

4장 연결 데이터 표현



순서

4.1 노드와 포인터

4.2 Java의 참조 변수

4.3 단순 연결 리스트

4.4 자유 공간 리스트

4.5 원형 연결 리스트

4.6 이중 연결 리스트

4.7 헤더 노드

4.8 다항식의 리스트 표현과 덧셈

4.9 일반 리스트

순차 표현

◆ 장점

- 표현이 간단함
- 원소의 접근이 빠름
 - ◆ 인덱스는 직접 메모리 주소로 변환할 수 있기 때문에 빠른 임의 접근이 가능

◆ 단점

- 원소의 삽입과 삭제가 어렵고 시간이 많이 걸림
 - ◆ 원소의 삽입, 삭제시 해당 원소의 위치 뒤에 있는 모든 원소를 뒤로 물리거나 앞으로 당겨야만 함
- 저장공간의 낭비와 비효율성
 - ◆ 리스트의 길이가 임의로 늘어나고 줄어드는 상황에서 배열의 적정 크기를 미리 결정하기가 어려움
 - ◆ 배열의 오버플로우(overflow)나 과도한 메모리 할당 발생

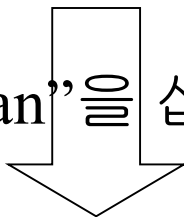
순서 리스트의 순차 표현

◆ 리스트 L의 순차 표현과 원소 삽입

- $L = (\text{Cho}, \text{Kim}, \text{Lee}, \text{Park}, \text{Yoo})$: 알파벳 순서로 된 성씨의 리스트

Cho	Kim	Lee	Park	Yoo	
L[0]	L[1]	L[2]	L[3]	L[4]	L[5]

“Han”을 삽입



Cho	Han	Kim	Lee	Park	Yoo
L[0]	L[1]	L[2]	L[3]	L[4]	L[5]

연결 표현

◆ 연결 표현 (linked representation) 또는 비순차 표현

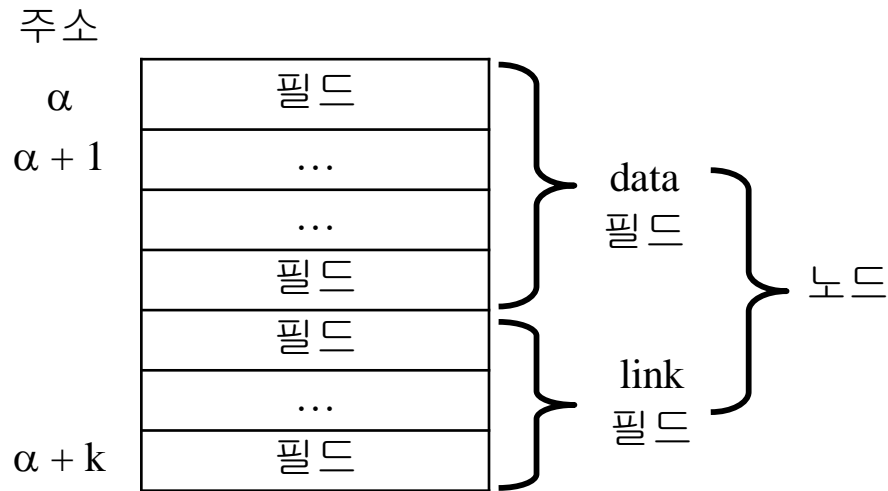
- 원소의 물리적 순서가 리스트의 논리적 순서와 일치할 필요 없음
- 원소를 저장할 때 이 원소의 다음 원소에 대한 주소도 함께 저장해야 함
- 노드 : <원소, 주소> 쌍의 저장 구조

◆ 노드 (node) : 데이터 필드와 링크 필드로 구성

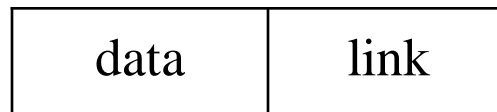
- 데이터 필드 – 리스트의 원소, 즉 데이터 값을 저장하는 곳
- 링크 필드 – 다른 노드의 주소값을 저장하는 장소 (포인터)

노드 구조

◆ 물리적 노드 구조



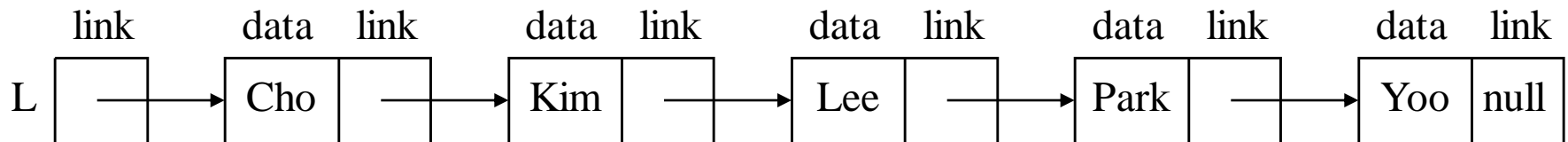
◆ 논리적 노드 구조



연결 리스트 (linked list)

◆ 연결 리스트

- 링크를 이용해 표현한 리스트
- 예
 - ◆ “리스트 L” – 리스트 전체를 가리킴
 - ◆ “노드 L” - 리스트의 첫번째 노드(“Cho”)를 가리킴



연결 리스트 표현 (1)

- ◆ 1. 링크에 실제로는 화살표 대신 주소값이 들어가 있음
 - 예) 원소 “Cho”, “Kim”, “Lee”, “Park”, “Yoo”가 저장된 노드들의 주소가 각각 1000, 1004, 990, 1110, 1120일 때

L	<table><tr><td>1000</td></tr></table>	1000		
1000				
1000	<table><tr><td>Cho</td></tr><tr><td>1004</td></tr></table>	Cho	1004	data link
Cho				
1004				
1004	<table><tr><td>Kim</td></tr><tr><td>990</td></tr></table>	Kim	990	data link
Kim				
990				
990	<table><tr><td>Lee</td></tr><tr><td>1110</td></tr></table>	Lee	1110	data link
Lee				
1110				
1110	<table><tr><td>Park</td></tr><tr><td>1120</td></tr></table>	Park	1120	data link
Park				
1120				
1120	<table><tr><td>Yoo</td></tr><tr><td>null</td></tr></table>	Yoo	null	data link
Yoo				
null				

연결 리스트 표현 (2)

- ◆ 2. 연결 리스트의 마지막 노드의 링크 값으로 **null**을 저장
- ◆ 3. 연결 리스트 전체는 포인터 변수 **L**로 나타냄. 여기에는 리스트의 첫 번째 노드의 주소가 저장됨
 - 공백 리스트의 경우 L의 값은 null이 됨. 이렇게 null 값을 가진 포인터를 널 포인터라 함
- ◆ 4. 노드의 필드 선택은 점연산자 (**· : dot operator**) 를 이용
 - 예) L.data : L이 가리키는 노드의 data 필드값, “Cho”
L.link.data : L이 가리키는 노드의 link 값이 가리키는 노드의 data 값, “Kim”

연결 리스트 표현 (3)

◆ 5. 포인터 변수에 대한 연산은 제한되어 있음

- ① $P = \text{null}$ 또는 $P \neq \text{null}$: 공백 포인터인가 또는 공백 포인터가 아닌가의 검사
- ② $P = P_1$: 포인터 P 와 P_1 이 모두 같은 노드 주소를 가리키는가?
- ③ $P \leftarrow \text{null}$: P 의 포인터 값으로 공백 포인터를 지정
- ④ $P \leftarrow P_1$: P_1 이 가리키는 노드를 P 도 같이 가리키도록 지정
- ⑤ $P.\text{link} \leftarrow P_1$: P 가 가리키는 노드의 링크 필드에 P_1 의 포인터값을 지정
- ⑥ $P \leftarrow P_1.\text{link}$: P_1 이 가리키는 노드의 링크 값을 포인터 P 의 포인터 값으로 지정

Java의 참조 변수 (1)

◆ 참조 변수 (reference variable)

- 객체가 만들어져 있는 메모리 주소를 저장하고 있는 변수
- 다른 언어의 포인터 변수와 비슷하나 산술 연산이 허용되지 않음

◆ Point 클래스 정의의 예

```
class Point {  
    int x;  
    int y;  
    public Point() {  
        x = 0;    y = 0;  
    }  
    public Point(int x1, int y1) {  
        x = x1;  y = y1;  
    }  
}
```

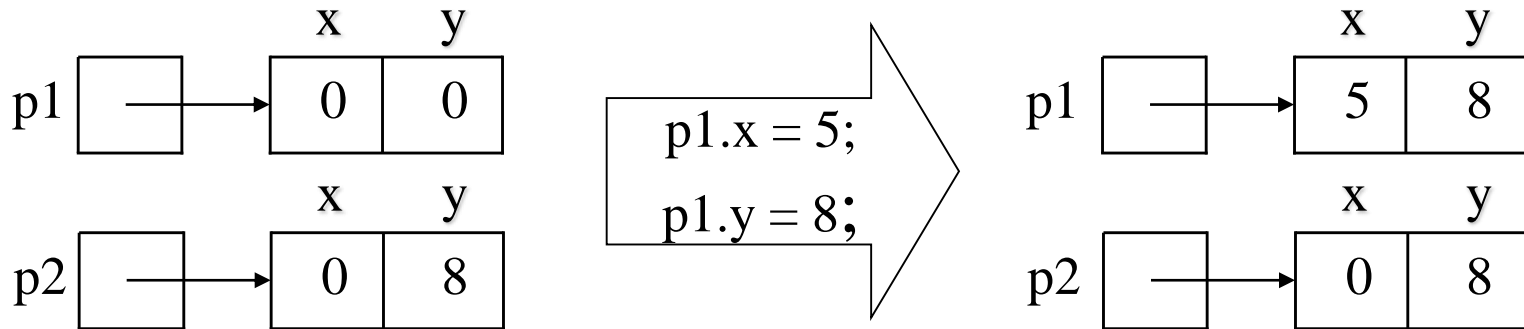
Java의 참조 변수 (2)

◆ Point 객체의 생성

- `Point p1 = new Point();`
 - ◆ (1) 새로운 Point 객체를 위해 메모리를 할당
 - ◆ (2) 이 객체의 모든 데이터 필드들을 0으로 초기화
 - ◆ (3) 이 객체에 대한 참조(주소)를 반환하여 참조 변수 p1의 초기값으로 저장
- `Point p2 = new Point(0, 8);`
 - ◆ 2개의 매개 변수를 가진 생성자를 이용하여 새로운 Point 객체 생성
 - ◆ 데이터 필드 값을 $x = 0$, $y = 8$ 로 초기화
 - ◆ 이 객체에 대한 참조를 참조 변수 p2의 값으로 저장

Java의 참조 변수 (3)

◆ 데이터 필드 값의 변경



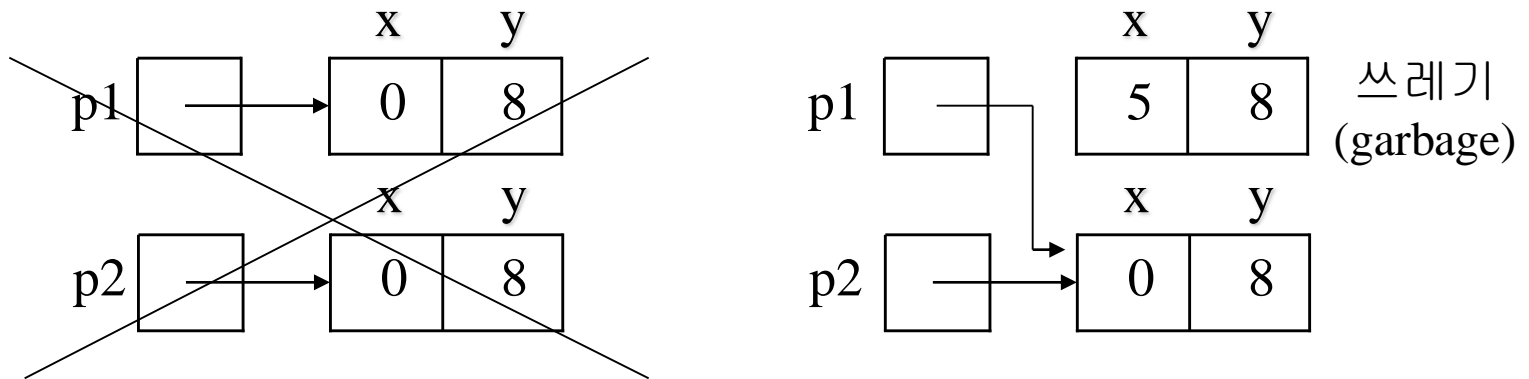
◆ 지정문 **p1 = p2;**

- p1이 가리키는 객체의 모든 데이터 필드 내용들을 p2가 가리키는 객체의 데이터 필드값들로 대체?
- p1의 참조값이 p2의 참조값으로 대체?

Java의 참조 변수 (4)

◆ p1 = p2; 의 결과

- 변수 p2에 저장되어 있는 참조값을 복사해서 변수 p1의 새로운 참조값으로 저장
- 변수 p1과 p2가 하나의 객체를 공동으로 참조
 - ◆ 어느 한 변수를 통해 객체의 필드값을 변경하면 또다른 변수로 이 값을 접근할 때 변경된 값을 접근



- 이 때 p1이 가리키는 객체의 저장소는 접근할 수 없는 쓰레기(garbage)가 됨
- Java 시스템이 저장소가 부족하게 될 때 쓰레기 수집기(garbage collector)를 기동시켜 쓰레기를 수집하여 재활용 (Java 시스템 내에서 자동으로 이루어짐)

Java로 정의한 노드 구조

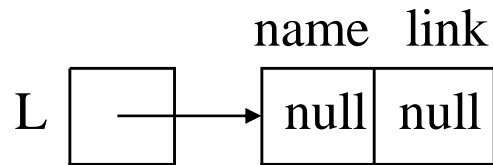
◆ 스트링 리스트에 대한 연결 표현을 위한 노드 구조

```
class ListNode {  
    String name;  
    ListNode link;  
}
```

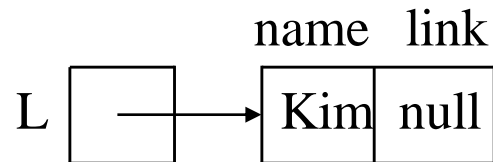
- name 필드 - String값을 가짐
- link 필드 - 다른 ListNode 객체를 가리키는 참조값을 가짐
- 리스트는 link 필드들을 통해 연결되는 ListNode 클래스의 객체들이 됨
- link 값이 null이면 그 리스트의 마지막 노드라는 것을 나타냄
- ListNode 변수가 null이면 공백리스트를 나타냄

리스트 생성의 예 (1)

- ◆ 1. L을 ListNode 타입의 변수로 선언하고 하나의 새로운 ListNode로 초기화
 - `ListNode L = new ListNode();`
 - 지정 생성자(default constructor)에 의해 name과 link 필드는 null로 초기화



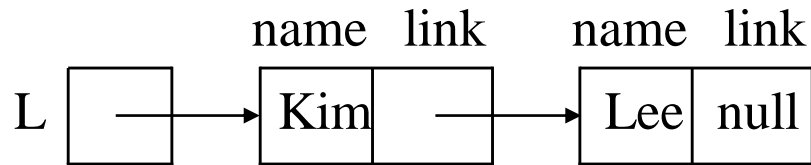
- ◆ 2. 노드 L의 name 필드에 “Kim”을 지정하고 link 필드에는 null을 지정
 - `L.name = “Kim”;`
 - `L.link = null;`



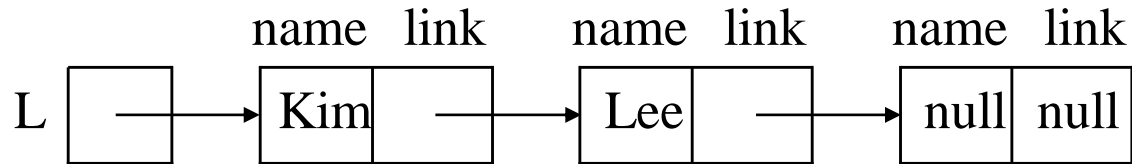
리스트 생성의 예 (2)

◆ 3. 리스트의 L에 “Lee”와 “Park”에 대한 두 개의 새로운 ListNode를 차례로 첨가

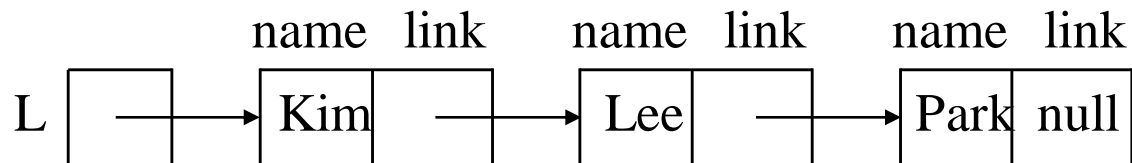
- L.link = new ListNode();
- L.link.name = “Lee”;



- L.link.link = new ListNode();



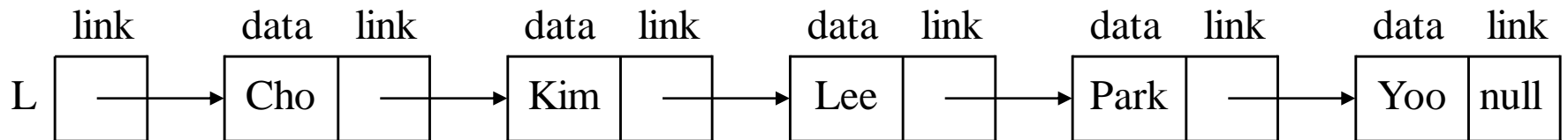
- L.link.link.name = “Park”;
- L.link.link.link = null;



단순 연결 리스트

◆ 단순 연결 리스트 (singly linked list)

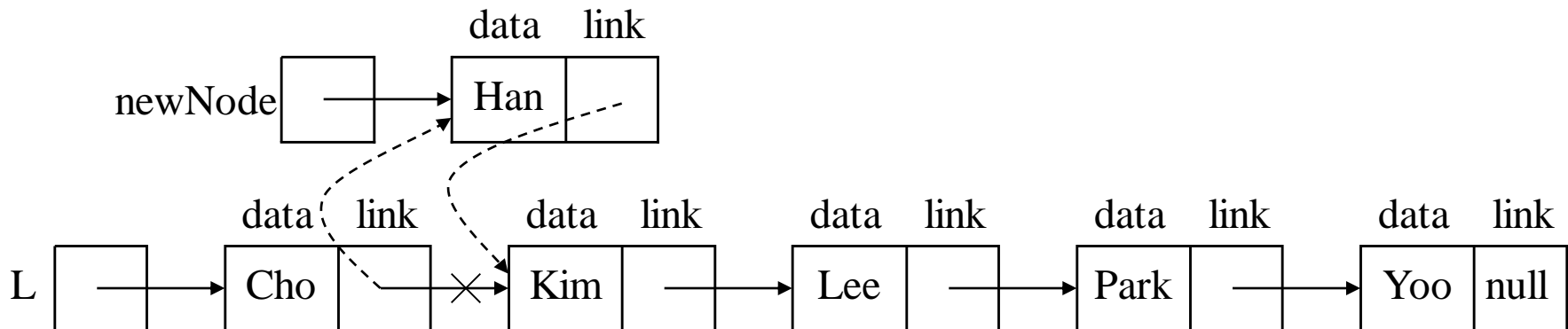
- 하나의 링크 필드를 가진 노드들이 모두 자기 후속노드와 연결되어 있는 노드 열
- 마지막 노드의 링크 필드는 리스트의 끝을 표시하는 null값을 가짐
- 별칭 : 선형 연결 리스트(linear linked list), 단순 연결 선형 리스트(singly linked linear list), 연결 리스트(linked list), 체인(chain)
- 단순 연결 리스트의 예



원소의 삽입

◆ 원소 삽입 알고리즘

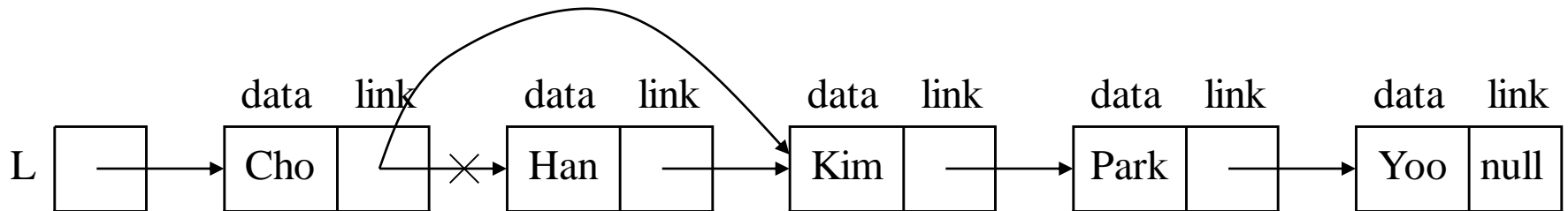
- 예) 리스트 L에 원소 “Han”을 “Cho”와 “Kim” 사이에 삽입
 - ◆ 1. 공백노드를 획득함. newNode라는 변수로 가리키게 함
 - ◆ 2. newNode의 data 필드에 “Han”을 저장
 - ◆ 3. “Cho”를 저장하고 있는 노드의 link 값을 newNode의 link 필드에 저장
(즉 “Kim”을 저장하고 있는 노드의 주소를 newNode의 link에 저장)
 - ◆ 4. “Cho”를 저장한 노드의 link에 newNode의 포인터 값을 저장
- 리스트의 기존 원소들을 이동시킬 필요 없음
- 부수적인 link 필드를 위해 저장 공간을 추가로 사용



원소의 삭제

◆ 원소 삭제 알고리즘

- 예) 리스트 L에서 원소 “Han”을 삭제
 - 1. 원소 “Han”이 들어 있는 노드의 선행자 찾음 (“Cho”가 들어있는 노드)
 - 2. 이 선행자의 link에 “Han”이 들어있는 노드의 link 값을 저장



메모리의 획득과 반납

- ◆ 연결 리스트가 필요로 하는 두 가지 연산
 - 데이터 필드와 하나의 링크 필드를 가진 하나의 공백 노드를 획득하는 방법
 - 사용하지 않는 노드는 다시 반납하여 재사용하는 방법
- ◆ 자유 공간 리스트 (free space list) 가 있는 경우
 - getNode()
 - ◆ 데이터와 링크 필드로 되어 있는 새로운 공백 노드를 가용 공간 리스트로부터 할당받아 그 주소를 반환하는 함수
 - returnNode()
 - ◆ 포인터 변수 P가 지시하는 노드를 가용 공간 리스트에 반환하는 함수
 - ◆ 프로그래밍 언어에 따라 필요한 경우도 있고 필요하지 않은 경우도 있음
 - ◆ Java와 같이 사용하지 못하는 노드를 자동으로 관리해 주는 언어에서도 사용자가 직접 관리할 필요가 있는 경우도 있음

리스트 생성 알고리즘 (1)

◆ 앞의 두 함수를 사용한 리스트 생성 알고리즘

```
newNode ← getNode();    // 첫번째 공백 노드를 할당받아
                          // newNode가 그 주소를
                          // 지시하도록 함
newNode.data ← "Cho";    // 원소 값을 저장

L ← newNode;             // 리스트 L을 만듦

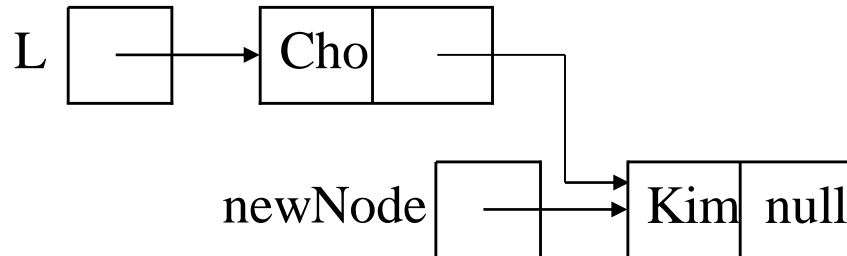
newNode ← getNode();    // 두 번째 공백 노드를 획득
                          // 두 번째 노드에 원소값과 null을 저장
newNode.data ← 'Kim';
newNode.link ← null;

L.link ← newNode;       // 두 번째 노드를 리스트 L에 연결
```

리스트 생성 알고리즘 (2)

◆ 리스트 생성 알고리즘을 점 표기식으로 작성한 경우

```
L ← getNode();  
L.data ← "Cho";  
L.link ← getNode();  
L.link.data ← "Kim";  
L.link.link ← null;
```



첫번째 노드 삽입

- ◆ 리스트 **L**의 첫번째 노드로 **data** 값이 **x**인 노드를 삽입

```
addFirstNode(L, x)
    // 리스트 L의 첫번째 노드로 원소 x를 삽입
    newNode ← getNode();
    newNode.data ← x;
    newNode.link ← L;
    L ← newNode;
end addFirstNode()
```

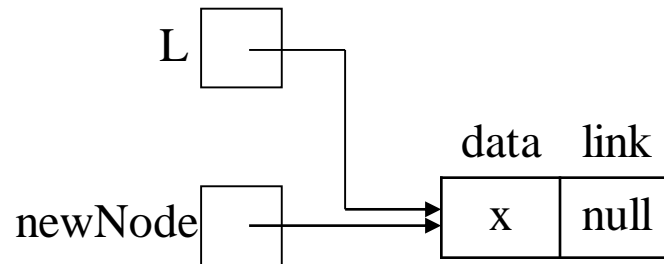

노드의 삽입 (1)

◆ 원소값이 x 인 노드를 p 가 가리키는 노드 다음에 삽입

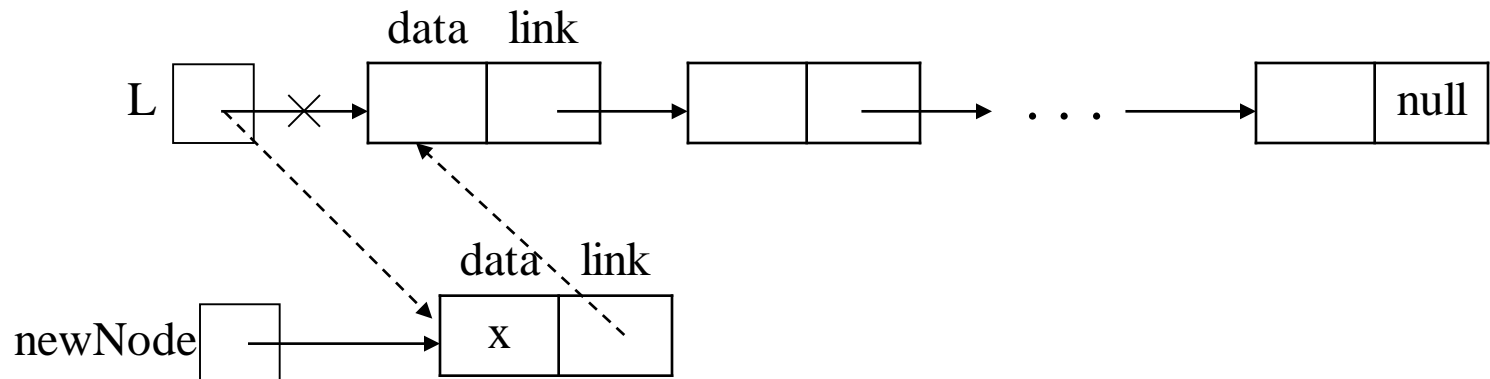
```
insertNode(L, p, x)
    // 리스트 L에서 p 노드 다음에 원소값 x를 삽입
    newNode ← getNode();    // 공백 노드를 newNode가 지시
    newNode.data ← x;        // 데이터값 x를 저장
    if L=null then {        // L이 공백 리스트인 경우
        L ← newNode;
        newNode.link ← null;
    }
    else if p=null then {    // p가 공백이면 L의 첫 번째 노드로 삽입
        newNode.link ← L;
        L ← newNode;
    }
    else {                  // p가 가리키는 노드 다음에 삽입
        newNode.link ← p.link;
        p.link ← newNode;
    }
end insertNode()
```

노드의 삽입 (2)

- (a) L이 공백 리스트인 경우

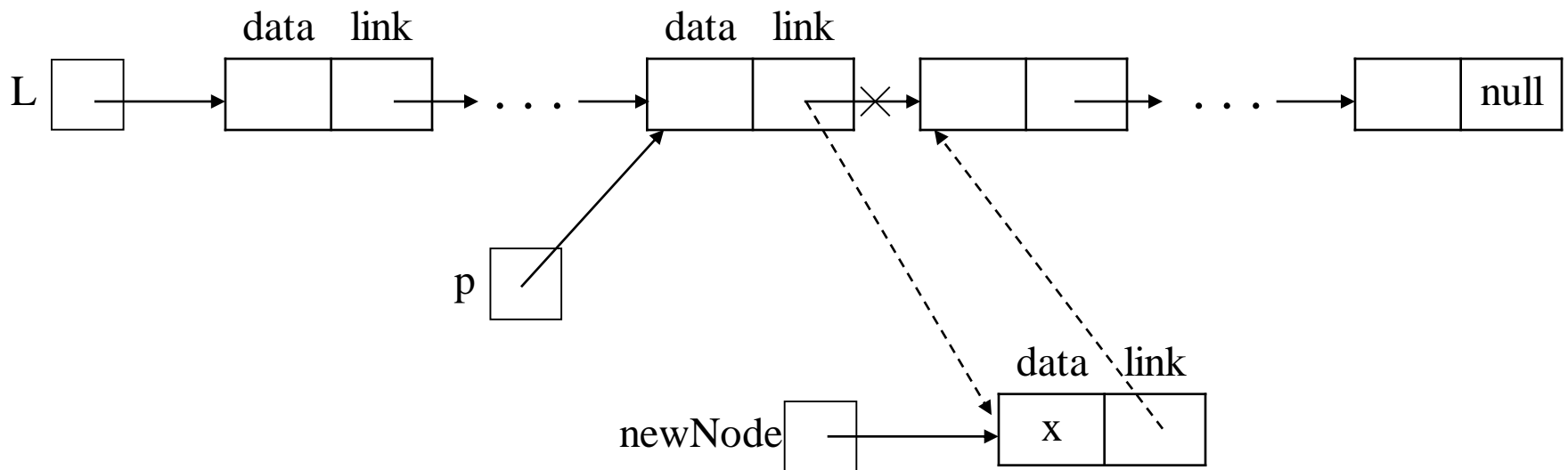


- (b) P가 null인 경우



노드의 삽입 (3)

- (c) L과 p가 null이 아닌 경우



마지막 노드 삽입

◆ 리스트 L의 마지막 노드로 원소값 x를 첨가

```
addLastNode(L, x)
    // 리스트 L의 끝에 원소 x를 삽입
    newNode ← getNode();    // 새로운 노드 생성
    newNode.data ← x;
    newNode.link = null;
    if L = null then {
        L ← newNode;
        return;
    }
    p ← L;    // p는 임시 순회 포인터 변수
    while p.link ≠ null do    // 리스트의 마지막 노드를 찾음
        p ← p.link;
    p.link ← newNode;    // 마지막 노드로 첨가
end addLastNode()
```

노드의 삭제

◆ 리스트 L에서 p가 가리키는 노드의 다음 노드를 삭제

```
delete(L, p)
  // p노드 다음 원소를 삭제
  if (L = null) then error;
  if p = null then {
    q ← L;   // q는 삭제할 노드
    L ← L.link;  // 첫 번째 노드 삭제
  } else {
    q ← p.link;  // q는 삭제할 노드
    if q = null then return;  // 삭제할 노드가 없는 경우
    p.link ← q.link;
  }
  returnNode(q); // 삭제한 노드를 자유 공간 리스트에 반환
end delete()
```

노드의 순서를 역순으로 변환

◆ 리스트를 역순으로 만드는 알고리즘

```
reverse(L)
  // L = (e1, e2, ..., en)을 L = (en, en-1, ..., e1)으로 변환
  // 순회 포인터로 p, q, r을 사용한다.
  p ← L;    // p는 역순으로 만들 리스트
  q ← null;  // q는 역순으로 만들 노드
  while (p ≠ null) do {
    r ← q;   // r은 역순으로 된 리스트
             // r은 q, q는 p를 차례로 따라간다.

    q ← p;
    p ← p.link;
    q.link ← r;  // q의 링크 방향을 바꾼다.
  }
  L ← q;
end reverse()
```

리스트 연결

◆ 두개의 리스트 L_1 과 L_2 를 하나로 만드는 알고리즘

```
addList(L1, L2)
  //  $L1 = (a1, a2, \dots, an)$ ,  $L2 = (b1, b2, \dots, bm)$  일 때,
  //  $L = (a1, a2, \dots, an, b1, b2, \dots, bm)$  을 생성
  case {
    L1 = null:  return L2;
    L2 = null:  return L1;
    else:  p  $\leftarrow$  L1;  // p는 순회 포인터
           while p.link  $\neq$  null do
             p  $\leftarrow$  p.link;
             p.link  $\leftarrow$  L2;
           return L1;
  }
end addList()
```

원소값 탐색 (1)

◆ 데이터 값이 x 인 노드를 찾는 알고리즘

```
searchNode(L, x)
  p ← L; // p는 임시 포인터
  while (p ≠ null) do {
    if p.data = x then return p; // 원소 값이 x인 노드를 발견
    p ← p.link;
  }
  return p; // 원소 값이 x인 노드가 없는 경우 null을 반환
end searchNode()
```


원소값 탐색 (2)

◆ 데이터 값이 x 인 노드를 찾는 알고리즘의 Java 구현

```
public ListNode searchNode(String x) {  
    ListNode p; // ListNode를 가리킬 수 있는 참조변수 선언  
    p = L;      // p는 L의 첫 번째 노드를 가리킴  
    while (p != null) {  
        if (x.equals(p.data)) // p의 data 필드 값을 비교  
            return p;  
        p = p.link;  
    }  
    return p; // x 값을 가진 노드가 없는 경우 null을 반환  
} // end searchNode()
```

마지막 노드 삭제 (1)

◆ 리스트에서 마지막 노드를 삭제하는 프로그램

```
public void deleteLastNode() {  
    ListNode previousNode, currentNode;  
        // 두 개의 ListNode에 대한 포인터 선언  
    if (L == null) return;  
        // L이 공백리스트이면 아무것도 하지 않음  
    if (L.link == null) L = null;    // 노드가 하나만 있는 경우  
    else {  
        previousNode = L;  
        // 초기에 previousNode는 첫 번째 노드를 지시  
        currentNode = L.link;  
        // 초기에 currentNode는 두 번째 노드를 지시
```

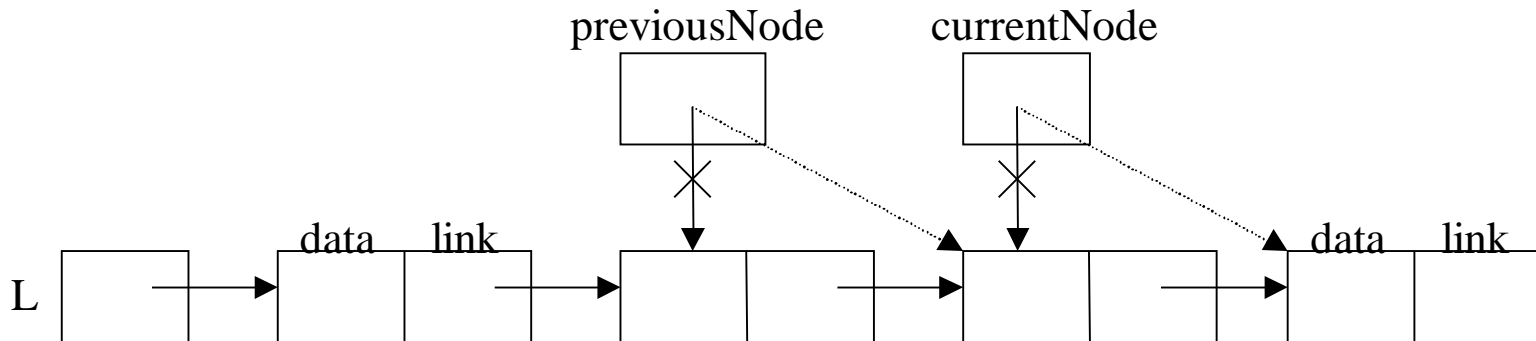
마지막 노드 삭제 (2)

```
while (currentNode.link != null) {  
    // currentNode가 마지막에 도착할 때까지 이동  
    previousNode = currentNode;  
    currentNode = currentNode.link;  
}  
previousNode.link = null;  
// previousNode가 지시하는 마지막 두 번째 노드를  
// 마지막 노드로 만듦  
}  
} // end deleteLastNode()
```

마지막 노드 삭제 (3)

◆ currentNode와 previousNode의 동작 과정

- currentNode 포인터가 어떤 노드를 가리키면 previousNode 포인터는 currentNode가 가리키는 노드의 바로 직전 노드를 가리키도록 함
- currentNode가 리스트의 마지막 노드를 가리키게 될 때 previousNode는 마지막 두 번째 노드를 가리키게 됨



리스트 출력

◆ 연결 리스트의 프린트

```
public void printList() {  
    ListNode p;  
    System.out.print("("); // 제일 먼저 "("를 프린트  
    p = L; // 순회 포인터로 사용  
    while (p != null) {  
        System.out.print(p.data); // data 필드 값을 프린트  
        p = p.link; // 다음 노드로 이동하여  
        if (p != null) { // 공백인가를 검사  
            System.out.print(","); // 원소 사이에 "," 프린트  
        }  
    }  
    System.out.print(")"); // 마지막으로 ")"를 프린트  
} // end printList()
```

예제 프로그램 (1)

◆ ListNode 클래스

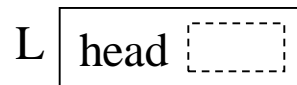
- data 필드와 link 필드를 가진 객체 구조를 정의
- 이 필드들을 초기화시키기 위해 생성자 정의

◆ LinkedList 클래스

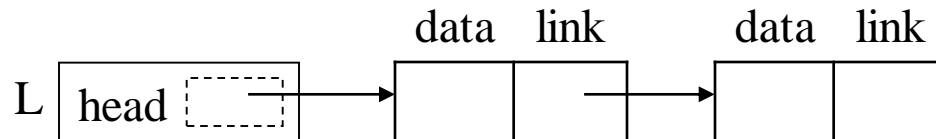
- 실제 연결 리스트를 정의
- head 필드 : private, ListNode 타입으로 선언, 연결 리스트의 첫 번째 노드를 가리킴

◆ LinkedList 생성의 예

- `LinkedList L = new LinkedList();`



- L에 노드가 연결되어 있는 경우



예제 프로그램 (2)

◆ 단순 연결 리스트의 처리

```
public class ListNode {  
    String data;  
    ListNode link;  
    public ListNode() {  
        data = link = null;  
    }  
    public ListNode(String val) {  
        data = val;  
        link = null;  
    }  
    public ListNode(String val, ListNode p) {  
        data = val;  
        link = p;  
    }  
} // end ListNode class
```

예제 프로그램 (3)

```
public class LinkedList{
    private ListNode head;
    public void addLastNode(String x) { // 알고리즘 4.2의 Java 구현
        // list의 끝에 원소 x를 삽입
        ListNode newNode = new ListNode();
        newNode.data = x;
        newNode.link = null;
        if (head == null) {
            head = newNode;
            return;
        }
        ListNode p = head;
        while (p.link != null) { // 마지막 노드를 탐색
            p = p.link;
        }
        p.link = newNode;
    } // end addLastNode()
}
```


예제 프로그램 (4)

```
public void reverse() { // 알고리즘 4.4의 Java 구현
    ListNode p = head;
    ListNode q = null;
    ListNode r = null;
    while (p != null) {
        r = q;
        q = p;
        p = p.link;
        q.link = r;
    }
    head = q;
} // end reverse()
```

```
public void deleteLastNode() { // 프로그램 4.2
    ListNode previousNode, currentNode;
    if (head == null) return;
    if (head.link == null) {
        head = null;
        return;
    }
}
```

예제 프로그램 (5)

```
else {  
    previousNode = head;  
    currentNode = head.link;  
    while (currentNode.link != null) {  
        previousNode = currentNode;  
        currentNode = currentNode.link;  
    }  
    previousNode.link = null;  
}  
} // end deleteLastNode()
```

```
public void printList() { // 프로그램 4.3  
    ListNode p;  
    System.out.print("(");  
    p = head;  
    while (p != null) {  
        System.out.print(p.data);  
        p = p.link;  
        if (p != null) {  
            System.out.print(", ");  
        }  
    }  
} // end printList()  
} // end LinkedList class
```

예제 프로그램 (6)

```
public class LinkedListTest {  
    public static void main(String args[]) {  
        LinkedList L = new LinkedList();  
        L.addLastNode("Kim");  
        L.addLastNode("Lee");  
        L.addLastNode("Park");  
        L.printList();    // (Kim, Lee, Park)가 프린트  
        L.addLastNode("Yoo");    // 원소 "Yoo"를 리스트 끝에 첨가  
        L.printList();    // (Kim, Lee, Park, Yoo)가 프린트  
        // 마지막 원소를 삭제하고 프린트  
        L.deleteLastNode();  
        L.printList();    // (Kim, Lee, Park)가 프린트  
        // 리스트 L을 역순으로 만들고 프린트  
        L.reverse();  
        L.printList();    // (Park, Lee, Kim)이 프린트  
    }  
} // end LinkedListTest class
```

Singly Linked List



ListNode.java



LinkedList.java



LinkedListTest.java



SinglyLinkedList.java

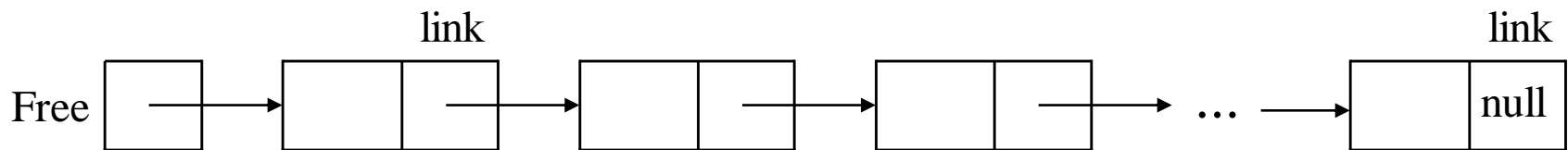


SinglyLinkedListTest.java

자유 공간 리스트

◆ 자유 공간 리스트 (free space list)

- 필요에 따라 요구한 노드를 할당할 수 있는 자유 메모리 풀
- 자유 공간 관리를 위해 연결리스트 구조를 이용하기 위해서는 초기에 사용할 수 있는 메모리를 연결 리스트로 만들어 놓아야 함
- 초기 자유 공간 리스트



- 노드 할당 요청이 오면 리스트 앞에서부터 공백 노드를 할당

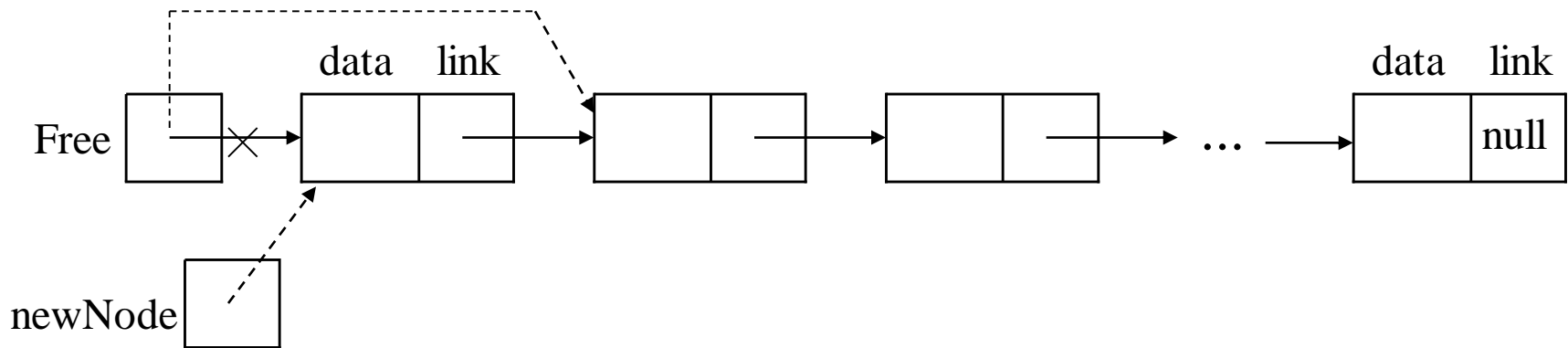
노드 할당 (1)

◆ 새로 노드를 할당하는 함수 (getNode)

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

노드 할당 (2)

◆ 노드를 할당한 뒤의 자유 공간 리스트



◆ 가용 공간이 모두 할당된 경우의 처리 방법

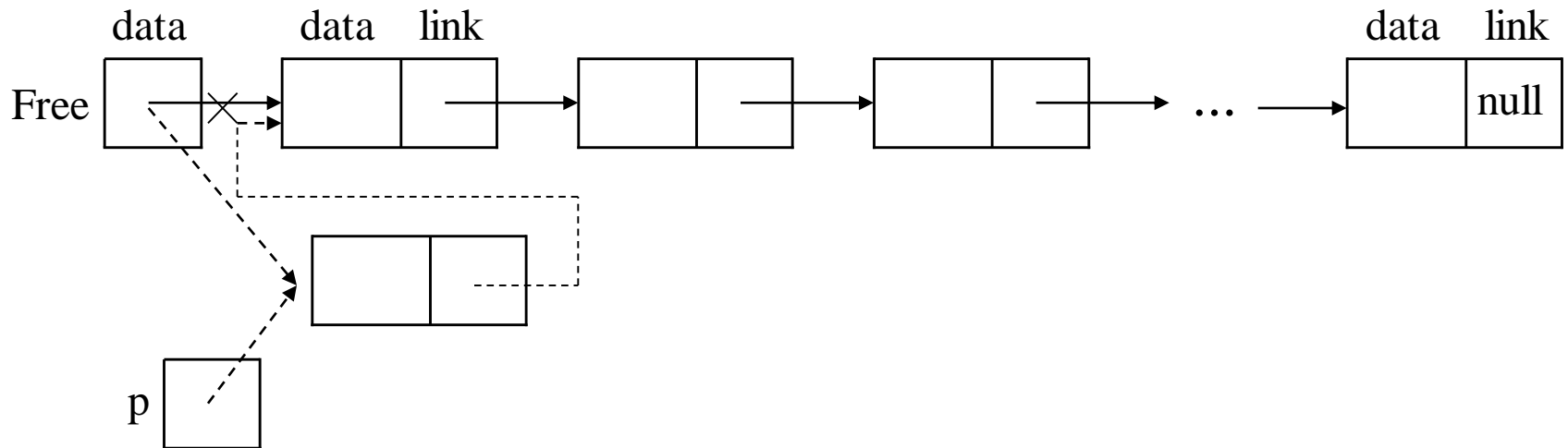
- 1. 프로그램이 더 이상 사용하지 않는 노드들을 자유 공간 리스트에 반환
- 2. 시스템이 자동적으로 모든 노드들을 검사하여 사용하지 않는 노드들을 전부 수집하여 자유 공간 리스트의 노드로 만든 뒤 다시 노드 할당 작업 재개 (garbage collection)

노드 반환

- ◆ 삭제된 노드를 자유 공간 리스트에 반환
 - 알고리즘

```
returnNode(p)
  // p는 삭제된 노드에 대한 포인터
  p.link ← Free;
  Free ← p;
end returnNode()
```

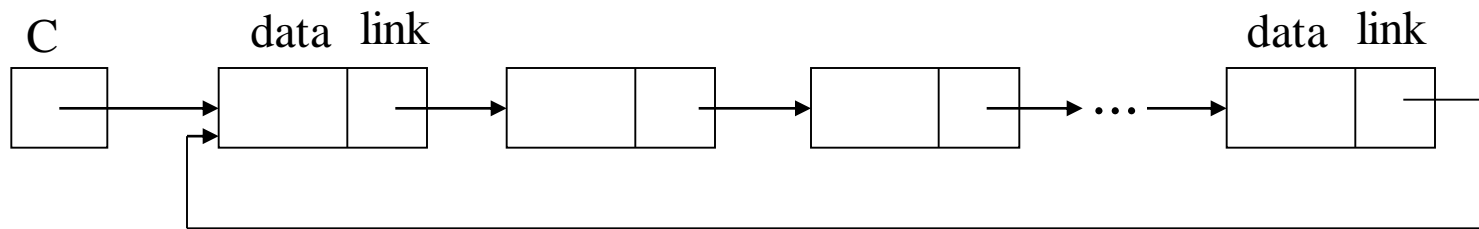
- ◆ 반환된 노드가 자유 공간 리스트에 삽입되는 과정



원형 연결 리스트

◆ 원형 연결 리스트 (circular linked list)

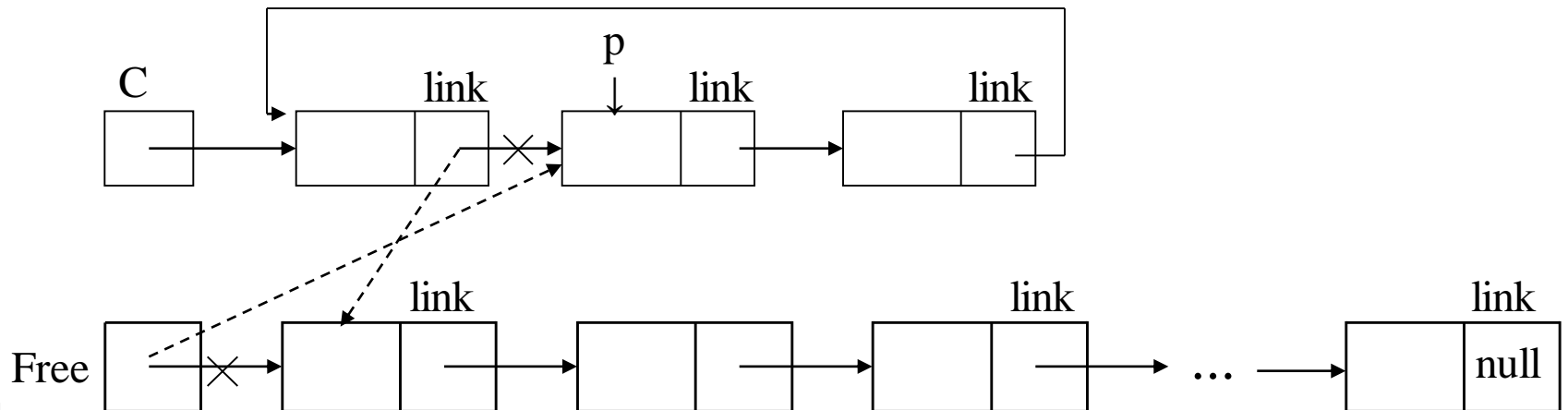
- 마지막 노드가 다시 첫 번째 노드를 가리키는 리스트
- 한 노드에서 다른 어떤 노드로도 접근할 수 있음
- 리스트 전체를 가용공간 리스트에 반환할 때 리스트의 길이에 관계없이 일정 시간에 반환할 수 있음
- 예



원형 연결 리스트 연산 (1)

◆ 원형 연결 리스트를 자유 공간 리스트에 반환

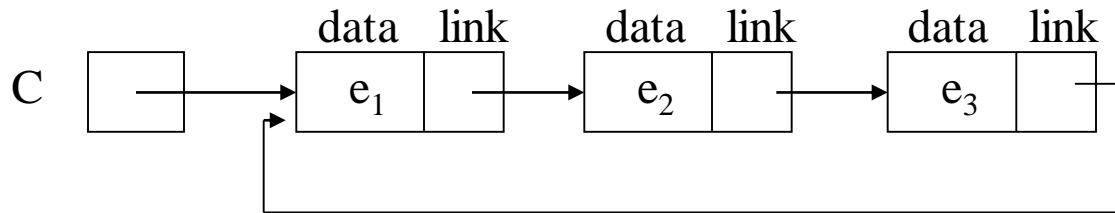
```
returnCList(C)
  // 원형 연결 리스트 C를 자유 공간 리스트에 반환
  if (C = null) then return;
  p ← C.link;
  C.link ← Free;
  Free ← p;
end returnCList()
```



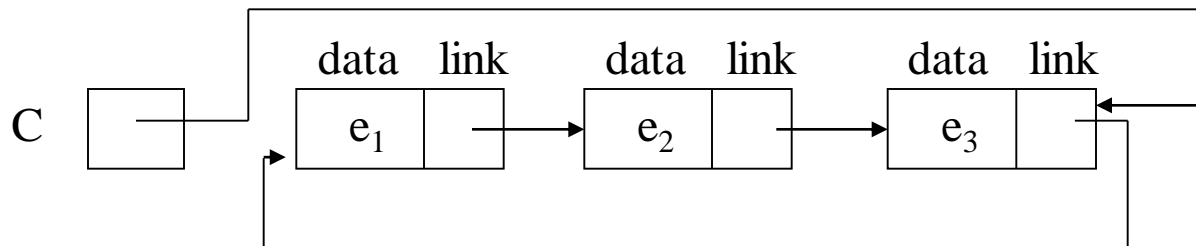
원형 연결 리스트 연산 (2)

◆ 원형 리스트에서의 노드 삽입

- 예) $CL = (e_1, e_2, e_3)$



- ◆ 첫 번째 노드로 새로운 노드 p 를 삽입할 때 마지막 노드의 **link** 필드를 변경시키기 위해 마지막 노드를 찾아야 함 (리스트 CL 의 길이만큼 순회)
- ◆ CL 을 첫 번째 노드가 아니고 마지막 노드를 지시하도록 하면 리스트의 길이에 상관없이 일정 시간에 노드 삽입 가능



원형 연결 리스트 연산 (3)

◆ 노드 삽입 알고리즘

```
insertFront(C, p)
    // C는 원형 리스트의 마지막 노드를 지시
    // p는 삽입할 노드를 지시
    if (C = null) then {
        C ← p;
        p.link ← C;
    }
    else {
        p.link ← C.link;
        C.link ← p;
    }
end insertFront()
```

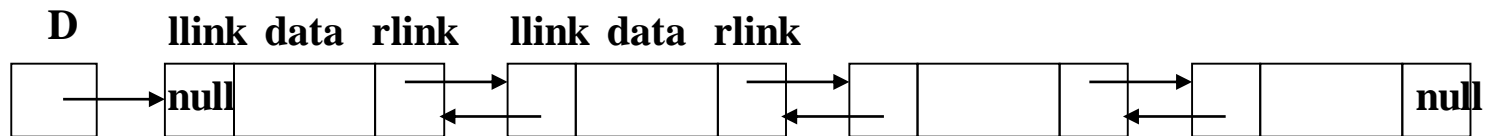
원형 연결 리스트 연산 (4)

◆ 리스트 길이 계산

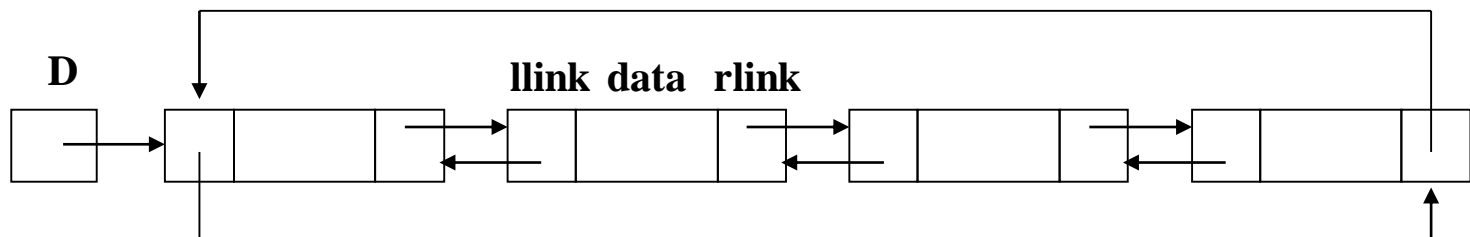
```
lengthC(C)
    // 원형 리스트 C의 길이 계산
    if (C = null) then return 0;
    length ← 1;
    p ← C.link;          // p는 순회 포인터
    while (p ≠ C) do {   // p가 처음 출발한 위치에 되돌아 왔는지 검사
        length ← length + 1;
        p ← p.link;
    }
    return length;
end lengthC()
```

이중 연결 리스트

- ◆ 단순 연결 리스트나 원형 연결 리스트의 문제점
 - 어떤 노드 p가 있을 때, 이 p의 선행자를 쉽게 찾을 수 없음
- ◆ 이중 연결 리스트 (doubly linked list)
 - 선행 노드를 검색하기 쉬움
 - 노드는 data, llink(왼쪽 링크), rlink(오른쪽 링크) 필드 가짐
 - 선형이 될 수도 있고 원형이 될 수도 있음
 - $p = p.llink.rlink = p.rlink.llink$



(a)



(b)

이중 연결 리스트 연산 (1)

◆ 이중 연결 리스트에서의 노드 삭제

```
deleteD(D, p)
  // D는 공백이 아닌 이중 연결 리스트
  // p는 삭제할 노드
  if (p = null) then return;
  p.llink.rlink ← p.rlink;
  p.rlink.llink ← p.llink;
end deleteD()
```

◆ 이중 연결 리스트에서의 노드 삽입

```
insertD(D, p, q)
  // D는 이중 연결 리스트이고
  // 노드 q를 노드 p 다음에 삽입
  q.llink ← p;
  q.rlink ← p.rlink;
  p.rlink.llink ← q;
  p.rlink ← q;
end insertD()
```

이중 연결 리스트 연산 (2)

- ◆ 이중 연결 리스트를 구현하기 위한 **DoubleListNode** 클래스

```
class DoubleListNode {  
    Object data;  
    DoubleListNode rlink;  
    DoubleListNode llink;  
}
```


헤더 노드 (1)

◆ 기존의 연결 리스트 처리 알고리즘

- 첫 번째 노드나 마지막 노드, 그리고 리스트가 공백인 경우를 예외적인 경우로 처리해야 함

◆ 헤더 노드 (header node) 추가

- 예외 경우를 제거하고 코드를 간단하게 하기 위한 방법
- 연결 리스트를 처리하는 데 필요한 정보를 저장
- 헤더 노드의 구조가 리스트의 노드 구조와 같을 필요는 없음
- 헤더 노드에는 리스트의 첫 번째 노드를 가리키는 포인터, 리스트의 길이, 참조 계수, 마지막 노드를 가리키는 포인터 등의 정보를 저장
- 객체지향 프로그래밍에서 유용
 - ◆ 공백 리스트도 head가 null인 객체로 나타낼 수 있음
 - ◆ 헤더 노드 없이 공백 리스트를 단순히 null 값으로만 표시하면 공백 리스트에 메소드들이 적용될 수 없음

헤더 노드 (2)

◆ 헤더 노드 기능을 갖는 연결 리스트를 Java로 구현

```
class ListNode {  
    String name;  
    ListNode link;  
}  
  
public class LinkedList {  
    private int length;    // 리스트의 노드 수  
    private ListNode head;  
        // 리스트의 첫 번째 노드에 대한 포인터  
    private ListNode tail;  
        // 리스트의 마지막 노드에 대한 포인터  
    public LinkedList() {    // 공백 리스트 생성  
        length = 0;  
        head = null;  
        tail = null;  
    }  
}
```

헤더 노드 (3)

◆ 헤더 노드 기능을 갖는 연결 리스트를 Java로 구현

```
public void addLastNode(String x) {  
    :  
}  
public void reverse() {  
    :  
}  
public void deleteLastNode() {  
    :  
}  
public void printList() {  
    :  
}  
  
// 기타 다른 메소드 정의  
}
```

헤더 노드 (4)

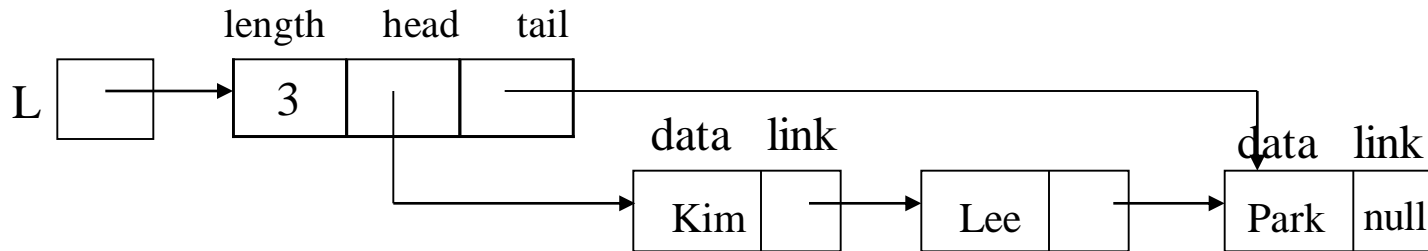
◆ LinkedList 클래스

- 각 LinkedList 객체
 - ◆ length, head, tail 필드 가짐
- 무인자 생성자 LinkedList()
 - ◆ 공백 LinkedList 객체를 생성
- addLastNode(), deleteLastNode() 메소드
 - ◆ 노드의 첨가나 삭제가 일어날 때마다 헤더의 length나 tail 값을 갱신시킬 명령문 포함

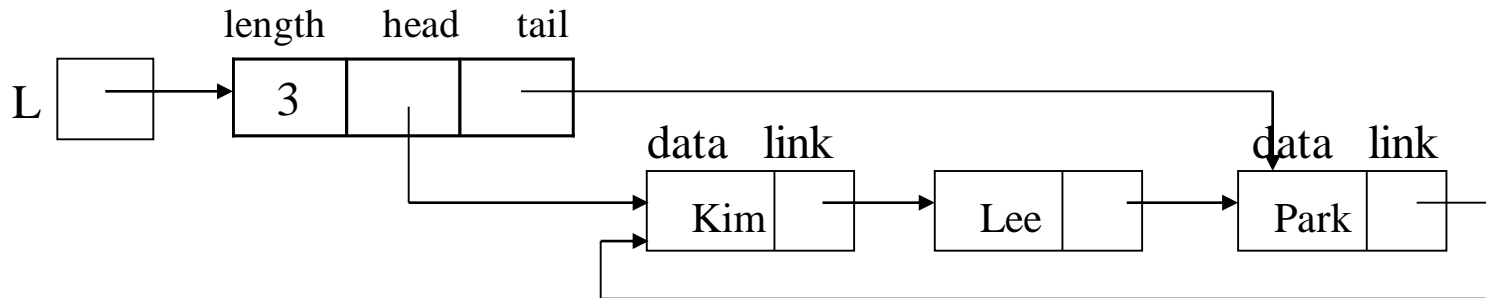
헤더 노드 (5)

◆ 헤더 노드를 가진 연결 리스트 표현

- 단순 연결 리스트



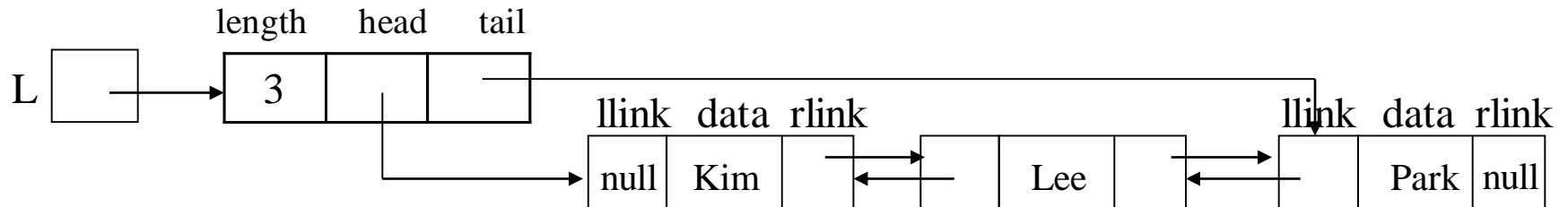
- 원형 연결 리스트



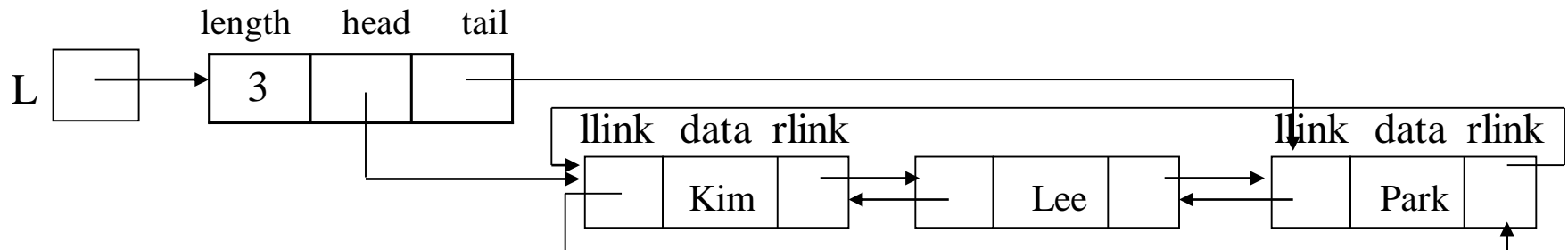
헤더 노드 (6)

◆ 헤더 노드를 가진 연결 리스트 표현

- 이중 연결 리스트



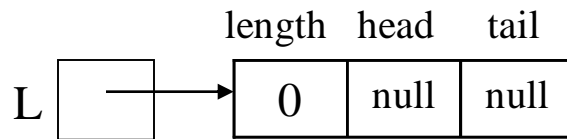
- 이중 원형 연결 리스트



헤더 노드 (7)

◆ 공백 리스트

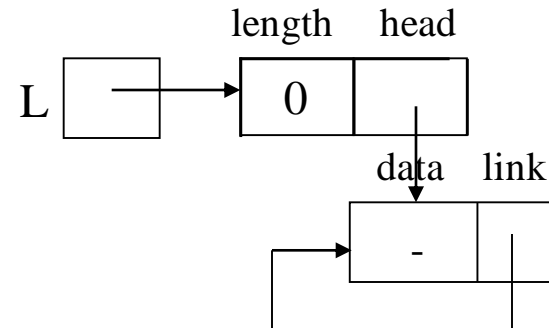
- length가 0이고 head가 null, tail이 null



◆ 공백 리스트를 표현하는 단순 연결 연형 리스트의 헤더 노드의 구조

- 생성자

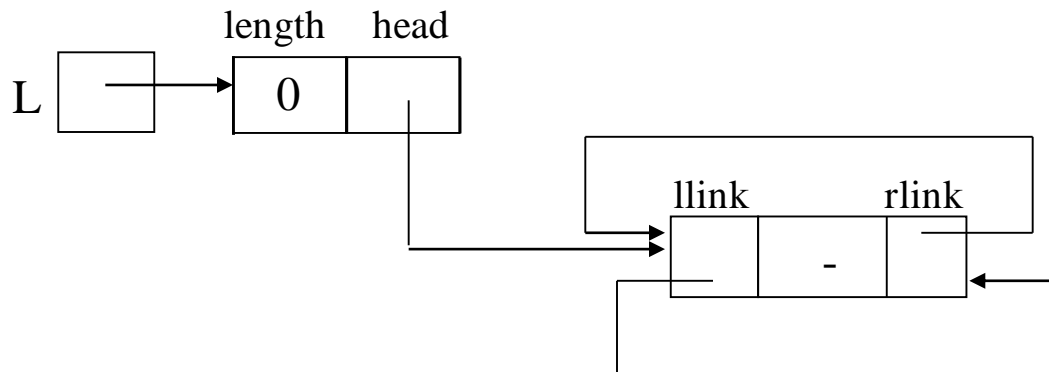
```
LinkedList() {  
    head = new ListNode();  
    head.link = head;  
    length = 0;  
}
```



헤더 노드 (8)

- ◆ 공백 리스트를 표현하는 이중 연결 원형 리스트의 헤더 노드의 구조
 - 생성자

```
DoubleLinkedList() {  
    head = new DoubleListNode();  
    head.rlink = head;  
    head.llink = head;  
    length = 0;  
}
```



다항식의 리스트 표현

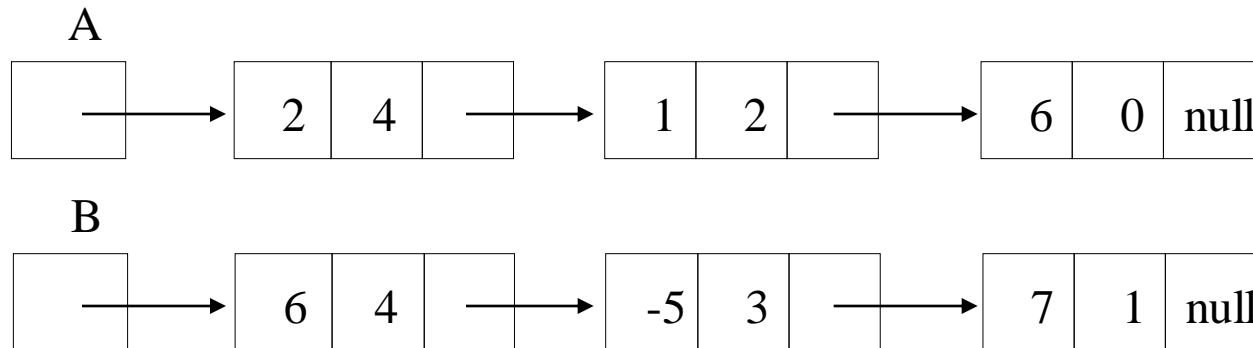
◆ 다항식의 표현

- 다항식은 일반적으로 0이 아닌 항들의 합으로 표현
- 다항식을 단순 연결 리스트로 표현 가능한데 각 항을 하나의 노드로 표현.
- 각 노드는 계수(coef)와 지수(exp) 그리고, 다음 항을 가리키는 링크(link) 필드로 구성

다항식 노드 :

coef	exp	link
------	-----	------

- 예) 다항식 $A(x) = 2x^4 + x^2 + 6$ 과 $B(x) = 6x^4 - 5x^3 + 7x$ 을 표현



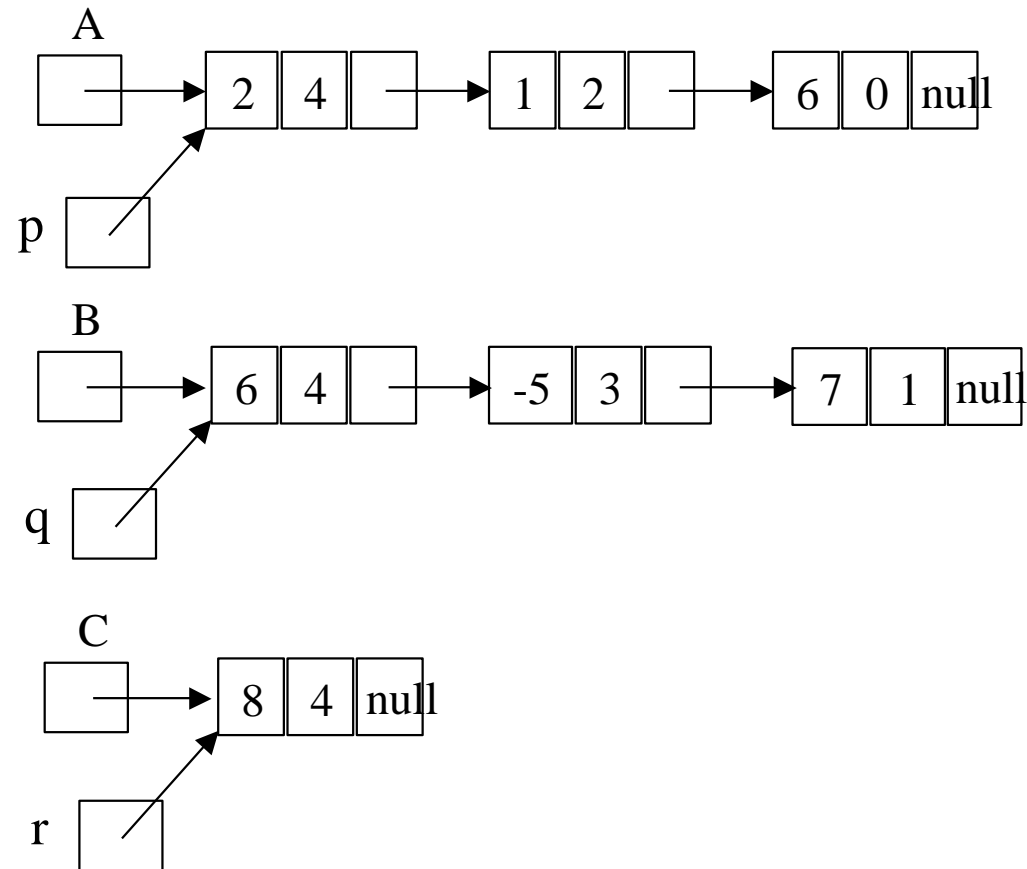
다항식의 덧셈 (1)

◆ 두 다항식을 더하는 연산 $C(x) \leftarrow A(x) + B(x)$

- 포인터 변수 p 와 q : 다항식 A 와 B 의 항들을 따라 순회하는 데 사용
- p 와 q 가 가리키는 항의 지수에 따라 3가지 경우에 따라 처리
 - ① $p.exp = q.exp$:
 - ◆ 두 계수를 더해서 0이 아니면 새로운 항을 만들어 결과 다항식 C 에 추가한다. 그리고 p 와 q 는 모두 다음 항으로 이동한다.
 - ② $p.exp < q.exp$:
 - ◆ q 가 지시하는 항을 새로운 항으로 복사하여 결과 다항식 C 에 추가한다. 그리고 q 만 다음 항으로 이동한다.
 - ③ $p.exp > q.exp$:
 - ◆ p 가 지시하는 항을 새로운 항으로 복사하여 결과 다항식 C 에 추가한다. 그리고 p 만 다음 항으로 이동한다.

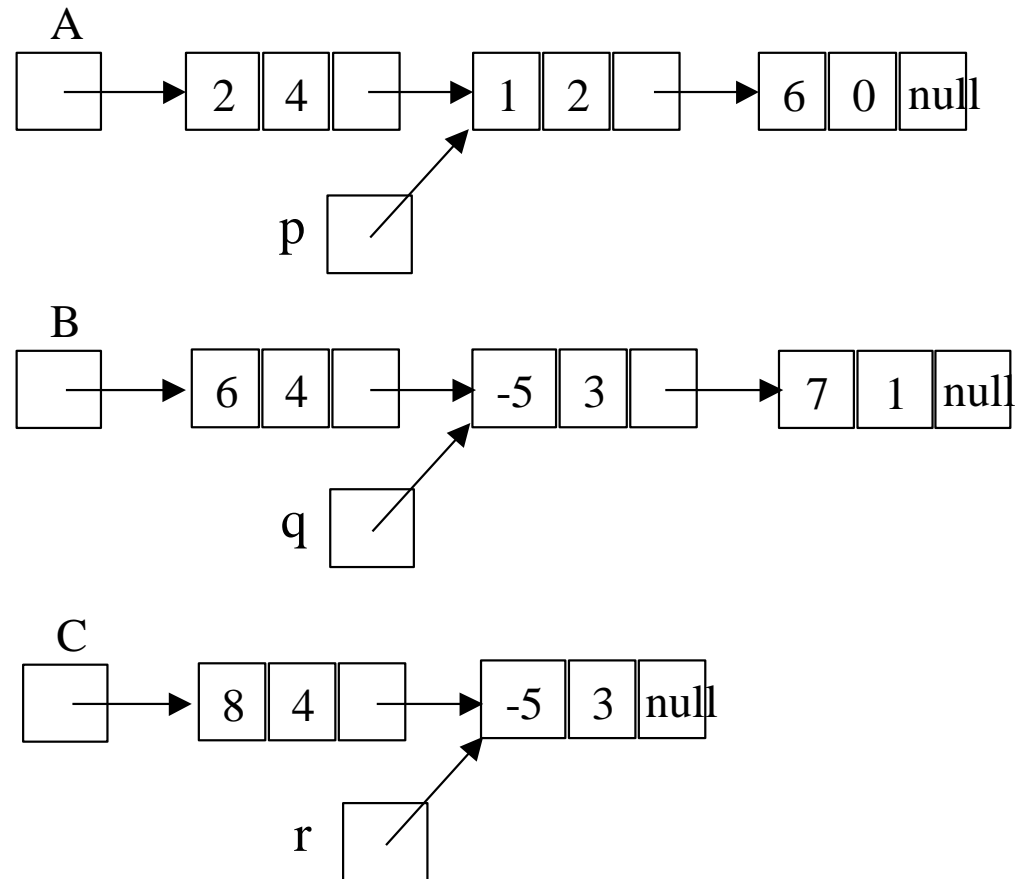
다항식의 덧셈 (2)

◆ $P.exp = Q.exp$



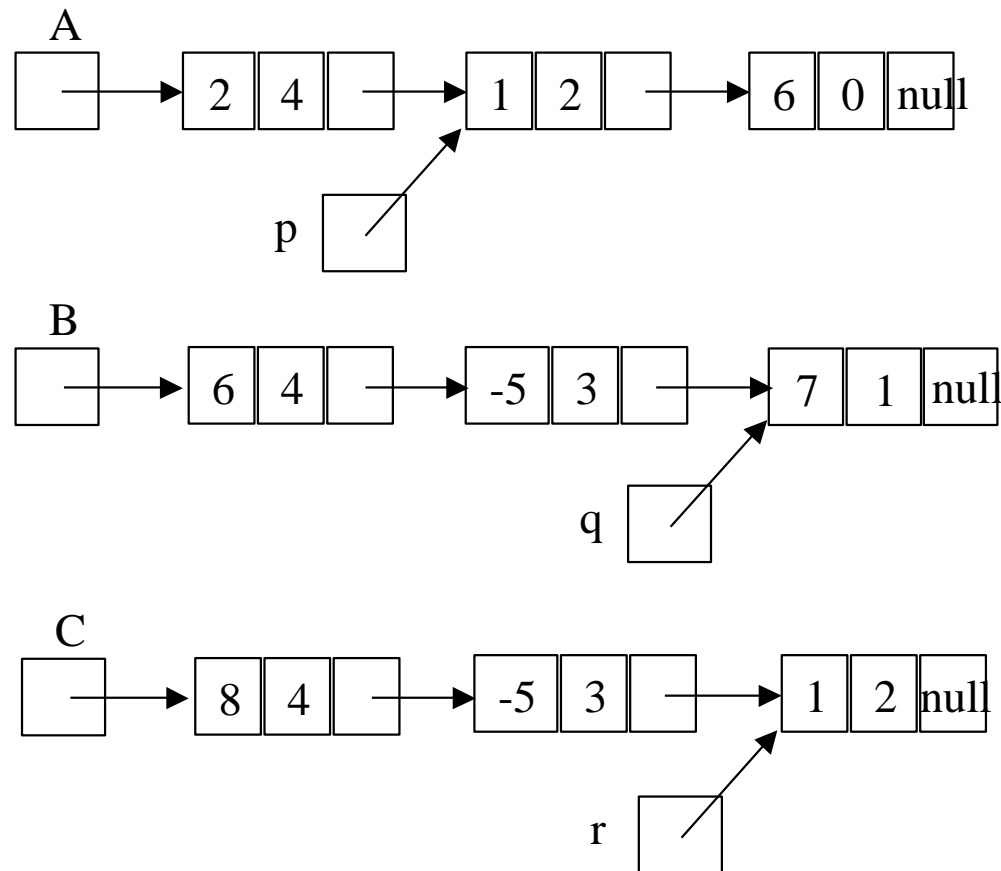
다항식의 덧셈 (3)

◆ $P.exp < Q.exp$:



다항식의 덧셈 (4)

◆ $P.exp > Q.exp$:



다항식의 덧셈 (5)

◆ 다항식에 새로운 항을 첨가

```
appendTerm(poly, c, e, last)
// c는 계수, e는 지수, last는 다항식 poly의 마지막 항을 가리키는 포인터
  newNode ← getNode();
  newNode.exp ← e;
  newNode.coef ← c;
  if (poly = null) then do {
    poly ← newNode;
    last ← newNode;
  }
  else {
    last.link ← newNode;
    last ← newNode;
  }
end appendTerm()
```

다항식 덧셈 알고리즘(1)

◆ 연결 리스트로 표현된 다항식의 덧셈

```
polyAdd(A, B)
```

```
// 단순 연결 리스트로 표현된 다항식 A와 B를 더하여 새로운 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 순회 포인터
```

```
  case {
```

```
    p.exp = q.exp :
```

```
      sum ← p.coef + q.coef;
```

```
      if (sum ≠ 0) then appendTerm(C, sum, p.exp, r);
```

```
      p ← p.link;
```

```
      q ← q.link;
```

```
    p.exp < q.exp :
```

```
      appendTerm(C, q.coef, q.exp, r);
```

```
      q ← q.link;
```

다항식 덧셈 알고리즘(2)

```
        else :    // p.exp > q.exp인 경우
            appendTerm(C, p.coef, p.exp, r);
            p ← p.link;
        } // end case
    } // end while
    while (p ≠ null) do { // A의 나머지 항들을 복사
        appendTerm(C, p.coef, p.exp, r);
        p ← p.link;
    }
    while (q ≠ null) do { // B의 나머지 항들을 복사
        appendTerm(C, q.coef, q.exp, r);
        q ← q.link;
    }
    r.link ← null;
    return C;
end polyAdd()
```


다항식 예



PolyNode.java



Polynomial.java

일반 리스트

◆ 일반 리스트 (general list)

- $n \geq 0$ 개의 원소 e_1, e_2, \dots, e_n 의 유한 순차
- 리스트의 원소가 원자(atom)일 뿐 아니라 리스트도 허용
- 리스트의 원소인 리스트를 서브리스트(sublist)라 함.
- 리스트는 $L = (e_1, e_2, \dots, e_n)$ 과 같이 표현함. L 은 리스트 이름이고 n 은 리스트의 원소수 즉 리스트의 길이가 됨
- $n \geq 1$ 인 경우 첫번째 원소 e_1 을 L 의 head 즉 $\text{head}(L)$ 라 하고, 첫번째 원소를 제외한 나머지 리스트 (e_2, \dots, e_n) 을 L 의 tail 즉 $\text{tail}(L)$ 이라 함
- 공백 리스트에 대해서는 head와 tail은 정의되지 않음
- 정의 속에 다시 리스트를 사용하고 있기 때문에 순환적 정의

일반 리스트 예

◆ 일반 리스트의 예

(1) $A=(a, (b, c))$: 길이가 2이고 첫 번째 원소는 a 이고 두 번째 원소는 서브리스트 (b, c) 이다.

- ◆ 리스트 A 에 대해 $\text{head}(A) = a$, $\text{tail}(A) = ((b, c))$
- ◆ $\text{tail}(A)$ 에 대해 $\text{head}(\text{tail}(A)) = (b, c)$, $\text{tail}(\text{tail}(A)) = ()$

(2) $B=(A, A, ())$: 길이가 3이고 처음 두 원소는 서브리스트 A 이고 세 번째 원소는 공백 리스트이다. 여기서 리스트 A 를 공유하고 있다.

- ◆ 리스트 B 에 대해 $\text{head}(B) = A$, $\text{tail}(B) = (A, ())$
- ◆ $\text{tail}(B)$ 에 대해 $\text{head}(\text{tail}(B)) = A$, $\text{tail}(\text{tail}(B)) = ()$

(3) $C=(a, C)$: 길이가 2인 순환리스트로서 두 번째 원소 C 는 무한리스트 $(a, (a, (a, \dots)))$ 에 대응된다.

(4) $D=()$: 길이가 0인 널(null) 또는 공백 리스트이다.

일반 리스트 표현 (1)

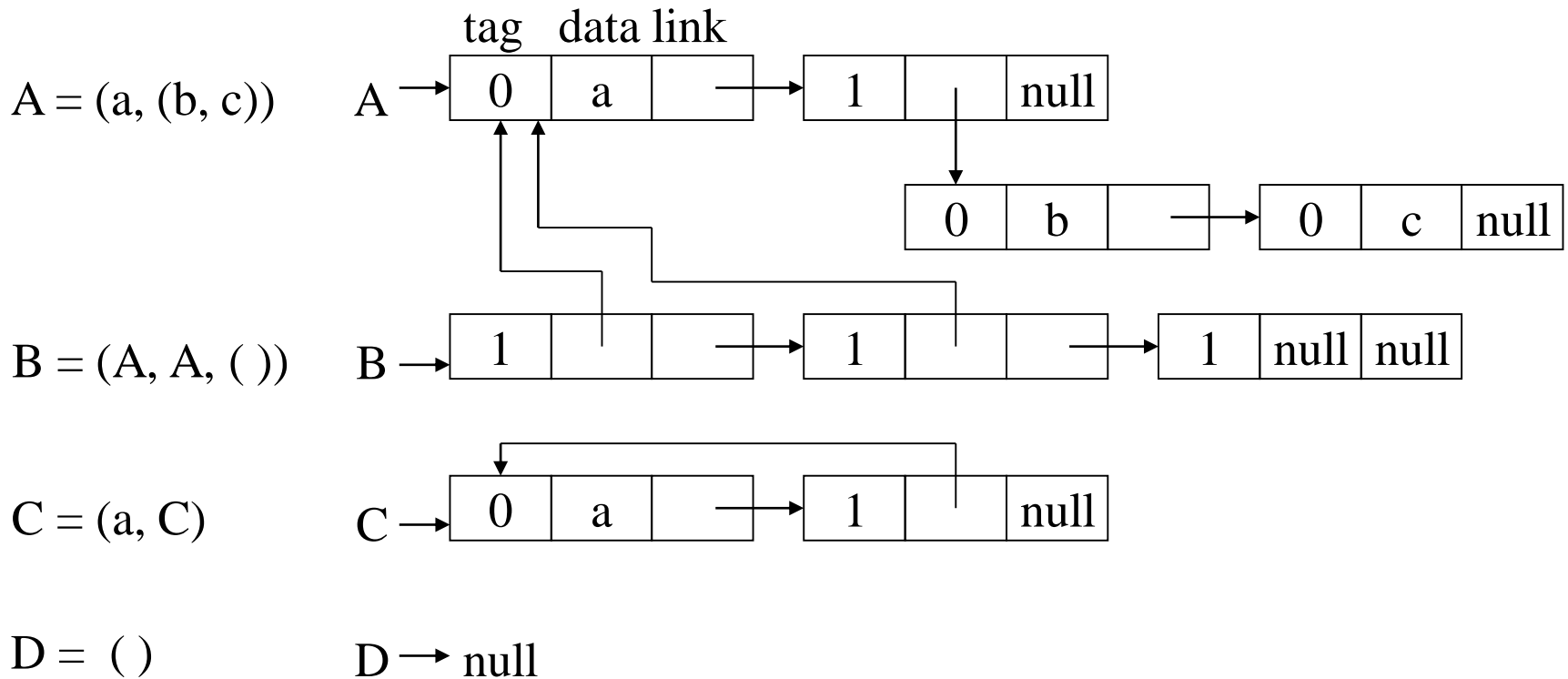
◆ 일반 리스트의 노드 구조

tag	data	link
-----	------	------

- data 필드 : 리스트의 head를 저장
 - ◆ head(L)이 원자인가 또는 서브리스트인가에 따라 원자값 또는 서브리스트의 포인터가 저장됨
- tag 필드 : data 필드 값이 원자인지 포인터 값인지를 표시
 - ◆ data 값이 원자값 : tag는 0
 - ◆ data 값이 서브리스트에 대한 포인터 값 : tag는 1
- link 필드 : 리스트의 tail에 대한 포인터를 저장

일반 리스트 표현 (2)

◆ 앞의 리스트의 예



공용 리스트와 참조 계수 (1)

◆ 서브리스트의 공용

- 저장 공간을 절약할 수 있는 장점
- 공용 리스트 앞에 노드를 새로 삽입하거나 삭제할 때 그 리스트를 공용하는 리스트도 변경되어야 함
- 앞의 일반 리스트 예에서
 - ◆ 리스트 A의 첫 번째 노드 삭제 : A를 가리키는 포인터 값을 A의 두 번째 노드를 가리키도록 변경
 - ◆ 새로운 노드를 리스트 A의 첫 번째 노드로 추가 : B의 포인터들을 이 새로 삽입된 첫 번째 노드를 가리키도록 변경
- 한 리스트가 어떤 리스트에 의해 참조되는지 알 수 없으므로 연산 시간이 많이 걸림

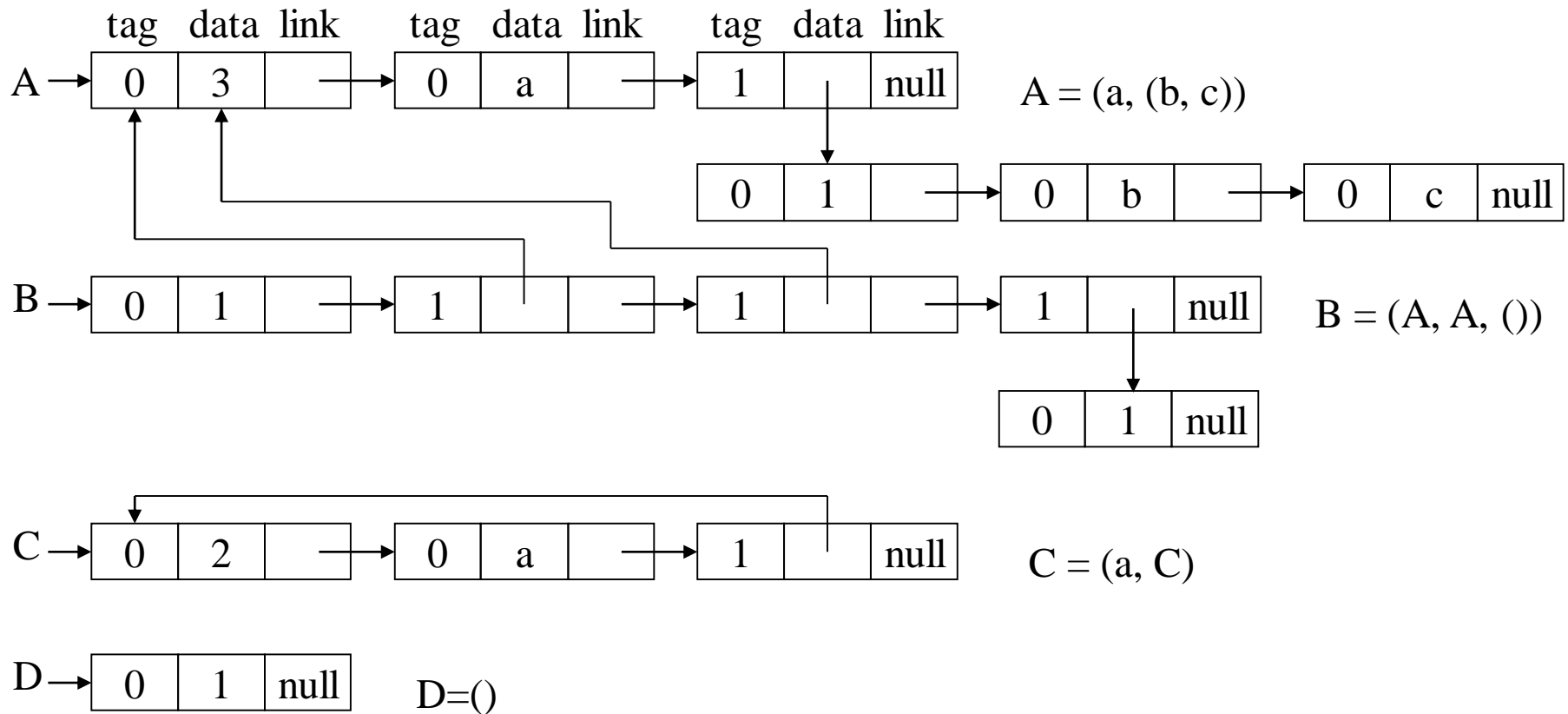
◆ 헤더 노드 추가로 문제 해결

- 공용 리스트를 가리킬 때 헤더 노드를 가리키게 함
- 공용 리스트 내부에서 노드 삽입과 삭제가 일어나더라도 포인터는 영향을 받지 않게 됨

공용 리스트와 참조 계수 (2)

◆ 헤더 노드가 첨가된 리스트 표현

- 헤더 노드의 data 필드는 참조 계수(자기를 참조하고 있는 포인터 수)를 저장하는 데 사용



공용 리스트와 참조 계수 (3)

◆ 참조 계수 (reference count)

- 각 리스트를 참조하고 있는 포인터수를 헤더 노드의 data 필드에 저장
- 리스트를 자유 공간 리스트에 반환할 것인가를 결정할 때, 리스트 헤더에 있는 참조 계수가 0인가만 확인하고 0일 때 반환하면 됨
- 예)
 - ◆ A.ref = 3 : A와, B의 두 곳에서 참조
 - ◆ B.ref = 1 : B만 참조
 - ◆ C.ref = 2 : C와, 리스트 자체 내에서 참조
 - ◆ D.ref = 1 : D만 참조

공용 리스트와 참조 계수 (4)

◆ 자유 공간 리스트로 리스트 반환

- p가 가리키는 리스트 삭제하려면 p의 참조 계수 1 감소
- p의 참조 계수 p.ref가 0이 되면 p의 노드들은 반환
- p의 서브리스트에 대해서도 순환적으로 수행

```
removeList(L)
```

```
    // 헤드 노드의 ref 필드는 참조 계수를 저장
```

```
    L.ref ← L.ref - 1; // 참조 계수를 1 감소시킴
```

```
    if (L.ref ≠ 0) then return;
```

```
    p ← L; // p는 순환 포인터
```

```
    while (p.link ≠ null) do {
```

```
        p ← p.link;
```

```
        if p.tag = 1 then removeList(p.data);
```

```
            // tag=1이면 서브리스트로 순환
```

```
    }
```

```
    p.link ← Free; // Free는 자유 공간 리스트
```

```
    Free ← L;
```

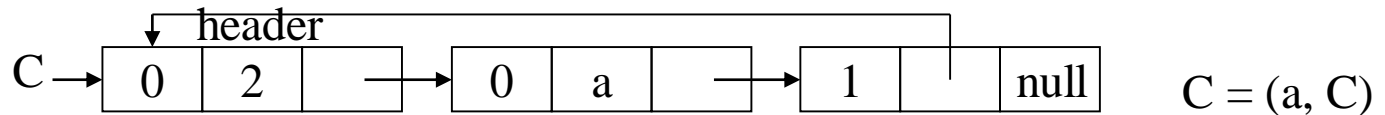
```
end removeList()
```

쓰레기 수집 (1)

◆ 참조 계수 사용의 한계

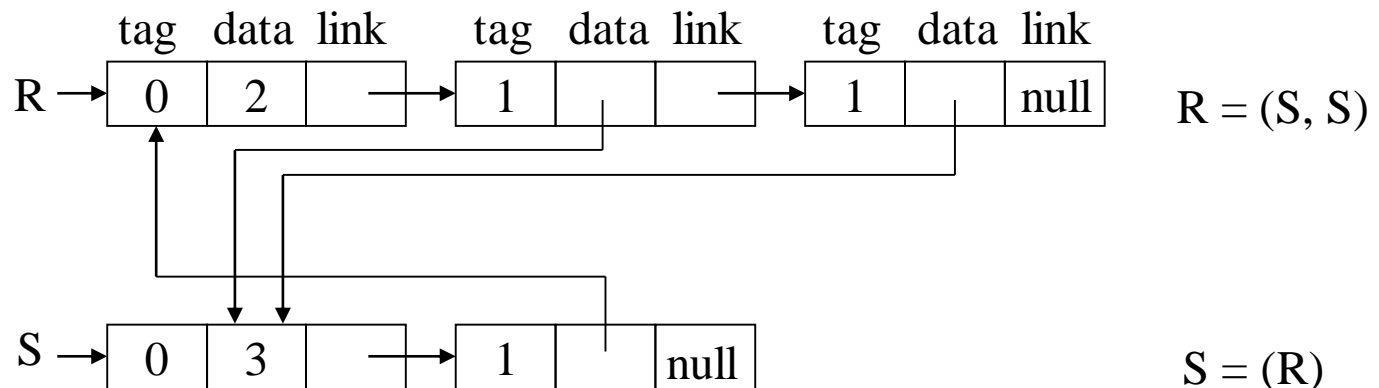
● 순환 리스트

- ◆ 반환되어야 될 리스트인데도 그 참조 계수가 결코 0이 되지 않음
- ◆ 예) `removeList(C)`: C의 참조 계수를 1로 만들지만 (0이 되지 않지만) 이 리스트는 다른 포인터나 리스트를 통해 접근이 불가능



● 간접 순환

- ◆ 예) `removeList(R)`과 `removeList(S)`가 실행된 뒤에 $R.ref = 1$, $S.ref = 2$ 가 되지만 더 이상 참조할 수 있는 것이 아님. 그러나 참조 계수가 0이 아니기 때문에 반환될 수 없음



쓰레기 수집 (2)

◆ 쓰레기 (garbage)

- 실제로 사용하고 있지 않으면서도 자유 공간 리스트에 반환될 수 없는 메모리
- 시스템 내에 쓰레기가 많이 생기면 가용공간 리스트가 고갈되어 프로그램 실행이 중지되는 경우 발생
- 자유 공간이 고갈되어 프로그램 실행을 더 이상 진행할 수 없는 경우 발생 가능

◆ 쓰레기 수집 (garbage collection)

- 사용하지 않는 모든 노드들을 수집하여 자유 공간 리스트에 반환시켜 시스템 운영 지속
- 모든 리스트 노드가 크기가 일정하다고 가정
- 각 노드에 추가로 할당된 markBit라는 특별한 비트는 0과 1만을 갖도록 하여 노드의 사용 여부(사용되지 않음 : 0, 사용중 : 1)를 표현

쓰레기 수집 (3)

◆ 쓰레기 수집 과정

1. 초기화 단계 (initialization)

- ◆ 메모리에 있는 모든 노드의 마크 비트를 0으로 설정하여 사용하지 않는 것으로 표시

2. 마크 단계 (marking phase)

- ◆ 현재 사용되고 있는 리스트 노드들을 식별해서 마크 비트를 1로 변경한 뒤 이 노드의 data 필드를 검사
- ◆ data 필드가 다른 리스트를 참조하면 이 필드에서부터 이 단계를 순환적으로 진행
- ◆ data 필드를 따라 처리를 끝낸 뒤에는 다시 link 필드를 따라 다음 노드를 처리 (이 때 mark bit가 1이면 그 노드를 통한 경로는 진행할 필요 없음)

3. 수집 단계 (collecting phase)

- ◆ 모든 노드의 mark bit를 검사해서 0으로 마크된 모든 노드들을 자유 공간 리스트에 반환

쓰레기 수집 (4)

◆ 쓰레기 수집 알고리즘

```
garbageCollection()
```

```
    // listNode는 markBit, tag, data, link로 구성
```

```
    // listNode들은 listNodeArray[]라는 노드의 배열로부터 할당되어진다고  
    가정
```

```
    // listNodeArray[]의 크기(listNode의 수)는 listNodeArraySize로 가정
```

```
    // 초기화 단계 : 모든 listNode들의 markBit을 0으로 설정
```

```
    for (i ← 0; i < listNodeArraySize; i ← i+1) do
```

```
        listNodeArray[i].markBit ← 0;
```

```
        // 마크 단계 : 사용중인 노드의 markBit을 모두 1로 표시
```

```
    for (i ← 0; i < numberOfPointers; i ← i+1) do
```

```
        markListNode(p[i]); // p[i]는 사용중인 포인터 변수
```

쓰레기 수집 (5)

```
// 수집 단계 : markBit이 0인 노드들을 Free 리스트에 연결
for (i ← 0; i < listNodeArraySize; i ← i+1) do {
    if (listNodeArray[0].markBit = 0) then {
        listNodeArray[i].link ← Free;
        Free ← listNodeArray[i];
    }
}

markListNode(p)
    // 사용중인 노드를 마크, p는 포인터 변수
    if ((p≠null) and (p.markBit = 0)) then
        p.markBit ← 1;
    if (p.tag = 1) then markListNode(p.data);
        // data 경로를 따라 markBit을 검사
    markListNode(p.link); // link 경로를 따라 markBit을 검사
end markListNode()
end garbageCollection()
```

일반 리스트를 위한 함수 (1)

◆ 리스트의 복사본 생성

```
copyList(L)
  // L은 비 순환 리스트로서 공용 서브 리스트가 없음
  // L과 똑같은 리스트 p를 만들어 그 포인터를 반환
  p ← null;
  if (L ≠ null) then {
    if (L.tag = 0) then q ← L.data; // 원자 값을 저장
    else q ← copyList(L.data); // 순환 호출
    r ← copyList(L.link); // tail(L)을 복사
    p ← getNode(); // 새로운 노드 생성
    p.data ← q;
    p.link ← r; // head와 tail을 결합
    p.tag ← L.tag;
  }
  return p;
end copyList()
```

일반 리스트를 위한 함수 (2)

◆ 두 리스트의 동일성 검사

- 구조가 같고 대응되는 필드의 데이터 값이 같아야 동일

```
equalList(S, T)
    // S와 T는 비 순환 리스트, 각 노드는 tag, data, link 필드로 구성
    // S와 T가 똑 같으면 true, 아니면 false를 반환
    b ← false;
    case {
        S = null and T = null : b ← true;
        S ≠ null and T ≠ null :
            if (S.tag = T.tag) then {
                if (S.tag = 0) then b ← (S.data = T.data);
                else b ← equalList(S.data, T.data);
                if (b) then b ← equalList(S.link, T.link);
            }
    }
    return b;
end equalList()
```


일반 리스트를 위한 함수 (3)

◆ 리스트의 깊이 계산

```
depthList(L)
    // L은 비 순환 리스트, 노드는 tag, data, link로 구성
    // 리스트 L의 깊이를 반환
    max ← 0;
    if (L = null) then return(max); // 공백 리스트의 깊이는 0
    p ← L;
    while (p ≠ null) do { // p는 순환 포인터
        if (p.tag = 0) then d ← 0;
        else d ← depthList(p.data); // 순환
        if (d > max) then max ← d; // 새로운 max
        p ← p.link;
    }
    return max+1;
end depthList()
```

Java에서의 일반 리스트 구현 (1)

◆ ListNode 클래스

- 노드에 어떤 객체도 저장할 수 있도록 data 필드를 Object로 지정

◆ GenList 클래스

- 연결 리스트의 첫 번째 ListNode에 대한 참조를 저장할 head 필드를 포함한 헤더 노드를 가짐

◆ 프로그램 4.6

```
class ListNode {  
    Object data; // 모든 Java 객체가 data 값이 될 수 있다.  
    ListNode link;  
    public ListNode() {  
        data = link = null;  
    }  
}
```

Java에서의 일반 리스트 구현 (2)

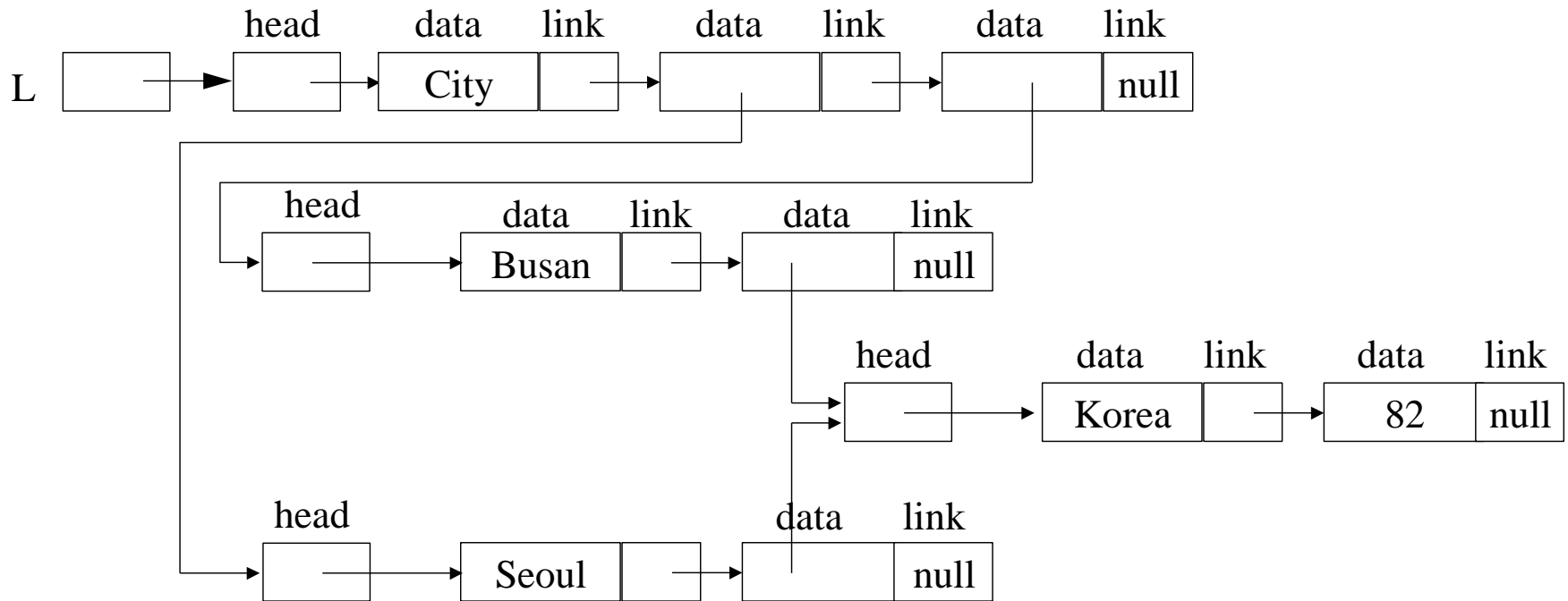
```
class GenList {  
    private ListNode head;  
        // 리스트의 첫 번째 ListNode에 대한 참조를 저장  
  
    void insertData (Object x) {  
        // 리스트 head 다음에 새로운 ListNode를 삽입  
        ListNode newNode = new ListNode();  
        newNode.data = x;  
        newNode.link = head;  
        head = newNode;  
    } // insertData()  
}
```

Java에서의 일반 리스트 구현 (3)

```
void printGL() {  
    // 일반 리스트를 프린트  
    System.out.print("(");  
    ListNode p = head;  
    while ( p != null ) {    // 공백 리스트가 아닌 경우  
        if (p.data instanceof GenList) {  
            // data값이 GenList 참조값인가를 검사  
            ((GenList)p.data).printGL(); //서브리스트를 순환적으로 프린트  
        } else {  
            System.out.print(p.data);  
        }  
        if ((p = p.link) != null) {  
            System.out.print(", ");  
        }  
    } // while  
    System.out.print(")");  
} // printGL()  
} // end GenList class
```

Java에서의 일반 리스트 구현 (4)

◆ 공용 서브 리스트를 가진 일반리스트



$L = (\text{City}, (\text{Seoul}, (\text{Korea}, 82)), (\text{Busan}, (\text{Korea}, 82)))$

Java에서의 일반 리스트 구현 (5)

◆ 프로그램 4.7 (일반 리스트 생성 및 프린트)

```
import java.io.*;

public class GenListPrint {
    public static void main(String[] args) {
        GenList p = new GenList();
        p.insertData(new Integer(82));
        p.insertData("Korea");
        GenList q = new GenList();
        q.insertData(p);
        q.insertData("Seoul");
        GenList r = new GenList();
        r.insertData("Busan");
        GenList L = new GenList();
```

Java에서의 일반 리스트 구현 (6)

```
L.insertData(r);  
    L.insertData(q);  
    L.insertData("City");  
    L.printGL();  
    System.out.println();  
} // end main()  
} // end GenListPrint class  
// <프로그램 4.6을 여기에 삽입>
```

```
public GenList copy() { // 공용 서브리스트가 없는 경우.
```

```
    GenList gn = new GenList();
```

```
    gn.head = theCopy(head);
```

```
    return gn;
```

```
}
```

```
private ListNode theCopy(ListNode h) {
```

```
    ListNode p = null;
```

```
    Object q;
```

```
    ListNode r;
```

```
    if (h != null) {
```

```
        if (!(h.data instanceof GenList)) q = h.data;
```

```
        else q = ((GenList)h.data).copy();
```

```
        r = theCopy(h.link);
```

```
        p = new ListNode();
```

```
        p.data = q;
```

```
        p.link = r;
```

```
    }
```

```
    return p;
```

```
} © DBLAB, SNU
```



```
public boolean equal(GenList T) {  
    return theEqual(this.head, T.head);  
}
```

```
private boolean theEqual(ListNode s, ListNode t) {  
    if (s == null && t == null) return true;  
    if (s == null && t != null) { return false;}  
    if (s != null && t == null) { return false;}  
    if (s.data instanceof GenList && t.data instanceof GenList) {  
        if (((GenList)s.data).equal((GenList)t.data)) return theEqual(s.link,  
t.link);  
        else { return false; }  
    } else if (!(s.data instanceof GenList) && !(t.data instanceof GenList))  
{  
        if (s.data.equals(t.data)) return theEqual(s.link, t.link);  
        else {return false; }  
    } else {return false;}  
}
```

```
public int depth() {  
    return theDepth(head);  
}
```

```
private int theDepth(ListNode h) {  
    int max = 0;  
    int d;  
    ListNode p;  
    if (h == null) return 0;  
    p = h;  
    while (p != null) {  
        if (p.data instanceof GenList) {  
            d = ((GenList)p.data).depth();  
        } else {  
            d = 0;  
        }  
        if (d > max) max = d;  
        p = p.link;  
    }  
    return max+1;  
}
```