

제 2장 연결리스트

리스트

- 일반적인 리스트(List)는 일련의 동일한 타입의 항목(item)들
- 실생활의 예: 학생 명단, 시험 성적, 서점의 신간 서적, 상점의 판매 품목, 실시간 급상승 검색어, 버킷 리스트 등
- 일반적인 리스트의 구현:
 - 1차원 파이썬 리스트(list)
 - 단순연결리스트
 - 이중연결리스트
 - 원형연결리스트

2.1 단순연결리스트

- 단순연결리스트(Singly Linked List)는 동적 메모리 할당을 이용해 리스트를 구현하는 가장 간단한 형태의 자료구조
- 동적 메모리 할당을 받아 노드(node)를 저장하고, 노드는 레퍼런스를 이용하여 다음 노드를 가리키도록 만들어 노드들을 한 줄로 연결시킴



- 연결리스트에서는 삽입이나 삭제 시 항목들의 이동이 필요 없음
- 배열(자바, C, C++언어)의 경우 최초에 배열의 크기를 예측하여 결정해야 하므로 대부분의 경우 배열에 빈 공간을 가지고 있으나, 연결리스트는 빈 공간이 존재하지 않음
- 연결리스트에서는 항목을 탐색하려면 항상 첫 노드부터 원하는 노드를 찾을 때까지 차례로 방문하는 순차탐색(Sequential Search)을 해야

단순연결리스트를 위한 SList 클래스

```
01 class SList:
02     class Node:
03         def __init__(self, item, link):
04             self.item = item
05             self.next = link
06
07     def __init__(self):
08         self.head = None
09         self.size = 0
10
11     def size(self): return self.size
12     def is_empty(self): return self.size == 0
13
14     def insert_front(self, item):
15         if self.is_empty():
16             self.head = self.Node(item, None)
17         else:
18             self.head = self.Node(item, self.head)
19         self.size += 1
20
```

노드 생성자
항목과 다음 노드 레퍼런스

단순연결리스트 생성자
head와 항목 수(size)로 구성

empty인 경우

head가 새
노드 참조

```
21 def insert_after(self, item, p):
22     p.next = SList.Node(item, p.next)
23     self.size += 1
```

새 노드가 p 다음
노드가 됨

```
24
25 def delete_front(self):
26     if self.is_empty():
27         raise EmptyError('Underflow')
28     else:
29         self.head = self.head.next
30         self.size -= 1
```

empty인 경우 에러 처리

head가 둘째 노드를 참조

```
31
32 def delete_after(self, p):
33     if self.is_empty():
34         raise EmptyError('Underflow')
35     t = p.next
36     p.next = t.next
37     self.size -= 1
```

empty인 경우 에러 처리

p 다음 노드를 건너뛰어 연결

```
39 def search(self, target):
40     p = self.head
41     for k in range(self.size):
42         if target == p.item: return k
43         p = p.next
44     return None
45
46 def print_list(self):
47     p = self.head
48     while p:
49         if p.next != None:
50             print(p.item, ' -> ', end='')
51         else:
52             print(p.item)
53         p = p.next
54
55 class EmptyError(Exception):
56     pass
```

head로부터 순차탐색

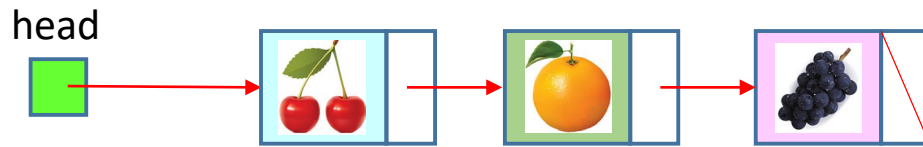
탐색 성공

탐색 실패

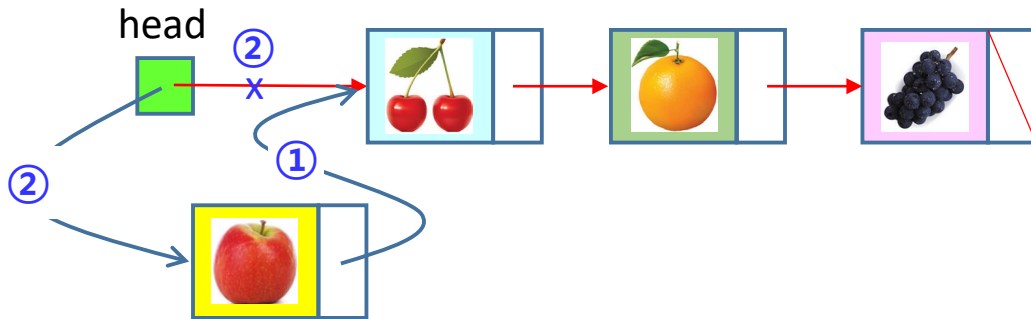
노드들을 순차탐색

underflow 시 에러 처리

[프로그램 2-1] slist.py



(a) 새 노드 삽입 전



```
18 self.head = self.Node(item, self.head)
```

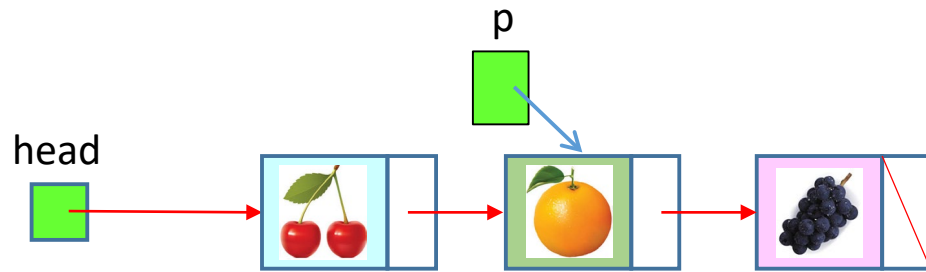
2



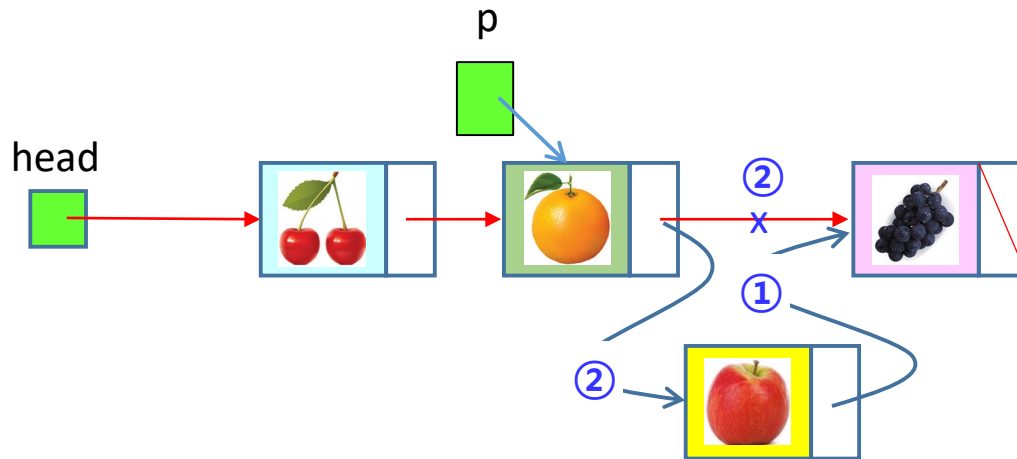
1

(b) 새 노드 삽입 후

[그림 2-2] insert_front() 함수



(a) 새 노드 삽입 전



```
22 p.next = SList.Node(item, p.next)
```

2



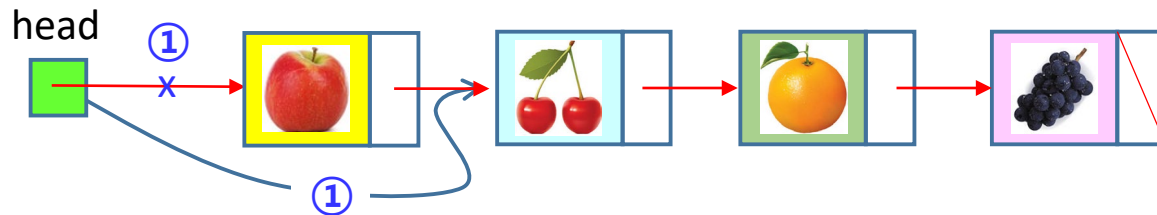
1

(b) 새 노드 삽입 후

[그림 2-3] insert_after() 함수



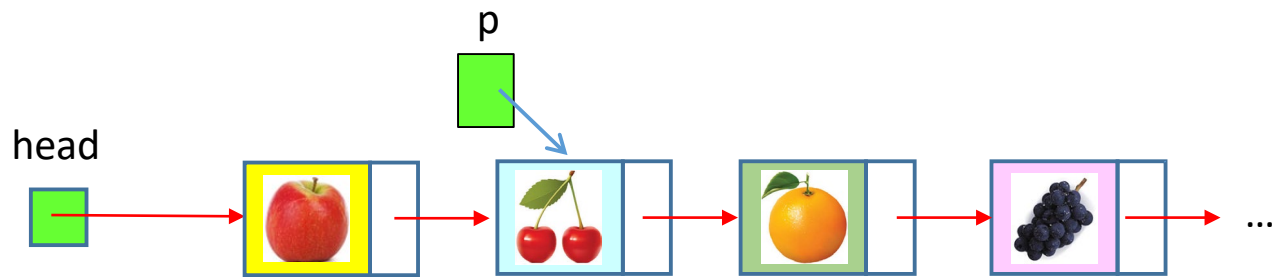
(a) 첫 노드 삭제 전



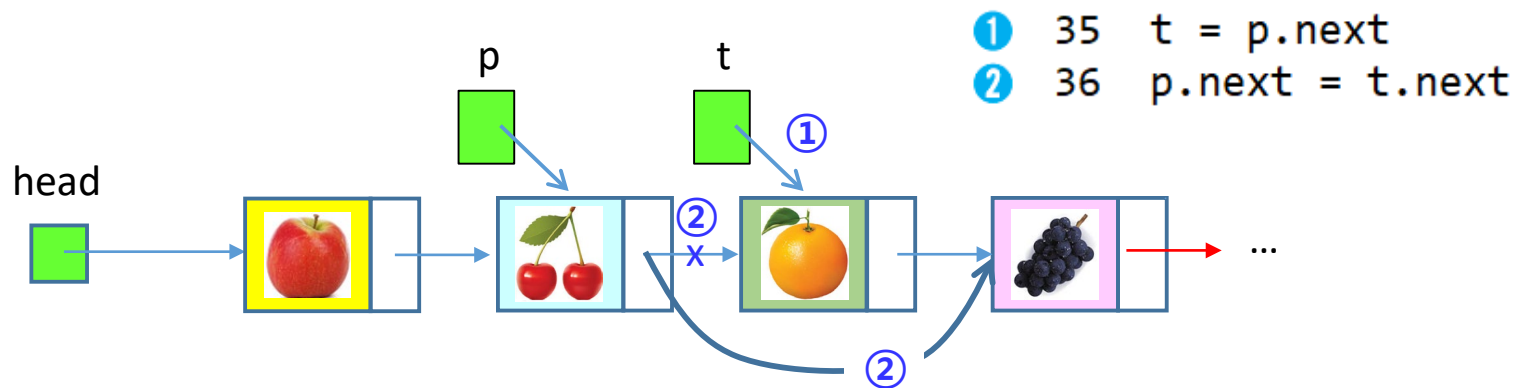
```
29 self.head = self.head.next
    ①
```

(b) 첫 노드 삭제 후

[그림 2-4] delete_front() 함수



(a) 노드 삭제 전



(b) 노드 삭제 후

[그림 2-5] delete_after() 함수

```

01 from slist import SList
02 if __name__ == '__main__':
03     s = SList()
04     s.insert_front('orange')
05     s.insert_front('apple')
06     s.insert_after('cherry', s.head.next)
07     s.insert_front('pear')
08     s.print_list()
09     print('cherry는 %d번째' % s.search('cherry'))
10     print('kiwi는', s.search('kiwi'))
11     print('배 다음 노드 삭제 후:\t\t', end='')
12     s.delete_after(s.head)
13     s.print_list()
14     print('첫 노드 삭제 후:\t\t', end='')
15     s.delete_front()
16     s.print_list()
17     print('첫 노드로 망고, 딸기 삽입 후:\t', end='')
18     s.insert_front('mango')
19     s.insert_front('strawberry')
20     s.print_list()
21     s.delete_after(s.head.next.next)
22     print('오렌지 다음 노드 삭제 후:\t', end='')
23     s.print_list()

```

slist.py에서 SList를 import

이 파이썬 파일(모듈)이 메인이면

단순연결리스트
생성

연결리스트의 삽입, 삭제, 탐색, 연산 수행

Console PyUnit

<terminated> main.py [C:\Users\sbayang\AppData\Local\Programs\Python\Python36-3

pear -> apple -> orange -> cherry

cherry는 3번째

kiwi는 None

배 다음 노드 삭제 후:

pear -> orange -> cherry

첫 노드 삭제 후:

orange -> cherry

첫 노드로 망고, 딸기 삽입 후:

strawberry -> mango -> orange -> cherry

오렌지 다음 노드 삭제 후:

strawberry -> mango -> orange

[프로그램 2-1, 2]의 수행 결과

Applications

- 단순연결리스트는 매우 광범위하게 활용되는데, 그 중에 3장의 스택과 큐 자료구조
- 6장 해싱의 체이닝(Chaining)에 사용
- 4장의 트리도 단순연결리스트의 개념을 확장시킨 자료구조

수행시간

- `search()`는 탐색을 위해 연결리스트의 노드들을 첫 노드부터 순차적으로 방문해야 하므로 $O(N)$ 시간 소요
- 삽입이나 삭제 연산은 각각 $O(1)$ 개의 레퍼런스만을 갱신하므로 $O(1)$ 시간 소요

단, `insert_after()`나 `delete_after()`의 경우에 특정 노드 `p`의 레퍼런스가 주어지지 않으면 `head`로부터 `p`를 찾기 위해 `search()`를 수행해야 하므로 $O(N)$ 시간 소요

2.2 이중연결리스트

- 이중연결리스트(Doubly Linked List)는 각 노드가 두 개의 레퍼런스를 가지고 각각 이전 노드와 다음 노드를 가리키는 연결리스트



- 단순연결리스트는 삽입이나 삭제할 때 반드시 이전 노드를 가리키는 레퍼런스를 추가로 알아내야 하고, 역방향으로 노드들을 탐색할 수 없음
- 이중연결리스트는 단순연결리스트의 이러한 단점을 보완하나, 각 노드마다 추가로 한 개의 레퍼런스를 추가로 저장해야 한다는 단점을 가짐

이중연결리스트를 위한 DList 클래스

```
01 class DList:
02     class Node:
03         def __init__(self, item, prev, link):
04             self.item = item
05             self.prev = prev
06             self.next = link
07
08     def __init__(self):
09         self.head = self.Node(None, None, None)
10         self.tail = self.Node(None, self.head, None)
11         self.head.next = self.tail
12         self.size = 0
13
14     def size(self): return self.size
15     def is_empty(self): return self.size == 0
16
```

노드 생성자
항목과 앞뒤 노드 레퍼런스

이중연결리스트 생성자
head와 tail, 항목 수(size)로 구성

```
17 def insert_before(self, p, item):
18     t = p.prev
19     n = self.Node(item, t, p)
20     p.prev = n
21     t.next = n
22     self.size += 1
23
24 def insert_after(self, p, item):
25     t = p.next
26     n = self.Node(item, p, t)
27     t.prev = n
28     p.next = n
29     self.size += 1
30
31 def delete(self, x):
32     f = x.prev
33     r = x.next
34     f.next = r
35     r.prev = f
36     self.size -= 1
37     return x.item
38
```

새 노드와 앞뒤
노드 연결

새 노드 생성하여
n이 참조

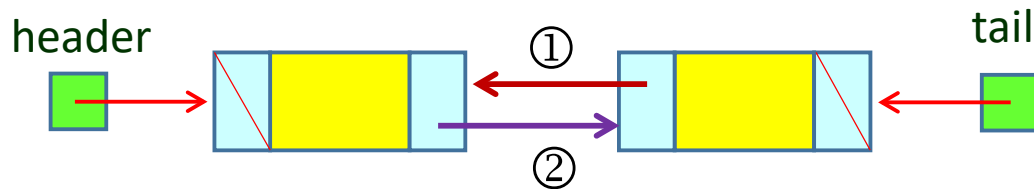
x를 건너 띄고 x의 앞뒤
노드를 직접 연결

```
39     def print_list(self):
40         if self.is_empty():
41             print('리스트 비어있음')
42         else:
43             p = self.head.next
44             while p != self.tail:
45                 if p.next != self.tail:
46                     print(p.item, ' <=> ', end='')
47                 else:
48                     print(p.item)
49                 p = p.next
50
51 class EmptyError(Exception):
52     pass
```

노드들을 차례로 방문하기 위해

underflow 시 에러 처리

[프로그램 2-3] dlist.py



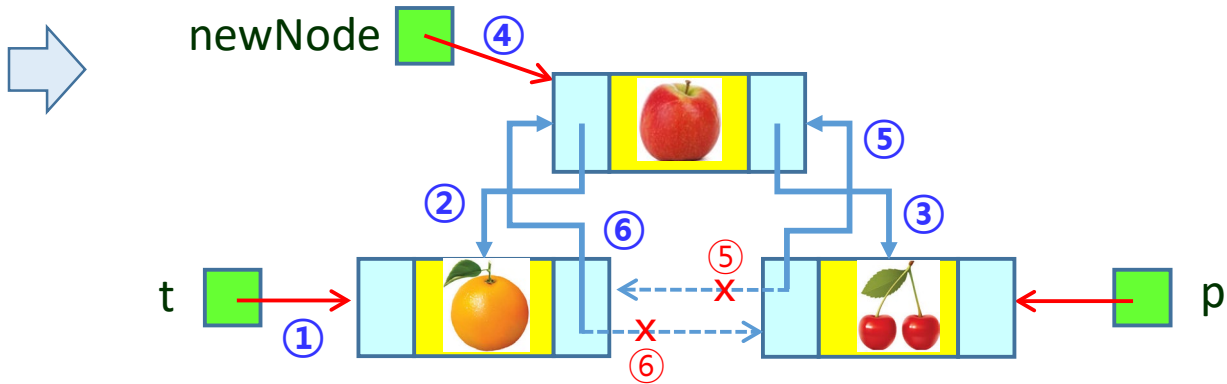
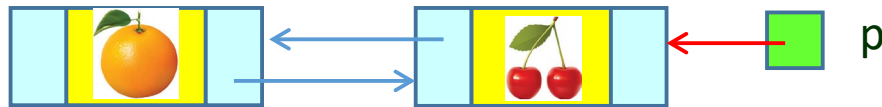
[그림 2-8] DList 객체 생성

```
09 self._head = self._Node(None, None, None)
10 self._tail = self._Node(None, self._head, None)
11 self._head._next = self._tail    ①
                                   ②
```

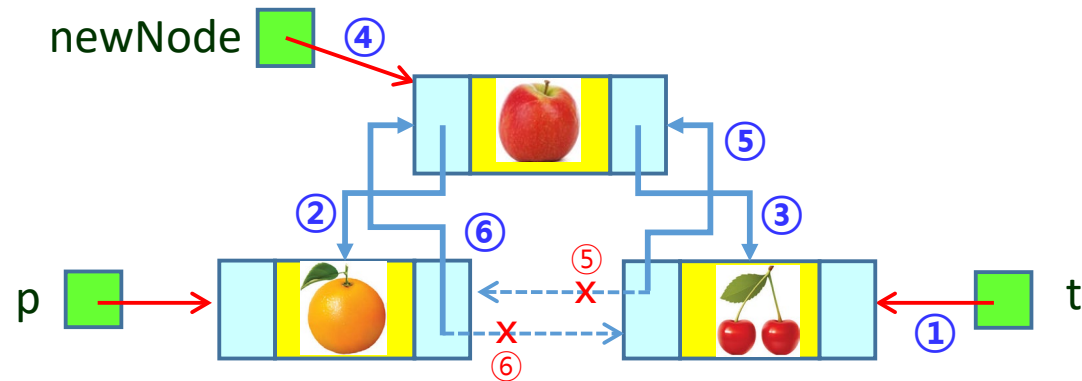
```

18     t = p._prev
19     n = self._Node(item, t, p)
20     p._prev = n
21     t._next = n

```

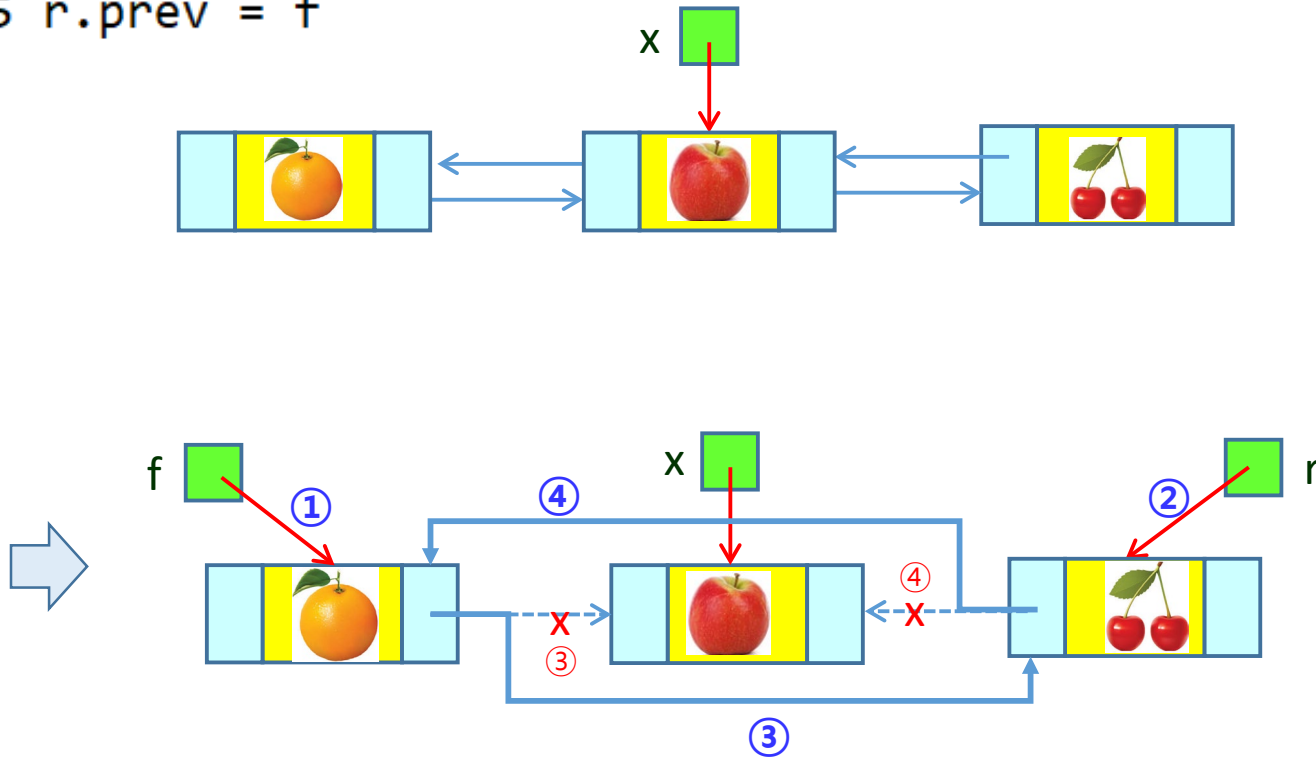


[그림 2-9] `insert_before()`의 삽입 수행



[그림 2-10] insert_after()의 삽입 수행

- ① 32 `f = x.prev`
- ② 33 `r = x.next`
- ③ 34 `f.next = r`
- ④ 35 `r.prev = f`



[그림 2-11] delete()의 삭제 수행

```

01 from dlist import DList
02 if __name__ == '__main__':
03     s = DList()
04     s.insert_after(s.head, 'apple')
05     s.insert_before(s.tail, 'orange')
06     s.insert_before(s.tail, 'cherry')
07     s.insert_after(s.head.next, 'pear')
08     s.print_list()
09     print('마지막 노드 삭제 후:\t', end='')
10     s.delete(s.tail.prev)
11     s.print_list()
12     print('맨 끝에 포도 삽입 후:\t', end='')
13     s.insert_before(s.tail, 'grape')
14     s.print_list()
15     print('첫 노드 삭제 후:\t', end='')
16     s.delete(s.head.next)
17     s.print_list()
18     print('첫 노드 삭제 후:\t', end='')
19     s.delete(s.head.next)
20     s.print_list()
21     print('첫 노드 삭제 후:\t', end='')
22     s.delete(s.head.next)
23     s.print_list()
24     print('첫 노드 삭제 후:\t', end='')
25     s.delete(s.head.next)
26     s.print_list()

```

이중연결리스트 생성


dlist.py에서 DList를 import

이 파이썬 파일(모듈)이 메인이면

일련의 삽입 삭제 탐색 연산 수행

[프로그램 2-4] main.py

[프로그램 2-3, 4] 수행 결과

Console  PyUnit

<terminated> main.py [C:\Users\wsbyang\AppData\Local\Programs\Python\Python36-32\python.exe]

apple <=> pear <=> orange <=> cherry

마지막 노드 삭제 후: **apple <=> pear <=> orange**

맨 끝에 포도 삽입 후: **apple <=> pear <=> orange <=> grape**

첫 노드 삭제 후: **pear <=> orange <=> grape**

첫 노드 삭제 후: **orange <=> grape**

첫 노드 삭제 후: **grape**

첫 노드 삭제 후: **리스트 비어있음**

수행시간

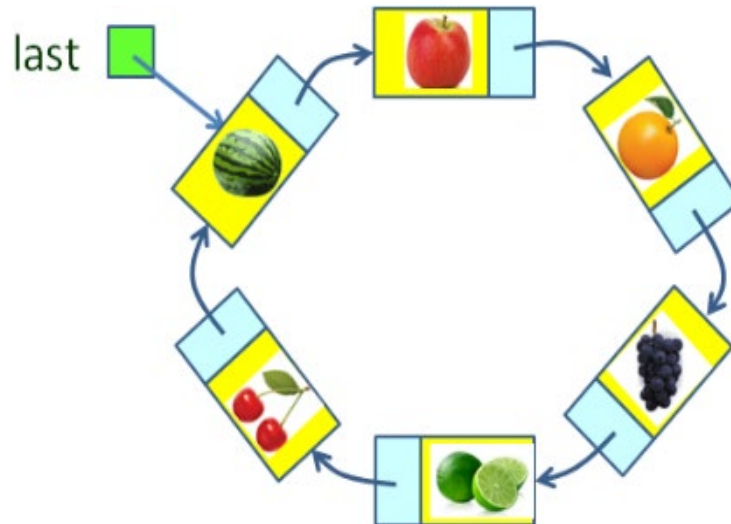
- 이중연결리스트에서 삽입이나 삭제 연산은 각각 상수 개의 레퍼런스만을 갱신하므로 $O(1)$ 시간에 수행
- 탐색 연산: head 또는 tail로부터 노드들을 순차적으로 탐색해야 하므로 $O(N)$ 시간 소요

Applications

- 이중연결리스트는 3장의 덱(Deque) 자료구조를 구현하는데 사용
- 이항힙(Binomial Heap)이나 피보나치힙(Fibonacci Heap)과 같은 우선순위큐를 구현하는 데에도 이중연결리스트가 부분적으로 사용

2-3 원형연결리스트

- 원형연결리스트(Circular Linked List)는 마지막 노드가 첫 노드와 연결된 단순연결리스트
- 원형연결리스트에서는 마지막 노드의 레퍼런스가 저장된 last가 단순연결리스트의 head와 같은 역할



- 마지막 노드와 첫 노드를 $O(1)$ 시간에 방문할 수 있는
장점
- 리스트가 empty가 아니면 어떤 노드도 None 레퍼런스를
가지고 있지 않으므로 프로그램에서 None 조건을
검사하지 않아도 되는 장점
- 원형연결리스트에서는 반대 방향으로 노드들을
방문하기 쉽지 않으며, 무한 루프가 발생할 수 있음에
유의할 필요

원형연결리스트를 위한 CList 클래스

```
01 class CList:
02     class _Node:
03         def __init__(self, item, link):
04             self.item = item
05             self.next = link
06
07     def __init__(self):
08         self.last = None
09         self.size = 0
10
11     def no_items(self): return self.size
12     def is_empty(self): return self.size == 0
13
14     def insert(self, item):
15         n = self._Node(item, None)
16         if self.is_empty():
17             n.next = n
18             self.last = n
19         else:
20             n.next = self.last.next
21             self.last.next = n
22         self.size += 1
23
```

노드 생성자
항목과 다음 노드 레퍼런스

원형연결리스트 생성자
last와 항목 수(size)로 구성

새 노드 생성하여
n이 참조

새 노드는 자신을 참조하고
last가 새 노드 참조

새 노드는 첫 노드를 참조하고
last가 참조하는 노드와 새 노드 연결

```
24 def first(self):
25     if self.is_empty():
26         raise EmptyError('Underflow')
27     f = self.last.next
28     return f.item
29
30 def delete(self):
31     if self.is_empty():
32         raise EmptyError('Underflow')
33     x = self.last.next
34     if self.size == 1:
35         self.last = None
36     else:
37         self.last.next = x.next
38     self.size -= 1
39     return x.item
40
```

empty 리스트가 됨

last가 참조하는 노드가 두 번째 노드를 연결

```
41 def print_list(self):
42     if self.is_empty():
43         print('리스트 비어있음')
44     else:
45         f = self.last.next
46         p = f
47         while p.next != f:
48             print(p.item, ' -> ', end='')
49             p = p.next
50         print(p.item)
51
52 class EmptyError(Exception):
53     pass
```

첫 노드가 다시 방문되면 루프 중단

노드들을 차례로 방문하기 위해

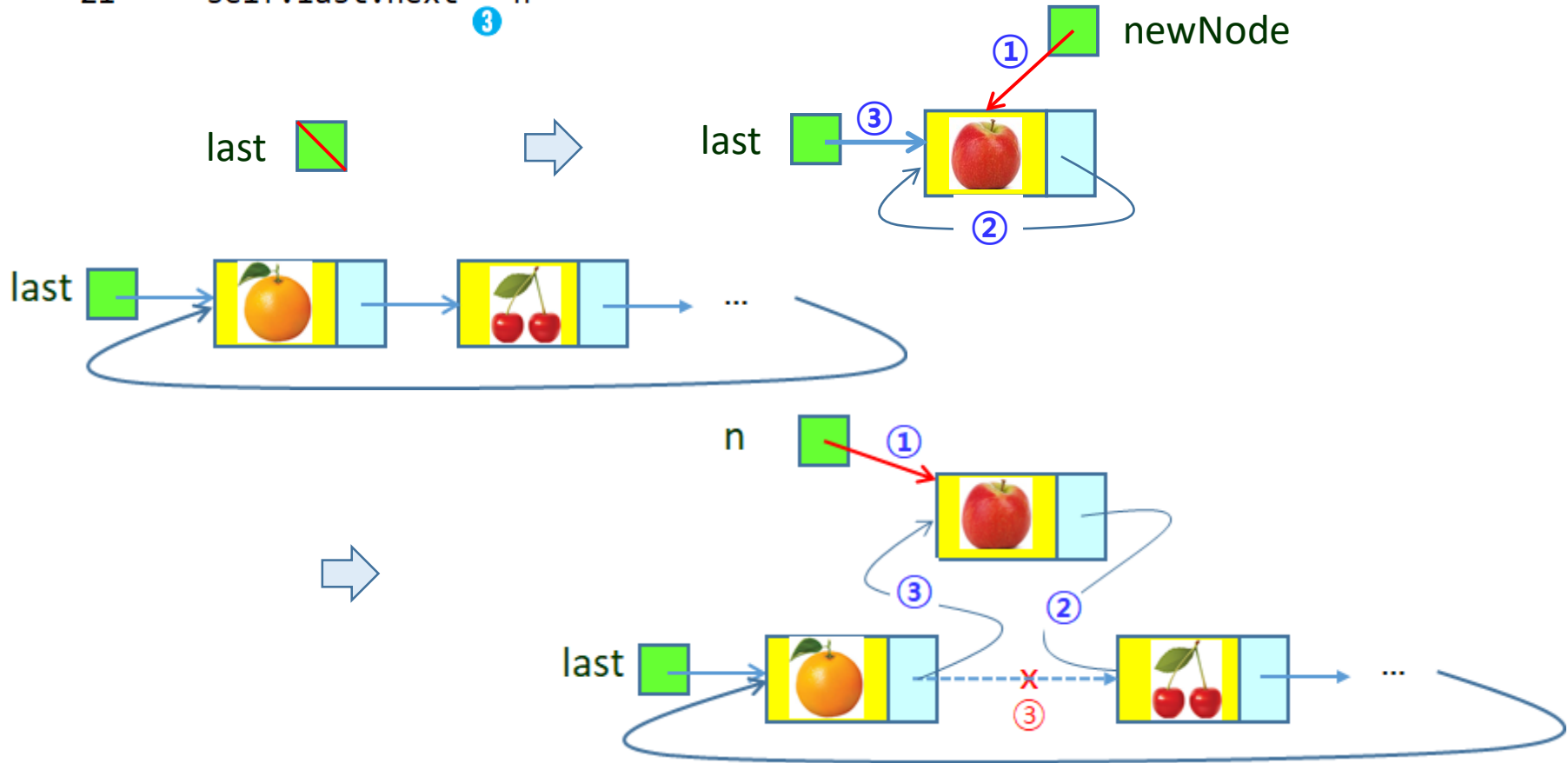
underflow 시 에러 처리

[프로그램 2-5] clist.py



```
15 n = self.Node(item, None)
16 if self.is_empty():
17     n.next = n
18     self.last = n
19 else:
20     n.next = self.last.next
21     self.last.next = n
```

[그림 2-14] insert()의 노드 삽입



```
33 x = self.last.next  
    ①
```

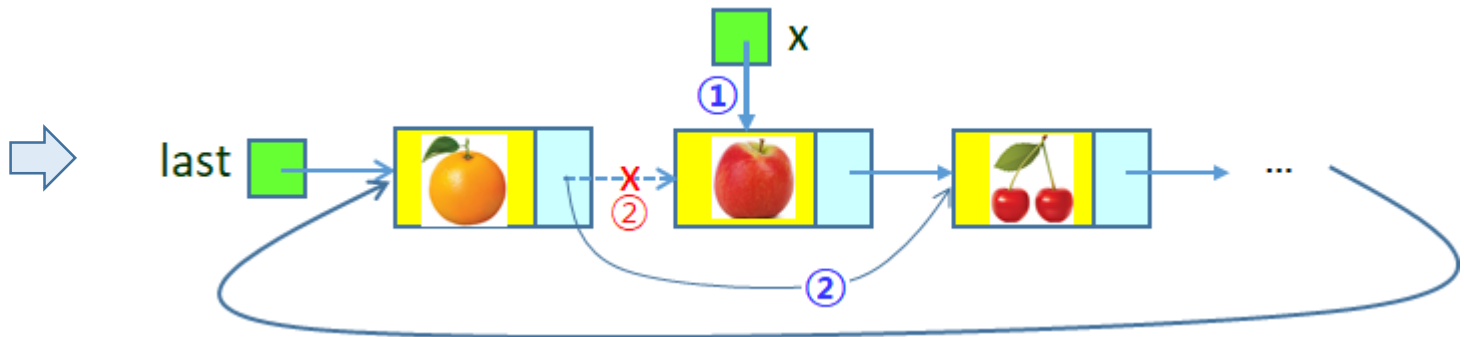
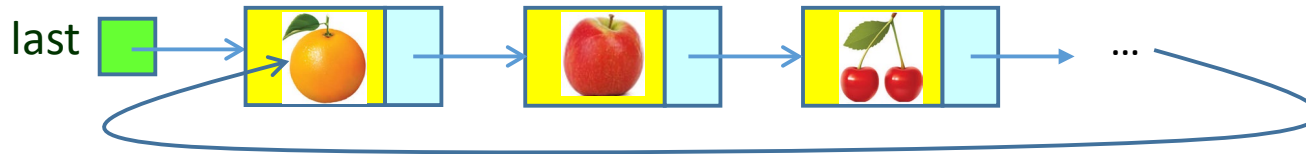
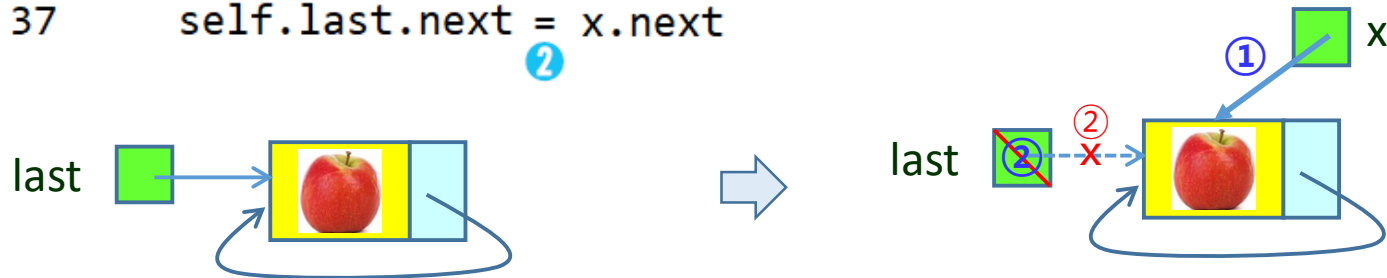
```
34 if self.size == 1:
```

```
35     self.last = None  
    ②
```

```
36 else:
```

```
37     self.last.next = x.next  
    ②
```

[그림 2-15] delete()의 노드 삭제



clist.py에서 CList를 import

이 파이썬 파일(모듈)이 메인이면


원형연결리스트 생성

일련의 삽입 삭제 탐색 연산 수행

```
01 from clist import CList
02 if __name__ == '__main__':
03     s = CList()
04     s.insert('pear')
05     s.insert('cherry')
06     s.insert('orange')
07     s.insert('apple')
08     s.print_list()
09     print('s의 길이 =', s.no_items())
10     print('s의 첫 항목:', s.first())
11     s.delete()
12     print('첫 노드 삭제 후: ', end='')
13     s.print_list()
14     print('s의 길이 =', s.no_items())
15     print('s의 첫 항목:', s.first())
16     s.delete()
17     print('첫 노드 삭제 후: ', end='')
18     s.print_list()
19     s.delete()
20     print('첫 노드 삭제 후: ', end='')
21     s.print_list()
22     s.delete()
23     print('첫 노드 삭제 후: ', end='')
24     s.print_list()
```

[프로그램 2-6] main.py

[프로그램 2-5, 6]의 수행 결과

Console  PyUnit

<terminated> main.py [C:\Users\wsbyang\AppData\Local\Programs\Python\Python36-32

apple -> orange -> cherry -> pear

s의 길이 = 4

s의 첫 항목 : apple

첫 노드 삭제 후: orange -> cherry -> pear

s의 길이 = 3

s의 첫 항목 : orange

첫 노드 삭제 후: cherry -> pear

첫 노드 삭제 후: pear

첫 노드 삭제 후: 리스트 비어있음

Applications

- 여러 사람이 차례로 돌아가며 하는 게임을 구현하는데 적합한 자료구조
- 많은 사용자들이 동시에 사용하는 컴퓨터에서 CPU 시간을 분할하여 작업들에 할당하는 운영체제에 사용
- 이항힙(Binomial Heap)이나 피보나치힙(Fibonacci Heap)과 같은 우선순위큐를 구현하는 데에도 원형연결리스트가 부분적으로 사용

수행시간

- 원형연결리스트에서 삽입이나 삭제 연산 각각 상수 개의 레퍼런스를 갱신하므로 $O(1)$ 시간에 수행
- 탐색 연산: last로부터 노드들을 순차적으로 탐색해야 하므로 $O(N)$ 소요



요약

- **리스트**: 일련의 동일한 타입의 항목들
- **단순연결리스트**: 동적 메모리 할당을 이용해 리스트를 구현하는 가장 간단한 형태의 자료구조
- 단순연결리스트에서는 삽입이나 삭제 시 항목들을 이동시킬 필요 없음
- 단순연결리스트는 항목을 접근하기 위해서 순차탐색을 해야 하고, 삽입이나 삭제할 때에 반드시 이전 노드를 가리키는 레퍼런스를 알아야 함

- 이중연결리스트는 각 노드에 2 개의 레퍼런스를 가지며 각각 이전 노드와 다음 노드를 가리키는 방식의 연결리스트
- 원형연결리스트는 마지막 노드가 첫 노드와 연결된 단순연결리스트
- 원형연결리스트는 마지막 노드와 첫 노드를 $O(1)$ 시간에 방문. 또한 리스트가 empty가 아닐 때, 어떤 노드도 None 레퍼런스를 갖지 않으므로 프로그램에서 None 조건을 검사하지 않아도 되는 장점

최악경우 수행시간 비교

자료구조	접근	탐색	삽입	삭제	비고
단순연결리스트	$O(N)$	$O(N)$	$O(1)^{\dagger}$	$O(1)^{\dagger}$	\dagger 이전 노드의 레퍼런스가 주어진 경우와 첫 노드의 경우
이중연결리스트					
원형연결리스트					