



부산대학교
PUSAN NATIONAL UNIVERSITY

Queues and Deques

First In, First Out

Circular Queues and Double-Ended Operations

DS3600082-140

자료구조

2026학년도 2학기

오용경

데이터사이언스전공

yongkyung.oh@pusan.ac.kr

The Queue ADT

먼저 들어온 값을 먼저 꺼내는 순서와 연산 조건을 정한다

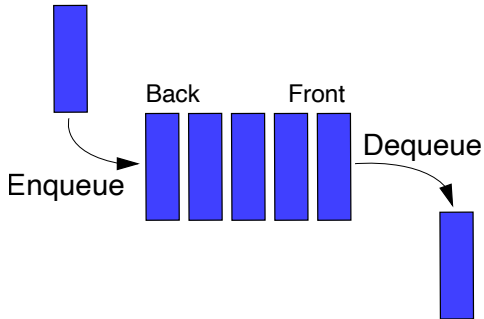
정의

큐(queue)

뒤쪽(rear)에서 값을 넣고 앞쪽(front)에서 꺼내는 ADT. first-in first-out(FIFO)는 가장 먼저 넣은 값을 가장 먼저 꺼내는 순서 규칙이다.

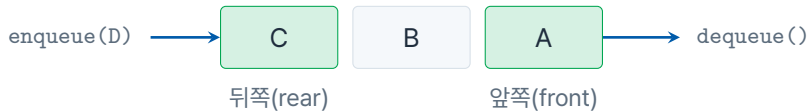
FIFO를 그림으로 보면

먼저 들어온 값은 앞쪽에서 먼저 나가고, 새 값은 뒤쪽으로 들어온다.



그림의 Back은 rear, Front는 front에 해당한다.

앞쪽과 뒤쪽의 역할



값은 뒤쪽으로 들어오고 앞쪽으로 나가므로 가장 먼저 넣은 A가 가장 먼저 나간다.

큐 연산

연산	반환값	효과
<code>enqueue(x)</code>	없음	x 를 뒤쪽에 추가
<code>dequeue()</code>	앞쪽 값	앞쪽 값을 삭제
<code>is_empty()</code>	True/False	비어 있는지 검사
<code>size()</code>	정수	저장한 값의 수를 반환

Caution 빈 큐에서 `dequeue()`는 돌려줄 값이 없으므로 먼저 `is_empty()`로 확인한다

Checkpoint

The Queue ADT

- 큐에서 값을 넣고 꺼내는 끝은 각각 어디인가?
- `enqueue(A)`, `enqueue(B)`, `dequeue()`의 반환값은 무엇인가?
- 빈 큐에서 별도의 조건이 필요한 연산은 무엇인가?

A Circular Queue

원소를 밀지 않고 앞쪽과 뒤쪽 위치만 옮기는 원형 큐를 만든다

Python list의 앞을 사용하면

앞 원소를 삭제하면 뒤의 원소를 모두 한 칸씩 옮겨야 한다.

연산	이동 수	시간
<code>append(x)</code>	0개(재할당 때만 n 개 복사)	상각 $O(1)$
<code>pop(0)</code>	$n - 1$ 개	$O(n)$
<code>insert(0, x)</code>	n 개	$O(n)$
<code>pop()</code>	0개	상각 $O(1)$

이후 비용표의 O 는 표시한 경우(최악, 상각, 기대)의 증가 차수를 나타낸다.

list로 만든 큐는 한쪽 연산이 $O(n)$ 이다

list에서 뒤쪽 위치	enqueue	dequeue
인덱스 0	$O(n)$	상각 $O(1)$
인덱스 $n - 1$	상각 $O(1)$	$O(n)$

Caution list에서 방향만 바꾸어서는 두 연산을 모두 $O(1)$ 로 만들 수 없다.

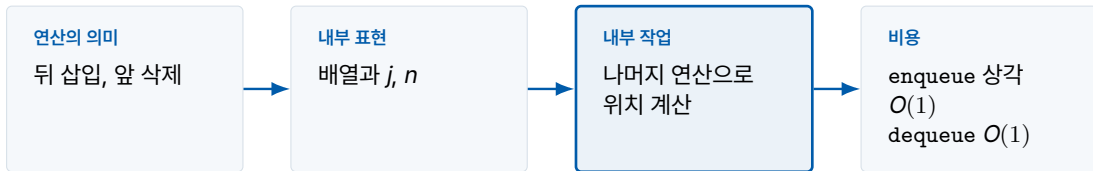
배열 끝을 다시 쓰는 원형 큐

정의

원형 큐(circular queue)

배열의 마지막 칸 다음을 첫 칸으로 이어서 사용하는 큐. `dequeue()` 때 원소를 앞으로 밀지 않고 앞쪽 위치를 가리키는 인덱스만 바꾼다.

원형 배열은 큐의 내부 표현이 된다



FIFO는 큐가 보장하는 순서이고 원형 배열은 그 순서를 구현하는 한 방법이다

원형 큐의 상태는 세 값이면 된다

상태	뜻
a	값을 저장하는 길이 m 의 배열
j	논리적 첫 원소가 놓인 배열 인덱스
n	현재 저장한 원소 수

j 와 n 을 따로 저장하면 빈 상태와 가득 찬 상태도 구분할 수 있다.

논리 인덱스를 배열 인덱스로 바꾼다

큐의 j 번째 값은 배열에서 $(j + i) \bmod m$ 위치에 있다.

j	m	논리 인덱스 i	배열 인덱스
5	8	0	5
5	8	1	6
5	8	2	7
5	8	3	0

나머지 연산이 배열 끝을 지날 때 인덱스를 0부터 다시 시작하게 한다.

배열 끝을 지나도 순서는 이어진다



논리 순서 $[A, B, C, D]$ 는 배열에서는 인덱스 5, 6, 7, 0에 나뉘어 저장된다

인덱스만 바꾸는 두 연산

배열이 가득 차지 않았고 dequeue 때는 비어 있지 않다고 가정하고 핵심 대입만 읽는다

```
1 def enqueue(self, value):
2     end = (self.j + self.n) % len(self.a)
3     self.a[end] = value
4     self.n += 1
5
6 def dequeue(self):
7     value = self.a[self.j]
8     self.j = (self.j + 1) % len(self.a)
9     self.n -= 1
10    return value
11
```

원소를 옮기지 않으므로 배열 용량이 충분할 때 두 연산은 $O(1)$ 이다.

빈 상태와 가득 찬 상태

다음 삽입 위치 $(j + n) \bmod m$ 은 빈 상태와 가득 찬 상태에서 모두 j 와 같으므로 n 으로 구분한다.

상태	조건	다음 동작
비어 있음	$n = 0$	dequeue 금지
저장 가능	$0 < n < m$	enqueue와 dequeue 가능
가득 참	$n = m$	배열을 늘린 뒤 enqueue

배열을 늘릴 때는 논리 순서로 복사한다

원형 배열에서 본 것처럼 앞쪽 원소를 새 배열의 인덱스 0에 놓아 순서를 다시 연속으로 만든다.

단계	동작
1	길이 $2m$ 인 새 배열 b 를 만든다
2	$b[i] \leftarrow a[(j + i) \bmod m]$ 로 n 개를 복사한다
3	$a \leftarrow b, j \leftarrow 0$ 으로 바꾼다

$m = 8, j = 5$ 인 상태를 인덱스 그대로 16칸에 복사하면 논리 인덱스 3의 위치 $(5 + 3) \bmod 16 = 8$ 이 비어 D를 잃는다.

가끔 $O(n)$ 으로 복사해도 길이를 두 배로 늘리면 enqueue의 상각 비용은 $O(1)$ 이다.

Checkpoint

A Circular Queue

- 길이 8인 배열에서 $j = 6$ 일 때 논리 인덱스 3의 배열 인덱스는 무엇인가?
- dequeue 뒤에 j 와 n 은 각각 어떻게 바뀌는가?
- 배열을 늘릴 때 원소를 논리 순서로 복사해야 하는 이유는 무엇인가?

Queue Simulations

큐를 돌려 발표 차례를 정하고 프린터 대기 시간을 계산한다

큐로 시뮬레이션할 수 있는 것

현실의 사건	큐 연산	기록할 값
새 작업 도착	enqueue	도착 시각
처리 시작	dequeue	대기 시간
처리 진행	큐 변화 없음	남은 처리량

도착 순서를 큐에 저장하면 대기 시간을 사건 순서대로 계산할 수 있다.

큐를 돌려 발표 차례를 정한다

대기 명단은 A, B, C, D다. 앞 사람을 뒤로 3번 보낸 뒤 다음 앞 사람에게 발표 차례를 배정한다.

단계	큐 연산
앞 사람을 뒤로 한 번 이동	<code>enqueue(dequeue())</code>
정해진 횟수만큼 이동	같은 연산을 3번 반복
차례를 배정하고 명단에서 삭제	반복 뒤 <code>dequeue()</code>
한 명만 남음	마지막 발표 차례 배정

Caution 무작위 추첨이 아니다. 처음 명단과 이동 횟수가 같으면 발표 순서도 같다.

앞쪽 사람을 뒤쪽으로 한 번 보낸다

단계	앞쪽에서 뒤쪽 순서	꺼낸 값
시작	A, B, C, D	없음
<code>person = dequeue()</code>	B, C, D	A
<code>enqueue(person)</code>	B, C, D, A	A를 다시 넣음

한 번 이동하면 사람 수는 그대로이고 앞쪽 사람만 뒤쪽으로 간다. 발표 차례는 아직 배정하지 않는다.

발표 차례를 정하는 큐 상태를 추적한다

이동	앞쪽에서 뒤쪽 순서	이동한 사람
0	A, B, C, D	없음
1	B, C, D, A	A
2	C, D, A, B	B
3	D, A, B, C	C
차례 배정	A, B, C	D 삭제

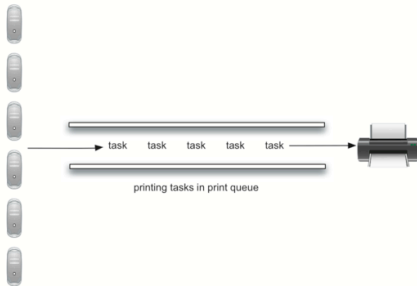
3번 이동하고 앞 사람을 선택하는 규칙을 반복하면 D, A, C 순으로 차례가 정해진다. 마지막 B까지 포함한 발표 순서는 D, A, C, B다.

원형 이동을 별도 원형 연결 구조 없이 큐의 두 연산으로 표현할 수 있다.

Adapted from Miller and Ranum (2013), Simulation: Hot Potato.

프린터 큐로 대기 시간을 계산한다

Lab Computers



컴퓨터가 만든 작업은 큐에서 기다리고 프린터는 앞쪽 작업부터 처리한다.

Source: Miller and Ranum (2013), Simulation: Printing Tasks.

매 시각에 확인할 세 가지

확인	상태 변화
작업이 도착했는가	도착 시각과 분량을 저장해 enqueue
프린터가 비었는가	큐가 비지 않았다면 dequeue하여 시작
작업이 끝났는가	남은 분량이 0이면 프린터를 빈 상태로 변경

Source: Miller and Ranum (2013), Simulation: Printing Tasks.

처리한 작업의 대기 시간을 평균한다

작업 k 의 대기 시간은 처리 시작 시각에서 도착 시각을 뺀 값이다.

$$w_k = s_k - a_k, \quad \bar{w} = \frac{1}{N} \sum_{k=1}^N w_k$$

a_k : 도착 시각 s_k : 처리 시작 시각 N : 처리한 작업 수

Caution 5 ppm 프린터로 같은 시뮬레이션을 10회 실행하면 평균 대기는 17.27초부터 376.05초까지 흩어지고 모든 작업을 끝낸 실행은 두 번뿐이다. N 은 처리한 작업만 세므로 남은 작업 수도 함께 본다.

Checkpoint

Queue Simulations

- 발표 대기 명단의 앞 사람을 뒤로 한 번 보내는 동작은 어떤 두 큐 연산으로 표현하는가?
- 작업이 도착할 때 반드시 함께 저장해야 하는 시각은 무엇인가?
- 평균 대기 시간만으로 시뮬레이션을 판단하기 어려운 이유는 무엇인가?

The Deque ADT

양 끝에서 값을 넣고 빼는 덱의 연산을 구분한다

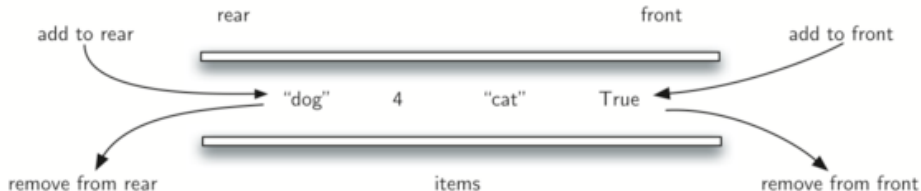
정의

덱(deque)

양 끝에서 값을 넣고 뺄 수 있는 ADT. 연산마다 앞쪽이나 뒤쪽 가운데 사용할 끝을 고른다. 철자가 비슷한 `dequeue()`는 덱이 아니라 큐의 삭제 연산이다.

덱은 양 끝에서 값을 넣고 뺀다

추가와 삭제가 앞쪽과 뒤쪽 양쪽에서 가능하다.



같은 값 순서에서도 다음 연산이 어느 끝을 쓰는지 명시해야 한다.

덱의 네 가지 끝 연산

연산	같은 뜻의 다른 이름	끝	효과
<code>add_front(x)</code>	<code>add_first(x)</code>	앞쪽	x 추가
<code>remove_front()</code>	<code>remove_first()</code>	앞쪽	값 삭제 후 반환
<code>add_rear(x)</code>	<code>add_last(x)</code>	뒤쪽	x 추가
<code>remove_rear()</code>	<code>remove_last()</code>	뒤쪽	값 삭제 후 반환

Source: Miller and Ranum (2013), The Deque Abstract Data Type.

스택과 큐는 덱으로 표현할 수 있다

규칙	추가	삭제
스택	같은 끝	같은 끝
큐	뒤쪽	앞쪽
덱	앞쪽 또는 뒤쪽	앞쪽 또는 뒤쪽

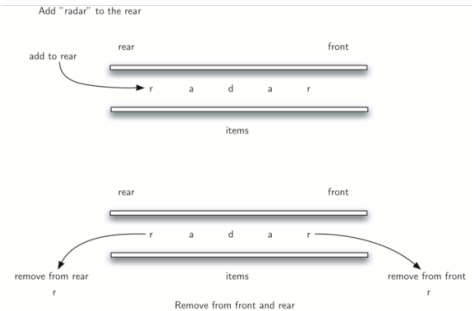
덱은 양 끝 연산을 제공하므로 스택과 큐의 동작을 모두 표현할 수 있다.

회문 검사는 덱의 양 끝을 비교한다

왼쪽 끝과 오른쪽 끝의 문자를 한 쌍씩 비교한다.

단계	동작
1	문자열의 문자를 차례로 덱에 넣는다
2	앞쪽과 뒤쪽에서 하나씩 삭제해 비교한다
3	다르면 중단하고, 한 글자 이하만 남으면 회문이다

양 끝 비교로 가운데를 향한다



radar의 양 끝 r을 삭제하면 다음 비교 대상은 a와 a다.

Checkpoint

The Deque ADT

- 덱과 dequeue는 각각 무엇을 뜻하는가?
- 덱으로 스택을 만들 때 추가와 삭제는 어느 끝에서 해야 하는가?
- 회문 검사에서 양 끝 연산이 필요한 이유는 무엇인가?

Python deque

Python의 덱 API를 연산 위치와 비용에 맞춰 선택한다

list는 양 끝 비용이 다르다

	추가	삭제	적합한 용도
오른쪽 끝	상각 $O(1)$	상각 $O(1)$	스택
왼쪽 끝	$O(n)$	$O(n)$	반복 사용 부적합

양 끝을 자주 사용한다면 list보다 deque가 적합하다.

collections.deque의 네 연산

Python 연산	사용하는 끝	ADT 연산
<code>appendleft(x)</code>	왼쪽	앞쪽에 추가
<code>popleft()</code>	왼쪽	앞쪽에서 삭제
<code>append(x)</code>	오른쪽	뒤쪽에 추가
<code>pop()</code>	오른쪽	뒤쪽에서 삭제

앞의 덱 그림은 오른쪽 끝을 앞쪽으로 그렸지만, `collections.deque`로 큐를 만들 때는 왼쪽 끝을 앞쪽으로 쓴다

큐를 deque로 구현하면

append로 오른쪽에 넣고 popleft로 왼쪽에서 꺼낸다

```
1 from collections import deque
2
3 q = deque()
4 q.append("A")
5 q.append("B")
6 first = q.popleft()
7
8 print(first, list(q))  # A ['B']
9
```

큐에서는 오른쪽에 넣고 왼쪽에서 꺼내면 FIFO가 된다.

Source: Python Standard Library (n.d.), collections.deque documentation.

양 끝 연산은 $O(1)$ 이다

deque는 양 끝의 추가와 삭제를 반복하는 용도로 설계되었다.

연산	시간	사용 예
appendleft, popleft	$O(1)$	큐의 앞쪽
append, pop	$O(1)$	뒤쪽 또는 스택

Source: Python Standard Library (n.d.), collections.deque documentation.

연산 위치에 따라 list와 deque를 고른다

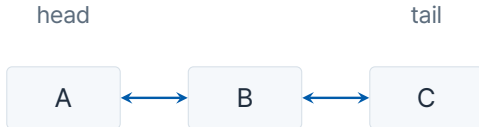
필요한 연산	우선 선택	이유
양 끝 추가·삭제	deque	양 끝 $O(1)$
인덱스로 자주 접근	list	list 임의 접근 $O(1)$, deque 가운데 $O(n)$
중간 삽입·삭제	상황에 따라 선택	위치 탐색 비용도 고려

Caution deque를 모든 리스트 연산이 빠른 구조로 해석하면 안 된다.

Source: Python Standard Library (n.d.), collections.deque documentation.

양 끝 $O(1)$ 을 만드는 한 가지 구조

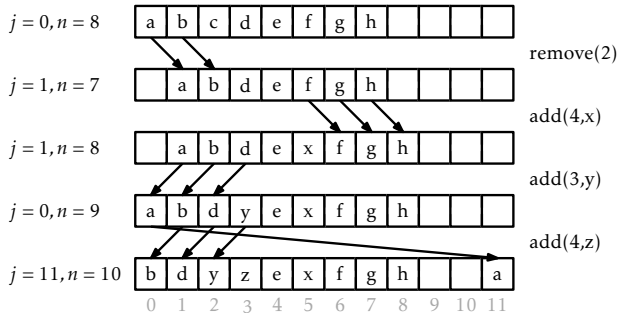
머리(head)와 꼬리(tail)를 가진 이중 연결 구조는 양 끝 노드에 바로 접근할 수 있다.



이 그림은 덱을 구현하는 한 방법이며, Python deque는 내부 구현과 상관없이 양 끝 연산을 $O(1)$ 로 제공한다

배열로 만든 덱

원형 배열 위의 덱은 j 만 움직여 양 끝을 처리하고, 가운데 연산은 가까운 끝 쪽 원소를 옮긴다



`add(3,y)`는 앞쪽 셋을 왼쪽으로 옮기고, `add(4,z)`에서 j 가 0 앞을 지나 11로 간다. `add(i,x)·remove(i)`는 List 연산이며 옮기는 원소는 $\min\{i, n - i\}$ 개 이하이다.

Source: Morin (2013), Figure 2.3.

세 ADT를 한 번에 정리하면

ADT	추가 위치	삭제 위치	순서
스택	맨 위	맨 위	LIFO
큐	뒤쪽	앞쪽	FIFO
덱	양 끝	양 끝	연산이 결정

자료구조 선택은 값의 종류보다 어떤 위치를 어떤 순서로 쓰는지에서 시작한다.

Checkpoint

Python deque

- `list.pop(0)` 대신 `deque.popleft()`를 쓰는 이유는 무엇인가?
- `deque`로 스택과 큐를 각각 어떻게 구현할 수 있는가?
- 인덱스 접근이 핵심일 때 `deque`보다 `list`가 나은 이유는 무엇인가?

자료구조

How Can a Key Find Its Place Without Searching?

Next Lecture: Hash Tables and Chaining

- 키를 위치로 바꾸는 해시 함수
- 버킷과 충돌
- 체이닝으로 충돌을 담는 방법