연결자료구조

1. 연결 자료구조와 연결 리스트의 이해

순차 자료구조의 문제점

- 삽입 연산이나 삭제 연산 후에 연속적인 물리 주소를 유지하기 위해서 원소들을 이동시키는 추가 작업과 시간 소요
  - 원소들의 이동 작업으로 인한 오버헤드로 원소의 개수가 많고 삽입・삭제 연산이 많이 발생하는 경우에 성능상의 문제 발생
- 순차 자료구조는 배열을 이용해 구현하기 때문에 배열이 갖고 있는 메모리 사용의 비효율성 문제(한 덩어리로 할당)를 그대로 가짐
- 순차 자료구조에서의 연산 시간에 대한 문제와 저장 공간에 대한 문제를 개선한 자료 표현 방법 필요

1. 연결 자료구조와 연결 리스트의 이해

연결 자료구조Linked Data Structure

- 자료의 논리적인 순서와 물리적인 순서가 불일치
  - 각 원소에 저장되어 있는 다음 원소의 주소에 의해 순서가 연결되는 방식
    - **물리적인 순서를 맞추기 위한 오버헤드가 발생하지 않음**
  - 여러 개의 작은 공간을 연결하여 하나의 전체 자료구조를 표현
    - **크기 변경이 유연하고 더 효율적으로 메모리를 사용**
- 연결 리스트
  - 리스트를 연결 자료구조로 표현
  - 연결하는 방식에 따라 단순 연결 리스트와 원형 연결 리스트, 이중 연결 리스트, 이중 원형 연결 리스트로 구분

1. 연결 자료구조와 연결 리스트의 이해

연결 리스트의 노드

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

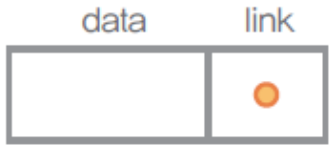


그림 4-2 노드의 논리적 구조

- 데이터 필드data field
  - 원소의 값을 저장
  - 저장할 원소의 형태에 따라서 하나 이상의 필드로 구성
- 링크 필드link field
  - 다음 노드의 주소를 저장
  - 포인터 변수를 사용하여 주소값을 저장

1. 연결 자료구조와 연결 리스트의 이해

- 리스트 week의 노드에 대한 구조체 정의

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

그림 4-9 리스트 week 노드의 구조체

1. 연결 자료구조와 연결 리스트의 이해

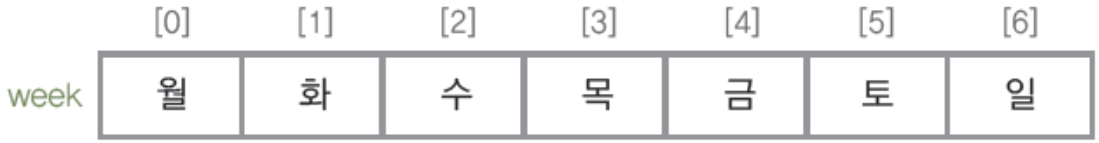
● 순차 자료구조와 연결 자료구조의 비교

표 4-1 순차 자료구조와 연결 자료구조의 비교

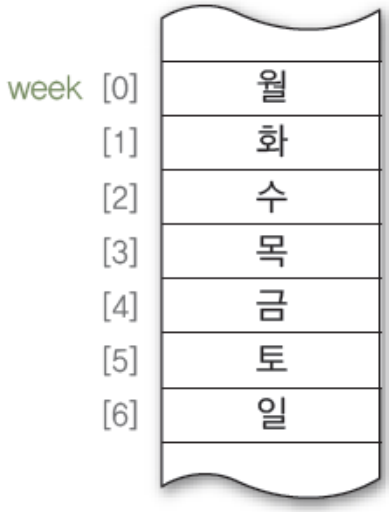
구분	순차 자료구조	연결 자료구조
메모리 저장 방식	필요한 전체 메모리 크기를 계산하여 할당하고, 할당된 메모리의 시작 위치부터 빈자리 없이 자료를 순서대로 연속하여 저장한다.	노드 단위로 메모리가 할당되며, 저장 위치의 순서와 상관없이 노드의 링크 필드에 다음 자료의 주소를 저장한다.
연산 특징	삽입·삭제 연산 후에도 빈자리 없이 자료가 순서대로 연속 저장되어, 변경된 논리적인 순서와 저장된 물리적인 순서가 일치한다.	삽입·삭제 연산 후 논리적인 순서가 변경되어도 링크 정보만 변경되고 물리적 위치는 변경되지 않는다.
프로그램 기법	배열을 이용한 구현	포인터를 이용한 구현

1. 연결 자료구조와 연결 리스트의 이해

- 선형 리스트 week의 순차 리스트 표현
  - 리스트 week=(월, 화, 수, 목, 금, 토, 일)



(a) 논리적 구조



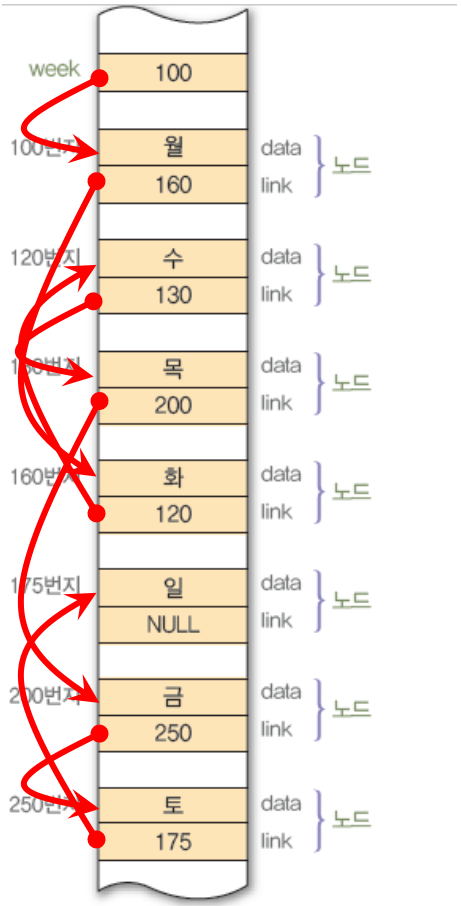
(b) 물리적 구조

그림 4-5 선형 리스트 week의 순차 리스트 표현

1. 연결 자료구조와 연결 리스트의 이해

- 선형 리스트 week의 연결 리스트 표현

(a) 논리적 구조



(b) 물리적 구조

그림 4-6 선형 리스트 week의 연결 리스트 표현

1. 연결 자료구조와 연결 리스트의 이해

- **선형 리스트 week의 연결 리스트 표현**
  - **리스트 이름** week : 연결 리스트의 시작을 가리키는 포인터 변수
    - week는 연결 리스트의 첫 번째 노드를 가리킴과 동시에 연결된 리스트 전체 의미
  - 연결 리스트의 마지막 노드의 링크 필드 : 노드 끝을 표시하기 위해 **NULL**(널) 저장
  - **공백 연결 리스트** : 포인터 변수 week에 NULL 저장(널 포인터)

그림 4-7 공백 연결 리스트

  - 각 노드의 필드에 저장한 값은 **점 연산자**를 사용해 액세스
    - week.data : 포인터 week가 가리키는 노드 데이터 필드 값 "월"
    - week.link : 포인터 week가 가리키는 노드 링크 필드에 저장된 주소값 "120"

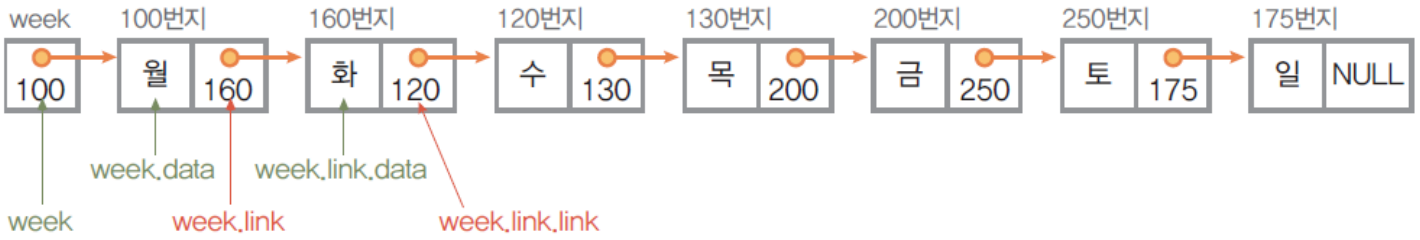


그림 4-8 선형 리스트 week의 연결 리스트 구조

2. 단순 연결 리스트 : 삽입

❶ 삽입할 노드를 준비한다.

❷ 새 노드의 데이터 필드에 값을 저장한다.

❸ 새 노드의 링크값을 지정한다.

❹ 리스트의 앞 노드에 새 노드를 연결한다.

그림 4-11 단순 연결 리스트에 노드를 삽입하는 방법

2. 단순 연결 리스트 : 삽입

- 단순 연결 리스트 week2=(월, 금, 일), '월'과 '금' 사이에 '수' 삽입 과정
  - 초기상태

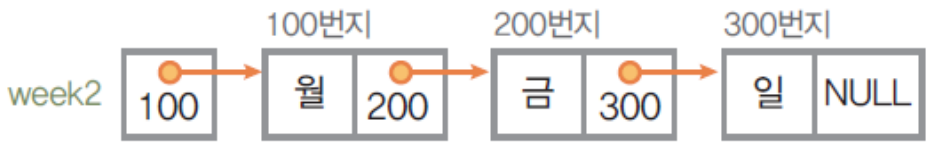
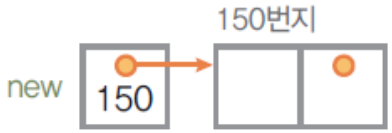
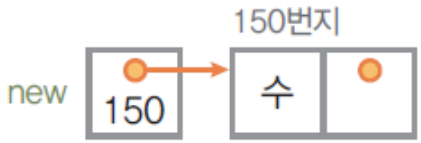


그림 4-12 단순 연결 리스트 week2에 노드를 삽입하기 전인 초기 상태

❶ **삽입할 노드 준비** : 공백 노드를 가져와 포인터 변수 new가 가리키게 함

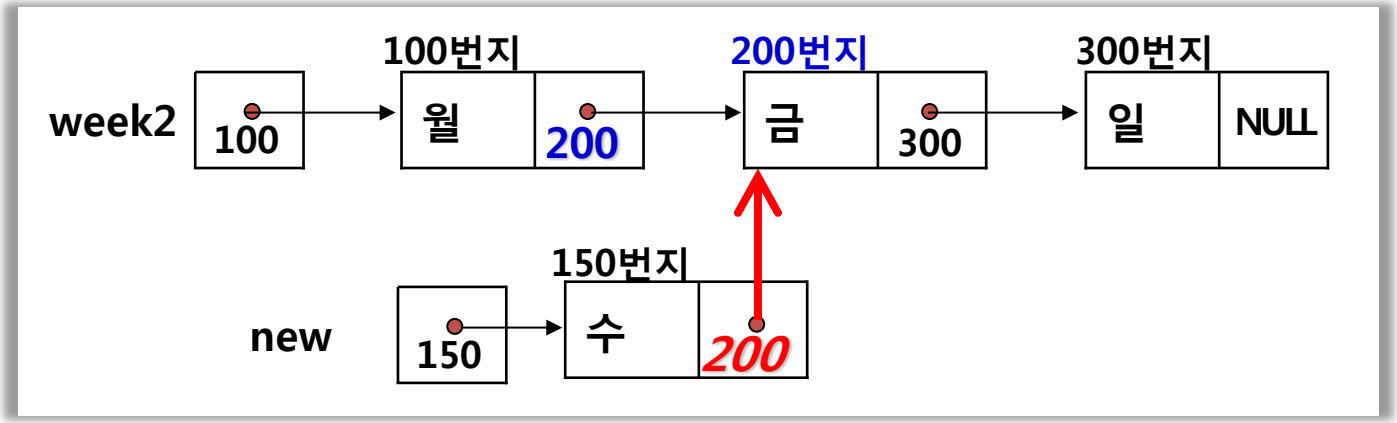


❷ **새 노드의 데이터 필드값 저장** : new의 데이터 필드에 "수"를 저장

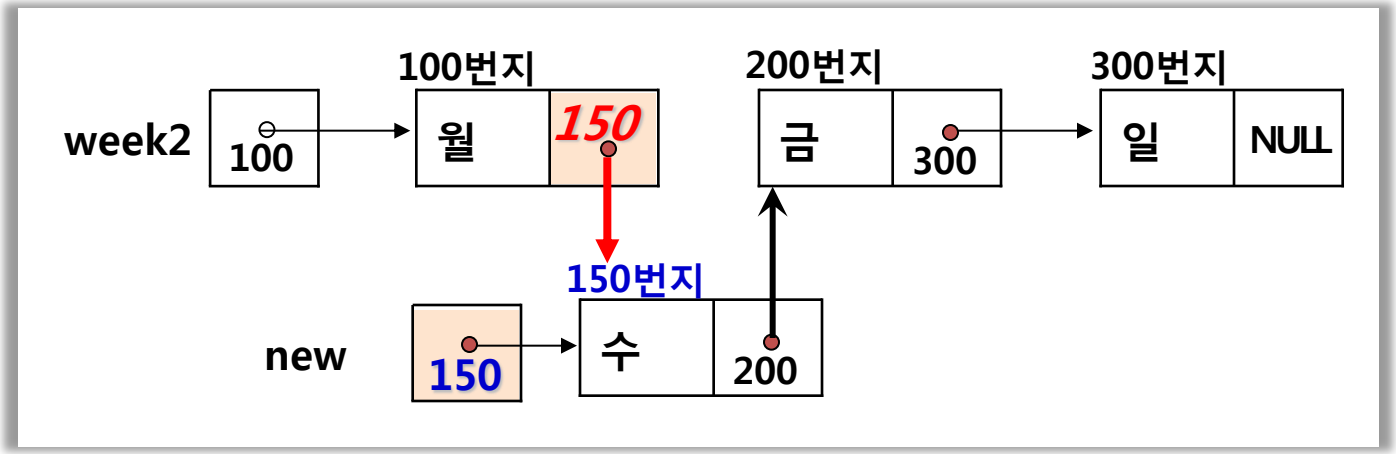


2. 단순 연결 리스트 : 삽입

❸ **새 노드의 링크 필드값 지정** : new의 앞 노드, 즉 "월"노드의 링크 필드 값을 new의 링크 필드에 저장



❹ **리스트의 앞 노드에 새 노드 연결** : new의 값을 "월"노드의 링크 필드에 저장



2. 단순 연결 리스트 : 첫 번째 노드로 삽입

- **단순 연결 리스트의 알고리즘**
  - 단순 연결 리스트의 첫 번째 노드로 삽입

알고리즘 4-1 단순 연결 리스트의 첫 번째 노드 삽입

```
insertFirstNode(L, x)
    ❶ new ← getNode();
    ❷ new.data ← x;
    ❸ new.link ← L;
    ❹ L ← new;
end insertFirstNode()
```

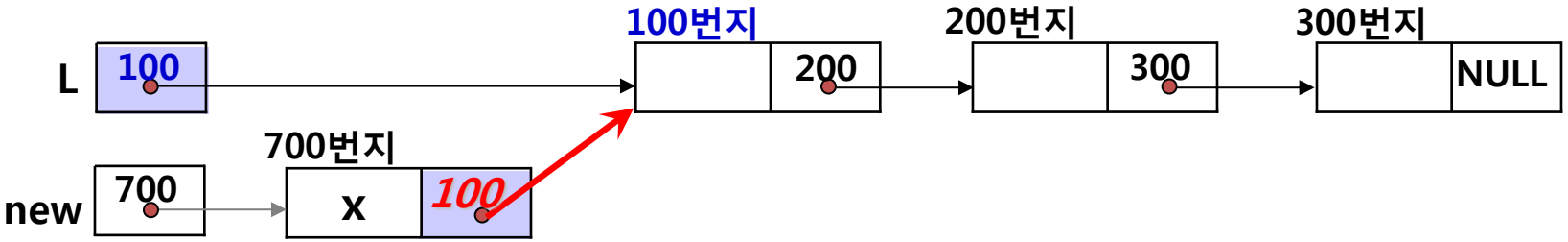
2. 단순 연결 리스트 : 첫 번째 노드로 삽입

❶ new ← getNode();

❷ new.data ← x;

❸ new.link ← L;

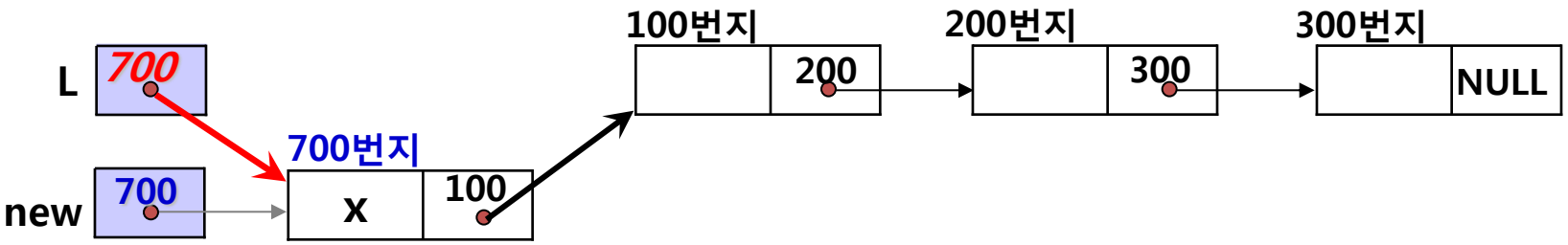
삽입할 노드를 연결하기 위해서 **리스트의 첫 번째 노드 주소(L)를** 삽입할 새 노드 **new의 링크 필드(new.link)에** 저장하여, 새 노드 new가 리스트의 첫 번째 노드를 가리키게 한다.



2. 단순 연결 리스트 : 첫 번째 노드로 삽입

❹ L ← new;

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



2. 단순 연결 리스트 : 마지막 노드로 삽입

- 단순 연결 리스트의 마지막 노드로 삽입

알고리즘 4-3 단순 연결 리스트의 마지막 노드 삽입

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

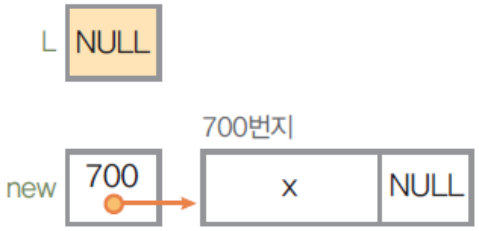
2. 단순 연결 리스트 : 마지막 노드로 삽입

❶ new ← getNode();

❷ new.data ← x;

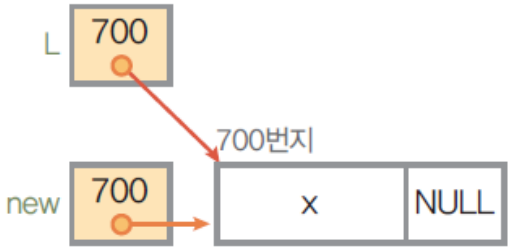
❸ new.link ← NULL;

❹ **공백 리스트인 경우**



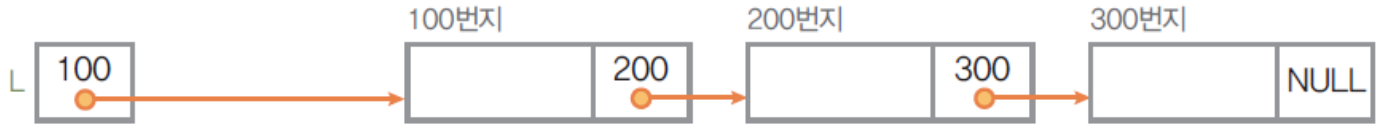
❹- a L ← new;

리스트 포인터 L에 새 노드 new의 주소(700) 저장. new는 리스트 L의 첫 번째 노드이자 마지막 노드



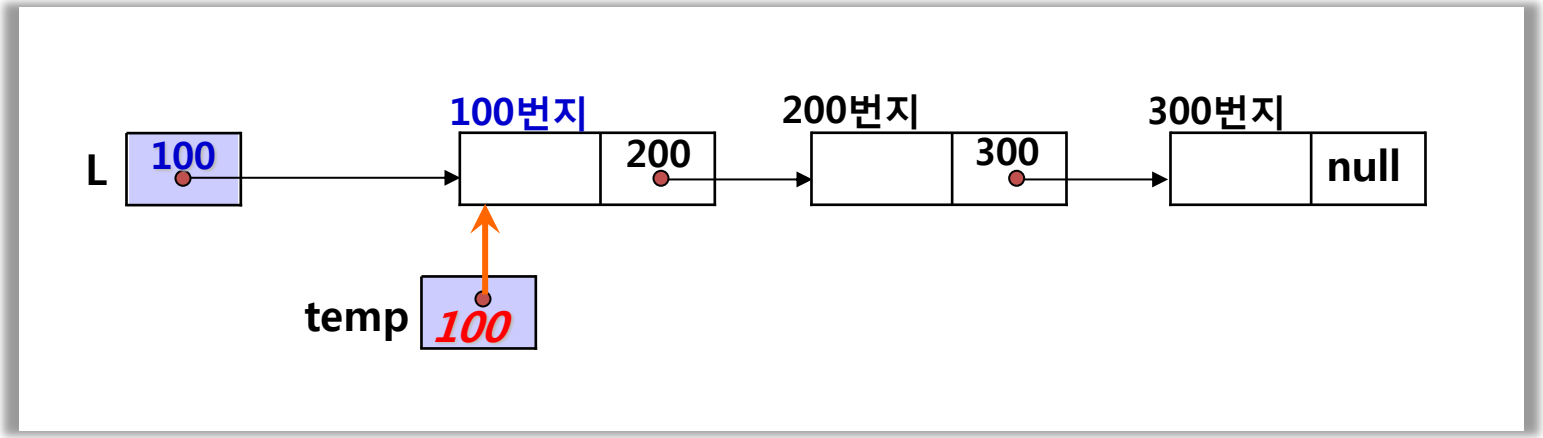
2. 단순 연결 리스트 : 마지막 노드로 삽입

❺ 공백 리스트가 아닌 경우



❺- a temp ← L;

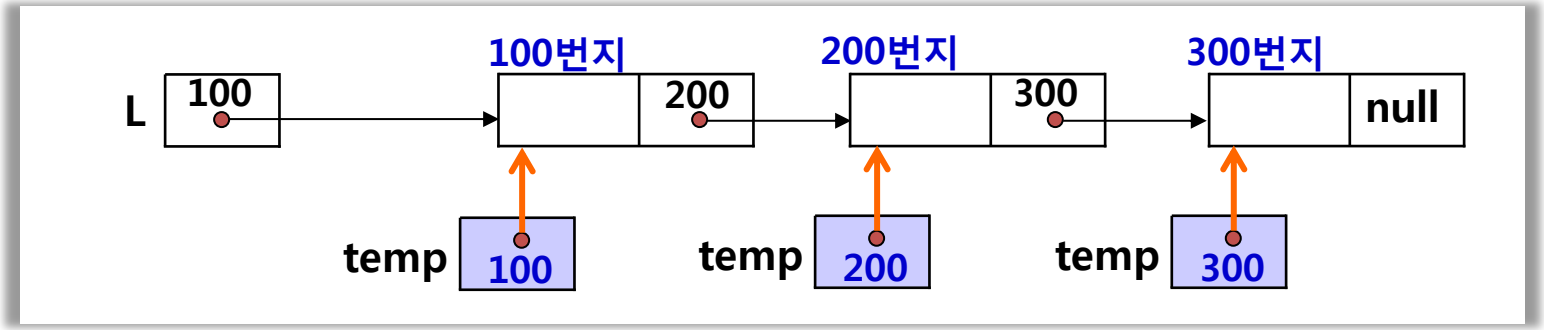
현재 리스트 L의 마지막 노드를 찾기 위해서 노드를 **순회할 임시포인터 temp에** 리스트의 **첫 번째 노드의 주소(L)를** 지정



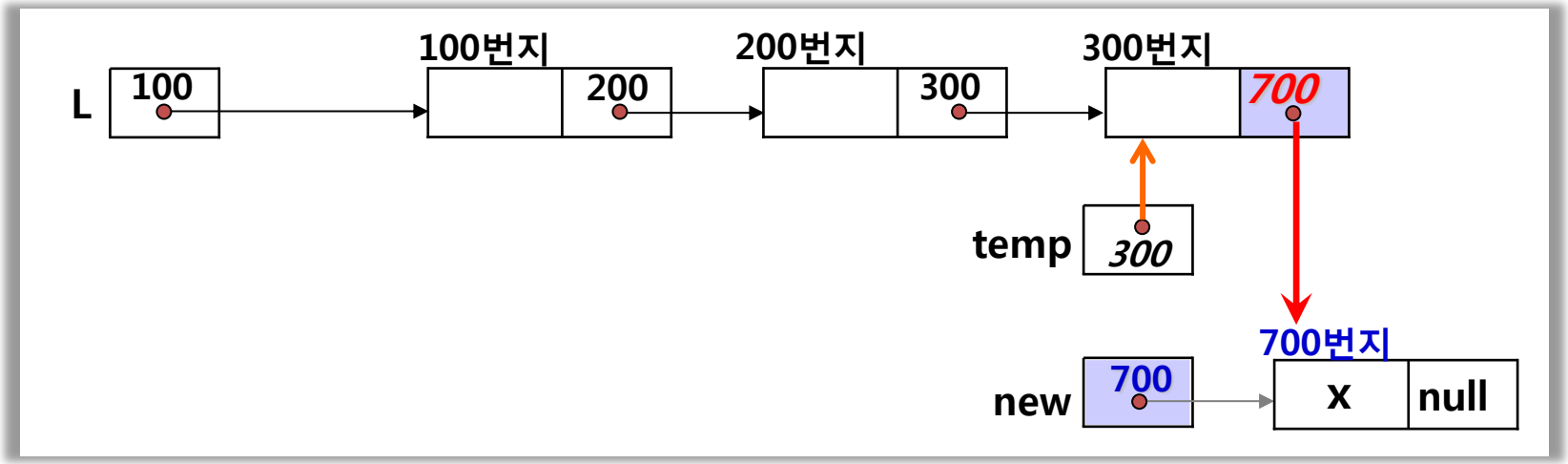
2. 단순 연결 리스트 : 마지막 노드로 삽입

❺- b temp ← temp.link;

순회 포인터 temp가 가리키는 노드의 링크 필드가 NULL이 아닌 동안(while (temp.link ≠ NULL)) 링크 필드를 따라 이동. 링크 필드가 NULL인 노드 즉, 마지막 노드를 찾으면 while 문을 끝내고 ❺- c를 수행



❺- c temp.link ← new;



2. 단순 연결 리스트 : 탐색

- **노드를 탐색하는 알고리즘과 프로그램**
  - 노드를 탐색하는 알고리즘

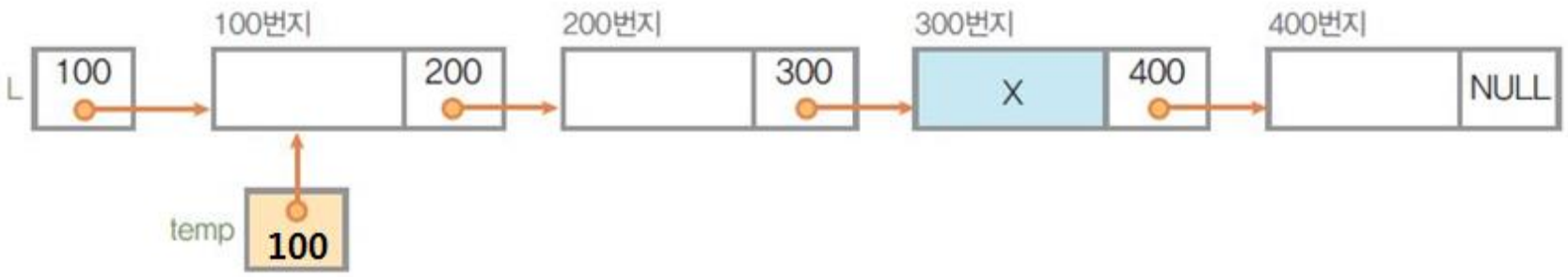
알고리즘 4-5 단순 연결 리스트의 노드 탐색

```
searchNode(L, x)
    ❶ temp ← L;
    ❷ while (temp ≠ NULL) do {
          ❷-ⓐ if (temp.data = x) then return temp;
          ❷-ⓑ else temp ← temp.link;
      }
    ❸ return temp;
end searchNode()
```

2. 단순 연결 리스트 : 탐색

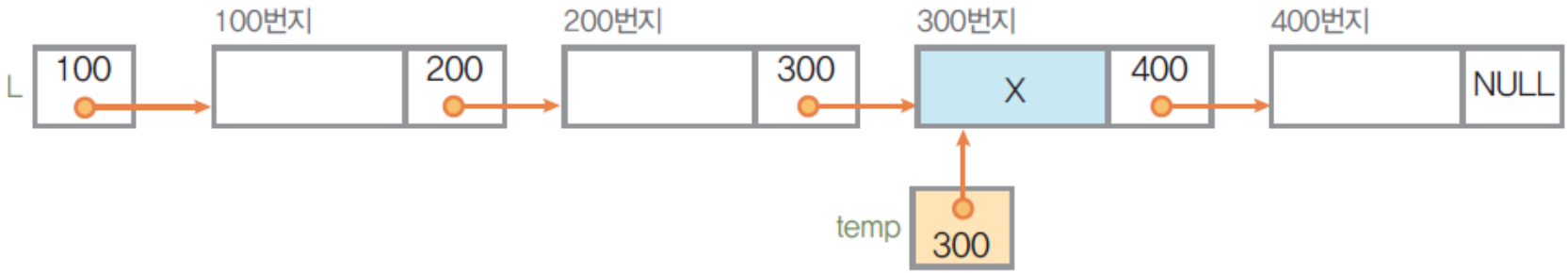
- 단순 연결 리스트에서 x 노드 탐색 과정

❶ temp ← L;



❷ 순회 포인터가 NULL이 아닌 경우

❷- a if (temp.data = x) then return temp;

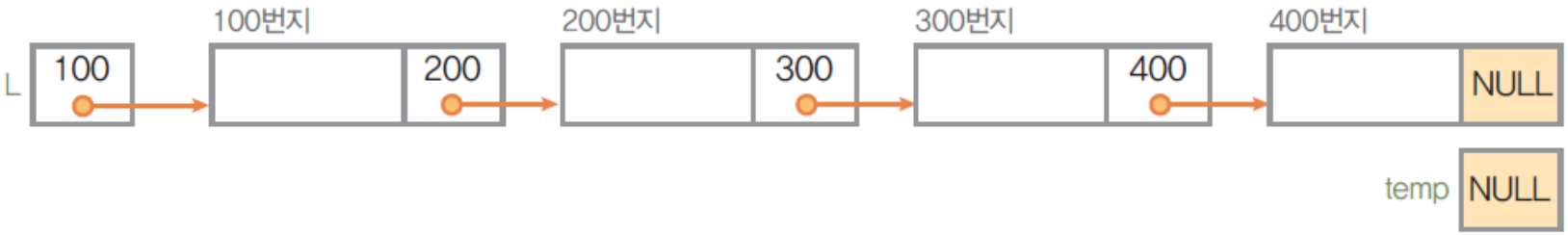


2. 단순 연결 리스트 : 탐색

- 단순 연결 리스트에서 x 노드 탐색 과정

  ❷- b else temp ← temp.link;

  ❸ return temp;



2. 단순 연결 리스트 : 중간 노드로 삽입

- 단순 연결 리스트 중간 노드로 삽입

알고리즘 4-2 단순 연결 리스트의 중간 노드 삽입

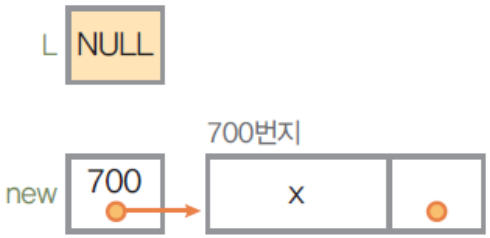
```
insertMiddleNode(L, pre, x)
    ❶ new ← getNode();
    ❷ new.data ← x;
    ❸ if (L = NULL) then {
          ❸-ⓐ L ← new;
          ❸-ⓑ new.link ← NULL;
      }
    ❹ else {
          ❹-ⓐ new.link ← pre.link;
          ❹-ⓑ pre.link ← new;
}
end insertMiddleNode()
```

2. 단순 연결 리스트 : 중간 노드로 삽입

❶ new ← getNode();

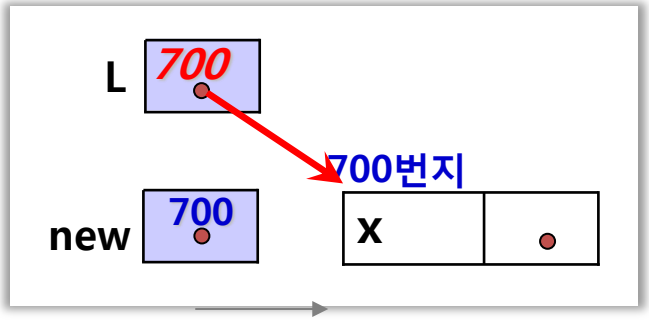
❷ new.data ← x;

❸ **공백 리스트인 경우**



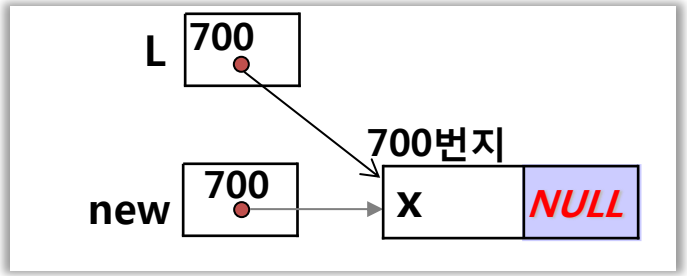
❸- a L ← new;

리스트 포인터 L에 새 노드 new의 주소를 저장하여, 새 노드 new가 리스트의 첫 번째 노드가 되도록 함



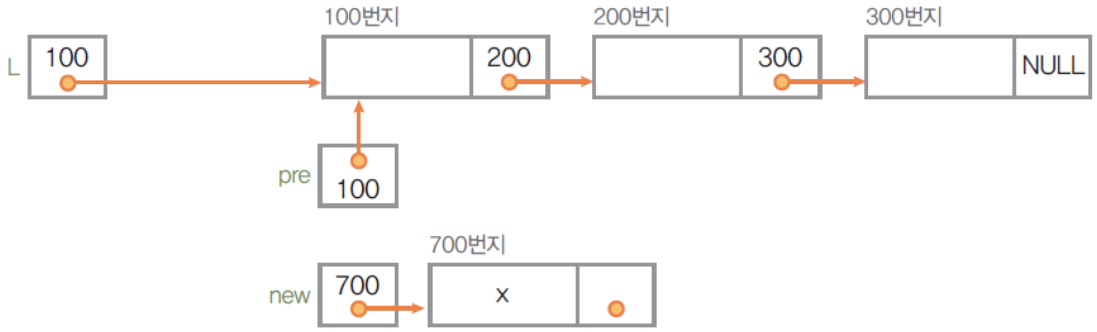
❸- b new.link ← NULL;

리스트의 마지막 노드인 new의 링크 필드에 NULL을 저장해 마지막 노드임을 표시



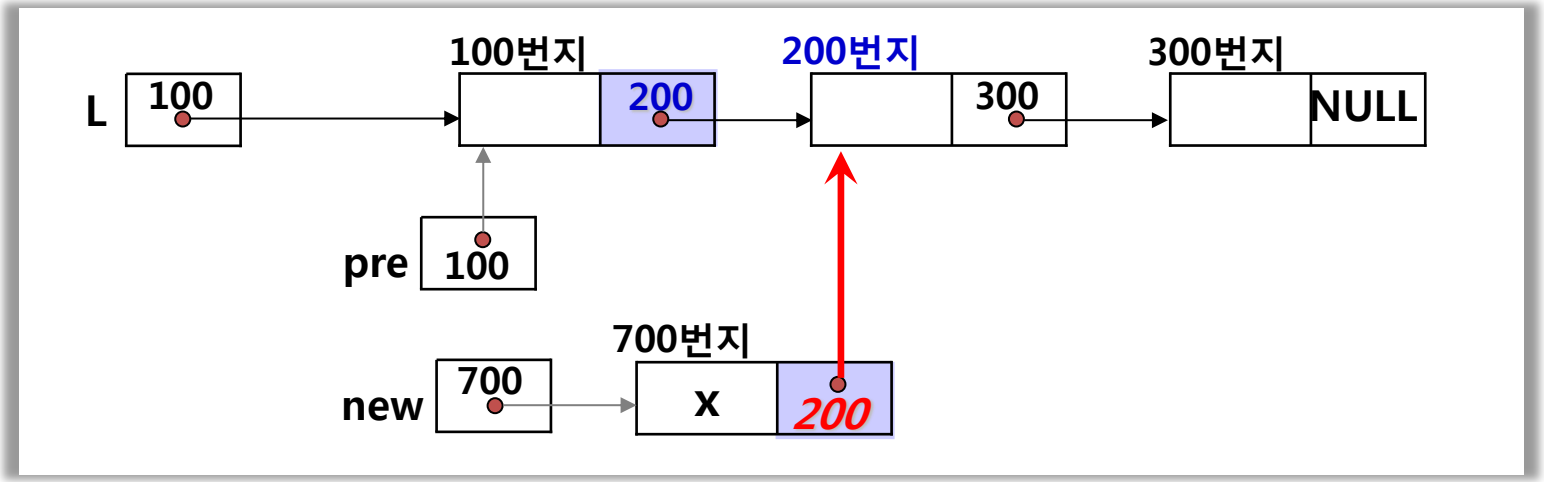
2. 단순 연결 리스트 : 중간 노드로 삽입

❹ 공백 리스트가 아닌 경우



❹- a new.link ← pre.link;

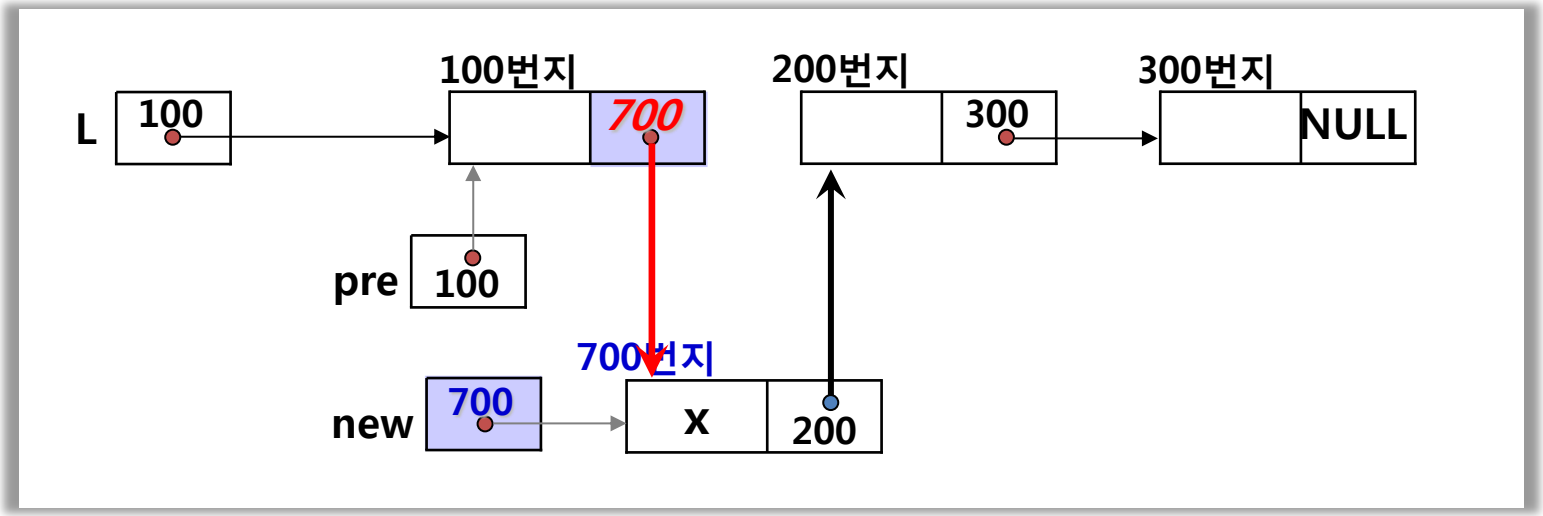
노드 pre의 링크 필드 값을 노드 new의 링크 필드에 저장하여, 새 노드 new가 노드 pre의 다음 노드를 가리키도록 함



2. 단순 연결 리스트 : 중간 노드로 삽입

❹- b pre.link ← new;

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



2. 단순 연결 리스트 : 삭제

❶ 삭제할 노드의 앞 노드를 찾는다.

❷ 앞 노드에 삭제할 노드의 링크 필드값을 저장한다.

❸ 삭제할 노드의 메모리 공간을 시스템에 반납한다.

그림 4-13 단순 연결 리스트에서 노드를 삭제하는 방법

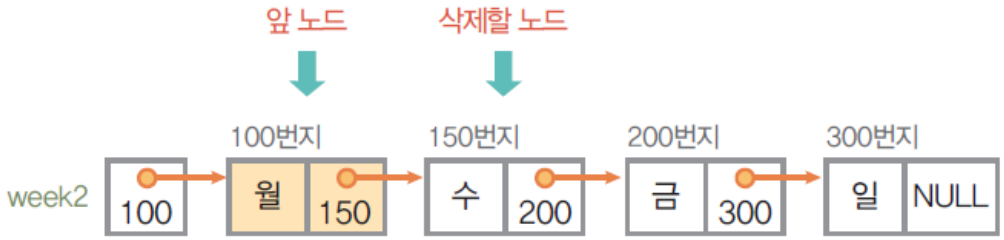
2. 단순 연결 리스트 : 삭제

- 단순 연결 리스트 week2=(월, 수, 금, 일)에서 원소 '수' 삭제 과정
  - 초기 상태

그림 4-14 단순 연결 리스트 week2에서 '수' 노드를 삭제하기 전의 초기 상태

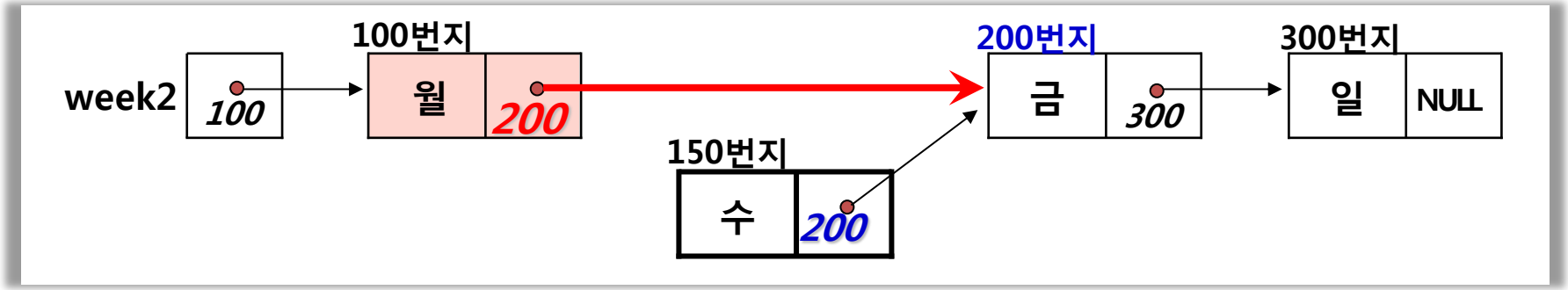
2. 단순 연결 리스트 : 삭제

❶ 앞 노드를 찾음 : 삭제할 원소의 앞 노드(선행자)를 찾음



❷ 앞 노드에 삭제할 노드의 링크 필드값 저장 : 삭제할 원소 "수"의 링크 필드 값을 앞 노드의 링크 필드에 저장

❸ 삭제한 노드의 앞뒤 노드 연결 : 삭제한 노드의 앞 노드인 '월' 노드를 삭제한 노드의 다음 노드인 '금' 노드에 연결



2. 단순 연결 리스트 : 삭제

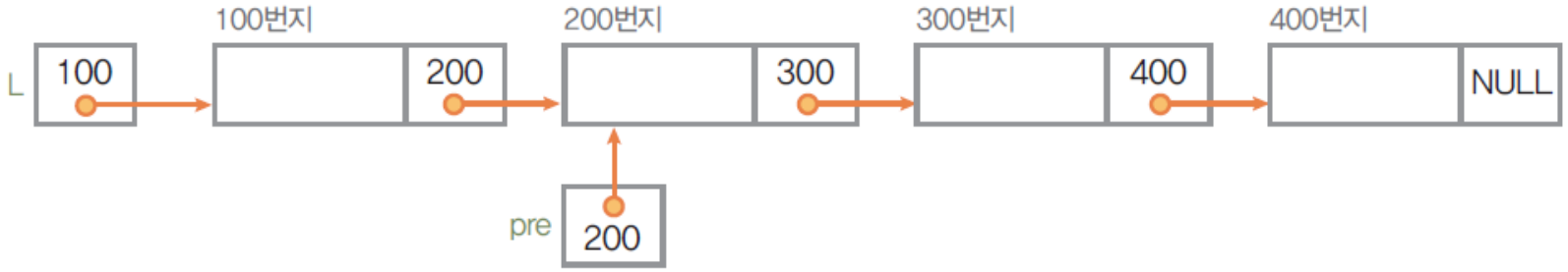
- 리스트 L에서 포인터 pre가 가리키는 노드의 다음 노드 삭제 알고리즘 : 포인터 old(삭제할 노드)

알고리즘 4-4 단순 연결 리스트의 노드 삭제

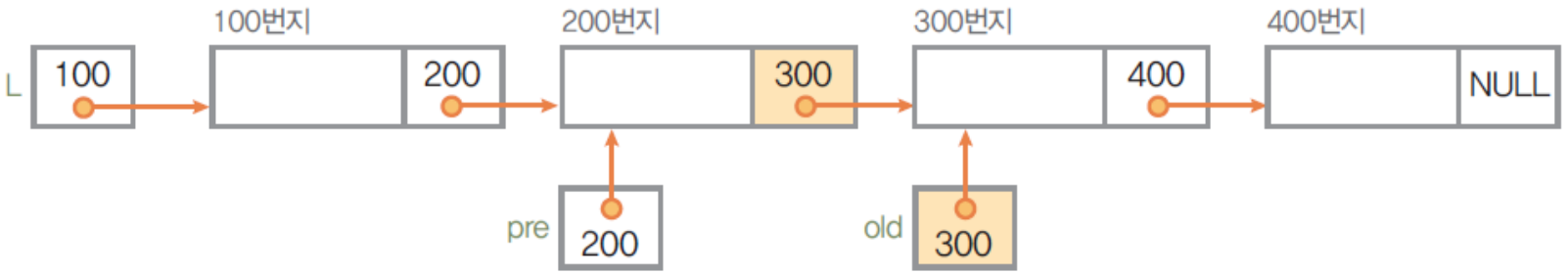
```
deleteNode(L, pre)
    ❶ if (L = NULL) then error;
    ❷ else {
        ❷-ⓐ old ← pre.link;
        ❷-ⓑ if (old = NULL) then return;
        ❷-ⓒ pre.link ← old.link;
        ❷-ⓓ returnNode(old);
      }
end deleteNode()
```

2. 단순 연결 리스트 : 삭제

- 삭제 연산 수행 과정
  - ❶ 공백 리스트인 경우
  - ❷ 공백 리스트가 아닌 경우

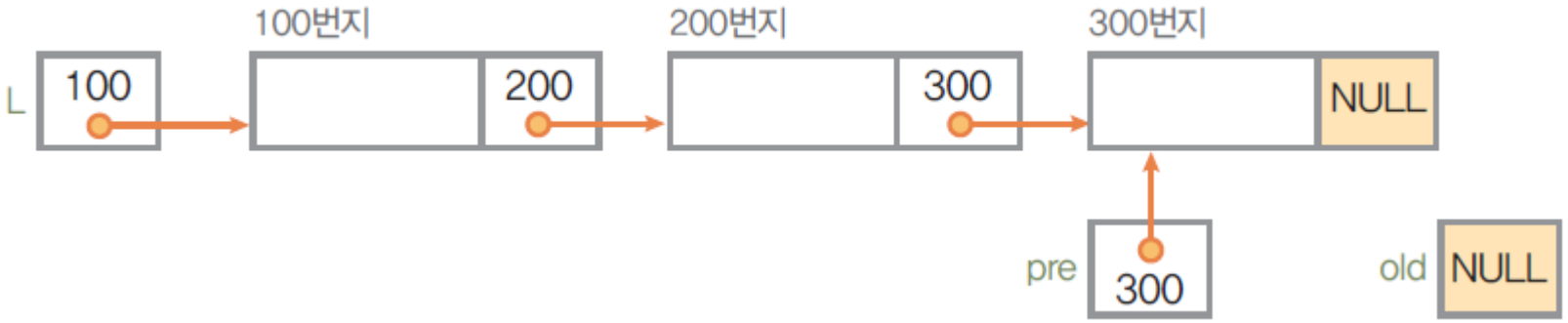


❷- a old ← pre.link;

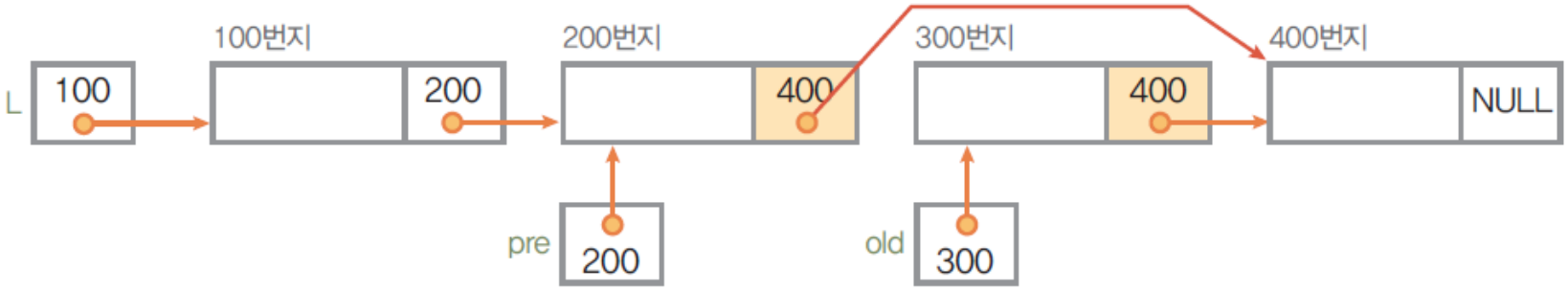


2. 단순 연결 리스트 : 삭제

❷- b if (old = NULL) then return;



❷- c pre.link ← old.link;



2. 단순 연결 리스트 : 삭제

❷- d returnNode(old);