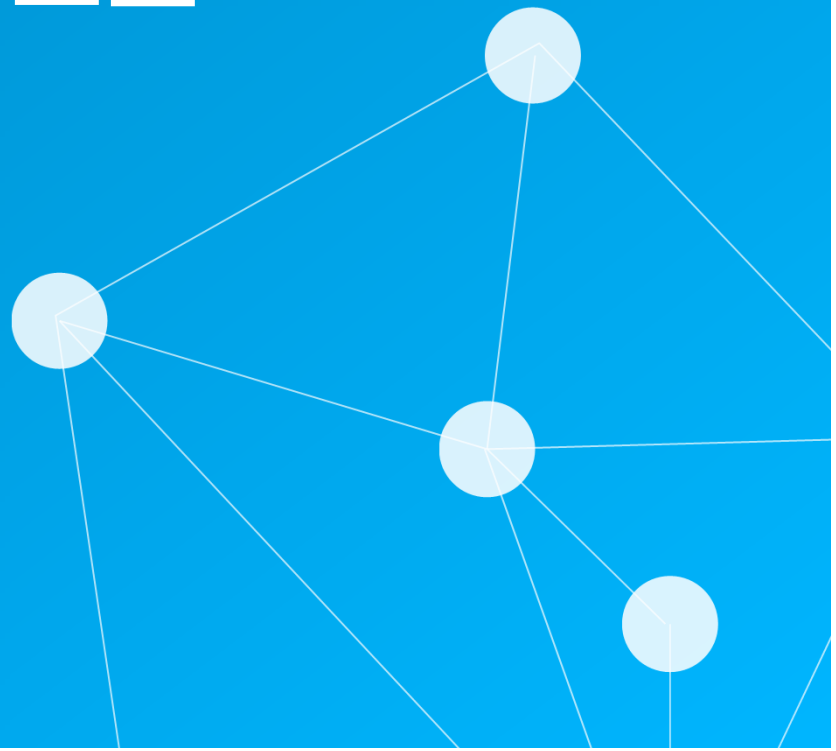

파이썬
자료구조

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?

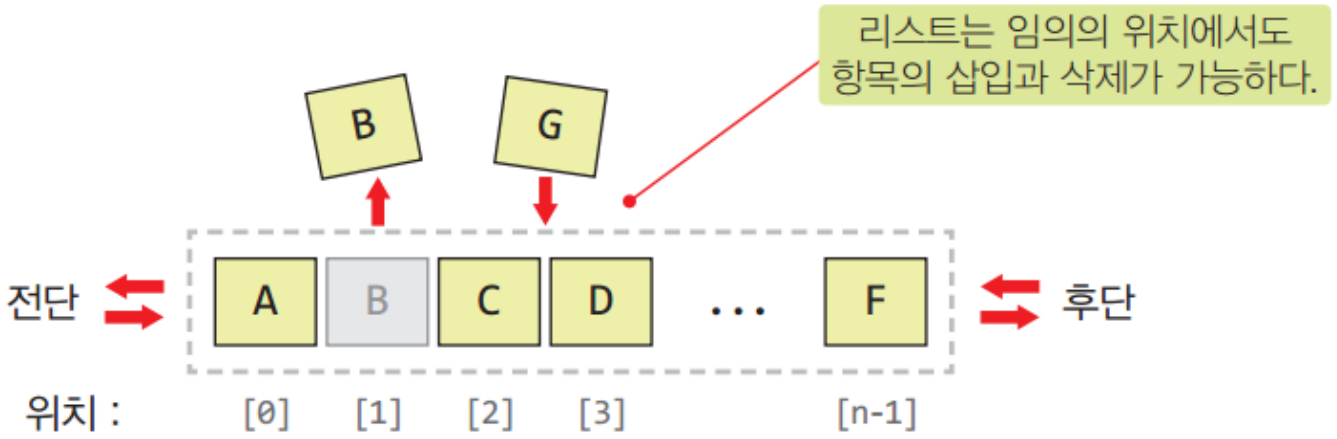
- 리스트(list), 선형리스트(linear list)
  - 순서를 가진 항목들의 모임
  - $L = [item_0,\ item_1,\ item_2,\ \ldots,\ item_{n-1}]$

    0번째 원소 1번째원소 n-1번째 원소

  - 집합: 항목간의 순서의 개념이 없음

리스트의 구조

- 리스트
  - 항목들이 순서대로 나열되어 있고, 각 항목들 은 위치를 갖는다.



- Stack, Queue, Deque과의 차이점
  - 자료의 접근 위치

리스트 ADT

정의 3.1 List ADT

데이터: 같은 유형의 요소들의 순서 있는 모임

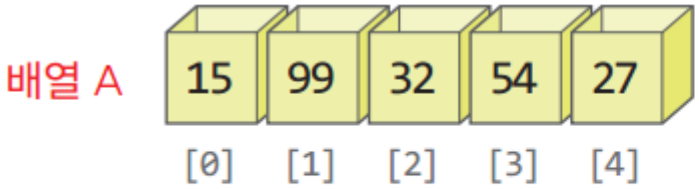
연산

- List(): 비어 있는 새로운 리스트를 만든다.
- insert(pos, e): pos 위치에 새로운 요소 e를 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. → 자주 사용하고 싶음
- clear(): 리스트를 초기화한다.
- find(item): 리스트에서 item이 있는지 찾아 인덱스를 반환한다.
- replace(pos, item): pos에 있는 항목을 item으로 바꾼다.
- sort(): 리스트의 항목들을 어떤 기준으로 정렬한다.
- merge(lst): 다른 리스트 lst를 리스트에 추가한다.
- display(): 리스트를 화면에 출력한다.
- append(e): 리스트의 맨 뒤에 새로운 항목을 추가한다.

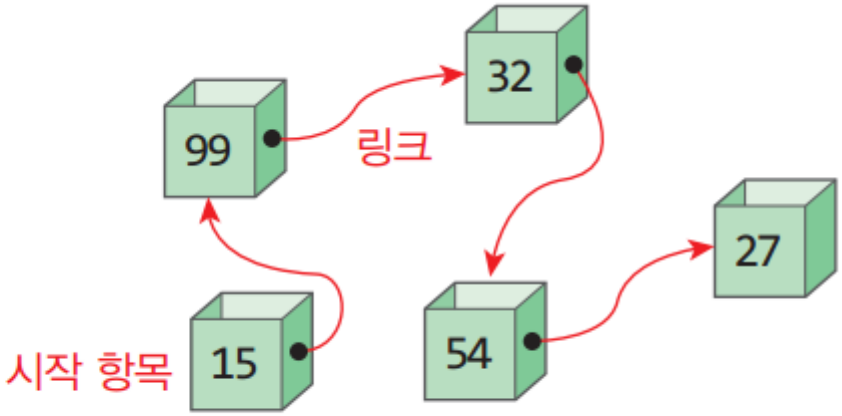
리스트 구현 방법

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

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



배열 구조의 리스트



연결된 구조의 리스트

리스트 용어 정리

파이썬 리스트	C언어에서의 배열이 진화된 형태의 스마트한 배열이다. 이 책에서는 배열 또는 배열 구조의 의미로 사용한다. 어떤 자료구조를 구현하기 위한 하나의 방법으로 사용한다.
연결 리스트	자료들이 일렬로 나열할 수 있는 연결된 구조를 말한다. 배열 구조(파이썬의 리스트)에 대응되는 의미로 사용한다.
자료구조 리스트	추상적인 의미의 자료구조 리스트를 의미한다. 앞에서 우리는 이 리스트의 ADT를 정의하였다. 이를 구현하기 위해 배열 구조(파이썬의 리스트)나 연결된 구조(연결 리스트)를 사용할 것이다.

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

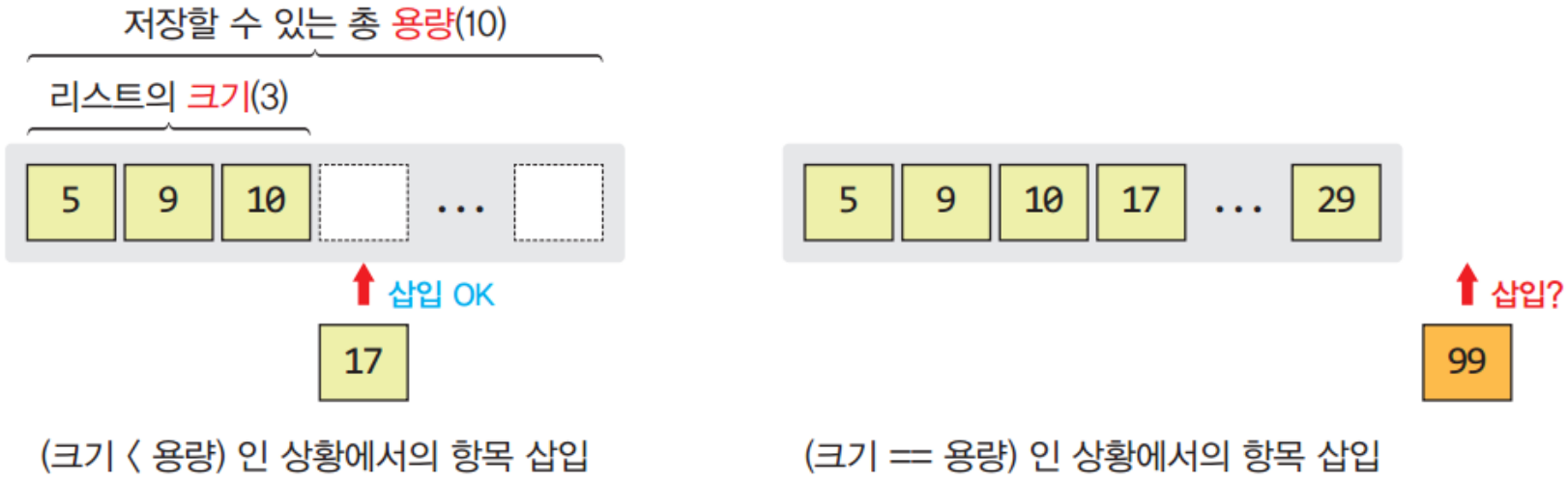
```
print('파이썬 리스트 A의 크기는 ', len(A))        # A의 크기(항목 수) 출력
```

- 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!



- 남은 공간이 없으면 어떻게 삽입할까?

동적 배열 구조에서의 용량 증가 과정

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

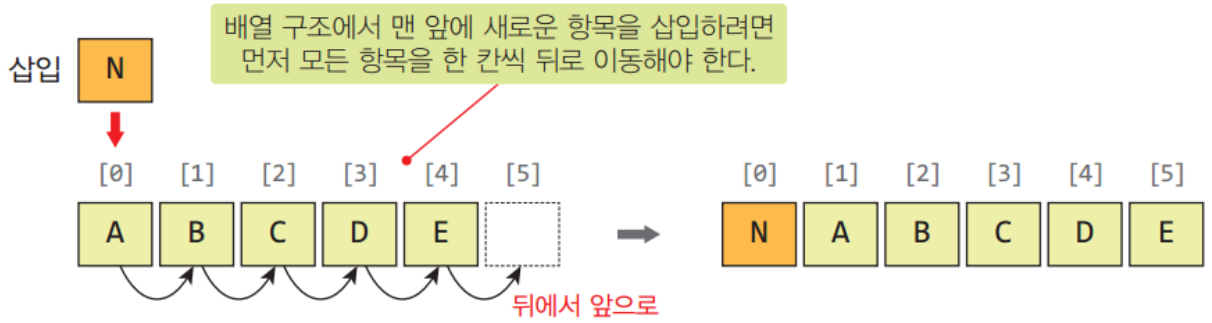
Step2: 기존의 배열을 새로운 배열에 복사

Step3: 항목을 삽입

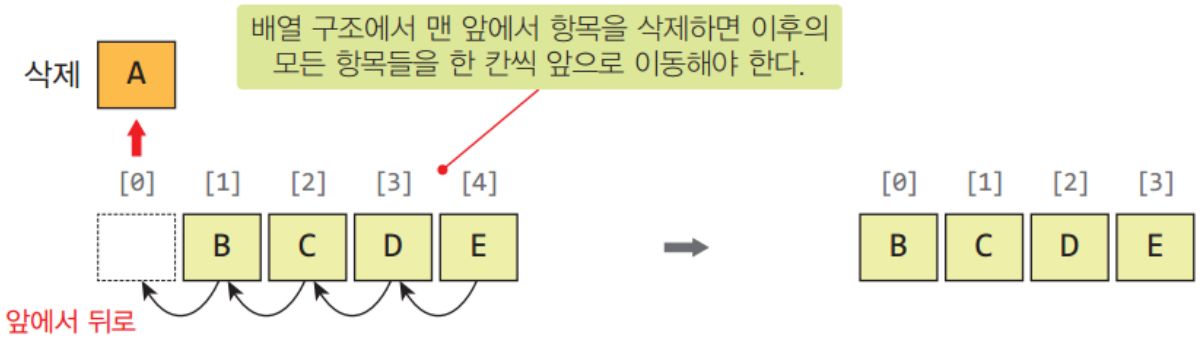
Step4: 기존 배열 해제, 리스트로 새 배열 사용

파이썬 리스트의 시간 복잡도

- append(e)연산: 대부분의 경우 $O(1)$
- insert(pos, e)연산: $O(n)$



- pop(pos)연산: $O(n)$



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)
    def delete(self, pos) :
        return self.items.pop(pos)
    def isEmpty( self ):
        return self.size() == 0
    def getEntry(self, pos) :
        return self.items[pos]
    def size( self ):
        return len(self.items)
    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)
```

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

```
class LinkedList:                                   # 연결된 리스트 클래스
    def __init__( self ):
        self.head = None                            head → None

    def isEmpty( self ): return self.head == None   # 공백상태 검사
    def clear( self ) : self.head = None            # 리스트 초기화
    def size( self ) : ...                          # self.top->self.head로 수정. 코드 동일
    def printList(self) : ...
```

연결 리스트 메소드

- pos번째 노드 반환: getNode(pos)

```
def getNode(self, pos) :                # pos번째 노드 반환
    if pos < 0 : return None
    node = self.head;                   # node는 head부터 시작
    while pos > 0 and node != None :    # pos번 반복
        node = node.link                # node를 다음 노드로 이동
        pos -= 1                        # 남은 반복 횟수 줄임
    return node                         # 최종 노드 반환
```

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

```
def getEntry(self, pos) :               # pos번째 노드의 데이터 반환
    node = self.getNode(pos)            # pos번째 노드
    if node == None : return None       # 찾는 노드가 없는 경우
    else : return node.data             # 그 노드의 데이터 필드 반환
```

연결 리스트 메소드

노드 개수 반환: size()

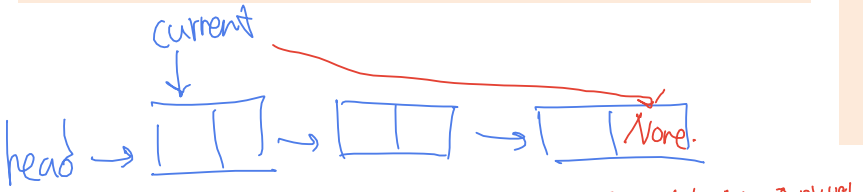
size 함수 구현

```
# iterative version (반복을 이용)
    def size(self):
        count = 0
        current  = self.head
        while current is not None:
            count += 1
            current = current.link
        return count
```

```
# recursive version
    def nodeCount(self, node):
        if node == None:
            return 0
        else:
            return self.nodeCount(node.link)+1

    def size(self):
        return self.nodeCount(self.head)
```

node가 참조하는 (가르키는) 연결리스트의 길이
= node.link 가 참조하는 연결리스트의 길이 + 1 if node != None.



current가 None을 만나면 return.

연결 리스트 메소드

노드 개수 반환: size()

- 다른 방법 : 길이를 저장하는 객체 변수 length를 둔다.

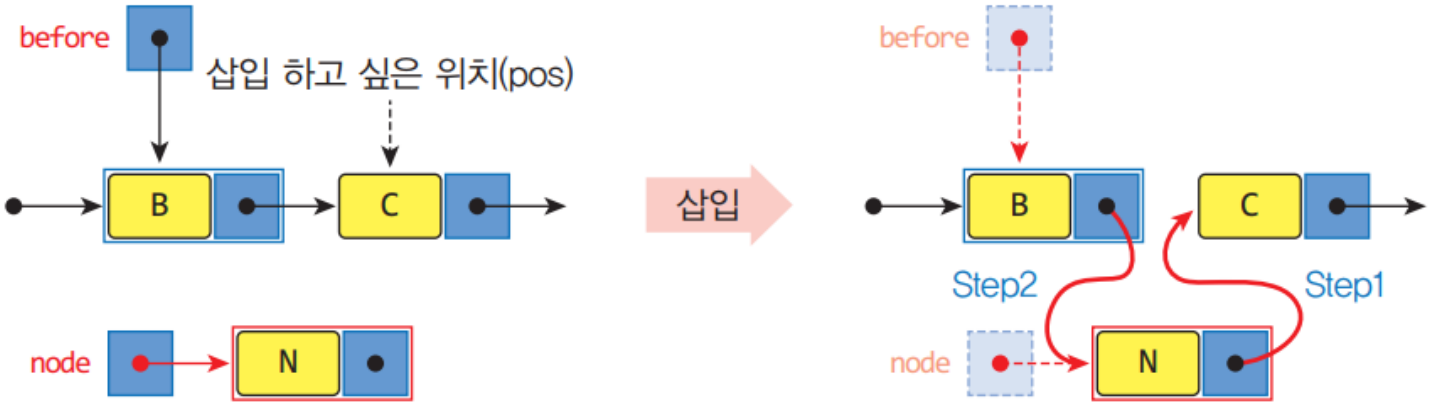
```
def __init__(self):
    self.head = None
    self.length = 0

def size(self):
    return length

def insert(self, pos, elem):
    length += 1
    ....

def delete(pos):
    length -= 1
```

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 찾음
    if before == None:              # 맨 앞에 삽입
        node.link = self.head
        self.head = node
    else:                           # 중간에 삽입
        node.link = before.link
        before.link = node
```

삭제 연산: delete(pos)

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

```
def delete(self, pos) :
    before = self.getNode(pos-1)                # before 노드를 찾음
    if before == None :                         # 시작노드를 삭제
        if self.head is not None :              # 공백이 아니면
            self.head = self.head.link          # head를 다음으로 이동
    elif before.link != None :                  # 중간에 있는 노드 삭제
        before.link = before.link.link          # Step1
```

출력 : printList(self) (ADT의 display)

```
# iterative version (반복 이용)
    def printList(self):
        node = self.head
        while node != None:
            print(node.data)
            node = node.link
```

```
# recursive version (재귀 이용)
    def printAll(self, node):
        if node is not None:
            print(node.data)
            self. printAll(node.link)

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