# 4장 연결 데이타 표현

# 순서

# 4.1 노드와 포인터

# 순차 표현

- 장점
  - 표현이 간단함
  - 원소의 접근이 빠름
    - 빠른 임의 접근
- 단점
  - 원소의 삽입과 삭제가 어렵고 시간이 많이 걸림
  - 저장공간의 낭비와 비효율성
  - 예) L=(Cho, Kim, Lee, Park, Yoo)

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

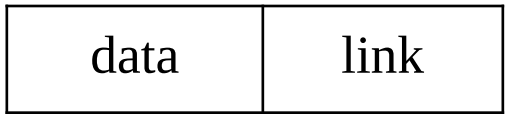
"Han"을 삽입

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

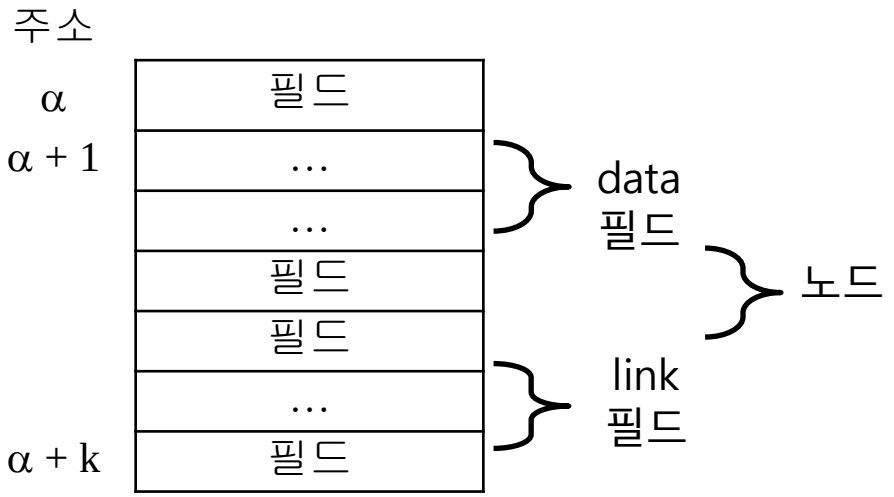
# 연결 표현(linked representation)

- 비순차 표현(non-sequential representation)
  - 원소의 물리적 순서가 리스트의 논리적 순서와 일치할 필요가 없음
  - 다음 원소에 대한 주소 저장
  - 노드 : <원소, 주소> 쌍의 구조

- 노드 (node)
  - 데이터(data) 필드
  - 링크(link) 필드



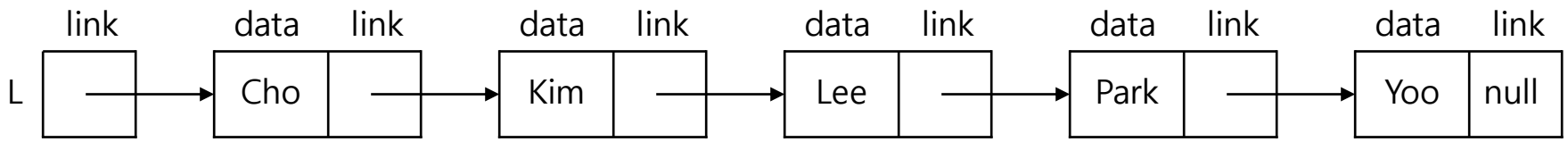

논리적 노드 구조



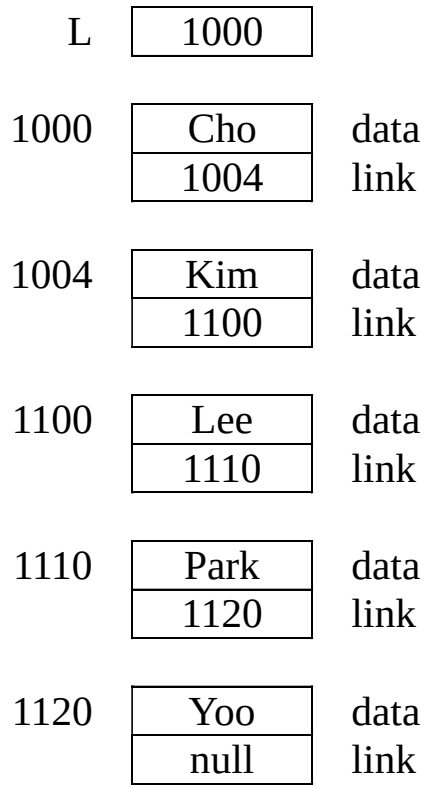

물리적 노드 구조

# 연결 리스트 (linked list)

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



# 연결 리스트 표현




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

```
ListNode L, p1, p2, p3, p4, p5;
p1 = new ListNode();
p2 = new ListNode();
p3 = new ListNode();
p4 = new ListNode();
p5 = new ListNode();

p1.data = "Cho";
p2.data = "Kim";
p3.data = "Lee";
p4.data = "Park";
p5.data = "Yoo";

p1.link = p2;
p2.link = p3;
p3.link = p4;
p4.link = p5;
p5.link = null;

L = p1;
```

# 4.2 Java의 참조 변수

# Java의 참조 변수

- 참조 변수 (reference variable)

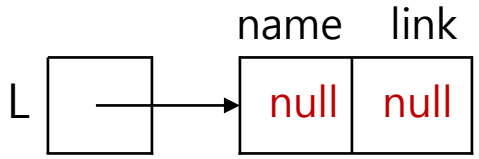
- Point 클래스에서의 생성자의 역할

```
class Point {
    int x;
    int y;

    public Point() {
        x = 0;
        y = 0;
    }

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

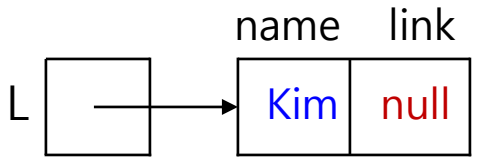
```
Point p1 = new Point();
Point p2 = new Point(0, 8);
```

# 리스트 생성의 예 (1)

1. L을 ListNode 타입의 변수로 선언하고 하나의 새로운 ListNode로 초기화
   - `ListNode L = new ListNode();`
   - null로 초기화

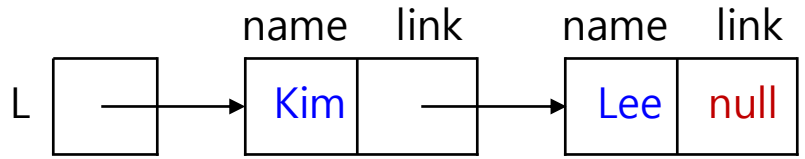



2. 노드 L의 name 필드에 "Kim"을 지정하고 link 필드에는 null을 지정
   - `L.name = "Kim";`
   - `L.link = null;`

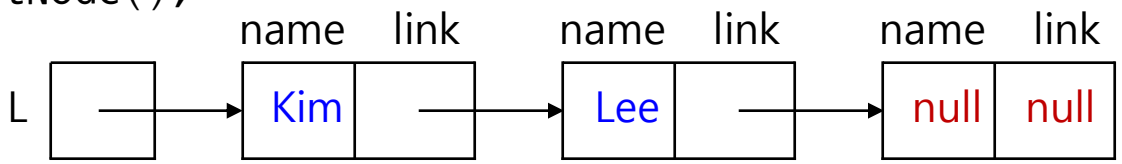


# 리스트 생성의 예 (2)

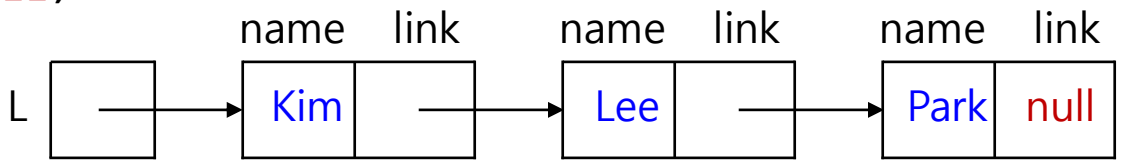
3. 리스트 L에 "Lee"와 "Park"에 대한 두 개의 새로운 노드를 차례로 첨가
   - `L.link = new ListNode();`
   - `L.link.name = "Lee";`




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



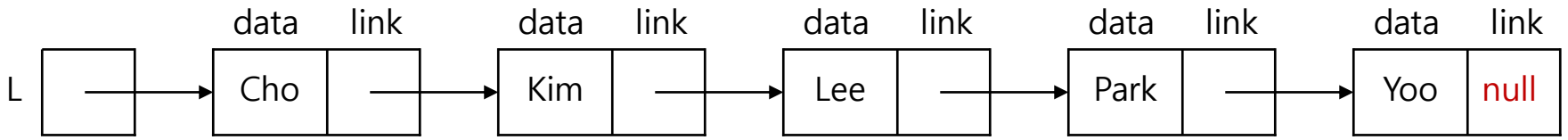

   - `L.link.link.name = "Park";`
   - `L.link.link.link = null;`



# 4.3 단순 연결 리스트

# 단순 연결 리스트 (singly linked list)

- 단순 연결 리스트
  - 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



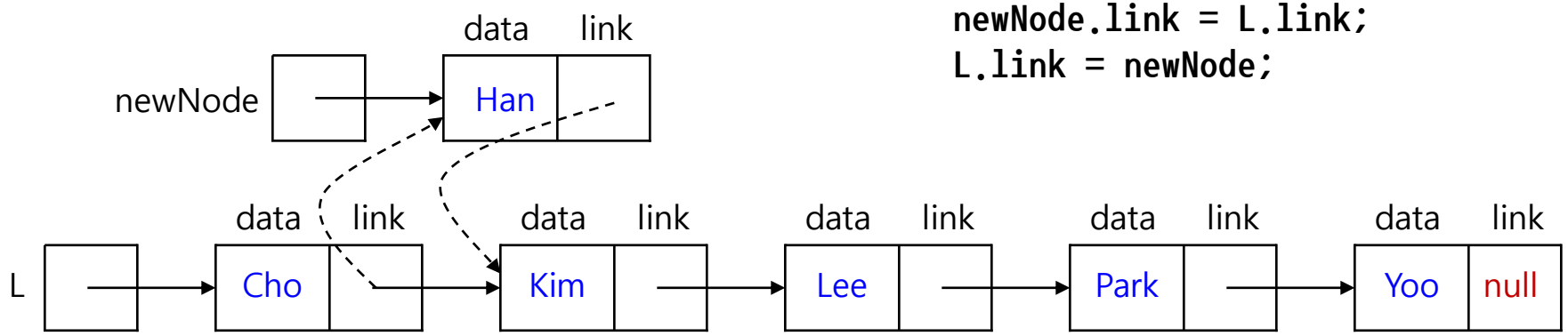
# 원소의 삽입

- 원소 삽입 알고리즘

  예) 리스트 L에 원소 "Han"을 "Cho"와 "Kim" 사이에 삽입

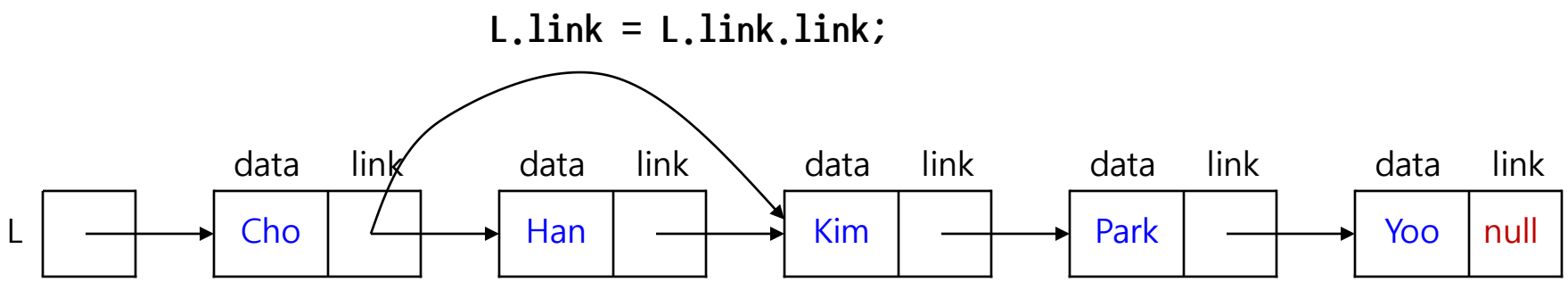
  1. 공백 노드를 생성하여 newNode라는 변수로 가리키게 함
  2. newNode의 data 필드에 "Han"을 저장
  3. "Cho"를 저장하고 있는 노드의 link 필드 값을 newNode의 link 필드에 저장
  4. "Cho"를 저장한 노드의 link에 newNode의 포인터 값을 저장

```
ListNode newNode = new ListNode();
newNode.data = "Han";
newNode.link = L.link;
L.link = newNode;
```



# 원소의 삭제

- "Han" 노드 삭제
  1. 원소 "Han"이 들어 있는 노드의 선행자를 찾음 ("Cho"가 들어있는 노드)
  2. 이 선행자의 link에 "Han"이 들어있는 노드의 link 값을 저장




  - "Han"노드는 쓰레기(garbage)가 됨

# 메모리의 획득과 반납

- 연결 리스트가 필요로 하는 두 가지 연산
  - 공백 노드 획득
  - 사용하지 않는 노드는 다시 반납

- 자유 공간 리스트 (free space list) 가 있는 경우
  - 미리 노드를 많이 만들어 둔 경우
  - `getNode()`
    - 새로운 공백 노드를 자유 공간 리스트로부터 할당 받아 주소 반환
  - `returnNode(p)`
    - 포인터 변수 p가 지시하는 노드를 자유 공간 리스트에 반환

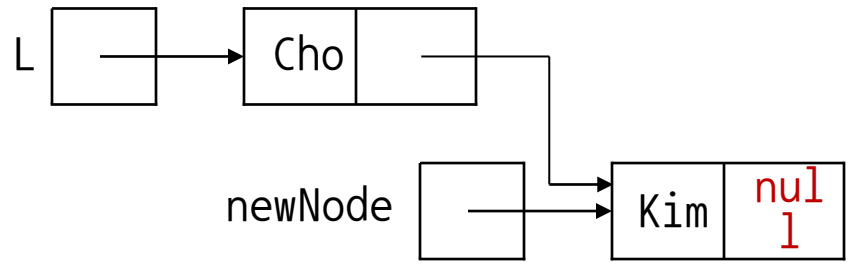
# 리스트 생성 알고리즘 (1)

- 앞의 두 함수를 이용한 리스트 생성 알고리즘

```
newNode ← getNode();          // 첫번째 공백 노드를 할당 받아
                              // newNode가 가리키도록 함
newNode.data ← "Cho";         // 원소 값을 저장

L ← newNode;                  // 리스트 L을 만듦

newNode ← getNode();          // 두 번째 공백 노드를 획득

newNode.data ← 'Kim';         // 두 번째 노드에 원소 값을 저장

newNode.link ← null;          //포인터 값 null을 저장

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

# 리스트 생성 알고리즘 (2)

- 점 표기식 만으로 리스트를 작성한 경우




Pseudo code

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

Java

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

```
L = new ListNode();
L.data = "Cho";
L.link = new ListNode();
L.link.data = “Kim”;
L.link.link = null;
```

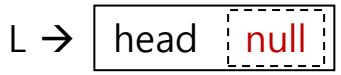
```
struct ListNode {
    char *data;
    struct ListNode *link;
};
```

C

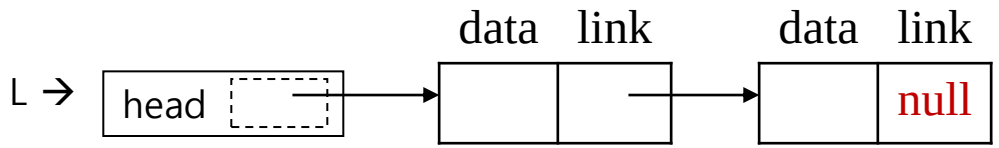
```
L = malloc(struct ListNode);
L->data = "Cho";
L->link = malloc(struct ListNode);
L->link->data = "Kim";
L->link->link = null;
```

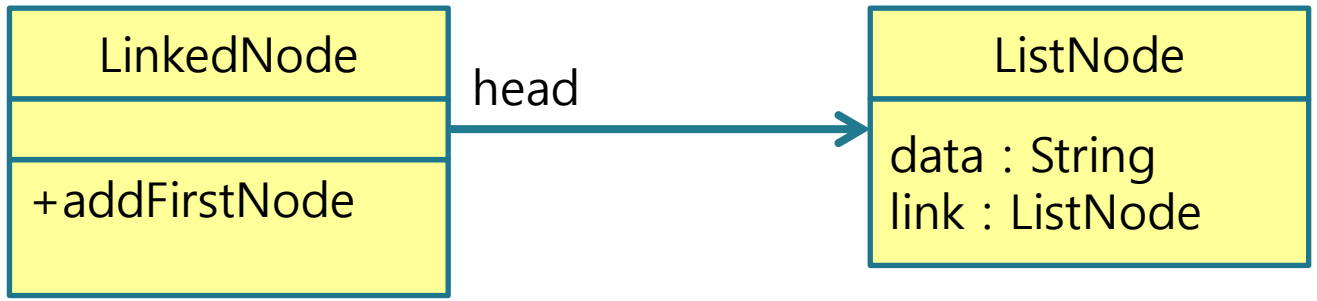
# 예제 프로그램

- LinkedList 생성의 예
  - `LinkedList L = new LinkedList();`




  - L에 노드가 연결되어 있는 경우



# ListNode

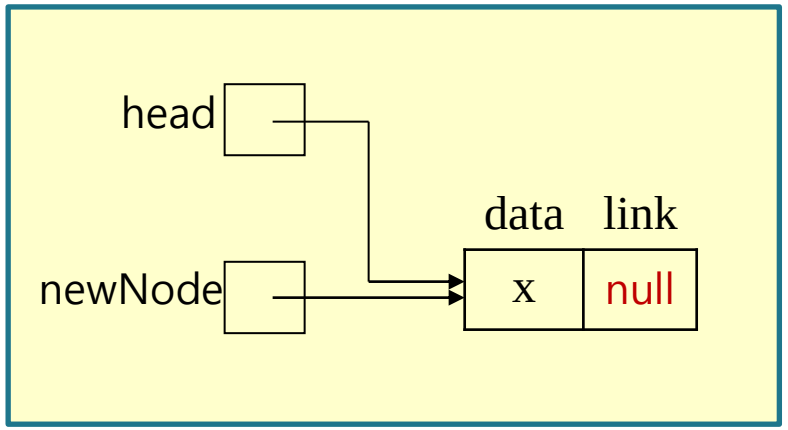
```
public class ListNode {
    String  data;
    ListNode  link;

    public ListNode() {
        data = null;
        link = null;
    }

    public ListNode(String val) {
        data = val;
        link = null;
    }

    public ListNode(String val, ListNode p) {
        data = val;
        link = p;
    }
}
```

# 원소를 첫 번째 노드로 삽입 (p.148)

```
public class LinkedList{

    private ListNode  head;

    /**
     * 리스트의 맨 앞에 원소 x를 삽입
     */
    public void addFirstNode(String x) {
        ListNode newNode = new ListNode();
        newNode.data = x;
        newNode.link = head;
        head = newNode;
    }
```

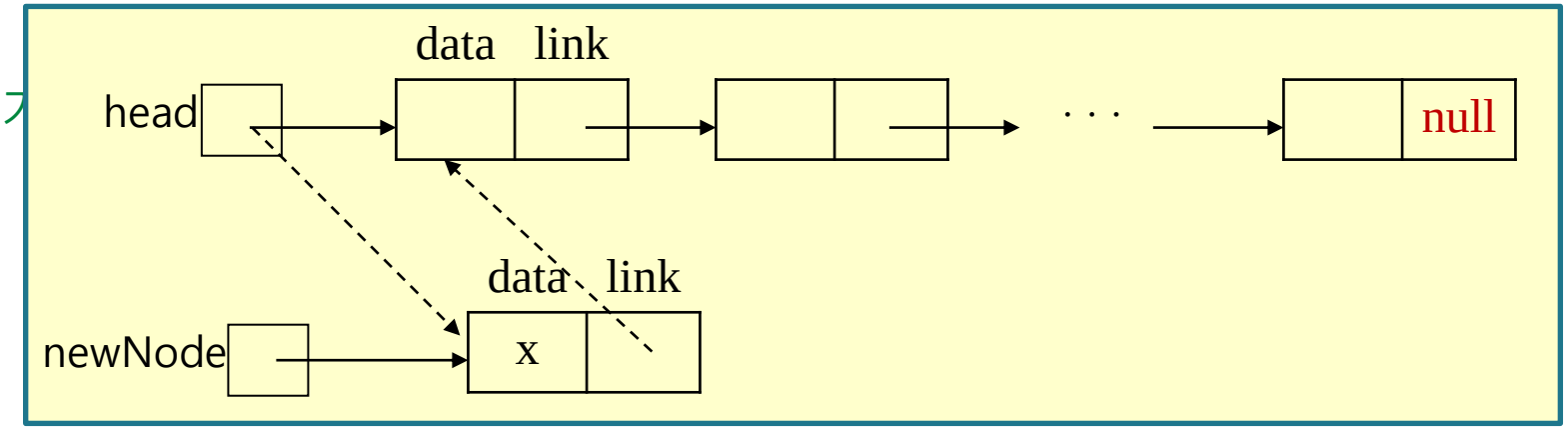
# p가 가리키는 노드 다음에 원소 x 삽입 (1)

```
/**
 * 리스트에서 p가 가리키는 노드 다음에 원소 x를 삽입
 * 알고리즘 4.1의 Java 구현
 */
public void insertNode(ListNode p, String x) {
    ListNode newNode = new ListNode();
    newNode.data = x;

    if (head == null) {       // 공백 리스트인 경우
        head = newNode;
        newNode.link = null;
    } else if (p == null) {   // p가 null이면 리스트의 첫 번째 노드로 삽입
        newNode.link = head;
        head = newNode;
    } else {                  // p가 가리키는 노드의 다음 노드로 삽입
        newNode.link = p.link;
        p.link = newNode;
    }
}
```



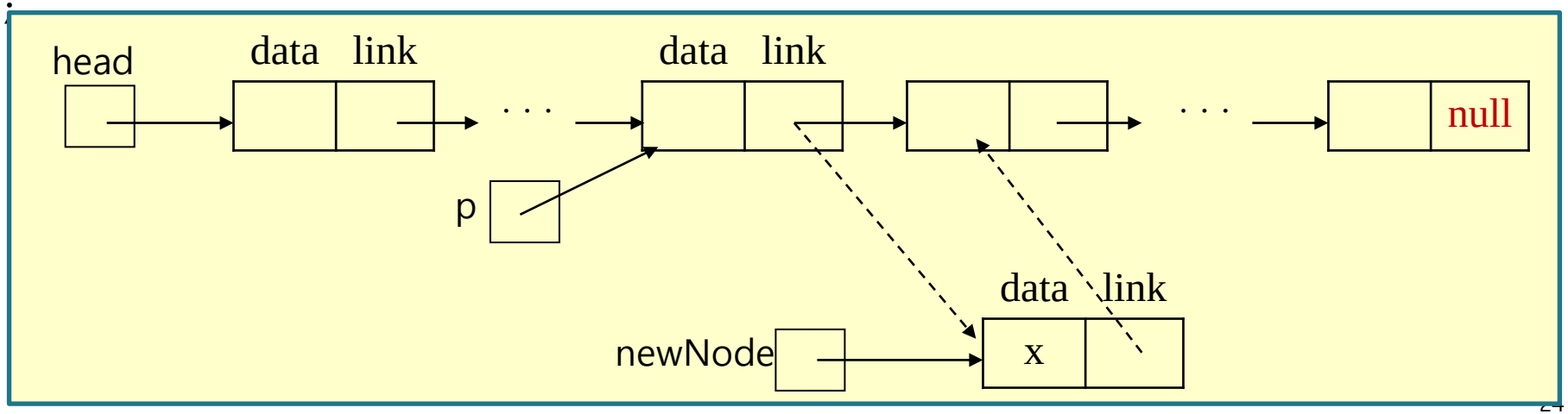
# p가 가리키는 노드 다음에 원소 x 삽입 (2)

```
/**
 * 리스트에서 p가 가리키는 노드 다음에 원소 x를 삽입
 * 알고리즘 4.1의 Java 구현
 */
public void insertNode(ListNode p, String x) {
    ListNode newNode = new ListNode();
    newNode.data = x;

    if (head == null) {       // 공백 리스트인 경우
        head = newNode;
        newNode.link = null;
    } else if (p == null) {  // p가 null이면 리스트의 첫 번째 노드로 삽입
        newNode.link = head;
        head = newNode;
    } else {                  // p가
        newNode.link = p.link;
        p.link = newNode;
    }
}
```



# p가 가리키는 노드 다음에 원소 x 삽입 (3)

```
/**
 * 리스트에서 p가 가리키는 노드 다음에 원소 x를 삽입
 * 알고리즘 4.1의 Java 구현
 */
public void insertNode(ListNode p, String x) {
    ListNode newNode = new ListNode();
    newNode.data = x;

    if (head == null) {      // 공백 리스트인 경우
        head = newNode;
        newNode.link = null;
    } else if (p == null) {  // p가 null이면 리스트의 첫 번째 노드로 삽입
        newNode.link = head;
        head = newNode;
    } else {                 // p가 가리키는 노드의 다음 노드로 삽입
        newNode.link = p.link;
        p.link = newNode;
    }
}
```



# 마지막 노드로 삽입

```
/**
 * list의 끝에 원소 x를 삽입
 * 알고리즘 4.2의 Java 구현
 */
public void addLastNode(String x) {
    ListNode newNode = new ListNode();  // 새로운 노드 생성
    newNode.data = x;
    newNode.link = null;

    if (head == null) {
        head = newNode;
        return;
    }

    ListNode p = head;          // p는 임시 순회 레퍼런스 변수
    while (p.link != null) {    // 마지막 노드를 탐색
        p = p.link;
    }
    p.link = newNode;           // 마지막 노드로 첨가
}
```

# 두 개의 리스트를 하나의 리스트로 연결

```
/**
 * 두 개의 리스트를 하나의 리스트로 연결
 * 알고리즘 4.5의 Java 구현
 */
public LinkedList addList(LinkedList list) {
    if (head == null) {
        head = list.head;
        return this;
    } else if (list.head == null) {
        return this;
    } else {
        ListNode p = head;      // p는 임시 순회 포인터
        while (p.link != null)
            p = p.link;
        p.link = list.head;
        return this;
    }
}
```

# 단순 연결 리스트에서 원소 값이 x인 노드 탐색

```java
/**
 * 단순 연결 리스트에서 원소 값이 x인 노드를 탐색
 * 알고리즘 4.6의 Java 구현
 */
public ListNode searchNode(String x) {
    ListNode p = head;          // p는 임시 포인터

    while (p != null) {
        if (x.equals(p.data))  // 원소 값이 x인 노드를 발견
            return p;
        p = p.link;
    }

    return p;  // 원소 값이 x인 노드가 없는 경우 null을 반환
}



    for (ListNode p = head; p != null; p = p.link) {
        if (x.equals(p.data))
            return p;
    }
```

# 리스트에서 p가 가리키는 노드의 다음 노드를 삭제

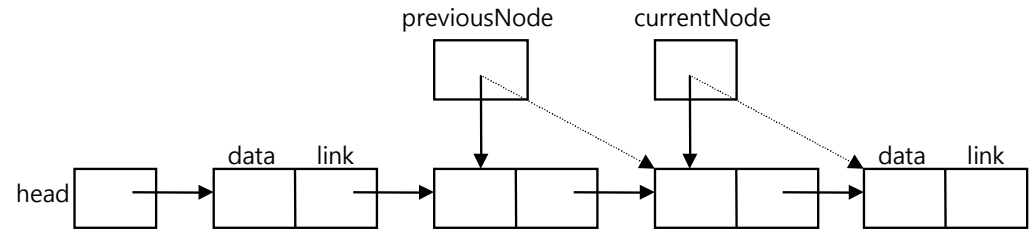
```java
/**
 * p가 가리키는 노드의 다음 노드를 삭제
 * 알고리즘 4.3의 Java 구현
 */
public void deleteNext(ListNode p) {
    if (head == null)    // head가 null이면 에러
        throw new NullPointerException();
    if (p == null)       // p가 null이면 첫 번째 노드 삭제
        head = head.link;
    else {
        ListNode q = p.link;
        if (q == null)   // 삭제할 노드가 없는 경우
            return;
        p.link = q.link;
    }
}
```

# 리스트의 원소를 역순으로 변환

```java
/**
 * 리스트의 원소를 역순으로 변환
 * 알고리즘 4.4의 Java 구현
 */
public void reverse() {
    ListNode p = head;  // p는 역순으로 변환될 리스트
    ListNode q = null;  // q는 역순으로 변환될 노드
    ListNode r = null;
    while (p != null) {
        r = q;          // r은 역순으로 변환된 리스트
                        // r은 q, q는 p를 차례로 따라간다.
        q = p;
        p = p.link;
        q.link = r;     // q의 링크 방향을 바꾼다.
    }
    head = q;
}
```

# 리스트의 마지막 노드의 삭제

```java
/**
 * 리스트의 마지막 원소를 삭제
 * 알고리즘 4.2의 Java 구현
 */
public void deleteLastNode() {
    ListNode previousNode, currentNode;

    if (head == null)
        return;

    if (head.link == null) {  // 원소가 하나 밖에 없는 경우
        head = null;
        return;
    } else {
        previousNode = head;
        currentNode = head.link;
        while (currentNode.link != null) {
            previousNode = currentNode;
            currentNode = currentNode.link;
        }
        previousNode.link = null;
    }
}
```



# 연결 리스트의 프린트

```
/**
 * 리스트의 내용을 화면에 출력
 * 프로그램 4.3
 */
public void printList() {
    ListNode p;
    System.out.print("(");
    p = head;
    while (p != null) {
        System.out.print(p.data);
        p = p.link;
        if (p != null) {
            System.out.print(", ");
        }
    }
    System.out.println( “ ) ” );
}
```

# 출력

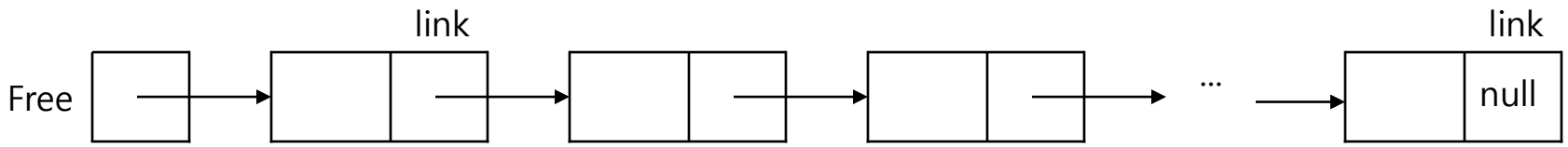
```
    public static void main(String args[]) {
        LinkedList list = new LinkedList();
        list.addLastNode("Kim");
        list.addLastNode("Lee");
        list.addLastNode("Park");
        list.printList();       // (Kim, Lee, Park)가 프린트
        list.addLastNode("Yoo");        // 원소 “Yoo”를 리스트 끝에 첨가
        list.printList();       // (Kim, Lee, Park, Yoo)가 프린트

        list.deleteLastNode();
        list.printList();       // (Kim, Lee, Park)가 프린트

        list.reverse();
        list.printList();       // (Park, Lee, Kim)이 프린트
    }

}
```

# 4.4 자유 공간 리스트

# 자유 공간 리스트 (free space list)

- 필요에 따라 요구한 노드를 할당할 수 있는 자유 메모리 풀

- 초기 자유 공간 리스트

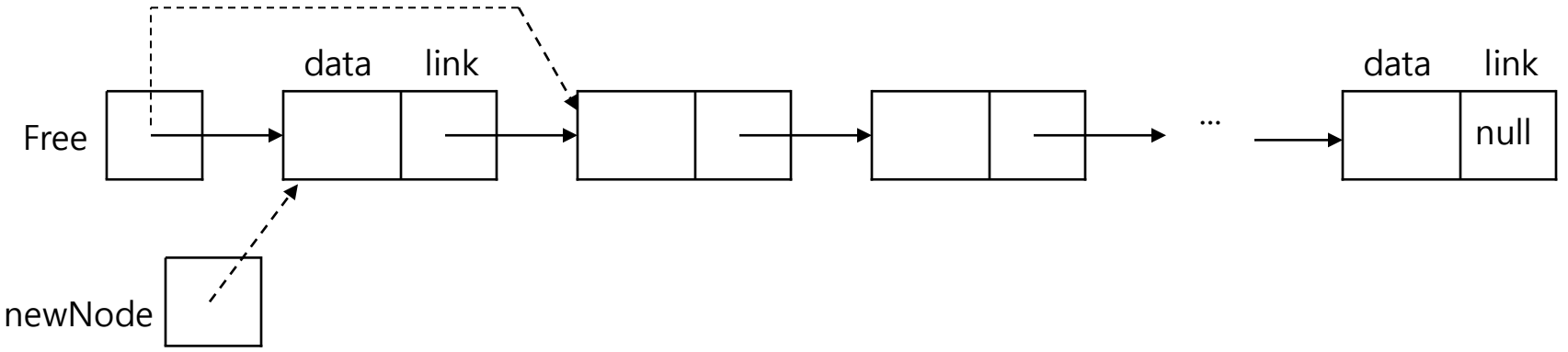



  - 노드 할당 요청을 받으면 자유 공간 리스트 앞에서부터 공백 노드를 할당

# 노드 할당

- 새로운 노드를 할당하는 함수 : getNode()

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

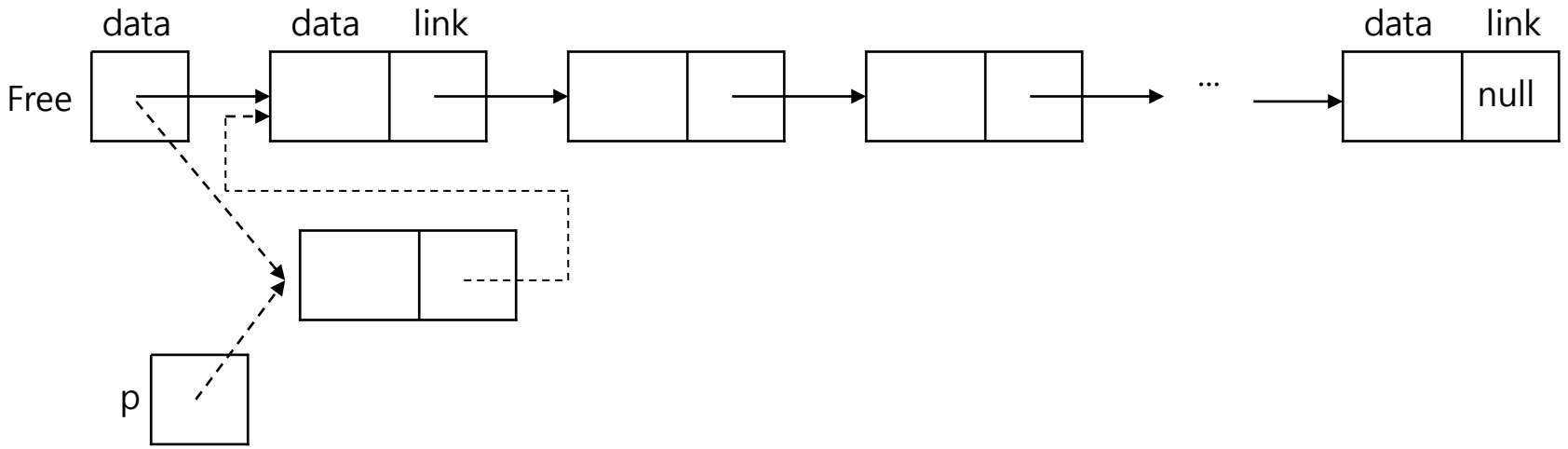


# 노드 반환

- 삭제된 노드를 자유 공간 리스트에 반환
  - 알고리즘

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

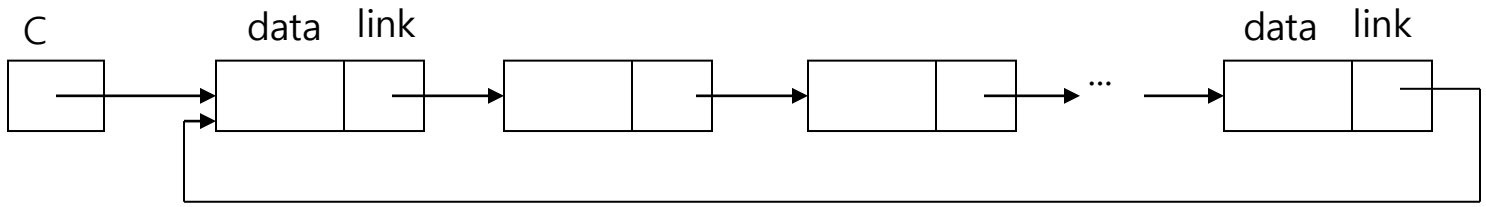
- 반환된 노드가 자유 공간 리스트에 삽입되는 과정



# 4.5 원형 연결 리스트

# 원형 연결 리스트

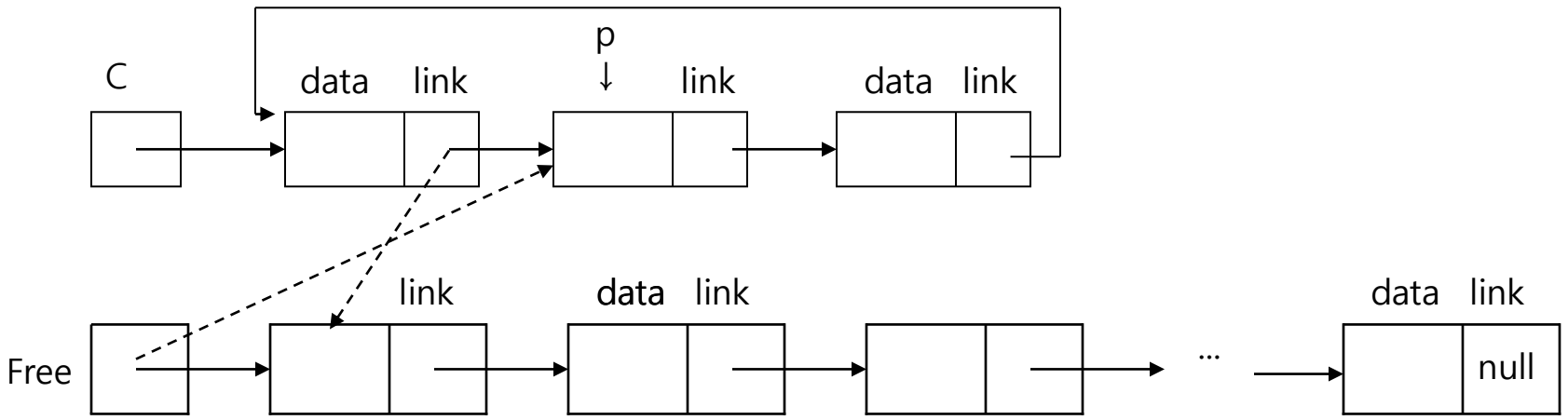
- 원형 연결 리스트 (circular linked list)
  - 마지막 노드의 링크가 다시 첫 번째 노드를 가리키는 리스트
  - 한 노드에서 다른 어떤 노드로도 접근할 수 있음
  - 리스트 전체를 자유 공간 리스트에 반환할 때 리스트의 길이에 관계없이 일정 시간에 반환할 수 있음
  - 예



# 원형 연결 리스트 연산 (1)

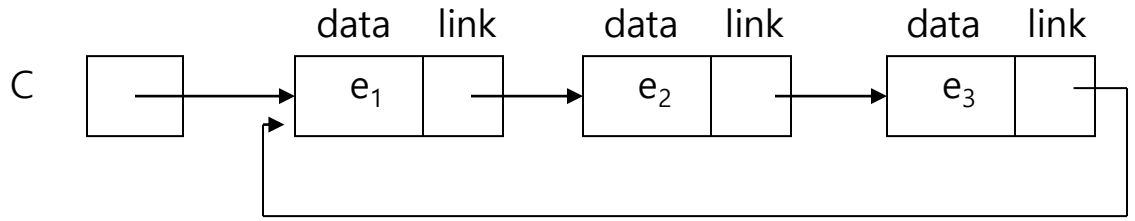
- 원형 연결 리스트를 자유 공간 리스트에 반환

```
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()
```



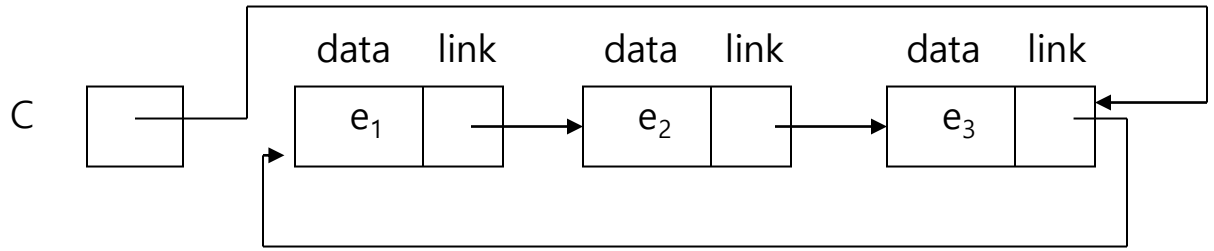
# 원형 연결 리스트 연산 (2)

- 원형 리스트에서의 노드 삽입
  - 예) C = ($e_1$, $e_2$, $e_3$)



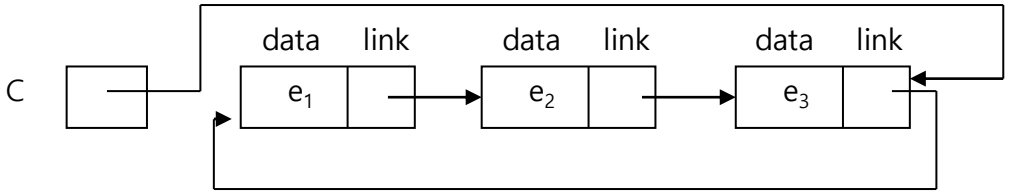

    - 문제점 : 리스트의 맨 뒤에 새로운 노드 삽입

- 개선책



# 원형 연결 리스트 – 리스트 처음/마지막에 삽입

```java
/**
 * 원형 연결 리스트의 처음에 노드를 삽입한다.
 * @param p 삽입하고자 하는 노드
 */
public void insertFront(ListNode p) {
    if (head == null) {
        head = p;
        p.link = head;
    } else {
        p.link = head.link;
        head.link = p;
    }
}

/**
 * 원형 연결 리스트의 마지막에 노드를 삽입한다.
 * @param p 삽입하고자 하는 노드
 */
public void insertLast(ListNode p) {
    if (head == null) {
        head = p;
        p.link = head;
    } else {
        p.link = head.link;
        head.link = p;
        head = p;
    }
}
```



# 원형 연결 리스트 – 리스트 길이 계산

```java
/**
 * 리스트의 길이를 반한하는 메소드
 * @return 리스트의 길이
 */
public int length() {
    if (head == null)
        return 0;

    int length = 1;
    ListNode p = head.link;
    while (p != head) {
        length++;
        p = p.link;
    }
    return length;
}
```

# 개선한 방법

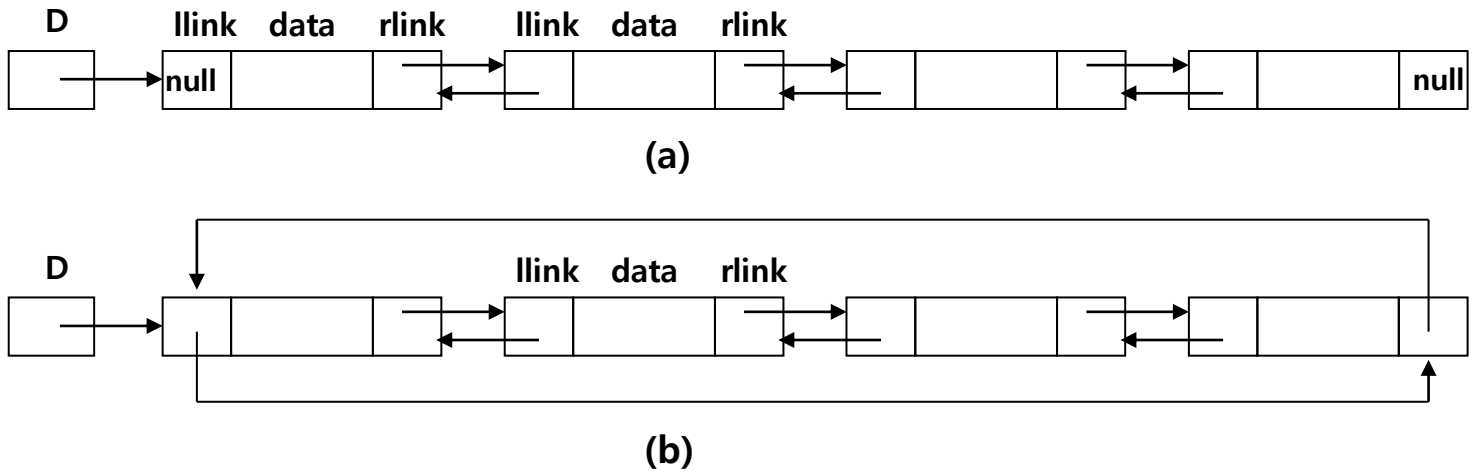
```
public class CircularLinkedList {
    private ListNode head;
    private int length;

    public CircularLinkedList() {
        head = null;
        length = 0;
    }

    public void insertFront(ListNode p) {
        if (head == null) {
            head = p;
            p.link = head;
        } else {
            p.link = head.link;
            head.link = p;
        }
        length++;
    }

    public void insertLast(ListNode p) {
        insertFront(p);     // 미리 만들어 둔 메소드 이용
        head = p;
    }

    public int length() {
        return length;      // 길이를 미리 저장하고 있으면 간단함
    }
}
```

# 4.6 이중 연결 리스트

# 이중 연결 리스트 (doubly linked list)

- 단순 연결 리스트나 원형 연결 리스트의 문제점
  - 어떤 노드 p에서 이 p의 선행자를 찾기가 어려움
- 이중 연결 리스트 (doubly linked list)
  - 노드는 data, llink(왼쪽 링크), rlink(오른쪽 링크) 필드를 가짐
  - 리스트는 선형이 될 수도 있고 원형이 될 수도 있음

  - `p = p.llink.rlink = p.rlink.llink`



# 이중 연결 리스트의 노드

```java
/**
 * 이중 연결 리스트에서 노드를 표현하기 위한 클래스이다.
 * 이 노드를 생성하면 기본적으로 모든 변수는 null이 된다.
 * Java에서는 변수의 값을 null로 설정하지 않아도 기본적으로 null이 된다.
 */
public class DoubleListNode {
    Object data = null;

    DoubleListNode rlink = null;
    DoubleListNode llink = null;
}
```

# 이중 연결 리스트 연산 – 삭제

```
public class DoubleLinkedList {
    DoubleListNode head = null;
    DoubleListNode tail = null;
    int length = 0;

    /**
     * @param p 삭제하고자 하는 노드
     * 노드 p를 삭제한다.
     */
    public void delete(DoubleListNode p) {
        if (p == null)
            throw new NullPointerException();

        if (p.llink != null)
            p.llink.rlink = p.rlink;
        else
            head = head.rlink;

        if (p.rlink != null)
            p.rlink.llink = p.llink;
        else
            tail = tail.llink;

        length--;
    }
```

```
/**
 * @param p 삭제하고자 하는 노드
 * 노드 p를 삭제한다.
 */
public void delete(DoubleListNode p) {
    if (p == null)
        throw new NullPointerException();

    if (length == 1) {
        head = tail = null;
    } else if (p == head) {
        head = head.rlink;
        head.llink = null;
    } else if (p == tail) {
        tail = tail.llink;
        tail.rlink = null;
    } else {
        p.llink.rlink = p.rlink;
        p.rlink.llink = p.llink;
    }

    length--;
}
```

# 이중 연결 리스트 연산 – 삽입

```
    /**
     * @param p 이 노드 뒤에 삽입
     * @param q 삽입할 노드
     * 노드 p 뒤에 노드 q를 삽입한다.
     */
    public void insert(DoubleListNode p, DoubleListNode q) {
        if (q == null)
            throw new NullPointerException();

        q.llink = p;
        if (p == null) {  // 제일 앞에 삽입
            q.rlink = head;
            if (head != null)
                head.llink = q;
            head = q;
        } else {
            q.rlink = p.rlink;
            if (p.rlink != null)
                p.rlink.llink = q;
            else
                tail = q;
            p.rlink = q;
        }
        length++;
    }
}
```

# 4.7 헤더 노드

# 헤더 노드

- 기존의 연결 리스트 처리 알고리즘
  - 객체지향 기법을 적용할 경우 리스트를 가리키는 객체를 정의
  - 이 객체를 교재에서는 헤더 노드라 부름

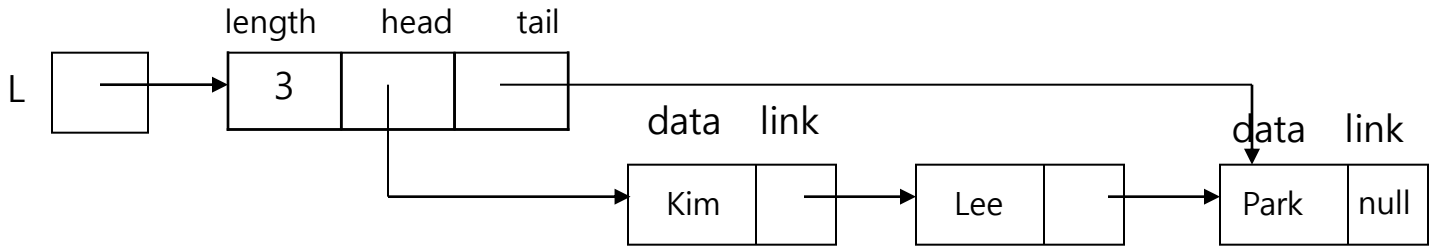
- 헤더 노드 (header node)를 추가하여 예외 제거
  - Java에서 연결 리스트 객체가 헤더 노드임
  - 알고리즘을 단순화
  - 연결 리스트를 처리하는 데 필요한 정보를 저장
  - 헤더 노드에는 리스트의 첫 번째 노드를 가리키는 포인터(head), 리스트의 길이(length), 참조 계수(ref), 마지막 노드를 가리키는 포인터(tail) 등의 정보를 저장

# 리스트 클래스 정의

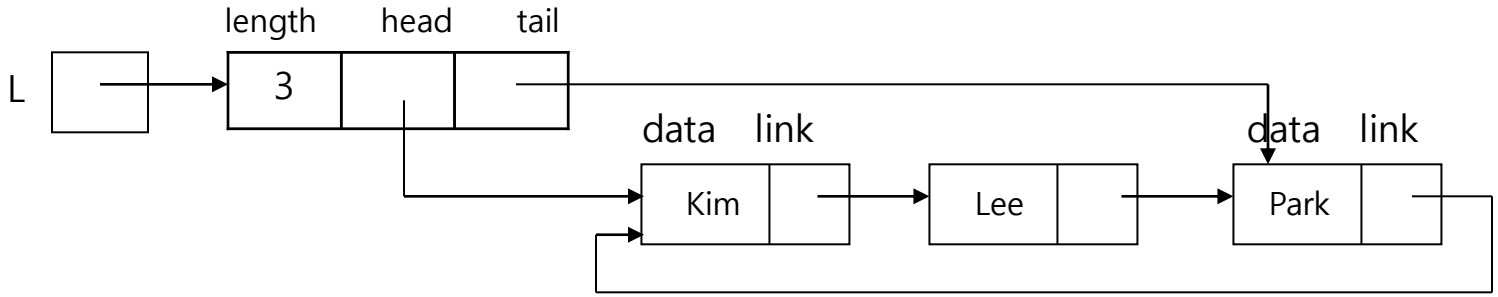
```
class ListNode {
    String    data;
    ListNode  link;
}

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 = tail = null;
    }

    public void addLastNode(String x) {
         . . .
    }

    public void reverse() {
         . . .
    }

    // 기타 다른 메소드 정의
}
```

# 리스트 객체의 연결 리스트 표현 (1)

- 단순 연결 리스트

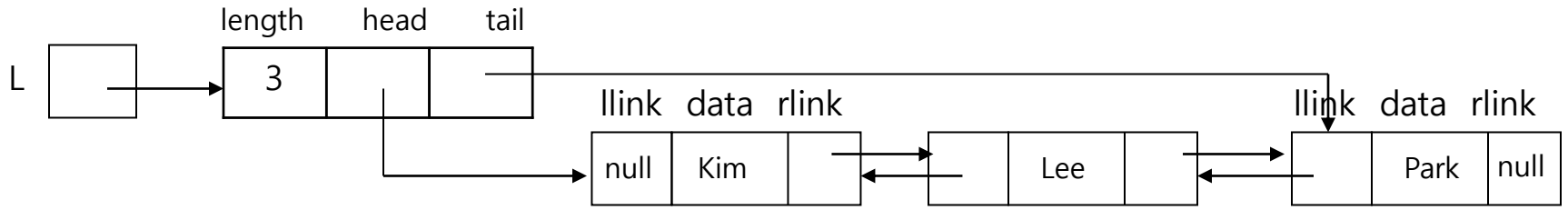



- 원형 연결 리스트

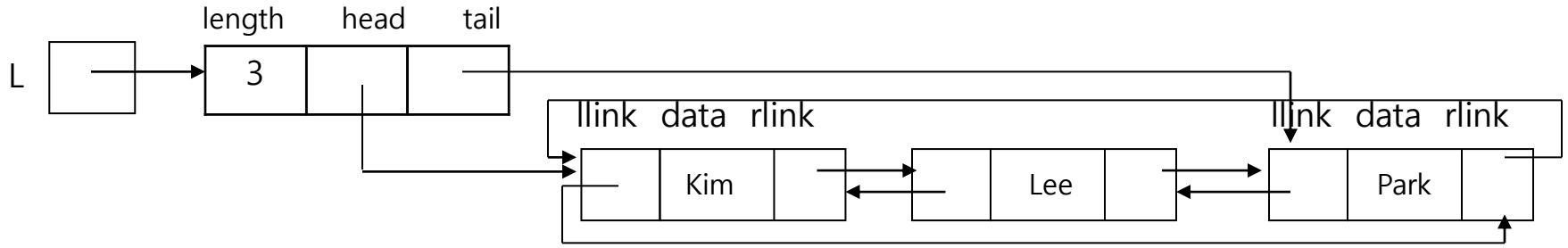


# 리스트 객체의 연결 리스트 표현 (1)

- 이중 연결 리스트

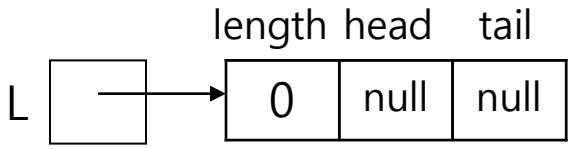



- 이중 연결 원형 리스트



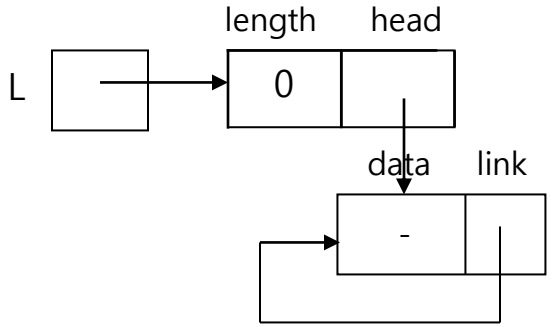
# 리스트 객체의 연결 리스트 표현 – 공백 리스트 (1)

- 공백 리스트
  - length가 0이고 head가 null, tail이 null인 헤더 노드로 표현




- 원형 연결 리스트에서의 공백 리스트

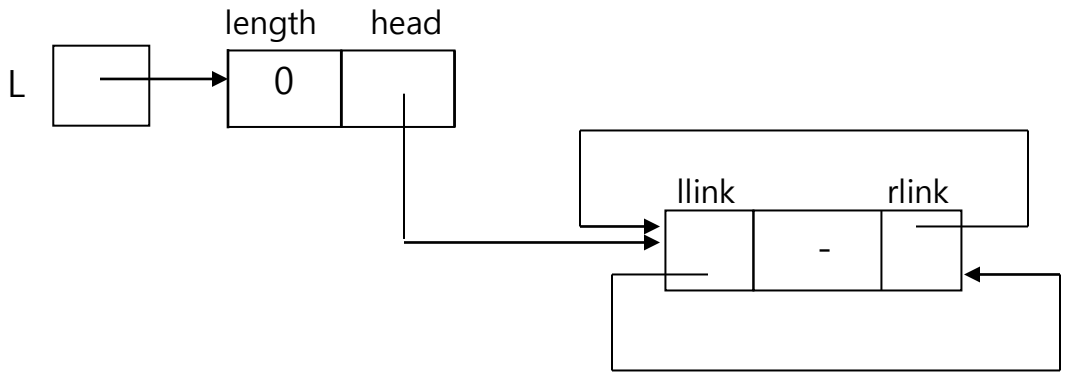
```
LinkedList() {
    head = new ListNode();
    head.link = head;
    length = 0;
}
```



# 리스트 객체의 연결 리스트 표현 – 공백 리스트 (2)

- 이중 연결 원형 리스트의 공백 리스트 구조

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



# 4.8 다항식의 리스트 표현과 덧셈

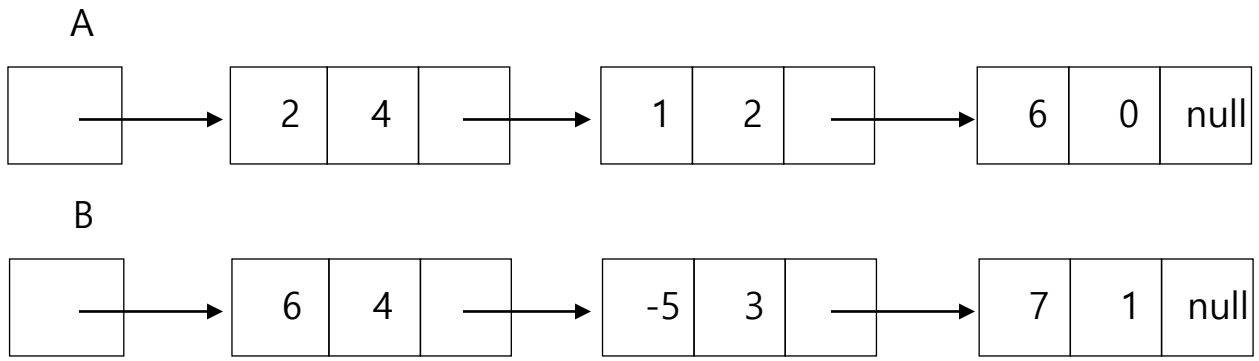
# 다항식의 리스트 표현

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

다항식 노드 :

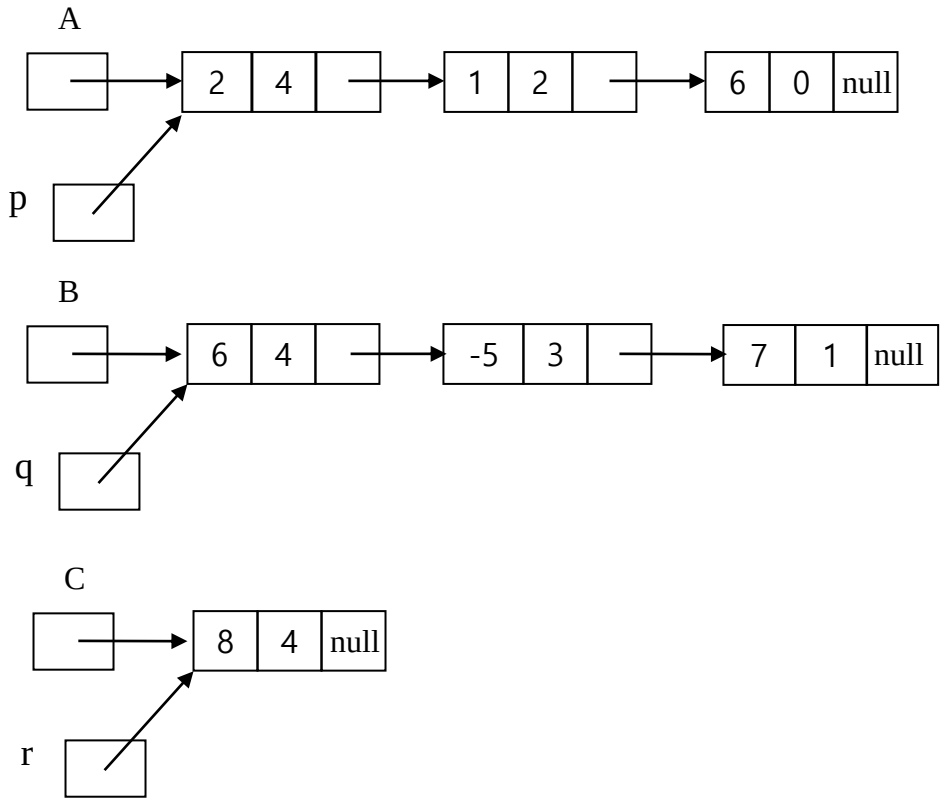

- $A(x) = 2x^4 + x^2 + 6$

  $B(x) = 6x^4 - 5x^3 + 7x$



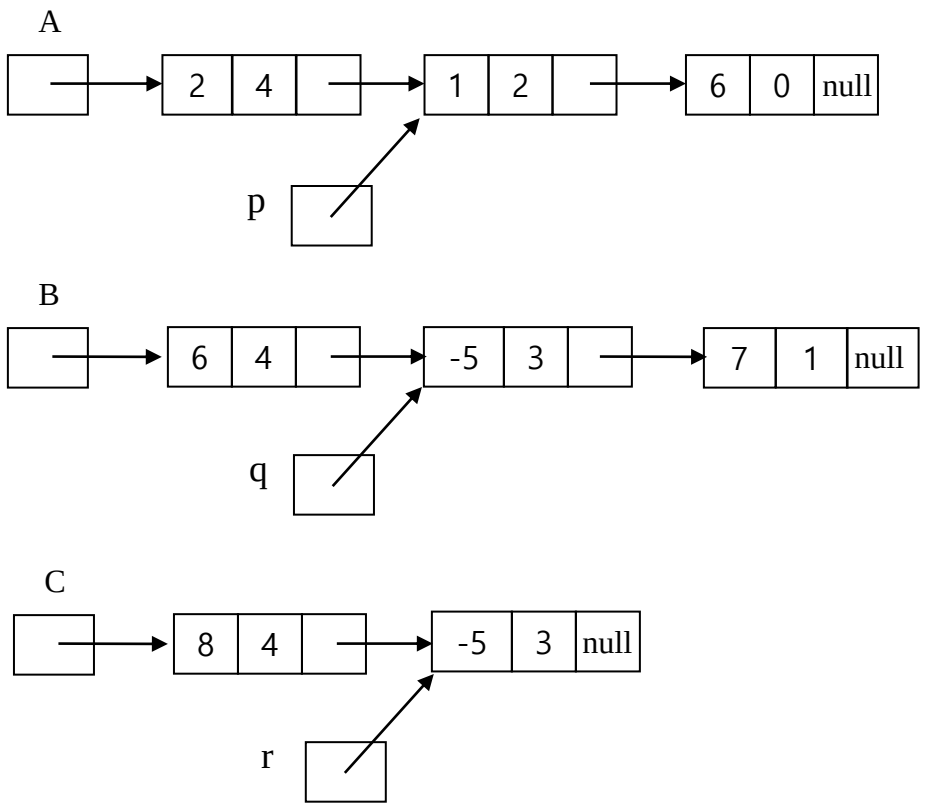
# 다항식의 덧셈 (1)

- p.exp = q.exp



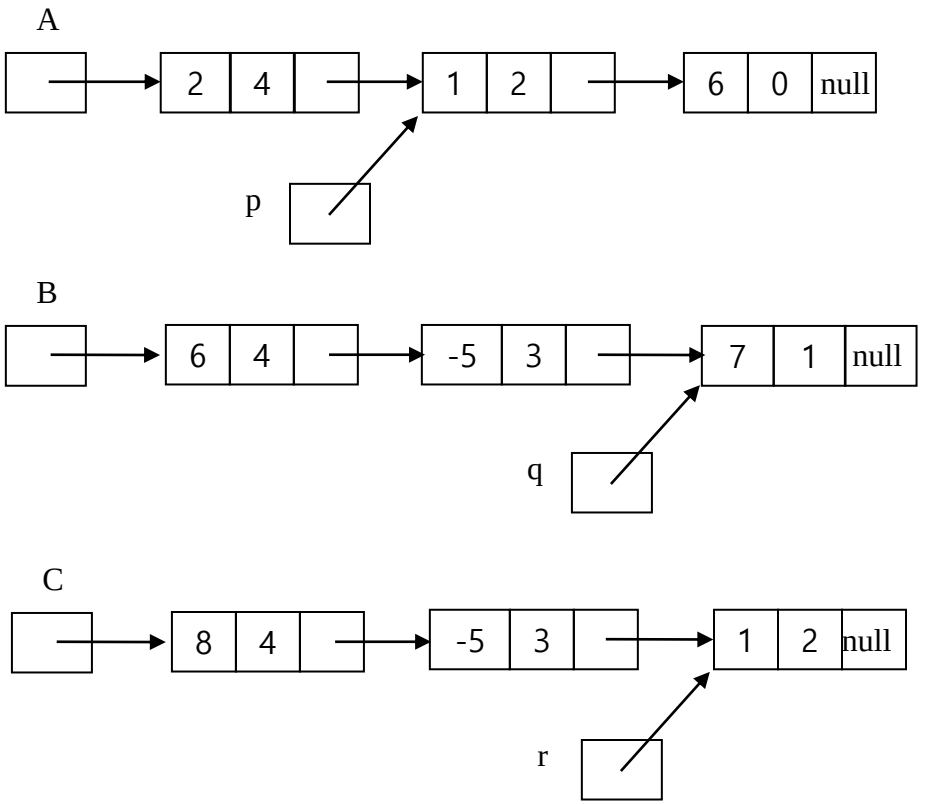
# 다항식의 덧셈 (2)

- p.exp < q.exp :



# 다항식의 덧셈 (3)

- p.exp > q.exp :



# 다항식에 새로운 항을 첨가

```
// c는 계수, e는 지수, last는 poly의 마지막 항을 가리키는 포인터
appendTerm(poly, c, e, last)
    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()
```

```
public class Polynomial {
    class TermNode {
        double coef;
        int exp;
        TermNode link;
    }

    private TermNode poly; // 첫번째 노드
    private TermNode last; // 마지막 노드

    public void appendTerm(double c, int e) {
        TermNode newNode = new TermNode();
        newNode.exp = e;
        newNode.coef = c;
        if (poly == null) {
            poly = last = newNode;
        } else {
            last.link = newNode;
            last = newNode;
        }
    }
```

# 연결 리스트로 표현된 다항식의 덧셈

```
// 단순 연결 리스트로 표현된 다항식 A와 B를 더하여 새로운 C를 반환.
polyAdd(A, B)
    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;
            else :        // p.exp > q.exp인 경우
                appendTerm(C, p.coef, p.exp, r);
                p ← p.link;
        }
    }

    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()
```

# 연결 리스트로 표현된 다항식의 덧셈의 Java 표현

```
public Polynomial polyAdd(Polynomial a) {
    TermNode p = this.poly;
    TermNode q = a.poly;
    Polynomial c = new Polynomial();

    while(p != null && q != null) {
        if (p.exp == q.exp) {
            double sum = p.coef + q.coef;
            if (sum != 0)
                c.appendTerm(sum, p.exp);
            p = p.link;
            q = q.link;
        } else if (p.exp < q.exp) {
            c.appendTerm(q.coef, q.exp);
            q = q.link;
        } else {
            c.appendTerm(p.coef, p.exp);
            p = p.link;
        }
    }

    while(p != null) {
        c.appendTerm(p.coef, p.exp);
        p = p.link;
    }

    while(q != null) {
        c.appendTerm(q.coef, q.exp);
        q = q.link;
    }

    return c;
}
```

# 4.9 일반 리스트

# 4.9.1 일반 리스트 구조

# 일반 리스트(general list)

- n≥0개의 원소 $e_1$, $e_2$, ... $e_n$의 유한 순차(finite sequence)
- 원소는 원자(atom)나 리스트(list)가 될 수 있음
- 리스트의 원소 리스트를 서브리스트(sublist)라 함.
- 리스트는 L=($e_1$, $e_2$, ... $e_n$)에서 L은 리스트 이름이고, n은 리스트의 원소수 즉 리스트의 길이가 됨
- n≥1인 경우 첫 번째 원소 $e_1$을 L의 head, 즉 head(L)로 표현하고, 첫 번째 원소를 제외한 나머지 리스트 ($e_2$, ... $e_n$)을  L의 tail, 즉 tail(L)로 표현함
- 공백 리스트(( ))에 대해서는 연산자 head와 tail은 정의되지 않음
- 일반 리스트 정의 속에 다시 리스트를 사용하고 있기 때문에 순환적 정의

# 일반 리스트 예

(1) A=(a, (b, c))

- 길이가 2이고 첫 번째 원소는 a이고 두 번째 원소는 서브리스트 (b, c)이다.
- head(A) = a, tail(A) = ((b, c))
- head(tail(A)) = (b, c), tail(tail(A)) = ()

(2) B=(A, A, ( ))

- 길이가 3이고 처음 두 원소는 서브리스트 A이고 세 번째 원소는 공백 리스트이다. 이 리스트 B는 리스트 A를 공용하고 있다.
- head(B) = A, tail(B) = (A, ())
- head(tail(B)) = A, tail(tail(B)) = (())

(3) C=(a, C)

- 길이가 2인 순환리스트
- (a, (a, (a, ...)

(4) D=( )

- 길이가 0인 널(null), 즉 공백 리스트.

# 4.9.2 일반 리스트 표현

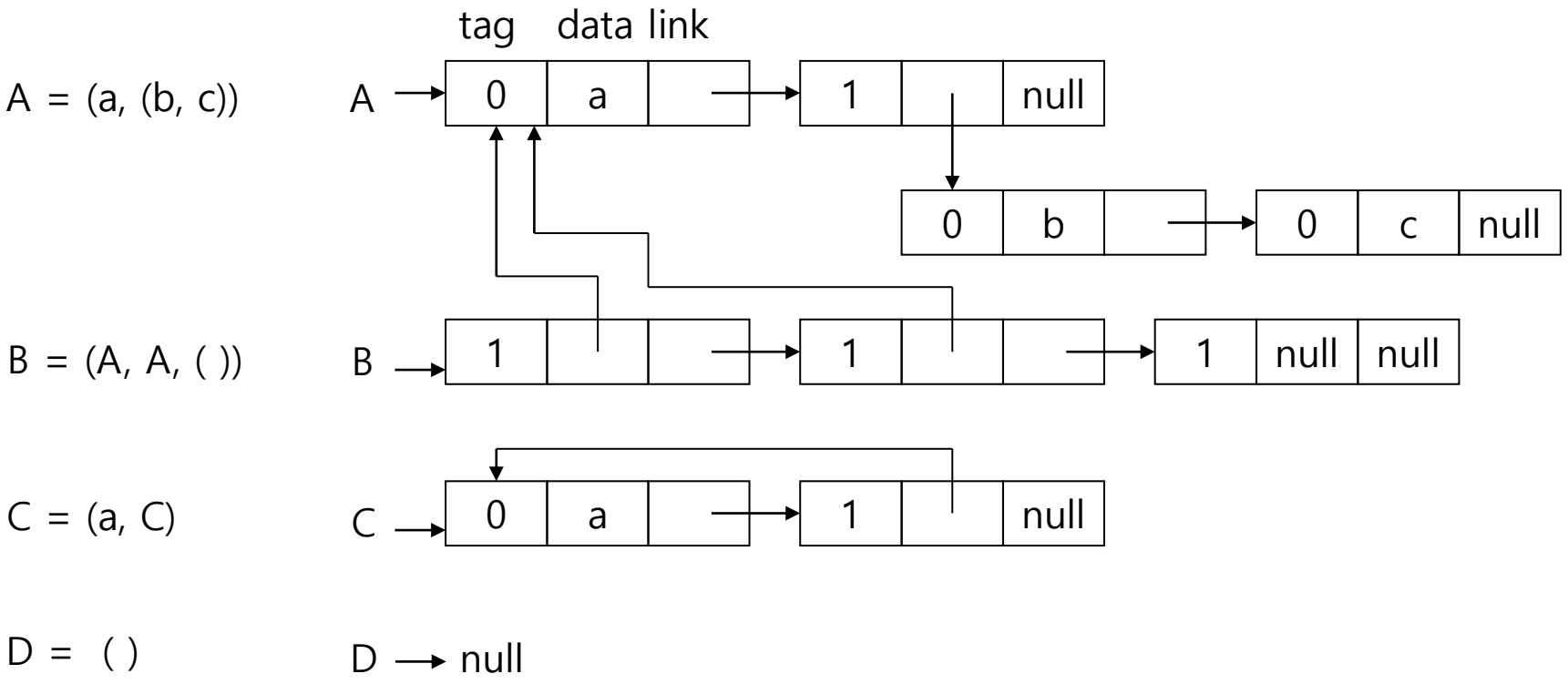
# 일반 리스트 표현 (1)

- 일반 리스트의 노드 구조


  - data 필드
    - 리스트의 head를 저장
    - head(L)이 원자인지 서브리스트인지에 따라 원자 값이나 서브리스트의 포인터를 저장
  - tag 필드
    - data 필드 값이 원자인지 포인터 값인지를 표시
    - tag = 0: data 값은 원자 값
    - tag = 1: data 값은 서브리스트에 대한 포인터
  - link 필드
    - 리스트의 tail에 대한 포인터를 저장

# 일반 리스트 표현 (2)

- 앞의 리스트의 예



# 4.9.3 공용 리스트와 참조 계수

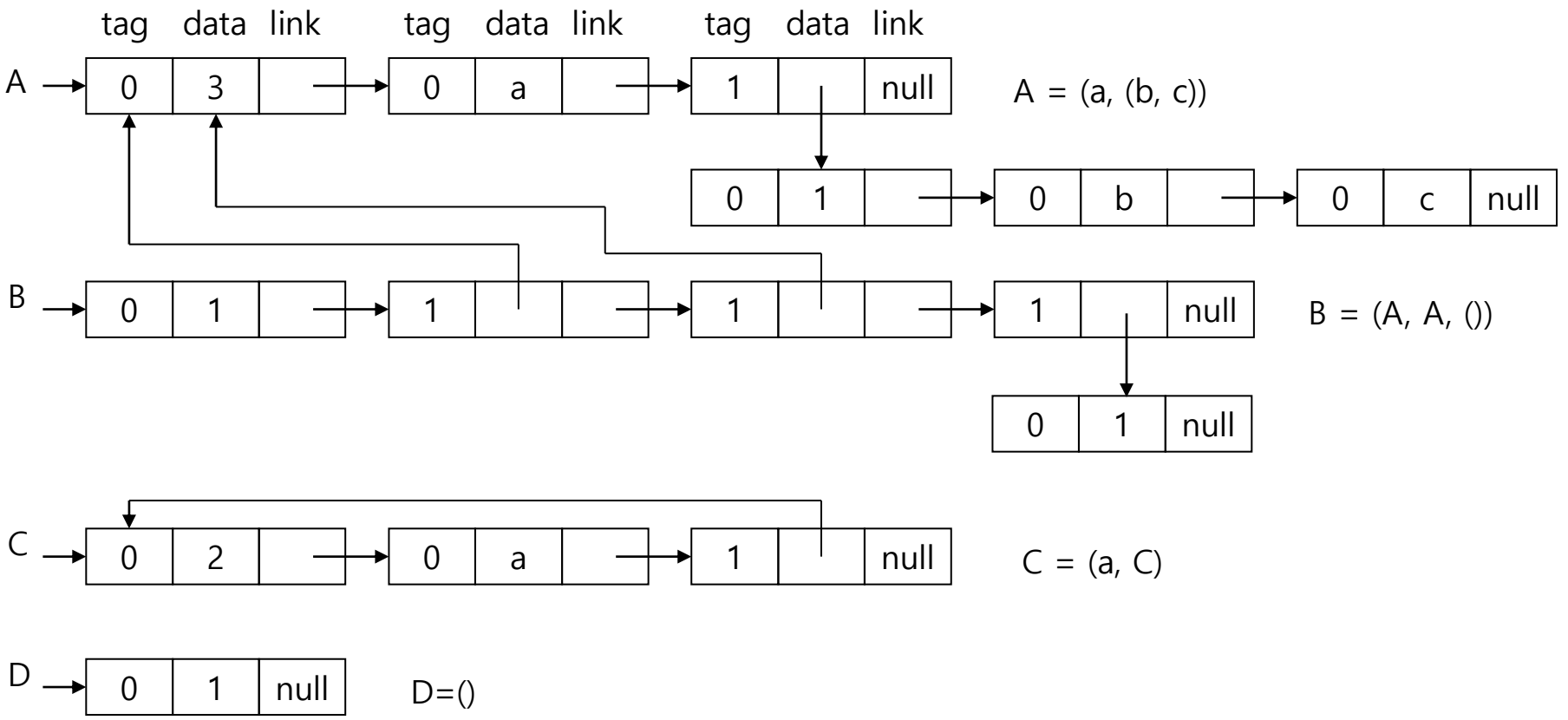
# 공용 리스트와 참조 계수 (1)

- 서브리스트의 공용
  - 리스트 공용은 저장 공간을 절약
  - 공용 리스트의 첫 번째 노드를 삽입하거나 삭제할 때 이 리스트를 공용하는 리스트도 변경해야 되는 문제가 있음
    - 리스트 A의 첫 번째 노드 삭제 : A를 가리키는 포인터 값을 A의 두 번째 노드를 가리키도록 변경
    - 새로운 노드를 리스트 A의 첫 번째 노드로 첨가 : B의 포인터들을 이 새로 삽입된 첫 번째 노드를 가리키도록 변경
  - 한 리스트가 어떤 리스트에 의해 참조되고 있는지 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 저장, tag 필드는 0으로 설정



# 공용 리스트와 참조 계수 (3)

- 참조 계수 (reference count)
  - 자기(리스트)를 참조하고 있는 포인터 수를 헤더 노드의 data 필드에 저장
  - 리스트를 자유 공간 리스트에 반환할 것인가를 결정할 때, 리스트 헤더에 있는 참조 계수를 검사하여 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)

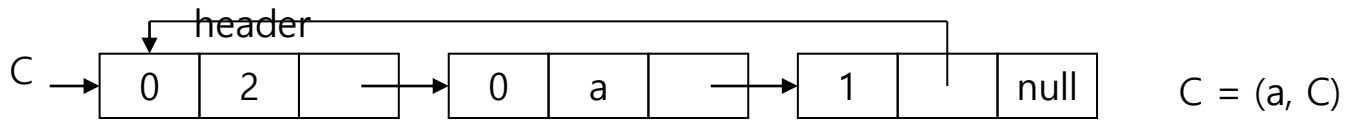
- 자유 공간 리스트로 리스트 반환
  - p가 가리키는 리스트를 삭제(removeList())하려면 먼저 p의 참조 계수(p.ref)를 1감소하고 0이 되면 모든 노드들을 자유 공간 리스트에 반환
  - p의 서브리스트에 대해서도 순환적으로 수행

```
removeList(L)
    // 헤더 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()
```

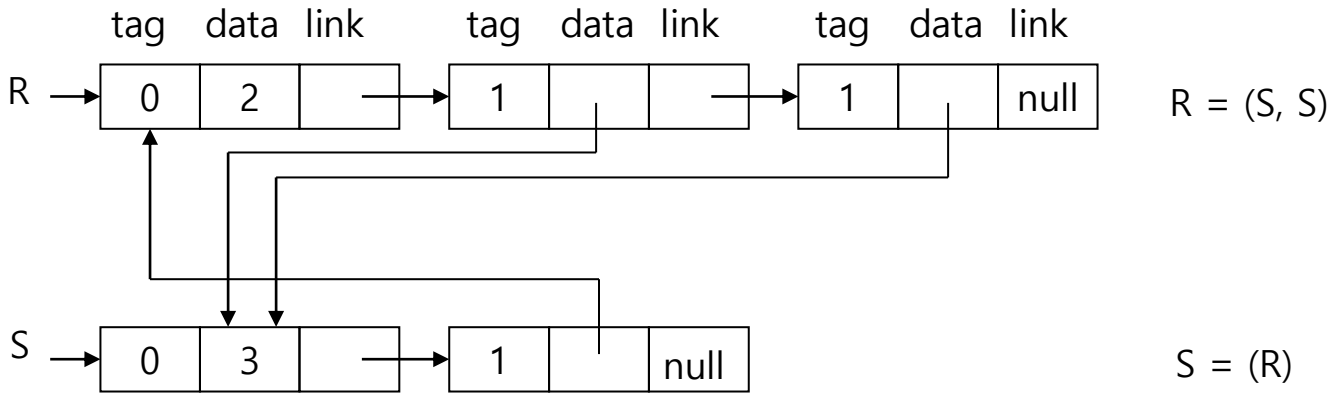
# 4.9.4 쓰레기 수집(garbage collection)

# 참조 계수 사용의 한계

- 순환 리스트
  - 반환되어야 될 리스트인데도 그 참조 계수가 결코 0이 되지 않음
  - 예) removeList(C) : C의 참조 계수가 1로 되지만 (0이 되지 않음) 이 리스트는 다른 포인터나 리스트를 통해 접근이 불가능




- 간접 순환
  - 예) removeList(R)과 removeList(S)가 실행된 뒤에 R.ref = 1, S.ref = 2가 되지만 더 이상 참조할 수 있는 리스트가 아님.
  - 또한 참조 계수가 0이 아니기 때문에 반환될 수도 없음



# Garbage Collection

- Garbage (쓰레기)
  - 실제로 사용되고 있지 않으면서도 자유 공간 리스트에 반환할 수 없는 메모리
  - 시스템 내에 쓰레기가 많이 생기면 자유공간 리스트가 소진되어 프로그램 실행을 더 이상 진행할 수 없는 경우가 발생 가능

- Garbage collection (쓰레기 수집)
  - 사용하지 않는 모든 노드들을 수집하여 자유 공간 리스트에 반환시켜 시스템 운영을 지속
  - 모든 리스트의 노드 크기가 일정하다고 가정
  - 각 노드에 추가로 markBit라는 특별한 비트를 할당하여 0과 1만을 갖도록 하고 노드의 사용 여부(0: 사용되지 않음, 1: 사용 중)를 표시

# Garbage Collection 과정

- 1. 초기화 단계 (initialization)
  - 메모리에 있는 모든 노드의 markBit 를 0으로 설정하여 일단 사용하지 않는 것으로 표시

2. 마크 단계 (marking phase)
  - 현재 사용하고 있는 리스트 노드들을 식별해서 markBit 를 1로 변경하고 이 노드의 data 필드를 검사
  - 리스트들은 모두 포인터 p[i]를 통해서만 접근한다고 가정
  - data 필드가 다른 리스트를 참조하면 이 필드에서부터 다시 이 단계2를 순환적으로 수행
  - data 필드를 따라 처리를 끝낸 뒤에는 다시 link 필드를 따라 다음 노드를 처리 (이 때 markBit가 1이면 그 노드를 통한 경로는 진행할 필요가 없음)

3. 수집 단계 (collecting phase)
  - 모든 노드의 markBit를 검사해서 0으로 마크된 노드들을 모두 자유 공간 리스트에 반환

# Garbage Collection Algorithm

```
// listNode는 markBit, tag, data, link로 구성
// listNode들은 listNodeArray[]라는 노드의 배열로부터
// 할당되어진다고 가정
// 포인터는 모두 배열 p[]로 표현된다고 가정
// listNodeArray[]의 크기(listNode의 수)는
// listNodeArraySize로 가정

garbageCollection()

    // 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]);   //사용중인 포인터 변수 p[i] 노드 마크

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


markListNode(p)
    // 포인터 p를 통해 사용중인 모든 노드를 마크
    if ((p≠null) and (p.markBit = 0)) then
        p.markBit ← 1;

     // data 경로를 따라 순환적으로 markBit을 검사
    if (p.tag = 1) then markListNode(p.data);
    markListNode(p.link);  // link 경로를 따라 markBit을 검사
end markListNode()
```

# 4.9.5 일반 리스트를 위한 함수

# 리스트의 복사본 생성

```
// L은 비 순환 일반 리스트로서 공용 서브 리스트가 없음
// L과 똑같은 리스트 p를 만들어 그 포인터를 반환
copyList(L)
    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 노드를 생성
        p.data ← q;
        p.link ← r;                            // head와 tail을 결합
        p.tag ← L.tag;
    }
    return p;
end copyList()
```

# 일반 리스트의 노드

```
public class ListNode implements Cloneable {
    Object data;
    ListNode link;

    public ListNode() {
        data = null;
        link = null;
    }

    public ListNode(Object data) {
        this.data = data;
        link = null;
    }

    public Object clone() {
        ListNode newNode = new ListNode();
        newNode.data = (data instanceof ListNode) ? ((ListNode)data).clone() : data;
        newNode.link = (ListNode)link.clone();
        return newNode;
    }

    public boolean equals(Object obj) {
        if (!(obj instanceof ListNode))
            return false;
        ListNode node = (ListNode)obj;
        boolean b = data.equals(node.data);
        if (b)
            b = link.equals(node.link);
        return b;
    }
}
```

# 리스트의 복사본 생성 – Java 표현

```
public class GenList implements Cloneable {
    private ListNode head;

    public GenList() {
        head = null;
    }

    // 알고리즘 4.16 (p.179-180)
    // 교과서에 나와있는 알고리즘은 Java 내부에서 처리해준다.
    public void removeList() {
        head = null;
    }


    // 알고리즘 4.18 (p.183)
    public Object clone() {
        GenList list = new GenList();
        list.head = (ListNode)head.clone();
        return list;
    }
```

# 두 리스트의 동일성 검사

```
// S와 T는 비 순환 일반 리스트, 각 노드는 tag, data, link 필드로 구성
// S와 T가 똑 같으면 true, 아니면 false를 반환
equalList(S, T)
    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()
```

```
// 알고리즘 4.19 (p.183-184)
public boolean equals(Object obj) {
    if (!(obj instanceof GenList))
        return false;

    return head.equals(((GenList)obj).head);
}
```

# 리스트의 깊이 계산

```
// L은 비 순환 일반 리스트, 노드는 tag, data, link로 구성
// 리스트 L의 깊이를 반환
depthList(L)
    max ← 0;
    if (L = null) then return(max); // 공백 리스트의 깊이는 0
    p ← L;
    while (p ≠ null) do {           // p는 순회 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()
```

```
// 알고리즘 4.20 (p.184)
public int depthList() {
    if (head == null)
        return 0;
    int max = 0;
    for(ListNode p = head; p != null; p = p.link) {
        int d;
        if (!(p.data instanceof ListNode)) {
            d = 0;
        } else {
            d = ((GenList)p.data).depthList();
        }
        if (d > max)
            max = d;
    }
    return max + 1;
}
```

# 4.9.6 Java에서의 일반 리스트 구현

# 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 저장하는 head 필드가 있는 헤더 노드를 가짐

- 프로그램 4.6

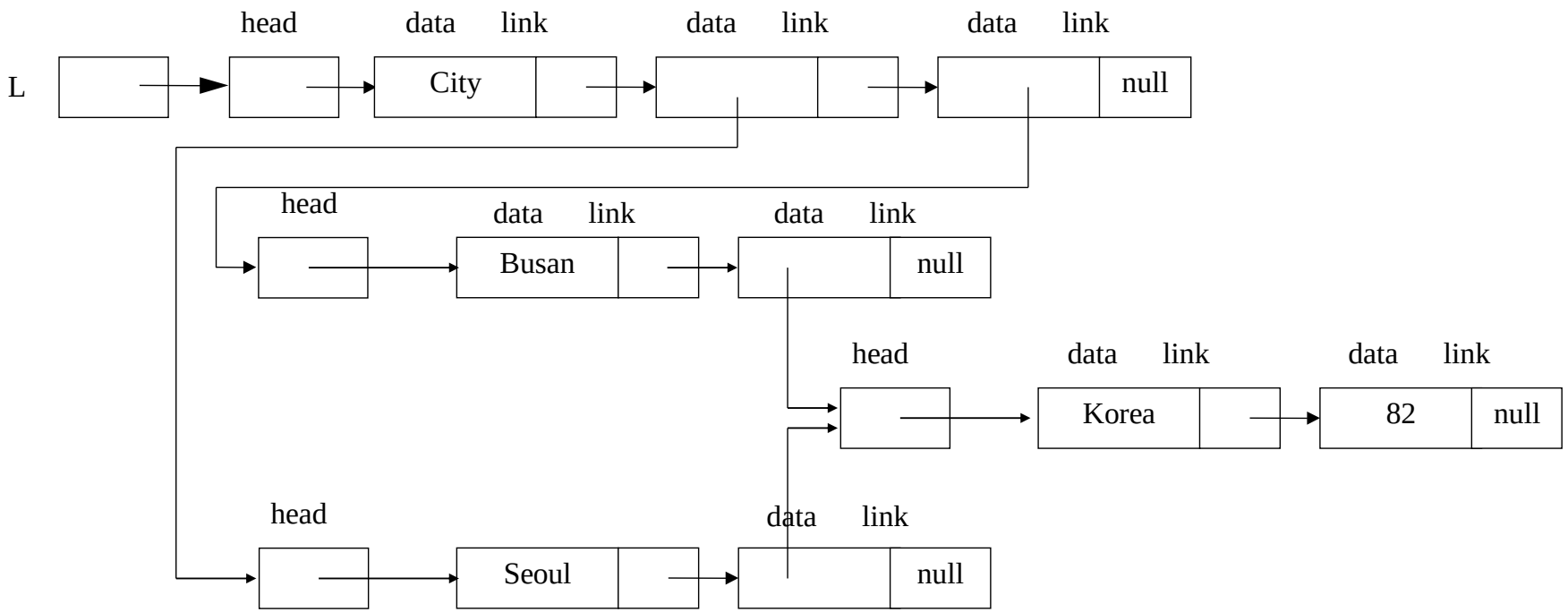
```
class ListNode {
    Object  data;  // 모든 Java 객체가 data 값이 될 수 있음.
    ListNode  link;

    public ListNode() {
        data =  link = null;
    }
}
```

# Java에서의 일반 리스트 구현 (2)

```
class GenList {
    /** 리스트의 첫 번째 ListNode에 대한 참조를 저장 */
    private ListNode  head;

    /** 리스트 head 다음에 새로운 ListNode를 삽입 */
    void insertData(Object x) {
       ListNode  newNode = new ListNode();
       newNode.data = x;
       newNode.link = head;
       head = newNode;
    }

    /** 일반 리스트를 프린트 */
    void printGL() {
        System.out.print("(");
        ListNode p = head;
        while ( p != null ) {      // 공백 리스트가 아닌 경우
            if (p.data instanceof GenList) {
                ((GenList)p.data).printGL();
            } else {
                System.out.print(p.data);
            }
            if ((p = p.link) != null) {
                System.out.print(", ");
            }
        }
        System.out.print(")");
    }

}
```

# Java에서의 일반 리스트 구현 (3)

- 공용 서브 리스트를 가진 일반리스트




L = (City, (Seoul, (Korea, 82)), (Busan, (Korea, 82)))

# 프로그램 4.7 (일반 리스트 생성 및 프린트)

```
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();
        L.insertData(r);
        L.insertData(q);
        L.insertData("City");

        L.printGL();
        System.out.ptinln();
    }

}
```