

# 제4장

# 연결 리스트

## 4.1 연결 리스트의 개념과 기본 연산

# 리스트

## ❶ 리스트(list)

기본적인 연산: 삽입(insert), 삭제(remove), 검색(search) 등

리스트를 구현하는 대표적인 두 가지 방법: 배열, 연결리스트

## ❷ 배열의 단점

크기가 고정 - `reallocation`이 필요

리스트의 중간에 원소를 삽입하거나 삭제할 경우 다수의 데이터를 옮겨야

## ❸ 연결리스트

다른 데이터의 이동없이 중간에 삽입이나 삭제가 가능하며,

길이의 제한이 없음

but, 랜덤 엑세스가 불가능

# 연결리스트

100	Monday
101	Tuesday
102	Friday
103	Saturday
104	
105	
106	
107	
108	

Array

100		
101		
102	Tuesday	107
103	Saturday	null
104		
105	Monday	102
106		
107	Friday	103
108		

Linked List

# 삽입과 삭제

100		
101		
102	Tuesday	108
103	Saturday	null
104		
105	Monday	102
106		
107	Friday	103
108	Thursday	107

Adding Thursday

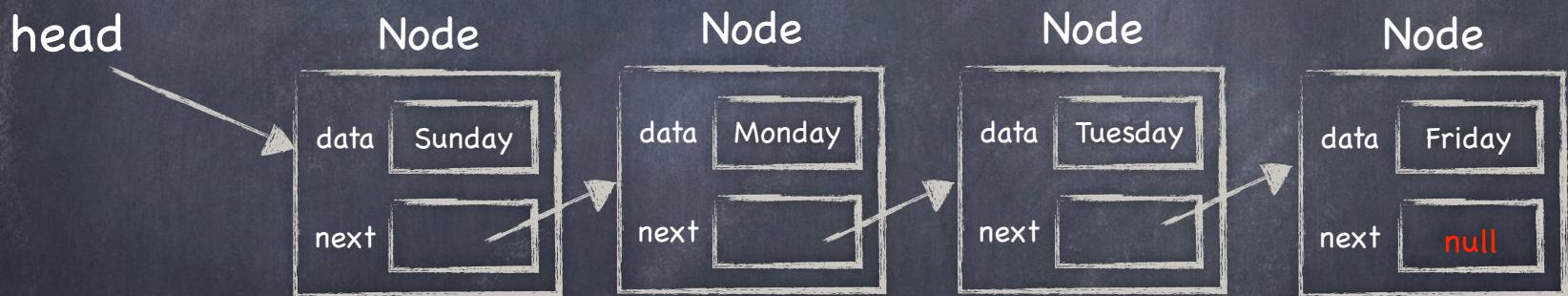
100		
101		
102		
103	Saturday	null
104		
105	Monday	108
106		
107	Friday	103
108	Thursday	107

Removing Tuesday



# 노드

- 각각의 노드는 “데이터 필드”와 하나 혹은 그 이상의 “링크 필드”로 구성
- 링크 필드는 다음 노드를 참조
- 첫 번째 노드의 주소는 따로 저장해야



# class Node<T>

연결 리스트에서 하나의 노드를 표현하기 위한 클래스이다. **Generics**로 작성해보자.

```
public class Node<T> {  
    public T data;  
    public Node<T> next;  
  
    public Node(T item) {  
        data = item;  
        next = null;  
    }  
}
```

하나의 데이터 객체를 저장할 필드 **data**와 다음 노드의 주소를 저장할 링크 필드 **next**를 가진다.

데이터 필드 **data**의 값을 받아서 초기화하는 가장 기본적인 생성자를 하나 만들었다.

# class MySingleLinkedList<T>

```
public class MySingleLinkedList<T> {  
    public Node<T> head;  
    public int size;
```

하나의 연결리스트를 표현하는 클래스  
MySingleLinkedList<T>를 만들어 보자.

```
public MySingleLinkedList() {  
    head = null;  
    size = 0;  
}
```

첫번째 노드의 주소 head와 노드의 개수 size를  
데이터 멤버로 가지도록 만들었다.

head를 null로, size는 0으로 초기화하는 생성자이다.

```
public void addFirst(T item) {  
    ...  
}
```

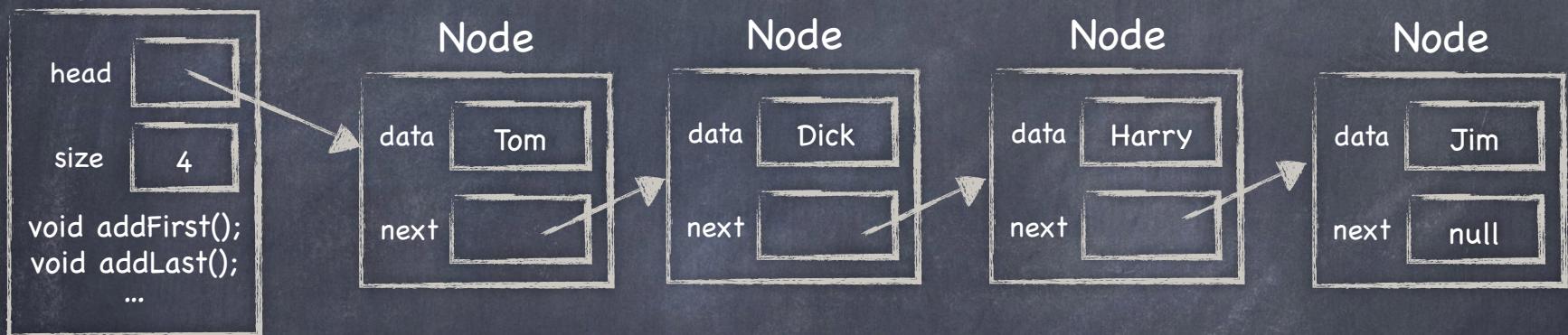
입력 받은 item을 저장하는 새로운 노드를 만들어 연결리스  
트의 맨 앞에 추가하는 일을 한다.

```
public void addAfter(Node<T> before, T item) {  
    ...  
}  
...  
}
```

입력 받은 item을 저장하는 새로운 노드를 만들어 연결리스트에서  
before가 가리키는 노드 바로 뒤에 추가하는 일을 한다.

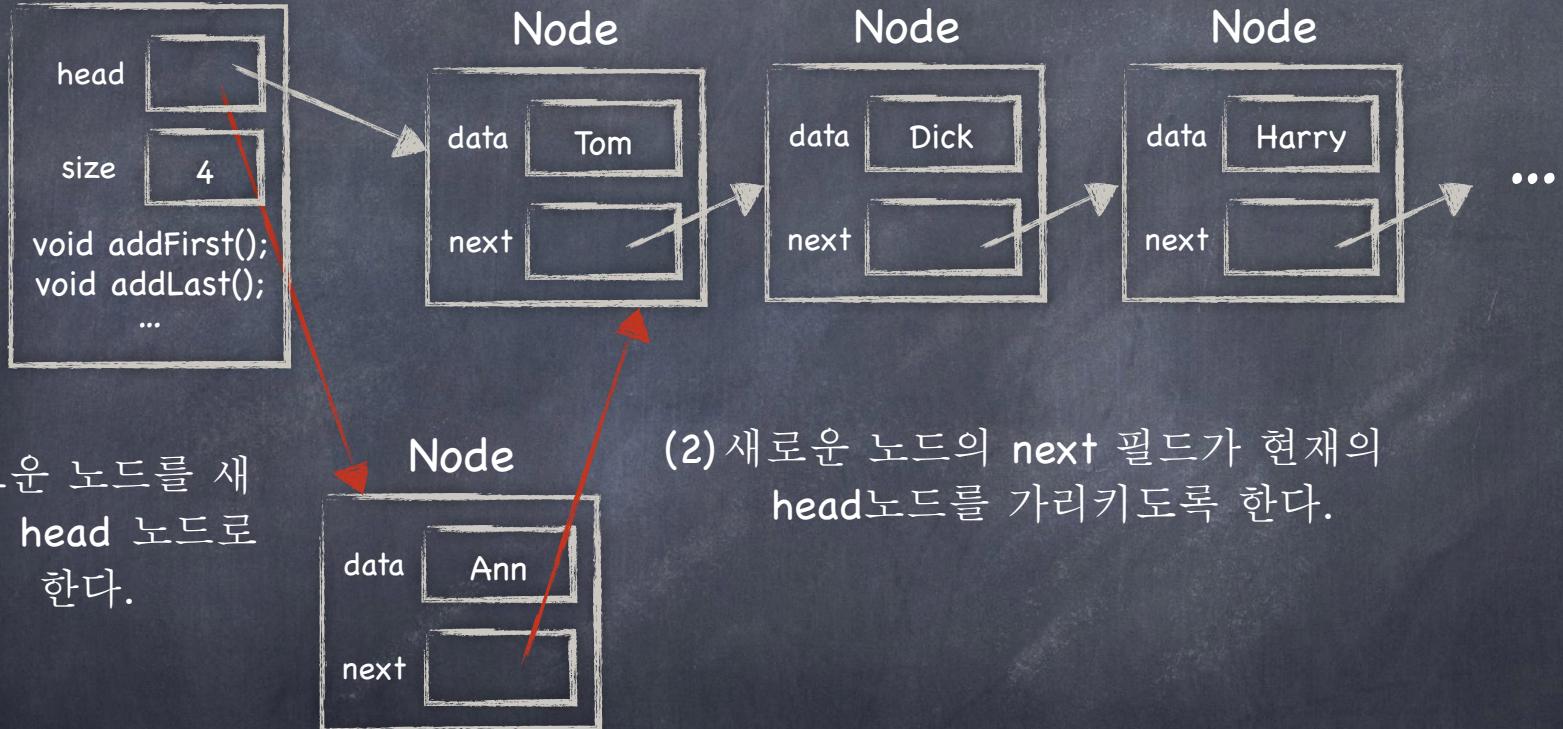
# 노드 연결

MySingleLinkedList



# addFirst("Ann")

MySingleLinkedList



(1) 새로운 노드를 만들고 추가할 데이터를 저장한다.

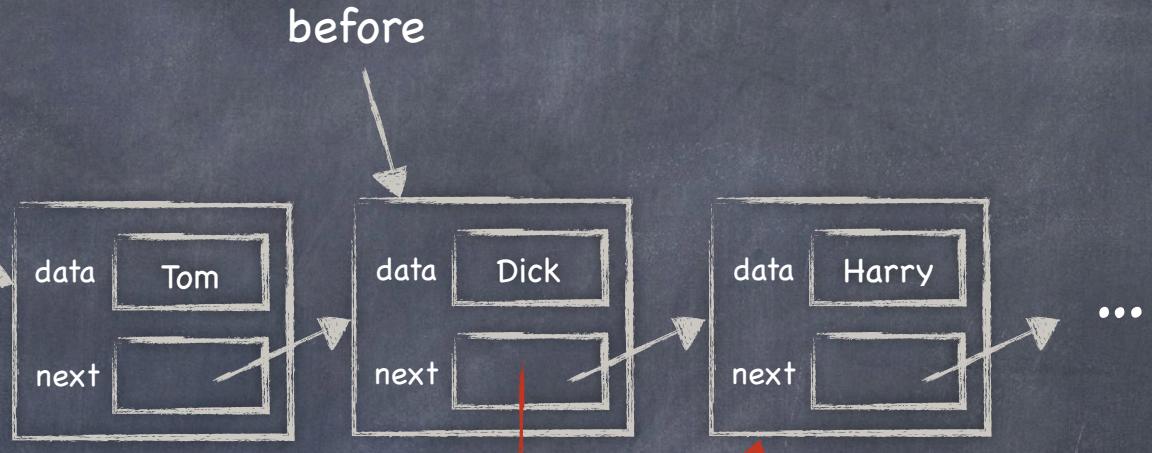
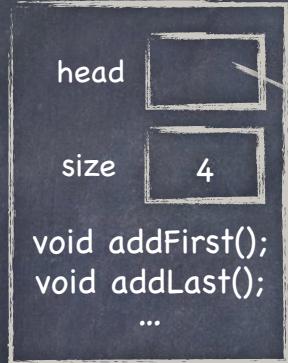
# addFirst(T item)

```
public void addFirst (T item) {  
    Node<T> temp = new Node<T>(item);  
    temp.next = head;  
    head = temp;  
    size++;  
}
```

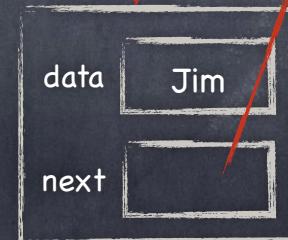
연결 리스트를 다루는 프로그램에서 가장 주의할 점은 내가 작성한 코드가 일반적인 경우만이 아니라 특수한 혹은 극단적인 경우에도 문제 없이 작동하는지 철저히 확인하는 것이다. 이 경우에는 기존의 연결 리스트의 크기가 0인 경우, 즉 `head`가 `null`인 경우에도 문제가 없는지 확인해야 한다. 문제가 없는가?

# addAfter(before, "Jim")

MySingleLinkedList



(3) 새로운 노드를  
**before**의 다음 노드  
로 만든다.



(2) 새로운 노드의 next 필드  
가 **before**의 다음 노드를  
가리키도록 한다.

(1) 새로운 노드를 만들고  
데이터를 저장한다.

# addAfter(Node<T> before, T item)

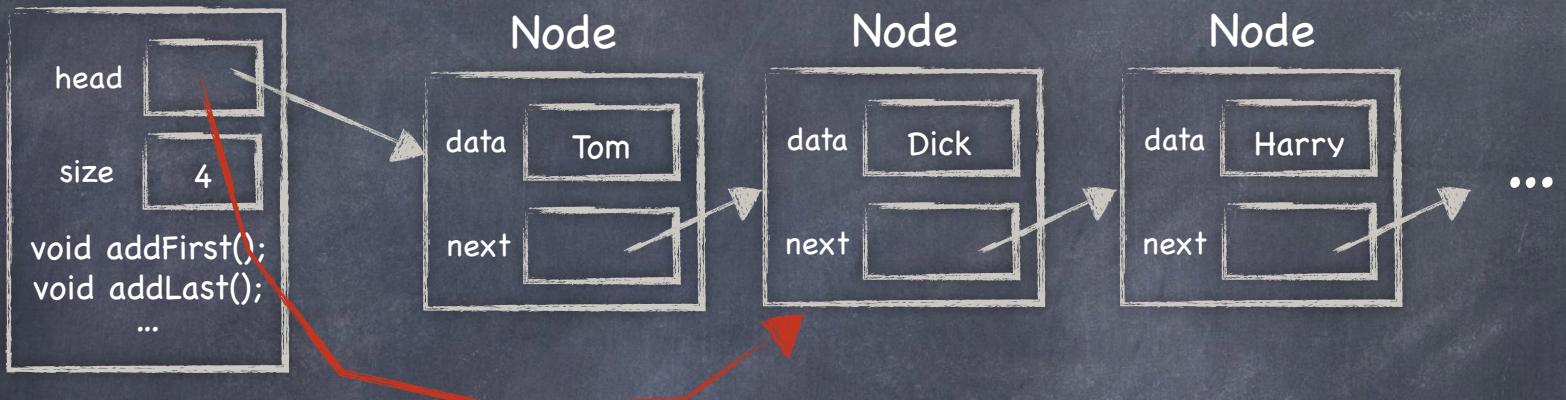
```
public void addAfter (Node<T> before, T item) {  
    Node<T> temp = new Node<T>(item);  
    temp.next = before.next;  
    before.next = temp;  
    size++;  
}
```

단순연결리스트에 새로운 노드를 삽입할 때 삽입할 위치의 바로 앞 노드의 주소가 필요하다. 즉 어떤 노드의 뒤에 삽입하는 것은 간단하지만, 반대로 어떤 노드의 앞에 삽입하는 것은 간단하지 않다.

# removeFirst()

연결리스트의 첫번째 노드를 삭제한다.

MySingleLinkedList



- (1) `head`가 `null`이 아니라면 `head`가 현재 `head` 노드의 다음 노드를 가리키게 만든다.

# removeFirst()

```
public T removeFirst () {  
    if (head == null) {  
        return null;  
    }  
    else {  
        Node<T> temp = head;  
        head = head.next;  
        size--;  
        return temp.data;  
    }  
}
```

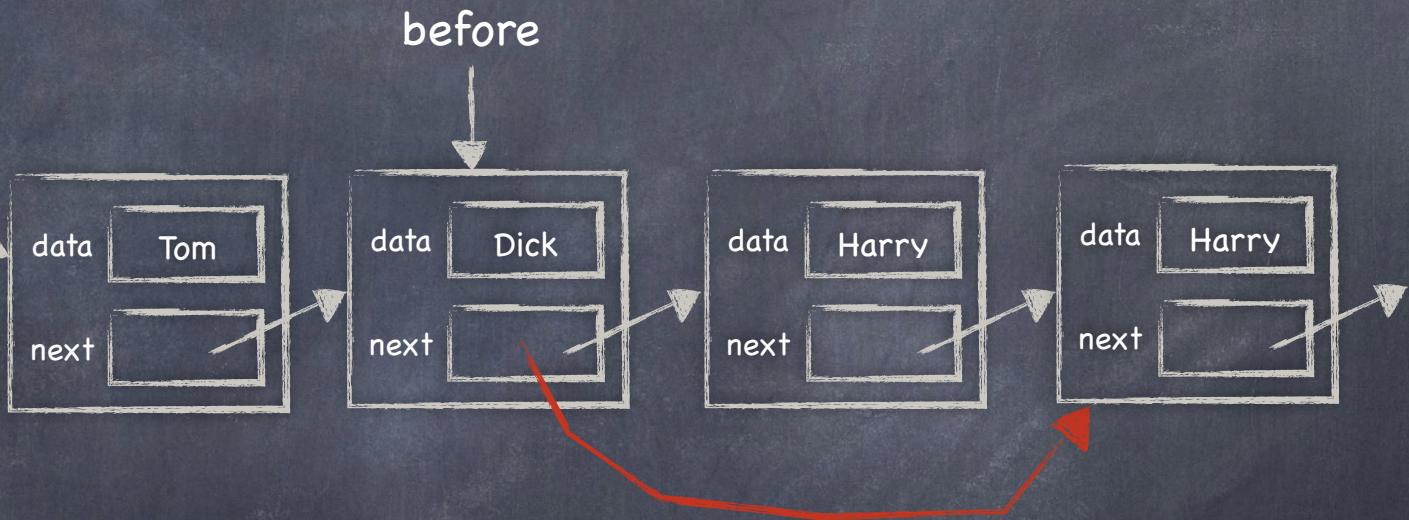
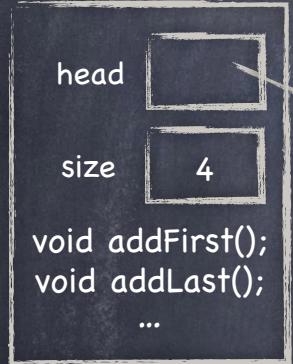
연결리스트의 첫번째 노드를 삭제하고 그 노드  
에 저장된 데이터를 반환한다.

head가 현재의 head 노드의 다음 노드를 가리키게 된다.

# removeAfter(Node<T> before)

`before`가 가리키는 노드의 다음 노드를 삭제한다.

MySingleLinkedList



(1) `before`의 다음 노드가 `null`이 아니라면  
`before`의 `next` 필드가 현재 `next` 노드의 다음  
노드를 가리키게 만든다.

# removeAfter(Node<T> before)

```
public T removeAfter (Node<T> before) {  
    Node<T> temp = before.next;  
    if (temp == null) {  
        return null;  
    }  
    else {  
        before.next = temp.next;  
        size--;  
        return temp.data;  
    }  
}
```

단순연결리스트에 어떤 노드를 삭제할 때는 삭제할 노드의 바로 앞 노드의 주소가 필요하다. 삭제할 노드의 주소만으로는 삭제할 수 없다.

# Traversing a Linked List

```
public int indexOf(T item) {  
    Node<T> p = head;  
    int index = 0;  
    while (p != null) {  
        if (p.data.equals(item))  
            return index;  
        p = p.next;  
        index++;  
    }  
    return -1;  
}
```

연결리스트의 노드들을 처음부터 순서대로 방문하는 것을 **순회** (**traverse**)한다고 말한다. `indexOf` 메서드는 입력된 데이터 `item`과 동일한 데이터를 저장한 노드를 찾아서 그 노드번호 (`index`)를 반환한다. 그것을 위해서 연결리스트를 순회한다.

# Traversing a Linked List

```
public Node<T> getNode(int index) {  
    if (index<0 || index>=size)  
        return null;  
    Node<T> p = head;  
    for (int i=0; i<index; i++)  
        p = p.next;  
    return p;  
}
```

연결리스트의 index번째 노드의 주  
소를 반환한다.

# get(int index)

index번째 노드에 저장된 데이터를 반환한다.

```
public T get (int index) {  
    if (index < 0 || index >= size)  
        return null;  
  
    Node<T> p = head;  
    for (int i=0; i<index; i++)  
        p = p.next;  
    return p.data;  
}
```

# void add(int index, T item)

연결리스트의 `index`번째 위치에 새로운 데이터를  
삽입한다.

```
public void add (int index, T item) {  
    if (index < 0 || index > size) {  
        return;  
    }  
    if (index == 0) {  
        addFirst(item);  
    }  
    else {  
        Node<T> node = getNode(index-1);  
        addAfter(node, item);  
    }  
}
```

# T remove(int index)

index번째 노드를 삭제하고, 그 노드에 저장된 데이터를 반환한다.

```
public T remove(int index) {  
    if (index < 0 || index >= size) {  
        return null;  
    }  
    if (index==0)  
        return removeFirst();  
    Node<T> prev = getNode(index-1);  
    return removeAfter(prev);  
}
```

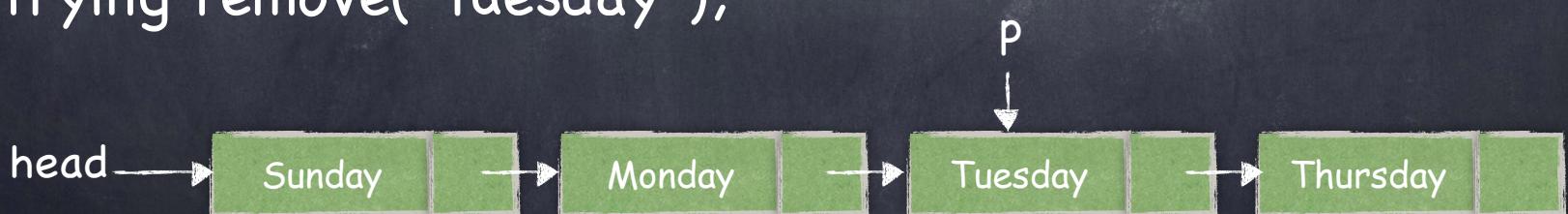
# T remove(T item)

```
public T remove(T item) {  
    Node<T> p = head;  
    while (p!=null && !p.data.equals(item))  
        p=p.next;  
  
    ???  
}
```

입력된 스트링을 저장한 노드를 찾아 삭제한다. 삭제된 노드에 저장된 스트링을 반환한다.

trying remove("Tuesday");

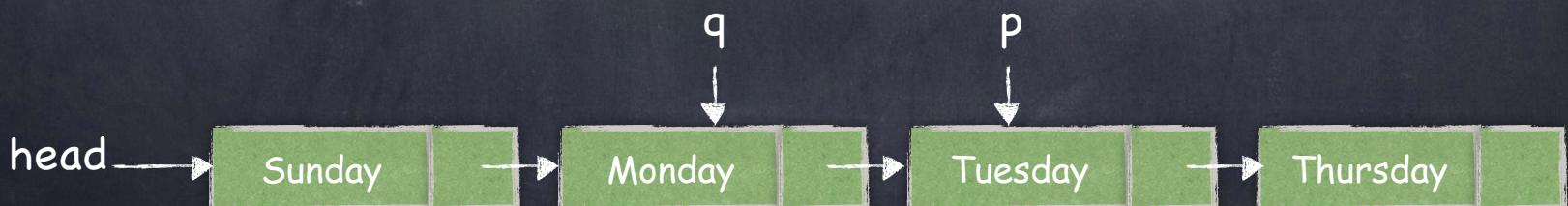
삭제할 노드를 찾았음. 하지만 노드를 삭제하기 위해서는 삭제할 노드가 아니라 직전 노드의 주소가 필요함.



# T remove(T item)

```
public T remove(T item) {  
    Node<T> p = head;  
    Node<T> q = null;  
    while (p!=null && !p.data.equals(item)) {  
        q = p;  
        p=p.next;  
    }  
    if (p==null)  
        return null;  
    if (q==null)  
        return removeFirst();  
    else  
        return removeAfter(q);  
}
```

q는 항상 p의 직전 노드를 가리킴. p가 첫번째 노드일 경우 q는 null임



# Test

```
public class TestLinkedList {  
    public static void main(String [] args) {  
        MySingleLinkedList<String> list = new MySingleLinkedList<>();  
        list.addFirst("Monday");  
        list.addFirst("Sunday");  
        list.add(2, "Saturday");  
        list.add(2, "Friday");  
        list.remove("Friday");  
        int index = list.indexOf("Saturday");  
        ...  
    }  
}
```



## 4.2 예제: 다항식 프로그램

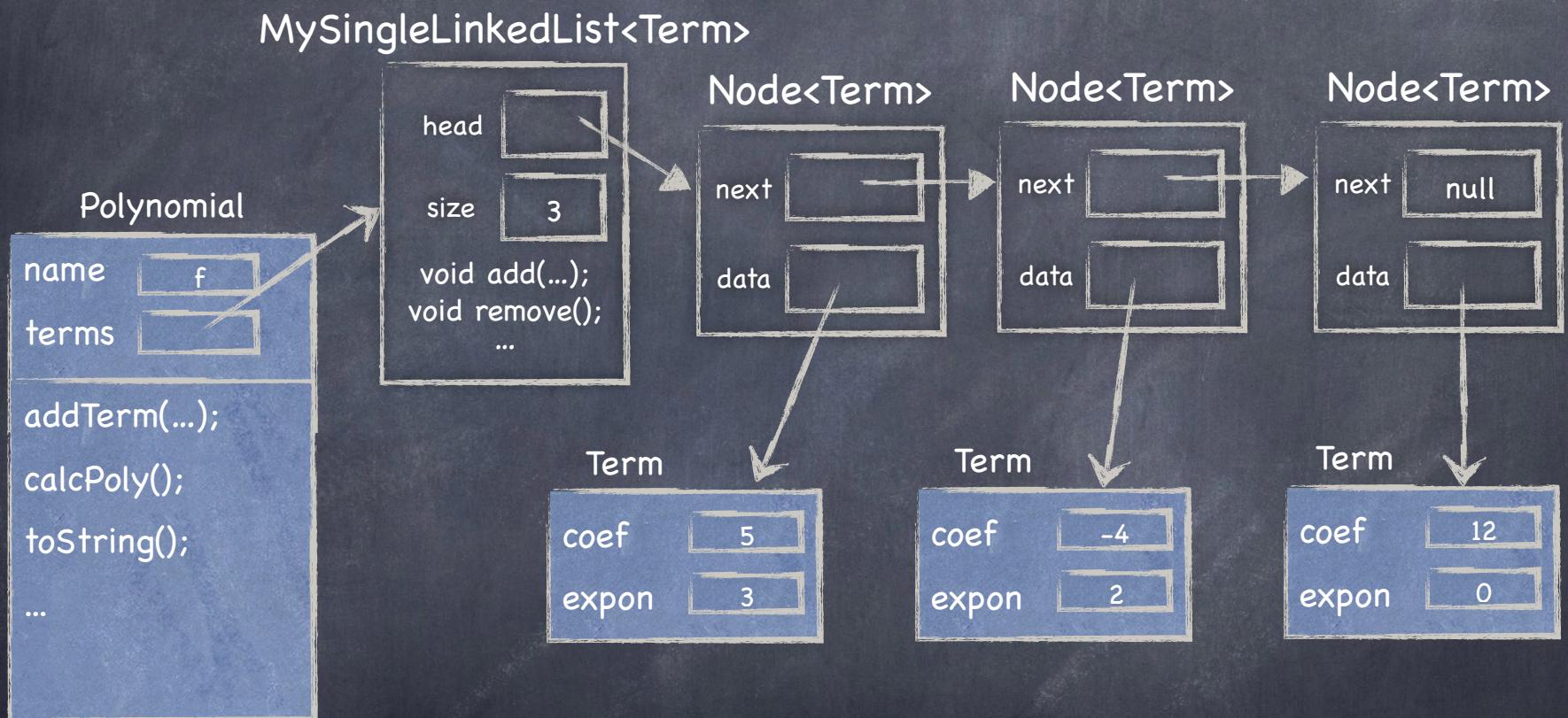
# 프로그램 실행 예

```
$ create f          // 다항함수  $f = 0$ 을 정의한다.  
$ add f 2 3        //  $f(x)$ 에  $2x^3$ 을 더한다. 따라서  $f(x) = 2x^3$  이 된다.  
$ add f -1 1       //  $f(x) = 2x^3 - x$  이 된다.  
$ add f 5 0        //  $f(x) = 2x^3 - x + 5$  이 된다.  
$ add f 2 1        //  $f(x) = 2x^3 - x + 5 + 2x = 2x^3 + x + 5$  이 된다.  
$ calc f 2          //  $x=2$ 일 때 다항함수  $f$ 의 값, 즉  $f(2) = 23$ 을 계산하여 출력한다.  
23  
$ print f          // 차수에 관한 내림차순으로 정렬하여 다음과 같이 출력한다.  
 $2x^3 + x + 5$       // 동일한 차수의 항은 하나로 합쳐져야 한다.  
$ create g          // 다른 다항 함수  $g$ 를 정의한다.  
....  
$ exit
```

# 다항식의 표현

- 연결리스트를 이용하여 하나의 다항식을 표현하는 클래스 `Polynomial` 을 작성한다.
- 다항식을 항들의 연결 리스트로 표현한다.
- 항들을 차수에 대해서 내림차순으로 정렬하여 저장하며, 동일 차수의 항을 2개 이상 가지지 않게 한다.
- 또한 계수가 0인 항을 가지지 않게 한다.
- 하나의 항은 계수와 지수에 의해 정의된다. 하나의 항을 표현하기 위해 클래스 `Term`을 정의한다.
- 클래스 `Polynomial`은 다항식을 계산하는 메서드와 출력하는 메서드를 제공한다.

# 자료구조



$$f(x) = 5x^3 - 4x^2 + 12$$

# class Term

```
public class Term {  
    public int coefficient;  
    public int exponent;  
    public Term(int coef, int exp) {  
        coefficient = coef;  
        exponent = exp;  
    }  
    public int eval(int x) {  
        return coefficient * (int) Math.pow(x, exponent);  
    }  
    public String toString() {  
        return coefficient + "x^" + exponent;  
    }  
}
```

이 `toString()` 메서드는 완벽하지 않다.  
개선하는 것은 과제로 남겨둔다.

# class Polynomial

```
public class Polynomial {  
    private char name;  
    private SingleLinkedList<Term> terms;  
  
    public Polynomial(char name) {  
        this.name = name;  
        terms = new SingleLinkedList<>();  
    }  
  
    public void addTerm( int c, int e ) {  
        ...  
    }  
  
    public int eval(int x) {  
        ...  
    }  
}
```

다항식에 하나의 새로운 항을 더하는 메서드이다. 새로운 항이 하나 생성될 수도 있고, 기존에 있던 항의 계수만 변경될 수도 있다.

변수 x의 값을 받아서 다항식의 값을 계산하여 반환한다.

# class Polynomial

```
public char getName() {  
    return name;  
}  
  
public String toString() {  
    ...  
}  
}
```

# addTerm(int coef, int expo)

- 기존의 다항식에 새로운 항을 추가하는 메서드
- 두 가지 경우

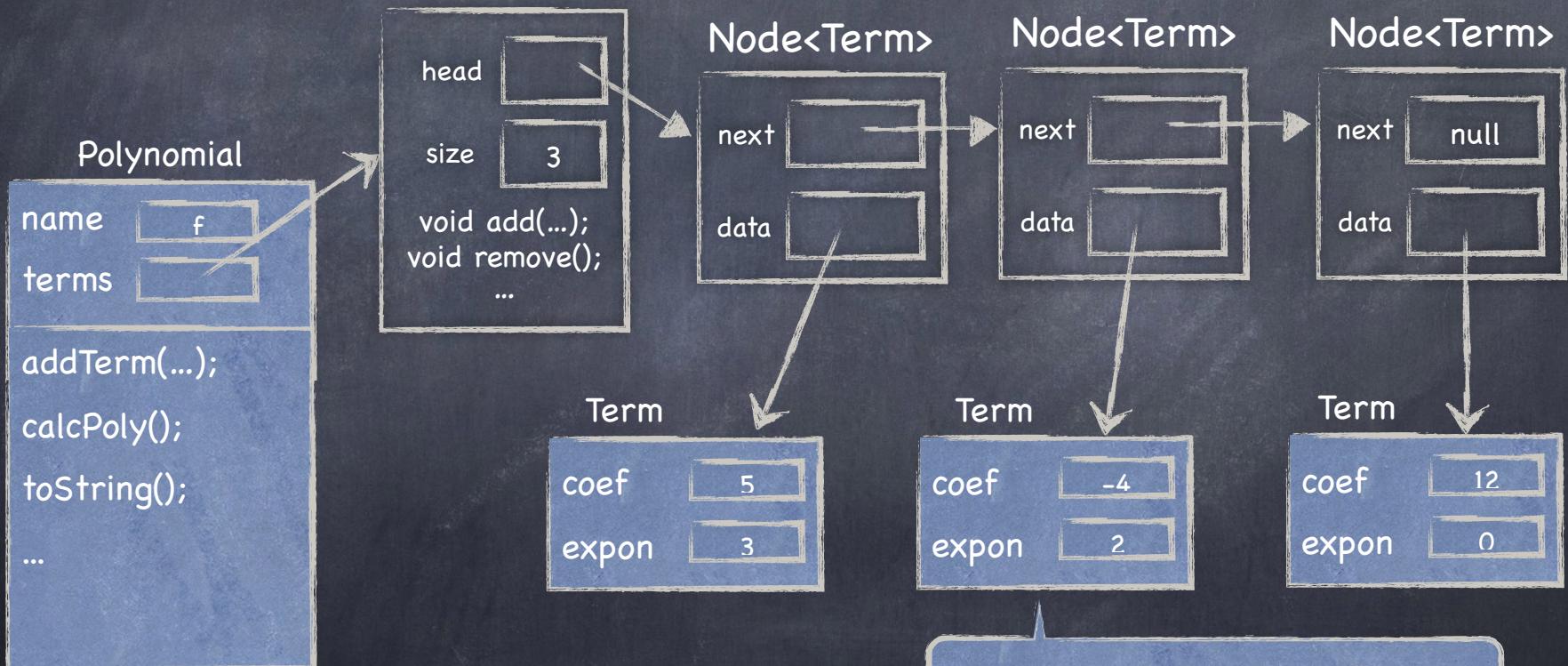
추가하려는 항과 동일 차수의 항이 이미 있는 경우: 기존 항의 계수만 변경

그렇지 않은 경우: 새로운 항을 삽입 (항들은 차수의 내림차순으로 항상 정렬 됨)

# addTerm(int coef, int expo)

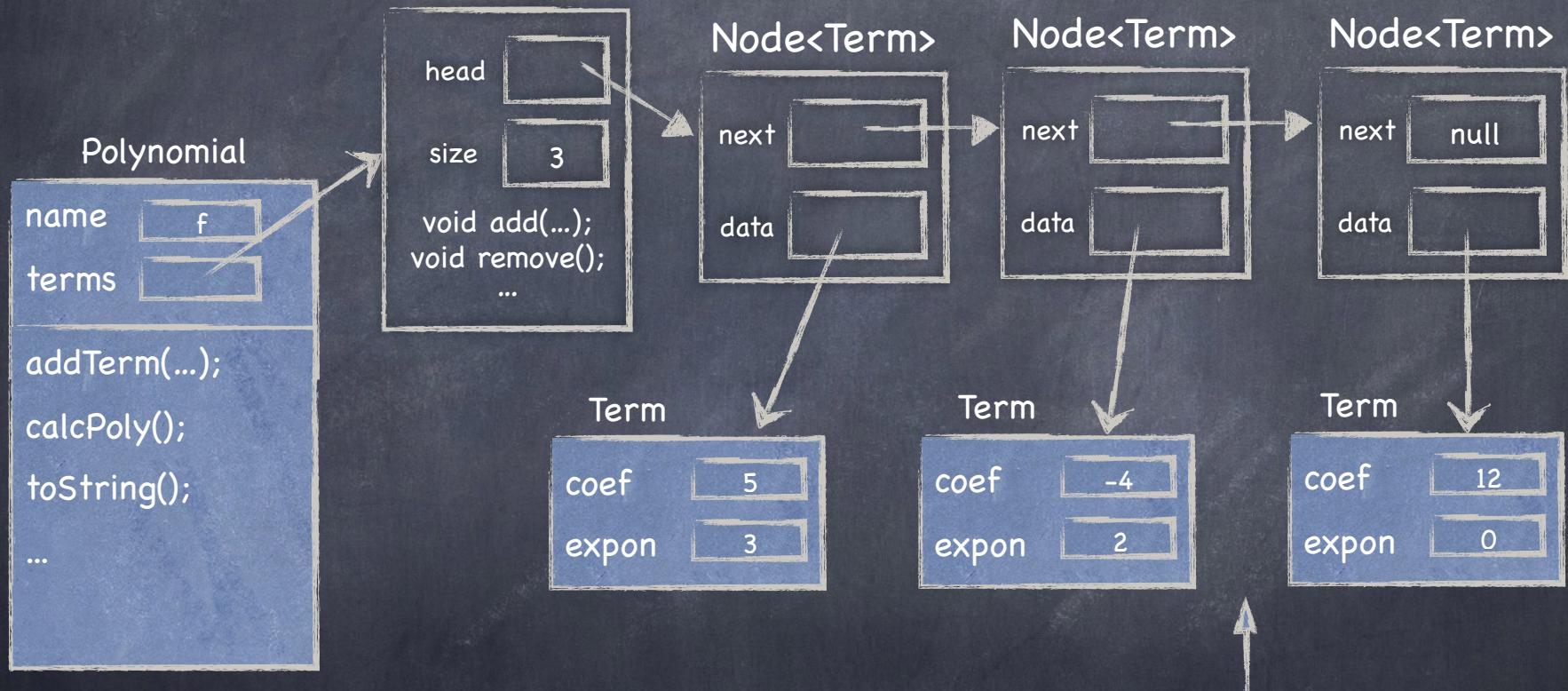
- 추가하려는 항과 동일 차수의 항이 이미 있는 경우: 기존 항의 계수만 변경

$$f(x) = 5x^3 - 4x^2 + 12$$



# addTerm(int coef, int expo)

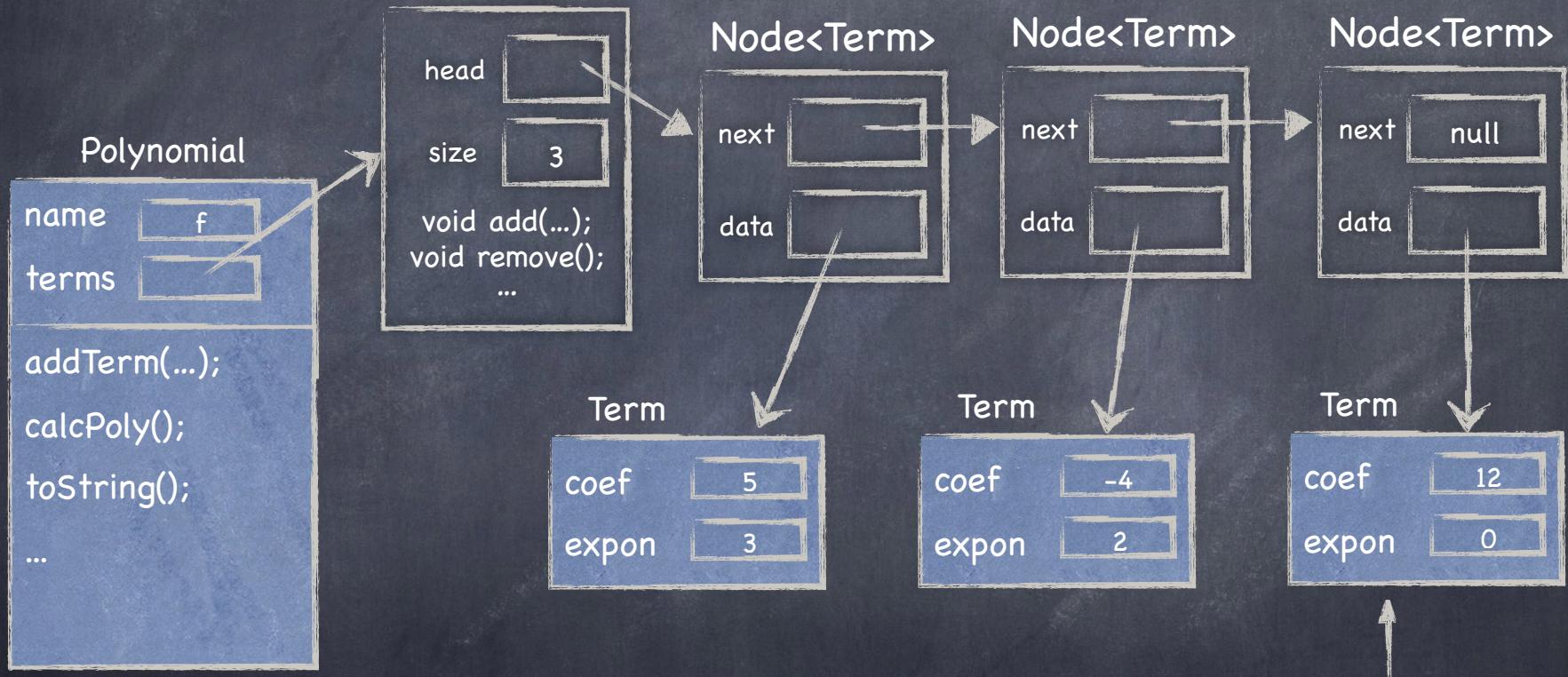
- 새로운 항을 삽입해야하는 경우: 예를 들어  $5x$ 를 삽입한다고 가정



새로운 항  $5x$ 가 삽입될 위치. 이 위치를 찾기 위해서는 새로운 항보다 차수가 작거나 같은 항이 나올때 까지 첫번째 항부터 순서대로 검사해야 한다.

# addTerm(int coef, int expo)

- 새로운 항을 삽입해야하는 경우: 예를 들어  $5x$ 를 삽입한다고 가정



새로운 항보다 차수가 작거나 같은 항이 나올때 까지 순서대로 검사하면 이 항에서 멈추게 된다. 하지만 연결리스트에서 노드를 삽입하기 위해서는 이 앞 노드의 주소가 필요하다.

그래서 1절의 `remove(T item)` 메서드에서와 같이 두 개의 변수 `p, q`를 이용한다.

# addTerm(int coef, int expo)

```
public void addTerm(int coef, int expo) {  
    if (coef == 0)  
        return;
```

```
Node<Term> p = terms.head, q=null;  
while (p != null && p.data.exponent > expo) {  
    q = p;  
    p = p.next;  
}  
q
```

q가 항상 p의 한 칸 뒤를 따라가도록 한다.

# addTerm(int coef, int expo)

```
if (p!=null && p.data.exponent == expo) {  
    p.data.coefficient += coef;  
    if (p.data.coefficient==0)  
        terms.removeAfter(q);  
}  
else {  
    Term t = new Term(coef, expo);  
    if (q==null)  
        terms.addFirst(t);  
    else {  
        terms.addAfter(q, t);  
    }  
}
```

동일 차수의 항이 존재하는 경우

더했더니 계수가 0이 되는 경우

# eval(int x)

```
public int eval(int x) {  
    int result = 0;  
    Node<Term> p = terms.head;  
    while(p != null) {  
        result += p.data.eval(x);  
        p = p.next;  
    }  
    return result;  
}
```

```
public char getName() {  
    return name;  
}
```

# toString()

```
public String toString()
{
    StringBuilder sb = new StringBuilder();
    sb.append(name + "(x)=");
    Term p = terms.head;
    while(p != null) {
        sb.append("+" + p.data.toString());
        p = p.next;
    }
    return sb.toString();
}
} // end of class Polynomial
```

이 `toString()` 메서드 역시 완벽하지 않다.  
개선하는 것은 과제로 남겨둔다.

# class MyPolynomialApp

```
public class MyPolynomialApp {  
    private static final int init_capacity = 100;  
    private Polynomial [] theList = new Polynomial [init_capacity];  
    private int n=0;  
    private Scanner kb = new Scanner(System.in);
```

# class MyPolynomialApp

```
public void processCommand() {  
    while(true) {  
        System.out.print("$ ");  
        String command = kb.next();  
  
        if (command.equals("print"))  
            handlePrint();  
        else if (command.equals("calc"))  
            handleCalc();  
        else if (command.equals("create"))  
            handleCreate();  
        else if (command.equals("add"))  
            handleAdd();  
        else if (command.equals("exit"))  
            break;  
    }  
    kb.close();  
}
```

# class MyPolynomialApp

```
private void handleCalc() {  
    char name = kb.next().charAt(0);  
    int x = kb.nextInt();  
    int index = find(name);  
    if (index < 0)  
        System.out.println("Undefined polynomial!");  
    else  
        System.out.println( theList[index].eval(x) );  
}  
  
private int find(char name) {  
    for (int i=0; i<n; i++) {  
        if (theList[i].getName() == name)  
            return i;  
    }  
    return -1;  
}
```

# class MyPolynomialApp

```
private void handlePrint() {
    char name = kb.next().charAt(0);
    int index = find(name);
    if (index>-1)
        System.out.println(theList[index].toString());
    else
        System.out.println("Undefined polynomial !");
}

private void insert(Polynomial polynomial)  {
    int index = find(polynomial.getName());
    if (index>-1)
        theList[index] = polynomial;
    else {
        if (theList.length<=n)
            reallocate();
        theList[n++] = polynomial;
    }
}
```

# class MyPolynomialApp

```
private void handleCreate() {  
    char name = kb.next().charAt(0);  
    Polynomial p = new Polynomial( name );  
    insert( p );  
}  
  
private void handleAdd() {  
    char name = kb.next().charAt(0);  
    int coef = kb.nextInt();  
    int expo = kb.nextInt();  
    int index = find( name );  
    if (index < 0)  
        System.out.println("Undefined polynomial!");  
    else  
        theList[index].addTerm( coef, expo );  
}
```

# class MyPolynomialApp

```
private void reallocate() {  
    Polynomial [] tmp = new Polynomial [2*theList.length];  
    System.arraycopy(theList, 0, tmp, 0, theList.length);  
    theList = tmp;  
}  
  
public static void main(String [] args)  
{  
    MyPolynomialApp theApp = new MyPolynomialApp();  
    theApp.processCommand();  
}  
}
```

## 4.3 연결리스트와 Iterator

# 객체지향 프로그래밍

- Information Hiding
- Data Encapsulation
- Abstract Data Type

인터페이스(Interface)와  
구현(implementation)의 분리

# 인터페이스와 구현의 분리

- 연결 리스트는 “리스트”라는 추상적인 데이터 타입을 구현하는 한 가지 방법일 뿐이다. 가령 배열 혹은 ArrayList는 또 다른 구현 방법의 예이다.
- 사용자는 리스트에 데이터를 삽입, 삭제, 검색할 수 있으면 된다. 그것의 구현에 대해서 세부적으로 알 필요는 없다.

사용자가 필요로 하는 이런 기능들을 public method들로 제공한다.

이 public method들은 가능한 한 내부 구현과 독립적이어야 한다.

- 인터페이스와 구현을 분리하면 코드의 모듈성(modularity)가 증가하며, 코드의 유지/보수, 코드의 재사용이 용이해진다.

# 인터페이스와 구현의 분리

## Implementation

```
Node<T> head;  
class Node<T> {  
    T data;  
    Node<T> next;  
};
```



연결리스트는 리스트라는 추상적인 데이터 타입을 구현하는 한 가지 방법일 뿐이다.

사용자에게 제공하는 **Interface**에는 리스트가 연결리스트로 구현되어 있다는 사실이 드러나지 않는다.

## Interface

```
T get(int index);  
void add(int index, T item);  
T remove(int index);  
boolean remove(T item);  
int indexOf(T item);  
int size();
```

사용자는 **Interface**만 알면 되고 **Implementation**에 대해서는 알 필요도 없고 알 수도 없게 ...

# 인터페이스와 구현의 분리

## Implementation

```
T [] data = (T []) new T [100];  
int size = 0;  
int capacity = 100;  
  
void reallocate() { ... }
```

data 

## Interface

```
T get(int index);  
void add(int index, T item);  
T remove(int index);  
boolean remove(T item);  
int indexOf(T item);  
int size();
```

배열을 이용해서 리스트를 구현하더라도 사용자에게는 동일한 **interface**를 제공한다.

배열은 리스트라는 추상적인 데이터 타입을 구현하는 또 다른 대표적인 한 가지 방법이다.

사용자는 **Interface**만 알면 되고 **Implementation**에 대해서는 알 필요도 없고 알 수도 없게...

# List Abstract Data Type

MySingleLinkedList 클래스가 제공할 인터페이스,  
즉 public method들이다.

```
T get(int index);  
void add(int index, T item);  
T remove(int index);  
boolean remove(T item);  
int indexOf(T item);  
int size();  
Iterator<T> iterator();
```

Java API는 List<E> 인터페이스를 정의하고 있다.  
<https://docs.oracle.com/javase/8/docs/api/java/util/List.html>  
MySingleLinkedList에서는 List<E> 인터페이스의 일부만을 제공한다.

# 연결리스트의 순회: Iterator의 필요성

- ☞ class MySingleLinkedList의 외부에서 연결리스트를 순회하려면?

```
MySingleLinkedList<String> aList = new MySingleLinkedList<String>();  
aList.add("Some string");
```

....

```
// 연결리스트를 순회하면서  
// 각 노드에 저장된 데이터에 대해서 어떤 일을 하고 싶다면...
```

```
Node p = aList.head;  
while( p!=null ) {  
    String str = p.data;  
    // do something with str  
    p = p.next;  
}
```

head와 class Node의 멤버들이 모두  
public으로 공개되어야 한다.  
이것은 바람직하지 않다.

# 연결리스트의 순회: Iterator의 필요성

- ☞ `class MySingleLinkedList`의 public 메서드들만을 이용하여 연결리스트를 순회하려면 ?

```
MySingleLinkedList<String> aList = new MySingleLinkedList<String>();
aList.add("Some string");
....
```

```
// 연결리스트를 순회하면서
// 각 노드에 저장된 데이터에 대해서 어떤 일을 하고 싶다면…
```

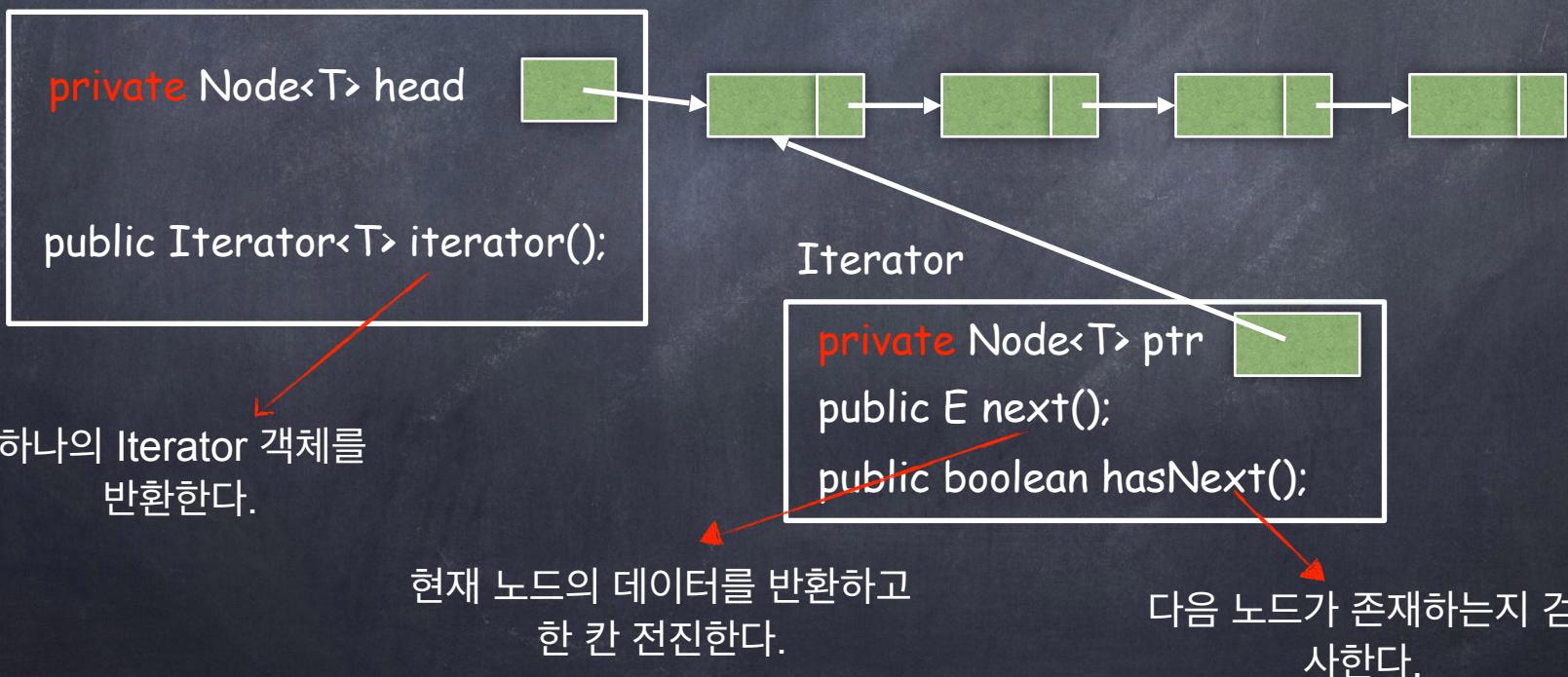
```
for (int i=0; i<aList.size(); i++)
    String str = aList.get(i);
    // do something with str
}
```

만약 `aList`가 `ArrayList`라면 이 코드는 별 문제가 없다. 하지만 연결리스트에서 이렇게 하는 것은 매우 비효율적이다. 왜냐하면 `get(i)` 메서드는 호출될 때마다 매번 연결리스트의 첫 번째 노드에서 시작하여 `i`번째 노드를 새로 찾아 가기 때문이다.

# Iterator

- 노드의 주소를 직접 사용자에게 제공하는 대신 그것을 **private** 멤버로 wrapping하고 있으면서 사용자가 필요로 하는 일을 (노드에 저장된 데이터를 엑세스하거나 한 칸 전진하는 일 등) 해주는 **public method**를 가진 **Iterator** 객체를 제공한다.

Class MySingleLinkedList



# 연결리스트의 순회

## Iterator를 이용한 순회

```
MySingleLinkedList<String> aList = new MySingleLinkedList<String>();  
aList.add("Some string");
```

....

```
// 연결리스트를 순회하면서  
// 각 노드에 저장된 데이터에 대해서 어떤 일을 하고 싶다면...
```

```
Iterator<String> iter = aList.iterator();  
while( iter.hasNext() ) {  
    String str = iter.next();  
    // do something with str  
}
```

Iterator의 hasNext() 메서드는 연결리스트의 끝에 도달하면 false를 반환하며, next() 메서드는 현재 노드에 저장된 데이터를 반환하고 자신은 한 칸 전진한다.

# Iterator<E> 인터페이스

- java.util의 Iterator<E> 인터페이스를 정의

Method	Behavior
boolean hasNext()	Returns true if the next method returns a value.
E next()	Returns the next element. If there are no more elements, throws the NoSuchElementException.
void remove()	Removes the last element returned by the next method.

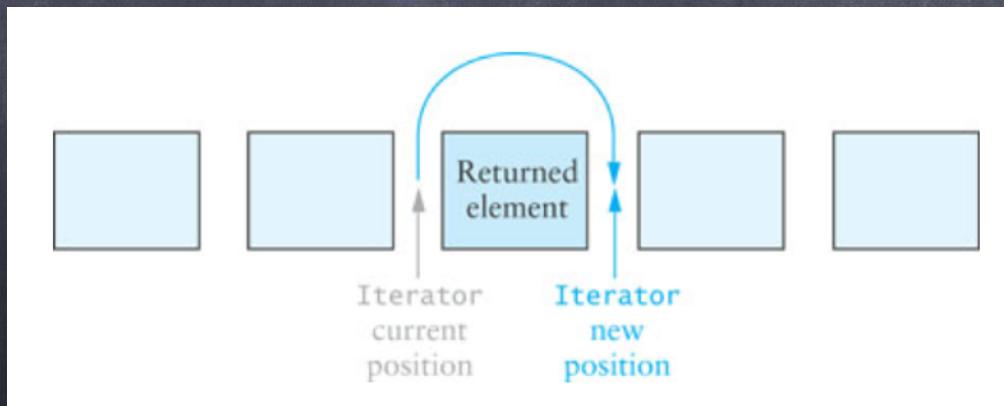
- MyLinkedList 클래스는 메서드 iterator()를 제공

```
public Iterator<E> iterator();
```

- MySingleLinkedList<E> 클래스는 Iterator<E> 인터페이스를 구현하는 하나의 클래스를 inner 클래스로 가지며, iterator() 메서드는 이 클래스의 객체를 생성하여 반환한다.

# 연결리스트에서의 Iterator

- Iterator는 개념적으로는 연결리스트의 노드와 노드 사이를 가리킨다.
- 초기상태의 iterator는 첫 번째 노드의 앞 위치를 가리킨다.
- next() 메서드는 한 칸 전진하면서 방금 지나친 노드의 데이터를 반환
- hasNext() 메서드는 다음 노드가 존재하면 true, 그렇지 않으면 false를 반환한다.
- remove() 메서드는 가장 최근에 next() 메서드로 반환한 노드를 삭제한다.



# remove()

```
public static void removeDivisibleBy(LinkedList<Integer> aList, int div) {  
    Iterator<Integer> iter = aList.iterator();  
    while (iter.hasNext()) {  
        int nextInt = iter.next();  
        if (nextInt % div == 0) {  
            iter.remove();  
        }  
    }  
}
```

aList에 저장된 정수들 중 div로 나누어 나누어지는 것들을 모두 삭제한다.

# MySingleLinkedList의 수정

- Node 클래스를 MySingleLinkedList 클래스의 private inner 클래스로 만든다.

MySingleLinkedList 클래스의 외부에서는 Node 클래스를 사용할 수 없게 만든다.

- MySingleLinkedList 클래스의 사용자는 단지 add, get, remove, indexOf, size 등의 public 메서드와 iterator를 통해서만 연결리스트에 접근할 수 있게 만든다.

MySingleLinkedList 클래스의 사용자는 이 클래스의 내부 구현에 대해서 아무것도 알 필요가 없다. 심지어는 내부가 연결리스트로 구현되어 있는지 배열로 구현되어 있는지도 알 필요가 없다.

# Interface Iterator<T>

Iterator 인터페이스는 Java API가 이미 제공하므로 우리가 정의할 필요는 없다.

```
public interface Iterator<T> {  
    public boolean hasNext();  
    public T next();  
    public void remove();  
}
```

# class MySingleLinkedList<T>

```
public class SingleLinkedList<T> {  
    private Node<T> head;  
    private int size;  
  
    public SingleLinkedList() {  
        head = null;  
        size = 0;  
    }  
  
    private static class Node<E> {  
        public E data;  
        public Node<E> next;  
        public Node(E item) {  
            data = item;  
            next = null;  
        }  
    }  
}
```

inner class의 type parameter가 바깥 클래스의 type parameter와 같은 이름일 필요는 없다.

클래스 Node를 private inner 클래스로 만들었다.  
private 멤버이므로 MySingleLinkedList 클래스의 외부  
에서는 class Node를 사용할 수 없다.  
inner 클래스 내부에서 바깥 클래스의 non-static 멤버  
를 access할 필요가 없을 때는 이렇게 static 클래스로  
만드는 것이 좋다.

# class MySingleLinkedList<T>

```
private class MyIterator implements Iterator<T> {
    private Node<T> nextNode;
    public MyIterator() {
        nextNode = head;
    }
    public boolean hasNext() {
        return (nextNode != null);
    }
    public T next() {
        if (nextNode == null)
            throw new NoSuchElementException();
        T val = nextNode.data;
        nextNode = nextNode.next;
        return val;
    }
    public void remove() {
        // ?
    }
}
```

Non-static inner 클래스의 경우 바깥 클래스의 type parameter T를 공유하므로 MyIterator<T>와 같이 generics로 정의할 필요는 없다.

바깥 클래스의 non-static 멤버인 head를 access하므로 MyIterator 클래스는 non-static 클래스로 만들었다.

remove 메서드를 구현하는 것은 다소 까다롭다.  
일단 나중으로 미뤄둔다.

# class MySingleLinkedList<T>

```
public Iterator<T> iterator() {  
    return new MyIterator<T>();  
}  
  
private void addFirst(T item) {  
    Node<T> temp = new Node<T>(item);  
    temp.next = head;  
    head = temp;  
    size++;  
}  
  
private void addAfter(Node<T> before, T item) {  
    Node<T> temp = new Node<T>(item);  
    temp.next = before.next;  
    before.next = temp;  
    size++;  
}
```

# class MySingleLinkedList<T>

```
private T removeFirst() {
    if (head == null)
        return null;
    else {
        Node<T> temp = head;
        head = head.next;
        size--;
        return temp.data;
    }
}

private T removeAfter( Node<T> before ) {
    Node<T> temp = before.next;
    if (temp == null)
        return null;
    else {
        before.next = temp.next;
        size--;
        return temp.data;
    }
}
```

# class MySingleLinkedList<T>

```
public int indexOf(T word) {  
    Node<T> p = head;  
    int index = 0;  
    while (p != null) {  
        if (p.data.equals(word))  
            return index;  
        p = p.next;  
        index++;  
    }  
    return -1;  
}  
  
private Node<T> getNode(int index) {  
    if (index<0 || index>=size)  
        return null;  
    Node<T> p = head;  
    for (int i=0; i<index; i++)  
        p = p.next;  
    return p;  
}
```

# class MySingleLinkedList<T>

```
public void add (int index, T item) {  
    if (index < 0 || index > size)  
        throw new IndexOutOfBoundsException();  
    if (index == 0)  
        addFirst(item);  
    else {  
        Node<T> node = getNode(index-1);  
        addAfter(node, item);  
    }  
}
```

IndexOutOfBoundsException을 throw하도록  
수정하였다.

```
public T remove(int index) {  
    if (index < 0 || index >= size)  
        throw new IndexOutOfBoundsException();  
    if (index==0)  
        return removeFirst();  
    Node<T> prev = getNode(index-1);  
    return removeAfter(prev);  
}
```

# class MySingleLinkedList<T>

```
public boolean remove(T item) {
    Node<T> p = head;
    Node<T> q = null;
    while (p!=null && !p.data.equals(item)) {
        q = p;
        p=p.next;
    }
    if (p==null)
        return false;
    if (q==null)
        removeFirst();
    else
        removeAfter(q);
    return true;
}

public int size() {
    return size;
}
```

Java의 List<T> 인터페이스는 삭제할 데이터가 존재하지 않 을 경우 false를 반환하도록 정의되어 있다.

# class Polynomial에서

```
public int eval(int x) {  
    int result = 0;  
    Iterator<Term> iter = terms.iterator();  
    while(iter.hasNext()) {  
        Term t = iter.next();  
        result += t.eval(x);  
    }  
    return result;  
}  
  
public String toString()  
{  
    StringBuilder sb = new StringBuilder();  
    Iterator<Term> iter = terms.iterator();  
    while(iter.hasNext()) {  
        Term t = iter.next();  
        sb.append("+" + t.toString());  
    }  
    return sb.toString();  
}
```

Iterator를 이용하여 연결리스트를 순회한다.

# class Polynomial에서

```
public void addTerm(int coef, int expo) {  
    if (coef == 0)  
        return;  
    Iterator<Term> p = terms.iterator(), q=null;  
    Term t = null;  
    int index = 0;  
    while (p.hasNext()) {  
        if (q == null)  
            q = terms.iterator();  
        else  
            q.next();  
        t = p.next();  
        index++;  
        if (t.exponent <= expo)  
            break;  
    }  
}
```

`Iterator`는 연결리스트를 단순하게 순회하는 정도의 용도에 적합하며 `addTerm` 메서드처럼 복잡한 작업을 하는 경우 그다지 적절하지 않다고 할 수 있다. 그래서 이런 경우에는 `ListIterator`를 사용하는 것이 좋다. `ListIterator`에 대해서는 나중에 다룬다.

# class Polynomial에서

```
if (t!=null && t.exponent==expo) {  
    t.coefficient += coef;  
    if (t.coefficient == 0)  
        terms.remove(index);  
}  
else {  
    Term term = new Term(coef, expo);  
    terms.add(index, term);  
}  
}
```

## 4.4 이중연결리스트와 `listIterator`

E.B. Koffman and P. Wolfgang, *Objects, Abstraction, Data Structures and Design using Java*, John Wiley & Sons, 2005.

# 이중연결리스트

## ① 단방향 연결 리스트의 한계:

단방향의 순회만이 가능

어떤 노드의 앞에 새로운 노드를 삽입하기 어려움

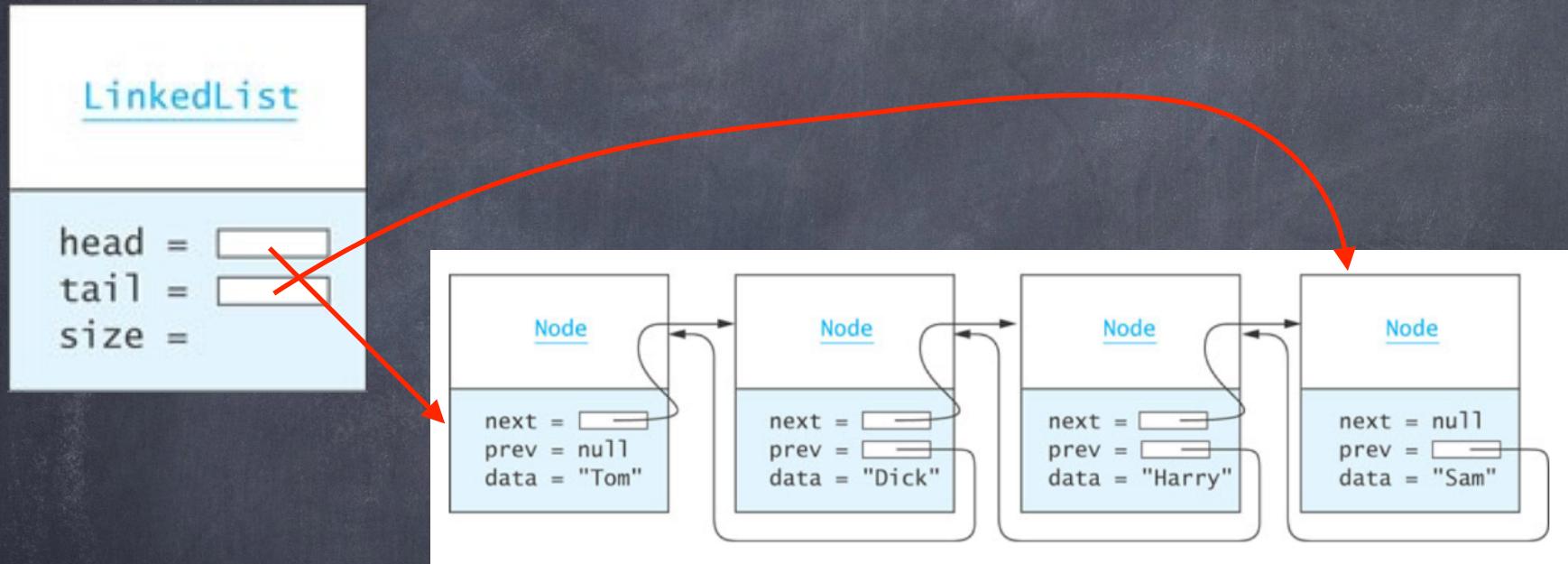
삭제의 경우에 항상 삭제할 노드의 앞노드가 필요

## ② 이중 연결 리스트

각각의 노드가 다음(next) 노드와 이전(previous) 노드의 주소를 가지는 연결 리스트

양방향의 순회가 가능

# 이중연결리스트



LinkedList 클래스는 첫 번째 노드와 마지막 노드를 참조하는  
head와 tail 필드를 가진다.

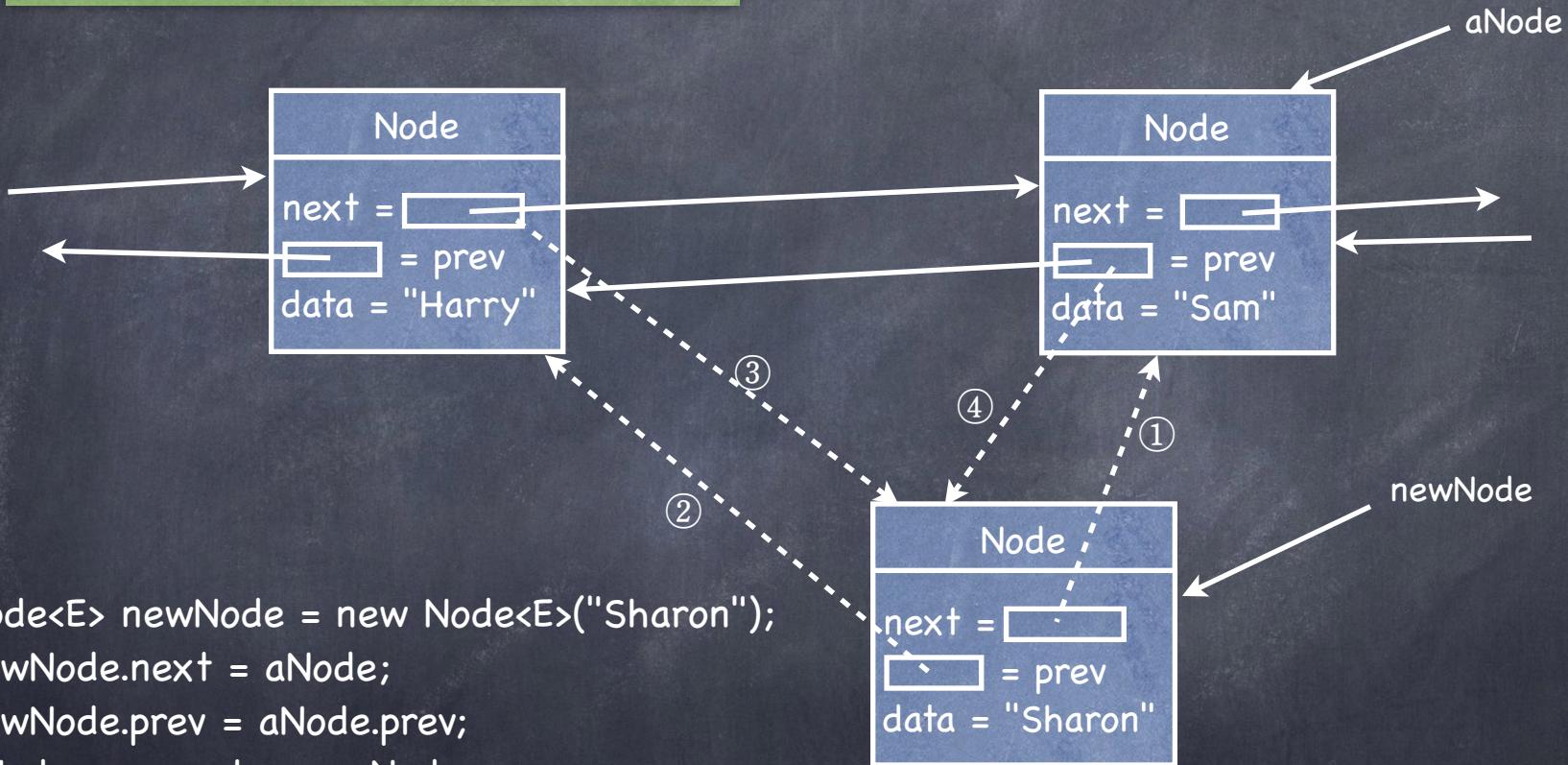
# class Node

```
private static class Node<E> {  
    private E data;  
    private Node<E> next = null;  
    private Node<E> prev = null;  
  
    private Node(E dataItem) {  
        data = dataItem;  
    }  
}
```

class Node를 LinkedList 클래스의  
private static inner class로 만든다.

# 노드 삽입

`aNode`가 가리키는 노드 앞에 새로운 노드를 삽입하는 경우

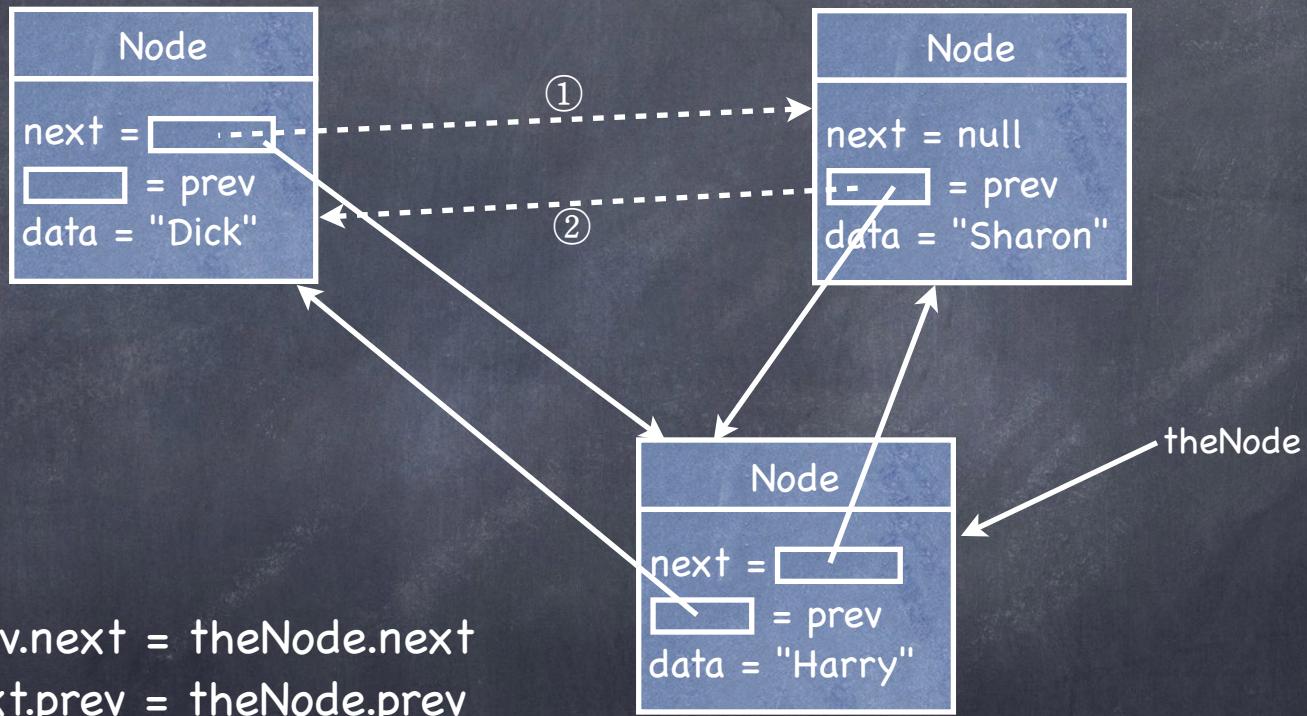


```
Node<E> newNode = new Node<E>("Sharon");
```

- ① `newNode.next = aNode;`
- ② `newNode.prev = aNode.prev;`
- ③ `aNode.prev.next = newNode;`
- ④ `aNode.prev = newNode`

# 노드 삭제

theNode가 가리키는 노드를 삭제하는 경우



# ListIterator<E> 인터페이스

## ② Iterator의 한계

단방향으로만 순회할 수 있다.

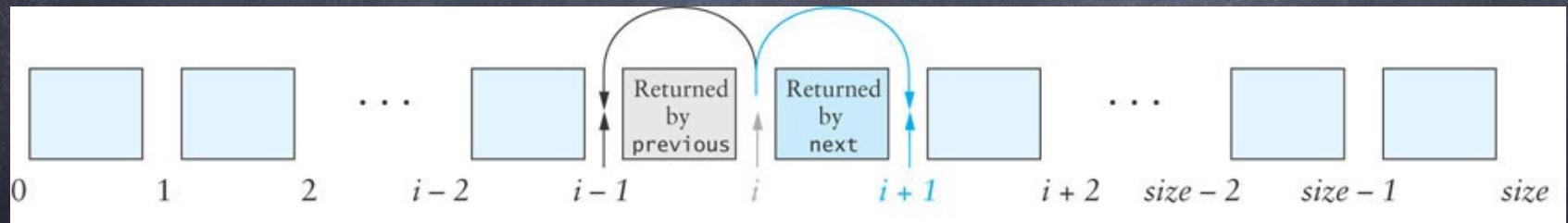
`remove()` 메서드는 지원하지만 `add()` 메서드를 지원하지 않는다.

항상 리스트의 처음에서 시작한다.

## ③ ListIterator는 Iterator를 확장한다.

# ListIterator 인터페이스

- Iterator 처럼 ListIterator 역시 개념적으로는 노드와 노드 사이를 가리킨다.
- ListIterator의 위치는 0에서 size까지의 인덱스로 표현한다.



# ListIterator 인터페이스

Method	Behavior
void add(E obj)	Inserts object obj into the list just before the item that would be returned by the next call to method <code>next</code> and after the item that would have been returned by method <code>previous</code> . If method <code>previous</code> is called after <code>add</code> , the newly inserted object will be returned.
boolean hasNext()	Returns <code>true</code> if <code>next</code> will not throw an exception.
boolean hasPrevious()	Returns <code>true</code> if <code>previous</code> will not throw an exception.
E next()	Returns the next object and moves the iterator forward. If the iterator is at the end, the <code>NoSuchElementException</code> is thrown.
int nextIndex()	Returns the index of the item that will be returned by the next call to <code>next</code> . If the iterator is at the end, the list size is returned.
E previous()	Returns the previous object and moves the iterator backward. If the iterator is at the beginning of the list, the <code>NoSuchElementException</code> is thrown.
int previousIndex()	Returns the index of the item that will be returned by the next call to <code>previous</code> . If the iterator is at the beginning of the list, <code>-1</code> is returned.
void remove()	Removes the last item returned from a call to <code>next</code> or <code>previous</code> . If a call to <code>remove</code> is not preceded by a call to <code>next</code> or <code>previous</code> , the <code>IllegalStateException</code> is thrown.
void set(E obj)	Replaces the last item returned from a call to <code>next</code> or <code>previous</code> with obj. If a call to <code>set</code> is not preceded by a call to <code>next</code> or <code>previous</code> , the <code>IllegalStateException</code> is thrown.

# listIterator() 메서드

- >List<E> 인터페이스는 iterator() 메서드와 함께 ListIterator를 만들어 주는 listIterator() 메서드를 요구한다.

Method	Behavior
public ListIterator<E> listIterator()	Returns a ListIterator that begins just before the first list element.
public ListIterator<E> listIterator(int index)	Returns a ListIterator that begins just before position index.

# Outline

```
public class KWLinkedList <E> {  
  
    private Node <E> head = null;  
    private Node <E> tail = null;  
    private int size = 0;  
  
    private static class Node<E> {  
        ...  
    }  
  
    private class KWListIterator implements ListIterator<E> {  
        ...  
    }  
}
```

ListIterator<E> 인터페이스는 Iterator<E>의 서브 인터페이스이므로 KWListIterator 클래스는 Iterator<E> 인터페이스도 구현하는 셈이다.

# Outline

Iterator<E> 인터페이스를 구현하는 하나의  
KWListIterator 객체를 생성하여 반환

```
public Iterator<E> iterator() {  
    return new KWListIterator(0);  
}
```

ListIterator<E> 인터페이스를 구현하는 하나의  
KWListIterator 객체를 생성하여 반환

```
public ListIterator<E> listIterator() {  
    return new KWListIterator(0);  
}
```

```
public ListIterator<E> listIterator(int index) {  
    return new KWListIterator(index);  
}
```

ListIterator<E> 인터페이스를 구현하는 하나의  
KWListIterator 객체를 생성하여 반환. 단 index번  
째 노드를 자신의 next노드로 가지는 상태

# Outline

```
public E get(int index) { ... }  
public void add(int index, E obj) { ... }  
public int indexOf(E obj) { ... }  
public E remove(int index) { ... }  
public boolean remove(E obj) { ... }  
public int size() { ... }  
}
```

# Get

```
public E get(int index) {  
    return listIterator(index).next();  
}
```

# Add

```
public void add(int index, E obj) {  
    listIterator(index).add(obj);  
}
```

index번째 노드를 next노드로 하는 listIterator를 만든 후  
그것의 add 메서드를 호출한다.

# indexOf, remove

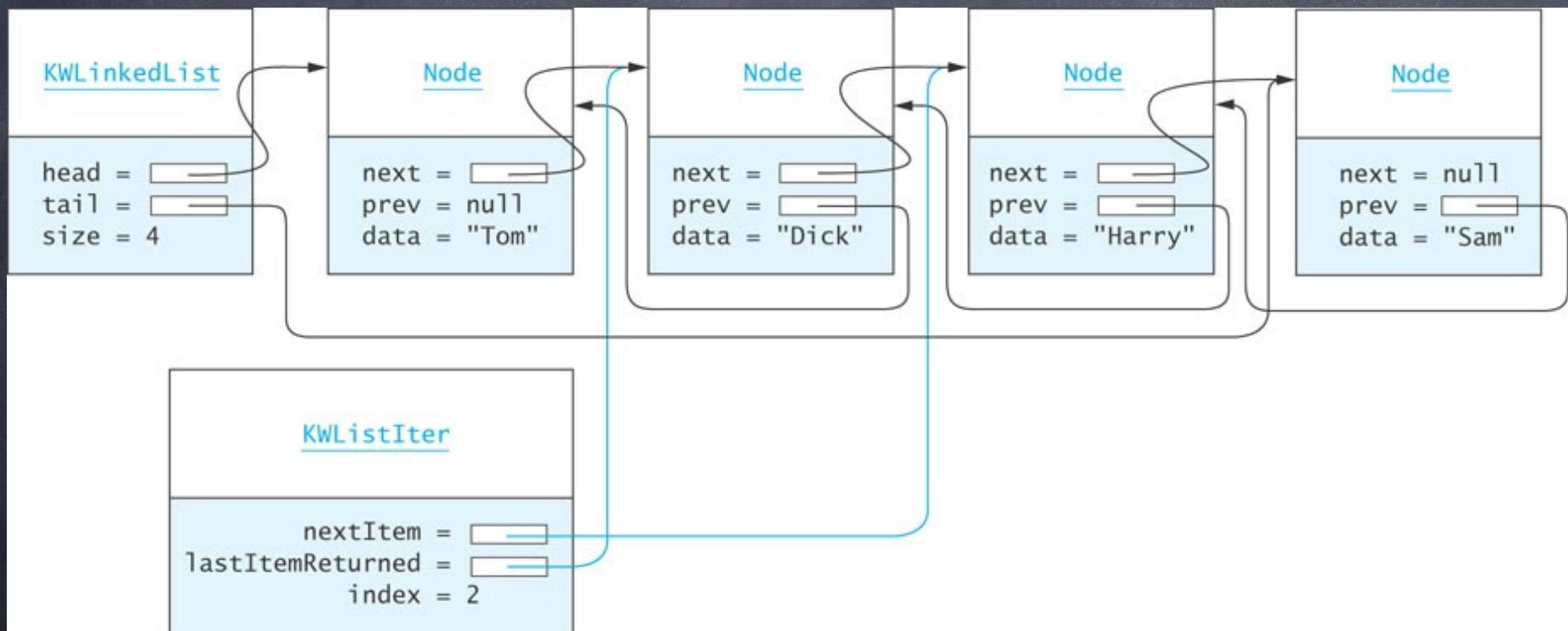
```
public int indexOf(E item) {  
    // MySingleLinkedList에서와 동일함  
}
```

```
public E remove(int index) {  
    if (index < 0 || index >= size)  
        throw new IndexOutOfBoundsException();  
    ListIterator<E> iter = listIterator(index);  
    E result = iter.next();  
    iter.remove();  
    return result;  
}
```

# KWListIterator 클래스: 데이터 멤버

private Node<E> nextItem	A reference to the next item.
private Node<E> lastItemReturned	A reference to the node that was last returned by <code>next</code> or <code>previous</code> .
private int index	The iterator is positioned just before the item at <code>index</code> .

# KWListIterator 클래스



# class KWListIterator

```
private class KWListIterator implements ListIterator<E> {  
    private Node <E> nextItem;  
    private Node <E> lastItemReturned;  
    private int index = 0;
```

# class KWListIterator: 생성자

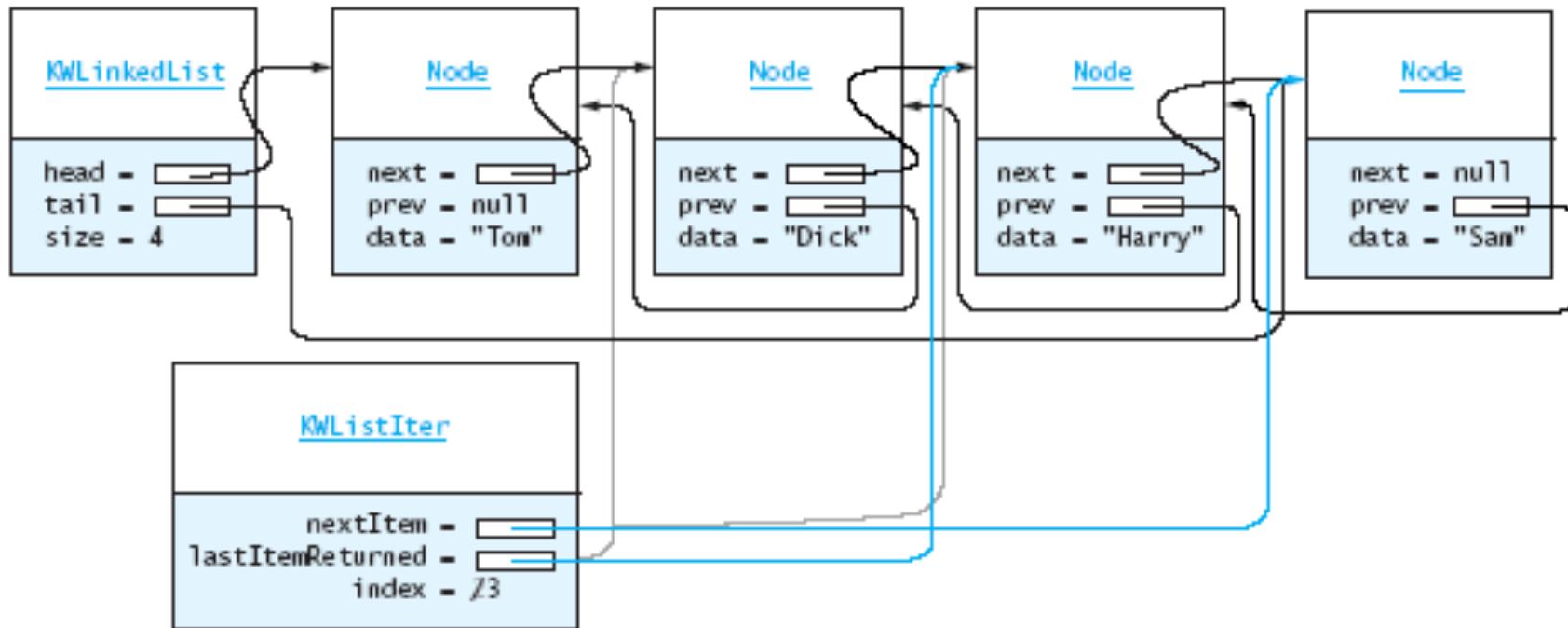
```
public KWListIterator(int i) {  
    if (i < 0 || i > size)  
        throw new IndexOutOfBoundsException("Invalid index " + i);  
  
    lastItemReturned = null;  
    if (i == size) {  
        index = size;  
        nextItem = null;  
    }  
    else {  
        nextItem = head;  
        for (index = 0; index < i; index++)  
            nextItem = nextItem.next;  
    }  
}
```

# class KWListIterator: hasNext()

```
public boolean hasNext() {  
    return nextItem != null;  
}
```

# Iterator 전진하기

**FIGURE 4.26**  
Advancing a KWListIter



# class KWListIterator: next()

```
public E next() {  
    if (!hasNext())  
        throw new NoSuchElementException();  
  
    lastItemReturned = nextItem;  
    nextItem = nextItem.next;  
    index++;  
    return lastItemReturned.data;  
}
```

# class KWListIterator: hasPrevious()

```
public boolean hasPrevious() {  
    return (nextItem == null && size != 0) || nextItem.prev != null;  
}
```

# class KWListIterator: previous()

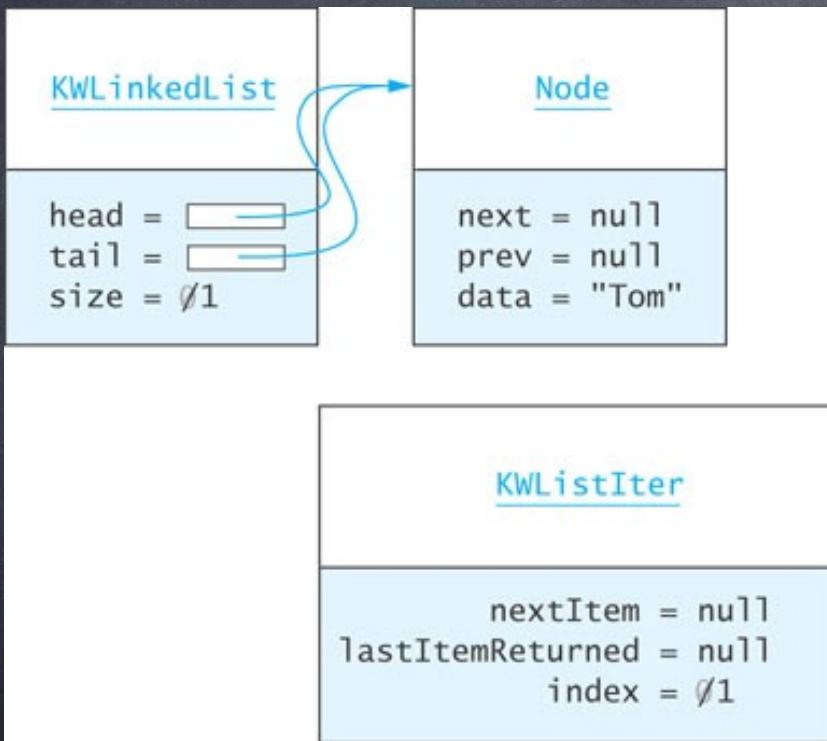
```
public E previous() {  
    if (!hasPrevious())  
        throw new NoSuchElementException();  
  
    if (nextItem == null)    // Iterator past the last element  
        nextItem = tail;  
    else  
        nextItem = nextItem.prev;  
    lastItemReturned = nextItem;  
    index--;  
    return lastItemReturned.data;  
}
```

# nextIndex() and previousIndex()

```
public int nextIndex() {  
    return index;  
}
```

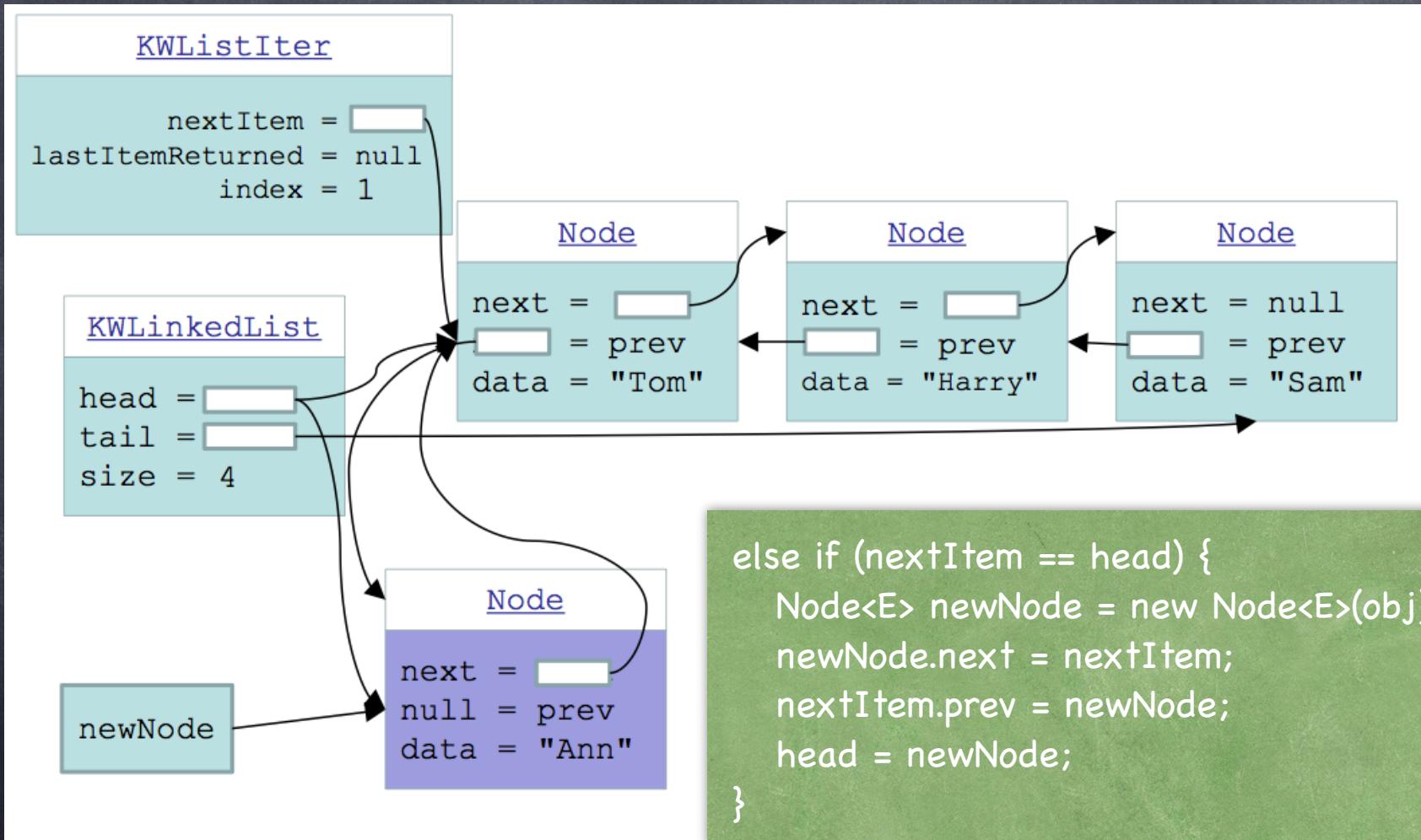
```
public int previousIndex() {  
    return index - 1;  
}
```

# add(E obj) - to empty list



```
public void add(E obj)
{
    if (head == null) {
        head = new Node<E>(obj);
        tail = head;
    }
}
```

# add() - at the head



# add() - at the tail

KWListIter

```
nextItem = null  
lastItemReturned = null  
index = 3
```

KWLinkedList

```
head = [ ]  
tail = [ ]  
size = 4
```

```
else if (nextItem == null) {  
    Node<E> newNode = new Node<E>(obj);  
    tail.next = newNode;  
    newNode.prev = tail;  
    tail = newNode;  
}
```

Node

```
next = [ ]  
prev = null  
data = "Tom"
```

Node

```
next = [ ]  
prev = [ ]  
data = "Ann"
```

Node

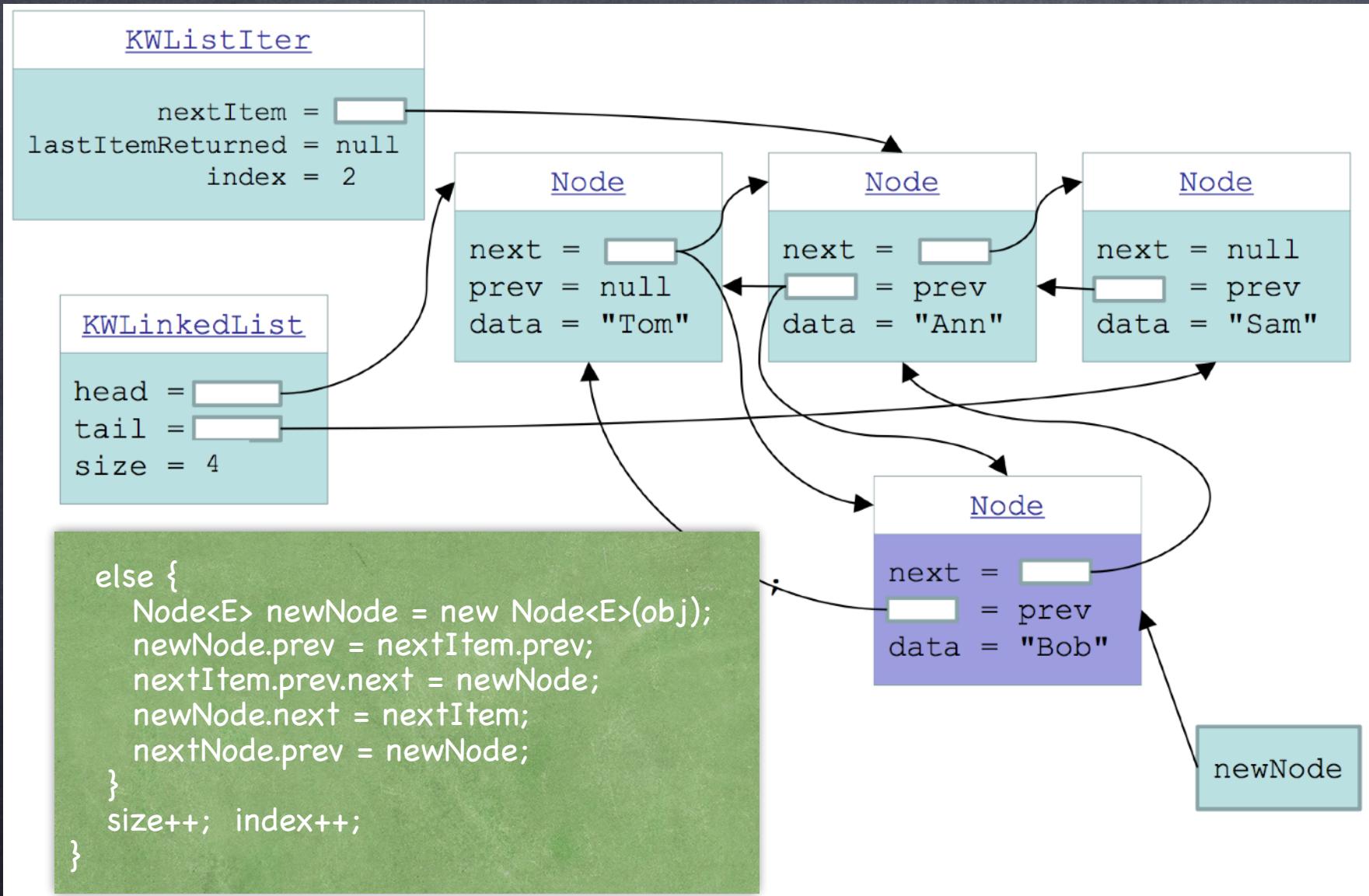
```
next = [ ]  
prev = [ ]  
data = "Sam"
```

Node

```
next = null  
prev = [ ]  
data = "Bob"
```

newNode

# add() - in the middle



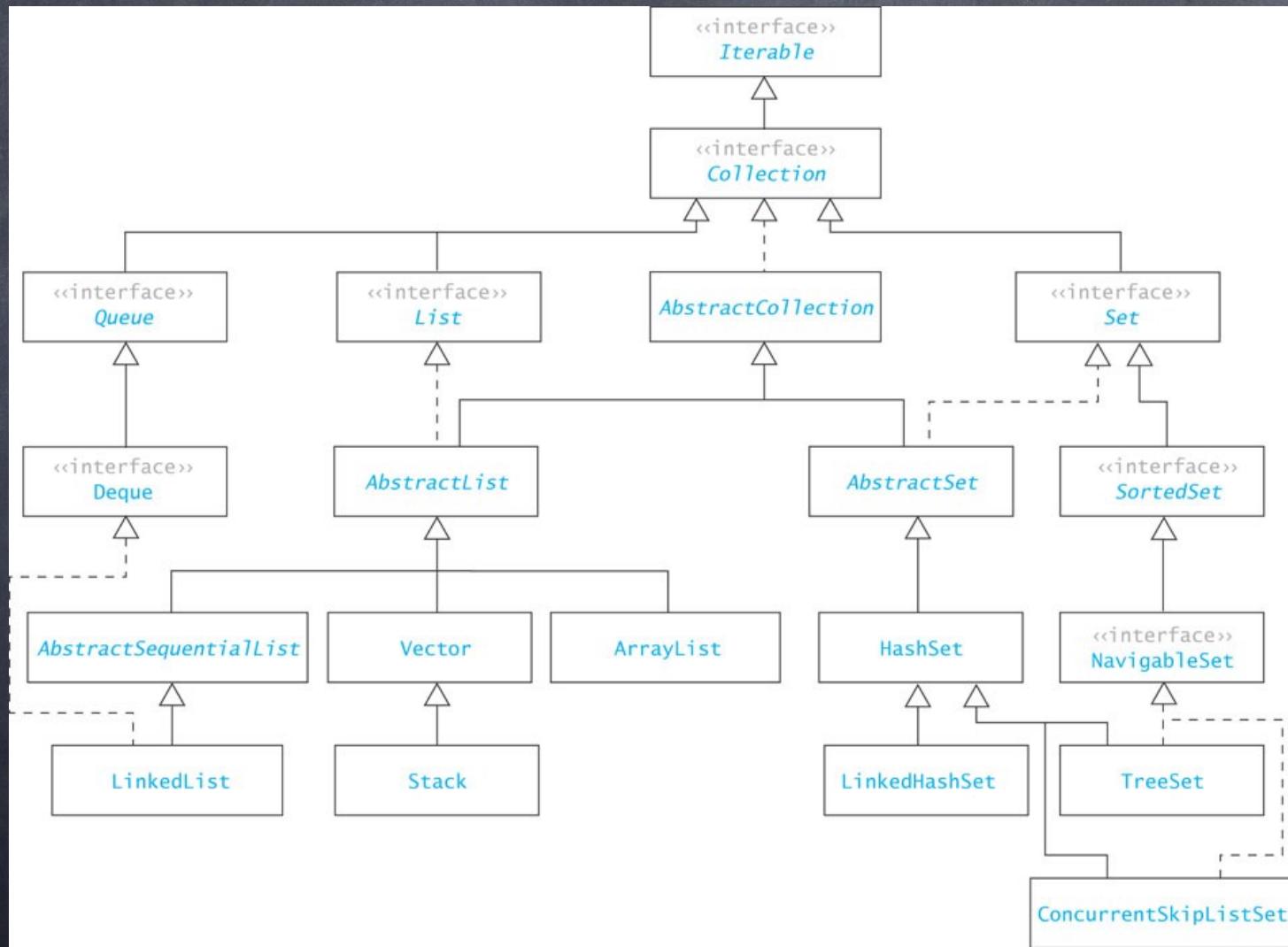
# *remove() and set()*

- ☞ 연습문제로 남겨둠

# Polynomial 프로그램

- ☞ KWLinkedList 클래스를 이용하도록 수정하라.

# Java Collection Framework



# Enhanced for 문

```
LinkedList<String> aList = new LinkedList<String>();  
aList.add("Some string");
```

....

```
// we want to traverse (or iterate) the list
```

```
for (String str : aList)  
    // do something to str  
    ...  
}
```

enhanced for문은 내부적으로 iterator를  
이용하여 구현됨

# 원형(circular) 리스트

## 원형 이중연결 리스트:

마지막 노드의 다음 노드가 첫번째 노드가 되고

첫 노드의 이전 노드가 마지막 노드가 됨

