



부산대학교
PUSAN NATIONAL UNIVERSITY

Dynamic Arrays

When the Array Fills Up

Capacity, Resizing, and Amortized Cost

DS3600082-140

자료구조

2026학년도 2학기

오용경

데이터사이언스전공

yongkyung.oh@pusan.ac.kr

Capacity and Resize

칸이 다 차면 더 큰 배열로 옮긴다

원소 수와 담을 수 있는 칸 수

정의

동적 배열(dynamic array)

필요할 때 내부 배열의 크기 재할당(resize)으로 저장 공간을 늘리는 자료구조다.
용량(capacity)은 담을 수 있는 칸 수이고, n 은 현재 담긴 원소 수다.

$n = 5$, 용량 8



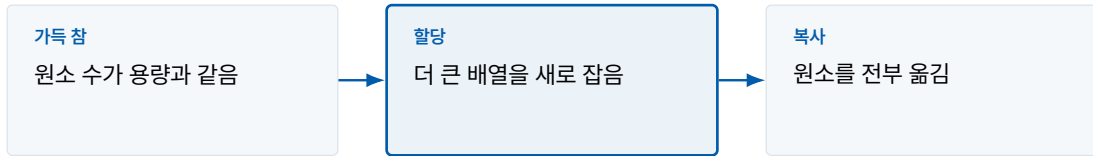
빈 칸에 바로 추가

$n = 8$, 용량 8



먼저 공간을 늘림

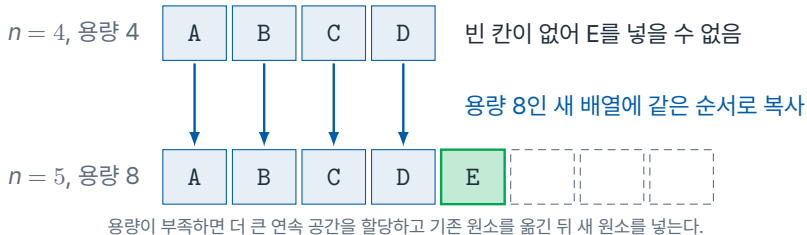
배열은 늘릴 수 없다



까닭	내용
연속 저장 재할당 비용	뒤 칸이 이미 다른 자료일 수 있어 이어 붙일 수 없음 새 배열을 잡고 옮긴 뒤 옛 배열을 버리는 일 원소 n 개를 옮기므로 $\Theta(n)$

가득 찬 배열에 E를 추가하면

더 큰 배열을 만들고 기존 원소를 같은 순서로 복사한다



고정 폭 증가와 배수 증가

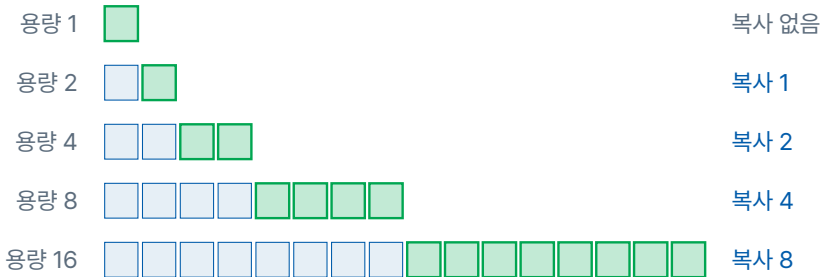
초기 용량 1에서 원소 16개를 담을 때까지 용량이 바뀌는 순서를 비교한다

전략	재할당 뒤 용량	재할당 횟수
두 칸씩 늘림	3, 5, 7, 9, 11, 13, 15, 17	8번
두 배로 늘림	2, 4, 8, 16	4번

원소 16개를 담는 동안 두 칸씩 늘리면 8번, 두 배로 늘리면 4번 재할당한다

두 배로 늘릴 때 배열의 모습

초기 용량 1에서 원소를 하나씩 추가하며, 가득 찬 순간의 재할당만 그린다



파란 칸은 재할당 때 옮긴 원소이고, 초록 칸은 그 뒤에 새로 채운 칸이다. 16칸을 채우는 동안 복사는 $1 + 2 + 4 + 8 = 15$ 번이다.

줄이기도 한다

용량 12, $n = 4$  용량 $\geq 3n$

용량 8  $2n$ 으로 줄여도 빈 칸 4개

언제	조건	하는 일
늘림	$n = \text{capacity}$	용량을 $\max(1, 2n)$ 로 변경
줄임	$\text{capacity} \geq 3n$	용량을 $\max(1, 2n)$ 로 변경

Caution 줄인 뒤에도 빈 칸을 남겨야 추가·삭제가 반복될 때 재할당을 매번 하지 않는다

Checkpoint

Capacity and Resize

- 용량과 현재 원소 수는 어떻게 다른가
- 배열을 그 자리에서 늘릴 수 없는 까닭은 무엇인가
- 줄인 뒤에도 빈 칸을 남겨 두는 까닭은 무엇인가

Section 2 • Dynamic Arrays

Counting the Copies

늘리는 전략별 복사량을 합으로 비교한다

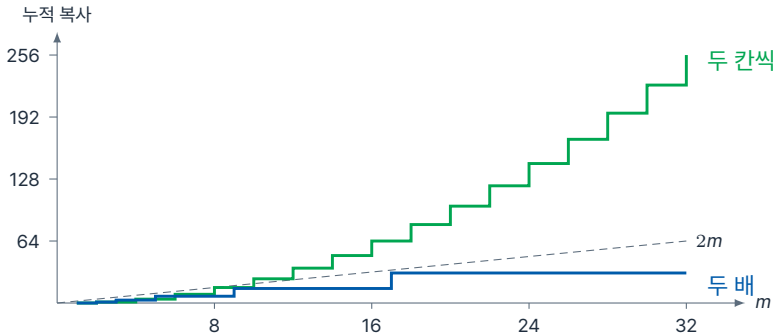
추가를 m 번 할 때 복사한 원소 수

m	두 배: 누적 복사	$\div m$	두 칸씩: 누적 복사	$\div m$
2	1	0.50	1	0.50
4	3	0.75	4	1.00
8	7	0.88	16	2.00
16	15	0.94	64	4.00
17	31	1.82	64	3.76
32	31	0.97	256	8.00

두 칸씩 늘리면 m 이 두 배가 될 때 한 번당 복사도 두 배가 된다

증가 전략에 따른 누적 복사량

초기 용량 1에서 추가를 32번 하는 동안의 누적 복사 수



두 칸씩 늘리면 계단이 점점 높아지고, 두 배로 늘리면 누적 복사가 $2m$ 아래에 머문다.

복사량의 합으로 증가 전략을 비교한다

고정 폭 증가에서는 등차수열, 배수 증가에서는 등비수열의 합이 나온다

전략	복사량의 합	m 번 추가	한 번당
두 칸씩	$\sum_{j=1}^{\lfloor m/2 \rfloor} (2j-1) = \lfloor m/2 \rfloor^2$	$\Theta(m^2)$	$\Theta(m)$
c 칸씩	$c + 2c + 3c + \dots \approx m^2/(2c)$	$\Theta(m^2)$	$\Theta(m)$
두 배	$1 + 2 + 4 + \dots + 2^k < 2m$ ($2^k < m$)	$\Theta(m)$	$\Theta(1)$

고정 폭으로 늘리면 폭을 크게 잡아도 총 복사는 $\Theta(m^2)$ 을 벗어나지 못한다

$m = 100$ 에 합 공식을 대입한다

2의 거듭제곱이 아닌 m 에서도 앞 쪽의 합으로 두 전략을 비교할 수 있다

전략	재할당 때 복사한 원소 수	합	한 번당
두 배	$1 + 2 + 4 + 8 + 16 + 32 + 64$	127	1.27
두 칸씩	$1 + 3 + 5 + \dots + 99$ (50개 항)	2500	25

두 배 전략의 합 127은 $2m = 200$ 보다 작다. 두 칸씩의 합은 $(100/2)^2 = 2500$ 이다. $m = 200$ 이면 두 합은 255와 10000이 된다.

m 이 두 배가 되면 두 배 전략의 한 번당 복사는 그대로이고, 두 칸씩은 두 배가 된다

Checkpoint

Counting the Copies

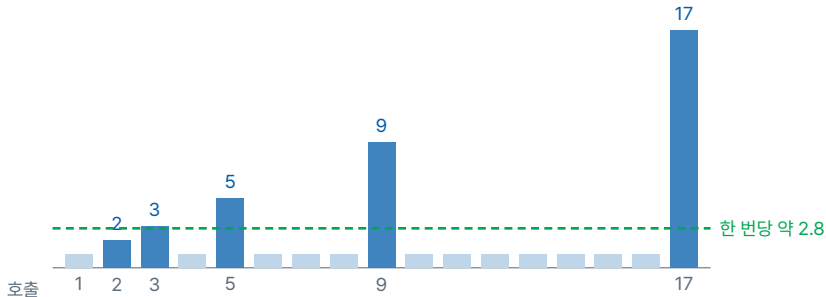
- 두 칸씩 늘리면 $m = 32$ 에서 누적 복사가 256이 되는 까닭은 무엇인가
- 두 배로 늘릴 때 복사량의 합이 $2m$ 보다 작은 이유는 무엇인가
- 고정 폭 증가와 배수 증가의 차이는 무엇인가

Amortized Cost

가끔 비싼 연산을 전체로 나누어 센다

add 호출별 쓰기·복사 횟수

초기 용량 1에서 끝에 17번 추가하며, 각 호출의 쓰기 1회와 기존 원소 복사 횟수를 센다



대부분의 호출은 비용이 1이고 재할당하는 호출만 높다. 17번의 총비용은 쓰기 17과 복사 31을 더한 48이다.

연속한 연산의 총비용

한 번의 최악값과 여러 add의 총비용은 서로 다른 양이다

분석 대상	계산하는 양	뜻
한 번의 재할당	$\Theta(n)$	현재 원소를 새 배열로 복사
연속한 연산	$\sum_{i=1}^m c_i$	각 연산의 실제 비용 c_i 를 합산
한 번당 비용	$\frac{1}{m} \sum_{i=1}^m c_i$	전체 비용을 연산 횟수로 나눔

연속한 연산의 총비용을 연산 수로 나누면 한 번당 비용이 된다

정의

상각 분석(amortized analysis)

연속한 연산의 총비용으로 한 번당 비용을 분석하는 방법이다.

어떤 순서로 m 번 호출해도 총비용이 $m \times c$ 를 넘지 않으면 c 를 상각 비용(amortized cost)이라 한다. 빈 동적 배열에 m 번 추가한 총비용이 $O(m)$ 이면 한 번당 상각 비용은 $O(1)$ 이다.

$$\frac{\text{총비용}}{m} = \frac{17 + 31}{17} \approx 2.8 \qquad \frac{m + \text{복사}}{m} < \frac{m + 2m}{m} = 3$$

왼쪽은 앞에서 본 17번 추가의 값이다. 오른쪽은 복사가 $2m$ 보다 적으므로(앞 절) m 에 관계없이 $c = 3$ 이 성립함을 보인다.

상각 비용은 입력 분포를 가정하지 않는다

용어	평균을 내는 대상	가정
평균 경우(average case)	입력의 분포	입력 분포가 필요
상각 비용	연속한 연산의 나열	확률 분포 가정 없음

Caution 상각 분석은 확률이 아니라 모든 연산 나열에 대한 총비용 상한을 다룬다

재할당의 총비용은 $O(m)$ 이다

보조정리

재할당의 총비용

빈 동적 배열(ArrayStack)에서 `add`와 `remove`를 m 번 부르면
모든 재할당에 드는 시간의 합은 $O(m)$ 이다



늘릴 때와 줄일 때의 복사 비용

직전 재할당 후 용량이 C 이면 원소는 약 $C/2$ 개다

단계	내용
늘림	C 개가 될 때까지 약 $C/2$ 번 추가한 뒤, C 개 복사
줄임	$C/3$ 개 이하가 될 때까지 약 $C/6$ 번 삭제한 뒤, 약 $C/3$ 개 복사
공통	각 복사 비용을 그 사이의 연산들에 상수만큼씩 나누어 부과

어느 방향이든 복사 한 번 앞에는 복사량에 비례하는 수의 연산이 있다

연산별 최악의 경우와 상각 비용

연산	최악의 경우	상각 비용
끝에 add	$\Theta(n)$	$\Theta(1)$
끝에서 remove	$\Theta(n)$	$\Theta(1)$
get, set	$\Theta(1)$	$\Theta(1)$
스택 push(x): add(n, x)	$\Theta(n)$	$\Theta(1)$
스택 pop(): remove(n-1)	$\Theta(n)$	$\Theta(1)$

스택 ADT가 요구한 두 연산이 배열 위에서 상각 $\Theta(1)$ 로 구현된다

Checkpoint

Amortized Cost

- 상각 비용과 평균 경우는 무엇이 다른가
- 재할당 사이의 연산 수와 복사량을 비교하는 까닭은 무엇인가
- 끝에 `add`를 최악의 경우만으로 설명하면 왜 충분하지 않은가

Inside Python Lists

Python list의 메모리 증가를 관찰해 재할당을 확인한다

Python list가 쓰는 방식

CPython은 원소의 참조를 배열에 저장하고 여유 칸이 부족하면 공간을 늘린다

연산	Python	상각 비용
끝에 추가	<code>append</code>	$\Theta(1)$
앞에 추가	<code>insert(0, x)</code>	$\Theta(n)$
위치로 읽기	<code>a[i]</code>	$\Theta(1)$

`append`와 `insert(0, x)`는 같은 배열 위에서 이동하는 원소 수가 다르다

원소 수와 메모리 사용량의 증가

관찰 코드

```
1 import sys
2
3 items = []
4 print(0, sys.getsizeof(items))
5 for i in range(25):
6     items.append(i)
7     print(len(items), sys.getsizeof(items))
```

각 append 뒤의 원소 수와 메모리 크기를 기록해 증가 구간을 찾는다

메모리는 몇 단계로 증가한다

원소 수	sizeof (bytes)	담을 수 있는 칸 수
0	56	0
1 이상 4 이하	88	4
5 이상 8 이하	120	8
9 이상 16 이하	184	16
17 이상 24 이하	248	24
25	312	32

빈 `list`는 56바이트, 참조 한 칸은 8바이트라 칸 수는 $(\text{bytes} - 56)/8$ 이다. 수치는 실행 환경에 따라 달라질 수 있다.

원소 수는 하나씩 늘지만 메모리는 몇 단계에 걸쳐 증가한다

Adapted from UBC DSCI 221 (n.d.), Dynamic Arrays.

append 한 번당 측정 시간

```
1 def build(n):  
2     a = []  
3     for i in range(n):  
4         a.append(i)  
5     return a
```

n	전체	한 번당
10,000	0.39 ms	39 ns
100,000	4.88 ms	49 ns
1,000,000	50.0 ms	50 ns

전체 시간은 5묶음 중앙값이고, 한 번당 시간은 전체를 n 으로 나눈 값이다.

n 이 100배가 되는 동안 한 번당 시간은 약 1.3배로, 한 번당 $\Theta(n)$ 일 때의 100배와 달리 상각 $\Theta(1)$ 예측과 맞는다

Checkpoint

Inside Python Lists

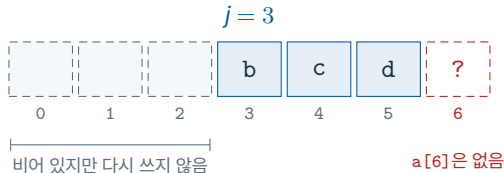
- `sys.getsizeof`가 같은 값에 머무는 구간은 무엇을 뜻하는가
- `append` 한 번당 시간을 전체 시간에서 구하는 방법은 무엇인가
- 이 측정만으로 재할당 배수를 단정할 수 없는 이유는 무엇인가

Circular Arrays

원형 배열로 양쪽 끝 연산의 원소 이동을 줄인다

선형 인덱스만 쓰면 앞쪽 빈 칸을 재사용하지 못한다

첫 원소의 인덱스 j 를 하나씩 늘리면 삭제할 때 원소를 옮기지 않지만 앞쪽에 빈 칸이 남는다



$n = 3$ 에서 $\text{add}(x)$
 $j + n = 6$ 은 배열 밖

쓸 수 있는 칸이 남았는데 자리가 없다고 판단하게 된다

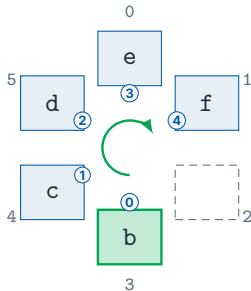
정의

원형 배열(circular array)

인덱스가 배열 끝을 지나면 앞으로 돌아오게 하는 방식이다. i 번째 원소는 $a[(j + i) \bmod \text{capacity}]$ 로 읽는다

j	i	$j + i$	실제 칸 (용량 12)
9	0	9	9
9	2	11	11
9	3	12	0

끝을 지나면 0번 칸으로 돌아온다



용량 6, $j = 3$, $n = 5$

$i = 0$ 은 $a[3]$, $i = 2$ 는 $a[5]$

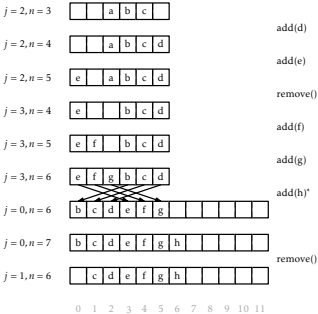
$i = 3$ 이면 $(3 + 3) \bmod 6 = 0$

다음 추가는 $(3 + 5) \bmod 6 = 2$ 번 칸

나머지 연산 덕분에 앞쪽 빈 칸을 원소 이동 없이 다시 쓴다

끝에 도달하면 앞쪽 빈 칸을 쓴다

j 는 첫 원소의 인덱스이고 n 은 담긴 원소 수다



원형 배열의 추가·삭제 과정이다. 별표는 재할당한 연산을 나타낸다.

Source: Morin (2013), Figure 2.2 (ArrayQueue).

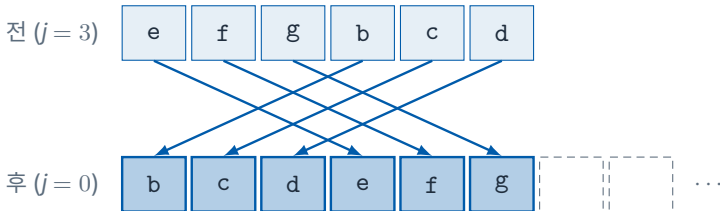
원형이어도 재할당은 필요하다

연산·상황	원형 배열에서의 처리	비용·결과
앞쪽 삭제	$j \leftarrow (j + 1) \bmod \text{capacity}$	상각 $\Theta(1)$
i 번째에 삽입	가까운 끝 쪽 원소를 한 칸씩 이동	상각 $\Theta(1 + \min\{i, n - i\})$
원소 수가 용량에 도달	더 큰 배열로 재할당	$\Theta(n)$

원형 배열은 앞쪽 삭제 비용을 줄이지만 용량 확장과 가운데 삽입 비용은 남는다

재할당하면 원형이 풀린다

재할당할 때는 j 부터 읽은 순서대로 새 배열의 0번 칸부터 담는다



재할당 뒤에는 첫 원소가 0번 칸에 오므로 j 를 0으로 되돌린다

Checkpoint

Circular Arrays

- $a[(j + i) \bmod \text{capacity}]$ 가 하는 일은 무엇인가
- 원형 배열이 해결하는 것과 해결하지 못하는 것은 각각 무엇인가
- 재할당할 때 j 를 0으로 되돌리는 까닭은 무엇인가

Why Can a Costly Operation Have Constant Amortized Cost?

Next Lecture: Singly Linked Lists

- 연속을 포기하고 노드로 잇는 표현
- 값과 다음 노드의 참조를 담는 노드
- head와 next로 노드를 따라가는 방법