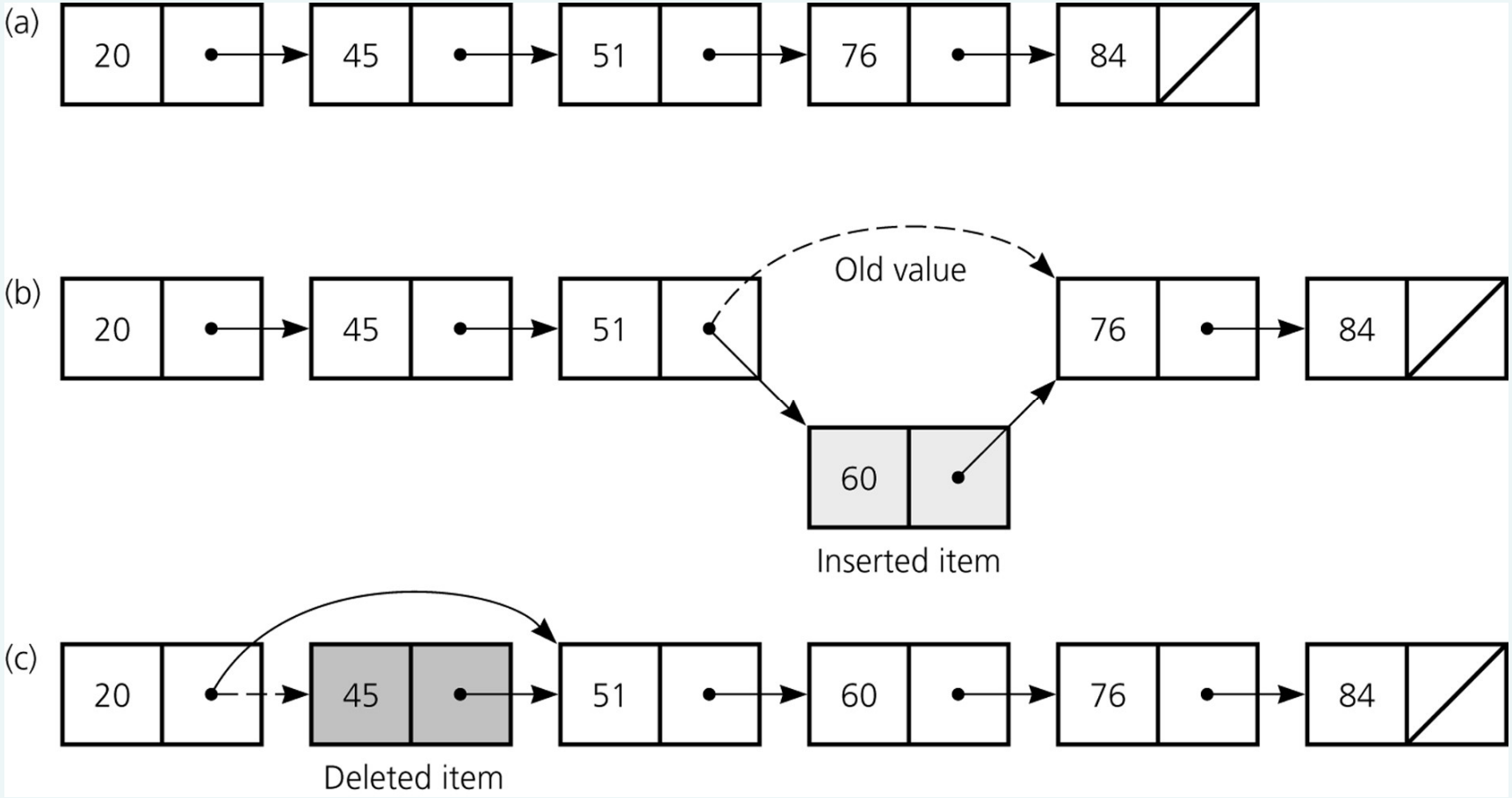
Linked Lists

Preliminaries

- Options for implementing an ADT (List)
  - Array
    - Has a fixed size
    - Data must be shifted during insertions and deletions
  - Linked list
    - Is able to grow in size as needed
    - Does not require the shifting of items during insertions and deletions

Preliminaries



a) A linked list of integers; b) insertion; c) deletion

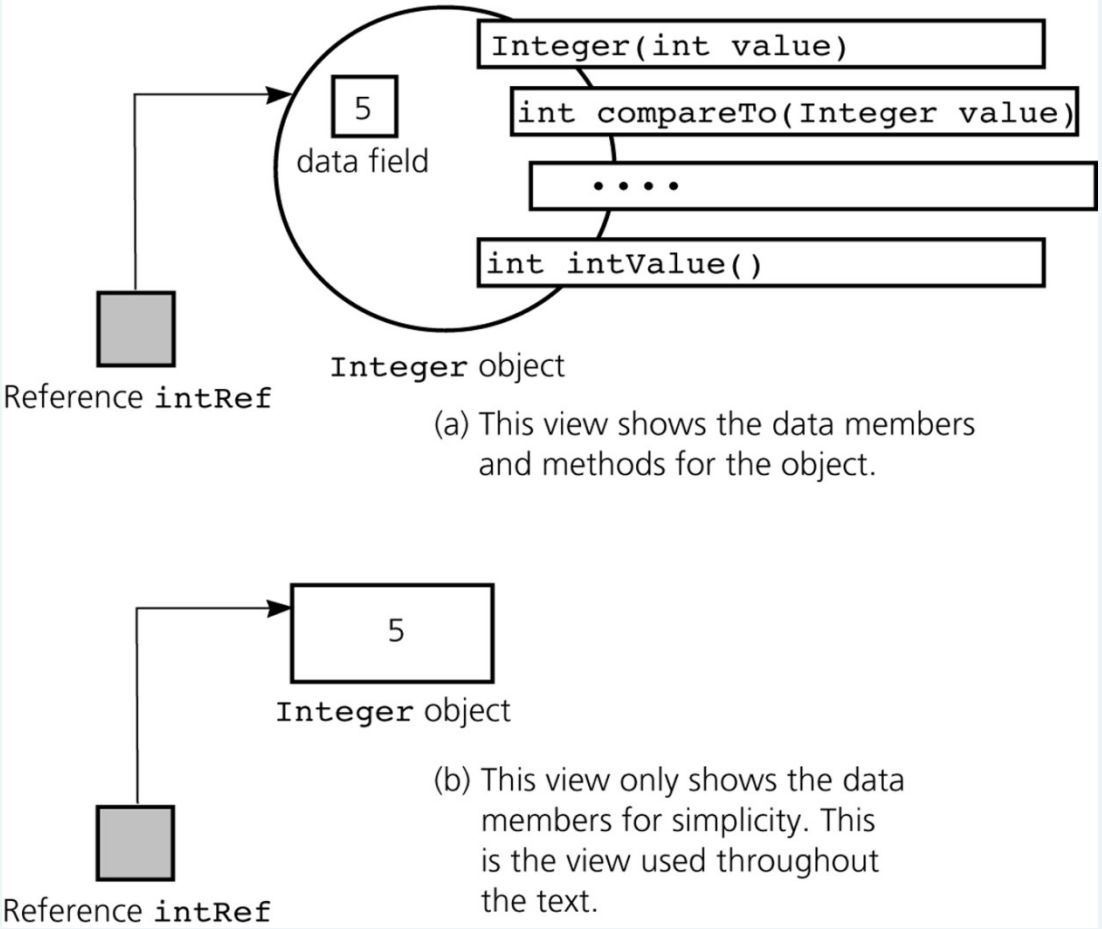
Object References

- A reference variable
  - Contains the location of an object
  - Example

```
Integer intRef;
intRef = new Integer(5);
```

  - As a data field of a class
    - Has the default value `null`
  - A local reference variable to a method
    - Does not have a default value

Object References



A reference to an ***`Integer`*** object

Object References

- When one reference variable is assigned to another reference variable, both references then refer to the same object

```
Integer p, q;
p = new Integer(6);
q = p;
```

- A reference variable that no longer references any object is marked for garbage collection

Object References

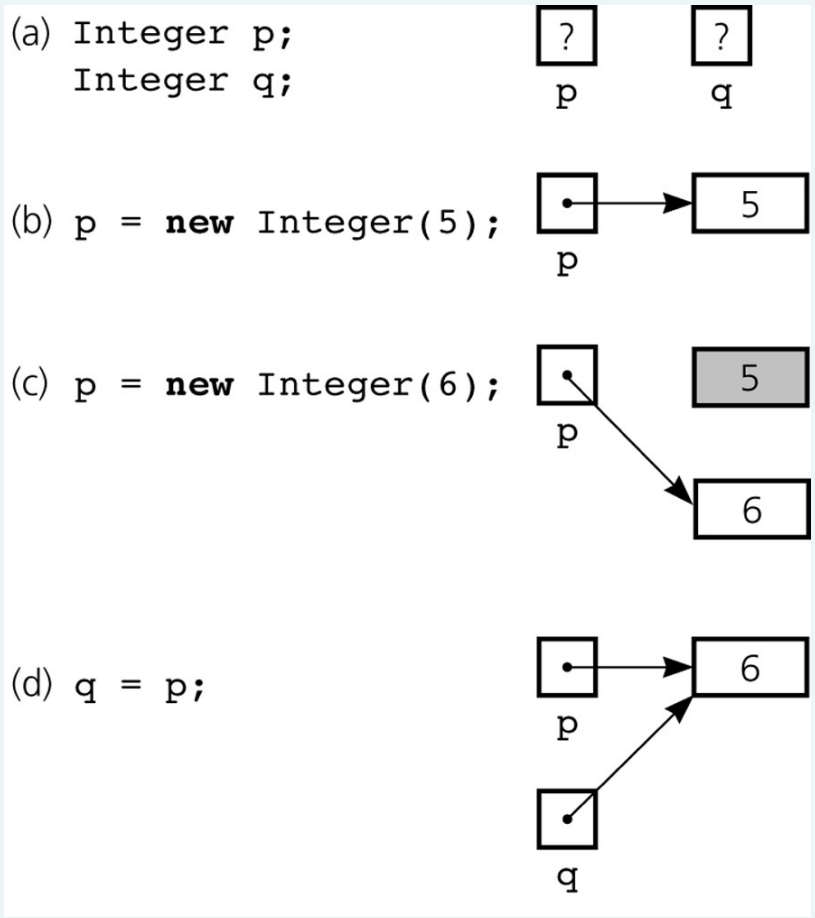


Figure 5-3a-d

a) Declaring reference variables; b) allocating an object; c) allocating another object, with the dereferenced object marked for garbage collection

Object References

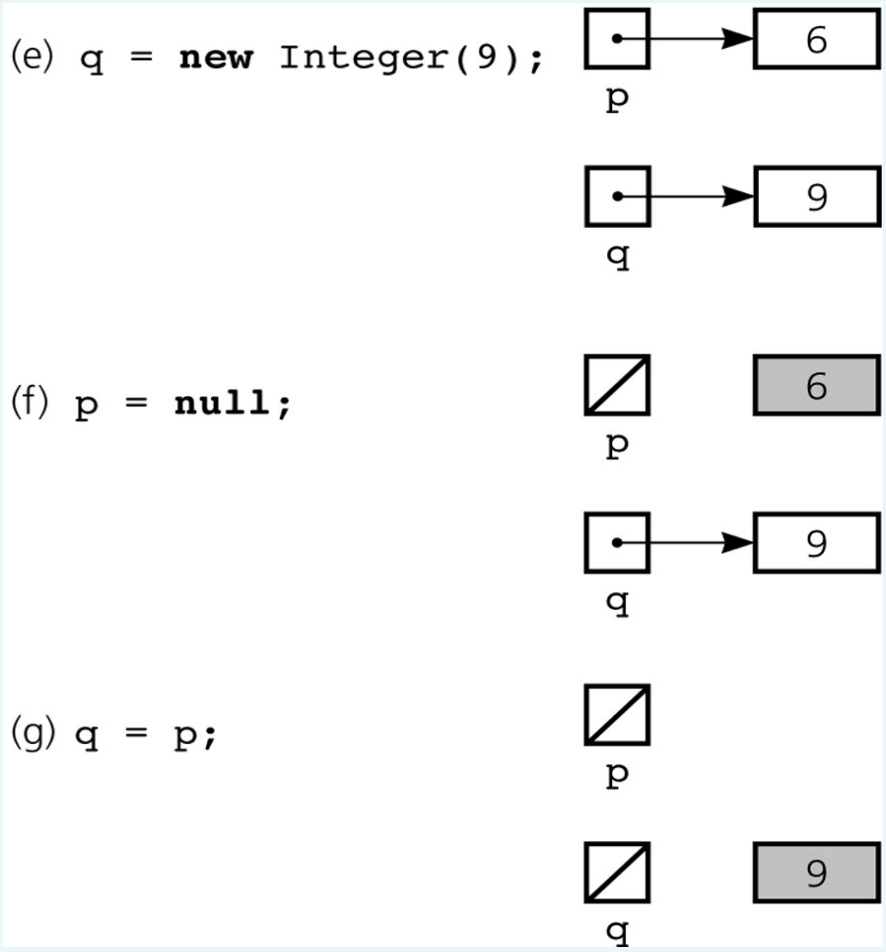


Figure 5-3e-g

e) allocating an object; f) assigning ***null*** to a reference variable; g) assigning a reference with a ***null*** value

Object References

- An array of objects
  - Is actually an array of references to the objects
  - Example

```
Integer[] scores = new Integer[30];
```

  - Instantiating Integer objects for each array reference

```
scores[0] = new Integer(7);
scores[1] = new Integer(9); // and so on …
```

Object References

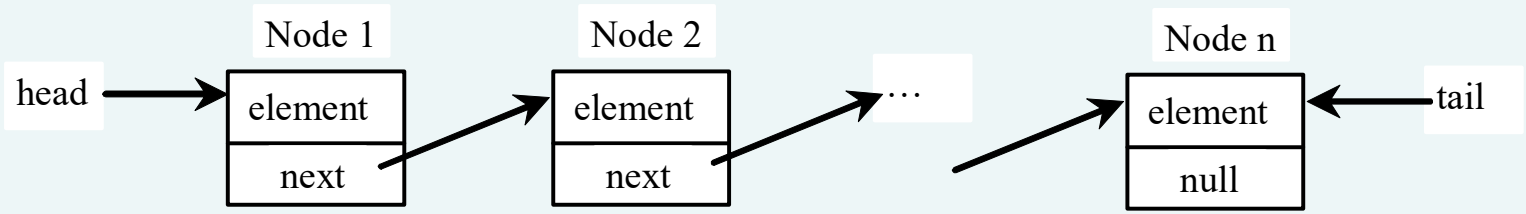
- Equality operators (== and `!=`)
  - Compare the values of the reference variables, not the objects that they reference
- `equals` method
  - Compares objects field by field
- When an object is passed to a method as an argument, the reference to the object is copied to the method's formal parameter
- Reference-based ADT implementations and data structures use Java references

Resizable Arrays

- The number of references in a Java array is of fixed size
- Resizable array
  - An array that grows and shrinks as the program executes
  - An illusion that is created by using an allocate and copy strategy with fixed-size arrays
- `java.util.Vector` class
  - Uses a similar technique to implement a growable array of objects

Nodes in Linked Lists

A linked list consists of nodes. Each node contains an element, and each node is linked to its next neighbor. Thus a node can be defined as a class, as follows:




```
class Node<E> {
  E element;
  Node<E> next;

  public Node(E o) {
    element = o;
  }
}
```

Adding Three Nodes

The variable head refers to the first node in the list, and the variable tail refers to the last node in the list. If the list is empty, both are null. For example, you can create three nodes to store three strings in a list, as follows:

Step 1: Declare head and tail:

```
Node<String> head = null;
Node<String> tail = null;
```

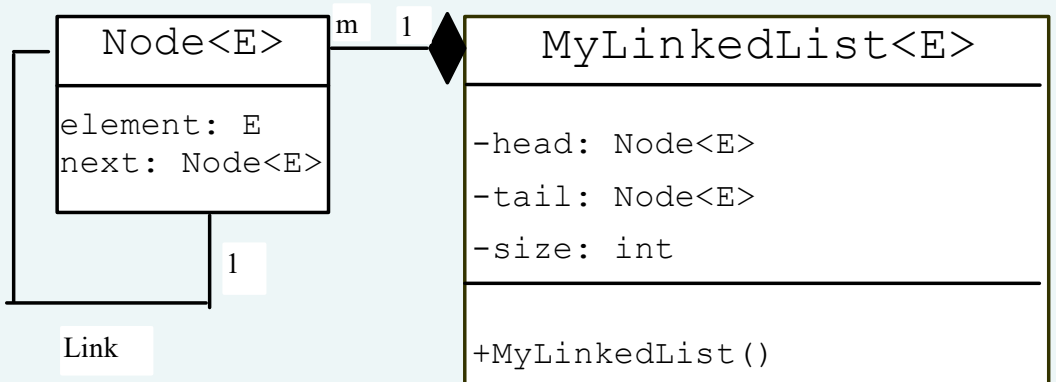
The list is empty now

Traversing All Elements in the List

Each node contains the element and a data field named *next* that points to the next node. If the node is the last in the list, its pointer data field next contains the value null. You can use this property to detect the last node. For example, you may write the following loop to traverse all the nodes in the list.

```
Node<E> current = head;
while (current != null) {
  System.out.println(current.element);
  current = current.next;
}
```

MyLinkedList



The head of the list.

The tail of the list.

The number of elements in the list.

Creates a default linked list.

Creates a linked list from an array of elements.

Adds an element to the head of the list.

Adds an element to the tail of the list.

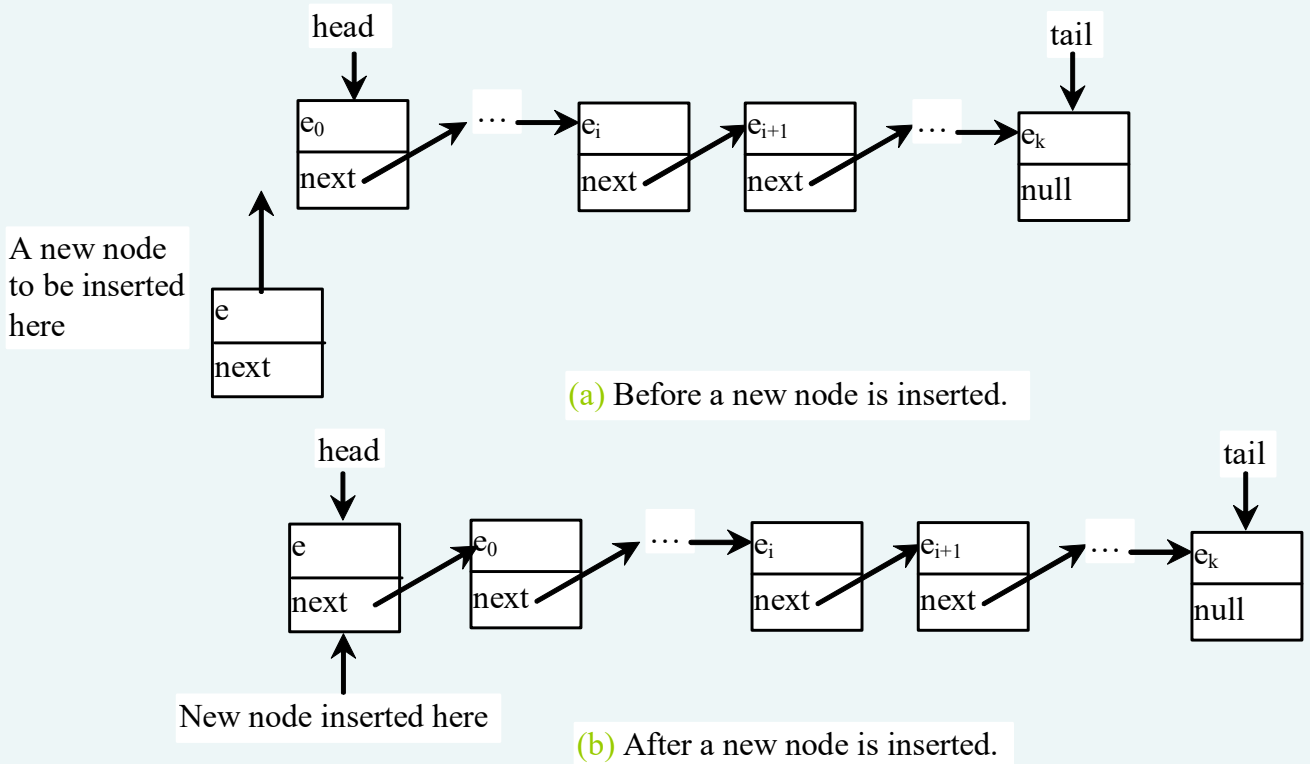
Returns the first element in the list.

Returns the last element in the list.

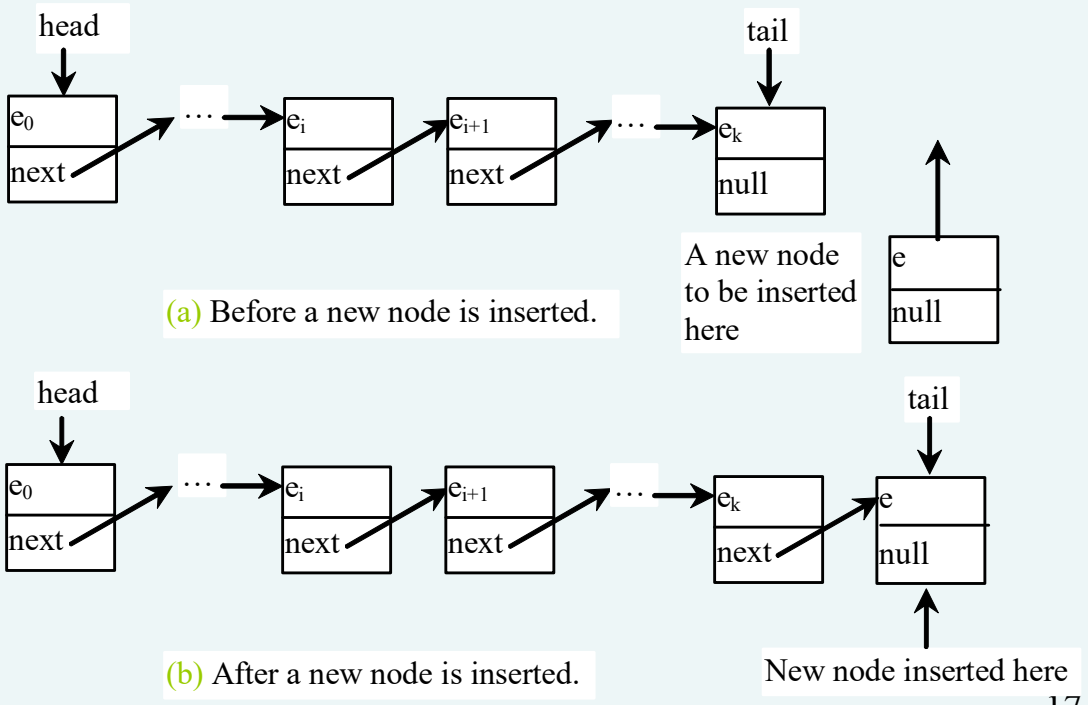
Removes the first element from the list.

Removes the last element from the list.

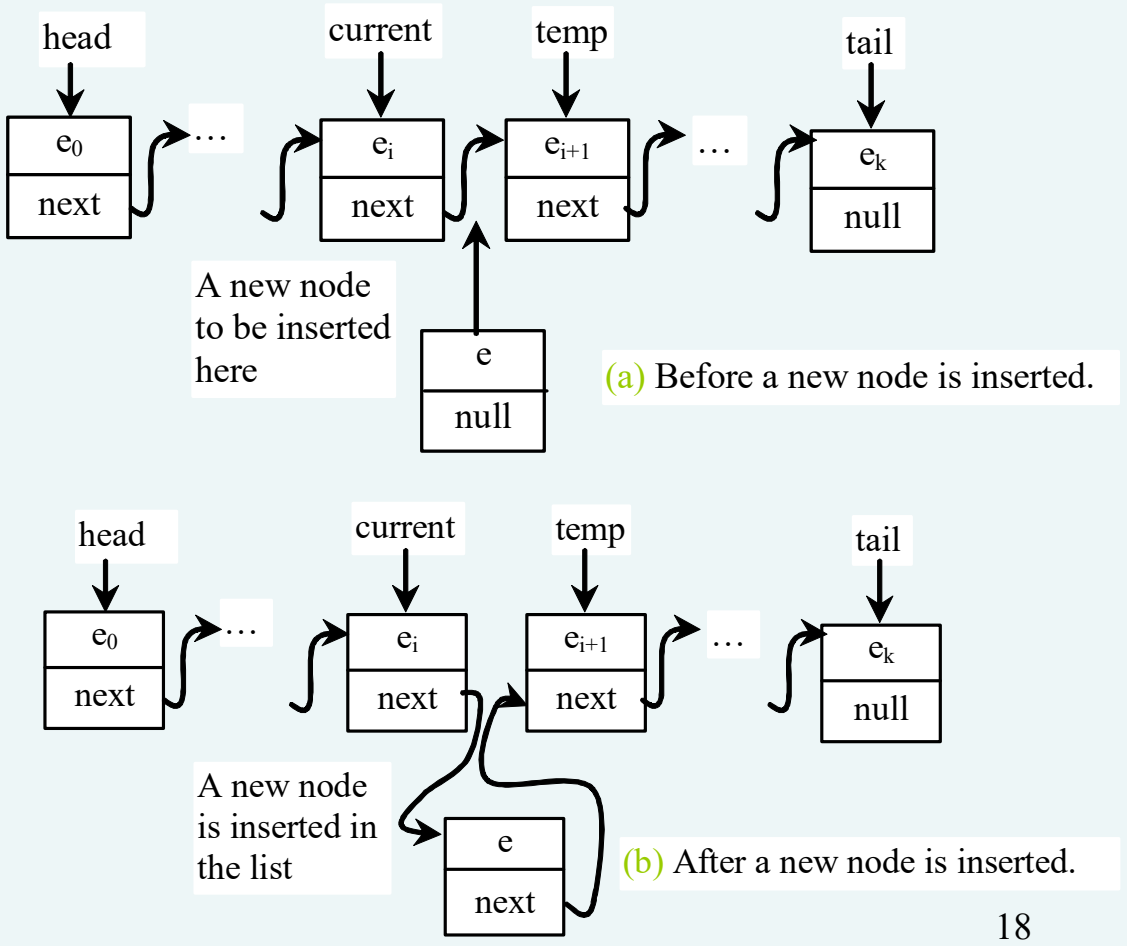
Implementing addFirst(E e)



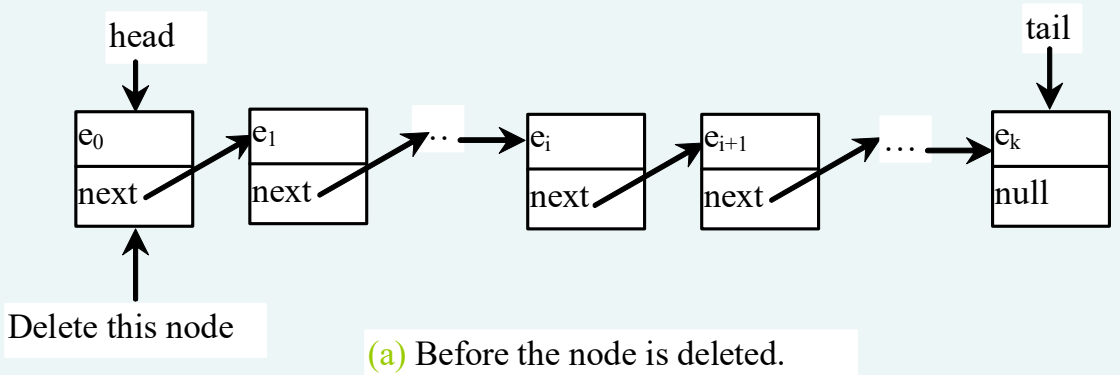
Implementing addLast(E e)



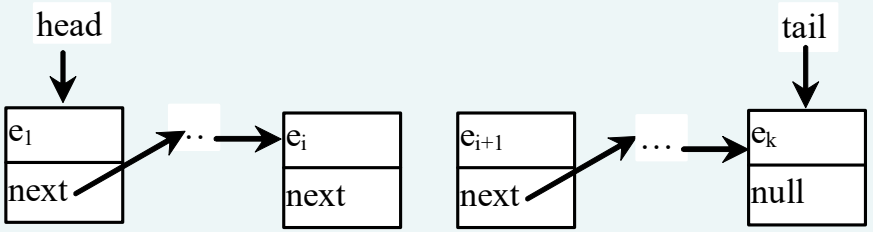
Implementing add(int index, E e)



Implementing removeFirst()

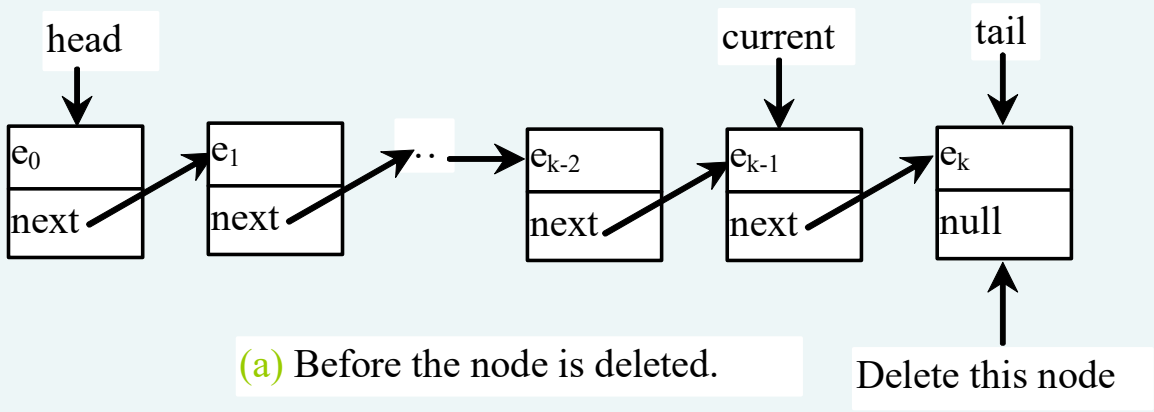


(a) Before the node is deleted.

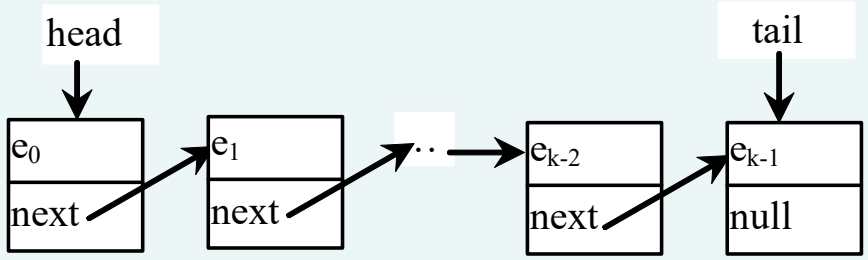


(b) After the first node is deleted

Implementing removeLast()

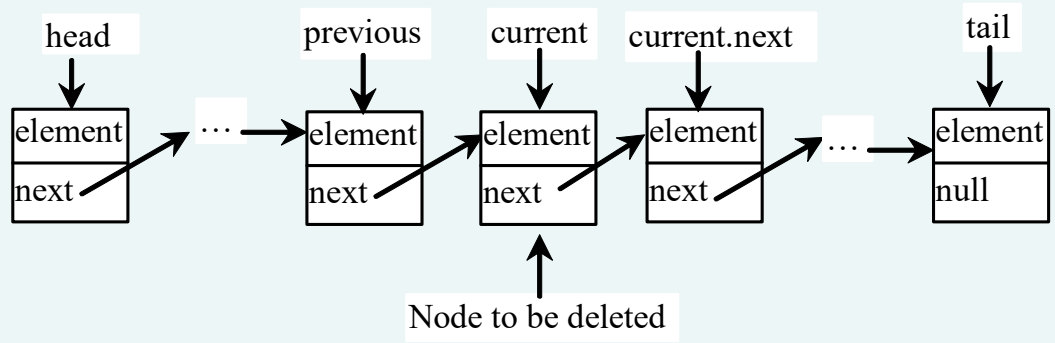


(a) Before the node is deleted.

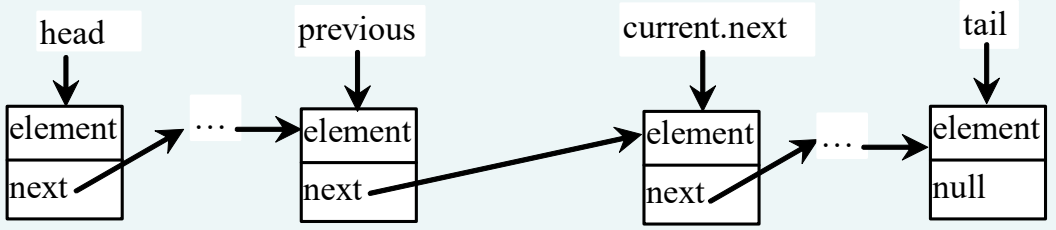


(b) After the last node is deleted

Implementing remove(int index)



(a) Before the node is deleted.



(b) After the node is deleted.

Comparing Array-Based and Referenced-Based Implementations

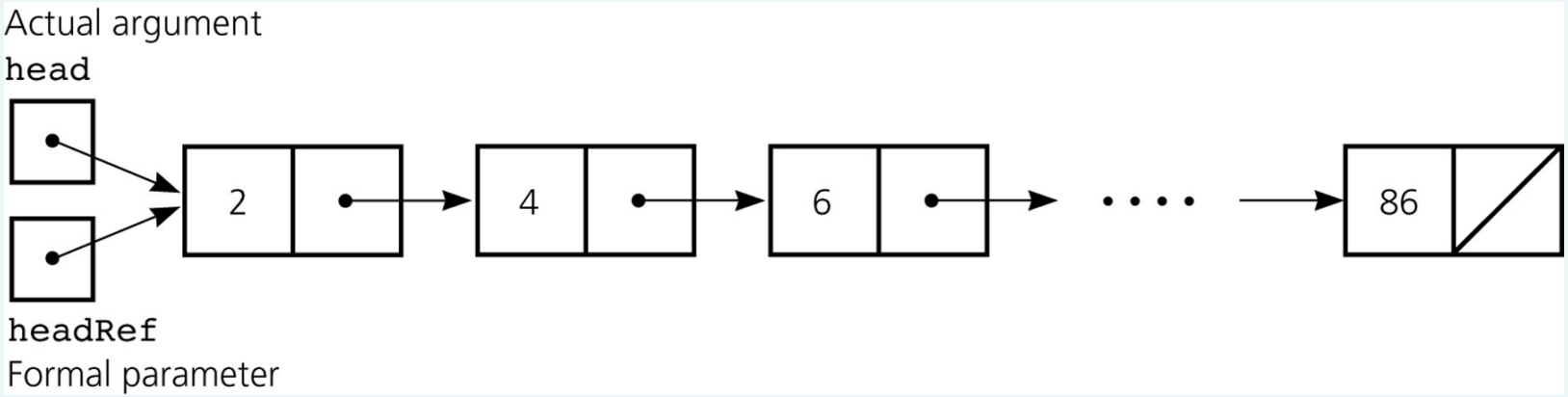
- Size
  - Array-based
    - Fixed size
    - Resizable array
- Access time
  - Array-based
  - Reference-based
- Insertion and deletions
  - Array-based
  - Reference-based

Linked List Efficiency

- Insertion and Deletion from the beginning are very fast. Efficiency is O(1).
- Finding, deleting or inserting next to a specific node requires searching through the list. This required O(N) comparisons.
  - Still faster than arrays because no shifting required.
- Expandable , initial size determination is not necessary.

Passing a Linked List to a Method

- A method with access to a linked list's `head` reference has access to the entire list
- When head is an actual argument to a method, its value is copied into the corresponding formal parameter

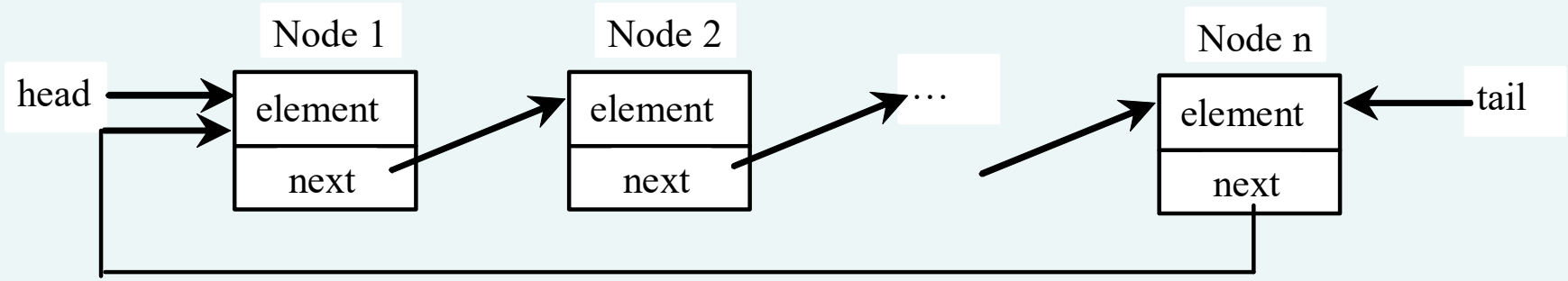


Processing Linked Lists Recursively

- Traversal
  - Recursive strategy to display a list
    - Write the first node of the list
    - Write the list minus its first node
  - Recursive strategies to display a list backward

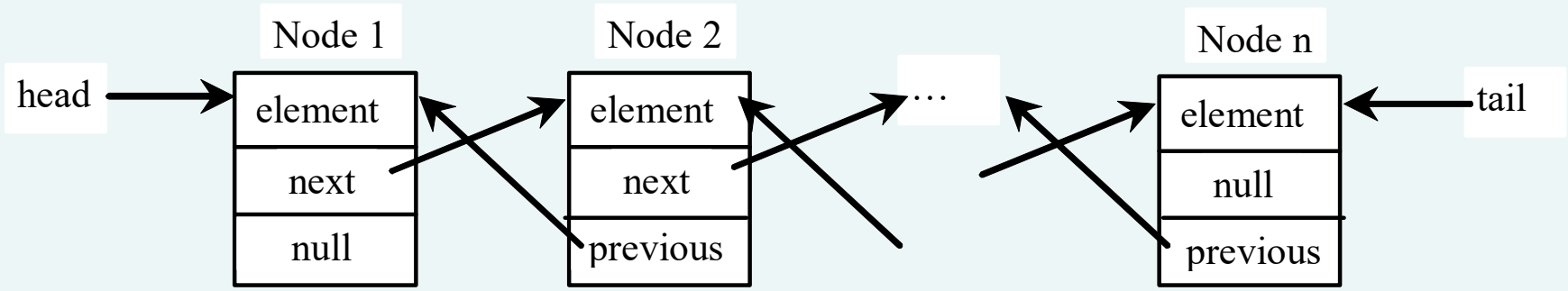
Circular Linked Lists

A *circular, singly linked list* is like a singly linked list, except that the pointer of the last node points back to the first node.

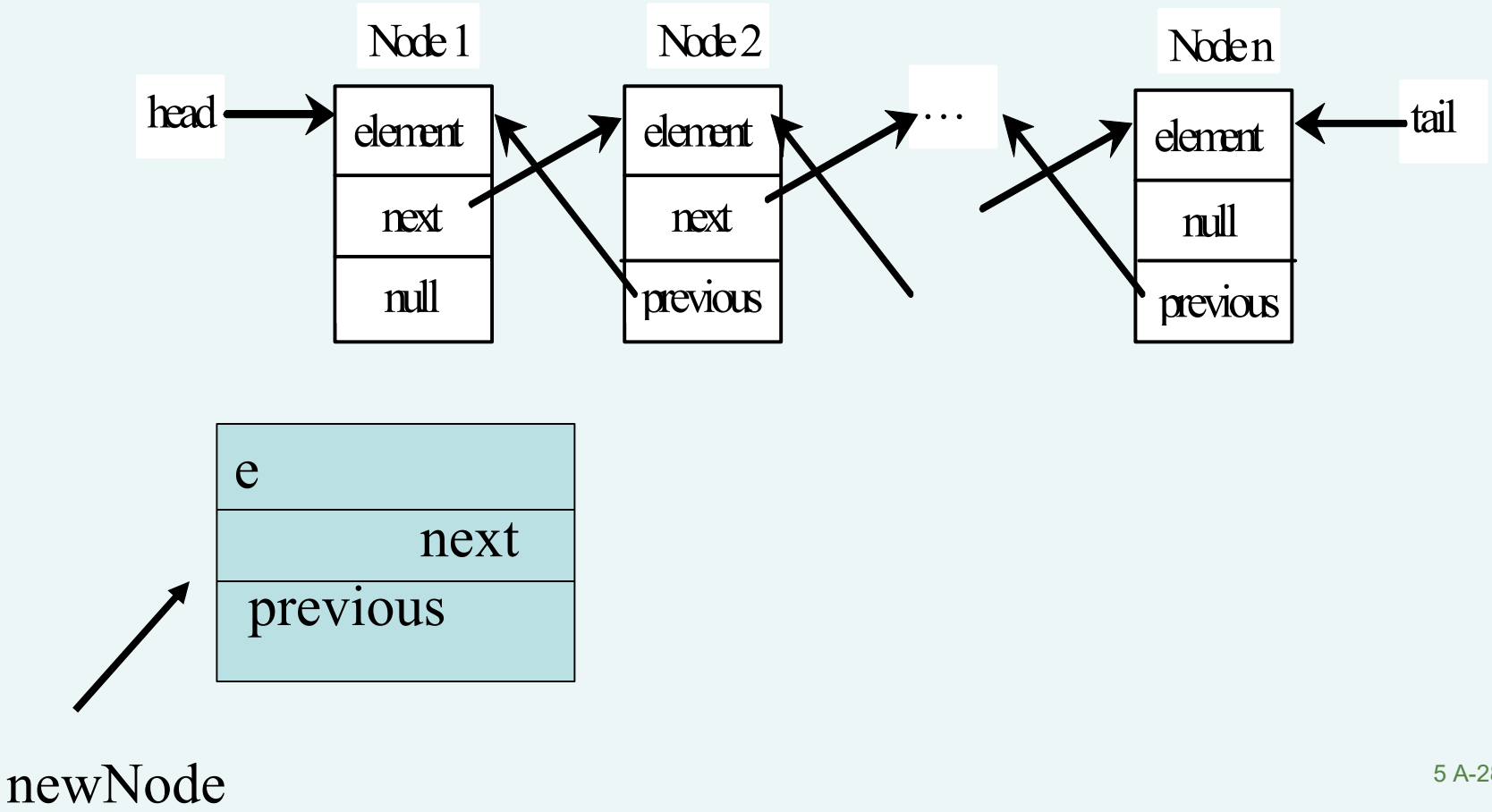


Doubly Linked Lists

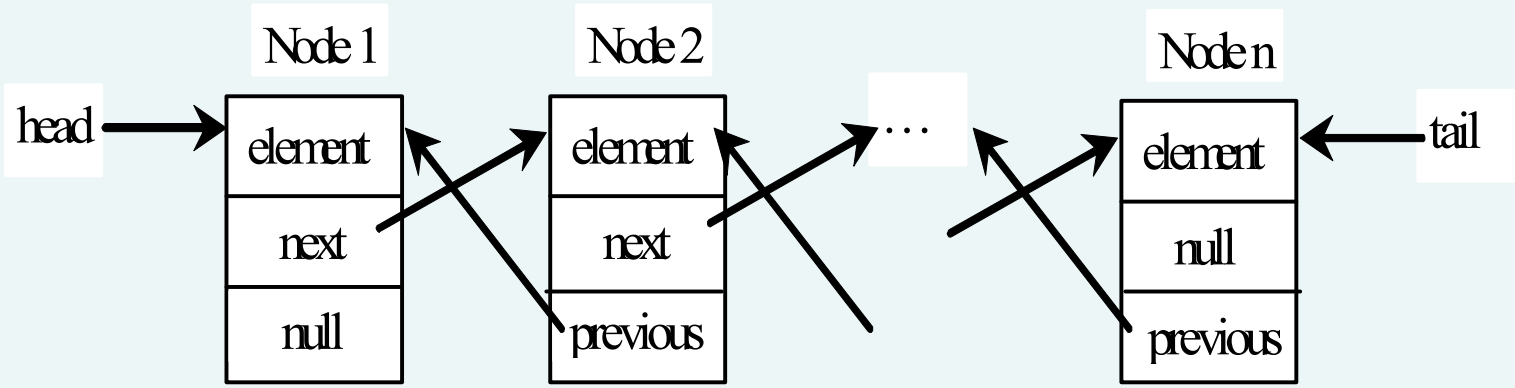
A *doubly linked list* contains the nodes with two pointers. One points to the next node and the other points to the previous node. These two pointers are conveniently called *a forward pointer* and *a backward pointer*. So, a doubly linked list can be traversed forward and backward.



Implementing addFirst(E e)

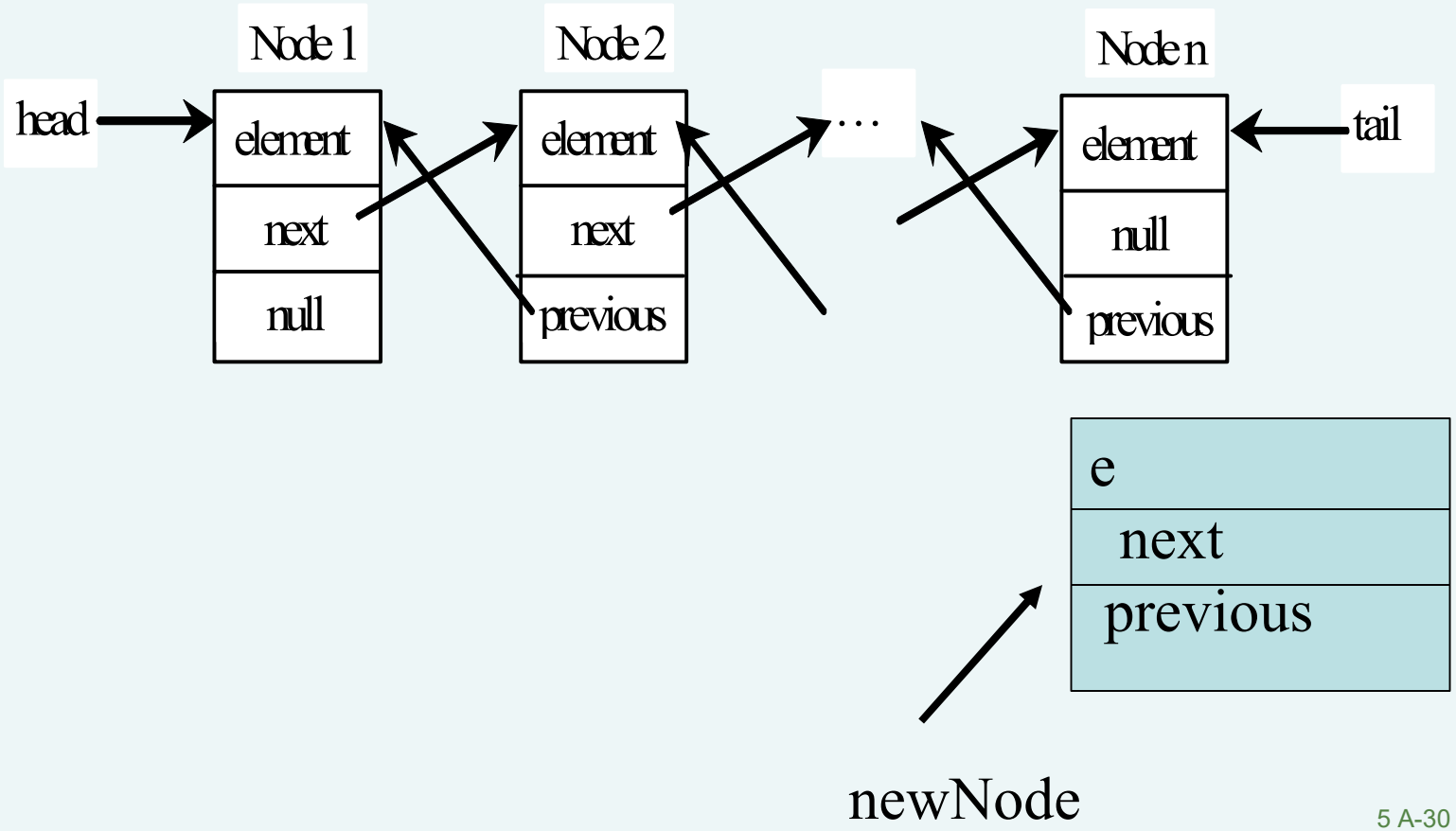


Implementing removeFirst(E e)

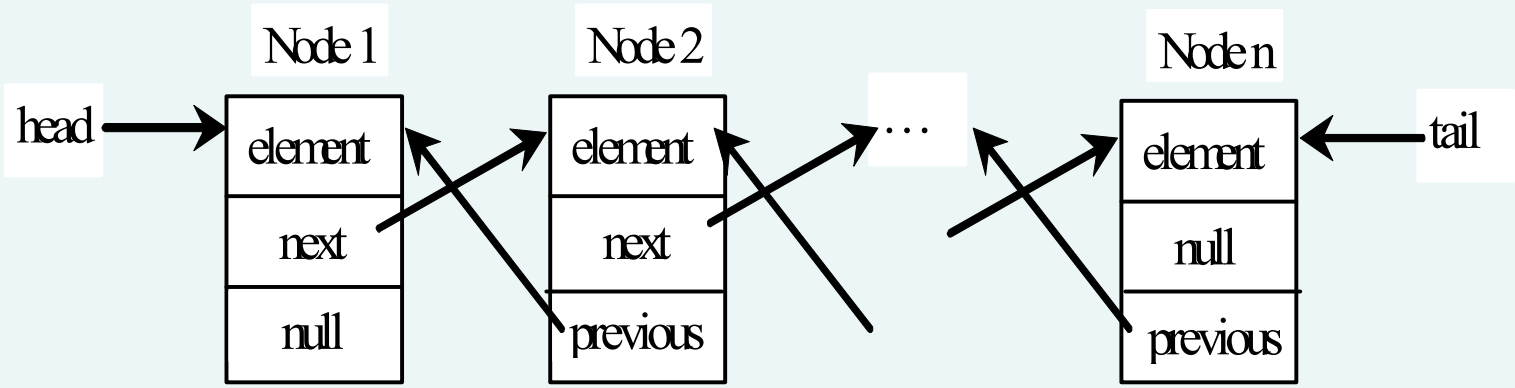


newNode

Implementing addLast(E e)



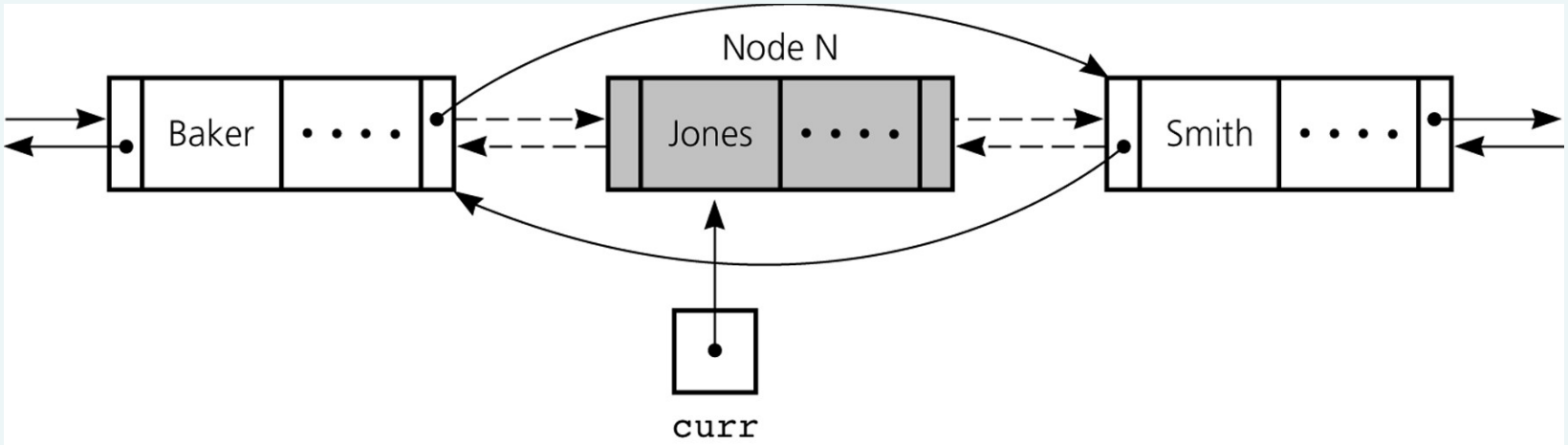
Implementing removeLast(E e)



newNode

Doubly Linked List

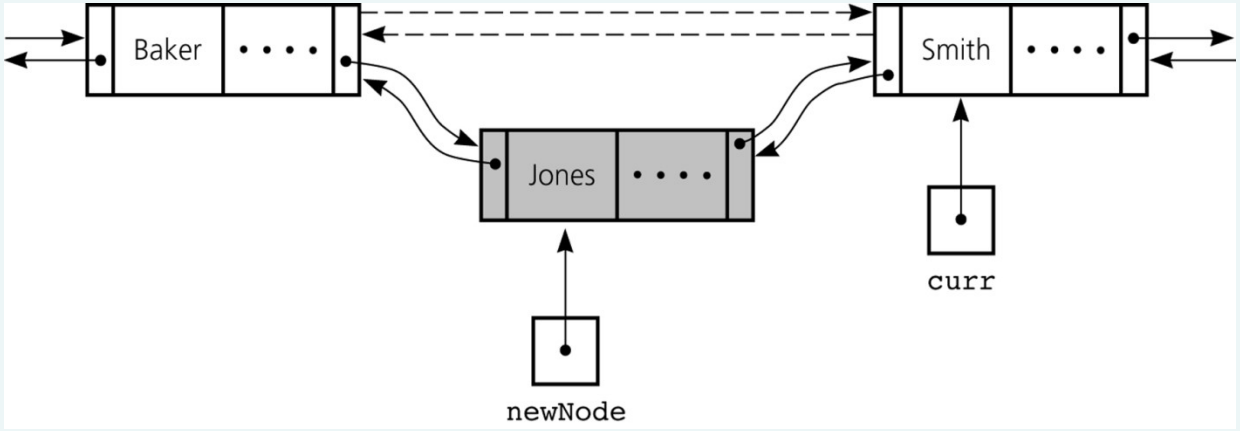
- To delete the node that `curr` references



Reference changes for deletion

Doubly Linked List

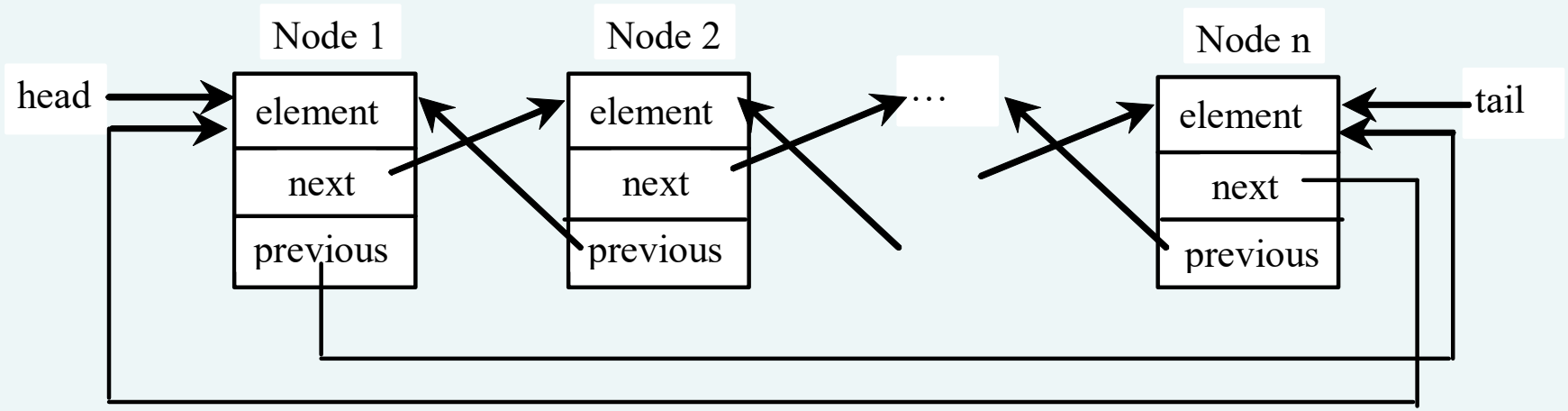
- To insert a new node that `newNode` references before the node referenced by `curr`



Reference changes for insertion

Circular Doubly Linked Lists

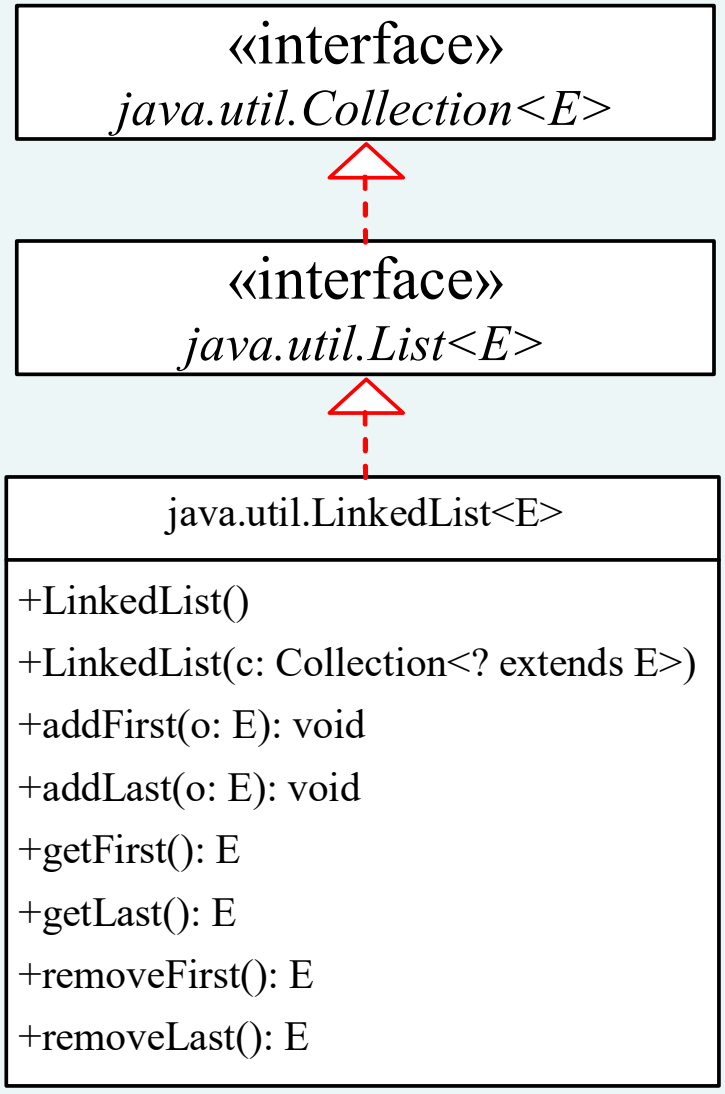
A *circular*, *doubly linked list* is doubly linked list, except that the forward pointer of the last node points to the first node and the backward pointer of the first pointer points to the last node.



JCF - LinkedList

- LinkedList class is concrete implementation of the List interface.
- If your application requires the insertion or deletion of elements from any place in the list, you should choose LinkedList.

java.util.LinkedList



Creates a default empty linked list.

Creates a linked list from an existing collection.

Adds the object to the head of this list.

Adds the object to the tail of this list.

Returns the first element from this list.

Returns the last element from this list.

Returns and removes the first element from this list.

Returns and removes the last element from this list.

Time Complexity for ArrayList and LinkedList

Methods	MyArrayList/ArrayList	MyLinkedList/LinkedList
`add(e: E)`	$O(1)$	$O(1)$
`add(index: int, e: E)`	$O(n)$	$O(n)$
`clear()`	$O(1)$	$O(1)$
`contains(e: E)`	$O(n)$	$O(n)$
`get(index: int)`	$O(1)$	$O(n)$
`indexOf(e: E)`	$O(n)$	$O(n)$
`isEmpty()`	$O(1)$	$O(1)$
`lastIndexOf(e: E)`	$O(n)$	$O(n)$
`remove(e: E)`	$O(n)$	$O(n)$
`size()`	$O(1)$	$O(1)$
`remove(index: int)`	$O(n)$	$O(n)$
`set(index: int, e: E)`	$O(n)$	$O(n)$
`addFirst(e: E)`	$O(n)$	$O(1)$
`removeFirst()`	$O(n)$	$O(1)$