

리스트

-단순, 환형 연결 리스트-

HaRim Jung, Ph.D.

Visiting Professor / Senior Researcher
SKKU Institute for Convergence / Convergence
Research Institute
Sungkyunkwan University, Korea

리스트

□ 리스트(List)

- 유한한(finite) 수의 항목(item)들이 **순서**를 이루어 나열되어 있는 논리적 선형 구조
- 리스트의 (물리적) 구현

– C와 JAVA의 정적 배열(Static Array)

– Python 리스트

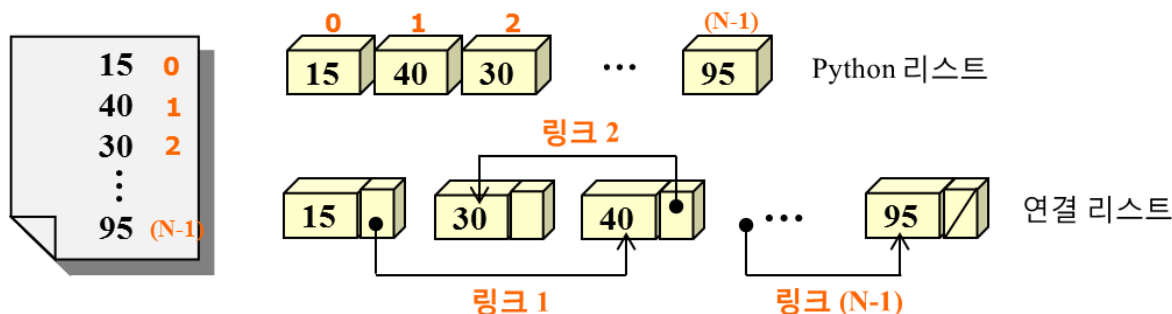
- 리스트의 항목들을 메모리에 **연속적으로(contiguously)** 저장
- JAVA의 ArrayList와 유사, i.e., 동적 배열(Dynamic Array)

정적 배열과 동적 배열의 차이점은?

- 정적 배열은 배열 선언과 동시에 해당 배열의 크기가 결정되고 변경하지 못함
- 동적 배열은 프로그램 실행 중에 메모리가 허용하는 범위 내에서 필요한 만큼 **동적으로** 메모리를 할당 받으므로 크기를 변경시킬 수 있음

– 단순 연결 리스트(Singly Linked List)

- 리스트의 항목들을 (1) 메모리에 분산하여 저장하고, i.e., 메모리에 연속적으로 저장할 필요가 없고, (2) 각 항목은 다음 순서의 항목이 저장된 위치를 가리키는 링크(Link)를 가짐으로써 **순서**를 유지



– 이중 연결 리스트(Doubly Linked List)

– 환형 연결 리스트(Circular Linked List)

단순 연결 리스트 (1/16)

□ 단순 연결 리스트(Singly Linked List)

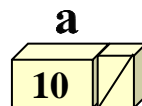
● 단순 연결 리스트를 구성하는 단위: 노드(Node)

- 리스트의 각 항목은 단순 연결 리스트의 기본 단위인 노드(사용자 정의 데이터 타입)에 저장
- 리스트의 각 항목에 대한 노드는 동적으로 메모리를 할당 받으며, 해당 항목을 저장하기 위한 데이터 필드(data field)와 다음 순서 노드의 참조(reference)를 저장하기 위한 링크 필드(link field)로 구성
- 노드 객체를 위한 Node 클래스

```
1 class Node:
2     def __init__(self, item):
3         self.item = item #data field
4         self.next = None #link field
5
6     def get_item(self): return self.item
7
8     def get_next(self): return self.next
9
10    def set_item(self, new_item):
11        self.item = new_item
12
13    def set_next(self, new_next):
14        self.next = new_next
15
16 if __name__ == "__main__":
17     a = Node(10)
18     print(a.get_item())
```

멤버변수 값 호출

멤버변수 값 변경



단순 연결 리스트 (2/16)

□ 단순 연결 리스트(Singly Linked List) contd.

● 단순 연결 리스트란(단순 연결 리스트의 정의)?

- 리스트의 각 항목(item)을 저장하고 있는 **노드**를 링크 필드의 참조를 이용하여 다음 순서의 노드를 가리키도록 만들어서(**NOTE: 마지막 노드의 링크 필드는 None**) 모든 노드들을 한 줄로 연결시킨 자료구조

● 단순 연결 리스트 ADT

왜? 모든 노드들을 한 줄로 연결시켰기 때문에

- 데이터: (1) 순차(sequential) 방식으로만 접근할 수 있는 **노드들의 집합**과 (2) 단순 연결 리스트의 시작을 가리키는(첫 번째 노드의 참조를 저장하는) 변수인 **head**

- 적용 가능한 연산

- **SList()**: 빈 (단순) 연결 리스트 생성
- **is_empty()**: 연결 리스트가 empty면 True 반환
- **add(item)**: 연결 리스트의 처음 위치에 item을 저장하는(**item의 참조를 저장하는**) 새로운 노드 삽입
- **size()**: 연결 리스트의 사이즈 반환

지금부터 설명의 편의를 위해:

- '객체 x를 저장하는'과 '객체 x의 참조를 저장하는'을 같은 의미로 interchangeably 사용
- '변수 y에 객체 z를 할당'과 '변수 y가 객체 z를 참조하게 함'과 '변수 y가 객체 z를 가리키게 함'을 같은 의미로 interchangeably 사용

- **search(item)**: 연결 리스트에 찾고자 하는 item을 저장하고 있는 노드가 존재하면 True를 반환
- **delete(item)**: 연결 리스트에서 특정 item을 저장하고 있는 기존 노드 삭제(삭제 연산 시 반드시 해당 item을 저장하는 노드가 존재한다고 가정)

⋮ **append(item), pop_first(), pop_last()**

단순 연결 리스트 (3/16)

□ 단순 연결 리스트(Singly Linked List) contd.

- 단순 연결 리스트 객체를 위한 클래스 정의

```
1 from node import Node
2 class SList:
3     def __init__(self):
4         self.head = None
5
6     def is_empty(self):
7         return self.head == None
8
9     def add(self, item):
10        temp = Node(item)
11        temp.set_next(self.head)
12        self.head = temp
13
14    def size(self):
15        current = self.head
16        count = 0
17        while current != None:
18            count = count + 1
19            current = current.get_next()
20        return count
```

계속

```
21
22    def search(self, item):
23        current = self.head
24        found = False
25        while current != None and not found:
26            if current.get_item() == item:
27                found = True
28            else:
29                current = current.get_next()
30        return found
31
32    def delete(self, item):
33        current = self.head
34        previous = None
35        found = False
36        while not found:
37            if current.get_data() == item:
38                found = True
39            else:
40                previous = current
41                current = current.get_next()
42        if previous == None:
43            self.head = current.get_next()
44        else:
45            previous.set_next(current.get_next())
```

단순 연결 리스트 (4/16)

□ 단순 연결 리스트(Singly Linked List) contd.

- `SList()` 시간복잡도: $O(1)$

```
3 def __init__(self):  
4     self.head = None
```

– 빈(empty) 연결 리스트를 생성

- 빈 연결 리스트므로 첫 번째 노드의 참조를 저장하는 변수인 `head`를 `None`으로 설정

```
s = SList()
```



- `is_empty()` 시간복잡도: $O(1)$

```
6 def is_empty(self):  
7     return self.head == None
```

– 연결 리스트가 비어 있으면 `True` 반환

```
s = SList()  
s.is_empty()  
True
```

단순 연결 리스트 (5/16)

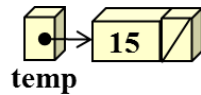
□ 단순 연결 리스트(Singly Linked List) contd.

- add(item) 시간복잡도: $O(1)$

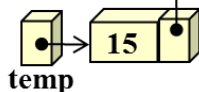
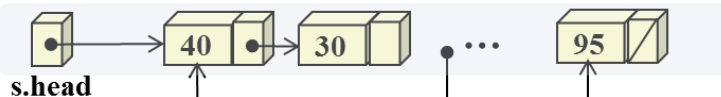
```
9  def add(self, item):  
10     temp = Node(item)  
11     temp.set_next(self.head)  
12     self.head = temp
```

– 연결 리스트의 처음 위치에 item을 저장하는 새로운 노드 삽입

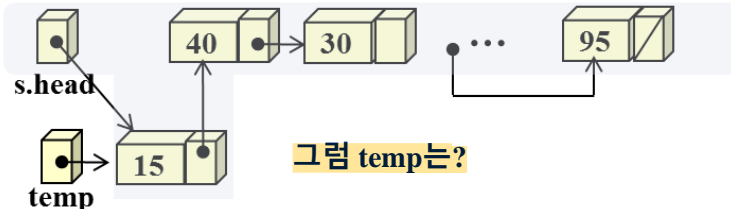
- 라인 10: 인자로 받은 item(e.g., 15)을 저장하는 새로운 노드 N_{new} 를 생성하여 지역변수(local variable) temp에 할당



- 라인 11: temp에 할당된 N_{new} 가 연결 리스트 head가 현재 참조하고 있는 기존 노드를 참조하게 함



- 라인 12: 연결 리스트 head가 temp에 할당된 N_{new} 를 참조하게 함



그럼 temp는?

연결 리스트의 마지막 위치에 item을 저장하는 새로운 노드를 삽입하는 append() 구현은?

순서 중요

단순 연결 리스트 (6/16)

□ 단순 연결 리스트(Singly Linked List) contd.

- add(item) – **인런의 삽입 연산 및 삽입 연산 결과를 확인**

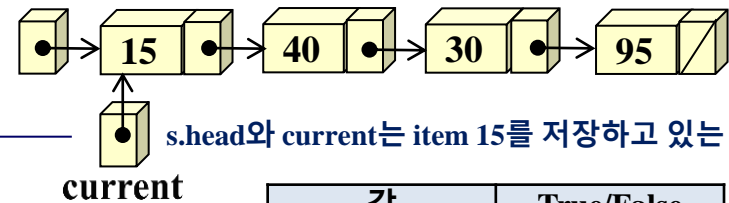
```
if __name__ == "__main__":
    s = SList()
    s.add(95)
    s.add(30)
    s.add(45)
    s.add(15)
```

```
current = s.head
```

```
while current:
    if current.get_next() != None:
        print(current.get_item(), '->', end = '')
    else:
        print(current.get_item())
    current = current.get_next()
```

순회
(traversal)

s.head



값	True/False
"python"	True
""	False
[1, 2, 3]	True
[]	False
1	True
0	False
None	False

- size() 시간복잡도: O(N)

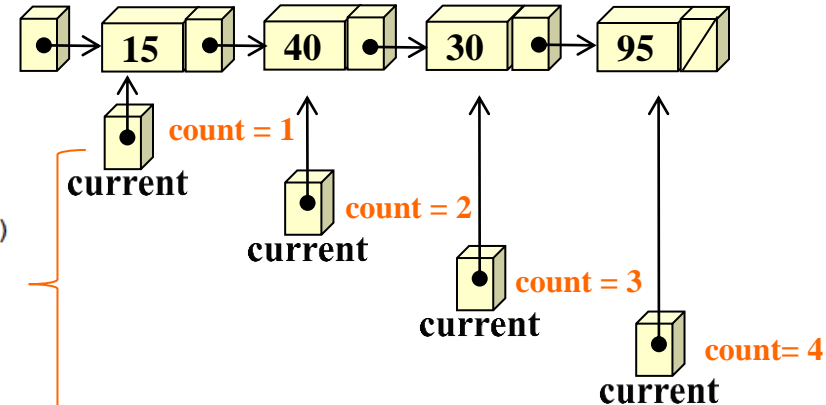
```
14 def size(self):
15     current = self.head
16     count = 0
17     while current != None:
18         count = count + 1
19         current = current.get_next()
20     return count
```

순회

```
s = SList()
s.add(95)
s.add(30)
s.add(45)
s.add(15)
print(s.size())
4
```

순회

s.head



- 연결 리스트의 사이즈(항목의 수) 반환

단순 연결 리스트 (7/16)

□ 단순 연결 리스트(Singly Linked List) contd.

- search(item) 시간복잡도: $O(N)$

```
22 def search(self, item):
23     current = self.head
24     found = False
25     while current != None and not found:
26         if current.get_item() == item:
27             found = True
28         else:
29             current = current.get_next()
30     return found
```

순회

or로 바꾸면?

- 연결 리스트에 찾고자 하는 item을 저장하고 있는 노드가 존재하면 True를 반환
 - 라인 23: 순회를 위한 지역 변수 current에 head가 참조하고 있는 노드 할당
 - 라인 24: 결과(True 혹은 False) 반환을 위한 지역 변수 found 선언하고 False를 할당
 - 라인 25: current가 마지막 노드를 참조하거나 (and) item을 찾을 때까지 while-루프 실행
 - 라인 26-27: if current가 참조하고 있는 노드 N이 저장하는 item이 찾고자 하는 item이라면, found에 True를 할당
 - 라인 28-29: else(찾고자 하는 item이 아니라면), current에 N의 다음 노드를 할당
 - 라인 30: 결과 반환

단순 연결 리스트 (8/16)

□ 단순 연결 리스트(Singly Linked List) contd.

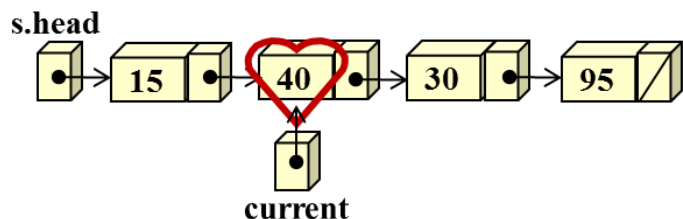
- delete(item) 시간복잡도: $O(N)$

<pre>32 def delete(self, item): 33 current = self.head 34 previous = None 35 found = False 36 while not found: 37 if current.get_item() == item: 38 found = True 39 else: 40 previous = current 41 current = current.get_next()</pre>	<pre>42 if previous == None: 43 self.head = current.get_next() 44 else: 45 previous.set_next(current.get_next())</pre>
---	--

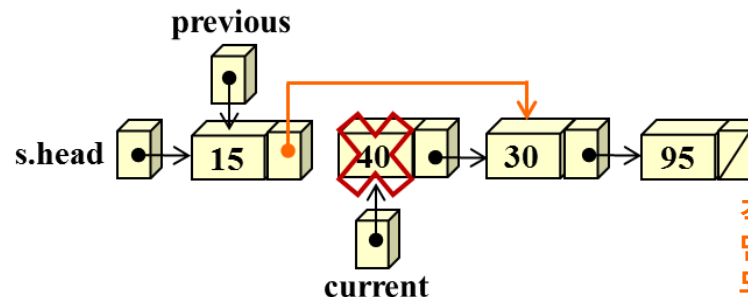
– 연결 리스트에서 item을 저장하고 있는 기존 노드 삭제

- 라인 33: 삭제할 item을 저장하고 있는 노드 N을 찾는 순회를 위한 지역 변수 current에 head가 참조하고 있는 노드 할당
- 라인 34: N 삭제 시 필요한 N 이전 노드를 할당하기 위한 previous 지역 변수 생성 및 초기화 (왜 previous가 필요할까?)

예: 40을 저장하고 있는 기존 노드 삭제



Step 1: 순회를 통해 40을 저장하고 있는 노드 N을 탐색



Step 2: N 이전 노드(15)가 N 다음 노드(30)를 참조하게 함으로써 N 삭제

각 노드가 다음 노드만 가리키는
단순 연결 리스트는 이전 노드로
되돌아갈 방법이 없음

단순 연결 리스트 (9/16)

□ 단순 연결 리스트(Singly Linked List) contd.

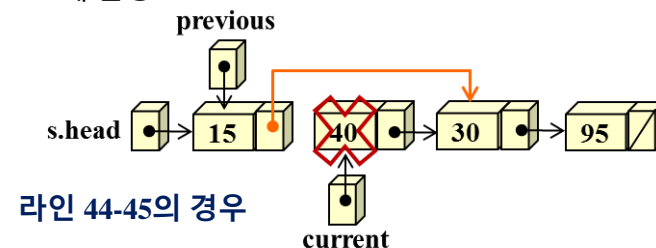
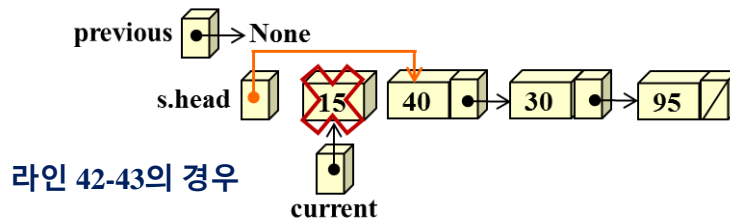
● delete(item) contd.

연결 리스트의 처음 노드를 삭제하는 pop_first() 구현은?
연결 리스트의 마지막 노드를 삭제하는 pop_last() 구현은?

```
32 def delete(self, item):
33     current = self.head
34     previous = None
35     found = False
36     while not found:
37         if current.get_item() == item:
38             found = True
39         else:
40             previous = current
41             current = current.get_next()
42
43     if previous == None:
44         self.head = current.get_next()
45     else:
46         previous.set_next(current.get_next())
```

– 연결 리스트에서 item을 저장하고 있는 기존 노드 삭제

- 라인 35: N 탐색 성공 여부 확인을 위한 found 지역 변수 선언 및 False 할당
- 라인 36: N을 찾을 때까지 while-루프 실행 (**NOTE: 삭제 연산 시 반드시 N이 존재한다고 가정**)
- 라인 37-38: if current가 참조하고 있는 노드 N이 저장하는 item이 삭제하고자 하는 item이라면, found에 True를 할당
- 라인 39-41: else(삭제하고자 하는 item이 아니라면), previous에 N을 할당하고 current에 N의 다음 노드를 할당
- 라인 42-43: if previous에 None이 할당되어 있다면, i.e., current is head, 현재 current가 참조하고 있는 노드 다음 노드를 head에 할당
- 라인 44-45: else, current가 참조하고 있는 노드 다음 노드를 previous에 할당



단순 연결 리스트 (10/16)

□ 정렬된 단순 연결 리스트(Ordered Singly Linked List)

● 정렬된 단순 연결 리스트 ADT

- 데이터: (1) 순차(sequential) 방식으로만 접근할 수 있는 **정렬된 노드들의 집합**과 (2) 단순 연결 리스트의 시작을 가리키는(첫 번째 노드의 참조를 저장하는) 변수인 **head**
- 적용 가능한 연산

단순 연결
리스트와
동일 ←

- **OList()**: (정렬된 단순) 연결 리스트 생성
- **is_empty()**: 연결 리스트가 empty면 True 반환
- **add(item)**: 연결 리스트의 (오름차순) **정렬된 순서에 맞는 위치**에 item을 저장하는 새로운 노드 삽입
- **size()**: 연결 리스트의 사이즈 반환
- **search(item)**: 연결 리스트에 찾고자 하는 item을 저장하고 있는 노드가 존재하면 True를 반환
 - » 배열·파이썬 리스트와 같이 **이진 탐색을 통해 시간복잡도를 $O(\log N)$ 으로 만들 수 있을까?**
- **delete(item)**: 연결 리스트에서 특정 item을 저장하고 있는 기존 노드 삭제(삭제 연산 시 반드시 해당 item을 저장하는 노드가 존재한다고 가정)

단순 연결 리스트 (11/16)

정렬된 단순 연결 리스트(Ordered Singly Linked List) contd.

- add(item) 시간복잡도: $O(N)$

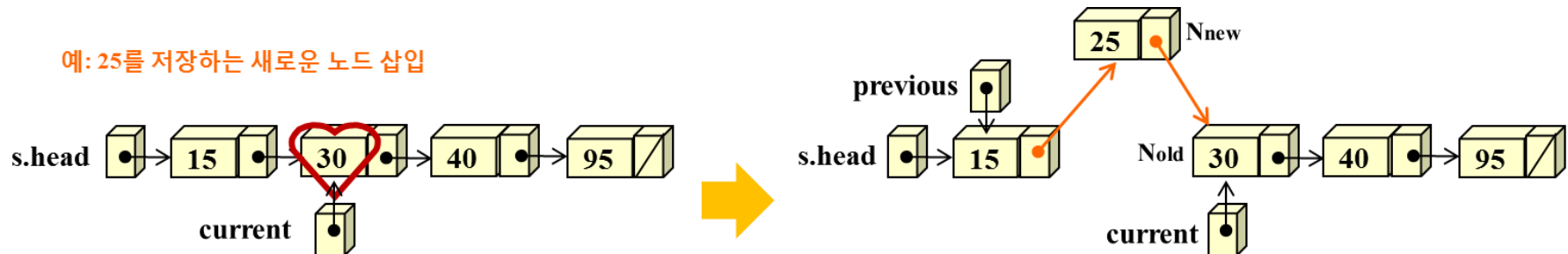
```
1 def add(self, item):
2     current = self.head
3     previous = None
4     stop = False
5     while current != None and not stop:
6         if current.get_item() > item:
7             stop = True
8         else:
9             previous = current
10            current = current.get_next()
```

```
11 temp = Node(item)
12 if previous == None:
13     temp.set_next(self.head)
14     self.head = temp
15 else:
16     temp.set_next(current)
17     previous.set_next(temp)
```

– 연결 리스트의 (오름차순) 정렬된 순서에 맞는 위치에 item을 저장하는 새로운 노드 삽입

- 라인 2: 삽입할 위치에 있는 기존 노드 N_{old} 를 찾는 순회를 위한 지역 변수 $current$ 에 $head$ 가 참조하고 있는 노드 할당
- 라인 3: 새로운 노드 N_{new} 를 삽입 시 필요한 N_{old} 이전 노드를 할당하기 위한 $previous$ 지역 변수 생성 및 초기화

예: 25를 저장하는 새로운 노드 삽입



Step 1: 순회를 통해 새로운 노드 N_{new} 가 삽입되어야 할 위치에 존재하는 기존 노드 N_{old} 를 찾음

Step 2: N_{new} 는 N_{old} 를 참조하게 하고 N_{old} 이전 노드(15)가 N_{new} 를 참조하게 함으로써 삽입

단순 연결 리스트 (12/16)

□ 정렬된 단순 연결 리스트(Ordered Singly Linked List) contd.

● add(item) contd.

<pre>1 def add(self, item): 2 current = self.head 3 previous = None 4 stop = False 5 while current != None and not stop: 6 if current.get_item() > item: 7 stop = True 8 else: 9 previous = current 10 current = current.get_next()</pre>	<pre>11 temp = Node(item) 12 if previous == None: 13 temp.set_next(self.head) 14 self.head = temp 15 else: 16 temp.set_next(current) 17 previous.set_next(temp)</pre>
---	---

- 연결 리스트의 (오름차순) 정렬된 순서에 맞는 위치에 **item**을 저장하는 **새로운 노드** 삽입
 - 라인 4: 삽입할 위치(i.e., 기존 노드 Nold) 탐색 성공 여부 확인을 위한 stop지역 변수 선언 및 False 할당
 - 라인 5: 삽입할 위치를 찾을 때까지 while-루프 실행
 - 라인 6-7: if current가 참조하고 있는 노드 N이 저장하는 item의 값이 삽입하고자 하는 item의 값보다 크다면, i.e., N이 Nold 라면, stop에 True 할당(삽입할 위치 탐색 성공)
 - 라인 8-10: else, previous에 N을 할당하고 current에 N의 다음 노드를 할당
 - 라인 11: 삽입하고자 하는 item을 저장하는 **새로운 노드 Nnew** 생성 후 지역 변수 **temp**에 할당

단순 연결 리스트 (13/16)

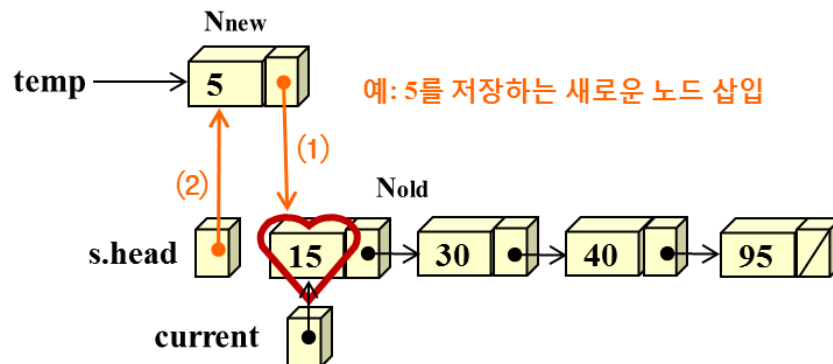
□ 정렬된 단순 연결 리스트(Ordered Singly Linked List) contd.

● add(item) contd.

<pre>1 def add(self, item): 2 current = self.head 3 previous = None 4 stop = False 5 while current != None and not stop: 6 if current.get_item() > item: 7 stop = True 8 else: 9 previous = current 10 current = current.get_next()</pre>	<pre>11 temp = Node(item) 12 if previous == None: 13 temp.set_next(self.head) 14 self.head = temp 15 else: 16 temp.set_next(current) 17 previous.set_next(temp)</pre>
---	---

– 연결 리스트의 (오름차순) 정렬된 순서에 맞는 위치에 item을 저장하는 새로운 노드 삽입

- 라인 12-14: if previous에 None이 할당되어 있다면, current가 참조하고 있는 노드 Nold는 head가 참조하고 있는 노드이므로(즉, None이거나 첫 노드의 item부터 Nnew의 item보다 큰 경우), (1) temp에 할당된 Nnew가 Nold를 참조하게 하고, (2) head는 Nnew를 참조하게 함 (순서 중요)



단순 연결 리스트 (14/16)

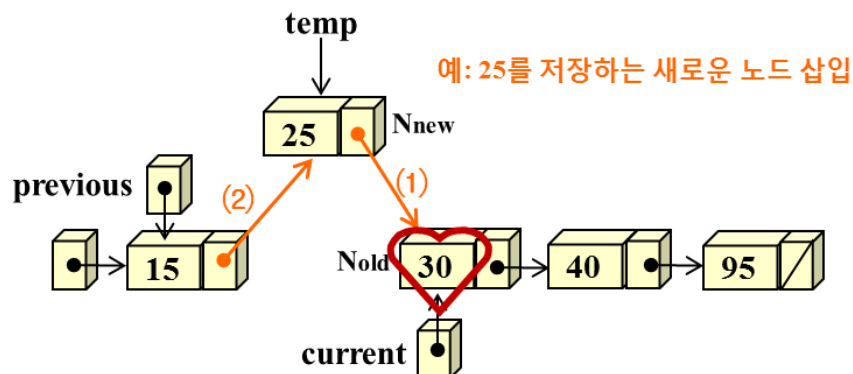
□ 정렬된 단순 연결 리스트(Ordered Singly Linked List) contd.

● add(item) contd.

<pre>1 def add(self, item): 2 current = self.head 3 previous = None 4 stop = False 5 while current != None and not stop: 6 if current.get_item() > item: 7 stop = True 8 else: 9 previous = current 10 current = current.get_next()</pre>	<pre>11 temp = Node(item) 12 if previous == None: 13 temp.set_next(self.head) 14 self.head = temp 15 else: 16 temp.set_next(current) 17 previous.set_next(temp)</pre>
---	---

– 연결 리스트의 (오름차순) 정렬된 순서에 맞는 위치에 item을 저장하는 새로운 노드 삽입

- 라인 15-17: else, (1) temp에 할당된 N_{new} 가 N_{old} 를 참조하게 하고, (2) previous에 할당된 노드(i.e., N_{old} 이전 노드)는 N_{new} 를 참조하게 함



단순 연결 리스트 (15/16)

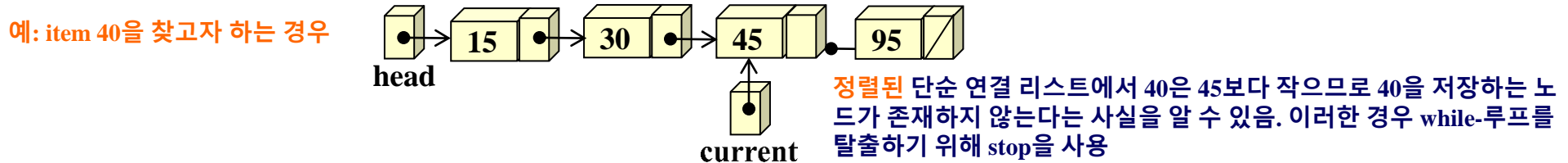
□ 정렬된 단순 연결 리스트(Ordered Singly Linked List) contd.

- search(item) 시간복잡도: $O(N)$

<pre>1 def search(self, item): 2 current = self.head 3 found = False 4 stop = False 5 while current != None and not stop and not found: 6 if current.get_item() == item: 7 found = True 8 else: 9 if current.get_item() > item: 10 stop = True</pre>	<pre>11 else : 12 current = current.get_next() 13 return found</pre>
--	--

– 연결 리스트에 찾고자 하는 item을 저장하고 있는 노드가 존재하면 True를 반환

- 라인 2-3: 단순 연결 리스트 search의 라인 23-24와 동일(슬라이드 9 참조)
- 라인 4: 찾고자 하는 item을 저장하고 있는 노드가 존재하지 않을 때 순회를 종료하기 위해 사용할 지역 변수 stop을 선언하고 False를 할당



- 라인 5: (1) current가 마지막 노드를 참조하거나, (2) 찾고자 하는 item이 존재하지 않는다는 사실을 알게 되거나, (3) item을 찾을 때까지 while-루프 실행

단순 연결 리스트 (16/16)

□ 정렬된 단순 연결 리스트(Ordered Singly Linked List) contd.

● search(item) contd.

<pre>1 def search(self, item): 2 current = self.head 3 found = False 4 stop = False 5 while current != None and not stop and not found: 6 if current.get_item() == item: 7 found = True 8 else: 9 if current.get_item() > item: 10 stop = True</pre>	<pre>11 else : 12 current = current.get_next() 13 return found</pre>
--	--

– 연결 리스트에 찾고자 하는 item을 저장하고 있는 노드가 존재하면 True를 반환

- 라인 6-7: if current가 참조하고 있는 노드 N이 저장하는 item이 찾고자 하는 item이라면, found에 True를 할당
- 라인 8-12: else(찾고자 하는 item이 아니라면), N이 저장하는 item이 찾고자 하는 item보다 크다면 stop에 True를 할당하여 while-루프를 탈출하고(라인 9-10) 아니라면 current에 N의 다음 노드를 할당(라인 11-12)
- 라인 13: 결과 반환

단순 연결 리스트 vs Python 리스트 (1/4)

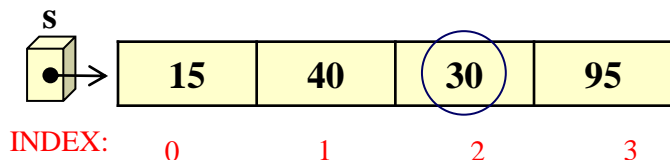
□ 검색 시간 비교

- 단순 연결 리스트는 찾고자 하는 item을 검색하기 위해 순차 접근(i.e., 순차 탐색)을 해야 하므로 $O(N)$ 시간이 필요
- Python 리스트(배열·동적 배열)는 인덱스를 이용하여 $O(1)$ 시간에 검색
- 검색 시간 면에서 Python 리스트(배열·동적 배열)가 유리

과연 항상 그럴까?

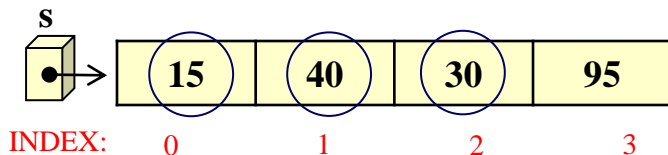
리스트의 세 번째 항목을 검색

Python 리스트: 인덱스를 통해, i.e., $s[2]$, $O(1)$

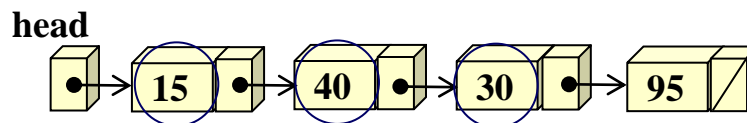


리스트에서 30을 검색

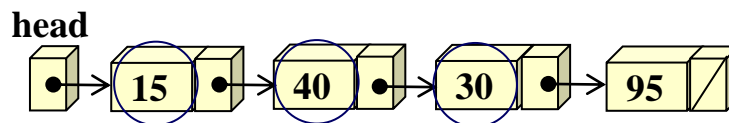
Python 리스트: 인덱스 무의미, 30을 찾을 때까지 순차 접근하므로 $O(N)$



단순 연결 리스트: head로부터 세 번째 노드까지 순차 접근하므로 $O(N)$



단순 연결 리스트: head로부터 30을 저장하는 노드를 찾기 위해 순차 접근하므로 $O(N)$



→ 결론: 검색 시간의 효율성은 **상황에 따라 다를 수 있음** (단, 정렬된 Python 리스트의 검색 성능은 정렬된 단순 연결 리스트의 검색 성능보다 항상 좋음 왜?)

단순 연결 리스트 vs Python 리스트 (2/4)

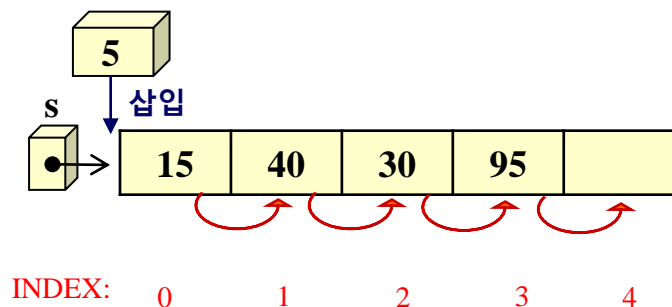
삽입·삭제 시간 비교

- 단순 연결 리스트는 특정 노드의 삽입·삭제 시, 해당 노드와 인접한 노드의 참조만을 변경하면 되므로 $O(1)$ 시간이 필요
- Python 리스트(배열·동적 배열)는 특정 항목의 삽입·삭제 시, 자리 이동을 해야 하므로 $O(N)$ 시간이 필요
- 삽입·삭제 시간 면에서 단순 연결 리스트가 유리

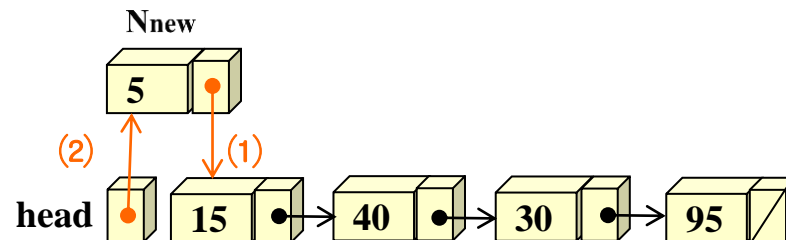
과연 항상 그럴까?

리스트의 처음 위치에 새로운 항목 5를 삽입

Python 리스트: 삽입위치 다음의 항목들을 오른쪽으로 이동시켜야 하므로, $O(N)$



단순 연결 리스트: (1) 새로운 항목 5를 저장하는 노드 N_{new} 가 head가 현재 참조하는 기존 노드를 참조하게 하고, (2) head는 N_{new} 를 참조하게 하는 두 번의 상수 연산이 필요하므로 $O(1)$

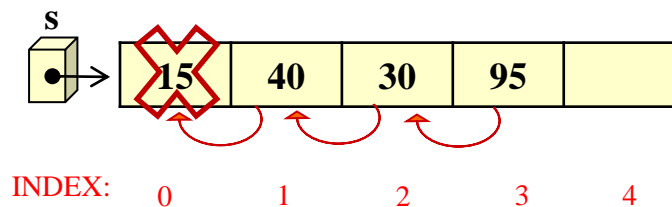


단순 연결 리스트 vs Python 리스트 (3/4)

삽입·삭제 시간 비교 contd.

리스트의 첫 항목 삭제

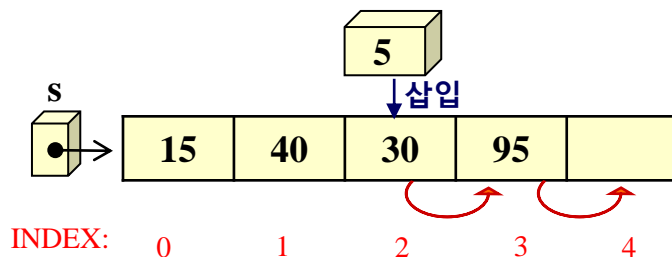
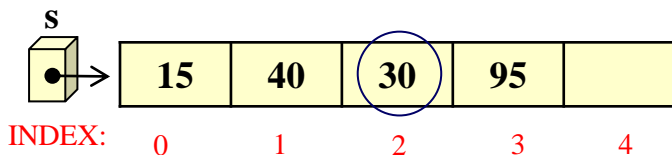
Python 리스트: 삭제위치 다음의 항목들을 왼쪽으로 이동시켜야 하므로, $O(N)$



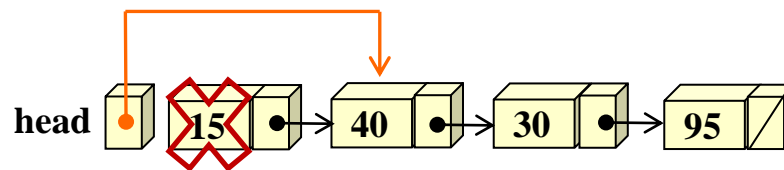
리스트의 세 번째 위치에 새로운 항목 5를 삽입

Python 리스트: $O(N)$

1. 인덱스를 통해 세 번째 위치를 $O(1)$ 시간에 접근
2. 세 번째 위치에 5를 삽입하고, 삽입위치 다음의 항목들을 오른쪽으로 이동시켜야 하므로, $O(N)$

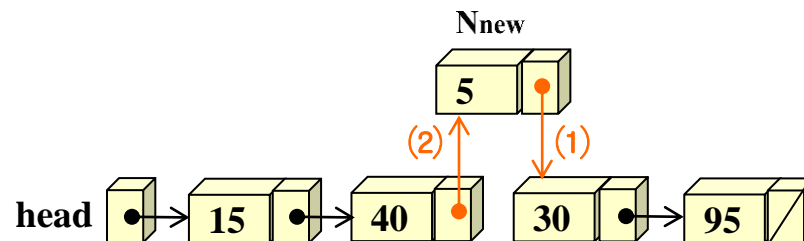
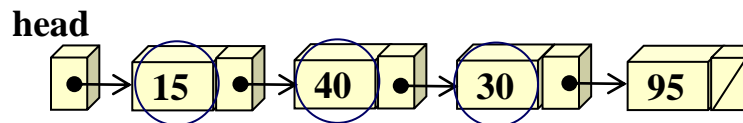


단순 연결 리스트: head를 처음 노드의 다음 노드를 참조하게 하는 상수 연산이 필요하므로, $O(1)$



단순 연결 리스트: $O(N)$

1. head로부터 세 번째 노드 N까지 순차 접근하므로 $O(N)$
2. (1) 새로운 항목 5를 저장하는 노드 N_{new} 가 N을 참조하게 하고, (2) N 이전 노드가 N_{new} 를 참조하게 하는 두 번의 상수 연산이 필요하므로 $O(1)$



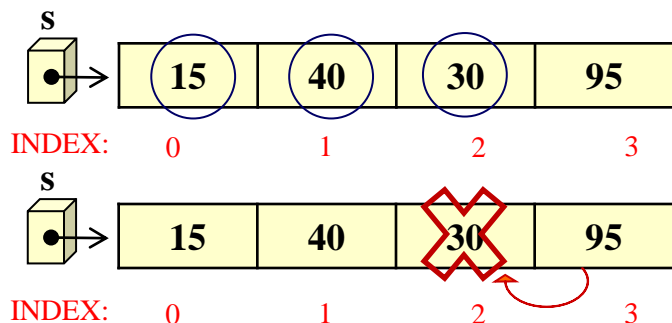
단순 연결 리스트 vs Python 리스트 (4/4)

삽입·삭제 시간 비교 contd.

리스트에서 30 삭제

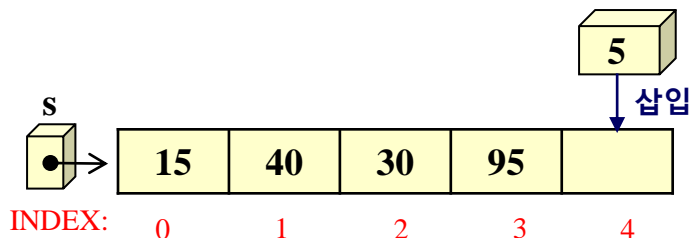
Python 리스트: $O(N)$

1. 인덱스 무의미, 30을 찾을 때까지 순차 접근하므로 $O(N)$
2. 30을 삭제하고, 삭제위치 다음의 항목들을 왼쪽으로 이동시켜야 하므로, $O(N)$



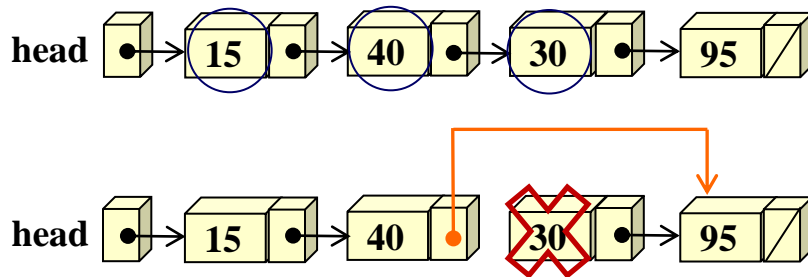
리스트의 마지막 위치에 새로운 항목 5를 삽입

Python 리스트: 자리 이동이 필요 없으므로 $O(1)$



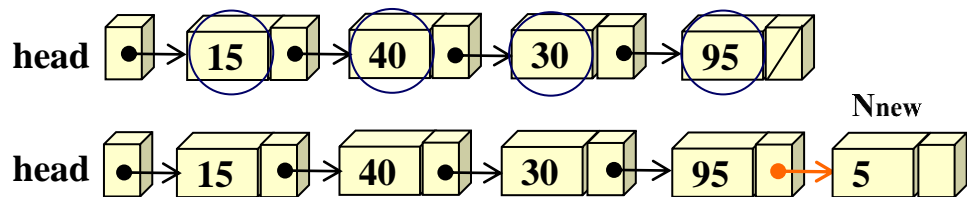
단순 연결 리스트: $O(N)$

1. head로부터 30을 저장하는 노드 N을 찾기 위해 순차 접근하므로 $O(N)$
2. N 이전 노드가 N 다음 노드를 참조하도록 하는 상수 연산이 필요하므로 $O(1)$



단순 연결 리스트: $O(N)$

1. head로부터 마지막 노드 N까지 순차 접근하므로 $O(N)$
2. 새로운 항목 5를 저장하는 노드 N_{new} 를 참조하도록 하는 상수 연산이 필요하므로 $O(1)$



→ 결론: 삽입·삭제 시간의 효율성은 상황에 따라 다를 수 있음

참고: 단순 연결 리스트 append(item), pop_first() , pop_last()

```
def append(self, item):
    new_node = Node(item)
    if self.is_empty():
        self.head = new_node
    else:
        current = self.head
        while current.get_next() != None:
            current = current.get_next()
        current.set_next(new_node)

def pop_first(self):
    self.head = self.head.get_next()

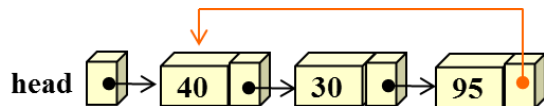
def pop_last(self):
    current = self.head
    previous = None
    while current.get_next() != None:
        previous = current
        current = current.get_next()
    previous.set_next(None)
```

환형 연결 리스트 (1/12)

□ 환형 연결 리스트(Circular Linked List)

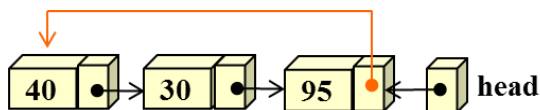
● 환형 연결 리스트란?

- 마지막 노드가 첫 노드와 연결된 단순 연결 리스트(Note: 이중 연결 리스트도 환형 연결 리스트로 변환할 수 있음)



마지막 노드가 첫 노드를 참조, 따라서 next 필드 값이 None인 노드가 존재하지 않음

- 환형이므로 연결 리스트 내 어떤 노드도 첫 노드가 될 수 있으며(i.e., head가 참조하게 할 수 있으며), 주로 head가 마지막 노드를 참조하게 함으로써 첫 노드와 마지막 노드를 $O(1)$ 시간에 접근할 수 있도록 함



- 리스트의 각 항목은 노드(단순 연결 리스트 노드 구조와 동일)에 저장

● 환형 연결 리스트 ADT

- 데이터: (1) 순차 방식으로만 접근할 수 있으며, 마지막 노드가 첫 노드를 참조하는 노드들의 집합과 (2) 환형 연결 리스트의 마지막을 가리키는(마지막 노드의 참조를 저장하는) 변수인 head

- 적용 가능한 연산

- CList(): 빈 (환형) 연결 리스트 생성
- is_empty(): 연결 리스트가 empty면 True 반환
- add(item): 연결 리스트의 처음 위치에 item을 저장하는 새로운 노드 삽입
- append(item): 연결 리스트의 마지막 위치에 item을 저장하는 새로운 노드 삽입
- search(item): 연결 리스트에 찾고자 하는 item을 저장하고 있는 노드가 존재하면 True를 반환
- pop_first(): 연결 리스트의 첫 노드를 삭제

참고: 추상자료형과 자료구조 (1/2)

□ 추상자료형(ADT: Abstract Data Type)

- 사용자의 입장에서 (1) 문제 해결의 대상이 되는 데이터들의 집합 D 와 (2) D 에 대한 연산들을 묶어서 정의해 놓은 것(A specification of (1) a set of data, denoted by D , and (2) the set of operations that can be performed on D)
 - 사용자의 입장에서는 구체적으로 데이터들이 어떻게 논리적으로 나열되고 물리적으로 저장되는지, 구체적으로 어떤 방법으로 연산이 수행되는지 알 바 아니고 알고 싶지도 않음
 - » 예 1: Python 사용자는 Python에 리스트라는 데이터 타입이 존재한다는 사실과 리스트의 사용법만 알고 있으면 됨

□ ADT와 자료구조

- ADT의 구현(implementation)이 자료구조
 - 데이터들에 대한 논리적·물리적 구조는 해당 데이터들에 대한 작업(연산)을 전제로 설계·구현 되어야 함 → 특정 문제 해결에 대한 ADT의 연산이 가장 효율적으로 수행될 수 있도록 데이터들에 대한 논리적 구조를 설계하고 구현하는 것이 자료구조의 역할

참고: 추상자료형과 자료구조 (2/2)

□ ADT와 자료구조 contd.

● 예: 도서관 ADT와 자료구조

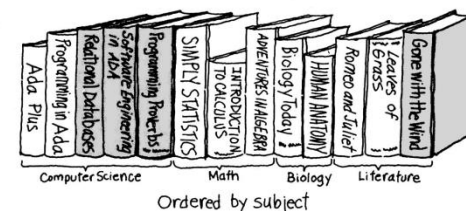
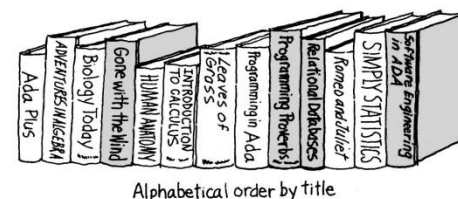
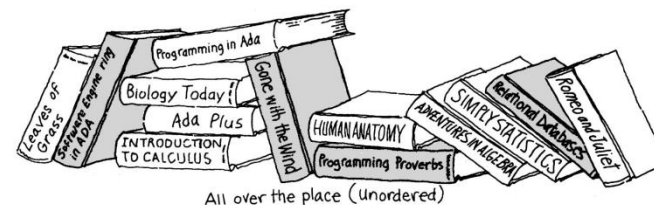
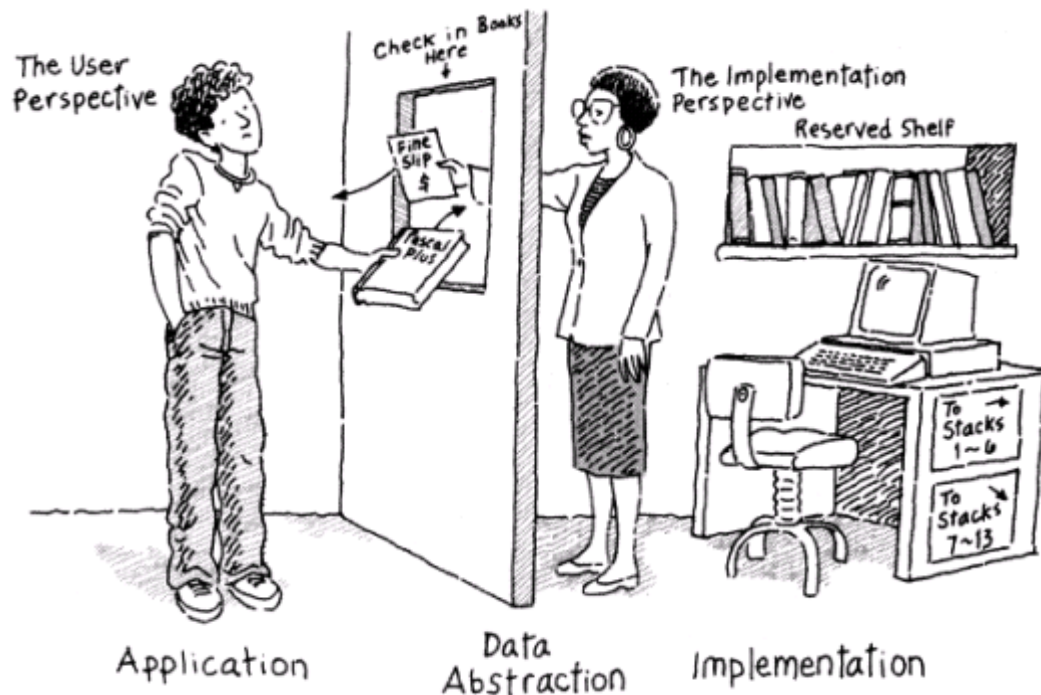
– ADT level:

- Data: a collection of books
- Operations: check a book out, return a book, pay fine, and reserve a book

– Implementation level:

→ 논리적으로 나열 후 코딩을 통해 물리적으로 구현

- Representation of the structure to hold books, and the coding for the above operations



환형 연결 리스트 (2/12)

□ 환형 연결 리스트(Circular Linked List) contd.

● 환형 연결 리스트 객체를 위한 클래스 정의

```
1 from node import Node
2 class CList:
3     def __init__(self):
4         self.head = None # CList의 마지막 노드
5
6     def is_empty(self):
7         return self.head == None
8
9     def add(self, item):
10        temp = Node(item)
11        if self.is_empty():
12            temp.set_next(temp)
13            self.head = temp
14        else:
15            temp.set_next(self.head.get_next())
16            self.head.set_next(temp)
17
18    def append(self, item):
19        temp = Node(item)
20        if self.is_empty():
21            temp.set_next(temp)
22            self.head = temp
23        else:
24            temp.set_next(self.head.get_next())
25            self.head.set_next(temp)
26            self.head = temp
```

계속

```
28
29 def pop_first(self):
30     if self.head == None:
31         print("List is empty.")
32     else:
33         temp = self.head.get_next()
34         if temp == temp.get_next():
35             self.head = None
36         else:
37             self.head.set_next(temp.get_next())
38
39 def search(self, item):
40     if self.head == None:
41         print("List is empty.")
42     else:
43         temp = self.head.get_next()
44         if self.head == temp:
45             if self.head.get_item() == item:
46                 return True
47             else:
48                 return False
49         found = False
50         current = temp
51         while True:
52             if current.get_item() == item:
53                 found = True
54             else:
55                 current = current.get_next()
56                 if current != temp and not found:
57                     continue
58             else:
59                 break
60         return found
```

환형 연결 리스트 (3/12)

□ 환형 연결 리스트(Circular Linked List) contd.


- CList() 시간복잡도: O(1)

```
3 def __init__(self):  
4     self.head = None # CList의 마지막 노드
```

– 빈(empty) 연결 리스트를 생성

- 빈 연결 리스트므로 **마지막 노드의 참조**를 저장하는 변수인 head를 None으로 설정

```
c = CList()  
c.head
```



- is_empty() 시간복잡도: O(1)

```
6 def is_empty(self):  
7     return self.head == None
```

– 연결 리스트가 비어 있으면 True 반환

```
c = CList()  
c.is_empty()  
True
```

환형 연결 리스트 (4/12)

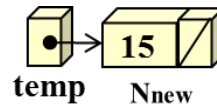
□ 환형 연결 리스트(Circular Linked List) contd.

- add(item) 시간복잡도: O(1)

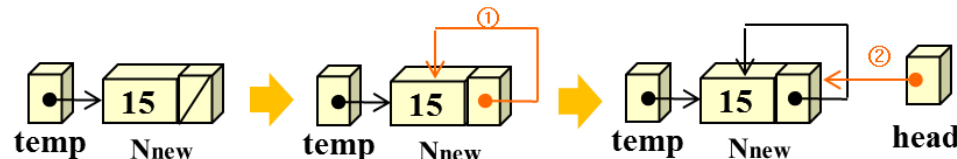
```
9  def add(self, item):
10      temp = Node(item)
11      if self.is_empty():
12          temp.set_next(temp)
13          self.head = temp
14      else:
15          temp.set_next(self.head.get_next())
16          self.head.set_next(temp)
```

– 연결 리스트의 처음 위치에 item을 저장하는 새로운 노드 삽입

- 라인 10: 인자로 받은 item(e.g., 15)을 저장하는 새로운 노드 N_{new} 을 생성하여 지역변수 temp에 할당



- 라인 11-13: if 빈 연결 리스트라면, (1) temp에 할당된 N_{new} 가 자기 자신을 next 필드로 참조하게 한 후, (2) head가 N_{new} 를 참조하게 함



N 이 첫 노드이자 마지막 노드이므로 환형 연결 리스트의 정의에 따라,

(1) 마지막 노드 N 이 첫 노드 N 을 참조하도록 하고(즉, 자기 자신을 참조하도록 하고), (2) head는 마지막 노드 N 을 참조하게 함

환형 연결 리스트 (5/12)

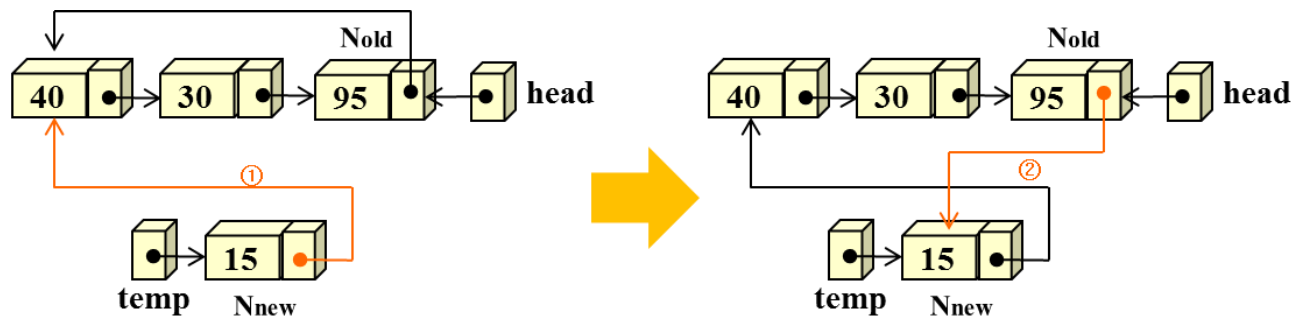
□ 환형 연결 리스트(Circular Linked List) contd.

● add(item) contd.

```
9     def add(self, item):
10         temp = Node(item)
11         if self.is_empty():
12             temp.set_next(temp)
13             self.head = temp
14         else:
15             temp.set_next(self.head.get_next())
16             self.head.set_next(temp)
```

– 연결 리스트의 처음 위치에 item을 저장하는 새로운 노드 삽입

- 라인 14-16: else(빈 연결 리스트가 아니라면), (1) temp에 할당된 N_{new}가 현재 head가 참조하는 노드(현재 마지막 노드) N_{old}의 다음 노드(현재 첫 노드)를 참조하도록 하고, (2) N_{old}가 next 필드로 N_{new}를 참조하도록 함



N_{new}(15) 삽입 후 N_{new}가 첫 노드가 되기 위해서,
(1) N_{new}가 기존 첫 노드(40)를 참조하게 하고, (2) 기존 마지막 노드(95)는 N_{new}를 참조하도록 하는 과정

환형 연결 리스트 (6/12)

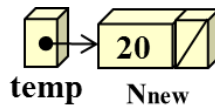
□ 환형 연결 리스트(Circular Linked List) contd.

● append(item) 시간복잡도: O(1)

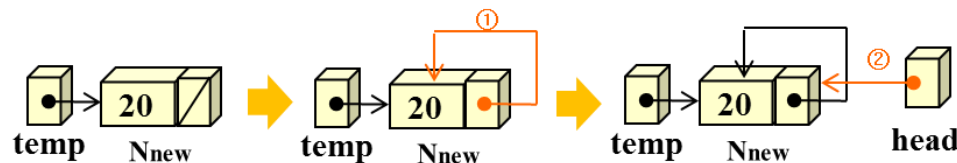
```
18 def append(self, item):
19     temp = Node(item)
20     if self.is_empty():
21         temp.set_next(temp)
22         self.head = temp
23     else:
24         temp.set_next(self.head.get_next())
25         self.head.set_next(temp)
26         self.head = temp
```

– 연결 리스트의 마지막 위치에 item을 저장하는 새로운 노드 삽입

- 라인 19: 인자로 받은 item(e.g., 20)을 저장하는 새로운 노드 N_{new} 를 생성하여 지역변수 temp에 할당



- 라인 20-22: if 빈 연결 리스트라면, (1) temp에 할당된 N_{new} 가 자기 자신을 next 필드로 참조하게 한 후, (2) head가 N_{new} 를 참조하게 함



빈 연결 리스트의 경우 add()와 append()는 동일

환형 연결 리스트 (7/12)

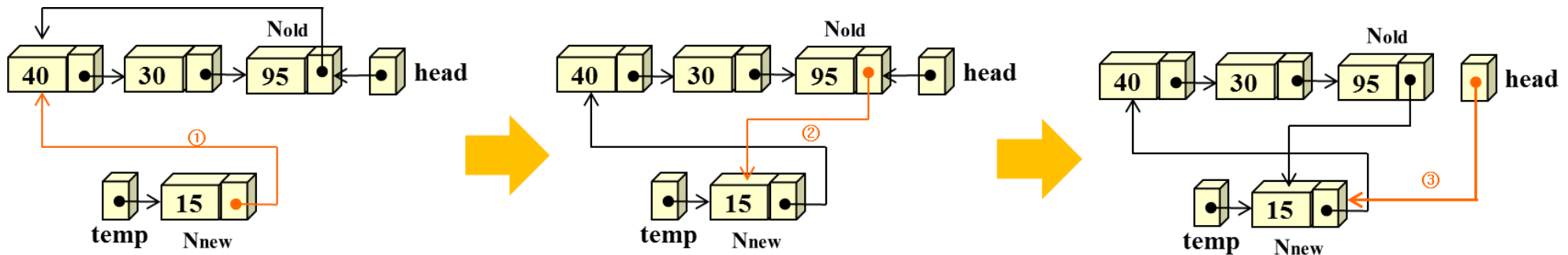
□ 환형 연결 리스트(Circular Linked List) contd.

● append(item) contd.

```
18 def append(self, item):
19     temp = Node(item)
20     if self.is_empty():
21         temp.set_next(temp)
22         self.head = temp
23     else:
24         temp.set_next(self.head.get_next())
25         self.head.set_next(temp)
26         self.head = temp
```

– 연결 리스트의 마지막 위치에 item을 저장하는 새로운 노드 삽입

- 라인 23-26: else(빈 연결 리스트가 아니라면), (1) temp에 할당된 Nnew가 현재 head가 참조하는 노드(현재 마지막 노드) Nold의 다음 노드(현재 첫 노드)를 참조하도록 하고, (2) Nold가 next 필드로 Nnew를 참조하도록 한 후, (3) head가 Nnew를 참조하도록 함



Nnew(15) 삽입 후 Nnew가 마지막 노드가 되기 위해서,

(1) Nnew가 기존 첫 노드(40)를 참조하게 하고, (2) 기존 마지막 노드(95)는 Nnew를 참조하도록 하고(즉, 마지막에서 두 번째에 위치하게 하고), (3) head가 Nnew를 참조하도록 하는 과정

환형 연결 리스트 (8/12)

□ 환형 연결 리스트(Circular Linked List) contd.

● pop_first() 시간복잡도: O(1)

```
28 def pop_first(self):
29     if self.head == None:
30         print("List is empty.")
31     else:
32         temp = self.head.get_next()
33         if temp == temp.get_next():
34             self.head = None
35         else:
36             self.head.set_next(temp.get_next())
```

– 연결 리스트의 첫 노드를 삭제

- 라인 29-30: if 빈 연결 리스트라면, “List is empty.” 출력
- 라인 31: else(빈 연결리스트가 아니라면),
 - » 라인 32: 지역 변수 temp를 선언하고 현재 head가 참조하는 노드(현재 마지막 노드)의 다음 노드(현재 첫 노드) N을 할당
 - » 라인 33-34: if **N이 첫 노드이자 마지막 노드라면**(즉, 연결 리스트에 N만 존재한다면), head에 None 할당



- » 라인 35-36: else, 마지막 노드가 N 다음 노드를 참조하게 함



환형 연결 리스트 (9/12)

□ 환형 연결 리스트(Circular Linked List) contd.

- search(item) 시간복잡도: $O(N)$

```
38 def search(self, item):
39     if self.head == None:
40         print("List is empty.")
41     else:
42         temp = self.head.get_next()
43         if temp == temp.get_next():
44             if temp.get_item() == item:
45                 return True
46             else:
47                 return False
48         found = False
49         current = temp
```

```
50 while True:
51     if current.get_item() == item:
52         found = True
53     else:
54         current = current.get_next()
55     if current != temp and not found:
56         continue
57     else:
58         break
59     return found
```

– 연결 리스트에 찾고자 하는 item을 저장하고 있는 노드가 존재하면 True를 반환

- 라인 39-40: if 빈 연결 리스트라면, “List is empty.” 출력

- 라인 41-59: else(빈 연결 리스트가 아니라면),

» 라인 42: 무한 루프 발생을 방지하기 위한 지역 변수 temp에 head가 참조하고 있는 노드의 다음 노드(즉, 첫 노드) N을 할당

환형 연결 리스트는 next 필드에 None 값을 가지는 노드가 없으므로 단순 연결 리스트의 종료 조건 (while current != None and not found:) 과 동일하게 작성할 경우, 찾고자 하는 item이 없다면 무한 루프가 발생

» 라인 43-47: if N이 첫 노드이자 마지막 노드라면(즉, 연결 리스트에 N만 존재한다면), N이 저장하는 item이 찾고자 하는 item일 경우 True를 반환 및 메소드를 종료하고(라인 44-45), (2) 찾고자 하는 item이 아닐 경우 False를 반환 및 메소드를 종료(라인 46-47)

환형 연결 리스트 (10/12)

□ 환형 연결 리스트(Circular Linked List) contd.

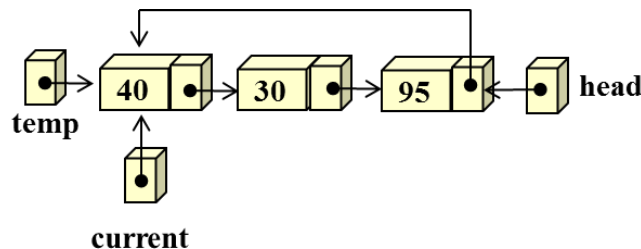
● search(item) contd.

```
38 def search(self, item):
39     if self.head == None:
40         print("List is empty.")
41     else:
42         temp = self.head.get_next()
43         if temp == temp.get_next():
44             if temp.get_item() == item:
45                 return True
46             else:
47                 return False
48         found = False
49         current = temp
```

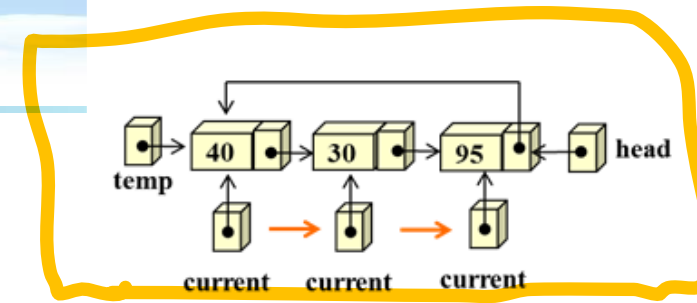
```
50
51     while True:
52         if current.get_item() == item:
53             found = True
54         else:
55             current = current.get_next()
56         if current != temp and not found:
57             continue
58         else:
59             break
    return found
```

– 연결 리스트에 찾고자 하는 item을 저장하고 있는 노드가 존재하면 True를 반환

- 라인 48: 결과(True 혹은 False) 반환을 위한 지역 변수 found 선언하고 False를 할당
- 라인 49: 순회를 위한 지역 변수 current에 temp에 할당된 노드(즉, 첫 노드) 할당



환형 연결 리스트 (11/12)



□ 환형 연결 리스트(Circular Linked List) contd.

● search(item) contd.

```
38 def search(self, item):
39     if self.head == None:
40         print("List is empty.")
41     else:
42         temp = self.head.get_next()
43         if temp == temp.get_next():
44             if temp.get_item() == item:
45                 return True
46             else:
47                 return False
48         found = False
49         current = temp
```

```
50
51 while True:
52     if current.get_item() == item:
53         found = True
54     else:
55         current = current.get_next()
56         if current != temp and not found:
57             continue
58         else:
59             break
return found
```

- 연결 리스트에 찾고자 하는 item을 저장하고 있는 노드가 존재하면 True를 반환

```
while current != temp and not found:
    if current.get_item() == item:
        found = True
    else:
        current = current.get_next()
return found
```

```
while current.get_next() != temp and not found:
    if current.get_item() == item:
        found = True
    else:
        current = current.get_next()
return found
```

- 라인 51-52: if current가 참조하고 있는 노드 N이 저장하는 item이 찾고자 하는 item이라면, found에 True를 할당
- 라인 53-54: else(찾고자 하는 item이 아니라면), current에 N의 다음 노드를 할당

환형 연결 리스트 (12/12)

□ 환형 연결 리스트(Circular Linked List) contd.

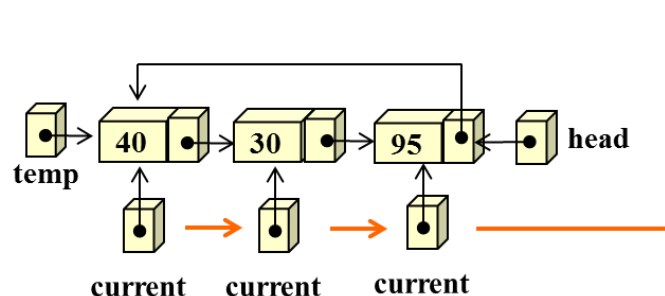
● search(item) contd.

```
38 def search(self, item):
39     if self.head == None:
40         print("List is empty.")
41     else:
42         temp = self.head.get_next()
43         if temp == temp.get_next():
44             if temp.get_item() == item:
45                 return True
46             else:
47                 return False
48         found = False
49         current = temp
```

```
50
51     while True:
52         if current.get_item() == item:
53             found = True
54         else:
55             current = current.get_next()
56         if current != temp and not found:
57             continue
58         else:
59             break
    return found
```

– 연결 리스트에 찾고자 하는 item을 저장하고 있는 노드가 존재하면 True를 반환

- 라인 55-56: while-루프 탈출 조건 검사: if current가 한 번의 리스트 순회를 마치지 않았고 item을 찾지도 못했다면, continue
- 라인 57-58: while-루프 탈출 조건 검사: else(current가 한 번의 리스트 순회를 마쳤거나 item을 찾았다면), break를 통해 while-루프 탈출



- 라인 59: 결과 반환