

6. Stacks, Queues, and Deques

2024학년도 가을학기

정보컴퓨터공학부 황원주 교수



부산대학교
PUSAN NATIONAL UNIVERSITY

강의내용

- 스택 (Stack)
 - 스택
 - 시간복잡도
- 큐 (Queue)
 - 큐
 - 시간복잡도
- 데크 (Double-ended queue, Deque)
 - 데크
 - 시간복잡도

필요성

- 배열은 random access이므로 유연성이 뛰어나나(flexible) 또한 의도하지 않은 접근이 일어날 수 있음
- 스택/큐/덱은 배열과 유사하게 값을 삽입하는 연산과 저장된 값을 삭제하는 연산이 제공되나, (의도하지 않은 접근이 덜 일어나도록) 이러한 연산을 수행할 때 LIFO (Last In First Out), FIFO (First In First Out) 등과 같은 매우 **제한적인 규칙**을 따름
 - 스택은 데이터의 출력 순서가 입력의 역순으로 이루어 질 경우 사용
 - 큐는 데이터의 출력 순서가 입력 순서와 같은 경우 사용
 - 덱은 **양쪽 끝**에서 삽입과 삭제를 **함께** 하고 싶을 경우 사용

스택 (Stack)

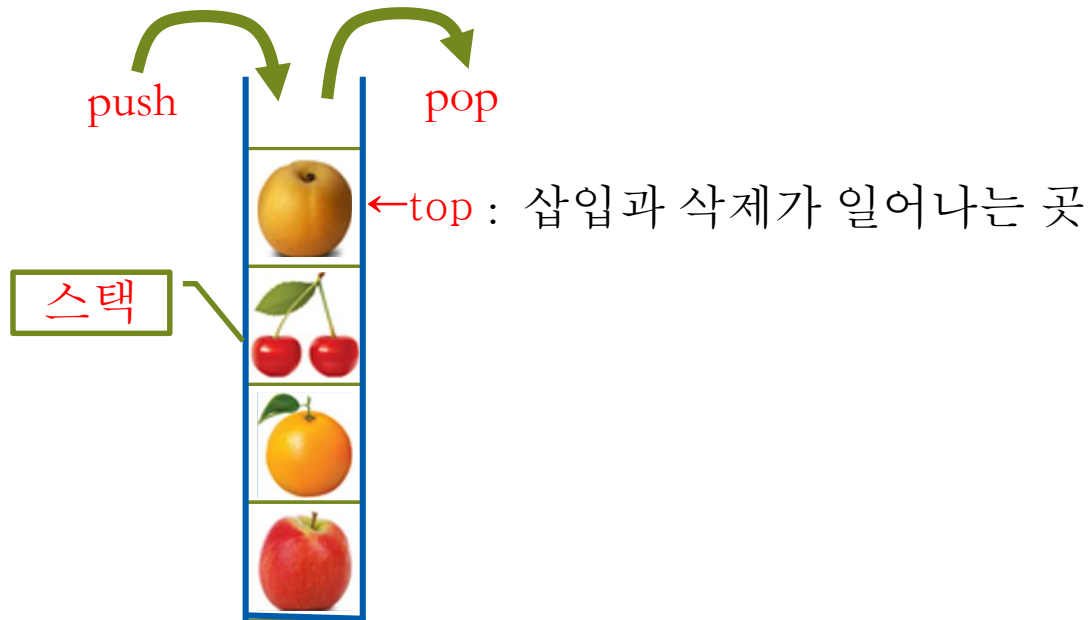
응용

- 회문 (Palindrome) 검사하기
 - 입력 문자열의 모든 문자를 스택에 삽입한 다음, 스택에서 문자들을 다시 꺼내면서 입력 문자열의 문자와 하나씩 맞추어 봄. 다르다면 계속할 필요없이 palindrome이 아님
- 컴파일러의 괄호 짝 맞추기
- 컴파일러의 후위표기법 (Postfix Notation) 수식 계산하기
- 컴파일러의 중위표기법 (Infix Notation) 수식의 후위표기법 변환
- 미로 찾기
- 이진트리의 순회 (전위순회, 중위순회, 후위순회) (8장)
- 그래프의 깊이우선탐색 (Depth-first search) (14장)

스택 (Stack)

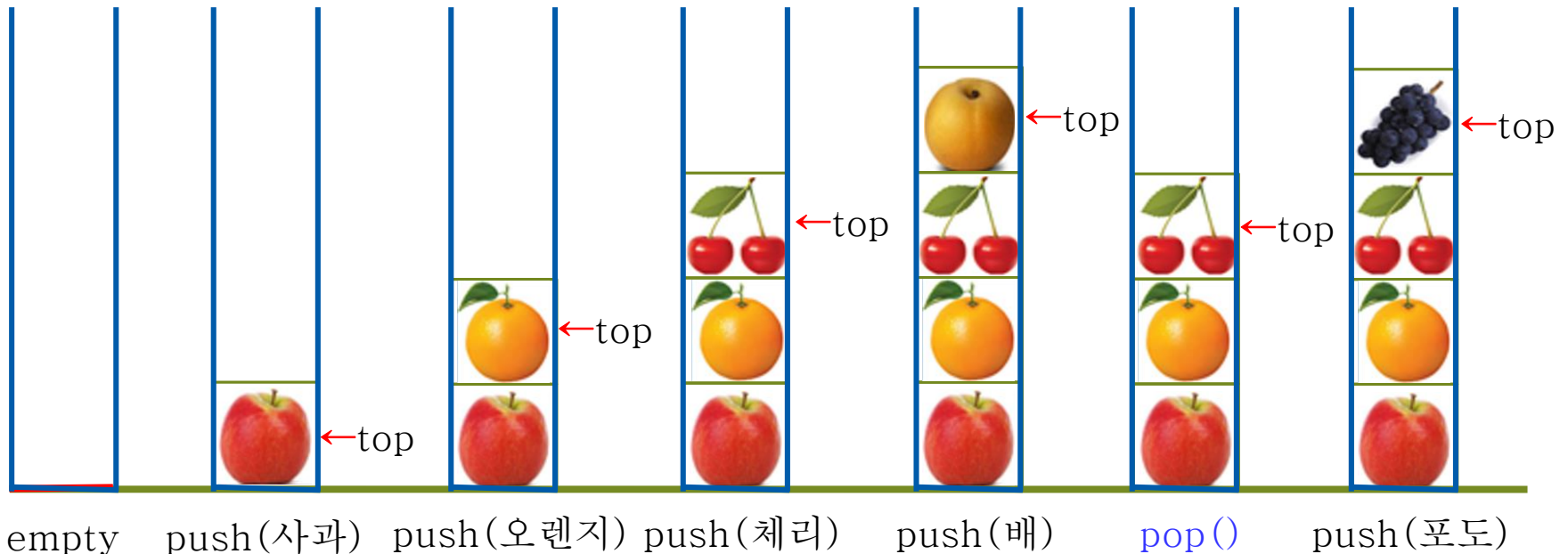
• 정의

- Stack은 가장 최근에 저장된 값 다음에 저장되며, 가장 최근에 저장된 값이 먼저 나간다: **LIFO (Last In First Out)** 원칙
- **한 쪽 끝에서만** 값 (item, element, 요소)을 삭제하거나 새로운 값을 삽입하는 자료구조
- 값을 가지고 있는 배열 또는 연결리스트의 맨 위를 가리키는 포인터를 **top**에 저장하고, top의 위치에 값을 삭제 (**pop**)하거나 삽입 (**push**)



• 예

- 읽기: `peek()`; top의 위치에 있는 값을 읽음
- 쓰기: 없음
- 삽입: `push(사과)`
- 삭제: `pop()`
- 탐색: 없음



스택 ADT

- 삽입

- `S.push(e)`: 스택 `S`의 `top`에 값 `e`를 삽입하는 연산

- 삭제

- `S.pop()`: 스택 `S`의 `top`이 가리키는 값을 반환한 후 삭제하는 연산

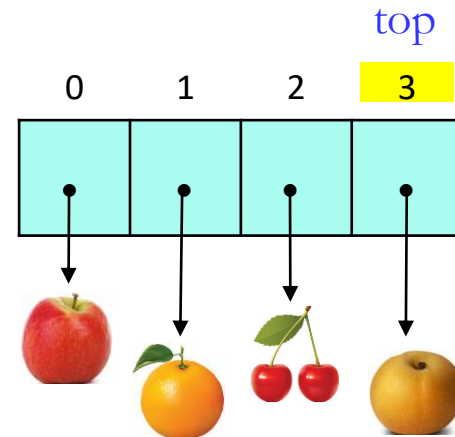
- 보조연산

- `S.top()`: 스택 `S`의 `top`이 가리키는 값의 주소값을 반환하는 연산 (그 값은 삭제하지는 않음)
- `S.is_empty()`: 만약 스택 `S`에 아무 값도 없으면 (즉, `empty`면) `True`를 반환하는 연산
- `len(S)`: 스택 `S`의 값의 개수를 반환하는 연산

스택 ADT의 구현

- 리스트로 구현

- 시간복잡도 때문에 리스트의 맨 뒤를 스택의 상단(top)으로 사용
- 파이썬에서는 push연산은 `append()` 로, pop연산은 `pop(-1)`으로 구현



리스트로 구현한 스택

Class ArrayStack:

```
def __init__(self):  
    self._data = []
```



```
def __len__(self):  
    return len(self._data)
```



```
def is_empty(self):  
    return len(self._data) == 0
```



```
def push(self, e):  
    self._data.append(e)
```



리스트의 맨 뒤에 추가
 $O(1)$

```
def top(self):  
    if self.is_empty():  
        raise Empty('Stack is empty')  
    return self._data[-1]
```



리스트의 맨 뒤 항목 리턴
 $O(1)$

```
def pop(self):  
    if self.is_empty():  
        raise Empty('Stack is empty')  
    return self._data.pop()
```



리스트의 맨 뒤에 있는 항목 제거
 $O(1)$

S = ArrayStack()	# contents: []	
S.push(5)	# contents: [5]	
S.push(3)	# contents: [5, 3]	
print(len(S))	# contents: [5, 3]	outputs 2
print(S.pop())	# contents: [5]	outputs 3
print(S.is_empty())	# contents: [5]	outputs False
print(S.pop())	# contents: []	outputs 5
print(S.is_empty())	# contents: []	outputs True
S.push(7)	# contents: [7]	
S.push(9)	# contents: [7, 9]	
print(S.top())	# contents: [7, 9]	outputs 9
S.push(4)	# contents: [7, 9, 4]	
print(len(S))	# contents: [7, 9, 4]	outputs 3
print(S.pop())	# contents: [7, 9]	outputs 4
S.push(6)	# contents: [7, 9, 6]	

프로세스가 시작되었습니다.(입력값을 직접 입력해 주세요)

```
> 2
3
False
5
True
9
3
4
```

시간복잡도

- 읽기/쓰기

- $O(1)$

- 삽입/삭제

- 파이썬의 **리스트**로 구현한 스택의 push와 pop 연산은 각각 **$O(1)$** 시간이 소요
 - 파이썬의 리스트는 크기가 동적으로 확대 또는 축소되며, 이러한 크기 조절은 사용자도 모르게 수행된다. 이러한 동적 크기 조절은 스택(리스트)의 모든 항목들을 새 리스트로 복사해야 하기 때문에 $O(n)$ 시간이 소요

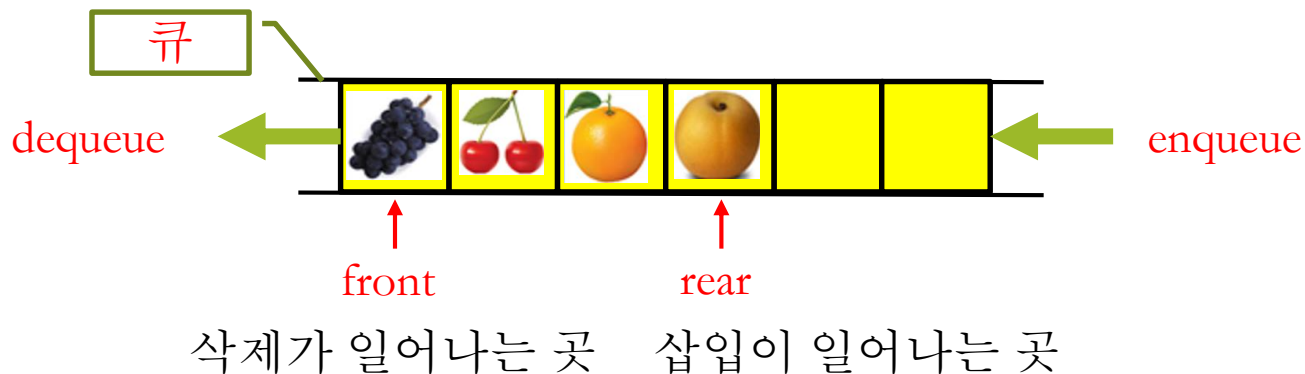
큐 (Queue)

응용

- CPU의 태스크 스케줄링 (Task scheduling)
- 네트워크 프린터
- 실시간 (Real-time) 시스템의 인터럽트 (Interrupt) 처리
- 다양한 이벤트 구동 방식 (Event-driven) 컴퓨터 시뮬레이션
- 이진트리의 **레벨순회 (Level-order traversal)** (8장)
- 그래프에서 **너비우선탐색 (Breadth-first search)** (14장)

큐 (Queue)

- Queue는 가장 최근에 저장된 값 다음에 저장되지만(stack과 동일) 가장 오래전에 저장된 된 값부터 나간다: **FIFO(First In First Out, 선착순)** 원칙
- 삽입과 삭제가 **양 끝에서 각각** 수행되는 자료구조
- 값을 가지고 있는 배열 또는 연결리스트의 맨 앞과 맨 뒤를 가리키는 포인터를 **front**와 **rear**에 각각 저장하고, **front**의 위치에서 값을 삭제(**dequeue**)하고 **rear+1**의 위치에 값을 삽입(**enqueue**)

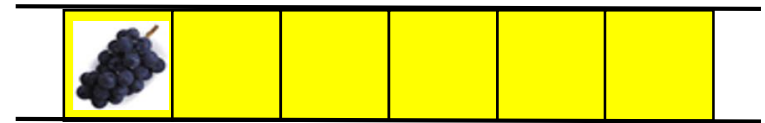


• 예

- 읽기: 없음
- 쓰기: 없음
- 삽입: enqueue (포도)
- 삭제: dequeue ()
- 탐색: 없음



front=None rear=None



front rear

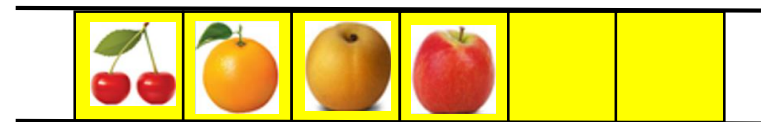
enqueue(포도)



front

rear

enqueue(체리), enqueue(오렌지),
enqueue(배), enqueue(사과)



front

rear

dequeue()

큐 ADT

- 읽기

- 삽입

- `Q.enqueue(e)`: 큐 Q의 rear가 가리키는 곳에 값 e를 삽입(enqueue)하는 연산

- 삭제

- `Q.dequeue()`: 큐 Q의 front가 가리키는 값을 삭제(dequeue)한 후 반환하는 연산

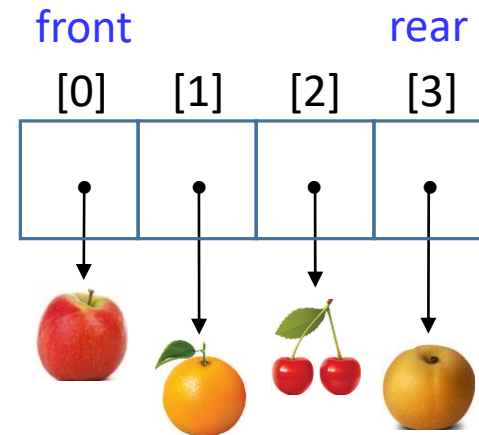
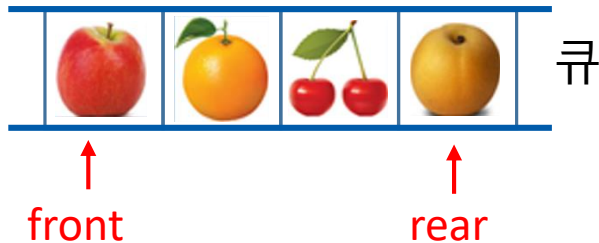
- 보조연산

- `Q.first()`: 큐 Q의 front가 가리키는 값의 주소값을 반환하는 연산 (그 값은 삭제하지는 않음)
- `Q.is_empty()`: 만약 큐 Q에 아무 값도 없으면 (즉, empty면) True를 반환하는 연산
- `len(Q)`: 큐 Q의 값의 개수를 반환하는 연산

큐 ADT의 구현

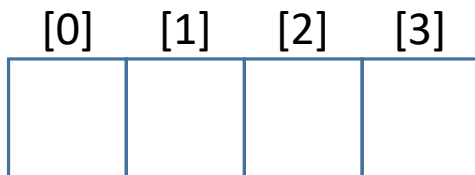
- 방법 1: 리스트의 맨 앞을 삭제

- 리스트의 맨 앞을 삭제하면 모든 항목들을 앞으로 한 칸씩 당겨야 해서 시간복잡도가 $O(n)$ 이 됨 (비추)
- 파이썬에서는 enqueue연산은 `append()`로, dequeue연산은 `pop(0)`으로 구현

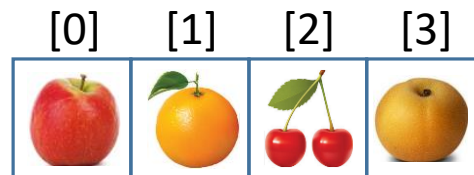


큐 ADT의 구현

- 방법 2: 리스트의 front를 뒤로 이동하면서 삭제
 - 고정된 길이의 배열 (static array)에 값을 삭제하면 $front+=1$, 삽입하면 $rear+=1$ 함
 - rear가 배열의 끝에 도달하면 큐가 full이므로 새로운 값이 있더라도 삽입할 수 없음
 - 큐가 empty가 되면 ($front==rear$) front와 rear를 0으로 초기화함
 - 삽입과 삭제의 시간복잡도는 $O(1)$

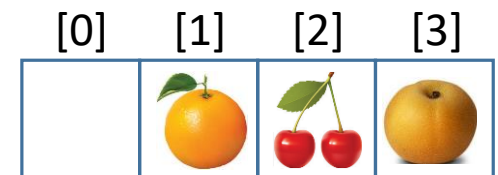


front=0
rear=0



front=0 rear=3

enqueue(사과), enqueue(오렌지),
enqueue(체리), enqueue(배)
더 이상 삽입 못함



front=1 rear=3

dequeue()
(빈 칸이 있는데도)
더 이상 삽입 못함

큐 ADT의 구현

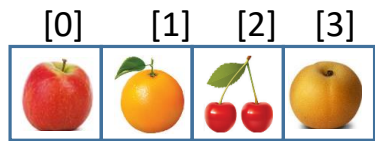
- 방법 3: 원형 큐를 이용하여 구현 (교과서 방법)

- 622. Design Circular Queue

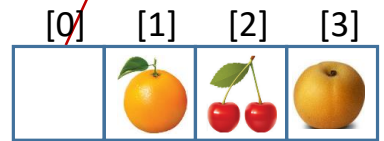
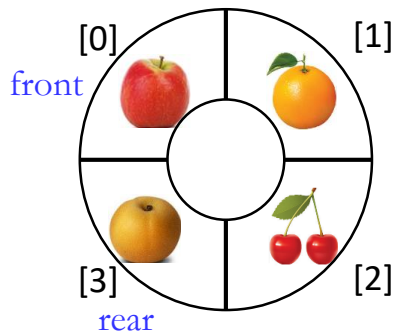
- 고정된 길이의 배열(static array)을 아래와 같이 업데이트하여 원형으로 순회

- $front = (front + 1) \% \text{큐의 길이}$

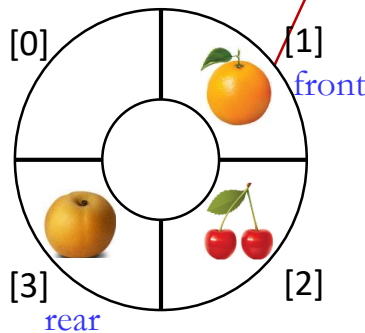
- $rear = (rear + 1) \% \text{큐의 길이}$



enqueue(사과), enqueue(오렌지),
enqueue(체리), enqueue(배)

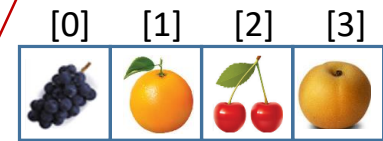


dequeue()

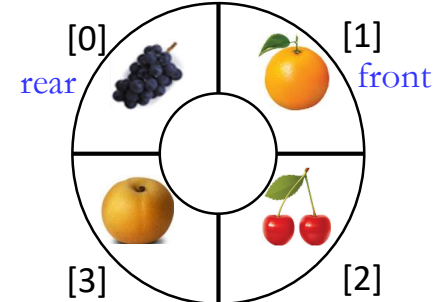


물리적인 모양

논리적인 모양

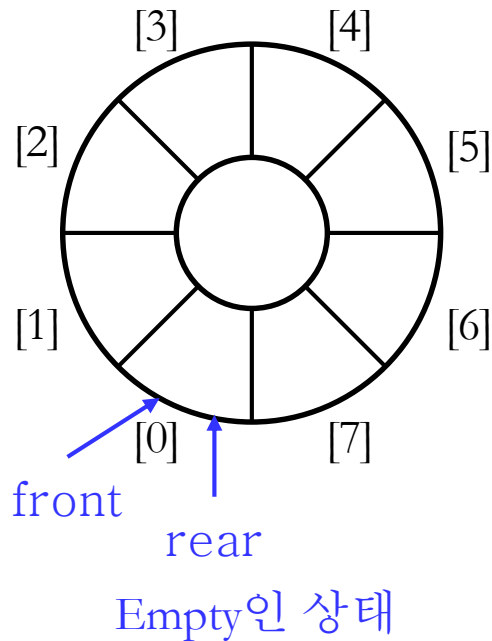


enqueue(포도)



원형 큐가 full일 때와 empty일 때 구분

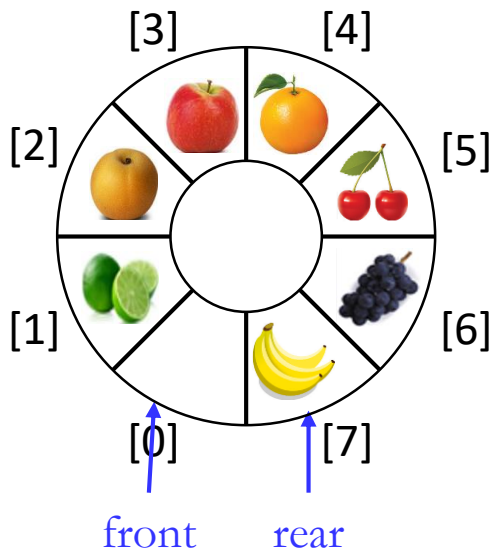
- (방법 1) 원형 큐 내의 항목의 개수를 세는 size를 정의:
 - Empty인 상태: `size == 0`
 - Full인 상태: `size == 큐의 길이`
- (방법 2) front는 항상 비워서 정의:
 - Empty인 상태: `front == rear`



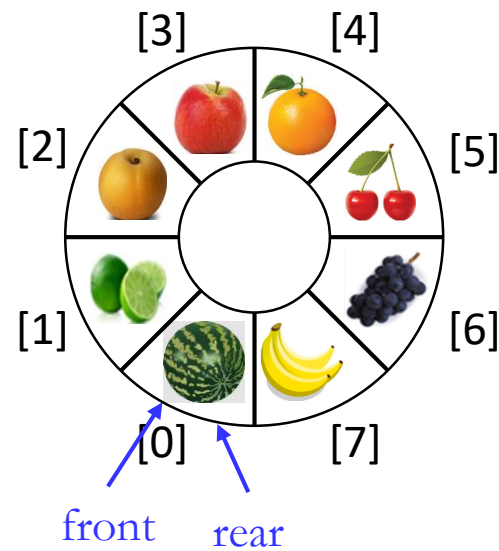
```
def is_empty(self):  
    if self.rear==self.front:  
        return True  
    return False
```

- Full인 상태 : $\text{front} == (\text{rear} + 1) \% \text{큐의 길이}$

```
def is_full(self):  
    if (self.rear+1)%self.MAX_SIZE == self.front:  
        return True  
    return False
```



Full인 상태



오류일 때

원형 큐로 구현한 큐

`Class` ArrayQueue:

`DEFAULT_CAPACITY = 10`

```
def __init__(self):  
    self._data = [None] * Array.DEFAULT_CAPACITY  
    self._size = 0  
    self._front = 0
```

빈 큐 생성

```
def __len__(self):  
    return self._size
```

길이 확인
 $O(1)$

```
def is_empty(self):  
    return self._size == 0
```

비었는지 확인
 $O(1)$


```
def first(self): ●
    if self.is_empty():
        raise Empty('Queue is empty')
    return self._data[self._front]
```

맨 앞의 요소 리턴
 $O(1)$

```
def dequeue(self): ●
    if self.is_empty():
        raise Empty('Queue is empty')
    answer = self._data[self._front]
    self._data[self._front] = None
    self._front = (self._front + 1) % len(self._data)
    self._size -= 1
    return answer
```

맨 앞의 요소 제거
 $O(1)$

```
def enqueue(self, e): ●
    if self._size == len(self._data):
        self._resize(2 * len(self._data))
    avail = (self._front + self._size) % len(self._data)
    self._data[avail] = e
    self._size += 1
```

맨 뒤에 요소 추가
 $O(1)$

```
def _resize(self, cap): ●
    old = self._data
    self._data = [None] * cap
    walk = self._front
    for k in range(self._size):
        self._data[k] = old[walk]
        walk = (1 + walk) % len(old)
    self._front = 0
```

큐 사이즈 증가
 $O(n)$

시간복잡도

- 읽기/쓰기

- $O(1)$

- 삽입/삭제

- 원형 큐로 구현한 큐의 enqueue와 dequeue 연산은 각각 $O(1)$ 시간이 소요됨
 - 단, enqueue의 경우 resize가 필요하다면 $O(n)$ 의 시간이 소요됨

덱 Deque (Double-ended queue)

응용

- 회문 (Palindrome) 검사하기

- Dequeue에 문자열을 저장한 후 양쪽에서 하나씩 빼면서 (pop과 popleft 이용) 같은지 비교하는 것을 반복함. 다르다면 계속할 필요없이 palindrome이 아님

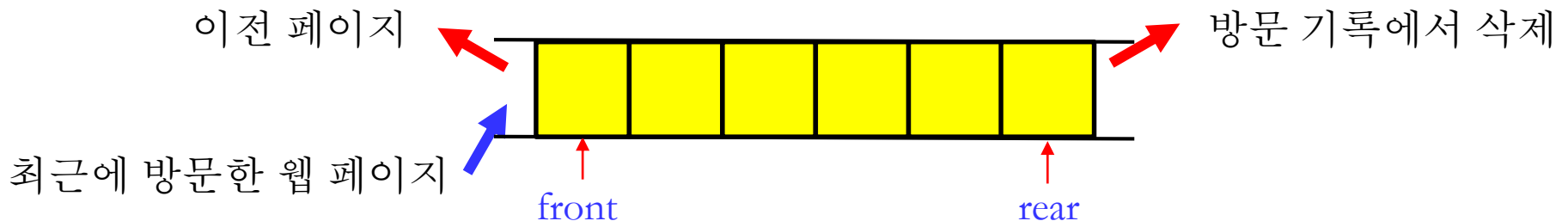
- Age항목 (undo나 history 특성을 가지는 데이터)에 사용

- 웹 브라우저의 방문 기록 등 (history feature)

웹 브라우저 방문 기록의 경우, 최근 방문한 웹 페이지 URL은 앞에 삽입하고, 일정 수의 새 URL들이 앞쪽에서 삽입되면 뒤에서 삭제가 수행

- 문서 편집기 등의 undo 연산 (undo feature)

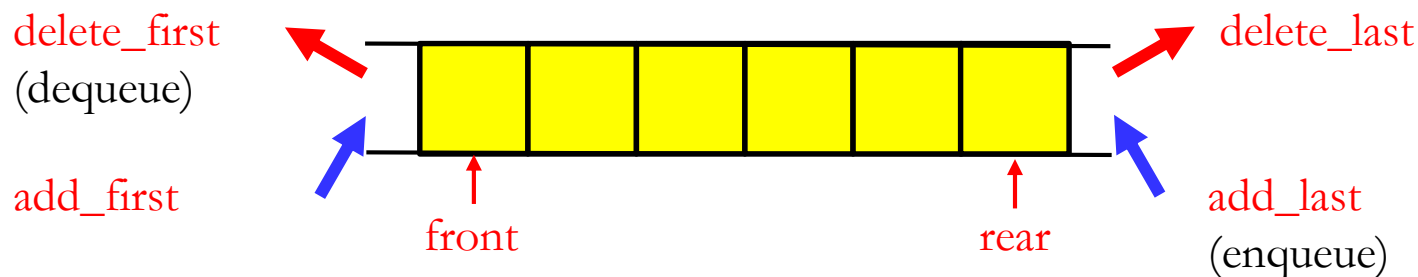
이전 페이지



덱

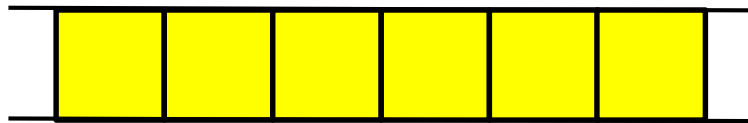
- 정의

- 덱 (Double-ended Queue, Deque)는 양쪽 끝에서 삽입과 삭제를 모두 허용하는 자료구조
- 값을 가지고 있는 배열 또는 연결리스트의 맨 앞과 맨 뒤를 가리키는 포인터를 **front**와 **rear**에 각각 저장하고, front의 위치에서 값을 삽입 (**add_first**) 및 삭제 (**delete_first**) 할 수 있고 rear의 위치에서도 값을 삽입 (**add_last**) 및 삭제 (**delete_last**) 할 수 있음



• 예

- 읽기: 없음
- 쓰기: 없음
- 삽입: `add_first(포도)`, `add_last(체리)`
- 삭제: `delete_first()`, `delete_last()`
- 탐색: 없음

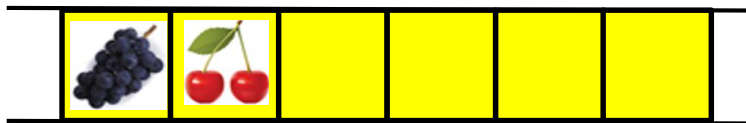


front=None rear=None



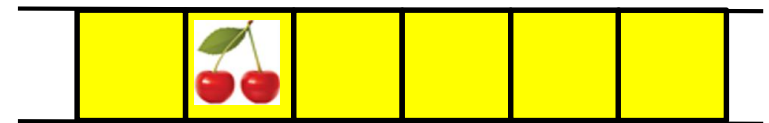
front rear

`add_first(포도)`



front rear

`add_last(체리)`



front rear

`delete_first()`

덱 ADT

- 읽기

- 삽입

- `D.add_first(e)`: 덱 D의 `front`가 가리키는 곳에 값 `e`를 삽입하는 연산
- `D.add_last(e)`: 덱 D의 `rear`가 가리키는 곳에 값 `e`를 삽입(enqueue)하는 연산

- 삭제

- `D.delete_first()`: 덱 D의 `front`가 가리키는 값을 삭제한 후 반환하는 연산
- `D.delete_last()`: 덱 D의 `rear`가 가리키는 값을 삭제한 후 반환하는 연산

- 보조연산

- `D.first()`: 덱 D의 `front`가 가리키는 값의 주소값을 반환하는 연산 (그 값은 삭제하지는 않음)
- `D.last()`: 덱 D의 `rear`가 가리키는 값의 주소값을 반환하는 연산 (그 값은 삭제하지는 않음)
- `Q.is_empty()`: 만약 덱 D에 아무 값도 없으면 (즉, empty면) `True`를 반환하는 연산
- `len(D)`: 덱 D의 값의 개수를 반환하는 연산

덱 ADT의 구현

- 우리 교과서에서는 collections라는 모듈의 deque란 클래스로 구현

Our Deque ADT	collections.deque	Description
len(D)	len(D)	number of elements
D.add_first()	D.appendleft()	add to beginning
D.add_last()	D.append()	add to end
D.delete_first()	D.popleft()	remove from beginning
D.delete_last()	D.pop()	remove from end
D.first()	D[0]	access first element
D.last()	D[-1]	access last element
	D[j]	access arbitrary entry by index
	D[j] = val	modify arbitrary entry by index
	D.clear()	clear all contents
	D.rotate(k)	circularly shift rightward k steps
	D.remove(e)	remove first matching element
	D.count(e)	count number of matches for e

Table 6.4: Comparison of our deque ADT and the collections.deque class.

collections 모듈의 deque 클래스

Python's deque is a low-level and highly optimized double-ended queue that's useful for implementing elegant, efficient, and Pythonic queues and stacks, which are the most common list-like data types in computing.

[출처: <https://realpython.com/python-deque/>]

- Python에서의 스택, 큐, 데크 구현

- collections 모듈의 deque 클래스 사용

`add_first = appendleft()`

`add_last, enqueue, push = append()`

- 스택은 리스트도 많이 사용

- 장단점

[장점] deque의 `popleft()`와 `appendleft(x)` 메서드는 모두 $O(1)$ 의 시간 복잡도를 가지기 때문에, list 자료 구조보다 성능이 훨씬 뛰어남!

[단점] random access의 시간 복잡도가 $O(n)$ 이고, 내부적으로 doubly linked list를 사용하고 있기 때문에 N번째 데이터에 접근하려면 N번 순회가 필요

- `delete_first, dequeue = popleft()`

- `delete_last, pop = pop()`

```

01 from collections import deque
02 dq = deque('data')
03 for elem in dq:
04     print(elem.upper(), end=' ')
05 print()
06 dq.append('r')
07 dq.appendleft('k')
08 print(dq)
09 dq.pop()
10 dq.popleft()
11 print(dq[-1])
12 print('x' in dq)
13 dq.extend('structure')
14 dq.extendleft(reversed('python'))
15 print(dq)

```

새 데크 객체를 생성

맨 뒤와 맨 앞에 항목 삽입

맨 뒤와 맨 앞의 항목 삭제

맨 뒤의 항목 출력

맨 뒤와 맨 앞에 여러 항목 삽입

Console PyUnit

<terminated> deque.py [C:\Users\wsbyang\AppData\Local\Programs\Python\Python36-32\python.exe]

DATA

deque(['k', 'd', 'a', 't', 'a', 'r'])

a

False

deque(['p', 'y', 't', 'h', 'o', 'n', 'd', 'a', 't', 'a', 's', 't', 'r', 'u', 'c', 't', 'u', 'r', 'e'])

시간복잡도

- 읽기/쓰기
 - $O(1)$
- 삽입/삭제
 - $O(1)$

파이썬 collections.deque객체의 시간복잡도

Operation	Average Case	Amortized Worst Case
Copy	$O(n)$	$O(n)$
append	$O(1)$	$O(1)$
appendleft	$O(1)$	$O(1)$
pop	$O(1)$	$O(1)$
popleft	$O(1)$	$O(1)$
extend	$O(k)$	$O(k)$
extendleft	$O(k)$	$O(k)$
rotate	$O(k)$	$O(k)$
remove	$O(n)$	$O(n)$

[출처: <https://wiki.python.org/moin/TimeComplexity>]

중간에 삽입/삭제할 때는 여전히 느림

-extend(): deque 의 maxlen 만큼 저장된 경우, extend() 메소드를 호출하면 오른쪽 끝에 값이 저장되고 입력된 값의 개수만큼 왼쪽 끝의 값이 제거된다.

-rotate(): 양수 인수를 받으면 오른쪽 끝에 있는 항목을 지정한 개수만큼 왼쪽 끝으로 이동하고, 음수 인수를 받으면 왼쪽 끝에 있는 항목을 지정한 개수만큼 오른쪽 끝으로 이동시킨다.

- remove(value): 첫 번째로 value 와 값이 일치하는 요소를 제거.

코딩 테스트

LeetCode 문제

[Stack]

- 772. Basic Calculator III (프리미엄, 구현문제)
- 155. Min Stack
- 1047. Remove All Adjacent Duplicates In String

[Queue]

- 622. Design Circular Queue

[Deque]

- 641. Design Circular Deque

772. Basic Calculator III

772. Basic Calculator III

Hard 644 231 Add to List Share

Implement a basic calculator to evaluate a simple expression string.

The expression string contains only non-negative integers, `'+'`, `'-'`, `'*'`, `'/'` operators, and open `'('` and closing parentheses `)'`. The integer division should **truncate toward zero**.

You may assume that the given expression is always valid. All intermediate results will be in the range of $[-2^{31}, 2^{31} - 1]$.

Note: You are not allowed to use any built-in function which evaluates strings as mathematical expressions, such as `eval()`.

Example 3:

Input: `s = "2*(5+5*2)/3+(6/2+8)"`

Output: 21

Example 4:

Input: `s = "(2+6*3+5-(3*14/7+2)*5)+3"`

Output: -12

- Infix 수식 \rightarrow Postfix 수식 \rightarrow 계산

수식의 표기법

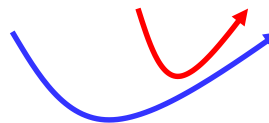
- 수식의 표기법
 - 중위 (infix) 수식: 일반적인 수식 작성법, $2 + 3$ 처럼 연산자가 가운데 위치
 - 전위 (prefix) 수식: $+ 2 3$ 처럼 연산자가 앞에 위치
 - 후위 (postfix) 수식: $2 3 +$ 처럼 연산자가 뒤에 위치
- 컴파일러 (또는 인터프리터)는 스택을 이용하여 프로그래머가 작성한 중위수식을 전위수식이나 후위 수식으로 변환하여 계산, 그 이유는?
 - 괄호를 사용하지 않아도 계산 순서를 알 수 있음
 - 식 자체에 우선순위가 이미 포함되어 있어, 우선순위를 생각할 필요 없음
 - 수식을 읽으면서 바로 계산 가능함. 중위 수식은 괄호와 연산자의 우선순위 때문에 수식을 끝까지 읽은 다음에야 계산 가능

Step 1: Infix→postfix 방법

- 사람이 하는 방법

1. 연산자 우선순위에 따라 괄호를 삽입한다 → 연산자마다 괄호 한 쌍씩 할당
2. 연산자를 자신의 괄호 쌍 중에서 오른쪽 괄호의 다음으로 이동
3. 괄호를 모두 지운다

(예) $A + B * C \rightarrow (A + (B * C)) \rightarrow A B C * +$



- 컴파일러가 하는 방법

- 사람이 하는 방법처럼 괄호를 모두 삽입한 후에 변환하지 않고, 입력을 왼쪽부터 차례로 검사하면서 연산자의 올바른 위치를 찾아 나감
- 입력: +, -, *, /, (,)와 영문 대문자가 섞인 infix 수식
- 출력: postfix 수식
- 관찰 1: $A + B * C \rightarrow A B C * +$
 1. 피연산자(operand)의 순서는 그대로 유지한다
 2. 수식의 왼쪽부터 고려하기 때문에, +를 *보다 먼저 고려한다
 3. +의 위치는 다음에 만나는 연산자 (여기선 *)에 의해 결정된다
 4. *이 + 보다 연산 순위가 높기 때문에 * 다음에 +가 위치해야 한다 \Rightarrow 우선순위가 높은 연산자부터 차례로 나열
- 관찰 2: $(A + B) * C \rightarrow A B + C *$
 1. +는 *보다 먼저)를 만나기 때문에,) 다음에 위치해야 올바르다
 2. 즉,)의 우선순위가 *보다 높기 때문에 + 연산자는) 다음에 위치하게 된다 \Rightarrow)를 만나면 대응되는 (를 만날 때까지 모든 연산자를 출력

- 결국, 어떤 연산자의 위치는 자신보다 우선순위가 높거나 같은 연산자가 오른쪽에서 나타날 때까지 기다린 후, 그 연산자 다음에 위치하면 된다. 이를 위해 연산자는 스택에 저장된 채로 대기해야 한다
- 알고리즘
 - 피연산자를 만나면 그대로 출력
 - 연산자를 만나면 스택에 저장했다가 스택보다 우선 순위가 낮은 연산자가 나오면 그때 출력
 - 왼쪽 괄호는 우선순위가 가장 낮은 연산자로 취급
 - 오른쪽 괄호가 나오면 스택에서 왼쪽 괄호 위에 쌓여있는 모든 연산자를 출력

- Infix→postfix 변환: $(A+B)*C$

토큰: 연산자 또는 피연산자

단계	중위표기 수식	스택	후위표기 수식
0	(A + B) * C	[]	
1	(A + B) * C	['(']	
2	(A + B) * C	['(']	A
3	(A + B) * C	['(', '+']	A
4	(A + B) * C	['(', '+']	A B
5	(A + B) * C	[]	A B +
6	(A + B) * C	['*']	A B +
7	(A + B) * C	['*']	A B + C
8	(A + B) * C	[]	A B + C *

- 풀이

```
def Infix2Postfix( expr ) :
    s = Stack()
    output = []
    for term in expr :
        if term in '(' :
            s.push('(')
        elif term in ')' :
            while not s.isEmpty() :
                op = s.pop()
                if op=='(' : break;
            else :
                output.append(op)
        elif term in "+-*/" :
            while not s.isEmpty() :
                op = s.peek()
                if( precedence(term) <= precedence(op) ) :
                    output.append(op)
                    s.pop()
            else: break
            s.push(term)
        else :
            output.append(term)

    while not s.isEmpty() :
        output.append(s.pop())

    return output
```

Step2: Postfix 수식의 계산

- 알고리즘

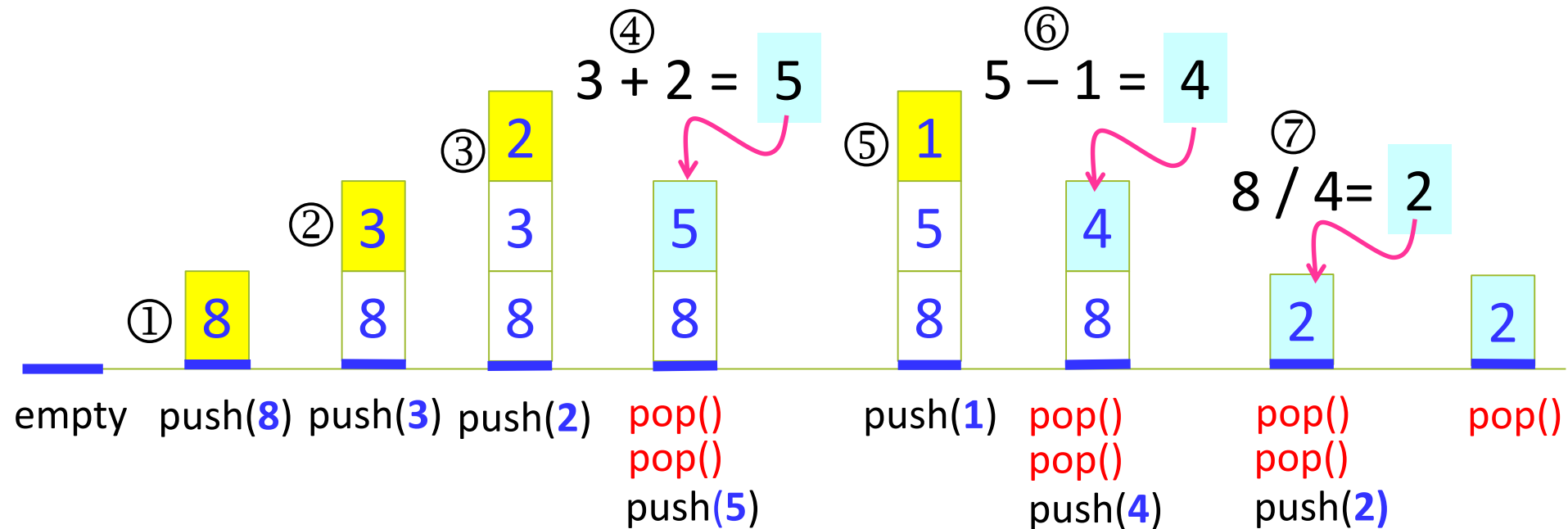
- 입력을 좌에서 우로 토큰을 한 개씩 읽는다.

1. 토큰이 피연산자이면 스택에 push한다.

2. 토큰이 연산자(op)이면 pop을 2회 수행하여 스택에 저장된 피연산을 꺼낸다. 먼저 pop된 피연산자가 A이고, 나중에 pop된 피연산자가 B라면, $(A \text{ op } B)$ 를 수행하여 그 결과 값을 다시 스택에 push한다.

① ② ③ ④ ⑤ ⑥ ⑦

• 예: 8 3 2 + 1 - /



- 풀이

```
def evalPostfix( expr ) :  
    s = Stack()  
    for token in expr :  
        if token in "+-*/" :  
            val2 = s.pop()  
            val1 = s.pop()  
            if (token == '+') : s.push(val1 + val2)  
            elif (token == '-') : s.push(val1 - val2)  
            elif (token == '*') : s.push(val1 * val2)  
            elif (token == '/') : s.push(val1 / val2)  
        else :  
            s.push( float(token) )  
  
    return s.pop()
```

622. Design Circular Queue

‘(방법 2) front는 항상 비워서 정의’방법으로 푸세요.

622. Design Circular Queue

Medium 1230 149 Add to List Share

Design your implementation of the circular queue. The circular queue is a linear data structure in which the operations are performed based on FIFO (First In First Out) principle and the last position is connected back to the first position to make a circle. It is also called "Ring Buffer".

One of the benefits of the circular queue is that we can make use of the spaces in front of the queue. In a normal queue, once the queue becomes full, we cannot insert the next element even if there is a space in front of the queue. But using the circular queue, we can use the space to store new values.

Implementation the `MyCircularQueue` class:

- `MyCircularQueue(k)` Initializes the object with the size of the queue to be `k`.
- `int Front()` Gets the front item from the queue. If the queue is empty, return `-1`.
- `int Rear()` Gets the last item from the queue. If the queue is empty, return `-1`.
- `boolean enQueue(int value)` Inserts an element into the circular queue. Return `true` if the operation is successful.
- `boolean deQueue()` Deletes an element from the circular queue. Return `true` if the operation is successful.
- `boolean isEmpty()` Checks whether the circular queue is empty or not.
- `boolean isFull()` Checks whether the circular queue is full or not.

You must solve the problem without using the built-in queue data structure in your programming language.

Example 1:

Input

```
["MyCircularQueue", "enqueue", "enqueue", "enqueue", "enqueue", "Rear",  
"isFull", "dequeue", "enqueue", "Rear"]  
[[3], [1], [2], [3], [4], [], [], [], [4], []]
```

Output

```
[null, true, true, true, false, 3, true, true, true, 4]
```

Explanation

```
MyCircularQueue myCircularQueue = new MyCircularQueue(3);  
myCircularQueue.enqueue(1); // return True  
myCircularQueue.enqueue(2); // return True  
myCircularQueue.enqueue(3); // return True  
myCircularQueue.enqueue(4); // return False  
myCircularQueue.Rear();    // return 3  
myCircularQueue.isFull();  // return True  
myCircularQueue.dequeue(); // return True  
myCircularQueue.enqueue(4); // return True  
myCircularQueue.Rear();    // return 4
```

- 풀이

```
class CircularQueue:
```

```
    def __init__(self, k: int):  
        self._data = [None] * k  
        self._rearIdx = -1  
        self._frntIdx = 0  
        self._size = 0
```

‘(방법 1) 원형 큐 내의 항목의 개수를 세는 size를 정의’ 방법으로 풀었습니다.

```
    def enqueue(self, value: int):  
        self._fullCheck()
```

```
        self._rearIdx += 1  
        self._rearIdx = self._rearIdx % len(self._data)  
        self._data[self._rearIdx] = value  
        self._size += 1
```

```
    def dequeue(self):  
        self._emptyCheck()
```

```
        self._frntIdx += 1  
        self._frntIdx = self._frntIdx % len(self._data)  
        self._size -= 1
```

```

def Rear(self) -> int:
    self._emptyCheck()
    return self._data[self._rearIdx]

def Front(self) -> int:
    self._emptyCheck()
    return self._data[self._frntIdx]

def _emptyCheck(self):
    if self._size == 0:
        raise RuntimeError('Queue is Empty')

def _fullCheck(self):
    cap = len(self._data)
    if self._size == cap:
        raise RuntimeError('Queue is full')

circularQ = CircularQueue(4)
circularQ.enqueue(1)
circularQ.enqueue(3)
circularQ.enqueue(5)
circularQ.dequeue()
print(circularQ.Front(), circularQ.Rear())

```

20. Valid Parentheses

20. Valid Parentheses

Easy

👍 8372

💬 340

♡ Add to List

🔗 Share

Given a string `s` containing just the characters `'('`, `')'`, `'{'`, `'}'`, `'['` and `']'`, determine if the input string is valid.

An input string is valid if:

1. Open brackets must be closed by the same type of brackets.
2. Open brackets must be closed in the correct order.

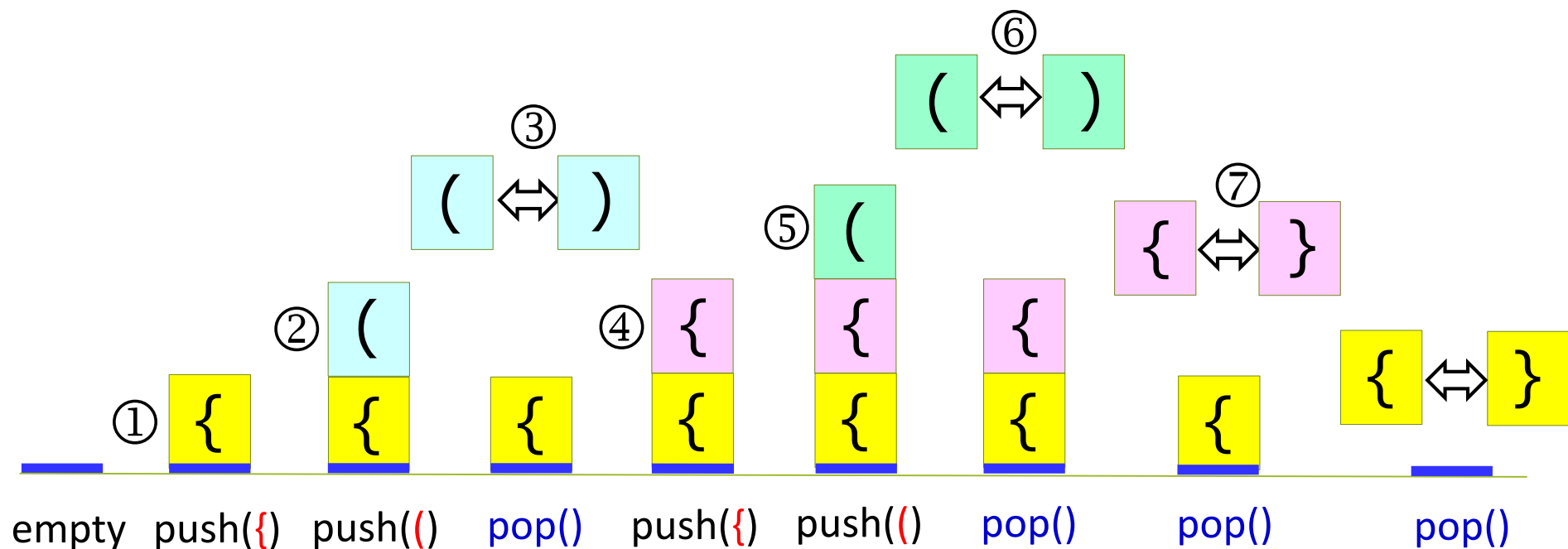
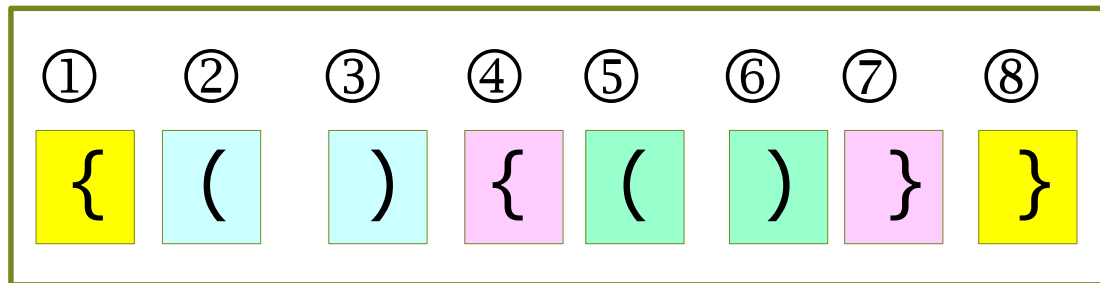
- 관찰 1: 괄호 맞추기

- 왼쪽 괄호 (이 등장하면 짝이 되는 오른쪽 괄호)이 나중에 반드시 등장해야 하고, 두 괄호 안에 포함되는 괄호들도 짝이 맞아야 한다
- 짝이 맞는 괄호들은 즉시 시퀀스에서 빼버린다면, 오른쪽 괄호)이 등장할 때, 자신의 왼쪽 괄호가 가장 최근에 짝을 못 맞춘 괄호로 기다리고 있어야 한다. 즉, 가장 최근에 짝을 못 맞춘 왼쪽 괄호가 가장 빨리 오른쪽 괄호와 짝을 맞추기 때문에 **스택의 LIFO 원칙과 일치함**
- 예: $(2+5)*7-((3-1)/2+7(4$ 는 () (() (이므로 짝이 맞지 않다

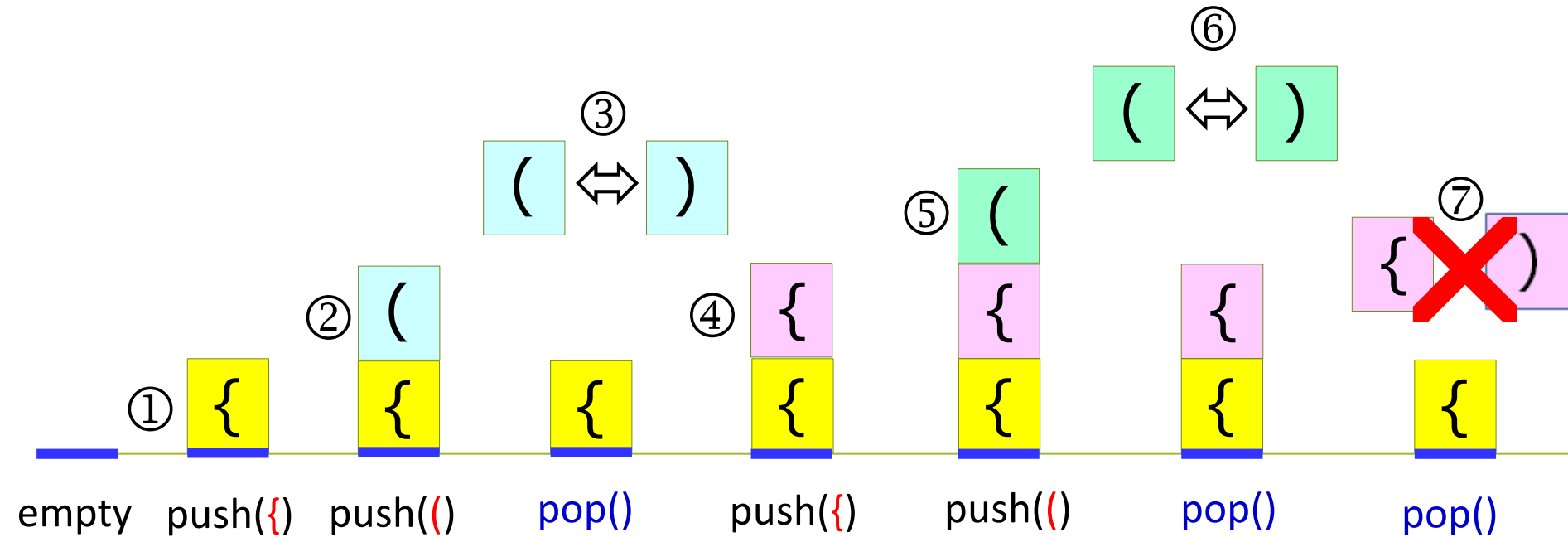
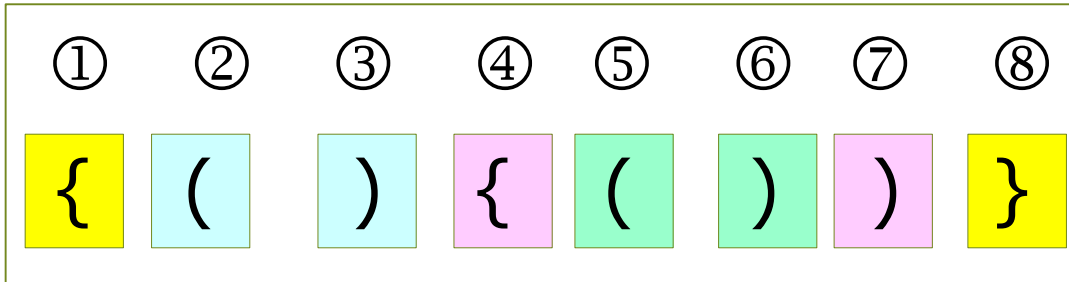
- 알고리즘

- 왼쪽 괄호는 스택에 push, 오른쪽 괄호를 읽으면 pop 수행
- pop된 왼쪽 괄호와 바로 읽었던 오른쪽 괄호가 다른 종류이면 에러 처리, 같은 종류이면 다음 괄호를 읽음
- 모든 괄호를 읽은 뒤 에러가 없고 스택이 empty이면, 괄호들이 짝이 맞는 것
- 만일 모든 괄호를 처리한 후 스택이 empty가 아니면 짝이 맞지 않는 괄호가 스택에 남은 것이므로 에러 처리

[예제 1]



[예제 2]



125. Valid Palindrome

125. Valid Palindrome

Easy



2274



4127



Add to List



Share

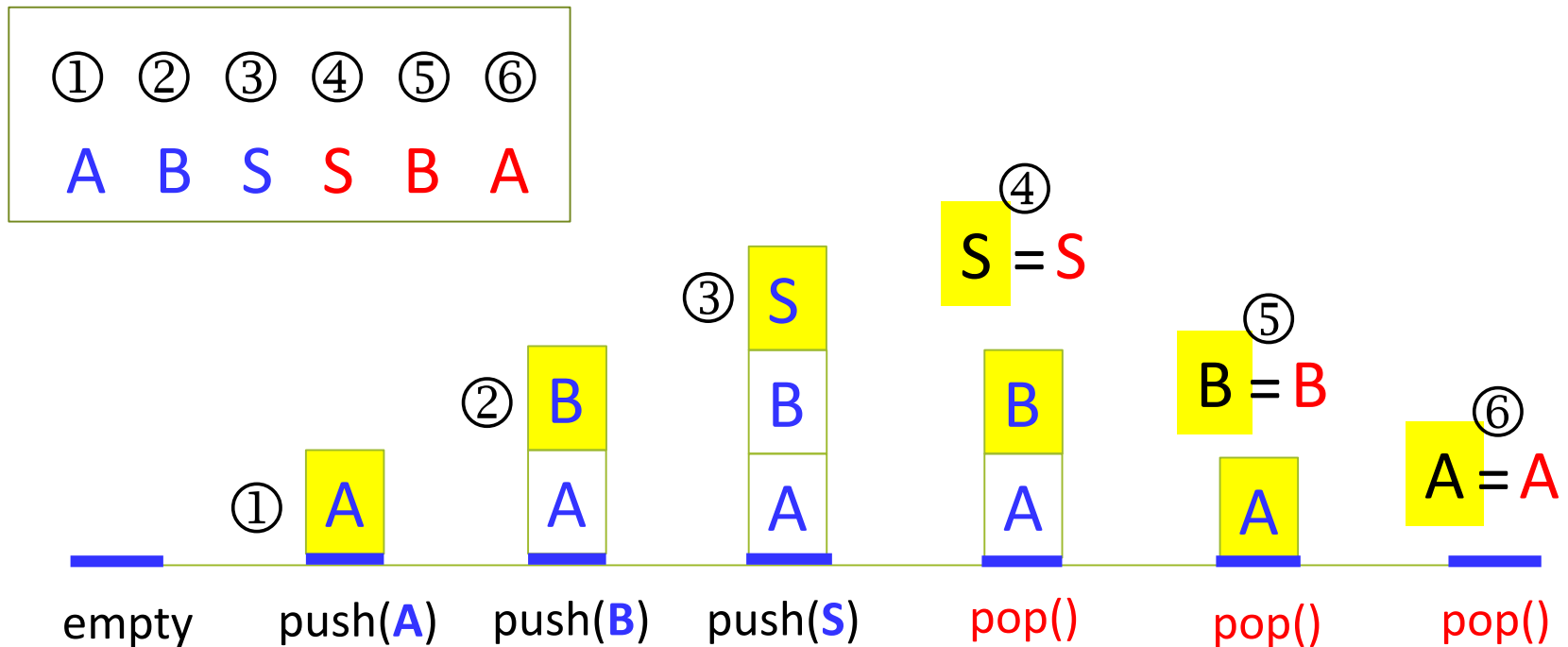
Given a string `s`, determine if it is a palindrome, considering only alphanumeric characters and ignoring cases.

- 우리는 이미 제2장에서 이 문제를 `two pointers technique`으로 풀어 보았음
- `slow-runner`가 읽은 문자를 스택에 저장, 스택에 저장된 것과 `slow-runner` 포인터 다음의 문자들을 비교

- Palindrome을 Equi-directional two pointers technique (slow-runner and fast-runner) + 스택을 이용해서 풀기
 - Palindrome 검사하기는 주어진 스트링의 앞부분 반을 차례대로 읽어 스택에 push한 후, 문자열의 길이가 짝수이면 뒷부분의 문자 1 개를 읽을 때마다 pop하여 읽어 들인 문자와 pop된 문자를 비교하는 과정을 반복 수행
 - 만약 마지막 비교까지 두 문자가 동일하고 스택이 empty가 되면, 입력 문자열은 palindrome

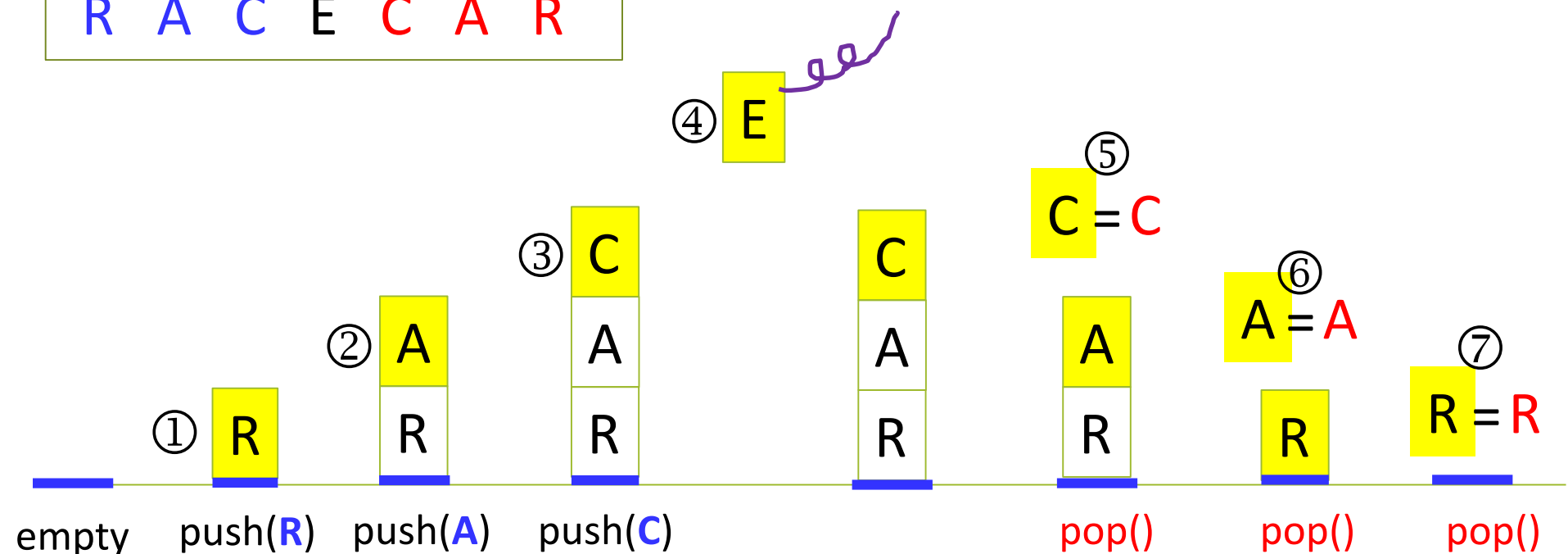
- 문자열의 길이가 홀수인 경우, 주어진 스트링의 앞부분 반을 차례로 읽어 스택에 push한 후, **중간 문자를 읽고 버린다**. 이후 짝수 경우와 동일하게 비교 수행

[예제 1]



[예제 2]

①	②	③	④	⑤	⑥	⑦
R	A	C	E	C	A	R



응용

- 회문 (Palindrome) 검사하기

- 방법1 (비추): 문자열 s와 s를 reverse한 문자열이 같다면 palindrome임

reversed(s)는 문자열 s를 거꾸로 한 iterable 객체임

```
>>> s == "".join(reversed(s))
```

- 방법2: Two pointers technique 사용

- 방법3: dequeue 사용

1. dequeue에 문자열을 저장한 후 양쪽에서 하나씩 빼면서(pop과 popleft 이용) 같은지 비교하는 것을 반복함. 다르다면 계속할 필요없이 palindrome이 아님

2. Pseudo 코드:

```
from collections import deque
def check_palindrome(s):
    dq = deque(s)
    palindrome = True
    while len(dq) > 1:
        if dq.popleft() != dq.pop():
            palindrome = False
    return palindrome
```

감사합니다!



부산대학교
PUSAN NATIONAL UNIVERSITY

왜 자료구조의 개념을 알아야 하나?

아직도 '코볼' 쓰는 KB금융

코볼 가르치는 곳 없어 유지보수 인력확보 어려워
"KB금융 경영진 IT현실 모르고 일만 키워" 지적도

강진규 기자 | 입력: 2014-05-29 20:26

KB국민은행과 KB국민카드의 전산시스템 교체 논란 이면에는 점차 사라져가고 있는 프로그래밍언어인 '코볼(COBOL)' 개발자 문제가 자리잡고 있다. 일각에서는 코볼이 향후 KB금융 IT 발전의 발목을 잡을 것이라는 우려도 나오고 있다.

29일 금융권과 IT업계에 따르면 KB국민은행과 KB국민카드는 그동안 메인프레임 시스템을 운영하면서 코볼로 개발된 프로그램을 유지보수하고 수정하는 데 골머리를 앓아 온 것으로 알려졌다.

코볼은 1959년 발표된 사무 처리를 위한 컴퓨터 프로그래밍 언어다. 코볼은 우수성을 인정받아 한 때 금융 및 기업용 소프트웨어(SW) 개발에 널리 사용됐다. 하지만 자바(JAVA), C언어, C++, C# 등 프로그래밍 언어들이 나오면서 사용이 줄고 있다.

- 실무에서는 다양한 프로그래밍언어로 구현된 프레임워크가 혼재되어 있음

IT인재들 "은행은 개발자 무덤"... 온갖 러브콜에도 외면

입력 : 2021-08-09 21:00:00 | 수정 : 2021-08-09 21:58:52

은행권 디지털 인재 채용 어려운 이유는

"은행 IT 옛날 언어 '코볼' 사용
자신만의 커리어쌓기 어려워"
영업점 의무근무도 기피 이유
"디지털 인력, 지원인력 인식
체질변화 없으면 생존 어려워"