

# <자료구조>

## 7. 연결 리스트 II

한국외국어대학교  
컴퓨터.전자시스템공학전공  
2016년 1학기  
고 석 훈

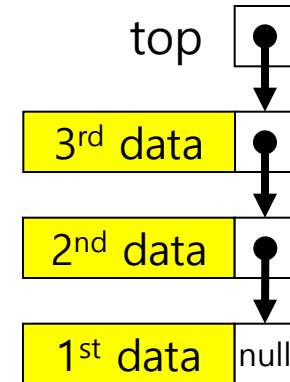
# 학습 목표

- 연결 리스트로 구현하는 스택과 큐
- 원형 연결 리스트의 구조를 이해하고, 삽입/삭제 알고리즘을 이해한다.
- 이중 연결 리스트의 구조를 이해하고, 삽입/삭제 알고리즘을 이해한다.

# 스택 구현: 연결 자료구조 [1/4]

## ● 단순 연결 리스트를 이용하여 구현

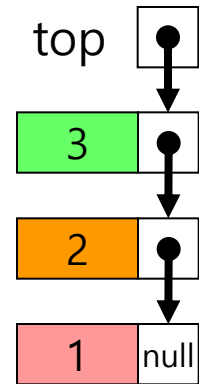
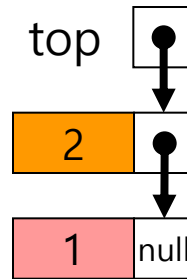
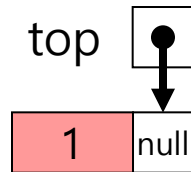
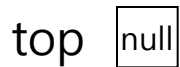
```
typedef struct _node {  
    int      data;  
    struct _node *link;  
} node;  
  
node* top;
```



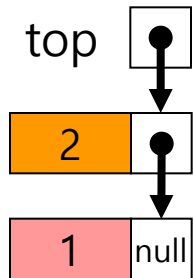
- 스택의 원소 : 단순 연결 리스트의 노드
- 변수 top : 단순 연결 리스트의 마지막 노드를 가리키는 포인터
  - ◆ 초기 상태 :
- push : 리스트의 맨 앞에 노드 삽입
- pop : 리스트의 맨 앞의 노드 삭제

# 스택 구현: 연결 자료구조 [2/4]

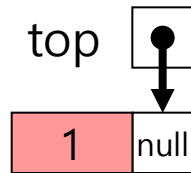
① create(stack, 5)    ② push(stack, 1);    ③ push(stack, 2);    ④ push(stack, 3);



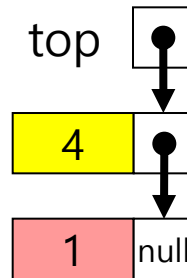
⑤ pop(stack);



⑥ pop(stack);

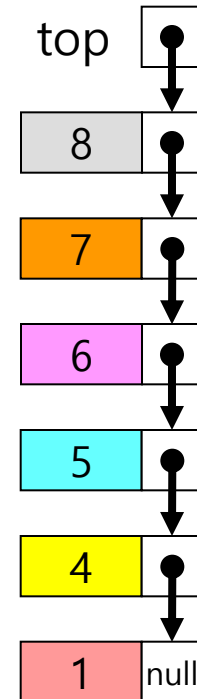
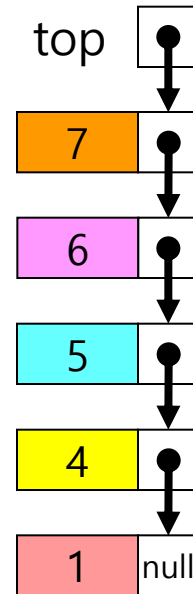
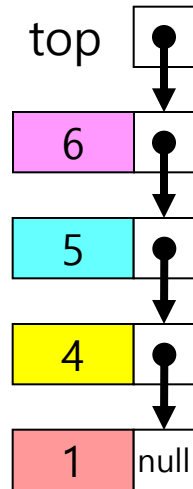
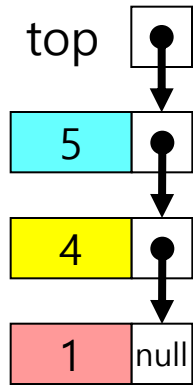


⑦ push(stack, 4);



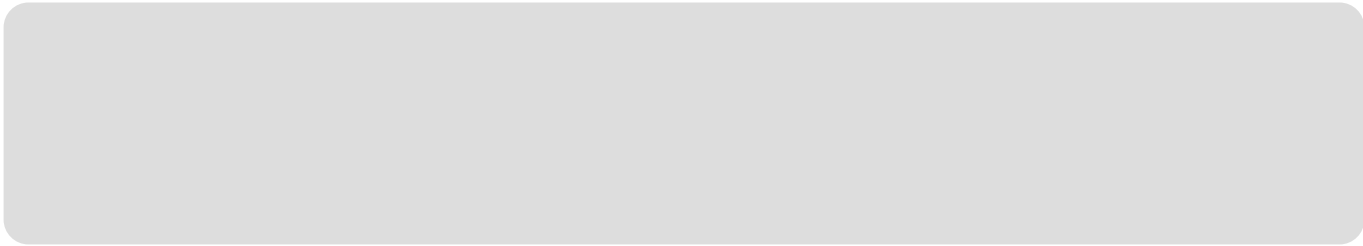
# 스택 구현: 연결 자료구조 [3/4]

⑧ push(stack, 5);    ⑨ push(stack, 6);    ⑩ push(stack, 7);    ⑪ push(stack, 8);



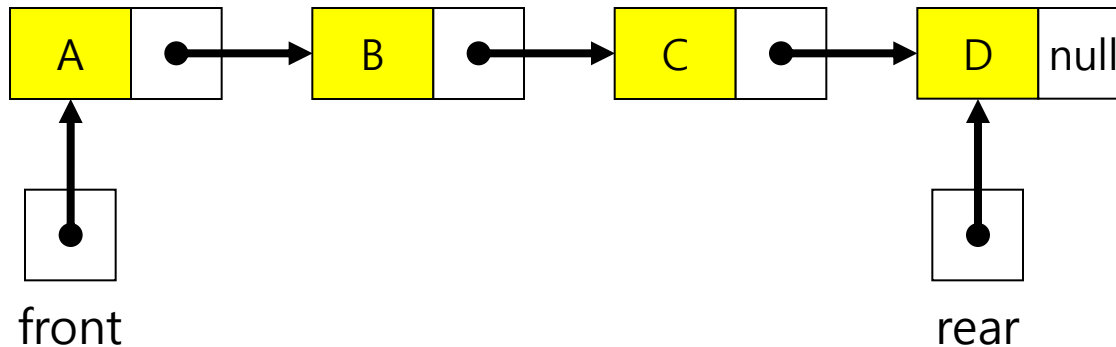
# 스택 구현: 연결 자료구조 [4/4]

- 연결 자료구조로 구현한 스택의 특징
  - 메모리가 허용하는 한 스택의 크기 제한 없음
  - 연결 자료구조의 장점을 그대로 가지고 있다.
  - 스택에 저장되어 있는 원소의 갯수는?



# 큐 구현: 연결 자료구조 [1/6]

- 단순 연결 리스트를 이용한 큐
  - 큐의 원소 : 단순 연결 리스트의 노드
  - 변수 front : 첫 번째 노드를 가리키는 포인터 변수
  - 변수 rear : 마지막 노드를 가리키는 포인터 변수
  - 초기 상태 (= 공백 상태) : front = null, rear = null



# 큐 구현: 연결 자료구조 [2/6]

- 연결 큐의 공백 검사 알고리즘
  - 공백 상태 : front = null

```
isEmpty(LQ)
    if (front = null) then return true;
    else return false;
end isEmpty( )
```



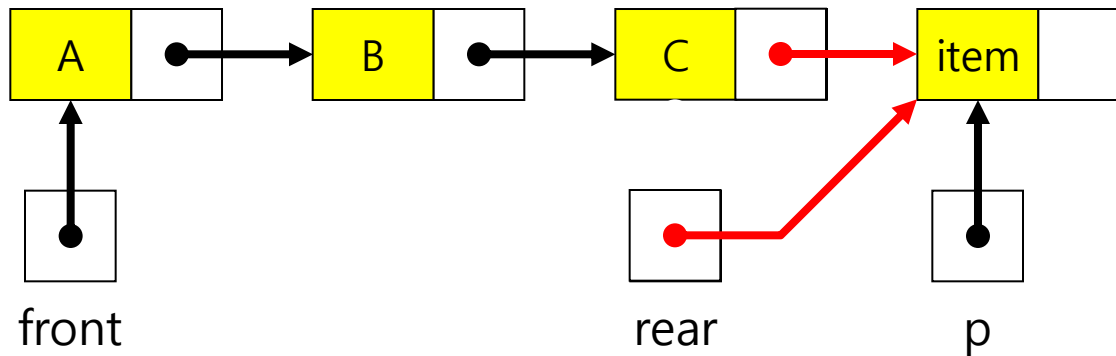
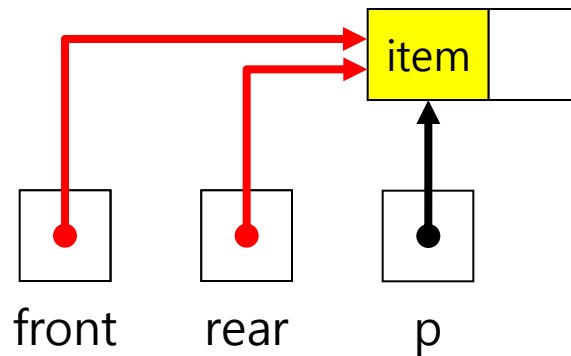
# 큐 구현: 연결 자료구조 [3/6]

## ● 연결 큐의 삽입 알고리즘

```
enqueue(LQ, item)  
  p ← getNode( );  
  p.data ← item;  
  p.link ← null;  
  if (front = null) then {  
    rear ← p;  
    front ← p;  
  }  
  else {  
    rear.link ← p;  
    rear ← p;  
  }  
end enqueue( )
```

# 큐 구현: 연결 자료구조 [4/6]

## ● 연결 큐의 삽입 알고리즘



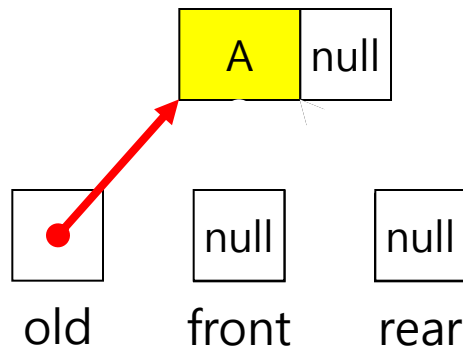
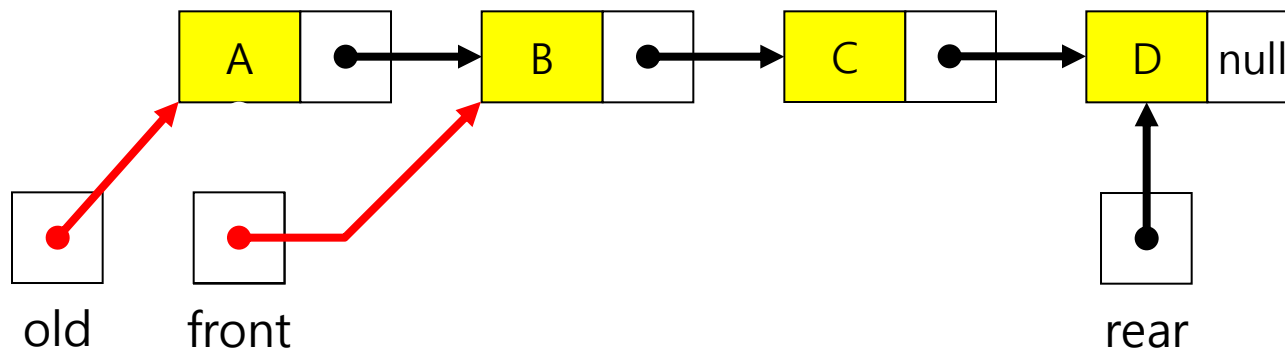
# 큐 구현: 연결 자료구조 [5/6]

## ● 연결 큐의 삭제 알고리즘

```
deQueue(LQ)  
  if (isEmpty(LQ)) then Queue_Empty( );  
  old ← front;  
  item ← front.data;  
  front ← front.link;  
  if (isEmpty(LQ)) then rear ← null;  
  returnNode(old);  
  return item;  
end deQueue( )
```

# 큐 구현: 연결 자료구조 [6/6]

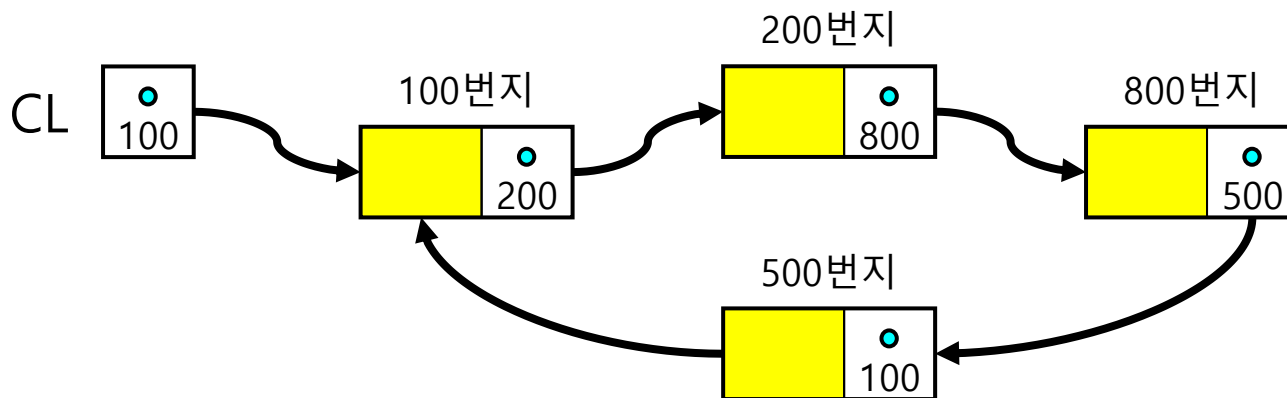
## ● 연결 큐의 삭제 알고리즘



# 원형 연결 리스트(Circular Linked List)

## ● 원형 연결 리스트(circular linked list)

- 단순 연결 리스트에서 마지막 노드가 리스트의 첫번째 노드를 가리키게 하여 리스트의 구조를 원형으로 만든 연결 리스트
- 단순 연결 리스트의 마지막 노드의 링크 필드에 첫번째 노드의 주소를 저장하여 구성
- 링크를 따라 계속 순회하면 이전 노드에 접근 가능



# 원형 연결: 첫번째 노드 삽입 [1/8]

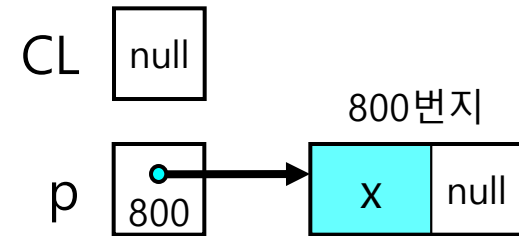
- 원형 연결 리스트 CL에 x 값을 갖는 노드 p를 삽입하는 알고리즘
  - 마지막 노드의 링크를 첫번째 노드로 연결하는 부분만 제외하고는 단순 연결 리스트에서의 삽입 연산과 같은 연산

```
insertFirstNode(CL, x)
  p ← getNode();
  p.data ← x;
  if (CL = null) then {                                // ①
    CL ← p;                                           // ②
    p.link ← p;                                       // ③
  }
  else {
    last ← CL;                                       // ④
    while (last.link ≠ CL) do                         // ⑤
      last ← last.link;
    p.link ← last.link;                             // ⑥
    last.link ← p;                                   // ⑦
    CL ← p;                                          // ⑧
  }
end insertFirstNode()
```

# 원형 연결: 첫번째 노드 삽입 [2/8]

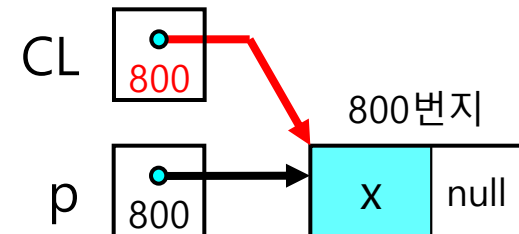
① **if** (CL = null) **then**

- 원형 리스트 CL이 공백 리스트인 경우



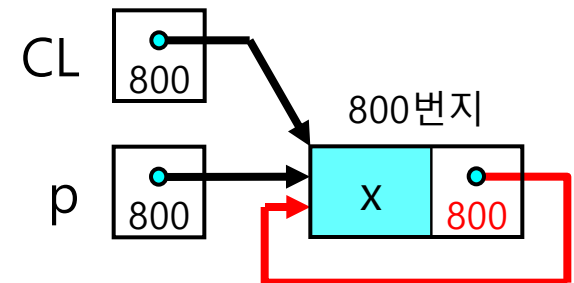
②  $CL \leftarrow p;$

- 원형 리스트 포인터 CL에 새 노드 p의 주소를 저장



③  $p.link \leftarrow p;$

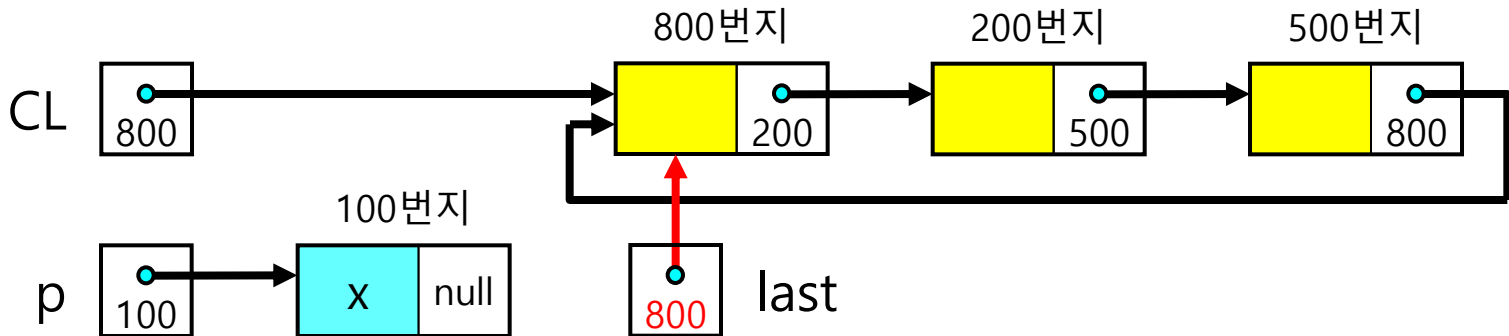
- 노드 p가 자신을 가리키게 하여 노드 p를 첫번째 노드이자 마지막 노드가 되도록 지정



# 원형 연결: 첫번째 노드 삽입 [3/8]

④  $last \leftarrow CL;$

- 첫번째 노드의 주소를 임시 순회 포인터 last에 저장하여 노드 순회의 시작점을 지정한다.

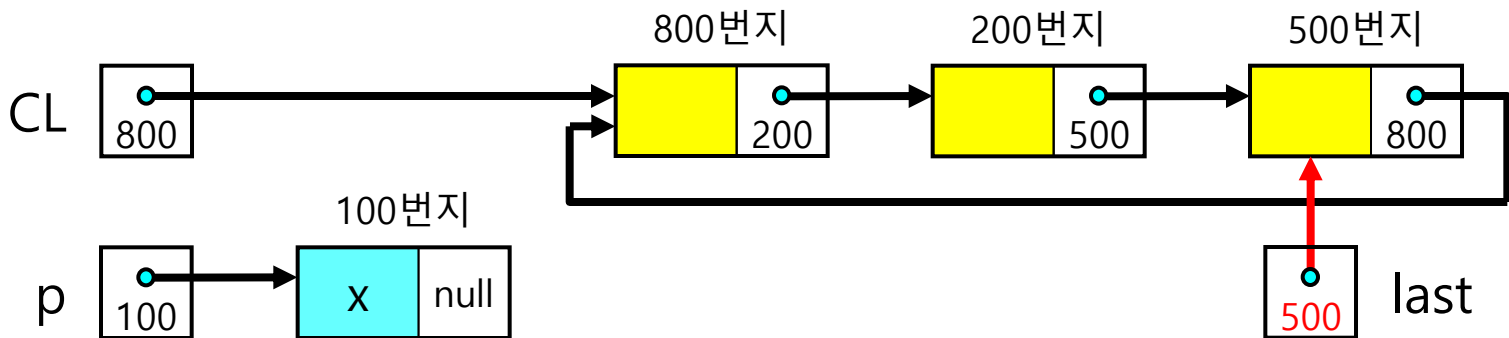




# 원형 연결: 첫번째 노드 삽입 [4/8]

⑤ **while** (last.link  $\neq$  CL) **do** last  $\leftarrow$  last.link;

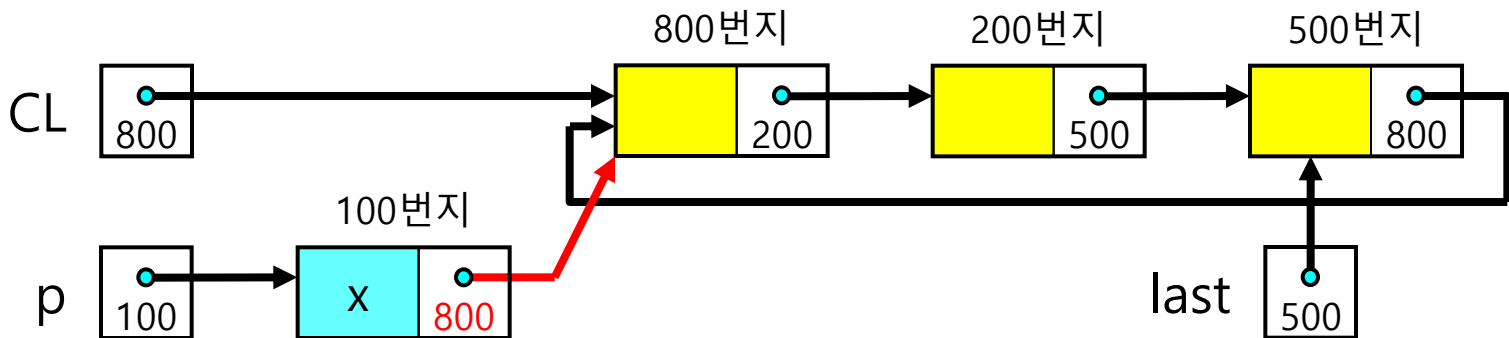
- while문을 수행하여 순회 포인터 last를 링크를 따라 마지막 노드까지 이동



# 원형 연결: 첫번째 노드 삽입 [5/8]

⑥  $p.link \leftarrow last.link;$

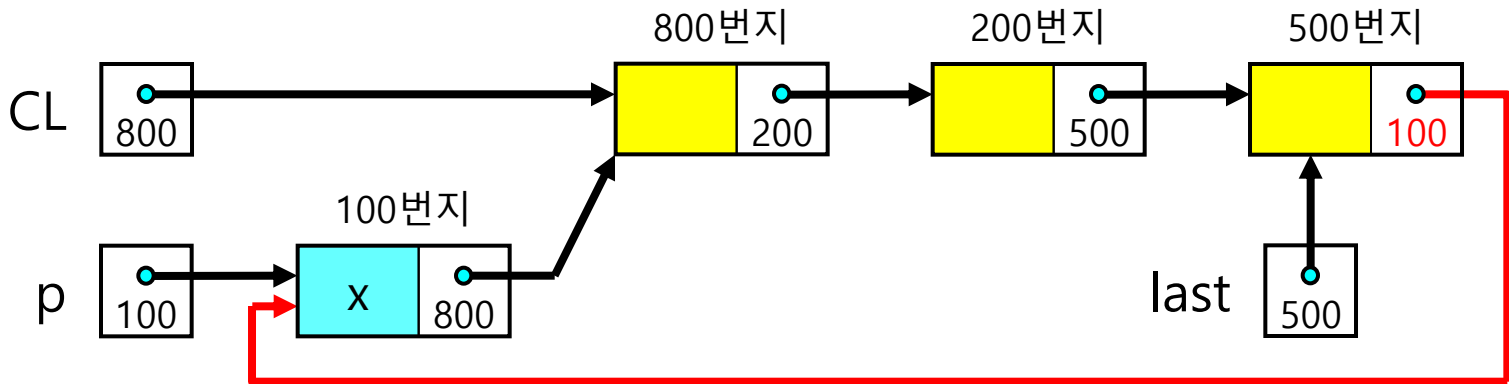
- 리스트의 마지막 노드의 링크 값을 노드 p의 링크에 저장하여, 노드 p가 노드 last의 다음 노드를 가리키게 한다.



# 원형 연결: 첫번째 노드 삽입 [6/8]

⑦  $\text{last.link} \leftarrow p;$

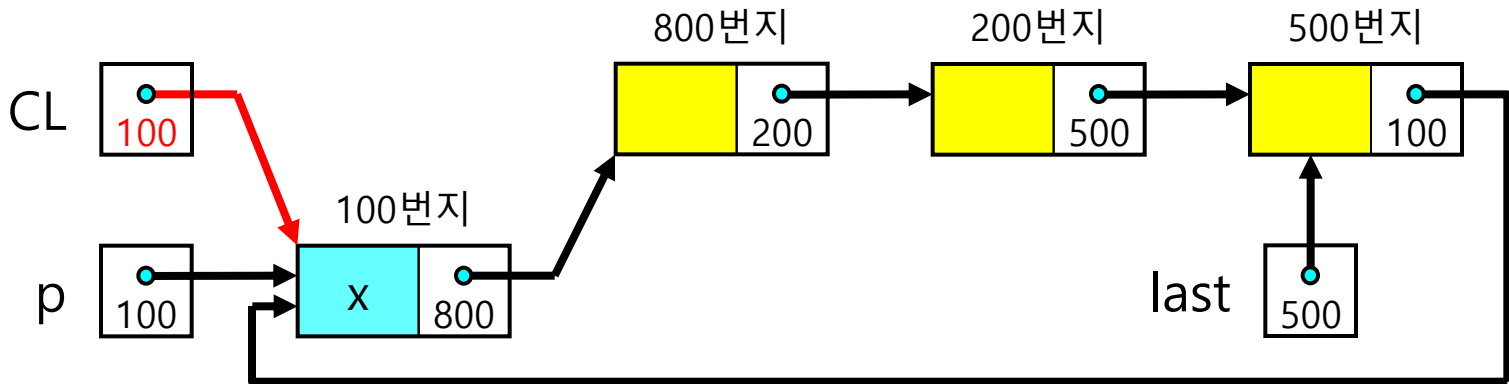
- 포인터 p의 값을 last가 가리키고 있는 마지막 노드의 링크에 저장하여, 리스트의 마지막 노드가 노드 p를 가리키게 한다.



# 원형 연결: 첫번째 노드 삽입 [7/8]

⑧  $CL \leftarrow p;$

- 노드 p의 주소를 리스트 포인터 CL에 저장하여 노드 p가 리스트의 첫번째 노드가 되도록 지정



# 원형 연결: 첫번째 노드 삽입 [8/8]

```
node *insertFirstNode(node *CL, int x)
{
    node *p, *last;

    p = getNode();
    p->data = x;

    if (CL == NULL) {
        CL = p;
        p->link = p;
    }
    else {
        last = CL;
        while (last->link != CL)
            last = last->link;
        p->link = last->link;
        last->link = p;
        CL = p;
    }
    return CL;
}
```

# 원형 연결: 중간 노드 삽입 [1/5]

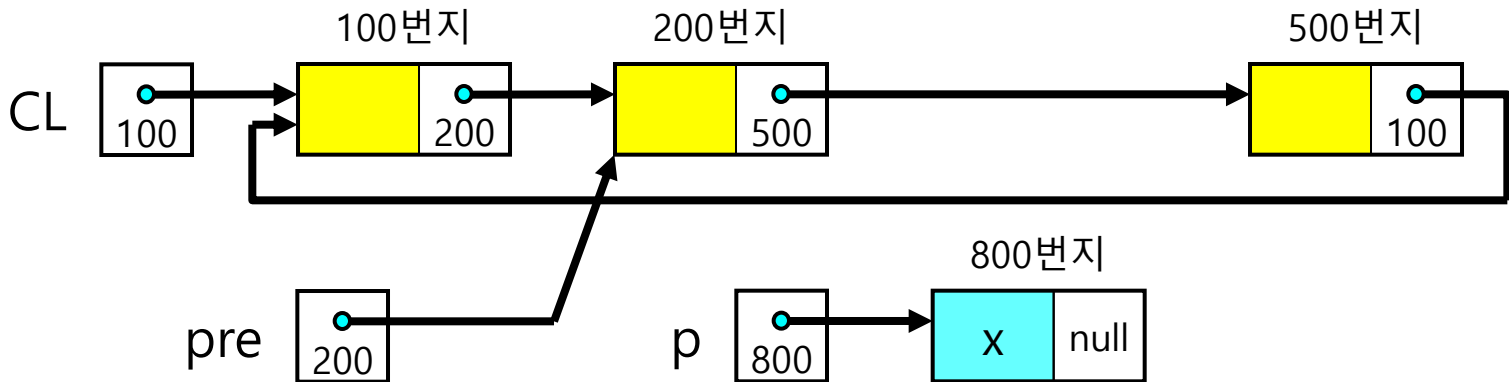
- 원형 연결 리스트 CL에 x 값을 갖는 노드 p를 포인터 pre가 가리키는 노드의 다음 노드로 삽입하는 알고리즘

- ① 원형 리스트 CL이 공백 리스트인 경우는 첫번째 노드 삽입과 동일

```
insertMiddleNode(CL, pre, x)
  p ← getNode();
  p.data ← x;
  if (CL = null) then {      // ①
    CL ← p;
    p.link ← p;
  }
  else {                     // ②
    p.link ← pre.link;      // ③
    pre.link ← p;           // ④
  }
end insertMiddleNode()
```

# 원형 연결: 중간 노드 삽입 [2/5]

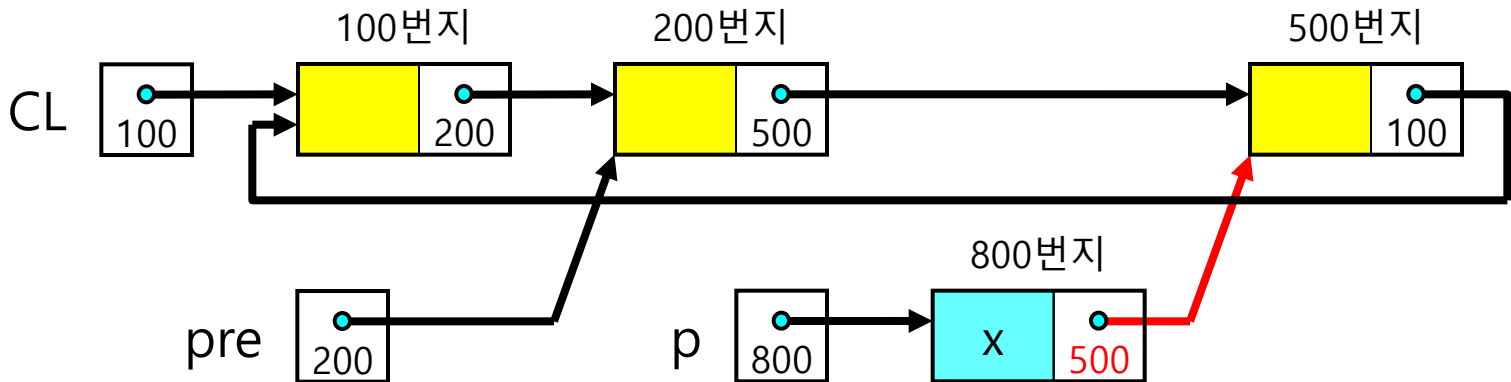
- ② 원형 연결 리스트 CL에서 노드 p를 삽입하려는 위치의 앞 노드를 포인터 pre가 가리키고 있다.



# 원형 연결: 중간 노드 삽입 [3/5]

③  $p.link \leftarrow pre.link;$

- 노드 pre의 링크가 가리키는 주소를 노드 p의 링크에 저장한다.

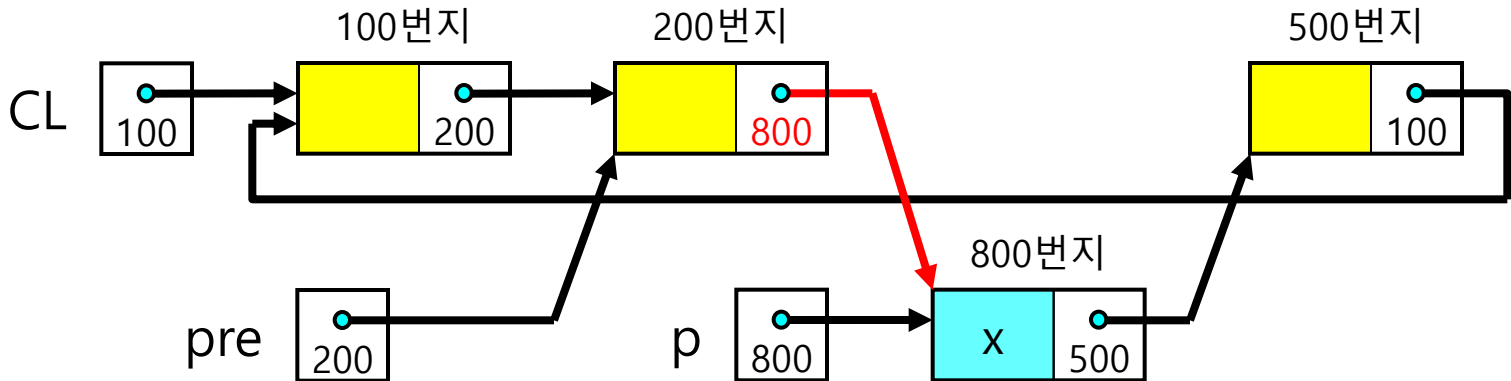




# 원형 연결: 중간 노드 삽입 [4/5]

④  $\text{pre.link} \leftarrow p;$

- 노드 p의 주소를 노드 pre의 링크에 저장하여, 노드 pre가 노드 p를 가리키게 한다.



# 원형 연결: 중간 노드 삽입 [5/5]

```
node *insertMiddleNode(node *CL, node *pre, int x)
{
    node *p;

    p = getNode();
    p->data = x;

    if (CL == NULL) {
        CL = p;
        p->link = p;
    } else {
        p->link = pre->link;
        pre->link = p;
    }

    return CL;
}
```

# 원형 연결 리스트의 노드 삭제 [1/6]

- 원형 연결 리스트 CL에서 포인터 pre가 가리키는 노드의 다음 노드를 삭제하고 삭제한 노드는 자유공간 리스트에 반환하는 연산

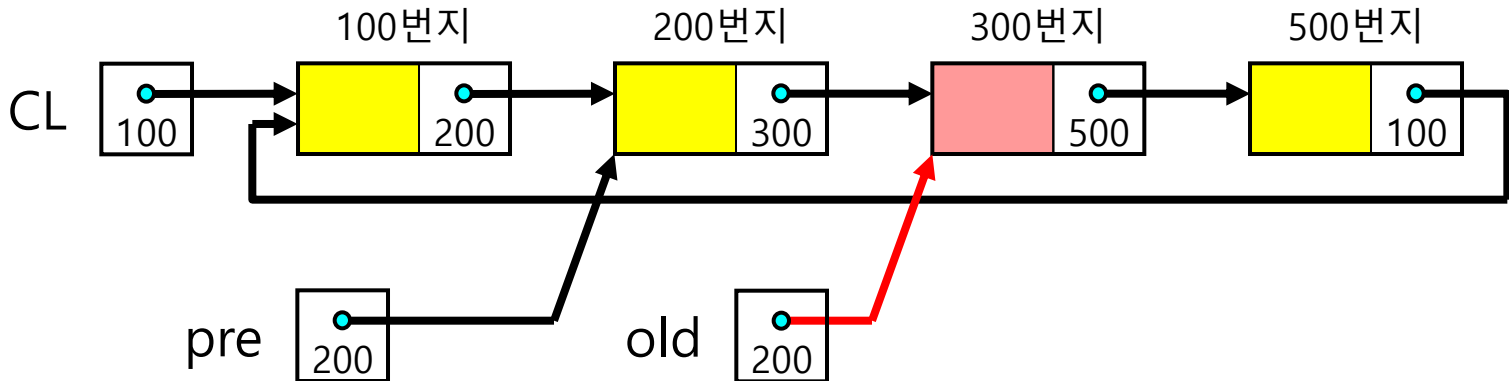
```
deleteNode(CL, pre)
    if (CL = null) then error;

    old ← pre.link;           // ①
    pre.link ← old.link;      // ②
    if (old = CL) then       // ③
        CL ← old.link;       // ④
    returnNode(old);
end deleteNode()
```

# 원형 연결 리스트의 노드 삭제 [2/6]

①  $old \leftarrow pre.link;$

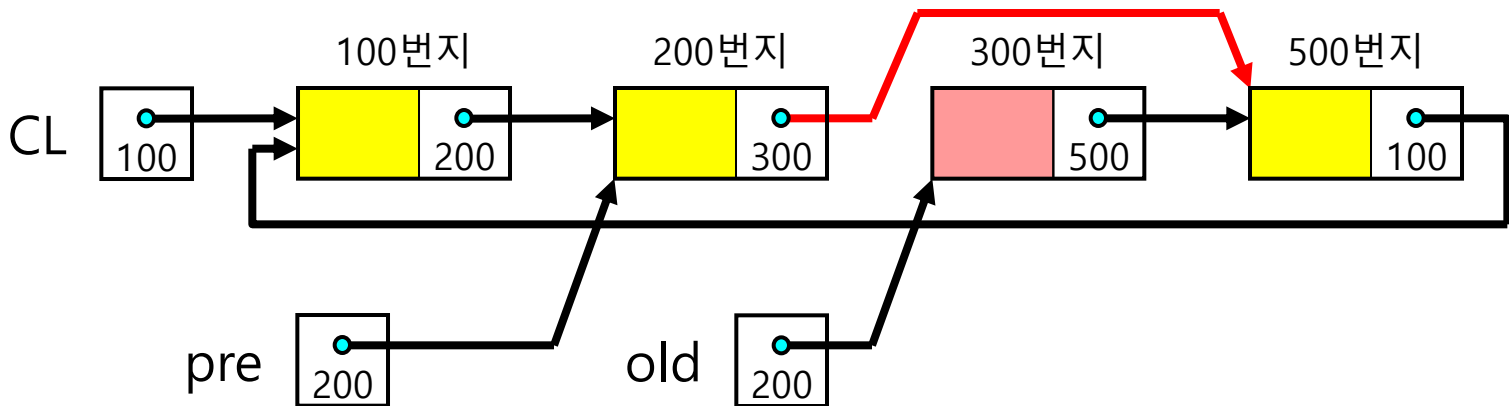
- 노드 pre의 다음 노드를 삭제할 노드 old로 지정



## 원형 연결 리스트의 노드 삭제 [3/6]

```
② pre.link ← old.link;
```

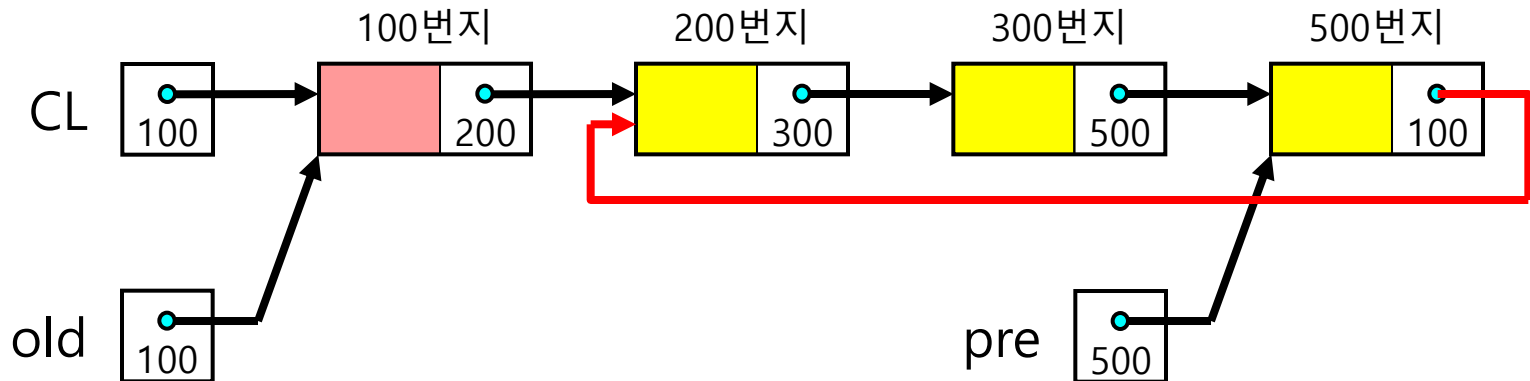
- 노드 old의 이전 노드와 다음 노드를 서로 연결



# 원형 연결 리스트의 노드 삭제 [4/6]

③ **if (old = CL) then**

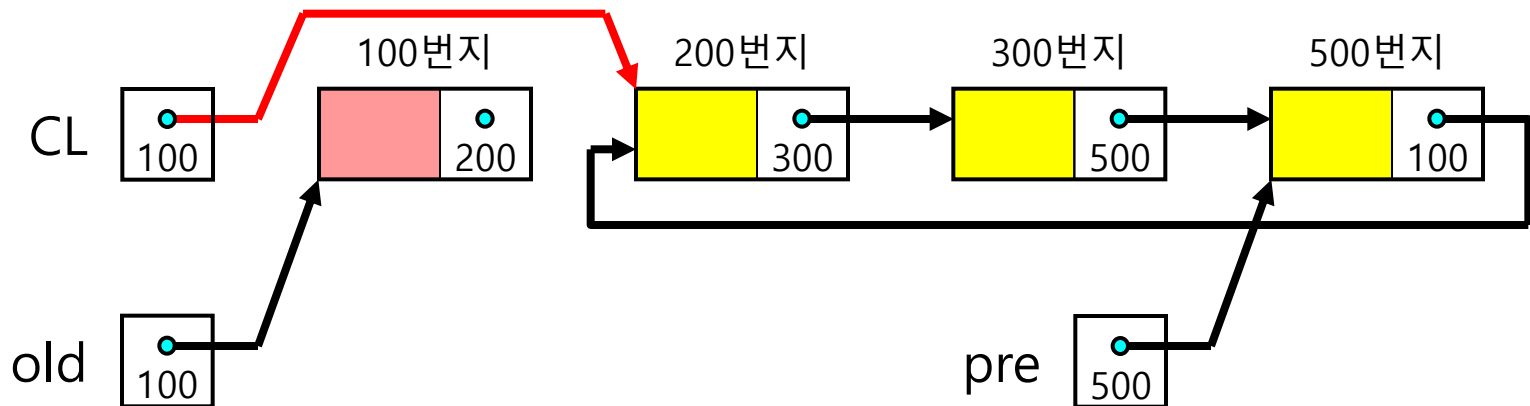
- 삭제할 노드 old가 리스트 포인터 CL인 경우



# 원형 연결 리스트의 노드 삭제 [5/6]

④  $CL \leftarrow old.link;$

- 노드 old의 링크 값을 리스트 포인터 CL에 저장하여 두 번째 노드가 리스트의 첫 번째 노드가 되도록 조정



# 원형 연결 리스트의 노드 삭제 [6/6]

```
node *deleteNode(node *CL, node *pre)
{
    node *old;

    if (CL == NULL) return CL;

    old = pre->link;
    pre->link = old->link;
    if (old == CL)
        CL = old->link;

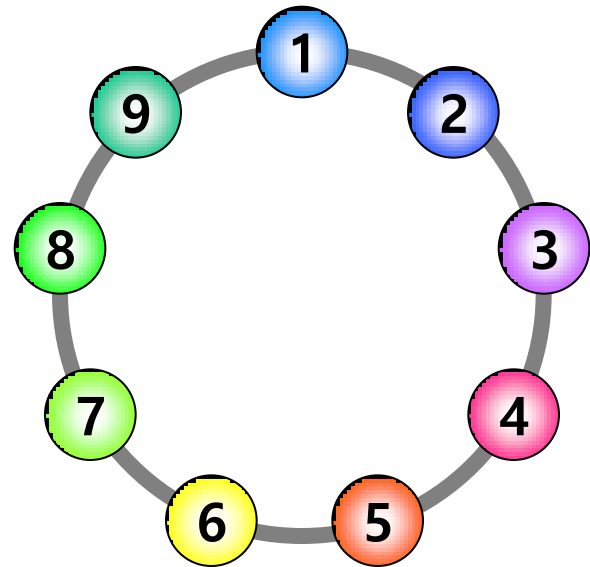
    returnNode(old);

    return CL;
}
```

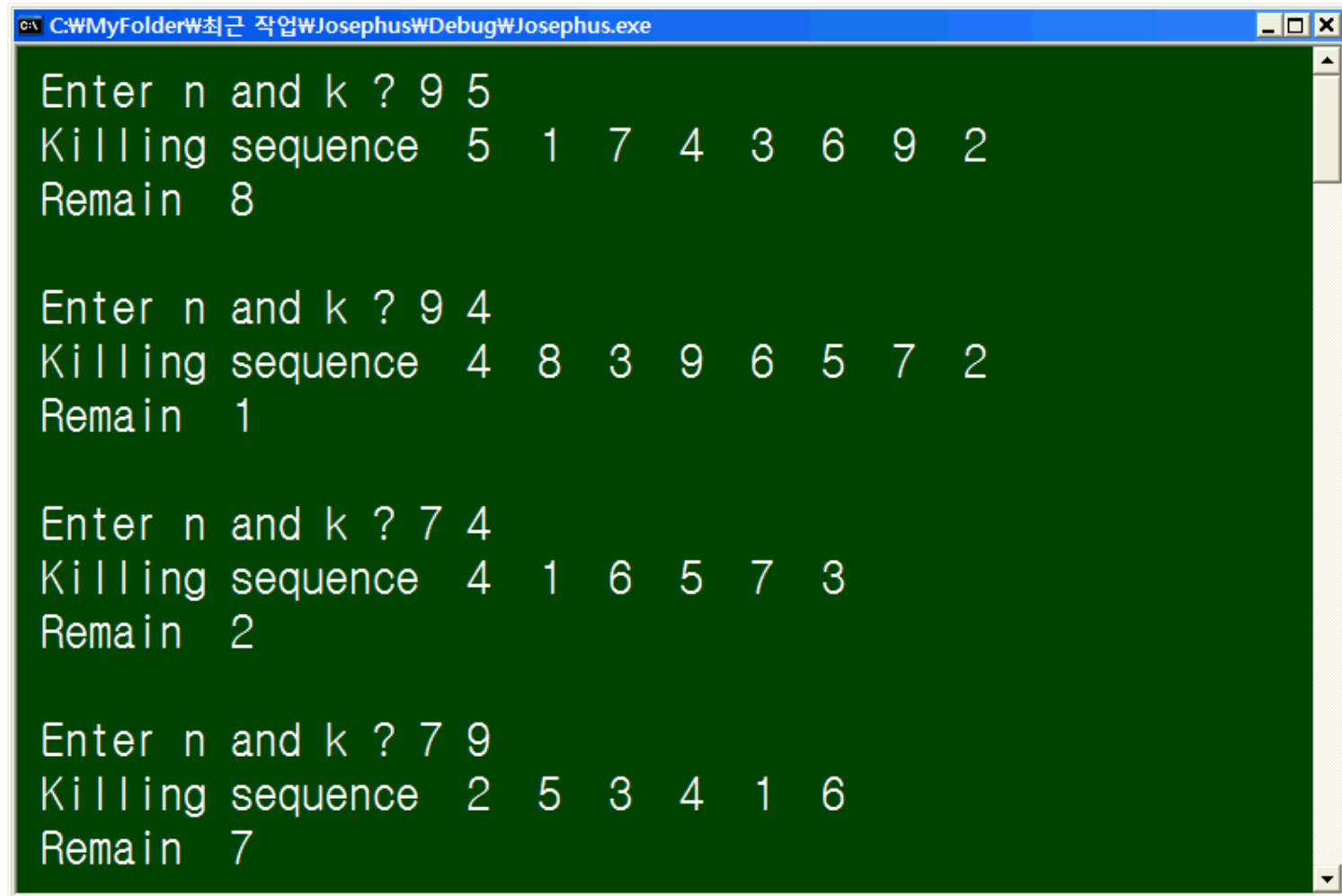


# Josephus problem [1/2]

- $n$ 명의 사람들이 원탁에 둘러 앉아 있다.
- 시작 위치로부터  $k$ 번째 사람은 원탁에서 빠진다. 그리고, 다음 사람을 시작위치로 하여 다시 반복한다.
- 마지막에 남은 한 사람은 누구인가?



# Josephus problem [2/2]



```
C:\MyFolder\최근 작업\Josephus\Debug\Josephus.exe

Enter n and k ? 9 5
Killing sequence 5 1 7 4 3 6 9 2
Remain 8

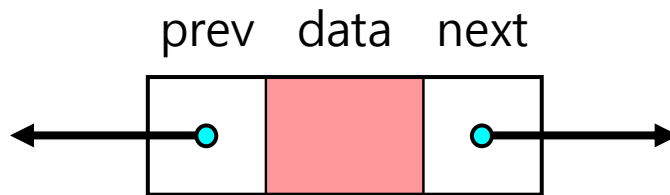
Enter n and k ? 9 4
Killing sequence 4 8 3 9 6 5 7 2
Remain 1

Enter n and k ? 7 4
Killing sequence 4 1 6 5 7 3
Remain 2

Enter n and k ? 7 9
Killing sequence 2 5 3 4 1 6
Remain 7
```

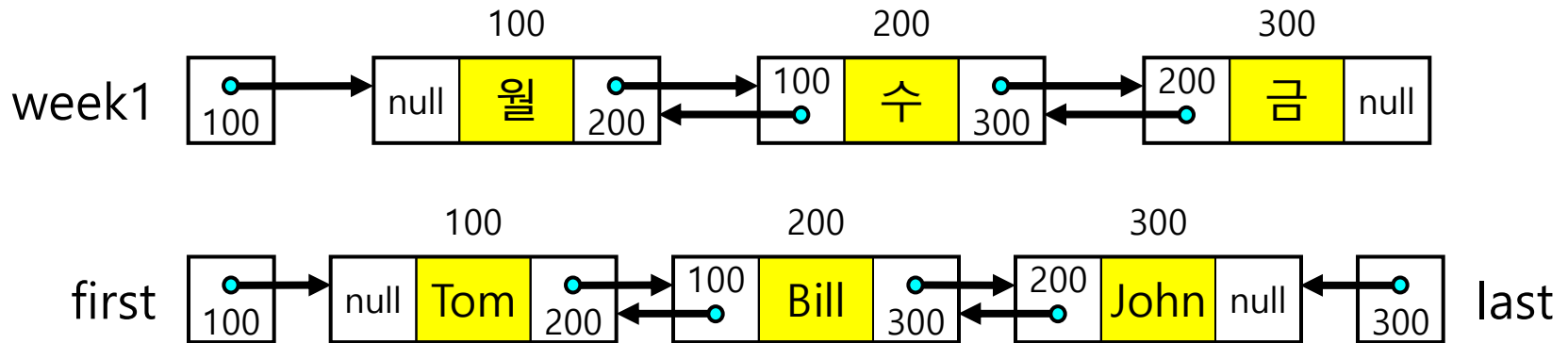
# 이중 연결 리스트 [1/3]

- 이중 연결 리스트(doubly linked list)
  - 양쪽 방향으로 순회할 수 있도록 노드를 연결한 리스트
  - 이중 연결 리스트의 노드 구조
    - ◆ 두 개의 링크 필드와 한 개의 데이터 필드로 구성
    - ◆ prev (앞쪽 링크 필드), next (뒤쪽 링크 필드)

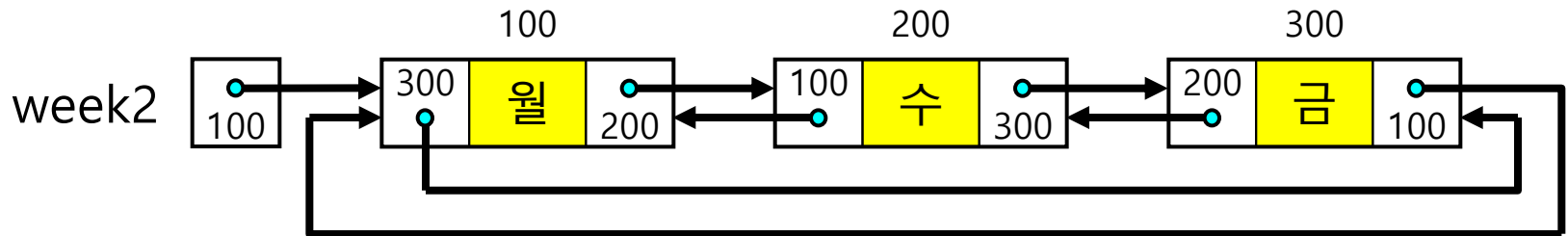


# 이중 연결 리스트 [2/3]

## ● 이중 연결 리스트



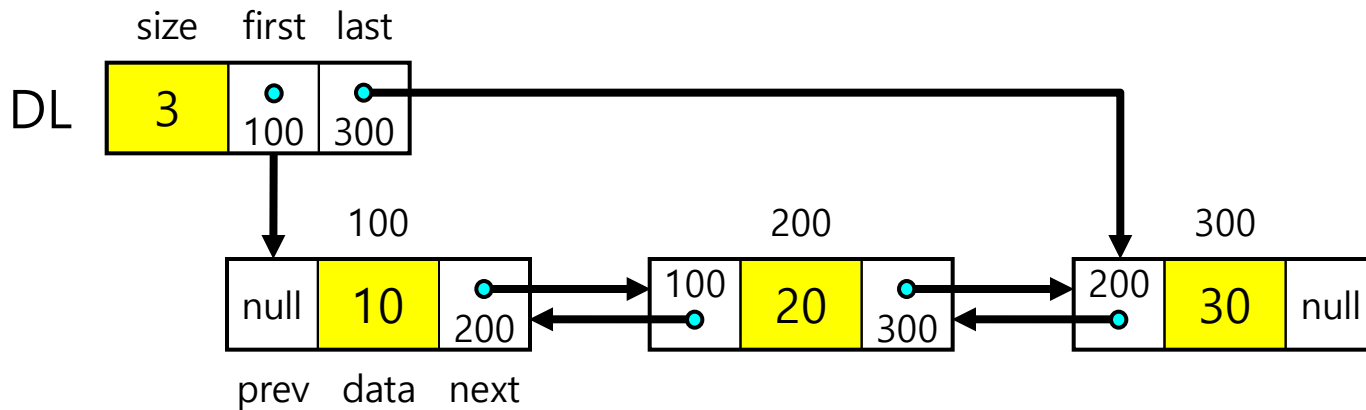
## ● 원형 이중 연결 리스트



# 이중 연결 리스트 [3/3]

```
typedef struct Dnode {  
    int data;  
    struct Dnode *prev;  
    struct Dnode *next;  
} dnode;
```

```
typedef struct DList {  
    int    size;  
    dnode  *first;  
    dnode  *last;  
} dlist;  
  
dlist DL;
```



# 이중 연결 리스트 ADT

이 름 : DLIST

데이터 : 정수 데이터

연 산 :

getNode( );	// 새로운 노드 할당
returnNode(p);	// p 노드를 반환
createHeader(DL);	// 이중 연결 리스트 DL을 초기화 한다.
insertFirstNode(DL, x);	// DL의 맨 앞에 x값을 갖는 노드 추가
insertLastNode(DL, x);	// DL의 맨 뒤에 x값을 갖는 노드 추가
deleteFirstNode(DL);	// 이중 연결 리스트 DL의 맨 앞의 노드 삭제
deleteLastNode(DL);	// 이중 연결 리스트 DL의 맨 뒤의 노드 삭제
printList(DL);	// 이중 연결 리스트 DL의 각 노드의 주소, // 데이터와 링크 필드 주소를 차례대로 출력한다

# 리스트 초기화

```
void createHeader(dlist *DL)
{
    DL->first = NULL;
    DL->last  = NULL;
    DL->size  = 0;
}
```

# 이중 연결: 첫번째 노드 삽입 [1/7]

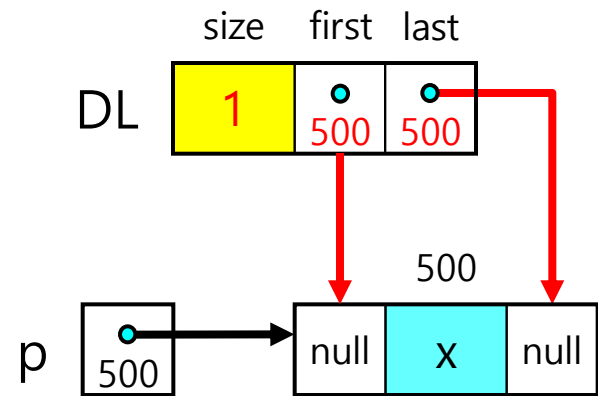
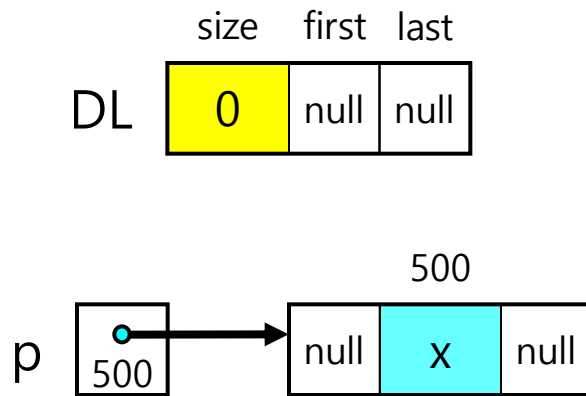
- 이중 연결 리스트 DL의 맨 앞에 x값을 갖는 노드 삽입

```
InsertFirstNode(DL, x)  
  p ← getNode();  
  p.data ← x;  
  if (DL.size = 0) then           // ①  
    DL.first ← p;  
    DL.last ← p;  
  else  
    p.next ← DL.first;           // ②  
    DL.first.prev ← p;           // ③  
    DL.first ← p;               // ④  
  end if  
  DL.size++;  
end InsertFirstNode()
```



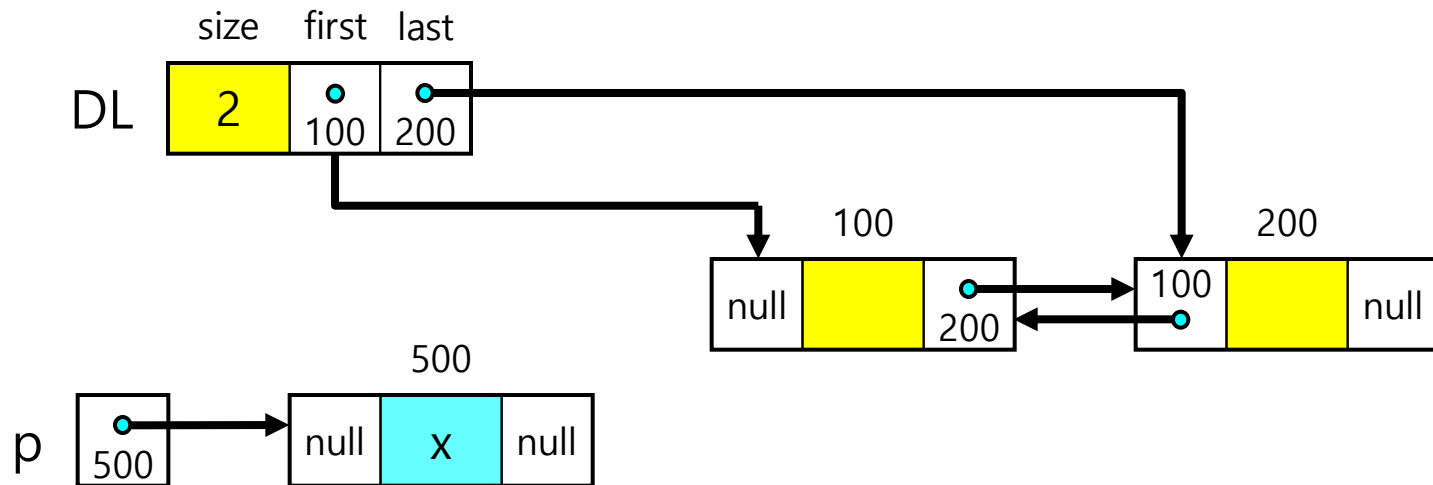
# 이중 연결: 첫번째 노드 삽입 [2/7]

① DL에 노드가 없으면 새로운 노드 p를 first, last에 연결



# 이중 연결: 첫번째 노드 삽입 [3/7]

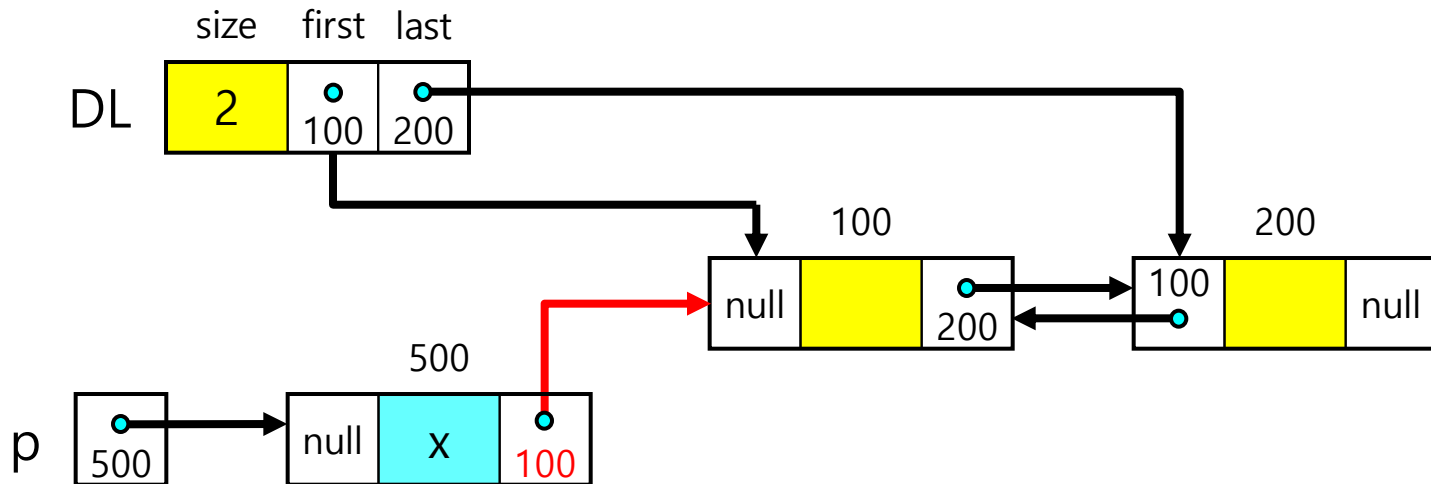
- 이중 연결 리스트 DL의 맨 앞에 x값을 갖는 노드 삽입



# 이중 연결: 첫번째 노드 삽입 [4/7]

②  $p.next \leftarrow DL.first;$

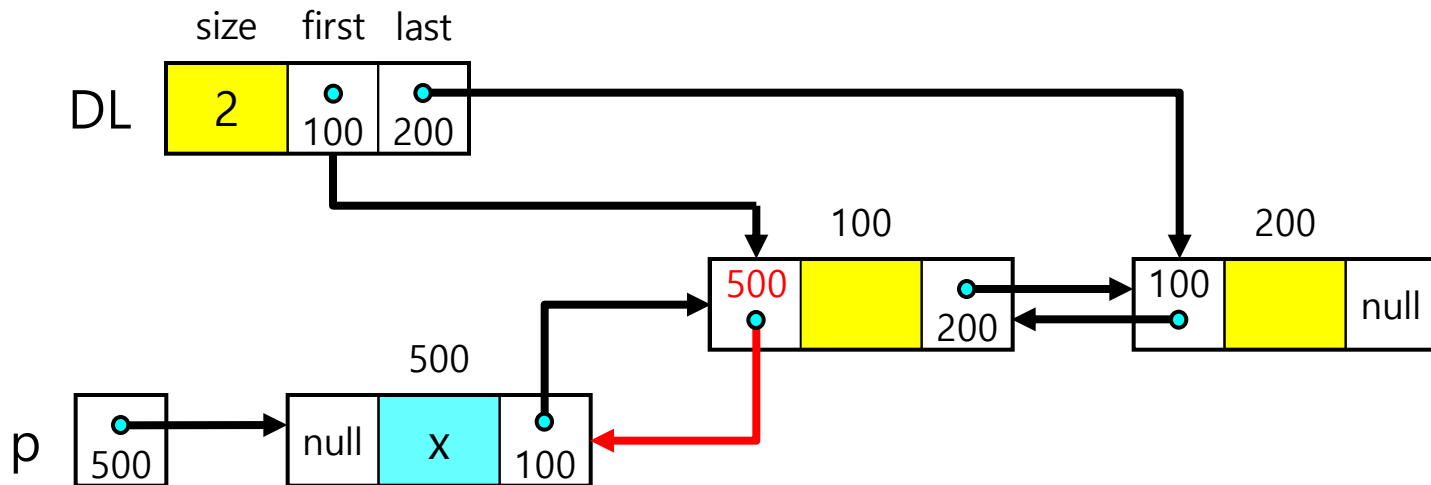
- 새로운 노드 p의 next를 리스트의 first 노드에 연결



# 이중 연결: 첫번째 노드 삽입 [5/7]

③  $DL.first.prev \leftarrow p;$

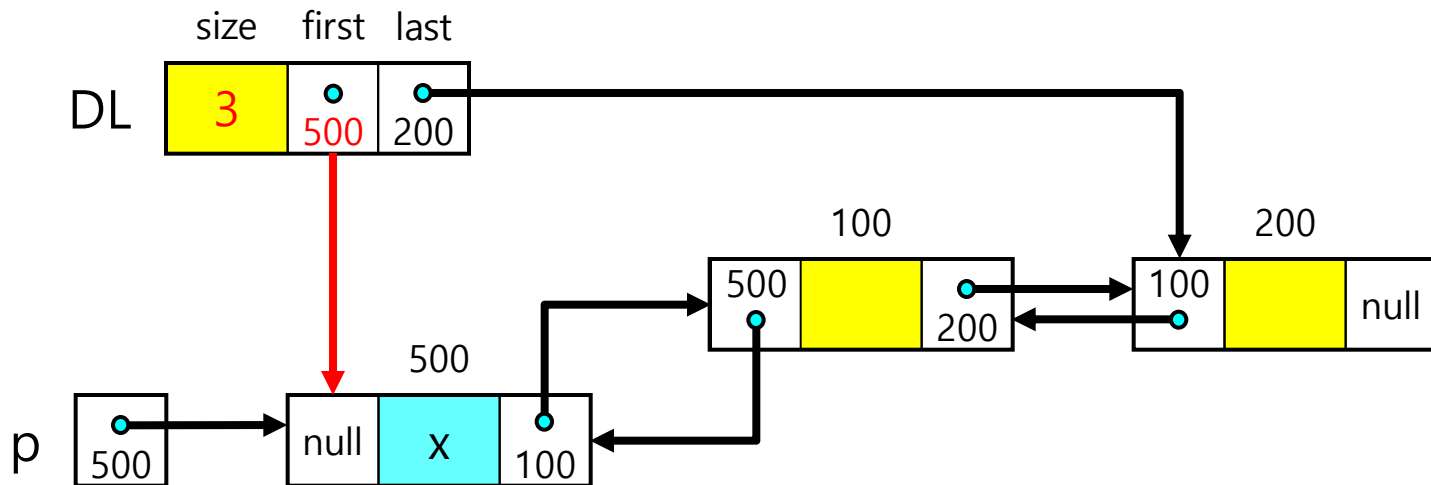
- 리스트의 first 노드의 prev 링크에 노드 p를 연결



# 이중 연결: 첫번째 노드 삽입 [6/7]

④  $DL.first \leftarrow p; (DL.size++;$  )

- 노드 p를 리스트의 새로운 first로 지정



# 이중 연결: 첫번째 노드 삽입 [7/7]

```
void insertFirstNode(dlist *DL, int x)
{
    dnode *p;

    p = getNode();
    p->data = x;

    if (DL->size == 0) {
        DL->first = p;
        DL->last  = p;
    } else {
        p->next = DL->first;
        DL->first->prev = p;
        DL->first = p;
    }
    DL->size++;
}
```

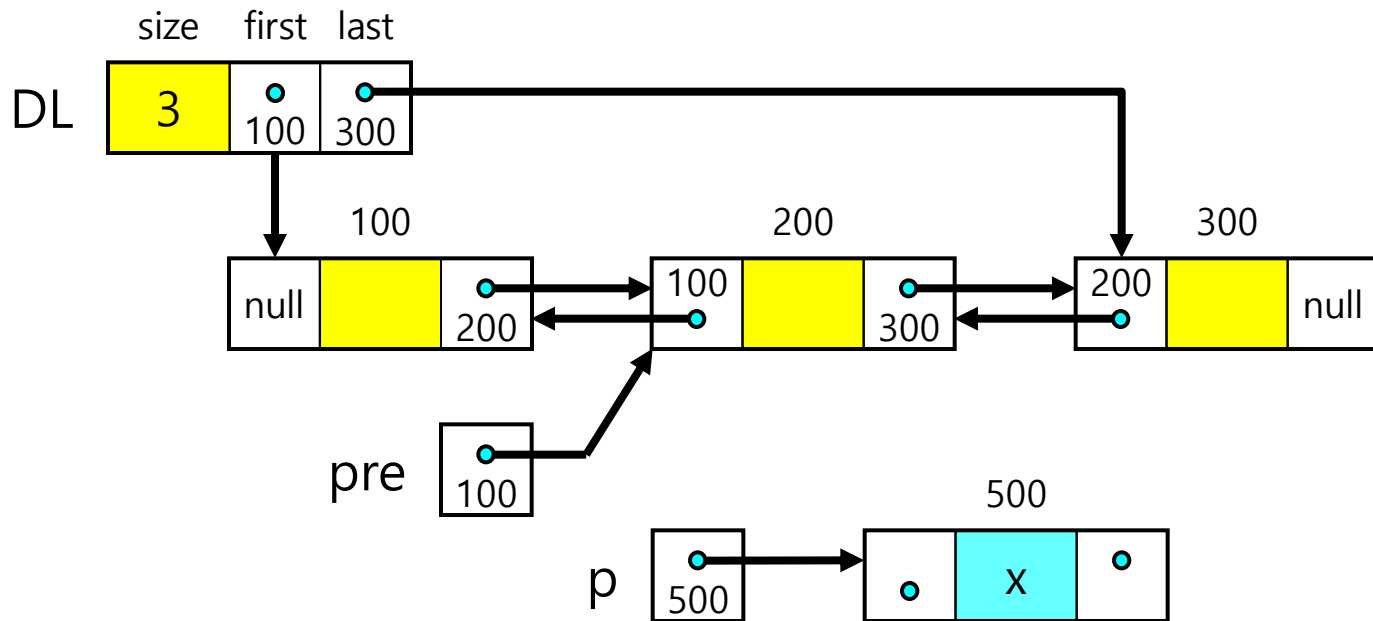
# 이중 연결: 중간 노드 삽입 [1/6]

- 이중 연결 리스트 DL에서 포인터 pre가 가리키는 노드의 다음 노드로 노드 p를 삽입하는 과정

```
InsertNode(DL, pre, x)
    p ← getNode();
    p.data ← x;
    p.next ← pre.next ;      // ①
    pre.next ← p;            // ②
    p.prev ← pre;            // ③
    p.next.prev ← p;         // ④
    DL.size++;
end InsertNode()
```

# 이중 연결: 중간 노드 삽입 [2/6]

- 이중 연결 리스트 DL에서 포인터 pre가 가리키는 노드의 다음 노드로 노드 p를 삽입하는 과정

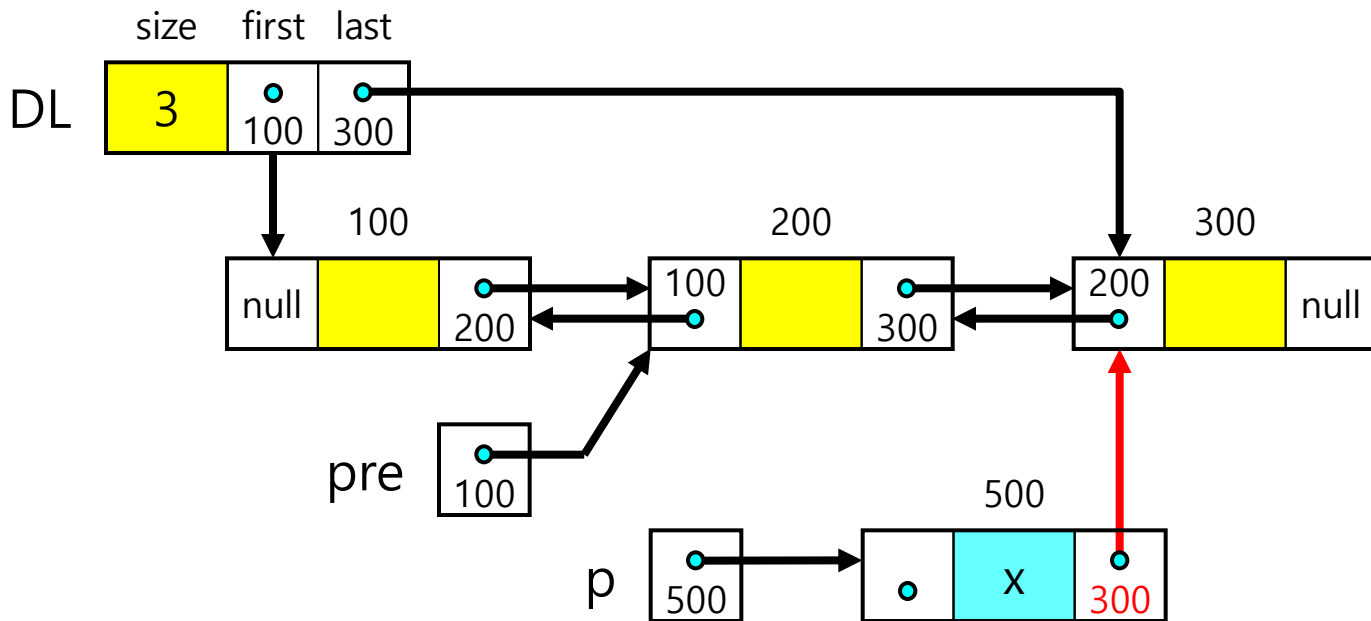




# 이중 연결: 중간 노드 삽입 [3/6]

①  $p.next \leftarrow pre.next;$

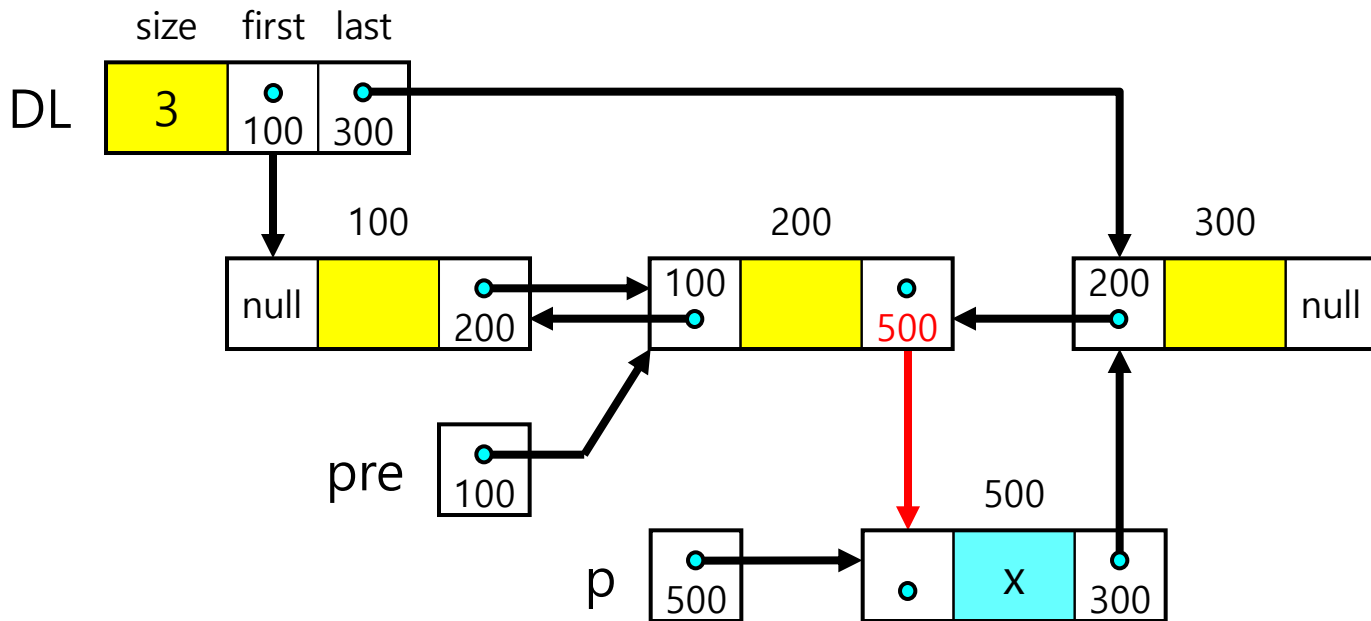
- 노드 pre의 next를 노드 p의 next에 저장하여,  
노드 pre의 다음 노드를 삽입할 노드 p의 다음 노드로 연결



# 이중 연결: 중간 노드 삽입 [4/6]

②  $pre.next \leftarrow p$ ;

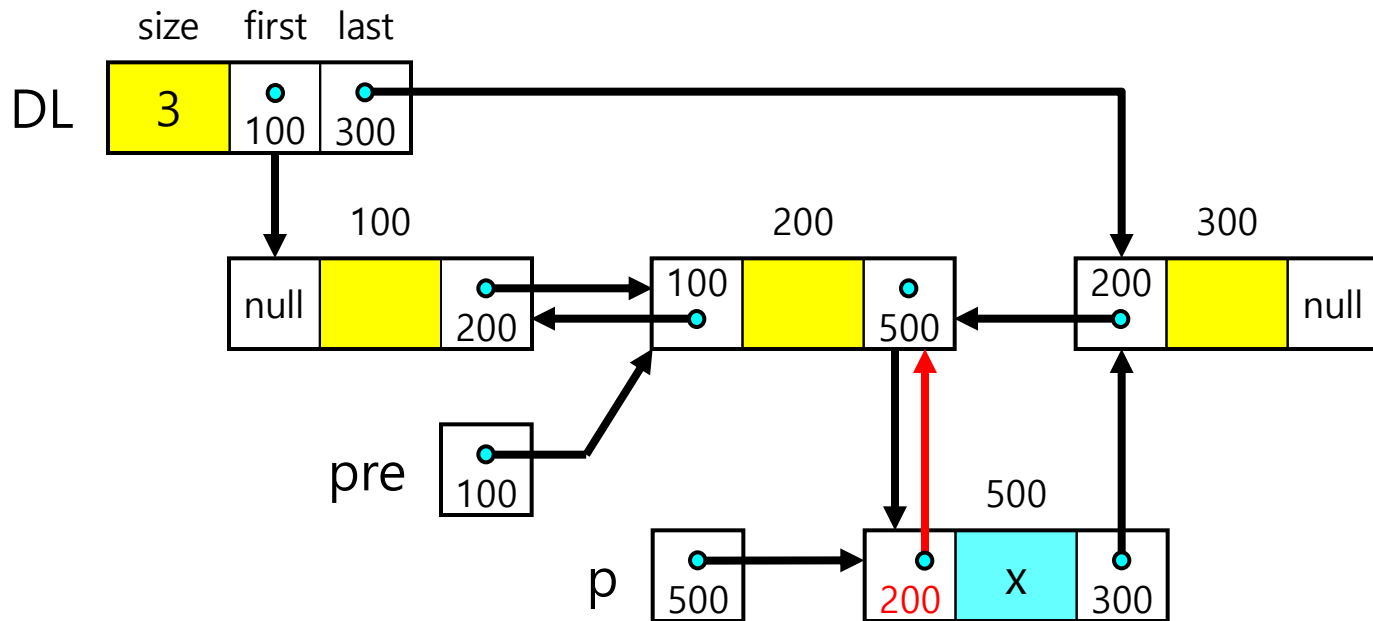
- 새 노드 p의 주소를 노드 pre의 next에 저장하여,  
노드 p를 노드 pre의 다음 노드로 연결



# 이중 연결: 중간 노드 삽입 [5/6]

③  $p.\text{prev} \leftarrow \text{pre};$

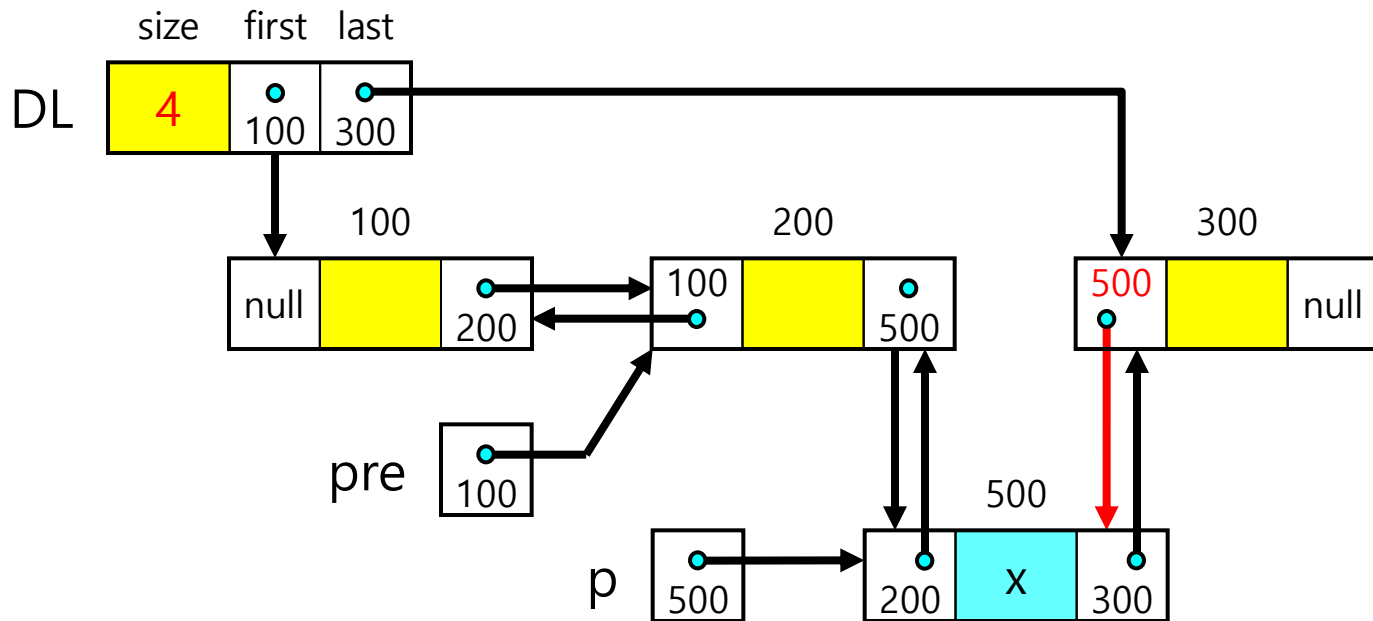
- 포인터 pre의 값을 삽입할 노드 p의 prev에 저장하여, 노드 pre를 노드 p의 이전 노드로 연결



# 이중 연결: 중간 노드 삽입 [6/6]

④  $p.next.prev \leftarrow p; (DL.size++;$  )

- 포인터 p의 값을 노드 p의 다음 노드(p.next)의 prev에 저장하여, 노드 p의 다음 노드의 이전 노드로 p를 연결



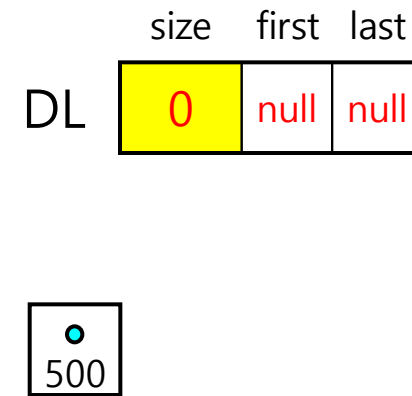
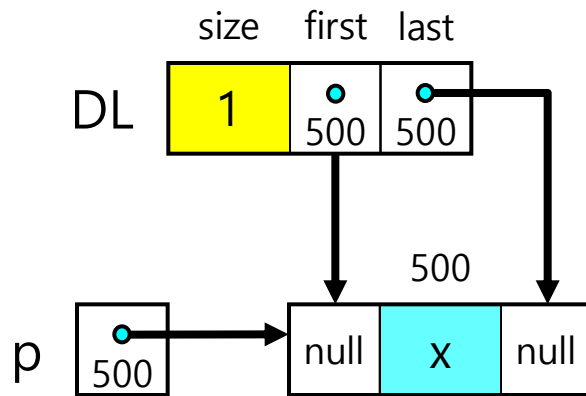
# 이중 연결: 첫번째 노드 삭제 [1/6]

- 이중 연결 리스트 DL의 맨 앞의 노드 삭제

```
deleteFirstNode(DL)  
  if (DL.size = 0) return;  
  if (DL.size = 1) then                                // ①  
    returnNode(DL.first);  
    DL.first ← null;  
    DL.last ← null;  
  
  else  
    DL.first ← DL.first.next;                            // ②  
    returnNode(DL.first.prev);                            // ③  
    DL.first.prev ← null;  
  
  end if  
  DL.size--;  
end deleteNode()
```

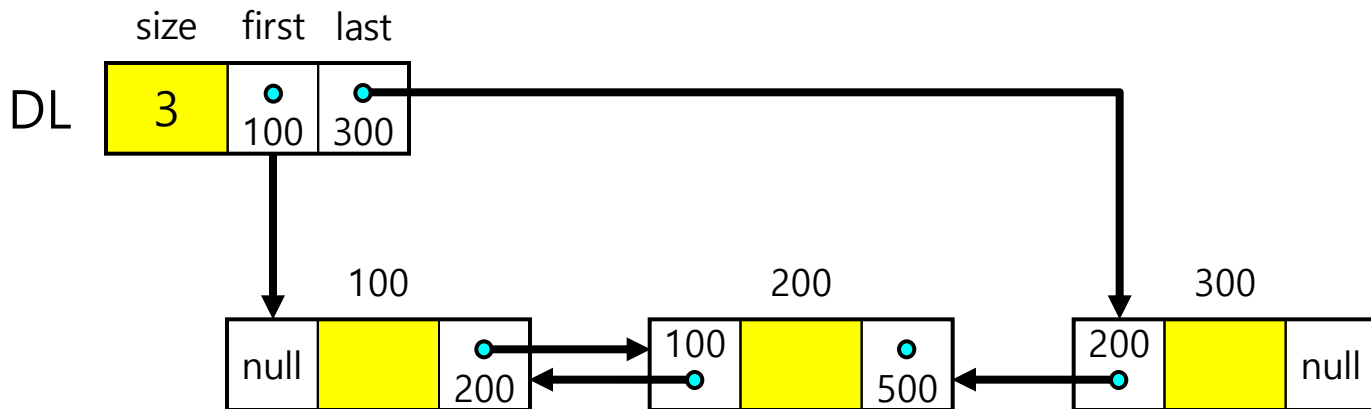
# 이중 연결: 첫번째 노드 삭제 [2/6]

① DL에 노드가 하나뿐이면 노드를 삭제하고, DL을 초기화



# 이중 연결: 첫번째 노드 삭제 [3/6]

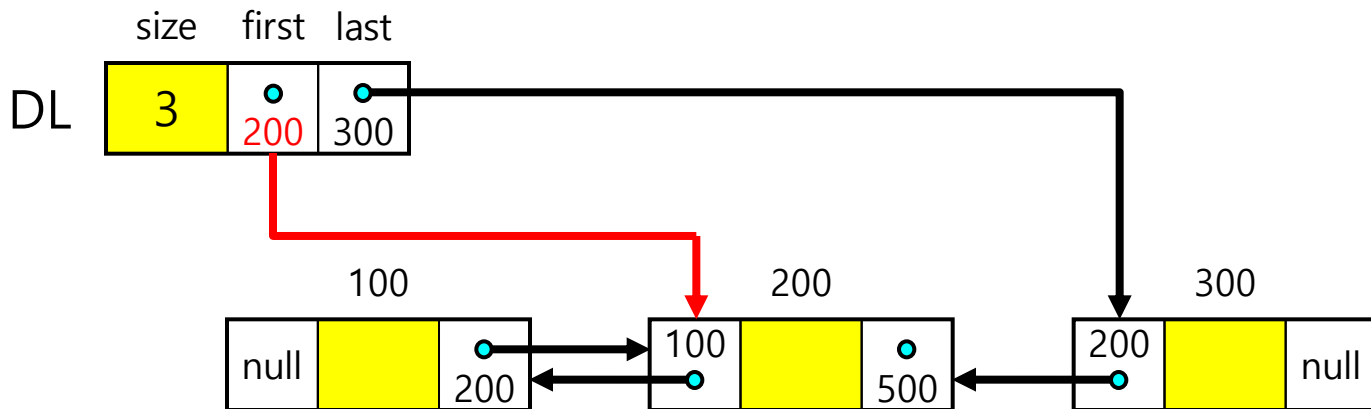
- 이중 연결 리스트 DL에 2개 이상의 노드가 있는 경우, 맨 앞의 노드 삭제



# 이중 연결: 첫번째 노드 삭제 [4/6]

② DL.first  $\leftarrow$  DL.first.next;

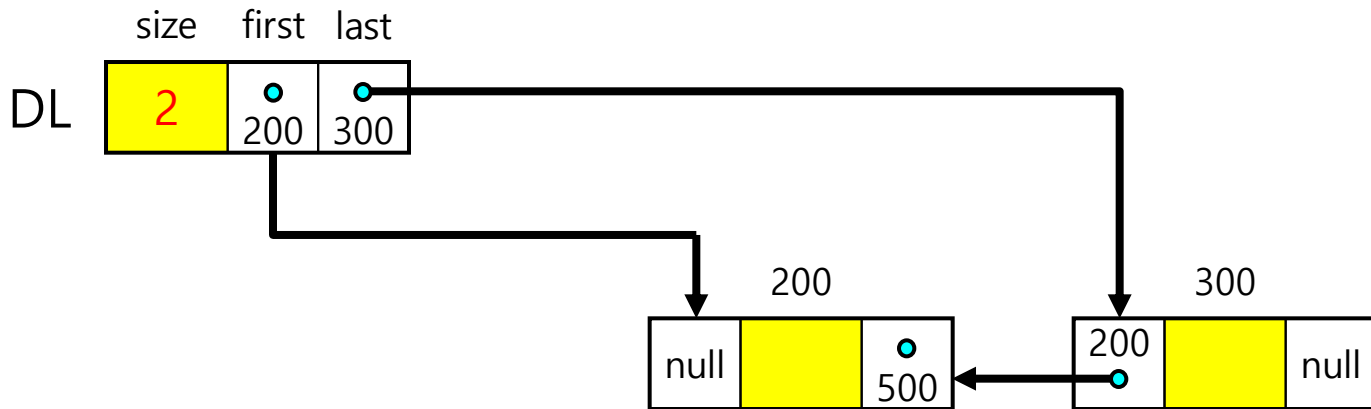
- DL의 first를 first의 다음 노드로 지정





# 이중 연결: 첫번째 노드 삭제 [5/6]

- ③ `returnNode(DL.first.prev); DL.first.prev ← null;`
- DL의 first의 prev 노드 (과거의 first 노드)를 삭제한다.



## 이중 연결: 첫번째 노드 삭제 [6/6]

```
void deleteFirstNode(dlist *DL)
{
    if (DL->size == 0)
        return;

    if (DL->size == 1) {
        returnNode(DL->first);
        DL->first = NULL;
        DL->last  = NULL;
    } else {
        DL->first = DL->first->next;
        returnNode(DL->first->prev);
        DL->first->prev = NULL;
    }
    DL->size--;
}
```

# 이중 연결: 중간 노드 삭제 [1/5]

- 이중 연결 리스트 DL에서 포인터 old가 가리키는 노드를 삭제하는 과정

**deleteNode(DL, old)**

old.prev.next  $\leftarrow$  old.next;      // ①

old.next.prev  $\leftarrow$  old.prev;      // ②

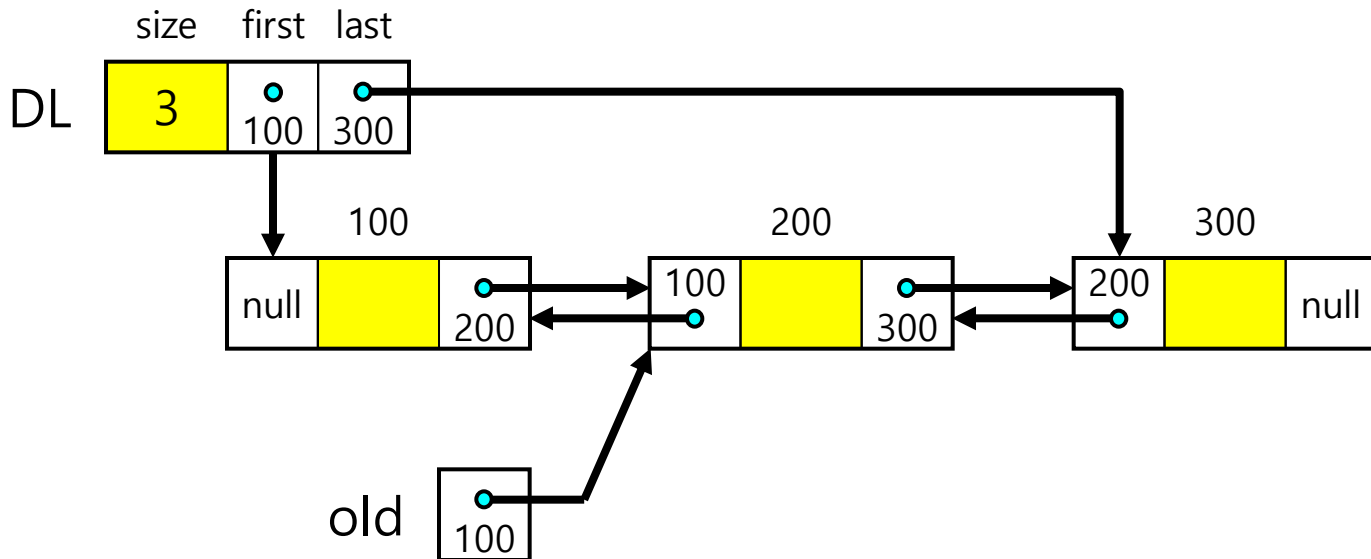
returnNode(old);      // ③

DL.size--;

**end deleteNode()**

# 이중 연결: 중간 노드 삭제 [2/5]

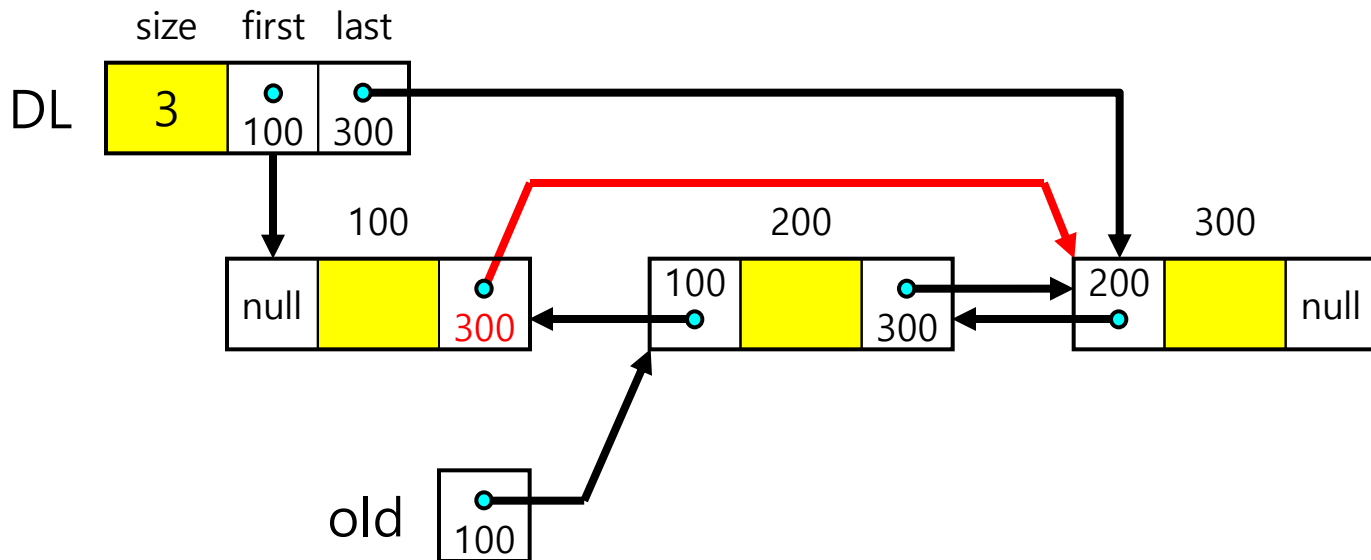
- 이중 연결 리스트 DL에서 포인터 old가 가리키는 노드를 삭제하는 과정



# 이중 연결: 중간 노드 삭제 [3/5]

① `old.prev.next ← old.next;`

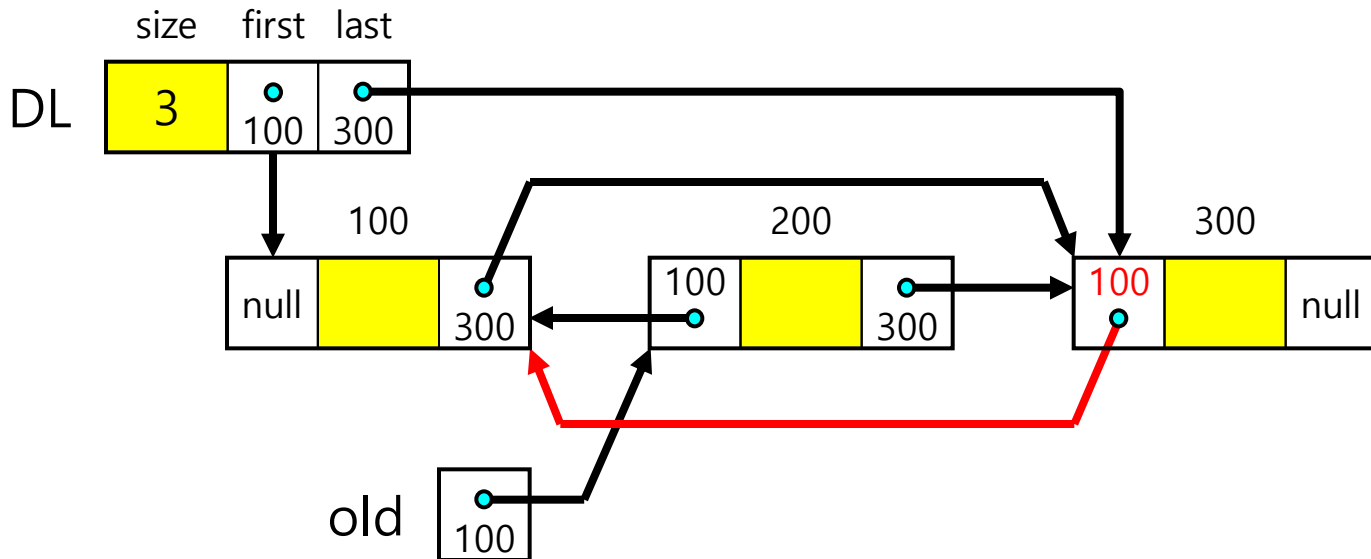
- 삭제할 노드 `old`의 다음 노드의 주소를 노드 `old`의 이전 노드의 `next`에 저장하여, 노드 `old`의 다음 노드를 노드 `old`의 이전 노드의 다음 노드로 연결



# 이중 연결: 중간 노드 삭제 [4/5]

②  $\text{old.next.prev} \leftarrow \text{old.prev};$

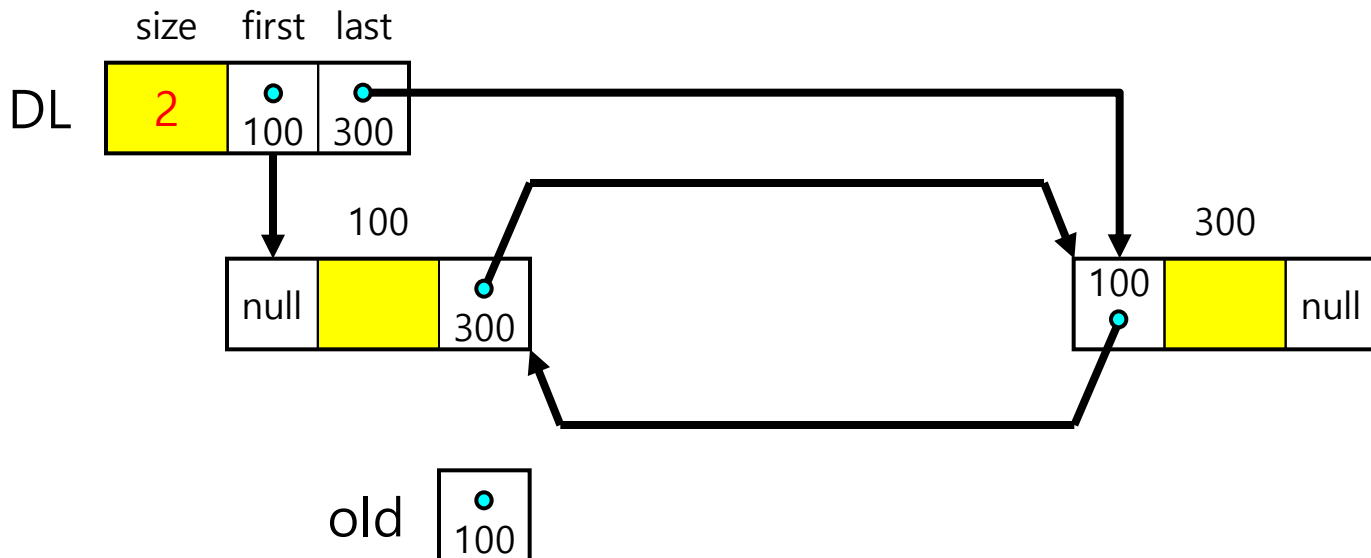
- 삭제할 노드 old의 이전 노드의 주소를 노드 old의 다음 노드의 prev에 저장하여, 노드 old의 이전 노드를 노드 old의 다음 노드의 이전 노드로 연결



# 이중 연결: 중간 노드 삭제 [5/5]

③ returnNode(old); ( DL.size-- )

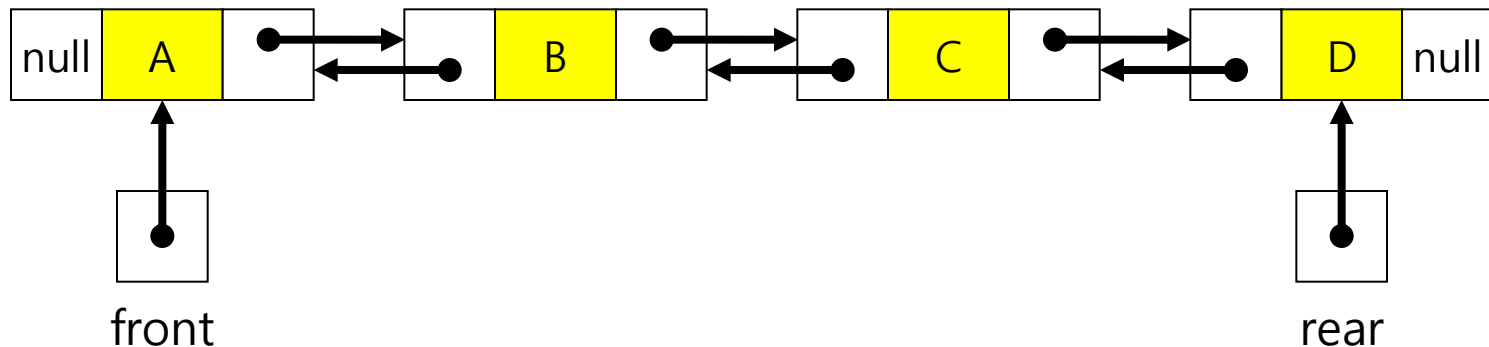
- 삭제된 노드 old는 자유공간리스트에 반환



# 연결리스트를 이용한 덱의 구현

## ● 이중 연결 리스트로 구현

- 양쪽 끝에서 삽입/삭제 연산을 수행하면서 크기 변화와 저장된 원소의 순서 변화가 많으므로 순차 자료구조는 비효율적
- 양방향으로 연산이 가능한 이중 연결 리스트를 사용





# 요약

- 원형 연결 리스트(circular linked list)
  - 단순 연결 리스트의 마지막 노드가 첫번째 노드를 가리켜 리스트의 구조를 원형으로 만든 연결 리스트
  - 링크를 따라 계속 순회하면 이전 노드에 접근 가능
- 이중 연결 리스트(doubly linked list)
  - 양쪽 방향으로 순회할 수 있도록 노드를 연결한 리스트
  - 두 개의 링크 필드와 한 개의 데이터 필드로 구성