

# 리스트 List

1. ADT List
2. Array List (List on Array)
3. Linked List (List by Link)
4. Extended Linked List

# 1. ADT List

## ADT(Abstract Data Type)

- 행하는 작업의 목록으로 데이터의 타입을 나타낸 것
- Implementation detail에 신경 쓰지 않고 추상적 레벨에서 데이터 타입을 정의함
- “어떻게 구현할까”가 아니라 “어떻게 사용할까”에 focusing

### ADT List

$k$ 번째 자리에 원소  $x$ 를 삽입한다

$k$ 번째 원소를 삭제한다

원소  $x$ 를 삭제한다

$k$ 번째 원소를 알려준다

원소  $x$ 가 몇 번째 원소인지 알려준다

리스트의 사이즈(원소의 총 수)를 알려준다

✓ A data structure  
is also a data type

# Java의 **Interface** 기능은 **ADT** 디자인과 잘 어울린다

---

```
public interface IntegerListInterface {  
    public void add(int i, Integer x);  
    public void append(Integer x);  
    public Integer remove(int i);  
    public boolean removeItem(Integer x);  
    public Integer get(int i);  
    public void set(int i, Integer x);  
    public int indexOf(Integer x);  
    public int len();  
    public boolean isEmpty();  
    public void clear();  
}
```

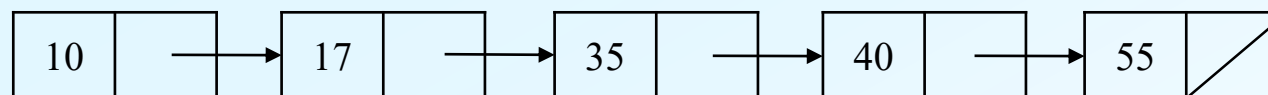
## Array로 구현할 수도 있고

- Consecutive space
- Intuitively simple
- Weak points
  - Overflow
  - Needs *shift* operation for insertion/deletion

item[]	10	35	40	17	95	50	48	33	9			
--------	----	----	----	----	----	----	----	----	---	--	--	--

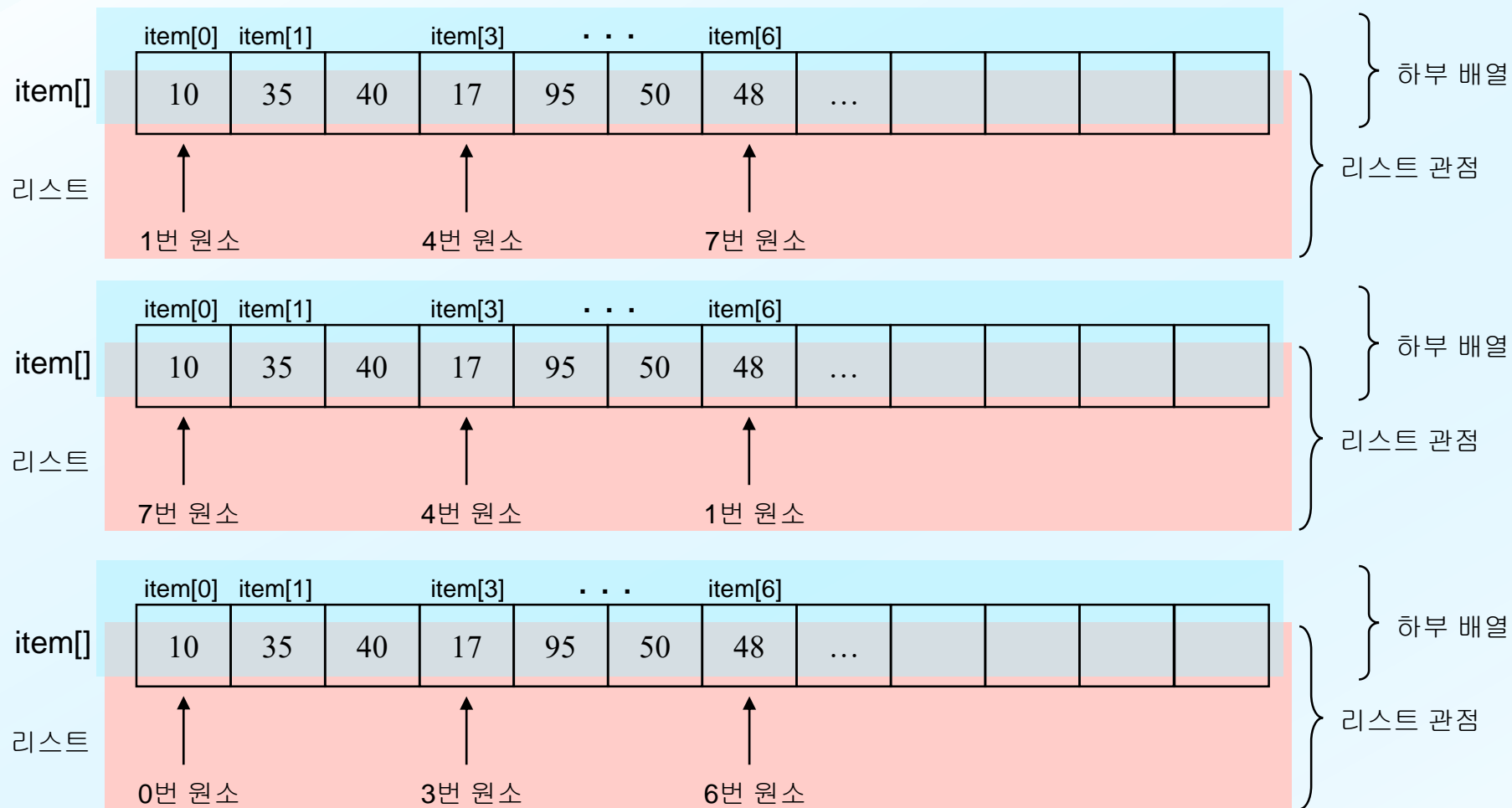
## Linked list로 구현할 수도 있다

- No consecutive space
- Free from shift overhead
- No overflow
- Overhead for linking



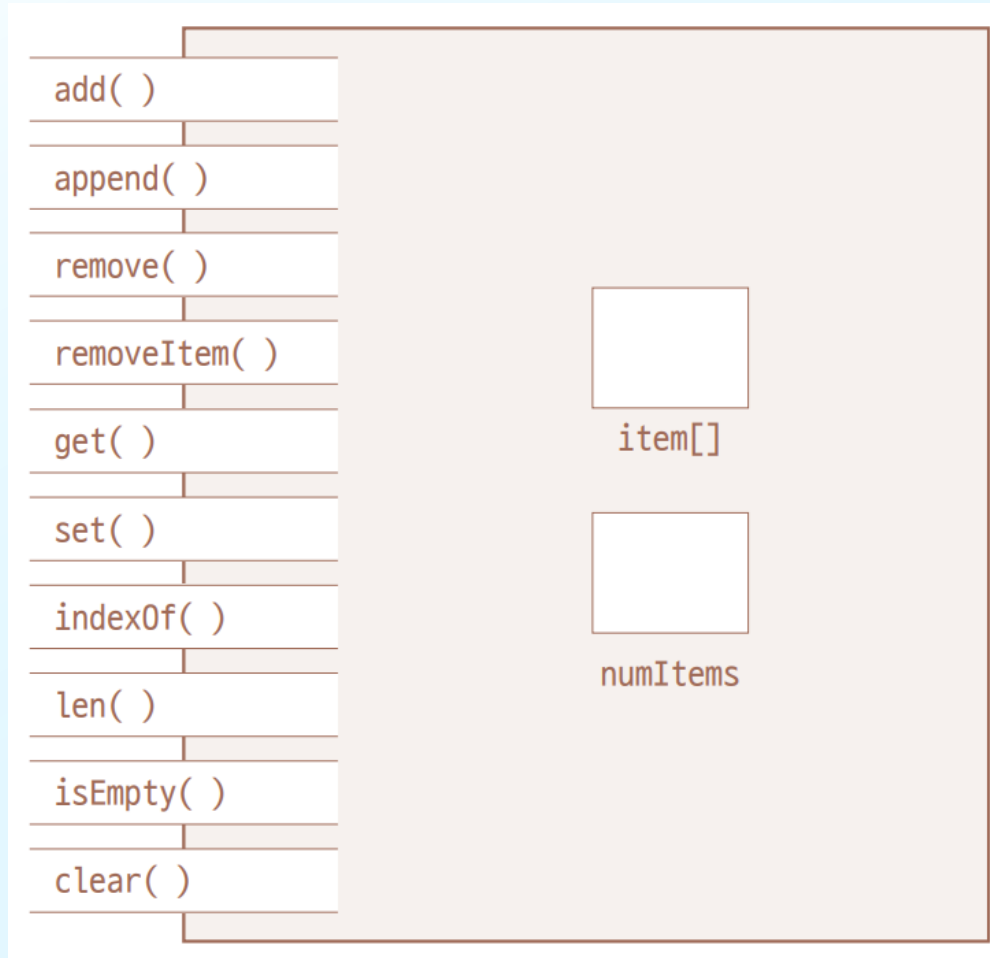
## 2. Array List

# 하부 배열과 리스트를 보는 여러 관점



- ✓ 배열을 리스트로 해석하는 관점은 하나가 아니다
- ✓ 여기서는 세번째 관점을 택한다

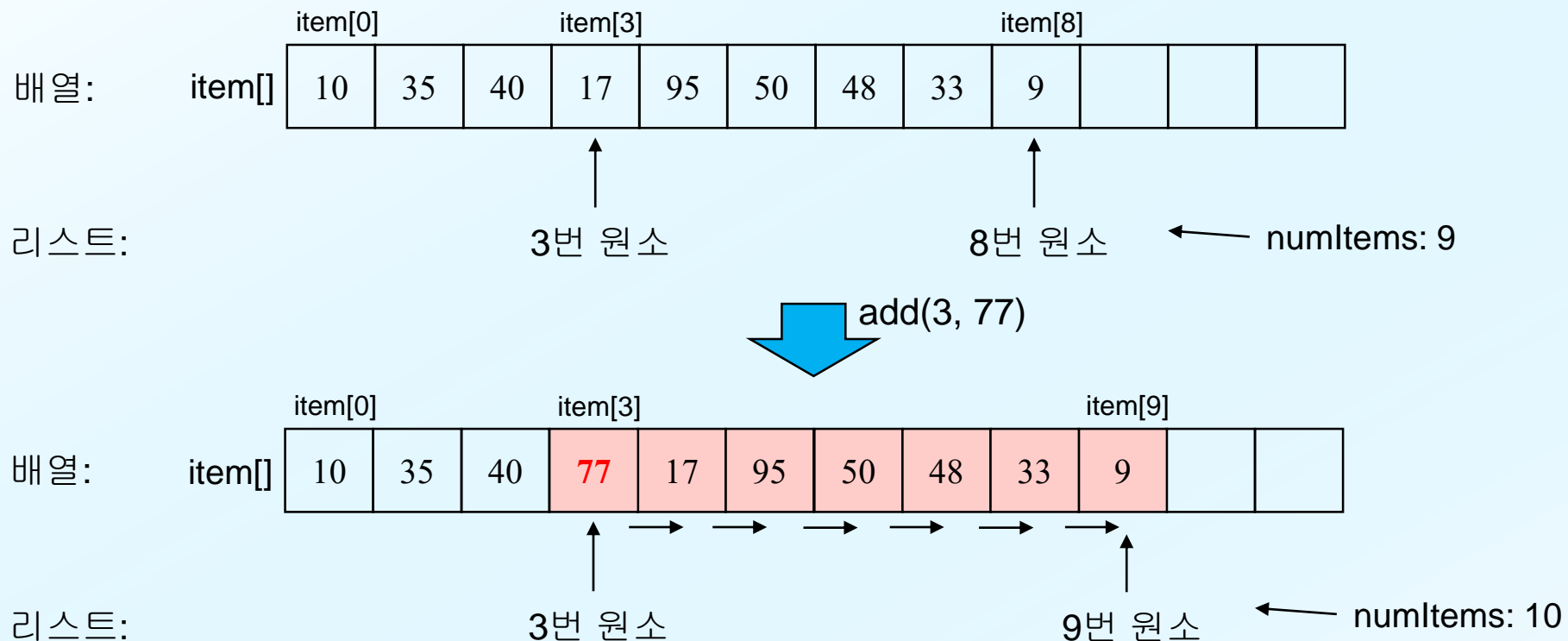
# Array List 객체 구조



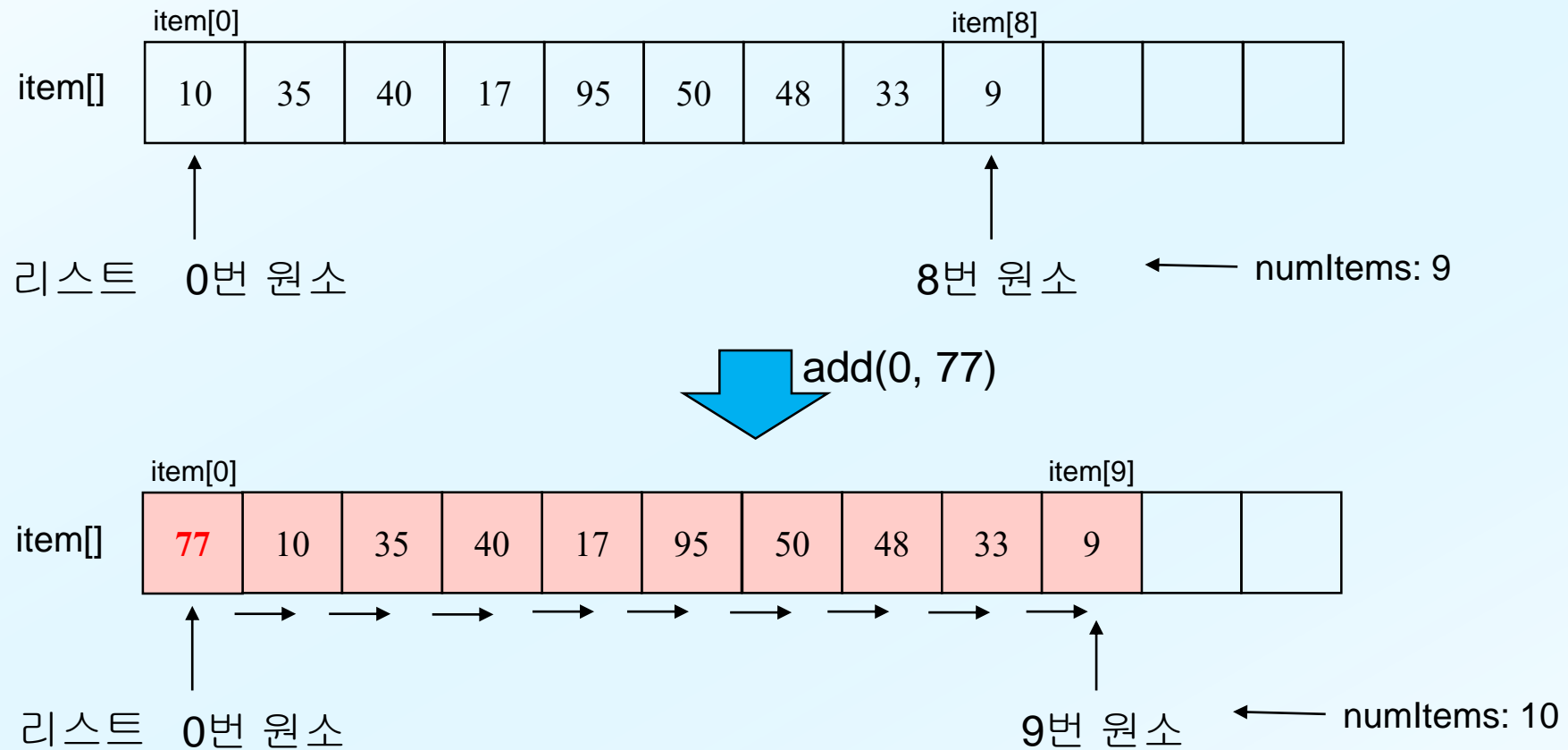
많은 가능한  
구조 디자인 중의 하나



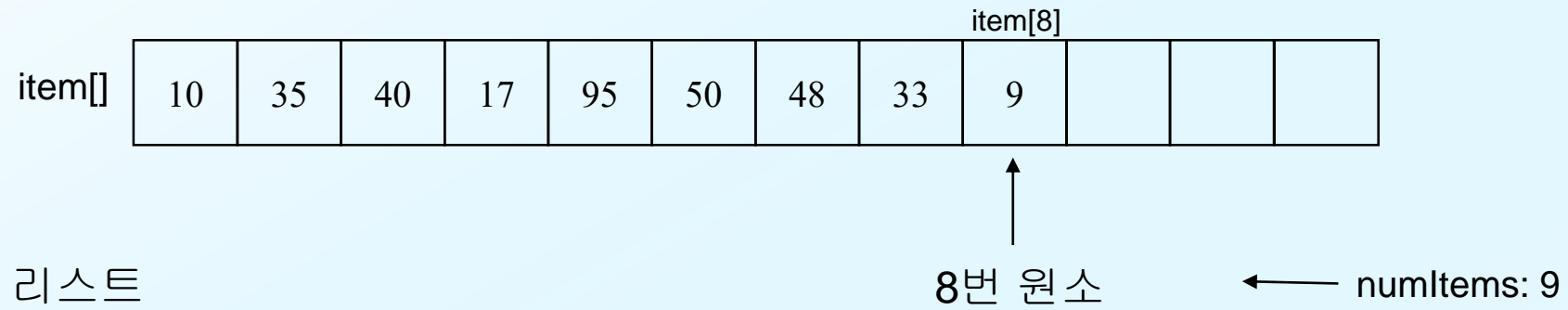
# Insertion



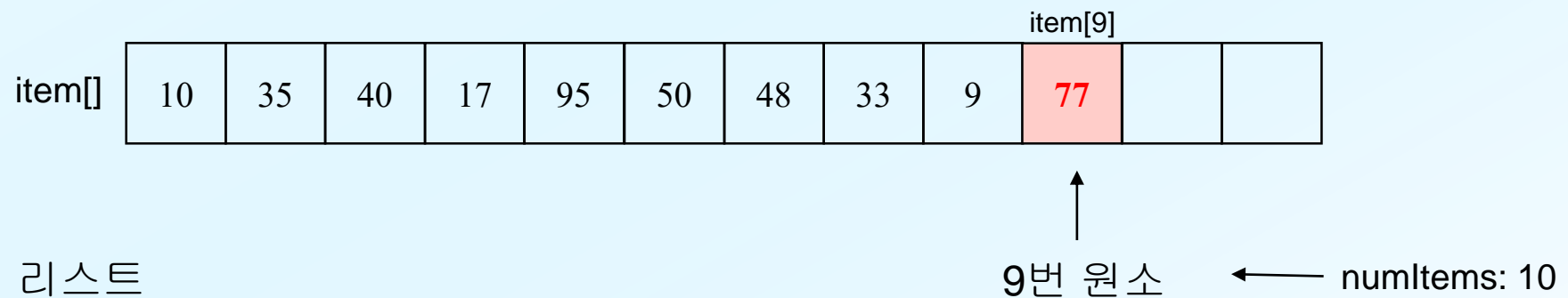
# Worst-Case Insertion



# Best-Case Insertion



append(77) 또는 add(9, 77)

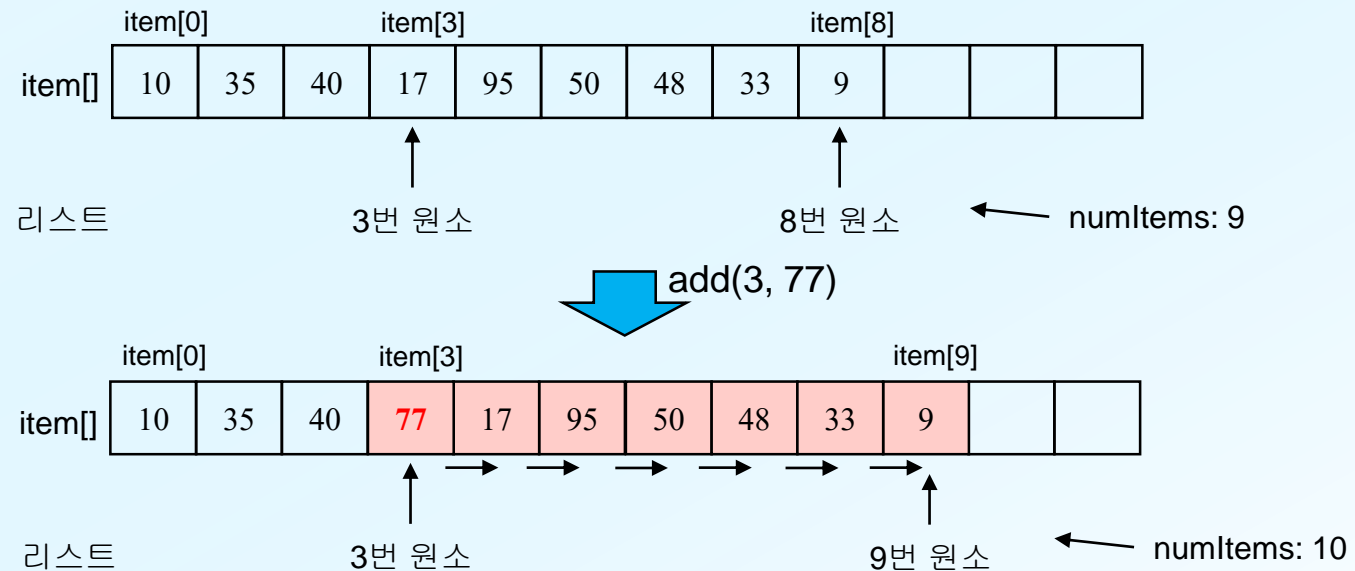


# Algorithm add()

```

add( $k, x$ ): ◀ Insert  $x$  as  $k^{\text{th}}$  element
  if ( $\text{numItems} \geq \text{item.length}$ )
    /*에러 처리*/ ◀ 배열 용량 초과
  else
    for  $i \leftarrow \text{numItems}-1$  downto  $k$ 
       $\text{item}[i+1] \leftarrow \text{item}[i]$  ◀ right shift
     $\text{item}[k] \leftarrow x$ 
     $\text{numItems}++$ 

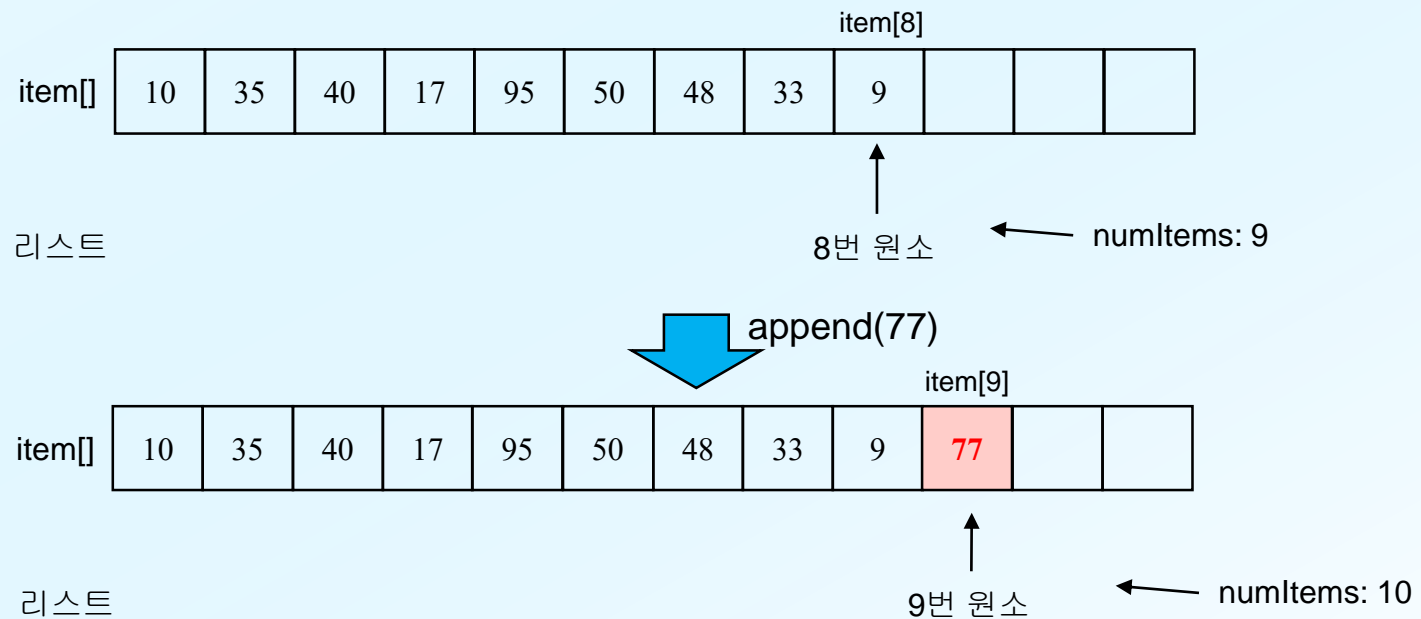
```



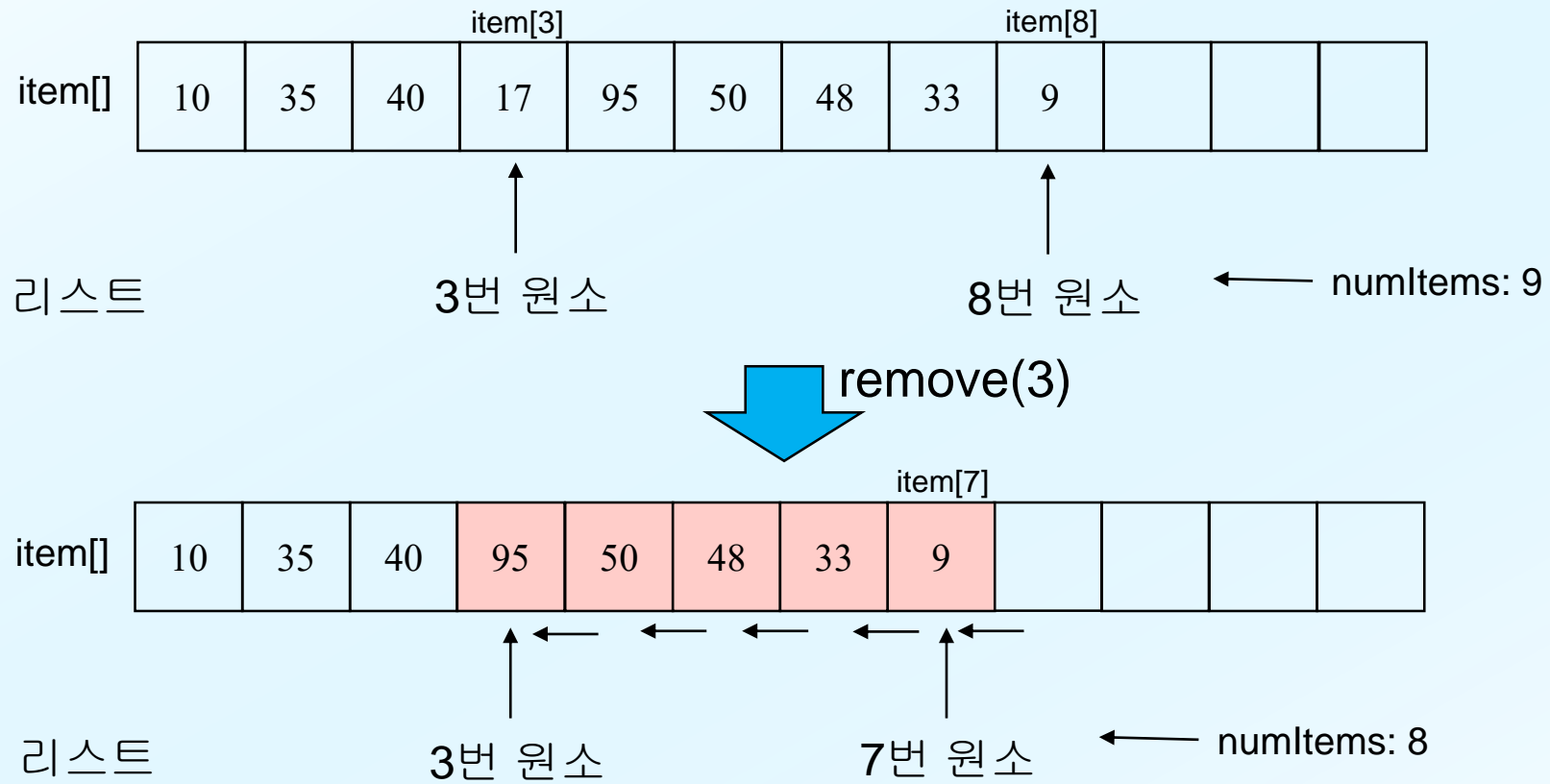
# Algorithm append()

```

append( $x$ ):  ◀ Append  $x$ 
  if (numItems >= item.length)
    /*에러 처리*/
  else
    item[numItems] ←  $x$ 
    numItems++
  
```



# Deletion

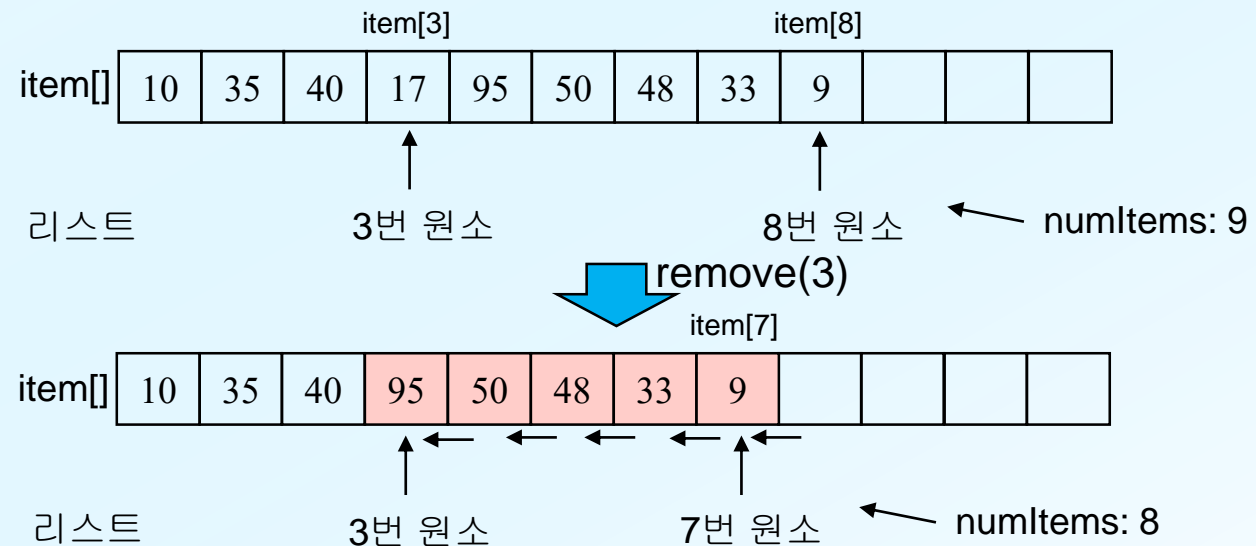


# Algorithm remove()

```

remove( $k$ ): ◀ Remove  $k^{\text{th}}$  element
  if (isEmpty() ||  $k < 0$  ||  $k > \text{numItems}-1$ )
    /*에러 처리*/
  else
    for  $i \leftarrow k$  to numItems-2
      item[ $i$ ]  $\leftarrow$  item[ $i+1$ ] ◀ shift left
    numItems--

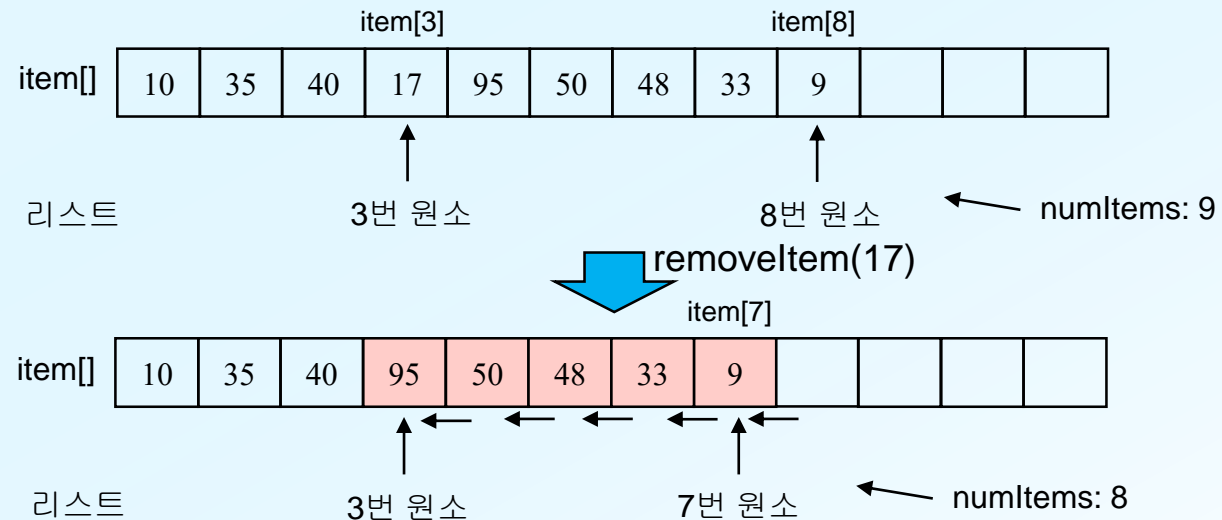
```



# Algorithm removeItem()

```

removeItem( $x$ ): ◀ Remove element  $x$ 
     $k \leftarrow 0$ 
    while (  $k < \text{numItems} \ \&\& \ \text{item}[k] \neq x$  ) ◀ 원소 찾기
         $k++$ 
    if (  $k = \text{numItems}$  ) return false ◀ 원소 없음
    else
        for  $i \leftarrow k$  to  $\text{numItems}-2$  ◀ 원소  $x = \text{item}[k]$  삭제
             $\text{item}[i] \leftarrow \text{item}[i+1]$  ◀ shift left
         $\text{numItems}--$ 
        return true
  
```





# 기타 작업

```
get(i):  
    if (i >= 0 && i <= numItems-1)  
        return item[i]  
    else  
        return OUT_OF_BOUND
```

```
set(i, x):  
    if (i >= 0 && i <= numItems-1)  
        item[i] ← x  
    else  
        /* 에러 처리 */
```

```
indexOf(x):  
    i ← 0  
    while ( i < numItems && item[i] != x)  
        i++  
    if (i = numItems)  
        return NOT_FOUND  
    else  
        return i    ◀ x = item[i]는 i번 원소
```

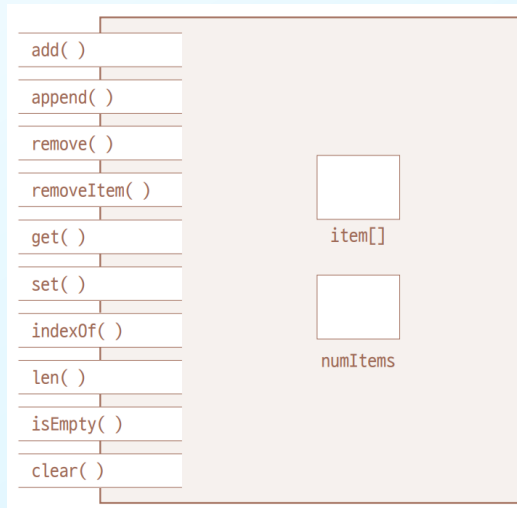
```
len():  
    return numItems
```

```
isEmpty():  
    if (numItems = 0)  
        return true  
    else  
        return false
```

```
clear():  
    numItems ← 0
```

# 자바 구현 1

시작을 위해 일단 원소 타입을 정수로 제한한다



```

public interface IntegerListInterface {
    public void add(int i, Integer x);
    public void append(Integer x);
    public Integer remove(int i);
    public boolean removeItem(Integer x);
    public Integer get(int i);
    public void set(int i, Integer x);
    public int indexOf(Integer x);
    public int size();
    public boolean isEmpty();
    public void clear();
}
  
```

```

public class IntegerArrayList implements IntegerListInterface {
    private Integer[] item;
    private int numItems;
    private static final int DEFAULT_CAPACITY = 64;
    public IntegerArrayList() { // 생성자 1
        item = new Integer[DEFAULT_CAPACITY];
        numItems = 0;
    }
    public IntegerArrayList(int n) { // 생성자 2
        item = new Integer[n];
        numItems = 0;
    }
    ...
}
  
```

이렇게 생성될 때

IntegerArrayList list = new IntegerArrayList();

이렇게 생성될 때

IntegerArrayList list = new IntegerArrayList(256);

# Wrapper Class

Java의 primitive type은 총 8개 { **byte, short, int, long**  
**float, double**  
**char**  
**boolean**

이들은 Object 클래스를 상속받지 못한다

그래서 패키지 java.lang에서는 이들 각각을 위해  
wrapper class(포장 클래스)를 준비해두고 있다

- 이름을 대문자로 시작
- 클래스 Object를 상속받는다
- 초기 Java에서는 사용이 번거로웠으나, 요즘은 그냥 primitive type처럼 쓰면 Java에서 알아서 포장해준다

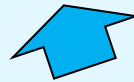
{ **Byte, Short, Integer, Long**  
**Float, Double**  
**Char**  
**Boolean**

```

void add(int k , Integer x) {
    if (numItems >= item.length)
        { /*에러 처리*/ } // 배열 용량 초과
    else {
        for (int i = numItems-1; i >= k; i--)
            item[i+1] = item[i]; // shift right
        item[k] = x;
        numItems++;
    }
}

```

Java 코드



```

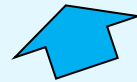
add(k, x):
    if (numItems >= item.length)
        /*에러 처리*/ ◀ 배열 용량 초과
    else
        for i ← numItems-1 downto k
            item[i+1] ← item[i] ◀ right shift
        item[k] ← x
        numItems++

```

알고리즘 in pseudo code(유사 코드)

```
void append(Integer x) {  
    if (numItems >= item.length) { /*에러 처리*/ }  
    else  
        item[numItems++] = x;  
}
```

Java 코드



```
append( $x$ ):  
    if (numItems >= item.length) /*에러 처리*/  
    else  
        item[numItems]  $\leftarrow$   $x$   
        numItems++
```

```

Integer remove(int k) { // 리턴값 추가
    if (isEmpty() || k < 0 || k > numItems-1)
        return null;
    else {
        Integer tmp = item[k];
        for (int i = k; i <= numItems-2; i++)
            item[i] = item[i+1]; // shift left
        numItems--;
        return tmp;
    }
}

```

Java 코드



```

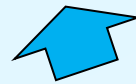
remove(k): ◀ Remove  $k^{\text{th}}$  element
    if (isEmpty() ||  $k < 0$  ||  $k > \text{numItems}-1$ )
        /*에러 처리*/
    else
        for  $i \leftarrow k$  to numItems-2
            item[i] ← item[i+1] ◀ shift left
        numItems--

```

```

boolean removeItem(Integer x) {
    int k = 0;
    while (k < numItems && item[k].compareTo(x) != 0)
        k++;
    if (k == numItems) return false;
    else {
        for (int i = k; i <= numItems-2; i++)
            item[i] = item[i+1]; // shift left
        numItems--;
        return true;
    }
}

```



Java 코드

```

removeItem(x): ◀ Remove element x
    k ← 0
    while ( k < numItems && item[k] != x) ◀ 원소 찾기
        k++
    if ( k = numItems) return false ◀ 원소 없음
    else
        for i ← k to numItems-2 ◀ 원소 삭제
            item[i] ← item[i+1] ◀ shift left
        numItems--
        return true

```

자바 코드

```
Integer get(int i) {
    if (i >= 0 && i <= numItems-1)
        return item[i];
    else return null;
}
```

```
void set(int i, Integer x) {
    if (i >= 0 && i <= numItems-1)
        item[i] = x;
    else { /* 에러 처리 */ }
}
```

```
private final int NOT_FOUND = -12345678;
public int indexOf(Integer x) {
    int i = 0;
    while (i < numItems &&
           ((Comparable)item[i]).compareTo(x) != 0)
        i++;
    if (i == numItems) return NOT_FOUND;
    else return i; // x = item[i]는 i번 원소
}
```

```
int len() {
    return numItems;
}
```

```
boolean isEmpty() {
    return numItems == 0;
}
```

```
void clear() {
    numItems = 0;
}
```



```
get(i):
    if (i >= 0 && i <= numItems-1)
        return item[i]
    else
        return OUT_OF_BOUND
```

```
set(i, x):
    if (i >= 0 && i <= numItems-1)
        item[i] ← x
    else
        /* 에러 처리 */
```

```
indexOf(x):
    i ← 0
    while ( i < numItems && item[i] != x)
        i++
    if (i = numItems)
        return NOT_FOUND
    else
        return i ◀ x = item[i]는 i번 원소
```

```
len():
    return numItems
```

```
isEmpty():
    if (numItems = 0)
        return true
    else
        return false
```

```
clear():
    numItems ← 0
```



# 자바 구현 2: 제네릭 Generic 버전

앞의 클래스 IntegerArrayList..

```
Public class IntegerArrayList {
    private Integer[] item;
    private int numItems;
    ...
    public void add(int index, Integer x) {
        if (numItems >= item.length || index < 0 || index > numItems)
            ...
        else {
            ...
        }
    }
    public void append(Integer x) {
        if (numItems >= item.length) { /*에러 처리*/ }
        else {
            item[numItems++] = x;
        }
    }
    public Integer remove(int index) {
        if (isEmpty() || index < 0 || index > numItems-1)
            return null;
        else {
            Integer tmp = item[index];
            for (int i = index; i <= numItems-2; i++)
                ...
        }
    }
    ...
}
```

**Integer**와 다른 타입이 요구되면,  
그 때마다 클래스를 하나씩 만들어야 한다

유용한 제네릭(generic)

제네릭 설명 참조:

<쉽게 배우는 자료구조>, p.84~86

이 클래스의 객체를 생성해서 사용할 때는

```
IntegerArrayList list = new IntegerArrayList();  
list.add(0, 300);  
list.add(0, 100);  
list.append(500);  
...
```

타입을 고정한 탓에 클래스를 여러 개 만들어야 할 가능성 크다

이렇게 할 수 있다

```
public class ArrayList<E> {
    private E[] item;
    private int numItems;
    ...
    public void add(int index, E x) {
        if (numItems >= item.length || index < 0 || index > numItems)
            ...
        else {
            ...
        }
    }
    public void append(E x) {
        if (numItems >= item.length) { /*에러 처리*/ }
        else {
            item[numItems++] = x;
        }
    }
    public E remove(int index) {
        if (isEmpty() || index < 0 || index > numItems-1)
            return null;
        else {
            E tmp = item[index];
            for (int i = index-1; i <= numItems-2; i++)
                ...
        }
    }
    ...
}
```

타입 **E**를 함수의 parameter처럼 매개변수화함

이 클래스의 객체를 생성해서 사용할 때는

```
ArrayList<Integer> list = new ArrayList<>();  
list.add(0, 300);  
list.add(0, 100);  
list.append(500);  
...
```

프로그램을 통틀어  
**Integer**는 이 곳에 한 번만 명시해주면 됨

이런 방식의 클래스를 **generic class**라 한다

인터페이스도 이런 건 불편하다

```
public interface InterfaceA {  
    public void add(int i, Integer x);  
    public void append(Integer x);  
    public Integer remove(int i);  
    public boolean removeItem(Integer x);  
    public Integer get(int i);  
    public void set(int i, Integer x);  
    public int indexOf(Integer x);  
    public int size();  
    public boolean isEmpty();  
    public void clear();  
}
```

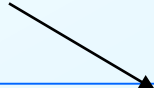
**Integer**와 다른 타입이 요구되면,  
그 때마다 별개의 인터페이스를 만들어야 한다

## 인터페이스를 따르는 클래스에서도 불편하다

```
public class ArrayList implements InterfaceA {
    private Integer[] item;
    private int numItems;
    ...
    public void add(int index, Integer x) {
        if (numItems >= item.length || index < 0 || index > numItems)
            ...
        else {
            ...
        }
    }
    public void append(Integer x) {
        if (numItems >= item.length) { /*에러 처리*/ }
        else {
            item[numItems++] = x;
        }
    }
    public Integer remove(int index) {
        if (isEmpty() || index < 0 || index > numItems-1)
            return null;
        else {
            Integer tmp = item[index];
            for (int i = index; i <= numItems-2; i++)
                ...
        }
    }
    ...
}
```

인터페이스 interfaceA를 만족시키기 위해  
Integer여야 하는 모든 곳에 **Integer** 명시

## 인터페이스도 Generic이 가능하다



```
public interface InterfaceA<E> {  
    public void add(int i, E x);  
    public void append(E x);  
    public E remove(int i);  
    public boolean removeItem(E x);  
    public E get(int i);  
    public void set(int i, E x);  
    public int indexOf(E x);  
    public int size();  
    public boolean isEmpty();  
    public void clear();  
}
```

클래스와 인터페이스를 모두 **generic**으로 하면..

```
public class ArrayList<E> implements InterfaceA<E> {
    private E[] item;
    private int numItems;
    ...
    public void add(int index, E x) {
        if (numItems >= item.length || index < 0 || index > numItems)
            ...
        else {
            ...
        }
    }
    public void append(E x) {
        if (numItems >= item.length) { /*에러 처리*/ }
        else {
            item[numItems++] = x;
        }
    }
    public E remove(int index) {
        if (isEmpty() || index < 0 || index > numItems-1)
            return null;
        else {
            E tmp = item[index];
            for (int i = index; i <= numItems-2; i++)
                ...
        }
    }
    ...
}
```

클래스와 인터페이스 둘 다 **E**를 parameter화 했다  
- Generic class, generic interface



이 클래스의 객체를 생성해서 사용할 때는

```
ArrayList<Integer> list = new ArrayList<>();  
list.add(0, 300);  
list.add(0, 100);  
list.append(500);  
...
```

프로그램을 통틀어  
**Integer**는 이 곳에 한 번만 명시해주면 됨

이런 구조가 될 수도 있다

```
public interface A<E> {
    public void ft1(int i, E x);
    public void ft2(E x);
    ...
}
```

```
public class Class1<E, T> implements A<E> {
    private E[] item;
    private int numItems;
    ...
    public void ft1(int index, E x) {
        ...
    }
    public void ft2(E x) {
        ...
    }
    public T ft3(int index) {
        if (isEmpty() || index < 0 || index > numItems-1)
            return null;
        else {
            T tmp = ...
            ...
            return tmp;
        }
    }
    ...
}
```

이 클래스의 객체를 생성해서 사용할 때는

```
Class1<Integer, Node> a1 = new Class1<>();
a1.ft1(0, 300);
a1.ft2(100);
Node n1 = a1.ft3(1);
...
```

프로그램을 통틀어  
E, T는 이 곳에 한 번만 명시해주면 됨.  
Node는 클래스 이름.

# 제네릭을 이용해서 범용성을 높인 리스트 구현

## Generic Interface

```
public interface ListInterface<E> {  
    public void add(int i, E x);  
    public void append(E x);  
    public E remove(int i);  
    public boolean removeItem(E x);  
    public E get(int i);  
    public void set(int i, E x);  
    public int indexOf(E x);  
    public int size();  
    public boolean isEmpty();  
    public void clear();  
}
```

## Generic Class

```

public class ArrayList<E> implements ListInterface<E>{
    private E[] item;
    private int numItems;
    private static final int DEFAULT_CAPACITY = 64;
    public ArrayList() { // 생성자 1
        item = (E[]) new Object[DEFAULT_CAPACITY];
        numItems = 0;
    }
    public ArrayList(int n) { // 생성자 2
        item = (E[]) new Object[n];
        numItems = 0;
    }
    public void add(int index, E x) {
        if (numItems >= item.length || index < 0 || index > numItems)
            { /* 에러 처리 */ }
        else {
            for (int i = numItems-1; i >= index; i--)
                item[i+1] = item[i]; // shift right
            item[index] = x;
            numItems++;
        }
    }
    public void append(E x) {
        if (numItems >= item.length) { /* 에러 처리 */ }
        else {
            item[numItems++] = x;
        }
    }
}

```

“item = new E[n];”를  
허용하지 않아 사용한 편법.  
허용하지 않을 타당성이 없는 불편.

<쉽게 배우는 자료구조>,  
p.187에 추가 설명

```
public E remove(int index) {  
    if (isEmpty() || index < 0 || index > numItems-1)  
        return null;  
    else {  
        E tmp = item[index];  
        for (int i = index; i <= numItems-2; i++)  
            item[i] = item[i+1]; // shift left  
        numItems--;  
        return tmp;  
    }  
}  
public boolean removeItem(E x) {  
    int k = 0;  
    while (k < numItems && ((Comparable)item[k]).compareTo(x) != 0)  
        k++;  
    if (k == numItems) return false;  
    else {  
        for (int i = k; i <= numItems-2; i++)  
            item[i] = item[i+1]; // shift left  
        numItems--;  
        return true;  
    }  
}
```

```

public E get(int index) { // 첫 번째 원소를 0번 원소로 표기
    if (index >= 0 && index <= numItems-1)
        return item[index];
    else return null;
}
public void set(int index, E x) {
    if (index >= 0 && index <= numItems-1)
        item[index] = x;
    else { /* 에러 처리 */ }
}
private final int NOT_FOUND = -1;
public int indexOf(E x) {
    int i = 0;
    while (i < numItems && ((Comparable)item[i]).compareTo(x) != 0)
        i++;
    if (i == numItems) return NOT_FOUND;
    else return i; // x = item[i]는 i번 원소
}
public int size() {
    return numItems;
}
public boolean isEmpty() {
    return numItems == 0;
}
public void clear() {
    item = (E[]) new Object[item.length];
    numItems = 0;
}
} // End ArrayList

```

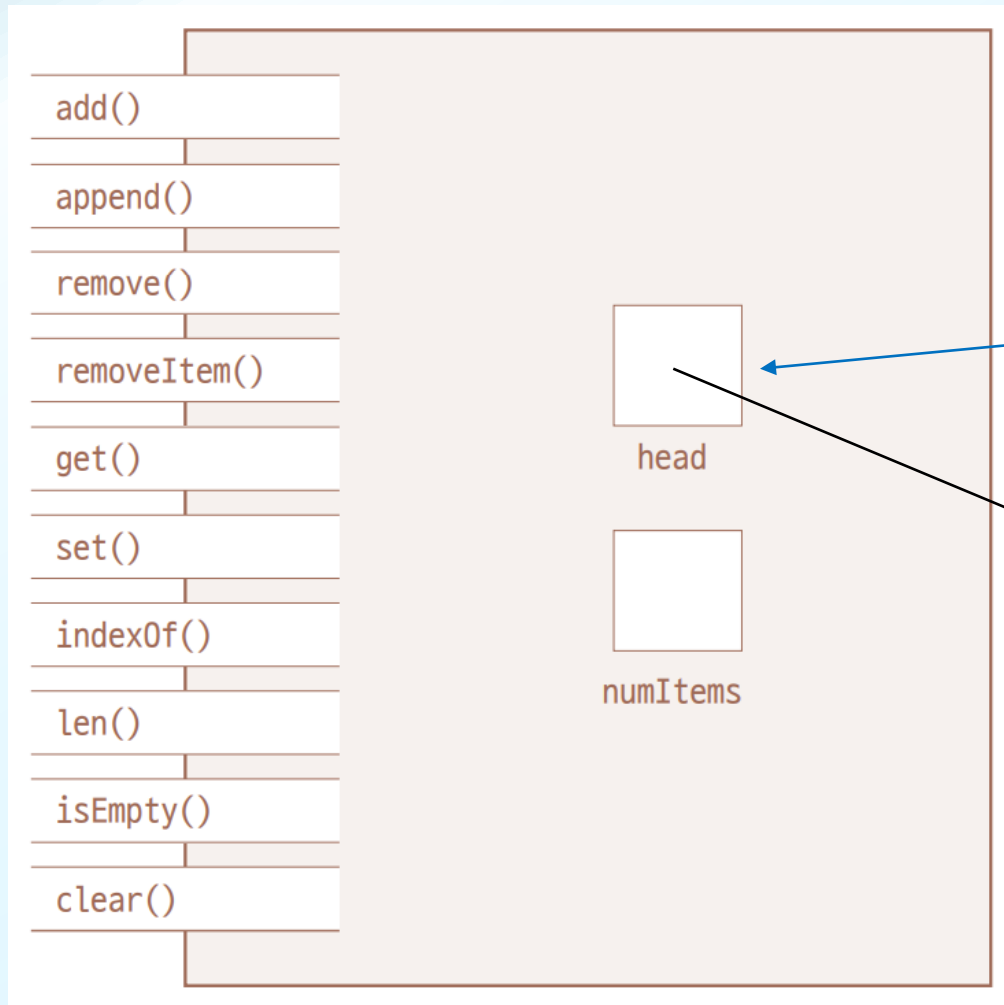
이 클래스의 객체를 생성해서 사용할 때는

```
ArrayList<Integer> list = new ArrayList<>();  
list.add(0, 300); // 오토박싱으로 300은 Integer(300)으로 취급해준다  
list.add(0, 100);  
list.append(500);  
list.remove(2);  
list.append(700);  
list.remove(1);
```

## 3. Linked List

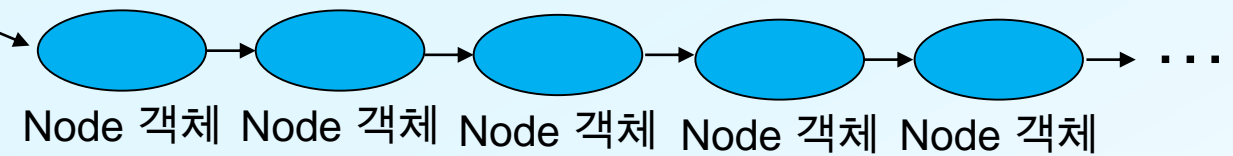


# Linked List 객체 구조



Linked List 객체

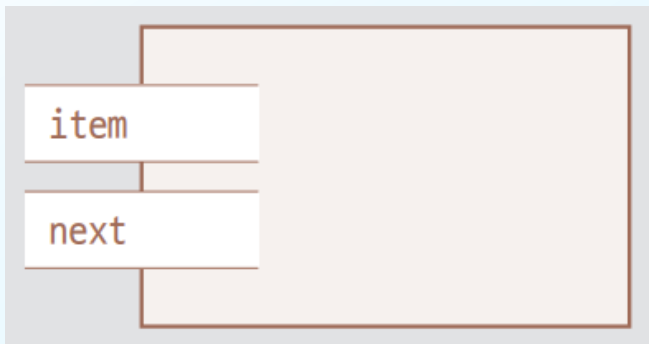
Linked list의  
시작 노드 레퍼런스만  
있으면 된다



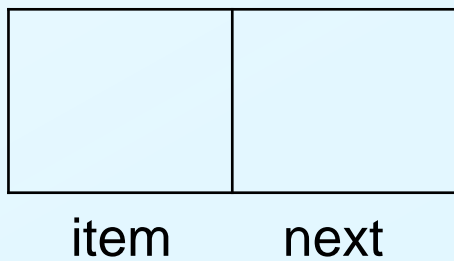
Linked List

사용자 입장에서 볼 때 객체 생성 이후는  
Array List나 Linked List나 동일하다

# Node 객체 구조



보통은 이렇게 그린다

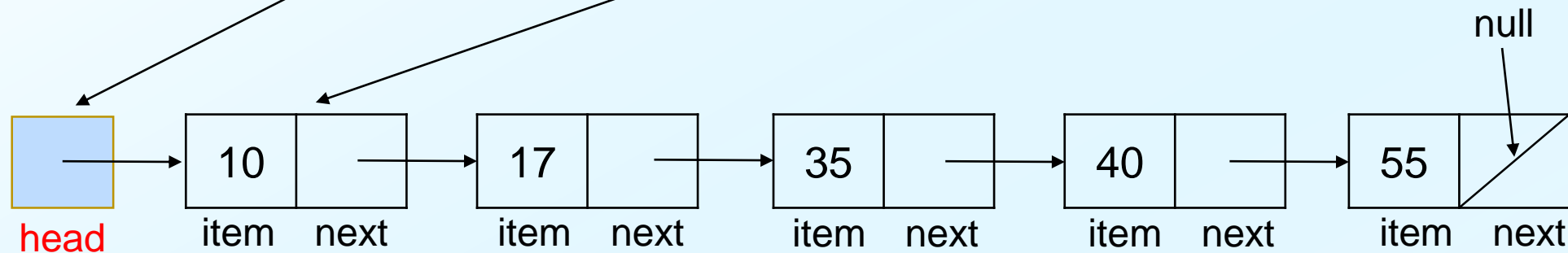


이런 디자인이 객체지향 철학에 가장 충실하기는 하지만..



# Head Node

A linked list has a reference for the head node



```
Node head = null;
```

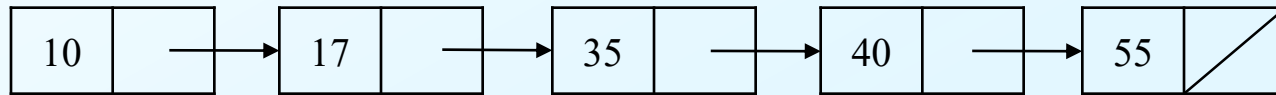
```
Node head = new Node(5);
```

```
...
```

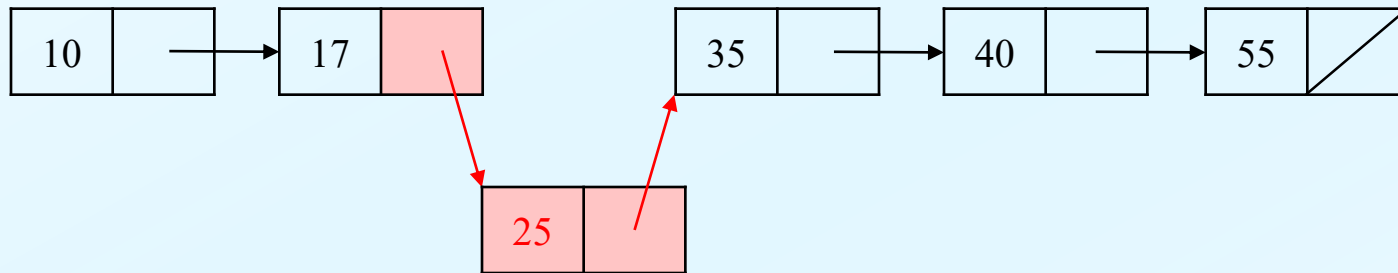
✓ Here, **head** is a simple **reference variable**

# 핵심 작업

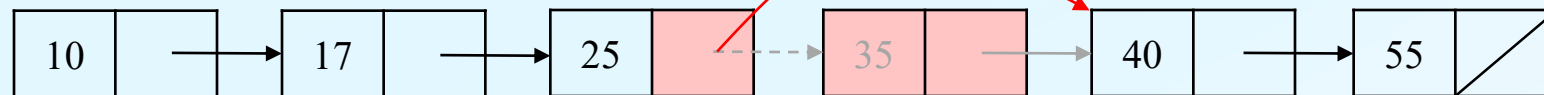
Linked list of integers



Insertion

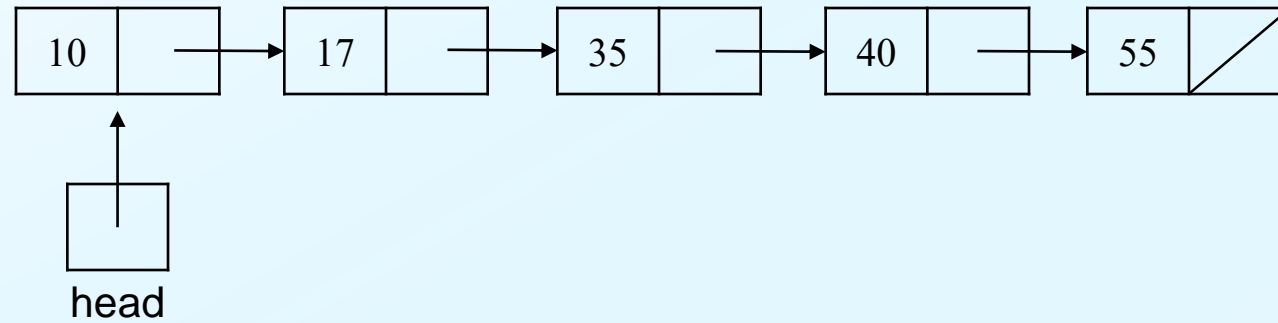


Deletion

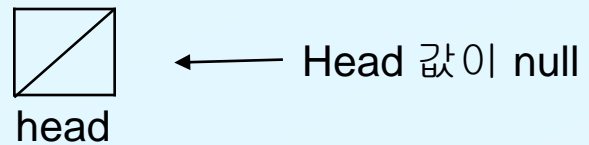


# 전형적 모양

대표적 상태

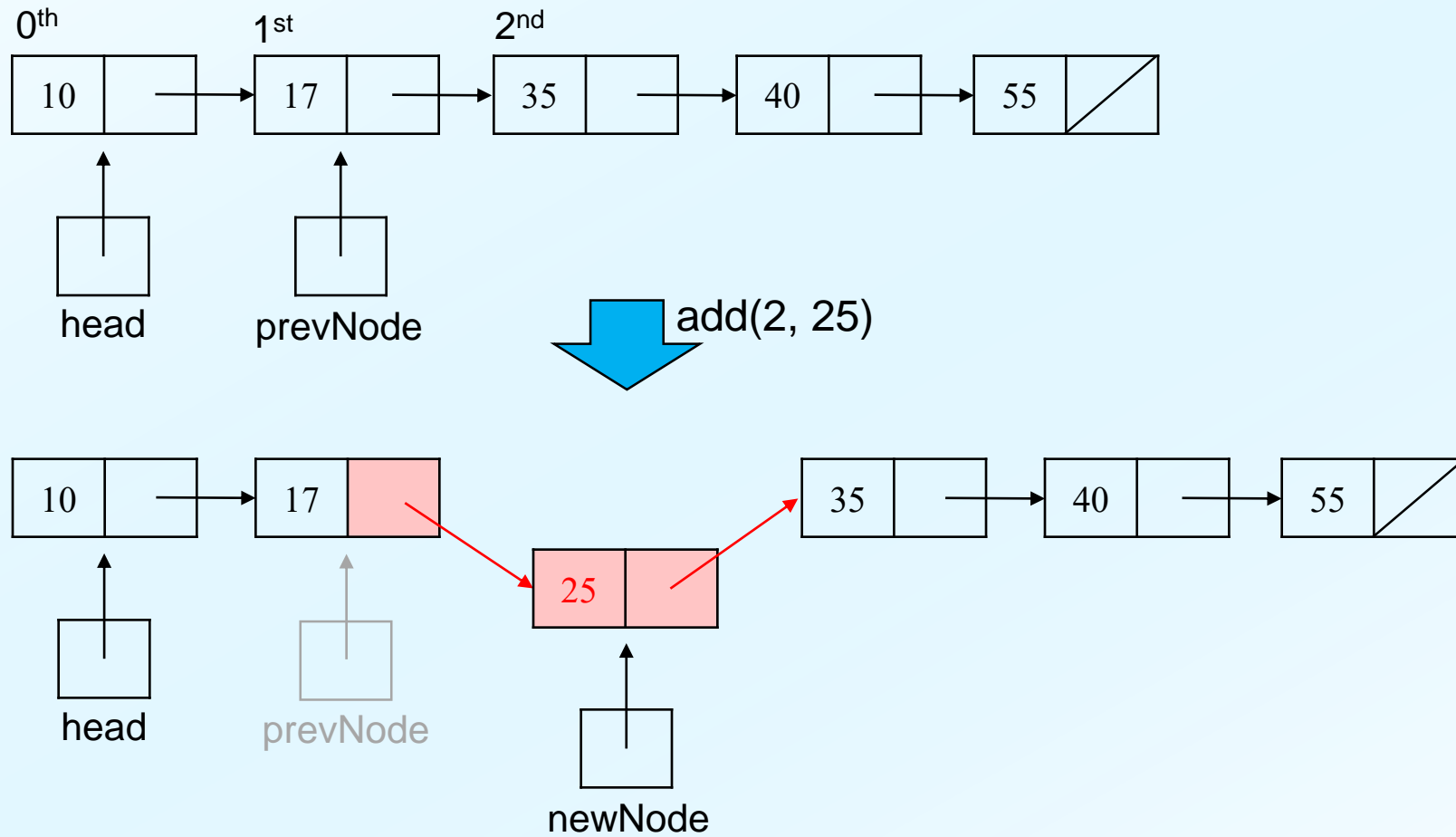


초기 상태



# Insertion

prevNode 다음에 새 노드를 삽입한다



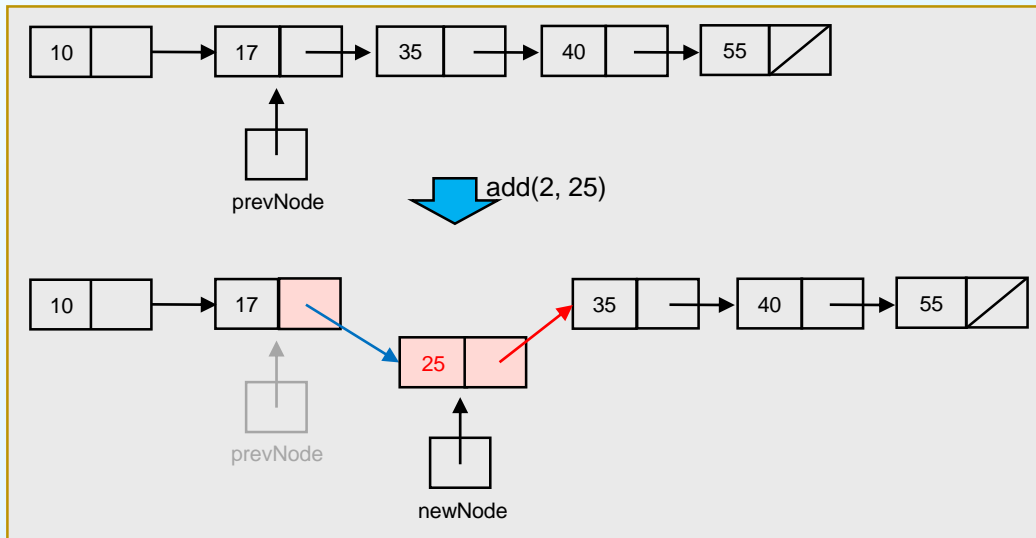
# Algorithm add()

```

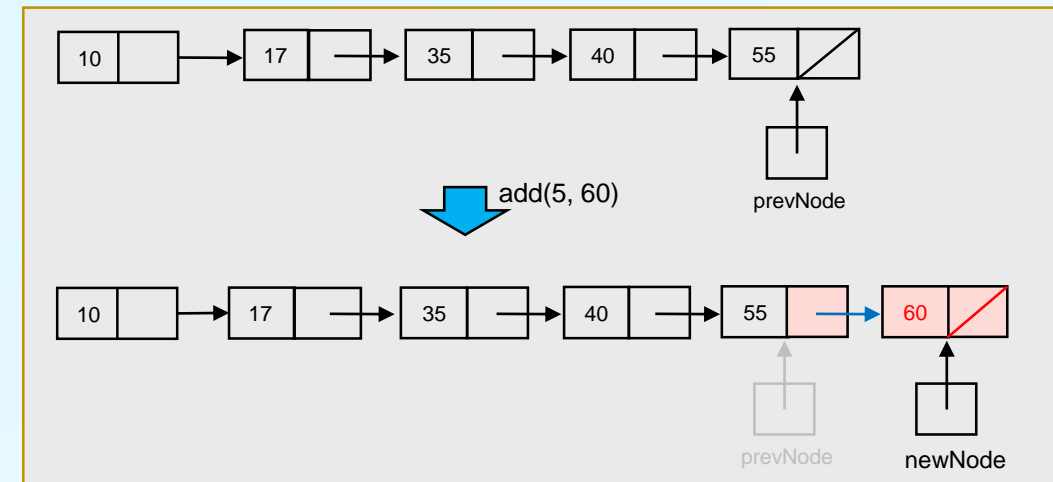
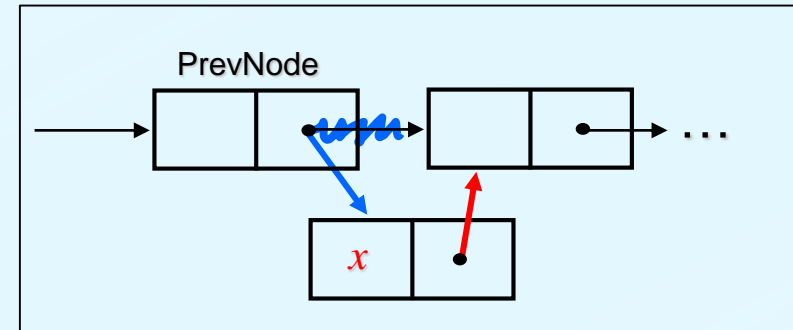
newNode.item ← x
newNode.next ← prevNode.next
prevNode.next ← newNode
numItems++

```

중간 삽입과 맨 끝 삽입은 이것으로 okay.  
But, 맨 앞 삽입은 작동하지 않는다.



중간에 삽입: Okay



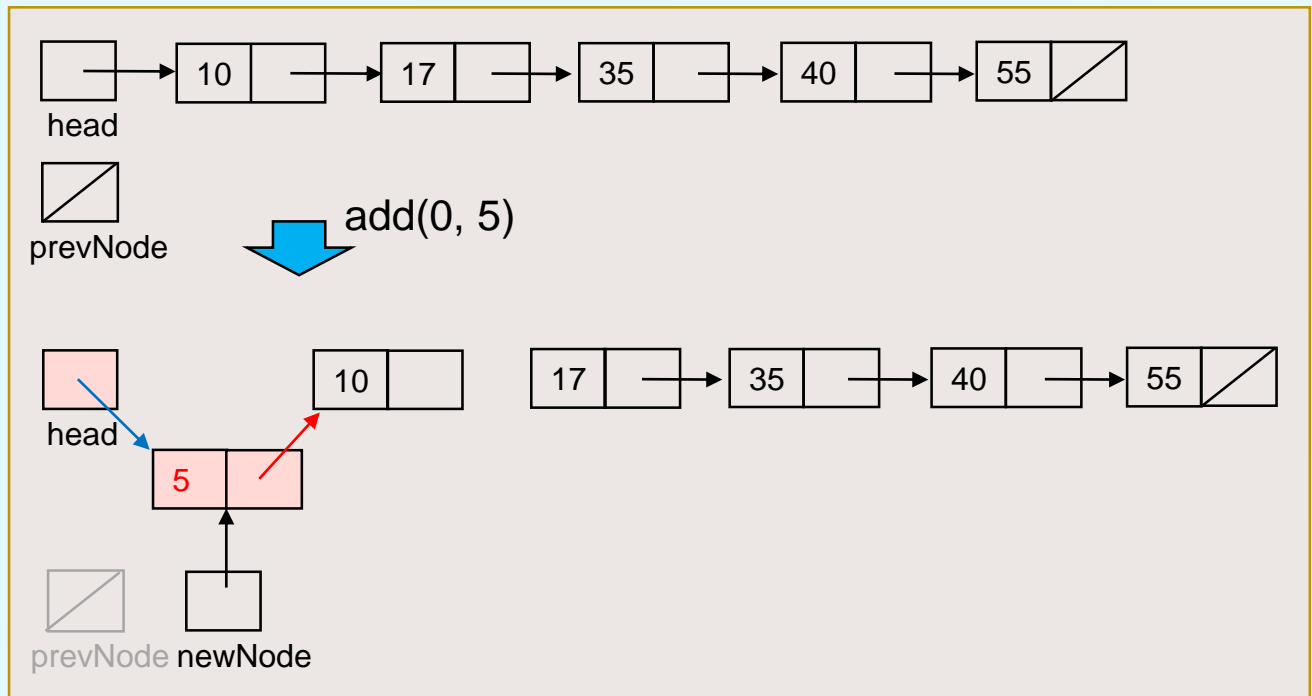
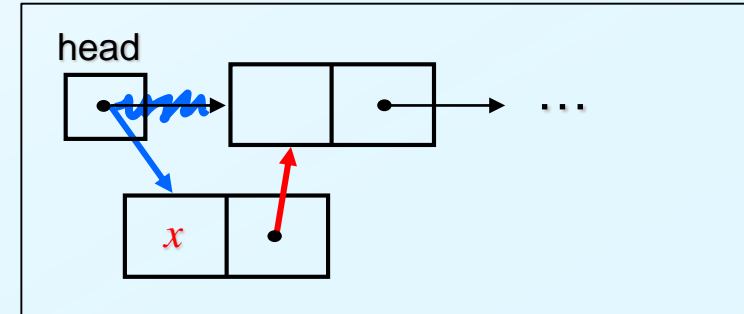
맨 끝에 삽입: Okay

```

newNode.item ← x
newNode.next ← head
head ← newNode
numItems++

```

맨 앞에 삽입할 때  
prevNode가 존재하지 않기 때문





```
add( $k, x$ ): ◀ Insert  $x$  as  $k^{\text{th}}$  element  
  if ( $k = 0$ )  
    newNode.item  $\leftarrow x$   
    newNode.next  $\leftarrow$  head  
    head  $\leftarrow$  newNode  
    numItems++  
  else  
    newNode.item  $\leftarrow x$   
    newNode.next  $\leftarrow$  prevNode.next  
    prevNode.next  $\leftarrow$  newNode  
    numItems++
```

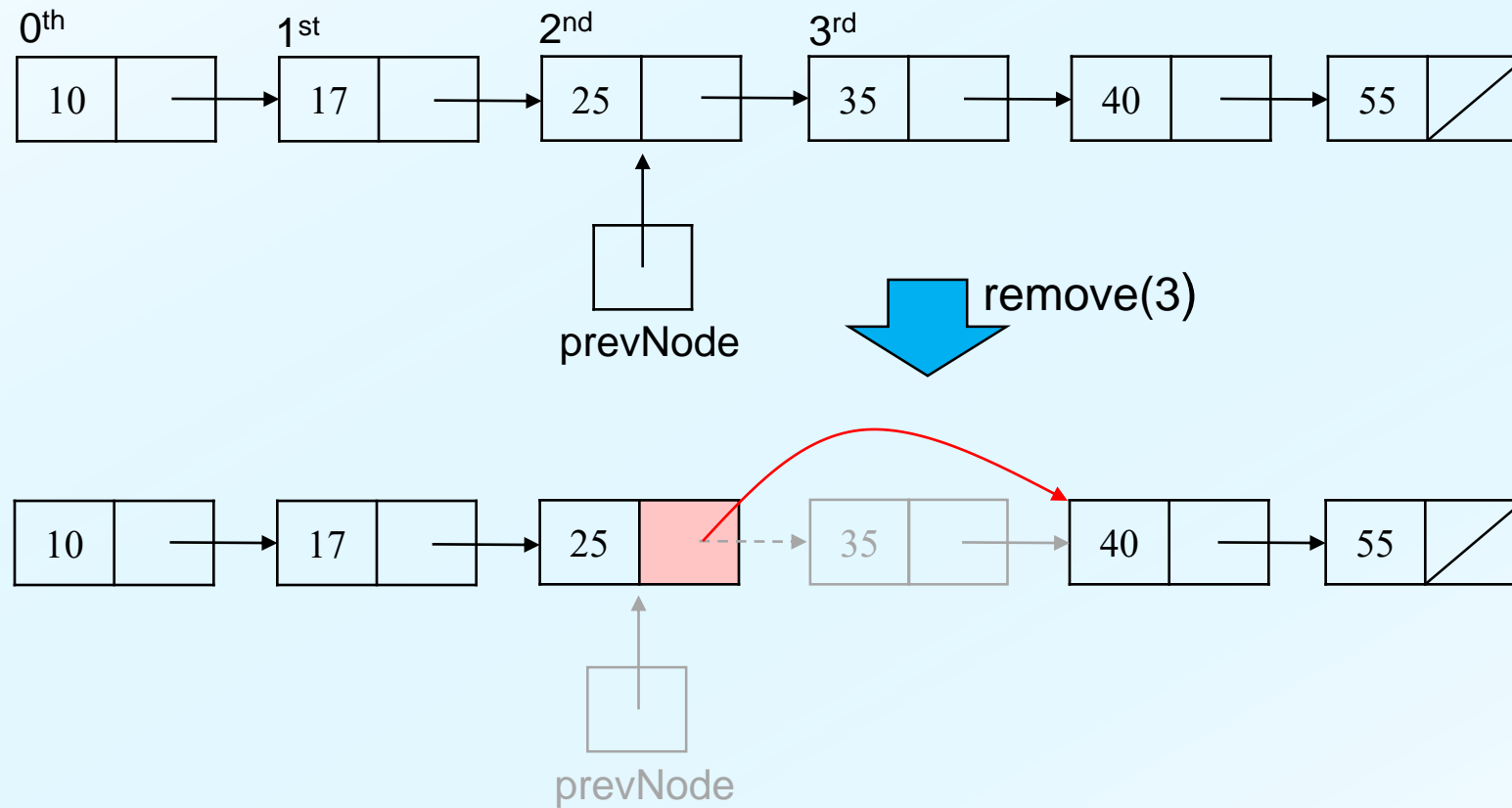
← 앞의 두가지 경우를 다 고려한

자주 나오지 않는 “맨 앞 삽입” 때문에  
이렇게 항상 두 가지 경우로 나누어 처리해야 하는가?

하나로 처리할 수 있는 방법이 있다 → dummy head node(slide #55부터 설명)

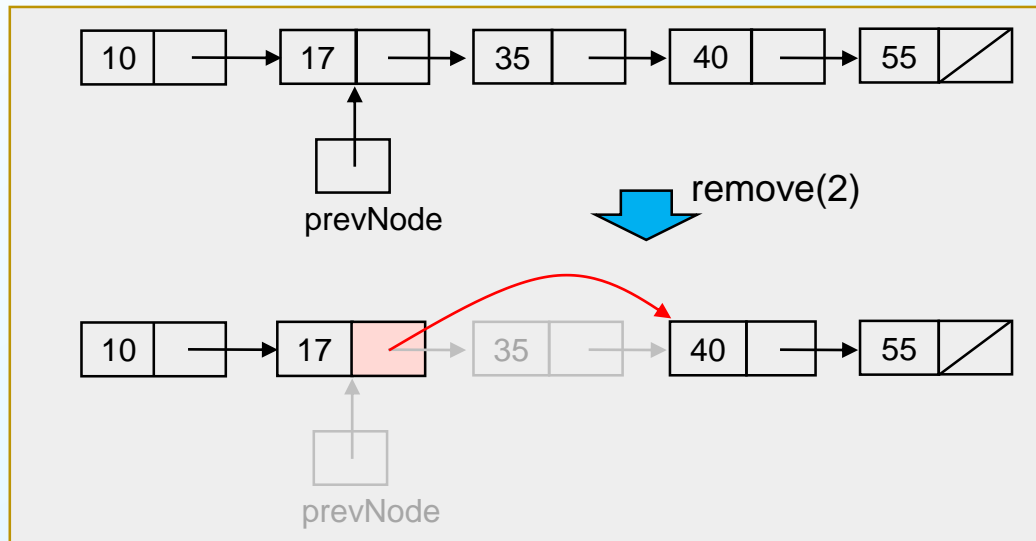
# Deletion

prevNode 다음 노드를 삭제한다

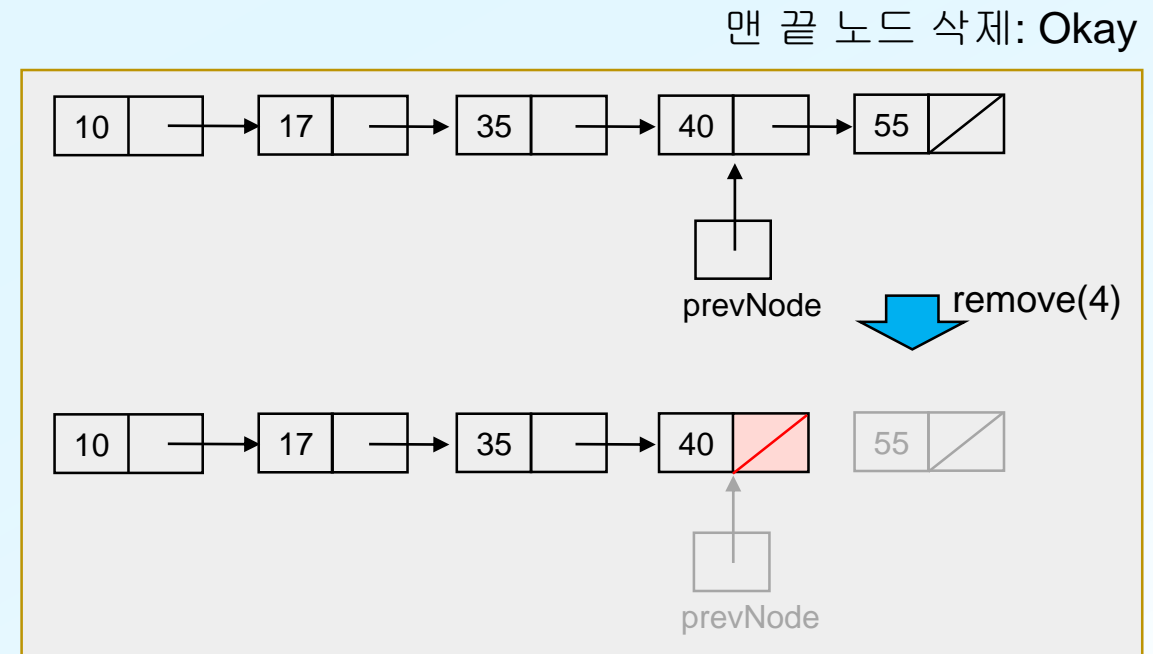
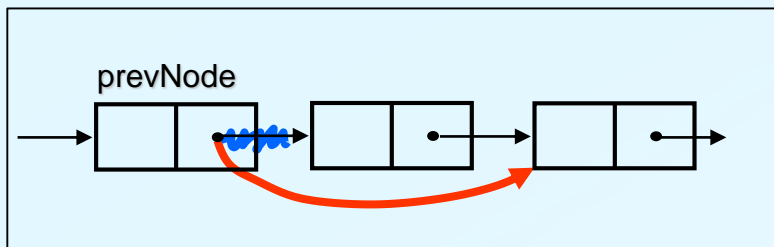


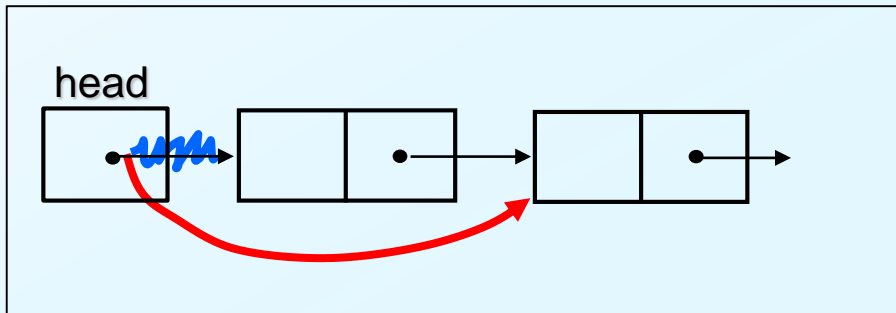
$\text{prevNode.next} \leftarrow \text{prevNode.next.next}$   
 $\text{numItems--}$

중간 노드 삭제와 맨 끝 노드 삭제는 이것으로 Okay  
 But, 맨 앞 노드 삭제는 작동하지 않는다



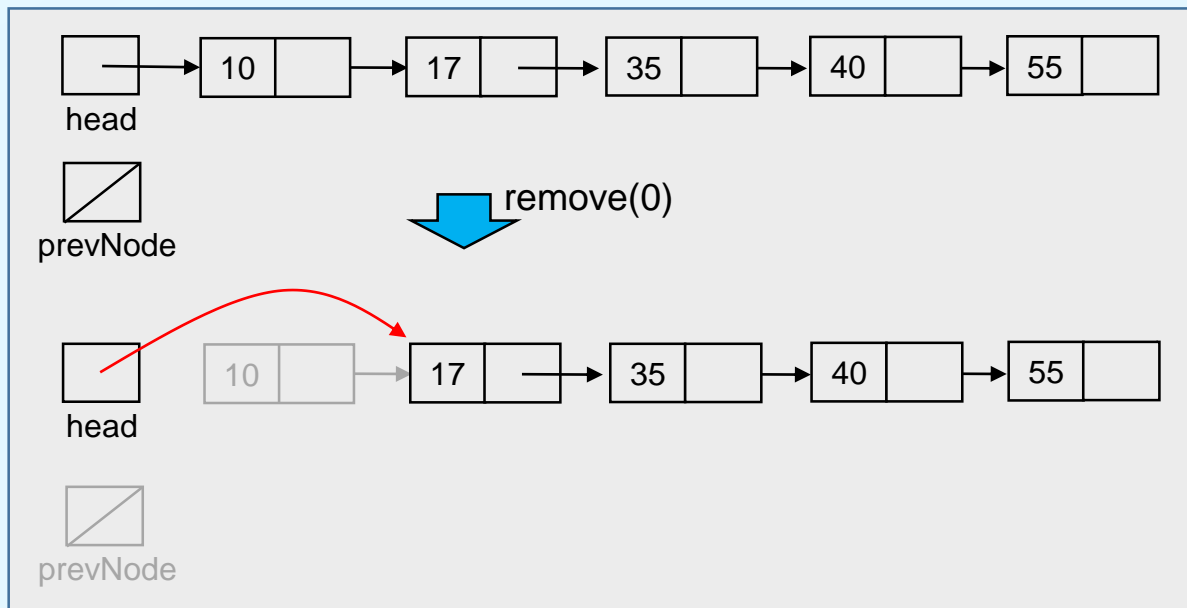
중간 노드 삭제: Okay





`head ← head.next`  
`numItems--`

← 맨 앞 노드를 삭제할 때  
`prevNode`가 존재하지 않기 때문



```
remove( $k$ ): ◀ Remove  $k^{\text{th}}$  element
```

```
  if ( $k = 0$ )
```

```
     $\text{head} \leftarrow \text{head.next}$ 
```

```
    numItems--
```

```
  else
```

```
     $\text{prevNode.next} \leftarrow \text{prevNode.next.next}$ 
```

```
    numItems--
```

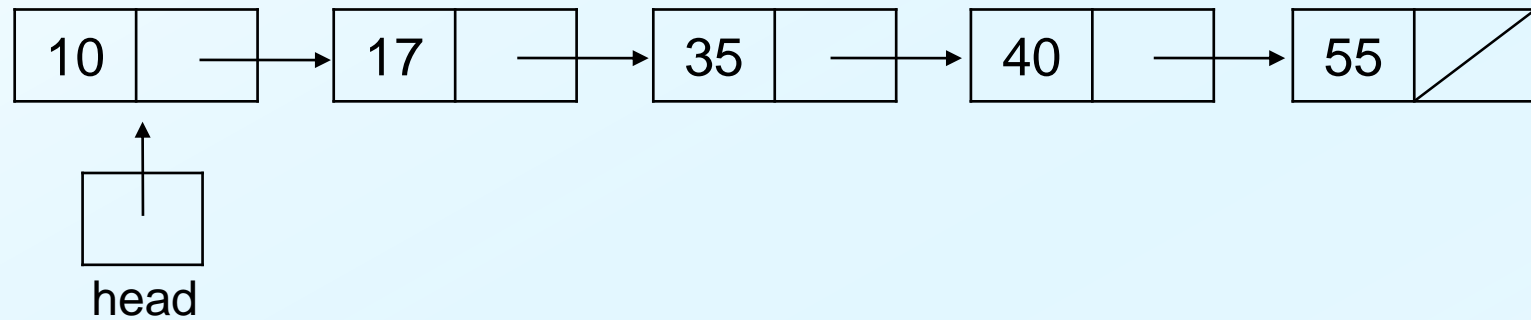
← 앞의 두가지 경우를 다 고려한

자주 나오지 않는 “맨 앞 삭제” 때문에  
이렇게 항상 두 가지 경우로 나누어 처리해야 하는가?

dummy head node를 두면 역시 하나로 처리된다

# Traversal thru the List

예: Linked List의 원소들을 앞에서부터 모두 프린트



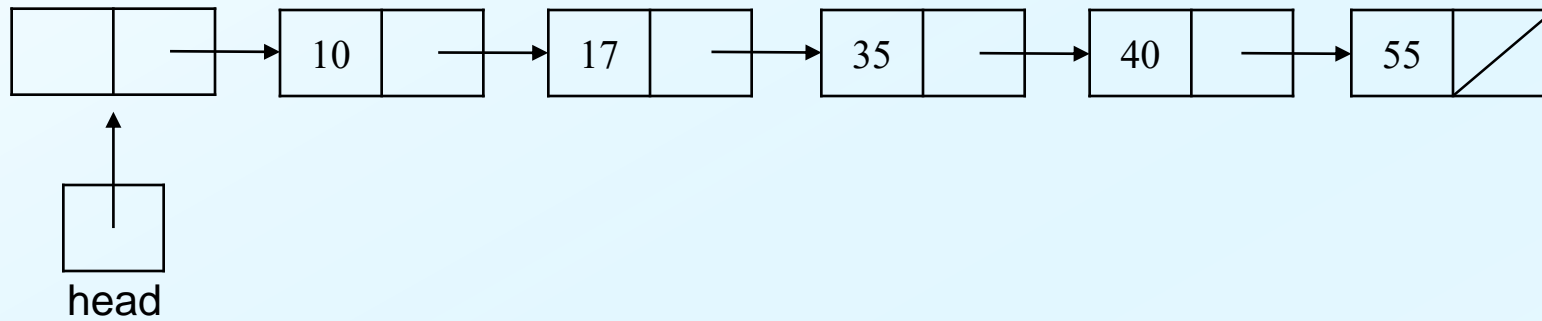
```
for (Node curr = head; curr != null; curr = curr.next) {  
    System.out.println(curr.item);  
}
```

```
10  
17  
35  
40  
55
```

# Dummy Head Node를 둔 Linked List

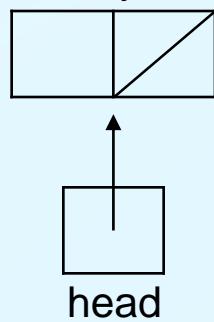
## 대표적 상태

dummy head



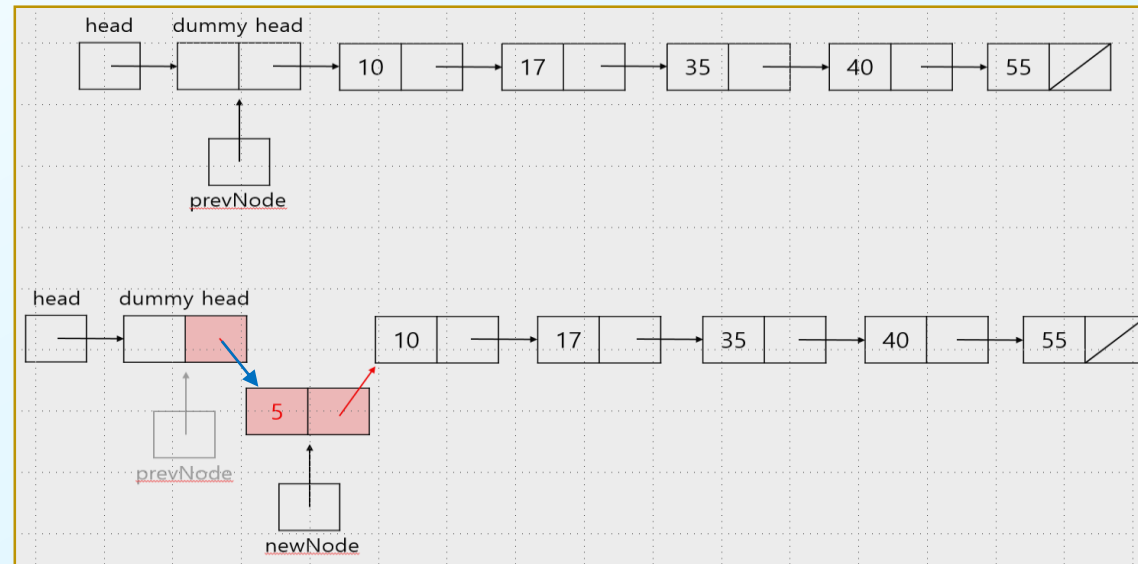
## 초기 상태

dummy head



← Empty list

# Insertion



```

newNode.next ← prevNode.next
prevNode.next ← newNode
numItems++

```

← Dummy head를 두면 이것으로 충분  
 항상 prevNode가 존재하기 때문



# Algorithm add()

```
add( $k, x$ ): ◀ Insert  $x$  as  $k^{\text{th}}$  element  
  prevNode  $\leftarrow$  getNode( $k - 1$ )  
  newNode.item  $\leftarrow x$   
  newNode.next  $\leftarrow$  prevNode.next  
  prevNode.next  $\leftarrow$  newNode  
  numItems++
```

```
getNode( $k$ ): ◀ 첫번째 노드를 0번 노드로 표기, dummy head는 -1번 노드로 간주  
  if ( $k \geq -1 \ \&\& \ k \leq \text{numItems}-1$ )  
    currNode  $\leftarrow$  head ◀ dummy head  
    for  $i \leftarrow 0$  to  $k$  ◀  $k^{\text{th}}$  node 찾기  
      currNode  $\leftarrow$  currNode.next  
    return currNode  
  else return null ◀ 범위 초과. 에러.
```

# Algorithm append()

맨 뒤에 원소  $x$ 를 추가

**append( $x$ ):**

prevNode  $\leftarrow$  끝 노드

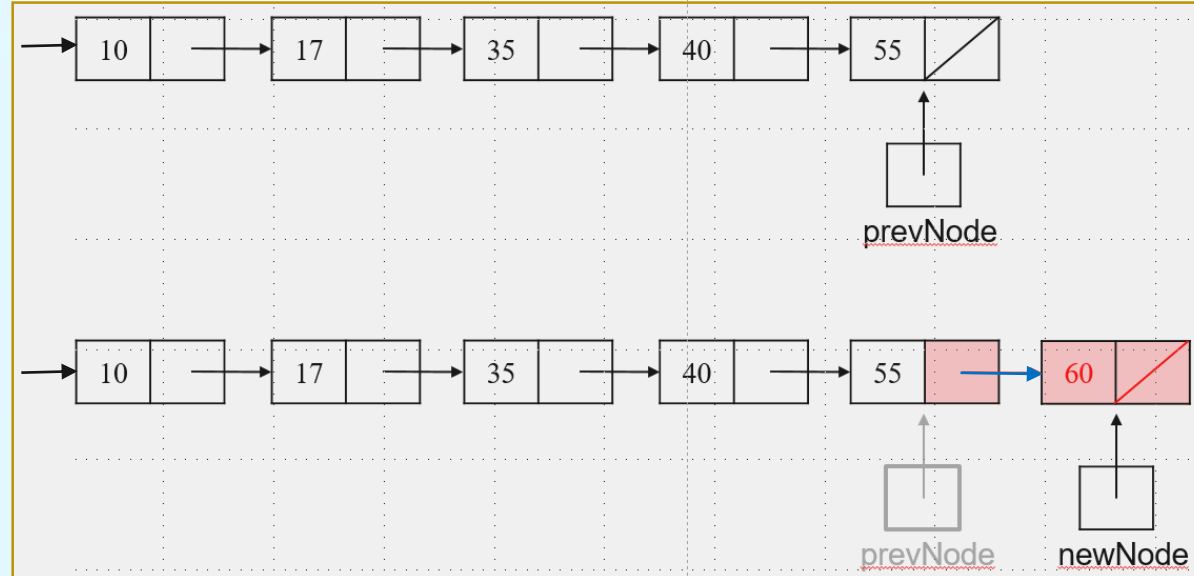
prevNode.item  $\leftarrow x$

newNode.next  $\leftarrow$  prevNode.next

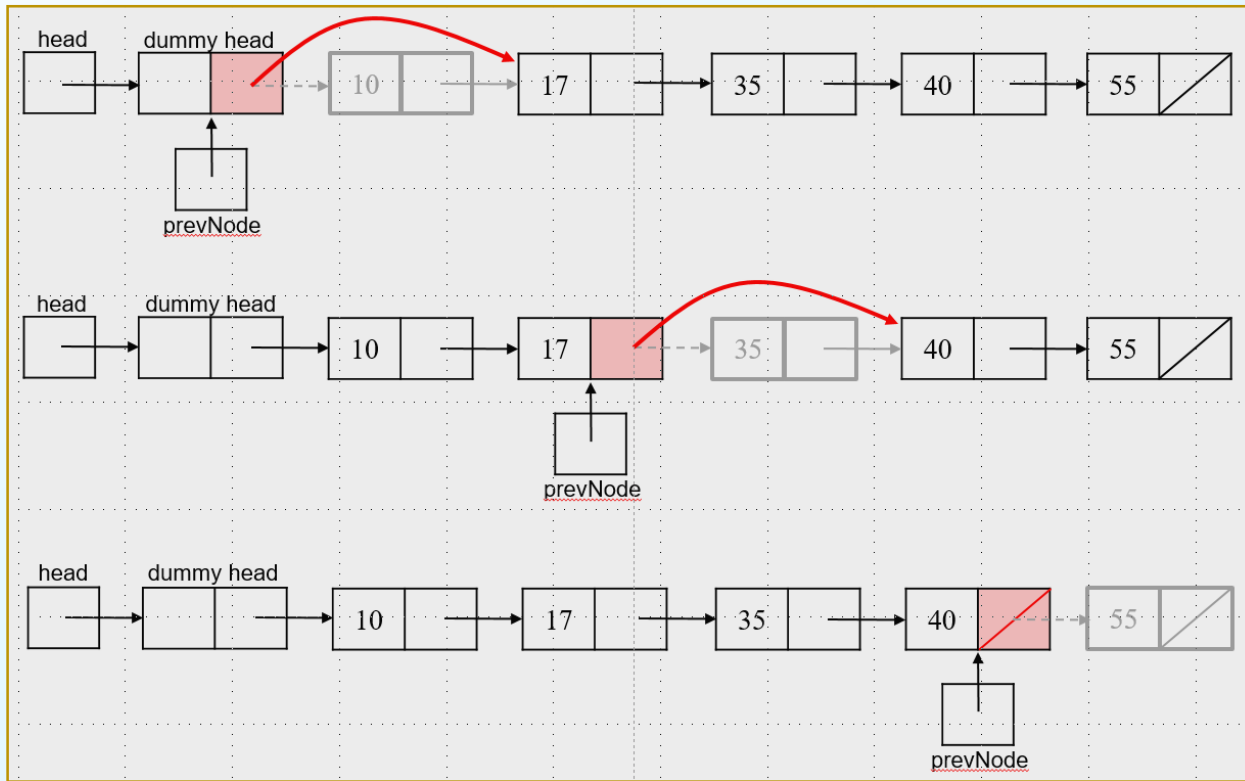
prevNode.next  $\leftarrow$  newNode

numItems++

← prevNode 찾는 부분 빼고는 add()의 코드와 동일



# Deletion



`prevNode.next ← prevNode.next.next`  
`numItems--`

← Dummy head를 두면 이것으로 충분  
 항상 `prevNode`가 존재하기 때문

# Algorithm remove()

$k$ 번째 원소 삭제

```
remove( $k$ ):  
    if ( $k \geq 0 \ \&\& \ k \leq \text{numItems}-1$ )  
        prevNode  $\leftarrow$  getNode( $k - 1$ ) ◀ prevNode 찾기. getNode(-1)은 dummy head  
        prevNode.next  $\leftarrow$  prevNode.next.next  
        numItems--  
        return currNode.item  
    else return null ◀ 에러
```

# Algorithm removeItem()

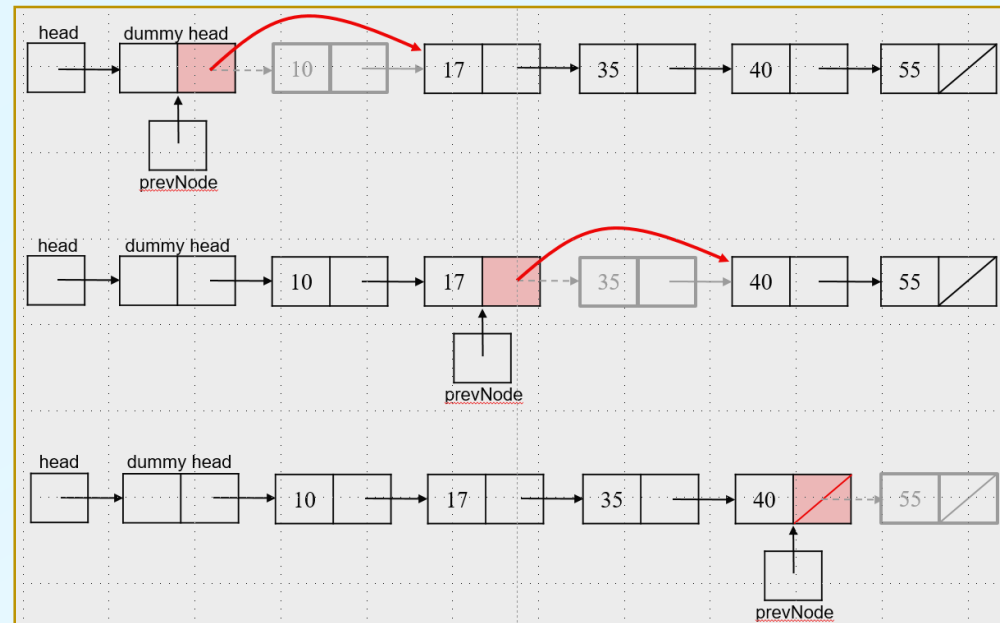
```

removeItem(x): ◀ Remove element x
  currNode ← head ◀ dummy head
  for i ← 0 to numItems-1
    prevNode ← currNode
    currNode ← currNode.next
    if (currNode.item = x)
      prevNode.next ← currNode.next
      numItems--
      return true
  return false;

```

원소  $x$  찾아 삭제

←  $k \leftarrow \text{indexOf}(x)$  후에  $\text{remove}(k)$  하면  
코드는 simple하다. But...



# 기타 작업

```

get(i):
    if ( $i \geq 0 \ \&\& \ i \leq \text{numItems}-1$ )
        return getNode(i).item
    else
        return OUT_OF_BOUND
  
```

```

set(i, x):
    if ( $i \geq 0 \ \&\& \ i \leq \text{numItems}-1$ )
        getNode(i).item  $\leftarrow$  x
    else return null ◀ 에러
  
```

```

indexOf(x):
    currNode  $\leftarrow$  head ◀ dummy head
    for  $i \leftarrow 0$  to numItems-1
        currNode  $\leftarrow$  currNode.next
        if (currNode.item = x) return i
    return NOT_FOUND
  
```

```

len():
    return numItems
  
```

```

isEmpty():
    if (numItems = 0)
        return true
    else
        return false
  
```

```

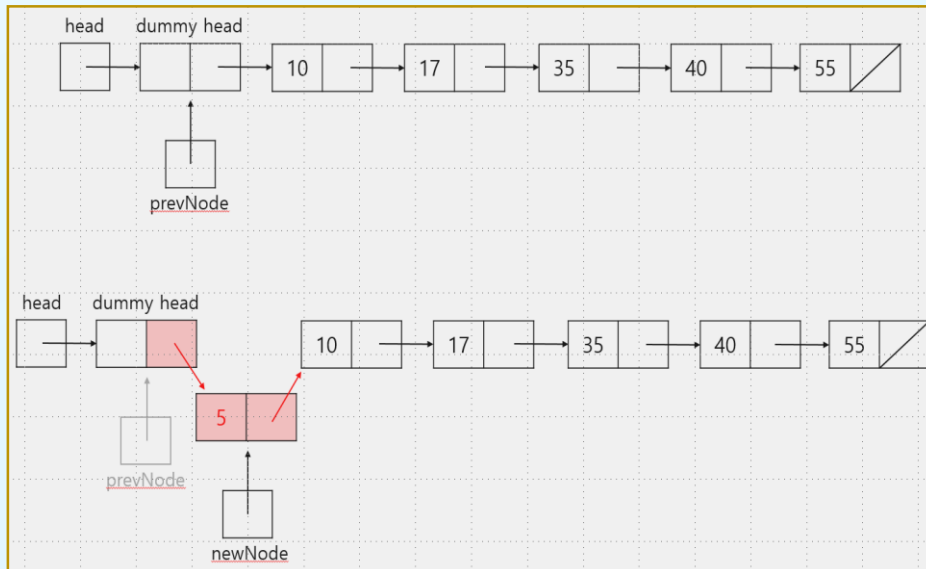
clear():
    newNode.next  $\leftarrow$  null
    head  $\leftarrow$  newNode
    numItems  $\leftarrow$  0
  
```

# Java 간이 코드

Node는 나중에  
Generic 버전으로 바꿈

```
void add(int k, Integer x) { // 첫번째 원소를 0번째 원소로 표기
    if (k >= 0 && k <= numItems) {
        Node prevNode = getNode(k - 1);
        Node newNode = new Node(x, prevNode.next);
        prevNode.next = newNode;
        numItems++;
    } else { /* 에러 처리 */ }
}
```

Java 코드



newNode.item  $\leftarrow x$   
 newNode.next  $\leftarrow$  prevNode.next  
 prevNode.next  $\leftarrow$  newNode  
 numItems++

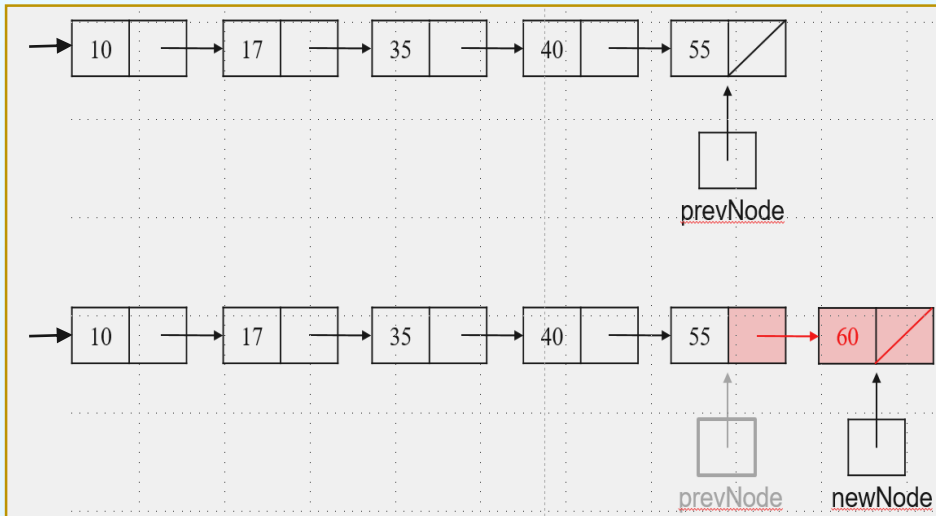
알고리즘 in pseudo code(유사 코드)

```

void append(Integer x) {
    Node prevNode = head; // 더미 헤드
    while (prevNode.next != null) // 끝 노드 찾기
        prevNode = prevNode.next;
    Node newNode = new Node(x, null);
    prevNode.next = newNode;
    numItems++;
}

```

Java 코드



```

newNode.item ← x
newNode.next ← prevNode.next
prevNode.next ← newNode
numItems++

```



```

Integer remove(int k) {
    if (k >= 0 && k <= numItems-1) {
        Node prevNode = getNode(k - 1);
        Node currNode = prevNode.next;
        prevNode.next = currNode.next;
        numItems--;
        return currNode.item;
    } else return null; // 예러
}

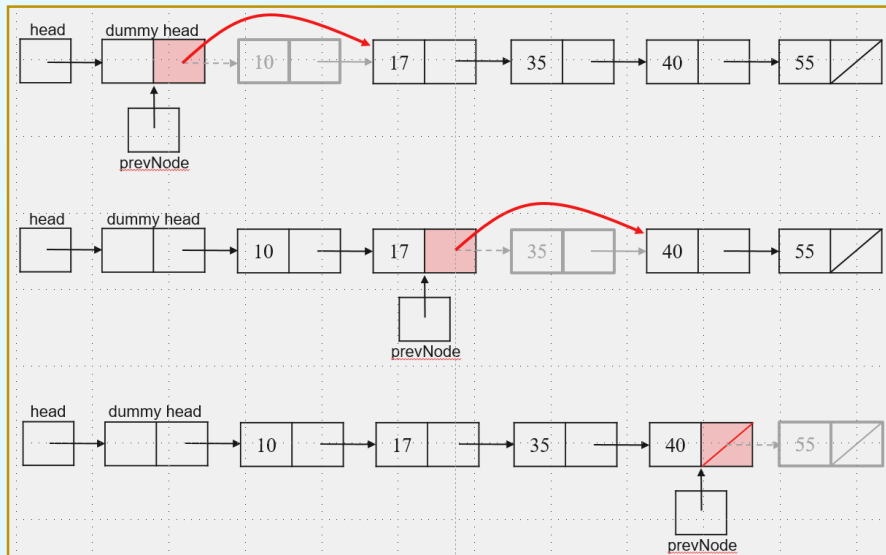
```

Java 코드

```

Node getNode(int k) { // 첫번째 노드를 0번째 노드로 표기
    // return reference to kth node
    if (k >= -1 && k <= numItems-1) {
        Node currNode = head; // dummy head
        for (int i = 0; i <= k; i++)
            currNode = currNode.next;
        return currNode;
    } else return null; // 예러
}

```



```

prevNode.next ← prevNode.next.next
numItems--

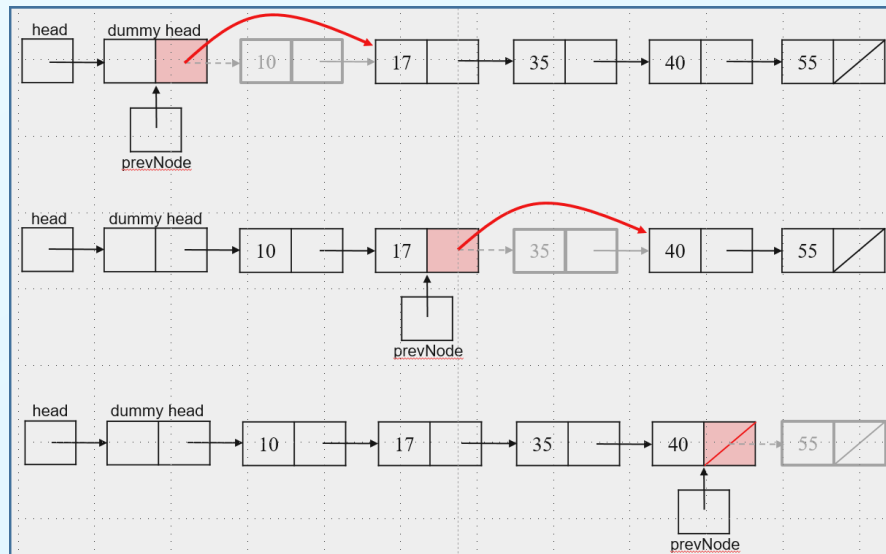
```

```

boolean removeItem(Integer x) {
    Node currNode = head, prevNode; // head: dummy node
    for (int i = 0; i <= numItems-1; i++) {
        prevNode = currNode;
        currNode = currNode.next;
        if (currNode.item.compareTo(x) == 0) {
            prevNode.next = currNode.next;
            numItems--;
            return true;
        }
    }
    return false;
}

```

Java 코드



**removeItem( $x$ ):** ◀ Remove element  $x$   
 $\text{currNode} \leftarrow \text{head}$  ◀ dummy node  
**for**  $i \leftarrow 0$  **to**  $\text{numItems}-1$   
      $\text{prevNode} \leftarrow \text{currNode}$   
      $\text{currNode} \leftarrow \text{currNode.next}$   
     **if** ( $\text{currNode.item} = x$ )  
          $\text{prevNode.next} \leftarrow \text{currNode.next}$   
          $\text{numItems}--$   
         **return true**  
**return false;**

자바 코드

```
Integer get(int k) {
    if (k >= 0 && k <= numItems-1)
        return getNode(k).item;
    else return null; // 에러
}
```

```
void set(int k, Integer x) {
    if (k >= 0 && k <= numItems-1)
        getNode(k).item = x;
    else { /* 에러 처리 */ }
}
```

```
int indexOf(Integer x) { // return item x's index
    Node currNode = head; // 더미 헤드
    for (int i = 0; i <= numItems-1; i++) {
        currNode = currNode.next;
        if (currNode.item.compareTo(x) == 0)
            return i;
    }
    return NOT_FOUND;
}
```

```
int len() {
    return numItems;
}
```

```
boolean isEmpty() {
    return numItems == 0;
}
```

```
void clear() {
    head = new Node(null, null);
    numItems = 0;
}
```



```
get(k):
    if (k >= 0 && k <= numItems-1)
        return getNode(k).item
    else
        return OUT_OF_BOUND
```

```
set(k, x):
    if (k >= 0 && k <= numItems-1)
        getNode(k).item ← x
    else return null ◀ 에러
```

```
indexOf(x):
    currNode ← head ◀ dummy head
    for i ← 0 to numItems-1
        currNode ← currNode.next
        if (currNode.item = x) return i
    return NOT_FOUND
```

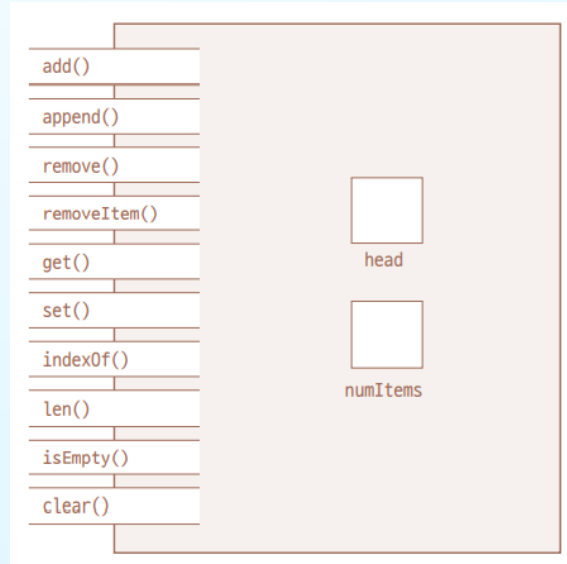
```
len():
    return numItems
```

```
isEmpty():
    if (numItems = 0)
        return true
    else
        return false
```

```
clear():
    newNode.next ← null
    head ← newNode
    numItems ← 0
```

# Java 제네릭 버전

## 객체 구조



```

public interface ListInterface<E> {
    public void add(int i, E x);
    public void append(E x);
    public E remove(int i);
    public boolean removeItem(E x);
    public E get(int i);
    public void set(int i, E x);
    public int indexOf(E x);
    public int size();
    public boolean isEmpty();
    public void clear();
}
  
```

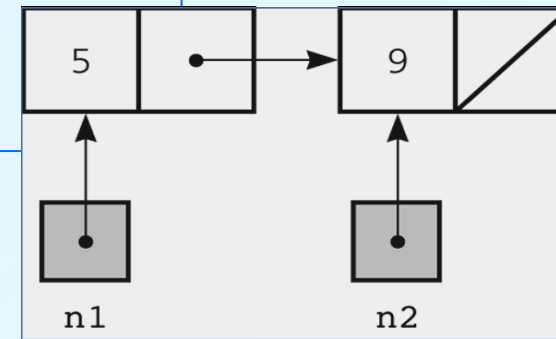
```

public class Node<E> {
    public E item;
    public Node<E> next;
    public Node(E newItem) {
        item = newItem; next = null;
    }
    public Node(E newItem, Node<E> nextNode) {
        item = newItem; next = nextNode;
    }
}
  
```

제네릭 인터페이스  
(Generic Interface)

# 생성자 Constructor 사용

```
public class Node {  
    public int item;  
    public Node next;  
    // constructors  
    public Node(int newItem) { // constructor 1  
        item = newItem;  
        next = null;  
    }  
    public Node(int newItem, Node nextNode) { // constructor 2  
        item = newItem;  
        next = nextNode;  
    }  
}
```



Example:

Node n2 = new Node(9);

Node n1 = new Node(5, n2);

# A Reusable Version

- ✓ The **Object** class is a superclass of every class

```
public class Node {  
    private Object item;  
    private Node next;  
    public Node(Object newItem) {  
        item = newItem;  
        next = null;  
    }  
    public Node(Object newItem, Node nextNode) {  
        item = newItem;  
        next = nextNode;  
    }  
}
```

- ✓ 범용성이 있다
- ✓ But, 최대한의 범용성을 갖는 것이 반드시 제일 좋은 것은 아니다
- ✓ 타입 관리가 느슨하다(**Object**가 쓰인 곳이 모두 같은 타입이라는 의미는 없다)

# Generic Class 사용

```
public class Node<E> {
    public E item;
    public Node<E> next;
    public Node(E newItem) {
        item = newItem; next = null;
    }
    public Node(E newItem, Node<E> nextNode) {
        item = newItem; next = nextNode;
    }
}
```

- ✓ 범용성을 가지면서
- ✓ 타입 관리가 강력하다

```
public class SampleClass<E, T> {
    public E item;
    public Node<E> next;
    ...
    public T function1(E newItem, ...) {
        ...
    }
}
```

- ✓ parameter가 여럿 있을 수도 있다

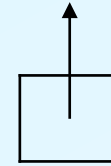
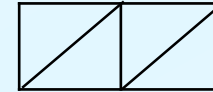
클래스 시작

타입 **E**는 Comparable을 상속(클래스 또는 인터페이스)

```
public class LinkedList<E extends Comparable> implements ListInterface<E> {
    private Node<E> head;
    private int numItems;
    public LinkedList() {
        numItems = 0;
        head = new Node<>(null, null);
    }
    ...
}
```

1/4 of class LinkedList

dummy head



head



## Methods 구현

```

public void add(int index, E x) { // 첫번째 원소를 0번째 원소로 표기
    if (index >= 0 && index <= numItems) {
        Node<E> prevNode = getNode(index - 1);
        Node<E> newNode = new Node<>(x, prevNode.next);
        prevNode.next = newNode;
        numItems++;
    } else { /* 에러 처리 */ }
}

public void append(E x) {
    Node<E> prevNode = head; // 더미 헤드
    while (prevNode.next != null)
        prevNode = prevNode.next;
    Node<E> newNode = new Node<>(x, null);
    prevNode.next = newNode;
    numItems++;
}

public E remove(int index) {
    if (index >= 0 && index <= numItems-1) {
        Node<E> prevNode = getNode(index - 1);
        prevNode.next = prevNode.next.next;
        numItems--;
        return currNode.item;
    } else return null; // 에러
}

```

```

public boolean removeItem(E x) {
    Node<E> currNode = head; // 더미 헤드
    for (i = 0; i <= numItems-1; i++) {
        Node<E> prevNode = currNode;
        currNode = prevNode.next;
        if (currNode.item.compareTo(x) == 0) {
            prevNode.next = currNode.next;
            numItems--;
            return true;
        }
    } else return false;
}

public E get(int index) { // 첫번째 원소를 0번째 원소로 표기
    if (index >= 0 && index <= numItems-1)
        return getNode(index).item;
    else return null; // 에러
}

public void set(int index, E x) {
    if (index >= 0 && index <= numItems-1)
        getNode(index).item = x;
    else { /* 에러 처리 */ }
}

private Node<E> getNode(int index) { // 첫번째 노드를 0번째 노드로 표기
    // return reference to indexth node
    if (index >= -1 && index <= numItems-1) {
        Node<E> currNode = head; // dummy head
        for (int i = 0; i <= index; i++)
            currNode = currNode.next;
        return currNode;
    } else return null; // 에러
}

```

p.72 선언부에 LinkedList<E extends Comparable> 대신  
LinkedList<E>로 했으면  
((Comparable)(currNode.item)).compareTo(x) == 0 로 typecasting 필요

```
private final int NOT_FOUND = -1;
public int indexOf(E x) { // return item x's index
    Node<E> currNode = head; // 더미 헤드
    for (int i = 0; i <= numItems-1; i++) {
        currNode = currNode.next;
        if (currNode.item.compareTo(x) == 0) return i;
    }
    return NOT_FOUND;
}
public int len() {
    return numItems;
}
public boolean isEmpty() {
    return numItems == 0;
}
public void clear() {
    numItems = 0;
    head = new Node(null, null);
}
} // End LinkedList
```

# Linked List 객체 사용하기

이 클래스의 객체를 생성해서 사용할 때는

```
LinkedList<Integer> list = new LinkedList<>();  
list.add(0, 300); // 오토박싱으로 300은 Integer(300)으로 취급해준다  
list.add(0, 100);  
list.append(500);  
list.remove(2);  
list.append(700);  
list.remove(1);
```

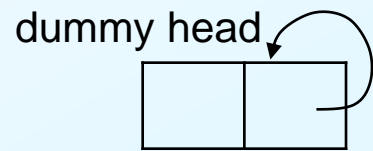
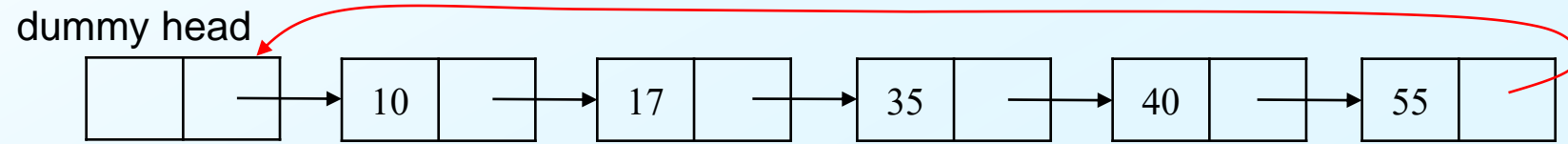
원소가 다른 타입이면 <E> 부분만 변경

```
LinkedList<String> list = new LinkedList<>();  
list.add(0, "Test String 1");  
list.add(0, "lion in oil");  
list.append("appended");  
list.remove(2);
```

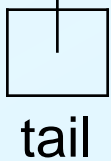
## **4. Extended Linked List**

# 원형 연결 리스트 Circular Linked List

끝 노드의 next가 null 값을 갖는 대신 첫 노드를 링크한다



초기 상태: Empty list

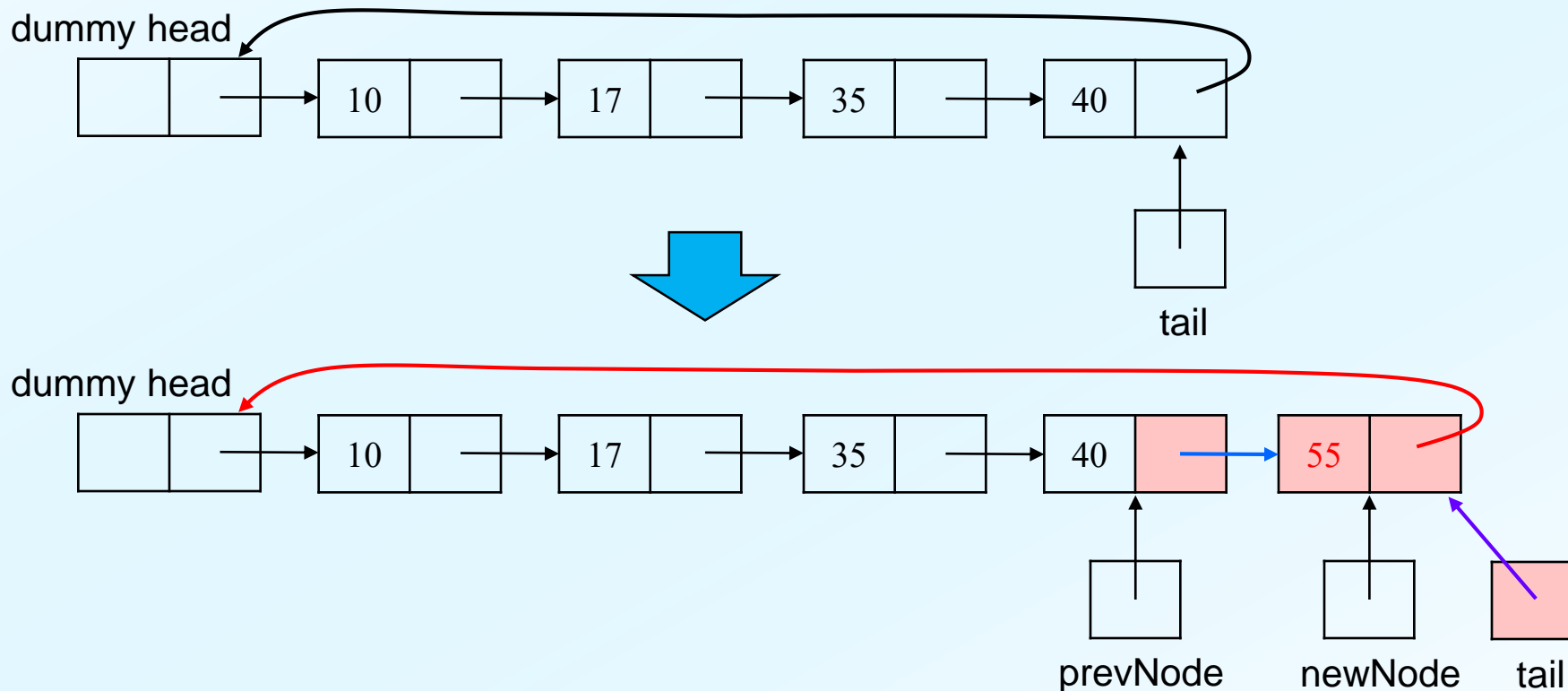


head 레퍼런스 대신 tail의 레퍼런스  
생각해 보기: 리스트의 맨 앞과 맨 뒤의 접근성

Dummy Head를 가진 Circular Linked List의 예

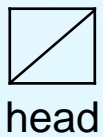
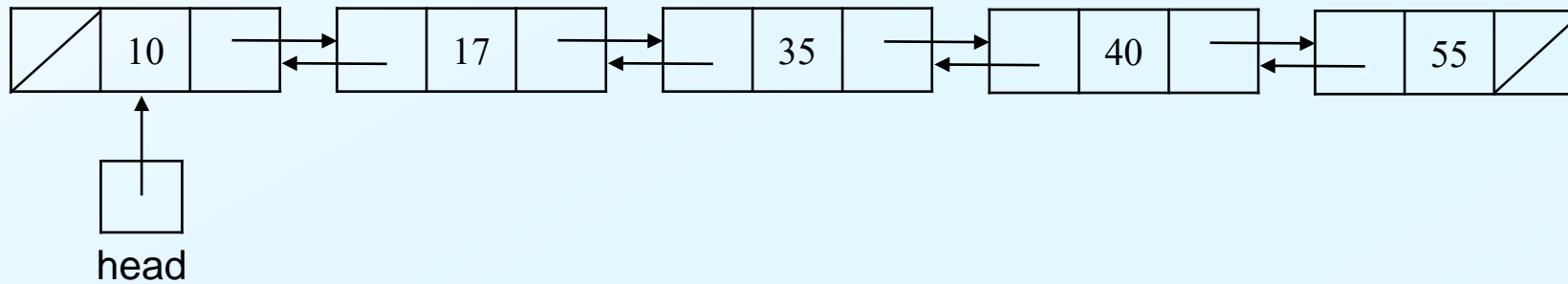
# Dummy Head를 가진 Circular Linked List에서의 삽입(위치 무관)

```
Node<E> prevNode = getNode(k - 1);
Node<E> newNode = new Node<>(x, prevNode.next);
prevNode.next = newNode;
if (k == numItems) tail = newNode;
numItems++;
```



# 양방향 연결 리스트 Doubly Linked List

Doubly Linked List의 예



초기 상태: Empty list



# 노드 구조



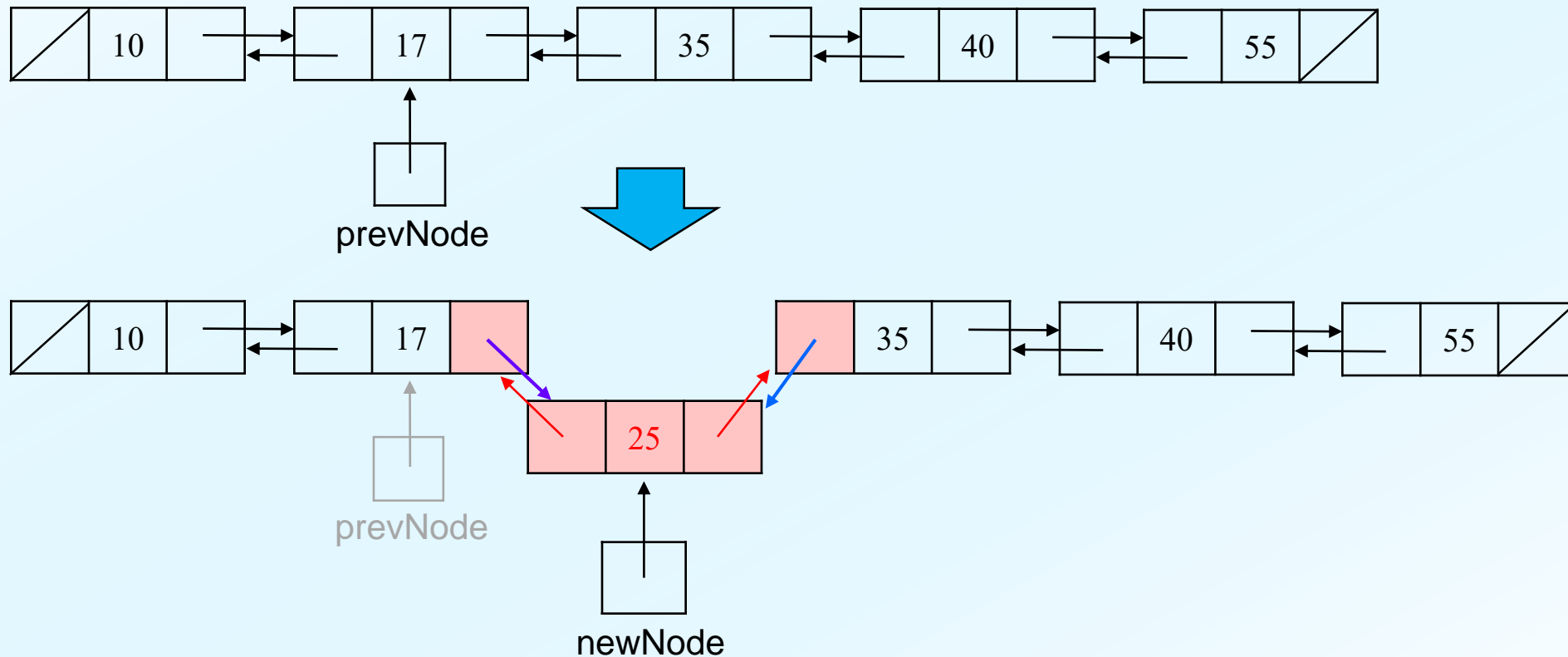
```
public class BidirectionalNode<E> {  
    public E item;  
    public BidirectionalNode<E> prev; // 앞 노드 레퍼런스  
    public BidirectionalNode<E> next; // 다음 노드 레퍼런스  
    public BidirectionalNode(E newItem) {  
        item = newItem;  
        prev = next = null;  
    }  
    public BidirectionalNode(E newItem,  
        BidirectionalNode<E> prevNode, BidirectionalNode<E> nextNode) {  
        item = newItem;  
        prev = prevNode;  
        next = nextNode;  
    }  
}
```

## 중간 삽입(맨 뒤 삽입 포함)

```

BidirectionalNode<E> newNode
    = new BidirectionalNode<>(newItem, prevNode, prevNode.next);
if (prevNode.next != null) prevNode.next.prev = newNode;
prevNode.next = newNode;
numItems++;

```

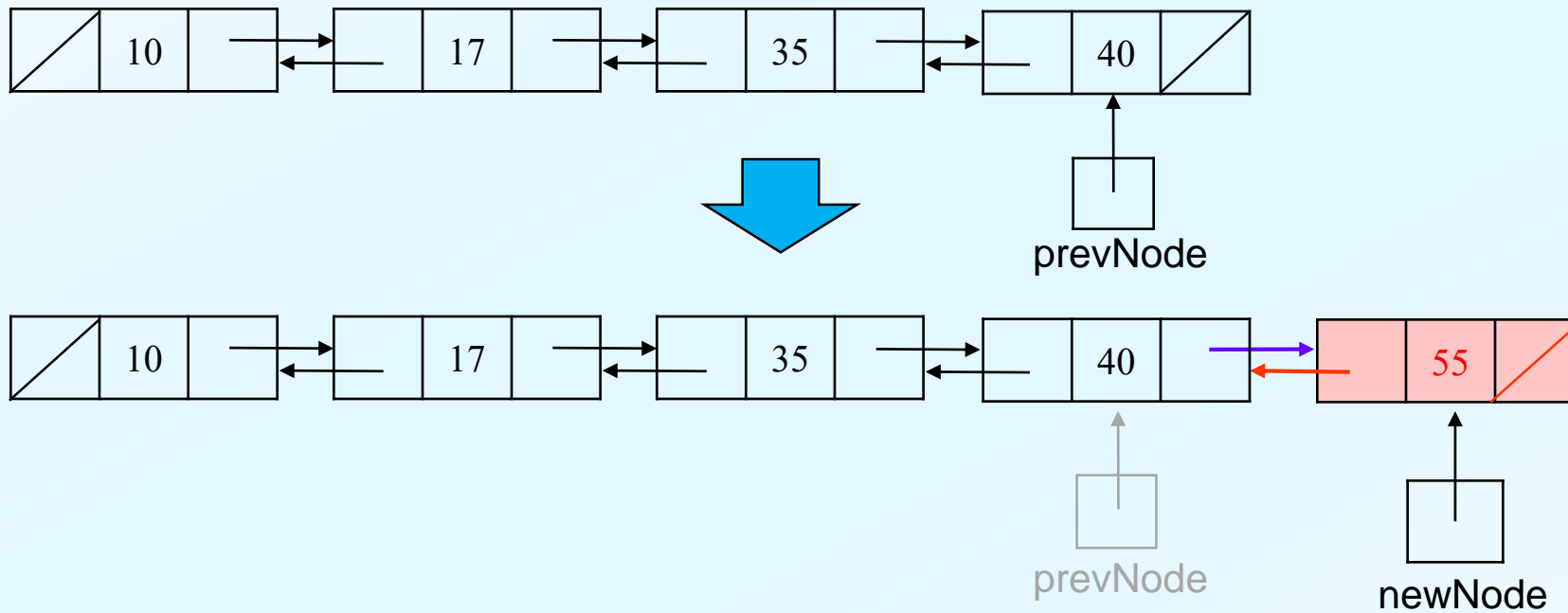


## 맨 뒤 삽입 확인 (앞과 동일 코드)

```

BidirectionalNode<E> newNode
    = new BidirectionalNode<>(newItem, prevNode, prevNode.next);
if (prevNode.next != null) prevNode.next.prev = newNode;
prevNode.next = newNode;
numItems++;

```

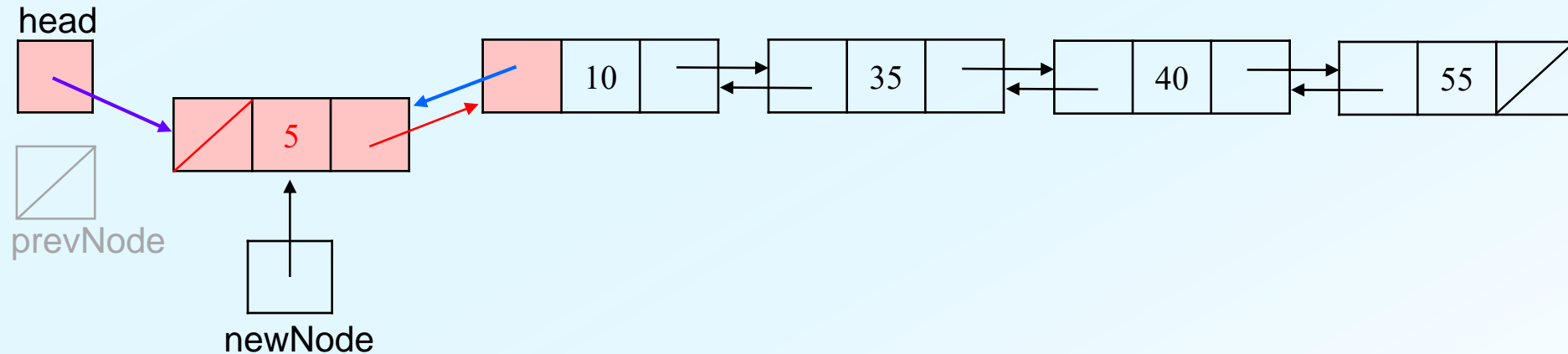
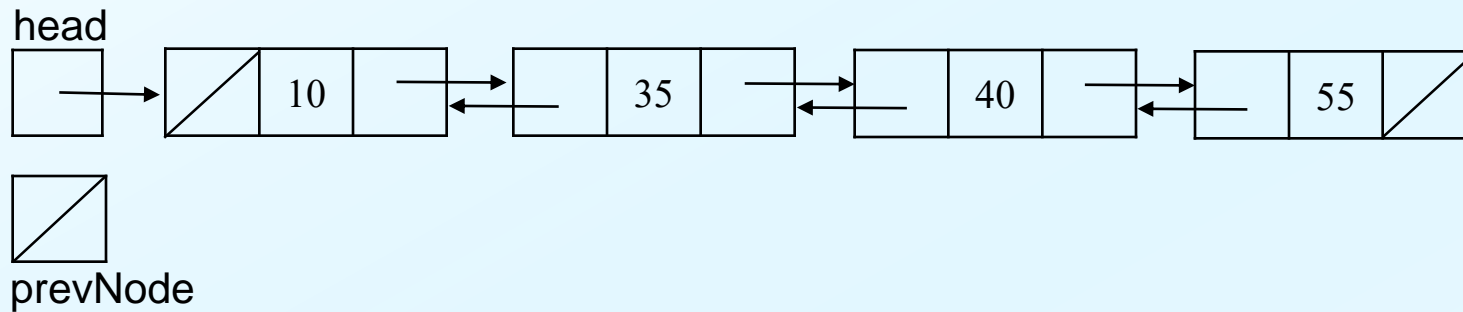


## 맨 앞 삽입 (앞과 다르다)

```

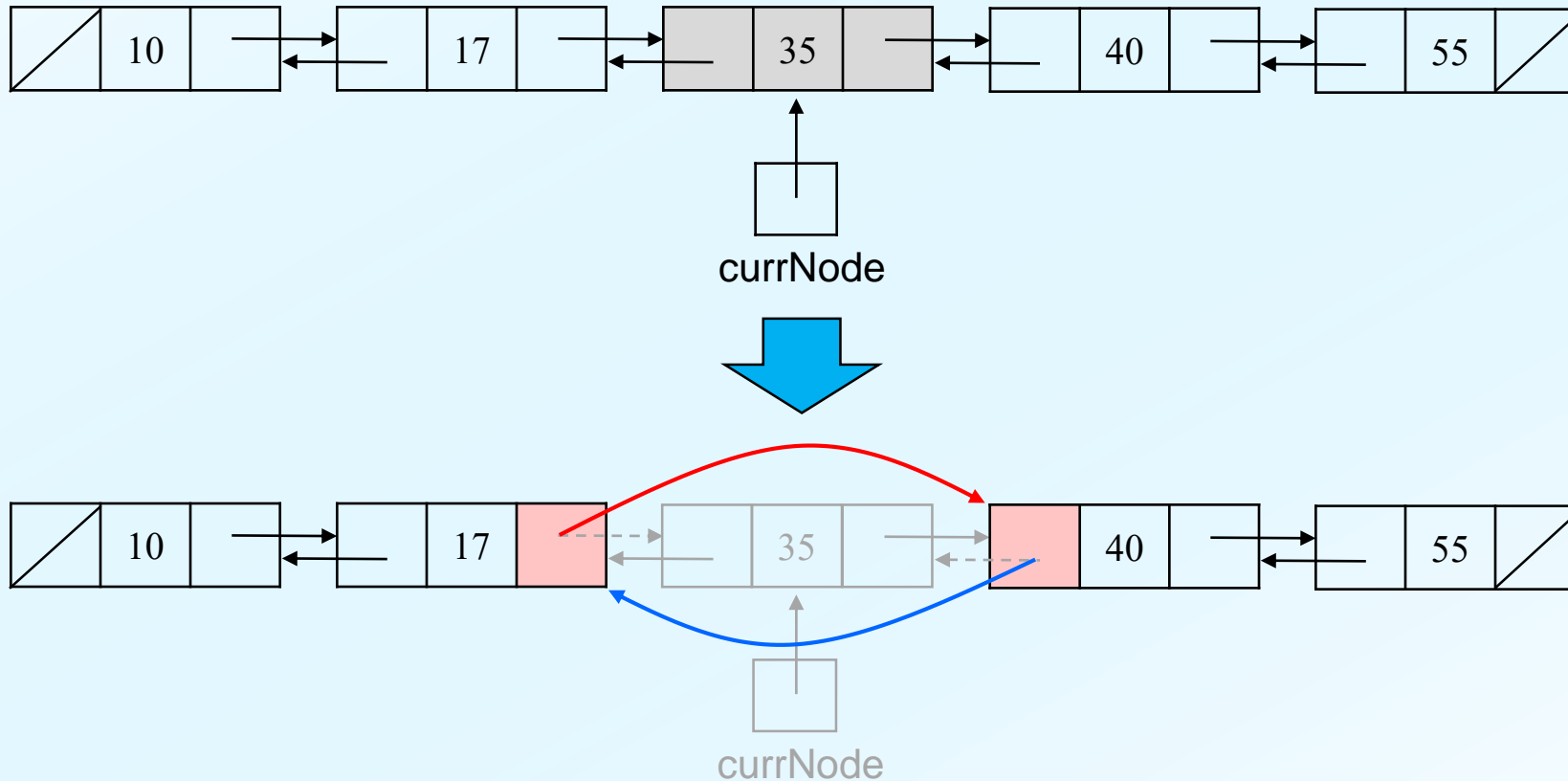
BidirectionalNode<E> newNode = new BidirectionalNode<>(newItem, null, head);
head.prev = newNode;
head = newNode;
numItems++;

```



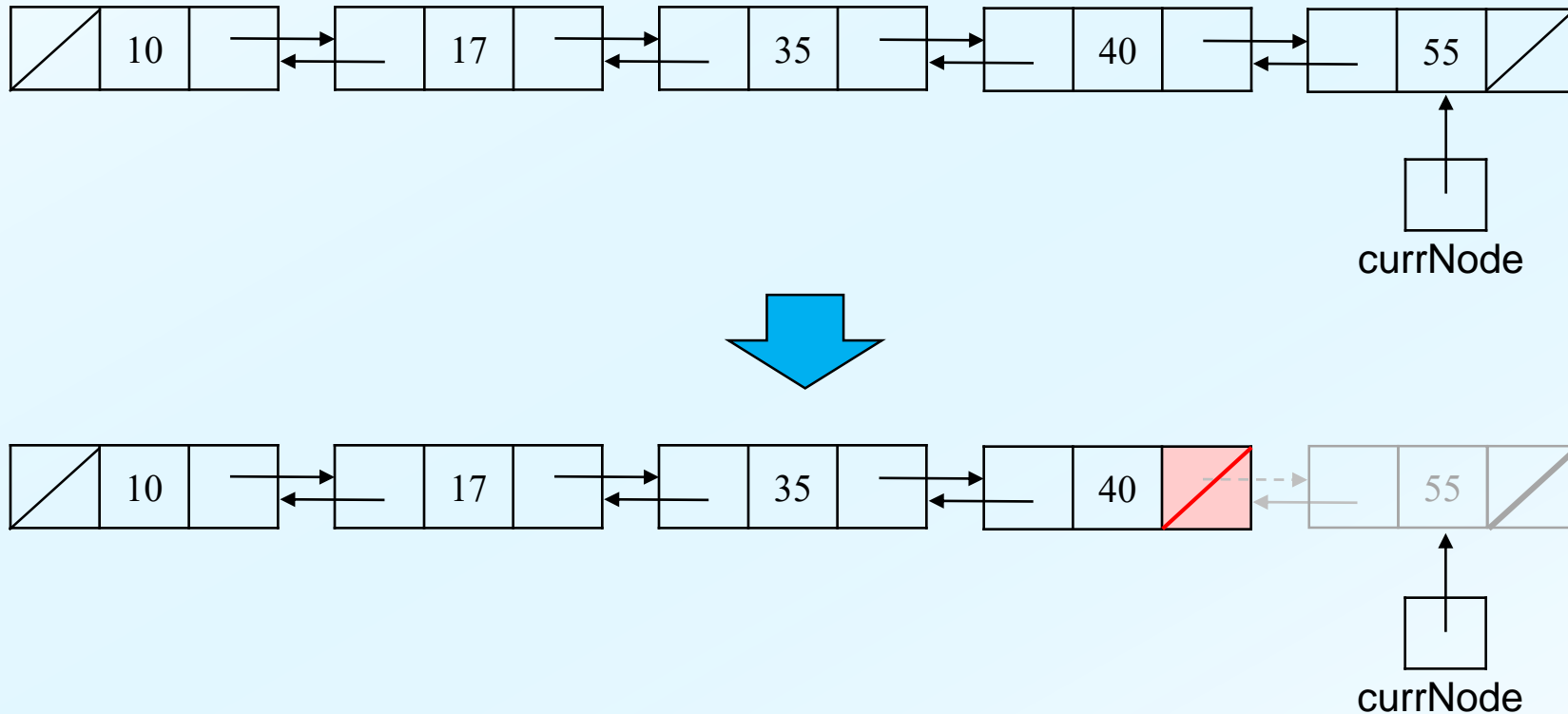
## 중간 삭제(맨 뒤 삭제 포함)

```
currNode.prev.next = currNode.next;  
if (prevNode.next != null) currNode.next.prev = currNode.prev;  
numItems--;
```



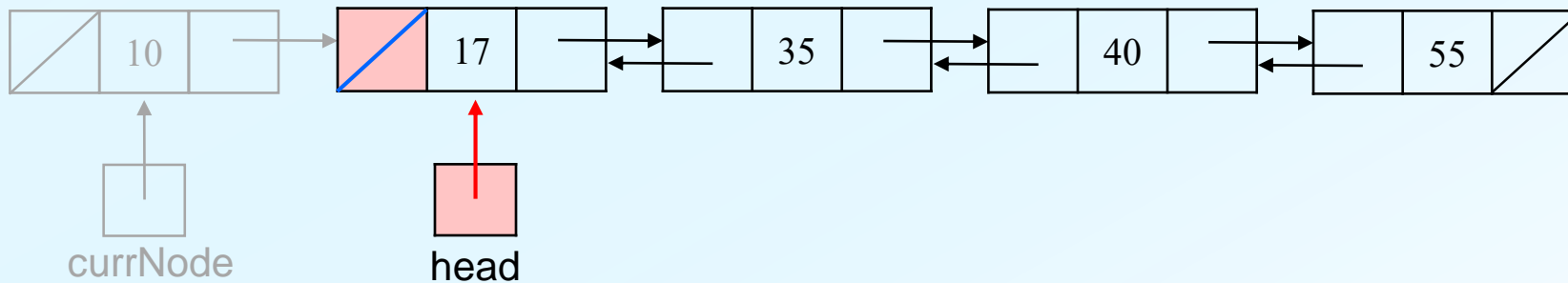
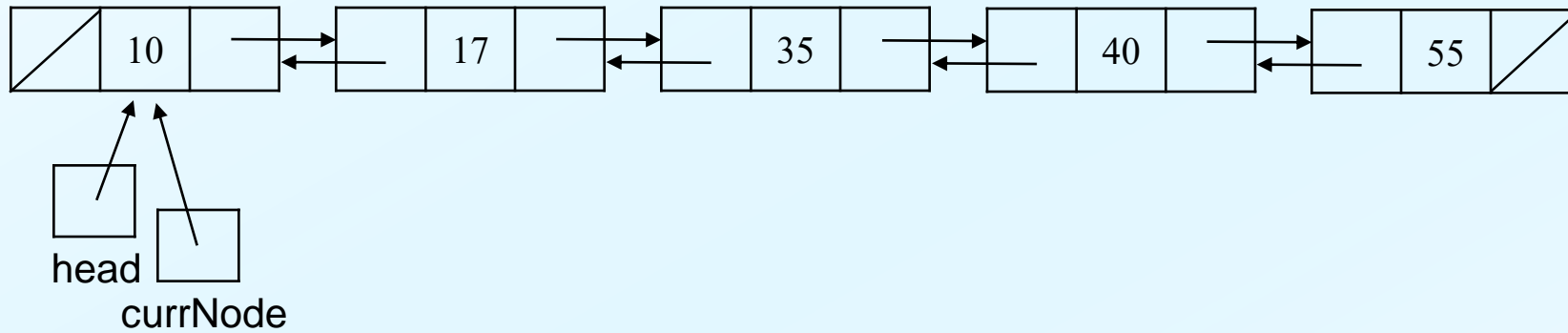
## 맨 뒤 삭제 확인 (앞과 동일 코드)

```
currNode.prev.next = currNode.next;  
if (prevNode.next != null) currNode.next.prev = currNode.prev;  
numItems--;
```



## 맨 앞 삭제 (앞과 다르다)

```
head = head.next;  
currNode.next.prev = null;  
numItems--;
```



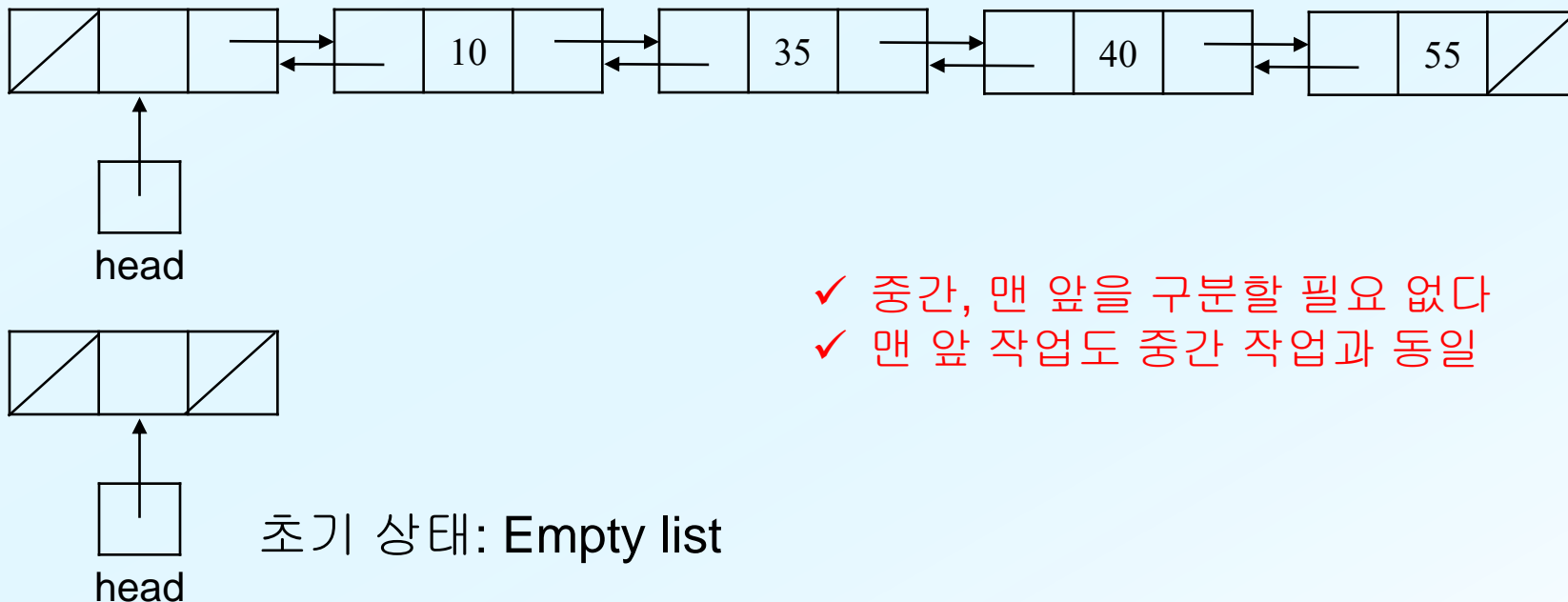
# Dummy Head를 가진 Doubly Linked List

삽입

```
BidirectionalNode<E> newNode
    = new BidirectionalNode<>(newItem, prevNode, prevNode.next);
if (prevNode.next != null) prevNode.next.prev = newNode;
prevNode.next = newNode;
numItems++;
```

삭제

```
currNode.prev.next = currNode.next;
if (prevNode.next != null) currNode.next.prev = currNode.prev;
numItems--;
```

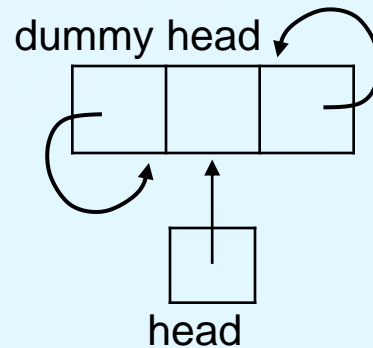
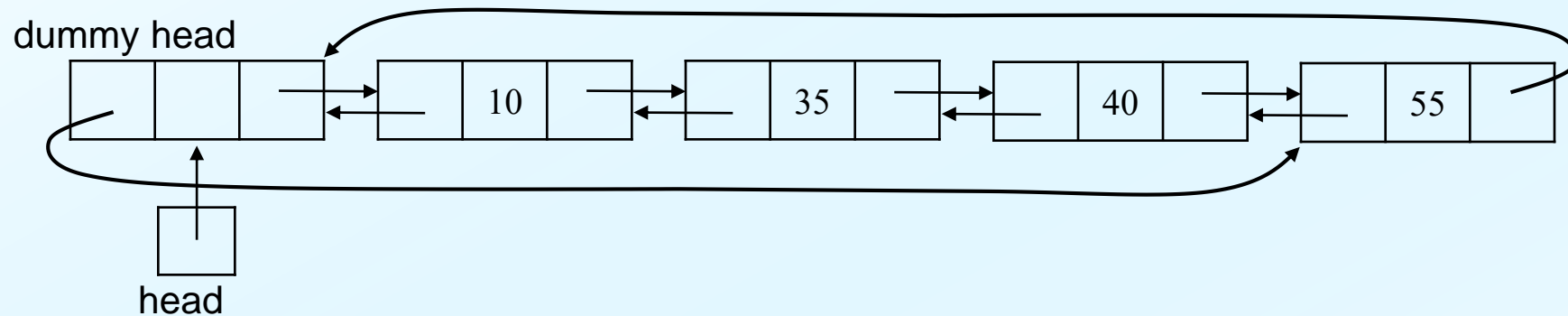


- ✓ 중간, 맨 앞을 구분할 필요 없다
- ✓ 맨 앞 작업도 중간 작업과 동일



# Circular Doubly Linked List w/ Dummy Head

지금까지 배운 모든 옵션이 다 포함된 Linked List



초기 상태: Empty list

# Insertion (삽입 위치에 무관)

```

BidirectionalNode<E> newNode
    = new BidirectionalNode<>(newItem, prevNode, prevNode.next);
if (prevNode.next != null) prevNode.next.prev = newNode;
prevNode.next = newNode;
numItems++;

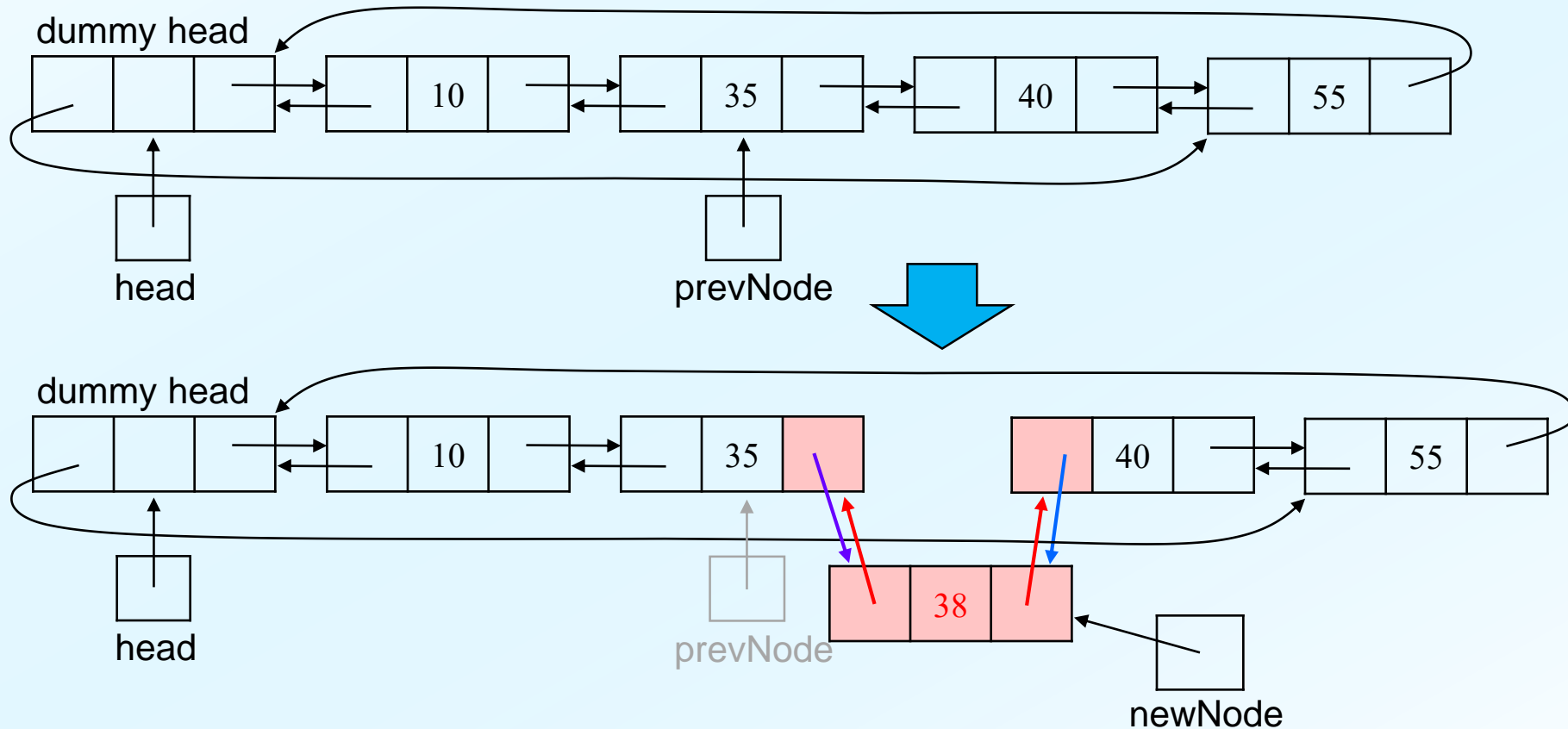
```

비교. P.88

```

BidirectionalNode<E> newNode
    = new BidirectionalNode<>(newItem, prevNode, prevNode.next);
prevNode.next.prev = newNode;
prevNode.next = newNode;
numItems++;

```

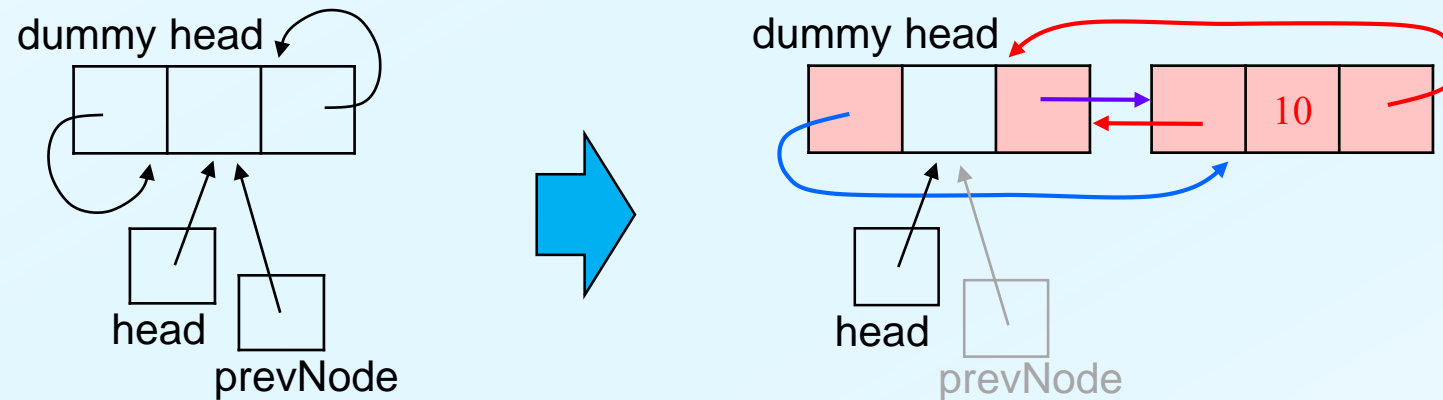


## 확인: 맨 앞 삽입도 이걸로 Okay

```

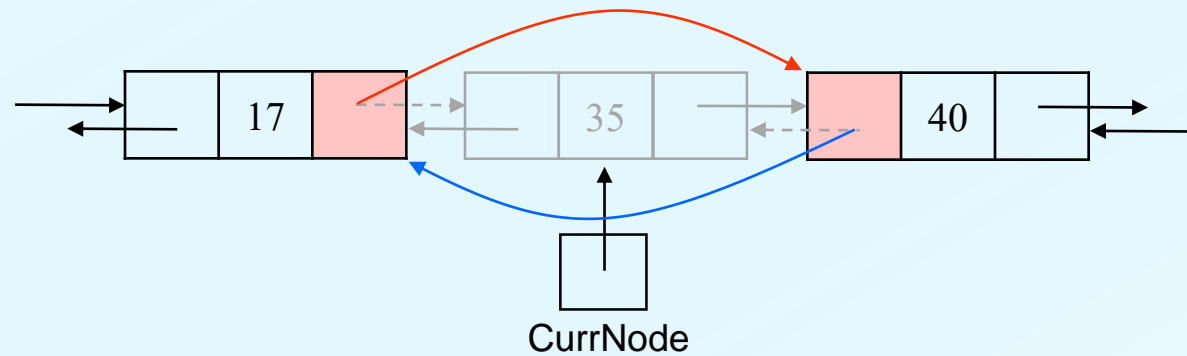
BidirectionalNode<E> newNode
    = new BidirectionalNode<>(newItem, prevNode, prevNode.next);
prevNode.next.prev = newNode;
prevNode.next = newNode;
numItems++;

```



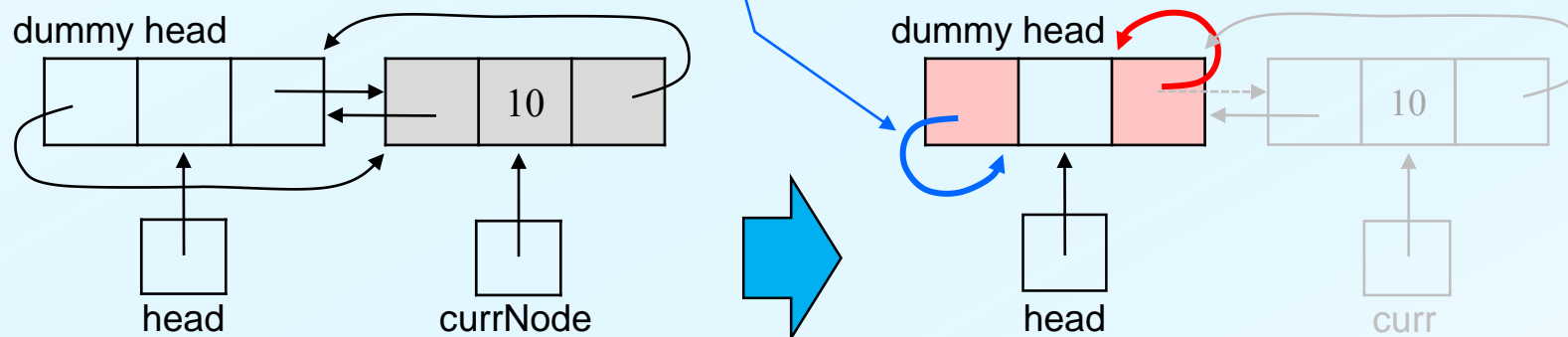
# Deletion (삭제 위치에 무관)

```
currNode.prev.next = currNode.next;  
currNode.next.prev = currNode.prev;  
numItems--;
```



확인: 맨 앞 삭제도 이걸로 Okay

```
currNode.prev.next = currNode.next;  
currNode.next.prev = currNode.prev;  
numItems--;
```



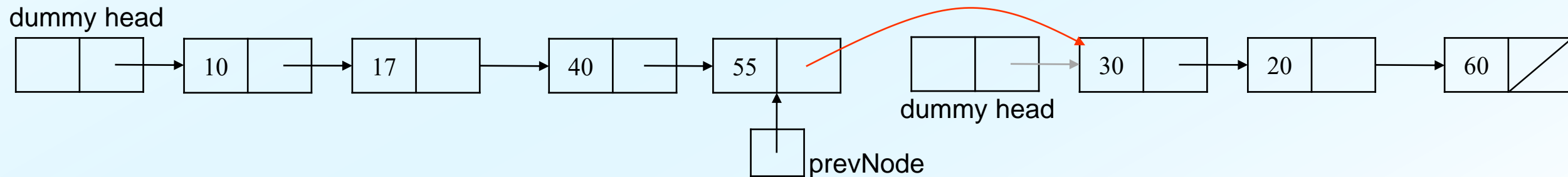
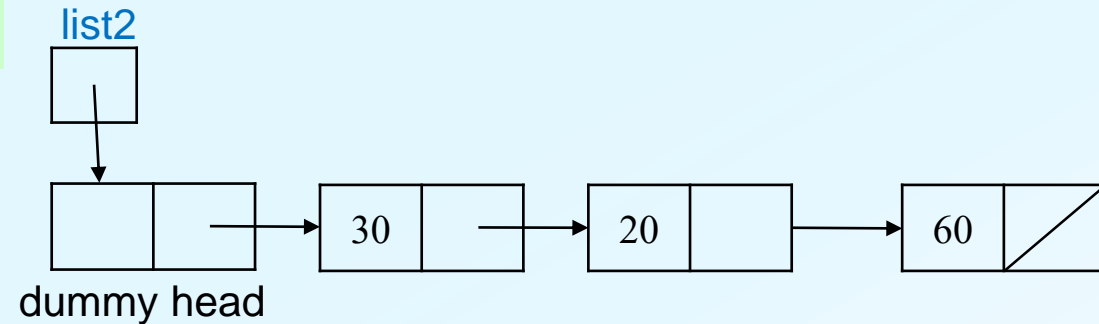
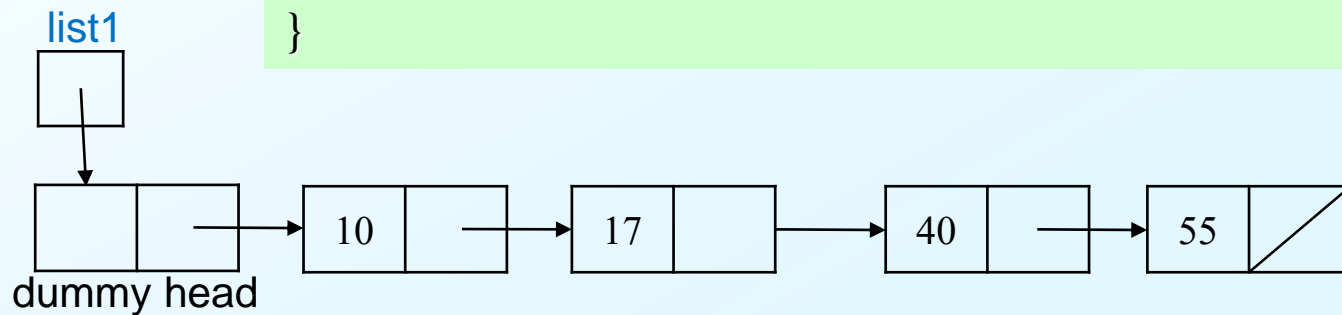
# 연결 리스트를 함수에 보내기

함수(메서드) 정의

```
public void appendList(LinkedList newList) {
    Node<E> prevNode = getNode(numItems-1);
    Node<E> startNode = newList.getNode(0);
    prevNode.next = startNode;
}
```

사용자 프로그램

```
...
list1.appendList(list2);
```



함수에서 리스트의 **reference**를 받는다.

Call by value지만 **reference** 값을 복사하므로 call by reference나 마찬가지.

dummy head

```
graph LR; Node1["5 | 9 | →"] --> Node2["7 | 9 | →"]; Node2 --> Node3["-4 | 7 | →"]; Node3 --> Node4["3 | 4 | →"]; Node4 --> Node5["-1 | 2 | →"]; Node5 --> Node6["15 | 0 | /"]; style Node1 fill:#fff,stroke:#000; style Node2 fill:#fff,stroke:#000; style Node3 fill:#fff,stroke:#000; style Node4 fill:#fff,stroke:#000; style Node5 fill:#fff,stroke:#000; style Node6 fill:#fff,stroke:#000;
```

#terms	degree	head	coeff	power	next
5	9	→	7	9	→
-4	7	→	3	4	→
-1	2	→	15	0	/