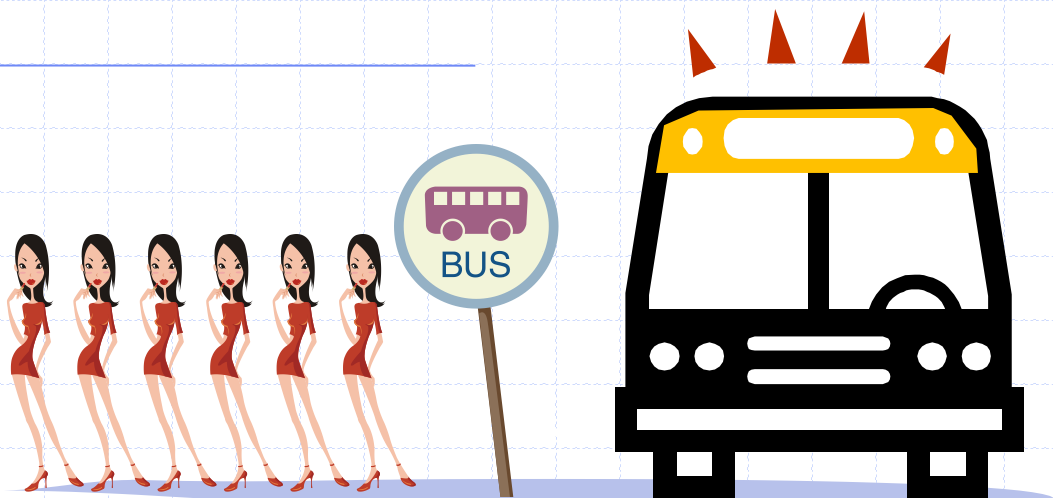


큐

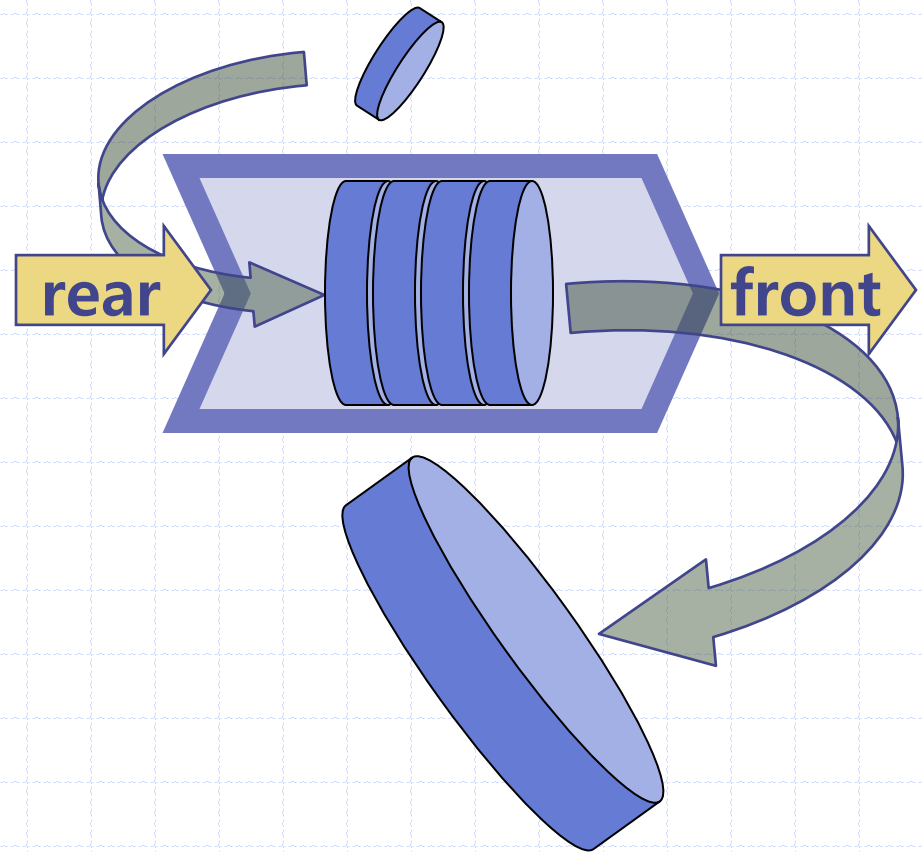


Outline

- ◆ 7.1 큐 ADT
- ◆ 7.2 큐 ADT 메소드
- ◆ 7.3 큐 ADT 구현
- ◆ 7.4 데크 ADT
- ◆ 7.5 응용문제

큐 ADT

- ◆ 큐 ADT는 임의의 개체들을 저장
- ◆ 삽입과 삭제는 **선입선출**(First-In First-Out, FIFO) 순서를 따른다
- ◆ 삽입은 큐의 **뒤**(rear), 삭제는 큐의 **앞**(front)이라 불리는 위치에서 수행



큐 ADT 메소드

◆ 주요 큐 메소드

- `enqueue(e)`: 큐의 뒤에 원소를 삽입
- `element dequeue()`: 큐의 앞에서 원소를 삭제하여 반환

◆ 보조 큐 메소드

- `element front()`: 큐의 앞에 있는 원소를 (삭제하지 않고) 반환
- `integer size()`: 큐에 저장된 원소의 수를 반환
- `boolean isEmpty()`: 큐가 비어 있는지 여부를 반환
- `iterator elements()`: 큐 원소 전체를 반환

◆ 예외

- `emptyQueueException()`: 비어 있는 큐에 대해 삭제 또는 `front`를 수행 시도할 경우 발령
- `fullQueueException()`: 만원 큐에 대해 삽입을 수행 시도할 경우 발령



큐 응용

◆ 직접 응용

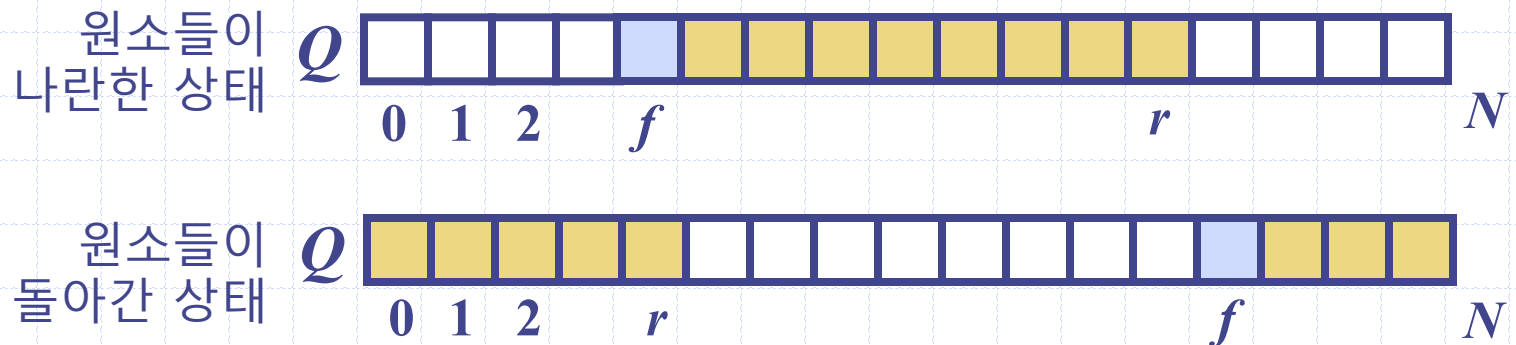
- 대기열, 관료적 체제
- 공유자원에 대한 접근, 예를 들어 프린터
- 멀티프로그래밍

◆ 간접 응용

- 알고리즘 수행을 위한 보조 데이터구조
- 다른 데이터구조를 구성하는 요소

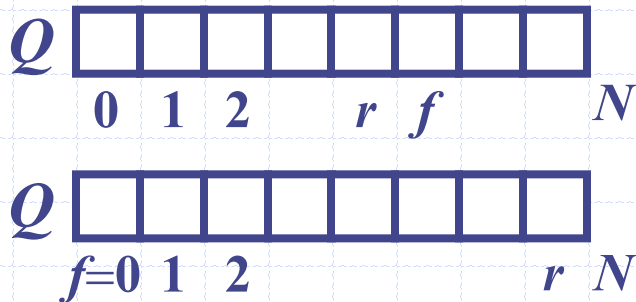
배열에 기초한 큐

- ◆ 크기 N 의 배열을 원형으로 사용
 - 선형배열을 사용하면 비효율적임
- ◆ 두 개의 변수를 사용하여 front와 rear 위치를 기억
 - f : front 원소의 첨자
 - r : rear 원소의 첨자
- ◆ 참고: f 가 front 원소가 저장된 위치의 한 셀 앞을 가리키도록 정의하는 방식도 가능 - 단, 이에 상응하여 큐 관련 메소드 수정 필요
- ◆ 빈 큐를 만원 큐로부터 차별하기 위해:
 - 한 개의 빈 방을 예비
 - 대안: 원소 개수를 유지

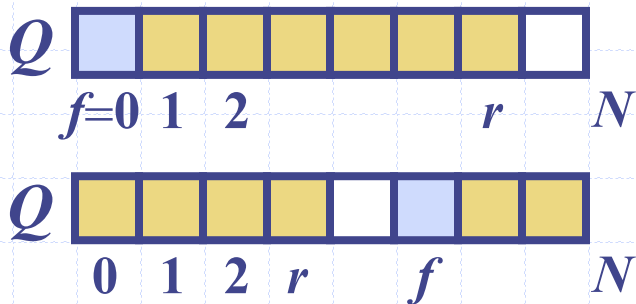


빈 큐 vs. 만원 큐

빈 큐



만원 큐



Alg **isEmpty()**

input queue Q , size N , front f ,
rear r

output boolean

1. return $(r + 1) \% N = f$

Alg **isFull()**

input queue Q , size N , front f ,
rear r

output boolean

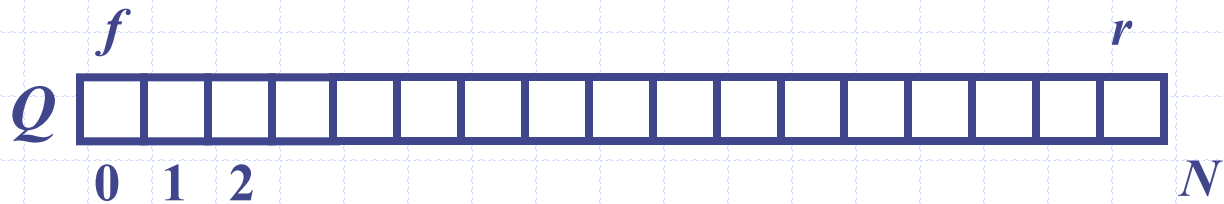
1. return $(r + 2) \% N = f$

초기화

- ◆ 초기에는 큐에 아무 원소도 없다

```
Alg initQueue()  
  input queue  $Q$ , size  $N$ , front  $f$ ,  
        rear  $r$   
  output an empty queue  $Q$ 
```

```
1.  $f \leftarrow 0$   
2.  $r \leftarrow N - 1$   
3. return
```



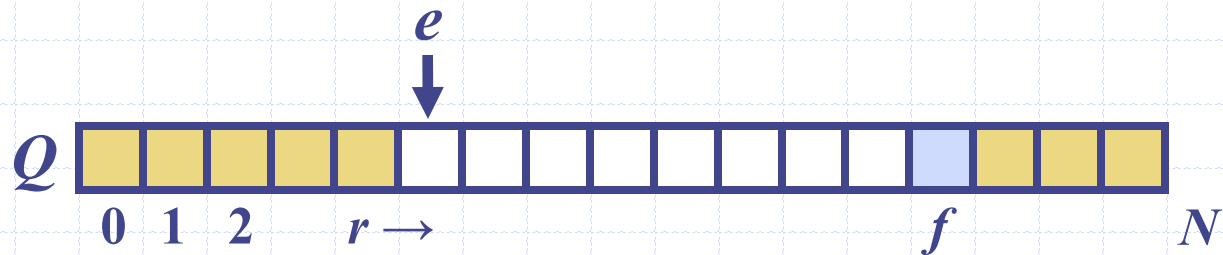
삽입

- ◆ 큐가 만원인 경우, **enqueue** 작업은 **fullQueueException**을 발령

- 배열에 기초한 구현의 한계
- 구현상의 오류일 뿐, **큐** ADT 취급 상 논리적 오류는 아님

Alg *enqueue*(*e*)
input queue *Q*, size *N*, front *f*,
rear *r*, element *e*
output none

1. **if** (*isFull*())
 fullQueueException()
2. $r \leftarrow (r + 1) \% N$
3. $Q[r] \leftarrow e$
4. **return**



삭제

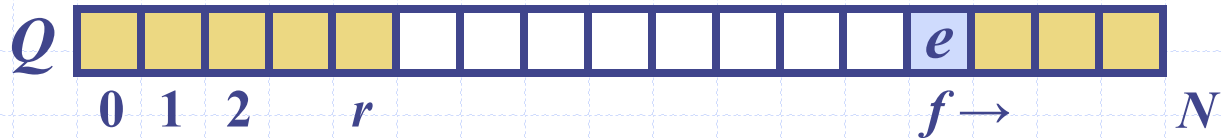
- ◆ 큐가 빈 경우, **dequeue** 작업은 **emptyQueueException**을 발령
 - 큐 ADT 취급 상 논리적 오류

Alg *dequeue()*

input queue Q , size N , front f ,
rear r

output element

1. **if** (*isEmpty()*)
 emptyQueueException()
2. $e \leftarrow Q[f]$
3. $f \leftarrow (f + 1) \% N$
4. **return** e



보조 메소드

- integer **size()**: 큐에 저장된 원소의 수를 반환
- element **front()**: 큐의 **front**에 있는 원소를 삭제하지 않고 반환

Alg **size()**

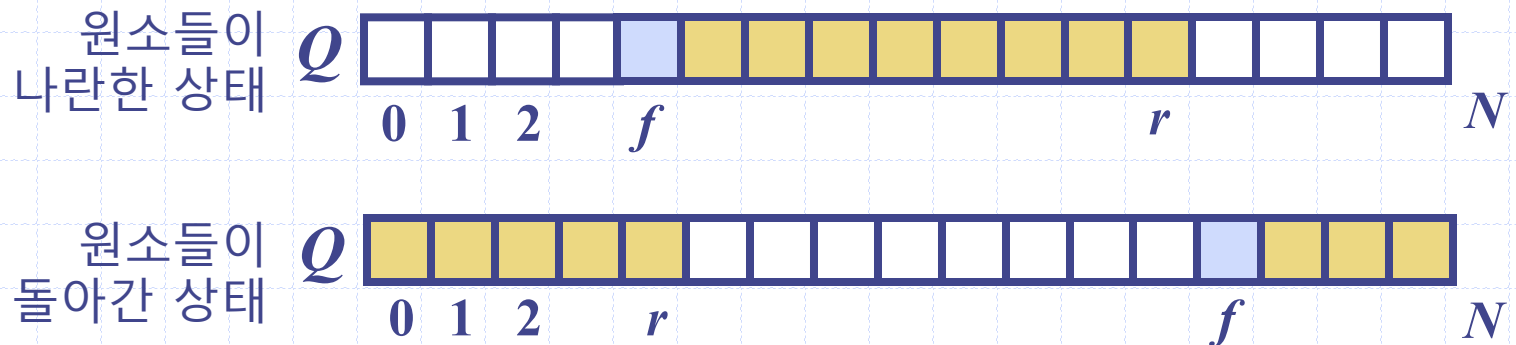
1. return $(N - f + r + 1) \% N$

Alg **front()**

1. if (*isEmpty()*)

emptyQueueException()

2. return $Q[f]$



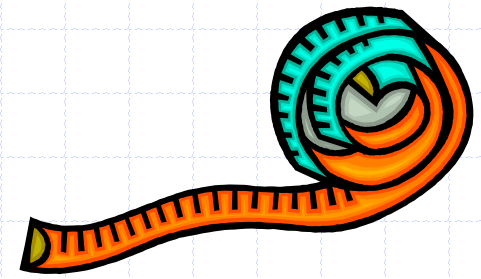
성능과 제약

◆ 성능

- 큐의 원소 개수를 n 이라 하면,
- 기억장소 사용: $O(n)$
- 각 작업의 실행시간: $O(1)$

◆ 제약

- 큐의 최대 크기를 예측할 수 있어야 하며 이 값은 실행 중 변경할 수 없다 (예외: 동적 할당)
- 만원 큐에 새로운 원소를 enqueue 시도할 경우 구현상의 오류를 일으킨다

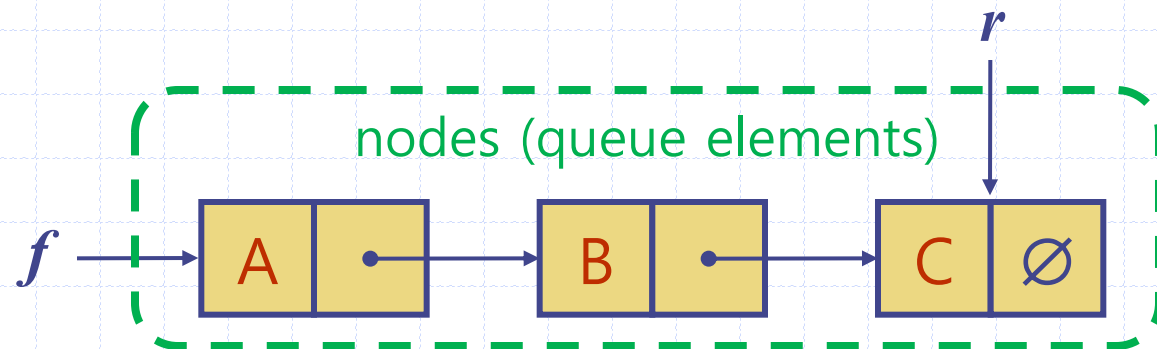


크기 기억

- ◆ 방 한 개를 예비하는 방식에 대한 대안
 - 변수 n 을 사용하여 큐에 저장된 원소 개수를 직접 관리
 - 이렇게 하면 큐에 할당된 기억장소 남김없이 활용 가능
- ◆ 필요한 코드 변경
 - `size`를 단순히 현재의 n 값을 반환하도록 수정
 - `isEmpty`와 `isFull`을 n 에 대한 간단한 테스트로 대체
 - `initQueue`, `enqueue`, `dequeue`는 n 값을 관리하도록 수정

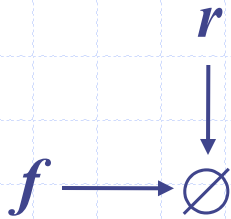
연결리스트에 기초한 큐

- ◆ 단일연결리스트를 사용하여 큐 구현 가능
 - 삽입과 삭제가 특정위치에서만 수행되므로, 역방향링크는 불필요 (참고: 스택의 경우 헤더노드 불필요)
- ◆ front 원소를 연결리스트의 첫 노드에, rear 원소를 끝 노드에 저장하고 f 와 r 로 각각의 노드를 가리키게 한다
- ◆ 기억장소 사용: $O(n)$
- ◆ 큐 ADT의 각 작업: $O(1)$



초기화

- ◆ 초기에는 아무 노드도 없다



Alg *initQueue()*

input front f , rear r

output an empty queue with front f and rear r

1. $f, r \leftarrow \emptyset$
2. **return**

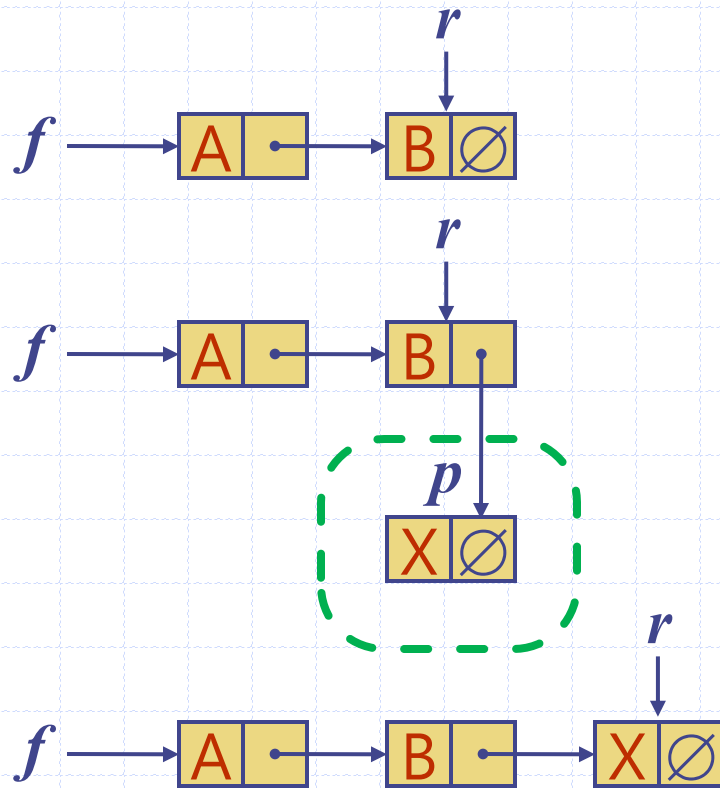
Alg *isEmpty()*

input front f , rear r

output boolean

1. **return** $f = \emptyset$ {or, $r = \emptyset$ }

삽입



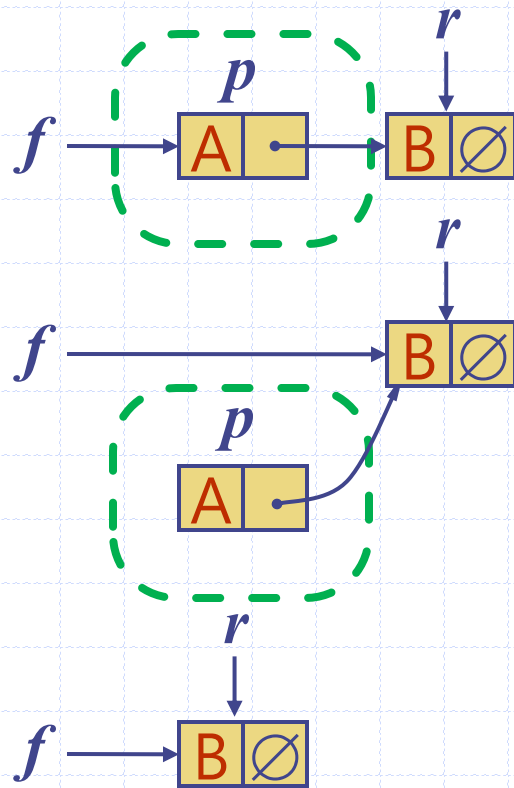
Alg **enqueue(e)**

input front f , rear r , element e

output none

1. $p \leftarrow \text{getnode}()$
2. $p.\text{elem} \leftarrow e$
3. $p.\text{next} \leftarrow \emptyset$
4. **if** ($\text{isEmpty}()$)
 $f, r \leftarrow p$
 else
 $r.\text{next} \leftarrow p$
 $r \leftarrow p$
5. **return**

삭제



Alg **dequeue()**

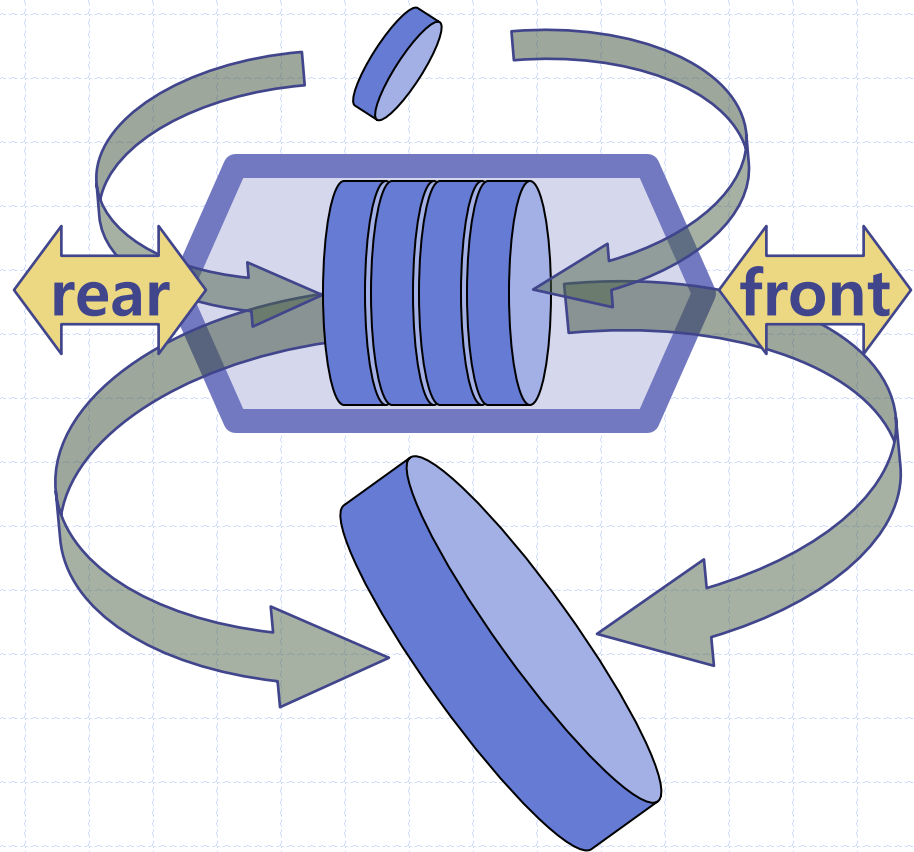
input front f , rear r

output element

1. if ($isEmpty()$)
 $emptyQueueException()$
2. $e \leftarrow f.elem$
3. $p \leftarrow f$
4. $f \leftarrow f.next$
5. if ($f = \emptyset$)
 $r \leftarrow \emptyset$
6. $putnode(p)$
7. return e

데크 ADT

- ◆ 데크 ADT는 임의의 개체들을 저장
- ◆ 데크(double-ended queue, **deque**)는 스택과 큐의 합체 방식으로 작동
- ◆ 삽입과 삭제는 앞(front)과 뒤(rear)라 불리는 양쪽 끝 위치에서 이루어진다



데크 ADT 메소드

◆ 주요 메소드

- **push**(e): front 위치에 원소를 삽입
- **element pop**(): front 위치의 원소를 삭제하여 반환
- **inject**(e): rear 위치에 원소를 삽입
- **element eject**(): rear 위치의 원소를 삭제하여 반환

◆ 보조 메소드

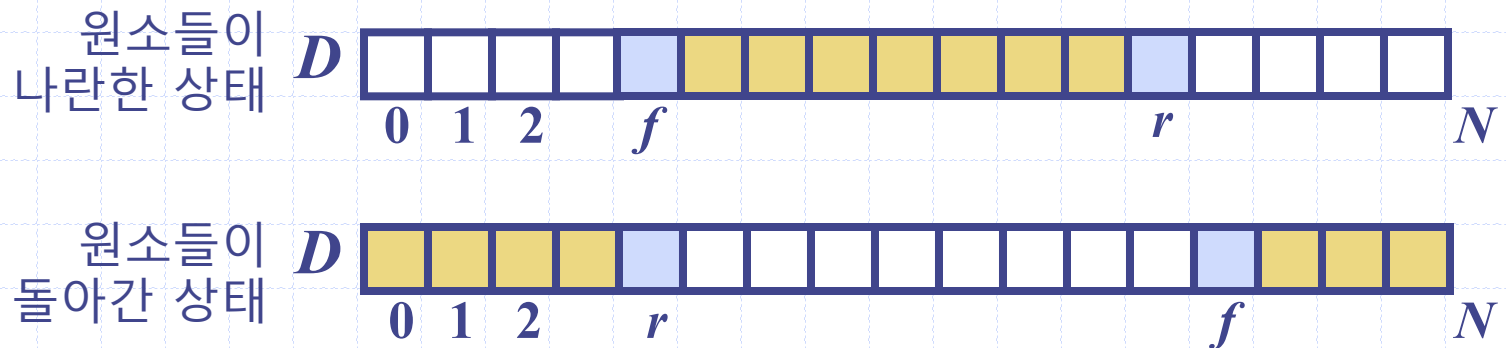
- **element front**(): front 위치의 원소를 반환
- **element rear**(): rear 위치의 원소를 반환
- **integer size**(): 데크에 저장된 원소의 수를 반환
- **boolean isEmpty**(): 데크가 비어 있는지 여부를 반환

◆ 예외

- **emptyDequeException**(): 비어 있는 데크로부터 삭제를 시도할 경우 발령
- **fullDequeException**(): 만원인 데크에 대해 삽입을 시도할 경우 발령

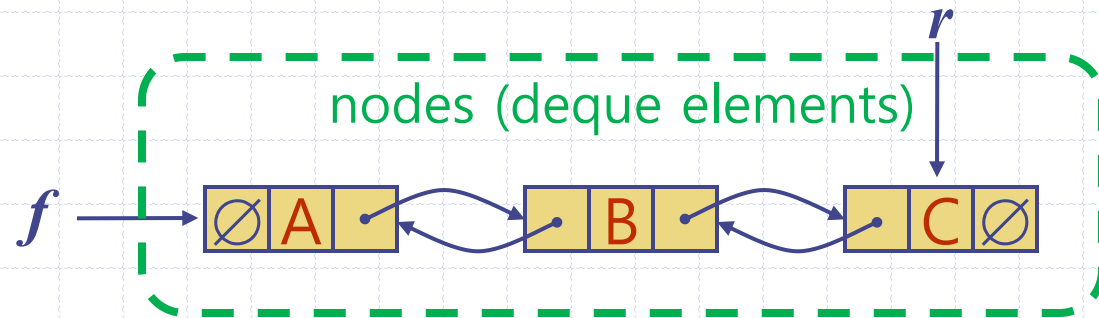
배열에 기초한 데크

- ◆ 크기 N 의 배열을 원형으로 사용
 - 선형배열을 사용하면 비효율적
- ◆ 두 개의 변수를 사용하여 front와 rear 위치를 관리
 - f : front 원소의 첨자
 - r : rear 원소의 첨자
- ◆ 빈 큐를 만원 큐로부터 차별하기 위해:
 - 한 개의 빈 방을 예비
 - 대안: 원소 개수를 기억



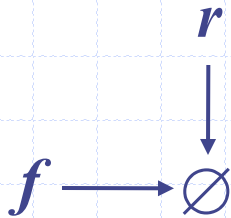
연결리스트에 기초한 데크

- ◆ 이중연결리스트를 사용하여 데크 구현 가능
 - 삽입과 삭제가 특정위치에서만 수행되므로, 특별노드는 불필요
- ◆ front 원소를 연결리스트의 첫 노드에, rear 원소를 끝 노드에 저장하고 각각의 노드를 f 와 r 로 가리키게 한다
- ◆ 기억장소 사용은 $O(n)$ 이며, 데크 ADT의 각 작업은 $O(1)$ 시간에 수행



초기화

- ◆ 초기에는 아무 노드도 없다



Alg *initQueue()*

input front f , rear r

output an empty queue with front f and rear r

1. $f, r \leftarrow \emptyset$
2. **return**

Alg *isEmpty()*

input front f , rear r

output boolean

1. **return** $f = \emptyset$ {or, $r = \emptyset$ }

삽입

Alg *add_rear*(*e*)
input front *f*, rear *r*, element *e*
output none

1. $p \leftarrow \text{getnode}()$
2. $p.\text{elem} \leftarrow e$
3. $p.\text{next} \leftarrow \emptyset$
4. $p.\text{prev} \leftarrow \emptyset$
5. **if** (*isEmpty*())
 $f, r \leftarrow p$
 else
 $p.\text{prev} \leftarrow r$
 $r.\text{next} \leftarrow p$
 $r \leftarrow p$
5. **return**

삽입

Alg *add_front*(*e*)

input front *f*, rear *r*, element *e*

output none

1. $p \leftarrow \text{getnode}()$
2. $p.\text{elem} \leftarrow e$
3. $p.\text{next} \leftarrow \emptyset$
4. $p.\text{prev} \leftarrow \emptyset$
5. **if** (*isEmpty*())
 $f, r \leftarrow p$
 else
 $p.\text{next} \leftarrow f$
 $f.\text{prev} \leftarrow p$
 $f \leftarrow p$
5. **return**

삭제

Alg *delete_front()*
input front *f*, rear *r*
output element

1. **if** (*isEmpty()*)
 emptyQueueException()
2. *e* \leftarrow *f.elem*
3. *p* \leftarrow *f*
4. *f* \leftarrow *f.next*
5. **if** (*f* = \emptyset)
 r \leftarrow \emptyset
 else
 f.prev \leftarrow \emptyset
6. *putnode(p)*
7. **return** *e*

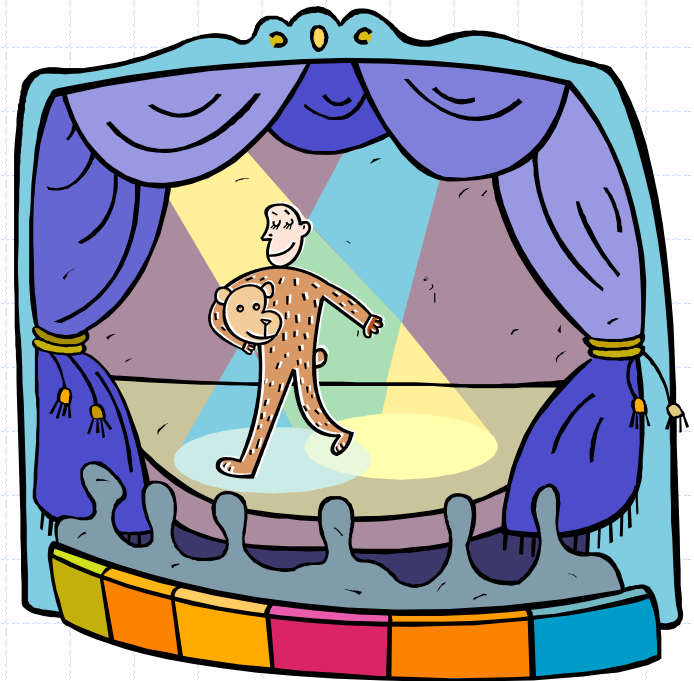
삭제

Alg *delete_rear()*
input front *f*, rear *r*
output element

1. **if** (*isEmpty()*)
 emptyQueueException()
2. *e* \leftarrow *f.elem*
3. *p* \leftarrow *r*
4. *r* \leftarrow *r.prev*
5. **if** (*r* = \emptyset)
 f \leftarrow \emptyset
 else
 r.next \leftarrow \emptyset
6. *putnode(p)*
7. **return** *e*

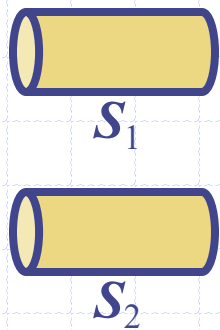
응용문제

- ◆ 두 개의 흥미로운 설계 문제를 통해 어떻게 스택과 큐가 서로를 위한 보조 데이터구조로 각각 사용되는지 공부한다
- ◆ 설계 문제
 - 두 개의 스택으로 큐 만들기
 - 두 개의 큐로 스택 만들기



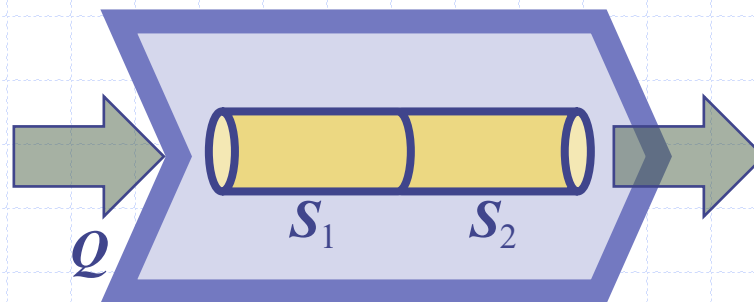
응용문제: 두 개의 스택으로 큐 만들기

- ◆ 두 개의 일반 스택을 이용하여 어떻게 큐 ADT를 구현할지 설명하라
- ◆ 전제: 주어진 스택들은 `isEmpty`, `top`, `push`, `pop` 등의 기본 메소드들을 상수시간에 수행(`size` 메소드는 지원하지 않음)
- ◆ 주의: 큐 ADT에는 중복 원소들의 저장이 가능

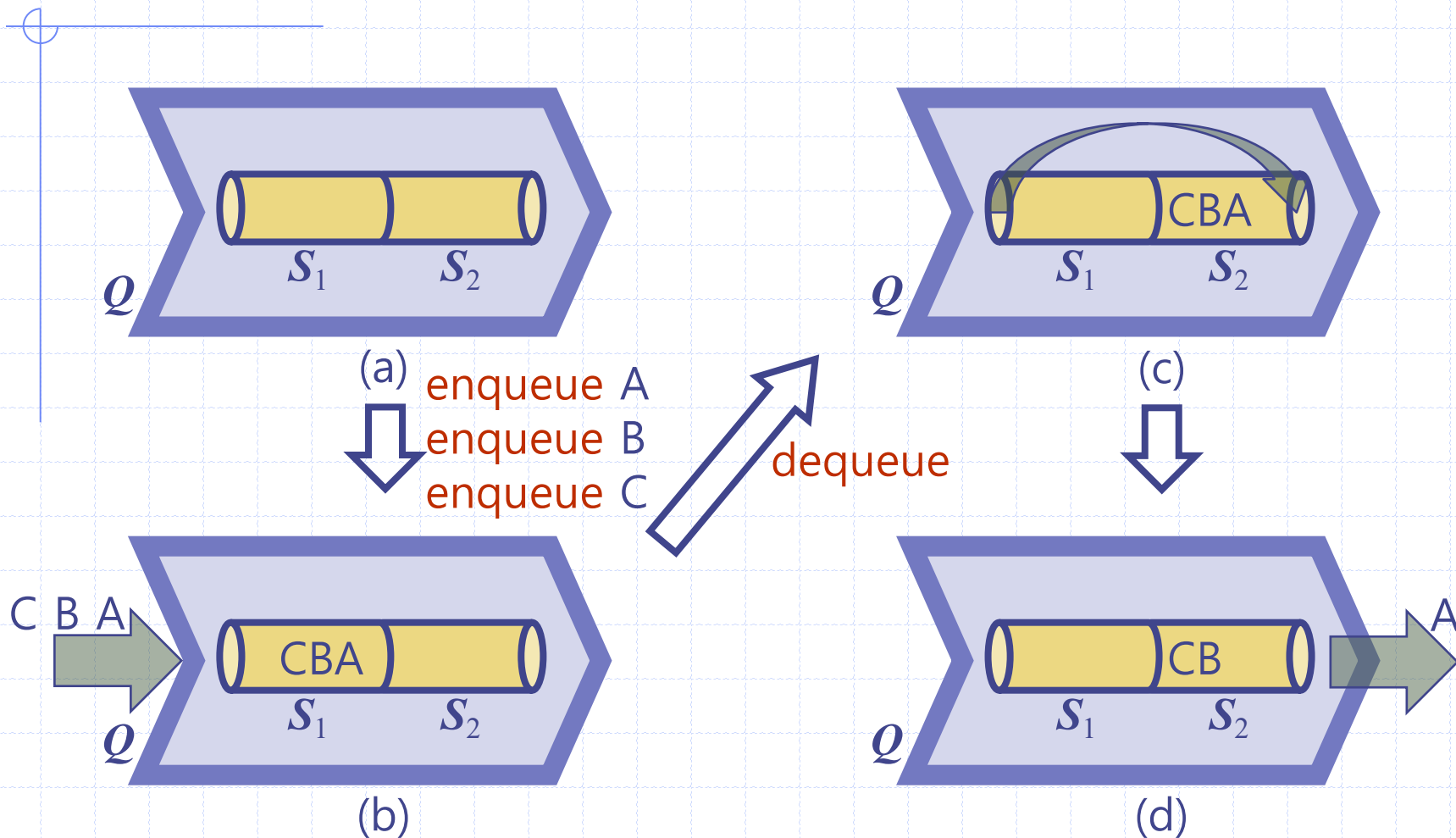


해결: 합동스택

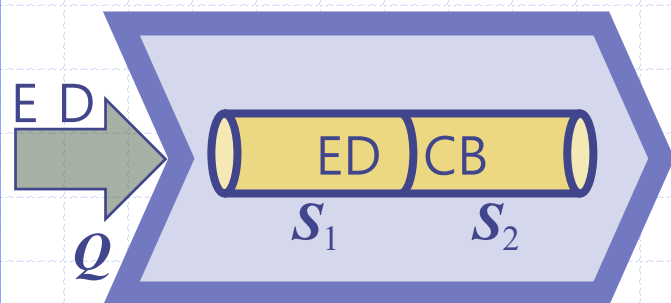
- ◆ 각각의 스택을 s_1 및 s_2 라고 하면,
 - s_1 을 enqueue 작업에,
 - s_2 를 dequeue 작업에 사용
- ◆ 실행시간
 - enqueue: $O(1)$ 시간
 - dequeue: $O(1)$ 시간: 상각실행시간(amortized running time)



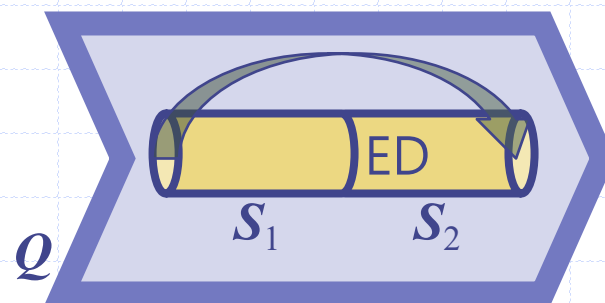
해결: 합동스택 수행 예



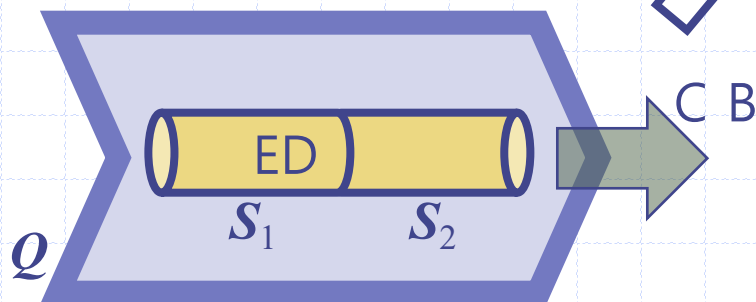
해결: 합동스택 수행 예 (conti.)



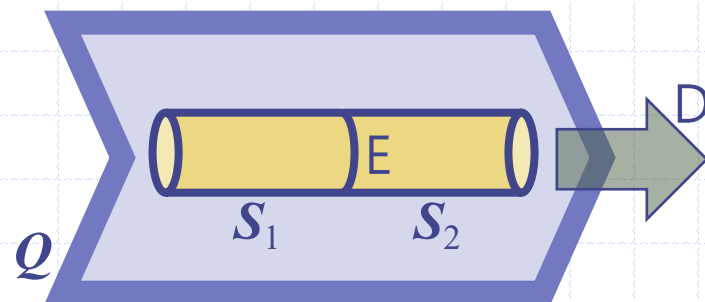
(e) dequeue
dequeue



dequeue



(f)



(h)

삭제에 소요되는 시간: 종합실행시간 분석



◆ 자판기에서 캔을 뽑는데 걸리는 시간

- 평상시: $O(1)$ 시간
- 최악: $O(n)$ 시간(리필 작업)

◆ 종합분석(aggregate analysis)

- n 회의 작업에 소요되는 총 시간을 구하고,
- 이를 작업 수 n 으로 나누어 1회 작업에 소요되는 시간을 구함: **상각실행시간**(amortized running time)

◆ 자판기 경우

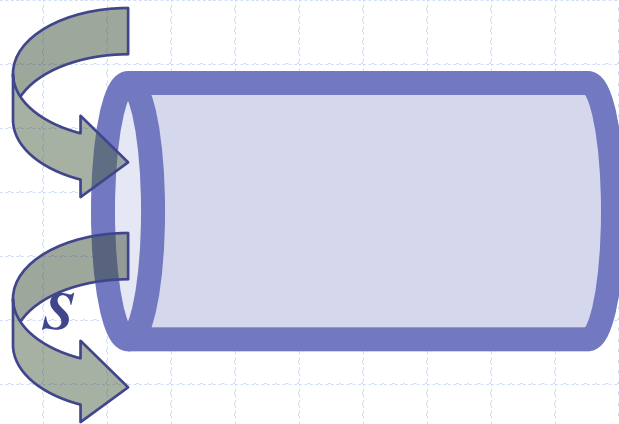
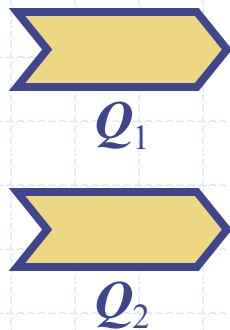
- $n + 1$ 회의 작업에 소요되는 총 시간
 $= (n + 1) \cdot O(1) + O(n) = O(n)$
- $O(n)$ 을 $n + 1$ 로 나누면 $O(1)$ 상각실행시간

◆ 합동스택으로 구현된 큐 ADT 경우

- n 회의 삭제 작업에 소요되는 총 시간 $= O(n) + O(n) = O(n)$
- $O(n)$ 을 n 으로 나누면 $O(1)$ 상각실행시간

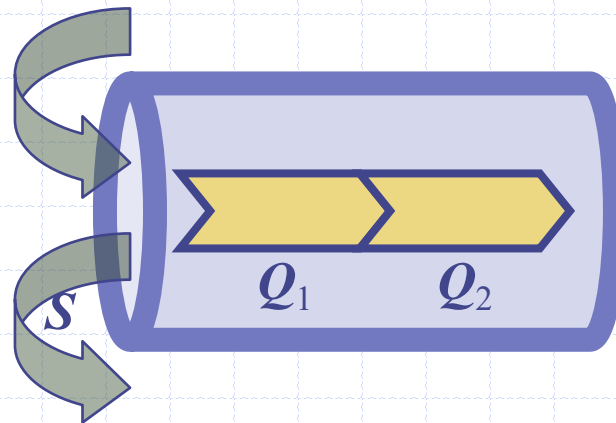
응용문제: 두 개의 큐로 스택 만들기

- ◆ 두 개의 일반 큐를 이용하여 어떻게 스택 ADT를 구현할지 설명하라
- ◆ 전제: 주어진 큐들은 isEmpty, front, enqueue, dequeue 등의 기본 메소드들을 상수시간에 수행 (size 메소드는 지원하지 않음)
- ◆ 주의: 스택 ADT에는 중복 원소들의 저장이 가능

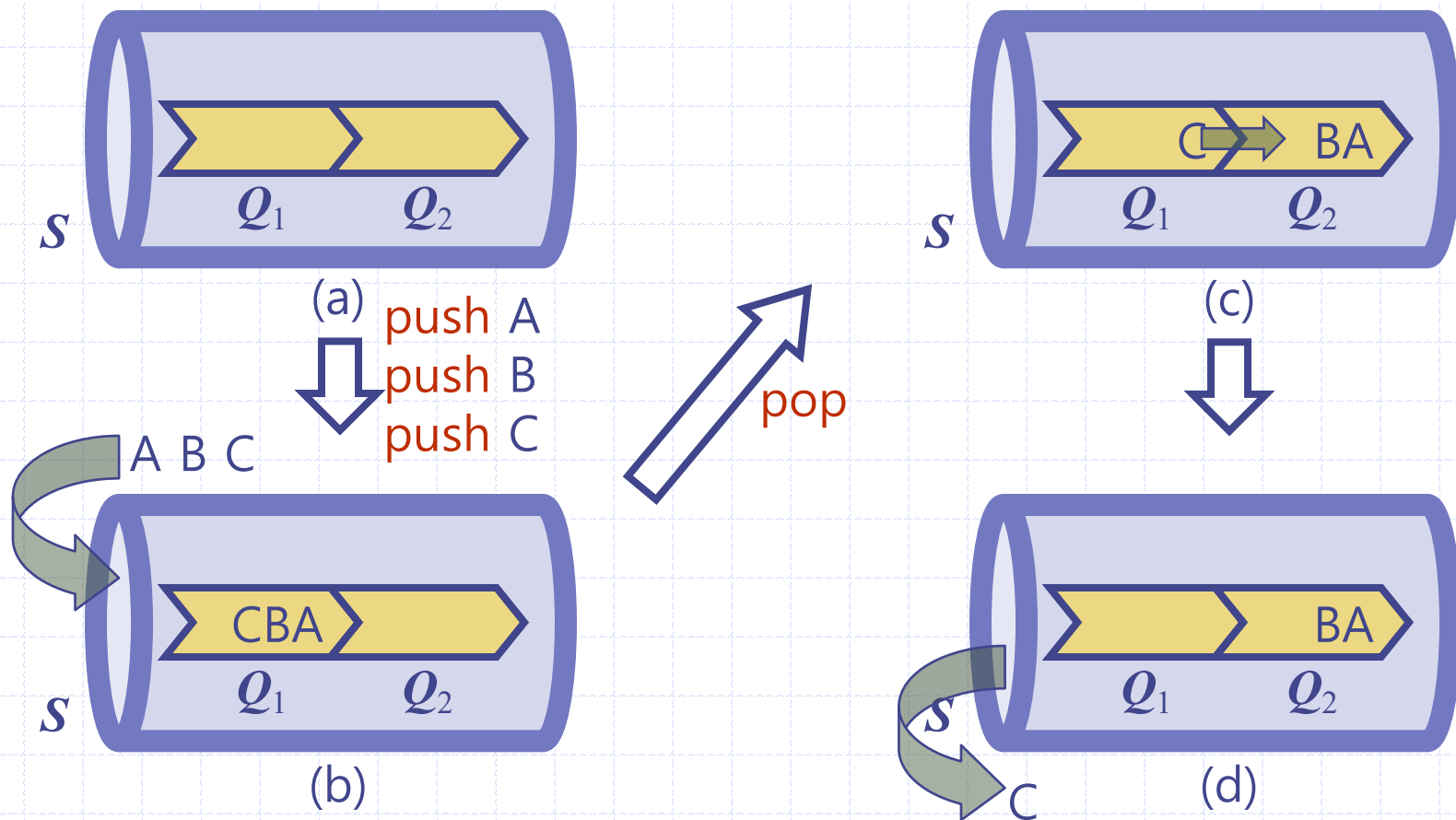


해결: 합동큐

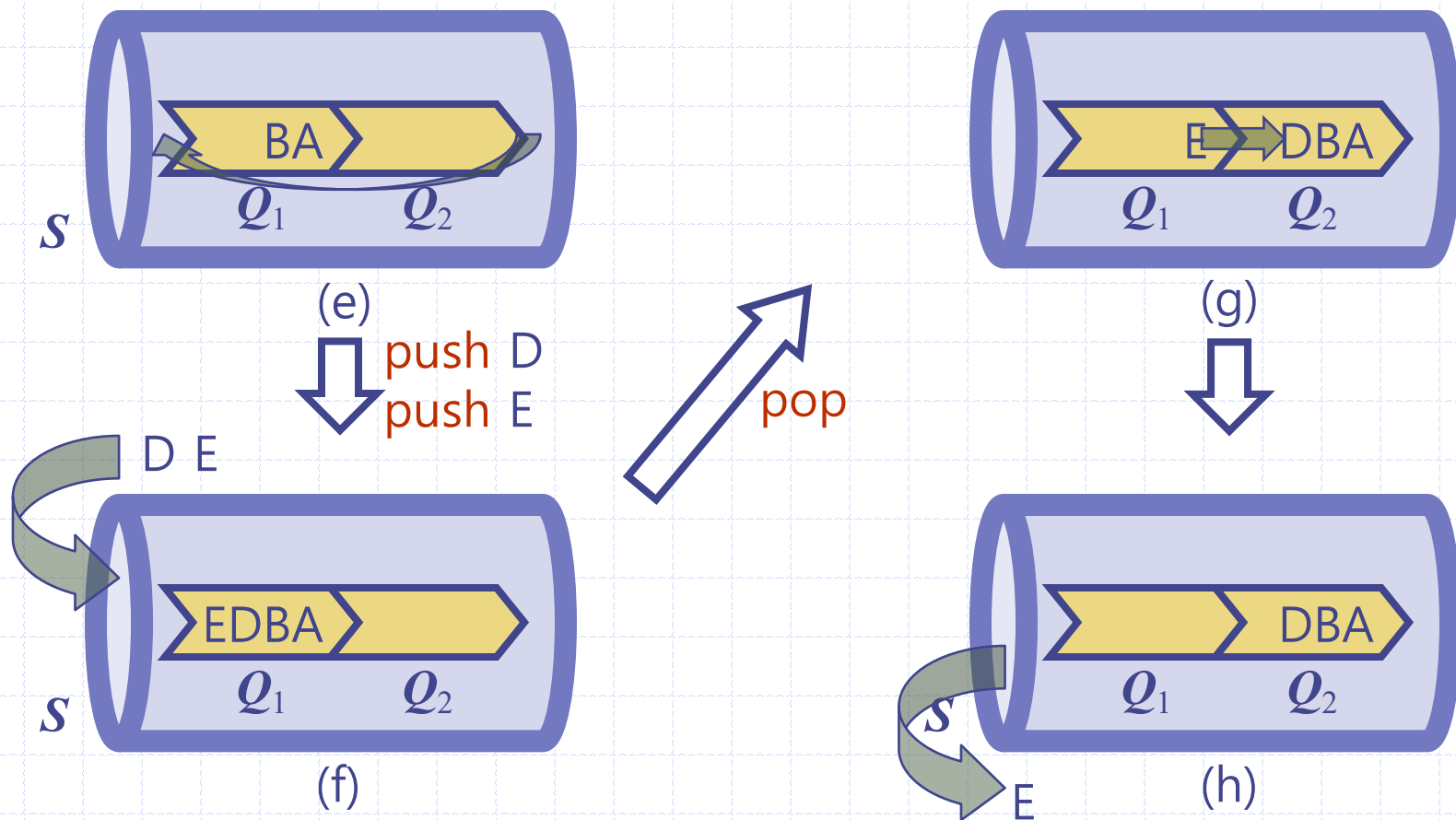
- ◆ 각각의 큐를 Q_1 및 Q_2 라고 하면,
 - Q_1 을 **push** 작업에,
 - Q_2 를 **pop** 작업에 사용
- ◆ 실행시간
 - **push**: $O(1)$ 시간
 - **pop**: $O(n)$ 시간



해결: 합동큐 수행 예



해결: 합동큐 수행 예



해결: 합동큐 수행 예 (conti.)

