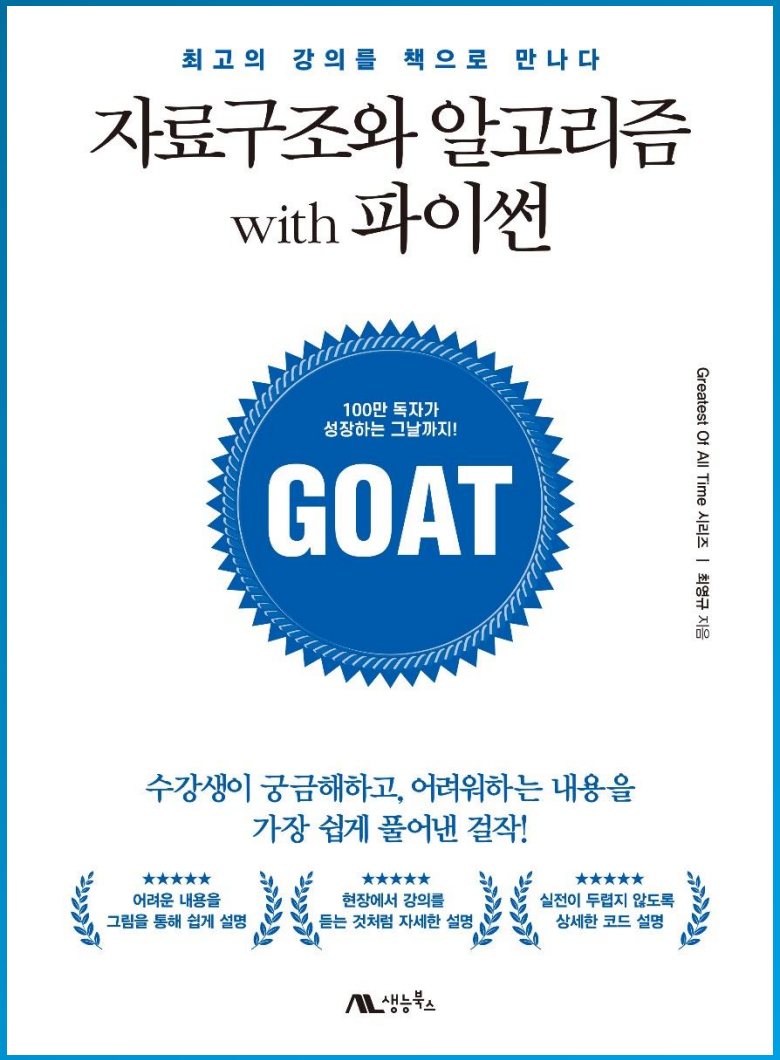
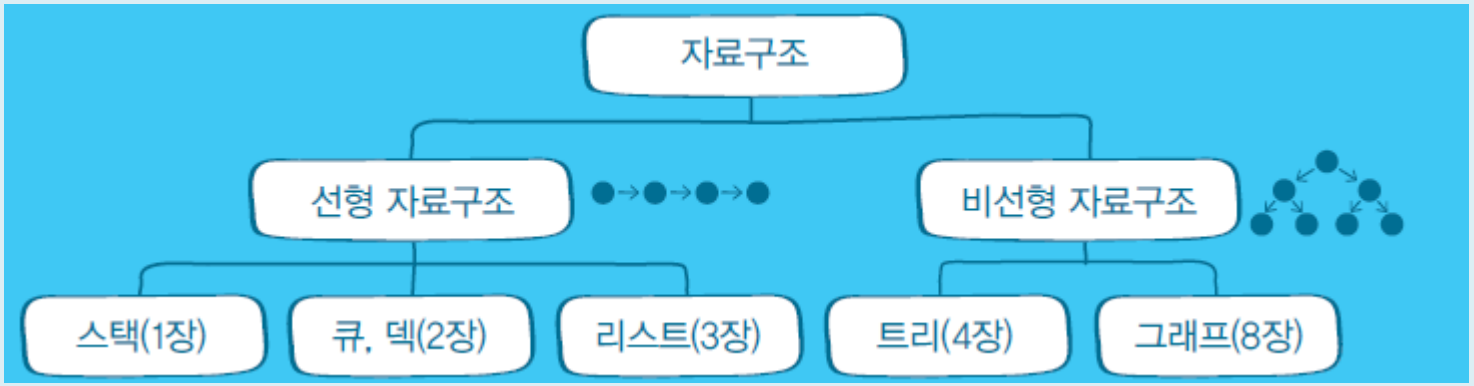
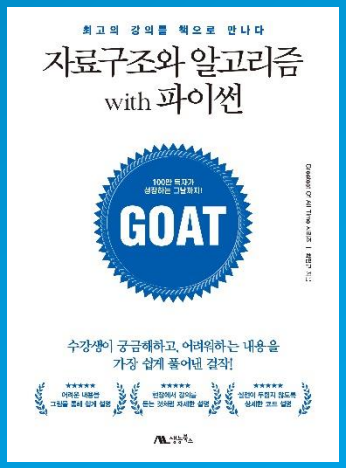


SW알고리즘개발
5주차

Part1. 자료구조



1장 스택 | 2장 큐 | 3장 리스트 | 4장 트리



03 CHAPTER

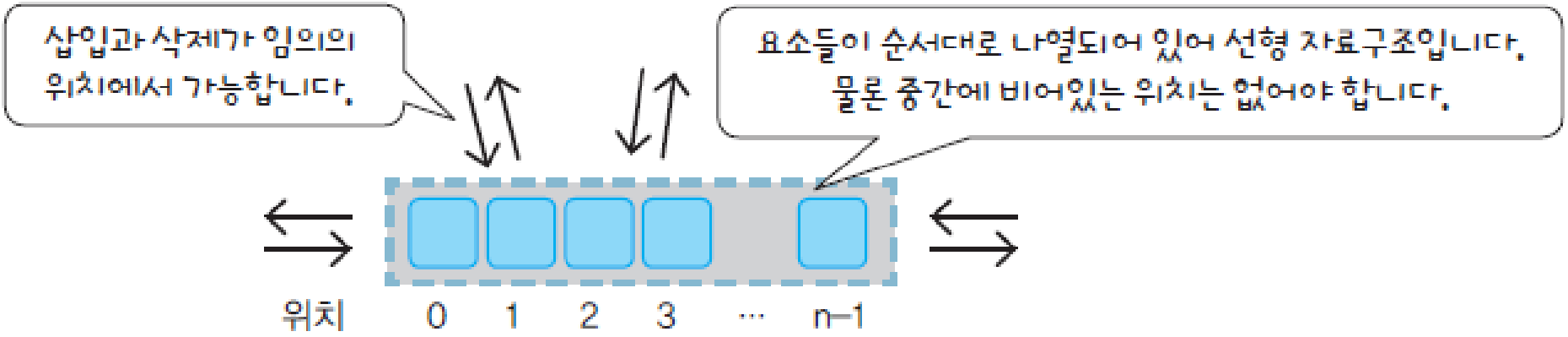
리스트

3장. 리스트

3.1 리스트란?

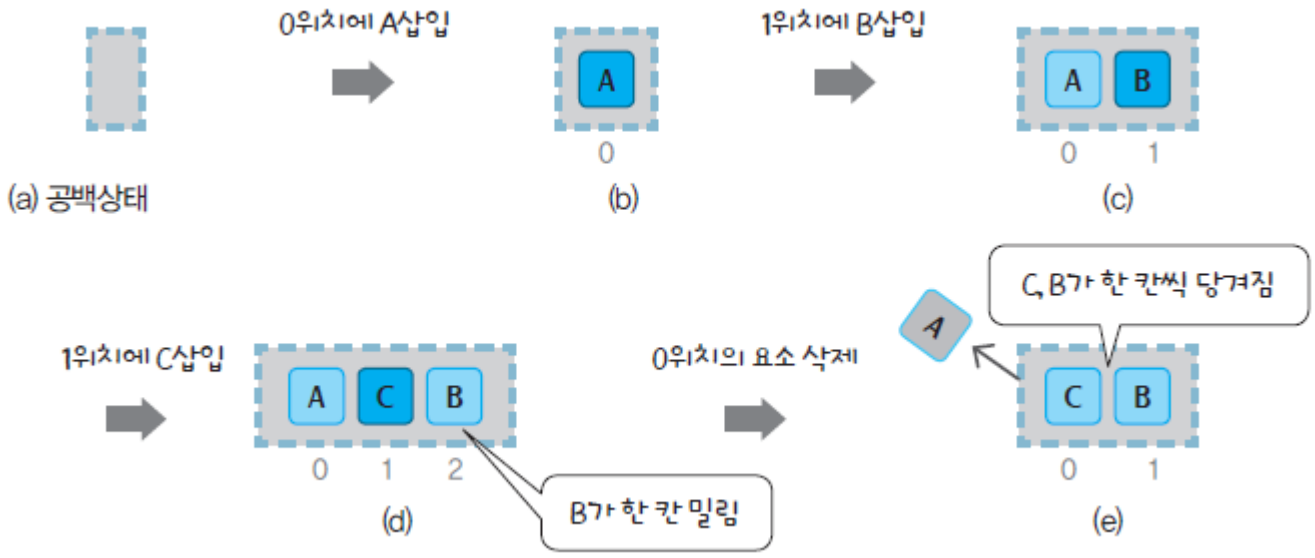
- 리스트(list) 또는 선형 리스트(linear list)
  - 자료들이 차례대로 나열된 자유로운 선형 자료구조
  - 각 자료는 순서 또는 위치(position)를 가짐

- 리스트의 구조



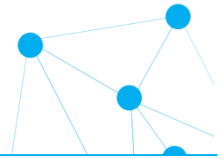
리스트란?

- 리스트의 연산 예



- 리스트와 집합(set)의 차이
  - 집합은 원소들 사이에 순서가 없고, 원소의 중복을 허용하지 않음. 특히 집합은 원소 사이에 순서의 개념이 없으므로 선형자료구조라 볼 수 없음

리스트의 연산들



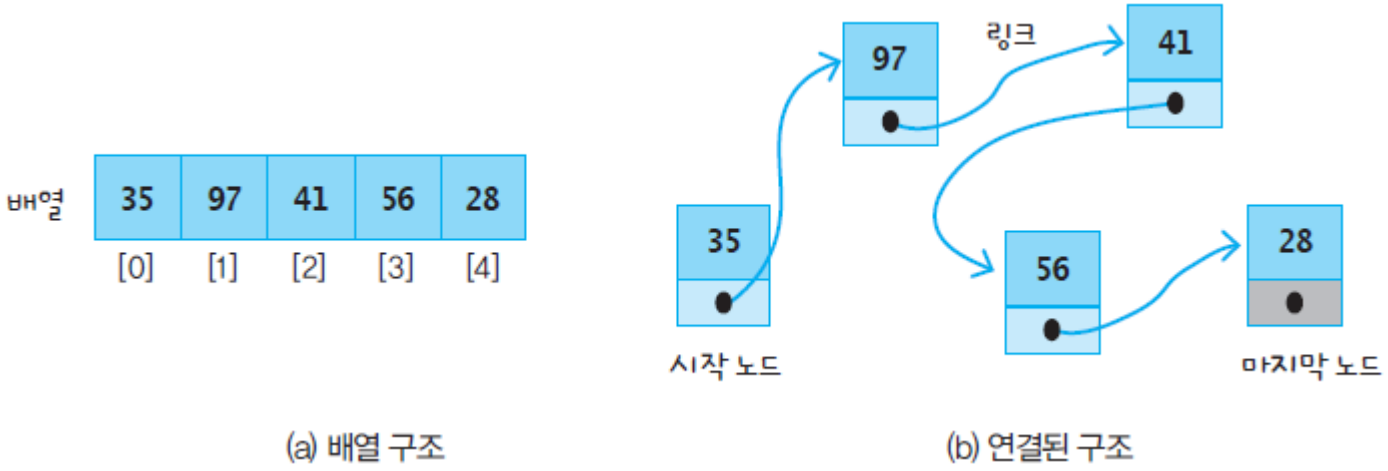
리스트의 연산

- insert(pos, e) : pos 위치에 새로운 요소 e를 삽입
- delete(pos) : pos 위치에 있는 요소를 꺼내서 반환
- getEntry(pos) : pos 위치에 있는 요소를 삭제하지 않고 반환
- isEmpty() : 리스트가 비어 있으면 True를 아니면 False를 반환
- isFull() : 리스트가 가득 차 있으면 True를 아니면 False를 반환
- size() : 리스트에 들어 있는 전체 요소의 수를 반환

- 삽입과 삭제 등의 연산에서 위치(pos)가 제공되어야 함
- 활용이 자유로워 추가적인 다양한 추가 연산이 가능
  - append(e), pop( ), find(e), replace(pos, e), display( ) 등

3.2 배열 구조 vs. 연결된 구조

- 리스트 구현 : 배열 구조와 연결된 구조
  - 배열: 메모리의 연속된 공간, 원하는 위치의 요소를 빠르게 참조
  - 연결된 구조: 메모리에 분산되어 있는 요소들을 링크를 이용하여 순서대로 연결해 하나로 관리
  - 연결된 리스트(linked list) : 자료들을 링크를 통해 일렬로 나열할 수 있는 연결된 구조

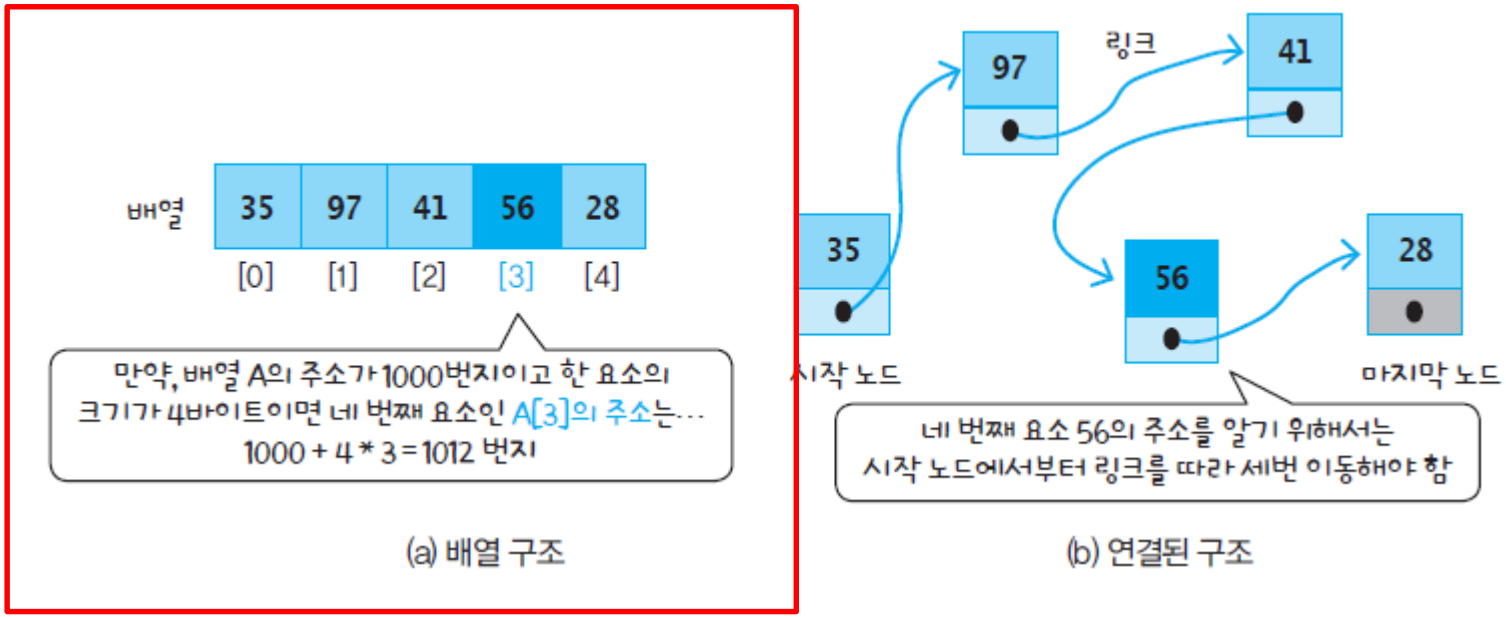


(a) 배열 구조 (b) 연결된 구조

- 연결된 구조
  - 노드(node) : 데이터 + 링크(link)

배열 구조의 리스트 vs. 연결 리스트

- **리스트 요소들에 대한 접근** : 배열 구조의 리스트



배열 구조의 리스트 vs. 연결 리스트

- **리스트의 용량** : 연결된 구조

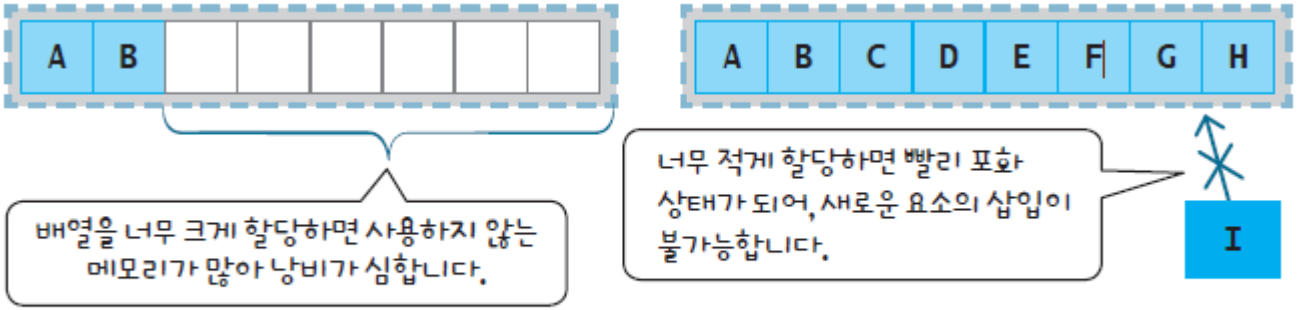


그림 3.5 | 배열은 용량이 고정됩니다.

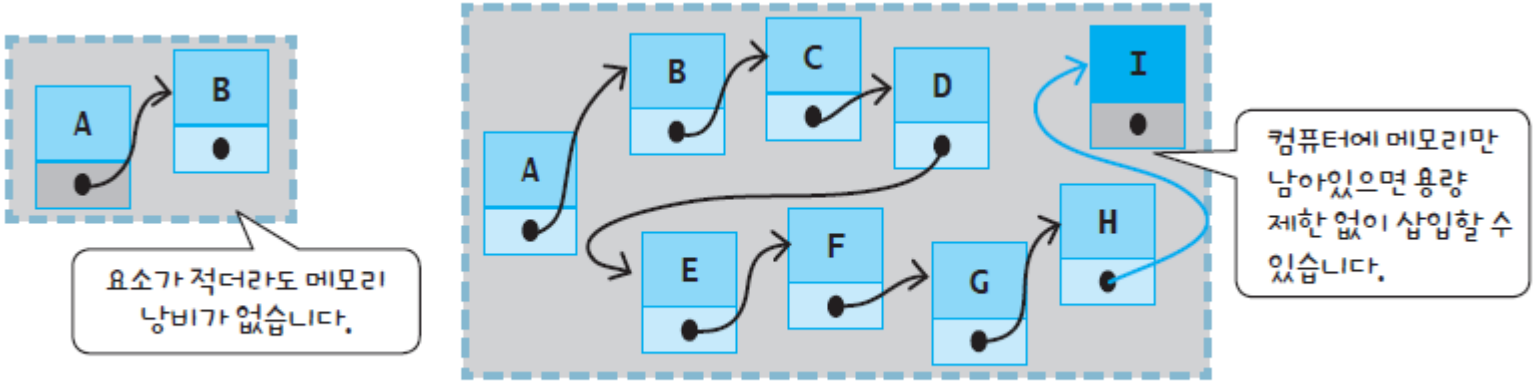
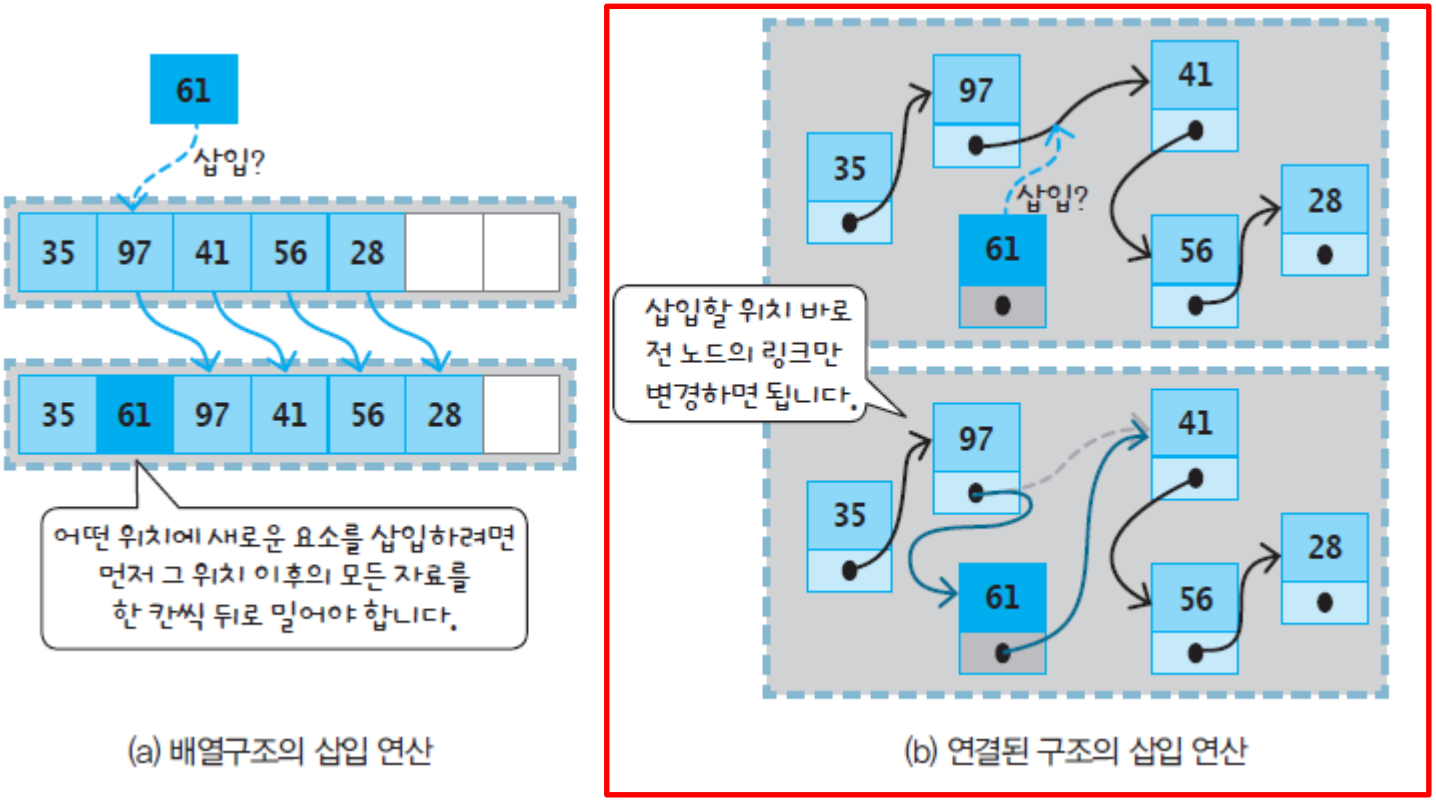


그림 3.6 | 연결된 구조는 용량이 고정되지 않습니다.

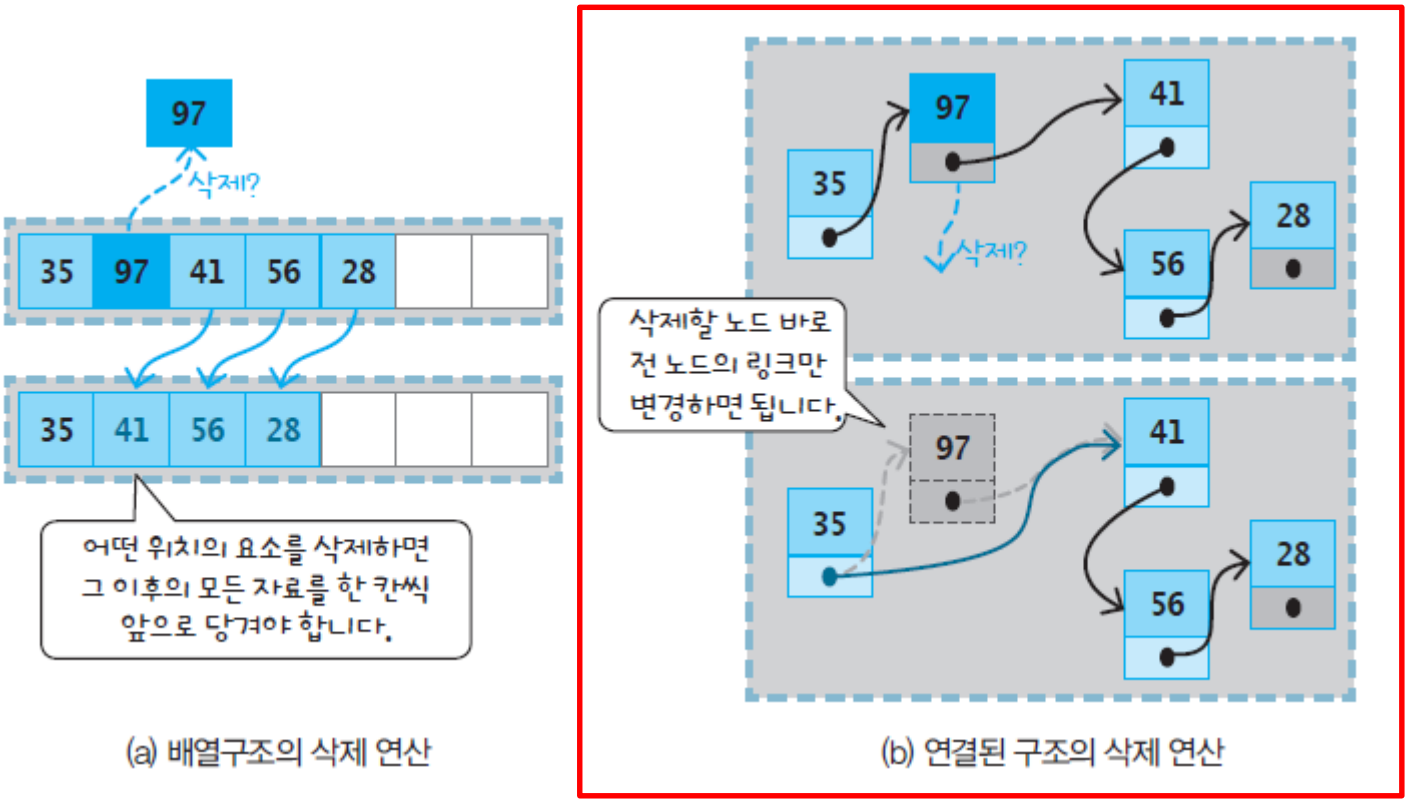
배열 구조의 리스트 vs. 연결 리스트

- **리스트의 삽입 연산 :** 연결된 구조



배열 구조의 리스트 vs. 연결 리스트

- **리스트의 삭제 연산 :** 연결된 구조



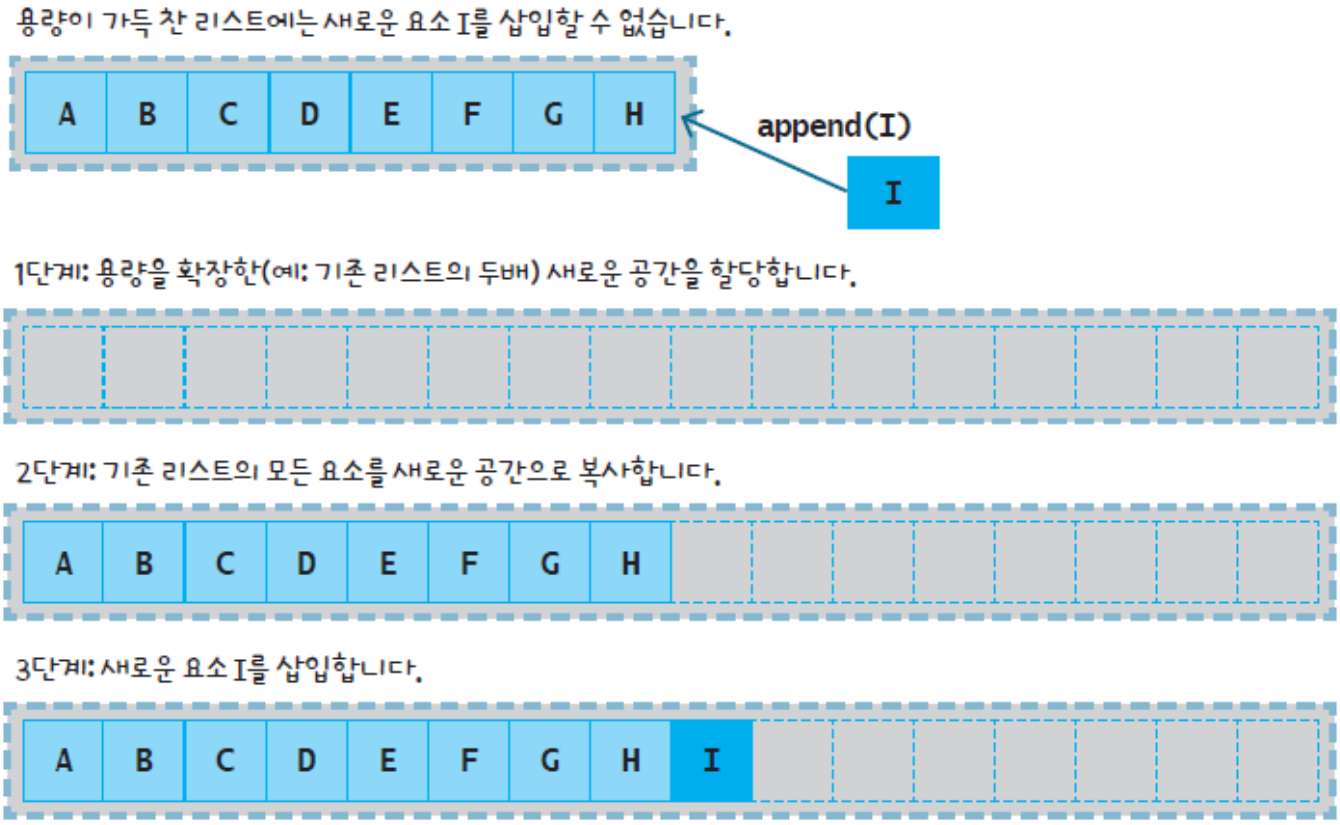
3.3 배열 구조의 리스트: 파이썬 리스트

- 파이썬의 리스트의 의미
  - 자료구조 리스트의 추상 자료형을 배열 구조로 구현
  - 원소의 접근은 효율적, 삽입과 삭제 연산은 비효율적

멤버함수(메서드)	설명
append(e)	새로운 요소 e를 추가합니다.
extend(lst)	리스트 lst를 리스트 s에 삽입합니다.
count(e)	리스트에서 요소 e의 개수를 세어 반환합니다.
index(e,[시작],[종료])	요소 e가 나타나는 가장 작은 위치(인덱스)를 반환합니다. 탐색을 위한 시작 위치와 종료 위치를 지정할 수도 있습니다.
insert(pos, e)	pos 위치에 새로운 요소 e를 삽입합니다.
pop(pos)	pos 위치의 요소를 꺼내고 반환합니다.
pop()	맨 뒤의 요소를 꺼내고 반환합니다.
remove(e)	요소 e를 리스트에서 제거합니다.
reverse()	리스트 요소들의 순서를 뒤집습니다.
sort([key], [reverse])	요소들을 key를 기준으로 오름차순으로 정렬합니다. reverse=True이면 내림차순으로 정렬합니다.

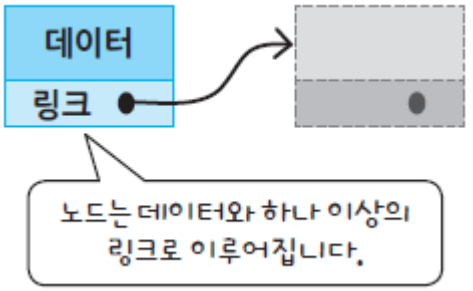
파이썬 리스트의 용량

- 용량이 제한되지 않도록 **동적 배열로 구현됨**
- 용량 제한을 해결 : **용량 확장** 방법 (내부적으로 처리)

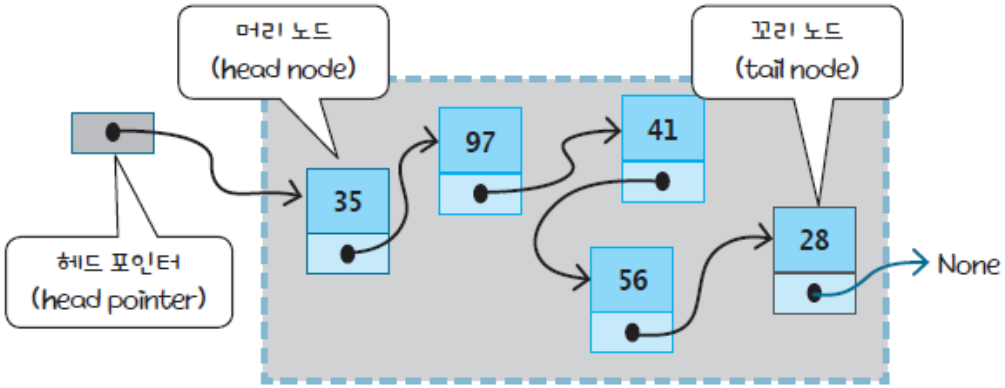


3.4 연결 리스트의 구조

- **노드(node)**는 하나의 데이터와 하나 이상의 링크(link)
  - 데이터: 배열 구조의 요소를 의미, 리스트에 저장
  - 링크 : 다른 노드의 주소를 저장하는 변수

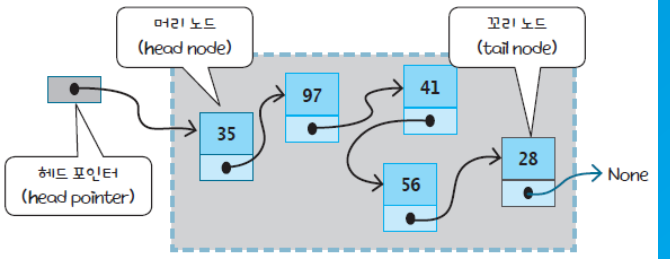


- 연결된 구조에서는 헤드 포인터를 잘 관리
  - 머리 노드만 알면 링크로 매달려 있는 모든 노드에 순차적 접근
  - 헤드 포인터 : 머리 노드의 주소를 저장하는 병수

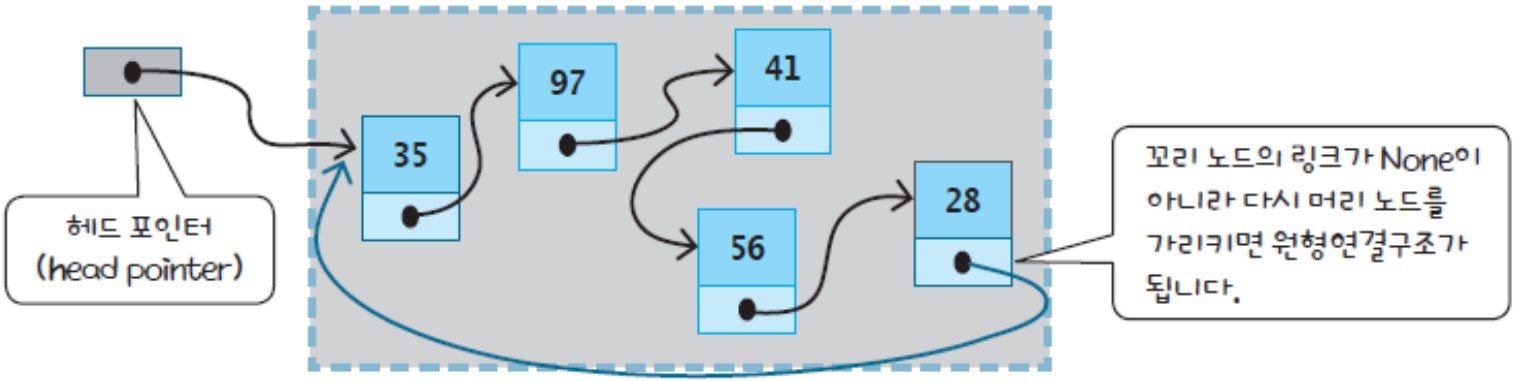


연결 리스트의 종류

- 노드의 형태와 연결된 구조에 따라 구분함
- 단순 연결 리스트(singly linked list)
  - 하나의 방향으로만 연결된 리스트
  - 노드는 하나의 링크만 갖음
  - 꼬리 노드의 링크가 None (마지막 노드을 의미)

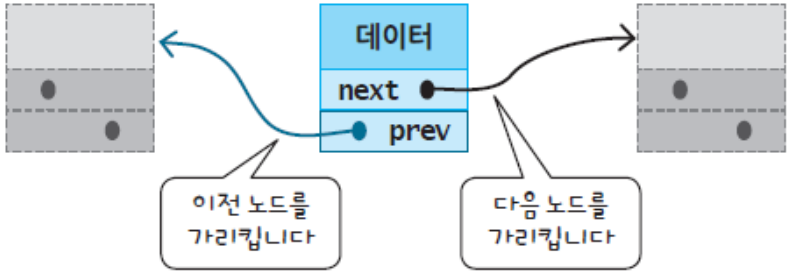


- 원형 연결 리스트(circular linked list)
  - 꼬리 노드의 링크가 머리 노드를 가리킴
  - 어떤 노드에서 시작해도 다른 모든 노드를 찾아 갈 수 있음

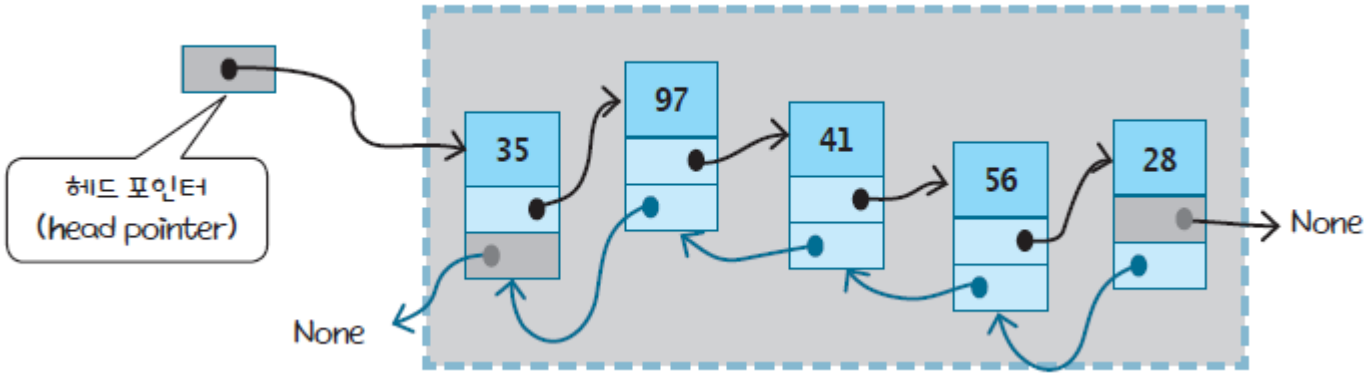


연결 리스트의 종류

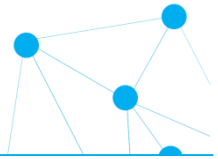
- 이중 연결 구조의 노드



- 이중 연결 리스트(doubly linked list)



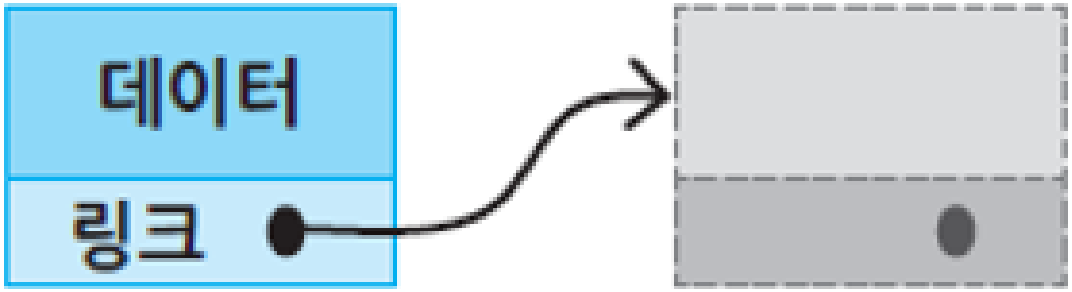
단순 연결 리스트의 응용 사례



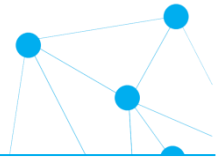
- **동적 메모리 관리**: 메모리를 효율적으로 관리하기 위해 사용 가능한 메모리 블록들을 추적하는데 사용
  - malloc(), free()
- **연결된 데이터 구조를 구현**: 순차적인 데이터를 저장하고 처리해야 할 때 사용
  - Undo/Redo 기능, 뒤로 가기/앞으로 가기
- **파일 시스템의 파일과 디렉토리 구조를 관리**: 각 디렉토리 안의 파일들이 노드로 연결되어 있으며, 하나의 파일이 다음 파일을 가리키는 방식으로 저장
  - FAT 파일 시스템
- **그래프와 트리의 인접 리스트 구현**
  - BFS, DFS, 소셜 네트워크 그래프
- **뮤직 플레이어의 재생 목록 관리:**
  - 각 노드는 음악 파일을 가리키며, 사용자는 다음 곡으로 이동하거나 특정 순서로 곡을 재생
- **폴더 탐색기에서의 최근 작업** : 사용자가 가장 최근에 열었던 문서 목록을 유지하여, 문서를 닫거나 다시 열 때 리스트에서 해당 항목을 빠르게 찾고 관리

3.5 단순 연결 구조로 리스트 구현하기

- **단순 연결 리스트**는 데이터를 노드 단위로 저장하고, 각 노드는 다음 노드를 가리키는 **링크**를 가지고 있는 구조
- **Node 클래스**와 **LinkedList 클래스**



Node 클래스



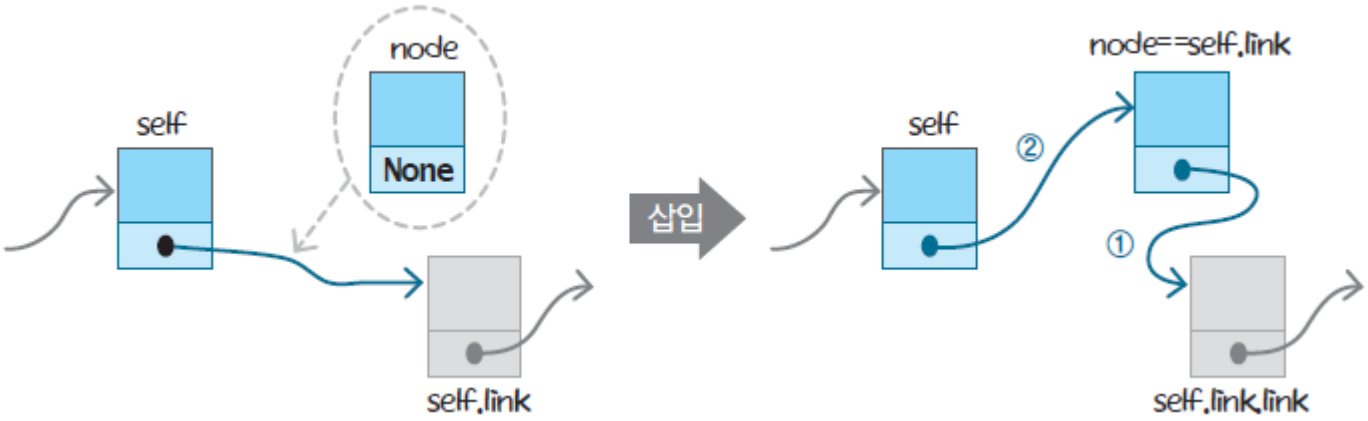
- 단순 연결 리스트의 Node 클래스 정의와 생성자
- '__init__(self, elem, next=None)' : 노드의생성자.
  - 'elem'에는 노드에 저장할 데이터가 들어감
  - 'link'에는 다음 노드를 가리키는 링크가 저장
  - 'link'가 'None'이면 마지막 노드임을 의미

디폴트 인수

```
class Node:
    def __init__ (self, elem, link=None):
        self.data = elem  # 데이터 멤버 생성 및 초기화
        self.link = link  # 링크 생성 및 초기화
```

Node 클래스

- 'append(self, node) 메서드 : 현재 노드(self) 뒤에 **새로운 노드(node)**를 추가하는 연산.
  - 새로운 추가된 노드는 현재 노드의 다음 노드와 연결됨

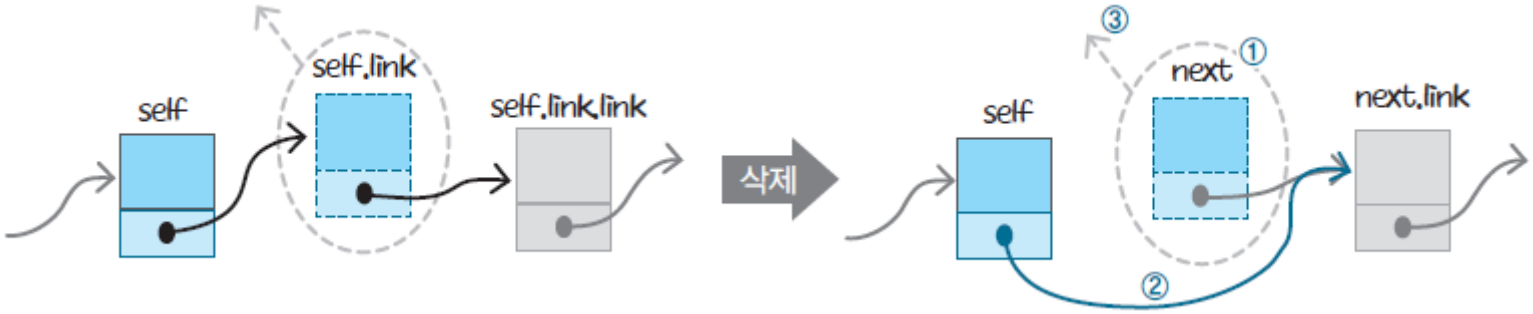



```
def append (self, node):      # self 다음에 node를 넣는 연산
    if node is not None :
        node.link = self.link
        self.link = node
```

← 삽입할 노드가 None이 아니면 ①과 ②단계를 통해 node를 다음 노드로 연결

Node 클래스

- **'popNext(self)'** 메서드 : 현재 노드(self)**의 다음 노드를** 삭제하는 연산
  - 삭제된 다음 노드를 반환
  - 현재 노드의 링크는 다음 노드의 링크로 갱신됨

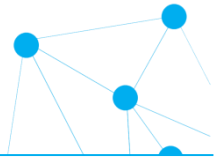



```
def popNext (self):                    # self의 다음 노드를 삭제하는 연산
    next = self.link                   # 현재 노드(self)의 다음 노드
    if next is not None :
        self.link = next.link
    return next                        # 다음 노드를 반환
```

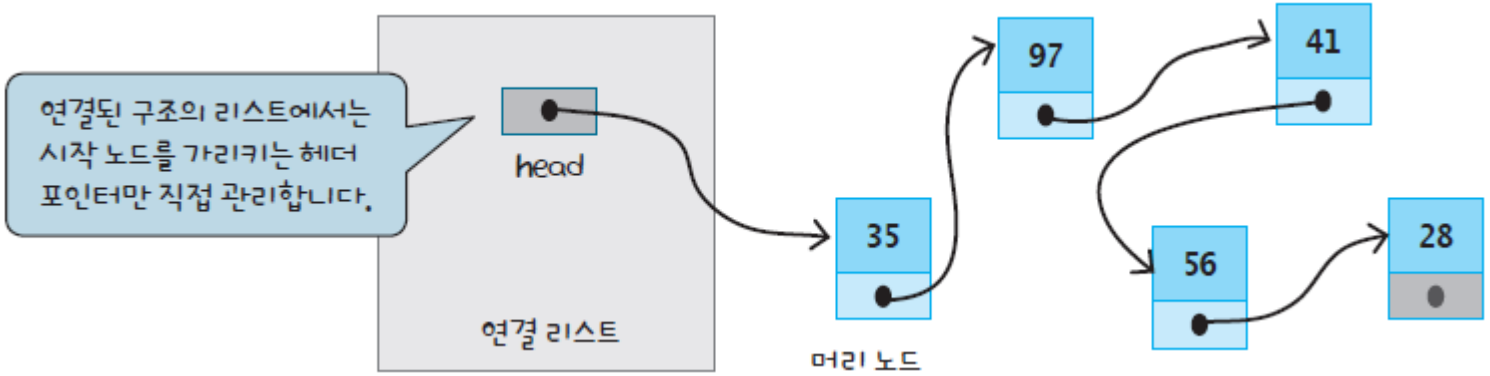
① 단계

← next가 None이 아니면 ②단계 처리

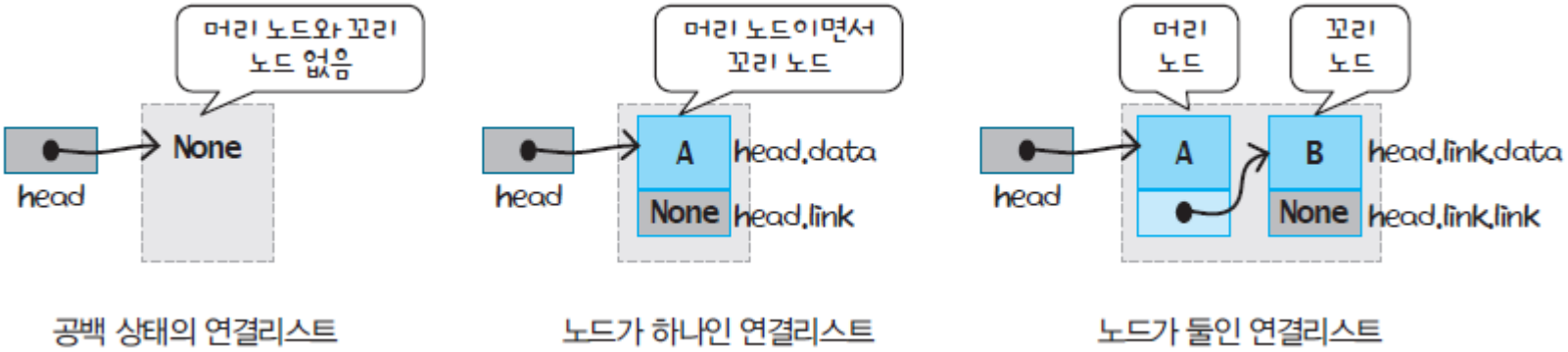
단순 연결 리스트 클래스



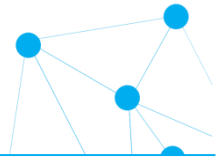
- 연결된 구조의 리스트에서는 헤더 포인터만 관리



- 예:



LinkedList 클래스



- 단순 연결 리스트 클래스 정의와 생성자
  - **'__init__(self)'** : 'head'는 리스트의 시작 노드를 가리킨다. 초기 값은 'None'으로 설정되어 리스트가 비어 있음을 의미

```
class LinkedList:                    # 단순 연결 리스트 클래스
    def __init__( self ):            # 생성자
        self.head = None             # head 선언 및 None으로 초기화
```

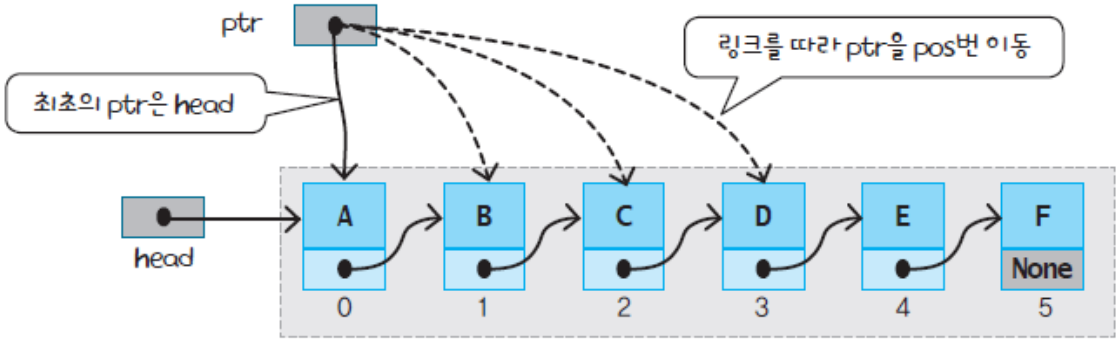
- 공백 상태와 포화 상태를 검사하는 isEmpty( )와 isFull( )
  - **'isEmpty(self)'** : 'head'가 'None'이면 비어있는 상태
  - **'isFull(self)'** : 단순 연결 리스트는 동적으로 크기가 변하기 때문에 포화 상태가 발생하지 않음

```
def isEmpty( self ):                 # 공백 상태 검사
    return self.head == None         # head가 None이면 공백

def isFull( self ):                  # 포화 상태 검사
    return False                     # 연결된 구조에서는 포화 상태 없음
```

LinkedList 클래스

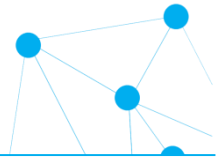
- **'getNode(self, pos)'** : 리스트에서 **pos 번째 노드를 반환**
  - 'pos'가 음수이면 None을 반환
  - 'head'부터 시작해 'pos-1' 번째까지 이동



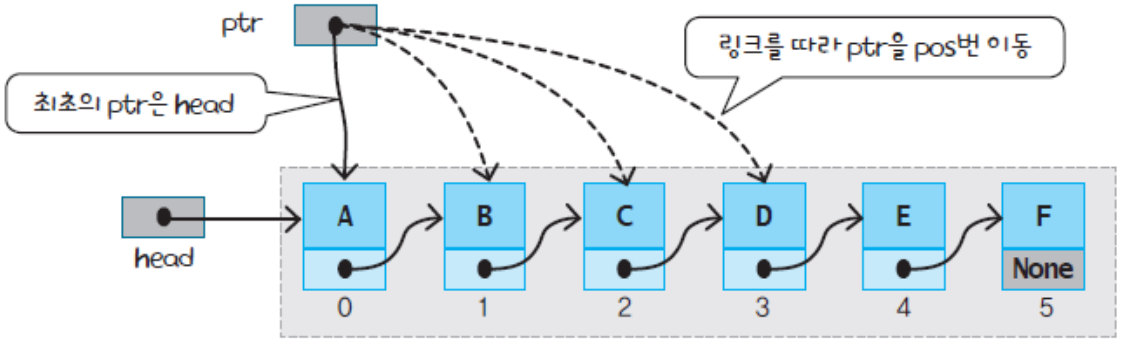

```
def getNode(self, pos) :
    if pos < 0 : return None          # 잘못된 위치 -> None 반환
    ptr = self.head                   # 시작 위치 -> head
    for i in range(pos):
        if ptr == None :
            return None
        ptr = ptr.link
    return ptr                        # 최종 노드를 반환
```

머리 노드에서부터 링크를 따라 pos번 이동하면 pos 위치의 노드에 도착.
위치는 0부터 시작한다고 가정함.

LinkedList 클래스



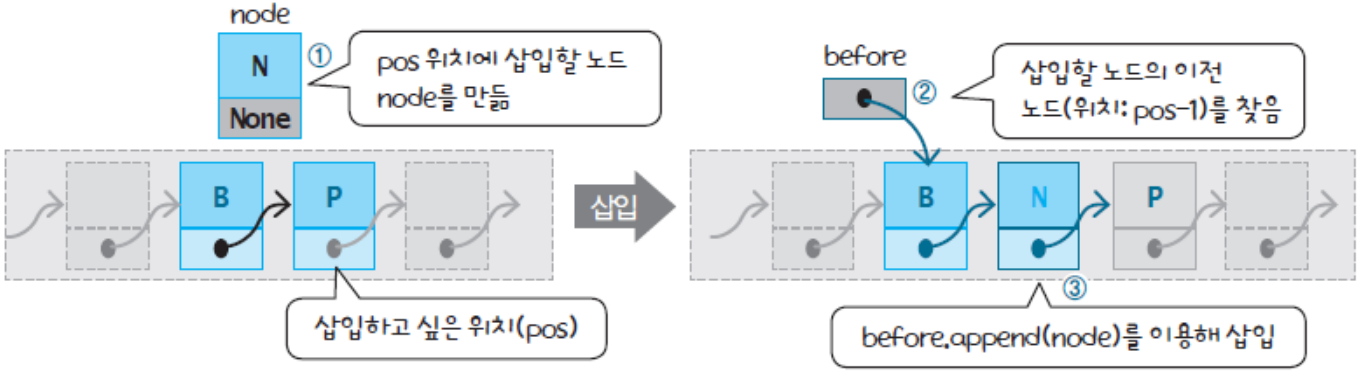
- **'getEntry(self, pos)'** : 리스트에서 **pos 번째 노드의 데이터를 반환**




```
def  getEntry(self, pos) :
        node = self.getNode(pos)   # pos번째 노드을 구함
        if node == None : return None   # 해당 노드가 없는 경우
        else :  return node.data        # 있는 경우 데이터 필드 반환
```

LinkedList 클래스

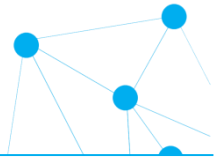
- **'insert(self, pos, e)'**: 리스트에서 pos 위치에 데이터 e를 삽입
  - 새로운 노드를 생성
  - 'pos-1' 번째 노드(before)를 찾은 후 그 다음에 새 노드를 추가
  - 'pos'가 0이면 'head' 노드로 삽입



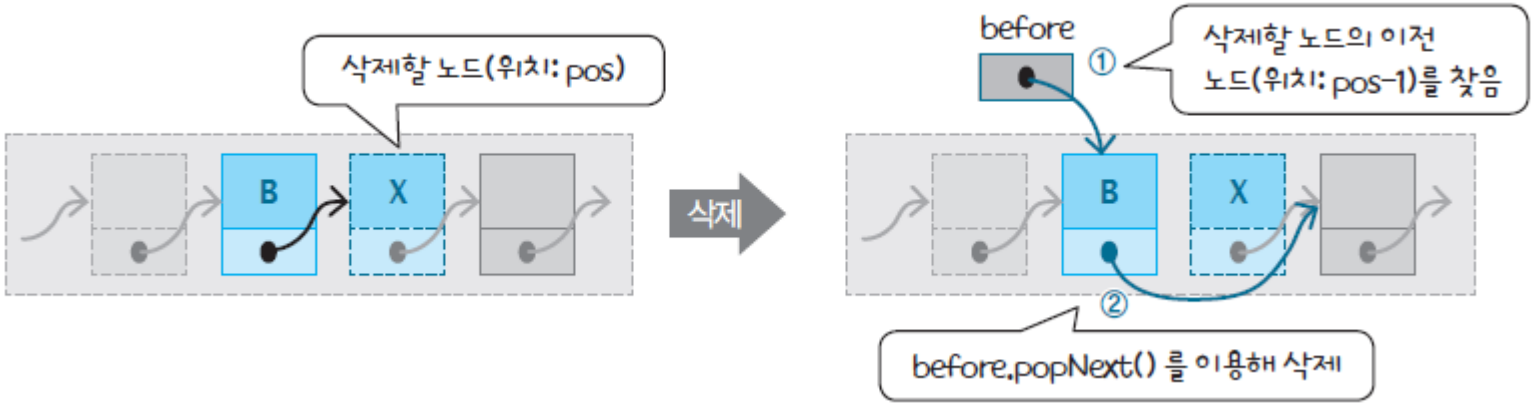

```
def insert(self, pos, e) :
    node = Node(e, None)                # 삽입할 새로운 노드를 만듦
    before = self.getNode(pos-1)        # 삽입할 위치 이전 노드 탐색
    if before == None :
        node.link = self.head
        self.head = node
    else : before.append(node)          # 아닌 경우: before뒤에 추가
```

← before가 None이면 맨 앞에 추가, 리스트의 머리노드(head)가 변경됨.

LinkedList 클래스



- **delete(self, pos):** 리스트에서 pos 번째 위치의 요소를 삭제
  - pos - 1 번째 노드(before)를 찾아서 그 다음 노드를 삭제
  - pos가 0이면 head 노드을 삭제

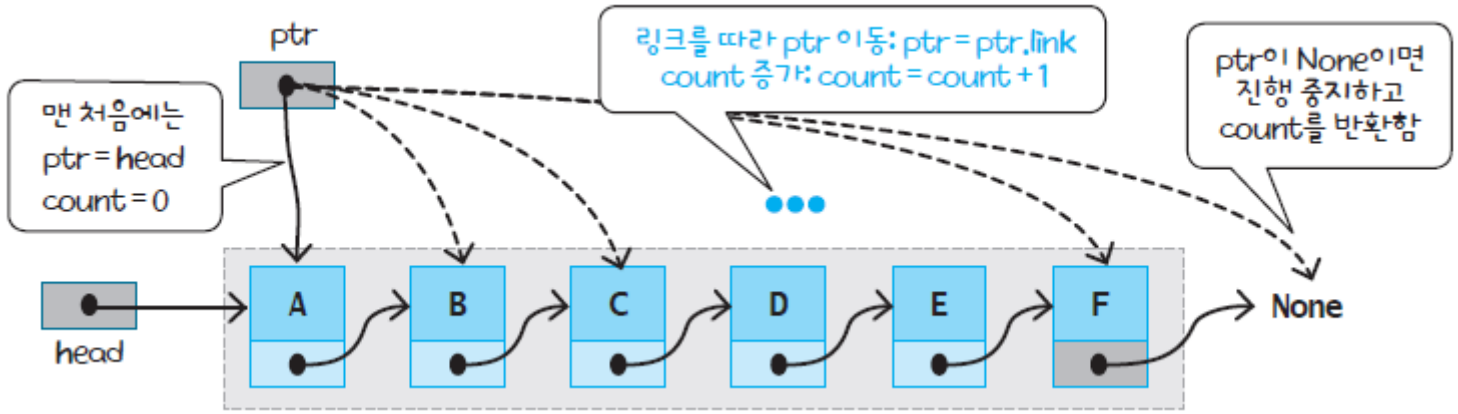



```
def delete(self, pos) :
    before = self.getNode(pos-1)          # 삭제할 위치 이전 노드 탐색
    if before == None :
        before = self.head
        if self.head is not None :
            self.head = self.head.link
        return before
    else: return before.popNext()         # before의 다음 노드 삭제
```

← 머리노드를 삭제하면 head가 다음 노드로 변경됨.

LinkedList 클래스

- **size( ) :** 리스트의 노드 수를 반환하는 함수. 'head' 부터 시작해서 노드를 하나씩 따라가며 카운트를 증가시켜 리스트의 크기를 계산

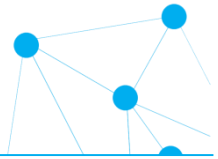



```
def size( self ) :
    ptr = self.head
    count = 0;
    while ptr is not None :
        ptr = ptr.link
        count += 1
    return count
```

머리노드부터 링크를 따라
None이 될 때까지 이동하면서
이동 횟수를 기록함.

```
    # ptr이 None이 아닌 동안
    # 링크를 따라 ptr 이동
    # 이동할 때마다 count 증가
    # count 반환
```

LinkedList 클래스



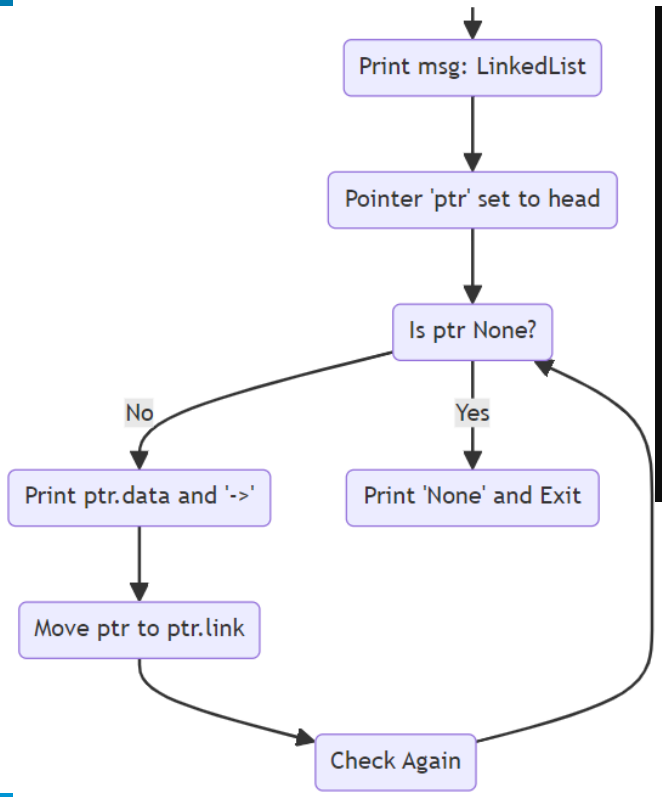
- **replace(pos, elem) :** 단순 연결 리스트에서 특정 위치의 데이터를 새 값으로 교체하는 역할.
  - getNode(pos) 함수를 사용하여 해당 노드를 찾아 데이터를 바꾸고, 유효하지 않은 위치에서는 아무 작업도 하지 않습니다.

```
def replace(self, pos, elem):
    node = self.getNode(pos)
    if node is not None:
        node.data = elem
```

1. **리스트 상태**: `10 -> 20 -> 30 -> 40 -> None`
2. **호출**: `replace(2, 50)`
   - 이 호출은 리스트의 2번째 노드의 데이터를 `50`으로 교체하라는 의미입니다. (0부터 시작하는 인덱스 기준)
3. **과정**:
   - `getNode(2)` 함수가 호출되어, 두 번째 노드 `30`을 가져옵니다.
   - 두 번째 노드가 유효하므로, 이 노드의 데이터 `30`을 `50`으로 교체합니다.
4. **결과**: `10 -> 20 -> 50 -> 40 -> None`

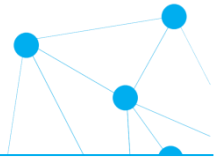
LinkedList 클래스

- **display( ) :** 단순 연결 리스트의 노드에 저장된 **데이터를 출력**
  - 각 노드의 데이터를 순차적으로 탐색하면서 출력
  - 노드 간의 연결을 화살표(->) 로 표시합니다. 마지막에 'None'로 출력하여 리스트의 끝을 나타냅니다.




```
def display(self, msg='LinkedList:'):
    print(msg, end='')
    ptr = self.head
    while ptr is not None:
        print(ptr.data, end='->')  # 각 노드의 데이터를 출력
        ptr = ptr.link
    print('None')  # 마지막에 None 출력
```

연결 리스트와 파이썬 리스트 비교



단순 연결 리스트(LinkedList)	파이썬의 리스트
s = LinkedList()	l = []
s.display('연결리스트(초기): ')	print('파이썬list(초기):', l)
s.insert(0, 10)	l.insert(0, 10)
s.insert(0, 20)	l.insert(0, 20)
s.insert(1, 30)	l.insert(1, 30)
s.insert(s.size(), 40)	l.insert(len(l), 40)
s.insert(2, 50)	l.insert(2, 50)
s.display("연결리스트(삽입x5): ")	print('파이썬list(삽입x5):', l)
s.replace(2, 90)	l[2] = 90
s.display("연결리스트(교체x1): ")	print('파이썬list(교체x1):', l)
s.delete(2)	l.pop(2)
s.delete(3)	l.pop(3)
s.delete(0)	l.pop(0)
s.display("연결리스트(삭제x3): ")	print('파이썬list(삭제x3):', l)

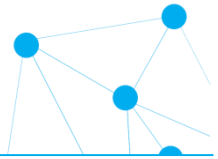
실행 결과

```
연결리스트( 초기 ): None
연결리스트(삽입x5): 20->30->50->10->40->None
연결리스트(교체x1): 20->30->90->10->40->None
연결리스트(삭제x3): 30->10->None
```

실행 결과

```
파이썬list( 초기 ): []
파이썬list(삽입x5): [20, 30, 50, 10, 40]
파이썬list(교체x1): [20, 30, 90, 10, 40]
파이썬list(삭제x3): [30, 10]
```

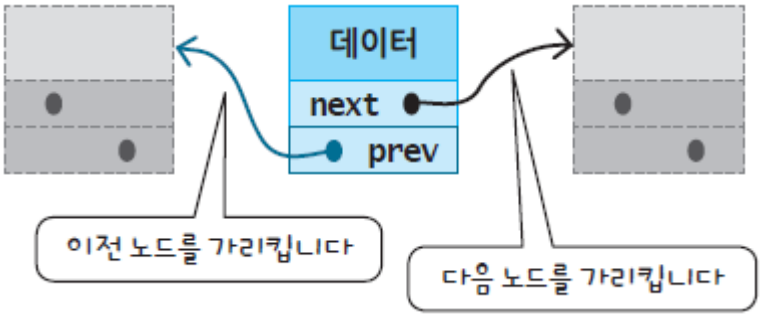
프로그래밍 문제: 도서 관리 시스템



- 단순 연결리스트를 활용하여 도서관의 도서 목록을 관리
- 이 프로그램은 도서 추가, 삭제, 검색, 목록 출력 기능
- 각 도서는 제목과 저자를 포함한 정보를 가지고 있으며, 동적으로 도서를 추가, 삭제, 검색
- **요구사항:**
  - **도서 추가**: 도서의 제목과 저자를 입력받아 도서 목록에 추가.
  - **도서 삭제**: 특정 위치에 있는 도서를 삭제.
  - **도서 검색**: 입력한 제목으로 도서를 검색하여, 해당 도서가 목록에 있는지 확인하고, 있으면 저자 정보를 출력.
  - **도서 목록 출력**: 현재 목록에 있는 모든 도서의 제목과 저자 정보를 출력
  - **도서 수 확인**: 현재 도서 목록에 있는 도서의 총 개수를 출력

3.6 이중 연결 구조로 리스트 구현하기

- 이중 연결 구조의 노드 클래스

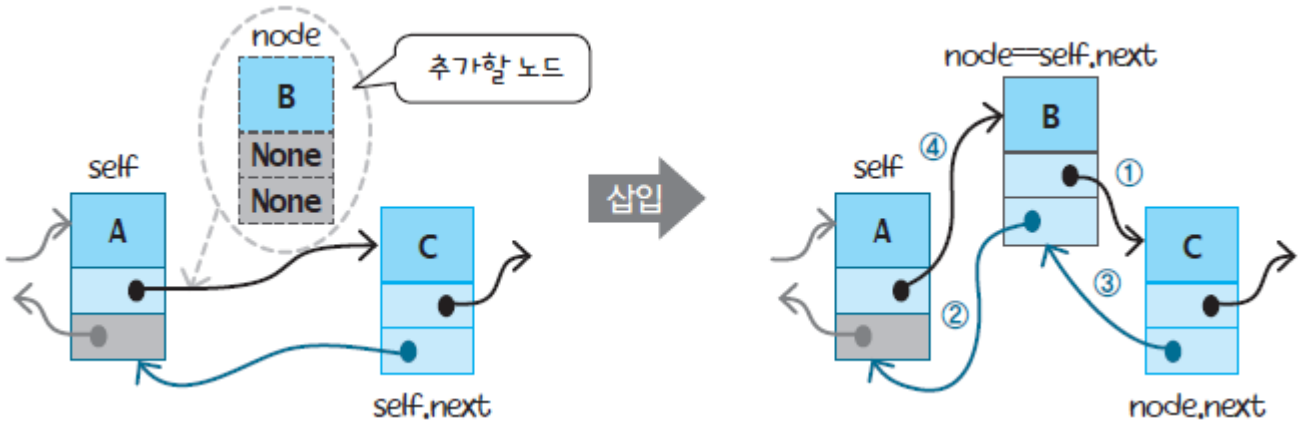



```
class DNode:
    def __init__ (self, elem, prev=None, next=None):
        self.data = elem  # 노드의 데이터 필드(요소)
        self.next = next  # 다음 노드를 위한 링크
        self.prev = prev  # 이전 노드를 위한 링크(추가됨)
```

← 이중 연결 노드의 생성자

DNode 클래스

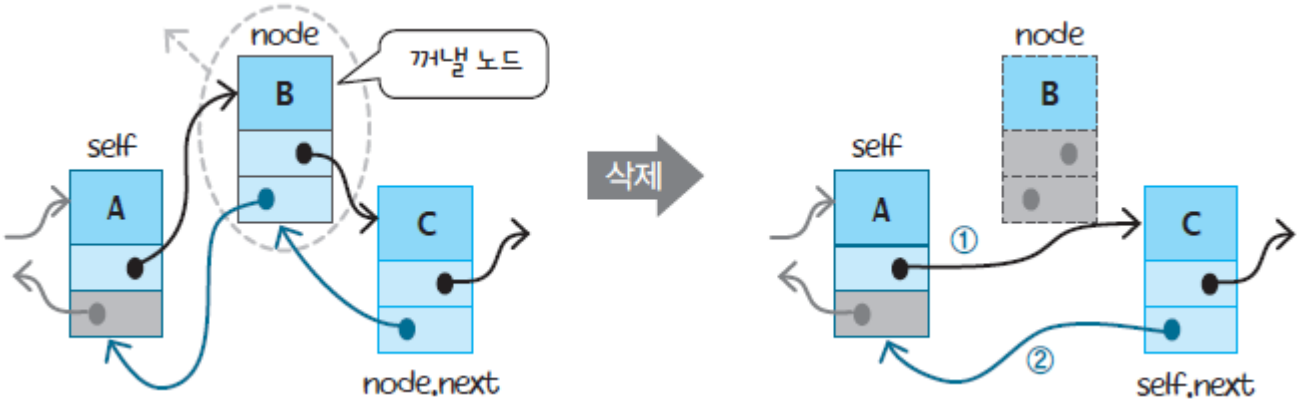
- 새로운 노드를 뒤에 추가하는 append( )




```
def append (self, node):              # self 다음에 node를 넣는 연산
    if node is not None :             # node가 None이 아니면
        node.next = self.next         # ①
        node.prev = self              # ②
        if node.next is not None:     # ③ self의 다음 노드가 있으면
            node.next.prev = node     # 그 노드의 이전 노드는 node
        self.next = node              # ④
```

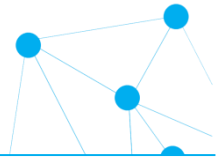
DNode 클래스

- 다음 노드를 연결 구조에서 꺼내는 popNext( )




```
def popNext (self):                 # self 다음 노드 삭제 연산
    node = self.next                # 삭제할 노드
    if node is not None :           # next가 None이 아니면
        self.next = node.next       # ①
        if self.next is not None:   # ② 다음 노드가 있으면
            self.next.prev = self   #   그 노드의 이전 노드는 self
    return node                     # 다음 노드를 반환
```

이중 연결 리스트 클래스

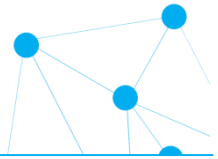


- 이중 연결 리스트 클래스 정의와 생성자

```
class DblLinkedList:                    # 이중 연결 리스트 클래스
    def __init__( self ):               # 생성자
        self.head = None                # head 선언 및 None으로 초기화
```

- 대부분의 연산들은 LinkedList 클래스에서와 거의 유사
  - Node를 DNode로 수정
  - .link를 .next로 수정
  - isEmpty( ), isFull( ), getEntry(pos), display( ), …

이중 연결 리스트 클래스 테스트



실행 결과

```
연결리스트( 초기 ): None
연결리스트(삽입 x 5): 20<=>30<=>50<=>10<=>40<=>None
연결리스트(교체 x 1): 20<=>30<=>90<=>10<=>40<=>None
연결리스트(삭제 x 3): 30<=>10<=>None
```

이중연결구조를 표시하기 위해 사용