CIS 351-Data Structure-LinkedList
Feb 27, 2020

Dr. Farzana Rahman

Syracuse University

Goals

- To learn how to use the linked lists provided in the standard library
- To be able to use iterators to traverse linked lists
- To understand the implementation of linked lists
- To distinguish between abstract and concrete data types
- To know the efficiency of fundamental operations of lists and arrays
- To become familiar with the stack and queue types

Using Linked List

- A linked list consists of a number of nodes, each of which has a reference to the next node
- Adding and removing elements in the middle of a linked list is efficient
- Visiting the elements of a linked list in sequential order is efficient
- Random access is not efficient

Java's `LinkedList` class

- Generic class
  - *Specify type of elements in angle brackets:* `LinkedList<Product>`
- Package: `java.util`

Table 1 LinkedList Methods

`LinkedList<String> lst = new LinkedList<String>();`	An empty list.
`lst.addLast("Harry")`	Adds an element to the end of the list. Same as `add`.
`lst.addFirst("Sally")`	Adds an element to the beginning of the list. `lst` is now `[Sally, Harry]`.
`lst.getFirst()`	Gets the element stored at the beginning of the list; here `"Sally"`.
`lst.getLast()`	Gets the element stored at the end of the list; here `"Harry"`.
`String removed = lst.removeFirst();`	Removes the first element of the list and returns it. `removed` is `"Sally"` and `lst` is `[Harry]`. Use `removeLast` to remove the last element.
`ListIterator<String> iter = lst.listIterator()`	Provides an iterator for visiting all list elements (see Table 2 on page 634).

List Iterator

- `ListIterator` type
- Gives access to elements inside a linked list
- Encapsulates a position anywhere inside the linked list
- Protects the linked list while giving access

A List Iterator

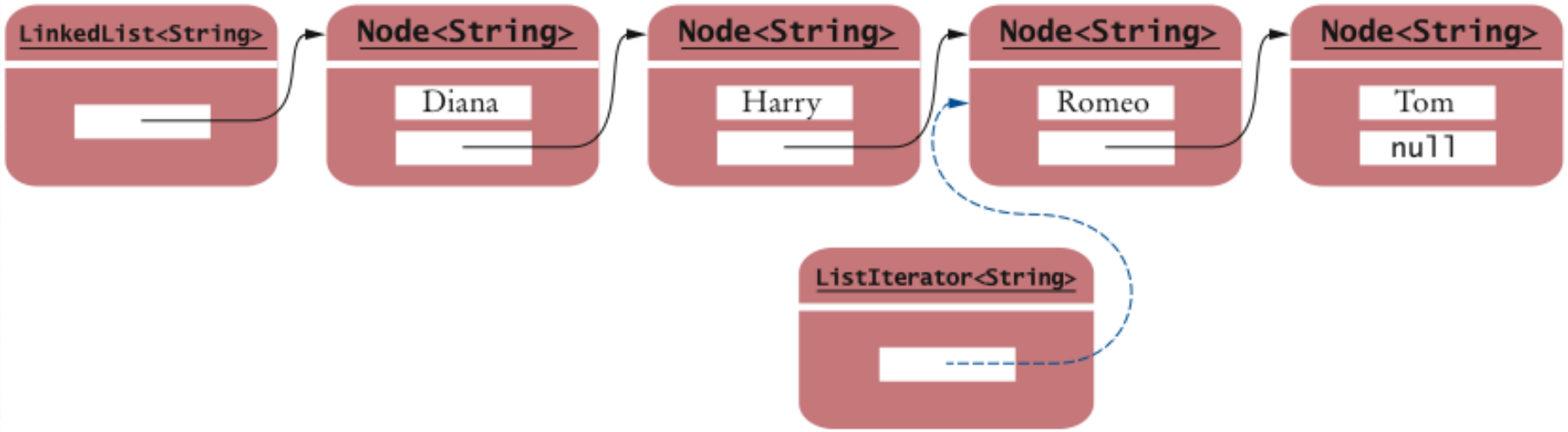


Figure 2 A List Iterator

A Conceptual View of the List Iterator

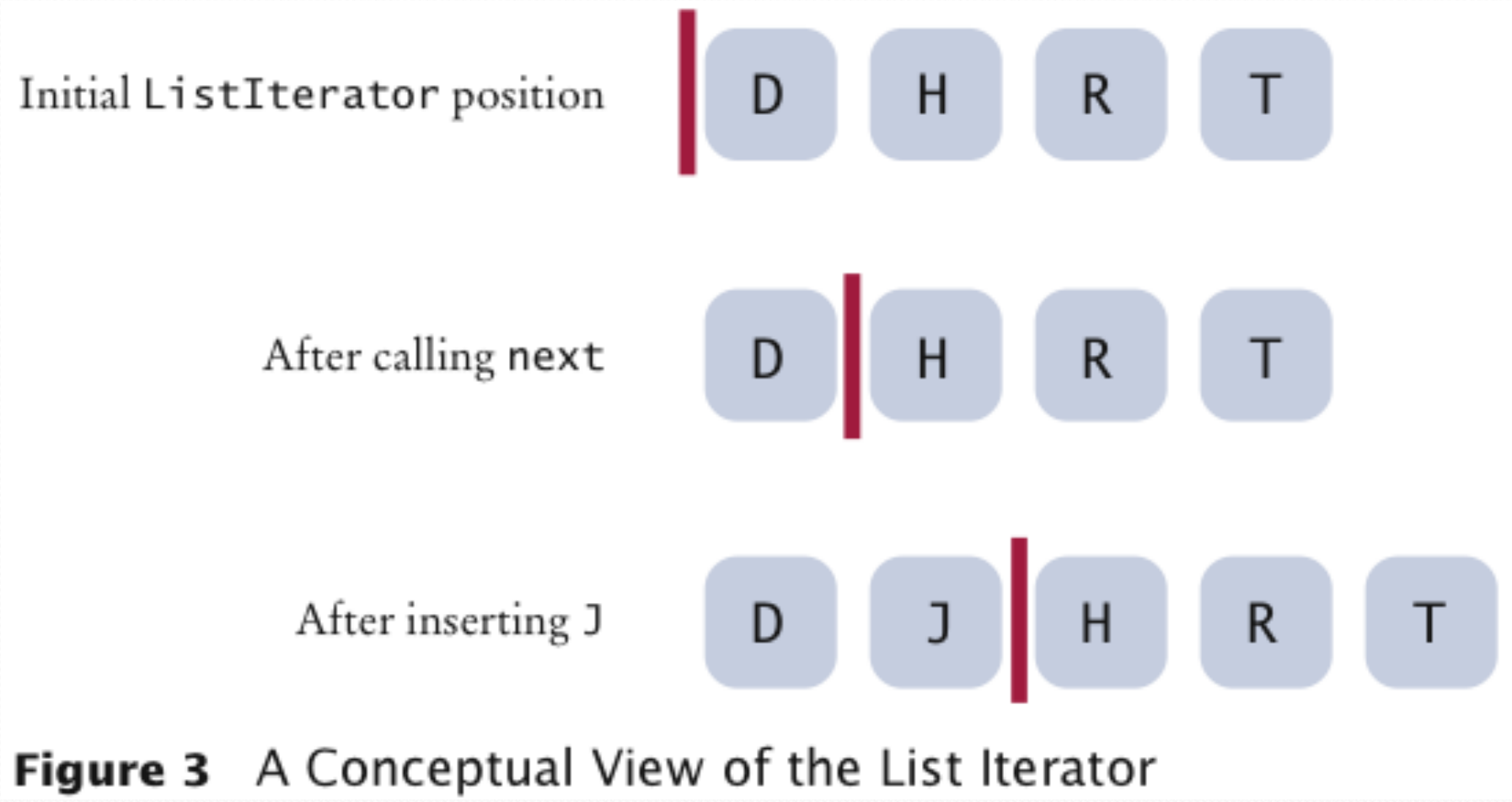


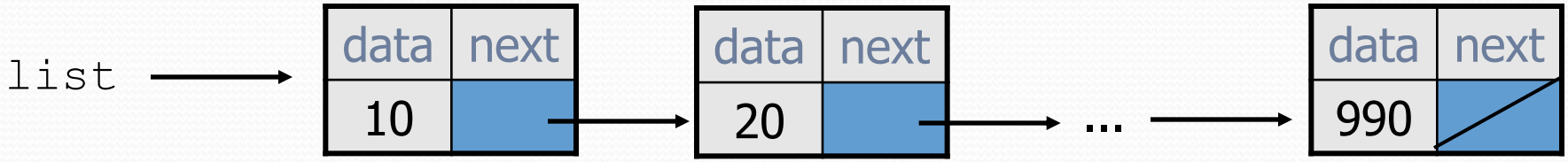
Figure 3 A Conceptual View of the List Iterator

The Java Collection's Framework `List` Interface

- JCF provides an interface `java.util.List`
- List interface supports an ordered collection - Also known as a sequence
  - `boolean add(E o)`
  - `void add(int index, E element)`
  - `void clear()`
  - `boolean contains(Object o)`
  - `boolean equals(Object o)`
  - `E get(int index)`
  - `int indexOf(Object o)`
  - `boolean isEmpty()`
  - `Iterator<E> iterator()`
  - `ListIterator<E> listIterator()`
  - `ListIterator<E> listIterator(int index)`
  - `E remove(int index)`
  - `boolean remove(Object o)`
  - `E set(int index, E element)`
  - `int size()`
  - `List<E> subList(int fromIndex, int toIndex)`
  - `Object[] toArray()`

Linked node question

- Suppose we have a long chain of list nodes:



  - We don't know exactly how long the chain is.

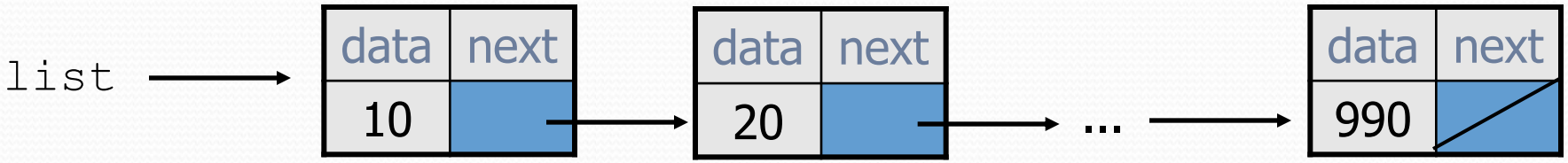
- How would we print the data values in all the nodes?

Algorithm pseudocode

- Start at the **front** of the list.
- While (there are more nodes to print):
  - Print the current node's **data**.
  - Go to the **next** node.

- How do we walk through the nodes of the list?

```
list = list.next;    // is this a good idea?
```

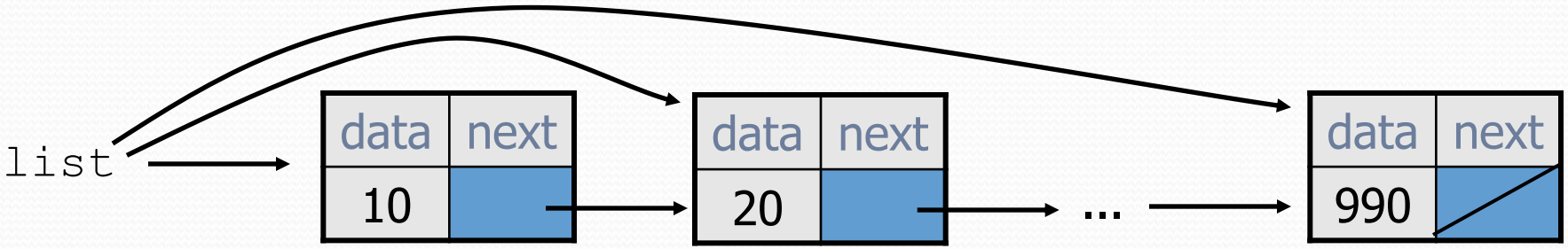


Traversing a list?

- One (bad) way to print every value in the list:

```
while (list != null) {
    System.out.println(list.data);
    list = list.next;     // move to next node
}
```

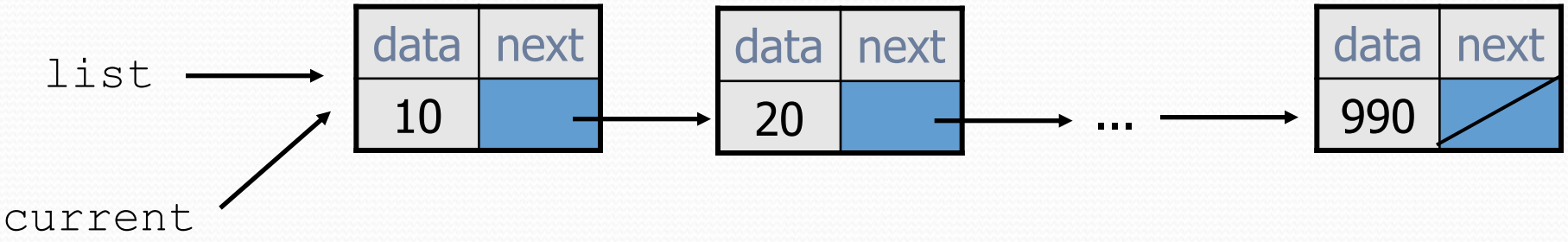
  - What's wrong with this approach?
    - (It loses the linked list as it prints it!)



A `current` reference

- Don't change `list`. Make another variable, and change it.
  - A `ListNode` variable is NOT a `ListNode` object

```
ListNode current = list;
```



- What happens to the picture above when we write:

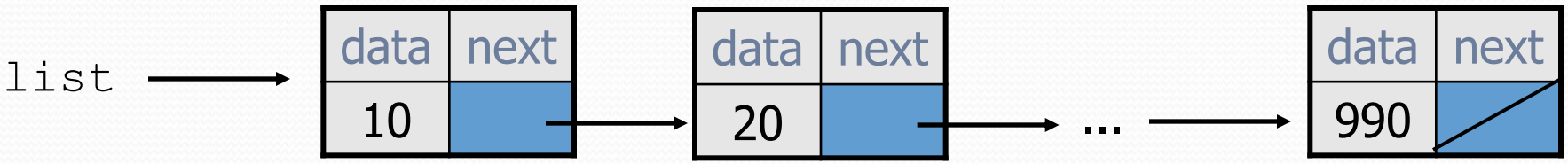
```
current = current.next;
```

Traversing a list correctly

- The correct way to print every value in the list:

```
ListNode current = list;
while (current != null) {
    System.out.println(current.data);
    current = current.next;  // move to next node
}
```

  - Changing `current` does not damage the list.



Linked List vs. Array

- Print list values:

```
ListNode list= ...;

ListNode current = list;
while (current != null) {
    System.out.println(current.data);
    current = current.next;
}
```

- Similar to array code:

```
int[] a = ...;

int i = 0;
while (i < a.length) {
    System.out.println(a[i]);
    i++;
}
```

Description	Array Code	Linked List Code
Go to front of list	`int i = 0;`	`ListNode current = list;`
Test for more elements	`i < size`	`current != null`
Current value	`elementData[i]`	`current.data`
Go to next element	`i++;`	`current = current.next;`

Abstract data types (ADTs)

- **abstract data type (ADT)**: A specification of a collection of data and the operations that can be performed on it.
  - Describes *what* a collection does, not *how* it does it

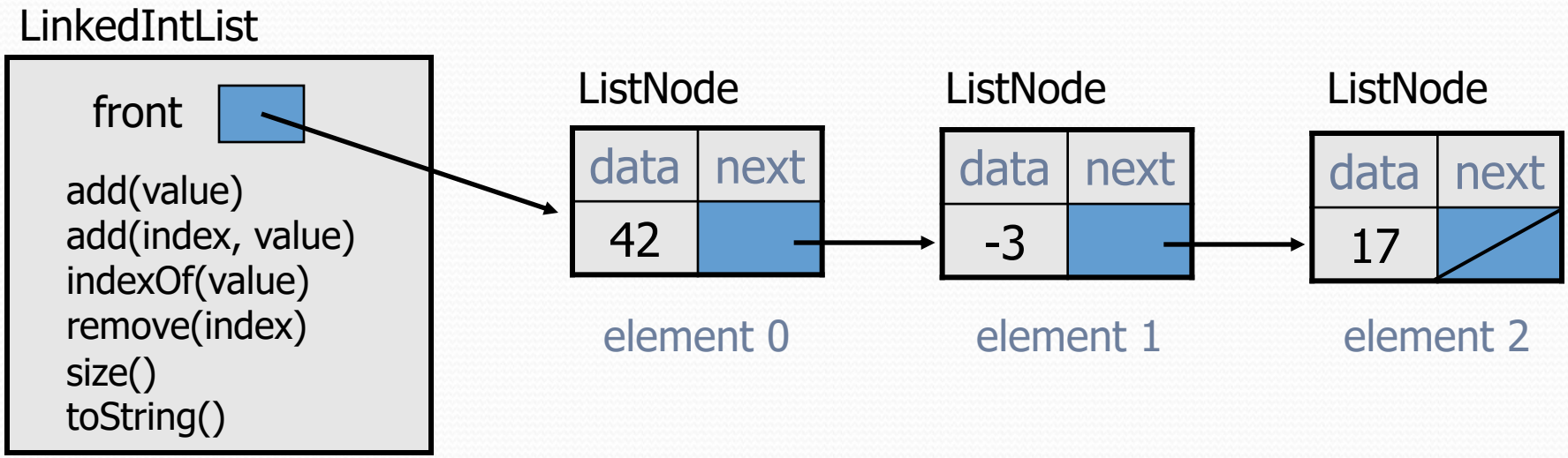
- Java's collection framework describes several ADTs:
  - `Queue`, `List`, `Collection`, `Deque`, `List`, `Map`, `Set`

- An ADT can be implemented in multiple ways:
  - `ArrayList` and `LinkedList` implement `List`
  - `HashSet` and `TreeSet` implement `Set`
  - `LinkedList`, `ArrayDeque`, etc. implement `Queue`

- The **same** external behavior can be implemented in many different ways, each with pros and cons.

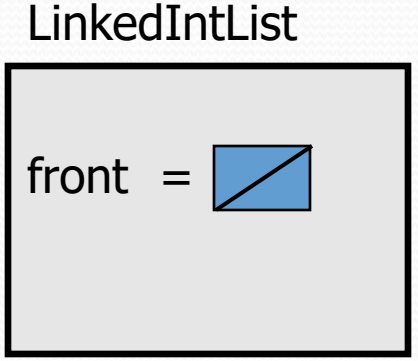
A `LinkedIntList` class

- Let's write a collection class named `LinkedIntList`.
  - Has the same methods as `ArrayIntList`:
    - `add`, `add`, `get`, `indexOf`, `remove`, `size`, `toString`
  - The list is internally implemented as a chain of linked nodes
    - The `LinkedIntList` keeps a reference to its `front` as a field
    - `null` is the end of the list; a `null` front signifies an empty list



`LinkedIntList` class v1

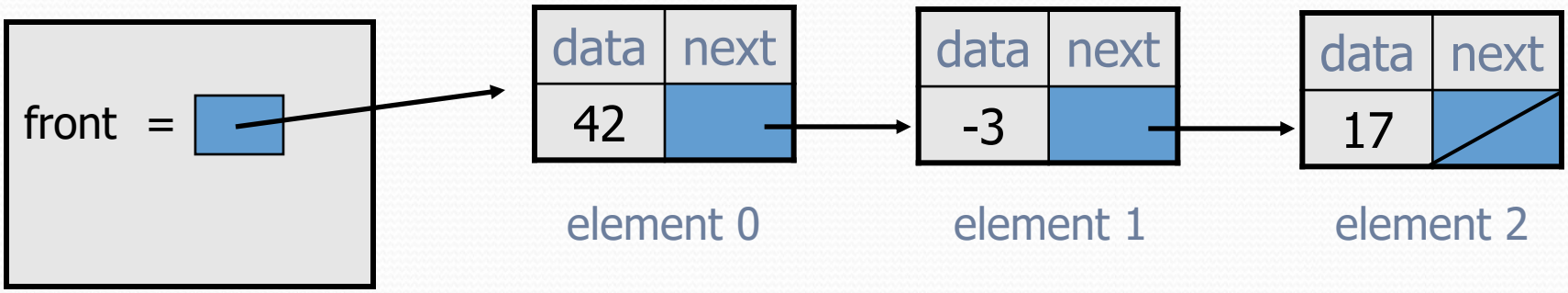
```
public class LinkedIntList {
    private ListNode front;

    public LinkedIntList() {
        front = null;
    }


    methods go here

}
```



Implementing add

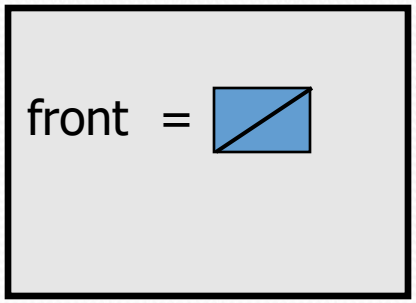
```
// Adds the given value to the end of the list.
public void add(int value) {
    …
}
```

- How do we add a new node to the end of a list?
- Does it matter what the list's contents are before the add?

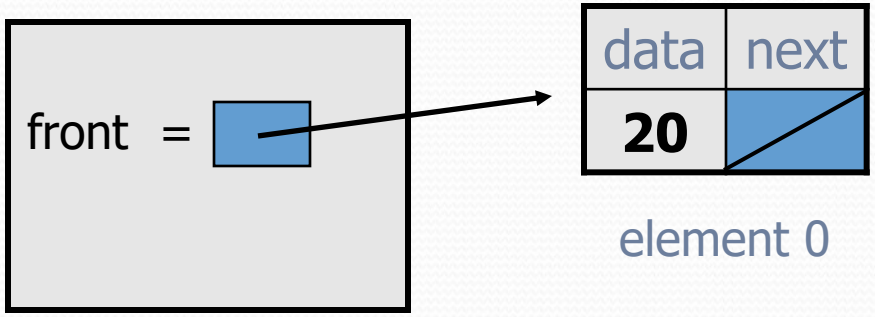


Adding to an empty list

- Before adding 20:



After:



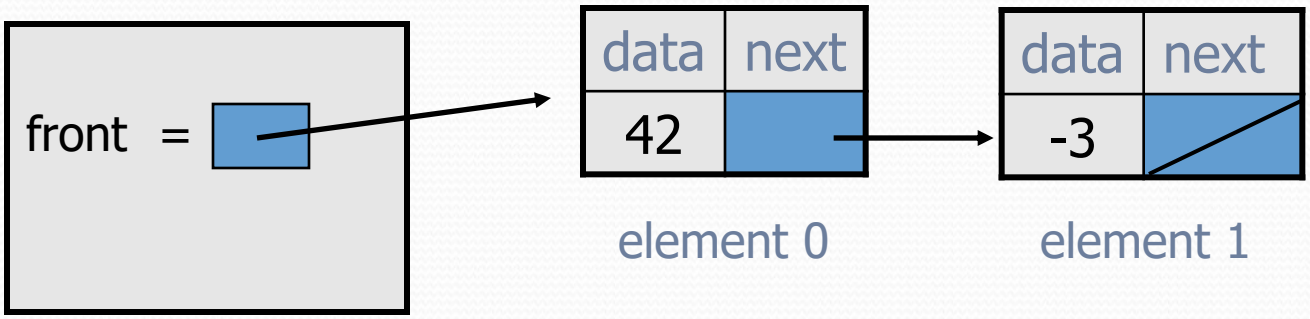
  - We must create a new node and attach it to the list.

The `add` method, 1st try

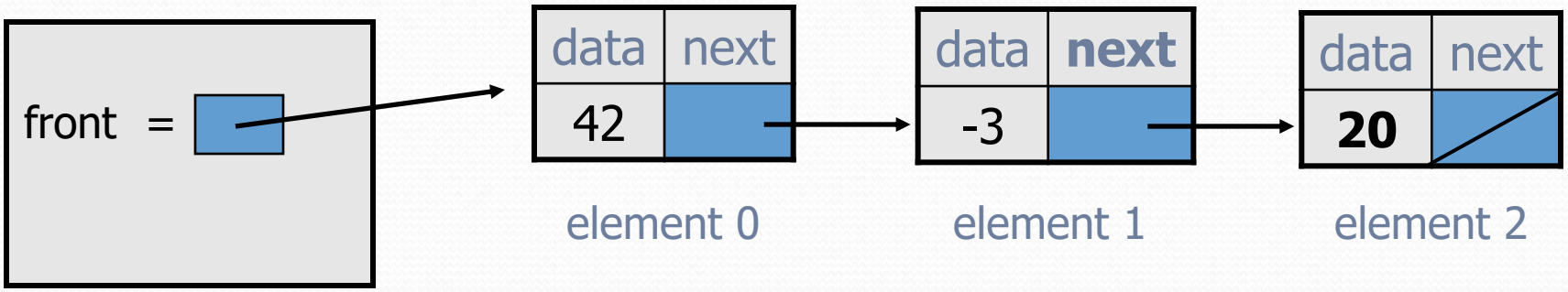
```
// Adds the given value to the end of the list.
public void add(int value) {
    if (front == null) {
        // adding to an empty list
        front = new ListNode(value);
    } else {
        // adding to the end of an existing list

        ...

    }
}
```

Adding to non-empty list

- Before adding value 20 to end of list:

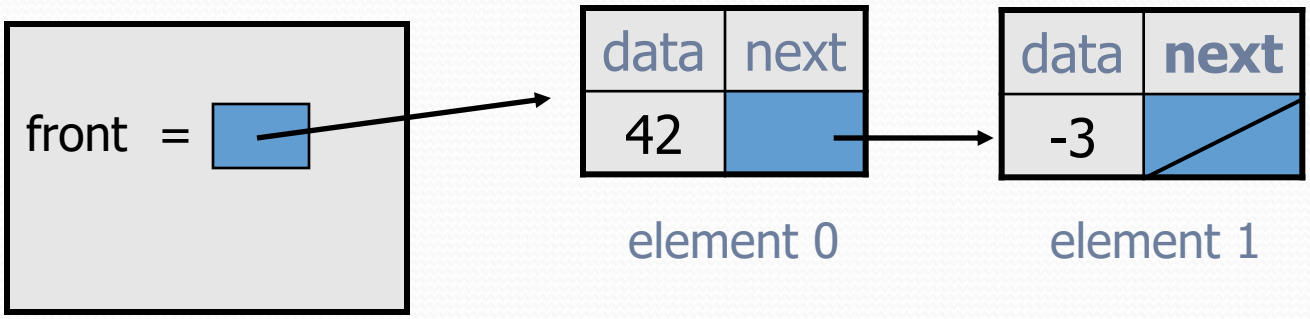


- After:



Don't fall off the edge!

- To add/remove from a list, you must modify the `next` reference of the node *before* the place you want to change.



  - Where should `current` be pointing, to add 20 at the end?
  - What loop test will stop us at this place in the list?

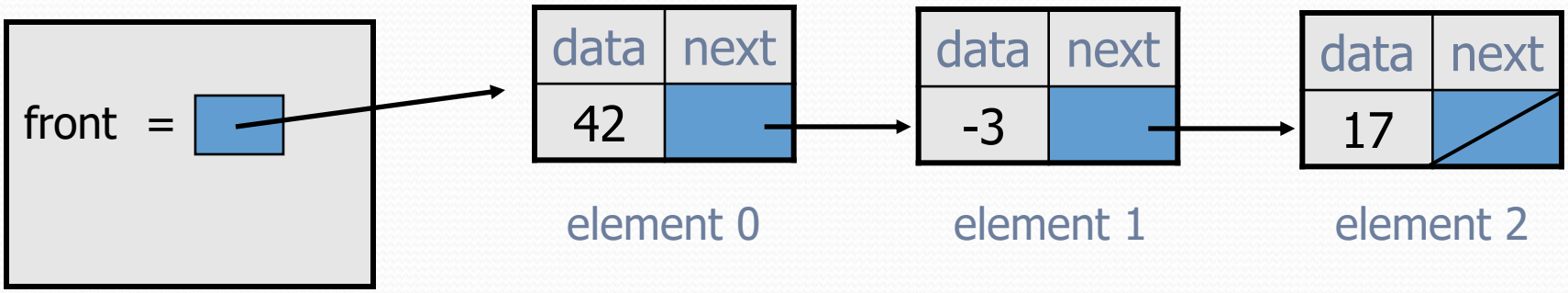
The `add` method

```
// Adds the given value to the end of the list.
public void add(int value) {
    if (front == null) {
        // adding to an empty list
        front = new ListNode(value);
    } else {
        // adding to the end of an existing list
        ListNode current = front;
        while (current.next != null) {
            current = current.next;
        }
        current.next = new ListNode(value);
    }
}
```

Implementing `get`

```
// Returns value in list at given index.
public int get(int index) {
    …
}
```

- Exercise: Implement the `get` method.



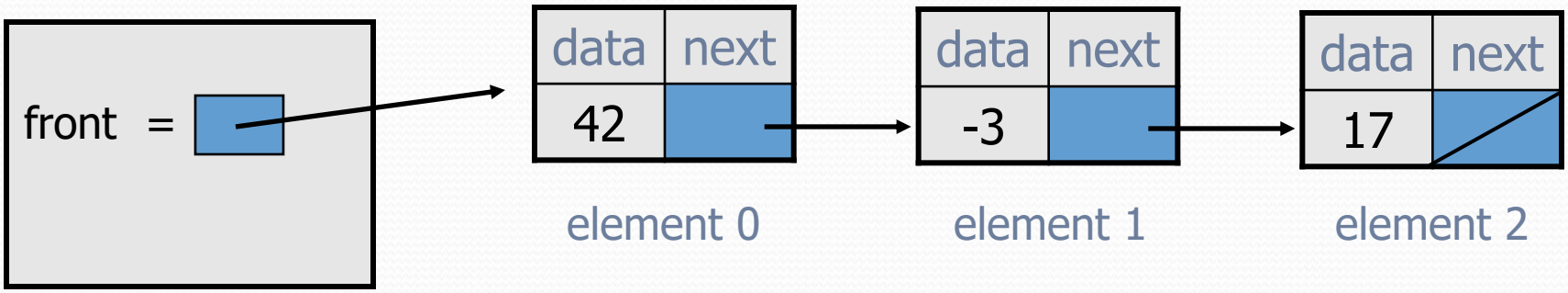
The `get` method

```
// Returns value in list at given index.
// Precondition: 0 <= index < size()
public int get(int index) {
    ListNode current = front;
    for (int i = 0; i < index; i++) {
        current = current.next;
    }
    return current.data;
}
```

Implementing `add` (2)

```
// Inserts the given value at the given index.
public void add(int index, int value) {
    …
}
```

- Exercise: Implement the two-parameter `add` method.

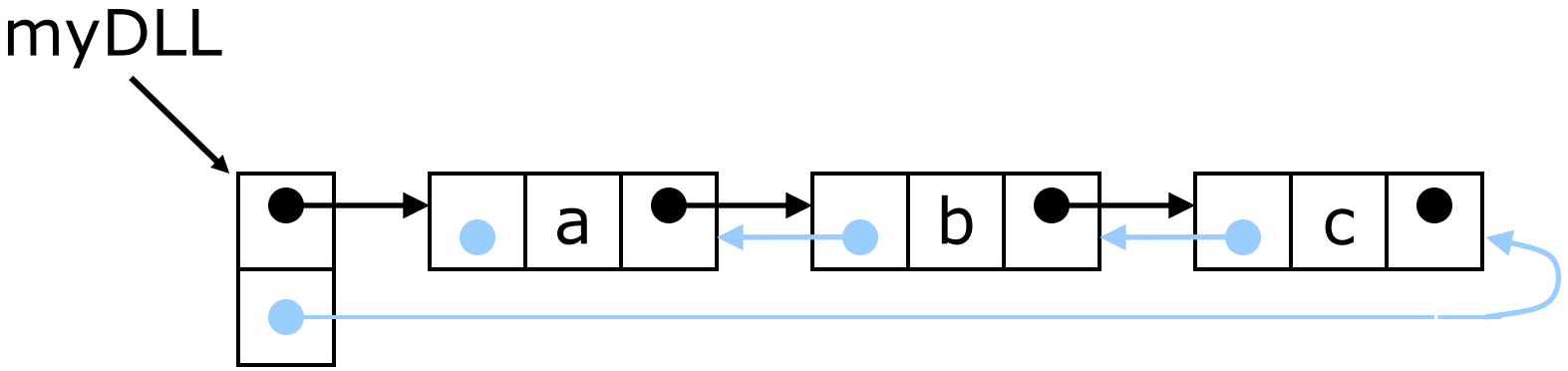


The `add` method (2)

```
// Inserts the given value at the given index.
// Precondition: 0 <= index <= size()
public void add(int index, int value) {
    if (index == 0) {
        // adding to an empty list
        front = new ListNode(value, front);
    } else {
        // inserting into an existing list
        ListNode current = front;
        for (int i = 0; i < index - 1; i++) {
            current = current.next;
        }
        current.next = new ListNode(value,
                                    current.next);
    }
}
```

Doubly-linked lists

- Here is a doubly-linked list (DLL):



- Each node contains a value, a link to its successor (if any), and a link to its predecessor (if any)
- The header points to the first node in the list *and* to the last node in the list (or contains null links if the list is empty)

DLLs compared to SLLs

- Advantages:
  - Can be traversed in either direction (may be essential for some programs)
  - Some operations, such as deletion and inserting before a node, become easier
- Disadvantages:
  - Requires more space
  - List manipulations are slower (because more links must be changed)
  - Greater chance of having bugs (because more links must be manipulated)

Constructing SLLs and DLLs (p. 74)

```
public class SLL {

    private SLLNode first;


    public SLL() {
        this.first = null;
    }


    // methods...
}
```

```
public class DLL {

    private DLLNode first;
    private DLLNode last;

    public DLL() {
        this.first = null;
        this.last = null;
    }

    // methods...
}
```

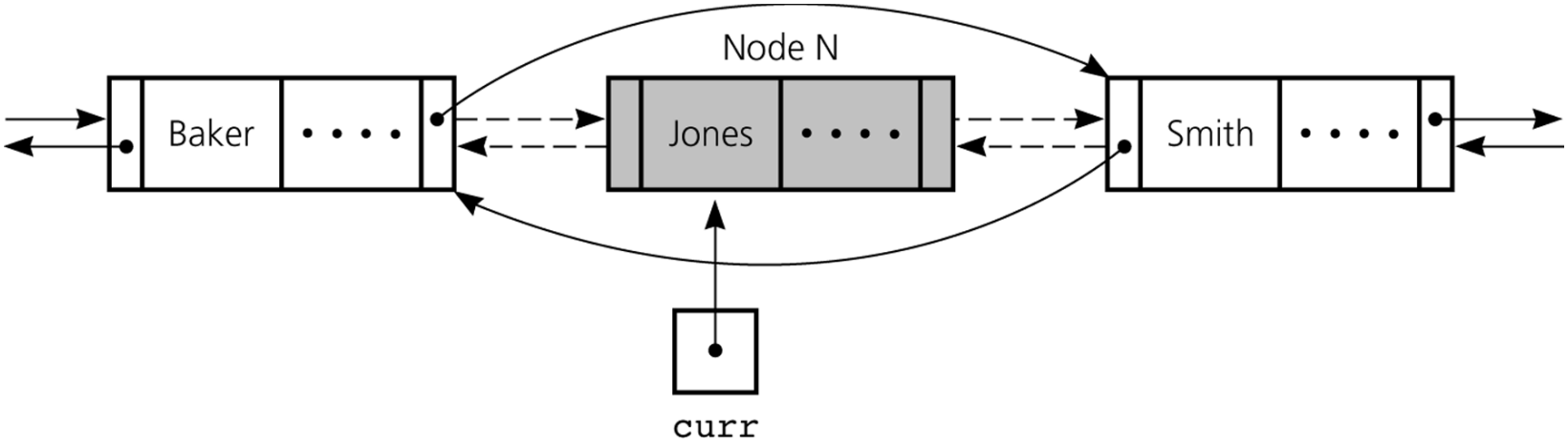
DLL nodes in Java

```
public class DLLNode {
    protected Object element;
    protected DLLNode pred, succ;

    protected DLLNode(Object elem,
                      DLLNode pred,
                      DLLNode succ) {
        this.element = elem;
        this.pred = pred;
        this.succ = succ;
    }
}
```

Doubly Linked List

- To **delete** the node that `curr` references

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



Doubly Linked List

- To **insert** a new node that `newNode` references before the node referenced by `curr`

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

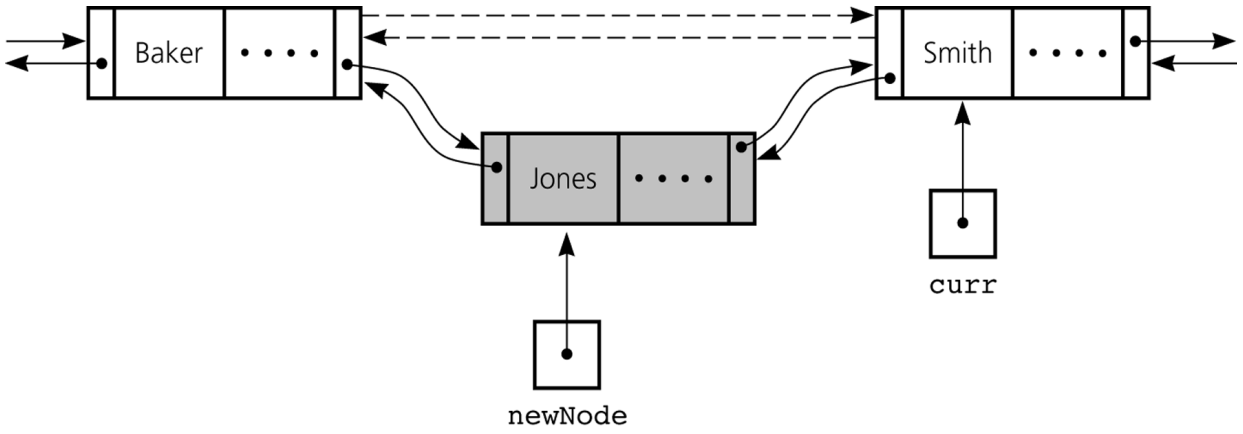


Figure 5-29

Reference changes for insertion

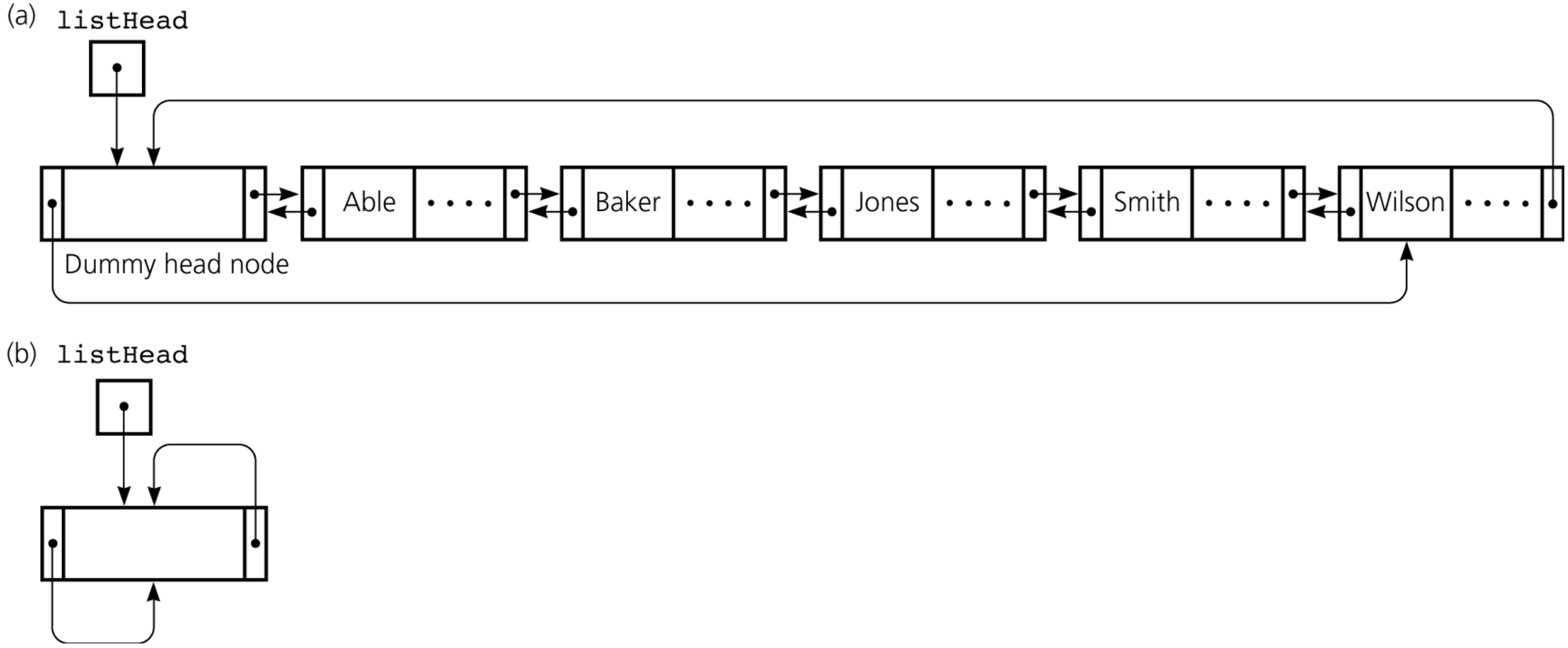
Application: Maintaining an Inventory

- Stages of the problem-solving process
  - Design of a solution
  - Implementation of the solution
  - Final set of refinements to the program
- Operations on the inventory
  - List the inventory in alphabetical order by title (L command)
  - Find the inventory item associated with title (I, M, D, O, and S commands)
  - Replace the inventory item associated with a title (M, D, R, and S commands)
  - Insert new inventory items (A and D commands)

Doubly Linked List

- Circular doubly linked list
  - `precede` reference of the dummy head node references the last node
  - `next` reference of the last node references the dummy head node
  - Eliminates special cases for insertions and deletions

Doubly Linked List



a) A circular doubly linked list with a dummy head node; b) an empty list with a dummy head node

Other operations on linked lists

- Most “algorithms” on linked lists—such as insertion, deletion, and searching—are pretty obvious; you just need to be careful

- Sorting a linked list is just messy, since you can’t directly access the n^{th} element—you have to count your way through a lot of other elements