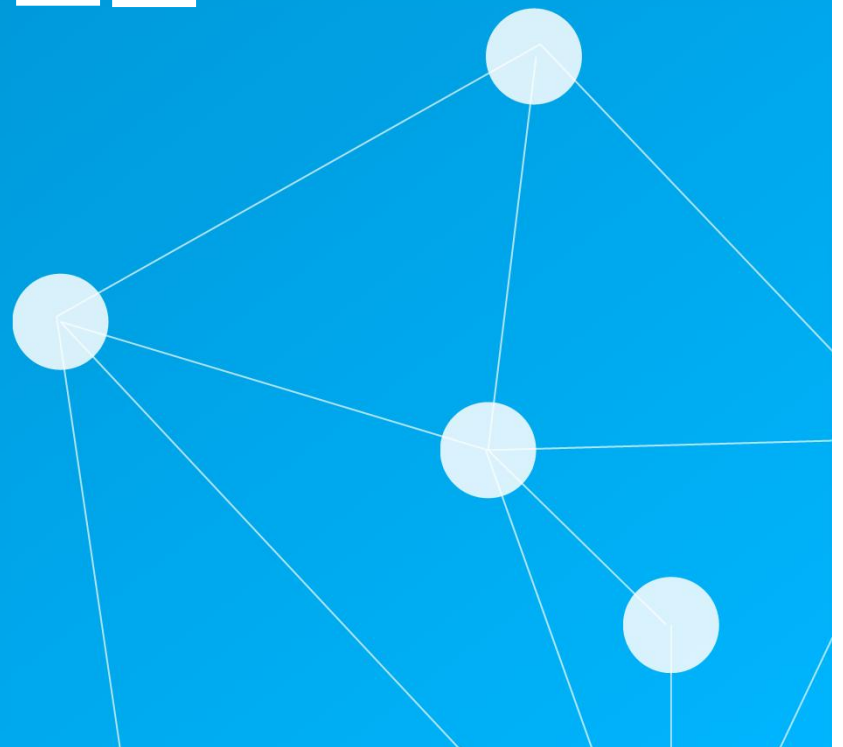
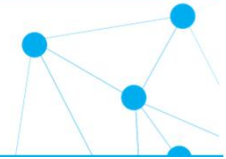


# 리스트

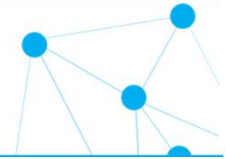


# 1. 리스트란?



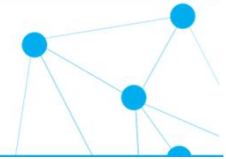
- 리스트는 가장 자유로운 선형 자료구조이다.
  - 리스트의 구조
  - 리스트의 추상 자료형
- 리스트의 구현 방법
  - 배열 구조와 연결된 구조로 구현할 수 있다.
  - 리스트와 관련된 용어의 정리

# 리스트란?

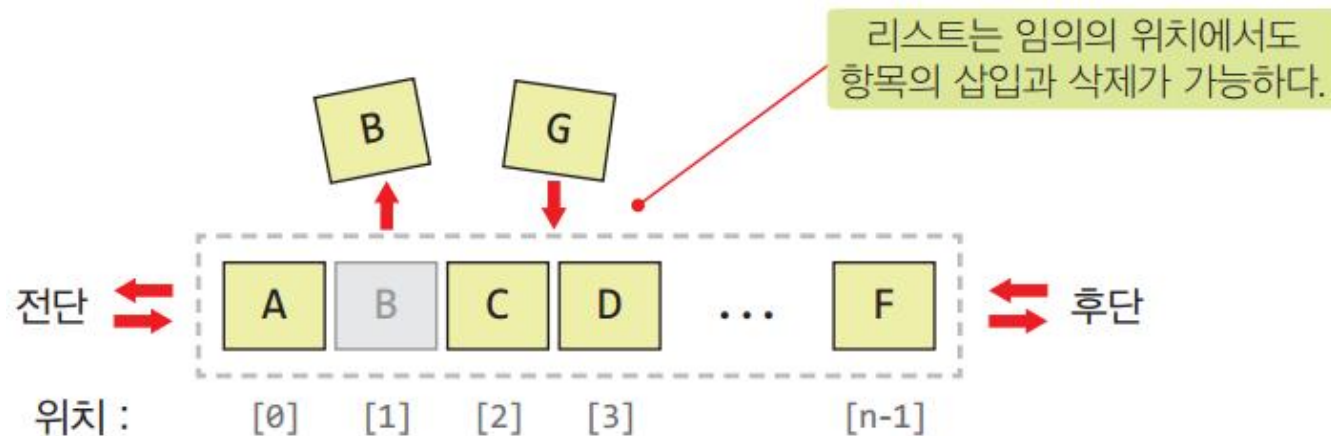


- 리스트(list), 선형리스트(linear list)
  - 순서를 가진 항목들의 모임
  - $L = [item_0, item_1, item_2, \dots, item_{n-1}]$ 
    - 0번째 원소   1번째 원소                      n-1번째 원소
  - 집합: 항목간의 순서의 개념이 없음  $\Rightarrow$  집합도 자료구조임. 원소들의 모임.

# 리스트의 구조

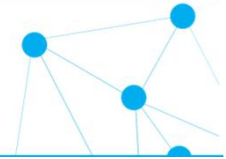


- 리스트
  - 항목들이 순서대로 나열되어 있고, 각 항목들은 위치를 갖는다.



- Stack, Queue, Deque과의 차이점
  - 자료의 접근 위치

# 리스트 ADT



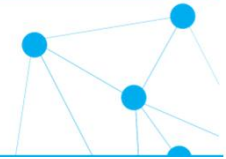
## 정의 3.1 List ADT

데이터: 같은 유형의 요소들의 순서 있는 모임

연산

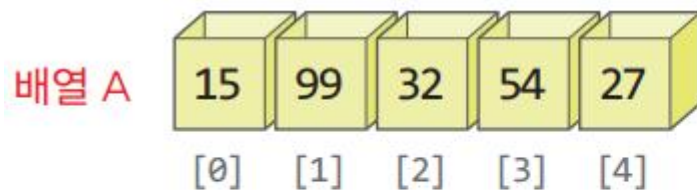
- `List()`: 비어 있는 새로운 리스트를 만든다.
- `insert(pos, e)`: `pos` 위치에 새로운 요소 `e`를 삽입한다.
- `delete(pos)`: `pos` 위치에 있는 요소를 꺼내고(삭제) 반환한다.
- `isEmpty()`: 리스트가 비어있는지를 검사한다.
- `getEntry(pos)`: `pos` 위치에 있는 요소를 반환한다.
- `size()`: 리스트안의 요소의 개수를 반환한다.
- `clear()`: 리스트를 초기화한다.
- `find(item)`: 리스트에서 `item`이 있는지 찾아 인덱스를 반환한다.
- `replace(pos, item)`: `pos`에 있는 항목을 `item`으로 바꾼다.
- `sort()`: 리스트의 항목들을 어떤 기준으로 정렬한다.
- `merge(lst)`: 다른 리스트 `lst`를 리스트에 추가한다.
- `display()`: 리스트를 화면에 출력한다.
- `append(e)`: 리스트의 맨 뒤에 새로운 항목을 추가한다.

# 리스트 구현 방법



- 배열 구조

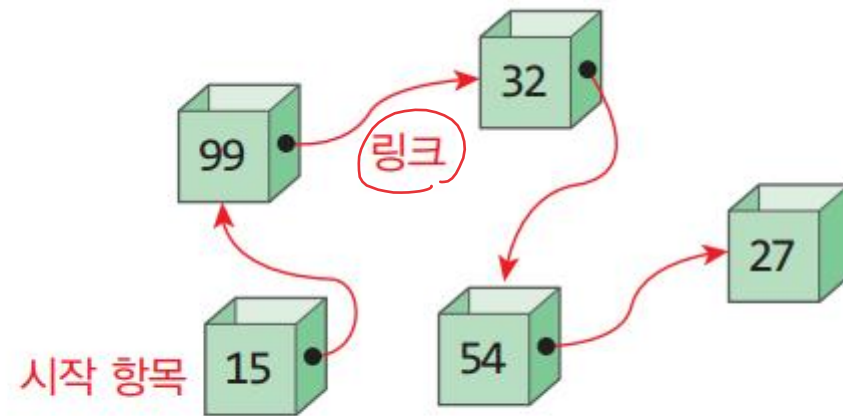
- 구현이 간단
- 항목 접근이  $O(1)$
- 삽입, 삭제가 오버헤드
- 항목의 개수 제한



배열 구조의 리스트

- 연결된 구조

- 구현이 복잡
- 항목 접근이  $O(n)$
- 삽입, 삭제가 효율적
- 크기가 제한되지 않음

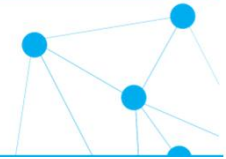


연결된 구조의 리스트

→ 파이썬으로 구현

- 배열 파이썬 리스트
- 연결 리스트

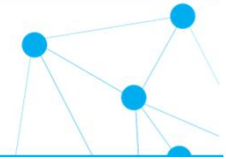
## 2. 파이썬 리스트



- 파이썬의 리스트는 스마트한 배열이다.
- 파이썬의 리스트는 동적 배열로 구현되었다.
- 파이썬 리스트의 시간 복잡도



# 파이썬 리스트



- 파이썬 리스트는 스마트한 배열이다
- 리스트 선언

```
A = [ 1, 2, 3, 4, 5 ]          # 파이썬 리스트 A
```

– C 언어의 배열 선언

```
int A[5] = { 1, 2, 3, 4, 5 };    // 정수 배열 A선언 및 초기화
```

- 항목의 수

```
print('파이썬 리스트 A의 크기는 ', len(A))    # A의 크기(항목 수) 출력
```

- 항목 추가: 용량을 늘릴 수 있다.

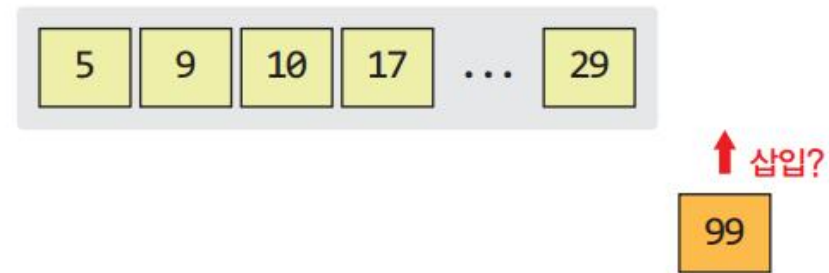
```
A.append(6)          # A = [1, 2, 3, 4, 5, 6]  
A.append(7)          # A = [1, 2, 3, 4, 5, 6, 7]  
A.insert(0, 0)       # A = [0, 1, 2, 3, 4, 5, 6, 7]
```

# 파이썬 리스트는 동적 배열로 구현되었다

- 필요한 양보다 넉넉한 크기의 메모리를 사용!



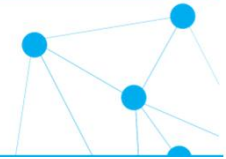
(크기 < 용량) 인 상황에서의 항목 삽입



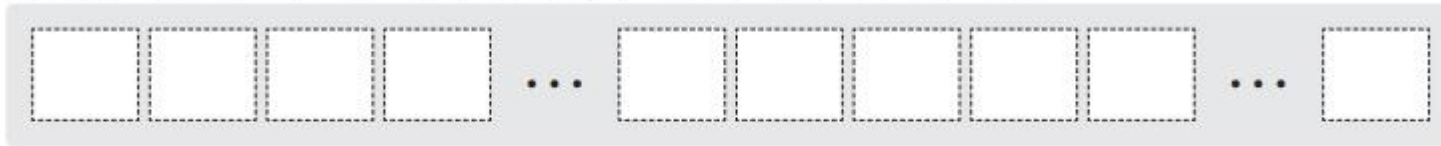
(크기 == 용량) 인 상황에서의 항목 삽입

- 남은 공간이 없으면 어떻게 삽입할까?

# 동적 배열 구조에서의 용량 증가 과정



Step1: 용량을 확장한 새로운 배열 할당. (예: 기존 배열 용량의 2배)



Step2: 기존의 배열을 새로운 배열에 복사



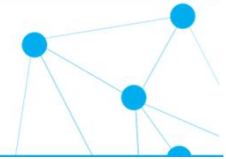
Step3: 항목을 삽입



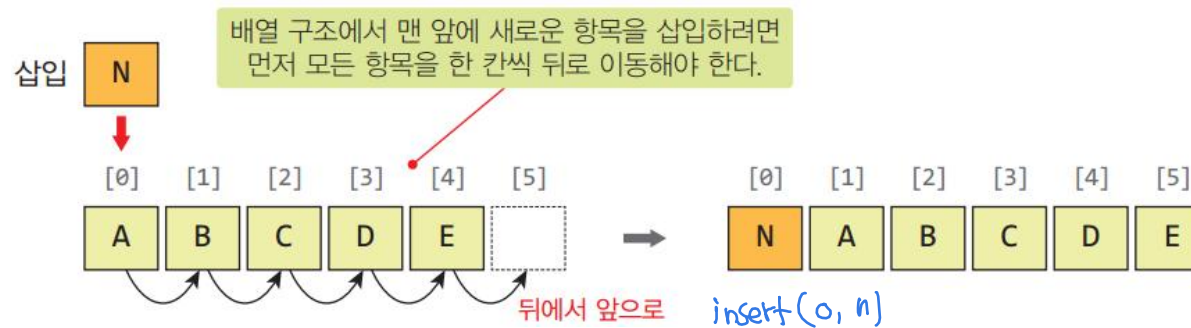
↑ 삽입!(현재 항목의 개수 증가)

Step4: 기존 배열 해제, 리스트로 새 배열 사용

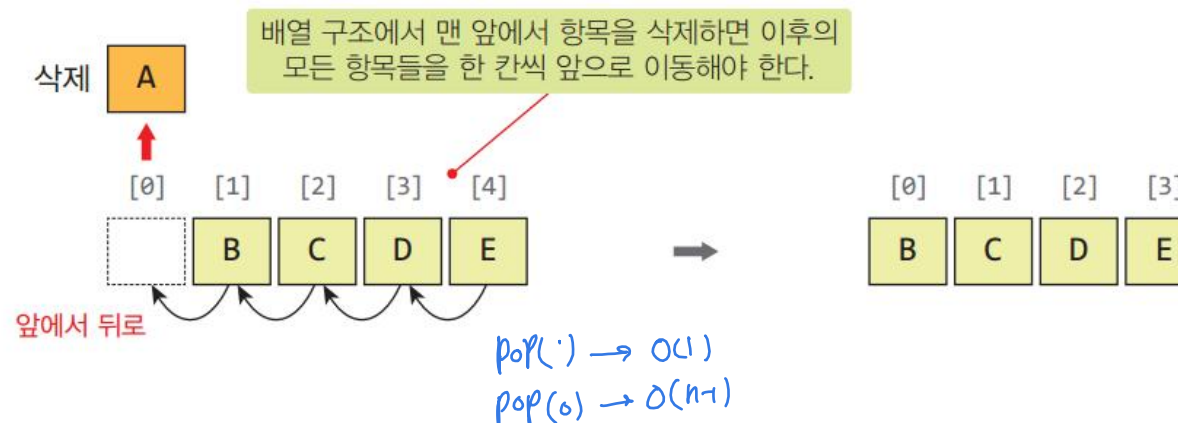
# 파이썬 리스트의 시간 복잡도



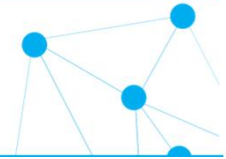
- `append(e)` 연산: 대부분의 경우  $O(1)$  동적배열 사용
- `insert(pos, e)` 연산:  $O(n)$  리스트의 원소 개수



- `pop(pos)` 연산:  $O(n)$   $pop$  연산 파라미터 (위치)를 줄 수 없음



### 3. 배열로 구현한 리스트



# 배열로 구현한 리스트(클래스 버전)

```
class ArrayList:
    # 이 책에서 클래스 코드는 살구색 바탕 이용
    def __init__( self ):
        # 생성자 (2.10절 참조)
        self.items = []
        # 클래스 변수 선언 및 초기화
```

```
def insert(self, pos, elem) :
    self.items.insert(pos, elem)  O(n)
```

```
def delete(self, pos) :
    # 위치 원소
    return self.items.pop(pos)  O(n)
```

```
def isEmpty( self ) :
    return self.size() == 0
```

```
def getEntry(self, pos) :
    # A[K] A의 시작주소 + d.K
    return self.items[pos]  # 위치 인덱스를 사용하여 값을 얻을 수 있음.
```

```
def size( self ) :
    return len(self.items)  # 원소의 개수를 반환하는 함수 len() 사용
```

```
def clear( self ) :
    self.items = []
    # items는
```

```
def find(self, item) :
    return self.items.index(item)
```

```
def replace(self, pos, elem) :
    self.items[pos] = elem
```

```
def sort(self) :
    self.items.sort()
```

```
def merge(self, lst) :
    self.items.extend(lst)
```

```
def display(self, msg='ArrayList:') :
    print(msg, '항목수=', self.size(), self.items)
```

찾으려는 원소가  
마지막에 있을 경우 다할  
다함

$O(n)$

index() 사용

원소의 위치를  
반환 해줌

$O(1)$

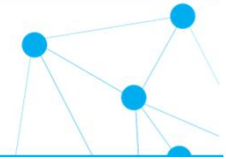
원소  
대체

배열의 크기

d.K  
원소



# 테스트 프로그램(클래스 버전)

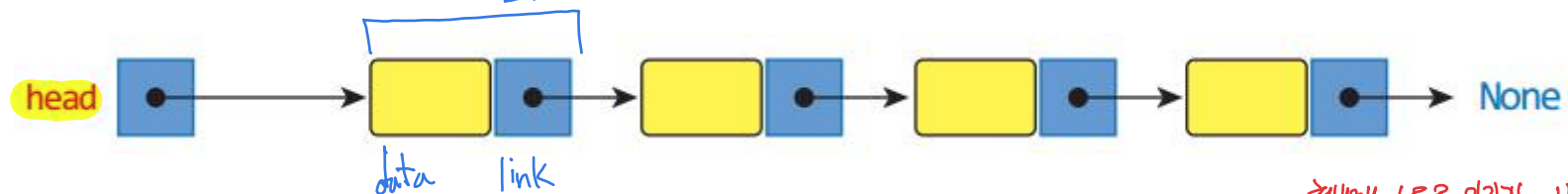


```
s = ArrayList()
s.display('파이썬 리스트로 구현한 리스트 테스트')
s.insert(0, 10);          s.insert(0, 20);          s.insert(1, 30)
s.insert(s.size(), 40);   s.insert(2, 50)
s.display("파이썬 리스트로 구현한 List(삽입x5): ")
s.sort()
s.display("파이썬 리스트로 구현한 List(정렬후): ")
s.replace(2, 90)
s.display("파이썬 리스트로 구현한 List(교체x1): ")
s.delete(2);   s.delete(s.size() - 1);   s.delete(0)
s.display("파이썬 리스트로 구현한 List(삭제x3): ")
lst = [ 1, 2, 3 ]
s.merge(lst)
s.display("파이썬 리스트로 구현한 List(병합+3): ")
s.clear()
s.display("파이썬 리스트로 구현한 List(정리후): ")
```

## 4. 연결리스트로 구현한 리스트



- 연결리스트 구조



- 노드 클래스: 연결된 스택에서와 동일
- 연결 리스트 클래스

첫번째 노드를 가리키는 변수  
head None 만 있음.  
→ 비어있음.

```
class LinkedList:
```

```
def __init__( self ):
```

```
    self.head = None
```

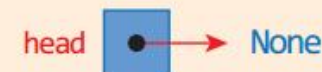
```
def isEmpty( self ): return self.head == None
```

```
def clear( self ) : self.head = None
```

```
def size( self ) : ...
```

```
def printAll(self) : ...
```

# 연결된 리스트 클래스



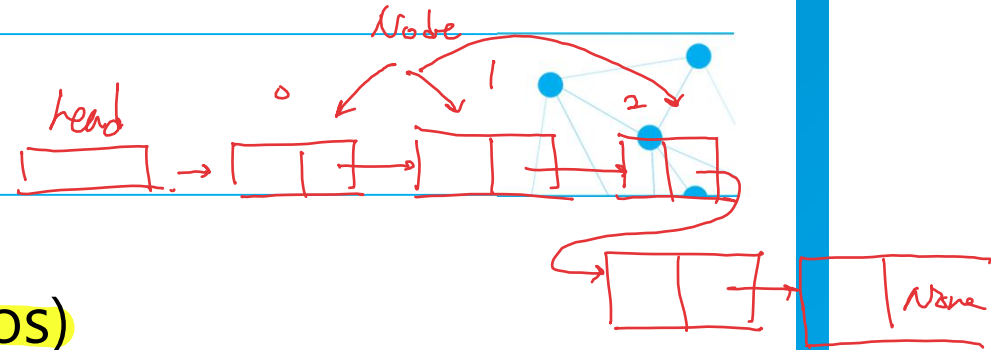
# 공백상태 검사

# 리스트 초기화

# self.top->self.head로 수정. 코드 동일



# 연결 리스트 메소드



- pos번째 노드 반환: getNode(pos)

```
def getNode(self, pos) :
```

```
    if pos < 0 : return None
```

```
    node = self.head;
```

```
    while pos > 0 and node != None :
```

```
        node = node.link
```

```
        pos -= 1
```

```
    return node
```

# pos번째 노드 반환

# node는 head부터 시작

# pos번 반복

# node를 다음 노드로 이동

# 남은 반복 횟수 줄임

# 최종 노드 반환

pos = 0  
pos = 1  
pos = 2  
⋮  
pos = n-1  
↓  
O(n)

→ 노드의 link를 따라가자.

pos 번 만큼 수행됨.

- getEntry(pos), replace(pos,elem), find(val)

$O(n)$

$O(n)$

$O(n)$

```
def getEntry(self, pos) :
```

```
    node = self.getNode(pos)
```

```
    if node == None : return None
```

```
    else : return node.data
```

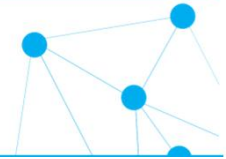
# pos번째 노드의 데이터 반환

# pos번째 노드

# 찾는 노드가 없는 경우

# 그 노드의 데이터 필드 반환

# 연결 리스트 메소드



- 노드 개수 반환: size()

def size(self):

count = 0

current = self.head

while current is not None:

count += 1

current = current.link

return count

개수를 세기 위해서는 node를 찾음.



# recursive version

def sizeRecur(self, node):

if node == None

return 0

else:

return self.sizeRecur(node.link)+1

def size(self):

return self.sizeRecur(self.head):

node 불러오기

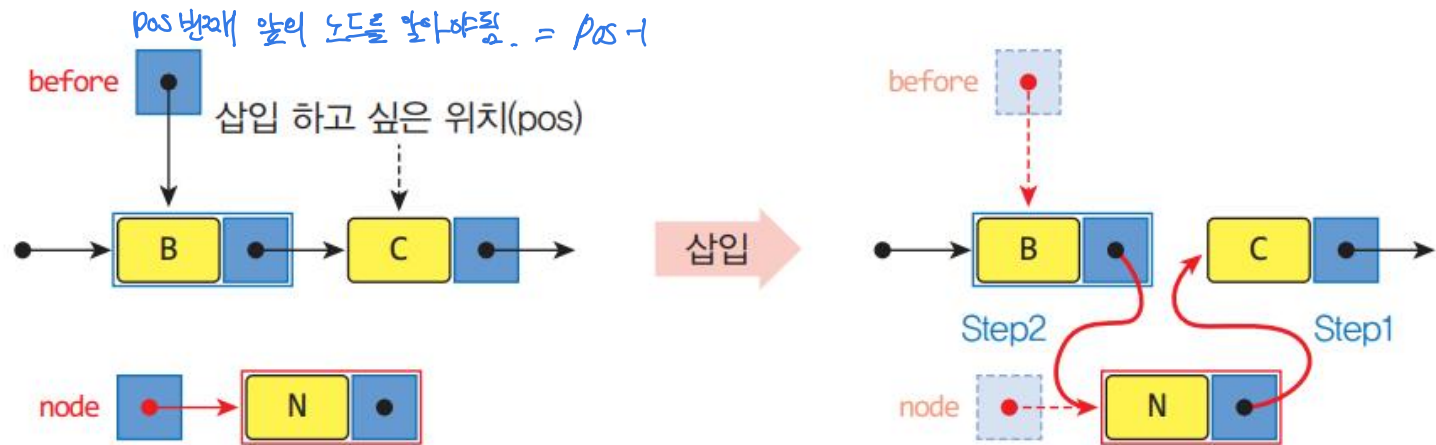
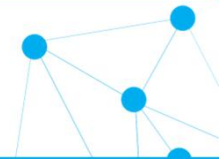
def printAll (self):

node = self.head

while node != None:

print (node.data)  
node = node.link

# 삽입 연산: insert(pos, elem)



① 노드 N이 노드 C를 가리키게 함: `node.link = before.link`

② 노드 B가 노드 N을 가리키게 함: `before.link = node`

```
def insert(self, pos, elem):
```

```
    node = Node(elem)
```

```
    before = self.getNode(pos-1) #before 노드 찾을
```

```
    if before == None:           # 맨 앞에 삽입
```

```
        node.link = self.head
```

```
        self.head = node
```

```
    else:                        # 중간에 삽입
```

```
        node.link = before.link
```

```
        before.link = node
```

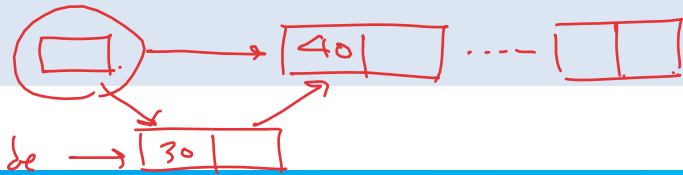
$(10, 30, 50, 20)$

↑  
insert(1, 40)

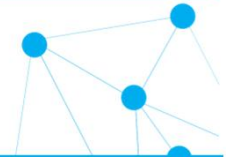
$(10, 40, 30, 50, 20)$

예외)

insert(0, 30)



# 삭제 연산: delete(pos)



① before의 link가 삭제할 노드의 다음 노드를 가리키도록 함 : before.link = before.link.link

link.link

$O(n)$  getNode와 연결되어 있음

```
def delete(self, pos) :
```

```
    before = self.getNode(pos-1)
```

```
    if before == None :
```

```
        if self.head is not None :
```

```
            self.head = self.head.link
```

```
    elif before.link != None :
```

```
        before.link = before.link.link
```

# before 노드를 찾음

# 시작노드를 삭제

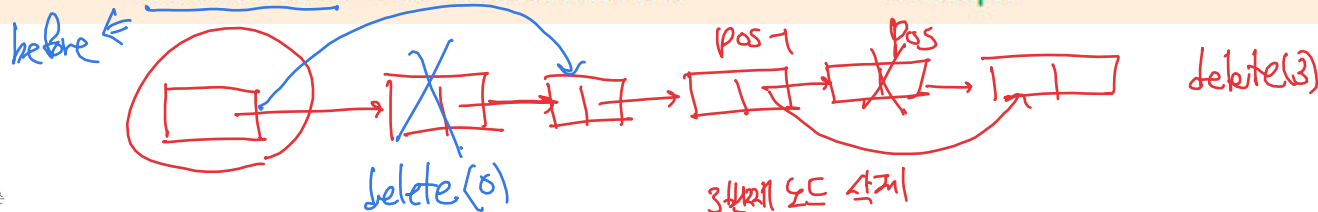
# 공백이 아니면

# head를 다음으로 이동

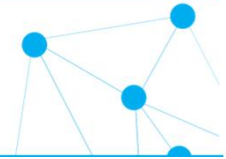
# 중간에 있는 노드 삭제

# Step1

head 값을 바꿔주면 연쇄 delete할 시에  
오류 발생 x.



## 출력 : printAll(self) (ADT의 display)



```
def printAll(self):
    node = self.head
    while node is not None:
        print(node.data)
        node = node.link

-----

# recursive version
def printRecur(self, node):
    if node is not None:
        print(node.data)
        self.printRecur(node.link)

def printAll(self):
    self.printRecur(self.head)
```