연결 자료구조 표현 방식

IT CookBook, 자바로 배우는 쉬운 자료구조

❑ 이 장에서 다룰 내용

❖ 순차 자료구조의 문제점

- 삽입연산이나 삭제연산 후에 연속적인 물리 주소를 유지하기 위해서 원소들을 이동시키는 추가적인 작업과 시간 소요
  - ➢ 원소들의 이동 작업으로 인한 오버헤드는 원소의 개수가 많고 삽입·삭제 연산이 많이 발생하는 경우에 성능상의 문제 발생
- 순차 자료구조는 배열을 이용하여 구현하기 때문에 배열이 갖고 있는 메모리 사용의 비효율성 문제를 그대로 가짐
- 순차 자료구조에서의 연산 시간에 대한 문제와 저장 공간에 대한 문제를 개선한 자료 표현 방법 필요

❖ 연결 자료구조(Linked Data Structure)

▪ 자료의 논리적인 순서와 물리적인 순서가 일치하지 않는 자료구조

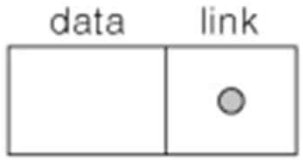
- 각 원소에 저장되어 있는 다음 원소의 주소에 의해 순서가 연결되는 방식
  - ➢ **물리적인 순서를 맞추기 위한 오버헤드가 발생하지 않는다.**
- 여러 개의 작은 공간을 연결하여 하나의 전체 자료구조를 표현
  - ➢ **크기 변경이 유연하고 더 효율적으로 메모리 사용**

▪ 연결 리스트

- 리스트를 연결 자료구조로 표현한 구조
- 연결하는 방식에 따라 단순 연결 리스트와 원형 연결 리스트, 이중 연결 리스트, 이중 원형 연결 리스트

❖ 노드

- 연결 자료구조에서 하나의 원소를 표현하기 위한 단위 구조
- <원소, 주소>의 구조

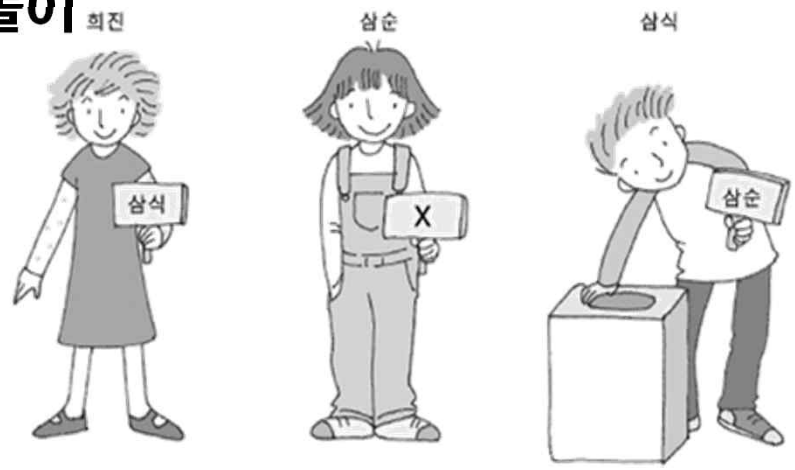


- 데이터 필드(data field)
  - 원소의 값을 저장
  - 저장할 원소의 형태에 따라서 하나 이상의 필드로 구성

- 링크 필드(link field)
  - 다음 노드의 주소를 저장
  - 포인터 변수를 사용하여 주소값을 저장

- **노드 연결 방법에 대한 이해 – 기차놀이**

① 이름표 뽑기



② 자기가 뽑은 이름의 사람을 찾아서 연결하기 : 희진→삼식 →삼순

- **X 표를 뽑은 사람은 마지막 기차**

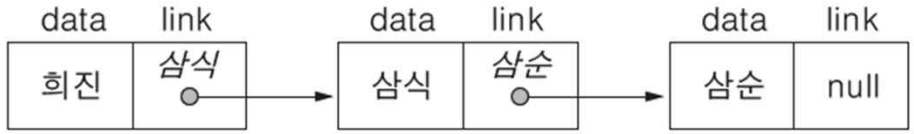
- 기차는 이름표를 들고 있는 방향으로 움직인다.

▪ 노드 연결 방법에 대한 이해 – 기차놀이

- 기차놀이와 연결 리스트

  ➢ 기차놀이 하는 아이들 : 연결리스트의 노드

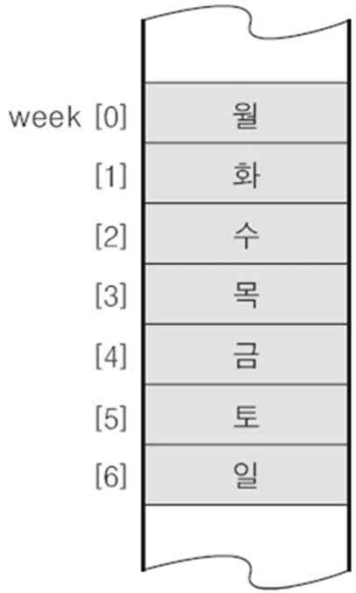
  ➢ 이름표 : 노드의 링크 필드



▪ 선형 리스트와 연결 리스트의 비교

- 리스트 week=(월, 화, 수, 목, 금, 토, 일)
- week에 대한 선형 리스트

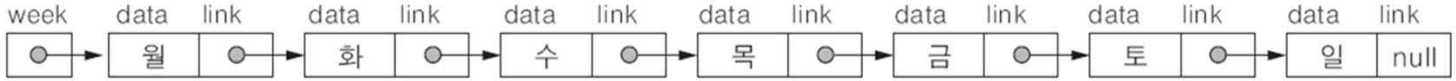
(a) 논리구조



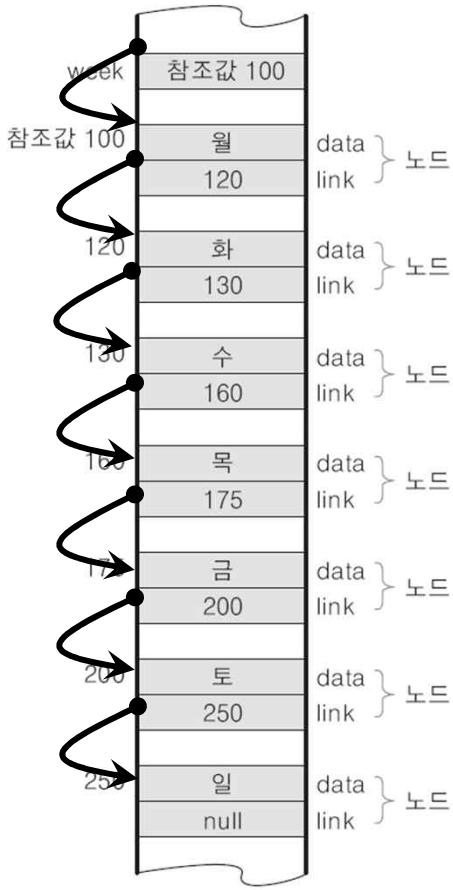
(b) 물리구조

❑ 연결 자료구조

- 리스트 week=(월, 화, 수, 목, 금, 토, 일)
- week에 대한 연결 리스트



(a) 논리구조



(b) 물리구조

- 리스트 이름 week – 연결 리스트의 시작을 가리키는 포인터변수
  - 포인터변수 week는 연결 리스트의 첫번째 노드를 가리키는 동시에 연결된 리스트 전체를 의미

- 연결 리스트의 마지막 노드의 링크필드 – 노드의 끝을 표시하기 위해서 null(널) 저장
- 공백 연결 리스트 – 포인터변수 week에 null을 저장 (널 포인터)
- 각 노드의 필드에 저장한 값은 포인터의 점 연산자를 사용하여 액세스
  - week.data : 포인터 week가 가리키는 노드의 데이터 필드 값 “월”
  - week.link : 포인터 week가 가리키는 노드의 링크 필드에 저장된 주소값
- 리스트 week의 노드에 대한 C 프로그램 구조체 정의

```
typedef struct Node{
        char data[4];
        struct Node* link;
};
```

❑ 연결 자료구조

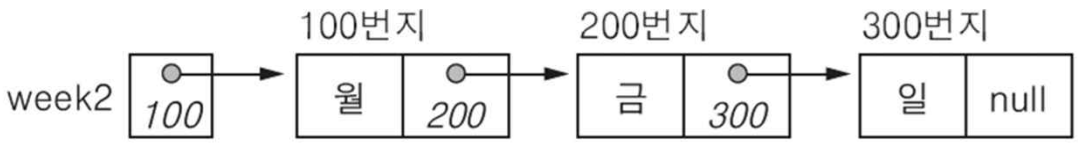
- 리스트 이름 week – 연결 리스트의 시작을 가리키는 포인터변수
  - ➢ 포인터변수 week는 연결 리스트의 첫번째 노드를 가리키는 동시에 연결된 리스트 전체를 의미

- 연결 리스트의 마지막 노드의 링크필드 – 노드의 끝을 표시하기 위해서 null(널) 저장
- 공백 연결 리스트 – 포인터변수 week에 null을 저장 (널 포인터)
- 각 노드의 필드에 저장한 값은 포인터의 점 연산자를 사용하여 액세스
  - ➢ week.data : 포인터 week가 가리키는 노드의 데이터 필드 값 “월”
  - ➢ week.link : 포인터 week가 가리키는 노드의 링크 필드에 저장된 주소값
- 리스트 week의 노드에 대한 C 프로그램 구조체 정의

```
typedef struct Node{
        char data[4];
        struct Node* link;
};
```

❖ 단순 연결 리스트(singly linked list)

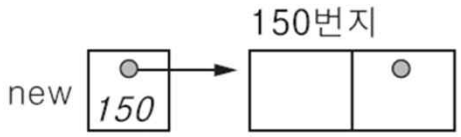
- 노드가 하나의 링크 필드에 의해서 다음 노드와 연결되는 구조를 가진 연결 리스트
- 연결 리스트, 선형 연결 리스트(linear linked list), 단순 연결 선형 리스트(singly linked linear list)
- 예)



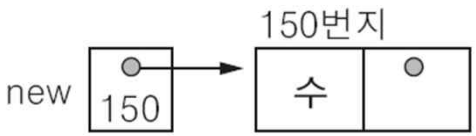
❖ 단순 연결 리스트의 삽입

- 리스트 week2=(월, 금, 일)에서 원소 “월” 과 “금” 사이에 새 원소 “수” 삽입하기

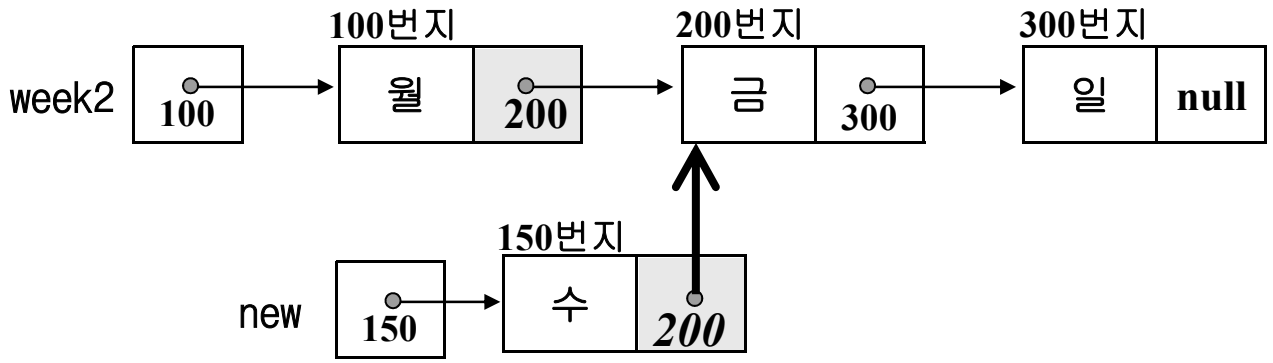
① 삽입할 새 노드를 만들 공백노드를 메모리에서 가져와서 포인터변수 new 가 가리키게 한다.



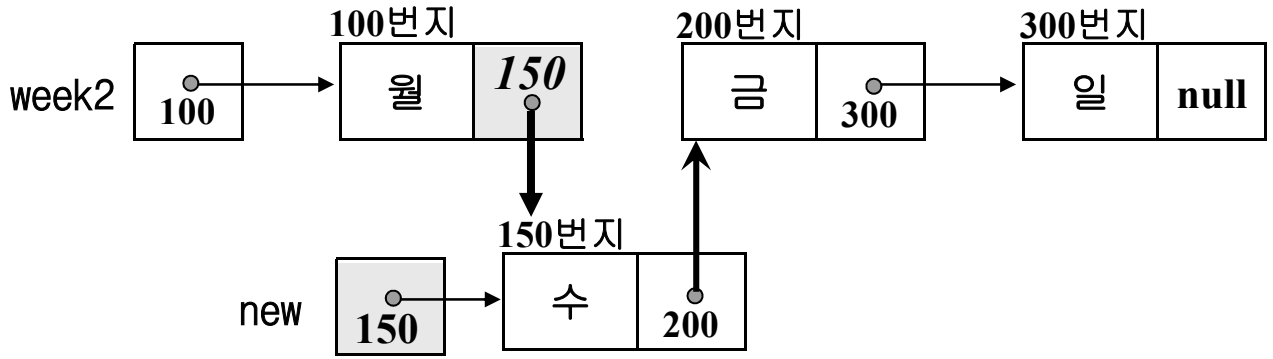
② new의 데이터 필드에 “수” 를 저장한다.



③ new의 앞 노드, 즉 "월" 노드의 링크 필드 값을 new의 링크 필드에 저장



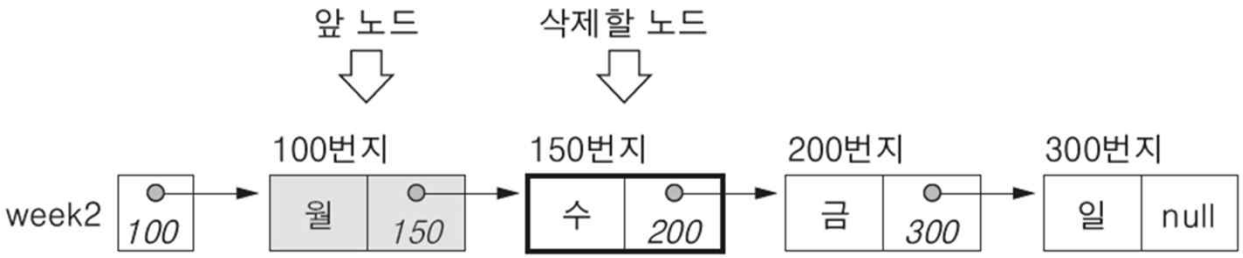
④ "월" 노드의 링크 필드에 new의 값(new가 가리키고 있는 새 노드의 주소)을 저장



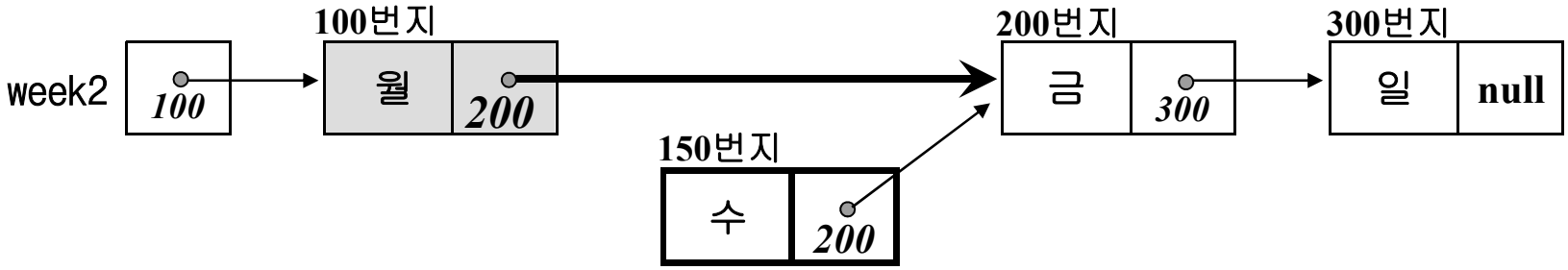
❖ 단순 연결 리스트의 삭제

- 리스트 week2=(월, 수, 금, 일)에서 원소 “수” 삭제하기

① 삭제할 원소의 앞 노드(선행자)를 찾는다.



② 앞 노드의 링크 필드에, 삭제할 원소 “수”의 링크 필드 값을 저장한다.

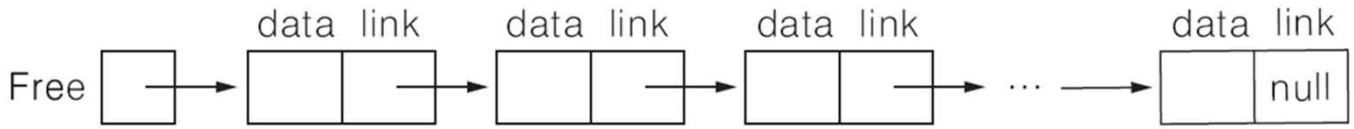


❖ 자유 공간 리스트(free space list)

- 사용하기 전의 메모리나 사용이 끝난 메모리를 관리하기 위해 노드로 구성하여 연결한 리스트

자유공간 리스트에서 대기 중인 노드들

사용 중인 노드들



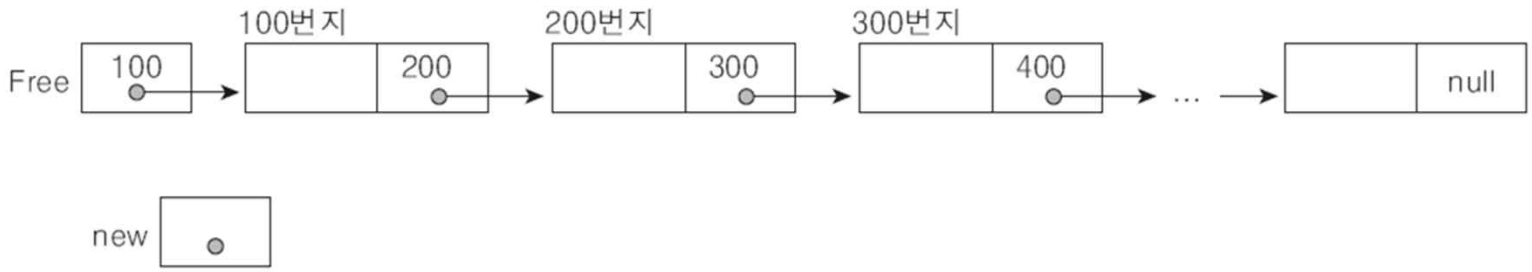
- **자유 공간 리스트에서의 노드 할당 알고리즘**

[알고리즘 6-1]

```
getNode( )
    if (Free = null) then
        underflow( );     // 언더플로우 처리 루틴
    new ← Free;           // ❶
    Free ← Free.link;     // ❷
    return new;
end getNode( )
```

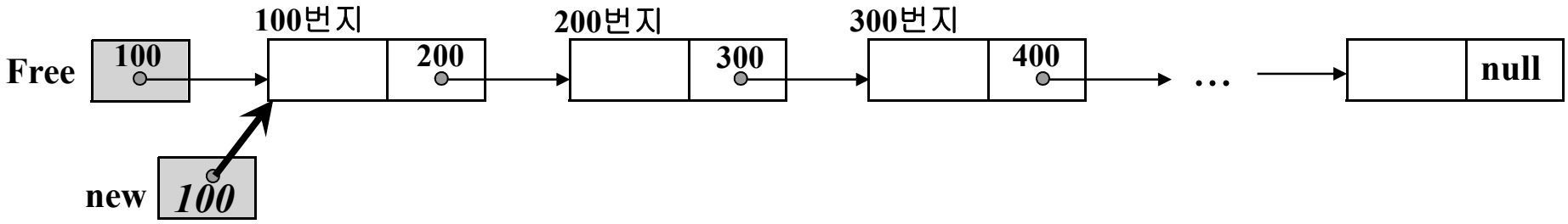
▪ 자유 공간 리스트에서의 노드 할당 과정

- 자유공간 리스트 free에서 노드를 할당할 때는 항상 첫 번째 노드 할당
- 초기 상태



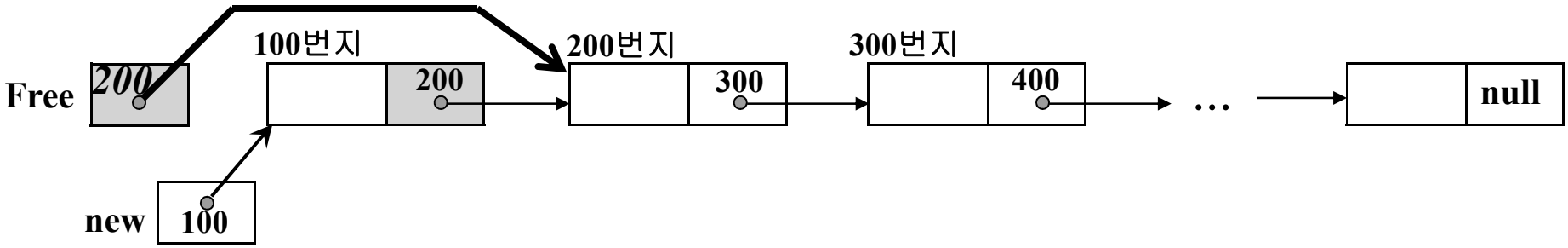
① **new ← Free;**

리스트 free의 첫 번째 노드의 주소를 포인터 new에 저장하여 포인터 new가 할당할 노드를 가리키게 한다.



② **Free ← Free.link;**

포인터 free에 리스트의 두 번째 노드의 주소(Free.link) 저장



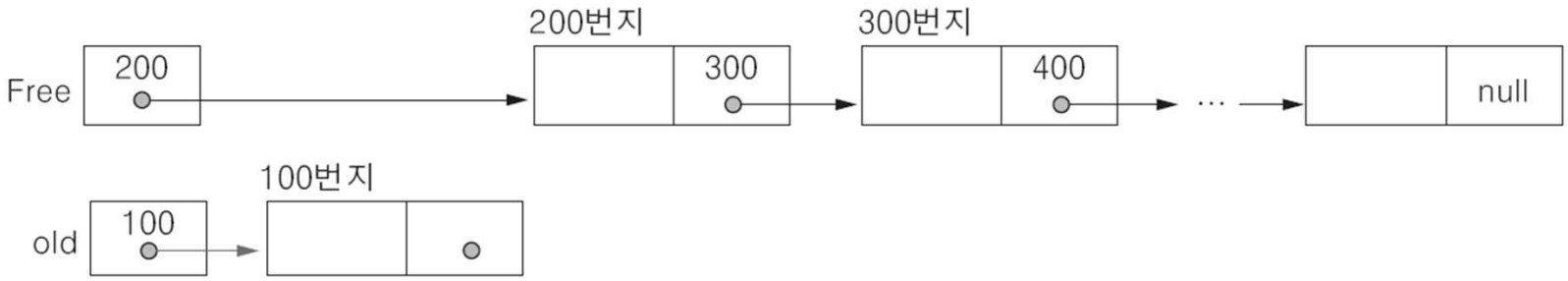
- 이제 자유공간 리스트 free의 첫 번째 노드는 리스트에서 의미적으로 분리된 상태이므로 포인터 new를 반환(return new;)해줌으로써 새 노드를 할당

- **자유 공간 리스트로의 노드 반환 알고리즘**

[알고리즘 6-2]

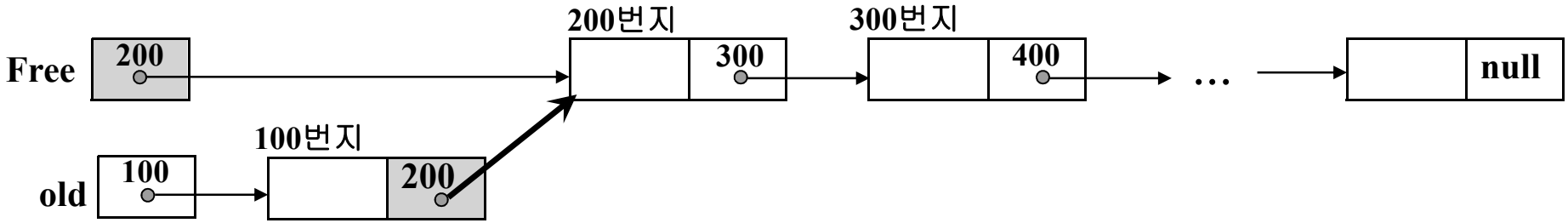
```
returnNode(old)
    old.link ← Free; // ❶
    Free ← old; // ❷
end returnNode( )
```

- **자유 공간 리스트로의 노드 반환 과정**
  - 반환되는 노드는 자유공간 리스트 free의 첫 번째 노드로 다시 삽입
  - 초기 상태



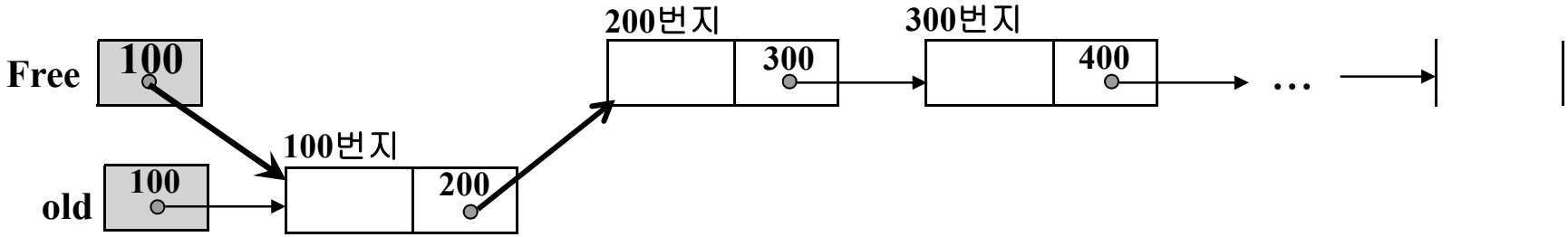
① **old.link ← Free;**

자유 공간리스트 free의 첫 번째 노드주소를 반환할 노드 포인터 old.link에 저장하여 포인터 old의 노드가 리스트 free의 첫 번째 노드를 가리키게 한다.



② **Free ← old;**

포인터 old의 값 즉, 반환할 노드의 주소를 포인터 free에 저장하여 리스트 free의 첫 번째 노드로 지정



❖ 단순 연결 리스트의 삽입 알고리즘

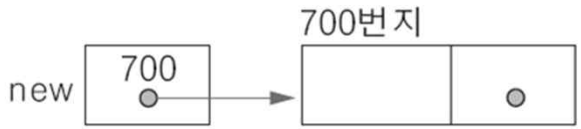
▪ 첫 번째 노드로 삽입하기

[알고리즘 6-3]

```
returnNode(old)
    old.link ← Free; // ❶
    Free ← old; // ❷
end returnNode( )
```

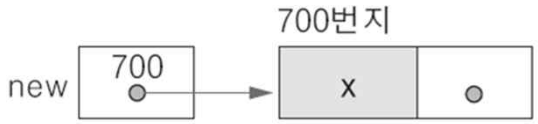
① **new ← getNode( );**

삽입할 노드를 자유 공간리스트에서 할당받는다.



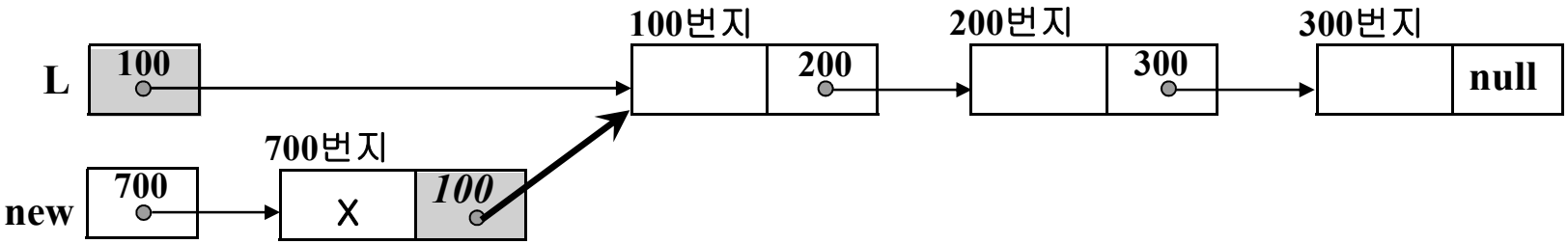
② **new.data ← x;**

새 노드의 데이터 필드에 x를 저장



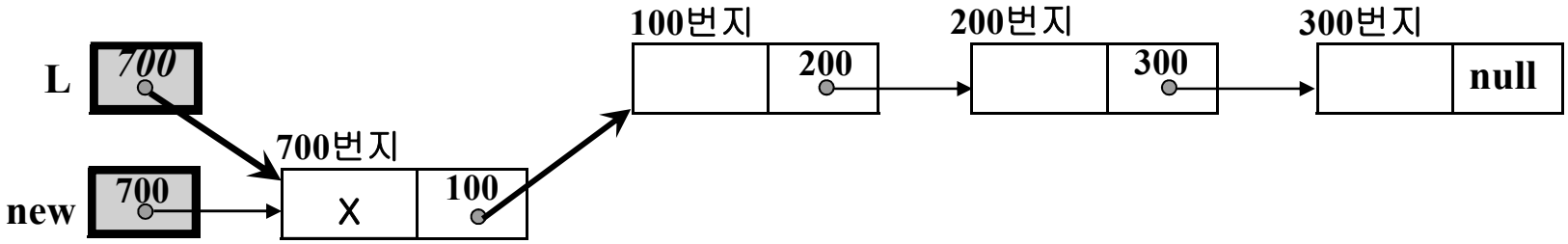
③ **new.link ← L;**

삽입할 노드를 연결하기 위해서 리스트의 첫 번째 노드 주소를 삽입할 새 노드 new의 링크 필드에 저장하여, 새 노드 new가 리스트의 첫 번째 노드를 가리키게 한다.



④ **L ← new;**

리스트의 첫 번째 노드 주소를 저장하고 있는 포인터 L에, 새 노드의 주소를 저장하여, 포인터 L이 새 노드를 첫 번째 노드로 가리키도록 지정



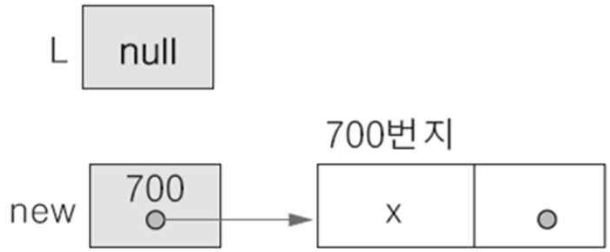
▪ 중간 노드로 삽입하기

- 리스트 L에서 포인터변수 pre가 가리키고 있는 노드의 다음에 데이터 필드 값이 x인 새 노드를 삽입하는 알고리즘
- 리스트의 중간 노드 삽입 알고리즘

[알고리즘 6-4]

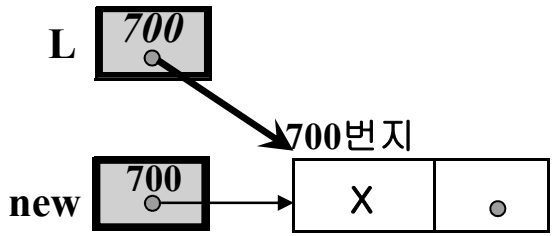
```
insertMiddleNode(L, pre, x)
    new ← getNode( );
    new.data ← x;
    if (L=null) then {      // ❶
        L ← new;            // ❶-ⓐ
        new.link ← null;    // ❶-ⓑ
    }
    else {                  // ❷
        new.link ← pre.link;  // ❷-ⓐ
        pre.link ← new;       // ❷-ⓑ
    }
end insertMiddleNode( )
```

➢ 리스트 L이 공백 리스트인 경우



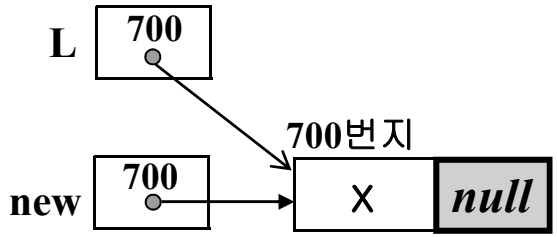
① L ← new;

리스트 포인터 L에 새 노드 new의 주소를 저장하여, 새 노드 new가 리스트의 첫 번째 노드가 되게 한다.

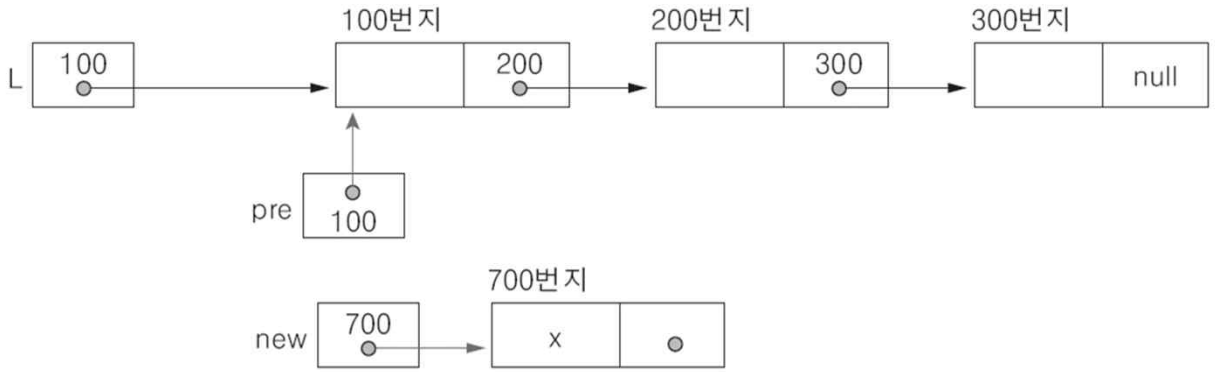


② new.link ← null;

리스트의 마지막 노드 new의 링크 필드에 null을 저장하여 마지막 노드임을 표시

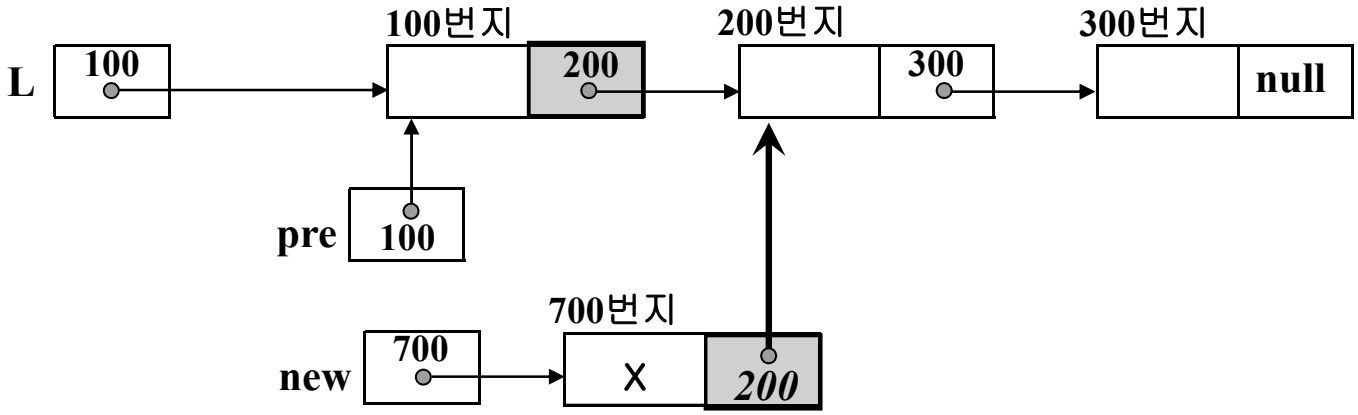


➢ 리스트 L이 공백 리스트가 아닌 경우



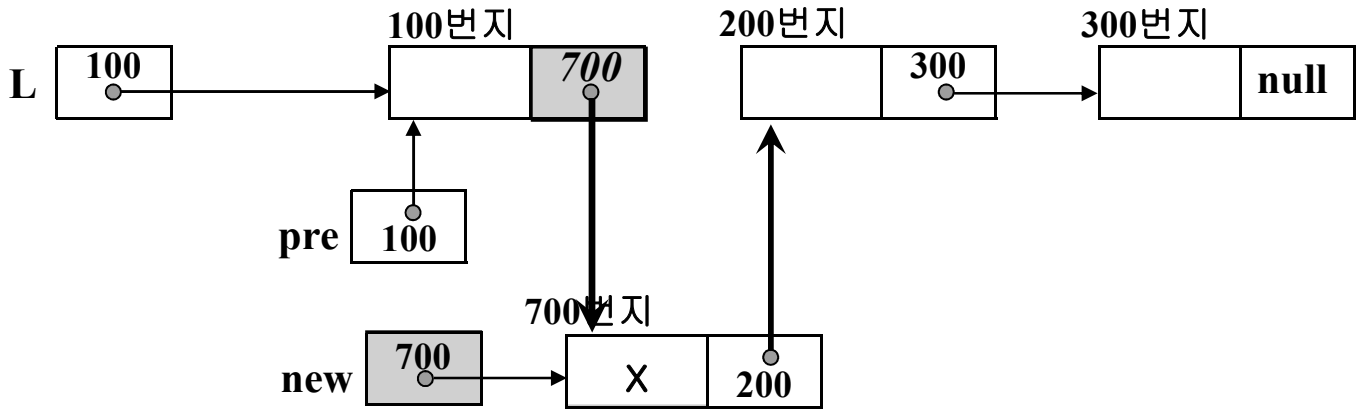
① **new.link ← pre.link;**

노드 pre의 링크 필드 값을 노드 new의 링크 필드에 저장하여, 새 노드 new가 노드 pre의 다음 노드를 가리키도록 한다.



② **pre.link ← new;**

포인터 new의 값을 노드 pre의 링크 필드에 저장하여, 노드 pre가 새 노드 new를 다음 노드로 가리키도록 한다.



▪ 마지막 노드로 삽입하기

- 새 노드 new를 마지막 노드로 삽입하는 알고리즘

[알고리즘 6-5]

```
insertLastNode(L, x)
    new ← getNode();
    new.data ← x;
    new.link ← null;
    if (L = null) then {              // ❶-ⓐ
        L ← new;                      // ❶-ⓑ
        return;
    }
    temp ← L;                         // ❷-ⓐ
    while (temp.link ≠ null) do       // ❷-ⓑ
            temp ← temp.link;
    temp.link ← new;                  // ❷-ⓒ
end insertLastNode()
```

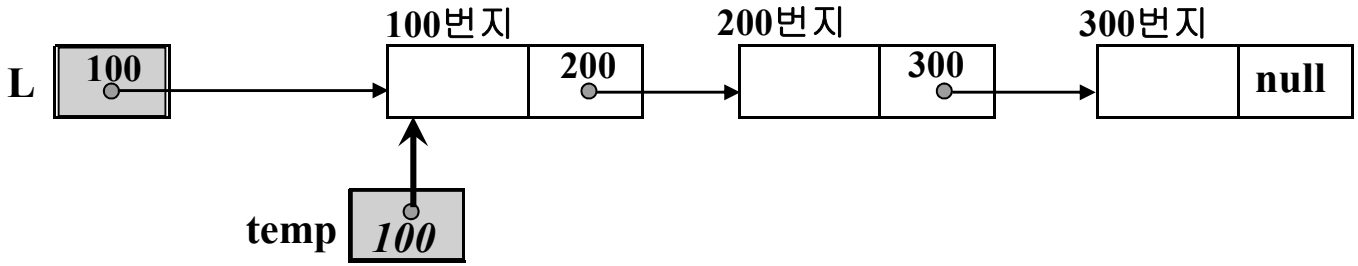
➢ 리스트 L이 공백 리스트인 경우

- ➢ [알고리즘 6-4]에서 공백리스트에 노드를 삽입하는 연산과 같은 연산
- ➢ 삽입하는 새 노드 new는 리스트 L의 첫 번째 노드이자 마지막 노드

➢ 리스트 L이 공백 리스트인 경우

① temp ← L;

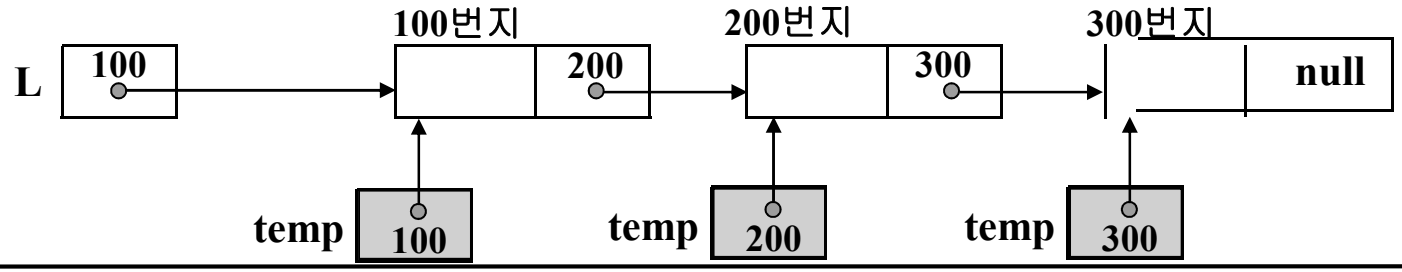
현재 리스트 L의 마지막 노드를 찾기위해 노드를 순회할 임시포인터 temp에 리스트의 첫 번째 노드의 주소를 지정



② while (temp.link ≠ null) do

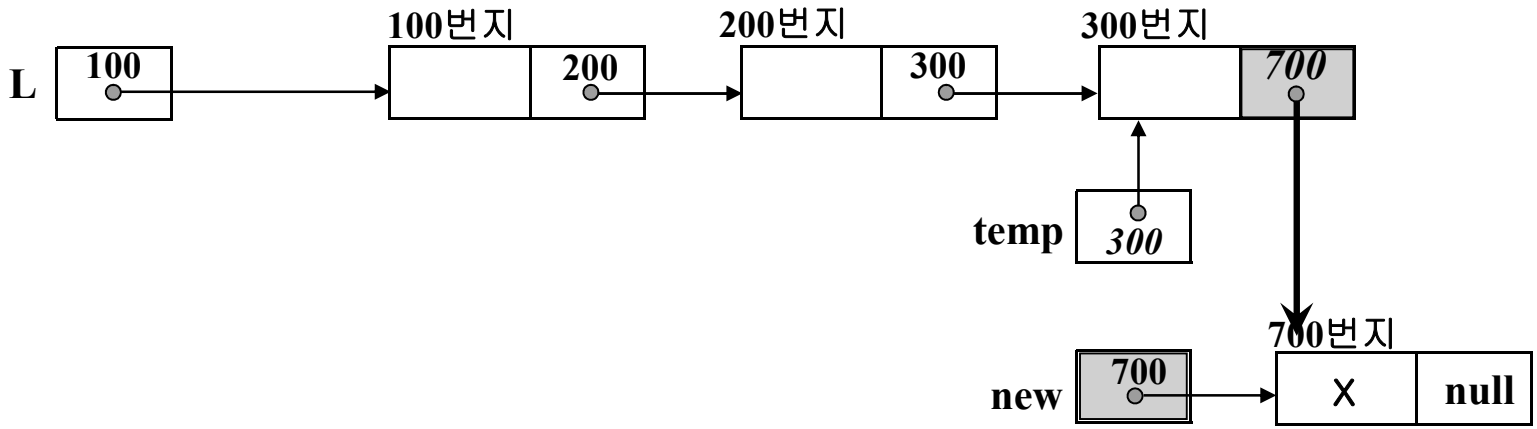
temp ← temp.link;

while 반복문을 수행하여 순회포인터 temp가 노드의 링크 필드를 따라 이동하면서 링크 필드가 null인 마지막 노드 찾기 수행



③ **temp.link ← new;**

순회포인터 temp가 가리키는 노드 즉, 리스트의 마지막 노드의 링크 필드에 삽입할 새 노드 new의 주소를 저장하여, 리스트의 마지막 노드가 새 노드 new를 연결



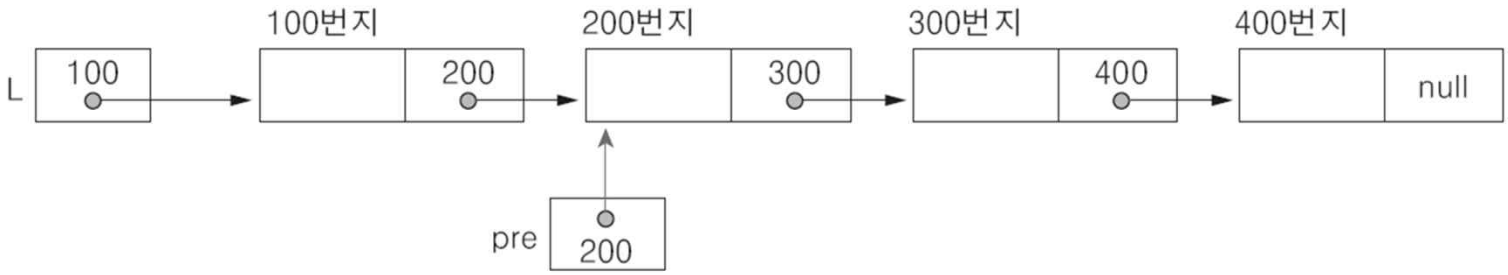
❖ 단순 연결 리스트의 삭제 알고리즘

- 리스트 L에서 포인터 pre가 가리키는 노드의 다음 노드를 삭제하는 알고리즘
  - 포인터 old : 삭제할 노드

[알고리즘 6-6]

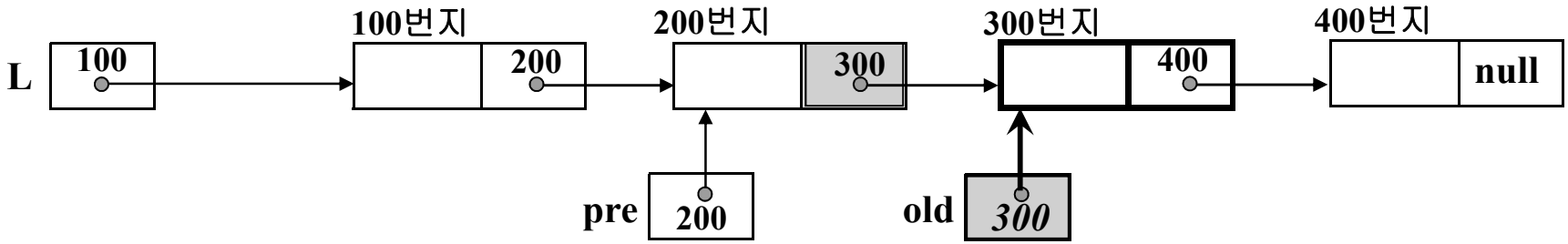
```
deleteNode(L, pre)
    if (L = null) then error;          // ❶
    else {                             // ❷
        old ← pre.link;                // ❷-ⓐ
        if (old = null) then return;   // ❸
        pre.link ← old.link;           // ❷-ⓑ
    }
    returnNode(old);                   // ❷-ⓒ
end deleteNode( )
```

➢ 리스트 L이 공백 리스트가 아닌 경우



① **old ← pre.link;**

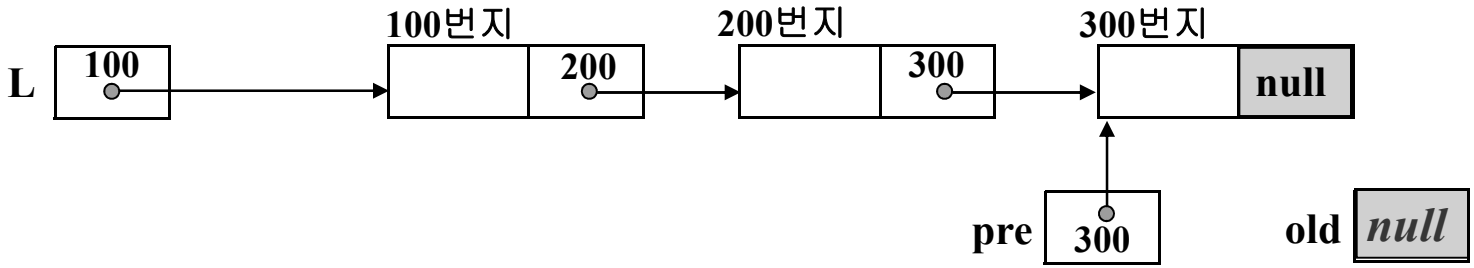
노드 pre의 다음노드의 주소를 포인터 old에 저장하여, 포인터 old가 다음 노드를 가리키게 한다.



② **if (old = null) then return;**

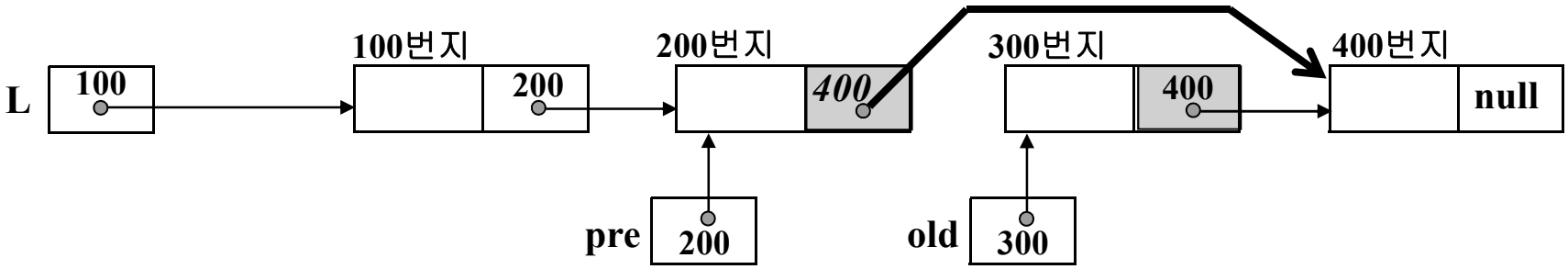
만약 노드 pre가 리스트의 마지막 노드였다면 :

- pre.link의 값은 null이므로 포인터 old의 값은 null이 된다.
  결국 노드 pre 다음의 삭제할 노드가 없다는 의미이므로
  삭제연산을 수행하지 못하고 반환(return).



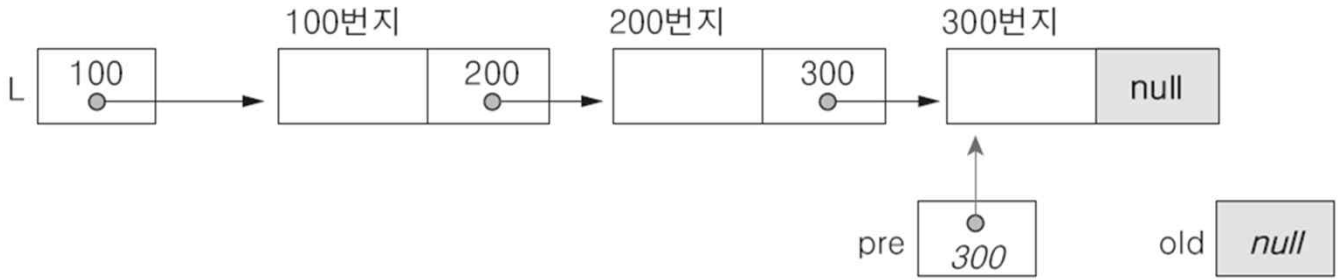
③ **pre.link ← old.link;**

삭제할 노드 old의 다음 노드(old.link)를 노드 pre의 다음 노드(pre.link)로 연결



④ **returnNode(old);**

삭제한 노드 old를 자유 공간리스트에 반환



❖ 단순 연결 리스트의 노드 탐색 알고리즘

- 리스트의 노드를 처음부터 하나씩 순회하면서 노드의 데이터 필드의 값과 x를 비교하여 일치하는 노드를 찾는 연산

[알고리즘 6-7]

```
searchNode(L, x)
    temp ← L;                                    // ❶
    while (temp ≠ null) do {
        if (temp.data = x ) then return temp;    // ❷
            temp ← temp.link;
        }
        return temp;                             // ❸
end searchNode()
```

- 단순 연결 리스트 프로그램

[예제 6-1]

```
001 public class Ex6_1{
002     public static void main(String args[]){
003         LinkedList L = new LinkedList();
004         System.out.println("(1) 공백 리스트에 노드 3개 삽입하기");
005         L.insertLastNode("월");
006         L.insertLastNode("수");
007         L.insertLastNode("일");
008         L.printList();
009
010         System.out.println("(2) 수 노드 뒤에 금 노드 삽입하기");
011         ListNode pre = L.searchNode("수");
012         if(pre == null)
013             System.out.println("검색실패>> 찾는 데이터가 없습니다.");
014         else{
015             L.insertMiddleNode(pre, "금");
016             L.printList();
017         }
```

- 단순 연결 리스트 프로그램

[예제 6-1]

```
018
019             System.out.println("(3) 리스트의 노드를 역순으로 바꾸기");
020             L.reverseList();
021             L.printList();
022
023             System.out.println("(4) 리스트의 마지막 노드 삭제하기");
024             L.deleteLastNode();
025             L.printList();
026         }
027 }
028
029 class LinkedList{
030         private ListNode head;
031         public LinkedList(){
032             head = null;
033         }
034         public void insertMiddleNode(ListNode pre, String data){
035             ListNode newNode = new ListNode(data);
```

- 단순 연결 리스트 프로그램

[예제 6-1]

```
036         newNode.link = pre.link;
037         pre.link = newNode;
038     }
039     public void insertLastNode(String data){
040         ListNode newNode = new ListNode(data);
041         if(head == null){
042             this.head = newNode;
043         }
044         else{
045             ListNode temp = head;
046             while(temp.link != null) temp = temp.link;
047             temp.link = newNode;
048         }
049     }
050     public void deleteLastNode(){
051         ListNode pre, temp;
052         if(head == null) return;
```

- 단순 연결 리스트 프로그램

[예제 6-1]

```
053         if(head.link == null){
054             head = null;
055         }
056         else{
057             pre = head;
058             temp = head.link;
059             while(temp.link != null){
060                 pre = temp;
061                 temp = temp.link;
062             }
063             pre.link = null;
064         }
065     }
066     public ListNode searchNode(String data){
067         ListNode temp = this.head;
068         while(temp != null){
069             if(data == temp.getData())
```

- 단순 연결 리스트 프로그램

[예제 6-1]

```
070                 return temp;
071                 else temp = temp.link;
072             }
073             return temp;
074         }
075         public void reverseList(){
076             ListNode next = head;
077             ListNode current = null;
078             ListNode pre = null;
079             while(next != null){
080                 pre = current;
081                 current = next;
082                 next = next.link;
083                 current.link = pre;
084             }
085             head = current;
086         }
```

- 단순 연결 리스트 프로그램

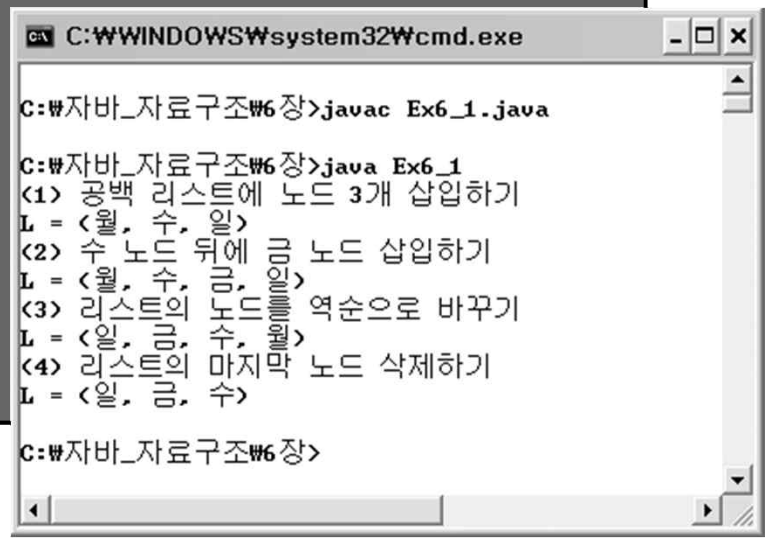
[예제 6-1]

```
087     public void printList(){
088         ListNode temp = this.head;
089         System.out.printf("L = (");
090         while(temp != null){
091             System.out.printf(temp.getData());
092             temp = temp.link;
093             if(temp != null){
094                 System.out.printf(", ");
095             }
096         }
097         System.out.println(")");
098     }
099 }
100
101 class ListNode{
102     private String data;
103     public ListNode link;
```

- 단순 연결 리스트 프로그램

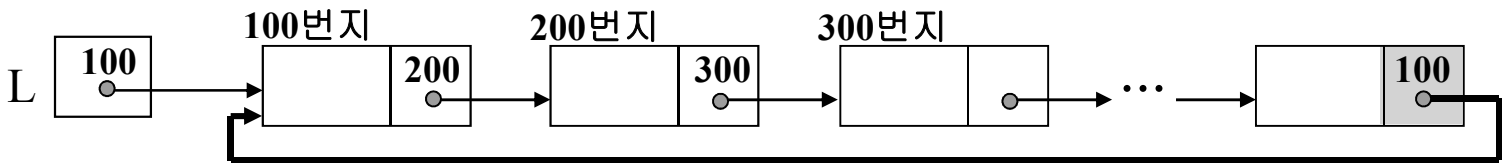
[예제 6-1]

```
104     public ListNode(){
105         this.data = null;
106         this.link = null;
107     }
108     public ListNode(String data){
109         this.data = data;
110         this.link = null;
111     }
112     public ListNode(String data, ListNode link){
113         this.data = data;
114         this.link = link;
115     }
116     public String getData(){
117         return this.data;
118     }
119 }
```



❖ 원형 연결 리스트(circular linked list)

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



❖ 원형 연결 리스트의 삽입 연산

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

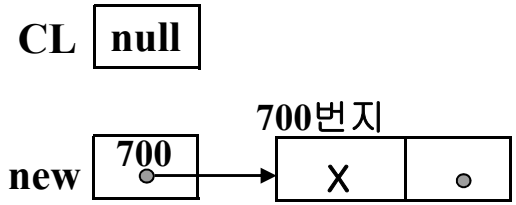
[알고리즘 6-8]

```
insertFirstNode(CL, x)
    new ← getNode();
    new.data ← x;
    if (CL = null) then { // ❶
        CL ← new; // ❶-ⓐ
        new.link ← new; // ❶-ⓑ
    }
    else{ // ❷
        temp ← CL; // ❷-ⓐ
        while (temp.link ≠ CL) do // ❷-ⓑ
        temp ← temp.link;
    new.link ← temp.link; // ❷-ⓒ
        temp.link ← new; // ❷-ⓓ
        CL ← new; // ❷-ⓔ
    }
end insertFirstNode()
```

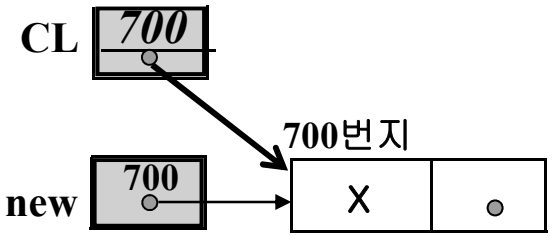
➢ 원형리스트가 공백 리스트인 경우

- 삽입하는 노드new는 리스트의 첫 번째 노드이자 마지막 노드가 되어야 함

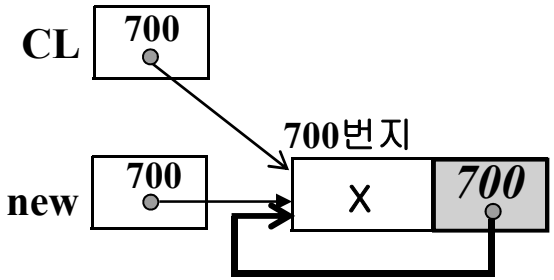
[초기상태]



① **CL ← new;**
포인터 CL이 노드 new를 가리키게 한다.



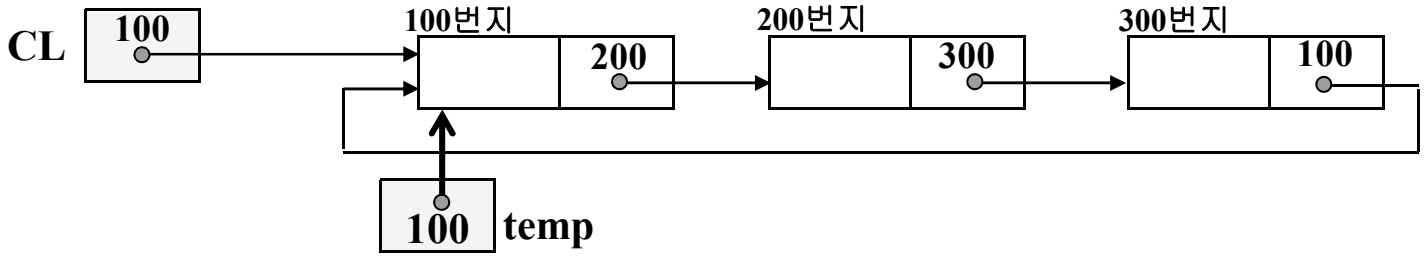
② **new.link ← new;**
노드 new가 자기자신을 가리키게 함으로써 노드 new를 첫 번째 노드이자 마지막 노드가 되도록 지정한다.



➢ 원형리스트가 공백 리스트가 아닌 경우

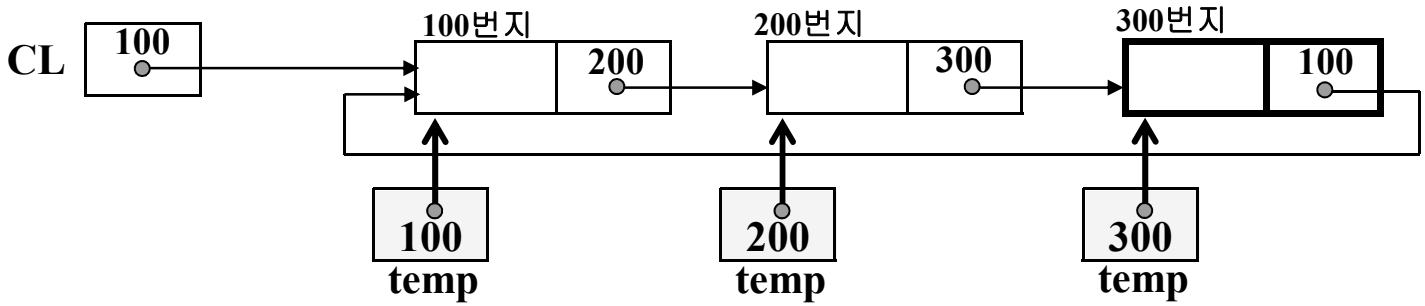
① temp ← CL;

리스트가 공백리스트가 아닌 경우에는 첫 번째 노드의 주소를 임시 순회 포인터 temp에 저장하여 노드 순회의 시작점을 지정한다.



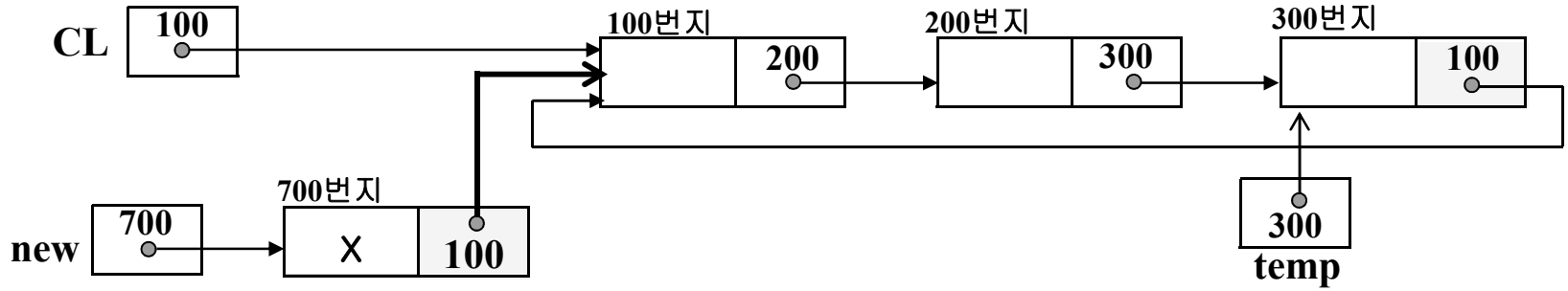
② while (temp.link ≠ CL) do temp ← temp.link;

while 반복문을 수행하여 순회 포인터 temp를 링크를 따라 마지막 노드까지 이동



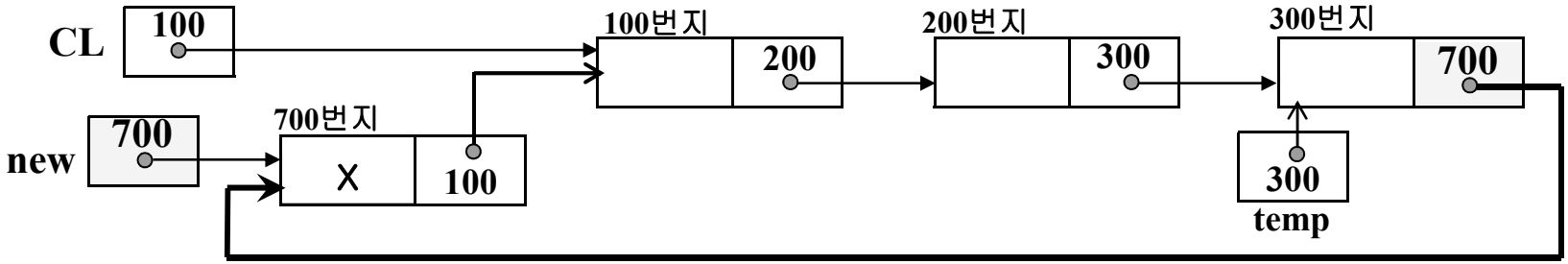
③ new.link ← temp.link;

리스트의 마지막 노드의 링크 값을 노드 new의 링크에 저장하여, 노드 new가 노드 temp의 다음 노드를 가리키게 한다. 리스트 CL은 원형 연결 리스트이므로 마지막 노드의 다음 노드는 리스트의 첫 번째 노드가 된다.



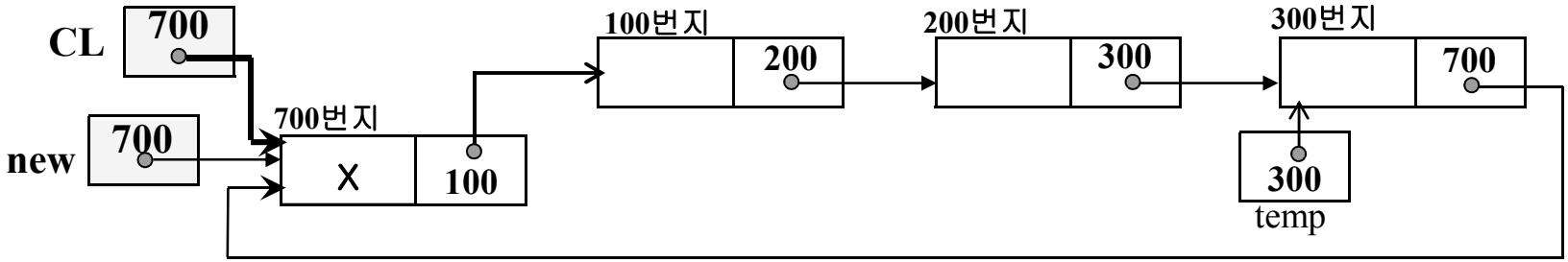
④ temp.link ← new;

포인터 new의 값을 포인터 temp가 가리키고 있는 마지막 노드의 링크에 저장하여, 리스트의 마지막 노드가 노드 new를 가리키게 한다.

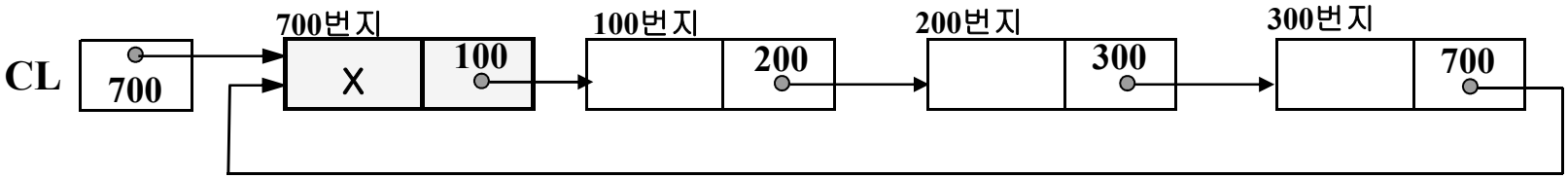


⑤ **CL ← new;**

- 노드 new의 주소를 리스트 포인터 CL에 저장하여 노드 new가 리스트의 첫 번째 노드가 되도록 지정



- **[알고리즘 6-8] 실행 결과**

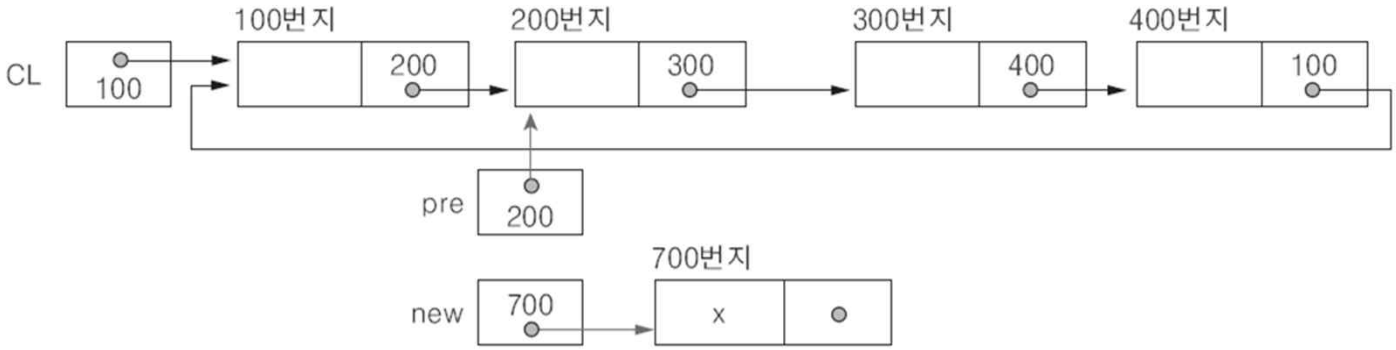


▪ 중간 노드로 삽입하기

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

[알고리즘 6-9]

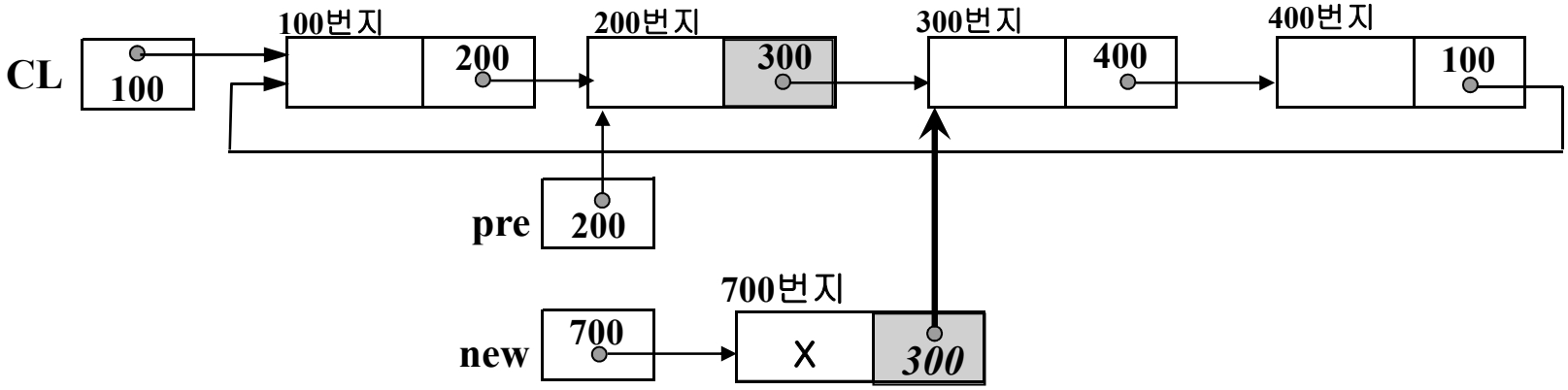
```
insertMiddleNode(CL, pre, x)
    new ← getNode();
    new.data ← x;
    if (CL=null) then { // ❶
        CL ← new;
        new.link ← new;
    }
    else { // ❷
        new.link ← pre.link; // ❷-ⓐ
        pre.link ← new; // ❷-ⓑ
    }
end insertMiddleNode()
```



① **new.link ← pre.link;**

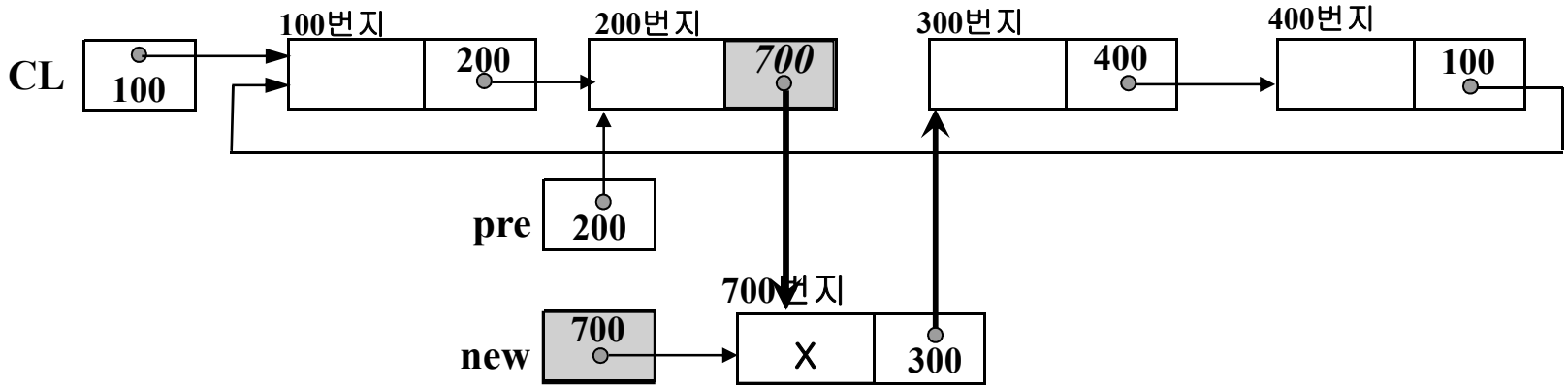
노드 pre의 다음 노드로 new를 삽입하기 위해서,

먼저 노드 pre의 다음 노드(pre.link)를 new의 다음 노드(new.link)로 연결



② **pre.link ← new;**

노드 new의 주소를 노드 pre의 링크에 저장하여, 노드 pre가 노드 new를 가리키도록 한다.



❖ 원형 연결 리스트의 삭제 연산

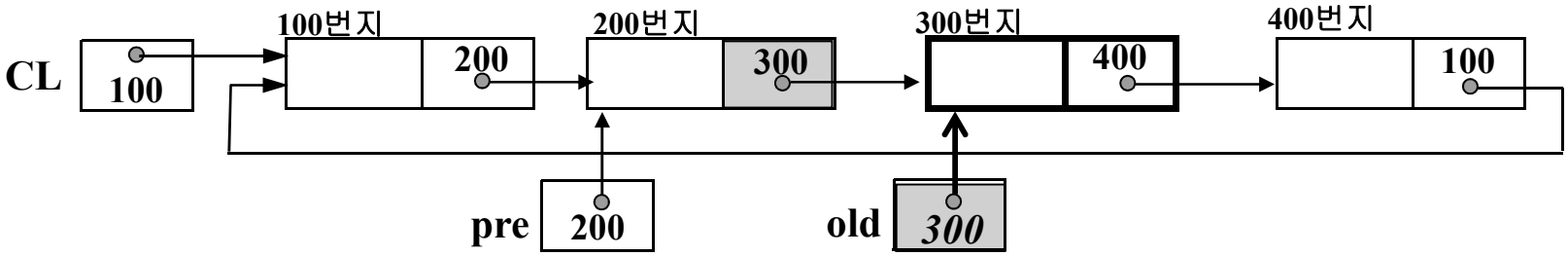
- **원형 연결 리스트 CL에서 포인터 pre가 가리키는 노드의 다음 노드를 삭제하고 삭제한 노드는 자유공간 리스트에 반환하는 연산**
  - 포인터 old는 삭제할 노드 지시

[알고리즘 6-10]

```
deleteNode(CL, pre)
    if (CL = null) then error;
    else {
        old ← pre.link; // ❶
        pre.link ← old.link; // ❷
        if (old = CL) then // ❸
            CL ← old.link; // ❸-ⓐ
        returnNode(old); // ❹
    }
end deleteNode()
```

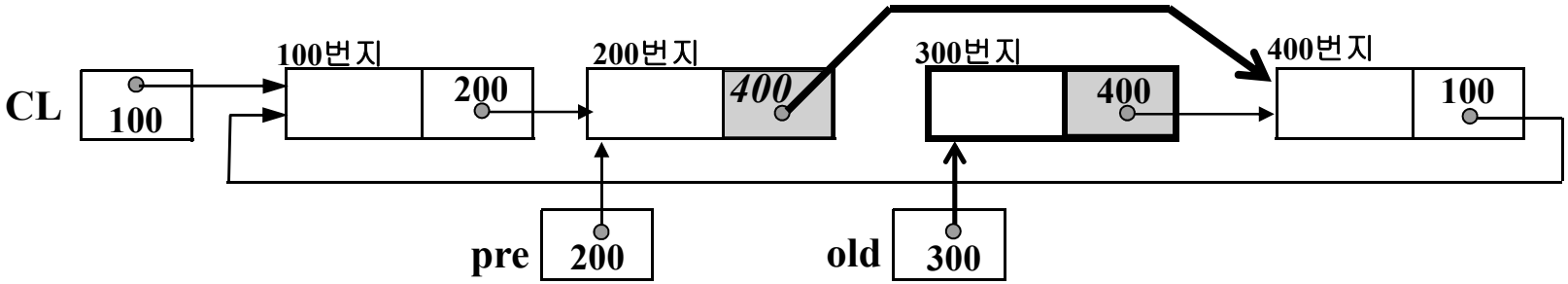
① **old ← pre.link;**

노드 pre의 다음 노드를 삭제할 노드 old로 지정

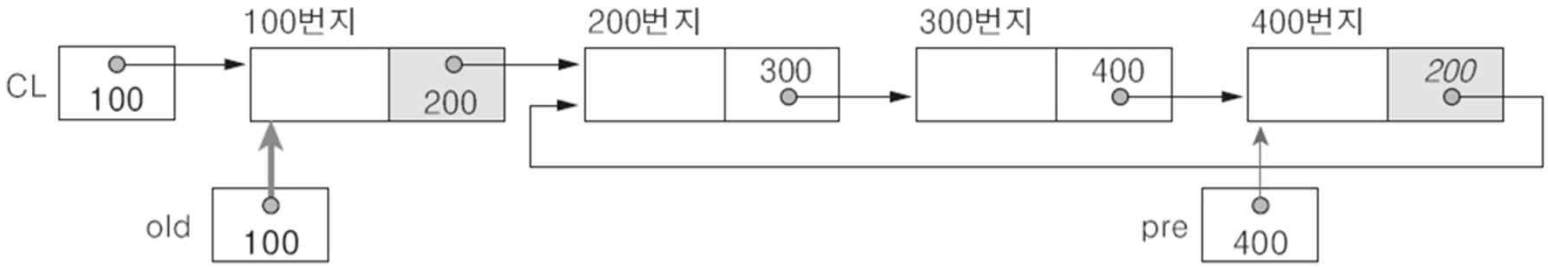


② **pre.link ← old.link;**

노드 old의 이전 노드와 다음노드를 서로 연결

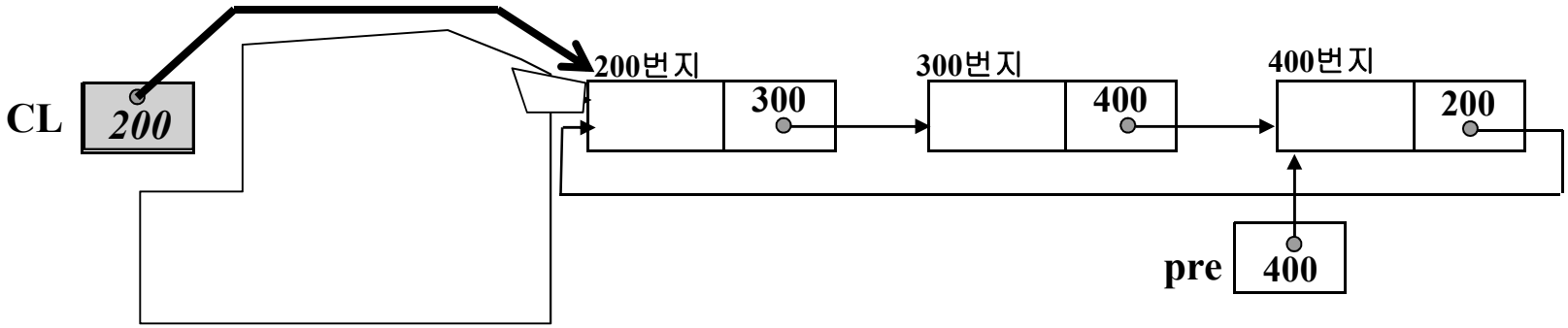


➢ 삭제할 노드 old가 리스트 포인터 CL인 경우



① CL ← old.link;

첫 번째 노드를 삭제하는 경우에는 노드 old의 링크 값을 리스트 포인터 CL에 저장하여 두 번째 노드가 리스트의 첫 번째 노드가 되도록 조정



❖ 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
  - llink(left link) 필드 : 왼쪽노드와 연결하는 포인터
  - rlink(right link) 필드 : 오른쪽 노드와 연결하는 포인터

- 노드 구조에 대한 구조체 정의

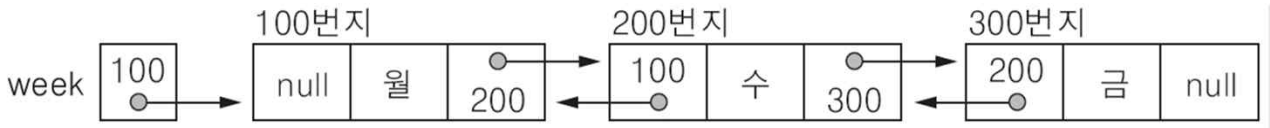
```
public class DblNode{
        DblNode llink;
        String data;
        DblNode rlink;
  } ;
```

❑ 이중 연결 리스트

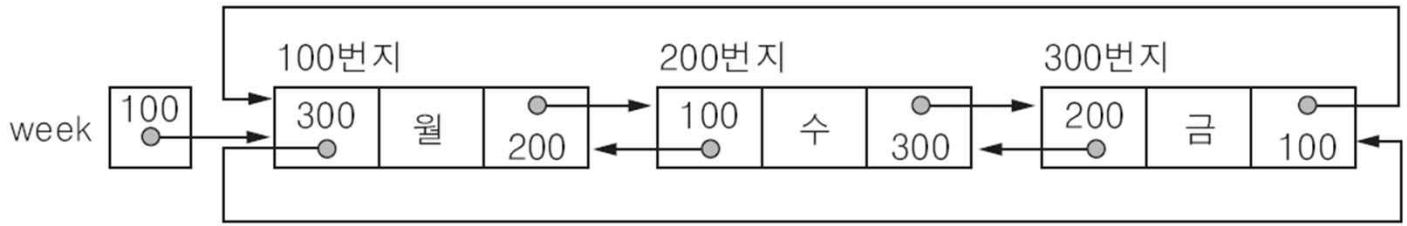
- 단순 연결 기차

- 이중 연결 기차

- 리스트 week=(월, 수, 금)의 이중 연결 리스트 구성

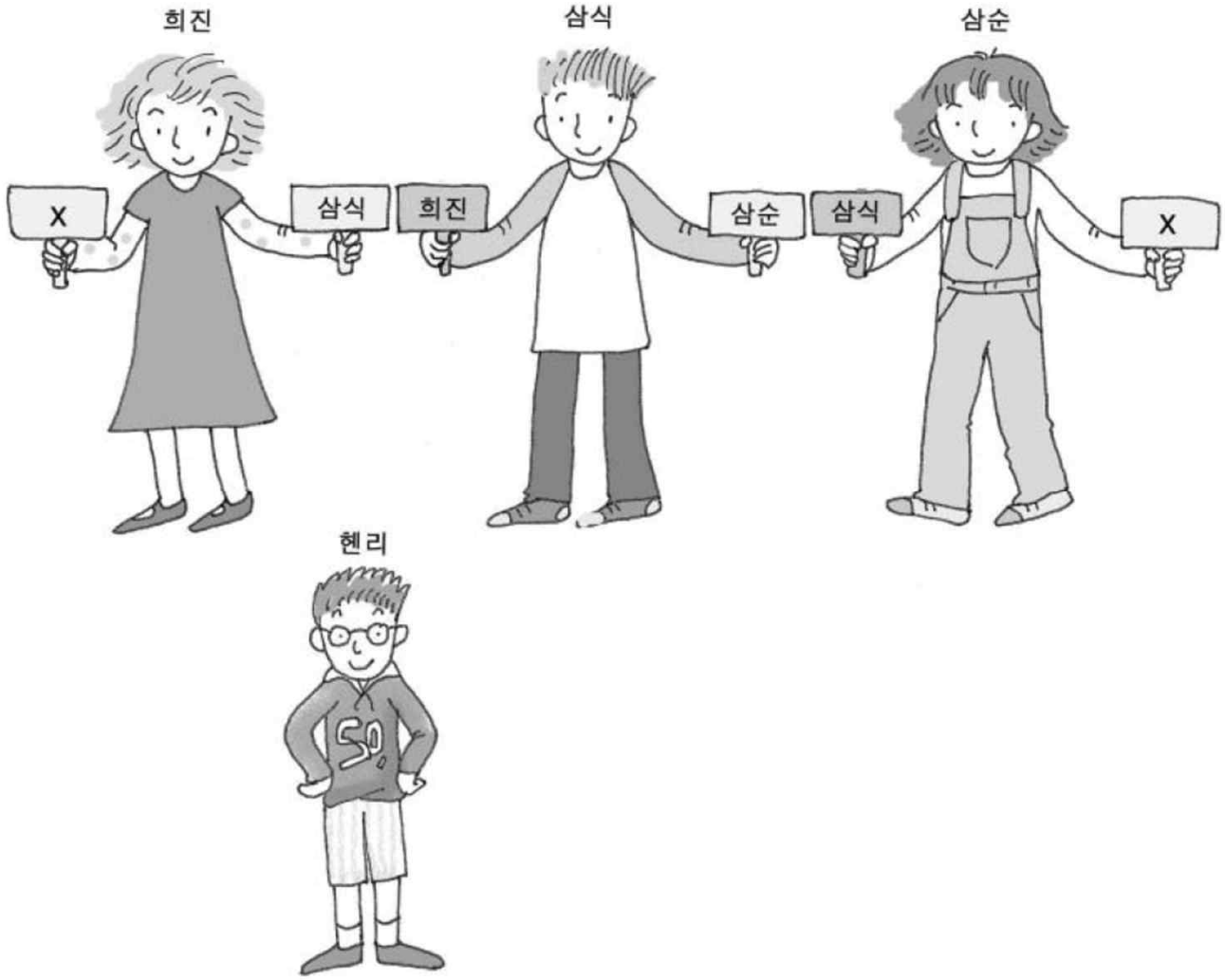


- 원형 이중 연결 리스트
  - 이중 연결 리스트를 원형으로 구성



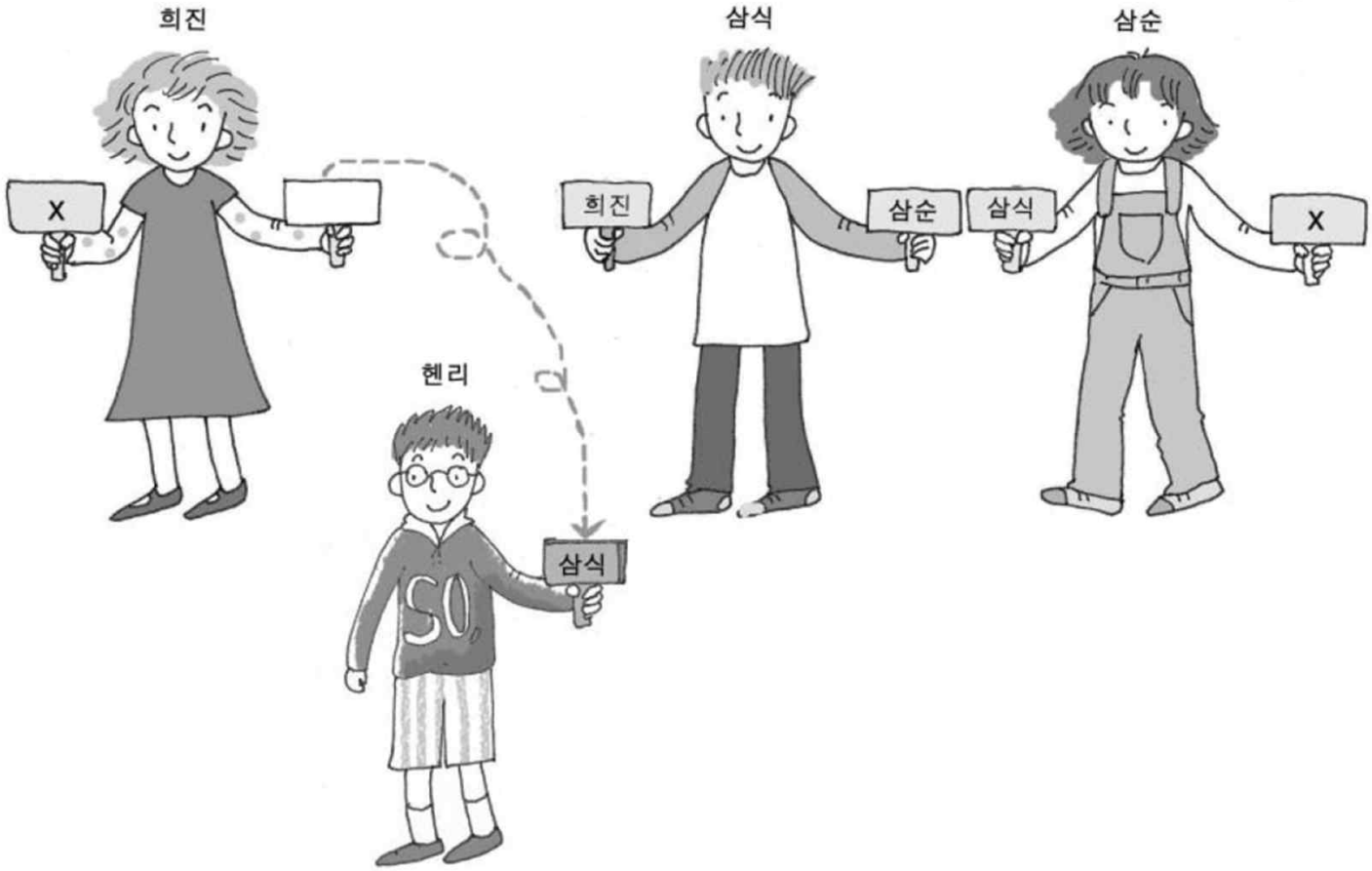
❑ 이중 연결 리스트

- 양방향 기차 만들기 준비



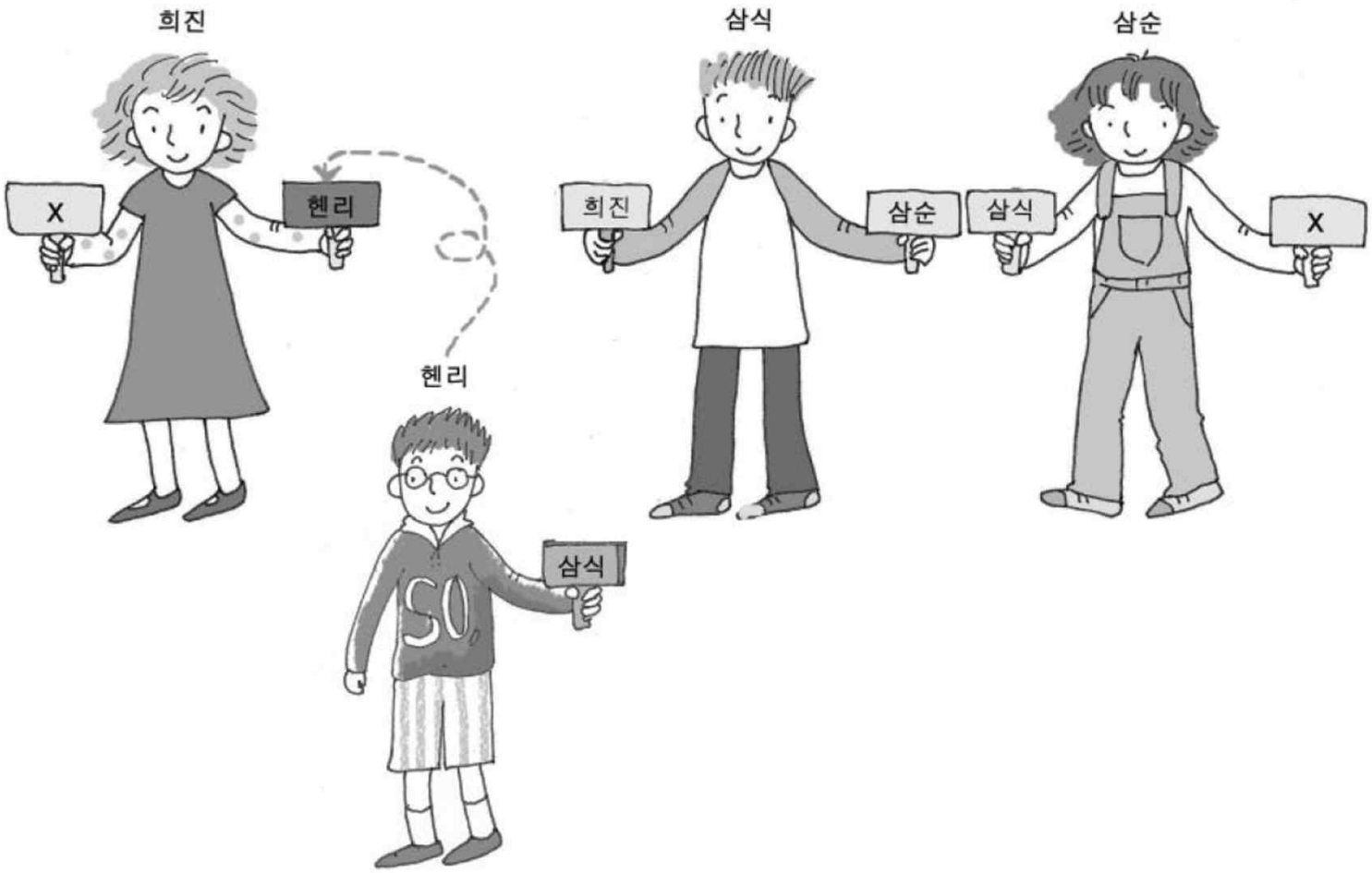
❑ 이중 연결 리스트

- **양방향 기차 만들기 1 : 왼쪽 사람의 오른손 이름표 받기**

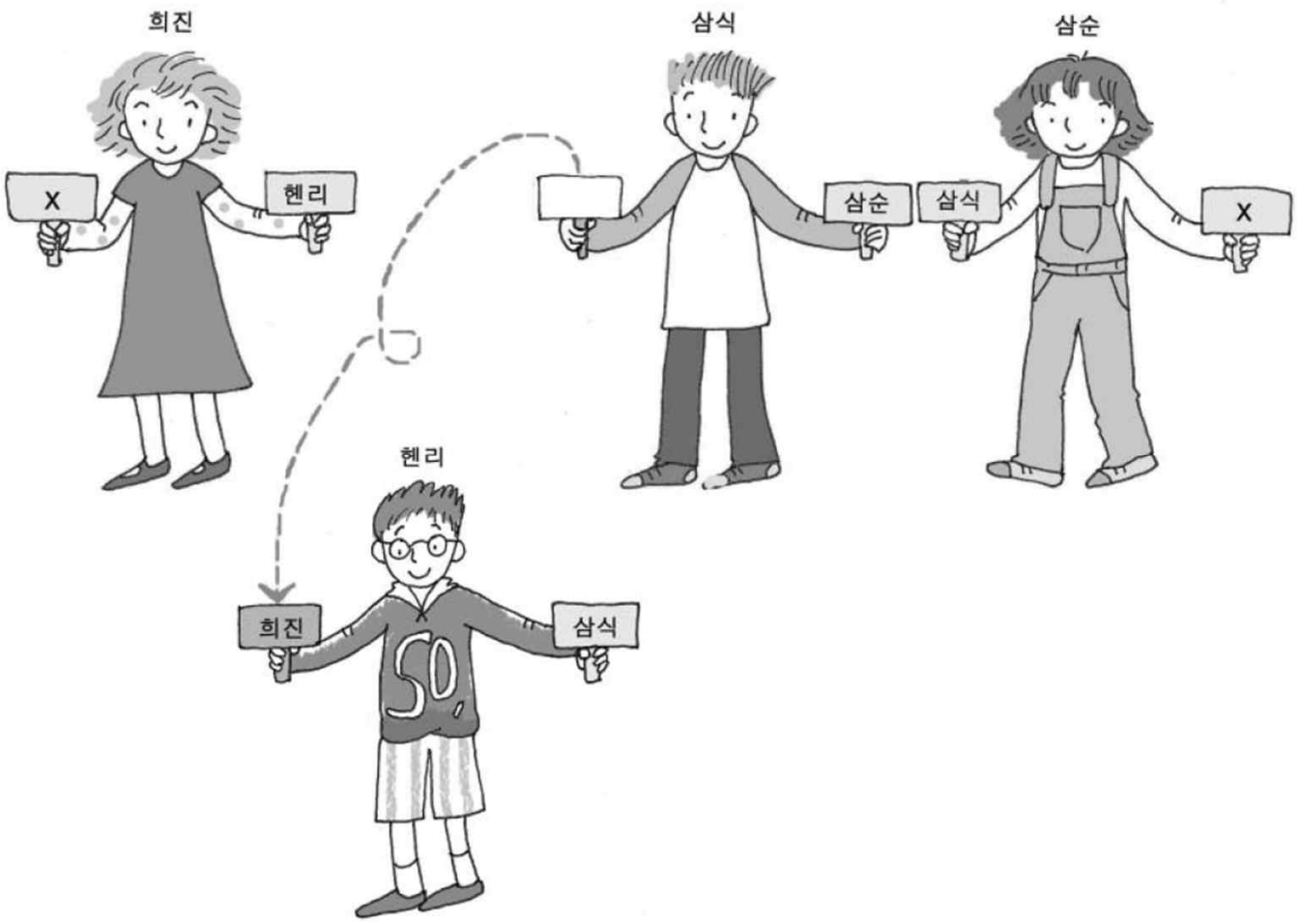


❑ 이중 연결 리스트

- 양방향 기차 만들기 1 : 왼쪽 사람에게 자기 이름 주기

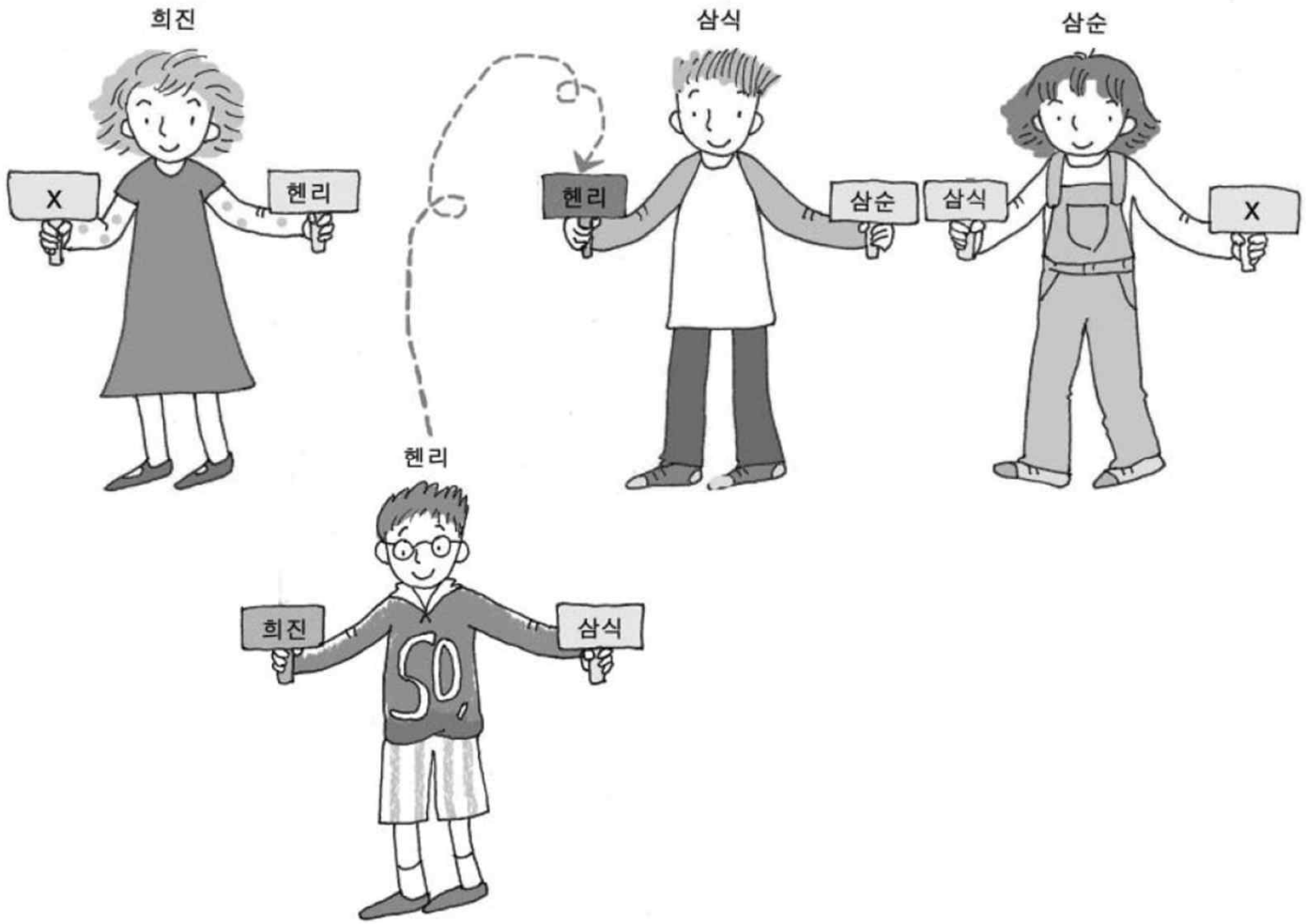


- 양방향 기차 만들기 2 : 오른쪽 사람의 왼손 이름표 받기

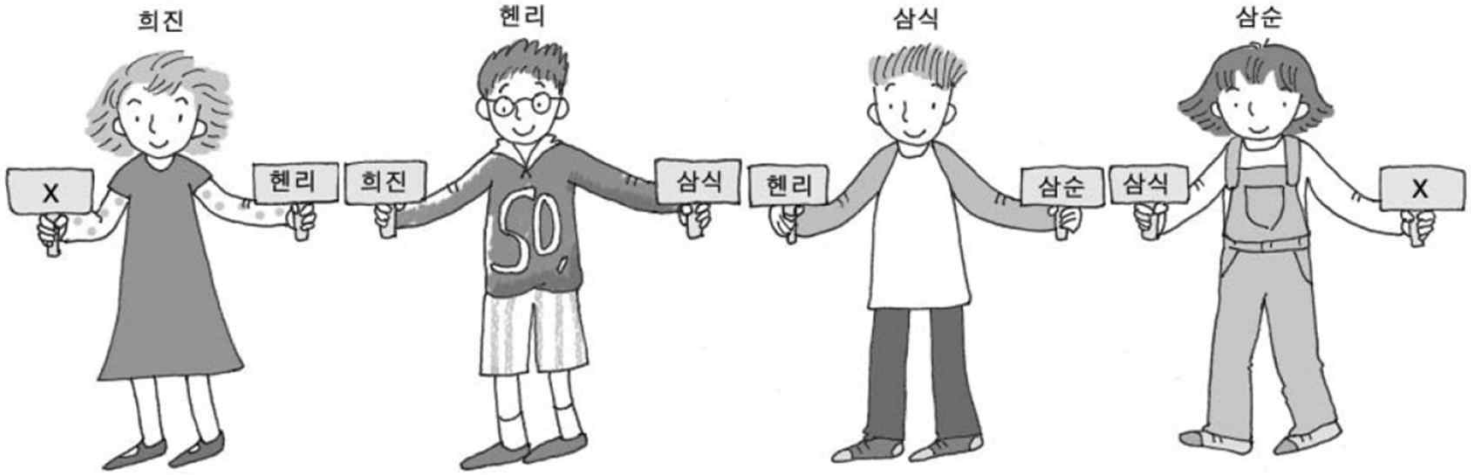


❑ 이중 연결 리스트

- 양방향 기차 만들기 2 : 오른쪽 사람에게 자기 이름 주기



- 양방향 기차 만들기 : 완성



▪ 이중 연결 리스트에서의 삽입 연산 과정

❶ 삽입할 노드를 가져온다.

❷ 새 노드의 데이터 필드에 값을 저장한다.

❸ 새 노드의 왼쪽 노드의 오른쪽 링크(rlink)를 새 노드의 오른쪽 링크(rlink)에 저장한다.

❹ 그리고 왼쪽 노드의 오른쪽 링크(rlink)에 새 노드의 주소를 저장한다.

❺ 새 노드의 오른쪽 노드의 왼쪽 링크(llink)를 새 노드의 왼쪽 링크(llink)에 저장한다.

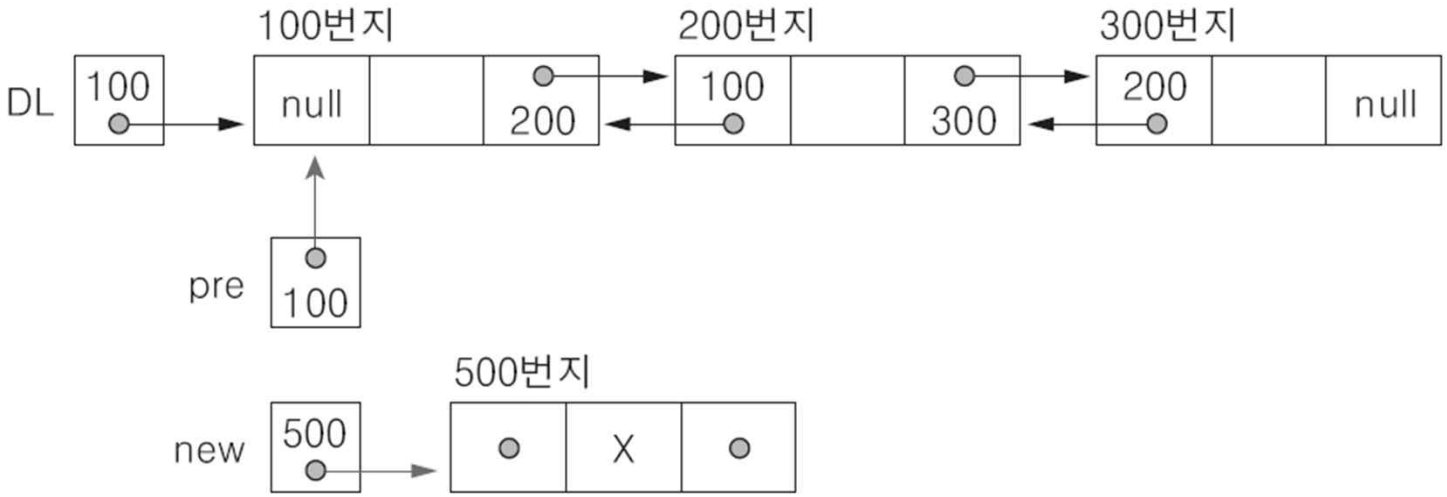
❻ 그리고 오른쪽 노드의 왼쪽 링크(llink)에 새 노드의 주소를 저장한다.

- 이중 연결 리스트에서의 삽입 알고리즘

[알고리즘 6-11]

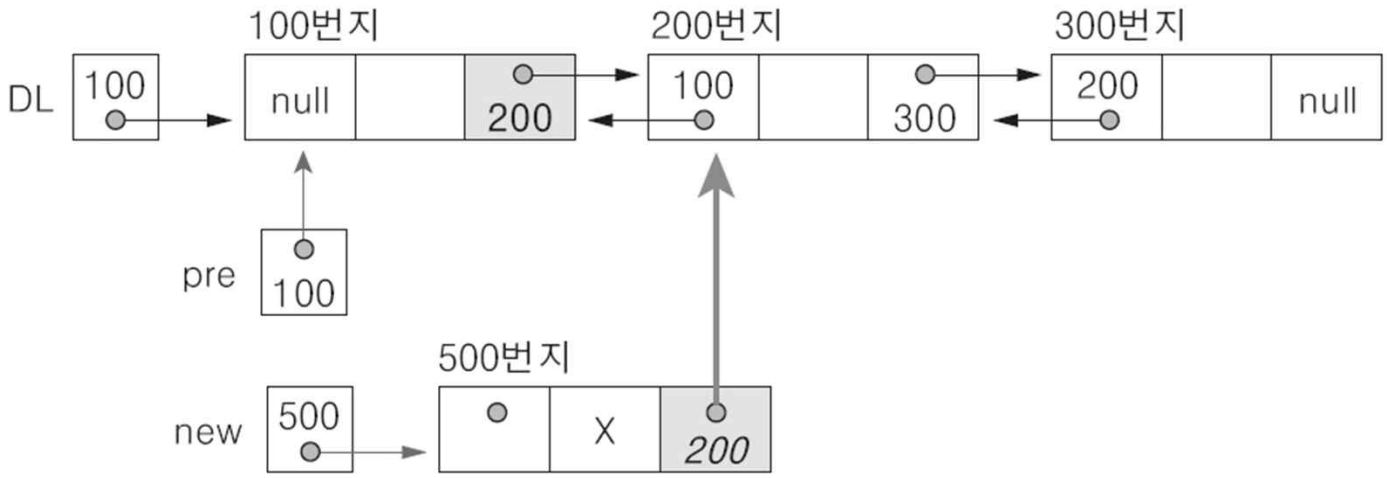
```
insertNode(DL, pre, x)
    new ← getNode();
    new.data ← x;
    new.rlink ← pre.rlink ;    // ❶
    pre.rlink ← new;           // ❷
    new.llink ← pre;           // ❸
    new.rlink.llink ← new;     // ❹
end insertNode()
```

- **이중 연결 리스트 DL에서 포인터 pre가 가리키는 노드의 다음노드로 노드 new를 삽입하는 과정**
  - 초기 상태



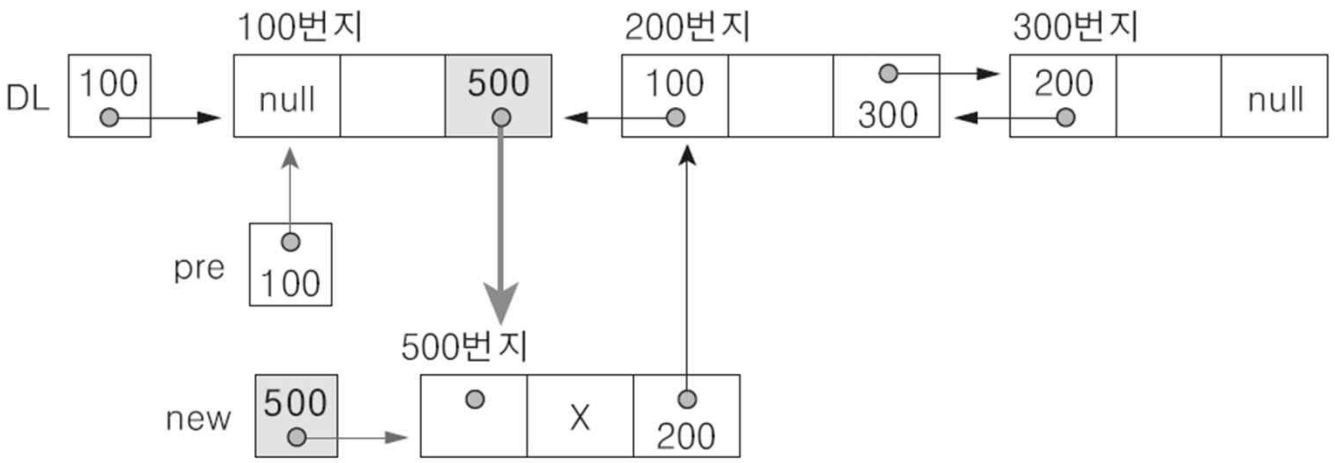
① **new.rlink ← pre.rlink ;**

노드 pre의 rlink를 노드 new의 rlink에 저장하여, 노드 pre의 오른쪽 노드를 삽입할 노드 new의 오른쪽 노드로 연결



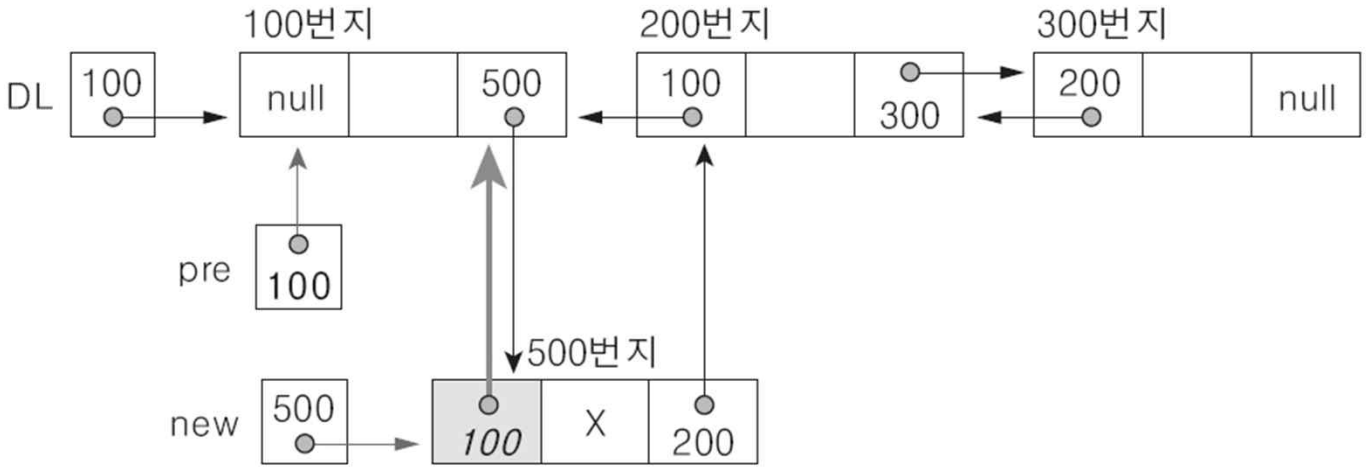
② **pre.rlink ← new;**

새 노드 new의 주소를 노드 pre의 rlink에 저장하여, 노드 new를 노드 pre의 오른쪽 노드로 연결



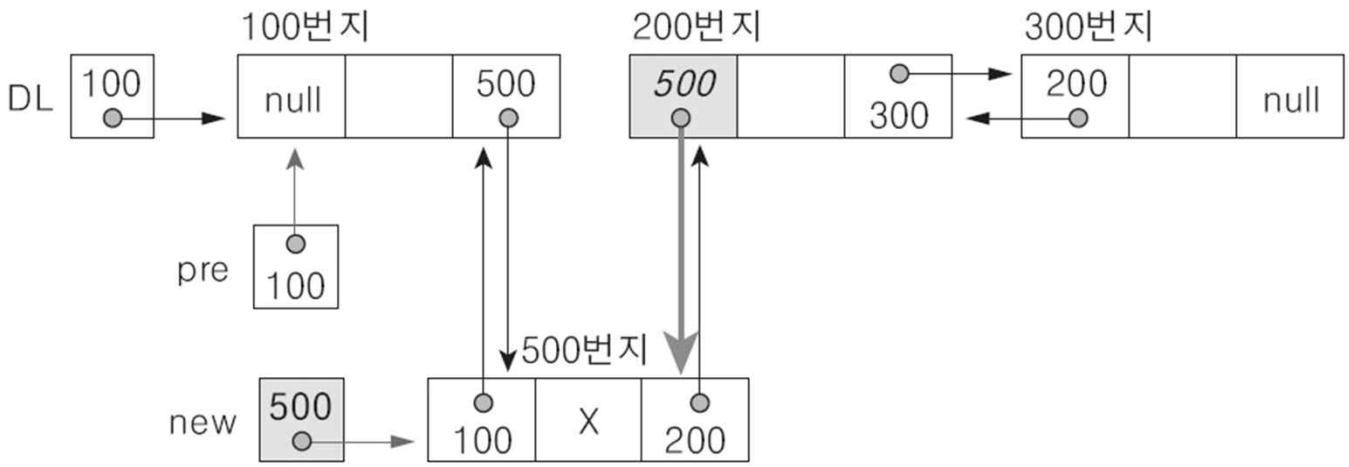
③ **new.llink ← pre;**

포인터 pre의 값을 삽입할 노드 new의 llink에 저장하여, 노드 pre를 노드 new의 왼쪽노드로 연결

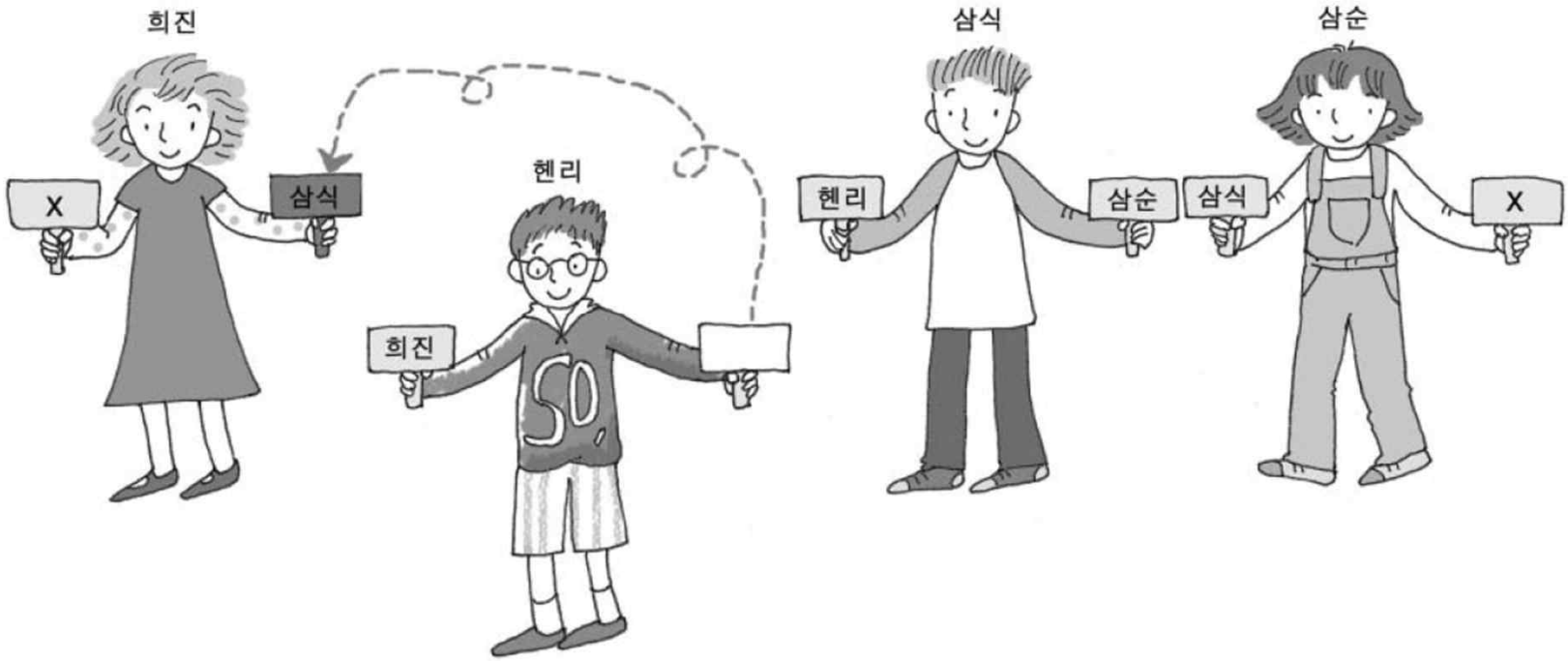


④ **new.rlink.llink ← new;**

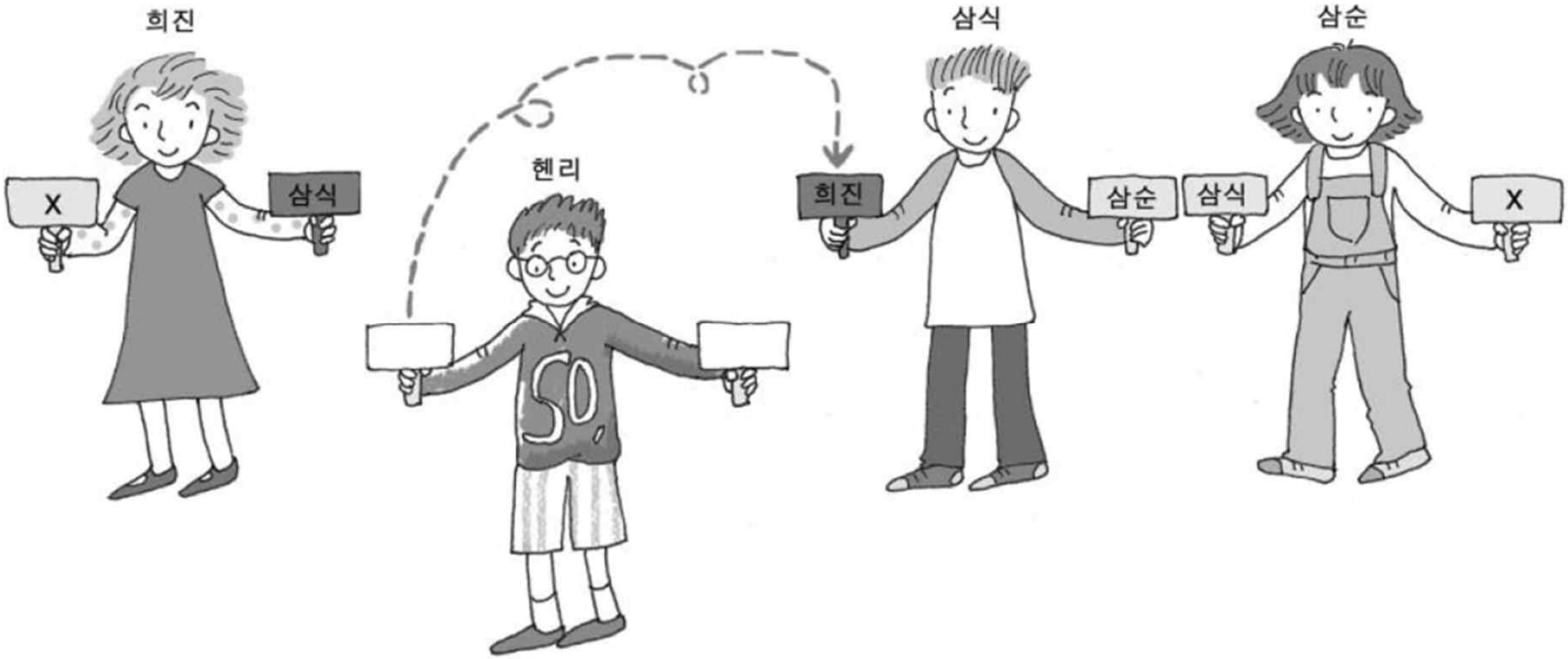
포인터 new의 값을 노드 new의 오른쪽노드(new.rlink)의 llink에 저장하여, 노드 new의 오른쪽노드의 왼쪽노드로 노드 new를 연결



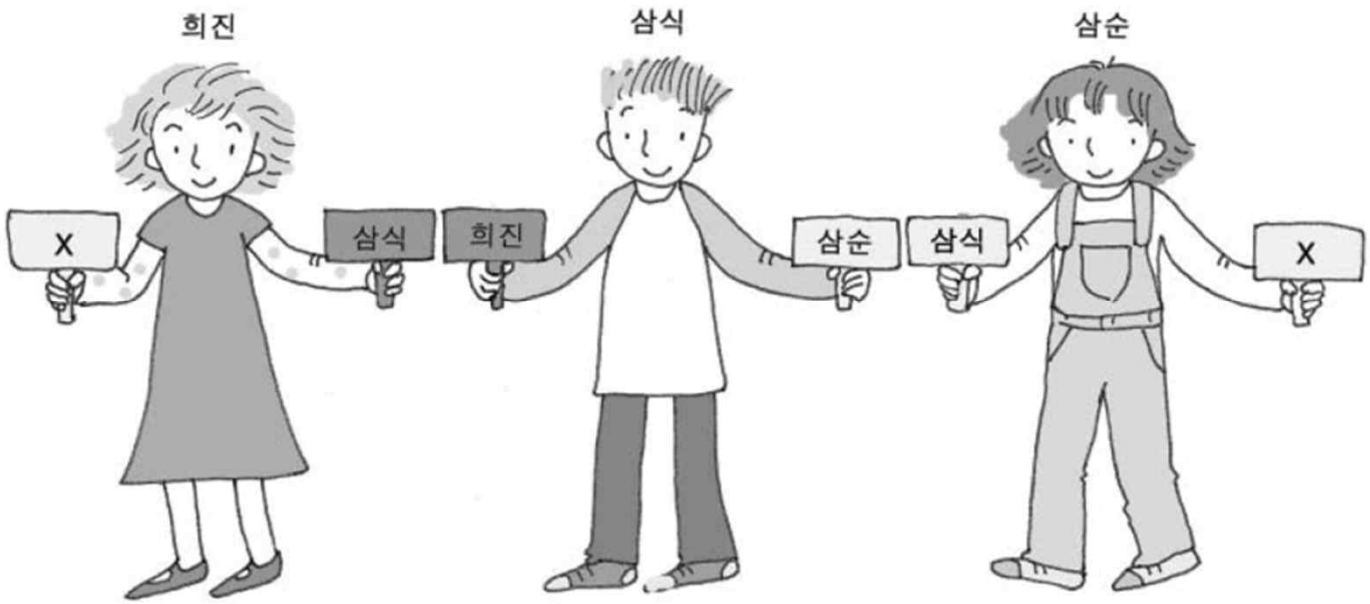
- 삭제 : 오른손 이름표 넘겨주기



- 삭제 : 왼손 이름표 넘겨주기



- 삭제 완료



▪ 이중 연결 리스트에서의 삽입 연산 과정

❶ 삭제할 노드의 오른쪽노드의 주소(old.rlink)를 삭제할 노드의 왼쪽노드(old.llink)의 오른쪽 링크(rlink)에 저장한다.

❷ 삭제할 노드의 왼쪽노드의 주소(old.llink)를 삭제할 노드의 오른쪽노드(old.rlink)의 왼쪽 링크(llink)에 저장한다.

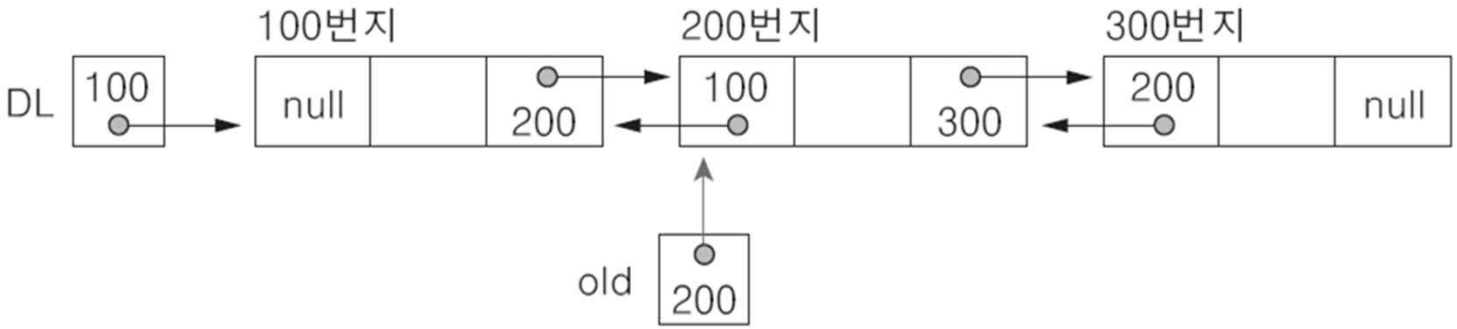
❸ 삭제한 노드를 자유공간리스트에 반환한다.

▪ 이중 연결 리스트에서의 삭제 알고리즘

[알고리즘 6-12]

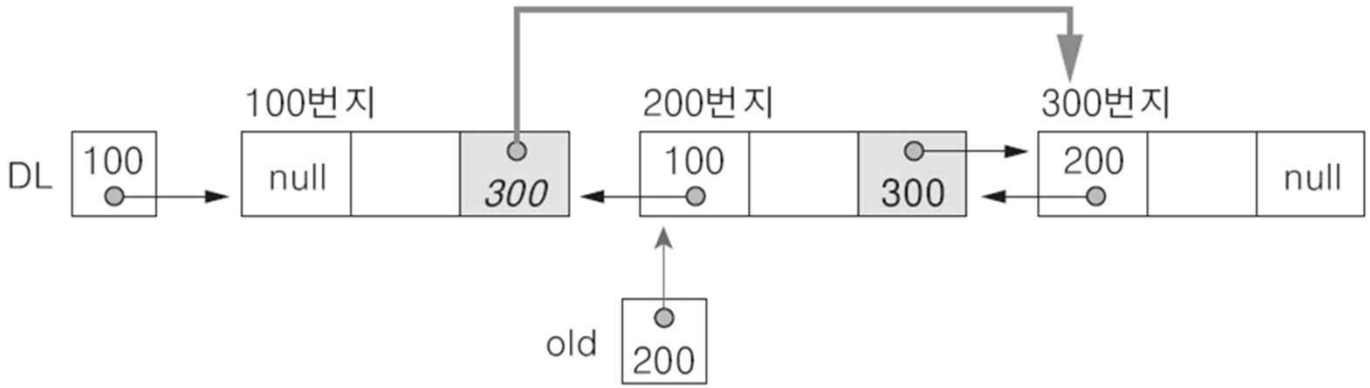
```
deleteNode(DL, old)
    old.llink.rlink ← old.rlink; // ❶
    old.rlink.llink ← old.llink; // ❷
    returnNode(old);             // ❸
end deleteNode( )
```

- **이중 연결 리스트 DL에서 포인터 old가 가리키는 노드를 삭제하는 과정**
  - 초기 상태



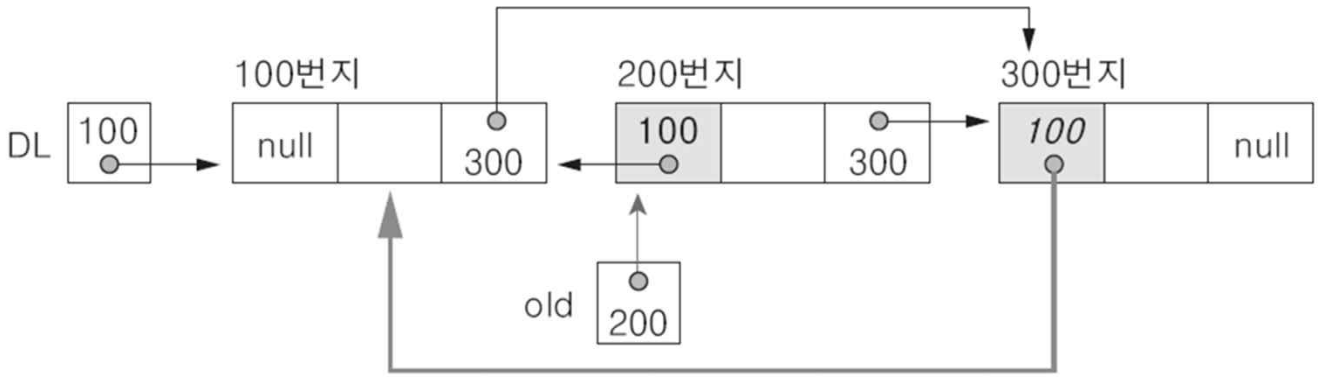
① old.llink.rlink ← old.rlink;

삭제할 노드 old의 오른쪽노드의 주소를 노드 old의 왼쪽노드의 rlink에 저장하여, 노드 old의 오른쪽노드를 노드 old의 왼쪽노드의 오른쪽노드로 연결



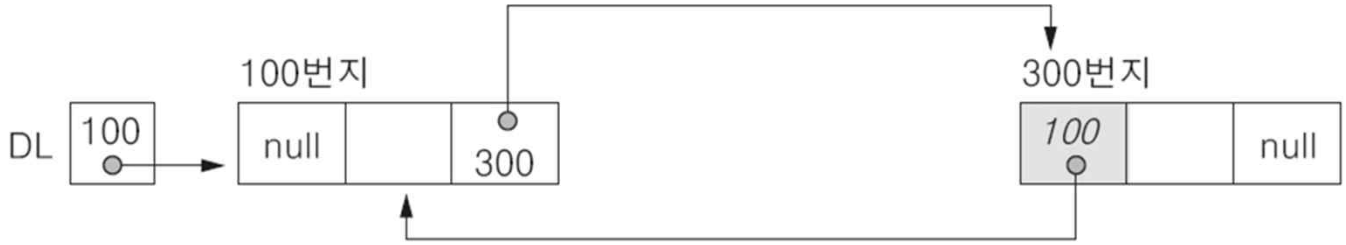
② **old.rlink.llink ← old.llink;**

삭제할 노드 old의 왼쪽노드의 주소를 노드 old의 오른쪽노드의 llink에 저장하여, 노드 old의 왼쪽노드를 노드 old의 오른쪽노드의 왼쪽노드로 연결



③ **returnNode(old);**

삭제된 노드 old는 자유공간리스트에 반환



❑ 다항식의 연결 자료구조 표현

❖ 단순 연결 리스트를 이용하여 다항식 표현

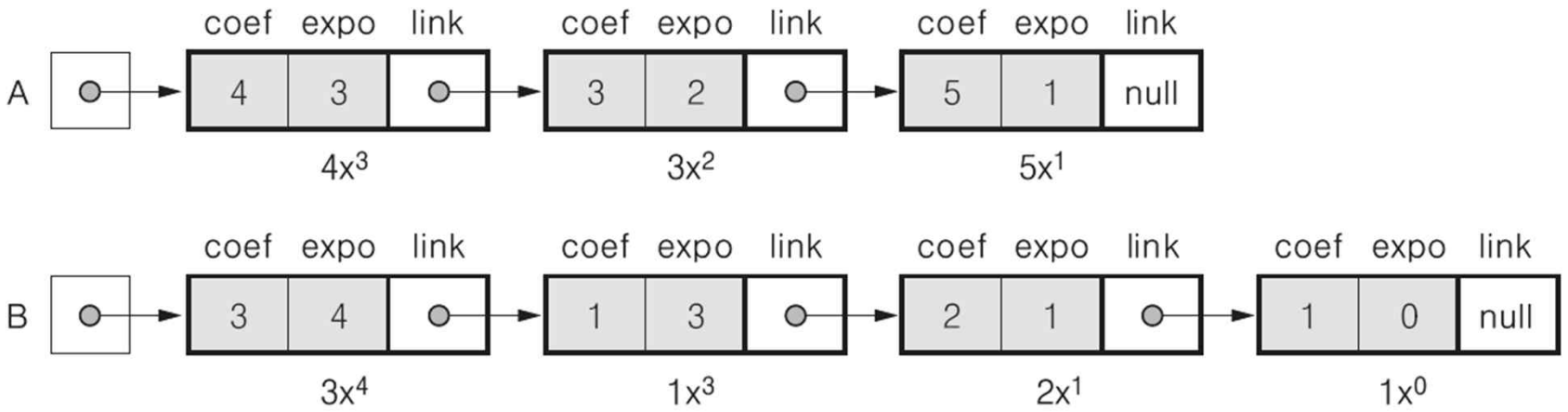
- **다항식의 항 : 단순 연결 리스트의 노드**
- **노드 구조**
  - 각 항에 대해서 계수와 지수를 저장
  - 계수를 저장하는 coef와 지수를 저장하는 expo의 두 개의 필드로 구성
  - 링크 필드 : 다음 항을 연결하는 포인터로 구성

  - 노드에 대한 구조체 정의

```
public class Node {
        float coef;
        int expo;
        Node link;
};
```

❑ 다항식의 연결 자료구조 표현

- **다항식의 단순 연결 리스트 표현 예**
  - 다항식 $A(x)=4x^3+3x^2+5x$
  - 다항식 $B(x)=3x^4+x^3+2x+1$



❖ 다항식 연결 자료구조의 삽입 연산

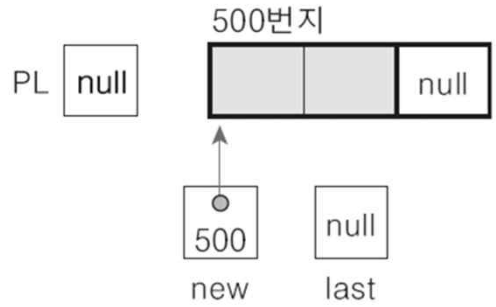
▪ 다항식에 항을 추가하는 알고리즘

- 다항식 리스트 포인터 PL, coef 필드 값을 저장한 변수 coef, expo 필드 값을 저장한 변수 expo
  리스트 PL의 마지막 노드의 위치를 지시하는 포인터 last 사용

[알고리즘 6-13]

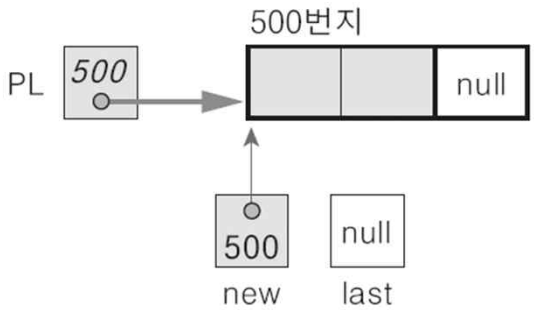
```
appendTerm(PL, coef, expo, last)
    new ← getNode( );
    new.expo ← expo;
    new.coef ← coef;
    new.link ← null;
    if (PL = null) then { // ❶
        PL ← new; // ❶-ⓐ
        last ← new; // ❶-ⓑ
    }
    else { // ❷
        last.link ← new; // ❷-ⓐ
        last ← new; // ❷-ⓑ
    }
end appendTerm( )
```

- **공백 다항식에서의 항 삽입 연산**
  - 초기 상태



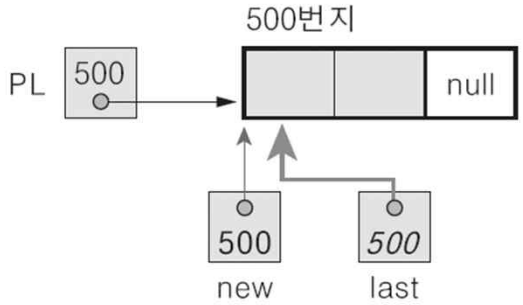
① **PL ← new;**

포인터 new의 값을 리스트 포인터 PL에 저장하여, 노드 new가 리스트 PL의 첫 번째 노드가 되도록 연결



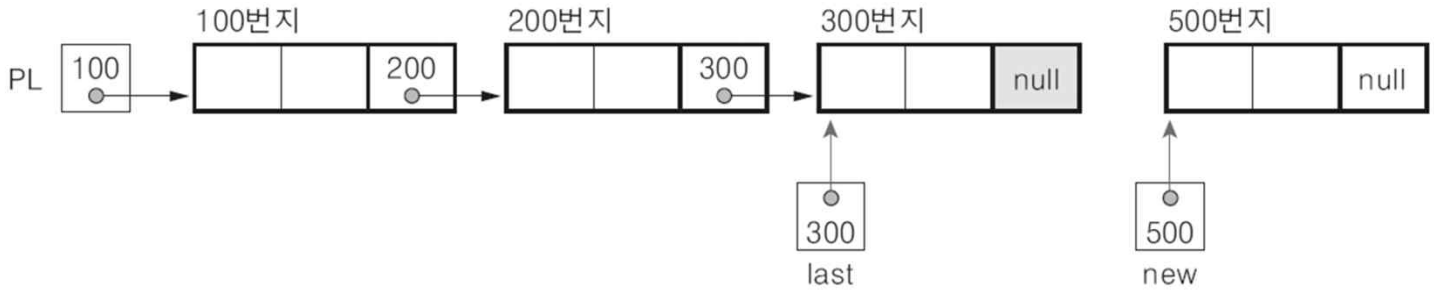
② **last ← new;**

포인터 new의 값을 포인터 last에 저장하여, 포인터 last가 리스트 PL의 마지막 노드인 노드 new를 가리키도록 지정



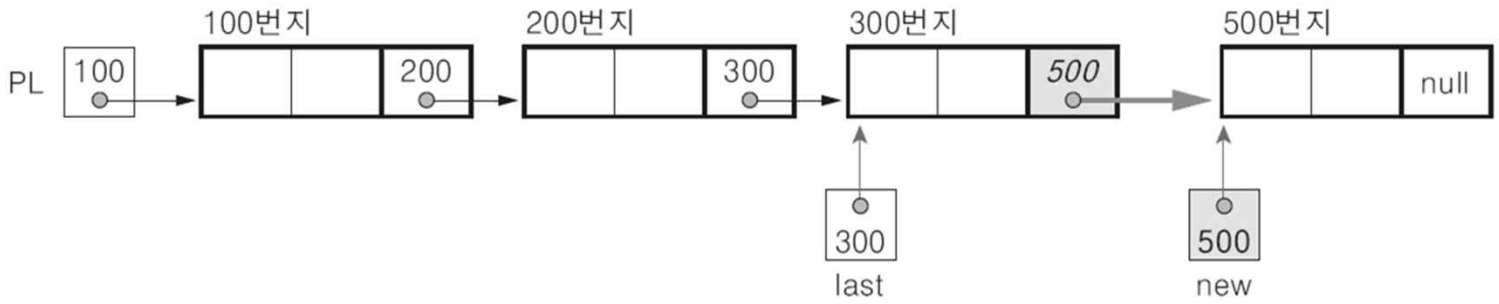
▪ 다항식에서의 항 삽입 연산

- 초기 상태



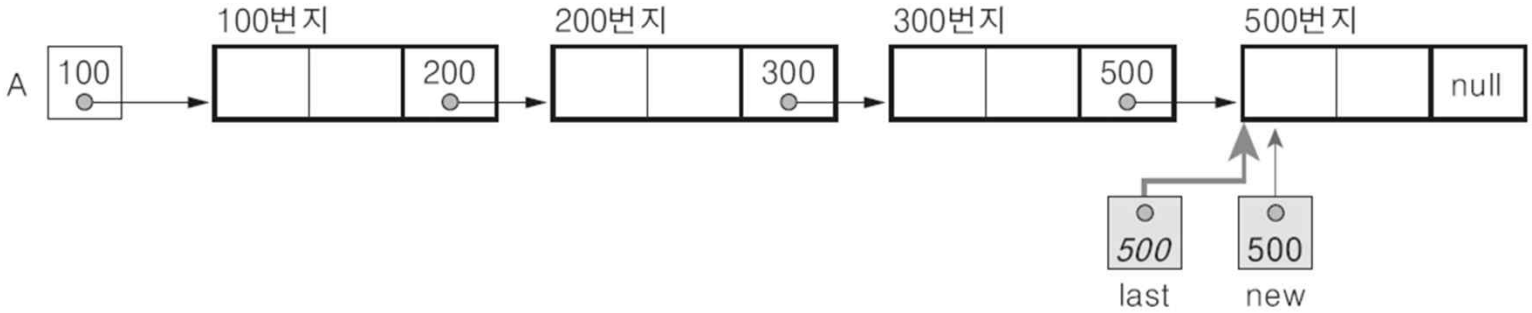
① **last.link ← new;**

포인터 new의 값을 노드 last의 링크에 저장하여, 노드 new를 노드 last의 다음 노드로 연결



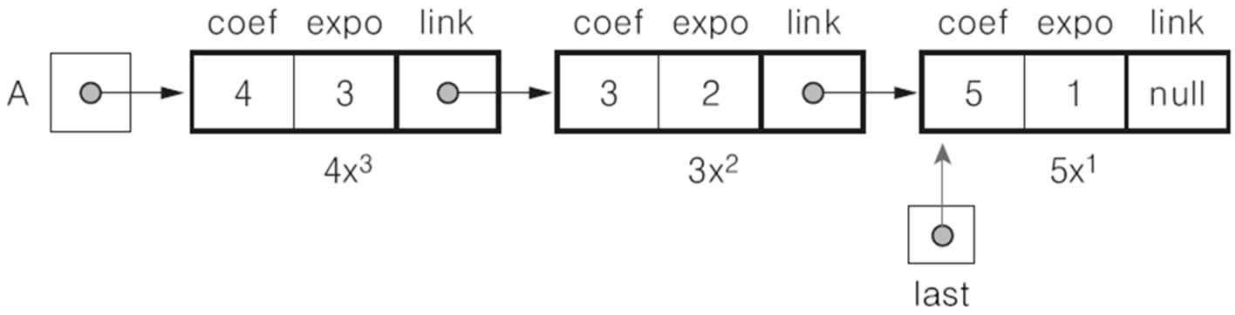
② **last ← new;**

포인터 new의 값을 포인터 last에 저장하여, 노드 new를 리스트 PL의 마지막 노드로 지정

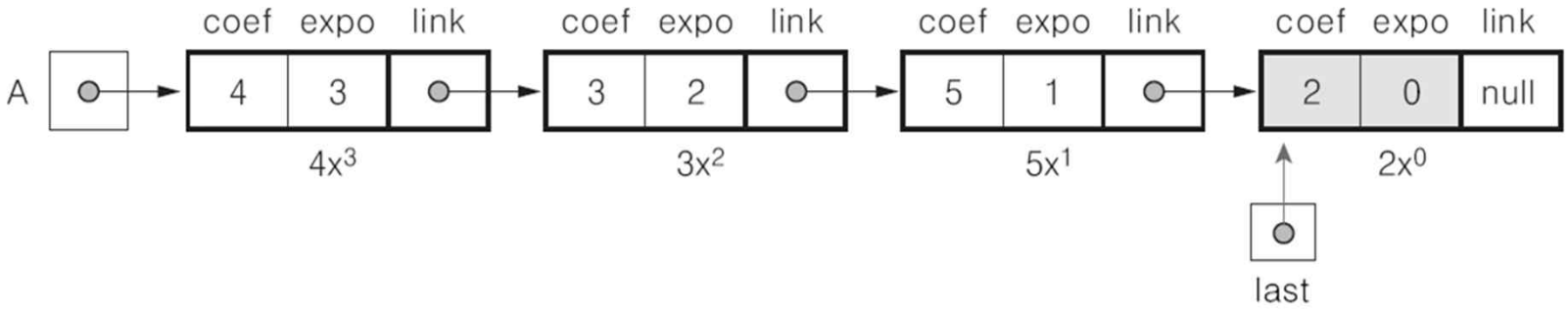


❑ 다항식의 연결 자료구조 표현

- 다항식 리스트 A에 appendTerm() 알고리즘을 사용하여 $2x^0$항 (상수항 2) 추가 예



(a) appendTerm(A,2,0,last) 함수 실행 전의 다항식 리스트 A



(b) appendTerm(A,2,0,last) 함수 실행 후의 다항식 리스트 A

❖ 다항식의 덧셈 연산

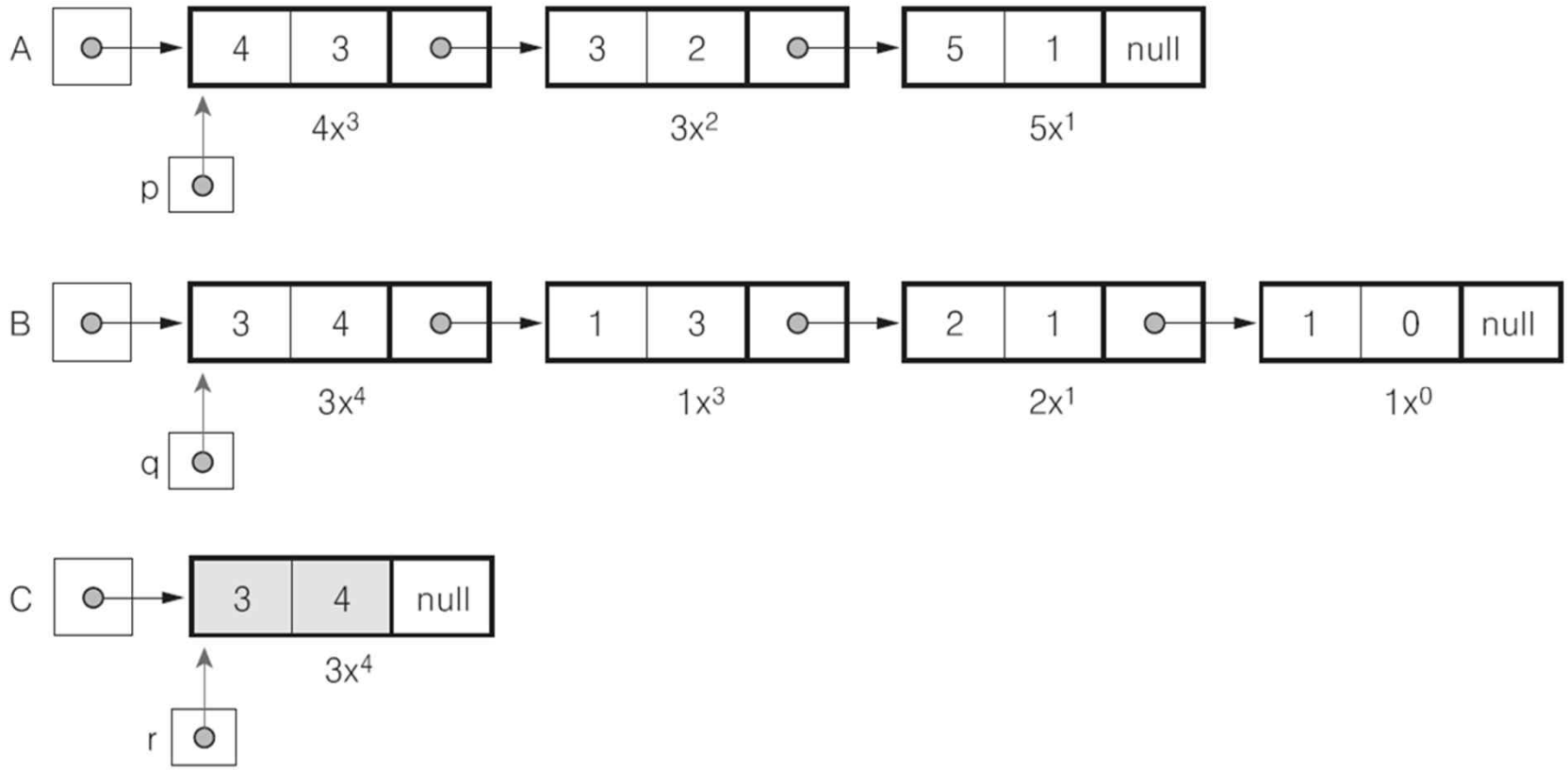
▪ 덧셈 A(x)+B(x)=C(x)를 단순 연결 리스트 자료구조로 연산

- 다항식 A(x)와 B(x), C(x)의 항을 지시하기 위해서 세 개의 포인터를 사용
- 포인터 p : 다항식 A(x)에서 비교할 항을 지시
- 포인터 q : 다항식 B(x)에서 비교할 항을 지시
- 포인터 r : 덧셈연산 결과 만들어지는 다항식 C(x)의 항을 지시

① p.expo < q.expo : 다항식 A(x) 항의 지수가 작은 경우

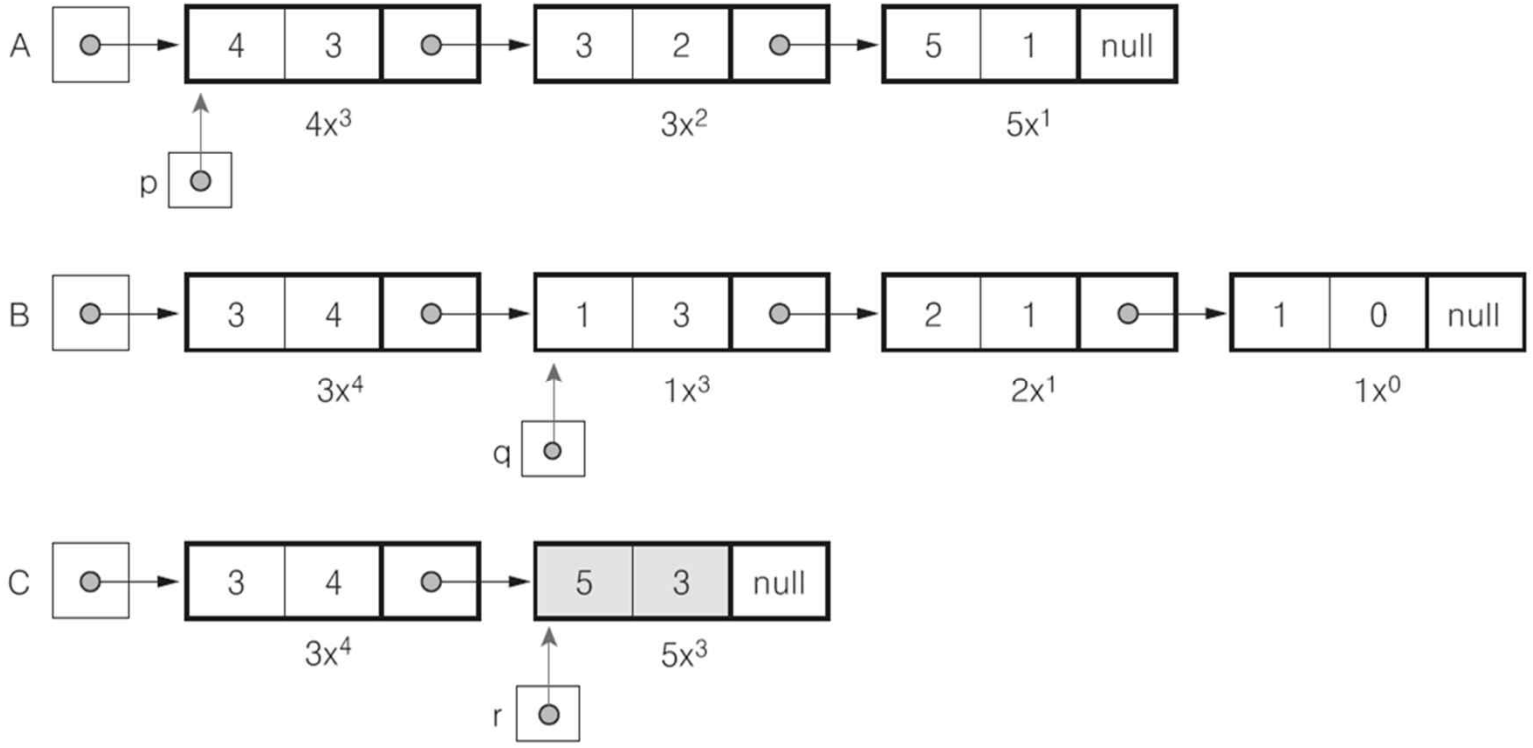
- 포인터 q가 가리키는 다항식 B(x)의 항을 C(x)의 항으로 복사
- q가 가리키는 항에 대한 처리가 끝났으므로 q를 다음 노드로 이동

❑ 다항식의 연결 자료구조 표현



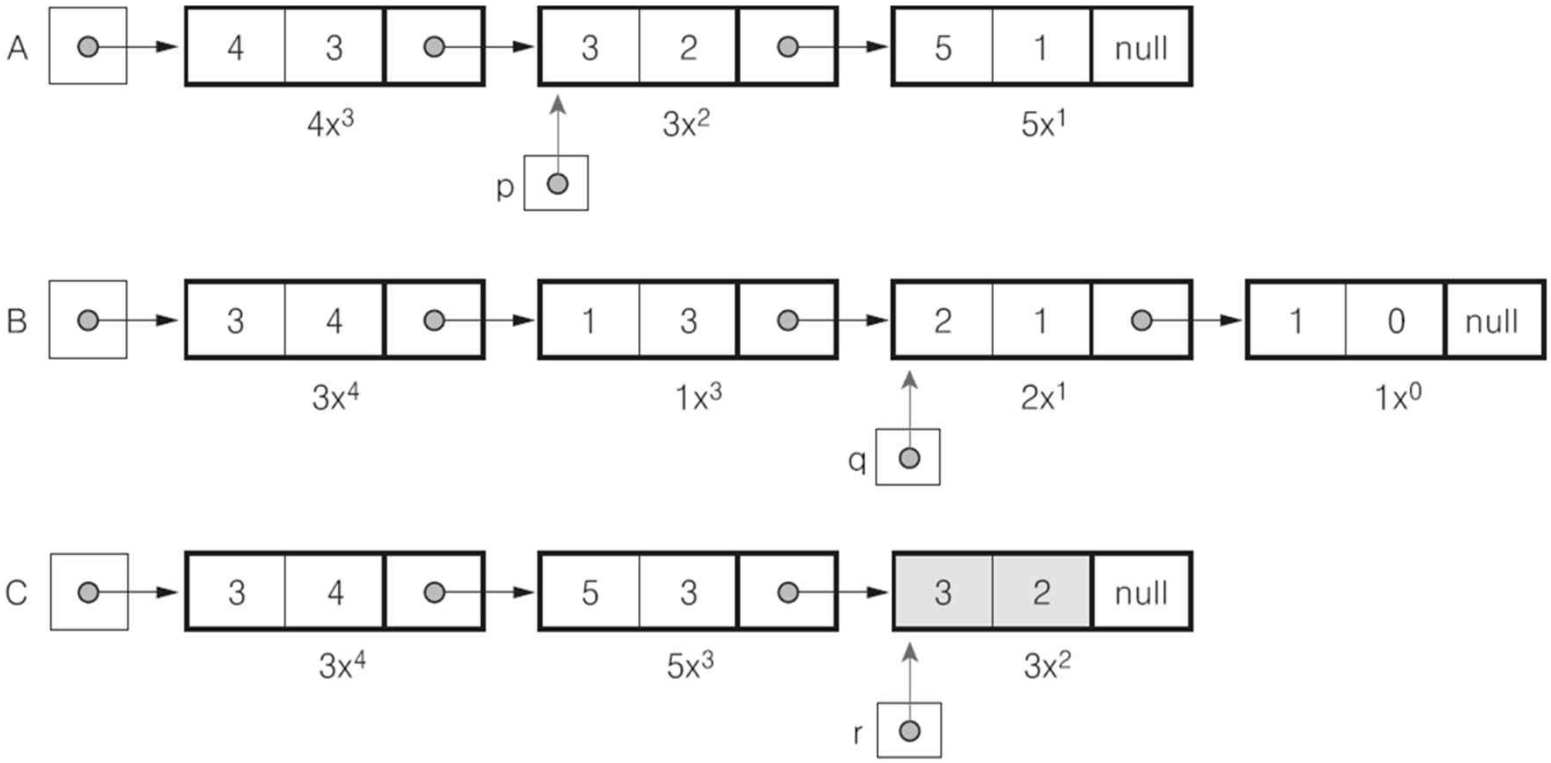
② p.expo = q.expo : 두 항의 지수가 같은 경우

- p.coef와 q.coef를 더하여 C(x)의 항, 즉 r.coef에 저장하고 지수가 같아야 하므로 p.expo(또는 q.expo)를 r.expo에 저장
- 다음 항을 비교하기 위해 포인터 p와 q를 각각 다음 노드로 이동



③ p.expo > q.expo : 다항식 A(x) 항의 지수가 큰 경우

- 포인터 p가 가리키는 다항식 A(x)의 항을 C(x)의 항으로 복사
- p가 가리키는 항에 대한 처리가 끝났으므로 p를 다음 노드로 이동



- **다항식의 덧셈 알고리즘**

[알고리즘 6-14]

```
addPoly(A, B)
// 단순 연결 리스트로 표현된 다항식 A와 B를 더하여 새로운 다항식 C를 반환
    p ← A;
    q ← B;
    C ← null; // 결과 다항식
    r ← null; // 결과 다항식의 마지막 노드를 지시
    while (p ≠ null and q ≠ null) do { // p, q는 순회용 참조변수
        case {
            p.expo = q.expo :
                sum ← p.coef + q.coef
                if (sum ≠ 0) then appendTerm(C, sum, p.expo, r);
                p ← p.link;
                q ← q.link;
            p.expo < q.expo :
                appendTerm(C, q.coef, q.expo, r);
                q ← q.link;
```

- 다항식의 덧셈 알고리즘

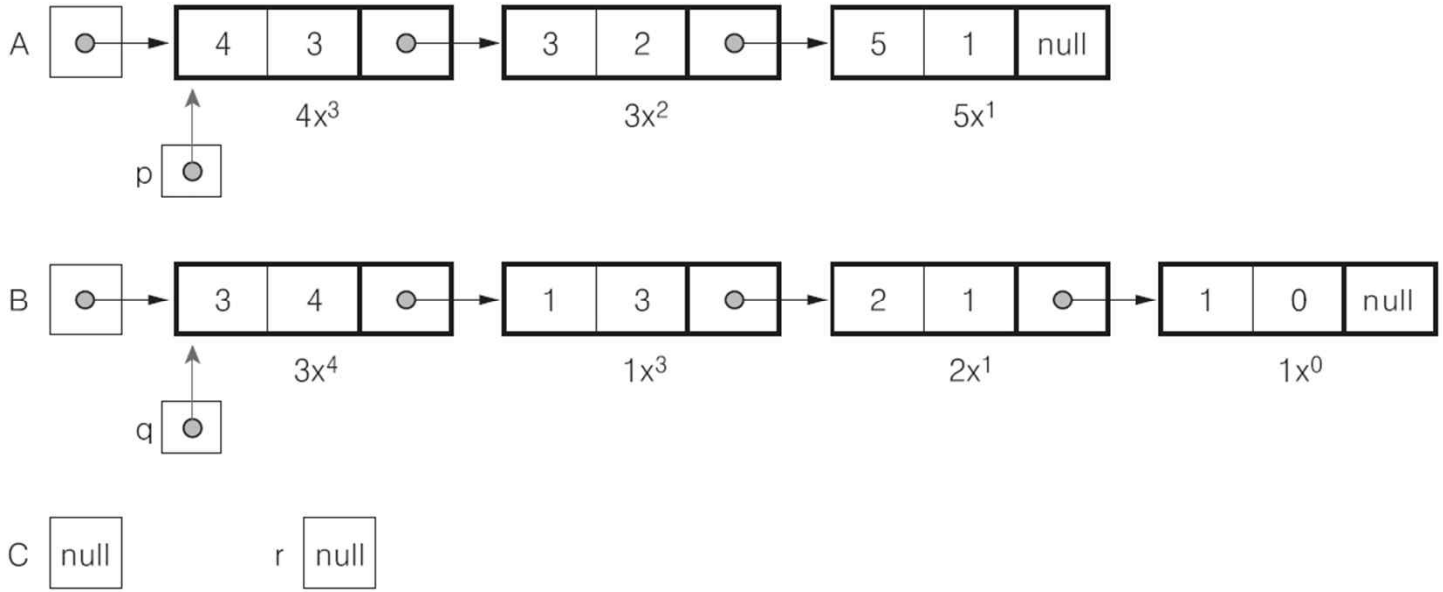
[알고리즘 6-14]

```
            else : // p.expo > q.expo인 경우
                appendTerm(C, p.coef, p.expo, r);
                p ← p.link;
            } // end case
        } // end while

        while (p ≠ null) do { // A의 나머지 항들을 C에 복사
            appendTerm(C, p.coef, p.expo, r);
            p ← p.link;
        }
        while (q ≠ null) do { // B의 나머지 항들을 C에 복사
            appendTerm(C, q.coef, q.expo, r);
            q ← q.link;
        }
    return C;
end addPoly()
```

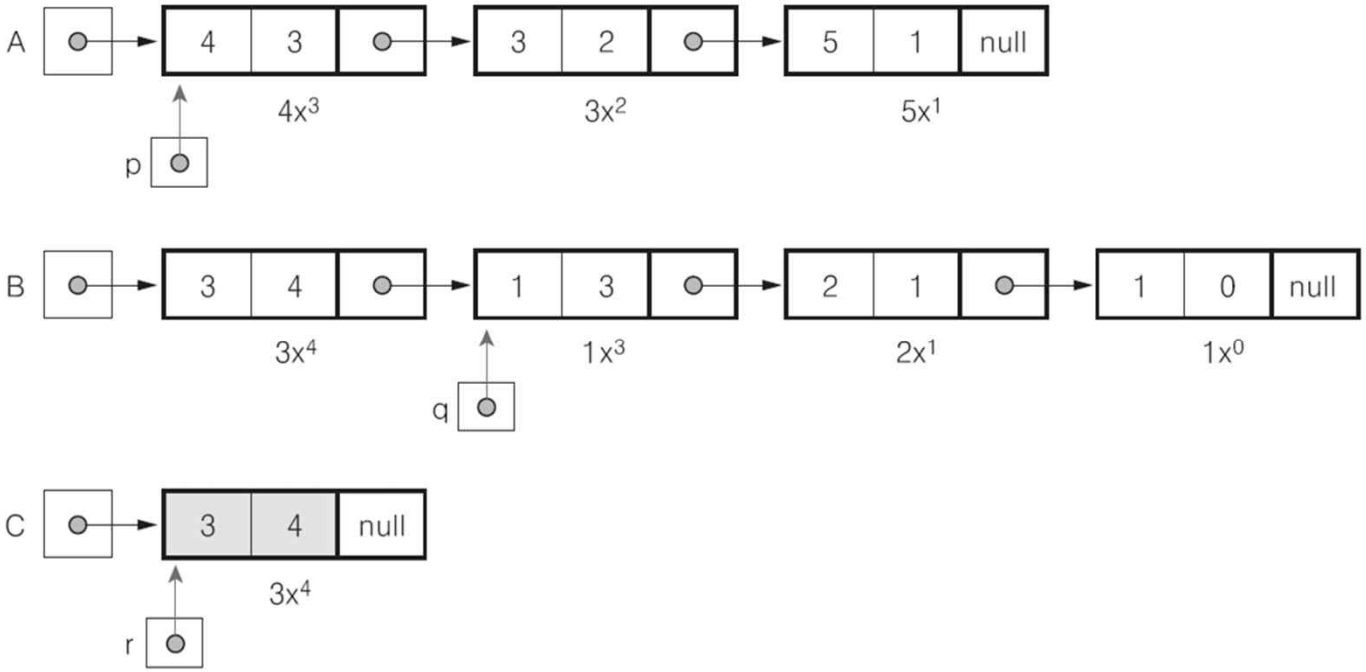
❑ 다항식의 연결 자료구조 표현

- **다항식의 덧셈 예**
  - $A(x)=4x^3+3x^2+5x$
  - $B(x)=3x^4+x^3+2x+1$
  - 초기 상태



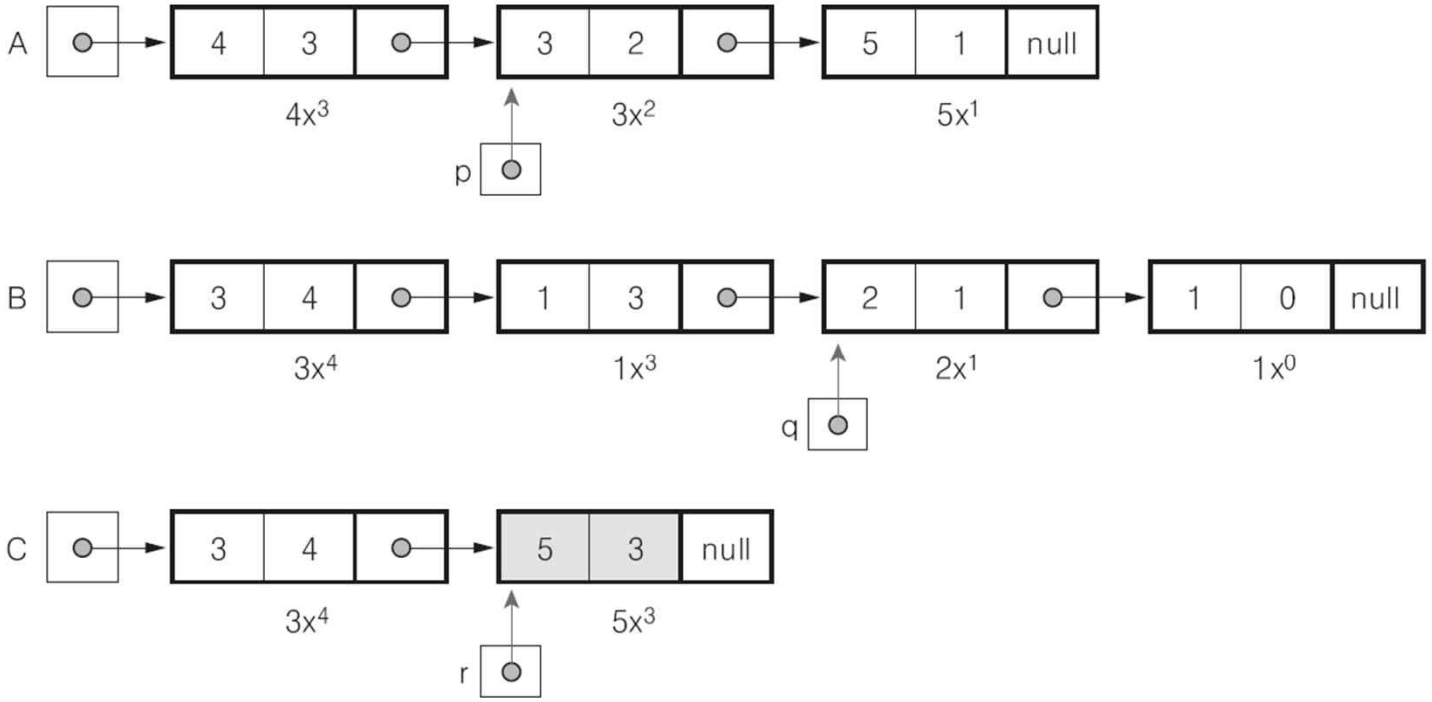
① $4x^3$항과 $3x^2$항에 대한 처리

- p.expo < q.expo이므로 지수가 큰 q 노드를 리스트 C에 복사
- 포인터 q는 다음 노드로 이동



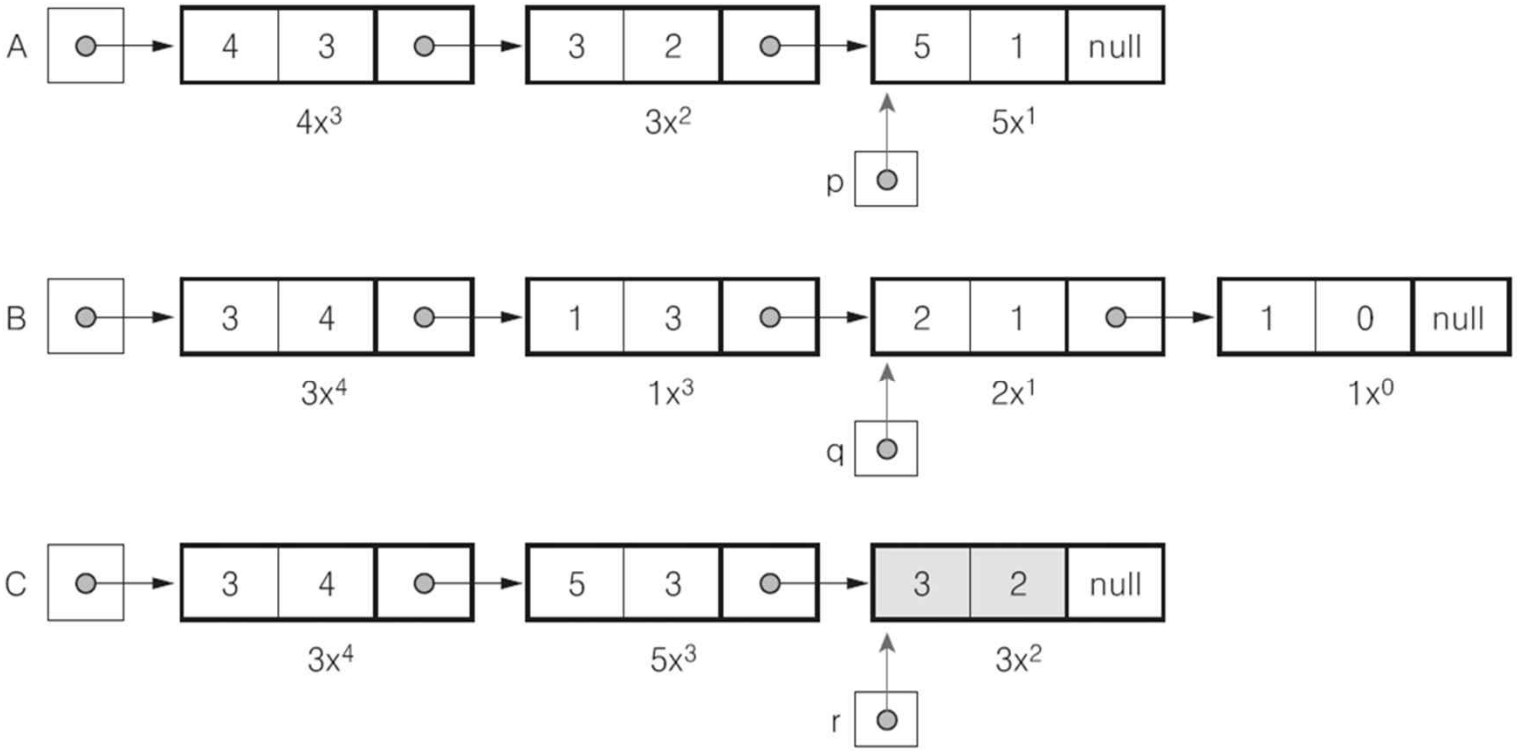
② $4x^3$항과 $1x^3$ 항에 대한 처리

- p.expo = q.expo 이므로 두 노드의 coef 필드의 값을 더하고, expo 필드의 값을 복사한 노드를 리스트 C에 추가
- 포인터 p와 q는 다음 노드로 이동



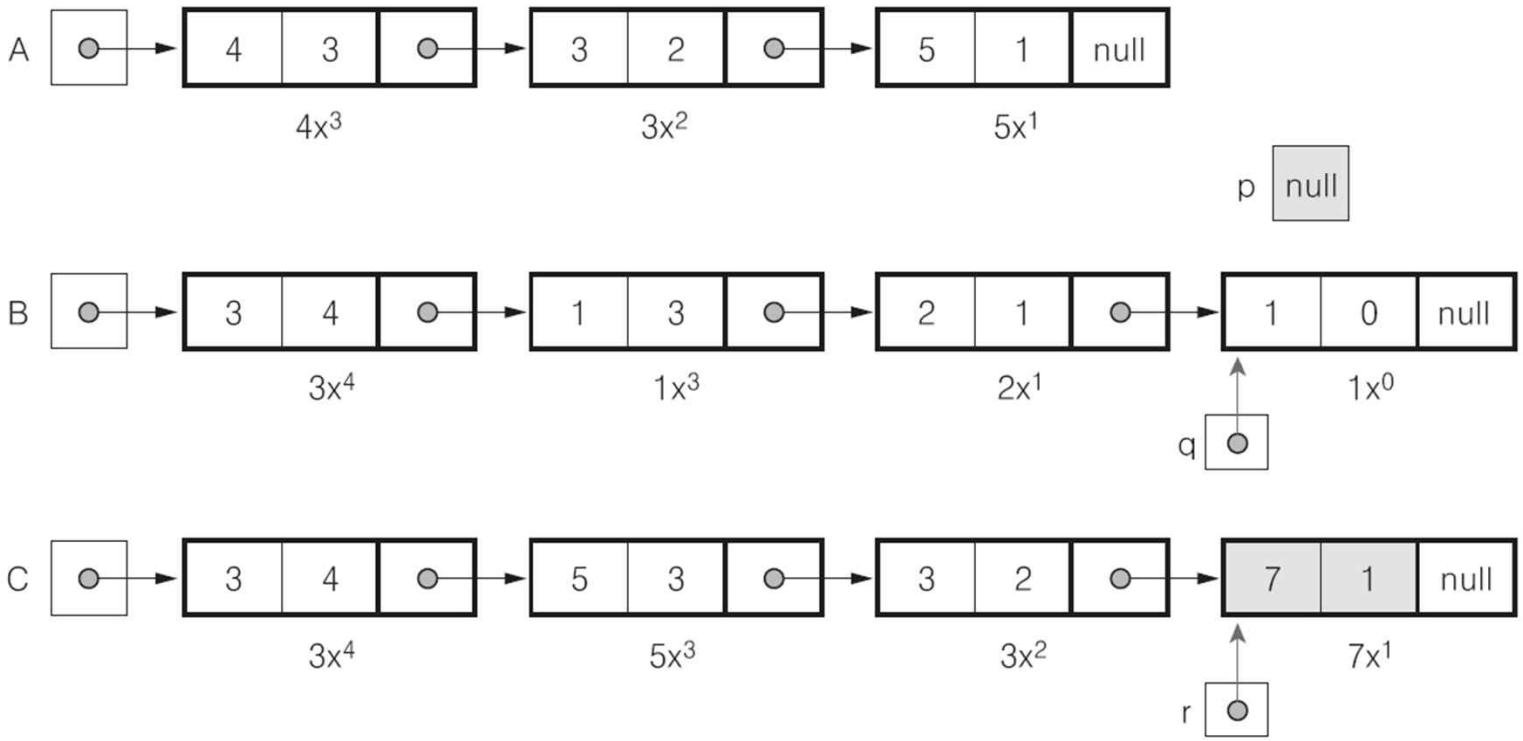
③ $3x^2$항과 $2x^1$ 항에 대한 처리

- p.expo > q.expo 이므로 지수가 큰 p 노드를 리스트 C에 복사
- 포인터 p는 다음 노드로 이동



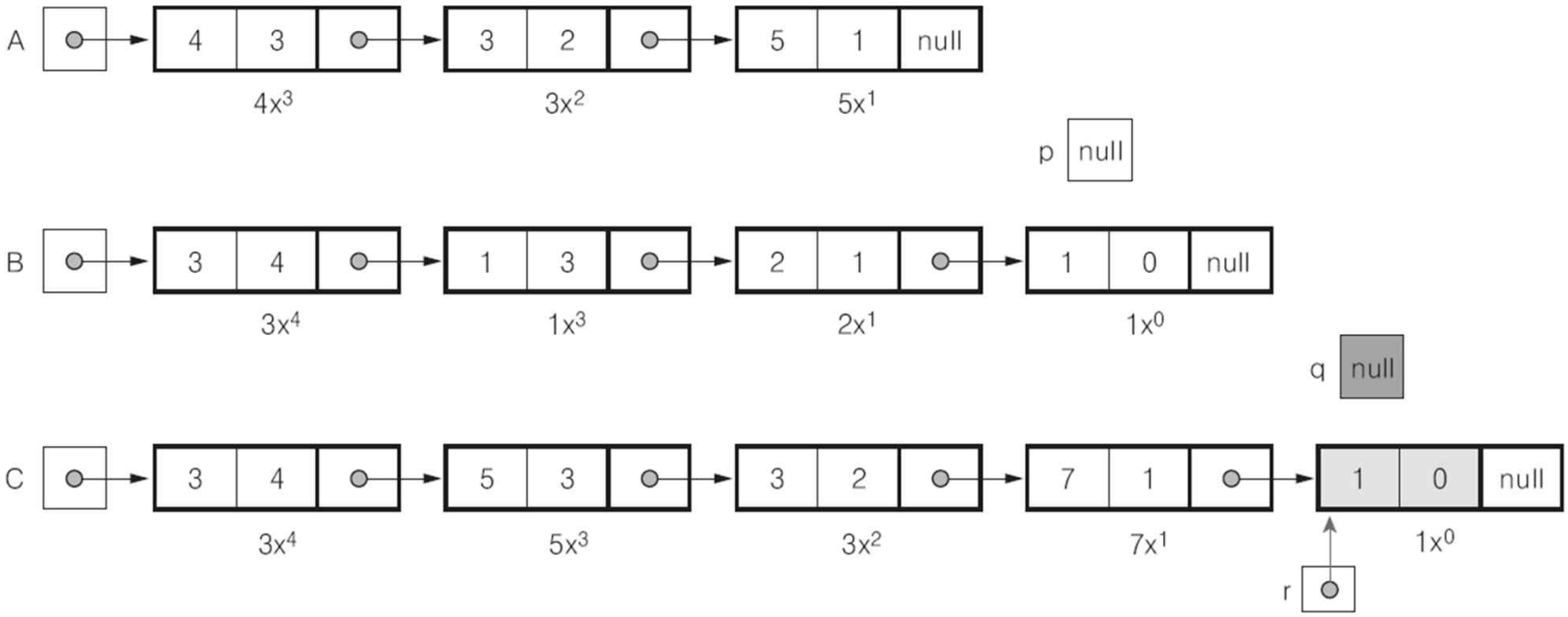
④ $5x^1$항과 $2x^1$ 항에 대한 처리

- p.expo = q.expo 이므로 두 노드의 coef 필드의 값을 더하고, expo 필드의 값을 복사한 노드를 리스트 C에 추가
- 포인터 p와 q는 다음 노드로 이동



⑤ B(x)의 남은 항에 대한 처리

- 포인터 p가 null이므로 다항식 A(x)의 항에 대한 처리 끝
- 처리할 항이 남아있는 다항식 B(x)의 포인터 q는 null이 될 때까지 모든 노드를 복사하여 리스트 C에 추가



Thank You !

IT CookBook 자바로 배우는 쉬운 자료구조 6장 끝