



부산대학교

PUSAN NATIONAL UNIVERSITY

Arrays and Indexed Access

Storing Elements in a Row

Contiguous Storage, Index, and Shifting Cost

DS3600082-140

자료구조

2026학년도 2학기

오용경

데이터사이언스전공

yongkyung.oh@pusan.ac.kr

Contiguous Storage

원소를 연속해서 저장하면 주소 계산으로 바로 찾는다

원소를 연속된 칸에 저장한다

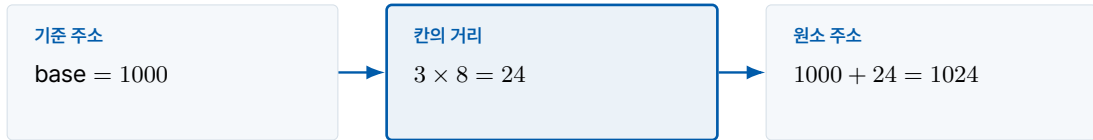
배열(array)은 같은 크기의 칸을 메모리에 연속해서 배치한다



용어	뜻
배열	같은 종류의 값을 정해진 개수만큼 담는 저장소
메모리 배치(memory layout)	원소가 메모리에 놓이는 방식. 연속 저장은 그 한 가지
연속 저장(contiguous storage)	칸이 메모리에서 물리적으로 이어져 있음
인덱스(index)	몇 번째 칸인지 가리키는 정수. 0부터 센다

인덱스로 원소 주소를 계산한다

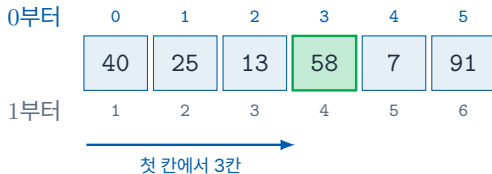
$\text{addr}(i) = \text{base} + i \times \text{size}$ 로 어느 칸이든 계산 한 번에 찾는다



$i = 0$ 의 주소는 1000이고, $i = 5$ 의 주소는 $1000 + 5 \times 8 = 1040$ 이다.

원소 수와 관계없이 곱셈 한 번과 덧셈 한 번으로 주소를 구한다

인덱스는 첫 칸에서 떨어진 칸 수다



0부터: 주소 = $\text{base} + 3 \times \text{size}$

번호가 곧 첫 칸에서 떨어진 칸 수

1부터: 주소 = $\text{base} + (4 - 1) \times \text{size}$

두 체계 모두 계산할 수 있으며 Python list와 List 인터페이스는 0부터 센다

유효한 인덱스의 범위

```
1 a = ["b", "r", "e", "d"]
2 print(a[3])      # d
3 print(a[-1])     # d
4 print(a[4])      # IndexError
```

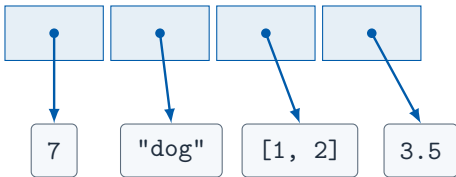
i	List 인터페이스	Python list
$0 \leq i < n$	유효한 위치	값을 반환
$i \geq n$	허용하지 않음	IndexError
$-n \leq i < 0$	허용하지 않음	끝에서부터 센 값을 반환
$i < -n$	허용하지 않음	IndexError

Caution 배열의 주소 계산과 공개된 List 인터페이스의 경계 검사는 서로 다른 일이다

Python list의 칸에는 참조가 들어간다

칸의 크기가 같아야 주소 계산이 성립하므로 객체(object) 대신 참조(reference)를 담는다

```
a = [7, "dog", [1, 2], 3.5]
```



칸: 참조 하나, 크기가 모두 같음

객체: 종류마다 크기가 다름

서로 다른 자료형을 담아도 칸의 크기는 같아 인덱스 접근이 $\Theta(1)$ 이다

List 인터페이스, Python list, 배열

층	정하는 것	예
List 인터페이스	연산의 이름과 결과	<code>get(i)</code> 는 $0 \leq i < n$ 에서만 정의
Python list	자료형이 허용하는 연산	<code>a[i]</code> , <code>a.insert(i, x)</code> , <code>a[-1]</code> 도 허용
배열	값을 메모리에 놓는 방식	연속된 칸, 칸마다 참조 하나

Python list는 배열에 참조를 담아 만든 자료형이며 인덱스 규칙은 List 인터페이스와 다르다

Checkpoint

Contiguous Storage

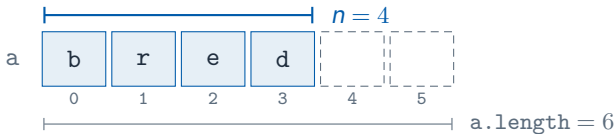
- 인덱스만으로 주소를 계산할 수 있는 조건은 무엇인가
- $\text{base} = 2000$, $\text{size} = 8$ 이면 인덱스 6의 주소는 얼마인가
- Python `list`가 서로 다른 자료형을 담아도 주소 계산이 성립하는 이유는 무엇인가

The Backing Array

ArrayStack의 원소 수와 내부 배열을 구분해 읽는다

ArrayStack의 내부 배열

ArrayStack은 내부 배열(backing array) a 와 원소 수 n 을 따로 둔다



이름	뜻	이 그림의 값
a	원소를 담는 내부 배열	b, r, e, d와 빈 칸 2개
n	크기(size): 지금 담긴 원소 수	4
$a.length$	확보한 칸 수	6

앞의 주소 공식에 쓴 size는 칸 하나의 바이트 수로, 여기의 크기 n 과 다른 값이다.

Adapted from Morin (2013), Figure 2.1.

ArrayStack은 배열을 내부 표현으로 사용한다

공개된 연산, 값을 저장하는 방식, 연산을 실행하는 코드를 구분한다

구분	역할	예
ADT	공개 연산과 그 의미	List 인터페이스
내부 표현	값을 저장하는 방식	연속된 배열
구현	표현으로 연산을 실행	ArrayStack 클래스

같은 List 인터페이스를 배열로도, 다른 표현으로도 구현할 수 있다

get은 한 칸만 읽는다

구현

```
1 class ArrayStack:
2     def get(self, i):
3         if i < 0 or i >= self.n:
4             raise IndexError
5         return self.a[i]
```

호출	반환	읽는 칸
get(2)	e	a[2]
get(4)	IndexError	없음

get은 원소 수와 관계없이 검사 뒤 한 칸을 읽는다

set은 한 칸을 바꾸고 이전 값을 돌려준다

구현

```
1 class ArrayStack:
2     def set(self, i, x):
3         if i < 0 or i >= self.n:
4             raise IndexError
5         old = self.a[i]
6         self.a[i] = x
7         return old
```

호출 전	호출과 반환	호출 뒤
[b,r,e,d,-,-]	set(2,i) → e	[b,r,i,d,-,-]

set은 지정한 한 칸만 바꾸며 배열의 길이는 그대로다

Checkpoint

The Backing Array

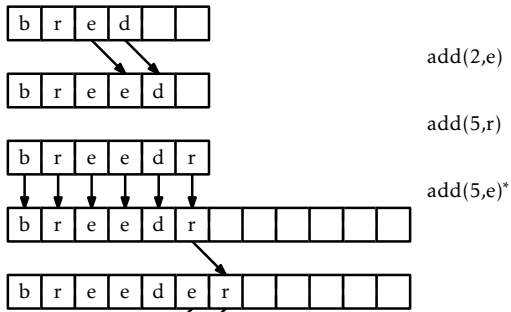
- 내부 배열 `a`와 원소 수 `n`은 각각 무엇을 나타내는가
- `a.length`가 6이고 `n = 4`일 때 `get(4)`를 허용하지 않는 이유는 무엇인가
- `set(i, x)`가 기존 값을 반환하면 사용자 코드에 어떤 정보가 생기는가

The Cost of Shifting

가운데를 건드리면 뒤쪽 전부가 움직인다

빈 칸을 만들어야 넣을 수 있다

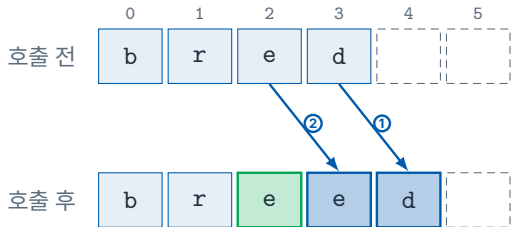
연속을 유지하려면 뒤쪽 원소를 한 칸씩 밀어야 한다



기울어진 화살표는 한 칸씩 밀리는 원소, 세로 화살표는 더 큰 새 배열로의 복사를 가리킨다.

add(2, e)는 뒤쪽 두 원소를 민다

인덱스 2부터 끝까지의 원소를 한 칸씩 오른쪽으로 옮긴 뒤 빈 칸에 넣는다



$$i = 2, n = 4$$

옮긴 원소: $n - i = 2$ 개

번호는 복사 순서

오른쪽 끝부터 복사해야 아직 옮기지 않은 값을 덮어쓰지 않는다

왼쪽부터 밀면 뒤쪽 원소를 잃는다

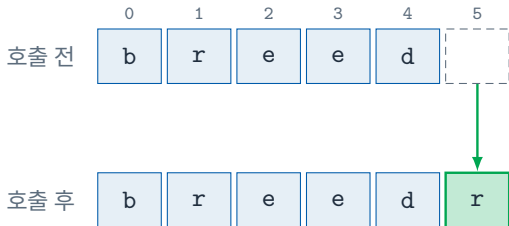
앞서 본 덮어쓰기 반례를 `[b,r,e,d]`에 `add(2, x)`를 하며 왼쪽부터 복사하는 경우로 다시 본다

한 일	배열 상태
시작	<code>[b,r,e,d,-,-]</code>
<code>a[3] = a[2]</code>	<code>[b,r,e,e,-,-]</code> : <code>d</code> 가 사라짐
<code>a[4] = a[3]</code>	<code>[b,r,e,e,e,-]</code> : <code>e</code> 를 다시 복사

삽입은 오른쪽 끝부터 밀어야 아직 복사하지 않은 `d`를 보존한다

add(5, r)는 옮길 원소가 없다

$i = n$ 이면 끝의 빈 칸에 바로 쓰고 n 만 늘린다



$$i = 5, n = 5$$

옮길 원소: $n - i = 0$ 개

빈 칸 하나에 쓰기만 한다

끝에 추가하면 원소 수가 아무리 많아도 이동이 없다

add(i, x)를 코드로 옮기면

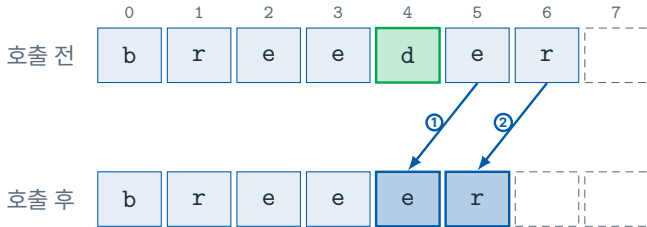
빈 칸이 하나 있다고 보고, 오른쪽 끝부터 한 칸씩 복사한 뒤 넣는다

```
1 class ArrayStack:
2     def add(self, i, x):
3         if i < 0 or i > self.n:
4             raise IndexError
5         # Assume self.n < len(self.a).
6         for j in range(self.n, i, -1):
7             self.a[j] = self.a[j - 1]
8         self.a[i] = x
9         self.n += 1
```

단계	하는 일
경계 검사	$0 \leq i \leq n$ 만 허용하고, $i = n$ 이면 끝에 붙인다
이동	$j = n, \dots, i + 1$ 순서로 오른쪽 칸에 복사해 $n - i$ 개를 옮긴다
쓰기	비운 $a[i]$ 에 x 를 쓰고 n 을 1 늘린다

remove(4)는 뒤쪽 원소를 당긴다

인덱스 4의 d를 꺼낸 뒤 그 뒤의 원소를 한 칸씩 왼쪽으로 옮긴다



`a.length = 12`, 8칸까지 표시

$i = 4, n = 7$

반환값 d

옮긴 원소: $n - i - 1 = 2$ 개

삭제는 왼쪽 원소부터 옮겨야 아직 옮기지 않은 값을 잃지 않는다

오른쪽부터 당기면 가운데 원소를 잃는다

[b,r,e,d]에서 인덱스 1을 지우면서 뒤쪽부터 복사한 경우

한 일	배열 상태
시작	[b,r,e,d,-,-]
a[2] = a[3]	[b,r,d,d,-,-]: e가 사라짐
a[1] = a[2]	[b,d,d,d,-,-]: d를 다시 복사

삭제는 왼쪽부터 당겨야 e를 옮긴 뒤 다음 원소를 읽을 수 있다

remove(i)를 코드로 옮기면

```
1 class ArrayStack:
2     def remove(self, i):
3         if i < 0 or i >= self.n:
4             raise IndexError
5         removed = self.a[i]
6         for j in range(i, self.n - 1):
7             self.a[j] = self.a[j + 1]
8         self.n -= 1
9         self.a[self.n] = None
10        return removed
```

단계	하는 일
보관	$0 \leq i < n$ 을 확인하고 덮어쓰기 전에 $a[i]$ 를 $removed$ 에 담는다
이동	$j = i, \dots, n - 2$ 순서로 왼쪽 칸에 복사해 $n - i - 1$ 개를 옮긴다
정리	n 을 1 줄이고 마지막 칸을 비운 뒤 $removed$ 를 돌려준다

각 연산이 옮긴 원소 수

연산	호출 전 n	옮긴 원소 수	계산
<code>add(2, e)</code>	4	2	$n - i$
<code>add(5, r)</code>	5	0	$n - i$
<code>add(5, e)*</code>	6	1	$n - i$
<code>remove(4)</code>	7	2	$n - i - 1$
<code>remove(4)</code>	6	1	$n - i - 1$
<code>remove(4)*</code>	5	0	$n - i - 1$
<code>set(2, i)</code>	4	0	이동 없음

별표 연산은 원소 이동과 별도로 내부 배열의 칸 수(`a.length`)를 바꾸는 작업도 한다. 이 표는 이동한 원소 수만 센다.

정리

ArrayStack의 연산 비용

내부 배열의 칸 수를 바꾸는 비용을 빼면 $\text{get}(i)$ 와 $\text{set}(i, x)$ 는 $\Theta(1)$ 이고
 $\text{add}(i, x)$ 와 $\text{remove}(i)$ 는 $\Theta(1 + n - i)$ 다

연산	옮기는 원소	비용
$\text{add}(0, x), \text{remove}(0)$	뒤쪽 거의 전부	$\Theta(n)$
$\text{add}(n, x), \text{remove}(n-1)$	없음	$\Theta(1)$

삽입은 $n - i$ 개, 삭제는 $n - i - 1$ 개를 옮긴다. +1은 원소 이동이 없어도 드는 비용이다.

삽입 위치에 따른 최선·평균·최악 비용

최선·최악·평균의 경우를 add의 삽입 위치에 따라 구분한다

경우	어떤 i 인가	비용
최선	$i = n$ (맨 끝)	$\Theta(1)$
최악	$i = 0$ (맨 앞)	$\Theta(n)$
평균	i 가 고르게 분포	$\Theta(n)$

Caution 평균 경우에도 약 $n/2$ 개를 옮기므로 최악의 경우와 같은 $\Theta(n)$ 이다

삽입 위치에 따른 이동 횟수

$n = 8$ 에서 시작해 `add`를 세 번 연속 실행한다

호출	옮기는 원소	개수	누적
<code>add(8, x)</code>	없음	0	0
<code>add(4, y)</code>	$a[4] \dots a[8]$	5	5
<code>add(0, z)</code>	전부	10	15

같은 `add`를 세 번 실행해도 삽입 위치에 따라 이동 비용은 0에서 10까지 달라진다

Checkpoint

The Cost of Shifting

- `add(0, x)`와 `add(n, x)`의 비용이 같리는 이유는 무엇인가
- `remove(i)` 뒤에 빈 칸이 남지 않게 하려면 어느 원소를 어떤 순서로 옮겨야 하는가
- $n = 8$ 일 때 `add(4, x)`가 옮기는 원소는 몇 개인가

What Arrays Are Good For

배열에서 비용이 작은 연산과 큰 연산을 구분한다

배열의 연산별 비용

하려는 일	배열	근거	쓰임 예
1번째 원소 읽기	$\Theta(1)$	주소 계산 한 번	번호로 성적 조회
빈 칸이 있을 때 끝에 추가	$\Theta(1)$	옮길 원소 없음	기록을 차례로 쌓기
앞·가운데 삽입·삭제	$\Theta(n)$	뒤쪽을 전부 이동	자주 끼워 넣으면 불리
값으로 찾기	최악 $\Theta(n)$	순서를 모르면 전부 확인	이름으로 찾으면 불리

배열의 연산 비용은 원소를 읽거나 바꾸는 위치에 따라 달라진다

ArrayStack으로 스택을 만들면

스택(stack)의 push와 pop을 배열의 끝에 대응시킨다

스택 연산	ArrayStack 연산	옮기는 원소	비용
push(x)	add(n, x)	0	$\Theta(1)$
pop()	remove(n-1)	0	$\Theta(1)$
앞을 쓰는 push	add(0, x)	n	$\Theta(n)$
앞을 쓰는 pop	remove(0)	$n - 1$	$\Theta(n)$

내부 배열의 칸 수(a.length)를 바꾸는 비용을 빼고 센 값이다.

배열의 끝에서 넣고 꺼내면 push와 pop 모두 원소를 옮기지 않는다

Python list 연산의 비용

Python list도 배열 표현이므로 연산이 건드리는 위치가 비용을 정한다

Python list	ArrayStack	옮기는 원소	비용
<code>a[i]</code>	<code>get(i)</code>	0	$O(1)$
<code>a[i] = x</code>	<code>set(i, x)</code>	0	$O(1)$
<code>a.append(x)</code>	<code>add(n, x)</code>	0	$O(1)$
<code>a.pop()</code>	<code>remove(n-1)</code>	0	$O(1)$
<code>a.insert(i, x)</code>	<code>add(i, x)</code>	$n - i$	$O(n)$
<code>a.pop(i)</code>	<code>remove(i)</code>	$n - i - 1$	$O(n)$

append가 원소를 옮기지 않는 것은 내부 배열에 빈 칸이 있을 때다. 표의 $O(n)$ 은 맨 앞을 건드릴 때의 최악이다.

같은 $\Theta(n)$ 도 실제 시간은 다를 수 있다

```
1 def shift_loop(a, i, n):
2     for j in range(n, i, -1):
3         a[j] = a[j - 1]
4 def shift_slice(a, i, n):
5     a[i + 1:n + 1] = a[i:n]
```

n	반복문	슬라이스
10,000	0.38 ms	0.08 ms
100,000	4.92 ms	0.87 ms

a 는 끝에 빈 칸 하나를 둔 길이 $n + 1$ 의 목록이다. 맨 앞 $i = 0$ 에서 이동 한 번의 시간이며 5묶음 측정의 중앙값이다. 슬라이스 대입은 인터프리터 안에서 원소 묶음을 한 번에 복사하고, 반복문은 원소마다 Python 문장을 실행한다.

두 방법 모두 원소 n 개를 옮기지만 원소 하나를 옮기는 비용이 다르다

Adapted from Morin (2013), Section 2.2.

배열을 고르는 기준

자주 부르는 연산이 배열의 어느 위치를 건드리는지 먼저 본다

상황	자주 부르는 연산	판단
학번 순서의 점수를 번호로 조회	<code>get(i)</code>	적합 · $\Theta(1)$
실행 기록을 끝에 계속 추가	<code>add(n, x)</code>	적합 · 옮길 원소 없음
대기 목록 맨 앞에 자주 끼워 넣기	<code>add(0, x)</code>	비효율적 · 매번 $\Theta(n)$

위치로 읽고 끝에서 바꾸는 일이 많을수록 배열이 유리하다

Checkpoint

What Arrays Are Good For

- 배열에서 $\Theta(1)$ 인 연산과 $\Theta(n)$ 인 연산은 각각 무엇인가
- 반복문 이동과 슬라이스 이동의 복잡도가 같은데 시간이 다른 이유는 무엇인가
- 자주 쓰는 연산의 위치가 배열 선택에 어떤 영향을 주는가

Why Does Position Decide the Cost?

Next Lecture: Dynamic Arrays

- 용량이 가득 찼을 때 하는 일
- 두 배로 늘리는 방법과 그 비용
- 상각 비용으로 본 끝 추가의 $\Theta(1)$