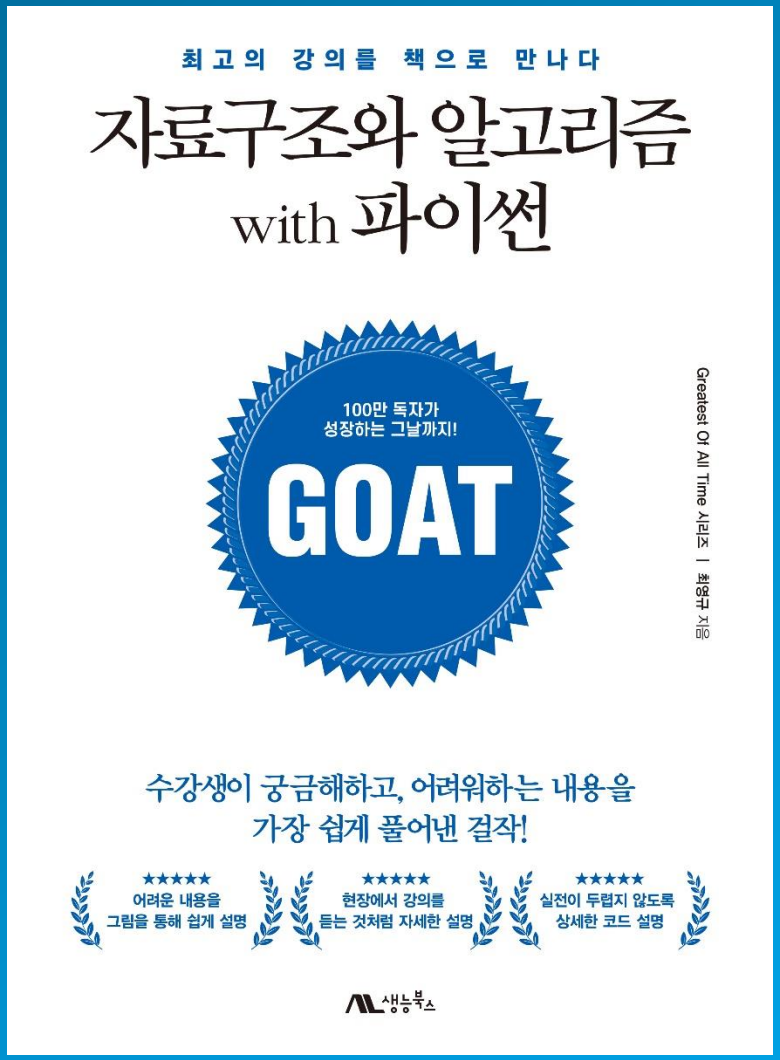
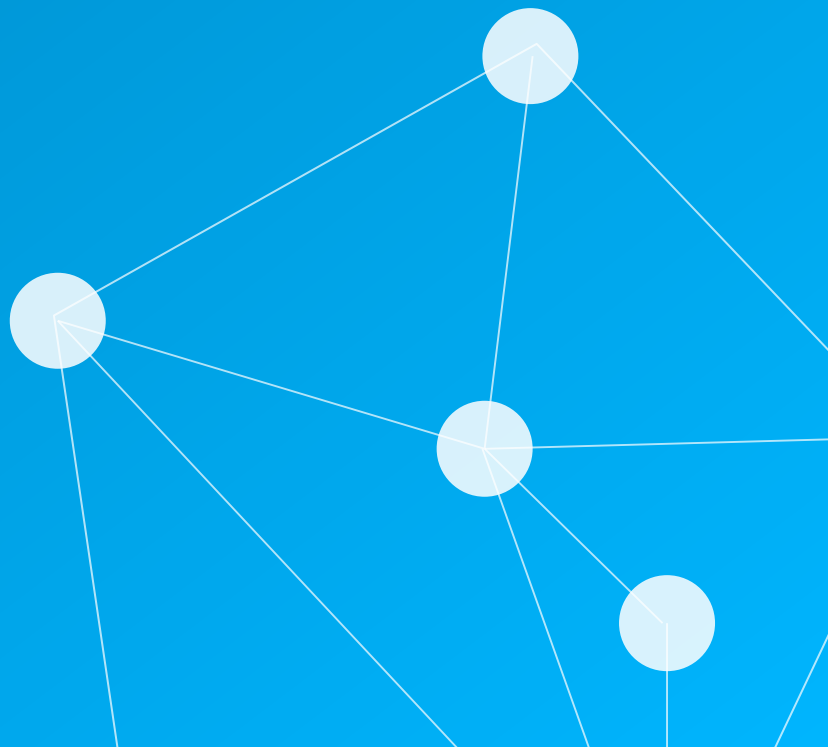
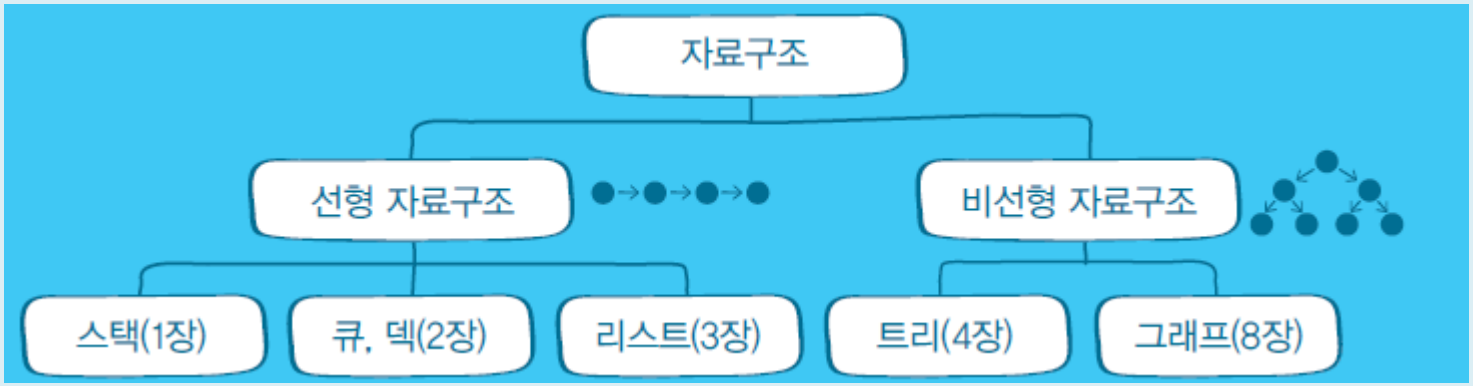

최고의 강의를 책으로 만나다
자료구조와 알고리즘
with 파이썬
100만 독자가
성장하는 그날까지!
GOAT
Greatest Of All Time 시리즈 | 최영규 지음
수강생이 궁금해하고, 어려워하는 내용을
가장 쉽게 풀어낸 걸작!
★★★★★
어려운 내용을
그림을 통해 쉽게 설명
★★★★★
현장에서 강의를
듣는 것처럼 자세한 설명
★★★★★
실전이 두렵지 않도록
상세한 코드 설명
생능북스

SW알고리즘개발

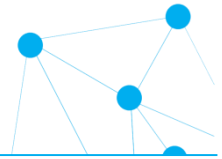


Part1. 자료구조



1장 스택 | 2장 큐 | 3장 리스트 | 4장 트리

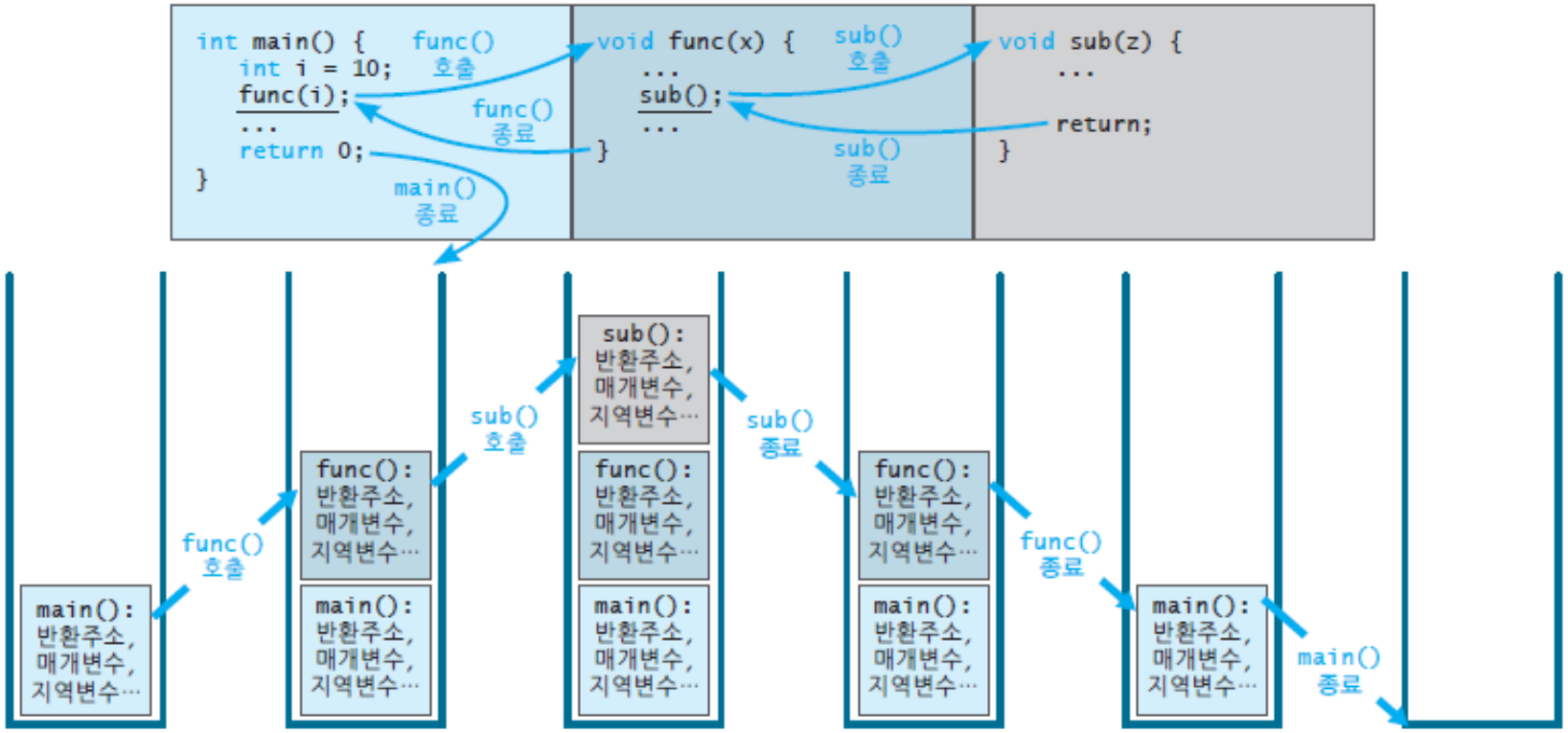
1.5 시스템 스택



- 운영체제에서 프로그램 실행 중에 함수 호출과 관련된 데이터를 임시로 저장하고 관리하는 구조.
- LIFO(Last In, First Out) 방식으로 동작,
- 함수가 호출되면 **함수의 반환 주소**가 스택에 저장. 함수 실행이 끝나면 이 주소를 참조하여 원래 위치로 복귀.
- 함수 내에서 선언된 **지역 변수**들은 스택에 저장되며, 함수 실행이 끝나면 스택에서 제거
- 함수 호출 시, **인자**들이 스택에 저장되고 함수가 이를 참조하여 처리
- **스택 오버플로우**(Stack Overflow): **시스템 스택은 크기가 제한**되어 있어 **너무 많은 함수 호출이 중첩되거나** 지나치게 많은 데이터를 스택에 쌓을 경우, "스택 오버플로우"가 발생. 이는 프로그램의 **비정상 종료로 이어짐**.

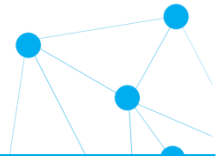
시스템 스택과 순환 호출

- **함수의 호출과 반환**을 위해 시스템 스택 예



- **시스템 스택을 적극적으로 사용하는 프로그래밍 기법**
  - 같은 일을 되풀이하는 방법: (1) 반복, (2) 순환(또는 재귀)

순환(Recursion) 기법



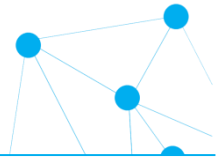
- 일반적인 순환:
  - 자연이나 사회 현상에서 어떤 과정이 주기적으로 반복되는 것
  - 예: 계절의 변화(봄, 여름, 가을, 겨울), 물의 순환(증발, 응축, 강수)
- 프로그래밍에서의 순환(반복)
  - **반복문(Loop)**을 통해 프로그램의 특정 부분이 반복해서 실행되는 것을 의미

- 재귀 함수는 **자기 자신을 호출하는 함수**
  - 문제를 더 작은 부분으로 나누어 반복적인 작업을 수행할 때 사용
  - 재귀 함수는 **기본 조건(base case)**을 만나면 자기 자신을 호출하지 않고 결과를 반환하여 재귀 호출을 종료

- 어떤 **함수가 자기 자신을 다시 호출하여 문제를 해결**하는 프로그래밍 기법
  - 문제 해결을 위한 독특한 구조를 제공
  - 많은 효율적인 알고리즘들에서 사용됨

- **문제 자체가 순환적이거나 순환적으로 정의되는 자료구조를 다루는데 적합**
  - 문제 자체가 순환적 : 팩토리얼 계산, 하노이 탑 등
  - 순환적으로 정의되는 자료구조 : 이진 트리

재귀 구조 vs. 순환(반복) 구조



- 모두 어떤 작업을 **반복적으로 수행하는데** 사용
- 순환 구조는 **'for', 'while' 과 같은 반복문**을 통해 구현
  - **반복구조**: 메모리 효율이 높고 큰 입력에 대해 성능이 좋다.
- 재귀 구조는 **재귀함수와 시스템 스택**을 이용하여 구현
  - **순환구조**: 코드가 더 간결하고 이해하기 쉬울 수 있지만, 재귀 깊이가 커지면 성능 저하가 발생
- **두 방식 모두 같은 결과를 반환하지만, 상황에 따라 적합한 방법을 선택하는 것이 중요**
- 재귀로 작성된 알고리즘은 반복문으로 변환할 수 있고, 그 반대도 가능
- 예시: Factorial 계산, Fibonacci 수열

예) n! 구하기

반복

$$n! = 1 \times 2 \times 3 \times \cdots \times (n-1) \times n$$

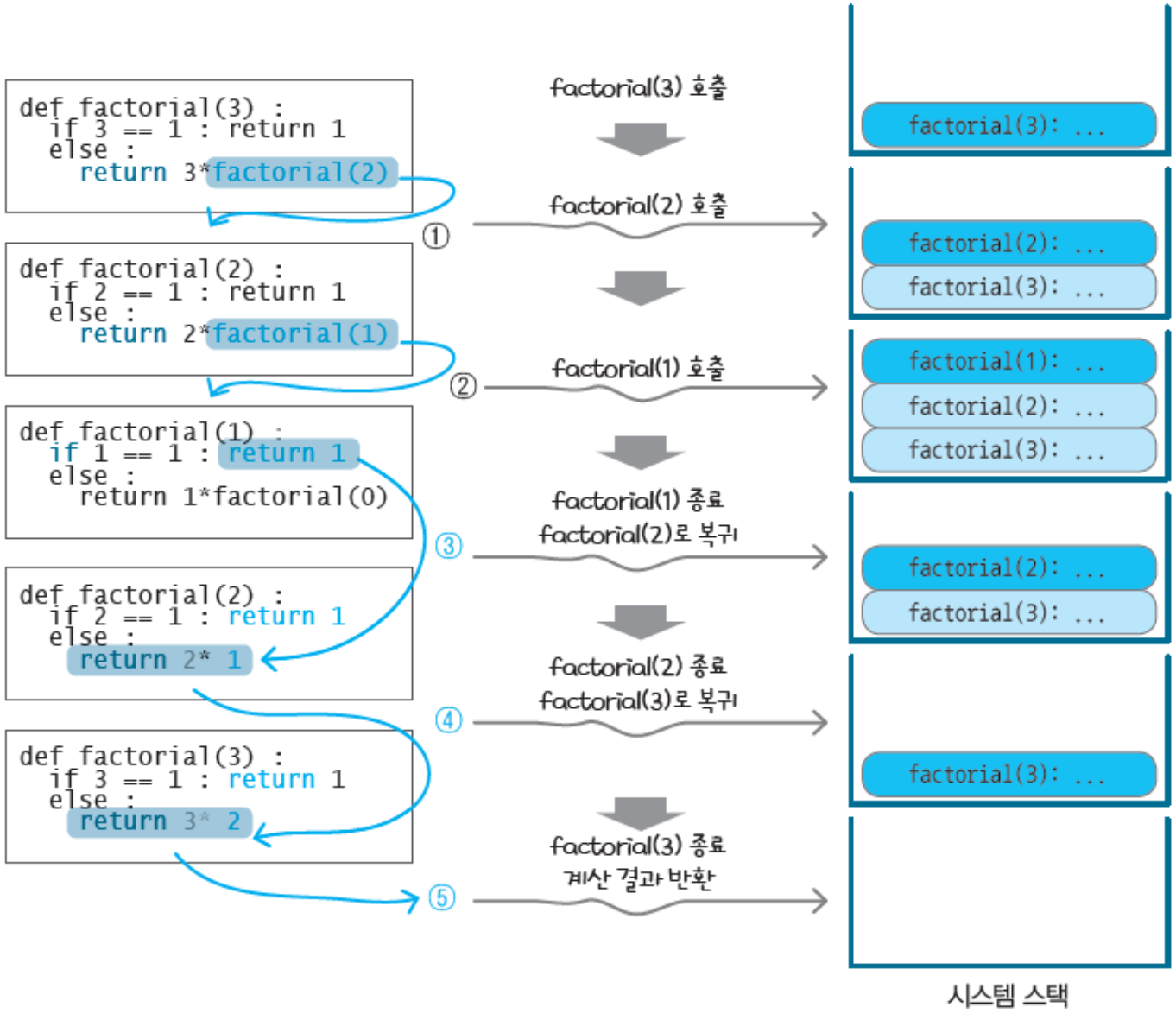
```
def factorial_iter(n) :
    result = 1
    for k in range(2, n+1) :
        result = result * k
    return result
```

순환

$$n! = \begin{cases} 1 & n=1 \\ n \times (n-1)! & n>1 \end{cases}$$

```
def factorial(n) :
    if n == 1 :
        return 1
    else :
        return n * factorial(n-1)
```

순환적인 팩토리얼 함수 동작의 이해



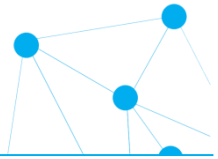
예: 피보나치 수열(Fibonacci Sequence)

- 정의: 수학에서 매우 유명한 수열 중 하나
  - 첫 번째 항과 두 번째 항은 각각 1
  - 세 번째 항부터는 바로 앞의 두 항의 합
- 1, 1, 2, 3, 5, 8, 13, 21, 34, 55, 89, …, F(n),

이를 수식으로 표현하면, 피보나치 수열의 n번째 항 $F(n)$

- $F(1) = 1$
- $F(2) = 1$
- $F(n) = F(n-1) + F(n-2)$ (n ≥ 3)

반복과 순환 함수 구현



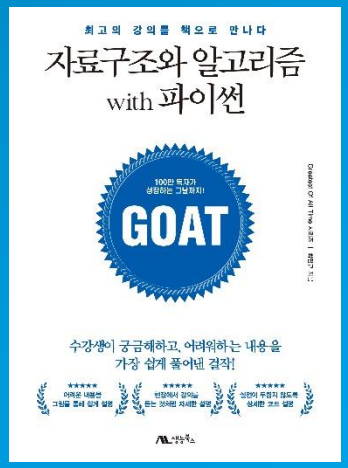
```python
def fibonacci_iterative(n):
    if n <= 0:
        return 0
    elif n == 1:
        return 1

    prev, current = 0, 1
    for _ in range(2, n + 1):
        prev, current = current, prev + current
    return current
```

반복

```python
def fibonacci_recursive(n):
    if n <= 0:
        return 0
    elif n == 1:
        return 1
    else:
        return fibonacci_recursive(n - 1) + fibonacci_recursive(n - 2)
```

순환


최고의 강의를 책으로 만나다
자료구조와 알고리즘
with 파이썬
100만 독자가
생활하는 그날까지!
GOAT
수강생이 궁금해하고, 어려워하는 내용을
가장 쉽게 풀어낸 걸작!

02 CHAPTER

큐

2장. 큐

02-1 큐란?

02-2 배열로 구현하는 큐

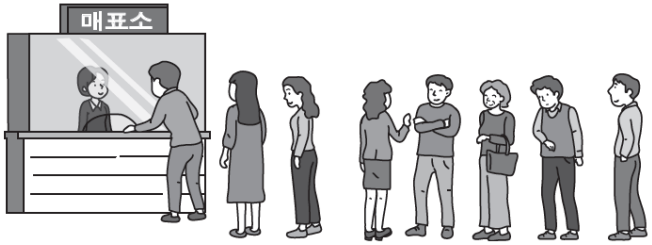
~~02-3 덱이란?~~

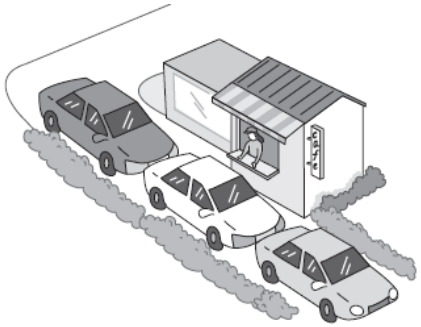
~~02-4 상속을 이용한 덱의 구현~~

02-5 파이썬에서 큐와 덱 사용하기

2.1 큐

- 큐(Queue)
  - 선입선출(FIFO: First-In First-Out)의 자료구조
  - 가장 먼저 들어온 데이터가 가장 먼저 나감

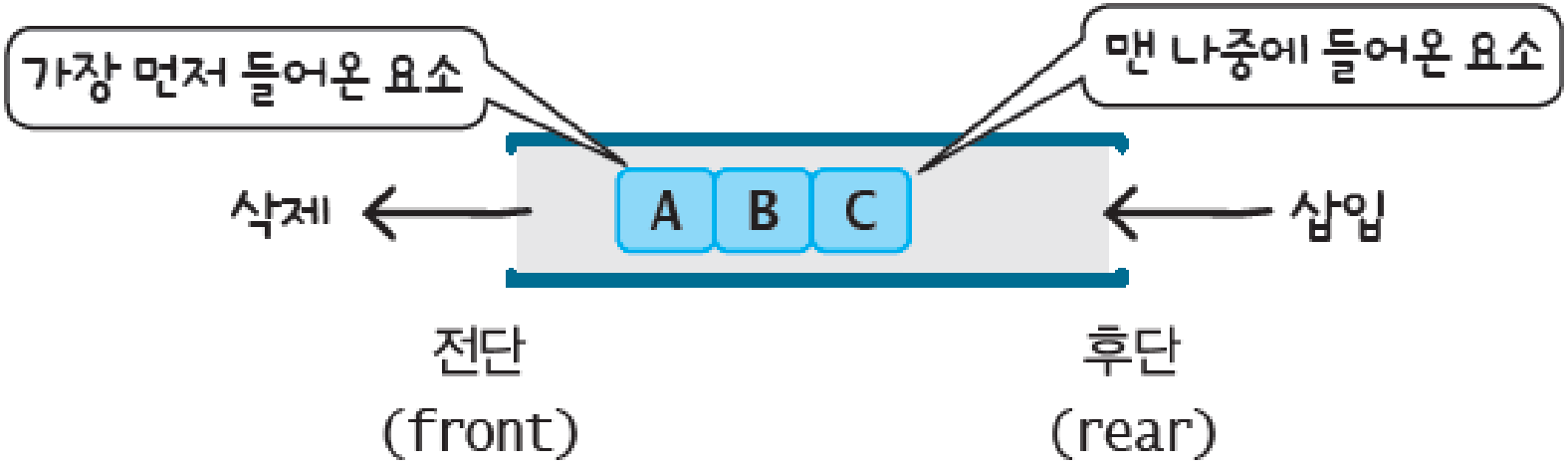




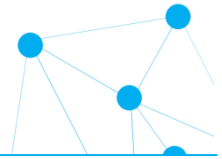
- 큐의 응용
  - 컴퓨터에서도 큐가 필요한 곳은 매우 광범위함
  - 버퍼(buffer)로 사용
    : 시간이나 속도 차이를 극복하기 위한 임시 기억 장치
    (예) CPU와 주변장치(프린트) 사이

큐의 구조

- 큐(Queue)
  - 선입선출(FIFO: First-In First-Out)의 순차 구조
  - 가장 먼저 들어온 데이터가 가장 먼저 나감

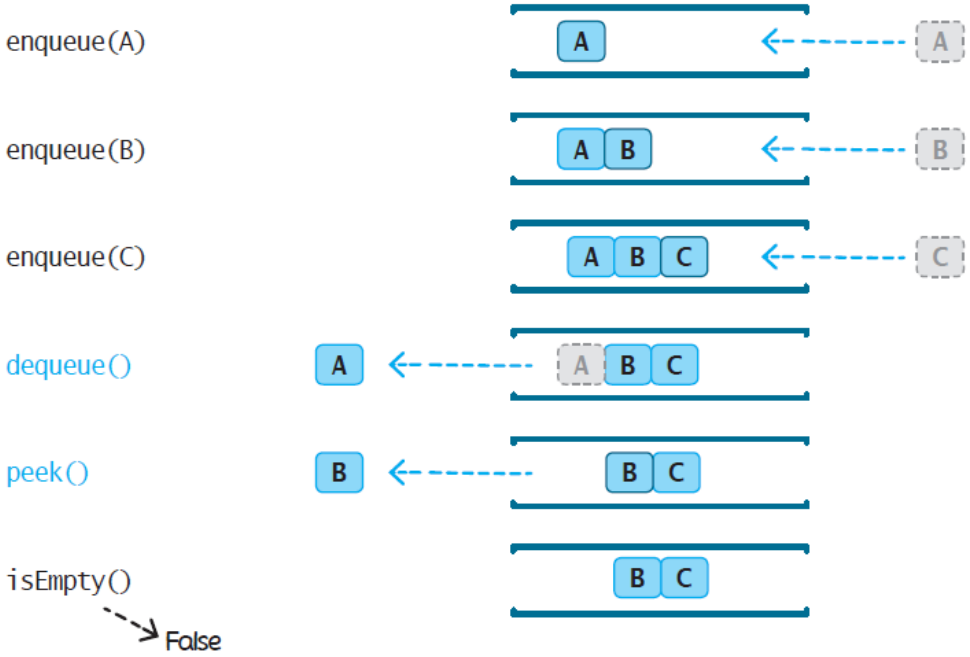


큐의 연산들

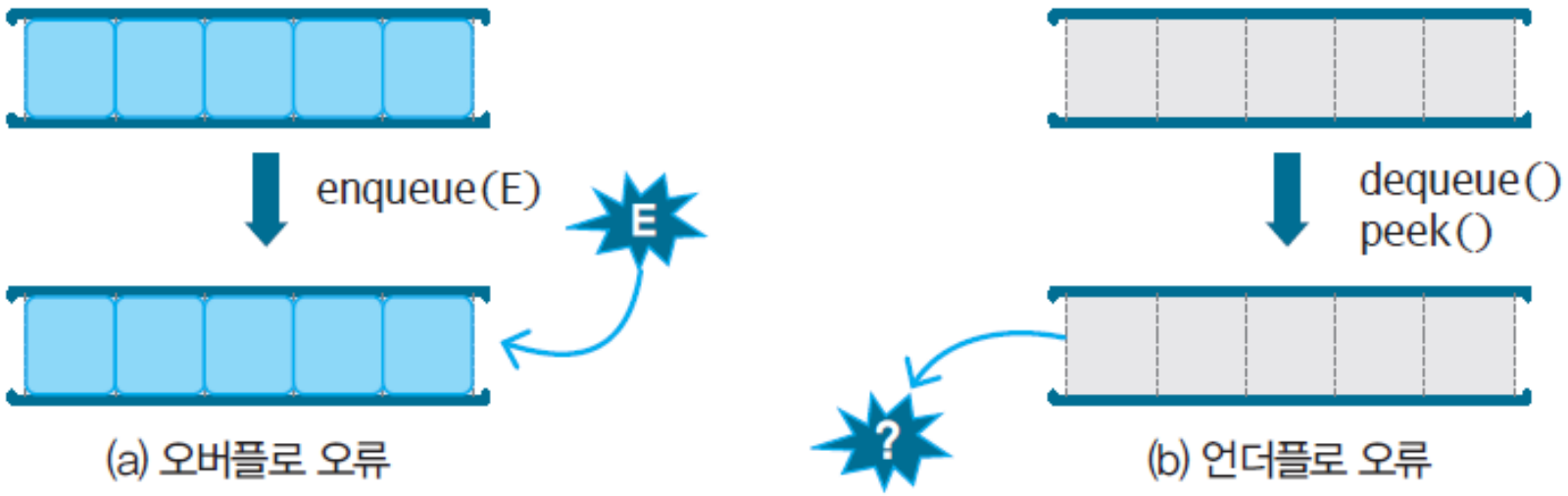


큐의 연산

- enqueue(e) : 새로운 요소 e를 큐의 맨 뒤에 추가
- dequeue() : 큐의 맨 앞에 있는 요소를 꺼내서 반환
- isEmpty() : 큐가 비어 있으면 True를 아니면 False를 반환
- isFull() : 큐가 가득 차 있으면 True를 아니면 False를 반환
- peek() : 큐의 맨 앞에 있는 요소를 삭제하지 않고 반환
- size() : 큐에 들어있는 전체 요소의 수를 반환

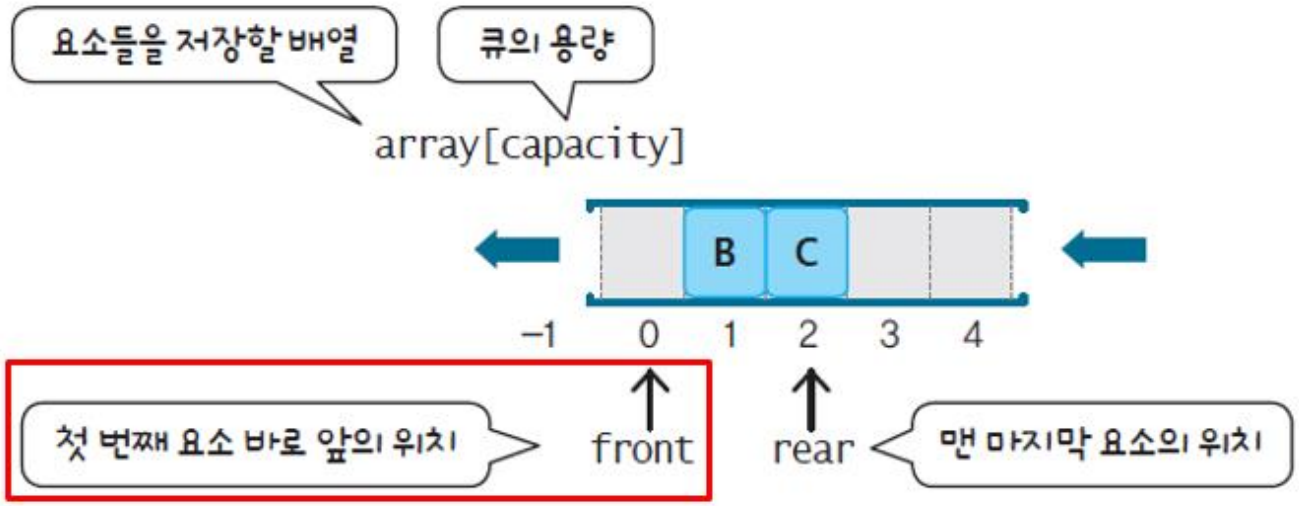


큐의 오류 상황



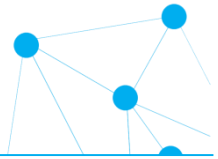
2.2 배열로 구현하는 큐

- 배열 구조의 큐를 위한 데이터



- array[] : 큐 요소들을 저장할 배열
- capacity : 큐에 저장할 수 있는 요소의 최대 개수
- rear : 맨 마지막(후단) 요소의 위치(인덱스)
- front : 첫 번째(전단) 요소 바로 이전의 위치(인덱스)

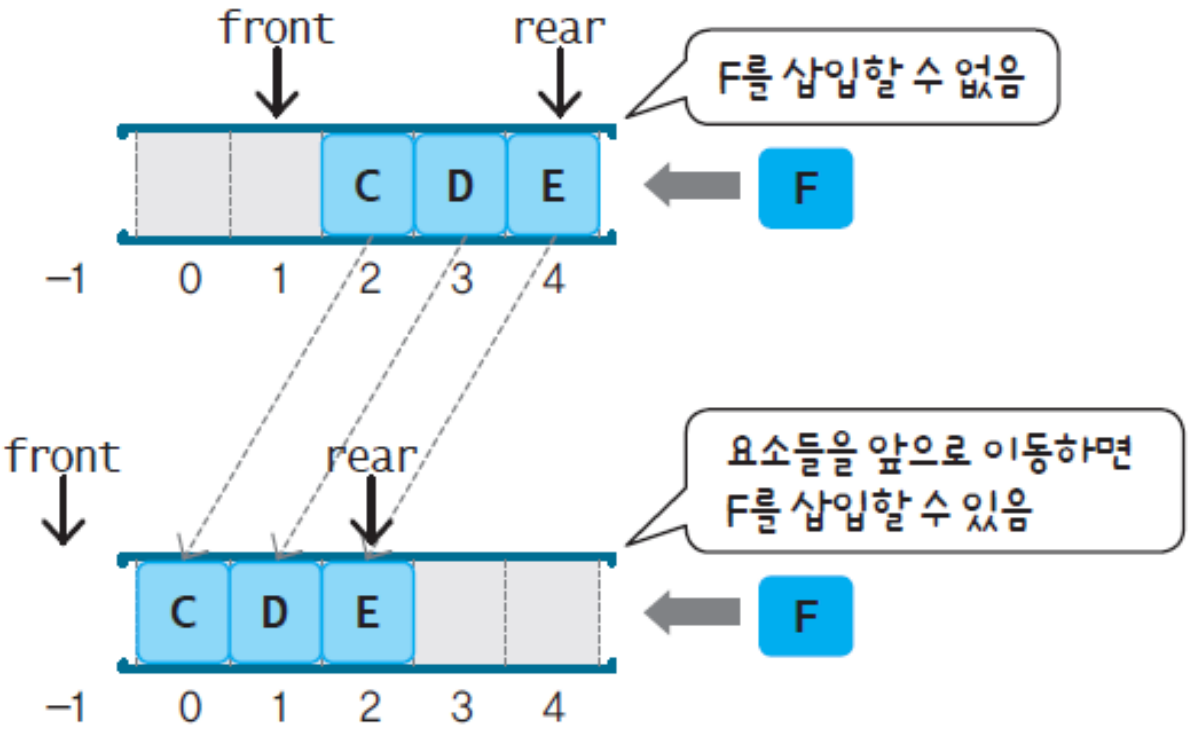
선형 큐의 문제점



- 요소들의 많은 이동이 필요함

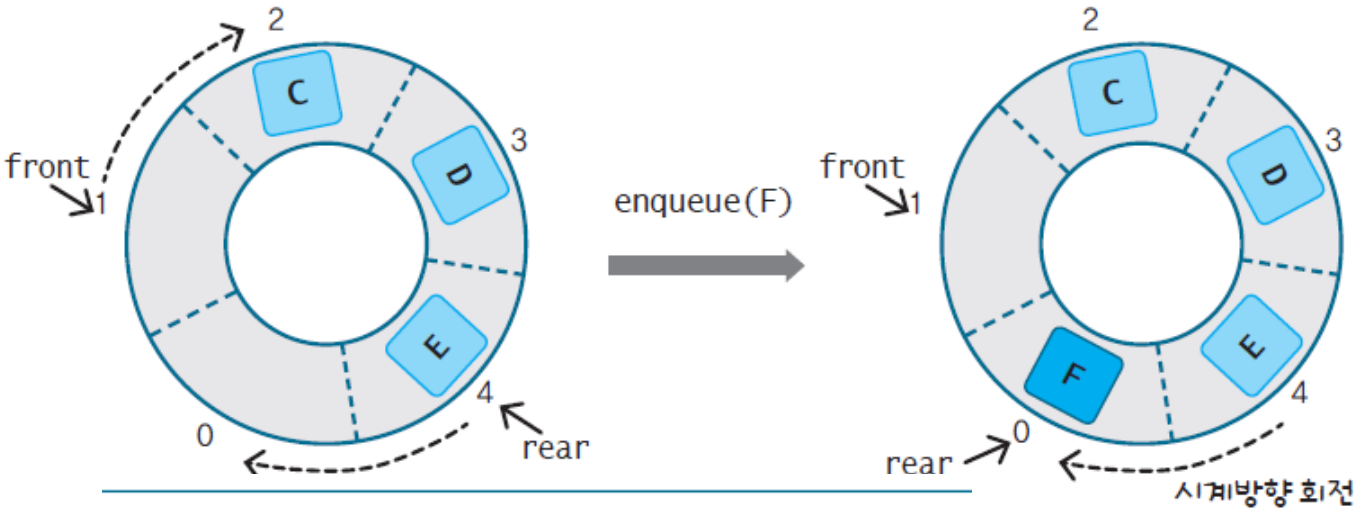
enqueue(A)
enqueue(B)
enqueue(C)
enqueue(D)
enqueue(E)
dequeue()
dequeue()

enqueue(F) ?



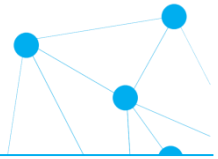
원형 큐의 원리

- **배열을** 선형이 아니라 **원형으로 생각하는 것**
  - 인덱스 front와 rear를 원형으로 회전시키는 개념
- **배열의 끝에 도달하면 다시 처음으로 돌아가는 "순환"**
  - 인덱스가 배열의 마지막 인덱스를 넘어가면 처음 인덱스으로 다시 돌아가는 구조



- 전단 회전 : front ← (front+1) % capacity
- 후단 회전 : rear ← (rear+1) % capacity

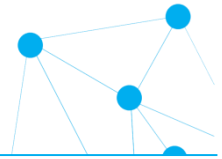
2.3 배열 기반의 원형 큐의 구조(1)



- 원형 큐: 고정된 크기를 가진 배열을 효율적으로 사용
  - 'rear'와 'front'가 배열의 끝에 도달할 때 **다시 처음으로 돌아가는 원형 구조**
  - 배열의 크기가 제한되어 있어도 메모리를 효율적으로 사용
  - 삽입 및 삭제 연산에서 배열의 크기를 넘는 인덱스 처리를 용이

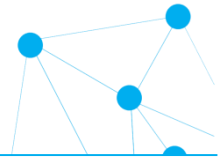
- **원형 큐의 rear (리어 포인터)**
  - 큐에서 **데이터가 삽입할 위치**를 가리킴
  - 데이터를 삽입할 때마다 rear가 가리키는 위치에 데이터를 넣고, rear는 한 칸씩 증가함
  - **'%capacity' 모듈 연산**을 사용하여 배열의 끝에 도달하면 다시 처음으로 돌아가도록 구현
  - 예: 큐의 크기가 5, rear=4인 경우, 삽입이 일어나면, (rear + 1) % 5 = 0 이 되어 다시 처음 위치로 이동

2.3 배열 기반의 원형 큐의 구조(2)



- **원형 큐의 front (프론트 포인터)**
  - 큐에서 **데이터가 삭제할 위치를** 가리킴
  - 데이터를 삭제할 때마다 front가 가리키는 위치의 데이터를 꺼내고, front는 한 칸씩 증가함
  - **'% capacity' 모듈 연산**을 사용하여 배열의 끝에 도달하면 다시 처음으로 돌아가도록 구현
  - 예: 큐의 크기가 5일 때, front가 배열의 마지막 위치(인덱스 4)에서 삭제 연산이 일어나면 (front + 1) % 5 = 0 이 되어 다시 배열의 첫 번째 위치로 이동

원형 큐의 클래스 구현



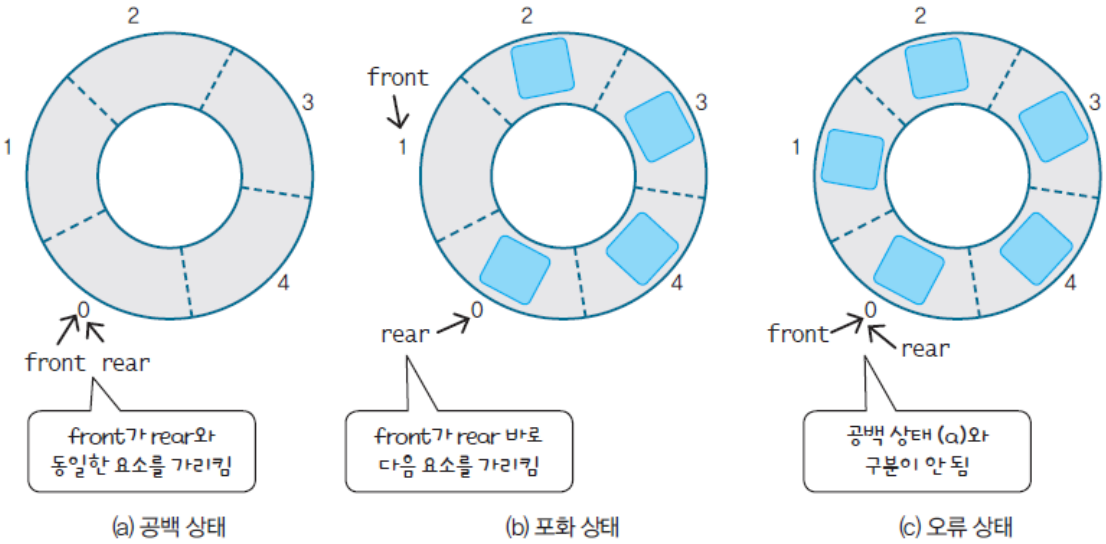
- 용량이 고정된 원형 큐 클래스 ArrayQueue

원형큐의 생성자

```
class ArrayQueue :
    def __init__( self, capacity = 10 ) :    # 생성자 정의
        self.capacity = capacity             # 용량(고정)
        self.array = [None] * capacity       # 요소들을 저장할 배열
        self.front = 0                       # 전단 인덱스
        self.rear = 0                        # 후단 인덱스
```

공백 상태와 포화 상태 검사

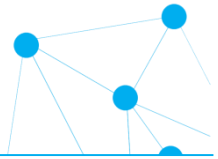
- 큐의 포화상태: 큐가 가득 찼는지 확인하는 방법은 front와 rear의 위치를 비교
  - Front == '(rear + 1) % capacity'
  - rear가 front 바로 앞에 도달하면 큐는 더 이상 데이터를 삽입할 수 없음
  - 이를 해결하기 위해서 **포화 상태일 때 큐의 한 칸을 비워두는 방법**을 사용. 즉, 큐가 실제로 가득 차더라도, 하나의 빈 칸을 남겨두고 포화 상태로 간주
- 큐의 공백 상태: 큐가 비어 있는지 확인하는 방법은 front와 rear의 위치를 비교
  - front == rear
  - 큐에는 데이터가 없는 상태이며, 데이터를 삽입하면 rear가 움직이기 시작함




```
def isEmpty( self ) :
    return self.front == self.rear

def isFull( self ) :
    return self.front == (self.rear+1)%self.capacity
```

원형 큐의 삽입/삭제/참조



- 삽입 연산: enqueue(e)

```
def enqueue( self, item ):                  # 삽입 연산
    if not self.isFull():                   # 포화 상태가 아닌 경우
        self.rear = (self.rear + 1) % self.capacity
        self.array[self.rear] = item
    else : pass                             # 오버플로 오류: 처리 안 함
```

- 삭제 연산: dequeue( )

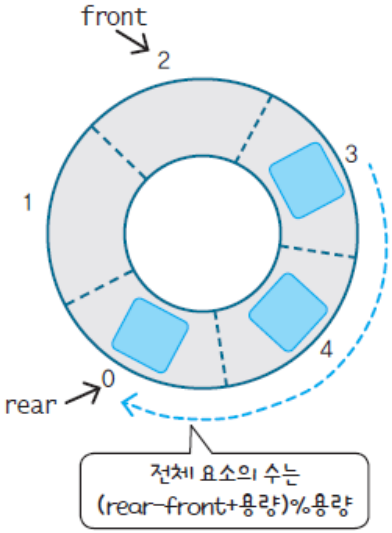
```
def dequeue( self ):
    if not self.isEmpty():
        self.front = (self.front + 1) % self.capacity
        return self.array[self.front]
    else : pass                             # 언더플로 오류: 처리 안 함
```

- 참조 연산: peek( )

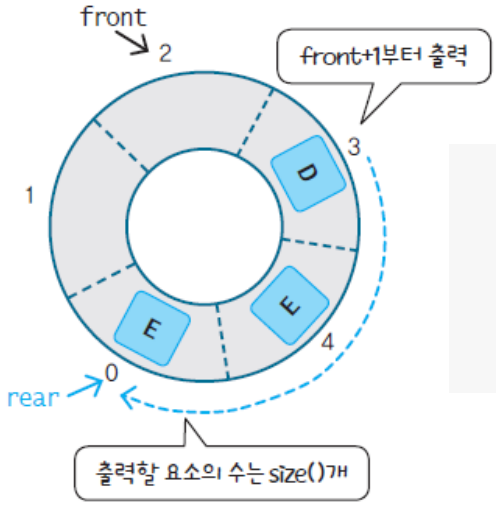
```
def peek( self ):
    if not self.isEmpty():
        return self.array[(self.front + 1) % self.capacity]
    else : pass                             # 언더플로 오류: 처리 안 함
```

기타 연산들

- 원형 큐 전체 요소의 수

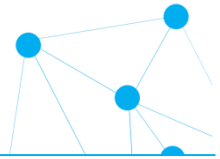


- 큐 내용을 출력 연산



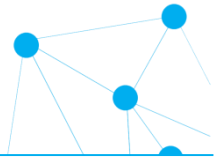

```
i = front
while i != rear:
    print(queue[i], end=' ')
    i = (i + 1) % capacity  # 인덱스를 순환하도록 처리
```

원형큐의 활용(1)



- 원형 큐는 효율적으로 메모리를 사용하고, 'FIFO'특성을 갖고 있기 때문에 고정된 크기의 데이터를 순차적으로 처리해야 하는 상황에서 유용
- **CPU 스케쥴링**: 운영체제에서 프로세스 관리 및 스케줄링에서 원형 큐가 사용. 라운드로빈 방식의 스케줄링에서는 각 프로세스가 고정된 시간 동안 CPU를 사용하며, 시간이 끝나면 다음 프로세스가 순차적으로 CPU를 사용
- **키보드 입력 버퍼** : 컴퓨터나 임베디드 시스템에서 키보드 입력을 관리할 때 원형 큐가 사용. 각 키보드 입력은 큐에 저장되며, 입력이 처리됨에 따라 큐에서 데이터를 순차적으로 삭제하고, 새로운 입력을 저장
- **프린터 작업 대기열** : 여러 개의 문서가 동시에 프린터로 보내질 때, 프린터가 문서들을 순서대로 처리하기 위해 원형 큐가 사용. 각 인쇄 작업은 큐에 저장되고, 프린터가 하나씩 문서를 처리하면서 큐에서 삭제

문제: 원형 큐의 활용



- 무작위로 발생한 정수(0~99)를 큐가 꽉 찰 때까지 삽입한 후, 다시 모든 숫자를 꺼내 출력하는 프로그램

```
import random                          # 난수 발생을 위해 random 모듈 포함
q = ArrayQueue(8)                      # 큐 객체를 생성(capacity=8)

q.display("초기 상태")
while not q.isFull() :
    q.enqueue(random.randint(0,100))
q.display("포화 상태")

print("삭제 순서: ", end='')
while not q.isEmpty() :
    print(q.dequeue(), end=' ')
print()
```

큐가 포화 상태가 될 때까지 0에서 99 사이의 정수를 무작위로 발생하여 큐에 삽입. 용량이 8이므로 7개까지 삽입됨.

큐가 공백 상태가 될 때까지 요소를 꺼내서 화면에 출력

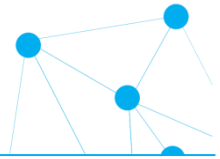
실행 결과

```
초기 상태= []
포화 상태= [31 8 5 64 9 17 26 ]
삭제 순서: 31 8 5 64 9 17 26
```

capacity가 8인 원형 큐에는 7개의 요소가 삽입되면 포화 상태가 됩니다.

큐에 입력된 순서대로 꺼내서 출력합니다.

원형큐의 활용(2)



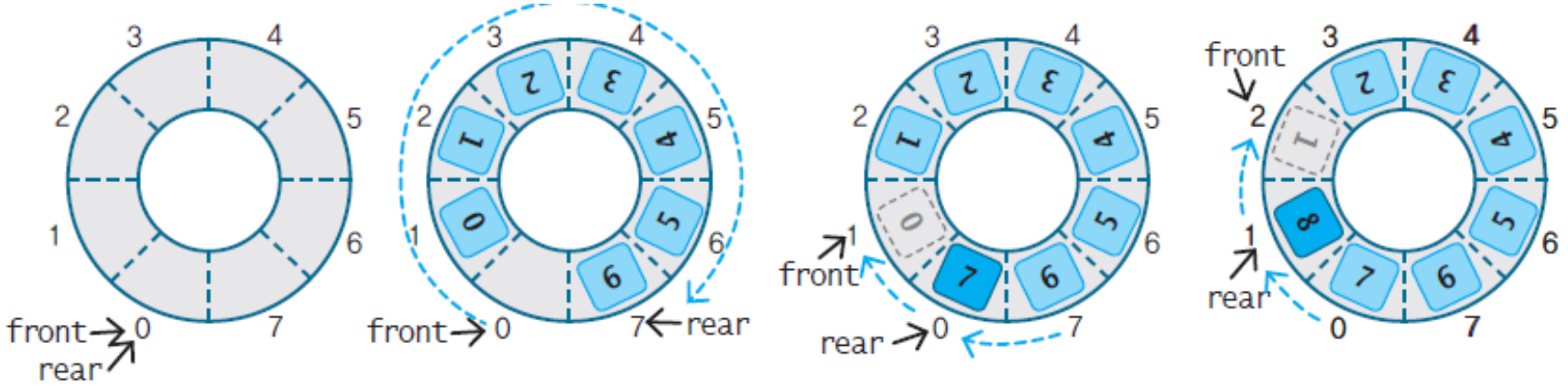
- **데이터 스트림 처리** : 스트리밍 데이터의 경우, 데이터가 지속적으로 들어오는 상황에서 버퍼를 관리할 때 원형 큐가 활용. 버퍼가 가득 차면, 데이터는 다시 처음부터 저장되며, **오래된 데이터를 덮어씌우거나 필요 시 삭제**하는 방식으로 처리.

- **네트워크 패킷 버퍼:** 네트워크 장비에서는 네트워크 패킷을 임시로 저장하고 처리해야 할 때 원형 큐를 사용. 네트워크를 통해 데이터를 보내거나 받을 때, 패킷은 순차적으로 큐에 저장되며, 라우터와 같은 네트워크 장치가 수많은 패킷을 관리할 때, 원형 큐를 이용해 패킷의 순서를 유지하고 효율적으로 관리

- **데이터 캐시(Circular Buffers for Caching)** : 데이터 캐시에서 고정된 크기의 메모리 공간을 활용하여 최근 데이터나 자주 사용하는 데이터를 저장하고 관리할 때 원형 큐가 사용. **캐시가 가득 차면 오래된 데이터를 제거하고 새로운 데이터를 추가**

2.4 응용: 원형 큐를 링 버퍼로 사용하기

- 링 버퍼(ring buffer) : 오래된 자료를 버리고 항상 최근의 자료를 유지하는 용도로 사용
- 링 버퍼는 **고정된 크기**를 가지는 원형 큐와 비슷한 자료구조.
  - 버퍼가 가득 차면 **오래된 데이터가 덮어씌워지는** 방식으로 동작
  - 항상 최신 데이터를 유지하는 데 유리
  - 링 버퍼는 일정한 공간을 반복적으로 활용하므로 **메모리 절약**
  - 특히 **스트리밍 데이터**나 **순환하는 데이터**를 처리하는 데 적합



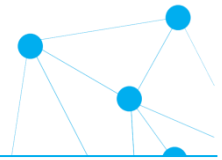
(a) 공백 상태 (capacity=8)

(b) 공백 상태에서 7개의 요소를 삽입하면 포화 상태가 됨

(c) 포화 상태에서 새로운 데이터 7을 삽입하면 가장 오래된 요소 0이 삭제됨

(d) 같은 방법으로 8을 삽입하면 가장 오래된 요소 1이 삭제됨

링 버퍼의 삽입 연산 : enqueue2()



- 삽입 연산 : enqueue함수
  - 링 버퍼에서 데이터를 삽입할 때 **덮어쓰기(overwriting)** 방식으로 동작
  - **원형 큐가 가득 찬 상태에서도 새 데이터를 삽입**할 수 있지만, 이 경우 **가장 오래된 데이터를 덮어씀**.
- 동작 원리
  1. 데이터 삽입 : 데이터를 **원형 큐에 삽입할** 때, rear가 가리키는 위치에 삽입

```
self.rear = (self.rear + 1) % self.capacity
```

     - rear가 큐의 끝에 도달하면 (rear+1) % capacity로 처음으로 돌아감
  2. 새로운 데이터 삽입 :

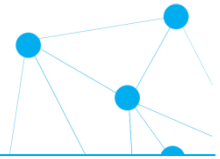
```
self.array[self.rear] = item
```

     - rear 가리키는 위치에 새로운 데이터을 삽입한다.
  3. 덮어 쓰기 발생 시 front 갱신 :

```
if self.is_full():
    self.front = (self.front + 1) % self.capacity
```

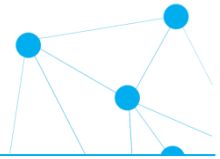
     1. **큐가 가득 찬** 경우
     2. **덮어 쓰기가** 발생하여, **가장 오래된 데이터가 삭제**되므로 **front 포인터도 한 칸 앞으로 이동**
        - front가 가리키는 위치는 큐에서 가장 오래된 데이터가 저장된 위치.
     - 큐가 가득 차고 새로운 데이터를 삽입할 때, 이 오래된 데이터를 덮어쓰게 되므로, front도 갱신되어야 함

덮어쓰기 방식의 장점



- **효율적인 메모리 사용** : 고정된 크기의 메모리 공간 내에서 데이터를 계속해서 순환시킬 수 있음
- **덮어쓰기 방식**: 큐가 가득 차더라도 데이터를 잃지 않고, 오래된 데이터를 덮어쓰는 방식으로 새로운 데이터를 처리. 특히, **실시간 시스템에서 데이터가 유실되지 않고 최신 상태 유지**

예제: 덮어쓰기 동작



가정:

- 큐의 용량(`capacity`) = 5
- `rear`와 `front`의 초기값 = 0
- 배열 상태: `[None, None, None, None, None]`

1. 처음 데이터 삽입 (`enqueue(10)`):

- `rear = (0 + 1) % 5 = 1` (rear가 1로 이동)
- `array = [None, 10, None, None, None]` (10이 후단에 삽입)
- `front`는 변하지 않음 (`front = 0`)

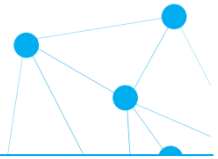
2. 두 번째 데이터 삽입 (`enqueue(20)`):

- `rear = (1 + 1) % 5 = 2` (rear가 2로 이동)
- `array = [None, 10, 20, None, None]` (20이 후단에 삽입)
- `front`는 변하지 않음 (`front = 0`)

4. [illegible] 번째 데이터 삽입 (`enqueue(50)`) -> 덮어쓰기 발생:

- `rear = (4 + 1) % 5 = 0` (rear가 배열 처음으로 돌아감)
- `array = [50, 10, 20, 30, 40]` (새로운 데이터 50이 배열의 처음에 삽입되고, 이전에 있었던 `None`이 덮어써짐)
- 큐가 가득 찼으므로, 가장 오래된 데이터인 `10`을 삭제하기 위해 `front`를 이동시킴:
  - `front = (0 + 1) % 5 = 1` (front도 한 칸 앞으로 이동하여, 이제 `front`는 배열의 두 번째 위치를 가리킴)

링 버퍼 테스트



- 삽입 연산 수정

```
def enqueue2( self, item ):                    # 링 버퍼 삽입 연산
    self.rear = (self.rear + 1) % self.capacity
    self.array[self.rear] = item                          ← 일단, 무조건 삽입
    if self.isEmpty():                         # front == rear
        self.front = (self.front + 1) % self.capacity
```

- 테스트 프로그램

```
q = ArrayQueue(8)

q.display("초기 상태")
for i in range(6) :
    q.enqueue2(i)
q.display("삽입 0-5")

q.enqueue2(6); q.enqueue2(7)
q.display("삽입 6,7")          ← 이제

q.enqueue2(8); q.enqueue2(9)
q.display("삽입 8,9")

q.dequeue(); q.dequeue()
q.display("삭제  x2")
```

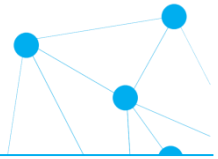
실행 결과

```
초기 상태= []
삽입 0-5= [0 1 2 3 4 5 ]
삽입 6,7= [1 2 3 4 5 6 7 ]
삽입 8,9= [3 4 5 6 7 8 9 ]
삭제  x2= [5 6 7 8 9 ]
```

포화 상태가 아니면 정상적으로 동작함

포화 상태에서 8, 9를 삽입하면 가장 먼저 들어온 요소들(1, 2)이 자동으로 삭제됨

2.5 파이썬에서 큐 사용하기



- 방법 1) queue 모듈의 Queue 사용하기

```
import queue                          # 파이썬의 queue 모듈 포함
q = queue.Queue(maxsize=20)           # 큐 객체 생성(최대크기 20)
```

연산	ArrayQueue	queue.Queue
삽입/삭제	enqueue(), dequeue()	put(), get()
공백/포화 상태 검사	isEmpty(), isFull()	empty(), full()
전단 들여다보기	peek()	제공하지 않음