WIA 1002 DATA STRUCTURE

SEM 2, SESSION 2024/205

NURUL JAPAR
nuruljapar@um.edu.my

HOO WAI LAM
wlhoo@um.edu.my

Home of the Bright, Land of the Brave

Di Sini Bermulanya Pintar, Tanah Tumpahnya Berani

UNIVERSITI MALAYA

Linked List

- Part 1: Linked List
- Part 2: Doubly Linked List

Part 1: Linked List

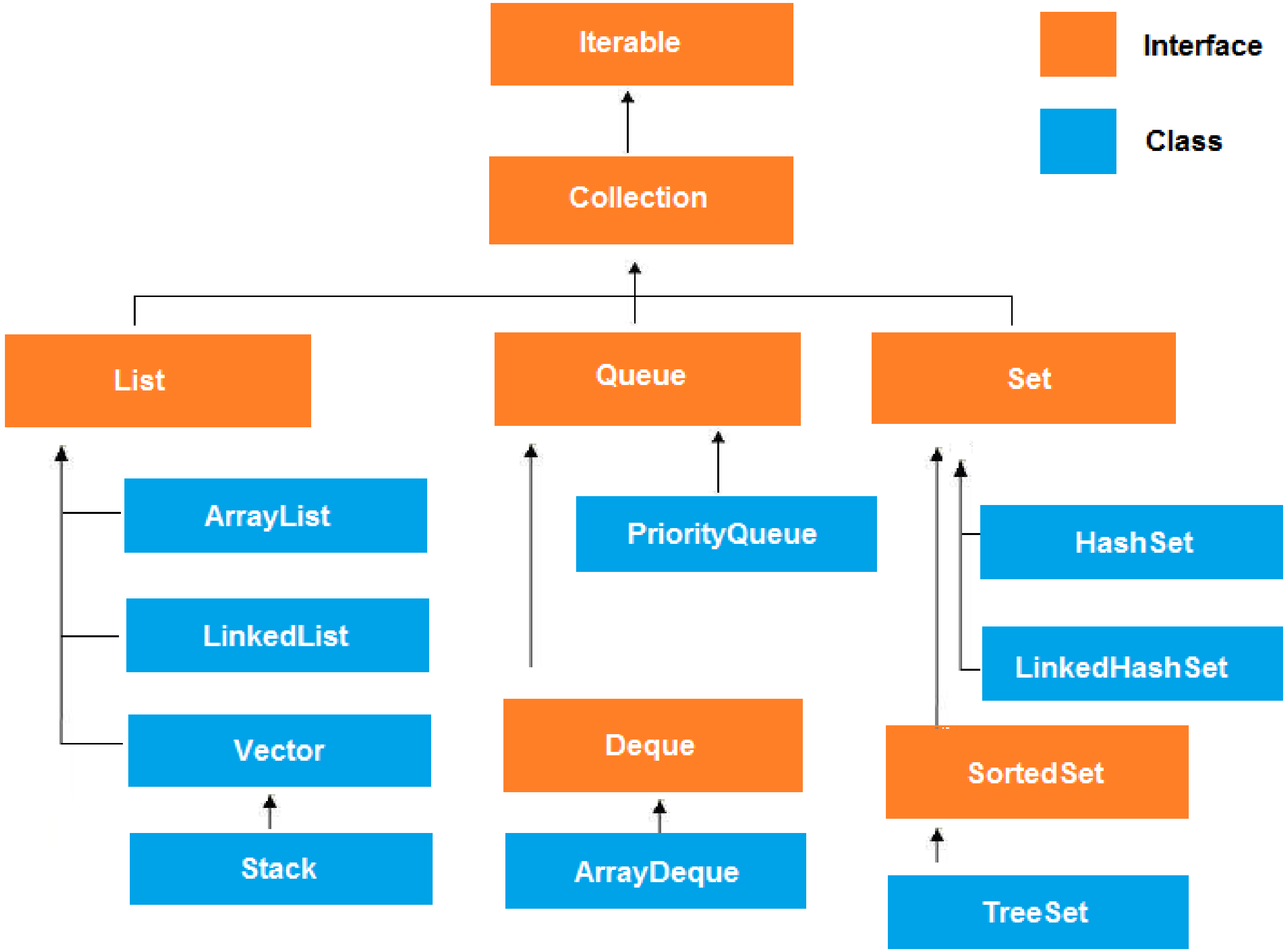
Part 1: Linked List

- Java Collection Framework Hierarchy
- List
- Linked-List
- Implementation of Linked-List
- Types of Linked-List

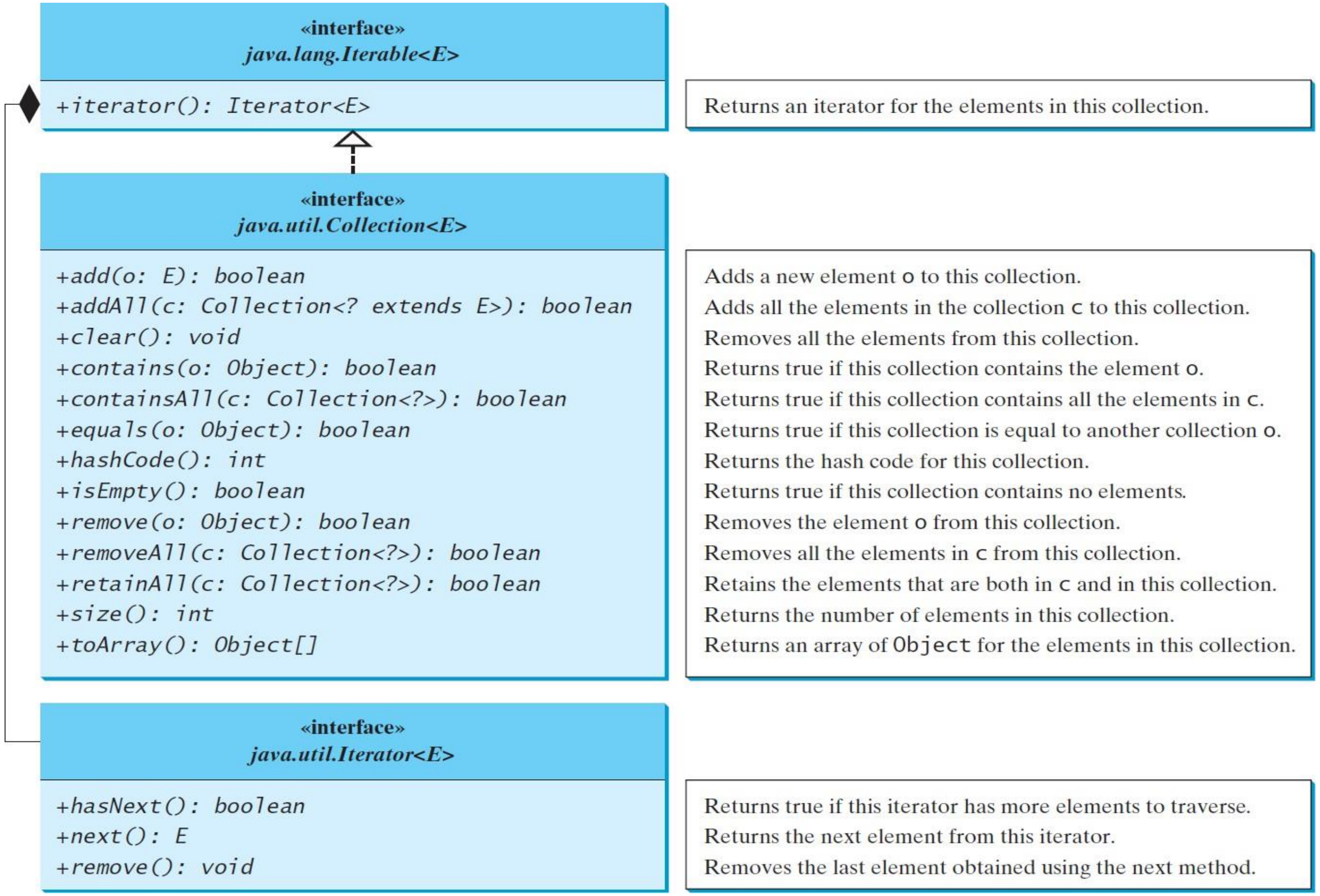
Java Collection Framework Hierarchy

A *collection* is a container object that holds a group of objects, often referred to as *elements*.

Java Collection Framework Hierarchy



The Collection Interface



«interface»
java.lang.Iterable<E>

```
+iterator(): Iterator<E>
```

Returns an iterator for the elements in this collection.

«interface»
java.util.Collection<E>

```
+add(o: E): boolean
+addAll(c: Collection<? extends E>): boolean
+clear(): void
+contains(o: Object): boolean
+containsAll(c: Collection<?>): boolean
+equals(o: Object): boolean
+hashCode(): int
+isEmpty(): boolean
+remove(o: Object): boolean
+removeAll(c: Collection<?>): boolean
+retainAll(c: Collection<?>): boolean
+size(): int
+toArray(): Object[]
```

Adds a new element o to this collection.
Adds all the elements in the collection c to this collection.
Removes all the elements from this collection.
Returns true if this collection contains the element o.
Returns true if this collection contains all the elements in c.
Returns true if this collection is equal to another collection o.
Returns the hash code for this collection.
Returns true if this collection contains no elements.
Removes the element o from this collection.
Removes all the elements in c from this collection.
Retains the elements that are both in c and in this collection.
Returns the number of elements in this collection.
Returns an array of Object for the elements in this collection.

«interface»
java.util.Iterator<E>

```
+hasNext(): boolean
+next(): E
+remove(): void
```

Returns true if this iterator has more elements to traverse.
Returns the next element from this iterator.
Removes the last element obtained using the next method.

The Collection interface is the root interface for manipulating a collection of objects.

List

- A list is a popular *Abstract Data Type* that stores data in sequential order.
- Examples: a list of students, a list of available rooms, a list of cities, and a list of books, etc.
- The common operations on a list are:
  - ✓ Retrieve an element from this list.
  - ✓ Insert a new element to this list.
  - ✓ Delete an element from this list.
  - ✓ Find how many elements are in this list.
  - ✓ Find if an element is in this list.
  - ✓ Find if this list is empty.

Two Ways to Implement Lists

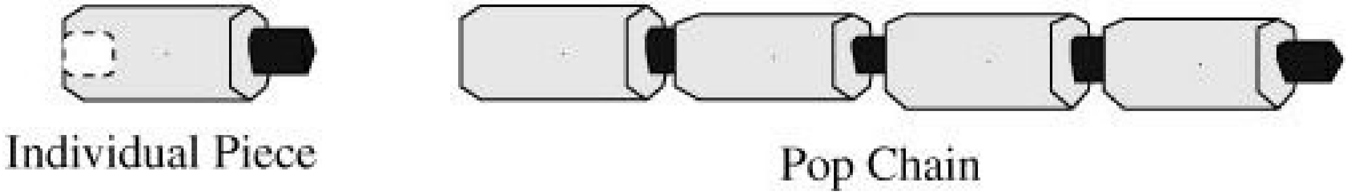
1. Using an array to store the elements
   - » The array is dynamically created.
   - » If array capacity is exceeded, create a new larger array and copy all the elements from the current array to the new array.
2. Using linked list
   - » A linked structure consists of nodes.
   - » Each node is dynamically created to hold an element.
   - » All the nodes are linked together to form a list.

Array or Linked List?

1. Use an array
   - » get(int index) and set(int index, Object o) through an index and add(Object o) for adding an element at the end of the list are efficient.
   - » add(int index, Object o) and remove(int index) are inefficient - shift potentially large number of elements.
2. Use a linked list
   - » improve efficiency for adding and removing an element anywhere in a list.

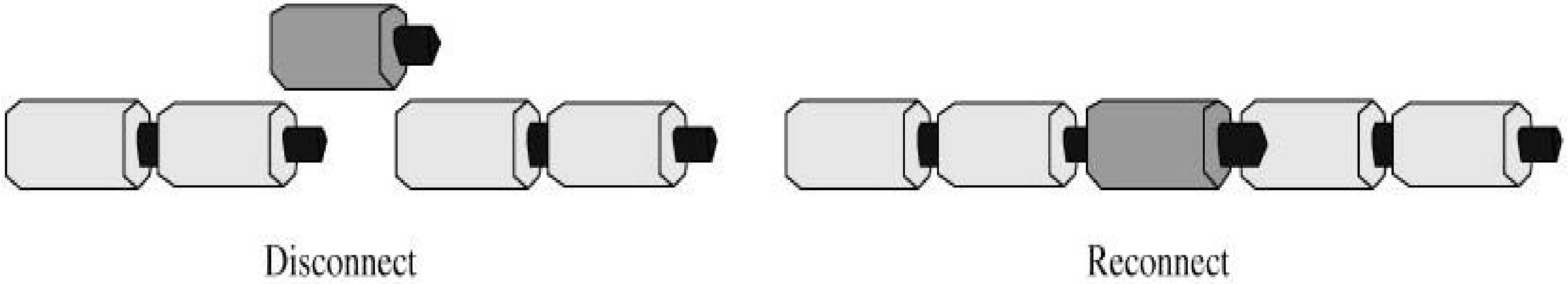
Introducing Linked List

- Think of each element in a linked list as being an individual piece in a child's pop chain. To form a chain, ***insert*** the connector into the back of the next piece



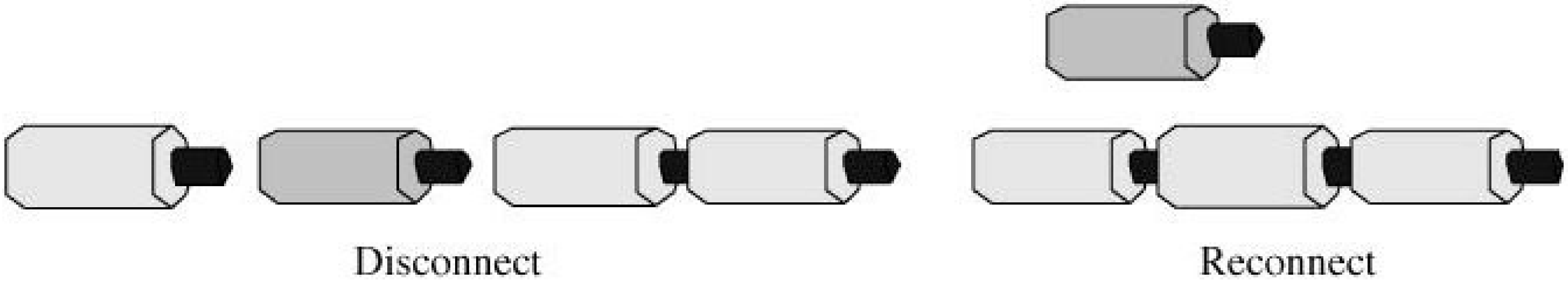
Introducing Linked List

- ***Inserting*** a new piece into the chain involves merely breaking a connection and reconnecting the chain at both ends of the new piece.



Introducing Linked List

- ***Removal*** of a piece from anywhere in the chain requires breaking its two connections, removing the piece, and then reconnecting the chain.



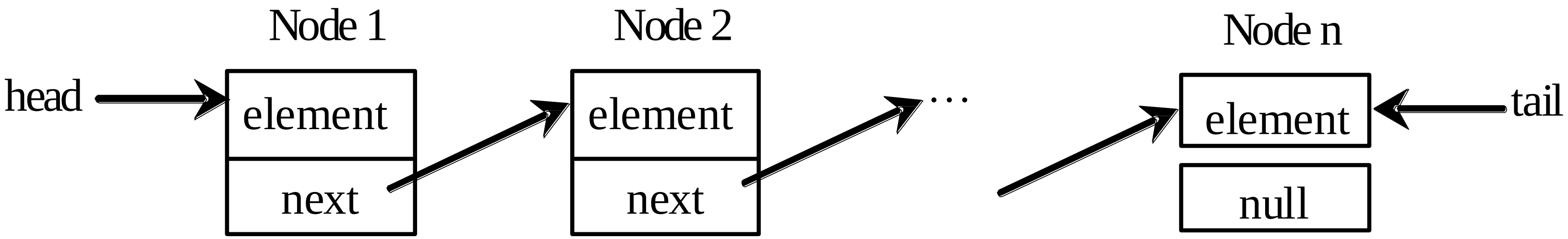
Introducing Linked List

Inserting and deleting an element is a local operation and *requires updating only the links adjacent to the element*. The other elements in the list are not affected.

Nodes in Linked Lists

- A linked list consists of nodes.
- Each node contains an element, and each node is linked to its next neighbor.
- A node with its two fields can reside anywhere in memory.

Nodes in Linked Lists



```
class Node<E> {
   E element;     //contains the element
   Node<E> next; // a reference to the next node

   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:

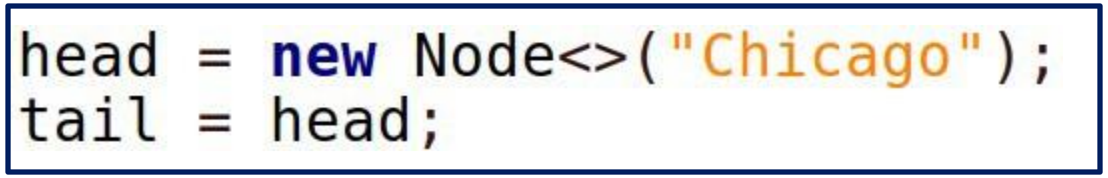
- Step 1: Declare **head** and **tail**.

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

The list is empty now

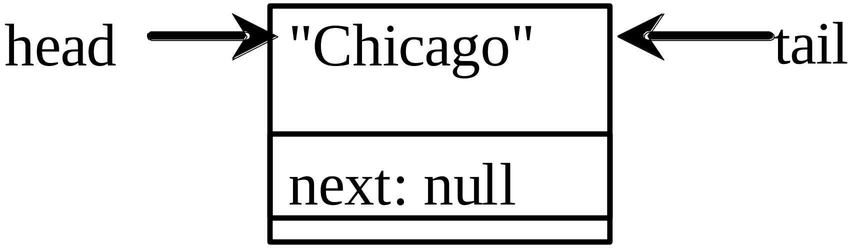
Adding Three Nodes

- Step 2: Create the first node and insert it to the list.



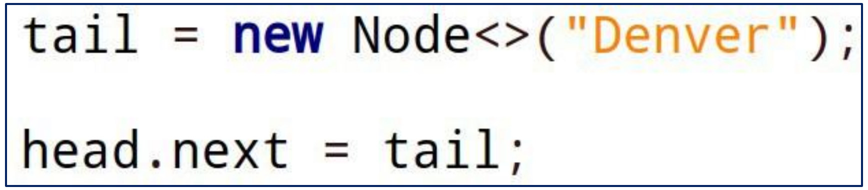
```
head = new Node<>("Chicago");
tail = head;
```

After the first node is inserted

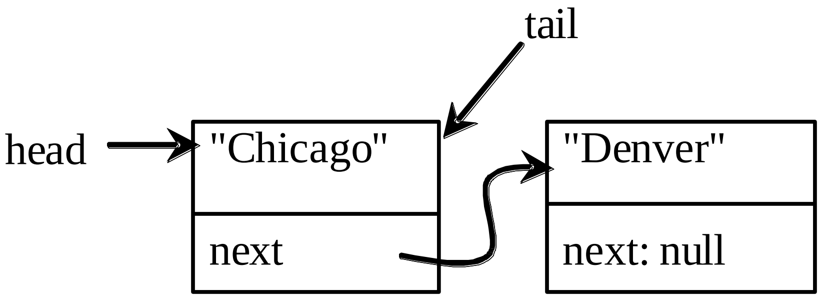


Adding Three Nodes

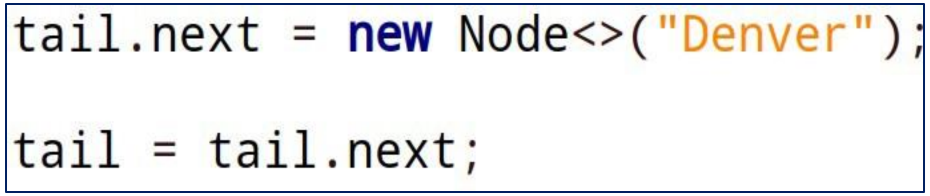
- Step 3: Create the second node and insert it to the list.



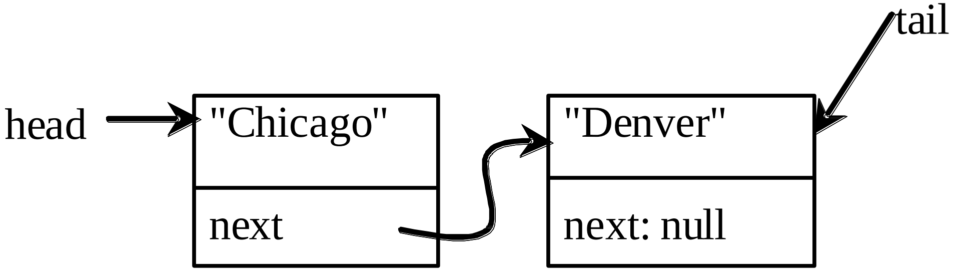
```
tail = new Node<>("Denver");

head.next = tail;
```



or

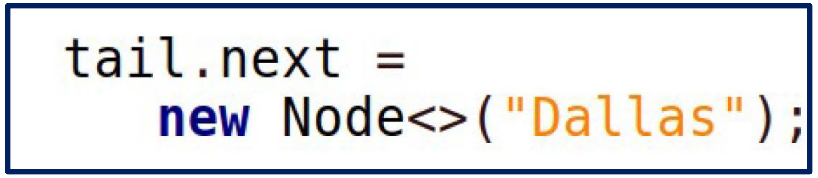


```
tail.next = new Node<>("Denver");

tail = tail.next;
```

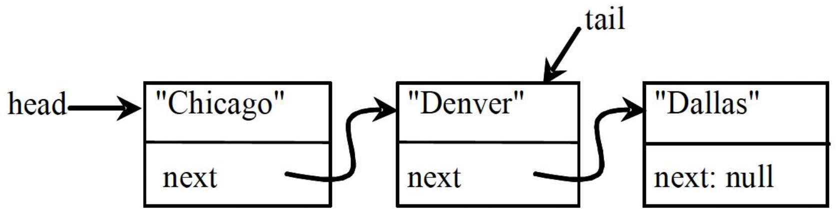


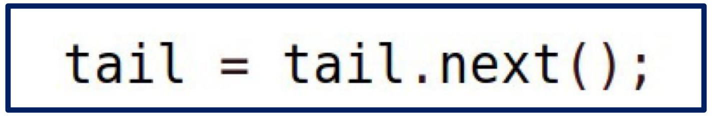
Adding Three Nodes, cont.

- Step 4:Create the third node and insert it to the list.

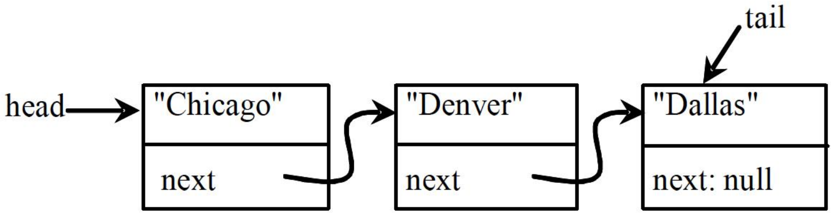


```
tail.next =
    new Node<>("Dallas");
```





```
tail = tail.next();
```



Traversing All Elements in the List

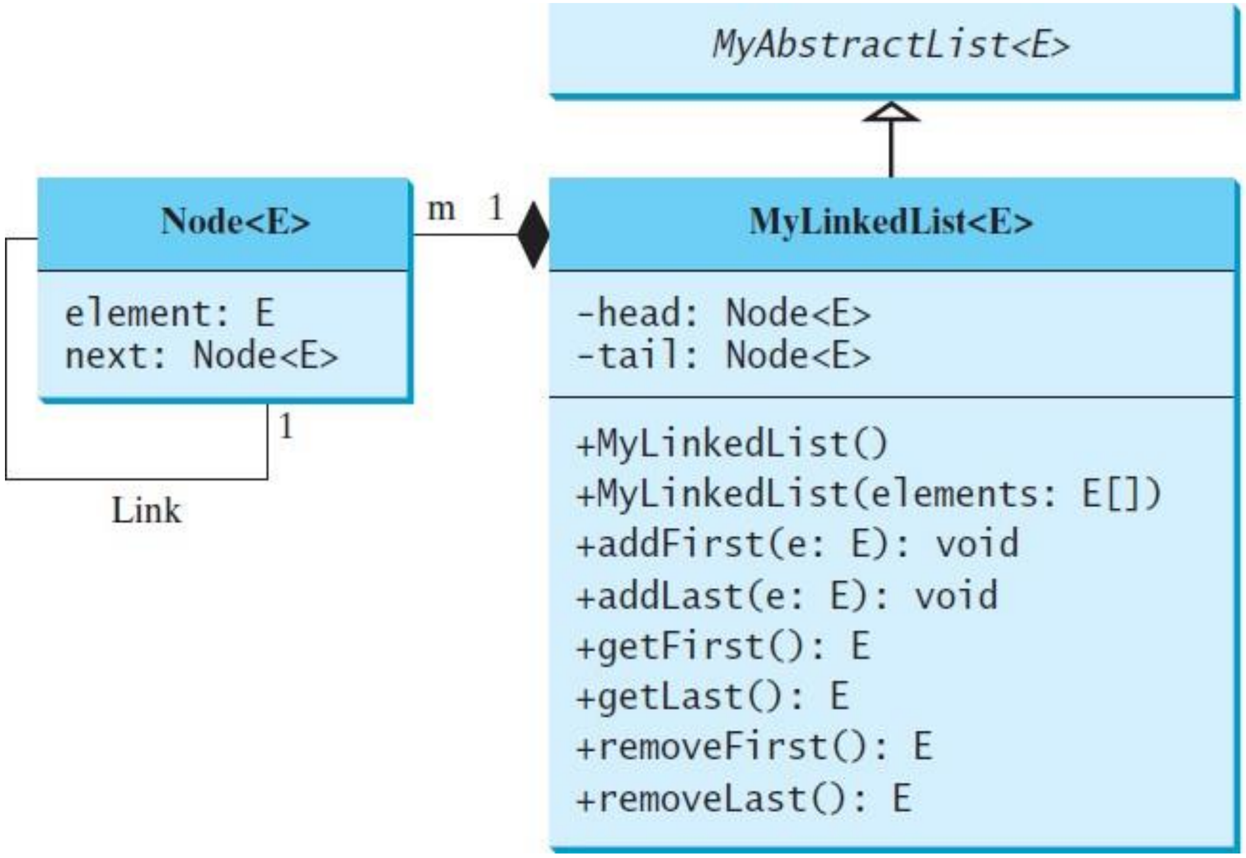
- Each node contains the element and a data field named next that points to the next node.
- If the node is the last in the list, its pointer data field next contains the value null. You can use this property to detect the last node.

Traversing All Elements in the List

- Loop to traverse all the nodes in the list:

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

MyLinkedList



MyAbstractList<E>

Node<E>

element: E
next: Node<E>

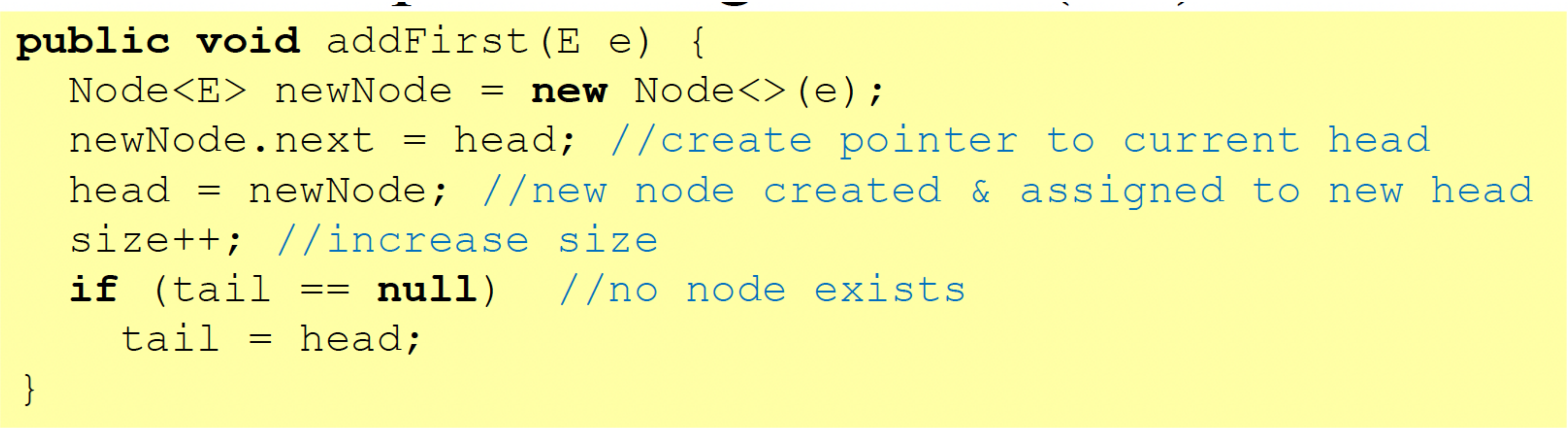
MyLinkedList<E>

-head: Node<E>
-tail: Node<E>

+MyLinkedList()
+MyLinkedList(elements: E[])
+addFirst(e: E): void
+addLast(e: E): void
+getFirst(): E
+getLast(): E
+removeFirst(): E
+removeLast(): E

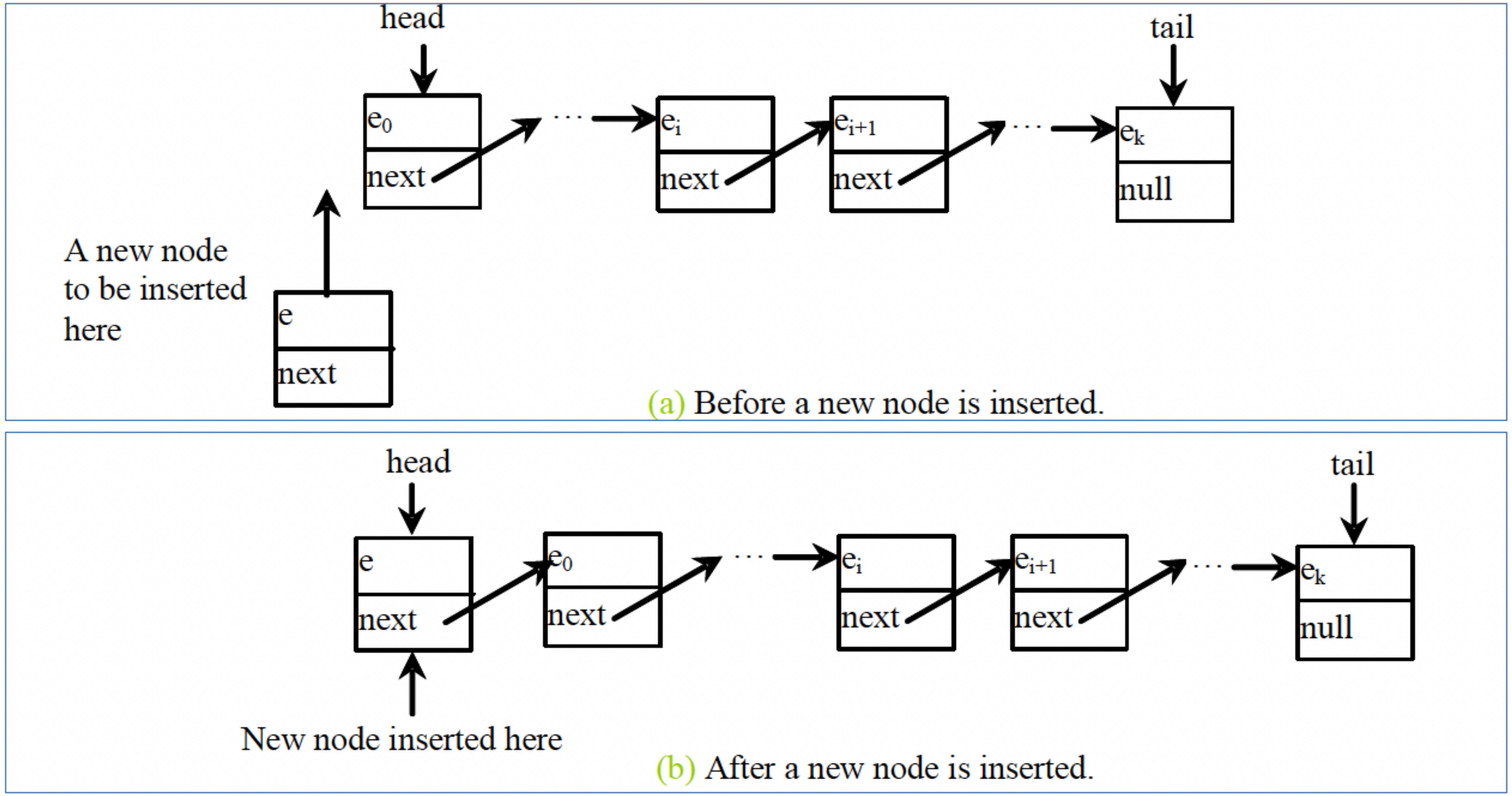
Creates a default linked list.
Creates a linked list from an array of elements.
Adds an element to the head of the list.
Adds an element to the tail of the list.
Returns the first element in the list.
Returns the last element in the list.
Removes the first element from the list.
Removes the last element from the list.

Implementing addFirst(E e)



```java
public void addFirst(E e) {
  Node<E> newNode = new Node<>(e);
  newNode.next = head; //create pointer to current head
  head = newNode; //new node created & assigned to new head
  size++; //increase size
  if (tail == null)  //no node exists
    tail = head;
}
```

Implementing addFirst(E e)



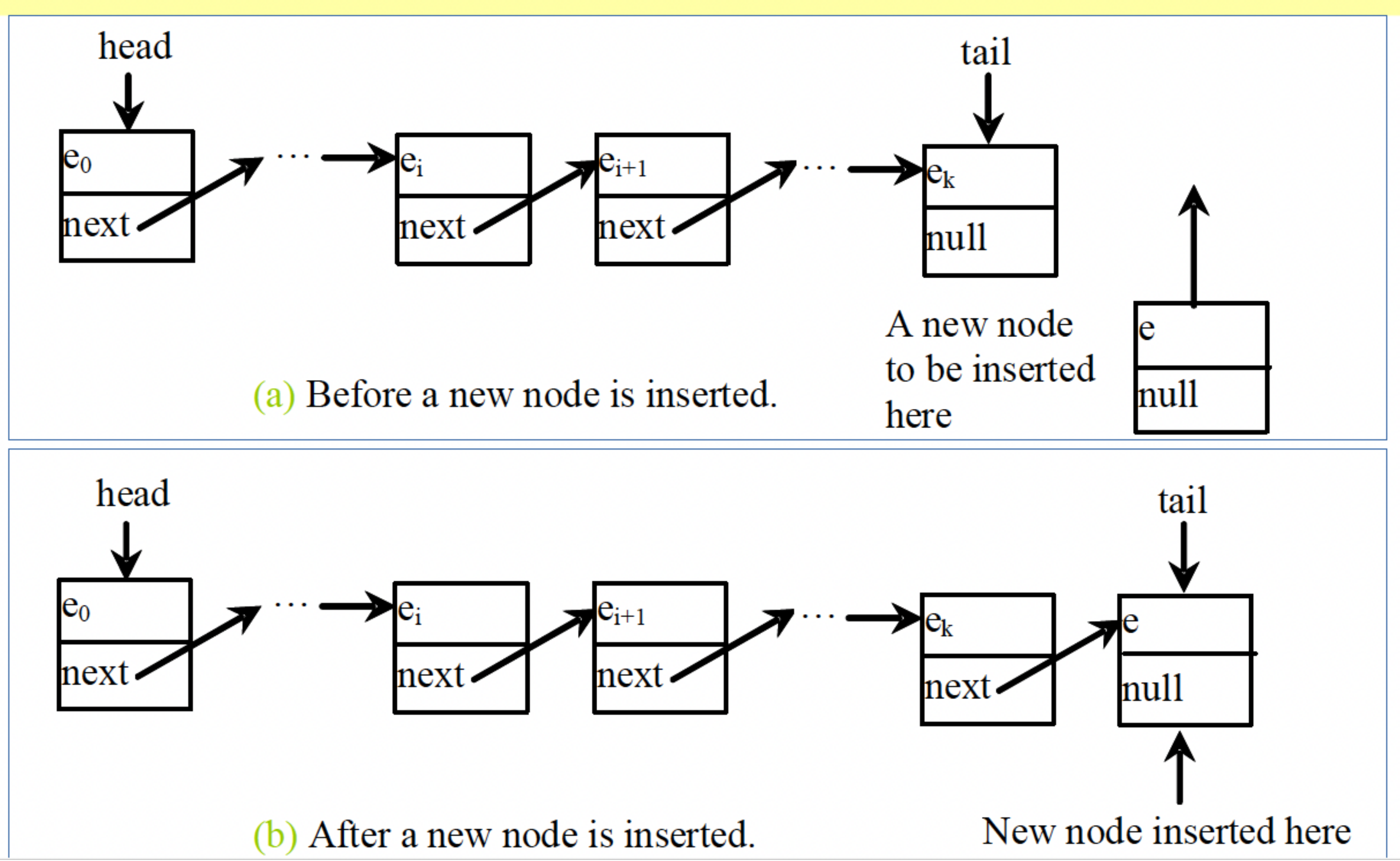
(a) Before a new node is inserted.

(b) After a new node is inserted.

Implementing addLast(E e)

```
public void addLast(E e) {
  if (tail == null) {  //no node exist
    head = tail = new Node<>(e);
  }
  else {
    tail.next = new Node<>(e);  //tail.next point to new Node
    tail = tail.next;  //new tail updated from tail.next
  }
  size++;
}
```

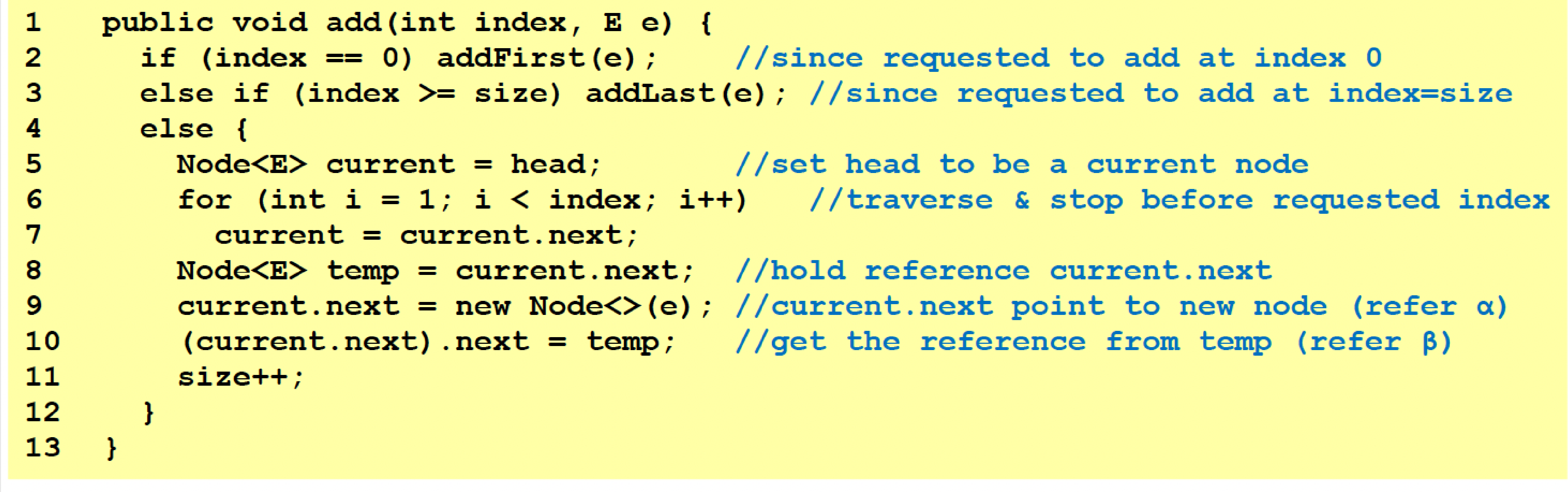
Implementing addLast(E e)



(a) Before a new node is inserted.

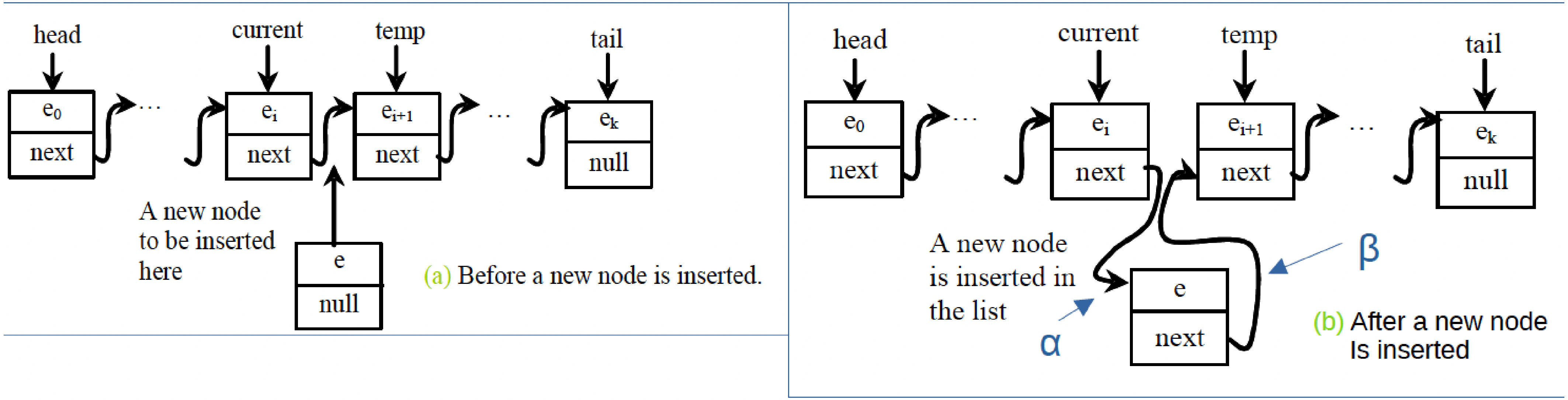
(b) After a new node is inserted.

Implementing add(int index, E e)



```java
public void add(int index, E e) {
  if (index == 0) addFirst(e);    //since requested to add at index 0
  else if (index >= size) addLast(e); //since requested to add at index=size
  else {
    Node<E> current = head;       //set head to be a current node
    for (int i = 1; i < index; i++)   //traverse & stop before requested index
      current = current.next;
    Node<E> temp = current.next;  //hold reference current.next
    current.next = new Node<>(e); //current.next point to new node (refer α)
    (current.next).next = temp;   //get the reference from temp (refer β)
    size++;
  }
}
```

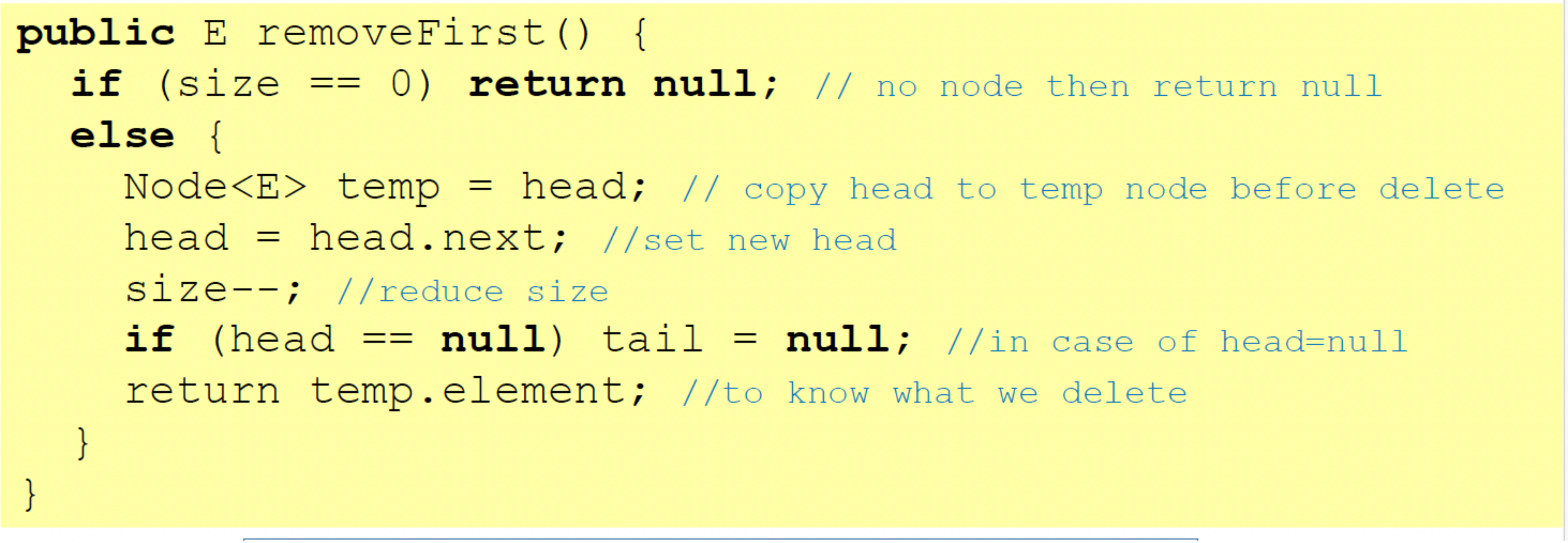
Implementing add(int index, E e)



(a) Before a new node is inserted.

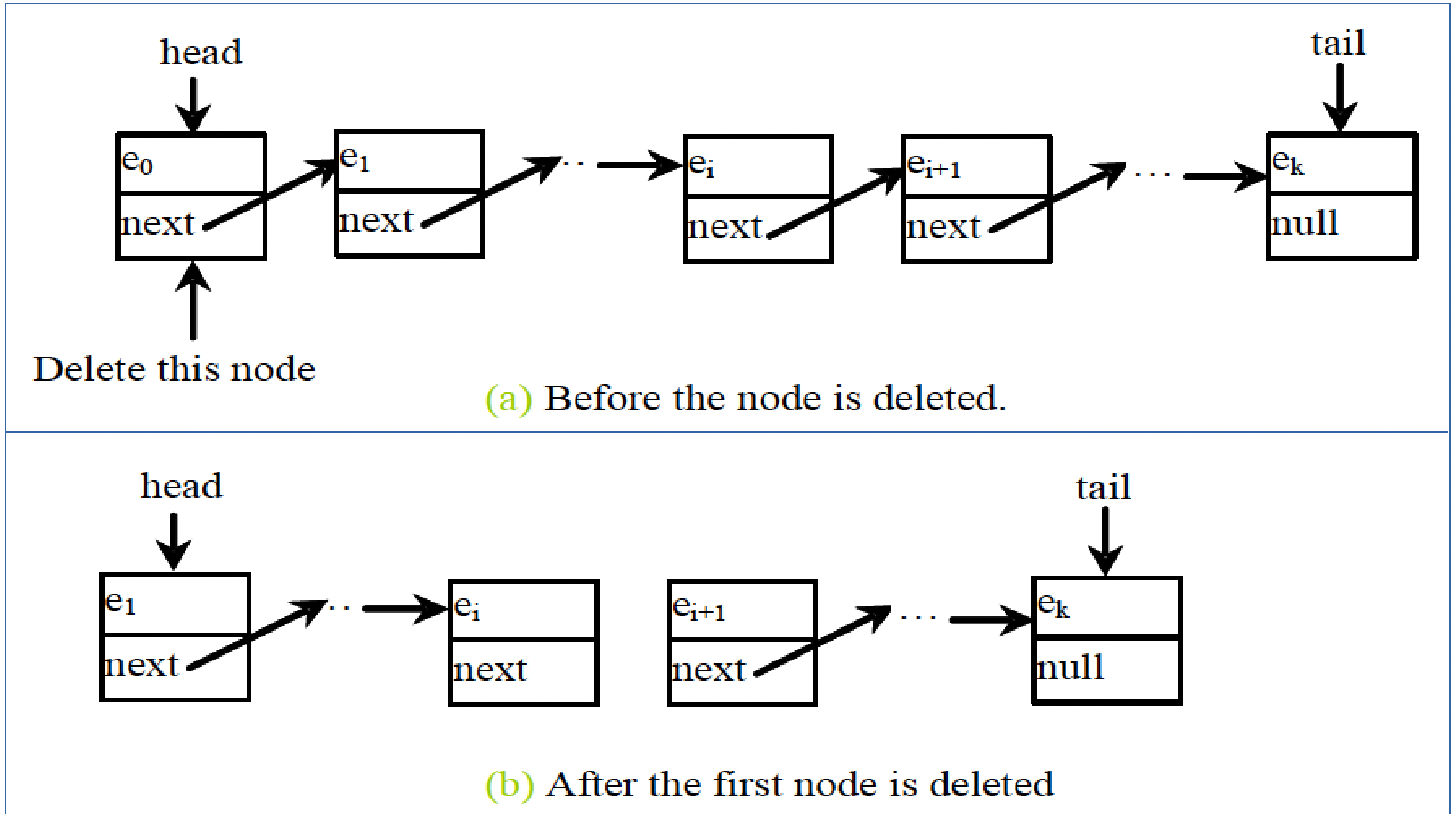
(b) After a new node
Is inserted

Implementing removeFirst()



```
public E removeFirst() {
  if (size == 0) return null; // no node then return null
  else {
    Node<E> temp = head; // copy head to temp node before delete
    head = head.next; //set new head
    size--; //reduce size
    if (head == null) tail = null; //in case of head=null
    return temp.element; //to know what we delete
  }
}
```

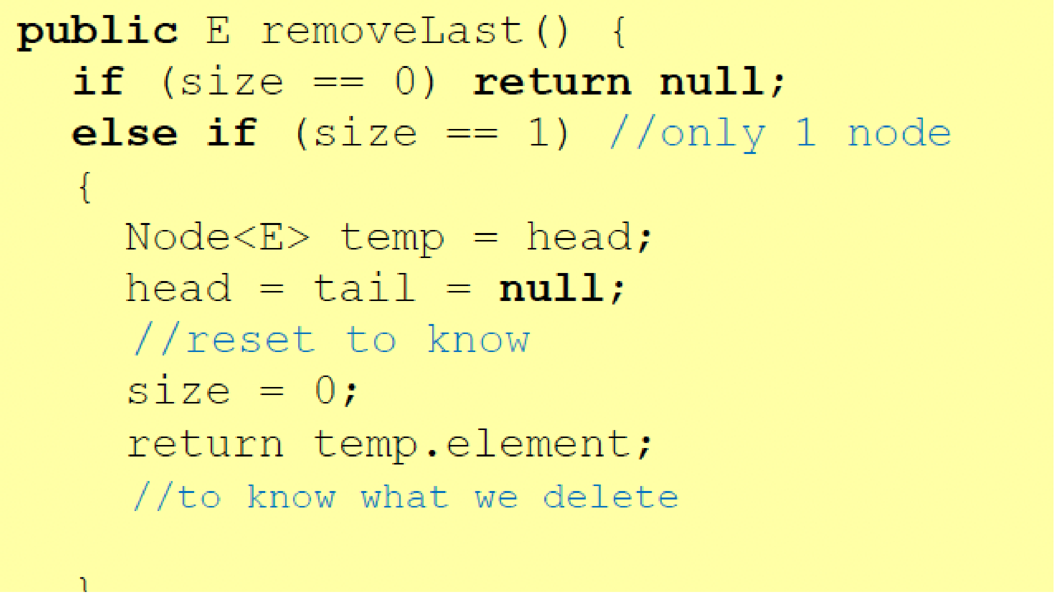
Implementing removeFirst()



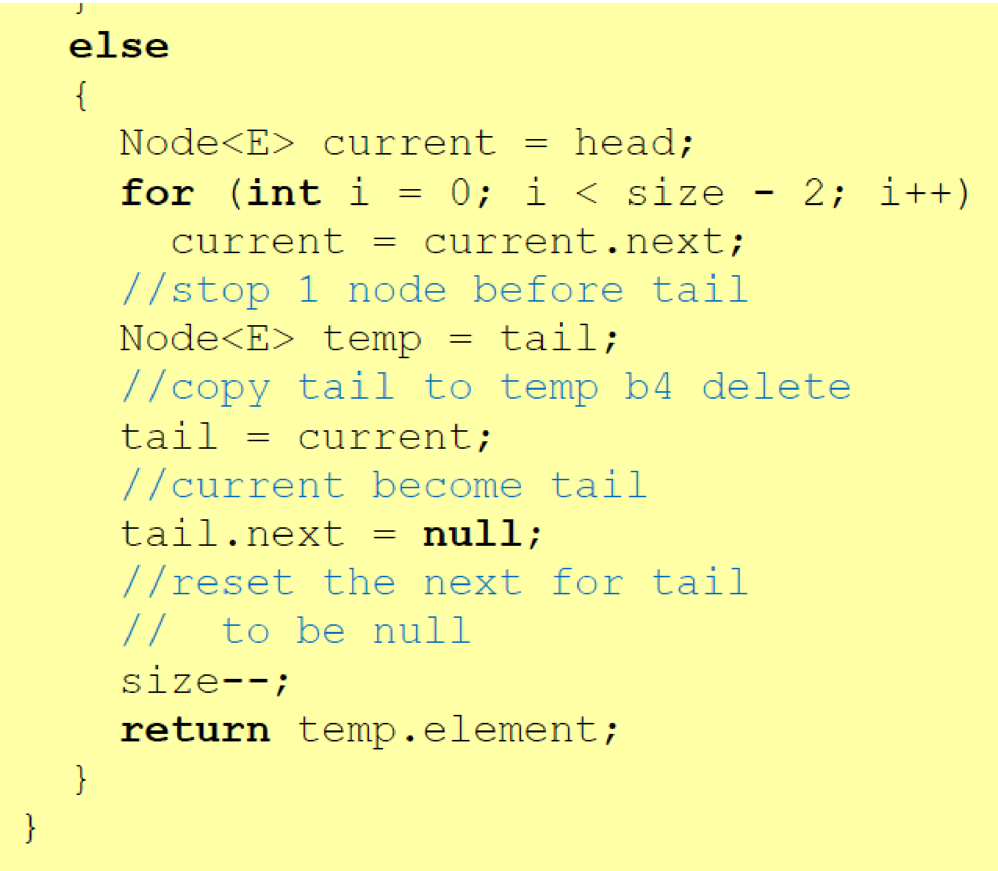
(a) Before the node is deleted.

(b) After the first node is deleted

Implementing removeLast()

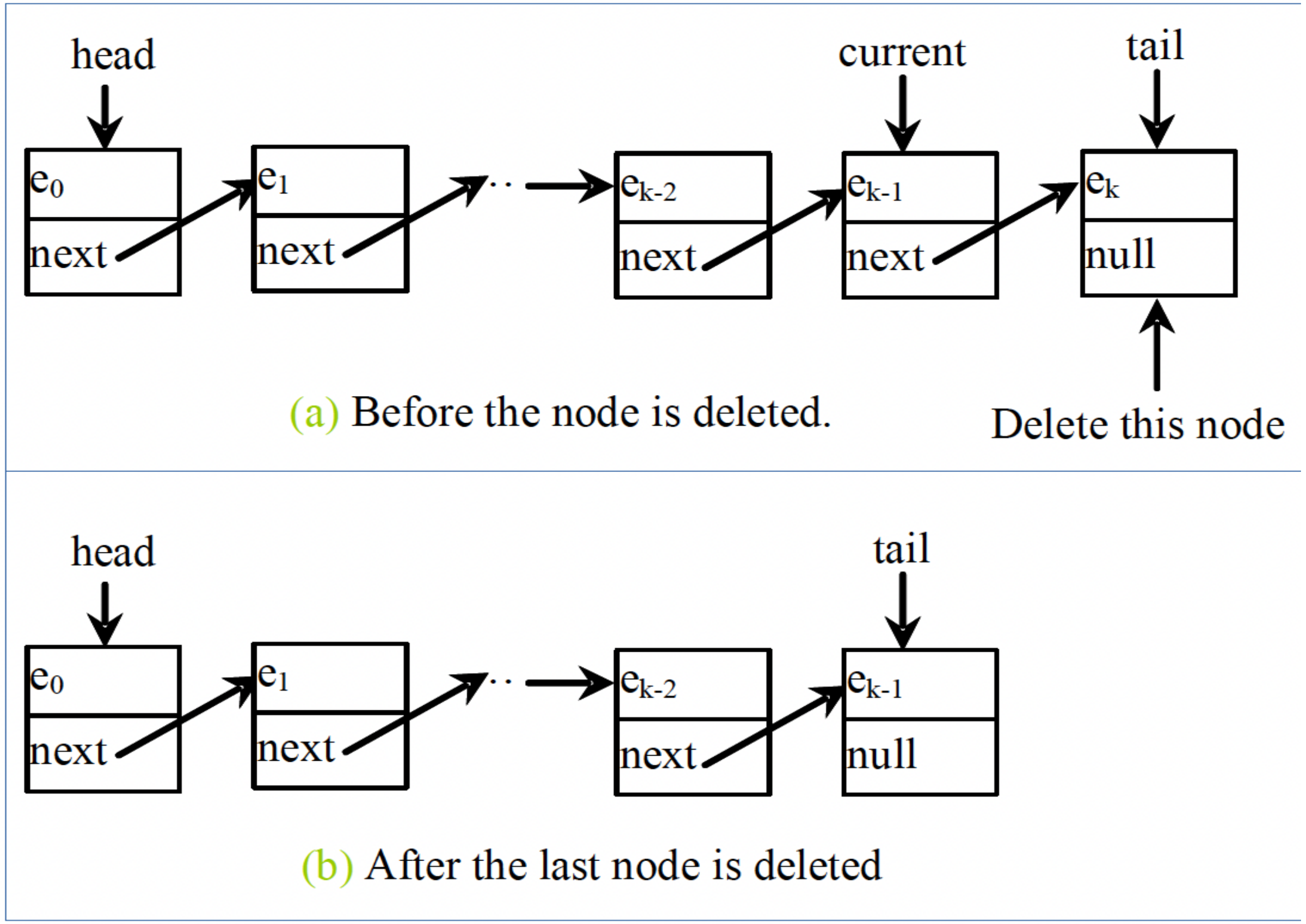


```
public E removeLast() {
  if (size == 0) return null;
  else if (size == 1) //only 1 node
  {
    Node<E> temp = head;
    head = tail = null;
    //reset to know
    size = 0;
    return temp.element;
    //to know what we delete

  }
```



```
  }
  else
  {
    Node<E> current = head;
    for (int i = 0; i < size - 2; i++)
      current = current.next;
    //stop 1 node before tail
    Node<E> temp = tail;
    //copy tail to temp b4 delete
    tail = current;
    //current become tail
    tail.next = null;
    //reset the next for tail
    //  to be null
    size--;
    return temp.element;
  }
}
```

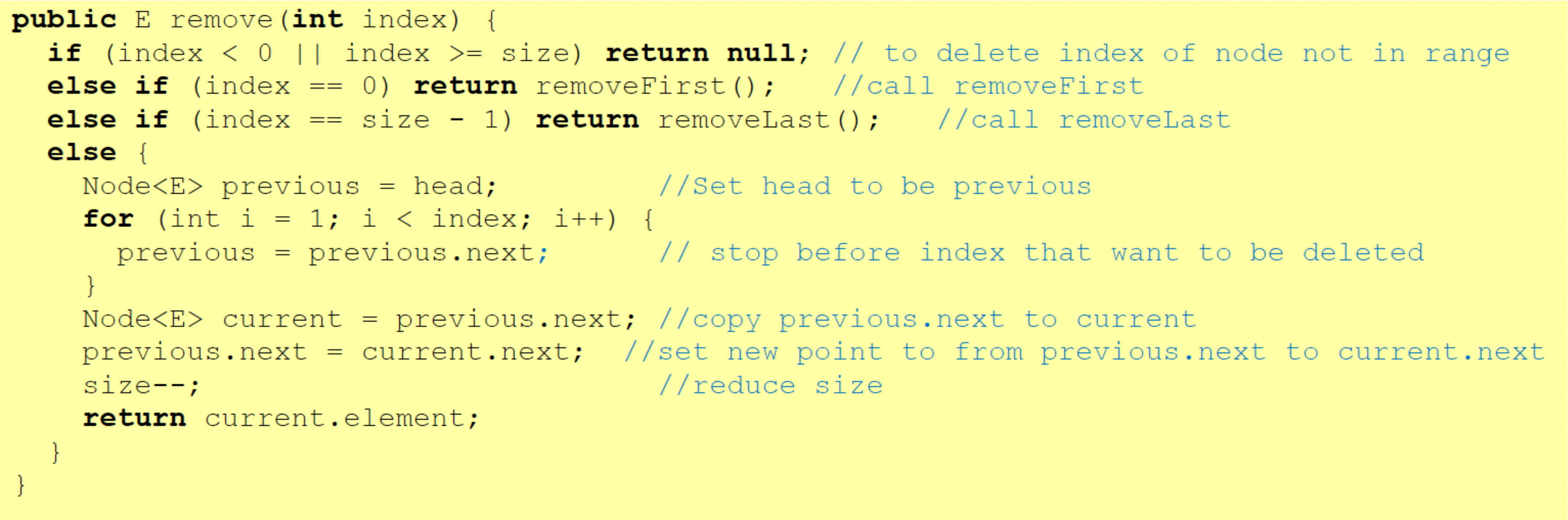
Implementing removeLast()



(a) Before the node is deleted.

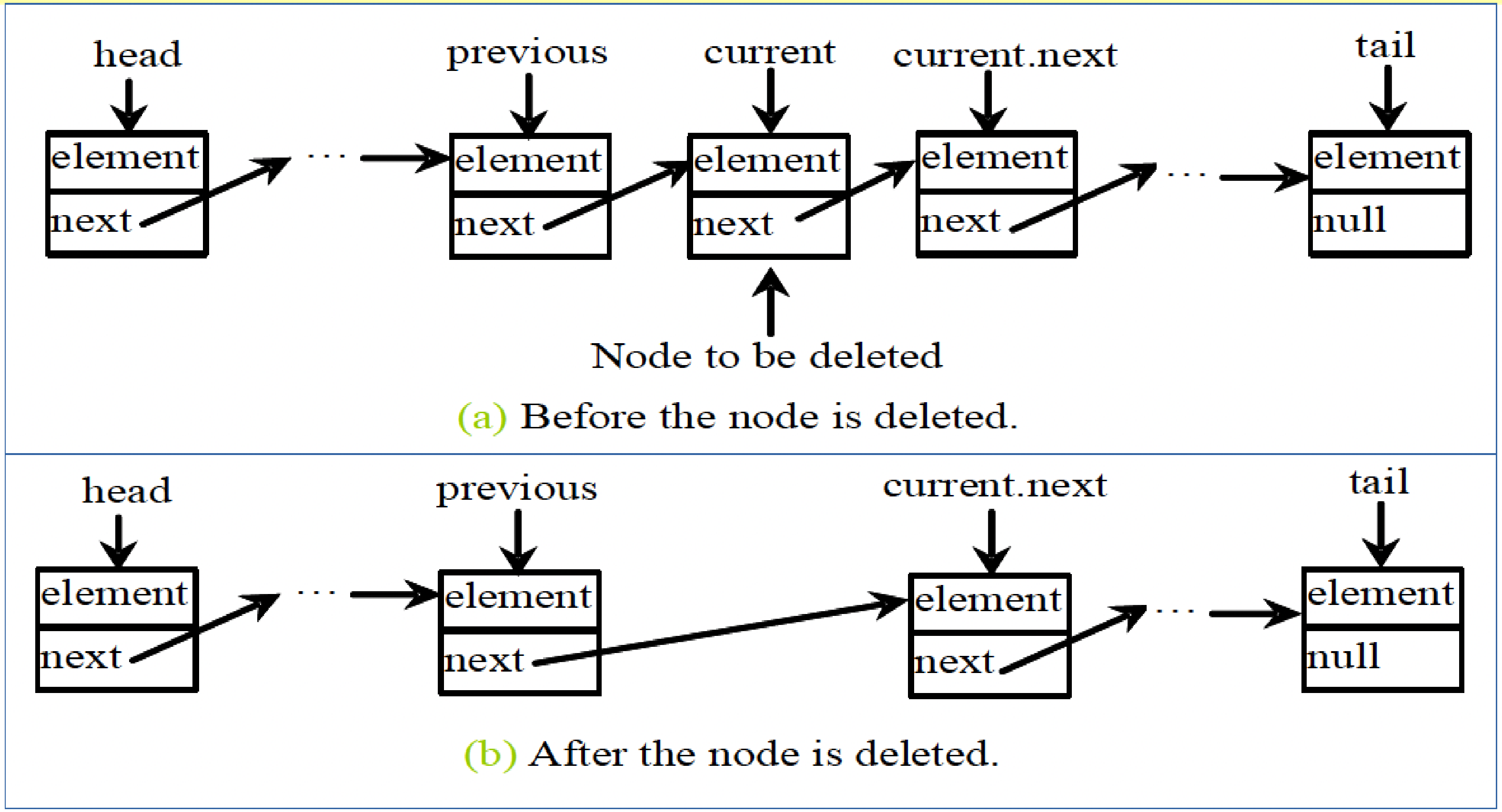
(b) After the last node is deleted

Implementing remove(int index)



```
public E remove(int index) {
  if (index < 0 || index >= size) return null; // to delete index of node not in range
  else if (index == 0) return removeFirst();   //call removeFirst
  else if (index == size - 1) return removeLast();   //call removeLast
  else {
    Node<E> previous = head;            //Set head to be previous
    for (int i = 1; i < index; i++) {
      previous = previous.next;         // stop before index that want to be deleted
    }
    Node<E> current = previous.next; //copy previous.next to current
    previous.next = current.next;  //set new point to from previous.next to current.next
    size--;                             //reduce size
    return current.element;
  }
}
```

Implementing remove(int index)

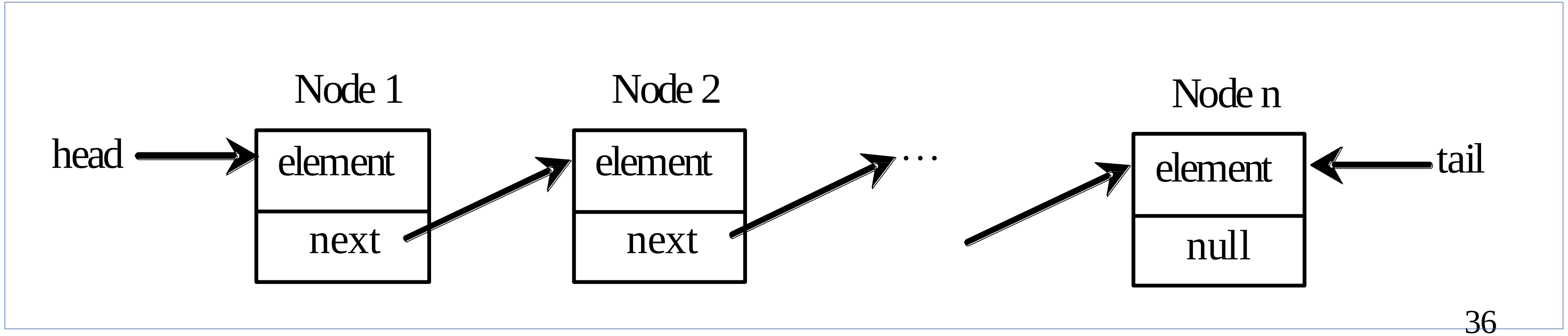


(a) Before the node is deleted.

(b) After the node is deleted.

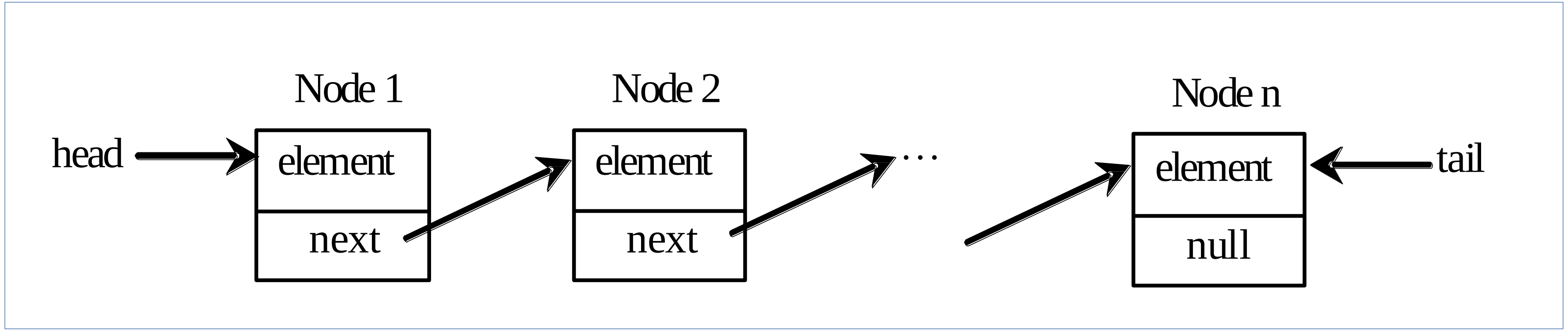
Singly Linked List

- What you have seen so far is singly linked list (contains a pointer to the list's first node, and each node contains a pointer to the next node sequentially.)
- Is not a direct access structure. It must be accessed sequentially by moving forward one node at a time



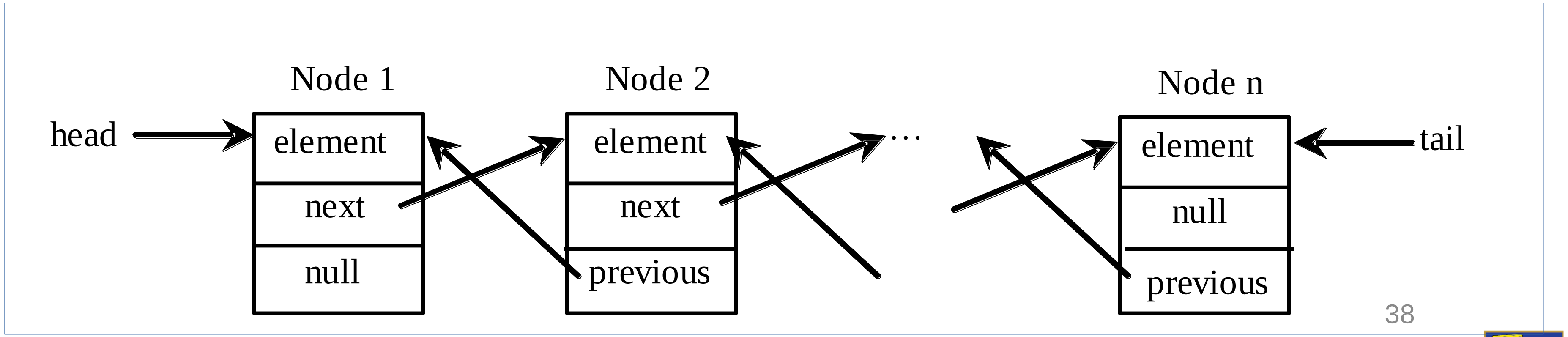
Circular Linked List

- 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**.



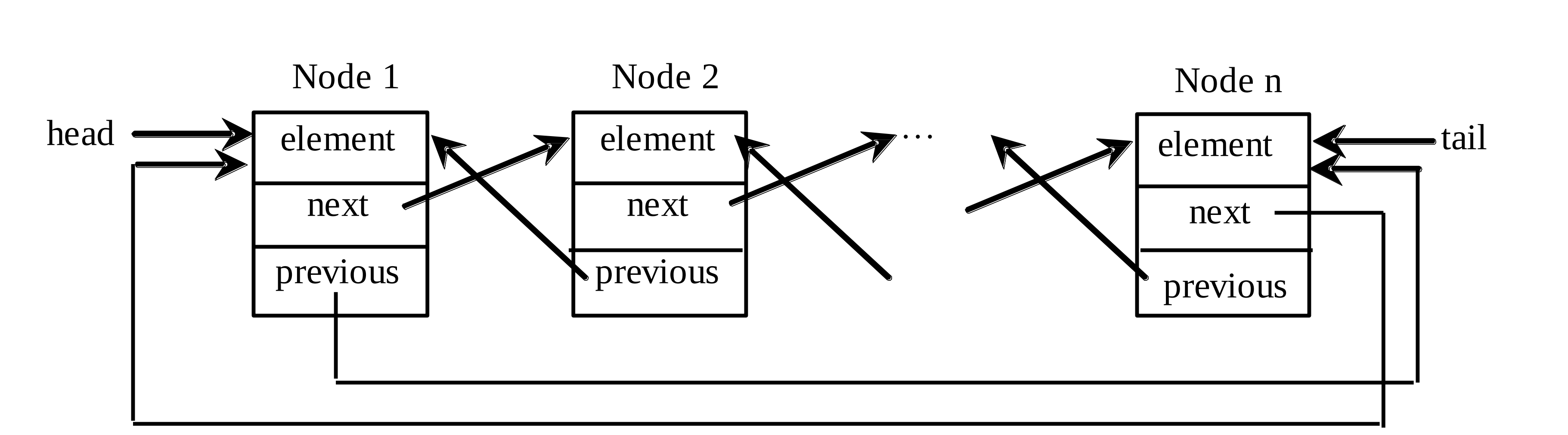
Doubly Linked List

- 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.



Circular Doubly Linked List

- 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.



Part 2: Doubly Linked List

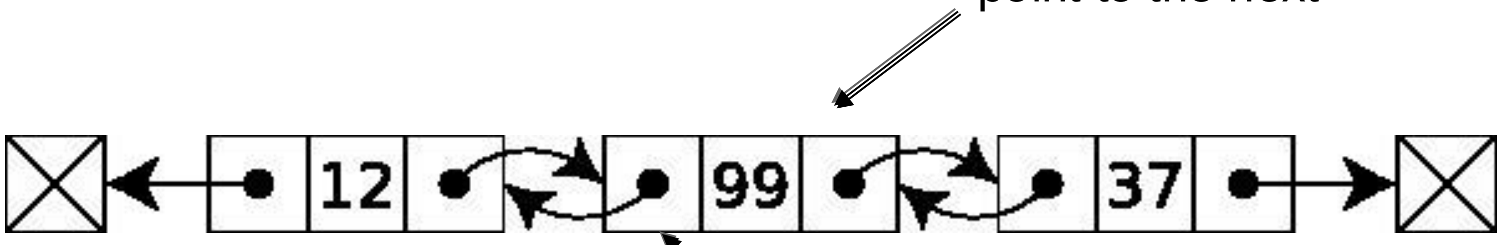
Doubly Linked List

- A doubly-linked list (also called two-way linked list) is a linked data structure that consists of a set of sequentially linked records called nodes.
- Each node contains two fields, called links(**pointer**), that are references to the ***previous*** and to the ***next*** node in the sequence of nodes.

Doubly Linked List

- The beginning and ending nodes previous and next links, respectively, point to some kind of terminator, typically a sentinel node or null, to facilitate traversal of the list.
- If there is only one sentinel node, then the list is circularly linked via the sentinel node.
- It can be conceptualized as ***two singly linked lists*** formed from the same data items, but in opposite sequential orders.

Pictorial View of Doubly Linked List



point to the next

point to the previous

Pictorial View of Doubly Linked List

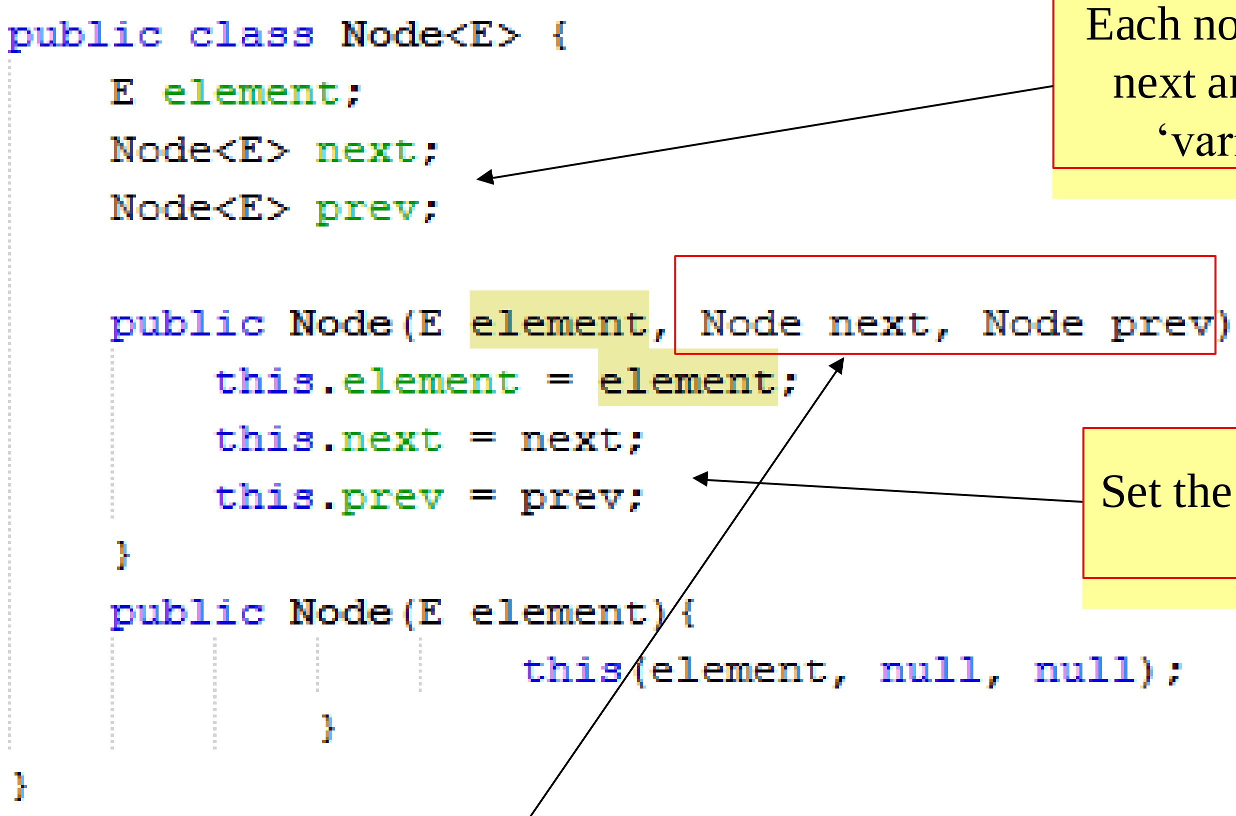
- The two node links allow traversal of the list in either direction.
- While adding or removing a node in a doubly-linked list requires changing more links than the same operations on a singly linked list, the operations are simpler and potentially more efficient, because there is no need to keep track of the previous node during traversal or no need to traverse the list to find the previous node, so that its link can be modified.

Disadvantages

❿ Each node requires an extra pointer, requiring more space

❿ The insertion or deletion of a node takes a bit longer (more pointer operations)

The Node Class for Doubly Linked List



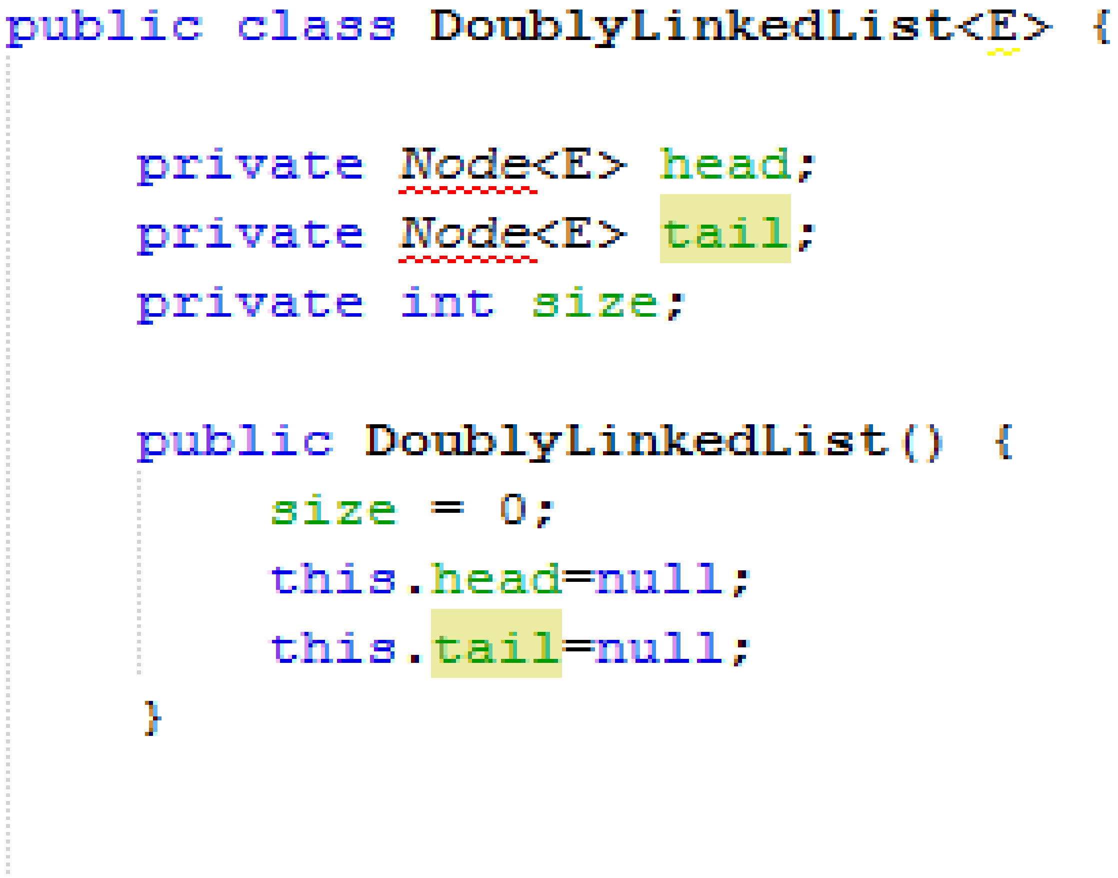
```
public class Node<E> {
    E element;
    Node<E> next;
    Node<E> prev;

    public Node(E element, Node next, Node prev) {
        this.element = element;
        this.next = next;
        this.prev = prev;
    }
    public Node(E element){
            this(element, null, null);
        }
}
```

Each node consist of 2 pointers next and prev also known as ‘variable of type object’

Set the value and the pointers to the nodes

These are the nodes

Class Definition for DoublyLinkedList

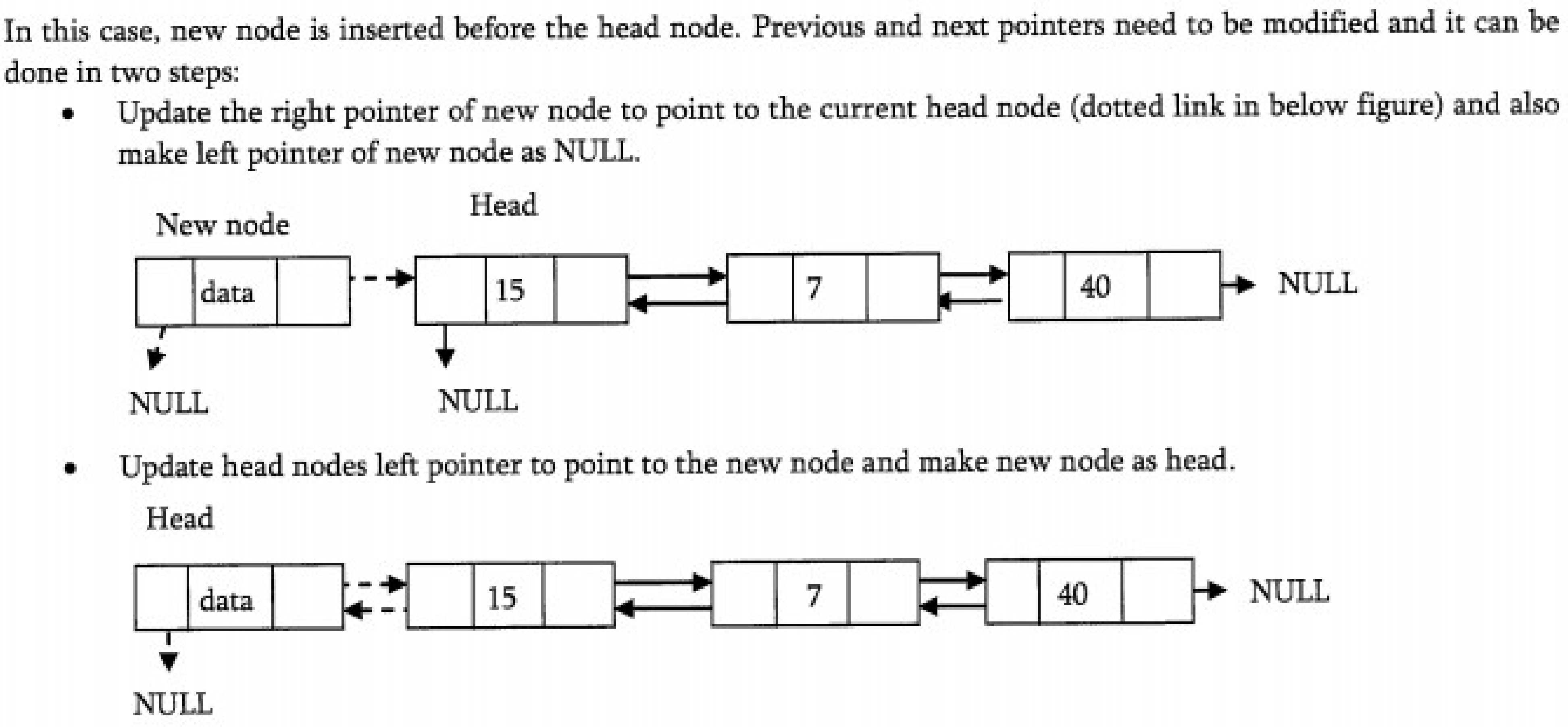


```
public class DoublyLinkedList<E> {

    private Node<E> head;
    private Node<E> tail;
    private int size;

    public DoublyLinkedList() {
        size = 0;
        this.head=null;
        this.tail=null;
    }
```

Doubly Linked List Insertion

- Insertion into a doubly-linked list has three cases (same as singly linked list)
  - Inserting a new node before the head
    - addFirst(E element)
  - Inserting a new node after the tail
    - addLast(E element)
  - Inserting a new node a the middle of the list
    - add(int index, E element)

Inserting a Node at the Beginning



In this case, new node is inserted before the head node. Previous and next pointers need to be modified and it can be done in two steps:

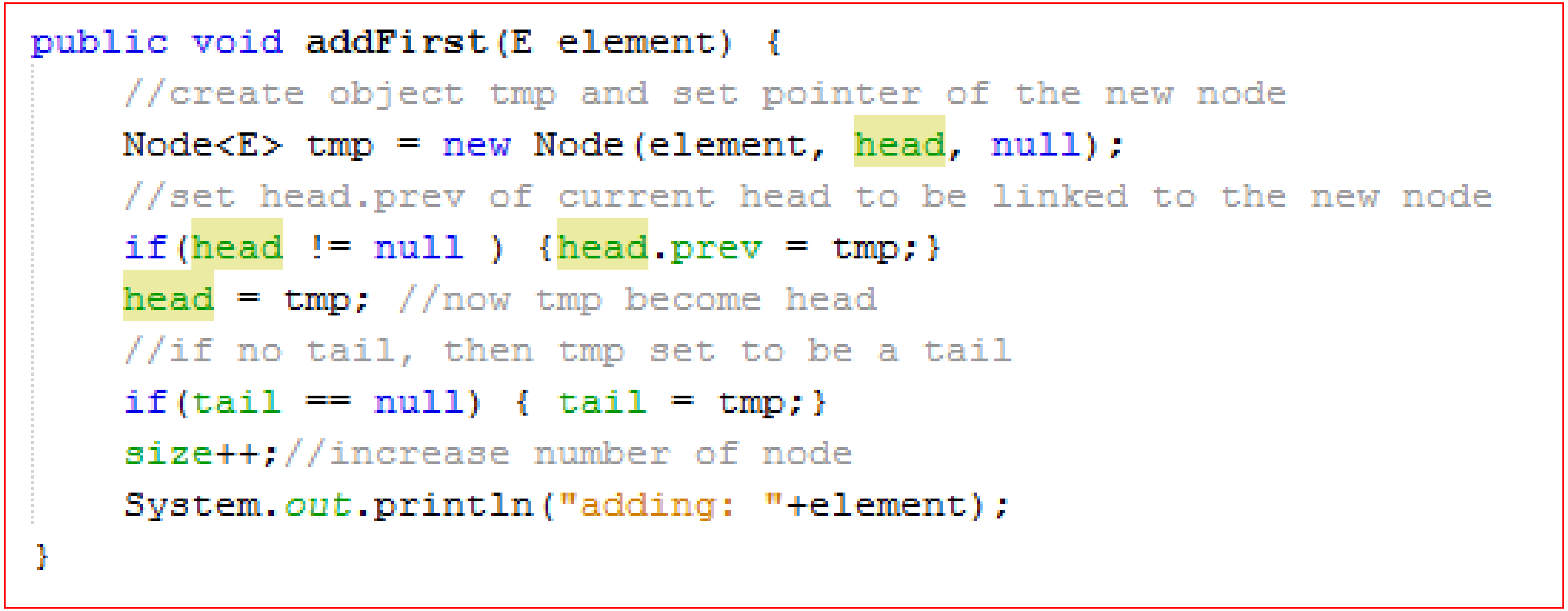
- Update the right pointer of new node to point to the current head node (dotted link in below figure) and also make left pointer of new node as NULL.

New node
Head
data
15
7
40
NULL
NULL
NULL

- Update head nodes left pointer to point to the new node and make new node as head.

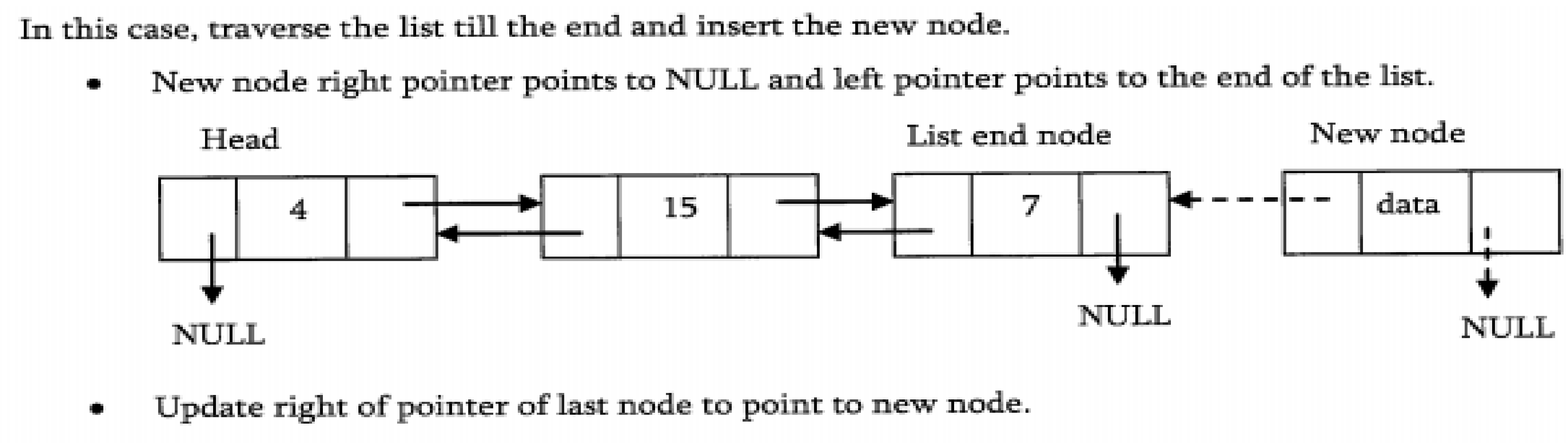
Head
data
15
7
40
NULL
NULL

Inserting a Node at the Beginning



```java
public void addFirst(E element) {
    //create object tmp and set pointer of the new node
    Node<E> tmp = new Node(element, head, null);
    //set head.prev of current head to be linked to the new node
    if(head != null ) {head.prev = tmp;}
    head = tmp; //now tmp become head
    //if no tail, then tmp set to be a tail
    if(tail == null) { tail = tmp;}
    size++;//increase number of node
    System.out.println("adding: "+element);
}
```

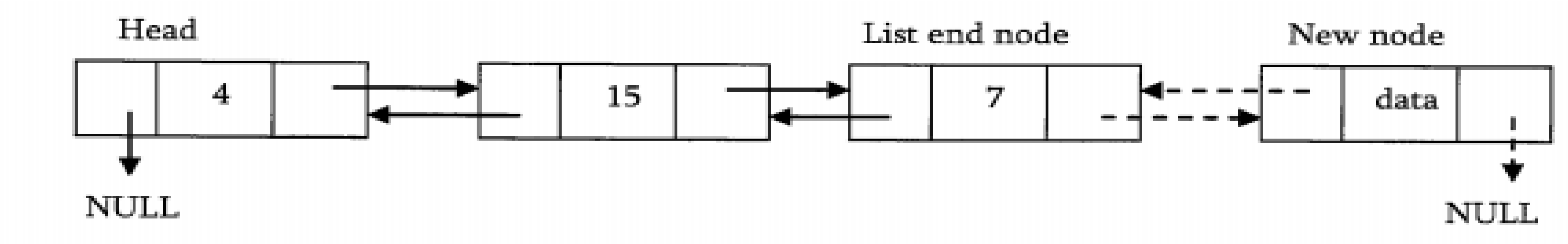
Inserting a Node at the Ending



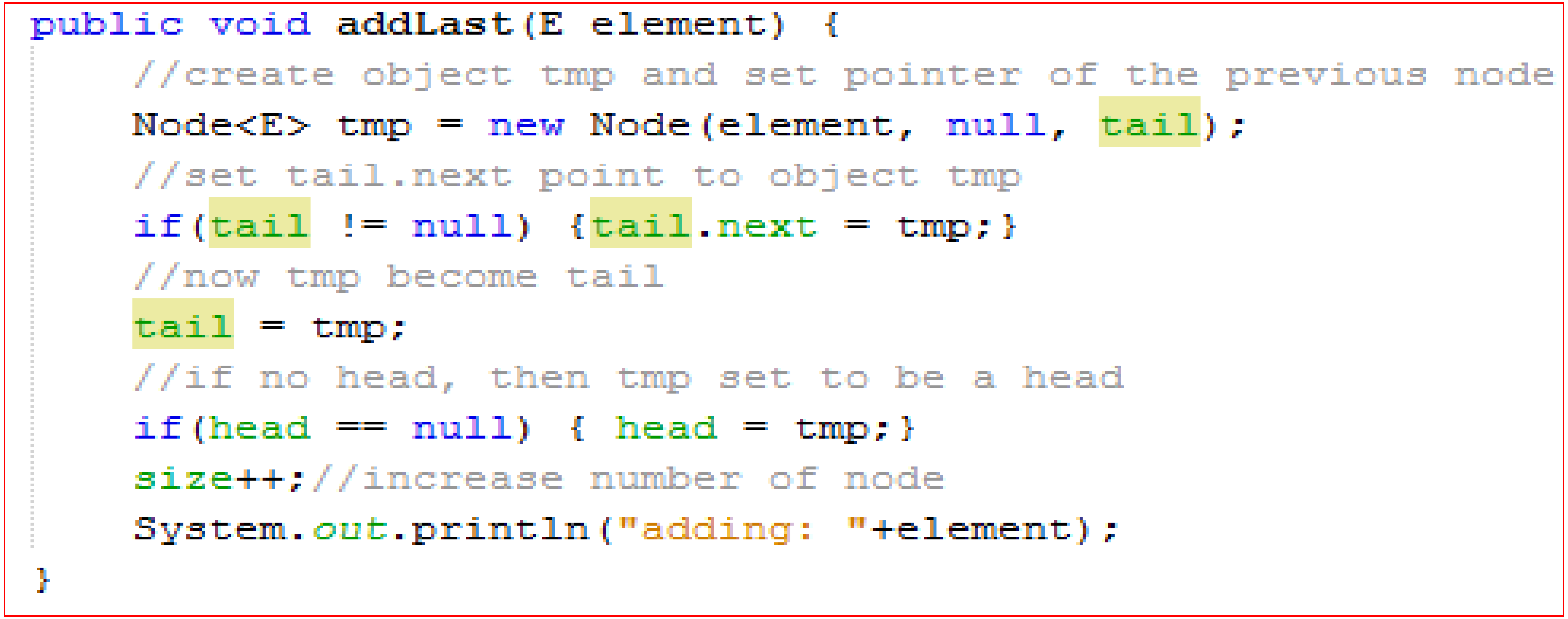
In this case, traverse the list till the end and insert the new node.

- New node right pointer points to NULL and left pointer points to the end of the list.

- Update right of pointer of last node to point to new node.

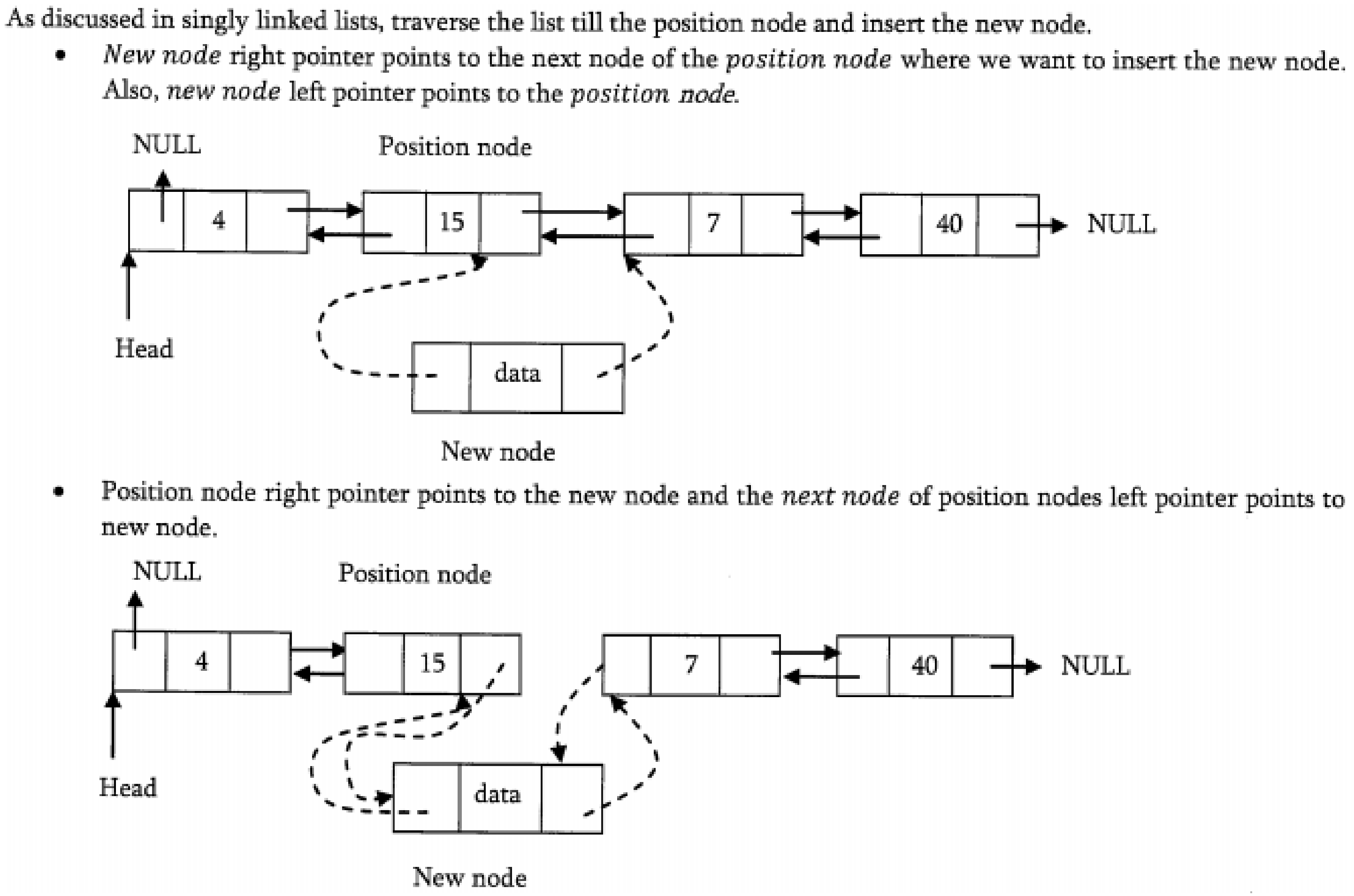


Inserting a Node at the Ending



```
public void addLast(E element) {
    //create object tmp and set pointer of the previous node
    Node<E> tmp = new Node(element, null, tail);
    //set tail.next point to object tmp
    if(tail != null) {tail.next = tmp;}
    //now tmp become tail
    tail = tmp;
    //if no head, then tmp set to be a head
    if(head == null) { head = tmp;}
    size++;//increase number of node
    System.out.println("adding: "+element);
}
```

Inserting a Node in the Middle



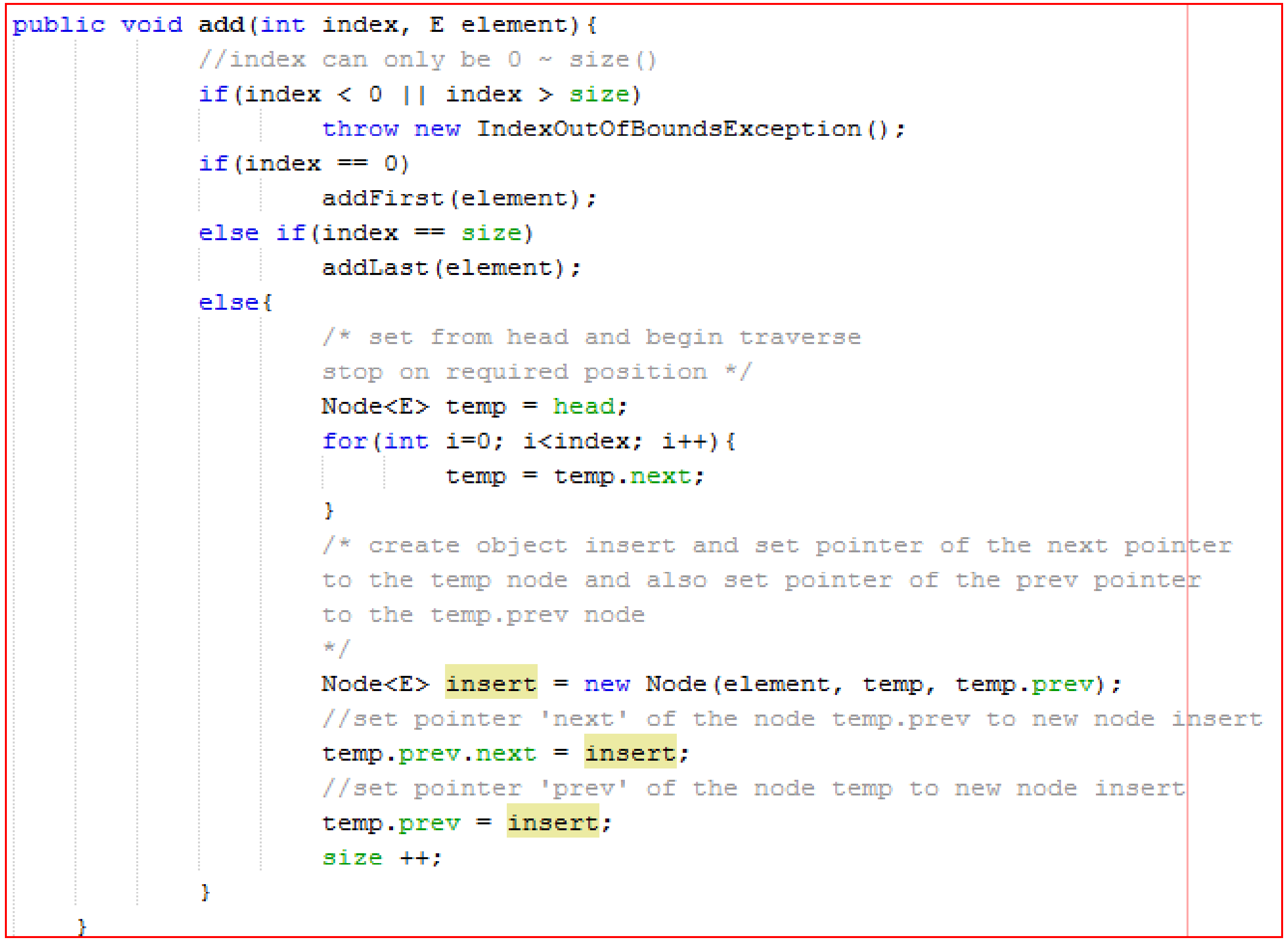
As discussed in singly linked lists, traverse the list till the position node and insert the new node.

- *New node* right pointer points to the next node of the *position node* where we want to insert the new node. Also, *new node* left pointer points to the *position node*.

NULL
Position node
4
15
7
40
NULL
Head
data
New node

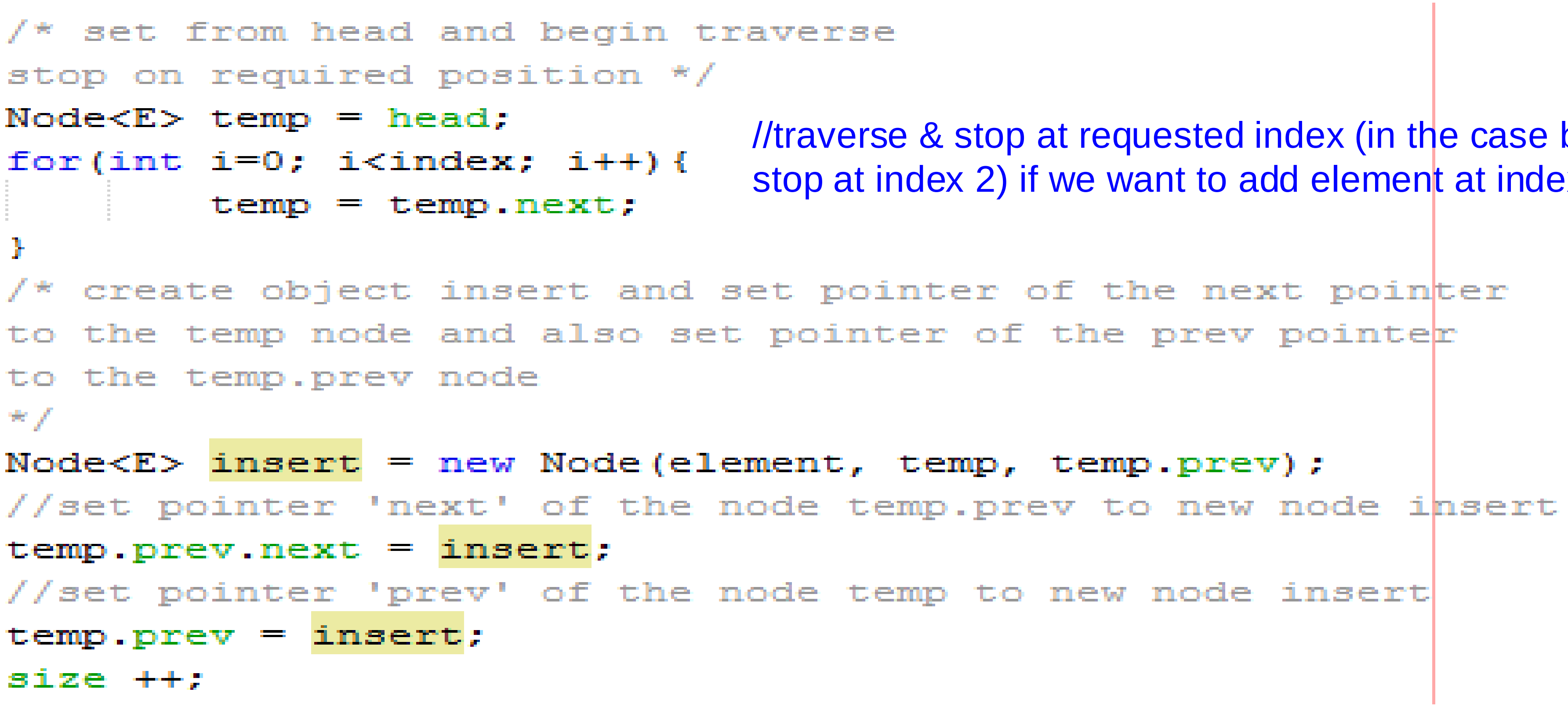
- Position node right pointer points to the new node and the *next node* of position nodes left pointer points to new node.

Inserting a Node in the Middle



```
public void add(int index, E element){
            //index can only be 0 ~ size()
            if(index < 0 || index > size)
                    throw new IndexOutOfBoundsException();
            if(index == 0)
                    addFirst(element);
            else if(index == size)
                    addLast(element);
            else{
                    /* set from head and begin traverse
                    stop on required position */
                    Node<E> temp = head;
                    for(int i=0; i<index; i++){
                            temp = temp.next;
                    }
                    /* create object insert and set pointer of the next pointer
                    to the temp node and also set pointer of the prev pointer
                    to the temp.prev node
                    */
                    Node<E> insert = new Node(element, temp, temp.prev);
                    //set pointer 'next' of the node temp.prev to new node insert
                    temp.prev.next = insert;
                    //set pointer 'prev' of the node temp to new node insert
                    temp.prev = insert;
                    size ++;
            }
    }
```

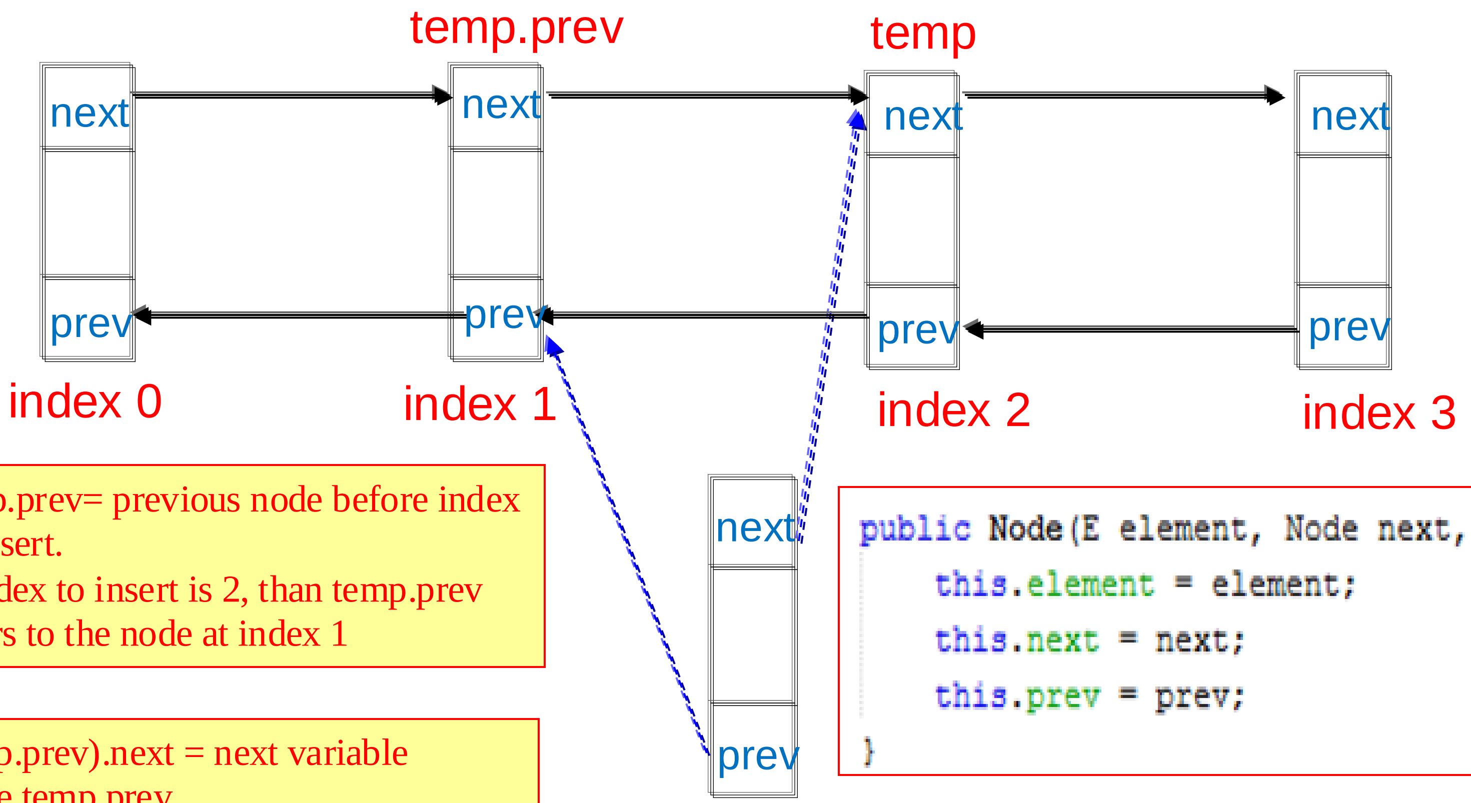
Inserting a Node in the Middle



```
/* set from head and begin traverse
stop on required position */
Node<E> temp = head;
for(int i=0; i<index; i++){
        temp = temp.next;
}
/* create object insert and set pointer of the next pointer
to the temp node and also set pointer of the prev pointer
to the temp.prev node
*/
Node<E> insert = new Node(element, temp, temp.prev);
//set pointer 'next' of the node temp.prev to new node insert
temp.prev.next = insert;
//set pointer 'prev' of the node temp to new node insert
temp.prev = insert;
size ++;
```

//traverse & stop at requested index (in the case below stop at index 2) if we want to add element at index 2.

Inserting a Node in the Middle



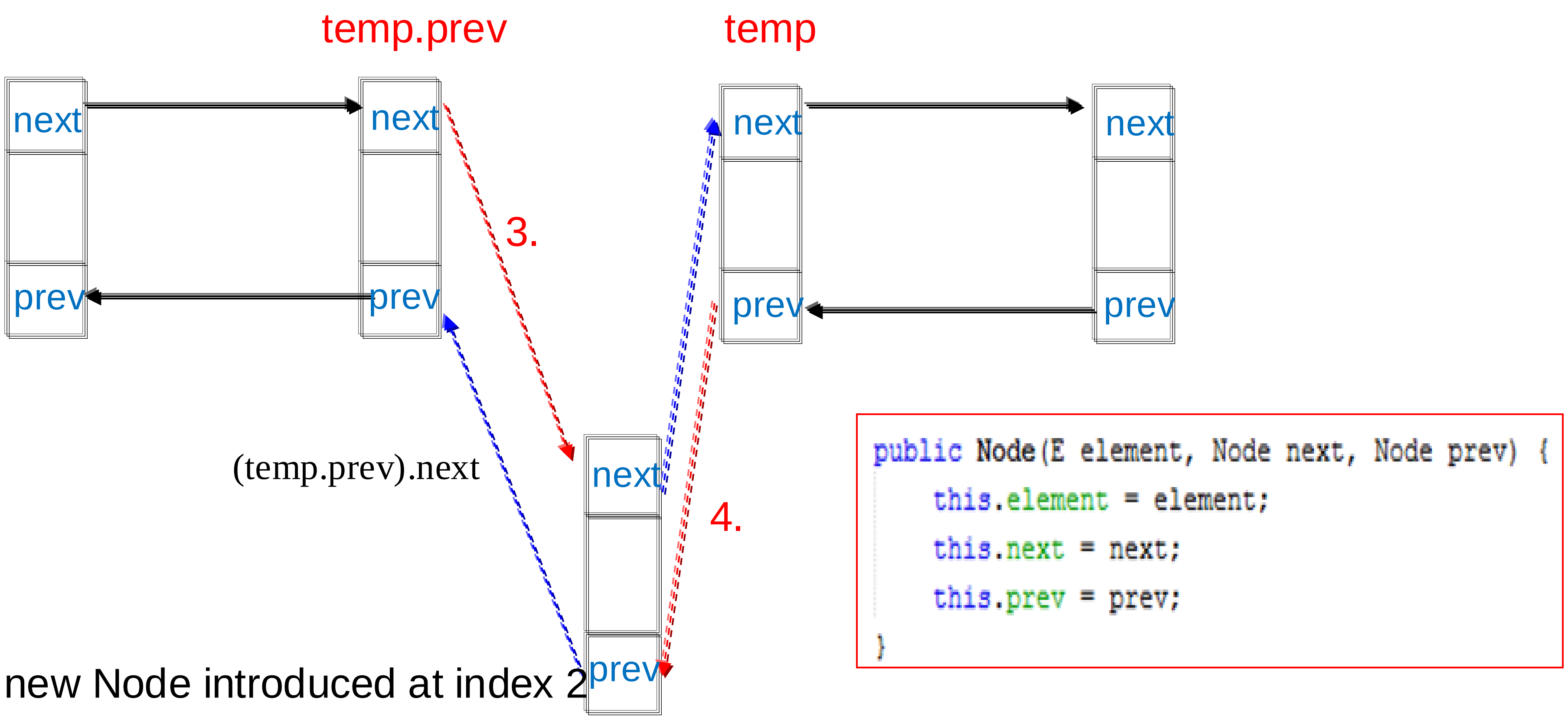
temp.prev= previous node before index to insert.
if index to insert is 2, than temp.prev refers to the node at index 1

(temp.prev).next = next variable of the temp.prev

```
public Node(E element, Node next, Node prev) {
    this.element = element;
    this.next = next;
    this.prev = prev;
}
```

new Node introduced at index 2

Inserting a Node in the Middle

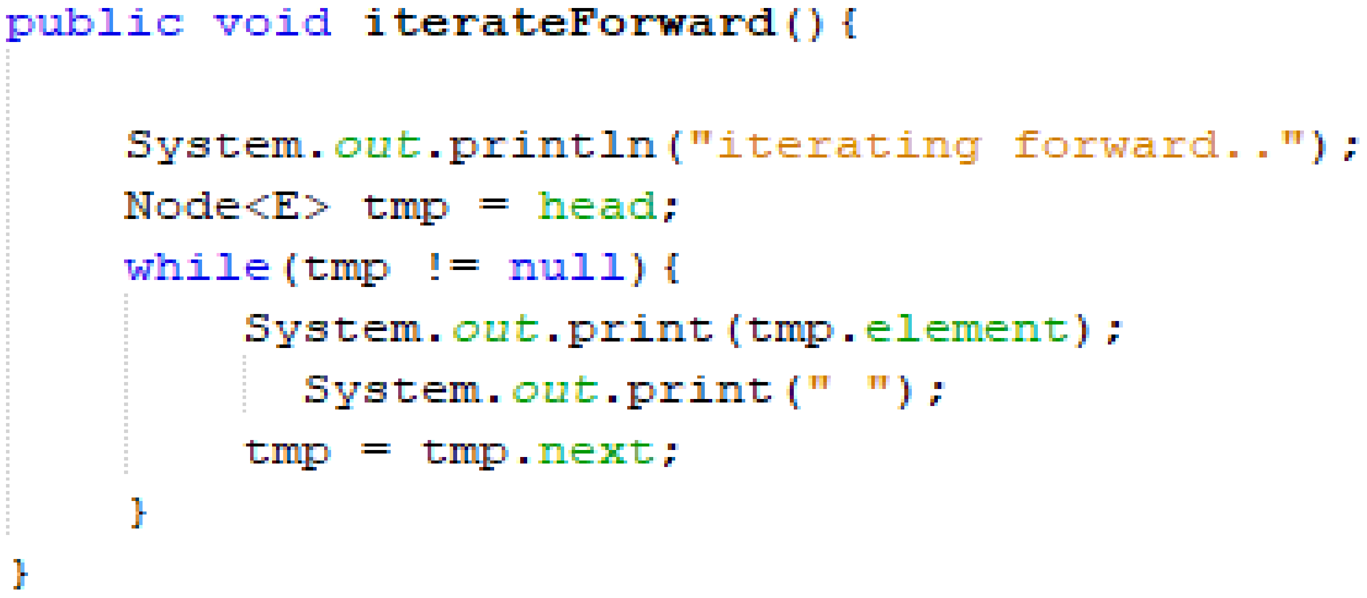


new Node introduced at index 2

```
public Node(E element, Node next, Node prev) {
    this.element = element;
    this.next = next;
    this.prev = prev;
}
```

Traversing Forward

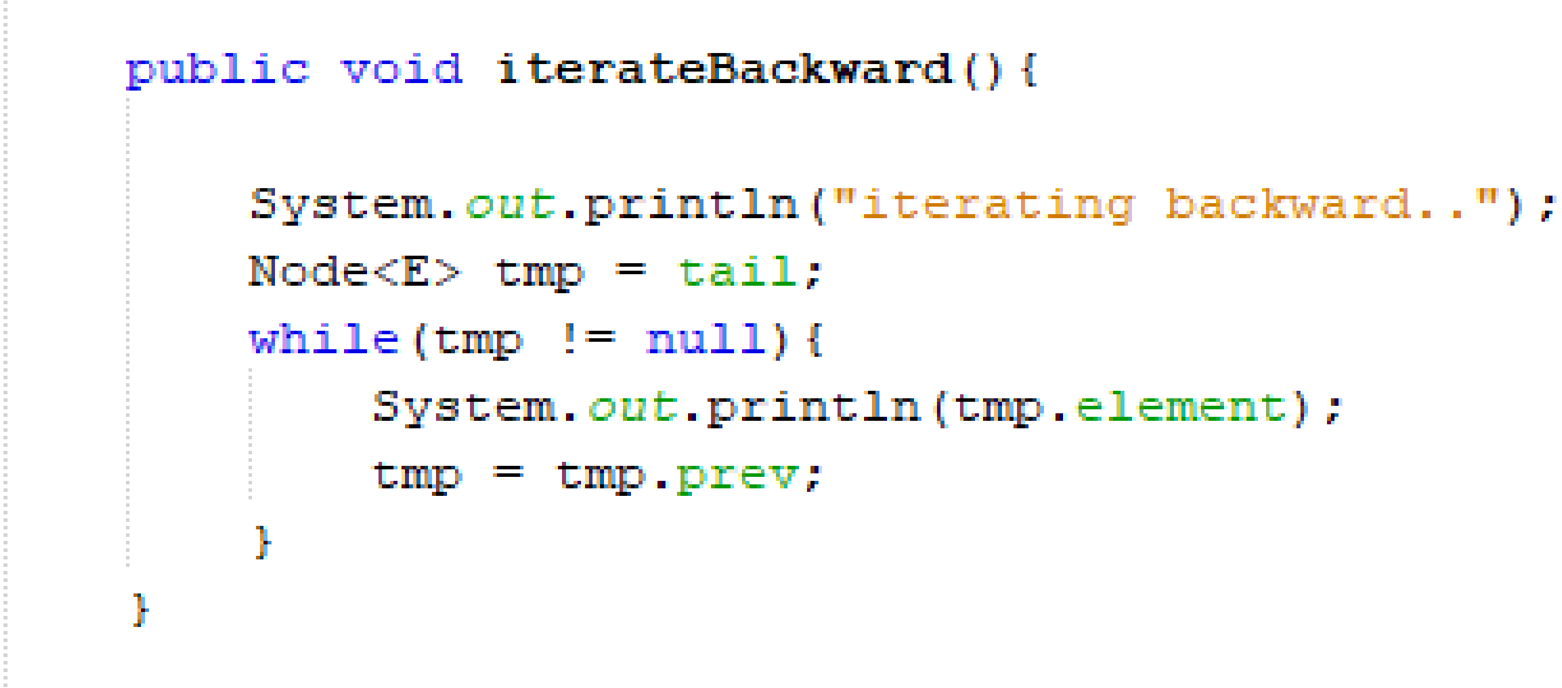
- Similar as Singly Linked List



```java
public void iterateForward(){

    System.out.println("iterating forward..");
    Node<E> tmp = head;
    while(tmp != null){
        System.out.print(tmp.element);
          System.out.print(" ");
        tmp = tmp.next;
    }
}
```

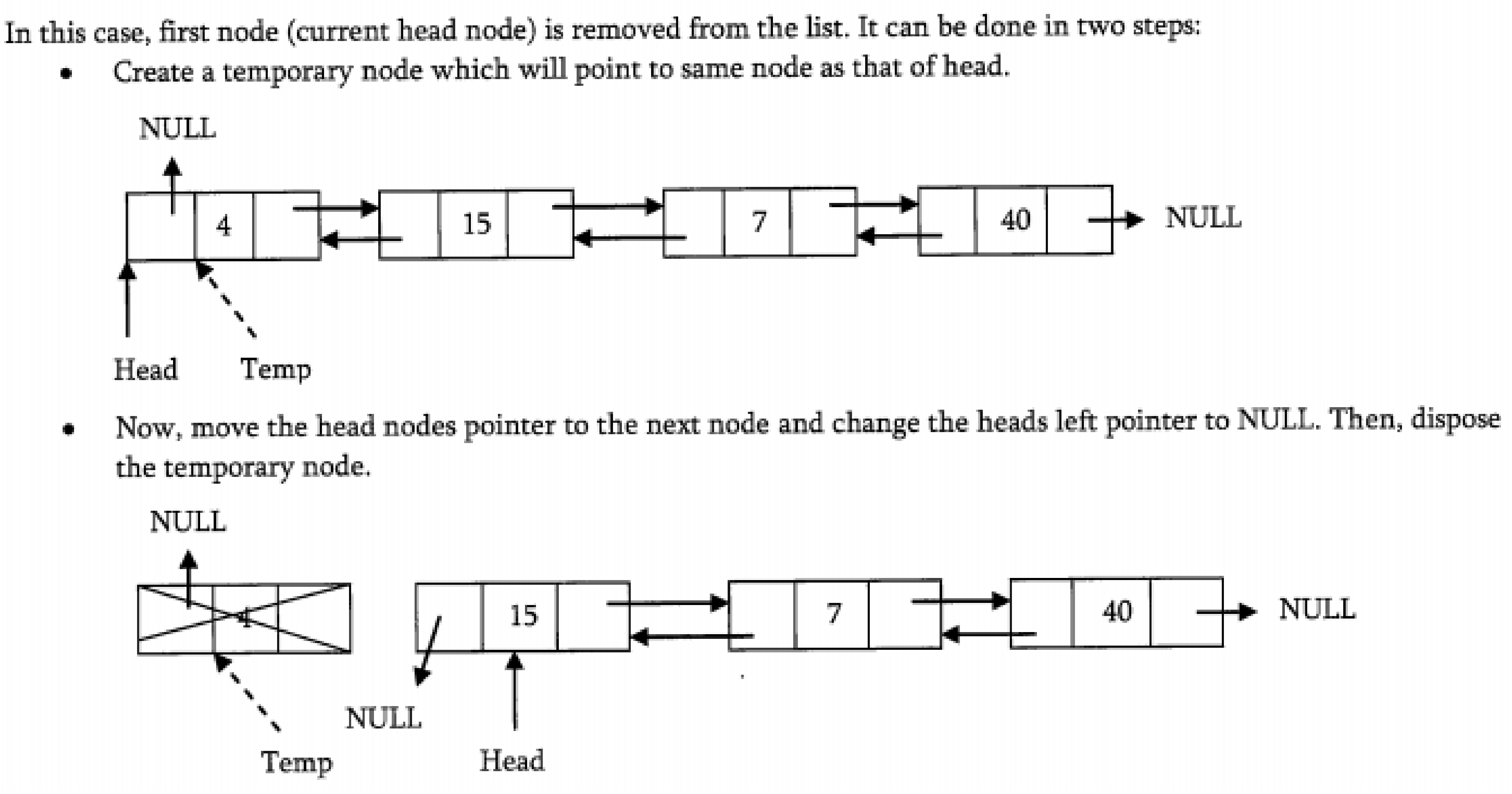
Traversing Backward

- Try to code!



```
public void iterateBackward(){

    System.out.println("iterating backward..");
    Node<E> tmp = tail;
    while(tmp != null){
        System.out.println(tmp.element);
        tmp = tmp.prev;
    }
}
```

Deleting the First Node

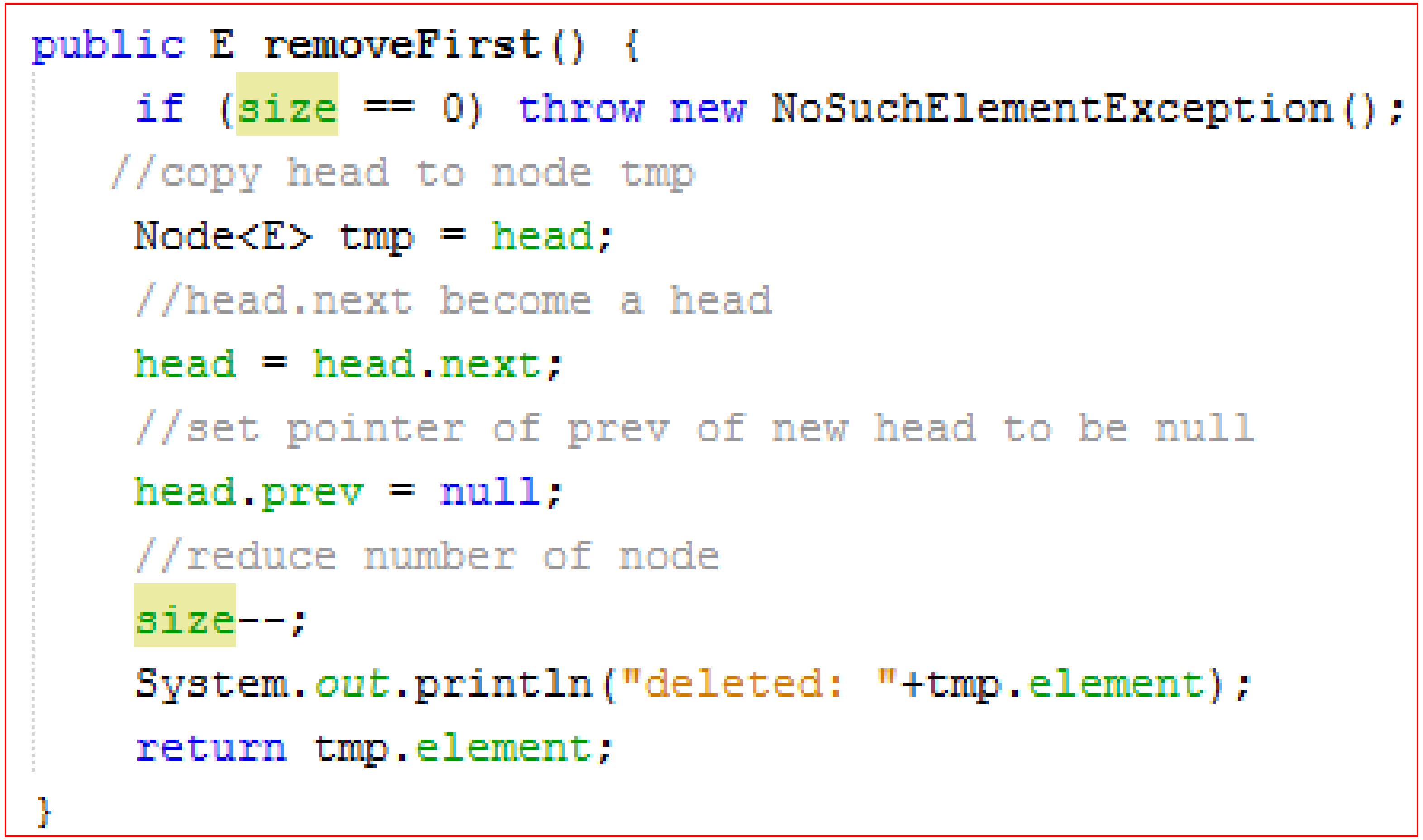


In this case, first node (current head node) is removed from the list. It can be done in two steps:

- Create a temporary node which will point to same node as that of head.

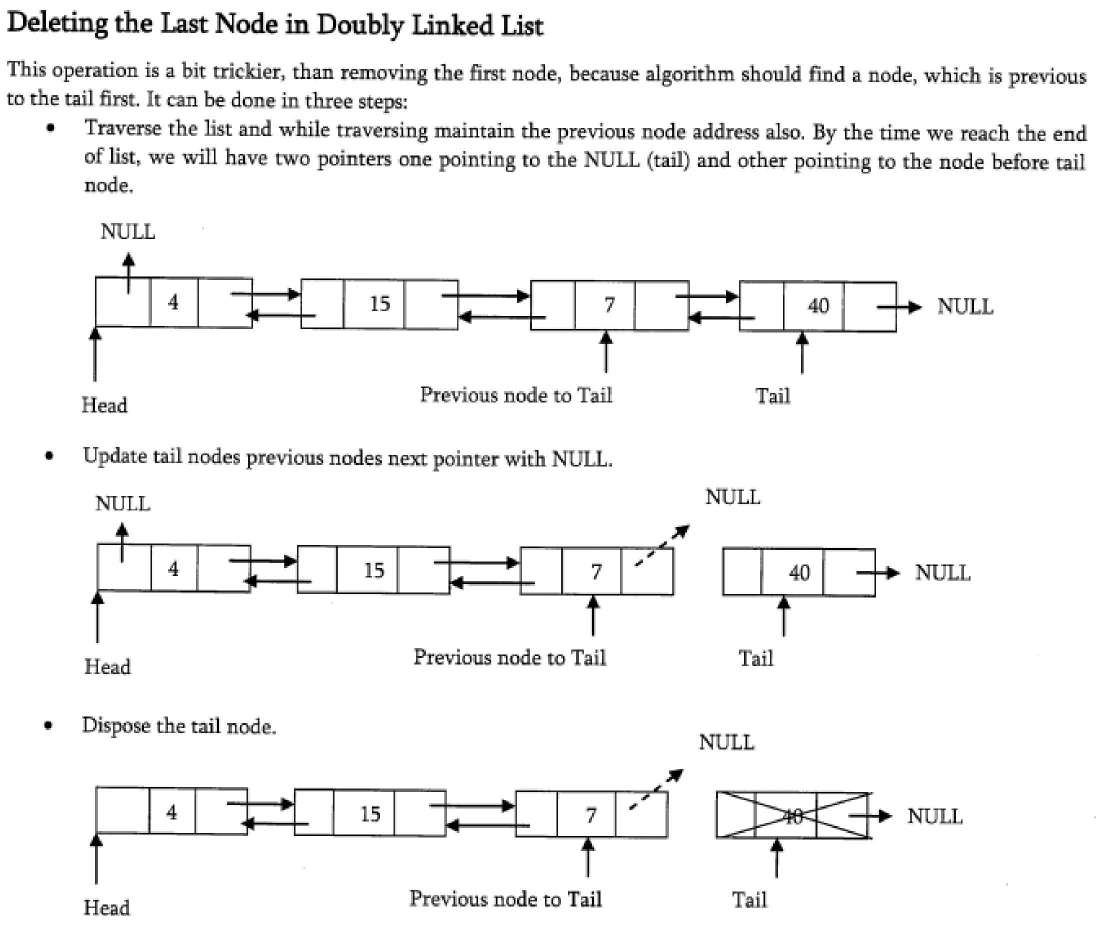
- Now, move the head nodes pointer to the next node and change the heads left pointer to NULL. Then, dispose the temporary node.

Deleting the First Node



```
public E removeFirst() {
    if (size == 0) throw new NoSuchElementException();
   //copy head to node tmp
    Node<E> tmp = head;
    //head.next become a head
    head = head.next;
    //set pointer of prev of new head to be null
    head.prev = null;
    //reduce number of node
    size--;
    System.out.println("deleted: "+tmp.element);
    return tmp.element;
}
```

Deleting the Last Node



Deleting the Last Node in Doubly Linked List

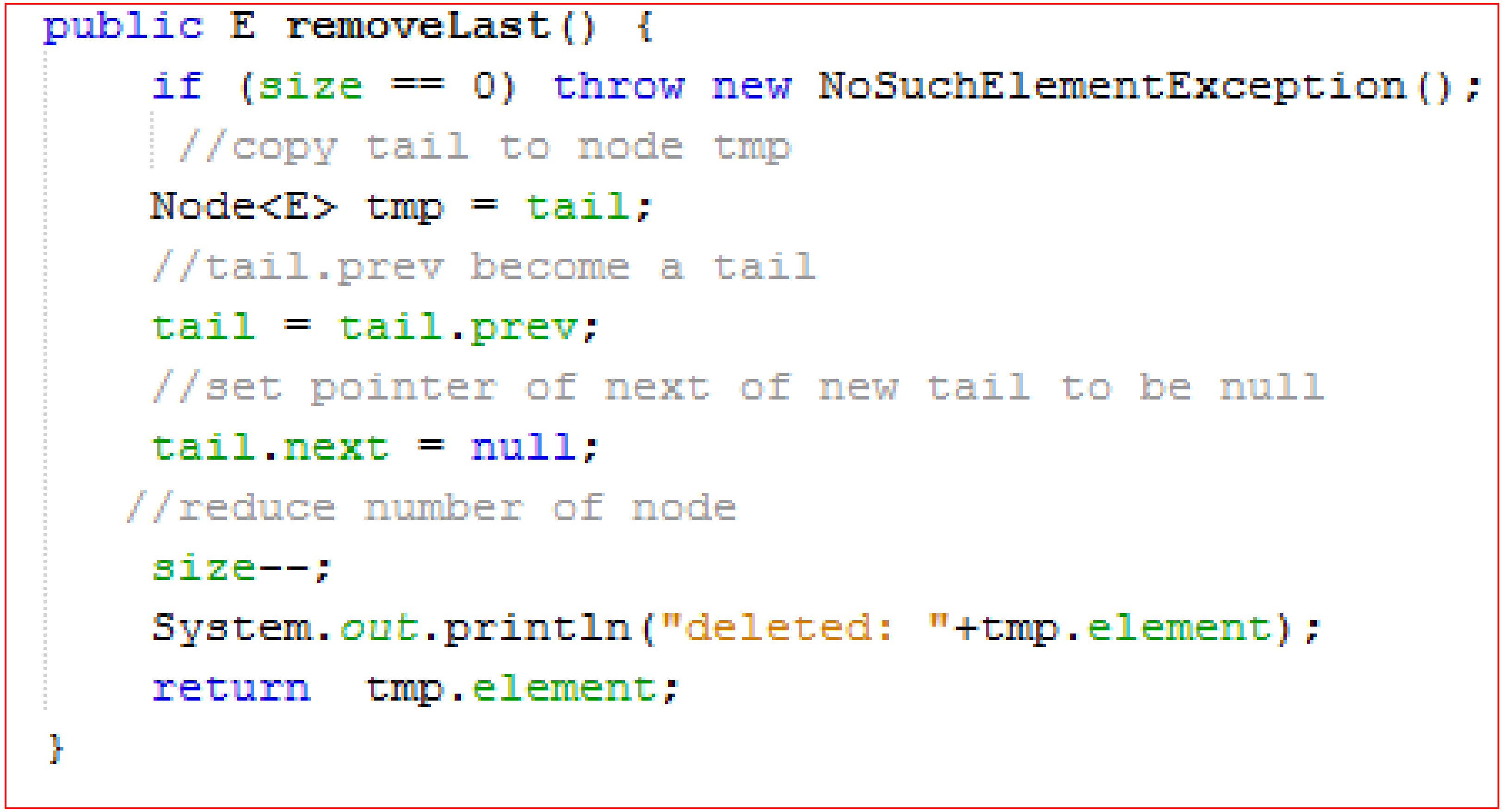
This operation is a bit trickier, than removing the first node, because algorithm should find a node, which is previous to the tail first. It can be done in three steps:

- Traverse the list and while traversing maintain the previous node address also. By the time we reach the end of list, we will have two pointers one pointing to the NULL (tail) and other pointing to the node before tail node.

- Update tail nodes previous nodes next pointer with NULL.

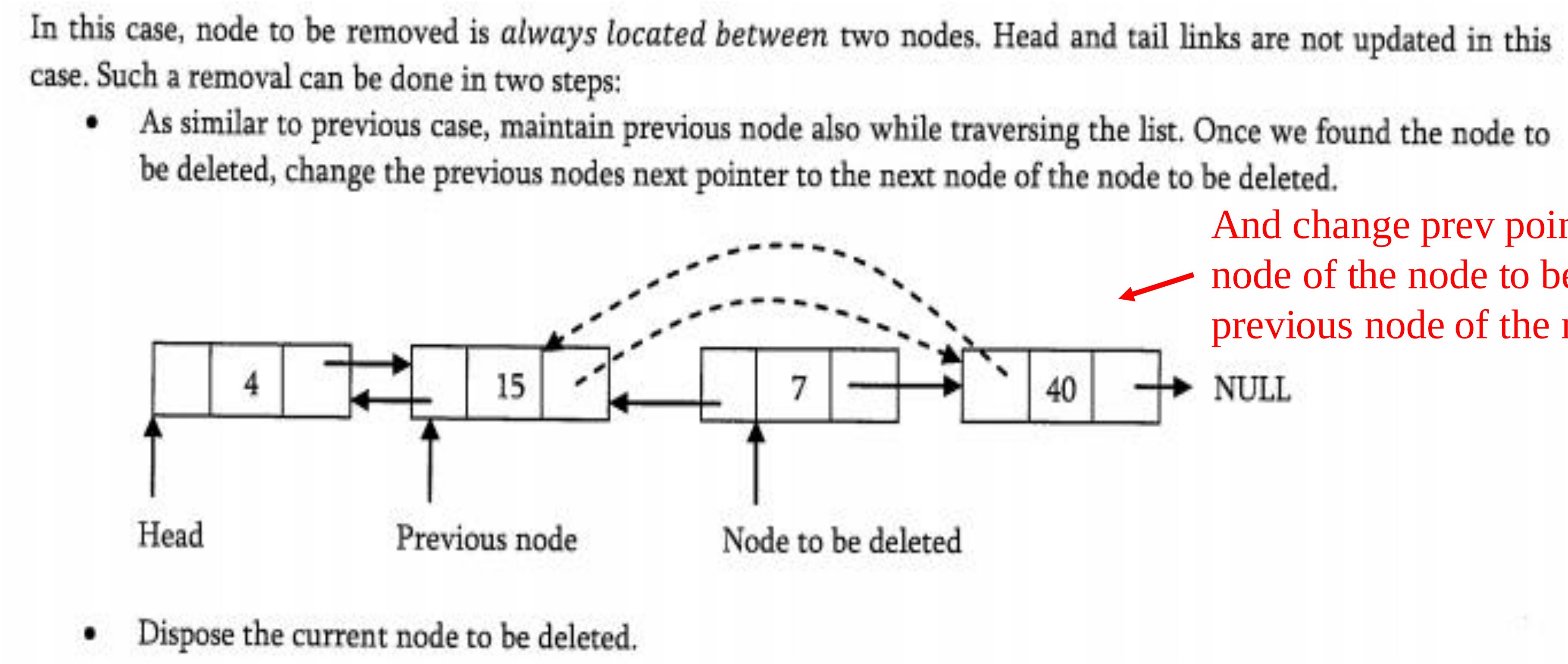
- Dispose the tail node.

Deleting the Last Node



```
public E removeLast() {
    if (size == 0) throw new NoSuchElementException();
     //copy tail to node tmp
    Node<E> tmp = tail;
    //tail.prev become a tail
    tail = tail.prev;
    //set pointer of next of new tail to be null
    tail.next = null;
   //reduce number of node
    size--;
    System.out.println("deleted: "+tmp.element);
    return  tmp.element;
}
```

Deleting the Intermediate Node



In this case, node to be removed is *always located between* two nodes. Head and tail links are not updated in this case. Such a removal can be done in two steps:

- As similar to previous case, maintain previous node also while traversing the list. Once we found the node to be deleted, change the previous nodes next pointer to the next node of the node to be deleted.

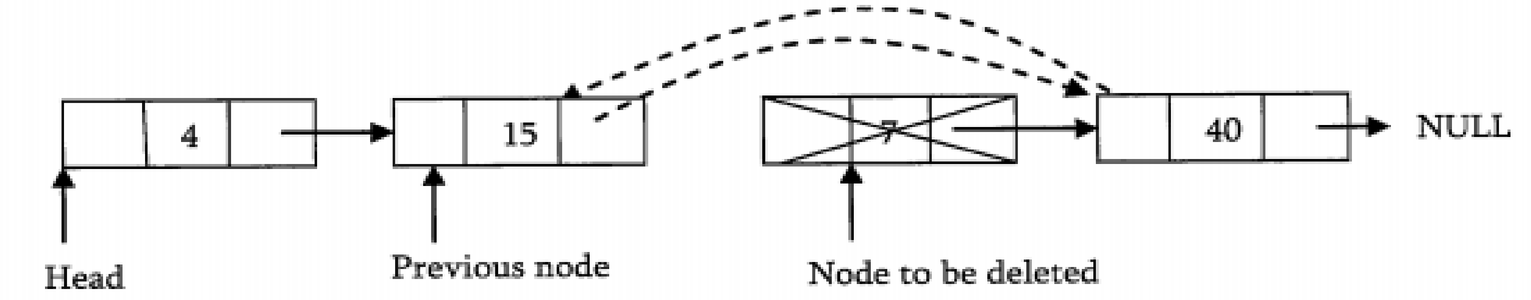
4
15
7
40
NULL
Head
Previous node
Node to be deleted

And change prev pointer of the next node of the node to be deleted to previous node of the node to be deleted

- Dispose the current node to be deleted.

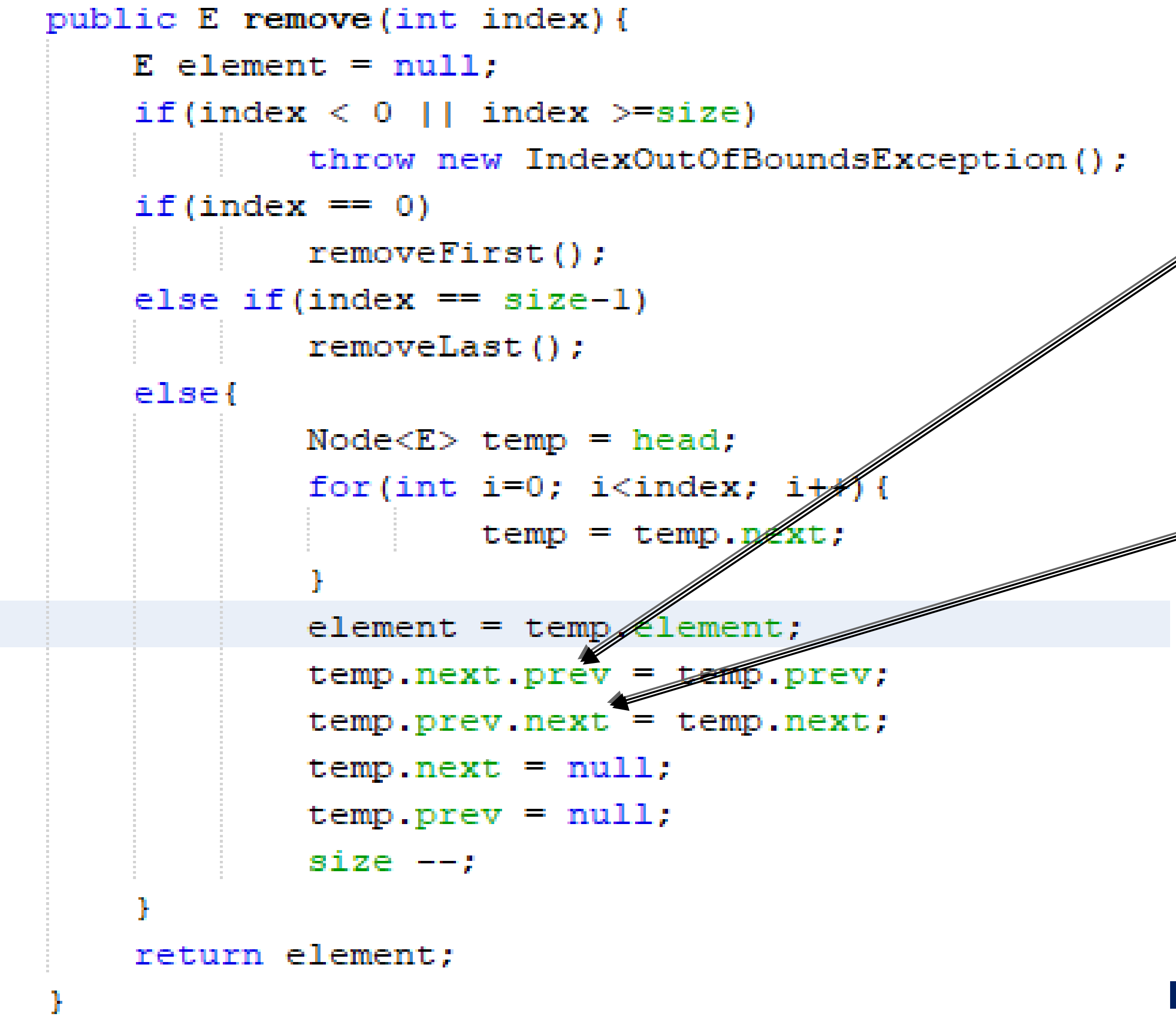
Deleting the Intermediate Node

- Dispose the current node to be deleted.



Try to code!

Ans: Deleting the Intermediate Node

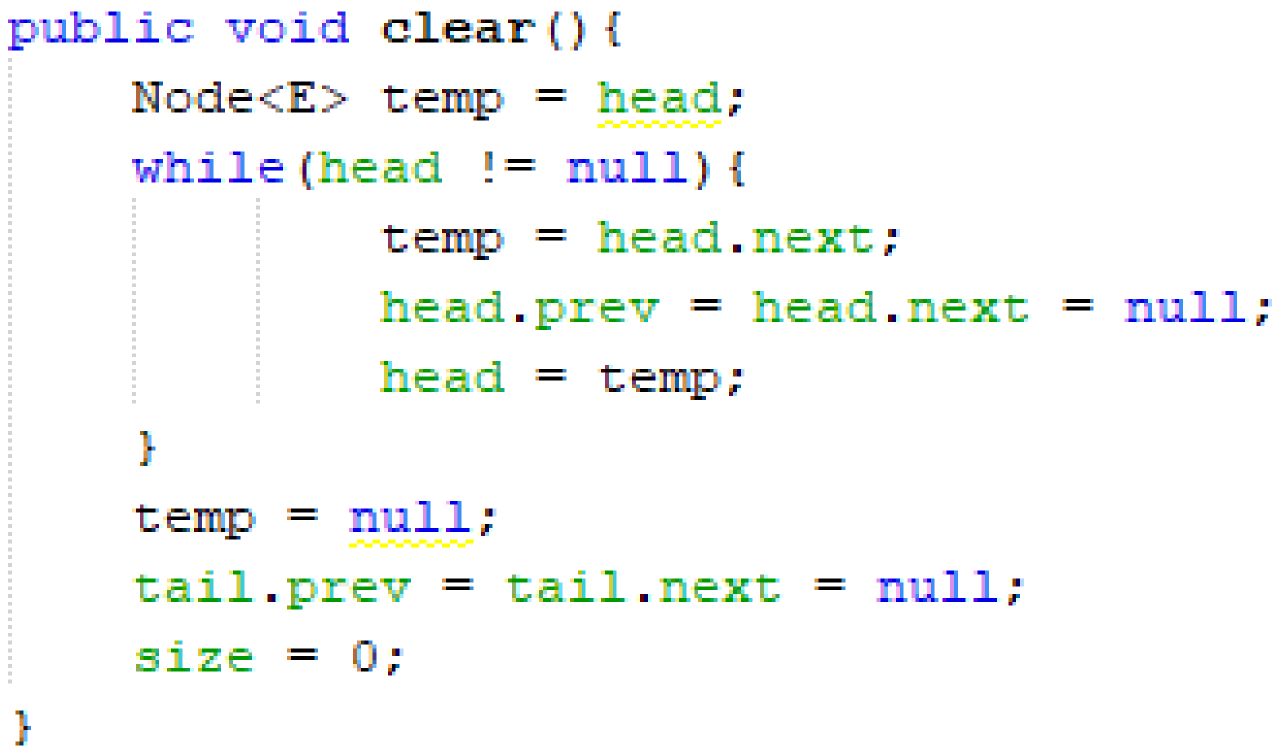


```
public E remove(int index){
    E element = null;
    if(index < 0 || index >=size)
            throw new IndexOutOfBoundsException();
    if(index == 0)
            removeFirst();
    else if(index == size-1)
            removeLast();
    else{
            Node<E> temp = head;
            for(int i=0; i<index; i++){
                    temp = temp.next;
            }
            element = temp.element;
            temp.next.prev = temp.prev;
            temp.prev.next = temp.next;
            temp.next = null;
            temp.prev = null;
            size --;
    }
    return element;
}
```

temp.next.prev
prev here is referring to the *prev* variable of the *next* node after index 2, namely, the node at index 3

temp.prev.next
next here is referring to the *next* variable of the *prev* node before index 2, namely, the node at index 1

Clear all nodes in the List



```
public void clear(){
    Node<E> temp = head;
    while(head != null){
            temp = head.next;
            head.prev = head.next = null;
            head = temp;
    }
    temp = null;
    tail.prev = tail.next = null;
    size = 0;
}
```

References

- Chapter 20, Liang, Introduction to Java Programming, 10th Edition, Global Edition, Pearson, 2015
- Chapter 24, Liang, Introduction to Java Programming, 10th Edition, Global Edition, Pearson, 2015