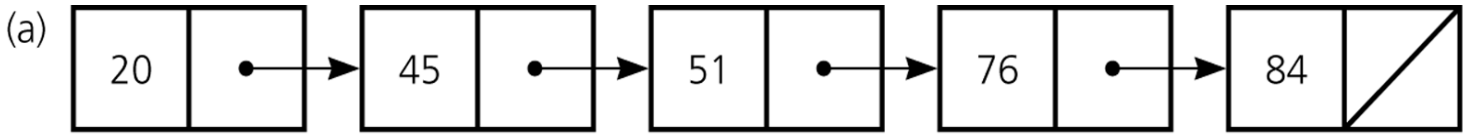
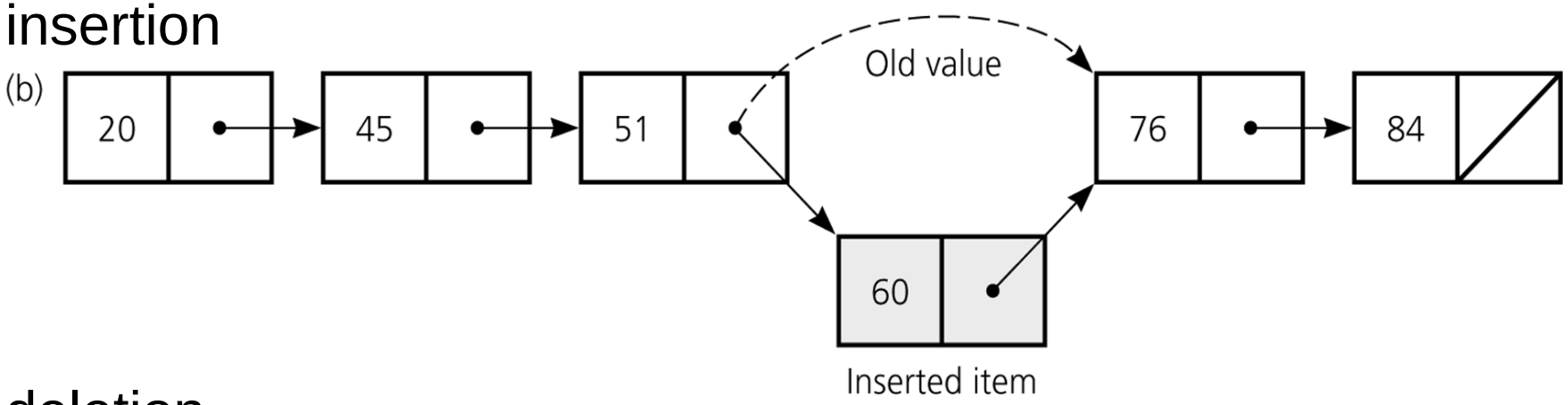
Ch. 4 Linked List

- When a list is implemented by an array
  - Consecutive space
  - Intuitively simple
  - Weak points
    - Overflow
    - Needs *shift* operation for insertion/deletion
- Linked list
  - (usually) no consecutive space
  - Free from shift overhead
  - No overflow
  - Overhead for linking

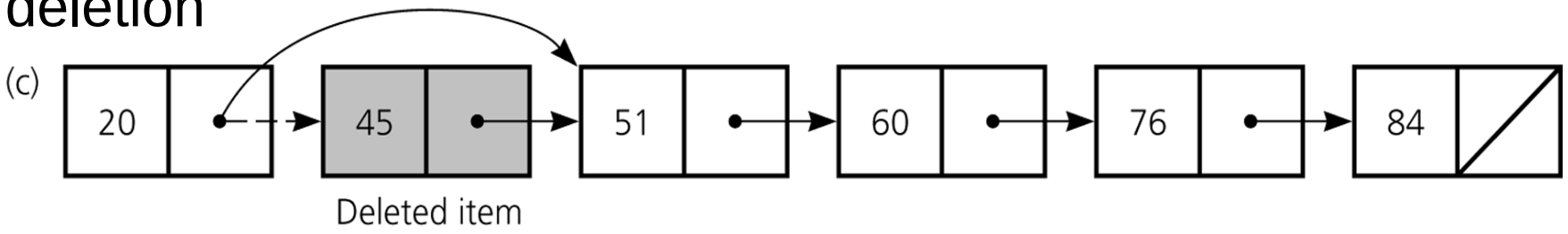
A linked list of integers



insertion

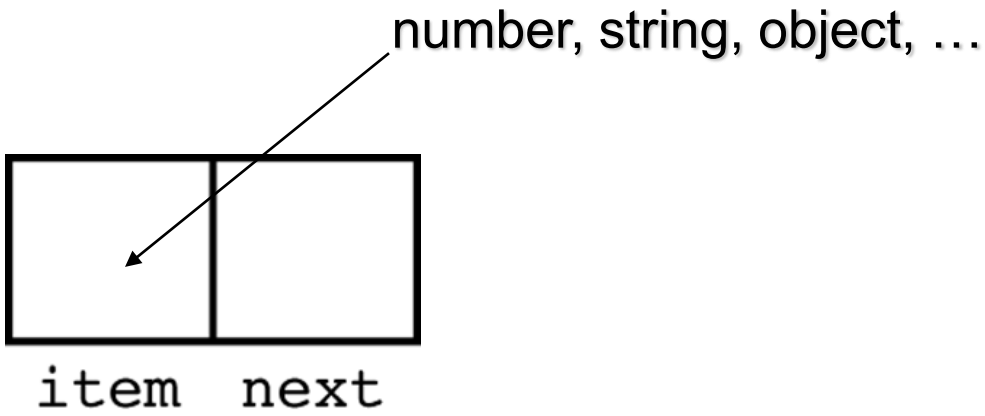


deletion



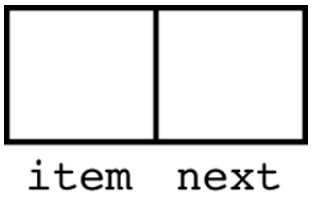
Linked List

- Each node contains
  - Data (item)
  - Link (to next item)



A naïve structure

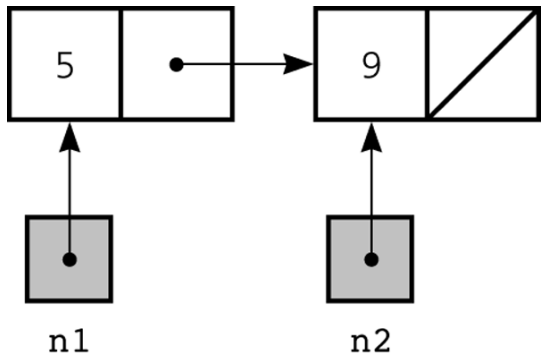
```
public class IntegerNode {
        public int item;
        public IntegerNode next;
}
```



✓ No good information hiding
✓ No good data abstraction

Example:

```
IntegerNode n1 = new IntegerNode( );
IntegerNode n2 = new IntegerNode( );
n1.item = 5;
n2.item = 9;
n2.next = null;
n1.next = n2;
```

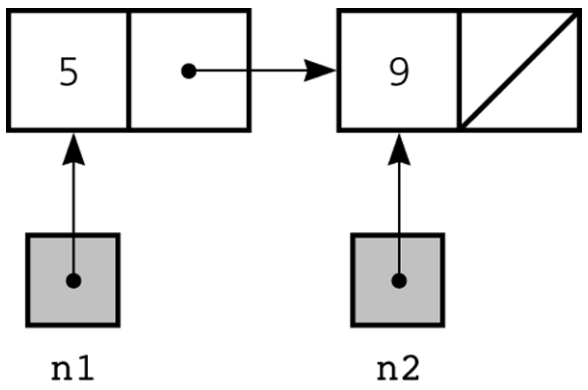


An Intermediate Version

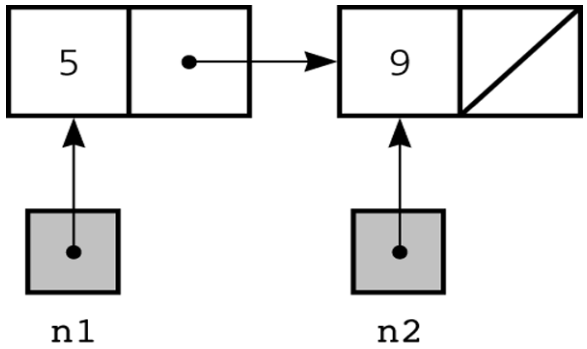
```
public class IntegerNode {
        private int item;
        private IntegerNode next;
        public void setItem(int newItem) {
                item = newItem;
        }
        public int getItem( ) {
                return item;
        }
        public void setNext(IntegerNode nextNode) {
                next = nextNode;
        }
        public IntegerNode getNext( ) {
                return next;
        }
}
```

Example:

```
IntegerNode n1 = new IntegerNode( );
IntegerNode n2 = new IntegerNode( );
n1.setItem(5);
n2.setItem(9);
n2.setNext(null);
n1.setNext(n2);
```



An Improved Version



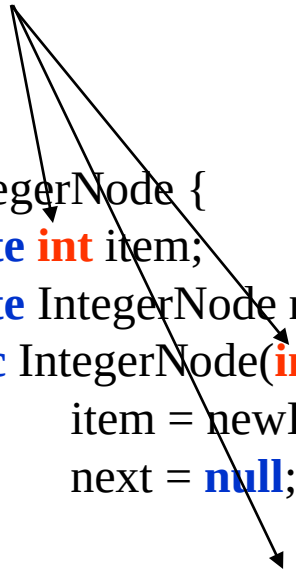

```
public class IntegerNode {
        private int item;
        private IntegerNode next;
        // constructors
        public IntegerNode(int newItem) {
                item = newItem;
                next = null;
        }
        public IntegerNode(int newItem, IntegerNode nextNode) {
                item = newItem;
                next = nextNode;
        }
        // setItem, getItem, setNext, getNext as before
        …
}
```

Example:

```
IntegerNode n2 = new IntegerNode(9);
IntegerNode n1 = new IntegerNode(5, n2);
```

Problems Still Remained

- ✓ restricted to a single integer field
- ✓ low reusability



```
public class IntegerNode {
        private int item;
        private IntegerNode next;
        public IntegerNode(int newItem) {
                item = newItem;
                next = null;
        }
        public IntegerNode(int newItem, IntegerNode nextNode) {
                item = newItem;
                next = nextNode;
        }
        …
}
```

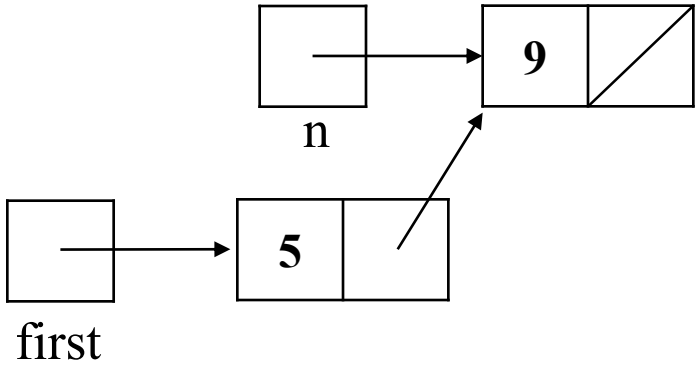
A Reusable Version

```
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;
        }
        public Object getItem( ) {
                return item;
        }
        // setItem, setNext, getNext similar
        …
}
```

✓ The Object class is a superclass of every class

Example:

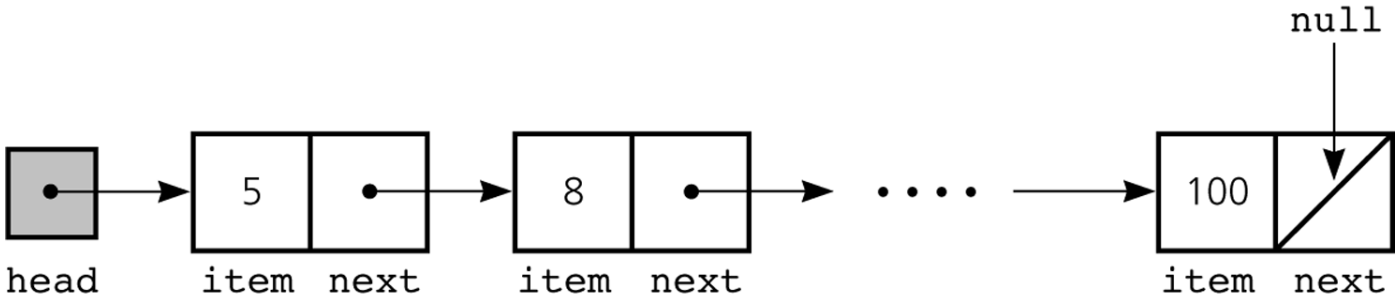
```
Node n = new Node(new Integer(9));
Node first = new Node(new Integer(5), n);
```



✓ Since **int** is a primitive type, it cannot be an inherited class of **Object**.
- **Integer** is a class in package *java.lang*.

Head Node

- Linked lists usually have a head reference




```
Node head = null;
```

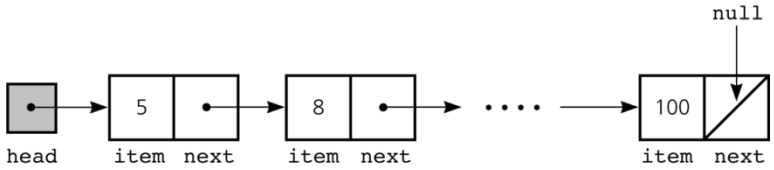
```
Node head = new Node(new Integer(5));
```

```
. . .
```

✓ Here, *head* is a simple reference variable

Displaying the Contents

- Sequential display of the contents of the linked list referenced by *head*



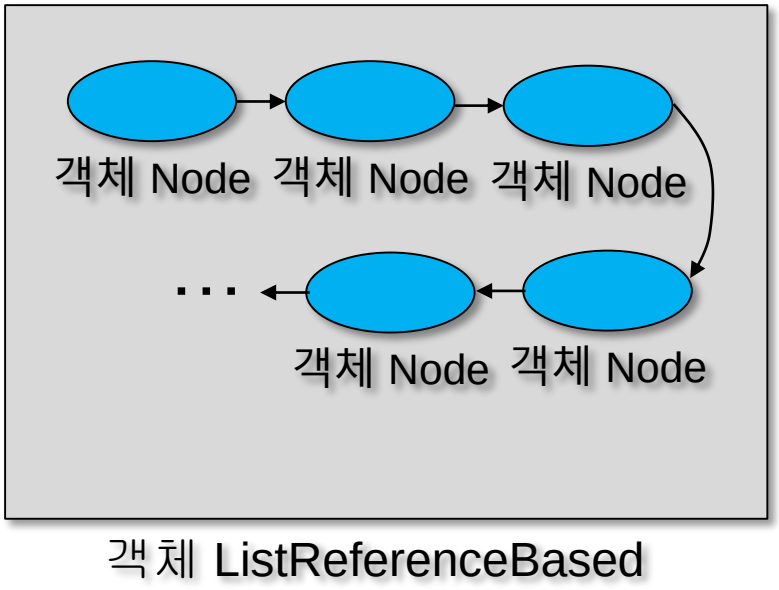

```
for (Node curr = head; curr != null; curr = curr.getNext( )) {
        System.out.println(curr.getItem( ));
}
```

```
5
8
.
.
.
100
```

Implementation of Linked List

```
Public class Node {
        See the slide ahead;
}
public interface ListInterface {
        // list operators
        public boolean isEmpty( );
        public int size( );
        public void add(int index, Object item);
        public void remove(int index);
        public Object get(int index);
        public void removeAll( );
}
```

```
Public class ListReferenceBased implements ListInterface {
        private Node head;
        private int numItems;
        // constructor
        public ListReferenceBased( ) {
                numItems = 0;
                head =  null;
        }
        // operations
        public boolean isEmpty( ) {
                return numItems == 0;
        }
        public int size( ) {
                return numItems;
        }
        …
```



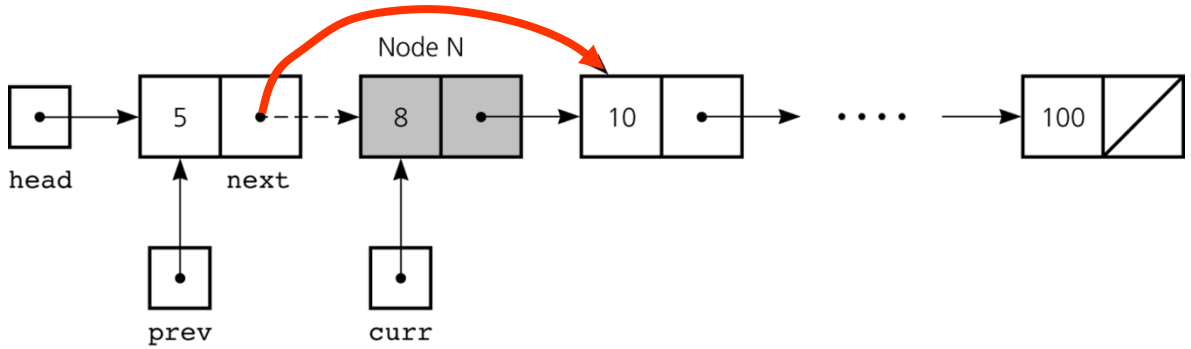
객체 ListReferenceBased

```
private Node find(int index) {
// return reference to i th node
        Node curr = head;  // 1st node
        for (int i = 1; i < index; i++) {
                curr = curr.getNext( );
        }
        return curr;
}
public Object get(int index) {
        if (index >= 1 && index <= numItems) {
                Node curr = find(index);
                return curr.getItem( );
        } else {
                Exception handling;
        }
}
…
```

Deleting a Specified Node

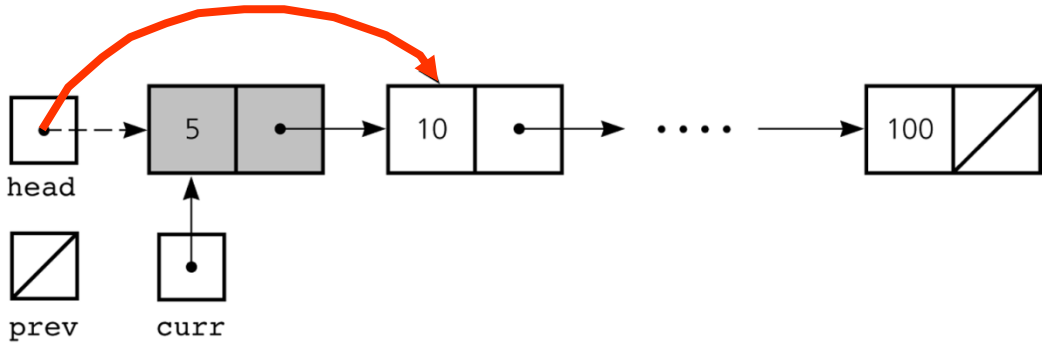
- *curr*가 가리키는 노드 삭제하기
- *curr* 의 바로 앞 노드는 *prev*가 가리킨다 가정

prev.setNext(curr.getNext( )); ← In C, prev->next = curr->next;



➢Removing the 1st node

head = curr.getNext( );



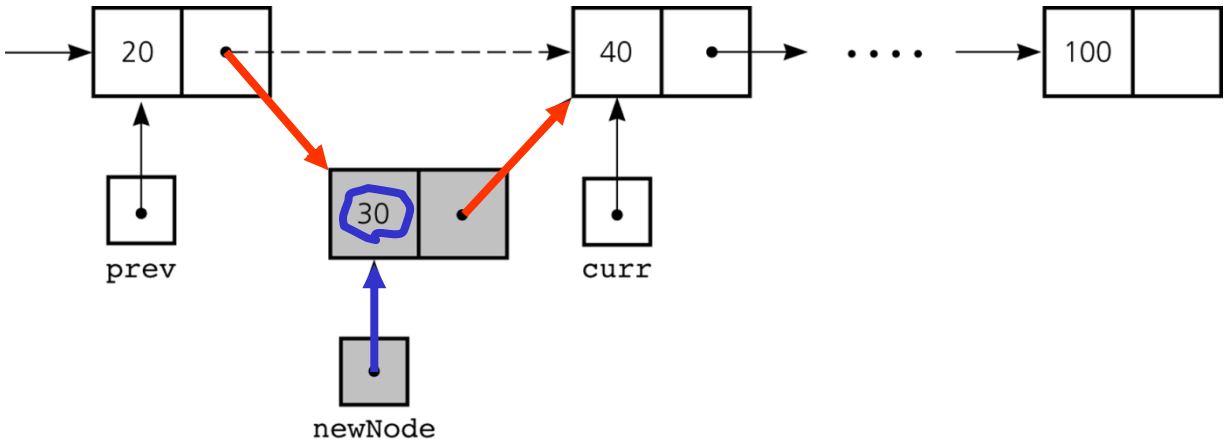
Inserting a Node

- *prev*와 *curr* 사이에 노드 삽입하기

```
newNode = new Node(new Integer(30));
newNode.setNext(curr);
prev.setNext(newNode);
```

In C,

```
newNode = malloc(sizeof Node);
newNode->item = 30;
newNode->next = curr;
prev.next = newNode;
```

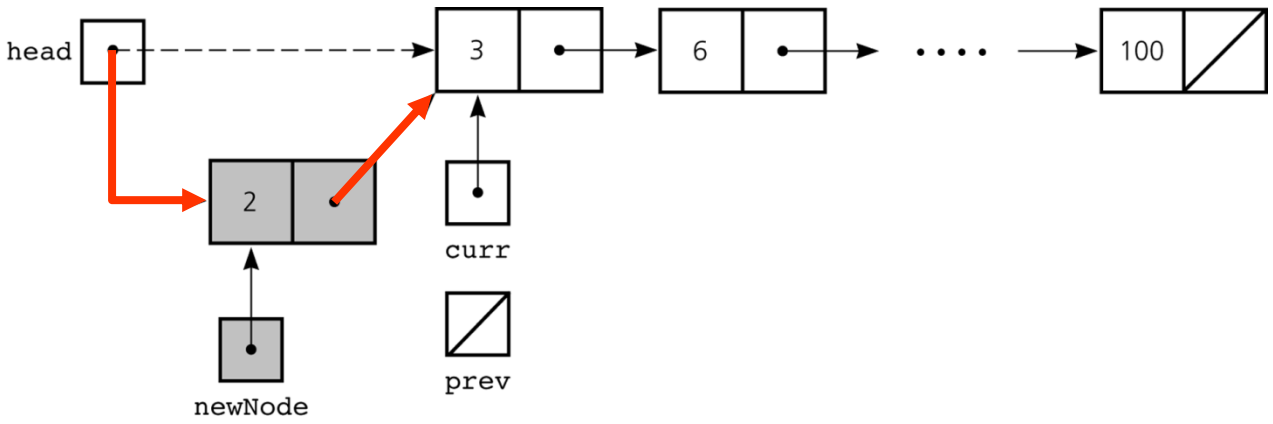


- 첫번째 노드 앞에 삽입하기

```
newNode = new Node(new Integer(2));
newNode.setNext(head);
head = newNode;
```

In C,

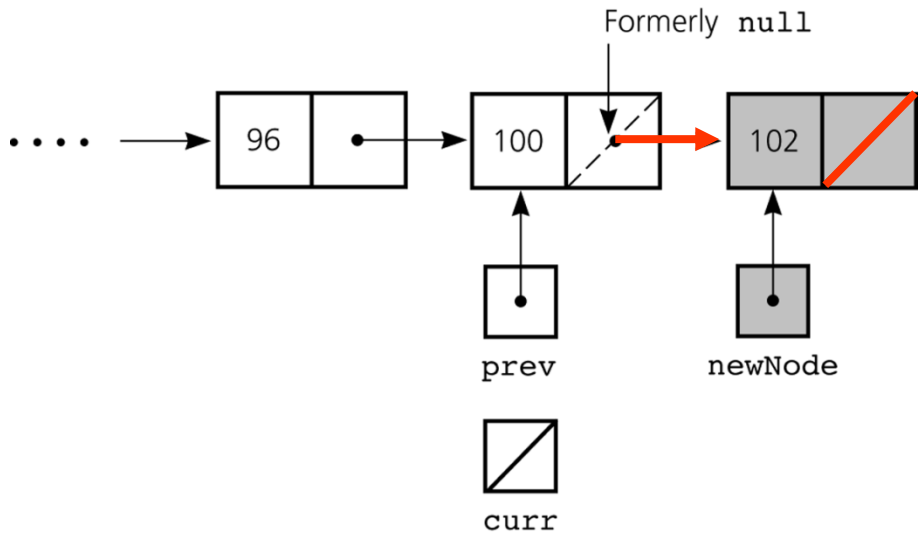
```
newNode = malloc(sizeof Node);
newNode->item = 2;
newNode->next = head;
head = newNode;
```



- 마지막 노드 다음에 삽입하기 (no special case)

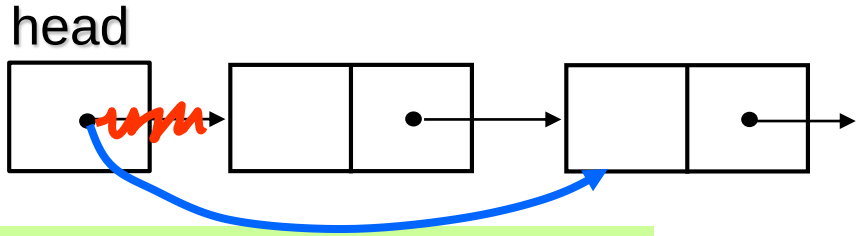
```
newNode = new Node(new Integer(102));
newNode.setNext(curr);
prev.setNext(newNode);
```

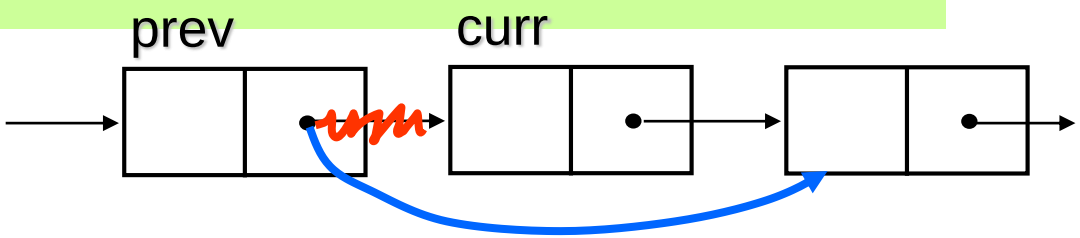
No need for special handling even if *curr* = null



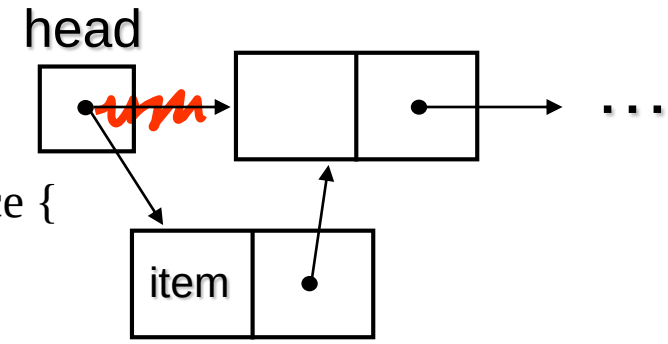
Implementation of Deletion

```
public class ListReferenceBased implements ListInterface {
        private Node head;
        private int numItems;
        …

        public void remove(int index) {
                if (index >= 1 && index <= numItems) {
                        if (index == 1) head = head.getNext( );
                        else {
                                Node prev = find(index – 1);
                                Node curr = prev.getNext( );
                                prev.setNext(curr.getNext( ));
                        }
                        numItems--;
                } else {Exception handling;}
        }
        …
}
```

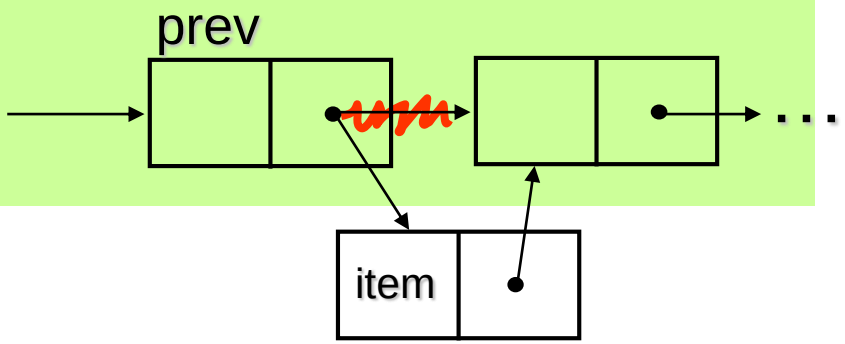




Implementation of Insertion

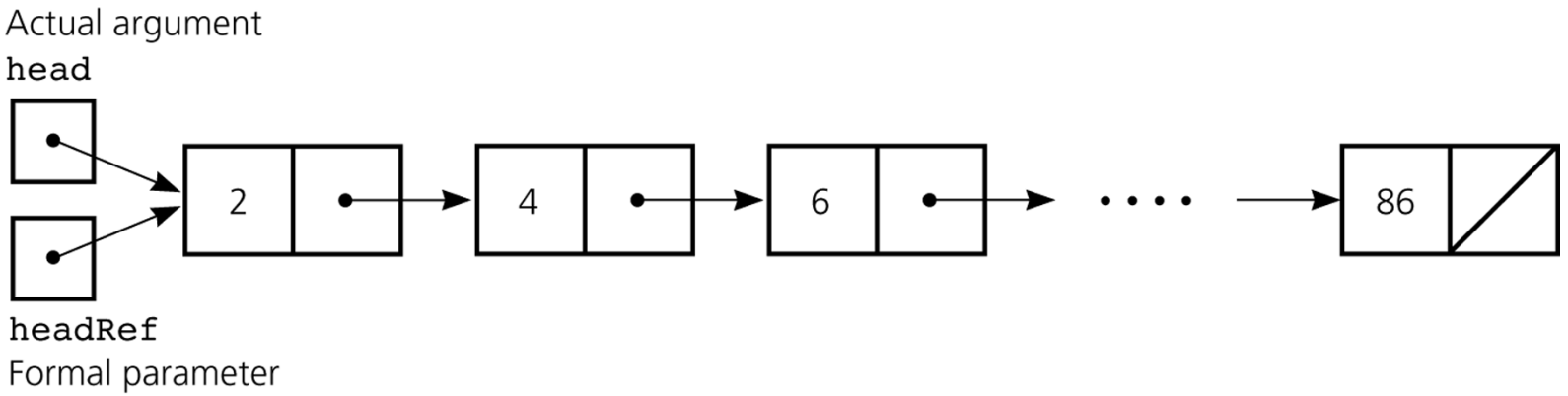



```
public class ListReferenceBased implements ListInterface {
        private Node head;
        private int numItems;
        …
        public void add(int index, Object item) {
                if (index >= 1 && index <= numItems+1) {
                        if (index == 1) {
                                Node newNode = new Node(item, head);
                                head = newNode;
                        } else {
                                Node prev = find(index – 1);
                                Node newNode = new Node(item, prev.getNext( ));
                                prev.setNext(newNode);
                        }
                        numItems++;
                } else {Exception handling;}
        }
        …
}
```



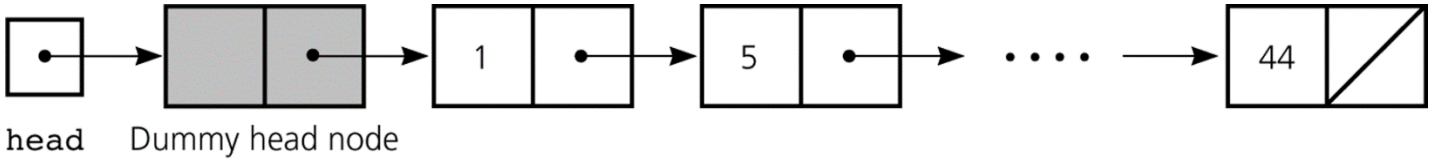
Passing a Linked List to a Method

```
printList(head);
```

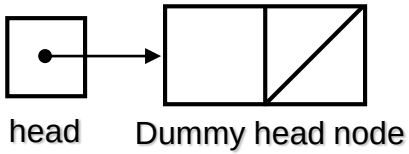


Dummy Head Node

- Put a dummy head node in front of the list
  - No need of special handling related to the 1st node



- Initialization/Constructor




```
public class ListReferenceBased implements ListInterface {
        private Node head;
        private int numItems;
        public ListReferenceBased( ) {
                numItems = 0;
                head = new Node( );
                head.setNext(null);
        }
        …
}
```

Cf. W/o Dummy Node

- Initialization/Constructor


```
public class ListReferenceBased implements ListInterface {
        private Node head;
        private int numItems;
        public ListReferenceBased( ) {
                numItems = 0;
                head = null;
        }
        …
}
```

Deletion When the Dummy Node Exists

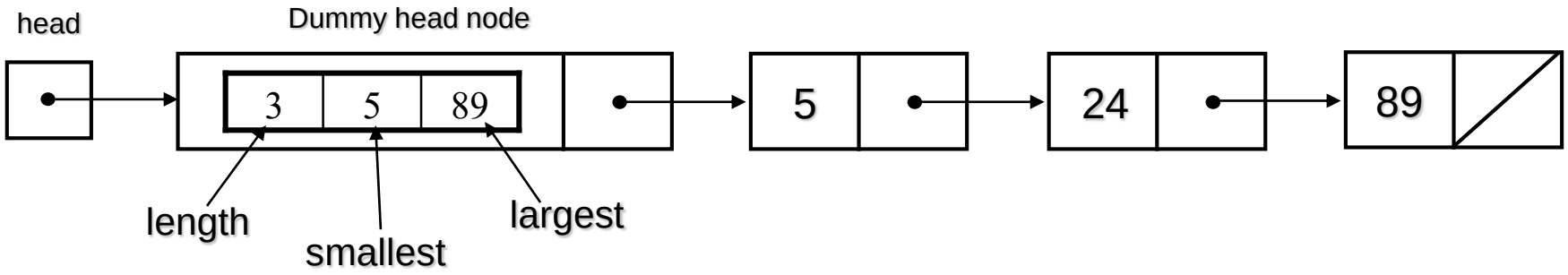
```
public class ListReferenceBased implements ListInterface {
        private Node head;
        private int numItems;
        …
        public void remove(int index) {
                if (index >= 1 && index <= numItems) {
                        Node prev = find(index – 1);
                        Node curr = prev.getNext( );
                        prev.setNext(curr.getNext( ));
                        numItems--;
                } else {Exception handling;}
        }
        …
}
```

Cf. W/o Dummy Node

```
public class ListReferenceBased implements ListInterface {
        private Node head;
        private int numItems;
        …
        public void remove(int index) {
                if (index >= 1 && index <= numItems) {
                        if (index == 1) head = head.getNext( );
                        else {
                                Node prev = find(index – 1);
                                Node curr = prev.getNext( );
                                prev.setNext(curr.getNext( ));
                        }
                        numItems--;
                } else {Exception handling;}
        }
        …
}
```

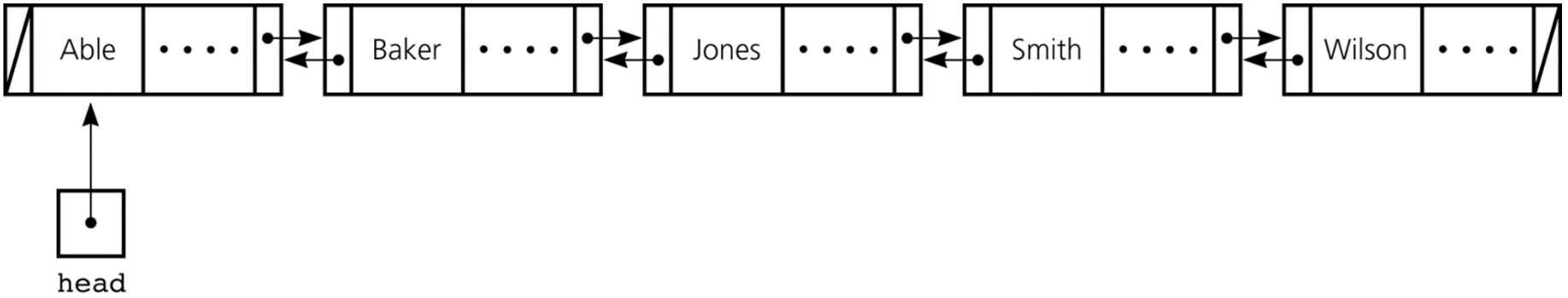
Utilization of the Dummy Node

- We can use the empty item field



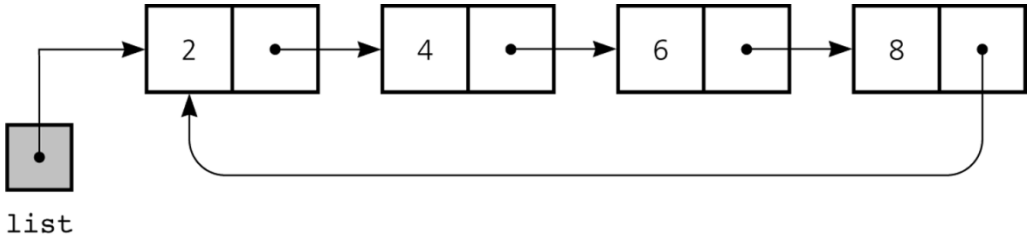

```
public class ListInfo {
        private int length;
        private Object smallestItem, largestItem;

        // methods for accessing the private data—length, smallestItem, largestItem—appear here
        …

} // end class
```

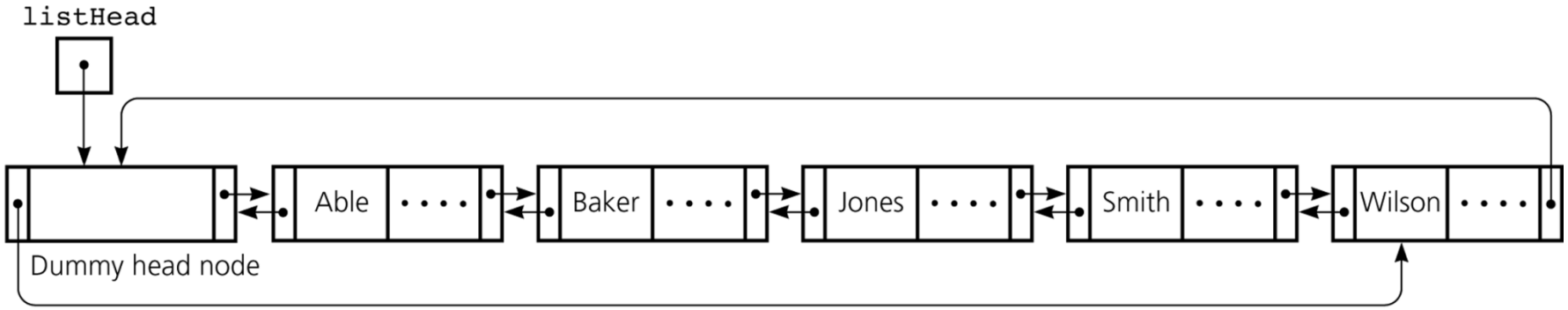
Doubly Linked List

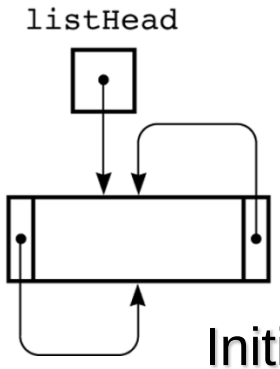


Circular Linked List

Singly linked list



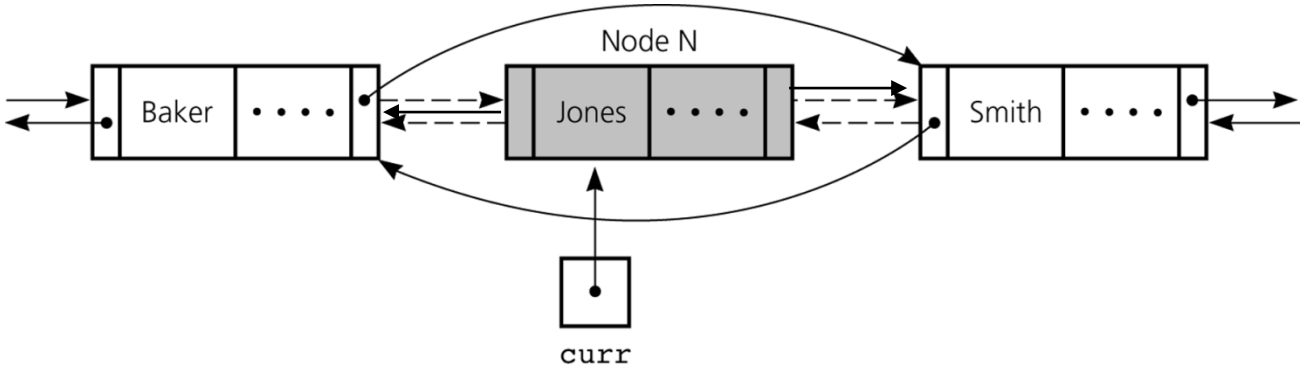




Circular doubly linked list

Initial state

Deletion in Doubly Linked List

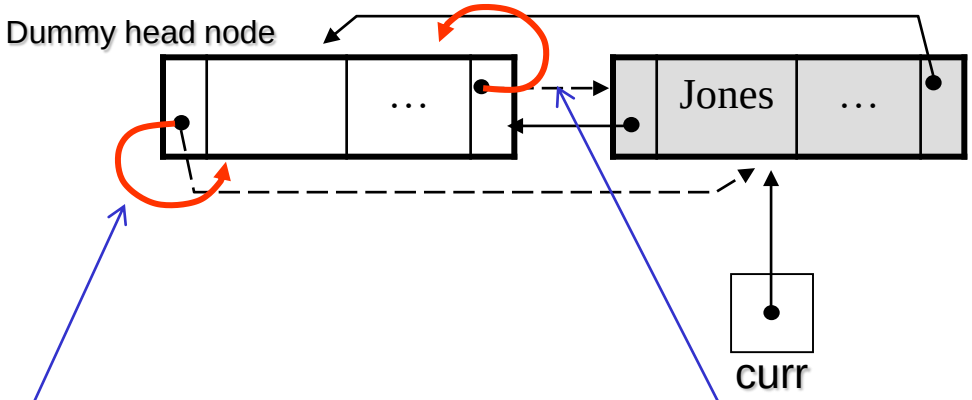


- No need to save the *prev* node
- When there exist nodes in both sides

```
curr.getPrecede( ).setNext(curr.getNext( ));
curr.getNext( ).setPrecede(curr.getPrecede( ));
```

- We still need a special treatment
  for the case that there is no node in at least one side

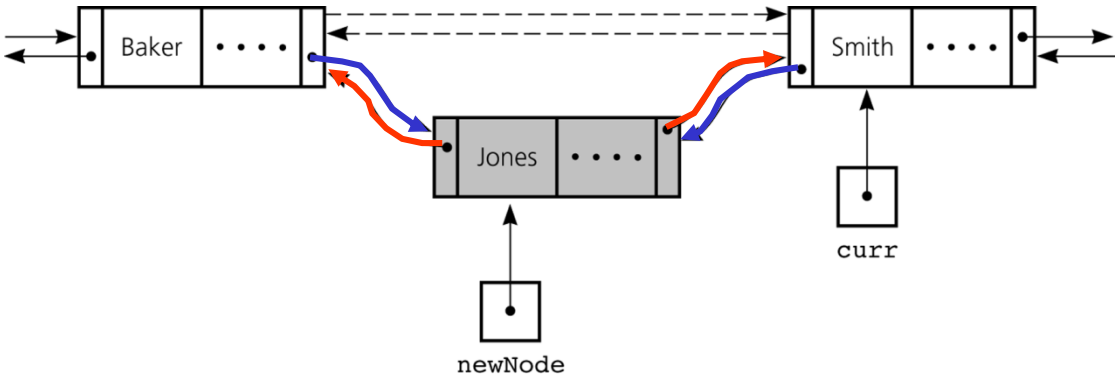
Deletion in Circular Doubly Linked List w/ Dummy Head Node



- No need of special treatment for the case of absence in any side

```
curr.getPrecede( ).setNext(curr.getNext( ));
curr.getNext( ).setPrecede(curr.getPrecede( ));
```

Insertion in Doubly Linked List



- When there exist nodes in both sides

```
newNode = new dListNode(…);
newNode.setNext(curr);
newNode.setPrecede(curr.getPrecede( ));
curr.getPrecede( ).setNext(newNode);
curr.setPrecede(newNode);
```

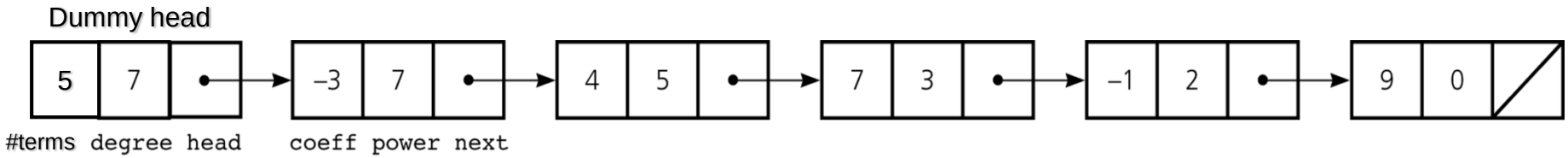
or

```
newNode = new dListNode(… curr.getPrecede( ), curr, …);
Curr.getPrecede( ).setNext(newNode);
Curr.setPrecede(newNode);
```

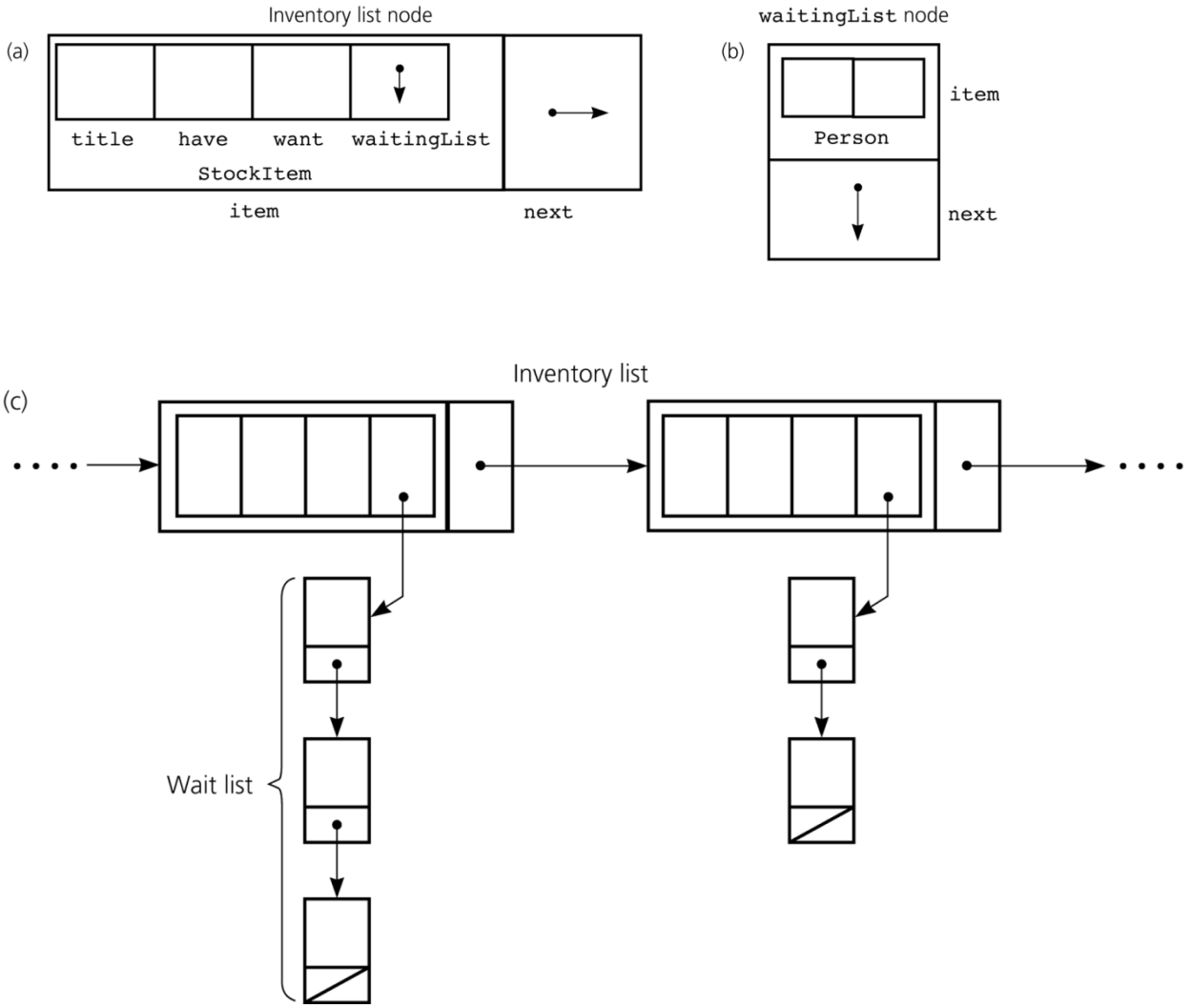
- Otherwise?

An Example: Sparse Polynomial

$-3x^7 + 4x^5 + 7x^3 - x^2 + 9$



A Mixed Structure



Call by Value

```
public void changeNumber(Integer n, Integer k) {
        n = k;
}
…
Integer x = new Integer(9);
```

// x → 9

// 하고싶은 일 x → 9 를 x → 5 로

```
changeNumber(x, new Integer(5)); // attempts to replace by Integer(5)
```

//실제로 일어나는 일