7. Linked Lists

2024학년도 가을학기

정보컴퓨터공학부 황원주 교수

부산대학교
PUSAN NATIONAL UNIVERSITY

강의내용

- 한방향 연결리스트(Singly linked list)
  - 한방향 연결리스트
  - 한방향 연결리스트 ADT의 구현
  - 시간복잡도
  - 순회(traversal)
- 양방향 연결리스트(Doubly linked list)
  - 양방향 연결리스트
  - 양방향 연결리스트 ADT의 구현
  - 시간복잡도
- 원형 연결리스트(Circular linked list)
  - 원형 연결리스트
  - 원형 연결리스트 ADT의 구현
  - 시간복잡도

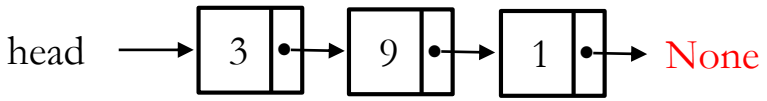
필요성

- 연결리스트는 1955-1956년에 RAND Corporation의 Allen Newell, Cliff Shaw 및 Herbert A. Simon이 Information Processing Language(IPL)의 기본 자료구조로 개발됨
  - **배열의 삽입/삭제 시 시간복잡도가 큰 것을 해소하기 위해 일부러 연결리스트를 만들었다기보다는** IPL을 개발하던 중 자연스럽게 개발된 것 같음
  - 왜냐면 초기에는 메모리의 용량이 지금처럼 크지 않았을 것이므로 메모리의 연속적인 공간에 데이터를 저장하는 것이 부담스러웠을 수도 있음 (뇌피셜). 즉, 양산캠에는 부지가 커서 큰 건물 한 채를 지을 수 있지만, 장전캠은 작은 건물을 여러 채 지을 수 밖에 없음
- IPL은 Logic Theory Machine, General Problem Solver 및 컴퓨터 체스 프로그램을 포함한 여러 초기 인공 지능 프로그램을 개발하는 데 사용됨

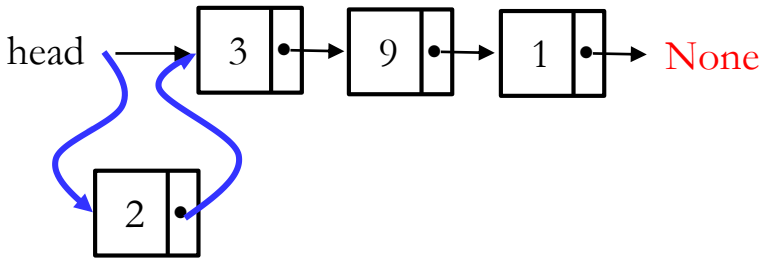
RAND Corporation 본사
(미국 Santa Monica)

Allen Newell
(1927-1992)

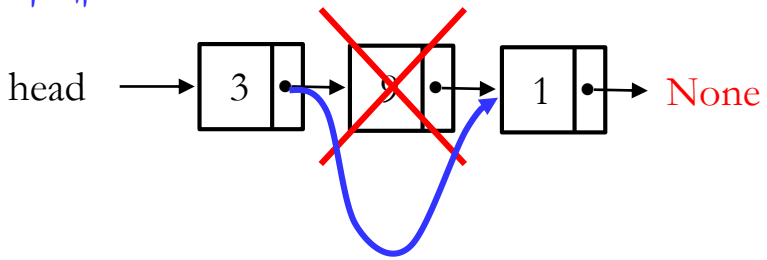
- 연결리스트는 배열에 비해 삽입과 삭제의 시간복잡도가 작다.
  - 배열, 리스트: 삽입(O(n)), 삭제(O(n))
  - 연결리스트: 삽입(O(1)), 삭제(O(1))



삽입



삭제



그러나…

응용

- 한방향 연결리스트는 널리 사용
  - 스택과 큐 자료구조에 사용할 수도 있음
  - 해싱의 체이닝(Chaining)에 사용할 수도 있음
  - 트리도 한방향 연결리스트의 개념을 확장시킨 자료구조

한방향 연결리스트
(Singly linked lists)

한방향 연결리스트

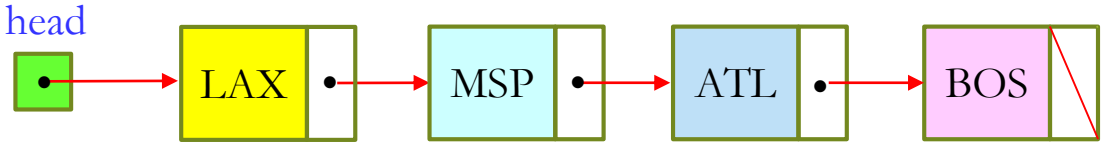
- 정의

  - 노드들이 한쪽 방향으로만(next link를 따라) 연결된 리스트

    노드(node): 실제 값을 위한 data 정보와 다음 노드(next)를 가리키는 link 정보(reference)로 구성된 클래스

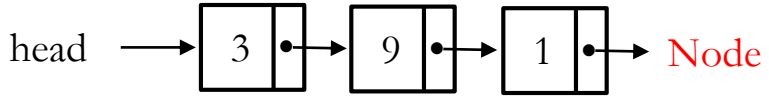
    가장 앞에 있는 노드를 특별히 head 노드라 부르고, head 노드를 통해 리스트의 노드를 접근함 (한방향 연결리스트에서는 head 노드부터 시작해 link를 계속 따라가면 모든 노드를 접근 할 수 있으므로, head 노드가 한방향 연결리스트를 대표한다고 말할 수 있음)

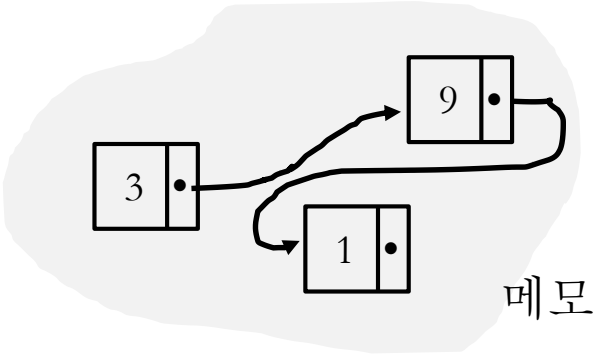
    가장 뒤에 있는 노드는 다음 노드가 없기 때문에 그 노드의 next link는 None을 저장함. 즉, next link가 None이면 그 노드가 마지막 노드라는 의미



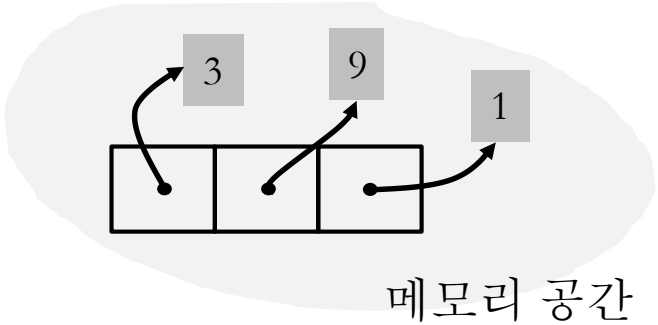
리스트 vs. 연결리스트 (특징)

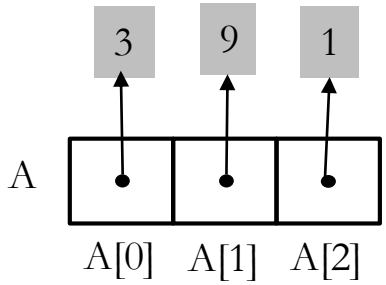
- 연결리스트(linked list)는 파이썬의 리스트(list)와는 이름만 비슷하지 다른 개념이다
  - 연결리스트는 노드(node)가 링크(link)에 의해 기차처럼 연결된 순차 (sequential) 자료 구조로 헤드노드에서 부터 link를 따라 원하는 노드의 데이터를 접근하고 수정한다





  - 배열 또는 리스트는 데이터를 연속적인 메모리 공간에 저장하고, 각 요소(element)는 인덱스를 이용하여 임의 접근한다






```
A=[3, 9, 1]
```

리스트 vs. 연결리스트 (비교)

- 리스트 vs 연결리스트

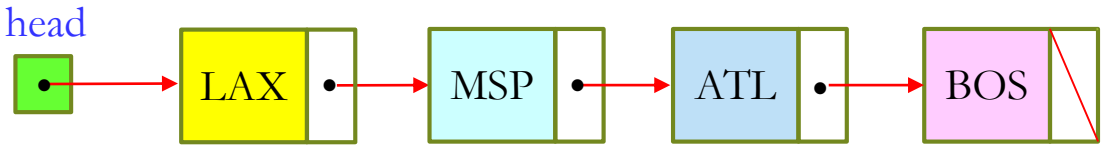
[장점]

- 연결리스트에서는 삽입이나 삭제 시 항목들의 이동이 필요 없음
- 연결리스트에서는 next link를 자유롭게 가져갈 수 있음

[단점]

- 연결리스트에서는 항목을 탐색하려면 항상 첫 노드부터 원하는 노드를 찾을 때까지 차례로 방문하는 순차탐색(sequential search)을 해야 함 (random access를 제공하지 않음!)

예

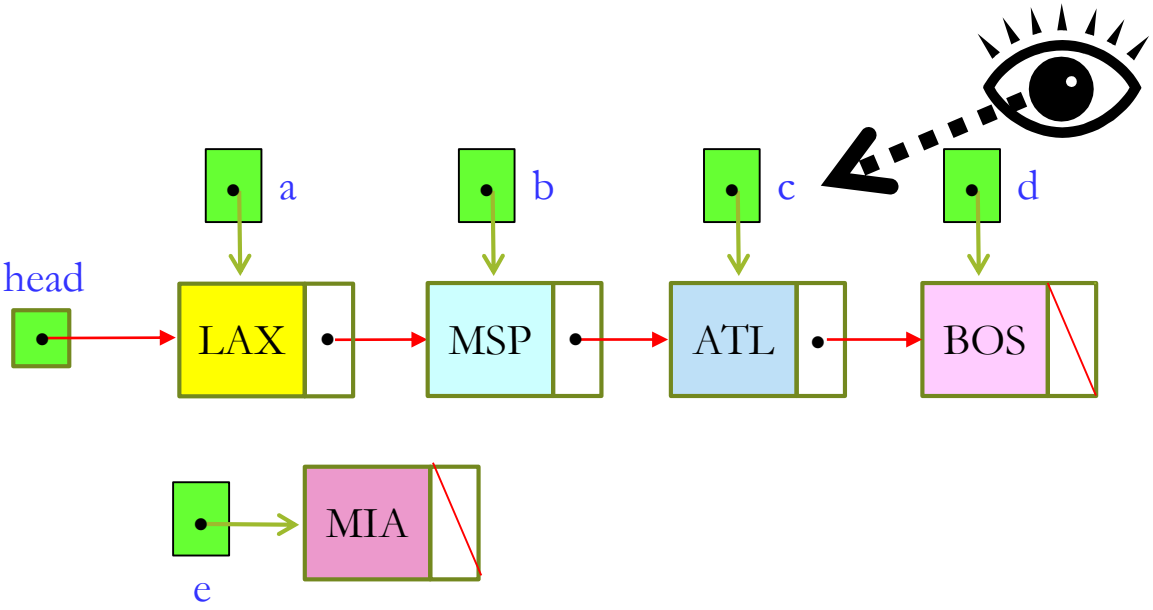


- 한방향 연결리스트 클래스와 노드 클래스 정의

```
class SList:
  class Node:
    __slots__ = "element", "next"
    def __init__(self, element, next):
      self.element = element
      self.next = next

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

- 예와 같은 한방향 연결리스트 만들기

```
s = SList()

a=s.Node('LAX', None)
b=s.Node('MSP', None)
c=s.Node('ATL', None)
d=s.Node('BOS', None)

s.head = a
a.next = b
b.next = c
c.next = d
```



access할 때 개별적으로 변수를 정의해서 접근하지 않음; head노드로 부터 접근함

- 읽기: `print(s.head.next.next.element)` #ATL 읽기
- 쓰기: `s.head.next.next.element = 'SFO'` #ATL에 SFO 쓰기
- 삽입: `e = s.Node('MIA', None)` #LAX 다음에 MIA 삽입

  `e.next=a.next`

  `a.next=e`
- 삭제: `e.next=b.next` #MSP 삭제
- 탐색: #반복의 방법으로 순회, 교과서에서는 search()와 같이 연결리스트의 크기로 순회하는 방법과, print_list()와 같이 마지막 노드의 link가 None인 것을 이용하는 방법을 사용하여 순회

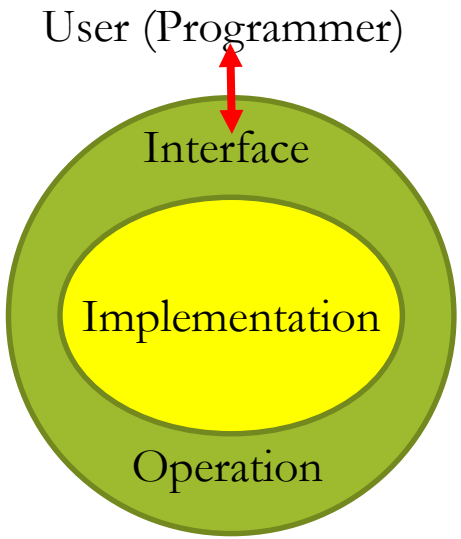
```
class SList:
  class Node:
    __slots__ = "element", "next"
    def __init__(self, element, next):
      self.element = element
      self.next = next

  def __init__(self):
    self.head = None

  def print_list(self):
    p = self.head
    while p:
      if p.next != None:
        print(p.element, ' -> ', end='')
      else:
        print(p.element)
      p = p.next
```

```
출력:
LAX  -> MSP  -> ATL  -> BOS
ATL
LAX  -> MSP  -> SFO  -> BOS
LAX  -> MIA  -> MSP  -> SFO  -> BOS
LAX  -> MIA  -> SFO  -> BOS
```

```
s = SList()

a=s.Node('LAX', None)
b=s.Node('MSP', None)
c=s.Node('ATL', None)
d=s.Node('BOS', None)

s.head = a
a.next = b
b.next = c
c.next = d

# 현제 만들어진 한방향 연결리스트
s.print_list()

# ATL 읽기
print(s.head.next.next.element)

# ATL에 SFO 쓰기
s.head.next.next.element = 'SFO'
s.print_list()

# LAX 뒤에 MIA 삽입
e = s.Node('MIA', None)
e.next = a.next
a.next = e
s.print_list()

# MSP 삭제
e.next = b.next
s.print_list()
```

추상데이터타입

- 추상데이터타입(ADT; Abstract Data Type)
  - '추상' 의 의미는 연산을 어떻게 구현해야한다는 구체적인 내용은 포함하지 않고, 사용자에게 interface(함수명, 반환값, 매개변수)와 operation(연산)만 제공. 건축설계도에 해당
  - 자료구조: ADT를 구체적으로, 즉 실제 프로그램으로 구현한 것을 의미. 실제 건물에 해당 (十人十色)



- 왜 ADT를 사용하나?
  - 자료구조를 구현할 때는 실제 저장되는 데이터를 처리하기 위한 interface와 operation을 사전에 반드시 정의해야 함
  - 대부분의 자료구조들은 읽기, 쓰기, 삽입, 삭제, 탐색을 기본 연산으로 지원하며, 자료구조에 따라 보조 연산들이 추가됨. 우리 교과서에서는 ADT의 정의가 간단 명료하기 때문에 생략되었을 뿐!!!
  - 협업 시 실무에서도 효율적인 업무 분담과 모듈화가 가능해져 널리 사용

미국 미시간대학의 연결리스트 ADT의 예

(Linked) List ADT

Operations

- size(): return number of nodes in list
- isempty(): return T if list is empty, F otherwise
- 읽기 – elemAtPos(p): return the element at position p
- 쓰기 – replaceAtPos(p,e): replace the element at position p with element e
- 삽입 – insertAfter(p,e): insert a new element e after position p
- 삽입 – insertBefore(p,e): insert a new element e before position p
- 삭제 – removeAt(p): remove the element at position p

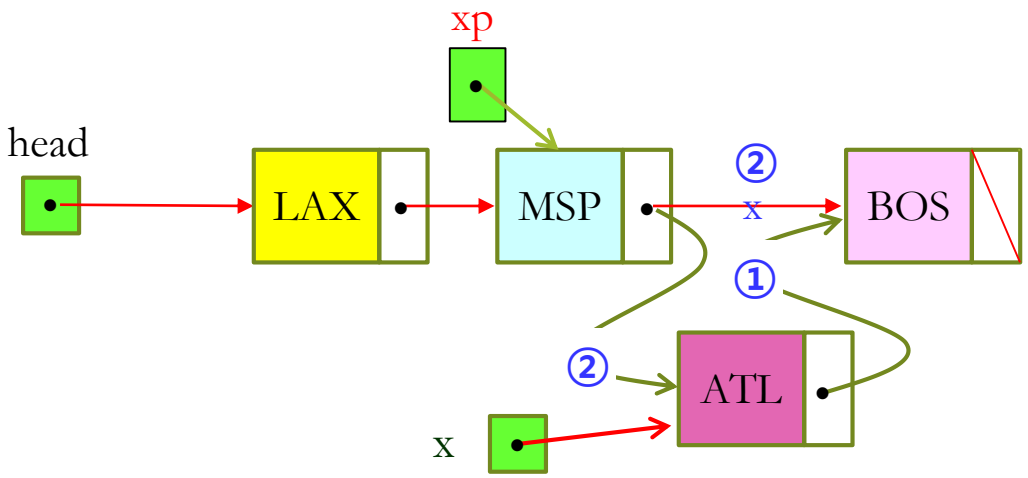
[출처: 미시간대학, https://www.eecs.umich.edu/courses/eecs281/f04/lecnotes/03-Array%20&%20List%20v2.pdf]

한방향 연결리스트 ADT

- **읽기**
  - SL.print_list(): 한방향 연결리스트 SL의 모든 노드 출력
- **삽입**:
  - SL.insert_front(item): 맨 앞에 노드(item을 가진)를 삽입
  - SL.insert_after(item, p): p의 다음에 노드(item을 가진)를 삽입
- **삭제**:
  - SL.delete_front(): 맨 앞에 노드를 삭제
  - SL.delete_after(p): p의 다음 노드를 삭제
- **탐색**:
  - SL.search(target): target을 탐색하여 성공하면 해당 위치를, 실패하면 None을 반환
- **보조 연산**:
  - SL.size(): 한방향 연결리스트 SL의 크기 반환
  - SL.is_empty(): 한방향 연결리스트 SL이 비었는지 체크

한방향 연결리스트 ADT의 구현

- 한방향 연결리스트에서 삽입의 관찰:



1. [중요] 전 노드의 link를 수정해야 하므로 삽입할 노드 x의 이전 노드(previous node)를 xp에 저장한다. 왜냐하면, 한방향 연결리스트에서는 이전 노드로 갈 수 있는 방법이 없기 때문

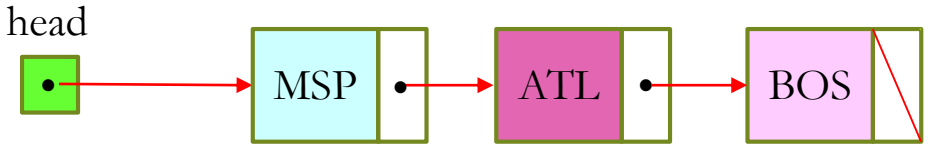
   - 맨 앞에 삽입하는 insert_front() 함수는 head가 삽입된 새 노드를 가리키면 되므로 (삽입된 노드가 새로운 head 노드가 되면 되므로), 즉 head 노드가 이전 노드이므로 별도의 동작이 불필요

   - 임의의 노드 다음에 삽입하는 insert_after() 함수는 이전 노드를 찾아서 저장해 두는 동작이 필요

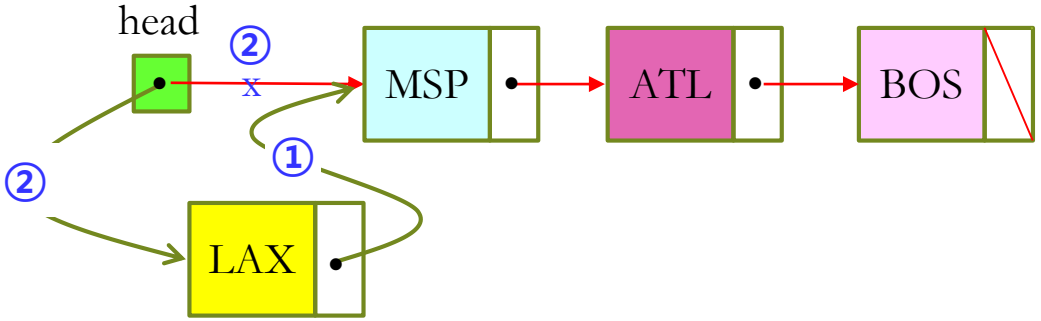
2. 삽입할 새 노드 x의 link를 x.next=xp.next로 갱신(①) 한다.

3. 이전 노드 xp의 link를 xp.next=x로 갱신(②) 한다.

• insert_front(LAX)의 삽입 수행: 맨 앞에 노드 삽입



(a) 새 노드 삽입 전

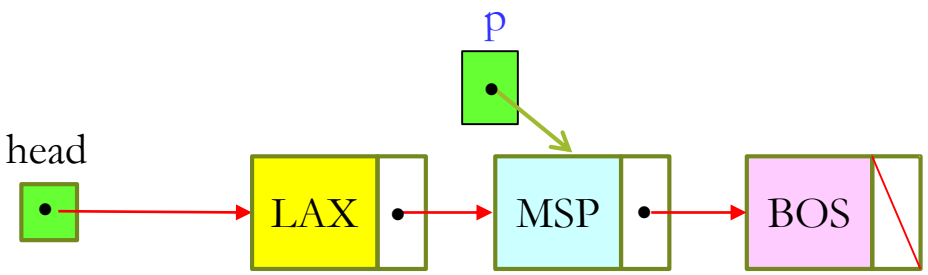


(b) 새 노드 삽입 후

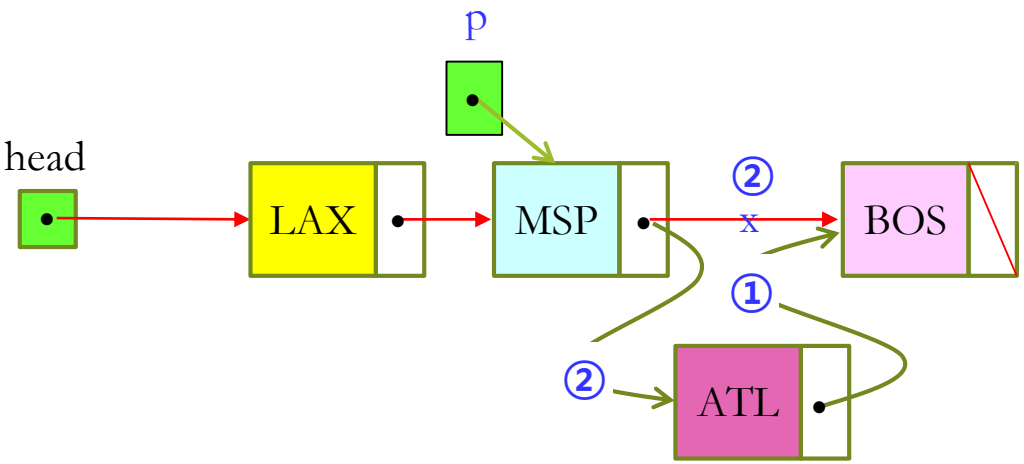
```
self.head = self.Node(item, self.head)
```

② LAX ①

- insert_after(ATL, p)의 삽입 수행: 인자인 p가 가리키는 노드의 다음에 삽입



(a) 새 노드 삽입 전

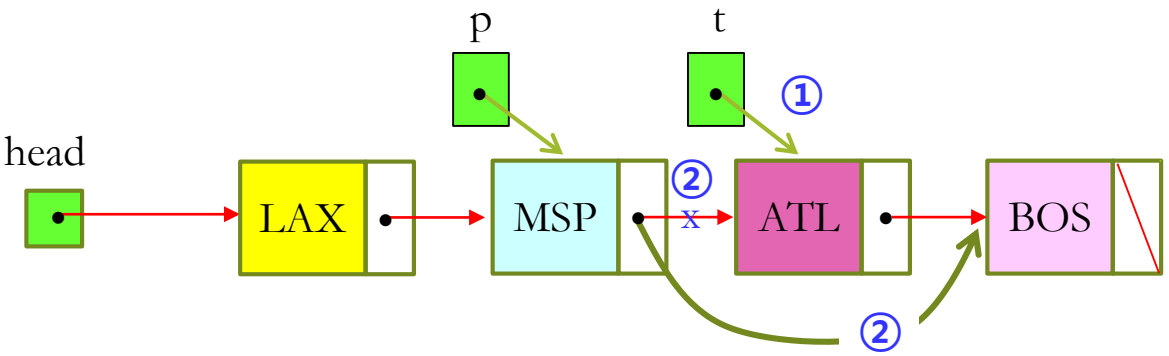


(b) 새 노드 삽입 후

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

②　　ATL　　①

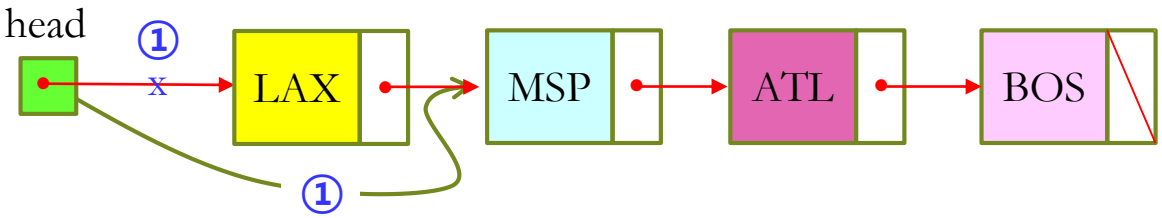
- 한방향 연결리스트에서 삭제의 관찰:



1. 삭제하는 경우, 먼저 빈 리스트인 경우와 그렇지 않은 경우로 나누어서 처리해야 함
2. [중요] 전 노드의 link를 수정해야 하므로 삭제하는 노드의 이전 노드(previous node)를 알아야 한다 (p에 저장).
   - 맨 앞의 head 노드를 삭제하는 delete_front() 함수는 head가 삭제된 노드의 다음 노드를 가리키게 하면 되므로 별도의 동작이 불필요
   - 임의의 노드 다음에 삭제하는 delete_after() 함수는 이전 노드를 찾아서 저장해 두는 동작이 필요
3. 삭제할 노드 t의 link(t.next)를 알기 위해 삭제할 노드(target node) t를 알아야 한다 (①). 즉 t=p.next
4. 이전 노드 p의 link(p.next)를 삭제된 노드 t의 link(t.next)로 갱신(②)한다. 즉 p.next=t.next

- delete_front()의 삭제 수행: 맨 앞의 노드를 삭제

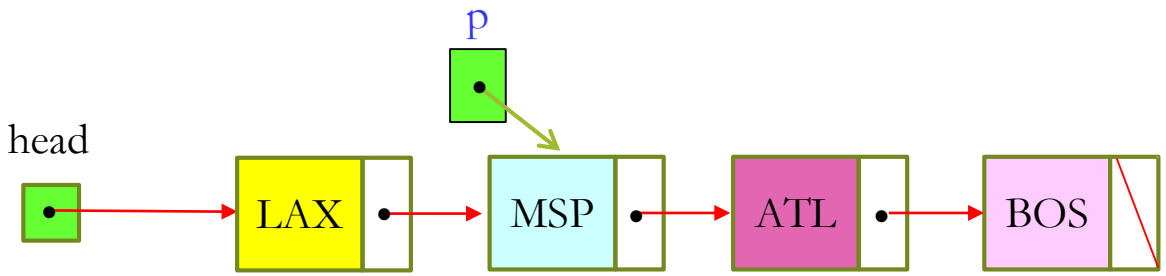
(a) 첫 노드 삭제 전



(b) 첫 노드 삭제 후

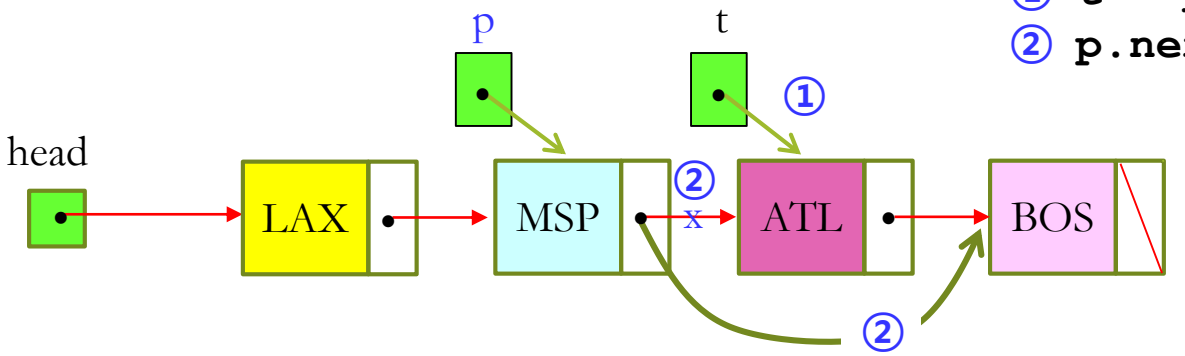
```
self.head = self.head.next
            ①
```

- delete_after(p)의 삭제 수행: 인자인 p가 가리키는 다음 노드를 삭제



(a) 노드 삭제 전

① t = p.next
② p.next = t.next



(b) 노드 삭제 후

한방향 연결리스트 구현 프로그램

```
class SList:
  class Node:
    __slots__ = "element", "next"
    def __init__(self, element, next):
      self.element = element
      self.next = next

  def __init__(self):
    self.head = None
    self.size = 0
  def size(self): return self.size
  def is_empty(self): return self.size == 0

  def insert_front(self, element):
    self.head = self.Node(element, self.head)
    self.size += 1
```

Node 클래스 정의

초기에 head는 None을 가르킴

head가 가르키는 노드 변경

```
def insert_after(self, element, p):
  p.next = self.Node(element, p.next)
  self.size += 1

def delete_front(self):
  if self.is_empty():
    raise EmptyError('Underflow')
  else:
    self.head = self.head.next
    self.size -= 1

def delete_after(self, p):
  if self.is_empty():
    raise EmptyError('Underflow')
  else:
    t = p.next
    p.next = t.next
    self.size -= 1
```

새 노드가 p 다음 노드가 됨

비어있으면 Underflow 에러

head가 2번째 노드를 참조

비어있으면 Underflow 에러

p 다음 노드를 건너뛰어 연결

```
  def search(self, target):
    p = self.head
    for k in range(self.size):
      if target == p.element:
        return k
      p = p.next
    return None

  def print_list(self):
    p = self.head
    while p:
      if p.next != None:
        print(p.element, ' -> ', end='')
      else:
        print(p.element)
      p = p.next

class EmptyError(Exception):
  pass
```

첫번째 노드부터 순차 탐색

탐색 성공

탐색 실패

노드 순차 탐색

Underflow 에러

- search()후에 반환된 노드를 이용하여 insert_after()와 delete_after()해 볼 것

```
from slist import SList
if __name__ == '__main__':
  s = SList()
  s.insert_front('BOS')
  s.insert_front('MSP')
  s.insert_after('ATL', s.head)
  s.insert_front('LAX')
  s.print_list()
  print("ATL은 %d번째" % s.search('ATL'))
  print("SFO는", s.search('SFO'))
  print("LAX 다음 노드 삭제 후:\t\t", end='')
  s.delete_after(s.head)
  s.print_list()
  print("첫 노드 삭제 후:\t\t", end='')
  s.delete_front()
  s.print_list()
  print("첫 노드로 SFO, JFK 삽입 후:\t", end='')
  s.insert_front('SFO')
  s.insert_front('JFK')
  s.print_list()
  s.delete_after(s.head.next)
  print("SFO 다음 노드 삭제 후:\t\t", end='')
  s.print_list()
```

```
출력:
LAX  -> MSP  -> ATL  -> BOS
ATL은 2번째
SFO는 None
LAX 다음 노드 삭제 후:          LAX  -> ATL  -> BOS
첫 노드 삭제 후:                ATL  -> BOS
첫 노드로 SFO, JFK 삽입 후:     JFK  -> SFO  -> ATL  -> BOS
SFO 다음 노드 삭제 후:          JFK  -> SFO  -> BOS
```

[프로그램 2-2] main.py

시간복잡도

- 읽기/쓰기
  - 대상이 되는 노드가 head node로 부터 k번째 떨어진 노드라면 head로 부터 대상 노드를 순차적으로 방문해야 하므로 O(n) 시간 소요
- 삽입/삭제
  - 대상이 되는 노드의 직전 노드를 안다면 삽입과 삭제에 O(1) 시간 소요
  - 마찬가지로 한방향 연결리스트의 head가 가리키는 곳(맨 앞)에 삽입하거나 head가 가리키는 노드를 삭제할 때 시간복잡도는 한 개의 링크정보만을 갱신하므로 O(1) 시간 소요
- 탐색
  - 탐색을 위해 연결리스트의 노드들을 첫 노드부터 순차적으로 방문해야 하므로 O(n) 시간 소요

순회(traversal)

- 자료구조를 순회한다는 것의 의미는 자료구조(배열, 연결리스트, 트리, 그래프 등)의 요소를 한번씩 방문(visiting)하고 그 데이터로 무엇인가를 하는 것을 의미
- 순회는 자료구조에 따라 (1)반복과/또는 (2)재귀의 방법으로 구현될 수 있음

https://www.youtube.com/watch?v=L1XOWwzv7fM

반복

```
class ListNode:
    def __init__(self,data):
        self.data=data
        self.next=None

def length_iter(head):
    tempNode=head
    count=0
    while(tempNode!=None):
        count=count+1
        tempNode=tempNode.next
    print(count)

head = ListNode('d')
node2 = ListNode('g')
node3 = ListNode('r')
node4 = ListNode('o')


head.next=node2
node2.next=node3
node3.next=node4
node4.next=None

print(length_iter(head))
```

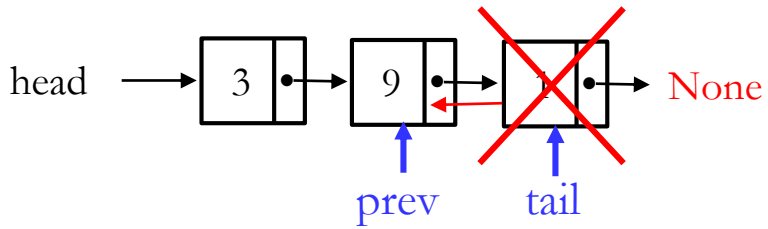
재귀

```
class ListNode:
    def __init__(self,data):
        self.data=data
        self.next=None

def length_recur(node):
  if (node == None):
     return 0
  else:
     return 1 + length_recur(node.next)


head = ListNode('d')
node2 = ListNode('g')
node3 = ListNode('r')
node4 = ListNode('o')


head.next=node2
node2.next=node3
node3.next=node4
node4.next=None

print(length_recur(head))
```

양방향 연결리스트
(Doubly linked lists)

한방향 연결리스트의 단점

- 한방향 연결리스트는 다음 노드에 대한 link(next)만 있어, 이전 노드를 알기 위해선 head 노드부터 차례로 탐색을 해야 하는 결정적인 단점을 가짐. 따라서 한방향 연결리스트는 삽입이나 삭제할 때 반드시 이전 노드를 가리키는 레퍼런스를 추가로 알아내야 하고, 역방향으로 노드들을 탐색할 수 없음



즉, tail을 알고 있더라도 tail에서 prev로 가는 link가 없기 때문에 결국 head에서 부터 prev를 찾아야 됨

- 양방향 연결리스트는 한방향 연결리스트의 이러한 단점을 보완하나, 각 노드마다 추가로 한 개의 레퍼런스를 추가로 저장해야 한다는 단점을 가짐

응용

- 양방향 연결리스트는 6장의 데크(Deque) 자료구조를 구현하는데 사용할 수도 있음
- 이항힙(Binomial Heap)이나 피보나치힙(Fibonacci Heap)과 같은 우선순위큐를 구현하는 데에도 양방향연결리스트가 부분적으로 사용할 수도 있음

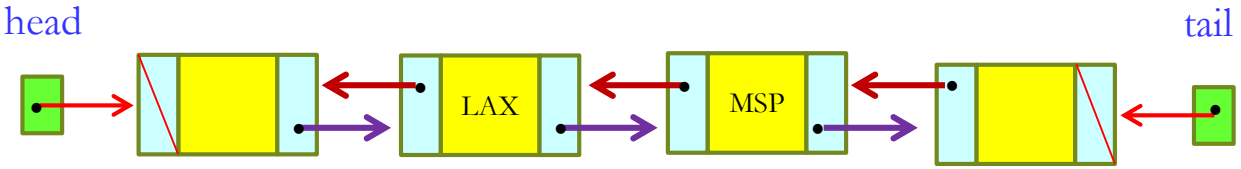
양방향 연결리스트

- 정의
  - 노드들이 양쪽 방향으로 연결된 리스트

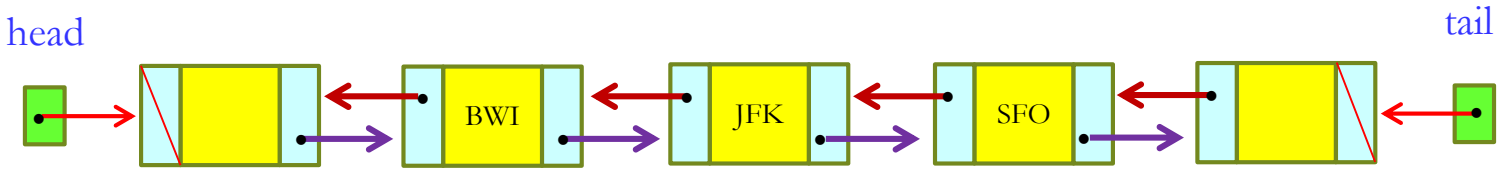
노드: 실제 값을 위한 data 정보와 이전 노드(prev)와 다음 노드(next)를 가리키는 두 개의 link 정보로 구성된 클래스

특별히 가장 앞에 있는 노드를 head 노드, 가장 뒤에 있는 노드를 tail 노드라고 부름. 양방향 연결리스트에서는 head 노드와 tail 노드로 부터 시작해 link를 계속 따라가면 모든 노드를 접근할 수 있음

가장 앞에 있는 노드는 이전 노드가 없기 때문에 그 노드의 prev link는 None을 저장하고, 가장 뒤에 있는 노드는 다음 노드가 없기 때문에 그 노드의 next link는 None을 저장함. 즉, prev link가 None이면 그 노드가 첫 노드이고 next link가 None이면 그 노드가 마지막 노드라는 의미



예시



- 양방향 연결리스트 클래스와 노드 클래스 정의

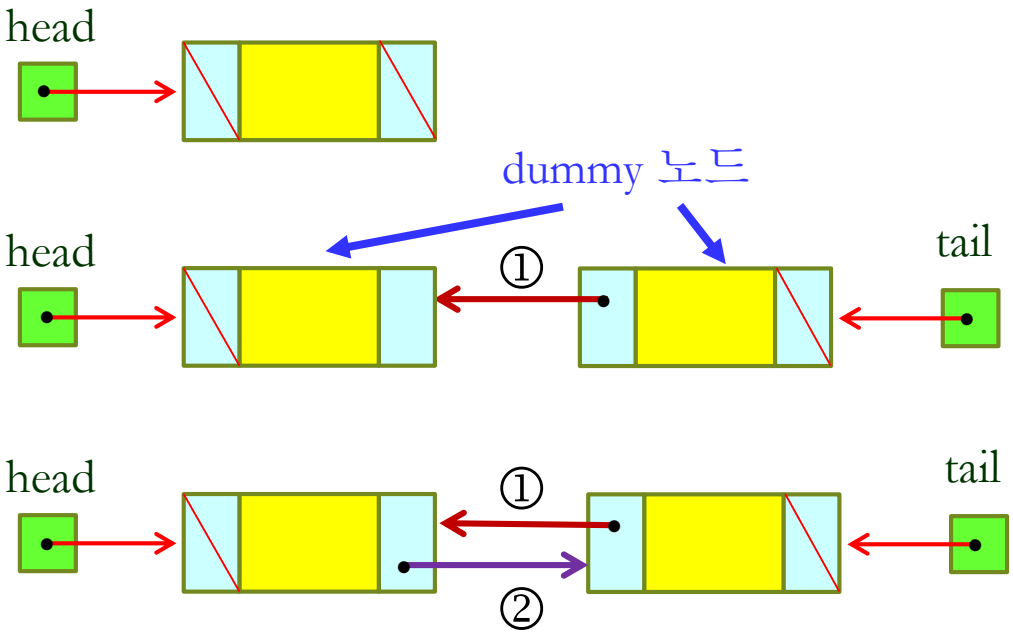
```
class DList:
  class Node:
    __slots__ = "element", "prev", "next"
    def __init__(self, element, prev, next):
      self.element = element
      self.prev = prev
      self.next = next

  def __init__(self):
    self.header = self.Node(None, None, None)
    self.trailer = self.Node(None, self.header, None)
    self.header.next = self.trailer
```

- 예와 같은 한방향연결리스트 만들기

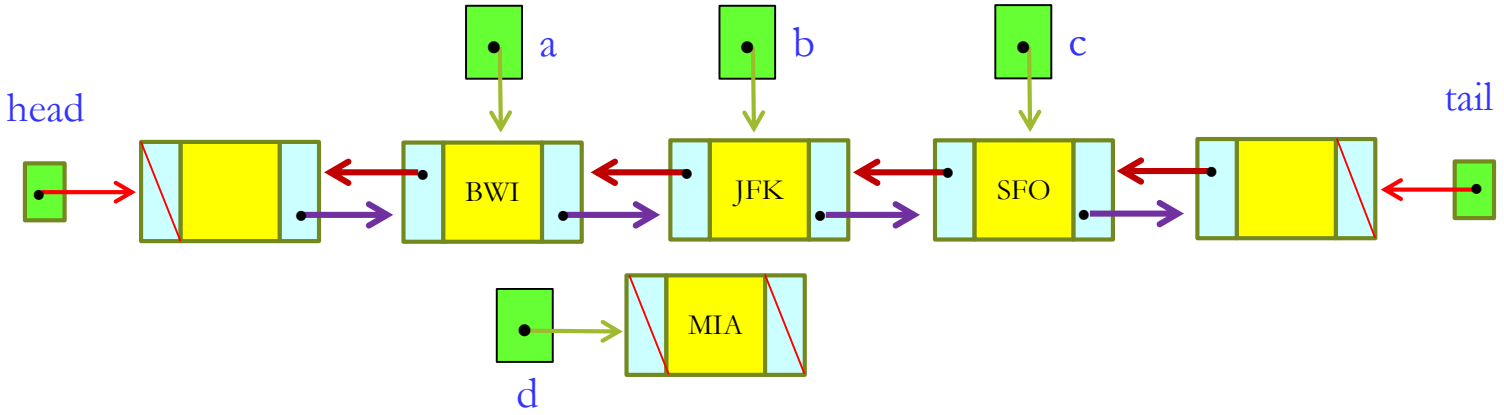
```
d = DList()
a = d.Node("BWI", None, None)
b = d.Node("JFK", None, None)
c = d.Node("SFO", None, None)

d.header.next = a
a.prev = d.header
a.next = b
b.prev = a
b.next = c
c.prev = b
c.next = d.trailer
d.trailer.prev = c

# 현제 만들어진 양방향 연결리스트
d.print_list()
```

```
self.header = self.Node(None, None, None)
self.trailer = self.Node(None, self.header, None)
self.header.next = self.trailer
```

① ②



(본 강의의 구현에서는) 첫 노드(head 노드)와 마지막 노드(tail 노드)는 항상 dummy 노드가 되도록 한다. Dummy 노드는 일종의 리스트의 처음과 마지막을 구분할 수 있는 'marker'의 기능을 하는 특별한 노드이다. 따라서 빈 양방향 연결리스트는 위의 그림처럼 dummy 노드 두개로 구성된다.



- 읽기: `print(d.header.next.next.element)` #JFK 읽기
- 쓰기: `d.header.next.next.element = "LAX"` #JFK에 LAX 쓰기
- 삽입:
```
c = d.Node("MIA", None, None) #BWI 다음에 MIA 삽입
a.next = c
c.prev = a
c.next = b
b.prev = c
```
- 삭제:
```
c.next = b.next #LAX 삭제
b.prev = c.prev
```
- 탐색: #반복의 방법으로 순회, 교과서에서는 tail노드에 도달할 때까지 순회

```
class DList:
  class Node:
    __slots__ = "element", "prev", "next"
    def __init__(self, element, prev, next):
      self.element = element
      self.prev = prev
      self.next = next

  def __init__(self):
    self.header = self.Node(None, None, None)
    self.trailer = self.Node(None, None, None)
    self.header.next = self.trailer
    self.trailer.prev = self.header

  def print_list(self):
    p = self.header.next
    while p != self.trailer:
      if p.next != self.trailer:
        print(p.element, ' <=> ', end='')
      else:
        print(p.element)
      p = p.next
```

출력:

BWI <=> JFK <=> SFO
JFK
BWI <=> LAX <=> SFO
BWI <=> MIA <=> LAX <=> SFO
BWI <=> MIA <=> SFO

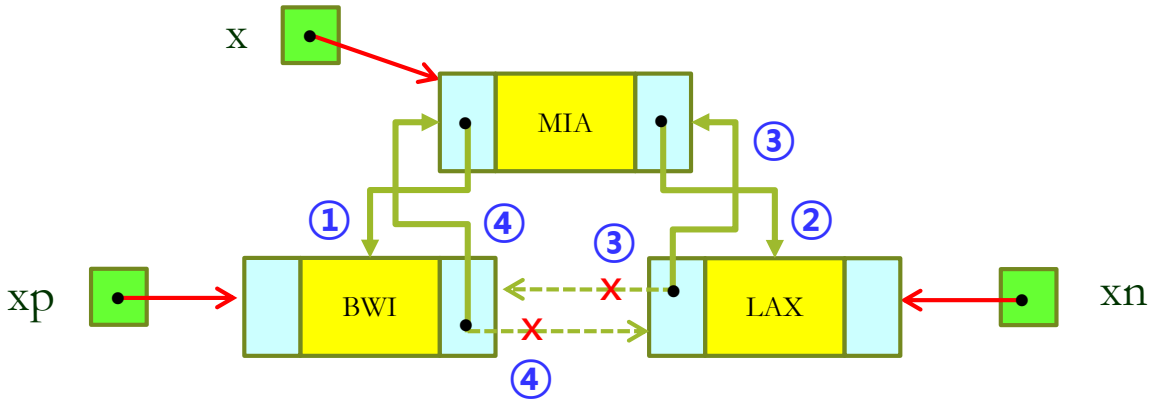
```
d = DList()
a = d.Node("BWI", None, None)
b = d.Node("JFK", None, None)
c = d.Node("SFO", None, None)

d.header.next = a
a.prev = d.header
a.next = b
b.prev = a
b.next = c
c.prev = b
c.next = d.trailer
d.trailer.prev = c

# 현제 만들어진 양방향 연결리스트
d.print_list()

# JFK 읽기
print(d.header.next.next.element)

# JFK에 LAX 쓰기
d.header.next.next.element = "LAX"
d.print_list()

# BWI 뒤에 MIA 삽입
c = d.Node("MIA", None, None)
a.next = c
c.prev = a
c.next = b
b.prev = c
d.print_list()

# LAX 삭제
c.next = b.next
b.prev = c.prev
d.print_list()
```

양방향 연결리스트 ADT

- **읽기:**
  - DL.print_list(): 양방향 연결리스트의 모든 노드 출력
- **쓰기:**
  - 없음
- **삽입:**
  - DL.insert_before(p, item): p의 앞에 노드(item을 가진)를 삽입
  - DL.insert_after(p, item): p의 다음에 노드(item을 가진)를 삽입
- **삭제:**
  - DL.delete(x): 노드 x를 삭제
- **탐색:**
  - 없음
- **보조 연산:**
  - DL.size(): 양방향 연결리스트의 크기 반환
  - DL.is_empty(): 양방향 연결리스트가 비었는지 체크

양방향 연결리스트 ADT의 구현

- 양방향 연결리스트에서 삽입의 관찰:

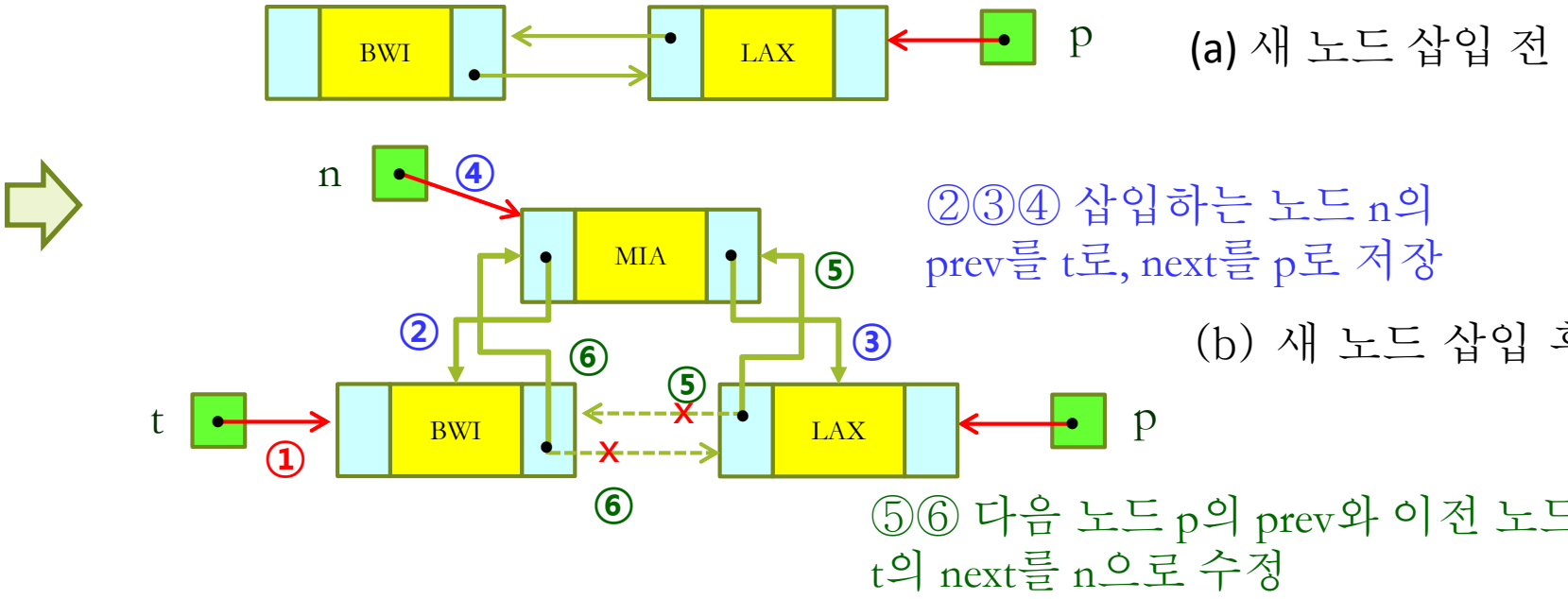


1. 삽입할 노드 x의 이전 노드와 다음 노드를 xp와 xn에 각각 저장한다.
2. 삽입할 노드 x의 link를 x.prev=xp와 x.next=xn으로 갱신한다.
3. 다음 노드 xn의 link를 xn.prev=x로 갱신한다.
4. 이전 노드 xp의 link를 xp.next=x로 갱신한다.

- insert_before(p,MIA)의 삽입 수행: 인자인 p가 가리키는 노드의 앞에 삽입
  - [중요] 이전과 다음 노드의 link를 수정해야 하므로 삽입하는 노드의 이전과 다음 노드를 알아야 한다.

① 삽입하는 노드의 이전 노드를 t에 저장
(삽입하는 노드의 다음 노드는 인자 p)

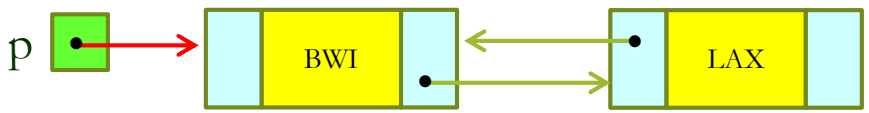
```
t = p.prev
①
n = self.Node(element, t, p)
④                    MIA   ②  ③
p.prev = n
⑤
t.next = n
⑥
```

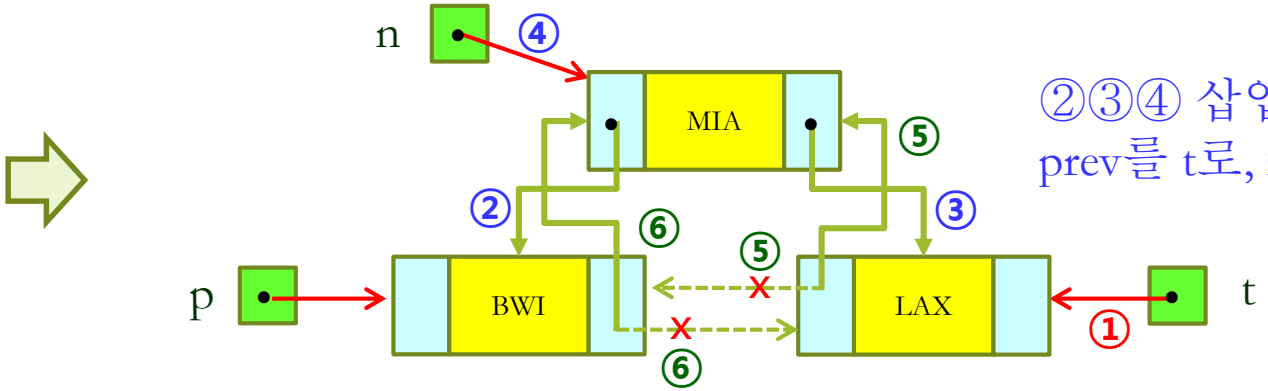


• insert_after(p, MIA)의 삽입 수행: 인자인 p가 가리키는 노드의 뒤에 삽입

① 삽입하는 노드의 다음 노드를 t에 저장
(삽입하는 노드의 이전 노드는 인자 p)

```
t = p.next
  ①
n = self.Node(element, p, t)
  ④                    MIA   ②  ③
t.prev = n
         ⑤
p.next = n
         ⑥
```



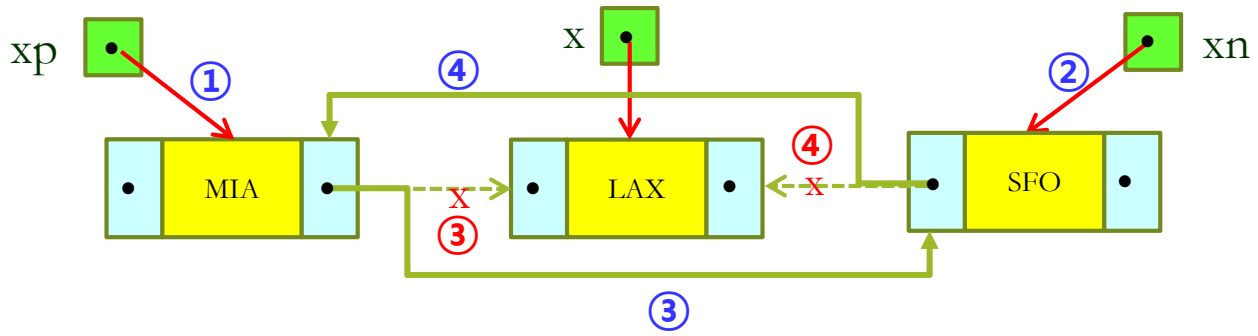


②③④ 삽입하는 노드 n의 prev를 t로, next를 p로 저장

⑤⑥ 다음 노드 t의 prev와 이전 노드 p의 next를 n으로 수정

• 양방향 연결리스트에서 삭제의 관찰:

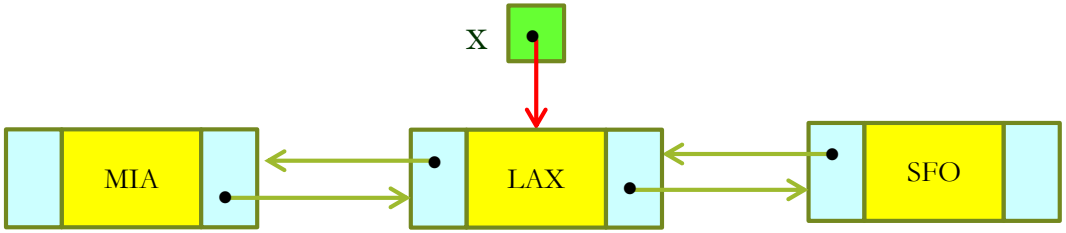
1. 삭제할 노드 x의 이전 노드와 다음 노드를 xp와 xn에 각각 저장한다.
2. 이전 노드 xp의 next link를 xp.next=xn로 갱신한다.
3. 다음 노드 xn의 prev link를 xn.prev=xp로 갱신한다.

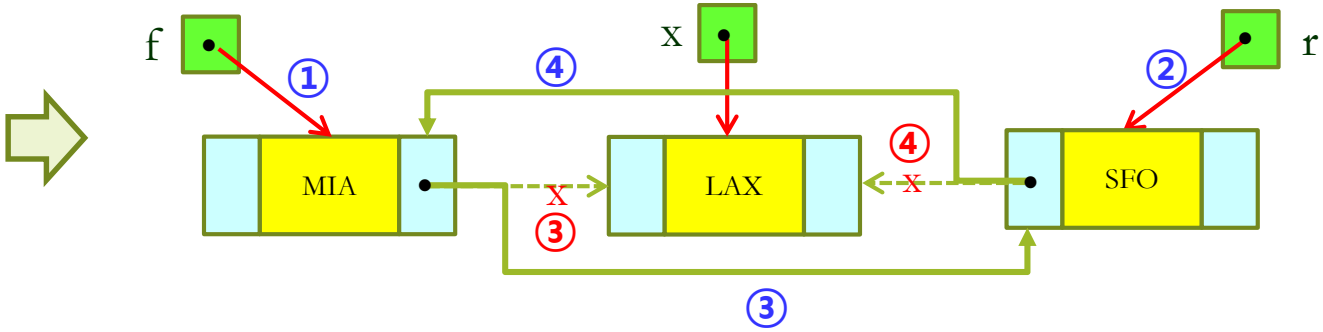


- delete(x)의 삭제 수행: 인자인 x가 가리키는 노드를 삭제
  - [중요] 이전과 다음 노드의 link를 수정해야 하므로 삭제하는 노드의 이전과 다음 노드를 알아야 한다.

```
① t = p.prev
② n = p.next
③ t.next = n
④ n.prev = t
```

① 삭제하는 노드의 이전 노드를 f에 저장
② 삭제하는 노드의 다음 노드를 r에 저장
③ 노드 f의 다음 노드를 r로 수정
④ 노드 r의 이전 노드를 f로 수정





양방향 연결리스트 구현 프로그램

```
class DList:
  class Node:
    __slots__ = "element", "prev", "next"
    def __init__(self, element, prev, next):
      self.element = element
      self.prev = prev
      self.next = next
  def __init__(self):
    self.header = self.Node(None, None, None)
    self.trailer = self.Node(None, None, None)
    self.header.next = self.trailer
    self.trailer.prev = self.header
    self.size = 0
  def size(self): return self.size
  def is_empty(self): return self.size == 0
```

Node 클래스 정의

이중연결리스트 생성자
header, tailer, size로 구성

```
def insert_before(self, p, element):
  t = p.prev
  n = self.Node(element, t, p)
  p.prev = n
  t.next = n
  self.size += 1

def insert_after(self, p, element):
  t = p.next
  n = self.Node(element, p, t)
  t.prev = n
  p.next = n
  self.size += 1

def delete(self, p):
  t = p.prev
  n = p.next
  t.next = n
  n.prev = t
  self.size -= 1
  return p.element

def print_list(self):
  p = self.header.next
  while p != self.trailer:
    if p.next != self.trailer:
      print(p.element, ' <=> ', end=' ')
    else:
      print(p.element)
    p = p.next
```

새 노드를 생성하여
n이 참조

새 노드와
앞뒤 노드 연결

p를 건너 띄고
p의 앞뒤 노드를 연결

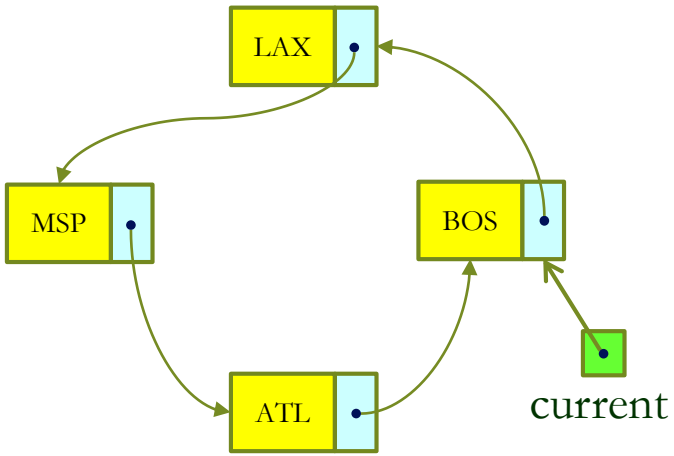
시간복잡도

- 읽기/쓰기
  - O(1)
- 삽입/삭제
  - 각각 상수 개의 레퍼런스만을 갱신하므로 O(1) 시간에 수행
  - 그러나, 양방향 연결리스트의 중간에서 삽입/삭제가 일어나는 경우 (insert_after(p, item), insert_before(p, item)나 delete(x)의 경우)에 삽입 또는 삭제하려는 노드의 link정보(p 또는 x)가 주어지지 않으면 head 또는 tail로부터 해당 노드를 찾기 위해 탐색해야 하므로 O(n) 시간 소요
- 탐색
  - head 또는 tail로부터 노드들을 순차적으로 탐색해야 하므로 O(n) 시간 소요

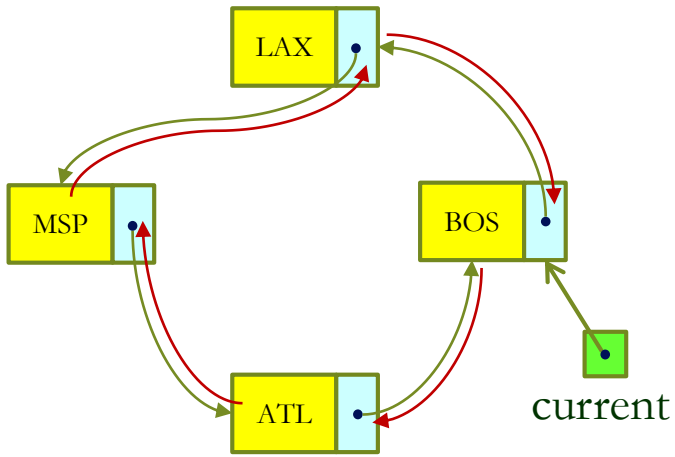
원형 연결리스트
(Circularly linked lists)

필요성

- 각 노드를 동등하게 방문해야 하는 경우
- 한방향 연결리스트에 비해서 원형연결리스트는 마지막 노드와 첫 노드를 O(1) 시간에 방문할 수 있다. 마지막 노드와 첫 노드를 빨리 접근해야 하는 경우



한방향 원형연결리스트



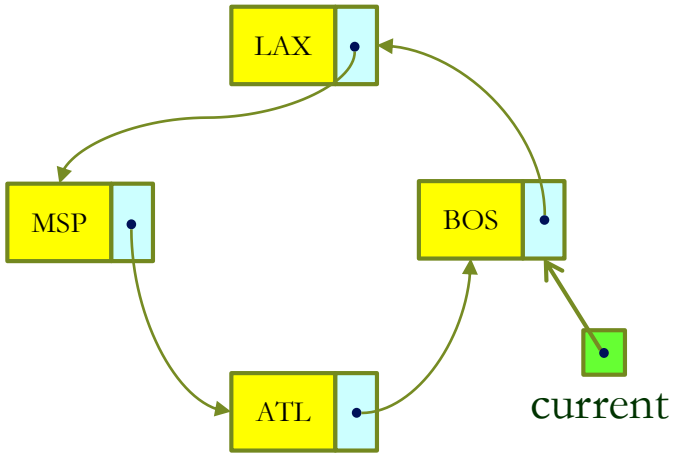
양방향 원형연결리스트

응용

- 여러 사람이 차례로 돌아가며 하는 게임을 구현하는데 적합한 자료 구조 (예, Josephus문제)
  - 1, 2, …, n번까지 원형 테이블에 앉아있다. 1번부터 시작해서 k번째 사람이 탈락하는 게임을 한다.
  - 예: n = 6이고, k = 2인 경우, 탈락하는 순서가 2→4→6→3→1가 되어 최종적으로 5가 생존한다.
- 많은 사용자들이 동시에 사용하는 컴퓨터에서 CPU 시간을 분할(time sharing 문제)하여 작업들에 할당하는 운영체제에 사용
- 마지막 노드가 첫 노드를 가리키는 속성을 이용하여 원형 큐(circular queue)를 구현하는 데 사용할 수도 있음

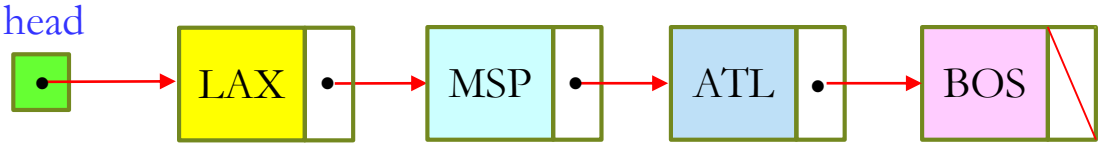
원형 연결리스트

- 정의
  - 마지막 노드가 첫 노드와 연결된 한방향연결리스트 (양방향 연결리스트로도 구현 가능)
  - 원형연결리스트에서는 마지막 노드의 레퍼런스가 저장된 current가 한방향 연결리스트의 head와 같은 역할
  - 첫 노드를 다시 방문하면 순회 중단
  - 빈 리스트가 아니면 어떤 노드도 마지막 노드임을 알려주는 None 레퍼런스를 가지고 있지 않으므로 무한 루프가 발생할 수 있으므로 유의

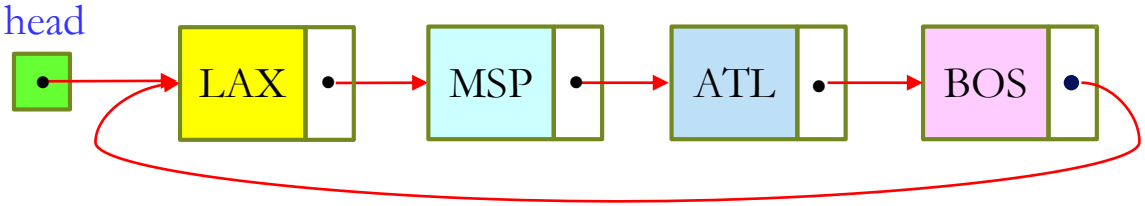


한방향 연결리스트 vs. 원형 연결리스트

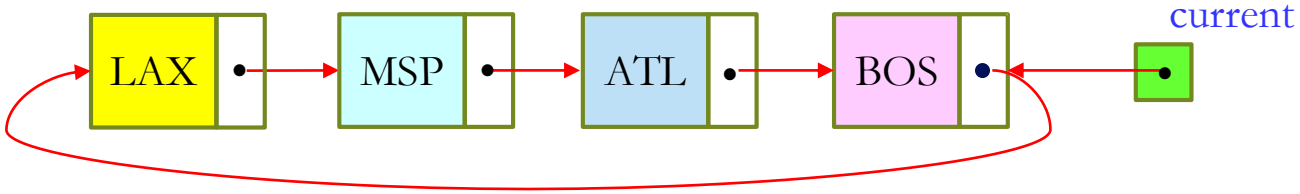
- 한방향 연결리스트
  - 연결리스트 마지막 노드의 link가 None을 가리킴



- 원형 연결리스트
  - 연결리스트 마지막 노드의 link가 첫 번째 노드를 가리킴
  - head 노드의 다음에 새 노드 삽입하기는 쉬우나, 마지막 노드 다음에 삽입하기 위해서는 모든 연결리스트를 따라가야 함 → 어떻게 해결?



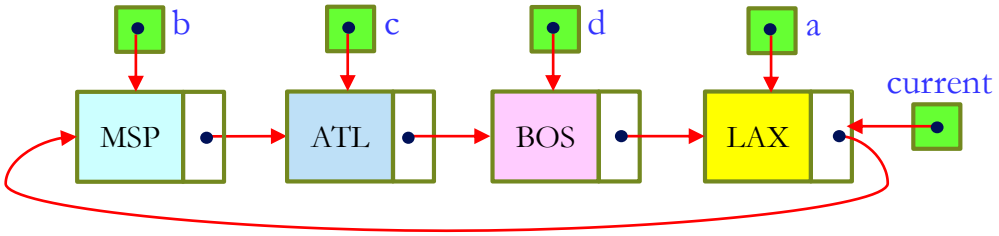
  - head 노드(current 노드라고 이름 붙임)가 마지막 노드를 가리키면 연결리스트의 첫 노드나 마지막 노드 다음에 노드를 삽입하기 쉬움



시간복잡도

- 읽기/쓰기
  - O(1)
- 삽입/삭제
  - 삽입이나 삭제 연산 각각 상수 개의 레퍼런스를 갱신하므로 O(1) 시간에 수행
  - 그러나, 원형 연결리스트의 중간에서 삽입/삭제가 일어나는 경우에 삽입하려는 노드의 이전 노드의 link정보가 주어지지 않으면 current로부터 이전 노드를 찾기 위해 탐색을 수행해야 하므로 O(n) 시간 소요
- 탐색
  - current로부터 노드들을 순차적으로 탐색해야 하므로 O(n) 소요

예

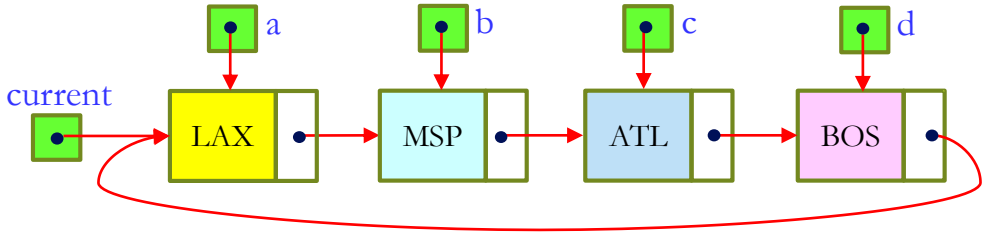


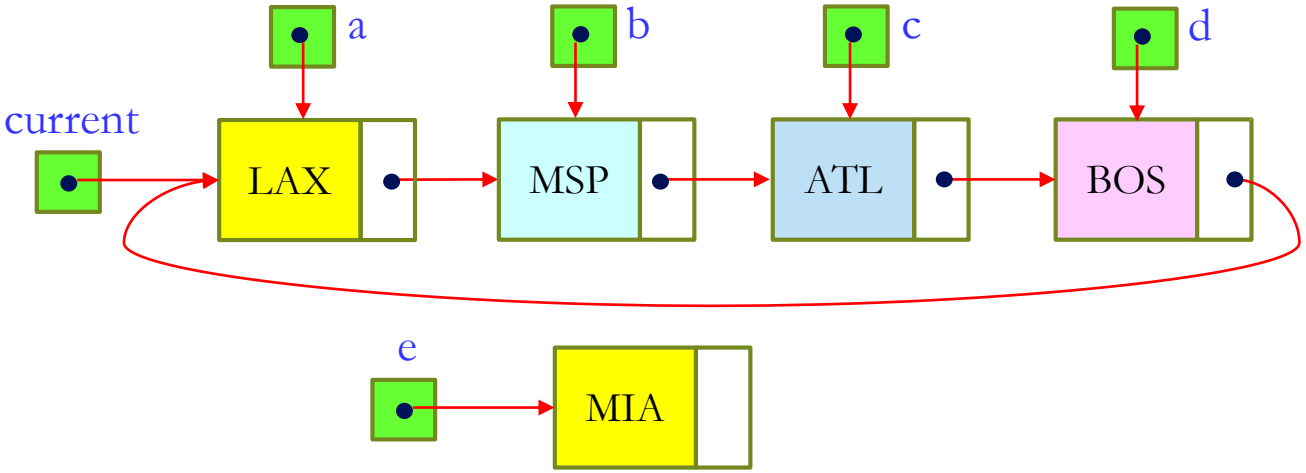
- 원형 연결리스트 클래스와 노드 클래스 정의

```
class CList:
  class Node:
    __slots__ = "element", "next"

    def __init__(self, element, next):
      self.element = element
      self.next = next

  def __init__(self):
    self.current = None
    self.size = 0
```

```
s = CList()
a = s.Node('LAX', None)
b = s.Node('MSP', None)
c = s.Node('ATL', None)
d = s.Node('BOS', None)

s.current = a
a.next = b
b.next = c
c.next = d
d.next = a
```

- 예와 같은 원형 연결리스트 만들기





- 읽기: `print(s.current.next.next.element)` #ATL 읽기
- 쓰기: `s.current.next.next.element = 'SFO'` #ATL에 SFO 쓰기
- 삽입: `e = s.Node('MIA', None)` #LAX 다음에 MIA 삽입

  `a.next=e`

  `e.next=b`
- 삭제: `e.next=b.next` #MSP 삭제
- 탐색: #반복의 방법으로 순회, (1) 첫 노드를 다시 방문하면 순회 중단하는 방법 (본 강의의 print_list())과 (2) 연결리스트의 크기로 순회하는 방법

```
class CList:

  class Node:
    __slots__ = "element", "next"

    def __init__(self, element, next):
      self.element = element
      self.next = next

  def __init__(self):
    self.current = None

  def print_list(self):
    f = self.current.next
    p = f
    while p.next != f:     # 첫 노드를 다시 방문하면
      print(p.element, ' -> ', end=' ')
      p = p.next
    print(p.element)
```

출력:

```
MSP -> ATL -> BOS -> LAX
ATL
MSP -> SFO -> BOS -> LAX
MIA -> MSP -> SFO -> BOS -> LAX
MIA -> SFO -> BOS -> LAX
```

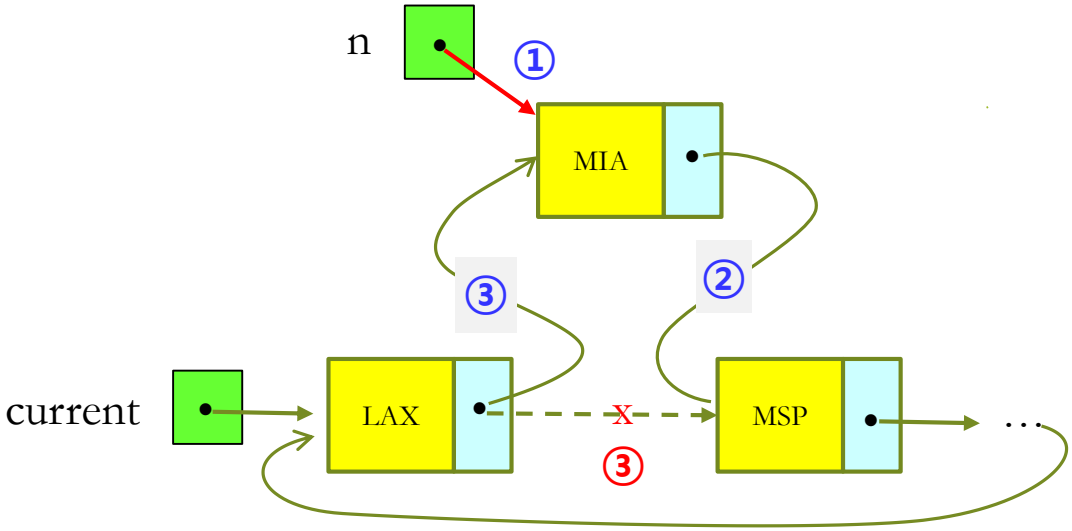
```
s = CList()
a = s.Node('LAX', None)
b = s.Node('MSP', None)
c = s.Node('ATL', None)
d = s.Node('BOS', None)

s.current = a
a.next = b
b.next = c
c.next = d
d.next = a

s.print_list()

# ATL 읽기
print(s.current.next.next.element
)

# ATL에 SFO 쓰기
s.current.next.next.element =
'SFO'
s.print_list()

# LAX 다음에 MIA 삽입
e = s.Node('MIA', None)
a.next = e
e.next = b
s.print_list()

# MAP 삭제
e.next = b.next
s.print_list()
```

원형 연결리스트 ADT

- **읽기:**
  - CL.print_list(): 원형 연결리스트 CL의 모든 노드 출력
  - CL.first(): 맨 앞 노드(current 노드의 다음 노드)의 값을 반환
- **삽입:**
  - CL.insert(item): 맨 앞에 노드(item을 가진)를 삽입
- **삭제:**
  - CL.delete(): 맨 앞에 노드를 삭제하고 값을 반환
- **보조 연산:**
  - CL.no_items(): 원형 연결리스트 CL의 크기 반환
  - CL.is_empty(): 원형 연결리스트 CL이 비었는지 체크

원형 연결리스트 ADT의 구현

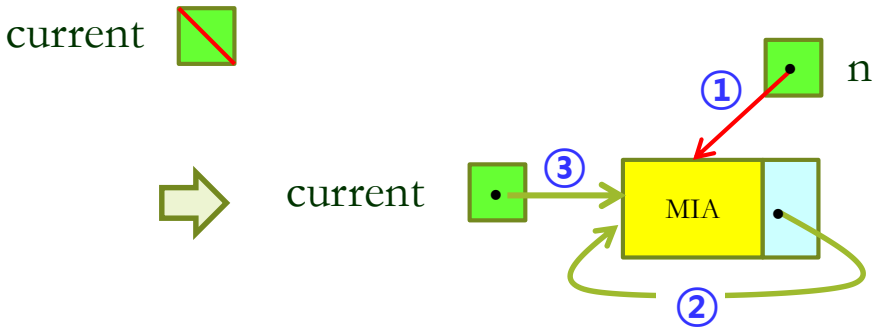
- 원형 연결리스트에서 삽입의 관찰:



- 빈 원형 연결리스트에 노드 n을 삽입하는 경우
  - 삽입할 노드 n의 link를 자신으로 저장; n.next=n
  - 마지막 노드를 자신으로 저장; current=n
- 비지않은 원형 연결리스트의 첫 노드에 노드 n을 삽입하는 경우
  - 삽입할 노드 n의 link를 첫 노드로 갱신(②); n.next=current.next
  - 첫 노드를 자신으로 갱신(③); current.next=n

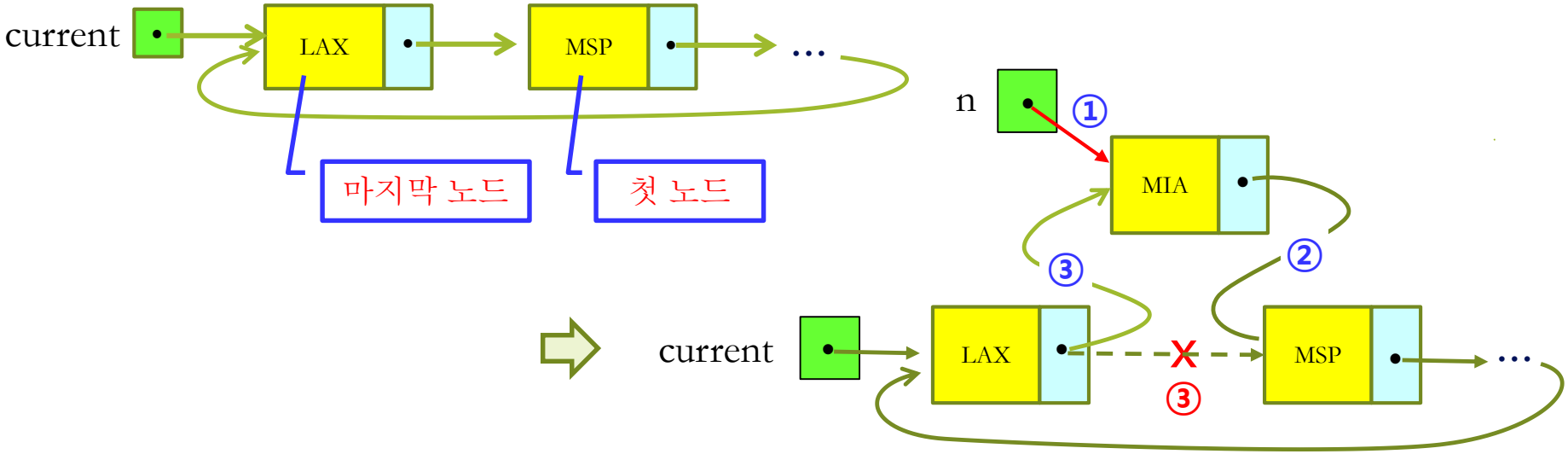
• insert(item)의 노드 삽입: 삽입할 노드 n을 리스트의 첫 노드에 삽입

(1) 빈 원형 연결리스트인 경우

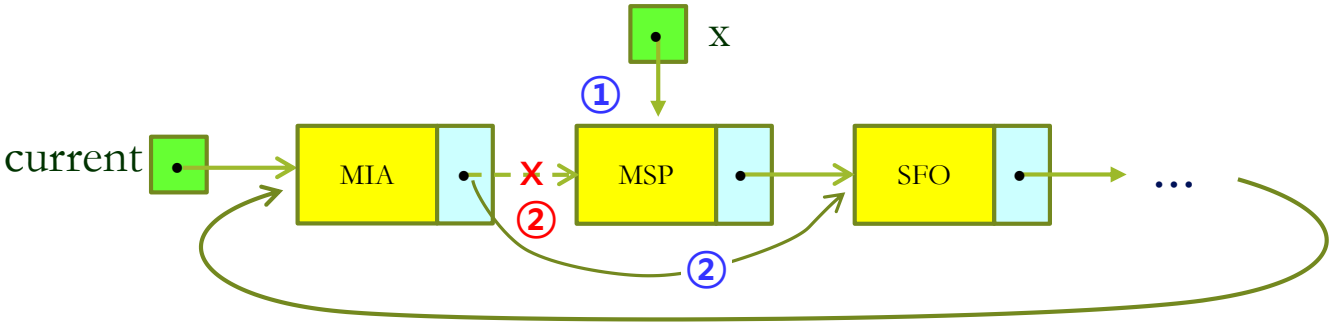



```
n = self.Node(element, None)
if self.is_empty():
  n.next = n
  self.current = n
else:
  n.next = self.current.next
  self.current.next = n
```

(2) 비지않은 원형 연결리스트인 경우



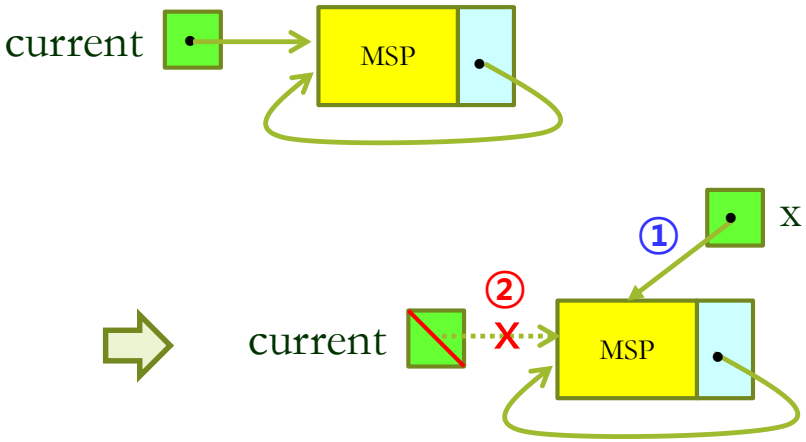
- 원형 연결리스트에서 삭제의 관찰:



- 첫 노드 x를 삭제하여 빈 원형 연결리스트가 되는 경우
  1. 마지막 노드를 None으로 저장; current=None
- 첫 노드 x를 삭제해도 비지않은 원형 연결리스트가 되는 경우
  1. 삭제할 노드 x를 첫 노드로 갱신(①); x=current.next
  2. 첫 노드를 다음 노드로 갱신(②); current.next=x.next

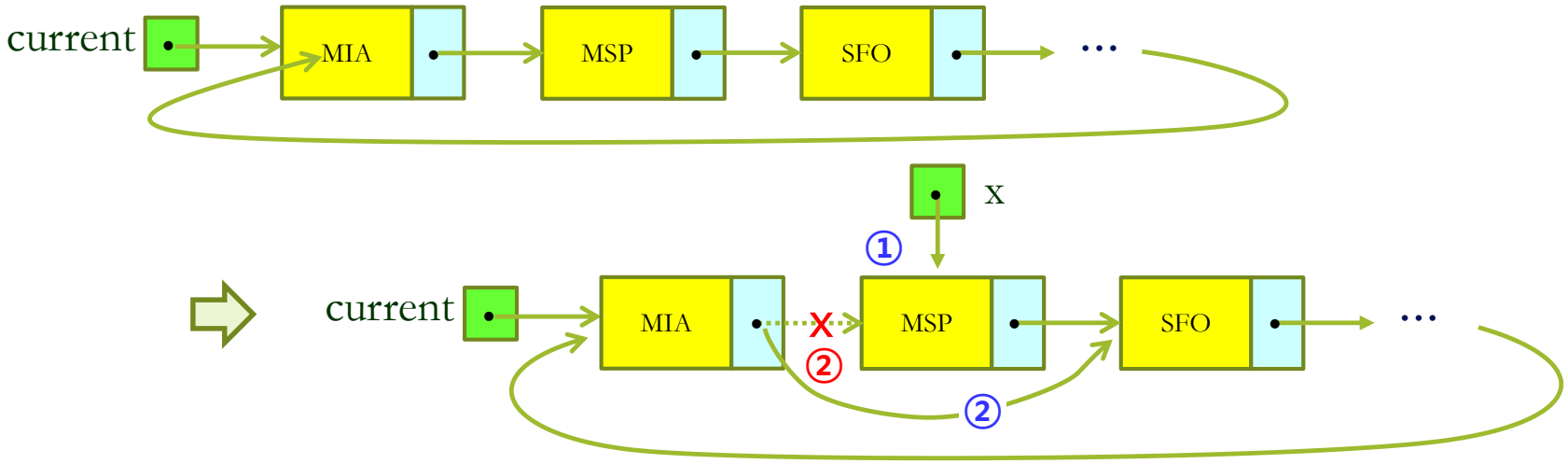
• delete()의 노드 삭제: 리스트의 첫 노드 x를 삭제

(1) 빈 원형 연결리스트가 되는 경우




```
x = self.current.next
①
if self.size == 1:
    self.current = None
                   ②
else:
    self.current.next = x.next
                        ②
```

(2) 비지않은 원형 연결리스트가 되는 경우



원형 연결리스트 구현 프로그램

```
class CList:
  class Node:
    __slots__ = "element", "next"

    def __init__(self, element, next):
      self.element = element
      self.next = next

  def __init__(self):
    self.current = None
    self.size = 0

  def no_items(self): return self.size
  def is_empty(self): return self.size == 0

  def insert(self, element):
    n = self.Node(element, None)
    if self.is_empty():
      n.next = n
      self.current = n
    else:
      n.next = self.current.next
    self.current.next = n
    self.size += 1
```

Node 클래스 정의

원형리스트 생성자
current와 size로 구성

새 노드 생성하여
n이 참조

새 노드는 자신을 참조
current는 새 노드 참조

새 노드는 첫 노드 참조
current의 다음노드는 새노드
참조

```
def first(self):
  if self.is_empty():
    raise EmptyError('Underflow’)
  f = self.current.next
  return f.element

def delete(self):
  if self.is_empty():
    raise EmptyError('Underflow’)
  x = self.current.next
  if self.size == 1:
    self.current = None
  else:
    self.current.next = x.next
  self.size -= 1
  return x.element
```

Empty 상태

current가 두번째 노드 참조

```
  def print_list(self):
    if self.is_empty():
      print('리스트가 비어있습니다.')
    else:
      f = self.current.next
      p = f
      while p.next != f:
        print(p.element, ' -> ', end=' ')
        p = p.next
      print(p.element)


class EmptyError(Exception):
  pass
```

첫 노드 재방문 시 루프 중단

노드를 차례로 방문

Underflow 에러

코딩테스트

LeetCode 문제

- 876. Middle of the Linked List: (1.기본동작-Middle)
- 206. Reverse Linked List: (2.기본동작-Reverse)
- 86. Partition List: (3.기본동작-Split)
- 143. Reorder List: (4.기본동작-Reorder)
- 21. Merge Two Sorted Lists (병합)
- 203. Remove Linked List Elements (삭제)
- 141. Linked List Cycle (루프 찾기)
- 160. Intersection of Two Linked Lists (교차점 찾기)

876. Middle of the Linked List

876. Middle of the Linked List

Easy 2938 81 Add to List Share

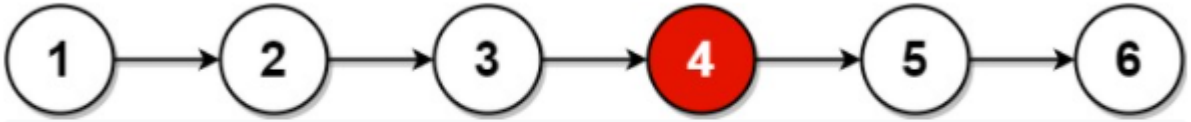
Given the `head` of a singly linked list, return *the middle node of the linked list.*

If there are two middle nodes, return **the second middle** node.

Example 1:


```
Input: head = [1,2,3,4,5]
Output: [3,4,5]
Explanation: The middle node of the list is node 3.
```

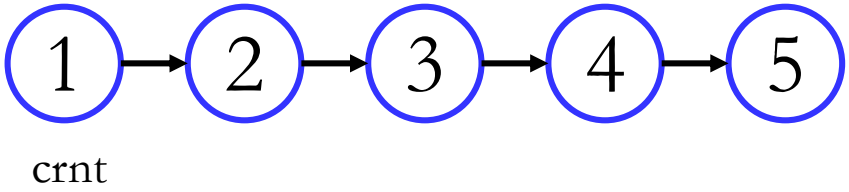
Example 2:




```
Input: head = [1,2,3,4,5,6]
Output: [4,5,6]
Explanation: Since the list has two middle nodes with values 3 and 4, we
return the second one.
```

- 관찰 1: 직관적인 방법
  - head노드에 crnt노드로 잡고 next로 이동하면서 count 값을 증가시켜가면서 전체 노드의 개수를 셈
  - 전체 노드의 개수를 반으로 나눈 위치로 crnt 노드를 이동하여 노드를 반환

초기상태:



count: 0

- 풀이

```
def indexWay(self, head: ListNode) -> ListNode:
    total_count = 0
    crnt_node = head
    while crnt_node:
        total_count += 1
        crnt_node = crnt_node.next

    half_count = int(total_count / 2)

    crnt_node = head
    for idx in range(0, half_count):
        crnt_node = crnt_node.next

    return crnt_node
```

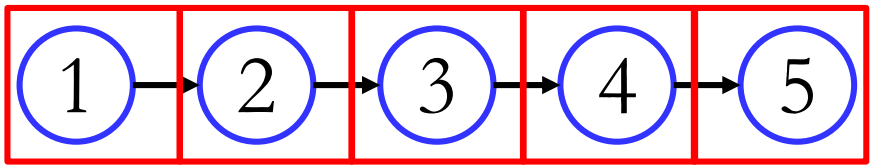
시간복잡도: O(n)

공간복잡도: O(1)

```
1  # Definition for singly-linked list.
2  # class ListNode(object):
3  #     def __init__(self, val=0, next=None):
4  #         self.val = val
5  #         self.next = next
6  class Solution(object):
7      def middleNode(self, head):
8          """
9          :type head: ListNode
10         :rtype: ListNode
11         """
12
```

- 관찰 2: 리스트에 저장
  - list를 사용한다면 linked list를 list에 저장하고 (O(n)), len() 함수 이용하여 길이 구한 후 (O(1)), 가운데 원소를 random access(O(1)) 해도 된다.
  - 시간복잡도 O(n), 공간복잡도는 별도의 list가 하나 더 필요하므로 O(n)이 필요.
  - len()은 O(1)의 시간 복잡도를 가지는데 list에 원소를 추가하거나 삭제할 때 list객체는 자동으로 갯수를 세고 이를 반환한다.

초기상태:

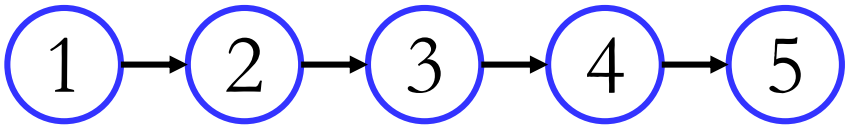


시간복잡도: O(n)

공간복잡도: O(n)

- 관찰 3: Equi-directional two pointers technique(slow-runner and fast-runner)

초기상태:



fast_node

slow_node

- 풀이

```
def fastSlow(self, head: ListNode) -> ListNode:
    slow_node = head
    fast_node = head

    while fast_node:
        if fast_node.next:
            slow_node = slow_node.next
            fast_node = fast_node.next.next
        else:
            break

    return slow_node
```

시간복잡도: O(n)

공간복잡도: O(1)

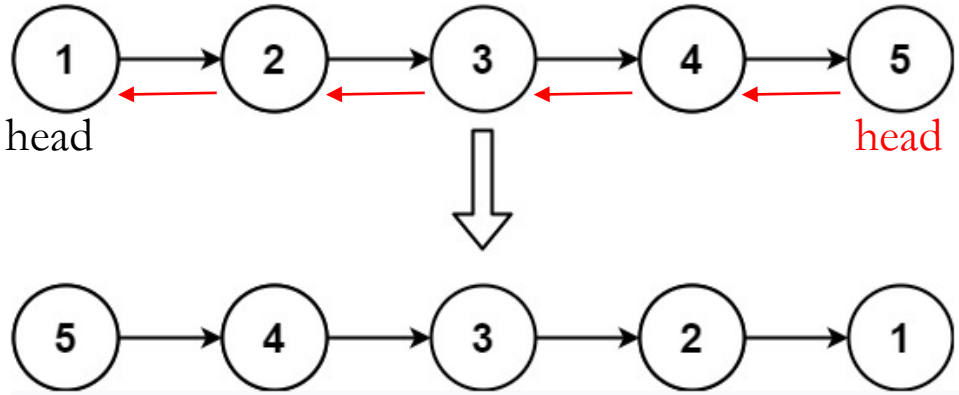
206. Reverse Linked List

206. Reverse Linked List

Easy 8012 148 Add to List Share

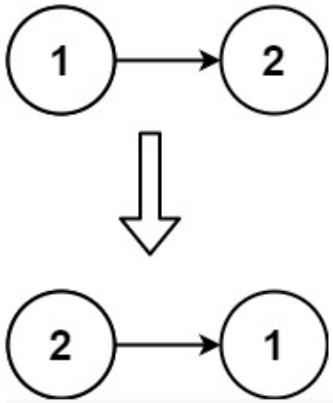
Given the `head` of a singly linked list, reverse the list, and return *the reversed list*.

Example 1:




```
Input: head = [1,2,3,4,5]
Output: [5,4,3,2,1]
```

Example 2:



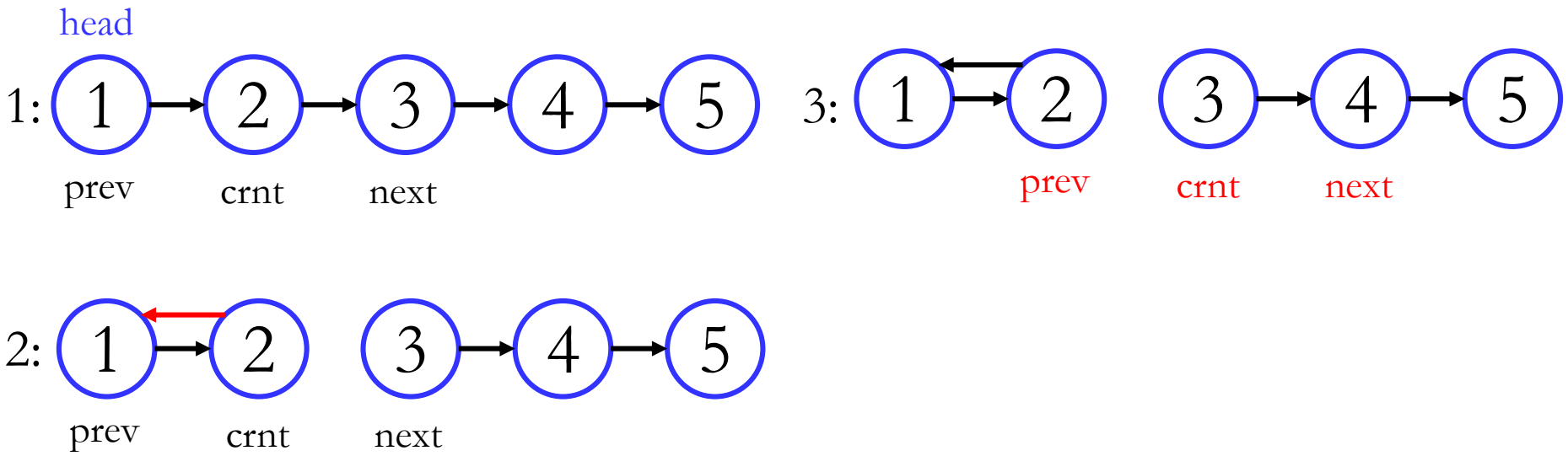

```
Input: head = [1,2]
Output: [2,1]
```

Example 3:

```
Input: head = []
Output: []
```

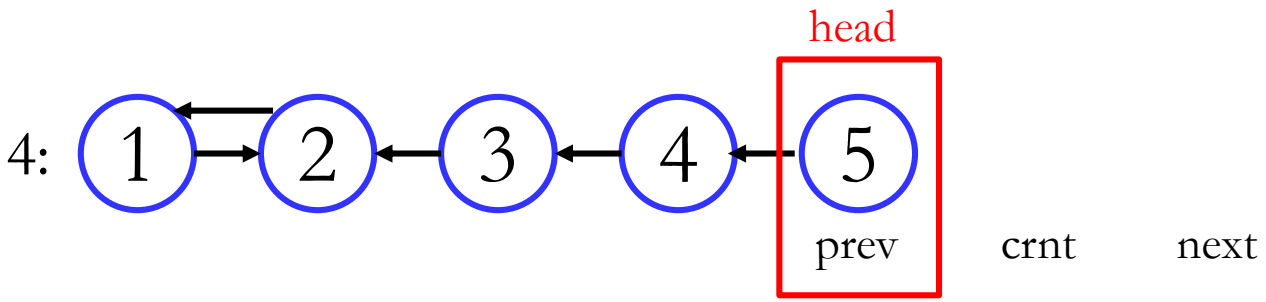
- 관찰 1: 반복 방법

    1. 현재 노드(crnt), 이전 노드(prev), 다음 노드(next)를 저장; 이전 노드에 대한 link가 없으므로 이전 노드를 저장하고, 현재 노드의 link를 변경하기 전에 다음 노드를 저장
    2. crnt 노드의 link가 prev 노드를 가리키도록 변경
    3. prev, crnt, next를 한 칸씩 오른쪽으로 이동



4. crnt 노드의 link가 None이 아닌 동안 계속 반복하여, 마지막 노드에 도착하면 prev 노드를 head 노드로 반환

- 첫 노드의 next link는 처리하지 않았는데 이것은 edge case로 가장 첫 노드는 아무것도 가리키지 않게 처리하면 됨



- 풀이

```
def iterativeWay(self, head: ListNode) -> ListNode:
    if head is None:
        return head
    elif head.next is None:
        return head

    crnt_node = head.next
    prev_node = head
    head.next = None  #가장 첫 노드는 아무것도 가리키지 않게 미리 처리해
                      둠
    while crnt_node:
        tmp_next_node = crnt_node.next
        crnt_node.next = prev_node
        prev_node = crnt_node
        crnt_node = tmp_next_node

    return prev_node
```

시간복잡도: O(n)
공간복잡도: O(1)

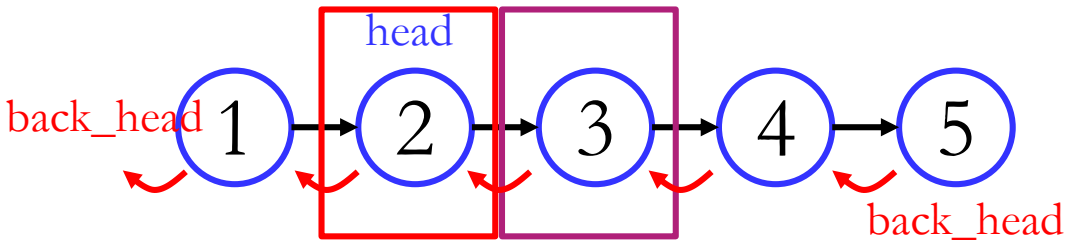
- 관찰 2: 재귀 방법

현재 스택 프레임이 노드 2라고 가정하면, 내부의 재귀함수는 다음 노드를 가리키는 스택 프레임을 만들게 될 것. 이는 반복적으로 연결리스트의 끝까지 진행됨.

연결리스트의 끝에 도달하면 재귀를 끝내는 base case를 만족해야 함.

- 마지막 노드의 주소(back_head) 반환

이렇게 base case에 도달해서 재귀함수가 pop될 때마다 각 스택 프레임(stack frame)은 연결리스트의 마지막 노드의 주소를 첫번째 노드를 가리키는 스택 프레임까지 반환해야 함. 왜냐하면 마지막 노드(back_head)가 reverse된 연결리스트의 head 노드가 되기 때문.



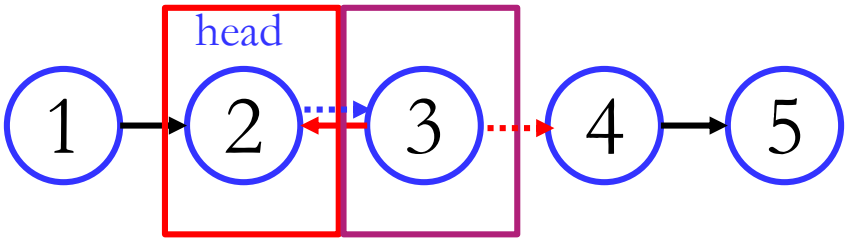

```
def recursiveWay(head)
#base case
  head.next 가 None일 때
#recursive case
  back_head = recursiveWay(head.next)
  link 조작
```

- link 조작

(1) 해당 노드(노드 2)의 next.next가 가리키는 link를 자신을 가리키게 (head.next.next = head)하고, (2) 해당 노드(노드 2)의 next가 가리키는 link를 제거(head.next = None)하면 됨.

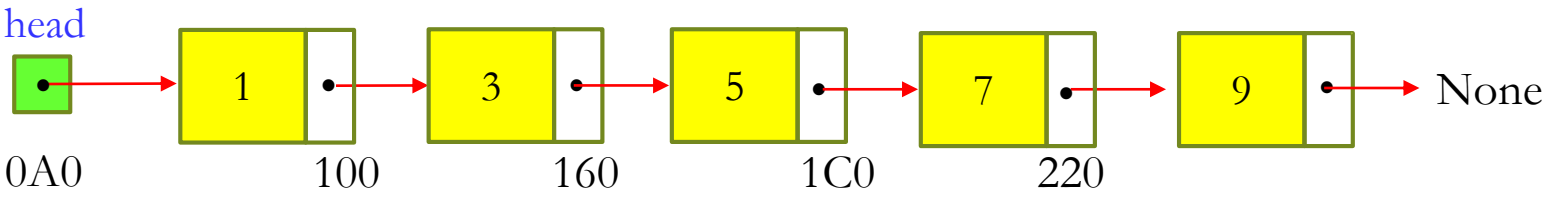
이 작업은 맨 마지막 노드에서부터 **역방향**으로 가면서 바꿔 줘야하므로 재귀 호출 후에 해야 됨!

순방향으로 가면서 어떤 동작을 해야 된다면 재귀 호출 전에 해야한다는 것도 알 수 있음!



- 풀이

```
def recursiveWay(self, head: ListNode) -> ListNode:
    # base case
    if head is None:
        return head
    elif head.next is None:
        return head
    # recursive case
    back_head = self.recursiveWay(head.next)
    # link 조작
    head.next.next = head
    head.next = None

    return back_head
```



86. Partition List

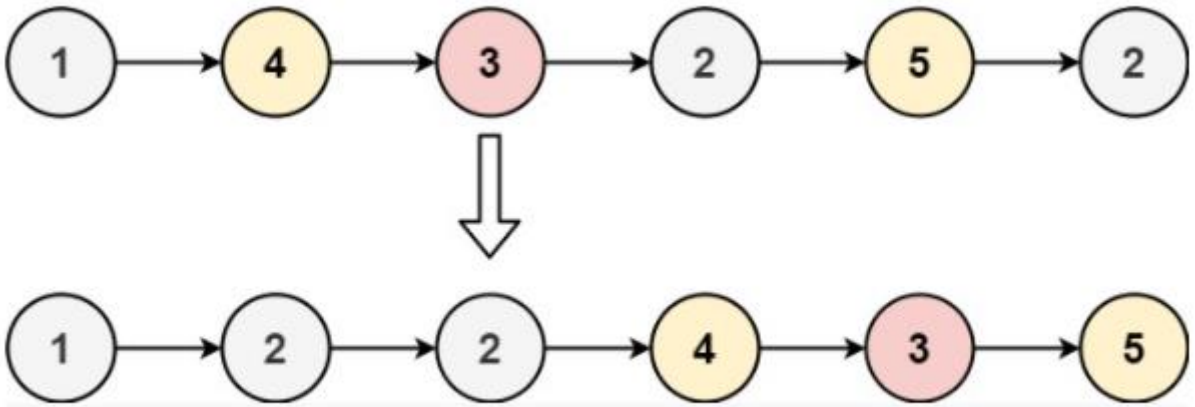
86. Partition List

Medium 2616 435 Add to List Share

Given the `head` of a linked list and a value `x`, partition it such that all nodes **less than** `x` come before nodes **greater than or equal** to `x`.

You should **preserve** the original relative order of the nodes in each of the two partitions.

Example 1:

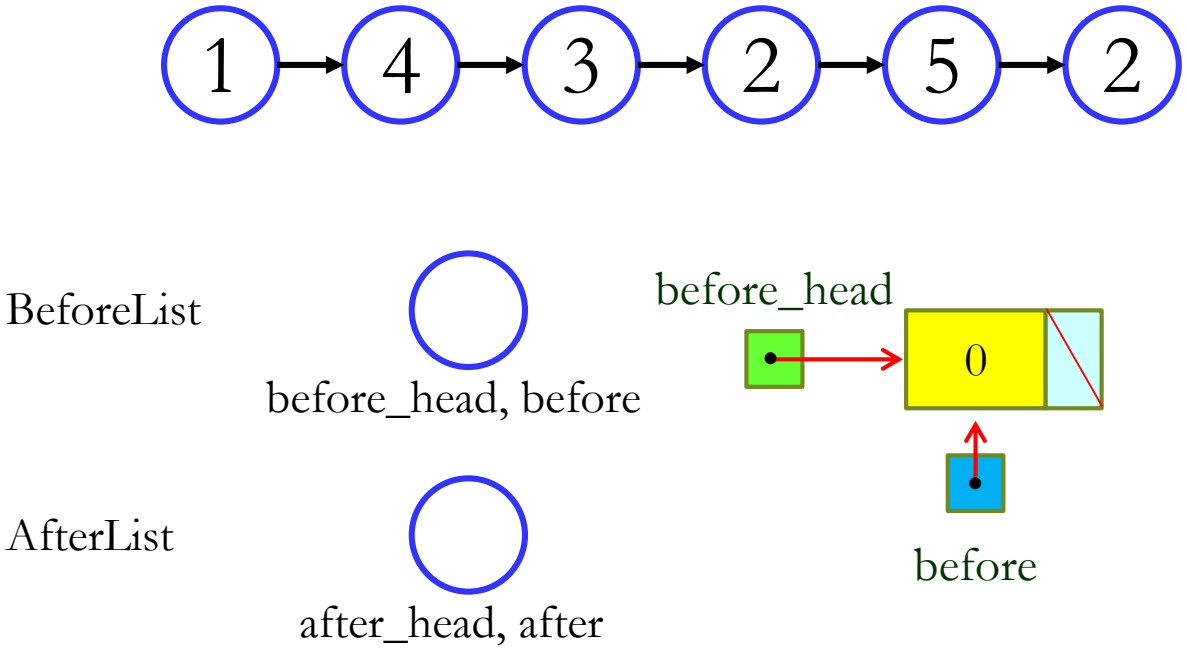


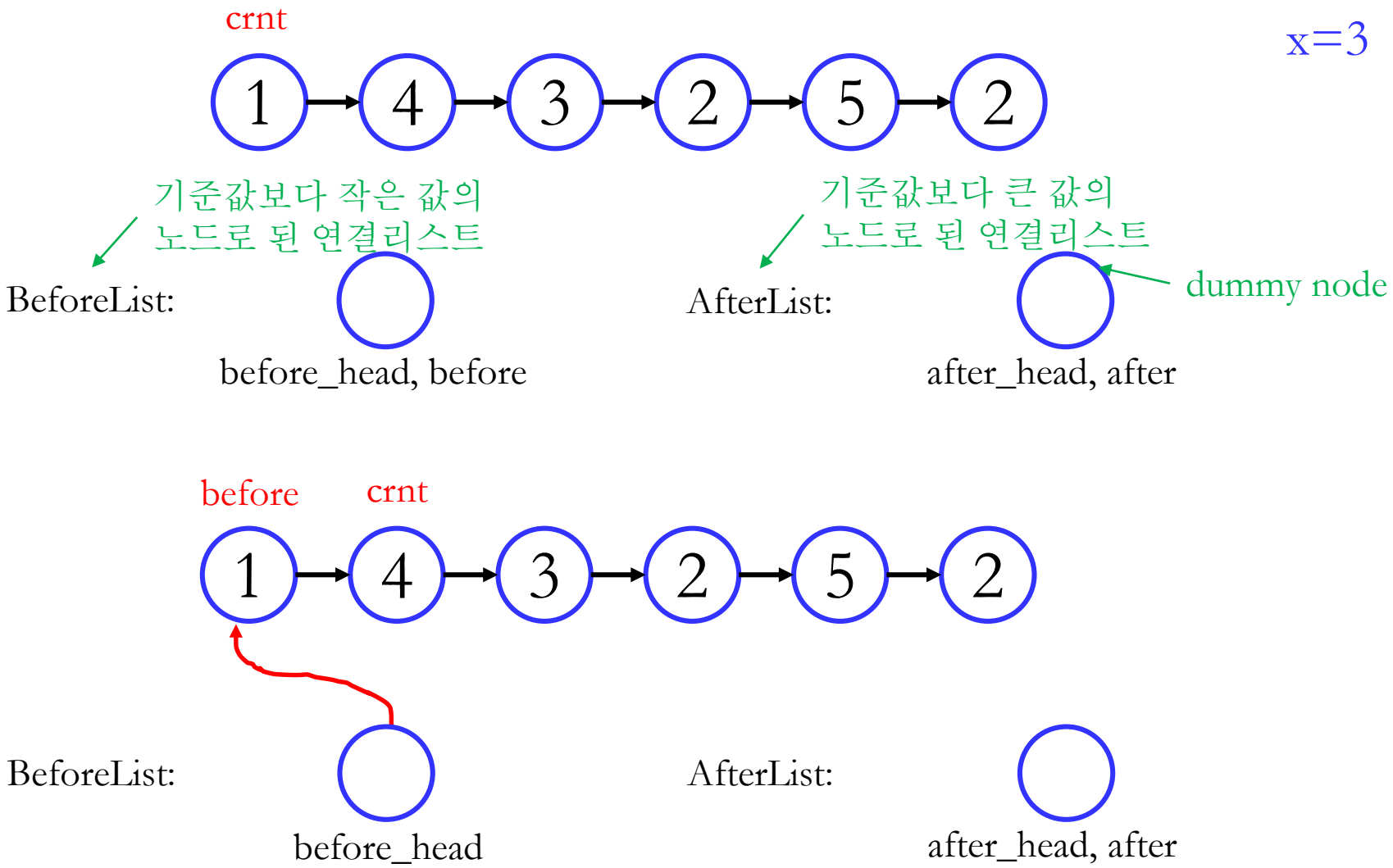

```
Input: head = [1,4,3,2,5,2], x = 3
Output: [1,2,2,4,3,5]
```

Example 2:

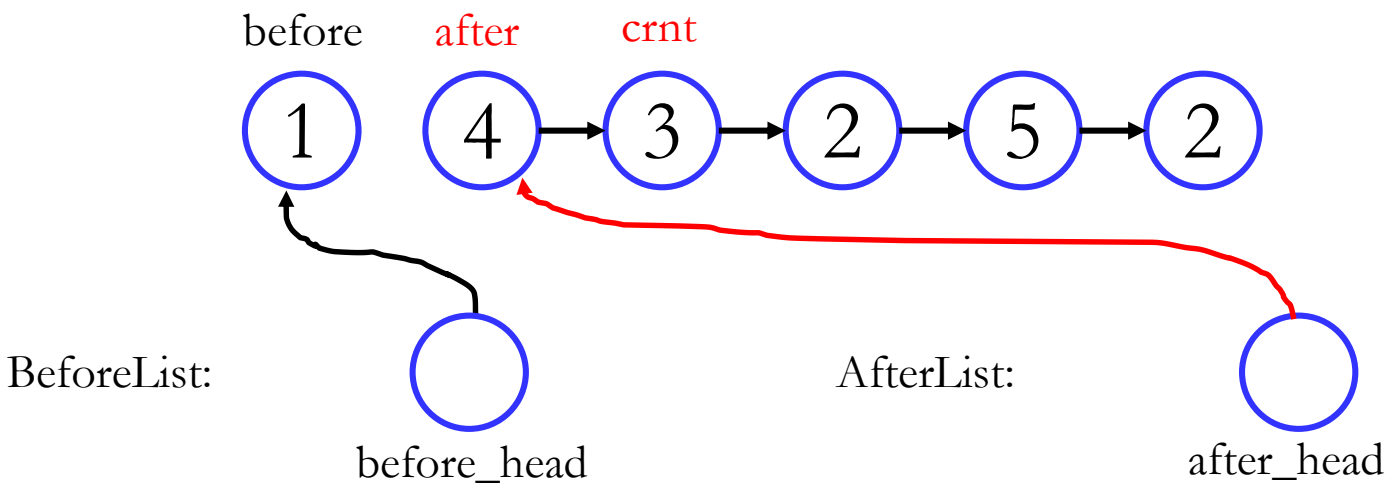
```
Input: head = [2,1], x = 2
Output: [1,2]
```

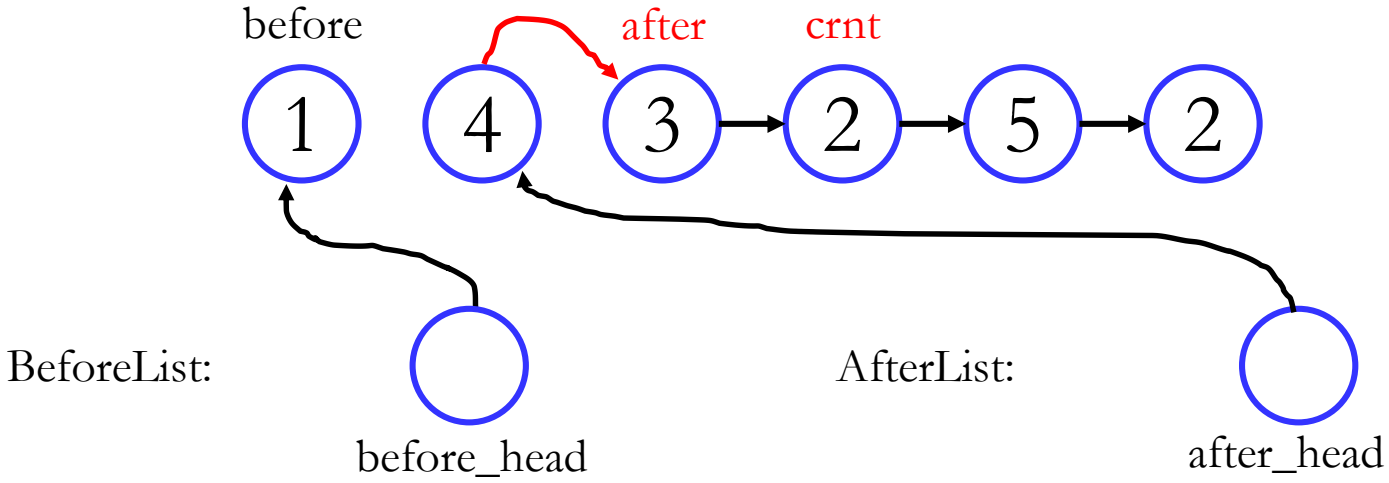
- 관찰:
  - 3보다 작은 값이 저장되는 BeforeList와 3보다 크거나 같은 값이 저장되는 AfterList을 만듦
  - before_head와 after_head는 두 연결리스트의 head 노드를 저장
  - before와 after는 두 연결리스트를 만들기 위해 이전 노드를 저장



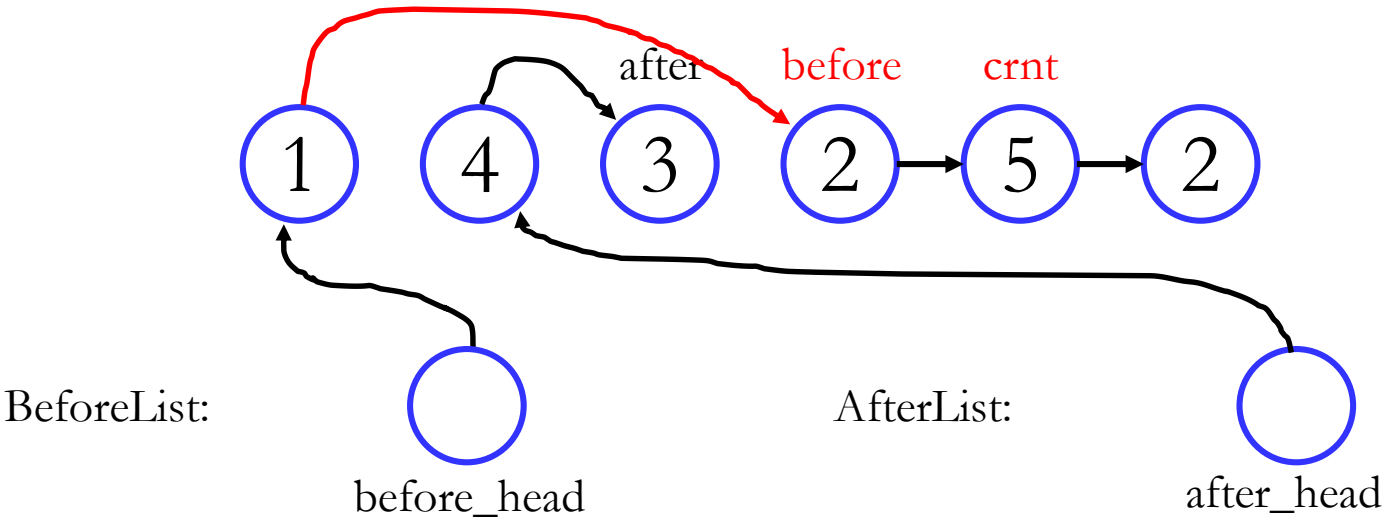

crnt
x=3
1
4
3
2
5
2
기준값보다 작은 값의
노드로 된 연결리스트
기준값보다 큰 값의
노드로 된 연결리스트
BeforeList:
AfterList:
dummy node
before_head, before
after_head, after
before
crnt
1
4
3
2
5
2
BeforeList:
AfterList:
before_head
after_head, after

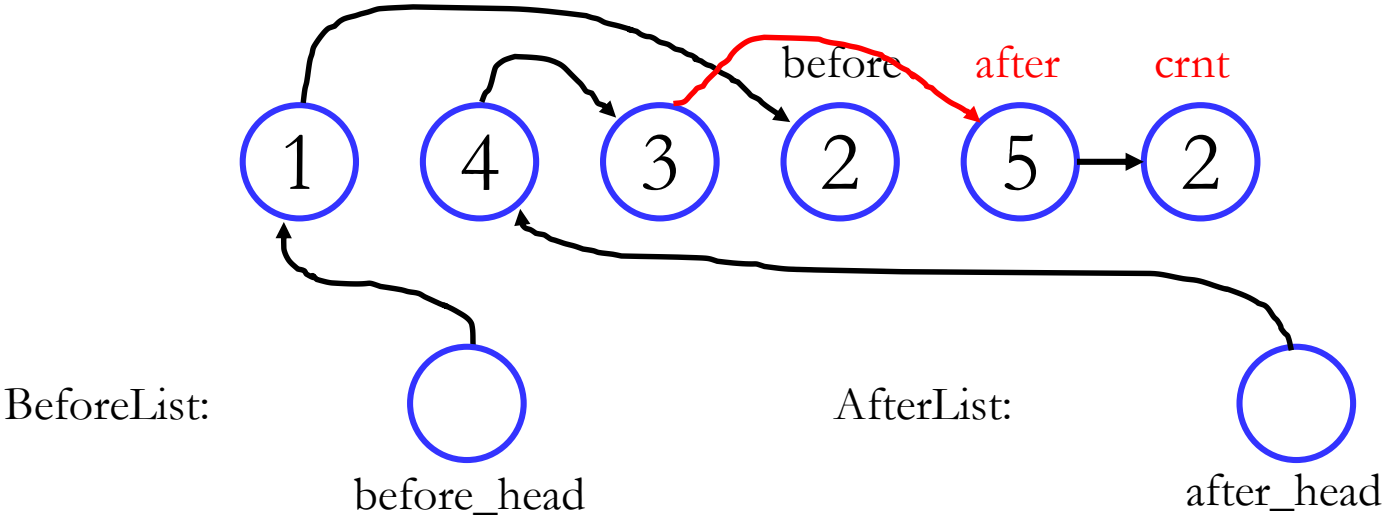
x=3


before
after
crnt
1
4
3
2
5
2
BeforeList:
AfterList:
before_head
after_head

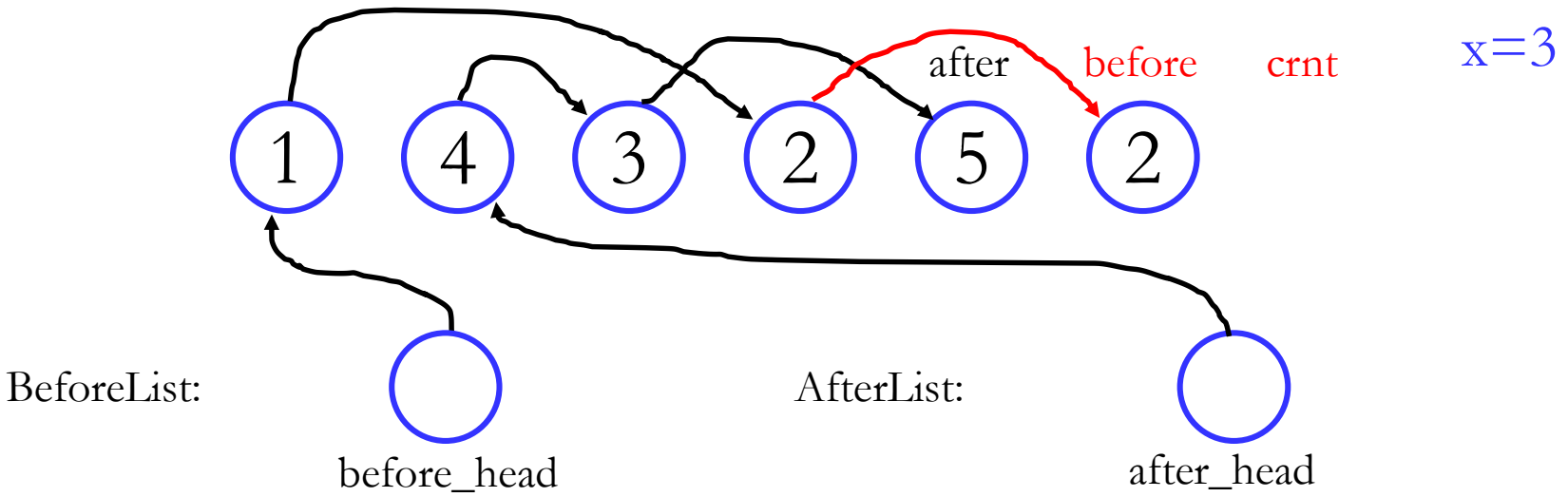

before
after
crnt
1
4
3
2
5
2
BeforeList:
AfterList:
before_head
after_head

x=3

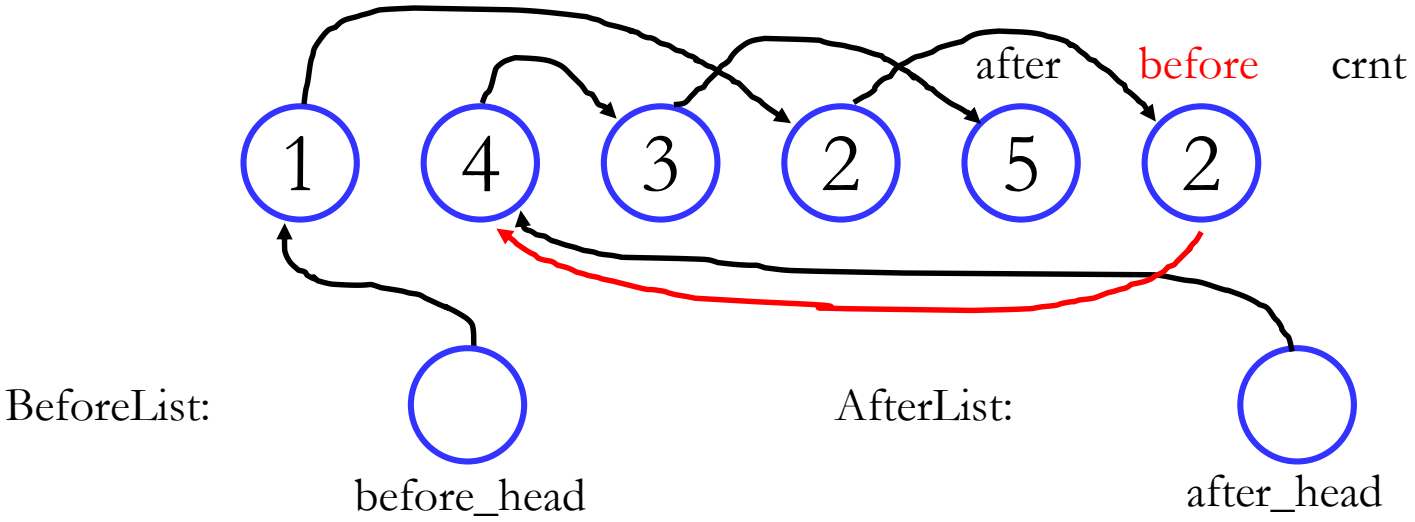

after
before
crnt
1
4
3
2
5
2
BeforeList:
AfterList:
before_head
after_head


before
after
crnt
1
4
3
2
5
2
BeforeList:
AfterList:
before_head
after_head

- crnt가 주어진 연결리스트의 끝을 만나면 종료



- before의 다음 노드를 after_head의 다음 노드로 갱신



- 풀이

```
def partition(self, head: ListNode, x: int) -> ListNode:
    before = before_head = ListNode(-1)
    after = after_head = ListNode(-1)

    crnt_node = head
    while crnt_node:
        val = crnt_node.val
        if x <= val:
            after.next = crnt_node
            after = after.next
            crnt_node = crnt_node.next
        else:
            before.next = crnt_node
            before = before.next
            crnt_node = crnt_node.next

    after.next = None
    before.next = after_head.next

    return before_head.next
```

시간복잡도: O(n)

143. Reorder List

143. Reorder List

Medium 3689 157 Add to List Share

You are given the head of a singly linked-list. The list can be represented as:

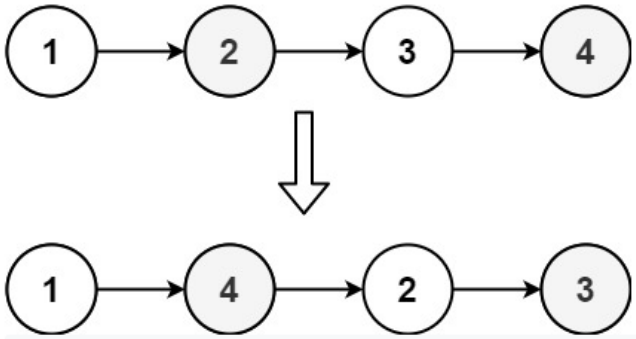
$$L_0 \rightarrow L_1 \rightarrow \dots \rightarrow L_{n-1} \rightarrow L_n$$

Reorder the list to be on the following form:

$$L_0 \rightarrow L_n \rightarrow L_1 \rightarrow L_{n-1} \rightarrow L_2 \rightarrow L_{n-2} \rightarrow \dots$$

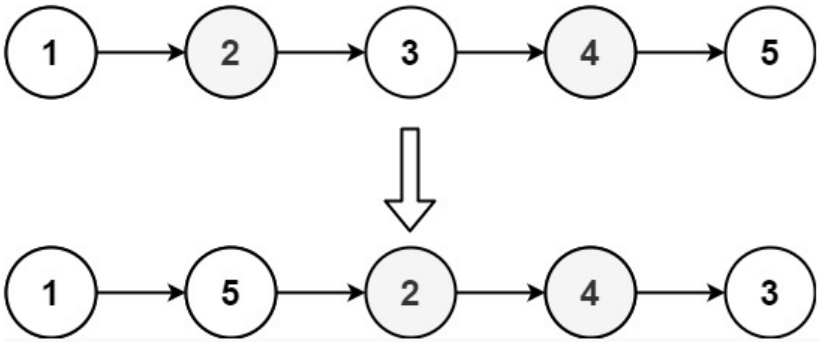
You may not modify the values in the list's nodes. Only nodes themselves may be changed.

Example 1:




```
Input: head = [1,2,3,4]
Output: [1,4,2,3]
```

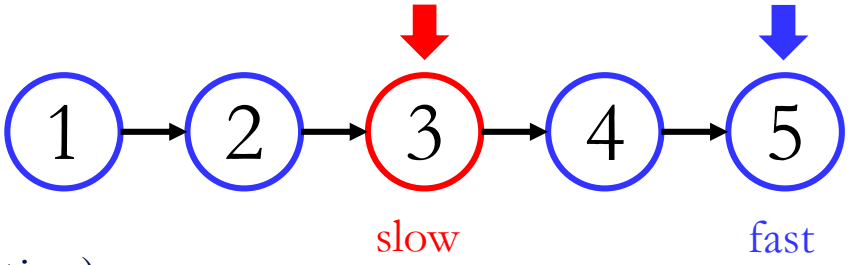
Example 2:



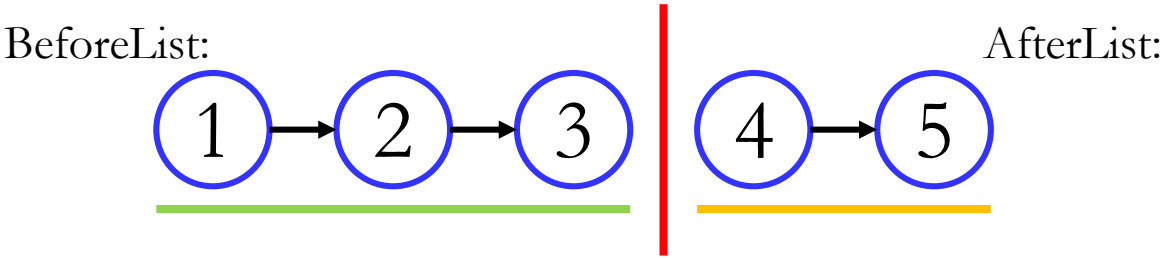

```
Input: head = [1,2,3,4,5]
Output: [1,5,2,4,3]
```

- 지금까지 배웠던 기본동작들을 이용

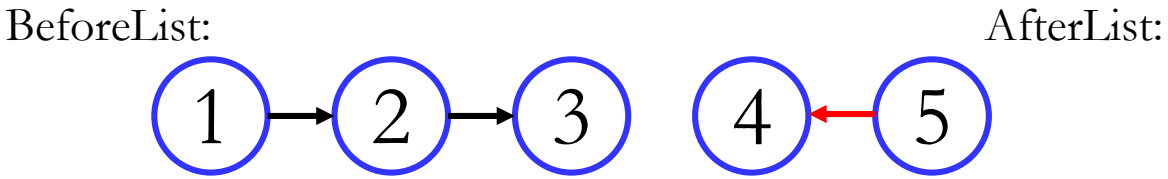
1. Middle



2. Split (Partition)



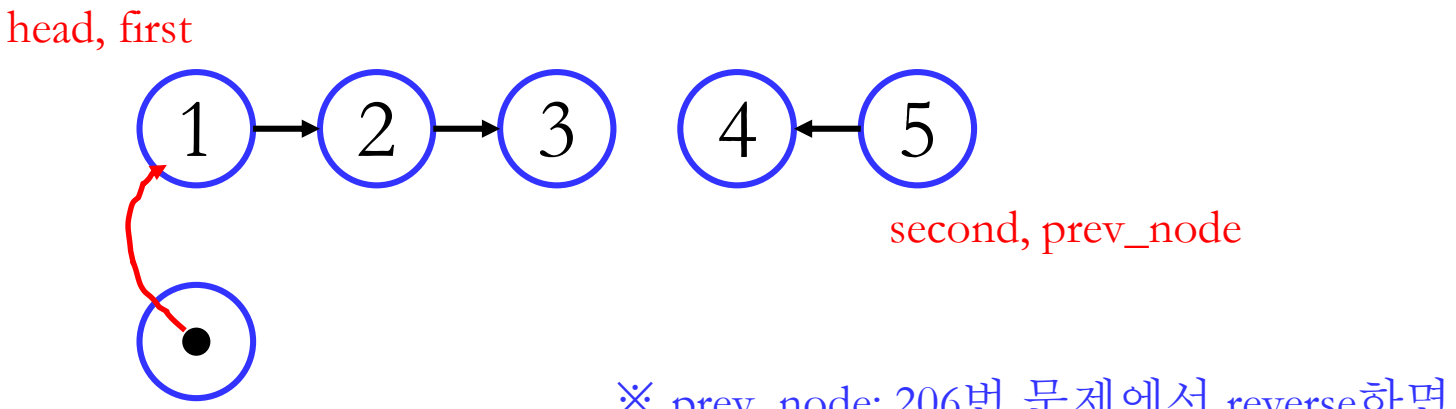
3. Reverse



4. Partition (Reorder)

가. 초기화:

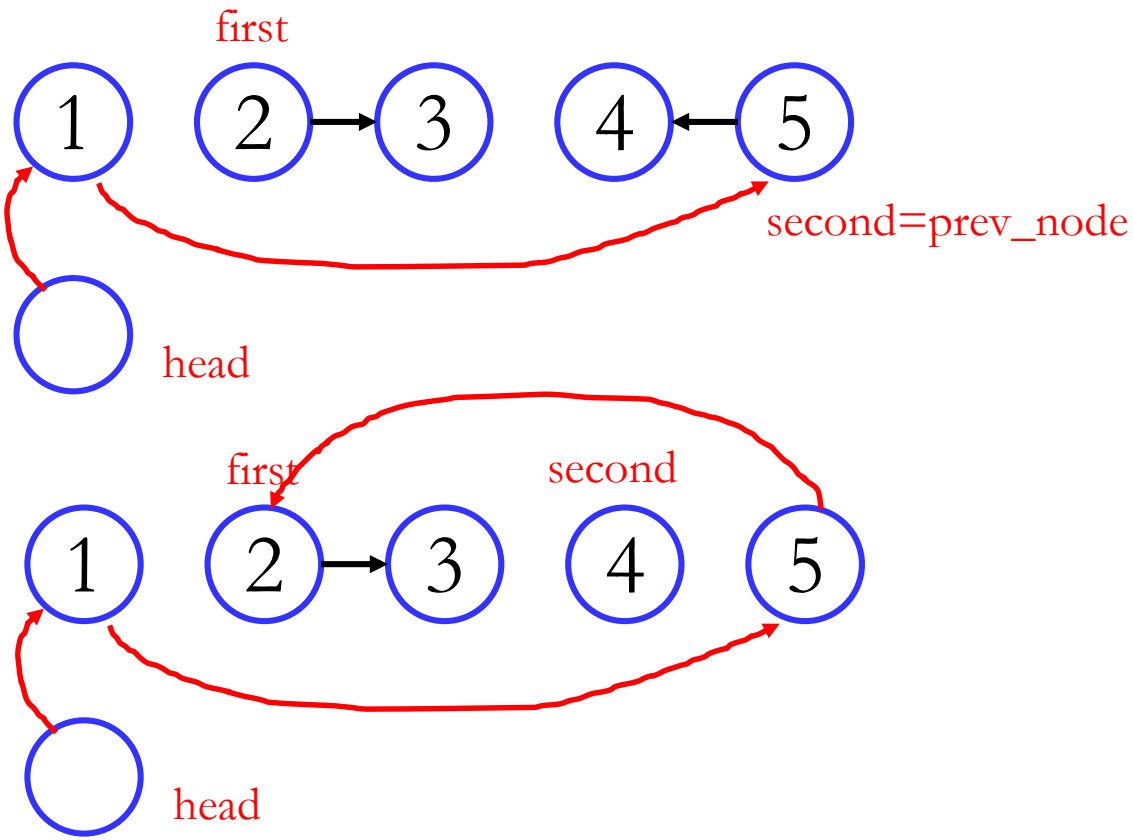
```
first, second = head, prev_node
```



※ prev_node: 206번 문제에서 reverse하면 prev_node가 head가 되었음을 상기

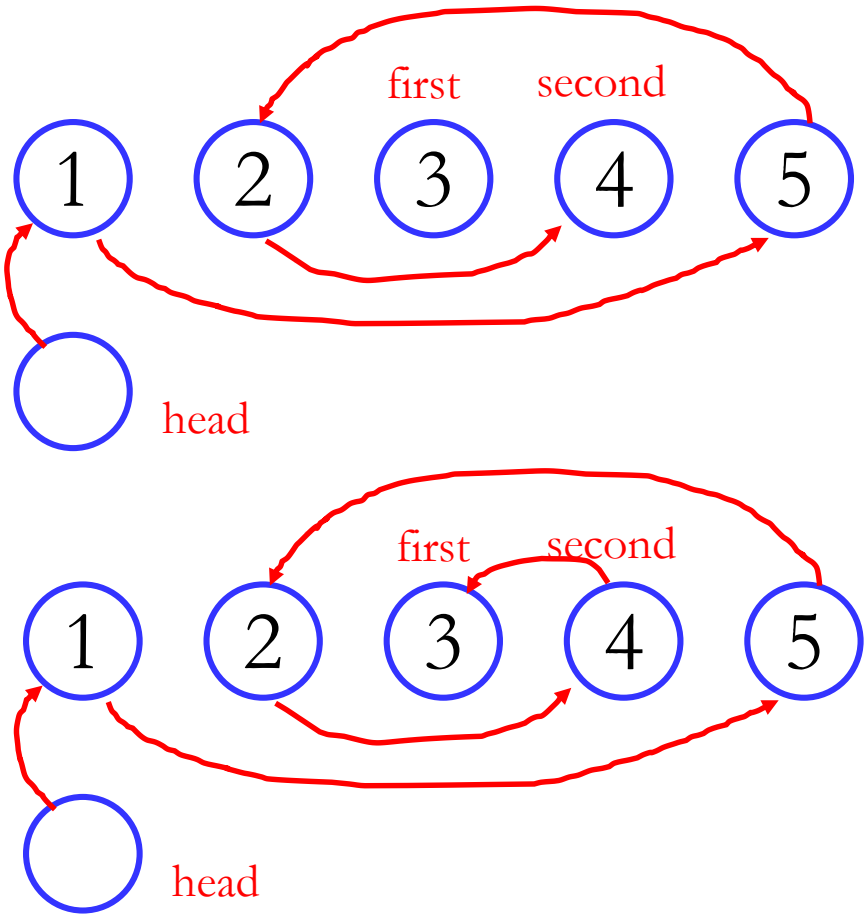
나. 1st iteration:

```
first.next, first = second, first.next
second.next, second = first, second.next
```



다. 2nd iteration:

```
first.next, first = second, first.next
second.next, second = first, second.next
```



```
def reorderList(self, head: ListNode) -> ListNode:
    if not head:
        return

    # find the middle of linked list [Problem 876]
    # in 1->2->3->4->5->6 find 4
    slow = fast = head
    while fast and fast.next:
        slow = slow.next
        fast = fast.next.next

    # reverse the second part of the list [Problem 206]
    # convert 1->2->3->4->5->6 into 1->2->3->4 and 6->5->4
    # reverse the second half in-place
    prev_node, crnt_node = None, slow
    while crnt_node:
        tmp_next_node = crnt_node.next
        crnt_node.next = prev_node
        prev_node = crnt_node
        crnt_node = tmp_next_node

    # merge two sorted linked lists [Problem 21]
    # merge 1->2->3->4 and 6->5->4 into 1->6->2->5->3->4
    first, second = head, prev_node
    while second.next:
        first.next, first = second, first.next
        second.next, second = first, second.next

    return head
```

감사합니다!

부산대학교
PUSAN NATIONAL UNIVERSITY