 빼는 연산의 경우 값을 뺀 이후 빈 공간을 채우기 위해 최악의 경우 남은 모든 값들을 이동시켜야 합니다. 따라서 시간복잡도는  $O(N)$  이 됩니다.
- ✓ **P**: B 연산과 마찬가지로 접근 자체는 문제가 없지만 값을 새롭게 넣으려면 새로운 공간을 만들기 위해 최악의 경우 남은 모든 값들을 이동시켜야 합니다. 역시 시간복잡도는  $O(N)$  입니다.
- ✓ B, P 연산에 대한 간단한 그림은 다음 페이지에 그려두었습니다.

B



P



- ✓ B, P 연산에 대한 그림을 보시고 데이터 하나 옮기기 위해 최악의 경우에는 데이터를 전부 다 옮겨야 하는 이 불편한 상황을 이해하셨다면 성공입니다!
- ✓ 이전의 간단한 발상으로는 이 문제를 풀 수 없습니다. B, P 연산이 너무 비효율적이기 때문입니다.
- ✓ 초기에 굉장히 긴 문자열이 입력되어 있고, 비효율적인 B, P 연산만 반복하면  $O(N)$  짜리 연산을 최대  $M$ 번 하게 되므로 시간복잡도는  $O(NM)$ 이 됩니다.
- ✓ 알고리즘 문제풀이에서 컴퓨터가 1초에 수행할 수 있는 연산 수는 보통 1억 번 ~ 수억 번 정도로 보는 경우가 많습니다.  $N \times M = 600\,000 \times 500\,000 = 300\text{억}$  정도로, 문제를 풀기에는 시간이 너무 부족함을 알 수 있게 됩니다.  $N$ 은 처음에 최대 100 000 이지만 600 000 까지 증가할 수 있음에 유의해 주세요. 또한, 시간복잡도를 고려할 때에 연산 횟수는 대략적으로 생각합니다.
- ✓ 코드를 직접 작성하기 전에 시간복잡도로 미리 문제를 풀 수 있는 지 풀이를 검토해 보는 연습을 해 보시면 학습에 도움이 많이 될 것이라 생각합니다.





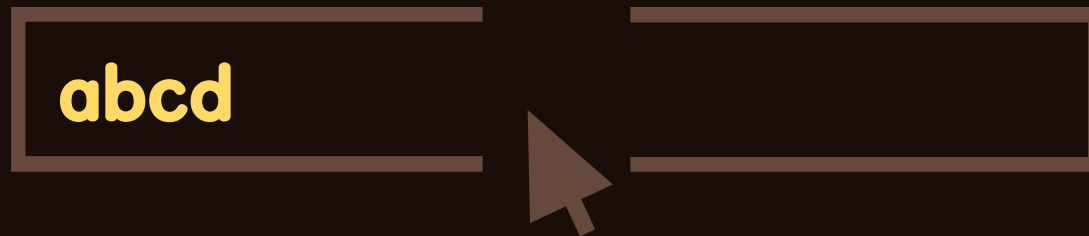
## D. 에디터

- ✓ 스택을 이용하여 더 효율적으로 문제를 풀어 봅시다.
- ✓ 스택을 이용하면 L, D, B, P 연산 모두의 시간복잡도를 무려  **$O(1)$** 로 만들 수 있습니다. 다음 페이지에서 그 방법을 설명해 보겠습니다.

## 2

## D. 에디터

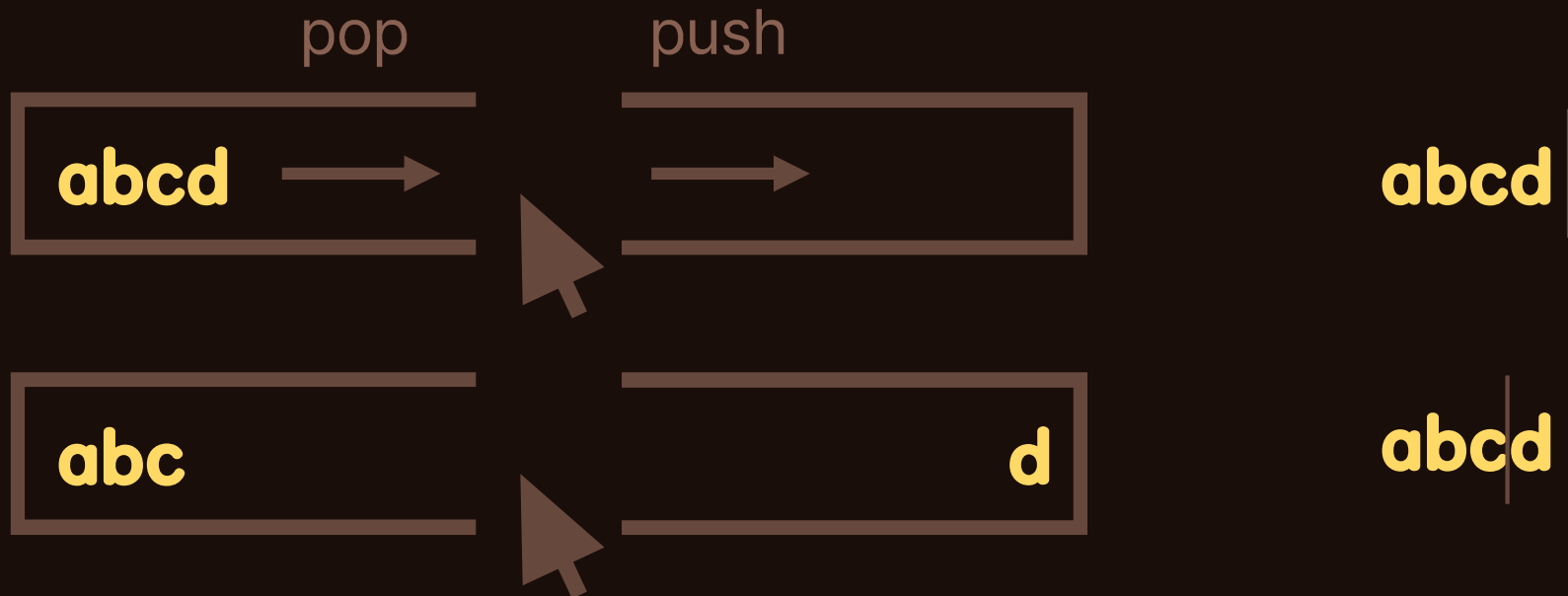
- ✓ 우선 스택 두 개를 준비합니다. 편의상 오른쪽 스택은 거꾸로 뒤집겠습니다.
- ✓ 이후 첫 번째 스택에, 입력으로 주어진 문자열을 넣어 줍시다. 여기까지 하면 시간복잡도는  $O(N)$ 이 됩니다.
- ✓ 커서의 위치는 더 이상 변수로 관리하지 않을 것입니다. 두 스택 사이의 빈 공간이 포인터의 위치가 되는데, 이는 다른 연산을 진행하면서 추가로 설명드리겠습니다.



## 2

## D. 에디터

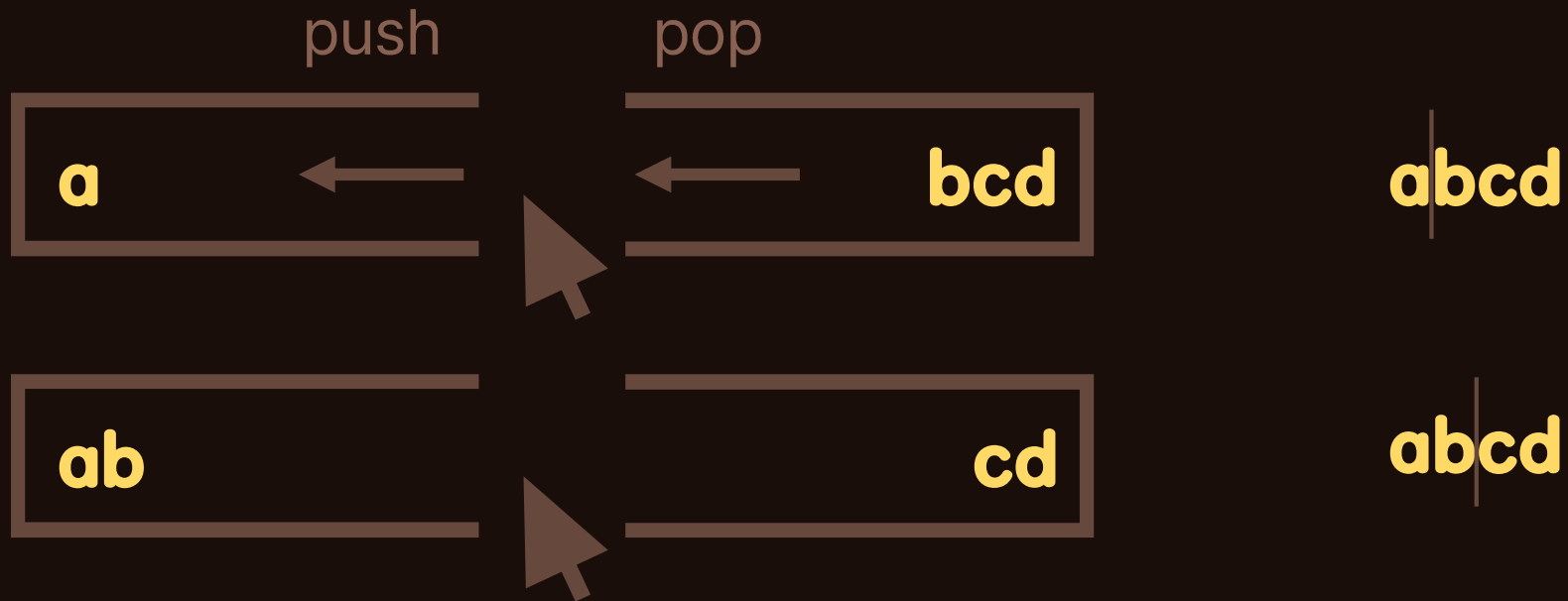
- ✓ **L** 연산의 수행 방법: 첫 번째 스택에서 값을 하나를 pop하고, pop한 값을 오른쪽 스택에 push합니다.
- ✓ 첫 번째 스택이 비어 있으면 커서가 가장 왼쪽에 있는 것이므로 아무 작업도 수행하지 않습니다.
- ✓ pop, push 모두  **$O(1)$** 이므로 우리가 구현한 L 연산의 시간복잡도는  **$O(1)$** 입니다.



## 2

## D. 에디터

- ✓ **R** 연산의 수행 방법: 두 번째 스택에서 값을 pop하고, pop한 값을 첫 번째 스택에 push합니다.
- ✓ 방법만 거꾸로일 뿐 L 연산과 거의 동일합니다. 시간 복잡도는  **$O(1)$** 입니다.



## 2

## D. 에디터

- ✓ **B** 연산의 수행 방법: 첫 번째 스택에서 값을 하나 pop합니다.
- ✓ 이 문제에서는 커서 주변에서 빈번하게 데이터 변경이 일어나고 있습니다. 기존 방법에서의 비효율적인 부분을 개선하기 위해 커서가 있는 부분을 두 스택의 입구에 두었고, 스택의 효율적인 pop, push 연산만을 이용해 연산을 수행할 수 있게 되었습니다.
- ✓ 시간 복잡도는  **$O(1)$** 입니다. 와~~



## 2

## D. 에디터

- ✓ **P** 연산의 수행 방법: 첫 번째 스택에서 지정된 값을 하나 push합니다.
- ✓ B 연산과 해야 하는 작업만 반대일 뿐 그 외에는 똑같습니다. 역시  **$O(1)$** 입니다.



## 2

## D. 에디터

- ✓ 이로써 모든 연산을 각각  $O(1)$ 에 수행할 수 있게 되었습니다.
- ✓ 시간복잡도는 맨 처음 주어지는 길이  $N$ 의 문자열을 스택에 저장하고,  $M$ 번의 연산을 수행하므로  $O(N + M)$ 이 됩니다.
- ✓ 몇백억을 해야 하던 연산 횟수는 이제 몇십만으로 줄었습니다. 이제 문제를 해결할 수 있을 것입니다. 이 발상을 활용하여 코드를 작성해 보시기 바랍니다.



## 4

## E. 문자열 폭발

- ✓ 우선 D번까지 문제를 푸시느라 수고 많으셨습니다. E번 역시 발상이 신기해서 넣은 문제이지만, 무조건 풀어야 한다고 생각하고 넣은 문제는 아닙니다. D번까지 푸신 걸로 이미 충분히 스택에 대해 학습하셨다고 생각되며, **E번은 시간이 남으면 도전해 보시기 바랍니다.**
- ✓ D번 문제와 마찬가지로 무식하지만 간단한 발상을 떠올려 보시고, 이 방법이 왜 안 통하는지를 생각해 보시면 좋을 것 같습니다.



## 4

### E. 문자열 폭발

- ✓ E번 문제는 문자열 하나가 지정되고, `replace(문자열, “”)` 을 반복했을 경우 결과적으로 나오게 되는 문자열이 무엇인지를 물어보는 문제입니다.
- ✓ `replace`를 반복해서 사용하는 풀이는 시간 초과를 받게 됩니다. 왜냐하면, 기본적으로 주어지는 문자열의 길이가 상당히 길어 `replace`를 한 번 사용하기 위한 연산이 무거우며, 새로 생긴 문자열에 폭발 문자열이 포함되어 있을 수도 있다는 문제의 조건 때문에 `replace`를 상당히 많이 사용하게 될 수도 있기 때문입니다.

## 4

## E. 문자열 폭발

- ✓ aaaaaaaaaabbbbbbbbbbb와 같은 문자열이 있고, 폭발 문자열이 ab인 경우는 replace를 정말 많이 진행해야 하는 케이스입니다.
- ✓ 문자열이 상당히 길어 중간에서 문자열을 아주 조금만 삭제하더라도 남은 문자열들을 모두 옮겨야 하는데(D번 문제와 동일한 이유), replace 요구 횟수 또한 상당히 많습니다.
- ✓ 이러한 접근 방법은 시간복잡도가  $O(N^2)$ 이며, 문자열의 길이가 최대 100만이기에 연산 수가 많아 문제를 풀기에는 시간이 너무 오래 걸립니다.

대충 길이 100만이라 가정

aaaaaaaaa**a**bbbbbbbbbb → aaaaaaaaa**a**bbbbbbbbbb → aaaaaaaa**a**bbbbbbbbbb → aaaaaa**a**bbbbbbbbbb → ...

한 번 지울 때마다 엄청난 길이의 문자열을 죄다 옮겨야 하는데  
지워진 문자열은 달랑 두 글자...

## 4

# E. 문자열 폭발

- ✓ 스택을 이용하면 아래와 같은 방법을 사용하여 문제를 해결할 수 있습니다.
- ✓ 1. 문자열의 처음부터 한 글자씩 차례대로 스택에 push합니다.
- ✓ 2. 만약 가장 마지막에 추가된 문자가 폭발 문자열의 마지막 문자라면, 방금 추가된 문자를 마지막으로 폭발 문자열이 포함되었을 수 있습니다. 이 경우에는 폭발 문자열인지를 확인해 봅니다.
  - 스택에서 폭발 문자열의 길이 횟수만큼 pop을 진행하여 확인해 봅니다.
- ✓ 3. 만약 폭발 문자열이 추가된 것이었다면, pop했던 문자열을 다시 넣지 않습니다. 이 경우 폭발 문자열이 지워진 것과 같은 효과를 누릴 수 있습니다.
- ✓ 4. 폭발 문자열이 아니었다면, 원상태로 복귀시켜야 합니다. pop했던 문자열들을 다시 스택에 넣어 줍니다.
- ✓ 5. 모든 작업을 진행한 후 남은 문자열을 스택에서 빼내 출력합니다.



## E. 문자열 폭발

- ✓ 이것만으로는 생소하니 직접 예를 들어 설명해 보겠습니다.
- ✓ `dababccadc`라는 문자열이 주어지고, 폭발 문자열은 `abc`라고 하겠습니다.

# 4

## E. 문자열 폭발

- ✓ d, a, b, a, b를 차례대로 push했습니다. 폭발 문자열의 끝 문자는 c로, 지금까지는 c가 등장하지 않았으므로 폭발 문자열이 발생했을 가능성이 없습니다.

dabab

dababccadc / abc

## 4

## E. 문자열 폭발

- ✓ **c**를 push했습니다. 폭발 문자열의 끝 문자와 일치하기에 방금 넣었던 **c**는 폭발 문자열에 포함되는 마지막 문자였을 수도 있습니다.
- ✓ 폭발 문자열의 길이는 3이므로 문자를 3개 pop한 후 폭발 문자열과 일치하는 지 확인해 보는 작업을 거칩니다. 만약 스택에 들어있는 문자열의 길이가 그보다 짧다면 폭발 문자열이 있을 리가 없으므로 넘어갑니다.
- ✓ 확인 결과 폭발 문자열이므로 뺀 문자열들을 그대로 버립니다.



dababccadc / abc

## 4

## E. 문자열 폭발

- ✓ 또 다시 **c**를 push했습니다.
- ✓ 이번에도 똑같이 확인 작업을 거치며, 역시 폭발 문자열이므로 pop했던 문자열들을 그대로 버립니다.
- ✓ 방금 지웠던 문자열은 **replace**를 한 번 사용했어야 지울 수 있었던 문자열이었습니다. 스택을 사용하여 모든 문자열에 연산을 취하지 않고 지울 수 있었습니다.



dababccadc / abc

## 4

## E. 문자열 폭발

- ✓ 이번에는 a, d, c를 차례로 push했습니다.
- ✓ 마지막에 c가 push되었으므로 확인 작업을 거쳐야 합니다. 이번에는 폭발 문자열이 아닙니다.
- ✓ 따라서 원래 상태로 스택을 복원합니다. pop했던 문자열들을 문자 하나하나씩 push하면 됩니다.



dad



d → adc



dad

dababccadc / abc



## 4

# E. 문자열 폭발

- ✓ 이로써 문자열에 있는 모든 문자를 차례대로 스택에 넣어보고, 적절한 연산을 취해 주었습니다.
- ✓ 스택에 끝까지 문자가 남았다는 것은 폭발되지 않았음을 의미합니다. 따라서, 스택에 남아 있는 문자열을 pop한 후, 문자열을 뒤집어 출력하면 그 문자열이 모든 폭발 이후 최종적으로 남아 있는 문자열이 됩니다.
- ✓ 폭발 문자열을 제거하거나 복원하는 과정에서 많은 횟수의 push와 pop을 진행해야 합니다. 하지만, 폭발 문자열의 길이는 길어봤자 36이기에 일일이 연산을 하더라도 문제를 풀 수 있습니다. (수상할 정도로 짧네요)

dadc

## 4

## E. 문자열 폭발

- ✓ 시간 복잡도는  $O(N)$ 이 됩니다.
- ✓ 폭발 문자열의 글자 수를 고려하면 push 연산 36번, pop 연산 36번으로 총 72라는 수를 연산 횟수에 곱하게 되는데, 그렇더라도 대략적인 연산 횟수는  $1\,000\,000 \times 72 = 72\,000\,000$  로 여전히 시간 내로 문제를 풀 수 있습니다.





The End