

Part 3

스택과 큐

목차

3.1 스택

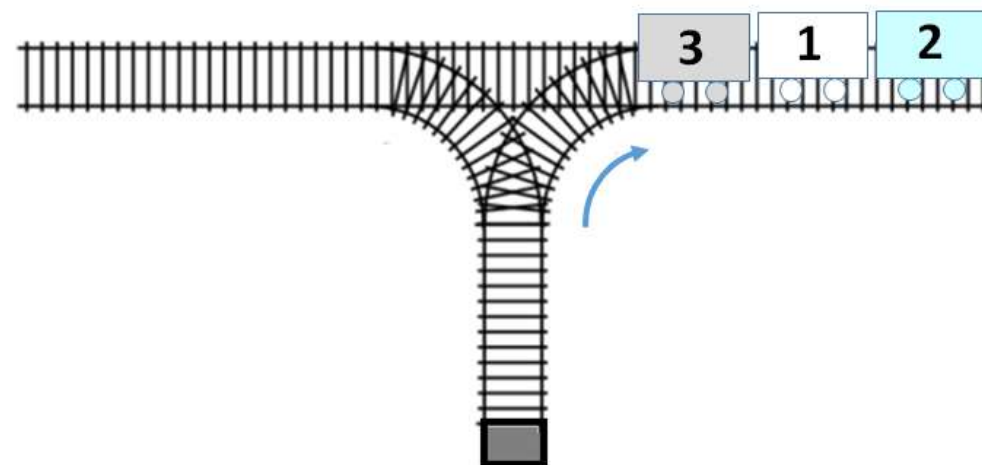
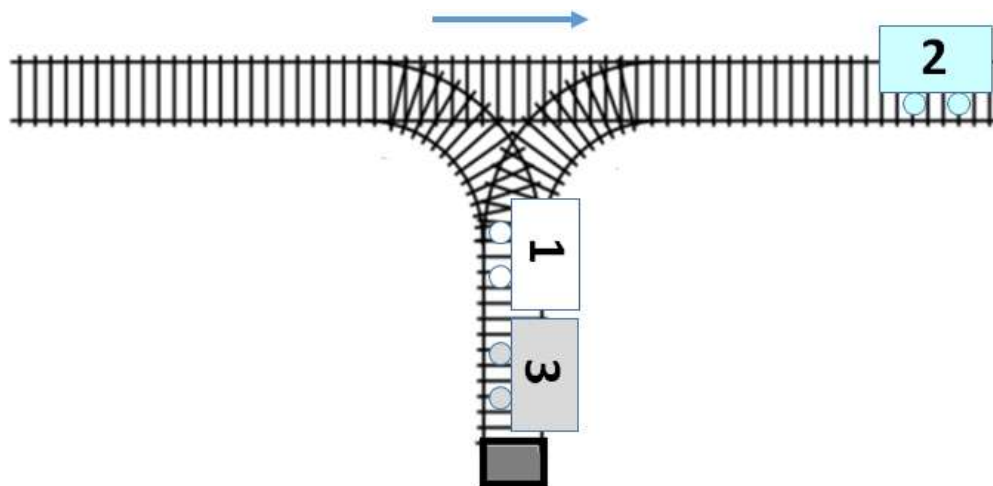
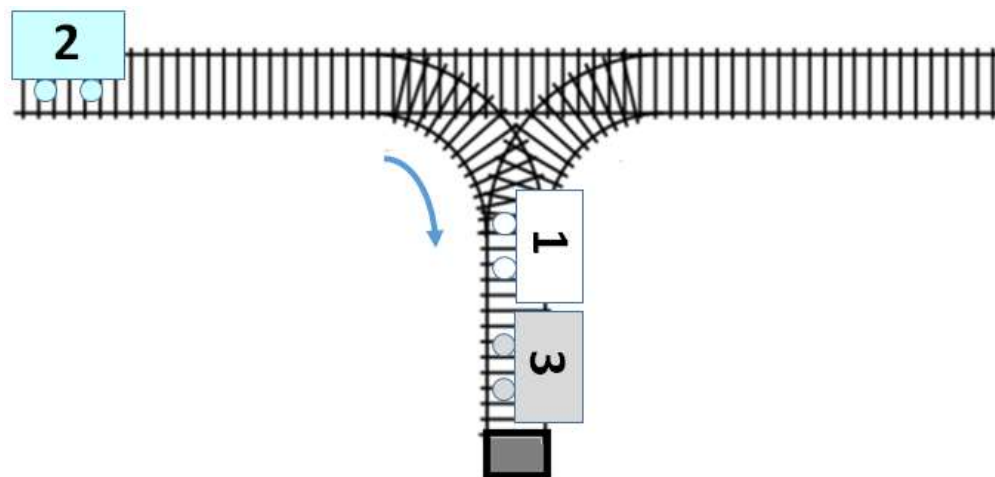
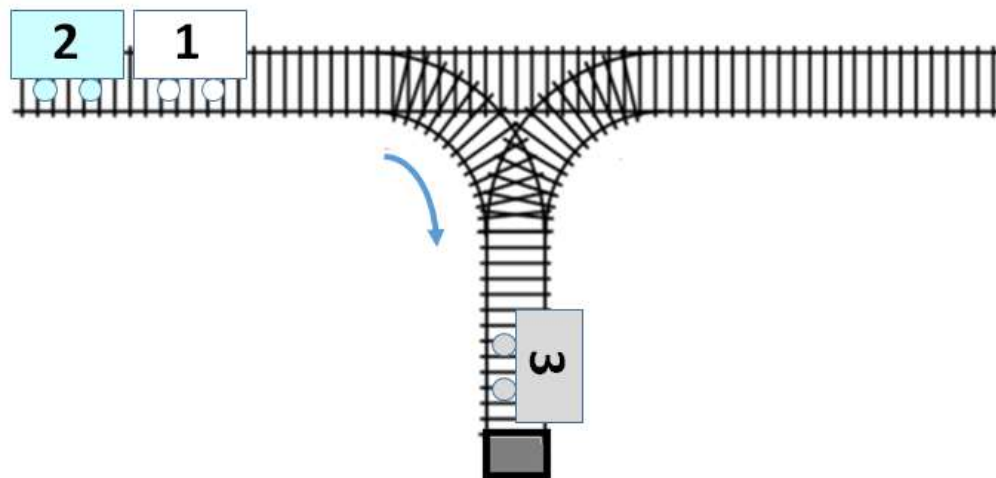
3.2 스택의 응용

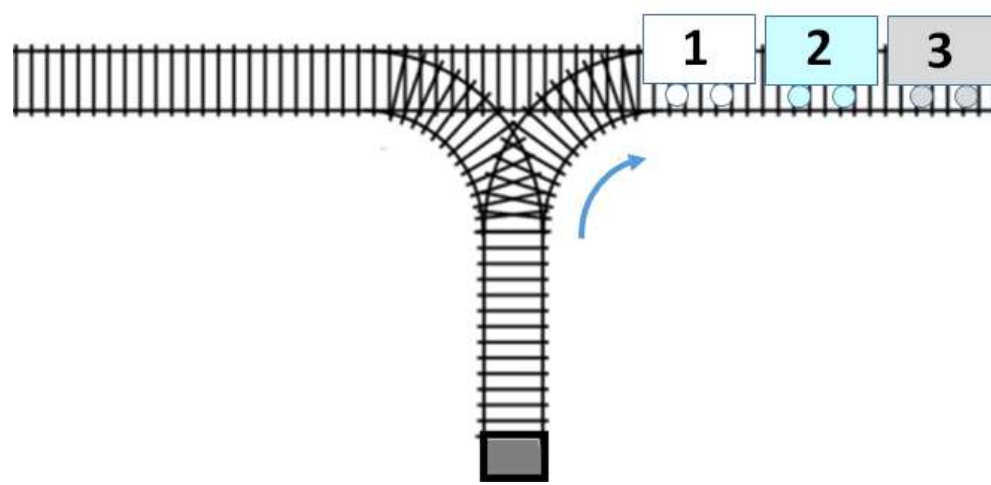
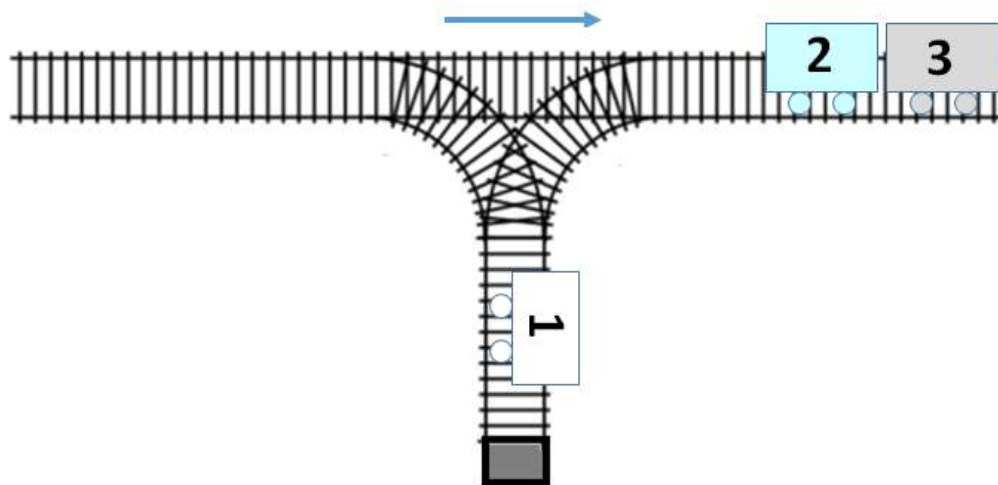
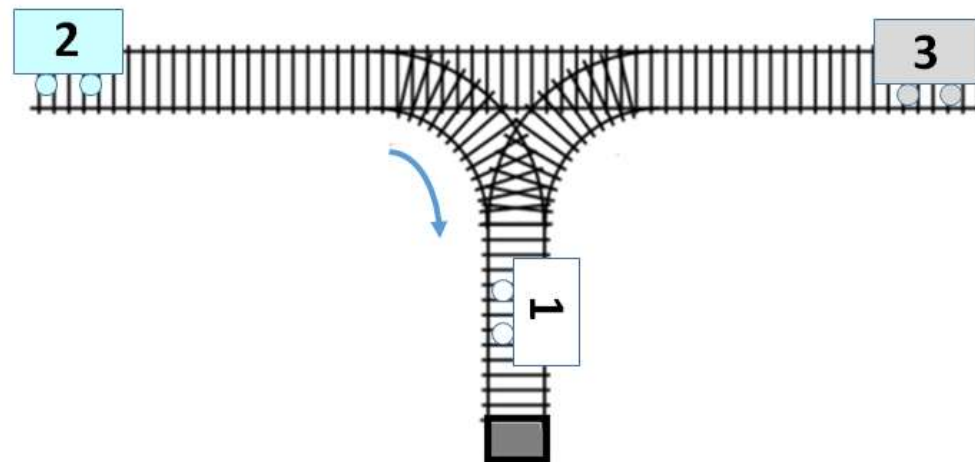
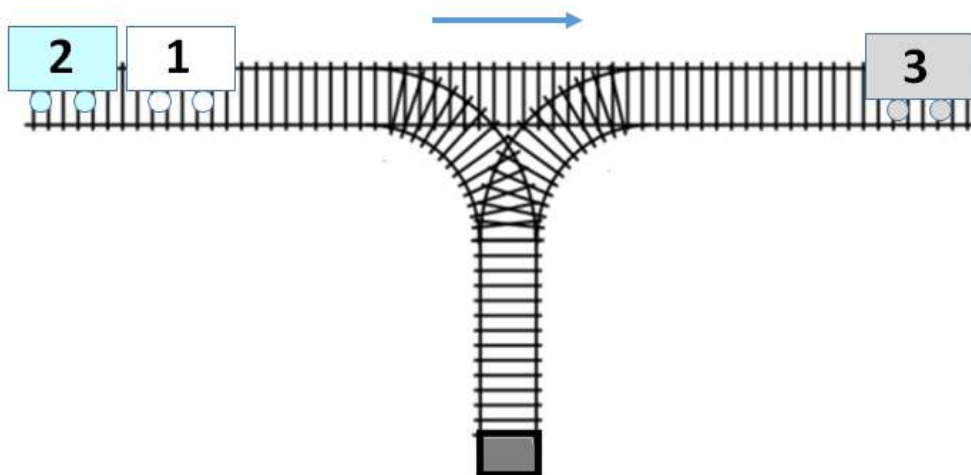
3.3 큐

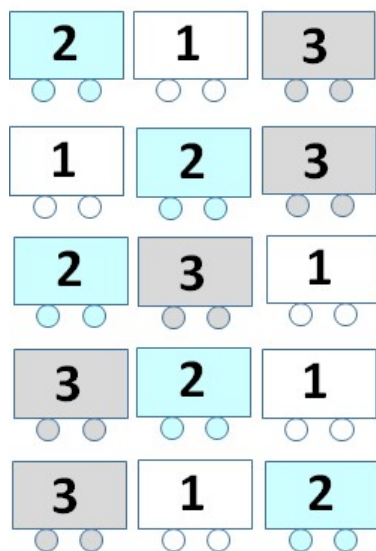
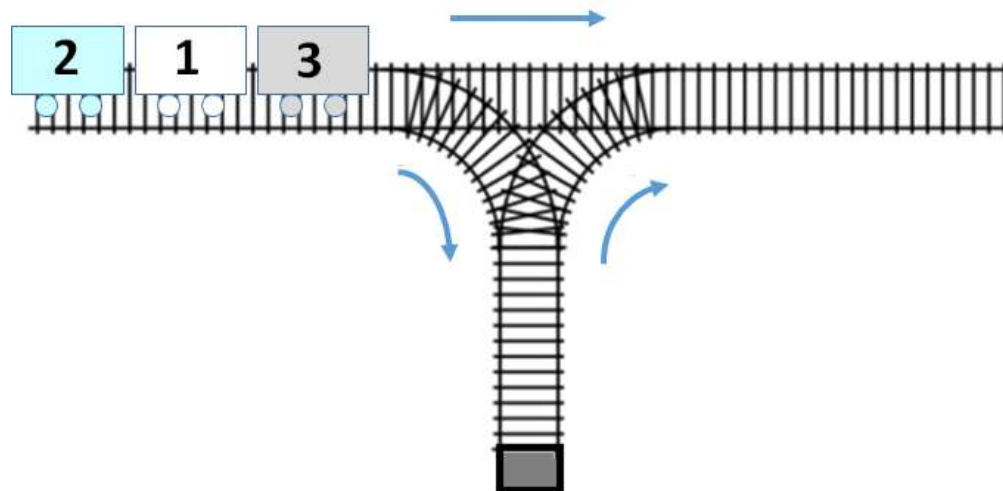
3.4 데크

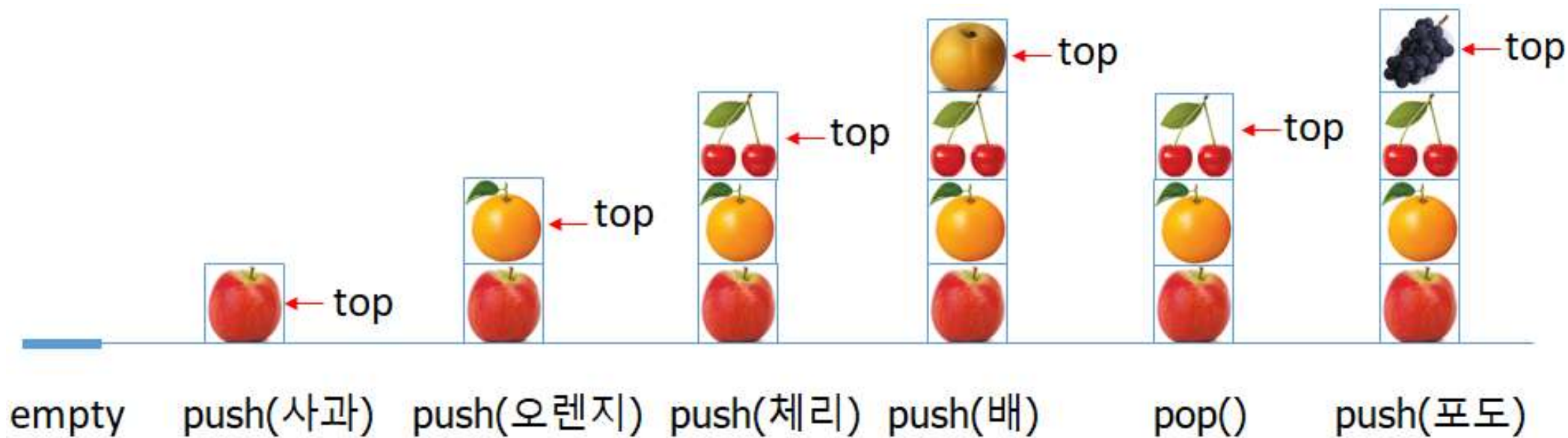
3.1 스택

- 한 쪽 끝에서만 item(항목)을 삭제하거나 새로운 item을 삽입하는 자료구조
- 새 item을 저장하는 연산: push
- Top item을 삭제하는 연산: pop
- 후입 선출(Last-In First-Out, LIFO)

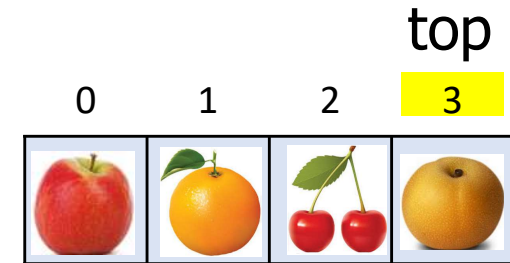
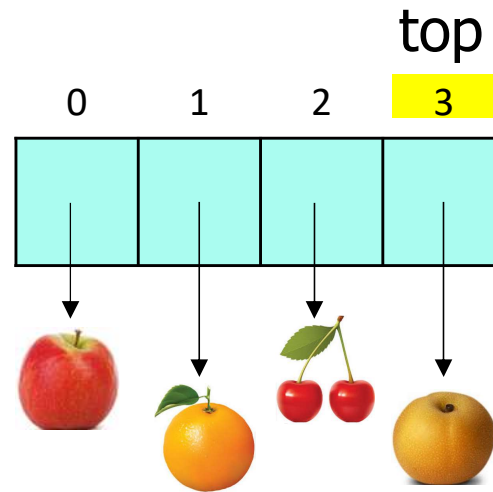
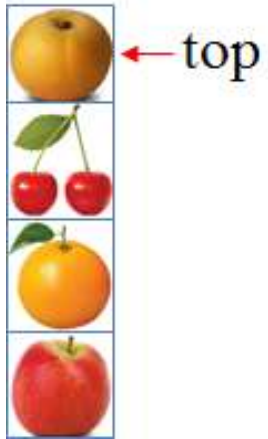








[그림 3-2] 스택의 push와 pop 연산



[그림 3-3] 리스트로 구현한 스택

리스트로 구현한 스택

```
01 def push(item): # 삽입 연산
02     stack.append(item) ●
```

push() = append()
리스트의 맨 뒤에 item 추가

```
03
04 def peek(): # top 항목 접근
05     if len(stack) != 0:
06         return stack[-1] ●
```

top 항목
= 리스트의 맨 뒤 항목 리턴

```
07
08 def pop(): # 삭제 연산
09     if len(stack) != 0:
10         item = stack.pop(-1) ●
11         return item
```

pop()
리스트의 맨 뒤에 있는 항목 제거

```
12 stack = [] ●
```

리스트 선언

```
13 push('apple')
14 push('orange')
15 push('cherry')
16 print('사과, 오렌지, 체리 push 후:\t', end='')
17 print(stack, '\t<- top')
18 print('top 항목: ', end='')
19 print(peek())
20 push('pear')
21 print('배 push 후:\t\t', end='')
22 print(stack, '\t<- top')
23 pop()
24 push('grape')
25 print('pop(), 포도 push 후:\t', end='')
26 print(stack, '\t<- top')
```

일련의 스택 연산과 출력

[프로그램 3-1]



프로그램 수행 결과

Console ✕ PyUnit

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

사과, 오렌지, 체리 push 후: ['apple', 'orange', 'cherry'] <- top

top 항목: cherry

배 push 후: ['apple', 'orange', 'cherry', 'pear'] <- top

pop(), 포도 push 후: ['apple', 'orange', 'cherry', 'grape'] <- top

단순 연결 리스트 스택

```
01 class Node: # Node 클래스
02     def __init__(self, item, link):
03         self.item = item
04         self.next = link
05
06 def push(item): # push 연산
07     global top
08     global size } — 전역 변수
09     top = Node(item, top) ● — 새 노드 객체를 생성하여
10     size += 1                연결리스트의 첫 노드로 삽입
11
12 def peek(): # peek 연산
13     if size != 0:
14         return top.item ● — top 항목만 리턴
15
```

```
16 def pop(): # pop 연산
17     global top
18     global size ] — 전역 변수
19     if size != 0:
20         top_item = top.item
21         top = top.next ● — 연결리스트에서 top이
22         size -= 1           참조하던 노드 분리시킴
23         return top_item ● — 제거된 top 항목 리턴

24 def print_stack(): # 스택 출력
25     print('top ->\t', end='')
26     p = top
27     while p:
28         if p.next != None:
29             print(p.item, '-> ', end='')
30         else:
31             print(p.item, end='')
32         p = p.next
33     print()
```

```
34 top = None ]
35 size = 0
36 push('apple')
37 push('orange')
38 push('cherry')
39 print('사과, 오렌지, 체리 push 후:\t', end='')
40 print_stack()
41 print('top 항목: ', end='')
42 print(peek())
43 push('pear')
44 print('배 push 후:\t\t', end='')
45 print_stack()
46 pop()
47 push('grape')
48 print('pop(), 포도 push 후:\t', end='')
49 print_stack()
```


초기화

비밀 거품 3-1 단 네오 함

[프로그램 3-2]



프로그램 수행 결과

Console  PyUnit

<terminated> linkedstack.py [C:\Users\wsbyang\AppData\Local\Programs\Python\Pyt

사과, 오렌지, 체리 push 후: top -> cherry -> orange -> apple

top 항목: cherry

배 push 후: top -> pear -> cherry -> orange -> apple

pop(), 포도 push 후: top -> grape -> cherry -> orange -> apple

수행 시간

- 파이썬의 리스트로 구현한 스택의 push와 pop 연산: 각각 $O(1)$ 시간

파이썬의 리스트는 크기가 동적으로 확대/축소되며, 크기 조절은 스택(리스트)의 모든 항목을 새 리스트로 복사해야 하기 때문에 $O(n)$ 시간 소요

- 단순 연결 리스트 스택의 push와 pop 연산: 각각 $O(1)$ 시간

연결 리스트의 맨 앞 부분에서 노드를 삽입하거나 삭제하기 때문

3.2 스택의 응용

- 컴파일러의 괄호 짝 맞추기
- 회문(Palindrome) 검사하기

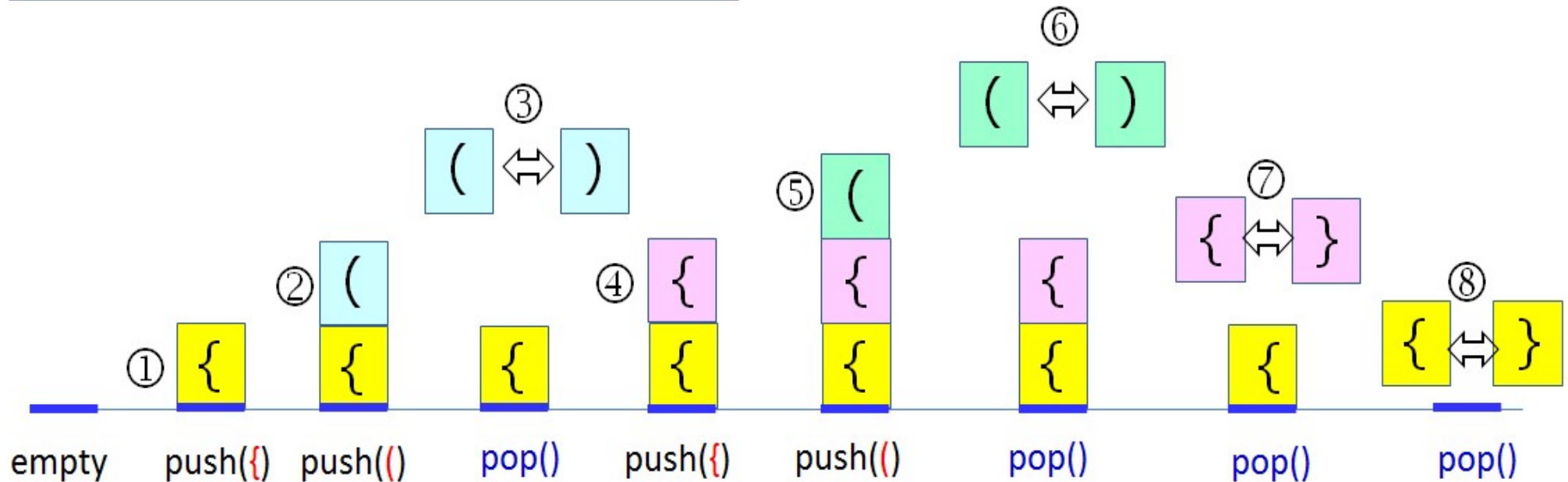
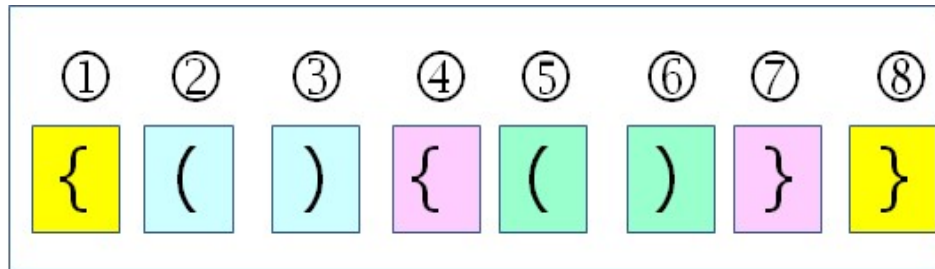
컴파일러의 괄호 짝 맞추기



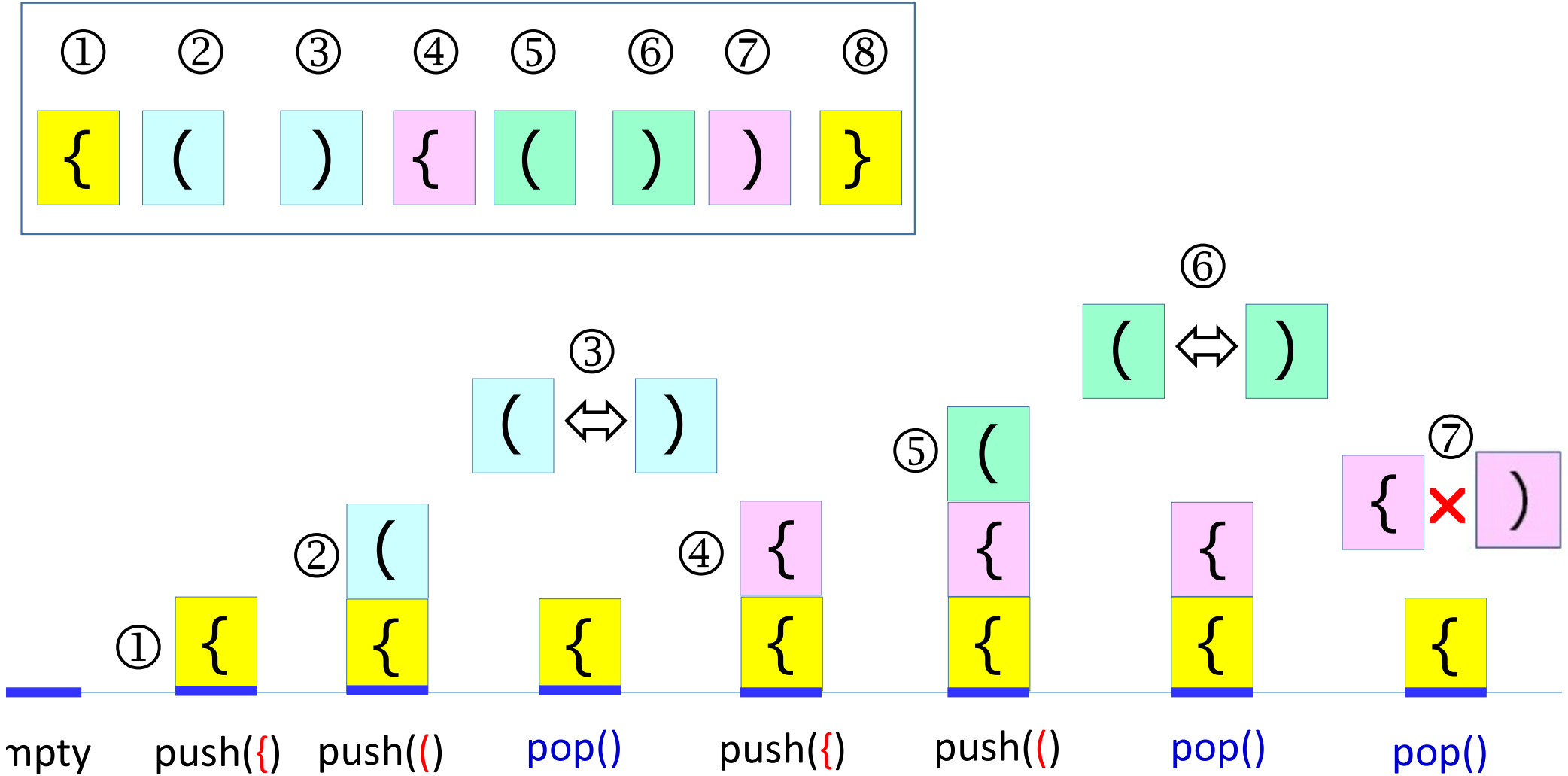
[핵심 아이디어] 왼쪽 괄호는 스택에 push, 오른쪽 괄호를 읽으면 pop 수행

- pop된 왼쪽 괄호와 바로 읽었던 오른쪽 괄호가 다른 종류이면 에러 처리, 같은 종류이면 다음 괄호를 읽음
- 모든 괄호를 읽은 뒤 에러가 없고 스택이 empty이면, 정상
- 모든 괄호를 처리한 후, 스택이 empty가 아니면 짝이 맞지 않는 괄호가 스택에 남은 것이므로 에러 처리

[예제 1] 괄호 짝 맞추기



[예제 2] 괄호 짝 맞추기



회문 검사

- 회문(Palindrome): 앞으로부터 읽으나 뒤로부터 읽으나 동일한 스트링

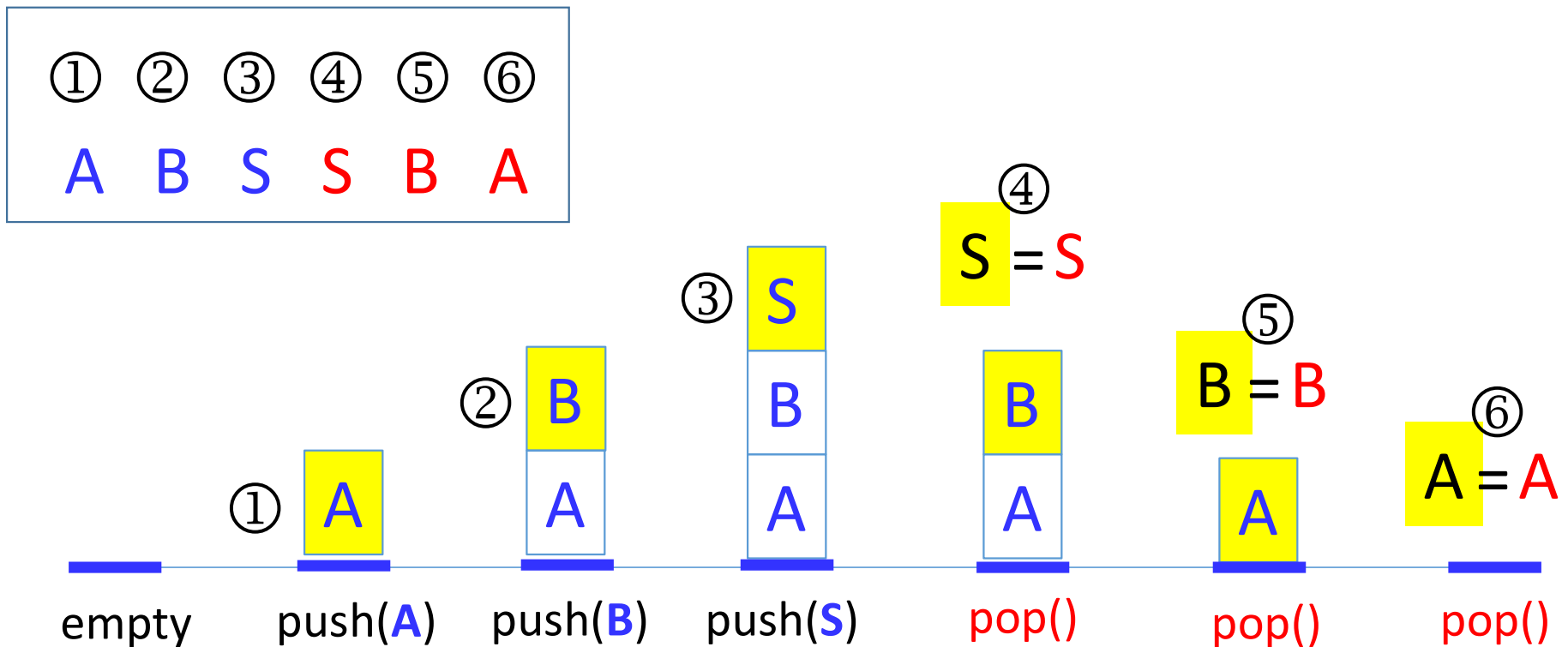


[핵심 아이디어] 전반부의 문자들을 스택에 push한 후, 후반부의 각 문자를 차례로 pop한 문자와 비교

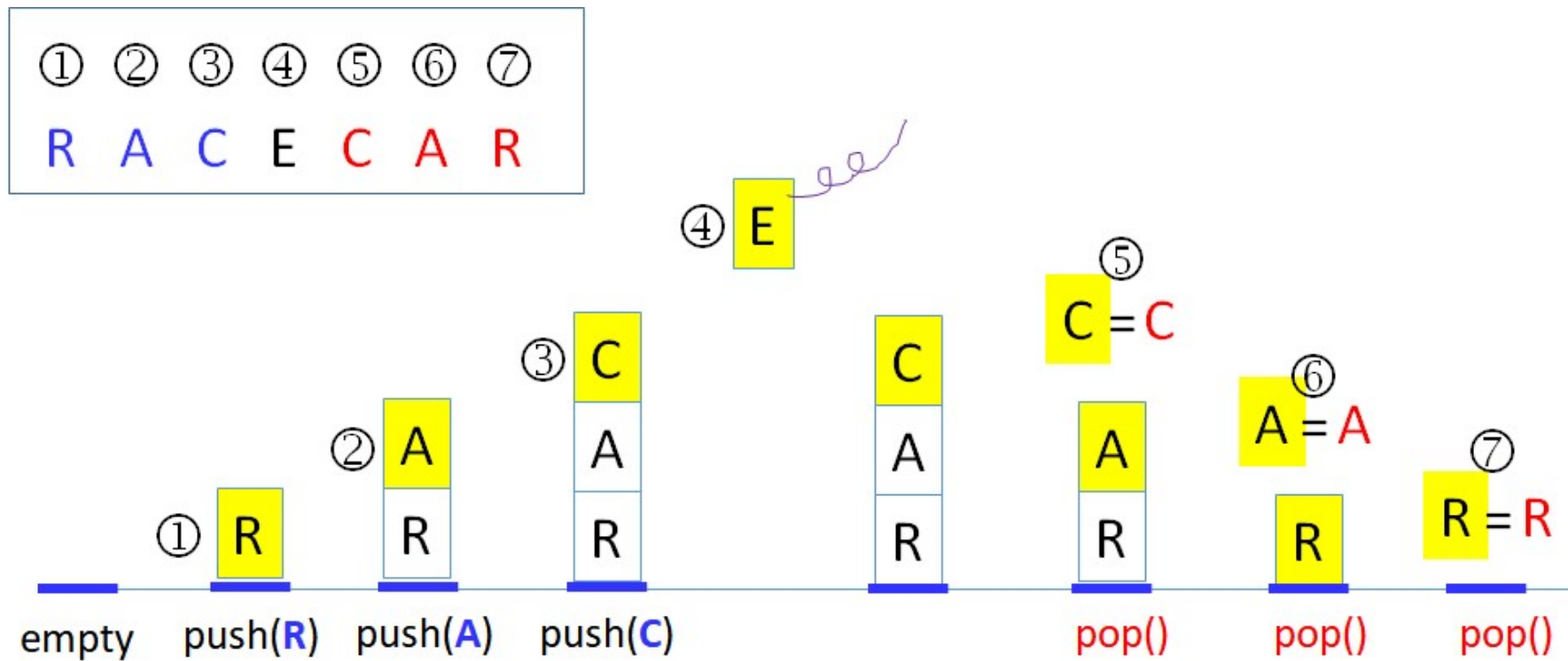
- 입력의 앞부분 1/2을 읽어 스택에 push한 후, 문자열의 길이가 짝수이면 뒷부분의 문자 1 개를 읽어서 pop한 문자와 비교하는 과정 반복
- 마지막 비교까지 두 문자가 동일하고 스택이 empty이면, 입력은 회문

- 입력의 길이가 홀수인 경우 중간 문자는 읽고 버린다.

[예제 1]



[예제 2]





스택의 기타 응용

- 후위 표기(Postfix Notation) 수식 계산하기
- 중위 표기(Infix Notation) 수식의 후위 표기법 변환
- 미로 찾기
- 트리의 순회(Part 4)
- 그래프의 DFS(Part 8)
- 프로그래밍에서 매우 중요한 함수/메소드 호출 및
순환 호출도 스택 자료구조를 바탕으로 구현

수식의 표기법

- 프로그램을 작성할 때 수식에서 $+$, $-$, $*$, $/$ 와 같은 이항 연산자는 2개의 피연산자 사이에 위치: **중위 표기(Infix Notation)**
- 컴파일러는 중위표기 수식을 **후위 표기(Postfix Notation)**으로 바꾼다.
 - 그 이유는 후위표기 수식은 괄호 없이 중위 표기 수식을 표현할 수 있기 때문
- **전위 표기(Prefix Notation)**: 연산자를 피연산자들 앞에 두는 표기

[연습문제]

중위 표기	후위 표기	전위 표기
$A + B$	$A B +$	$+ A B$
$A + B - C$	$A B + C -$	$+ A - B C$
$A + B * C - D$	$A B C * + D -$	$- + A * B C D$
$(A + B) / (C - D)$	$A B + C D - /$	$/ + A B - C D$

후위 표기 수식 계산



[핵심 아이디어] 피연산자는 스택에 push하고,
연산자는 2회 pop하여 계산한 후 push

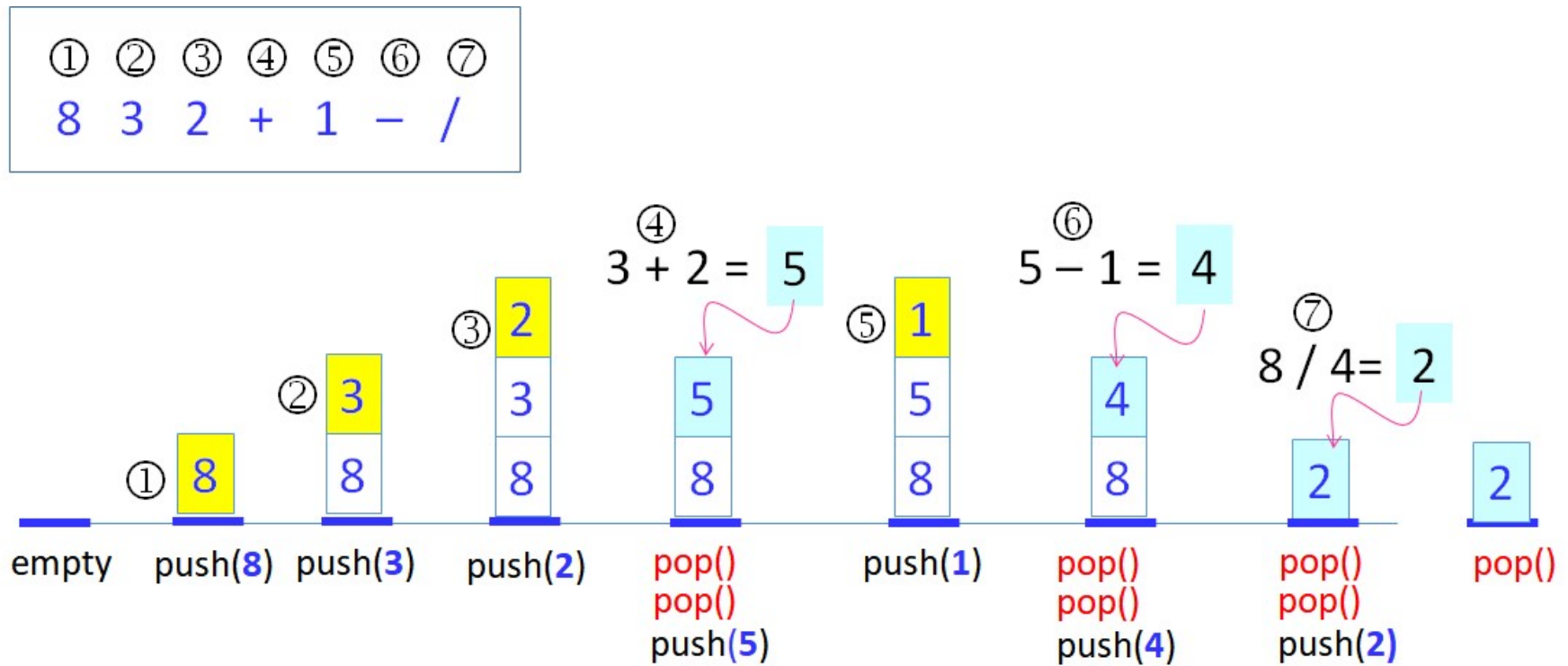
후위표기로 표현된 수식 계산 알고리즘

- 입력을 좌에서 우로 문자를 1개씩 읽고 이를 C라고하면

[1] C가 피연산자이면 C를 push

[2] C가 연산자(op)이면 pop을 2회 수행한다. 먼저 pop된 피연산자가 A이고, 나중에 pop된 피연산자가 B라면, $(A \text{ op } B)$ 를 수행하여 그 결과값을 push

[연습문제]



중위 표기 수식을 후위 표기로 변환

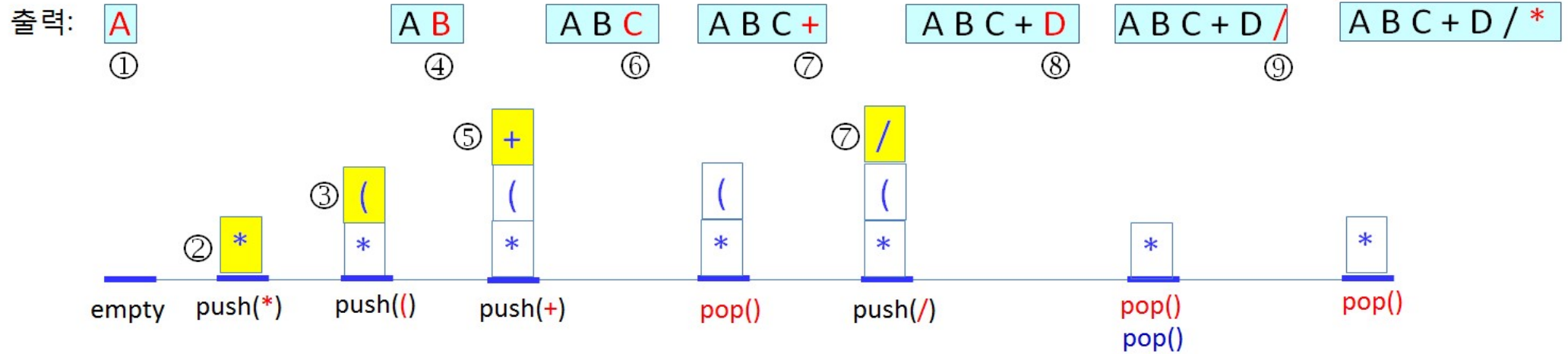


[핵심 아이디어] 왼쪽 괄호나 연산자는 스택에 push하고, 피연산자는 출력

- 입력을 좌에서 우로 문자를 1개씩 읽는다. 읽은 문자가
 1. 피연산자이면, 읽은 문자를 출력
 2. 왼쪽 괄호이면, push
 3. 오른쪽 괄호이면, 왼쪽 괄호가 나올 때까지 pop하여 출력. 단, 오른쪽이나 왼쪽 괄호는 출력하지 않음
 4. 연산자이면, 자신의 우선순위보다 낮은 연산자가 스택 top에 나타날 때까지 pop하여 출력하고 읽은 연산자를 push
- 입력을 모두 읽었으면 스택이 empty될 때까지 pop 출력

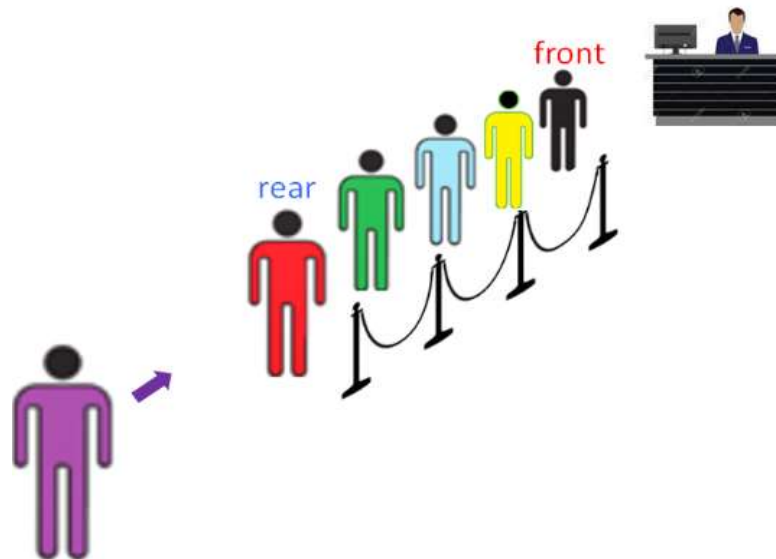
[연습문제]

① ② ③ ④ ⑤ ⑥ ⑦ ⑧ ⑨
A * (B + C / D)



3.3 큐

- 큐(Queue): 삽입과 삭제가 양 끝에서 각각 수행되는 자료구조
- 일상생활의 관공서, 은행, 우체국, 병원 등에서 번호표를 이용한 줄서기
- **선입 선출(First-In First-Out, FIFO)**



파이썬 리스트 큐

```
01 def add(item): # 삽입 연산
02     q.append(item)
```

맨 뒤에 새 항목 삽입

```
04 def remove(): # 삭제 연산
05     if len(q) != 0:
06         item = q.pop(0)
07         return item
```

맨 앞의 항목 삭제

```
09 def print_q(): # 큐 출력
10     print('front -> ', end='')
11     for i in range(len(q)):
12         print('{!s:<8}'.format(q[i]), end='')
13     print(' <- rear')
```

맨 앞부터 항목들을
차례로 출력


```
14 q = []
15 add('apple')
16 add('orange')
17 add('cherry')
18 add('pear')
19 print('사과, 오렌지, 체리, 배 삽입 후: \t', end='')
20 print_q()
21 remove()
22 print('remove한 후:\t\t', end='')
23 print_q()
24 remove()
25 print('remove한 후:\t\t', end='')
26 print_q()
27 add('grape')
28 print('포도 삽입 후:\t\t', end='')
29 print_q()
```

리스트 선언

일련의 큐 연산과 출력

[프로그램 3-3]



프로그램 수행 결과

Console PyUnit

<terminated> listqueue.py [C:\Users\sbayang\AppData\Local\Programs\Python\Python36-32\py

사과, 오렌지, 체리, 배 삽입 후: front -> apple orange cherry pear <- rear

remove한 후: front -> orange cherry pear <- rear

remove한 후: front -> cherry pear <- rear

포도 삽입 후: front -> cherry pear grape <- rear

단순 연결 리스트 큐

```
01 class Node:
02     def __init__(self, item, n):
03         self.item = item
04         self.next = n
05 def add(item): # 삽입 연산
06     global size
07     global front
08     global rear
09     new_node = Node(item, None)
10     if size == 0:
11         front = new_node
12     else:
13         rear.next = new_node
14     rear = new_node
15     size += 1
```

노드 생성자
항목과 다음 노드 레퍼런스

전역 변수

새 노드 객체를 생성

연결리스트의 맨 뒤에 삽입

```
16 def remove(): # 삭제 연산
```

```
17     global size
```

```
18     global front
```

```
19     global rear
```

전역 변수

```
20     if size != 0:
```

```
21         fitem = front.item
```

```
22         front = front.next
```

연결리스트에서 front가
참조하던 노드 분리시킴

```
23         size -= 1
```

```
24         if size == 0:
```

```
25             rear = None
```

```
26         return fitem
```

제거된 맨 앞의 항목 리턴

```

27 def print_q(): # 큐 출력
28     p = front
29     print('front: ', end='')
30     while p:
31         if p.next != None:
32             print(p.item, '-> ', end='')
33         else:
34             print(p.item, end = ' ')
35         p = p.next
36     print(' : rear')
37 front = None
38 rear = None
39 size = 0
40
54

```

단순연결리스트(스택)의 항목을 차례로 출력

초기화

[프로그램 3-3]의 line 15~29와 동일

[프로그램 3-4]



프로그램 수행 결과

Console PyUnit

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

```
사과, 오렌지, 체리, 배 삽입 후: front: apple -> orange -> cherry -> pear : rear
remove한 후:                 front: orange -> cherry -> pear : rear
remove한 후:                 front: cherry -> pear : rear
포도 삽입 후:                front: cherry -> pear -> grape : rear
```



큐 자료구조의 응용

- CPU의 태스크 스케줄링(Task Scheduling)
- 네트워크 프린터
- 실시간(Real-time) 시스템의 인터럽트(Interrupt) 처리
- 다양한 이벤트 구동 방식(Event-driven) 컴퓨터 시뮬레이션
- 콜 센터의 전화 서비스 처리 등
- Part 4의 이진 트리의 레벨 순회(Level-order Traversal)
- Part 8 그래프에서 너비 우선 탐색(Breath-First Search) 등

수행 시간

- 리스트로 구현한 큐의 add와 remove 연산: 각각 $O(1)$ 시간
 - 리스트 크기를 확대/축소시키는 경우에 큐의 모든 항목을 새 리스트에 복사해야 하므로 $O(n)$ 시간
- 단순 연결 리스트 큐의 add와 remove 연산은 각각 $O(1)$ 시간
 - 삽입 또는 삭제 연산이 rear 와 front로 인해 연결 리스트의 다른 노드를 방문할 필요 없음

3.4 데크

- **데크(Double-ended Queue, Deque)**: 양쪽 끝에서 삽입과 삭제를 허용하는 자료구조
- 데크는 스택과 큐 자료구조를 혼합한 자료구조
- 따라서 데크는 스택과 큐를 동시에 구현하는데 사용

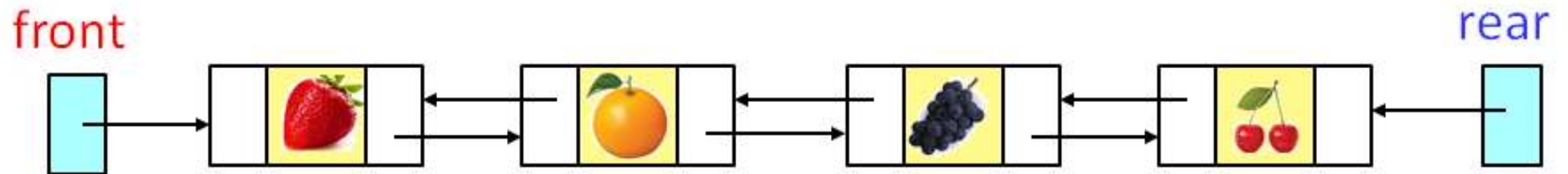




데크의 응용

- 스크롤(Scroll)
- 문서 편집기 등의 undo 연산
- 웹 브라우저의 방문 기록 등
 - 웹 브라우저 방문 기록의 경우, 최근 방문한 웹 페이지 주소는 앞에 삽입하고, 일정 수의 새 주소들이 앞쪽에서 삽입되면 뒤에서 삭제가 수행

- 데크를 이중 연결 리스트로 구현하는 것이 편리
- 단순 연결 리스트는 노드의 이전 노드의 레퍼런스를 알아야 삭제



- 파이썬에는 데크가 Collections 패키지에 정의되어 있음
- 삽입, 삭제 등의 연산은 파이썬의 리스트의 연산과 매우 유사

```
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



프로그램 수행 결과

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'])

수행 시간

- 데크를 배열이나 이중 연결 리스트로 구현한 경우, 스택과 큐의 수행 시간과 동일
- 양 끝에서 삽입과 삭제가 가능하므로 프로그램이 다소 복잡
- 이중 연결 리스트로 구현한 경우는 더 복잡함



요약

- **스택**은 한 쪽 끝에서만 item을 삭제하거나 새로운 item을 저장하는 **후입 선출(LIFO)** 자료구조
- 스택은 컴파일러의 괄호 짝 맞추기, 회문 검사하기, 후위 표기 수식 계산하기, 중위 표기 수식을 후위 표기로 변환하기, 미로 찾기, 트리의 노드 방문, 그래프의 DFS에 사용. 또한 프로그래밍에서 매우 중요한 메소드 호출 및 순환 호출도 스택 자료구조를 바탕으로 구현
- **큐**는 삽입과 삭제가 양 끝에서 각각 수행되는 **선입 선출(FIFO)** 자료구조

- 큐는 CPU의 태스크 스케줄링, 네트워크 프린터, 실시간 시스템의 인터럽트 처리, 다양한 이벤트 구동 방식 컴퓨터 시뮬레이션, 콜 센터의 전화 서비스 처리 등에 사용되며, 이진트리의 레벨순회와 그래프의 BFS에 사용
- **데크**는 양쪽 끝에서 삽입과 삭제를 허용하는 자료구조로서 스택과 큐 자료구조를 혼합한 자료구조
- 데크는 스크롤, 문서 편집기의 undo 연산, 웹 브라우저의 방문 기록 등에 사용

스택, 큐, 데크의 수행 시간

자료구조	구현	삽입	삭제	비고
스택 큐 데크	*파이썬 리스트	$O(1)$	$O(1)$	*타 언어의 배열
	연결 리스트 [†]	$O(1)$	$O(1)$	[†] 데크는 이중 연결 리스트로 구현

끝