



부산대학교

PUSAN NATIONAL UNIVERSITY

Data Structure Interfaces

Interfaces

Representation, Operations, and Client Code

DS3600082-140

자료구조

2026학년도 2학기

오용경

데이터사이언스전공

yongkyung.oh@pusan.ac.kr

Interface and Specification

인터페이스와 명세가 정하는 내용을 구분한다

인터페이스가 정하는 것

정의

인터페이스(interface)

자료구조가 제공하는 연산과 각 연산의 의미를 정한 것이며, 추상 자료형(ADT)이라고도 부른다. 그 의미를 입력·반환값·적용 조건·상태 변화로 정확히 적은 것이 명세다

항목	공개한 연산과 그 명세의 예
이름	<code>get(i)</code>
입력	길이 n 인 리스트에서 $0 \leq i < n$
결과	위치 i 의 값을 반환하고 리스트는 바꾸지 않음

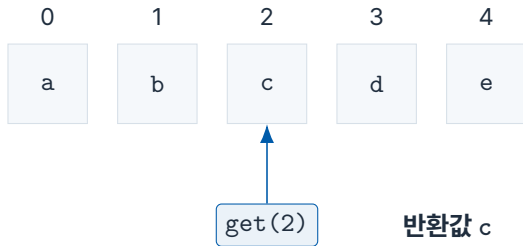
리스트 인터페이스

인덱스로 원소 위치를 지정하는 선형 나열

연산	입력	관찰할 수 있는 결과
<code>get(i)</code>	인덱스 i	위치 i 의 값을 반환
<code>set(i, x)</code>	인덱스와 값	기존 값을 반환하고 x 로 변경
<code>add(i, x)</code>	인덱스와 값	위치 i 에 x 를 추가
<code>remove(i)</code>	인덱스 i	위치 i 의 값을 삭제하고 반환

인덱스 범위 조회·교체·삭제는 $0 \leq i < n$, 삽입은 맨 뒤를 포함해 $0 \leq i \leq n$

get(i)의 의미



인덱스는 값이 아니라 나열 안의 위치를 가리킨다

set(i, x)의 의미

변경 전



set(2, x)

변경 후



길이

5로 유지

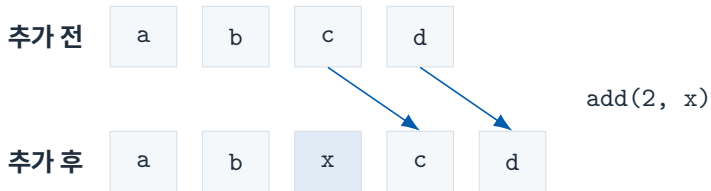
값

c가 x로 변경

반환

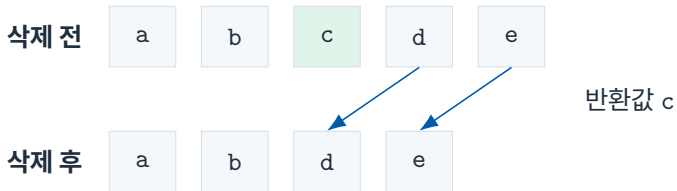
기존 값 c

add(i, x)의 의미



인터페이스는 인덱스의 변화를 정하지만 원소를 옮기는 방법은 정하지 않는다

remove(i)의 의미



remove(2)는 c를 돌려주고 뒤쪽 원소의 인덱스가 하나씩 줄어든다

클라이언트 코드에 공개되는 연산

클라이언트 코드(client code)는 자료구조를 호출하는 코드이며, 공개된 연산 결과에만 의존해야 한다

```
1 def swap_first_two(seq):  
2     first = seq.get(0)  
3     second = seq.get(1)  
4     seq.set(0, second)  
5     seq.set(1, first)
```

코드	하는 일
seq get(0), get(1) set(0, second), set(1, first)	get·set을 가진 리스트 구현(Python list 아님) 앞의 두 값을 먼저 저장 두 자리의 값을 서로 바꿔 저장

클라이언트 코드가 공개된 연산에만 의존하면 내부 저장 방식을 바꿀 수 있다

연산마다 허용되는 인덱스가 다르다

적용 조건

인덱스 범위

명세가 정하는 적용 조건의 하나다.

원소가 n 개일 때 조회·교체·삭제와 삽입의 인덱스 범위는 다르다.

연산	허용 범위	$n = 3$ 인 예
get, set, remove	$0 \leq i < n$	0, 1, 2
add	$0 \leq i \leq n$	0, 1, 2, 3

add(3, x)는 끝에 추가할 수 있지만 get(3)은 허용되지 않는다

Source: Morin (2013), Section 1.2.2.

Checkpoint

Interface and Specification

- `add(1, x)` 뒤에 기존 인덱스 1의 값은 어느 인덱스에 있는가
- `remove(2)`가 반환해야 하는 값은 무엇인가
- 클라이언트 코드가 내부 저장 방식에 직접 접근하면 무엇에 의존하게 되는가?

Representation and Work

배열 표현에서 각 연산이 읽고 옮기는 작업을 추적한다

정의

내부 표현(representation)

값과 관계를 저장하는 방식이다.

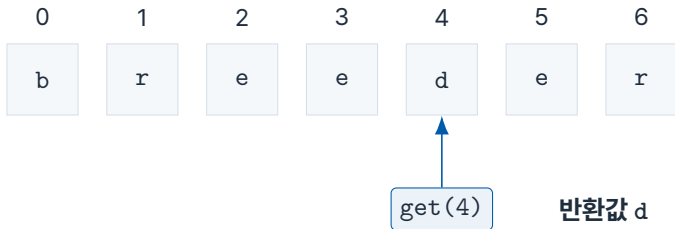
구현(implementation)은 내부 표현과 그 표현으로 연산을 실행하는 코드다.

구분	배열로 리스트를 구현할 때 묻는 질문
인터페이스	어떤 연산을 공개하는가
명세	호출 뒤 어떤 값을 반환하고 어떤 상태가 되어야 하는가
내부 표현	값을 배열의 어느 칸에 저장하는가
내부 작업	연산마다 무엇을 읽고 옮기는가
비용	읽거나 옮기는 작업이 입력 크기에 따라 얼마나 늘어나는가

Source: Morin (2013), Section 1.2.

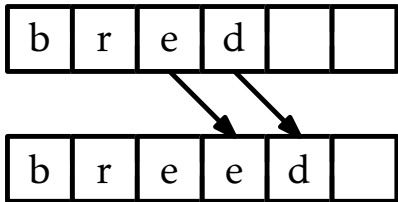
배열에서 `get(i)`

시작 위치와 인덱스를 알면 한 칸을 바로 선택



배열 삽입에서 이동하는 칸

빈 칸을 만들려면 삽입 위치부터 맨 뒤까지의 원소를 맨 뒤부터 한 칸씩 오른쪽으로 옮긴다

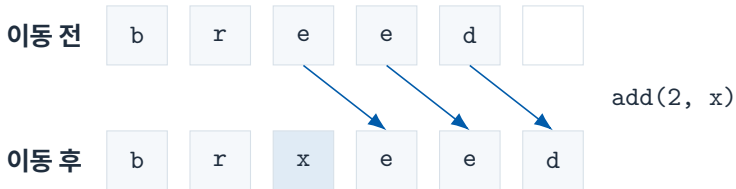


$\text{add}(2, e)$

아랫행 인덱스 2의 e가 새로 넣은 값이고, 화살표는 원래 e, d가 한 칸씩 오른쪽으로 복사된 것

배열에서 `add(i, x)`

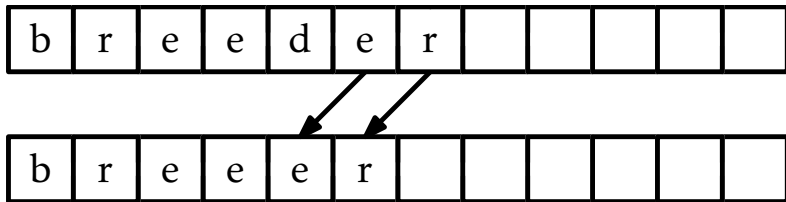
맨 뒤 원소부터 한 칸씩 오른쪽으로 옮겨 인덱스 `i`를 비운 뒤 `x`를 넣는다



`add(2, x)`는 원소 세 개를 옮긴다

배열 삭제에서 이동하는 칸

삭제 뒤 생긴 빈 칸은 뒤쪽 원소를 한 칸씩 왼쪽으로 옮겨 메운다

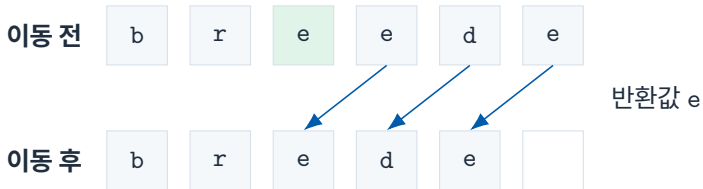


remove(4)

삽입과 삭제는 이동 방향이 반대이며, 이동하는 원소 수는 위치에 따라 달라진다

배열에서 `remove(i)`

삭제한 칸을 뒤쪽 원소가 차례로 덮어쓴다



`remove(2)`는 원소 세 개를 왼쪽으로 옮긴다

삽입할 때는 뒤에서부터 옮긴다

아직 복사하지 않은 값을 덮어쓰지 않도록 빈칸 쪽부터 이동한다

단계(j = 받는 칸)	실행문	a (None은 _)
시작	<code>add(1, "x"), n = 4</code>	p q r s _
$j = 4$	<code>a[4] = a[3]</code>	p q r s s
$j = 3$	<code>a[3] = a[2]</code>	p q r r s
$j = 2$	<code>a[2] = a[1]</code>	p q q r s
삽입	<code>a[1] = "x", n = 5</code>	p x q r s

앞에서부터 옮기면 원래 값을 덮어쓴다

앞선 대입이 아직 복사하지 않은 값을 바꾸므로 r과 s가 사라진다

단계(j = 받는 칸)	실행문	a (None은 _)
시작	<code>add(1, "x"), n = 4</code>	p q r s _
$j = 2$	<code>a[2] = a[1]</code>	p q q s _
$j = 3$	<code>a[3] = a[2]</code>	p q q q _
$j = 4$	<code>a[4] = a[3]</code>	p q q q q
삽입	<code>a[1] = "x"</code>	p x q q q

Caution r과 s를 복사하기 전에 덮어썼기 때문에 원래 값을 잃었다

삭제할 값은 이동 전에 보관한다

단계(j = 받는 칸)	실행문	상태와 반환값 (None은 _)
시작	<code>remove(1), n = 4</code>	<code>p q r s _</code>
보관	<code>removed = a[1]</code>	<code>removed = q, p q r s _</code>
$j = 1$	<code>a[1] = a[2]</code>	<code>p r r s _</code>
$j = 2$	<code>a[2] = a[3]</code>	<code>p r s s _</code>
정리	<code>n = 3, a[3] = None</code>	<code>p r s _ _</code>

반환값 `q`를 먼저 보관하고 뒤 원소를 한 칸씩 당기면 `[p, r, s]`가 남는다

자료구조는 연산의 의미를 내부 작업으로 실현한다

정의

자료구조(data structure)

ADT의 연산을 실행할 수 있도록 값을 저장하고 처리하는 구체적인 구현이다.
같은 ADT라도 저장 방식과 처리 코드는 달라질 수 있다.

구분	답하는 질문
ADT	어떤 연산을 제공하고 무슨 결과를 내는가
자료구조	값을 어떻게 저장하고 연산을 수행하는가
내부 작업	무엇을 읽고 옮기는가
비용	그 작업이 입력 크기에 따라 얼마나 늘어나는가

Source: Morin (2013), Section 1.2.

Checkpoint

Representation and Work

- 배열에서 `get(i)`는 어느 칸을 선택하는가?
- 배열 중간에 값을 삽입하면 어떤 내부 작업이 필요한가?
- 내부 표현을 바꾸어도 유지해야 하는 연산의 결과는 무엇인가?

Python Objects

공유된 객체의 상태 변화가 두 변수 이름에 어떻게 보이는지 추적한다

변수 이름과 객체

변수 이름(name)은 객체(object)를 가리키며 대입은 참조(reference)가 가리키는 대상을 정한다



변수 이름

코드에서 사용하는 식별자

참조

객체를 가리키는 연결

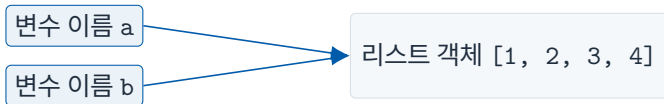
객체

값과 상태가 저장된 대상

두 변수 이름이 같은 객체를 가리킬 때

b = a는 새 리스트 객체를 만들지 않는다

```
1 a = [1, 2, 3]
2 b = a
3 b.append(4)
4
5 print(a)
6 # [1, 2, 3, 4]
```



별칭은 둘 이상의 변수 이름이 같은 객체를 가리키는 상태다

객체 변경과 변수 재대입

두 행은 모두 $a = [1, 2]$, $b = a$ 인 같은 시작 상태에서 따로 실행한다

구분	실행	변수 이름 a가 가리키는 값	변수 이름 b가 가리키는 값
객체 변경	<code>b.append(3)</code>	[1, 2, 3]	[1, 2, 3]
재대입	<code>b = [7, 8]</code>	[1, 2]	[7, 8]

`b.append(3)`는 같은 리스트 객체의 상태를 바꾸고, `b = [7, 8]`은 변수 이름 b가 새 객체를 가리키게 한다

같은 객체인지와 같은 값을 가지는지를 구분

a = [1, 2], b = a, c = [1, 2]

식	결과	확인하는 것
a is b	True	같은 객체를 가리키는가
a is c	False	같은 객체를 가리키는가
a == c	True	두 객체의 값이 같은가

리스트처럼 원소 값을 비교하는 객체에서는, 값이 같은 서로 다른 객체의 ==가 True, is가 False다

클래스와 self

self는 메서드가 호출된 인스턴스 객체를 가리킨다

```
1 class Box:
2     def __init__(self, value):
3         self.value = value
4
5 box = Box(7)
6 print(box.value)
```

변수 이름

box가 인스턴스 객체를 참조

인스턴스 객체

Box의 인스턴스, value = 7

메서드의 첫 매개변수

__init__ 실행 중 self도
같은 객체를 참조

같은 클래스에서 만든 객체는 상태를 따로 가진다

같은 메서드를 사용해도 `self`가 가리키는 객체에 따라 읽고 쓰는 상태가 달라진다

```
1 a = Box(7)
2 b = Box(12)
3 a.value = 9
4 print(a.value, b.value)    # 9 12
5 print(a is b)             # False
```

첫째 인스턴스

a가 가리키는 Box 객체, value = 9

둘째 인스턴스

b가 가리키는 Box 객체, value = 12

a.value를 바꾸면 a가 가리키는 객체만 바뀌고 b의 상태는 그대로다

대입과 새 리스트 만들기

`list(a)`는 새 리스트 객체를 만들지만 `b = a`는 기존 객체를 가리킨다

```
1 a = [1, 2]
2 b = list(a)
3 b.append(3)
4 print(a)      # [1, 2]
5 print(b)      # [1, 2, 3]
6 print(a is b) # False
```

원래 리스트

a가 가리키는 [1, 2]

새 리스트

b가 가리키는 [1, 2, 3]

Checkpoint

Python Objects

- `b = a` 뒤에 `b.append(x)`를 실행하면 `a`가 가리키는 객체는 어떻게 되는가
- `b = [x]`는 기존 객체와 변수 이름 `b` 중 무엇을 바꾸는가
- 객체 변경과 변수 재대입은 무엇이 다른가?

Two Implementations

같은 스택 인터페이스를 두 리스트 배치로 구현하고 결과를 비교

리스트의 뒤쪽을 맨 위로 사용

스택(stack)의 push와 pop을 같은 리스트 끝에 대응

```
1 class EndStack:
2     def __init__(self):
3         self._items = []
4
5     def push(self, item):
6         self._items.append(item)
7
8     def pop(self):
9         return self._items.pop()
```

스택 인터페이스

push(item), pop()

메서드 대응

append(item), pop()

내부 리스트

push(4), push("dog") 뒤
[4, "dog"], 맨 위는 오른쪽

Adapted from Miller and Ranum (2013), Implementing a Stack in Python.

리스트의 앞쪽을 맨 위로 사용

같은 스택 연산을 다른 리스트 메서드에 대응

```
1 class FrontStack:
2     def __init__(self):
3         self._items = []
4
5     def push(self, item):
6         self._items.insert(0, item)
7
8     def pop(self):
9         return self._items.pop(0)
```

스택 인터페이스

push(item), pop()

메서드 대응

insert(0,item), pop(0)

내부 리스트

같은 호출 뒤 ["dog", 4],
맨 위는 왼쪽

Adapted from Miller and Ranum (2013), Implementing a Stack in Python.

내부 순서는 달라도 삭제 순서는 같다

호출	EndStack 내부 리스트	FrontStack 내부 리스트
push(4)	[4]	[4]
push("dog")	[4, "dog"]	["dog", 4]
pop()	[4], 반환값 "dog"	[4], 반환값 "dog"
pop()	[], 반환값 4	[], 반환값 4

내부 작업 insert(0, item)은 앞 절의 add(0, x)처럼 나머지 원소를 한 칸씩 옮기는 작업이 따른다

내부 리스트를 그대로 비교하면 다르지만 스택으로 관찰한 결과는 같다

클라이언트 코드는 관찰 가능한 결과를 비교한다

구현을 바꾸어도 모든 호출 열에서 명세와 같은 결과를 내야 하며, `assert`는 그중 한 호출 열을 확인한다

```
1 def observe(stack):  
2     stack.push(4)  
3     stack.push("dog")  
4     return stack.pop(), stack.pop()  
5  
6 assert observe(EndStack()) == ("dog", 4)  
7 assert observe(FrontStack()) == ("dog", 4)
```

클라이언트 코드는 내부 `list`의 방향이 아니라 `push`와 `pop`의 결과에 의존한다

Checkpoint

Two Implementations

- 두 스택 클래스가 공통으로 제공하는 연산은 무엇인가?
- 내부 리스트의 순서가 달라도 같은 스택이라고 말할 수 있는 이유는 무엇인가?
- 클라이언트 코드가 내부 `list` 메서드를 직접 호출하면 어떤 문제가 생기는가?

How Do We Compare the Cost of Two Implementations?

Next Lecture: Measuring Running Time

- 입력 크기에 따른 실제 실행 시간 측정
- 코드가 수행하는 연산 수 세기
- 측정값의 변동과 해석 조건