

자료구조



더 복잡하게 연결된 구조



한국기술교육대학교
온라인평생교육원

학습내용

- 더 복잡하게 연결된 구조의 개념과 특징
- 원형 연결 구조 응용: 연결된 큐
- 이중 연결 구조 응용: 연결된 덱

학습목표

- 단순 연결 구조와 원형 연결된 구조의 차이를 설명할 수 있다.
- 원형 연결 구조로 큐를 구현할 수 있다.
- 이중 연결 구조로 덱을 구현할 수 있다.

더 복잡하게 연결된 구조의 개념과 특징



더 복잡하게 연결된 구조

더 복잡하게 연결된 구조의 개념과 특징

1 연결된 리스트의 종류

단순 연결 리스트의 개선 방향

- ▶ 마지막 노드의 링크가 `None`을 가리켜 낭비한다?
- ▶ 노드에서 2개 이상의 링크를 사용하면 더 편하지 않을까?

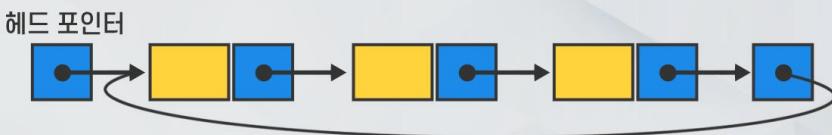
더 복잡하게 연결된 구조

더 복잡하게 연결된 구조의 개념과 특징

1 연결된 리스트의 종류

마지막 노드의 링크 활용

- ▶ 원형 연결 리스트(Circular Linked List)



둘 이상의 링크

- ▶ 이중 연결 리스트(Doubly Linked List)



더 복잡하게 연결된 구조의 개념과 특징



더 복잡하게 연결된 구조

더 복잡하게 연결된 구조의 개념과 특징

2 원형 연결 리스트

◎ 노드(Node)의 구조

원형 연결 리스트 노드(Node)의 구조

- 단순 연결 노드와 동일
- 하나의 링크 필드(Link Field)



더 복잡하게 연결된 구조

더 복잡하게 연결된 구조의 개념과 특징

2 원형 연결 리스트

◎ 노드(Node)의 구조

```
class Node:
    def __init__(self, elem, link):
        self.data = elem
        self.link = link
```



더 복잡하게 연결된 구조의 개념과 특징

더 복잡하게 연결된 구조

더 복잡하게 연결된 구조의 개념과 특징

2 원형 연결 리스트

원형 연결 리스트

- 마지막 노드의 링크가 첫 번째 노드를 가리킴



더 복잡하게 연결된 구조

더 복잡하게 연결된 구조의 개념과 특징

2 원형 연결 리스트

원형 연결 리스트의 마지막 노드 검사

- 마지막 노드의 링크가 **None**이 아님
- 마지막 노드의 링크가 **첫 번째 노드(헤드 포인터)**를 가리키는지를 통해 **마지막 노드 검사**를 할 수 있음
- 마지막 노드 검사 시, **무한 반복되는 알고리즘**이 나타날 수 있으므로 **구현에 주의**

더 복잡하게 연결된 구조의 개념과 특징



더 복잡하게 연결된 구조

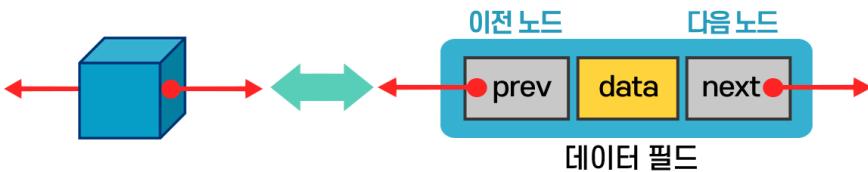
더 복잡하게 연결된 구조의 개념과 특징

3 이중 연결 리스트

◉ 노드(Node)의 구조

이중 연결 리스트 노드(Node)의 구조

- 이전 노드와 다음 노드를 가리키는 2개의 링크가 필요
- 이전 노드 = prev, 다음 노드 = next로 표현



더 복잡하게 연결된 구조

더 복잡하게 연결된 구조의 개념과 특징

3 이중 연결 리스트

◉ 노드(Node)의 구조

```
class DNode:
    def __init__(self, elem, prev, next):
        self.data = elem
        self.prev = prev
        self.next = next
```

더 복잡하게 연결된 구조의 개념과 특징



더 복잡하게 연결된 구조

더 복잡하게 연결된 구조의 개념과 특징

3 이중 연결 리스트

○ 이중 연결의 장단점

이중 연결의 장점	이중 연결의 단점
<ul style="list-style-type: none"> 선행 노드를 바로($O(1)$) 찾아갈 수 있음 단순 연결에서는 선행 노드 찾기가 $O(n)$에 가능 	<ul style="list-style-type: none"> 링크가 많으므로, 모든 연산에서 추가된 링크를 관리 링크가 많으므로, 오류 발생 가능성과 메모리 사용이 증가

더 복잡하게 연결된 구조

더 복잡하게 연결된 구조의 개념과 특징

3 이중 연결 리스트

이중 연결 리스트
<p>헤드 포인터</p> <ul style="list-style-type: none"> 마지막 노드와 첫 번째 노드가 서로 가리키고 있도록 함으로써, 이중 연결 리스트를 원형으로 구현 가능

원형 연결 구조 응용: 연결된 큐



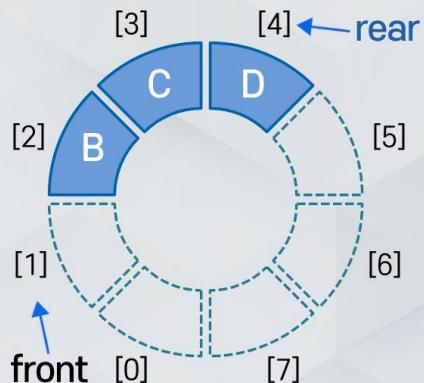
더 복잡하게 연결된 구조

원형 연결 구조 응용: 연결된 큐

1 원형으로 연결된 큐

배열 구조의 큐 (원형 큐)

- 삽입은 전단(front)
- 삭제는 후단(rear)



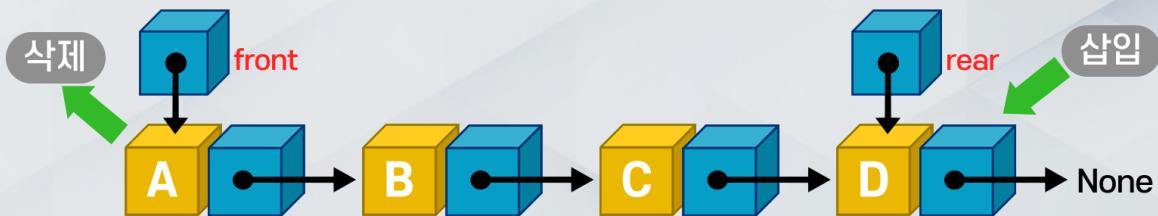
더 복잡하게 연결된 구조

원형 연결 구조 응용: 연결된 큐

1 원형으로 연결된 큐

연결된 구조의 큐

- 2개의 포인터가 필요함
 - front: 전단을 가리키는 포인터 (head)
 - rear: 후단을 가리키는 포인터 (tail)



원형 연결 구조 응용: 연결된 큐



더 복잡하게 연결된 구조

원형 연결 구조 응용: 연결된 큐

1 원형으로 연결된 큐

원형 연결 구조의 큐

연결된 구조의 큐

- rear의 link가 front를 가리키도록 함
- 두 개의 포인터**가 필요 없음

- 모든 항목들이 연결되어 있음
- 하나의 포인터로 처리 가능

포인터의 관리방법 1

front를 “head”란 이름으로 관리

포인터의 관리방법 2

rear를 “tail”이란 이름으로 관리

더 복잡하게 연결된 구조

원형 연결 구조 응용: 연결된 큐

1 원형으로 연결된 큐

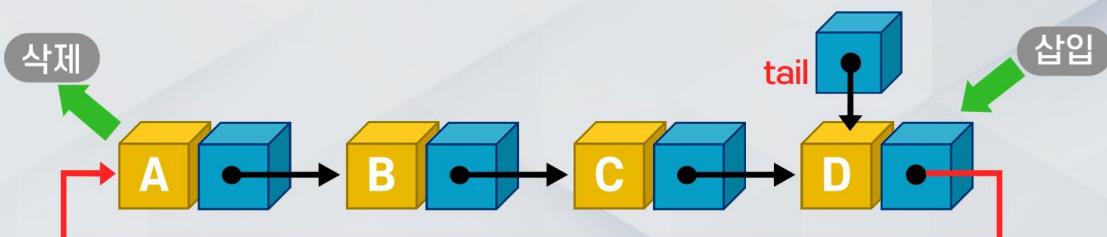
원형 연결 구조의 큐

포인터의 관리방법 2

rear를 “tail”이란 이름으로 관리

- » rear == tail
- » front == tail.link

→ front와 rear에 바로($O(1)$) 접근 가능하므로
포인터의 관리방법 2가 훨씬 효율적임





원형 연결 구조 응용: 연결된 큐

더 복잡하게 연결된 구조

원형 연결 구조 응용: 연결된 큐

1 원형으로 연결된 큐

원형 연결 구조의 큐

포인터의 관리방법 1

front를 “head”란 이름으로 관리

- » front == head
- » rear에 바로 접근 불가
- 하나씩 찾아서 $O(n)$ 시간 소요되므로 비효율적

더 복잡하게 연결된 구조

원형 연결 구조 응용: 연결된 큐

2 연결된 큐의 구현

클래스 정의와 멤버 변수

tail

꼬리(최근에 삽입된) 노드를 위한 포인터

count

큐의 항목 수(**선택사항**)

```
class CircularLinkedListQueue:
    def __init__( self ):
        self.tail = None
        self.count = 0
```



원형 연결 구조 응용: 연결된 큐

더 복잡하게 연결된 구조

원형 연결 구조 응용: 연결된 큐

2 연결된 큐의 구현

◎ 기본 연산들

» 항목의 수

```
def size( self ): return self.count
```

» 공백 상태 검사

```
def isEmpty( self ): return self.tail == None
```

» 포화 상태 검사

```
def isFull( self ): return False
```

더 복잡하게 연결된 구조

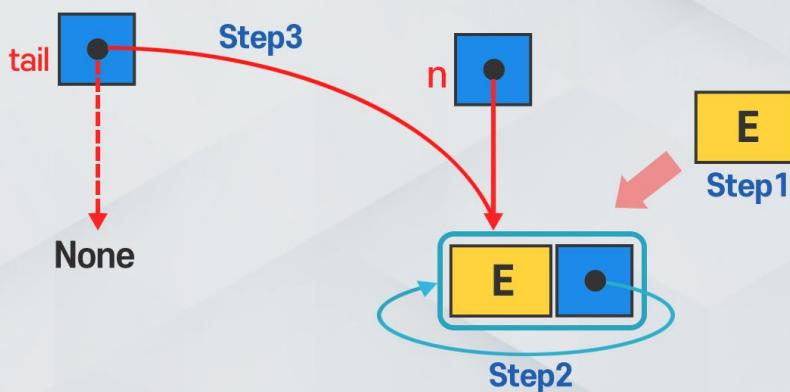
원형 연결 구조 응용: 연결된 큐

2 연결된 큐의 구현

◎ 삽입 연산

case 1

큐가 공백 상태인 경우의 삽입 연산



원형 연결 구조 응용: 연결된 큐



더 복잡하게 연결된 구조

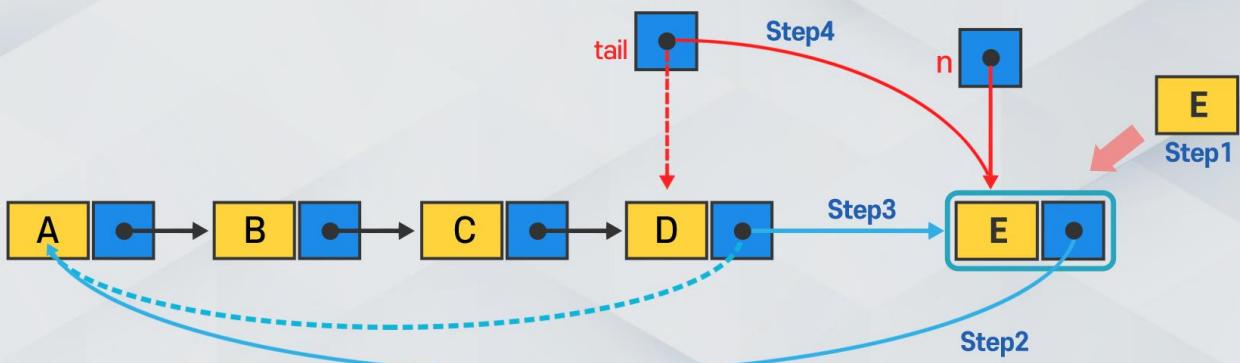
원형 연결 구조 응용: 연결된 큐

2 연결된 큐의 구현

◎ 삽입 연산

case 2

큐가 공백 상태가 아닌 경우의 삽입 연산



더 복잡하게 연결된 구조

원형 연결 구조 응용: 연결된 큐

2 연결된 큐의 구현

◎ 삽입 연산 알고리즘

```

def enqueue( self, item ):
    node = Node(item, None)
    if self.isEmpty() :
        self.tail = node
        node.link = node
    else :
        node.link = self.tail.link
        self.tail.link = node
        self.tail = node
    self.count += 1
  
```

원형 연결 구조 응용: 연결된 큐



더 복잡하게 연결된 구조

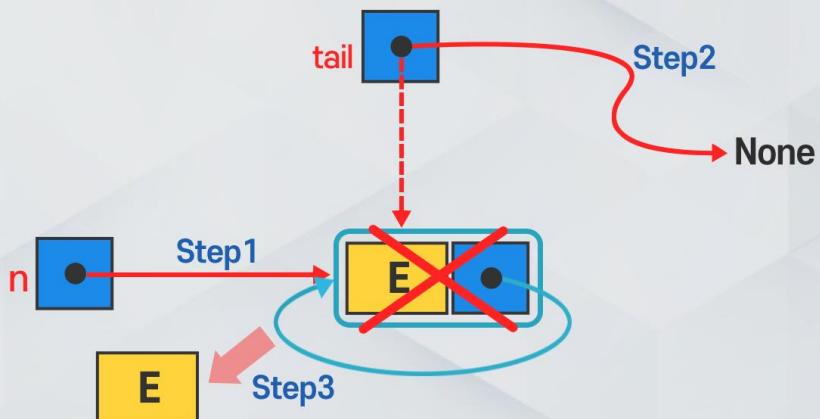
원형 연결 구조 응용: 연결된 큐

2 연결된 큐의 구현

◎ 삭제 연산

case 1

큐가 하나의 항목을 갖는 경우의 삭제 연산



더 복잡하게 연결된 구조

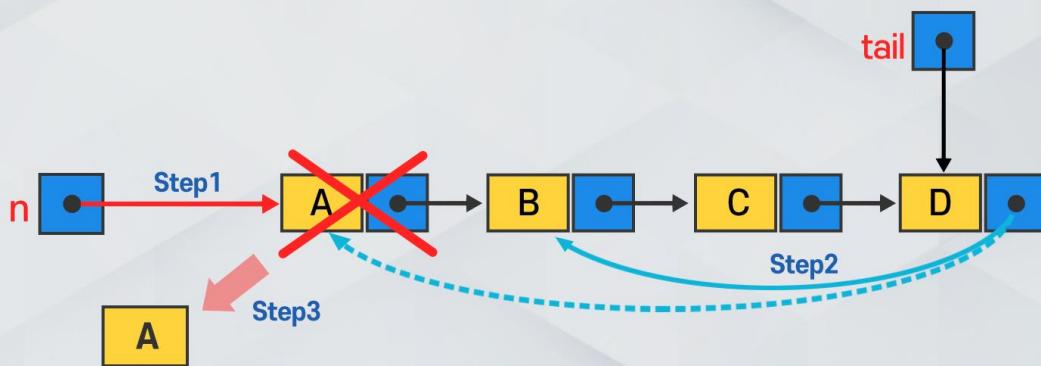
원형 연결 구조 응용: 연결된 큐

2 연결된 큐의 구현

◎ 삭제 연산

case 2

큐가 여러 개의 항목을 갖는 경우의 삭제 연산



원형 연결 구조 응용: 연결된 큐



더 복잡하게 연결된 구조

원형 연결 구조 응용: 연결된 큐

2 연결된 큐의 구현

◎ 삭제 연산 알고리즘

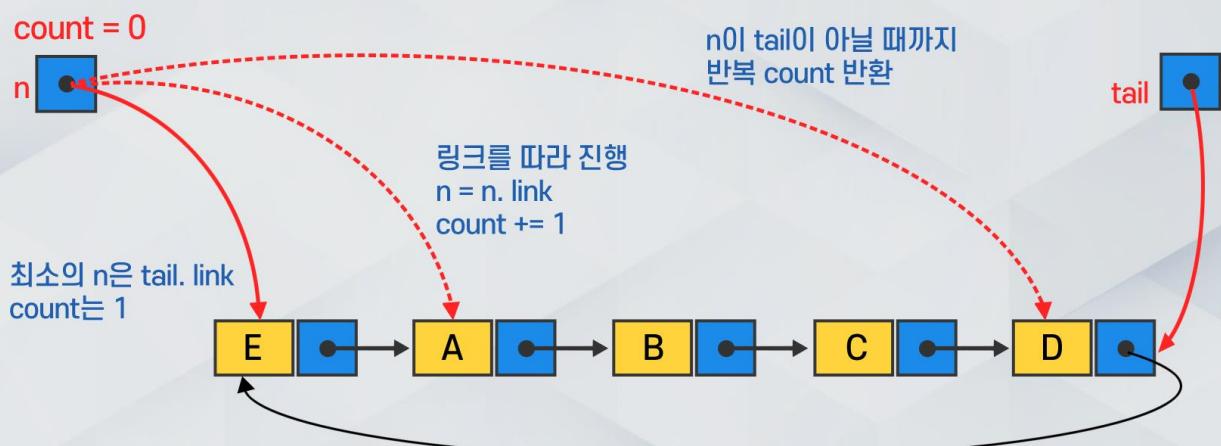
```
def dequeue( self ):
    if not self.isEmpty():
        self.count -= 1
        data = self.tail.link.data
        if self.tail.link == self.tail :
            self.tail = None
        else:
            self.tail.link = self.tail.link.link
    return data
```

더 복잡하게 연결된 구조

원형 연결 구조 응용: 연결된 큐

2 연결된 큐의 구현

◎ 전체 노드의 방문



원형 연결 구조 응용: 연결된 큐



더 복잡하게 연결된 구조

원형 연결 구조 응용: 연결된 큐

2 연결된 큐의 구현

◎ 전체 노드의 방문

```
def size_iter( self ):
    if self.isEmpty() : return 0
    else :
        count = 1
        node = self.tail.link
        while not node == self.tail :
            node = node.link
            count += 1
    return count
```

더 복잡하게 연결된 구조

원형 연결 구조 응용: 연결된 큐

2 연결된 큐의 구현

◎ 테스트 프로그램

```
if __name__ == "__main__":
    print('원형큐 테스트')
    q = CircularLinkedList()
    for i in range(9):
        q.enqueue(i)
    q.display(' enqueue()x9: ')
    print('   dequeue()-->', q.dequeue())
    print('   dequeue()-->', q.dequeue())
    print('   dequeue()-->', q.dequeue())
    q.display('   dequeue()x3: ')
    q.enqueue('홍길동')
    q.enqueue('이순신')
    q.enqueue('김연아')
    q.display('   enqueue()x3: ')
    print('   dequeue()-->', q.dequeue())
    q.display('   dequeue()x1 ')
    print('   peek()-->', q.peek())
```

C:\WINDOWS\system32#cmd.exe

원형 연결 구조의 큐 테스트

enqueue()x9: = [0 1 2 3 4 5 6 7 8]
dequeue()--> 0
dequeue()--> 1
dequeue()--> 2
dequeue()x3: = [3 4 5 6 7 8]
enqueue()x3: = [3 4 5 6 7 8] 홍길동 이순신 김연아
dequeue()--> 3
dequeue()x1 = [4 5 6 7 8] 홍길동 이순신 김연아
peek()--> 4

이중 연결 구조 응용: 연결된 덱

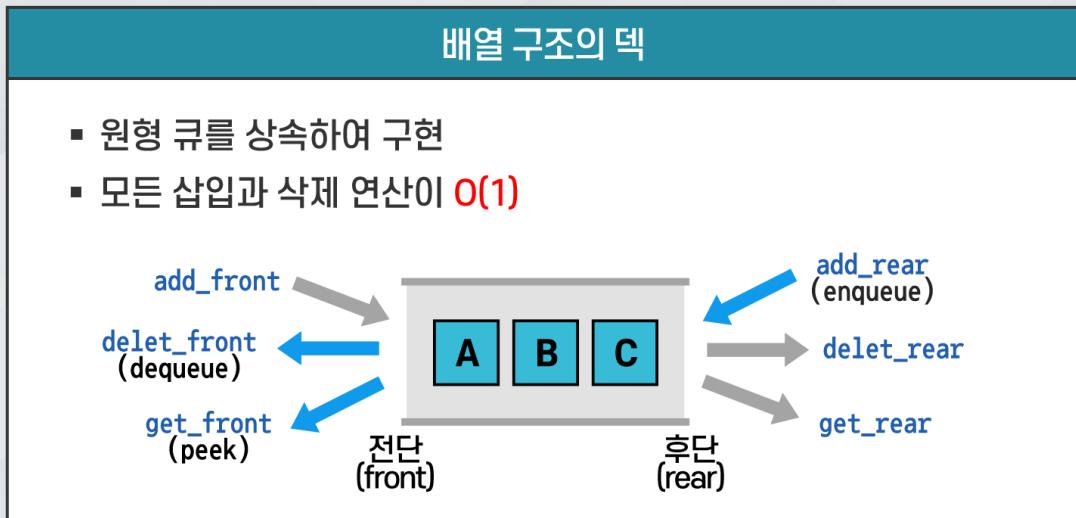


더 복잡하게 연결된 구조

이중 연결 구조 응용: 연결된 덱

1 연결된 구조의 덱

● 배열 구조의 덱



더 복잡하게 연결된 구조

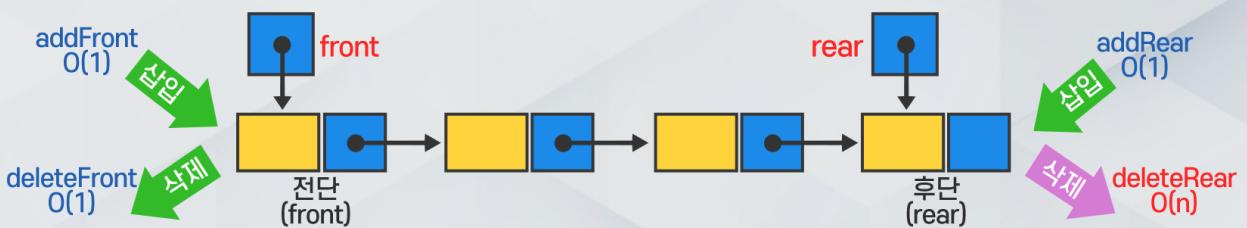
이중 연결 구조 응용: 연결된 덱

1 연결된 구조의 덱

● 단순 연결 구조의 덱

단순 연결 구조의 덱

- 대부분의 삽입과 삭제 연산은 $O(1)$ 시간이 소요
- **후단 삭제 연산**의 경우, $O(n)$!



▶ rear를 앞으로 한 칸 옮겨야 하는데, rear의 선형 노드를 바로 알 수가 없음

이중 연결 구조 응용: 연결된 덱

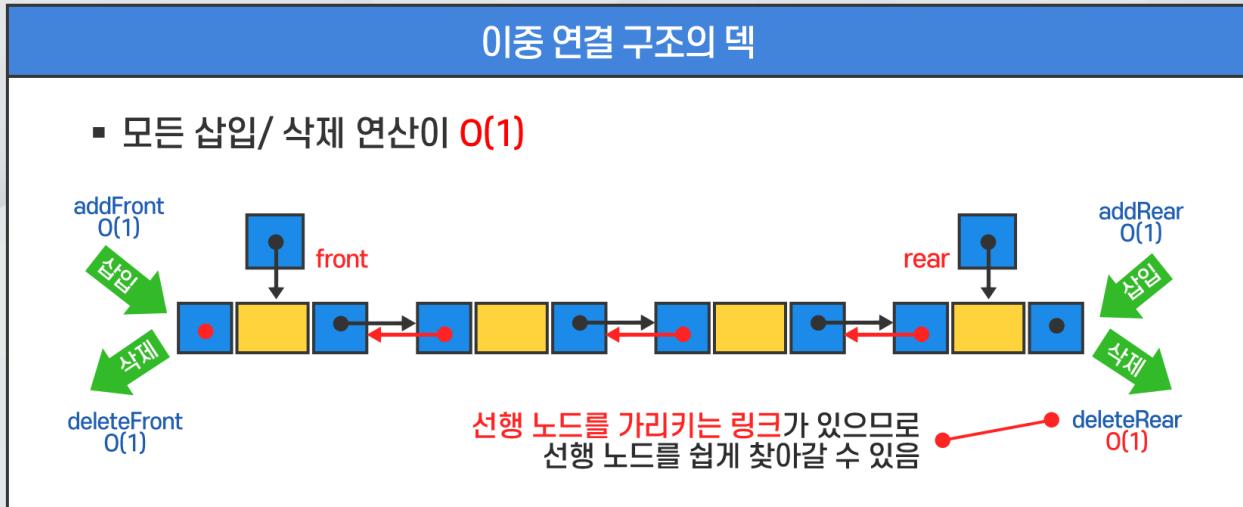


더 복잡하게 연결된 구조

이중 연결 구조 응용: 연결된 덱

1 연결된 구조의 덱

● 이중 연결 구조의 덱



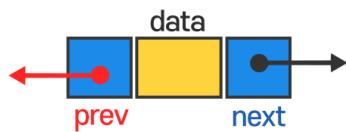
더 복잡하게 연결된 구조

이중 연결 구조 응용: 연결된 덱

2 이중 연결 구조의 덱 구현

● 이중 연결 구조를 위한 노드

```
class DNode:
    # 이중연결리스트를 위한 노드
    def __init__( self, elem, prev = None, next = None):
        self.data = elem
        self.prev = prev
        self.next = next
```





이중 연결 구조 응용: 연결된 덱

더 복잡하게 연결된 구조

이중 연결 구조 응용: 연결된 덱

2 이중 연결 구조의 덱 구현

● 클래스 정의와 멤버 변수

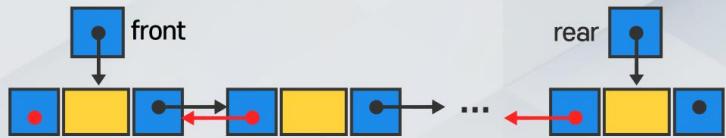
`front,rear`

전단과 후단 노드를 위한 포인터

`count`

덱의 항목 수(**선택사항**)

```
class DoublyLinkedListDeque:
    def __init__( self ):
        self.front = None
        self.rear = None
        self.count = 0
```



더 복잡하게 연결된 구조

이중 연결 구조 응용: 연결된 덱

2 이중 연결 구조의 덱 구현

● 기본 연산들

```
def isEmpty( self ): return self.front == None
def isFull ( self ): return False

def clear( self ):
    self.front = self.rear = None
    self.count == 0
```

이중 연결 구조 응용: 연결된 덱

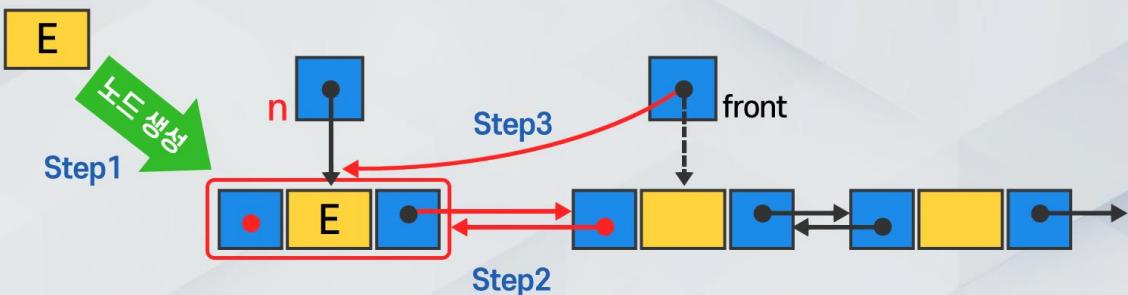


더 복잡하게 연결된 구조

이중 연결 구조 응용: 연결된 덱

2 이중 연결 구조의 덱 구현

- 전단 삽입: addFront()



더 복잡하게 연결된 구조

이중 연결 구조 응용: 연결된 덱

2 이중 연결 구조의 덱 구현

- 전단 삽입: addFront()

```
def addFront( self, item ):
    node = DNode(item, None, self.front)
    if( self.isEmpty()):
        self.front = self.rear = node
    else :
        self.front.prev = node
        self.front = node
    self.count += 1
```

이중 연결 구조 응용: 연결된 데크



더 복잡하게 연결된 구조

이중 연결 구조 응용: 연결된 데크

2 이중 연결 구조의 데크 구현

- 후단 삽입: addRear()

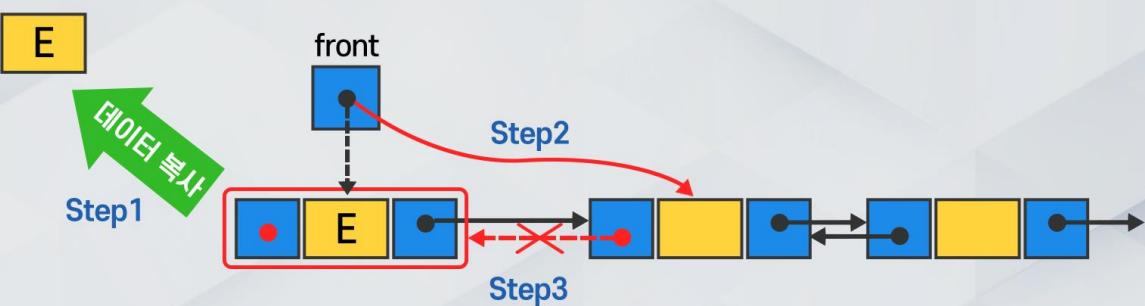
```
def addRear( self, item ):
    node = DNode(item, self.rear, None)
    if( self.isEmpty()):
        self.front = self.rear = node
    else :
        self.rear.next = node
        self.rear = node
    self.count += 1
```

더 복잡하게 연결된 구조

이중 연결 구조 응용: 연결된 데크

2 이중 연결 구조의 데크 구현

- 전단 삭제: deleteFront()



이중 연결 구조 응용: 연결된 덱



더 복잡하게 연결된 구조

이중 연결 구조 응용: 연결된 덱

2 이중 연결 구조의 덱 구현

- 전단 삭제: deleteFront()

```
def deleteFront( self ):
    if not self.isEmpty():
        self.count -= 1
        data = self.front.data
        self.front = self.front.next
        if self.front==None :
            self.rear = None
        else:
            self.front.prev = None
    return data
```

더 복잡하게 연결된 구조

이중 연결 구조 응용: 연결된 덱

2 이중 연결 구조의 덱 구현

- 후단 삭제: deleteRear()

```
def deleteRear( self ):
    if not self.isEmpty():
        self.count -= 1
        data = self.rear.data
        self.rear = self.rear.prev
        if self.rear==None :
            self.front = None
        else:
            self.rear.next = None
    return data
```

이중 연결 구조 응용: 연결된 덱



Solu션 탐색기 - 편집 보기 | 파일(F) 관리(R) 보기(V) 프로젝트(P) 디버그(D) 테스트(S) 브레이크(B) 도구(T) 확장(X) 찾(W) 도움말(H) 검색(Ctrl+Q) Python 3.6 (64-bit) 고체 파이썬

DequeueLinkedDoubly.py Node.py

```

9     from Node import DNode
10
11 class DoublyLinkedListDeque:
12     def __init__( self ):
13         self.front = None
14         self.rear = None
15         self.count = 0
16
17     def isEmpty( self ):
18         return self.front == None
19         # return self.count == 0
20
21     def clear( self ):
22         self.front = self.rear = None
23         self.count = 0
24
25     def addFront( self, item ):
26         node = DNode(item, None, self.front)
27         if( self.isEmpty() ):
28             self.front = self.rear = node
29         else :
30             self.front.prev = node

```

출처 보기 선택: 디버그 python.exe 프로그램이 종료되었습니다(코드: 1 (0x1)).

오류 목록 영업 창 출처

실습 단계

이중 연결 구조의 덱을 위해 DNode 클래스 필요

isEmpty, isFull, addFront, addRear, deleteRear 구현

deleteRear의 단순 연결 구조와 이중 연결 구조에서의 구현 시 다른 시간 복잡도

테스트 프로그램 출력 결과