한방향 연결리스트

한방향 연결리스트: 노트(node)와 연결리스트 선언

1. 연결리스트는 기차에 비유하면 이해하기 쉽다.
    - 기차는 객차가 연결된 형태이다. 객차에는 승객(데이터)이 타고 있다.
    - 연결리스트에서는 객차를 노트(node)라 부르고, 객차를 연결하는 연결선을 에지(edge) 또는 링크(link)라 부른다.
2. 노트에는 **데이터를 위한 속성**과 **다음 노드에 대한 링크 정보**(노드의 주소)가 있어야 한다.
3. 가장 간단한 형태의 Node 클래스 선언

```
class Node:
  def __init__(self, key, value=None):
    self.key = key      # 노드에 저장되는 key 값으로 이 값으로 노드를 구분함
    self.value = value  # 추가 정보가 있다면 value에 저장함 (optional)
    self.next = None    # 다음에 연결될 노드(의 주소 또는 reference): 초기값은 None

  def __str__(self):    # print함수를 이용해 출력할 때의 출력 문자열 리턴
    return str(self.key)
```

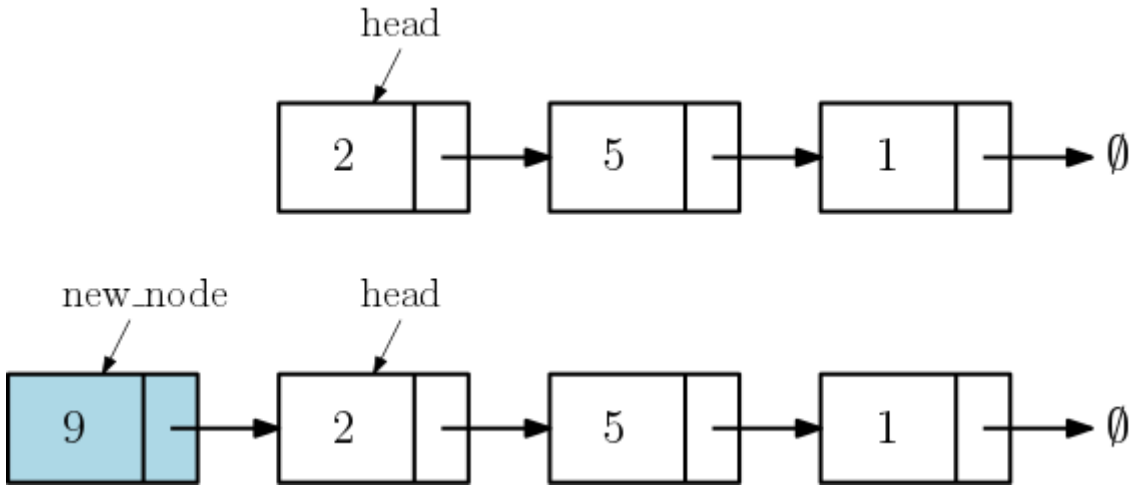
1. 이제 한방향 연결리스트 클래스 선언
    a. 연결리스트를 대표하는 노드는 가장 앞에 있는 head 노드 이므로, 이 head 노드를 기억해야 한다.
    b. 필요하다면 연결리스트에 연결된 노드의 수를 저장하는 size 정보도 포함 가능
2. 한방향 연결리스트 SinglyLinkedList 클래스 선언

```
class SinglyLinkedList:
  def __init__(self):
    self.head = None  # head 노드를 저장함
    self.size = 0     # 리스트의 노드 개수를 저장함

  def __str__(self):  # print() 출력용 문자열 리턴
    s = ""
    v = self.head
    while v:
      s += str(v.key) + " -> "
      v = v.next
    s += "None"
    return s

  def __len__(self):    # len(L): 리스트 L의 size 리턴
    return self.size
```

한방향 연결리스트 삽입+삭제 연산

리스트 가장 앞에 새로운 노드 삽입하기: pushFront: O(1)

- SinglyLinked 클래스의 메소드 중 하나로 아래 그림과 같이 새로운 노드를 삽입하는 연산

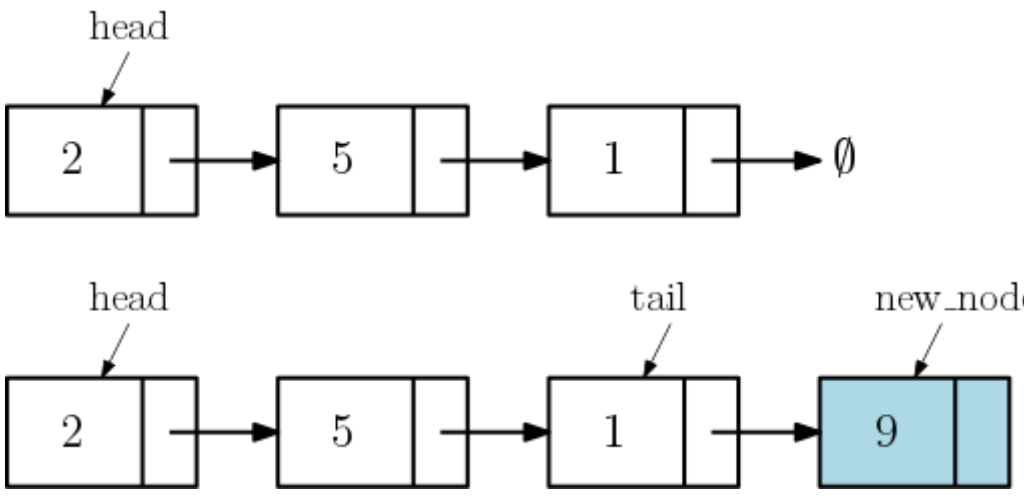



```
# class SinglyLinkedList의 메소드
  def pushFront(self, key, value=None):
    new_node = Node(key, value)
    new_node.next = self.head
    self.head = new_node      # head 노드가 바뀜
    self.size += 1
```

리스트의 tail 노드 다음에 삽입하기: pushBack: O(n)

- tail 노드 다음에 새로운 노드 삽입
  - 주의1: 빈 리스트라면 head = None이고 tail = None임에 유의!
  - 주의2: head 노드부터 링크를 따라 tail 노드를 찾아내야 함

```
def pushBack(self, key, value=None):
    new_node = Node(key, value)
    if self.size == 0:  # empty list --> new_node becomes a head!
      self.head = new_node
    else:
      tail = self.head
      while tail.next != None:  # follow links until tail
        tail = tail.next
      tail.next = new_node
    self.size += 1
```



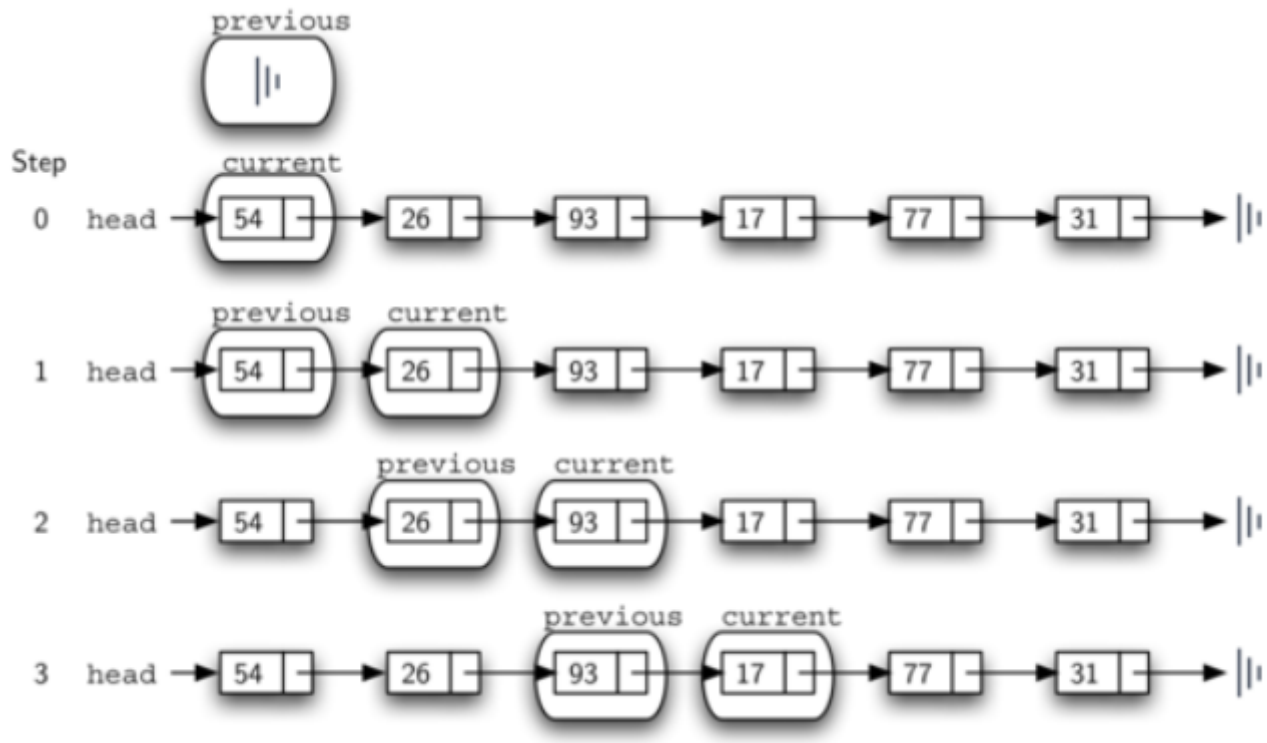
리스트의 head 노드 삭제하기: popFront: O(1)

- head가 가르키는 헤드 노드를 리스트에서 삭제한 후, 헤드 노드의 (key, value) 튜플 리턴
  - 만약에 빈 리스트라면 (None, None) 리턴

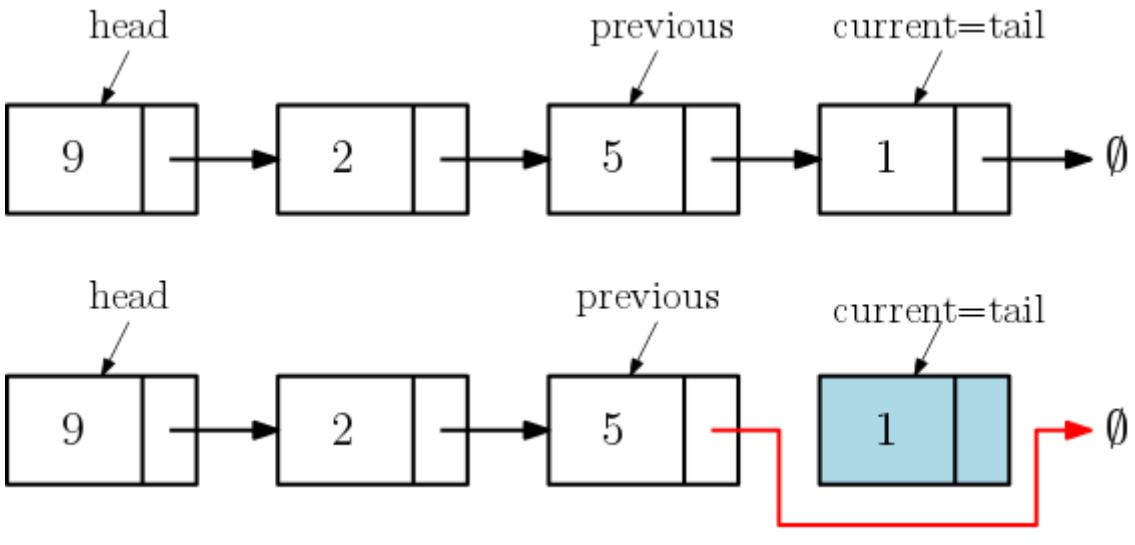
```
def popFront(self):
    key = value = None
    if len(self) > 0:
      key = self.head.key
      value = self.head.value
      self.head = self.head.next
      self.size -= 1
    return key, value
```

리스트의 가장 뒤(tail) 노드 삭제하기: popBack: O(n)

1. 가장 뒤에 있는 tail 노드를 리스트에서 삭제한 후 (key, value) 튜플 리턴
    a. 만약에 빈 리스트라면 (None, None) 리턴
    b. tail 노드와 tail 노드의 바로 전 previous 노드를 탐색 (previous 노드의 링크를 수정해야 하므로)
        i. previous = None, current = head로 초기화하여 아래 그림처럼 current가 마지막 노드에 도착할 때까지 링크를 하나씩 따라가면서 두 노드를 찾아냄
    c. 만약 리스트에 노드가 하나 뿐이라면? (head == tail이라면? 또는 previous == None이라면?) previous의 링크를 수정할 필요가 없어짐에 유의
    d. 리스트의 헤드 노드를 바꿔야 한다면(언제? head == tail인 경우) 수정 (다른 경우엔 헤드 수정할 필요 X)




```
def popBack(self):
    if self.size == 0:  # empty list (nothing to pop)
      return None, None
    else:
      # tail 노드와 그 전 노드인 previous를 찾는다
      previous, current = None, self.head
      while current.next != None:
        previous, current = current, current.next   # 한 노드씩 진행
      # 만약 리스트에 노드가 하나라면 그 노드가 head이면서 동시에 tail임
      # 그런 경우라면 tail을 지우면 빈 리스트가 되어 head = None으로 수정해야함!
      key, value = tail.key, tail.value
      tail = current
      if self.head == tail: # 또는 if previous == None:
        self.head = None
      else:
        previous.next = tail.next # previous가 새로운 tail이 됨!
      self.size -= 1
      return key, value
```



한방향 연결리스트 노드 탐색과 노드 제거

한방향 연결리스트에서 노드 차례로 방문하기: 탐색 search

1. 리스트의 노드를 차례로 방문하면서 연산은 매우 일반적임.
    a. 예를 들어, 모든 노드의 key 값을 차례대로 출력한다든지,
    b. key 값이 특정한 값을 갖는 노드를 탐색한다든지 하는 경우 등
2. 탐색함수 search(key)이면, key 값을 갖는 노드를 발견하면 그 노드를 리턴하고, 없다면 None 리턴

```
def search(self, key):
    v = self.head
    while v:
      if v.key == key:
        return v
      v = v.next
    return None (return v 해도 됨)
```

한방향 연결리스트에서 노드 차례로 방문하는 새로운 방법: generator 이용하기

- Python의 generator는 값들을 순차적으로 필요할 때마다 하나씩 생성하여 돌려주는 특수한 형태의 함수(또는 방복자: iterator)
- 예: 아래 코드에는 정수 n의 약수를 계산하는 두 함수가 설명되어 있다.
    - factor_fun은 일반적인 함수로 약수들을 모두 만들어 리스트에 담아 리턴
    - factor_gen은 generator로 약수들을 리스트로 만들어 리턴하지 않고, for문이 반복할 때마다 하나씩 값을 생성해(generate) for 문에 전달한다.
        - 이 경우엔 적은 메모리르 사용하고 훨씬 빠르게 동작
        - 생성하는 명령은 yield

```
def factor_fun(n):  # return a list of multiples of k in [1..n]
  results = [ ]
  for k in range(1, n+1):
    if n % k == 0:
      results.append(k)
  return results

def factor_gen(n):
  for k in range(1, n+1):
    if n % k == 0:
      yield k   # k를 한 번에 한 값씩 리턴한다

print("-----function-------")
for factor in factor_fun(100):
  print(factor, end=' ')
print("\n-----generator------")
for factor in factor_gen(100):
  print(factor, end=' ')
```

- 이제 SinglyLinkedList 클래스에 **generator**를 사용해보자.
  - 예를들어, 리스트에 관련된 코드를 찾아보자

```
a = [4, 3, -2, 9]
for x in a: # a의 첫 원소부터 시작해서 차례대로 x에 저장됨
    print(x)
```

- for 루프를 돌면서 처음엔 x = a[0], 다음 루프에선 x = a[1]이 지정되는 식으로 반복을 할 때마다 원소를 차례대로 지정한다.
- 이렇게 for 루프를 진행할 때마다 다음 원소를 가져와 지정해주는 함수는 리스트 클래스의 **__iter__** 라는 특별한 메소드이다.
  - 이 __iter__ 함수는 이미 작성되어 있으므로 우리는 신경쓸 필요 X
- 우리가 설계한 클래스의 특별한 메소드 __iter__를 작성하면, 헤드 노드부터 차례로 노드들을 for 루프를 통해 방문 가능
  - for 루프를 돌 때마다 **yield**로 전달된 객체가 다음 객체가 된다.
  - yield = return

```
def __iter__(self):
    v = self.head
    while v:
        yield v
        v = v.next
```

이러면 다음 코드가 작성 가능

```
L = SinglyLinkedList()
L.pushFront(10)
L.pushFront(20)
L.pushFront(2)
print(L)
for v in L: # 리스트와 for 루프를 사용하는 방식 그대로 사용 가능
    print(v, end = ' → ')
print('None')
```

- generator를 활용하면 search 함수를 다음과 같이 더 간단하게 작성 가능

```
def search(self, key):
    for v in self:
        if v.key == key:
            return v
    return None
```

한방향 연결리스트에서 특정 노드 삭제하기: remove

1. 특정 key 값을 갖는 노드를 리스트에서 삭제하는 함수 remove
   a. v = L.search(key) #key 값을 갖는 노드 v를 찾는다.
   b. L.remove(v) # L에서 노드 v를 삭제한다.
   c. [주의할 점]
      i. v가 None이라면?
      ii. v가 L.head라면?

iii. 위의 특수한 경우라면 주의해서 처리해야 함