



부산대학교
PUSAN NATIONAL UNIVERSITY

Doubly Linked Lists

Links Both Ways

Previous, Next, and the Sentinel Node

DS3600082-140

자료구조

2026학년도 2학기

오용경

데이터사이언스전공

yongkyung.oh@pusan.ac.kr

Adding prev

현재 노드만으로는 바로 앞 노드를 알 수 없다

단방향 연결의 한계

이전 노드를 가리키는 참조가 없으면 앞 노드를 찾으려 처음부터 따라가야 한다

하러던 일	SLList에서	까닭
끝에서 삭제	$\Theta(n)$	앞 노드를 처음부터 찾음
가운데 삭제	순회하며 기억	previous를 따로 관리
바로 앞 노드 확인	현재 노드만으로는 불가능	앞 노드 참조가 없음

마지막 노드의 앞을 찾으려면 처음부터 걷는다

head $\rightarrow$ a $\rightarrow$ b $\rightarrow$ c에서 tail이 c를 가리켜도 앞 노드를 찾으려면 head부터 따라가야 한다

걸음	현재 노드	기억한 앞 노드	다음
시작	a	없음	b
1	b	a	c
2	c	b	없음

끝에서 지울 때 필요한 b를 얻으려면 처음부터 연결을 따라가야 한다

prev 참조를 추가한다

정의

이중 연결리스트(doubly linked list)

각 노드가 다음 노드를 가리키는 `next`와 이전 노드(previous)를 가리키는 `prev`를 함께 갖는 표현

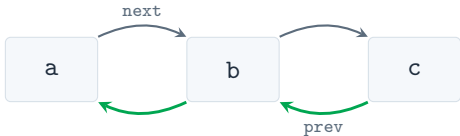
필드	나타내는 것
<code>u.data</code>	이 노드가 담은 값
<code>u.next</code>	뒤따르는 노드
<code>u.prev</code>	이전 노드

순회할 때 따로 들고 다니던 변수 `previous`와 달리 `prev`는 노드마다 저장한다.

Source: Morin (2013), Section 3.2.

끝에서 앞 노드로 바로 이동한다

c.prev가 b를, b.prev가 a를 가리킨다



마지막 노드의 참조를 이미 알면 바로 앞 노드를 한 번에 찾는 대신 노드마다 prev 참조 하나를 더 저장한다

Checkpoint

Adding prev

- prev를 더하면 무엇이 가능해지는가
- 그 대가로 무엇을 더 쓰는가
- 끝에서 생기는 특별한 경우는 몇 가지인가

The Sentinel

센티널로 양끝과 빈 목록을 같은 규칙으로 처리한다

빈 노드 하나를 놓는다

정의

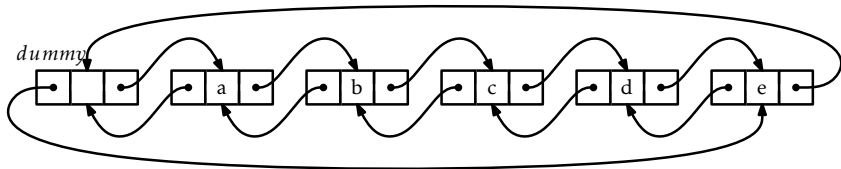
센티넬(sentinel)

값을 담지 않고 양끝을 대신해 연결만 맡는 노드. 코드에서는 dummy라 하며 마지막 노드의 뒤와 첫 노드의 앞을 이어 목록을 닫는다

목록이 고리가 되면 prev나 next가 비는 자리가 없어진다

고리로 닫힌 목록

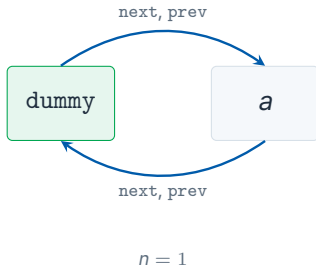
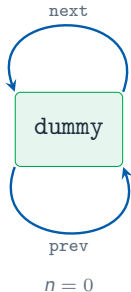
a, b, c, d, e 를 담은 DLList에서 센티널과 다섯 노드가 양방향으로 연결된다



dummy가 e 의 뒤이면서 a 의 앞이라 양끝이 이어진다. 센티널로 닫힌 이 구조가 원형 이중 연결리스트다

빈 목록에서는 dummy가 자신을 가리킨다

`dummy.next`와 `dummy.prev`가 모두 dummy인 상태가 빈 목록이다



Source: Morin (2013), Section 3.2, `initialize()`.

첫 노드를 넣으면 네 참조가 한 고리를 이룬다

빈 목록에 `u`를 넣으면 `dummy`의 두 참조와 `u`의 두 참조가 서로를 가리킨다

참조	빈 목록	<code>u</code> 를 넣은 뒤
<code>dummy.next</code>	<code>dummy</code>	<code>u</code>
<code>dummy.prev</code>	<code>dummy</code>	<code>u</code>
<code>u.prev</code>	없음	<code>dummy</code>
<code>u.next</code>	없음	<code>dummy</code>

첫 노드도 앞뒤에 센티넬이 있어 별도의 `head/tail` 분기가 없다

센티넬을 초기화하는 코드

만들 때 하는 일은 dummy를 자기 자신에 잇는 것뿐

```
1 def initialize(self):  
2     self.n = 0  
3     self.dummy = Node(None)  
4     self.dummy.prev = self.dummy  
5     self.dummy.next = self.dummy  
6
```

빈 목록에서도 prev와 next가 항상 유효한 노드를 가리킨다

센티널이 양끝의 별도 분기를 없앤다

head와 tail을 따로 갱신할 일이 사라진다

상황	센티널 없이	센티널과 함께
첫 노드 앞에 삽입	head 갱신	일반 삽입과 같음
마지막 노드 삭제	tail 갱신	일반 삭제와 같음
빈 목록에 삽입	둘 다 갱신	일반 삽입과 같음

센티널은 양끝과 빈 목록을 처리하는 별도 분기를 없앤다

Checkpoint

The Sentinel

- dummy는 어떤 값을 담는가
- 빈 목록에서 `dummy.next`는 무엇을 가리키는가
- 센티넬이 없애는 특별한 경우는 무엇인가

Finding and Changing

가까운 끝에서 출발하면 많아야 절반만 걷는다

어느 끝에서 출발할지 고른다

i 가 앞쪽이면 `dummy.next`에서 `next`로, 뒤쪽이면 `dummy`에서 `prev`로 이동한다

조건	출발점	걸음 수
$i < n/2$	<code>dummy.next</code>	i
$i \geq n/2$	<code>dummy</code>	$n - i$

$i = 6$ 을 찾아간다

$n = 8$ 이므로 dummy에서 prev로 $8 - 6 = 2$ 걸음이면 닿는다

걸음	앞에서 갔다면	뒤에서 간다면
출발	dummy.next (0번째)	dummy
1	1번째	7번째
2	2번째	6번째 도착
...	6걸음 필요	끝

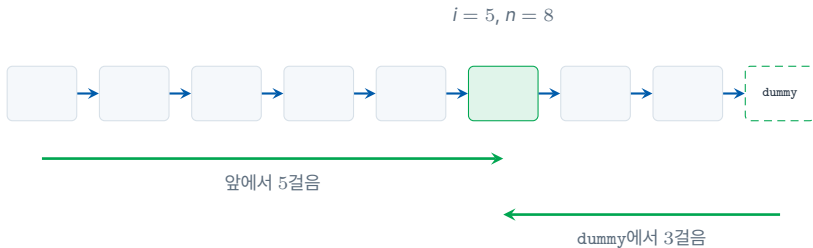
가까운 끝을 고르는 코드

앞 절반은 `next`로, 뒤 절반은 `prev`로 이동한다

```
1 def get_node(self, i):
2     if i < self.n / 2:
3         p = self.dummy.next
4         for _ in range(i):
5             p = p.next
6     else:
7         p = self.dummy
8         for _ in range(self.n, i, -1):
9             p = p.prev
10    return p
11
```

Adapted from Morin (2013), Section 3.2, `get_node(i)`.

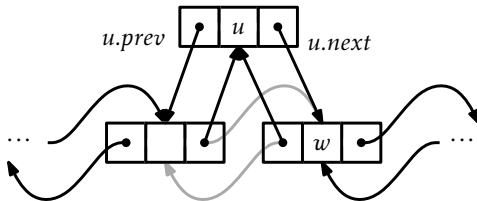
두 끝에서 걸음 수를 비교한다



가까운 끝을 고르면 최대 $\lfloor n/2 \rfloor$ 걸음만 이동한다

w 앞에 넣는다

w.prev를 바꾸기 전에 그 값을 u.prev에 옮겨 둔다



새 노드 u 의 prev와 next를 먼저 정하고 양옆이 u 를 가리키게 한다

네 줄의 순서



`add_before(w, x)`는 새 노드 `u = Node(x)`를 만든 뒤 네 줄을 실행하고 `self.n += 1`로 원소 수를 늘린다.

새 노드를 연결하는 동안 기존 연결을 보존한다

a와 w 사이에 u를 넣으며 참조를 하나씩 바꾼다

한 일	a에서 w로 가는 상태
시작	<code>a.next=w</code> , <code>w.prev=a</code>
<code>u.prev=a</code> , <code>u.next=w</code>	기존 연결 유지, u가 양옆을 기억
<code>w.prev=u</code>	역방향은 $w \rightarrow u \rightarrow a$
<code>a.next=u</code>	정방향도 $a \rightarrow u \rightarrow w$

u에 옛 앞 노드를 먼저 저장해야 마지막 연결까지 그 참조를 잃지 않는다

a, b, c에서 `add_before(b, x)`

w=b이고 새 노드 u가 값 x를 담는다

줄	실행한 대입	바뀐 참조
시작	없음	<code>a.next=b</code> , <code>b.prev=a</code>
1	<code>u.prev=w.prev</code>	<code>u.prev=a</code>
2	<code>u.next=w</code>	<code>u.next=b</code>
3	<code>u.next.prev=u</code>	<code>b.prev=u</code>
4	<code>u.prev.next=u</code>	<code>a.next=u</code>

정방향으로 `dummy.next`부터 읽으면 `a, x, b, c`, 역방향으로 `dummy.prev`부터 읽으면 `c, b, x, a`다.

삽입 뒤에는 정방향과 역방향이 모두 이어졌는지 함께 확인한다

순서를 뒤집으면

`w.prev`를 먼저 바꾸면 기존 앞 노드를 가리키던 참조가 사라진다

먼저 한 일	그다음 벌어지는 일
<code>u.prev, u.next</code> 먼저	양옆 노드의 참조를 <code>u</code> 에 보존
<code>w.prev = u</code> 먼저	기존 앞 노드의 참조를 잃어 순회 없이는 <code>u.prev</code> 를 정할 수 없음
<code>u.prev.next = u</code> 를 <code>u.prev</code> 설정 전에 실행	<code>u.prev</code> 가 아직 없어 앞 노드를 고칠 수 없음

Caution `u`를 먼저 채우는 것이 단일 연결 구조의 삽입과 같은 규칙이다

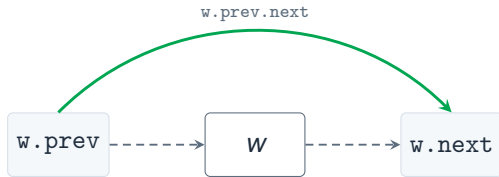
경계에서도 같은 삽입 규칙을 쓴다

센티넬이 있으므로 첫 자리와 빈 목록에서도 `w.prev`는 유효한 노드를 가리킨다

넣는 자리	<code>w.prev</code> 가 가리키는 것
목록 한가운데	앞 노드
첫 자리	<code>dummy</code>
빈 목록	<code>dummy</code> 자신
맨 끝 뒤	<code>w=dummy</code> 이고 <code>w.prev</code> 는 마지막 노드

w를 지운다

양옆 노드가 서로를 가리키게 하면 목록에서는 w에 더 이상 도달할 수 없다



w.next.prev도 w.prev를 가리키게 바꾼다

참조 두 개를 바꾸면 되고 경계 위치도 별도로 처리하지 않는다

a, x, b, c에서 remove(b)

w=b이면 앞 노드 x와 뒤 노드 c가 서로를 가리키게 된다

실행한 대입	바뀐 참조	원소 수
시작	없음	$n = 4$
<code>w.prev.next=w.next</code>	<code>x.next=c</code>	$n = 4$
<code>w.next.prev=w.prev</code>	<code>c.prev=x</code>	$n = 4$
<code>self.n -= 1</code>	없음	$n = 3$

정방향은 a, x, c, 역방향은 c, x, a이고 b는 어느 방향에서도 도달하지 않는다.

삭제는 참조 두 개만 바꾸며 양옆이 첫 노드나 dummy여도 같은 두 줄이다

마지막 원소를 지워도 센티널은 남는다

한 노드 w 만 남았을 때 앞뒤 노드는 모두 dummy다

한 일	남은 참조
시작	<code>dummy.next=w, dummy.prev=w</code>
<code>w.prev.next=w.next</code>	<code>dummy.next=dummy</code>
<code>w.next.prev=w.prev</code>	<code>dummy.prev=dummy</code>

원소 수가 0이 되면 센티널의 두 참조가 다시 자기 자신을 가리킨다

노드 참조를 알면 두 참조만 바꾼다

앞뒤 노드가 서로를 가리키게 하면 w 를 건너뛴다

```
1 def remove_node(self, w):
2     w.prev.next = w.next
3     w.next.prev = w.prev
4     self.n -= 1
5     return w.data
6
```

위치를 찾는 비용을 제외하면 참조 변경 자체는 $\Theta(1)$ 이다

Checkpoint

Finding and Changing

- $i = 1$ 과 $i = n - 1$ 을 찾을 때 각각 몇 걸음 이동하는가
- 삽입에서 u 쪽 참조를 먼저 채우는 까닭은 무엇인가
- 삭제에 특별한 경우가 없는 까닭은 무엇인가

What It Costs

노드를 찾는 비용이 삽입·삭제 비용을 지배한다

DLList의 연산 비용

하는 일	비용	비용의 원인
i 번째 노드 찾기	$\Theta(1 + \min\{i, n - i\})$	가까운 끝에서 순회
찾은 노드 앞에 삽입	$\Theta(1)$	참조 네 개 갱신
찾은 노드 삭제	$\Theta(1)$	참조 두 개 갱신

인덱스로 요청하면 찾는 비용이 들고, 노드 참조가 주어지면 연결 변경 비용만 든다

인덱스 연산은 `get_node` 위에 선다

네 인덱스 연산은 노드를 찾은 뒤 값을 읽거나 연결을 바꾼다

```
1 def get(self, i):
2     return self.get_node(i).data
3
4 def set(self, i, x):
5     u = self.get_node(i)
6     y = u.data
7     u.data = x
8     return y
9
```

```
1 def add(self, i, x):
2     self.add_before(self.get_node(i), x)
3
4 def remove(self, i):
5     w = self.get_node(i)
6     return self.remove_node(w)
7
```

각 연산의 비용은 찾기 $\Theta(1 + \min\{i, n - i\})$ 에 값 읽기나 연결 변경 $\Theta(1)$ 을 더한 것이다

노드당 추가 공간

이중 연결리스트의 노드 하나는 값 하나와 참조 두 개를 저장한다

구성	n 개를 담는 자리	추가 비용
값	n	원소마다 하나
next 참조	n	노드마다 하나
prev 참조	n	단일 연결리스트에는 없음

Caution 이전 노드로 이동할 수 있는 대신 단일 연결리스트보다 노드마다 참조 하나를 더 저장한다

left와 right로 쓴 이중 연결 노드

두 참조를 left와 right로 쓰고 센티널 없이 양끝을 None으로 두는 이중 연결 노드도 있다

```
1 class Node:
2     def __init__(self, data,
3                     left=None, right=None):
4         self.data = data
5         self.left = left
6         self.right = right
7
```

```
1 def print_deque(deque):
2     node = deque.head
3     while node:
4         print(node.data, end=', ')
5         node = node.right
6     print()
7     node = deque.tail
8     while node:
9         print(node.data, end=', ')
10        node = node.left
11
```

right를 따라가면 정방향, left를 따라가면 역방향이며 None을 만나면 끝이다

Checkpoint

What It Costs

- DLList에서 비싼 부분은 어디인가
- 노드 참조를 이미 알 때 삭제가 $\Theta(1)$ 인 까닭은 무엇인가
- 단일 연결리스트보다 노드마다 무엇을 더 저장하는가

Why Does One Extra Reference Change So Much?

Next Lecture: Arrays and Linked Lists

- 같은 연산이 두 표현에서 하는 내부 작업
- 연산 빈도로 표현을 고르는 방법
- 임의 접근과 순차 접근의 비용 차이