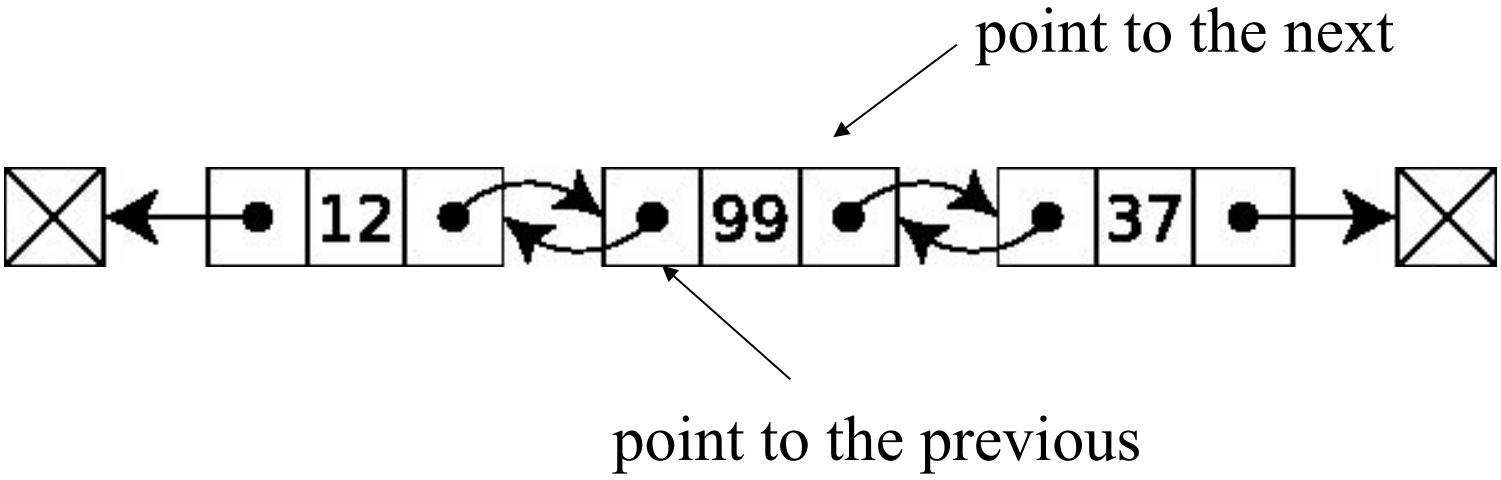
Linked List
Doubly Linked List

WIA1002/ WIB1002 :
Data Structures

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.
- 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



- 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

- Each node requires an extra pointer, requiring more space
- 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;
    }
```

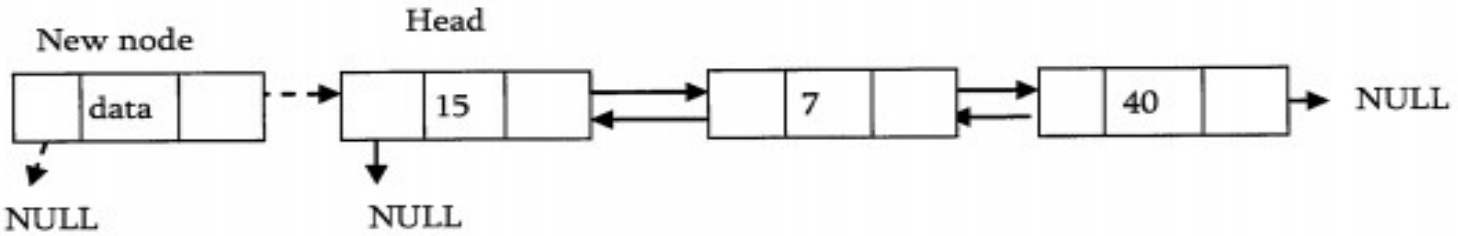
Doubly Linked List Insertion

- 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)

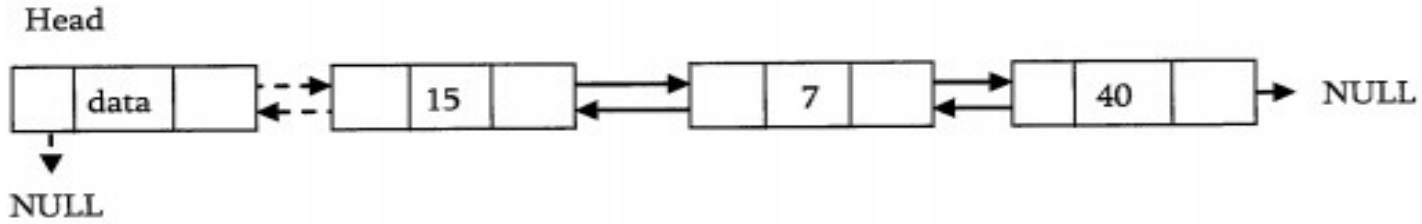
Inserting a Node at the Beginning

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.



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

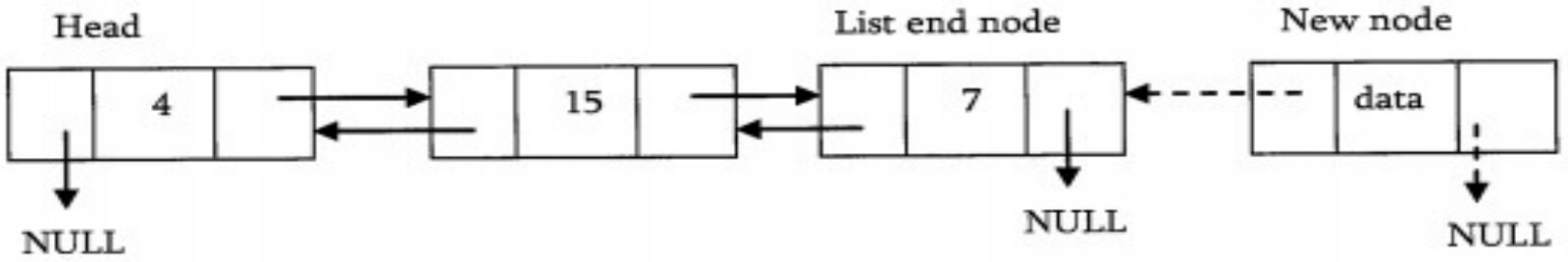



```
public void addFirst(E element) {
     Node<E> tmp = new Node(element, head, null);
     if(head != null ) {head.prev = tmp;}
     head = tmp;
     if(tail == null) { tail = tmp;}
     size++;
     System.out.println("adding: "+element);
}
```

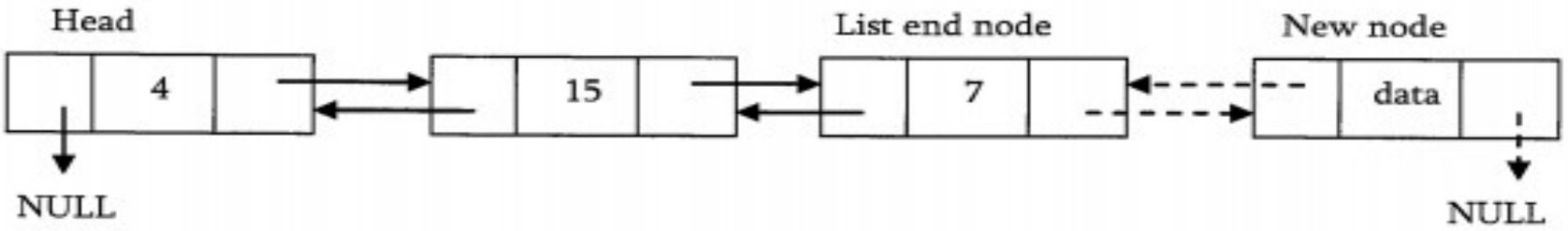
Inserting a Node at the Ending

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.

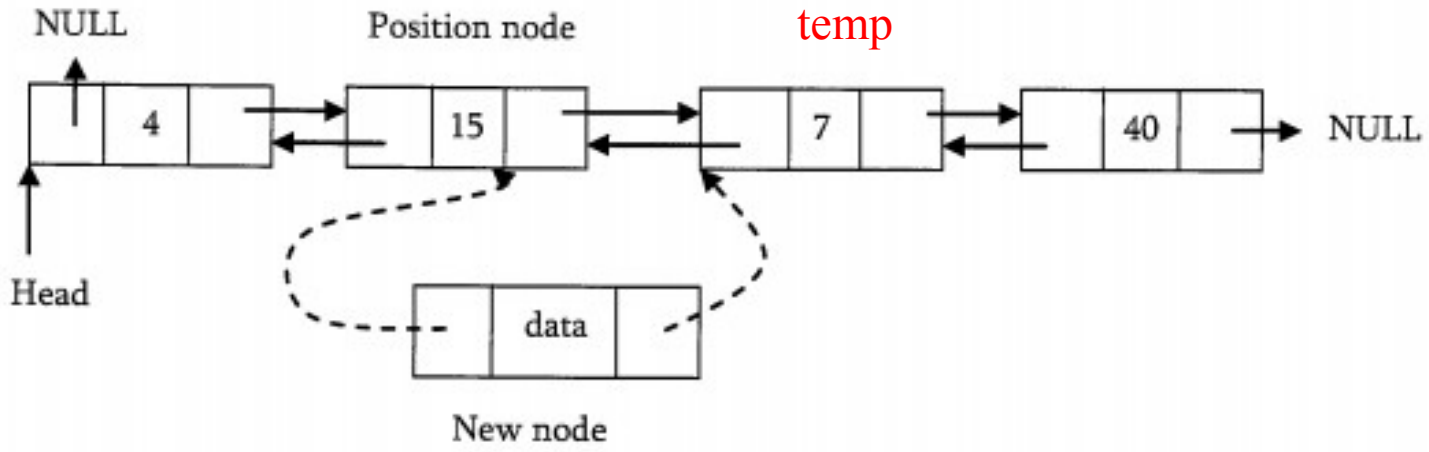



```
public void addLast(E element) {
    Node<E> tmp = new Node(element, null, tail);
    if(tail != null) {tail.next = tmp;}
    tail = tmp;
    if(head == null) { head = tmp;}
    size++;
    System.out.println("adding: "+element);
}
```

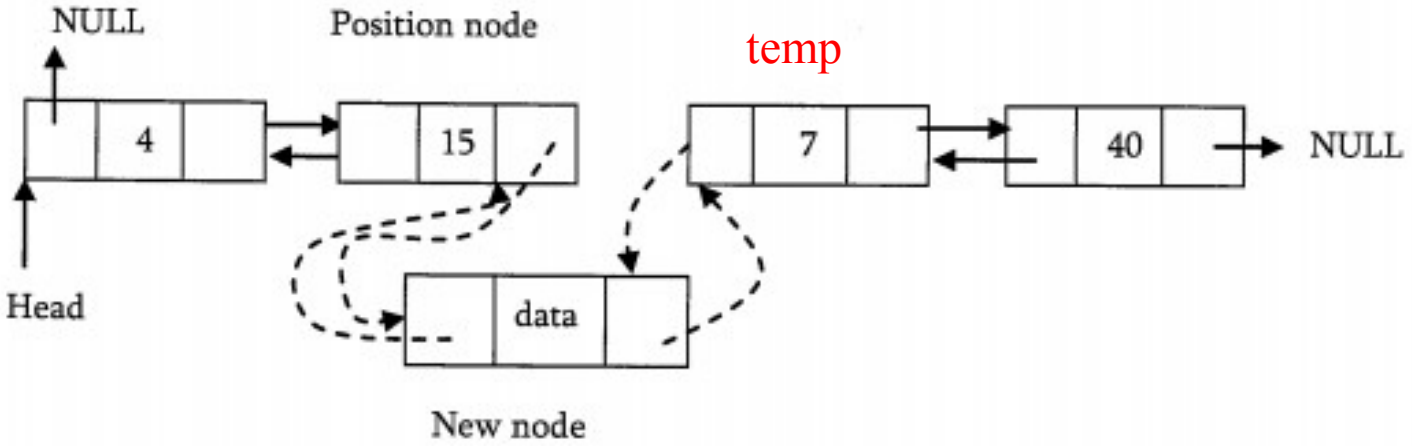
Inserting a Node in the Middle

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*.



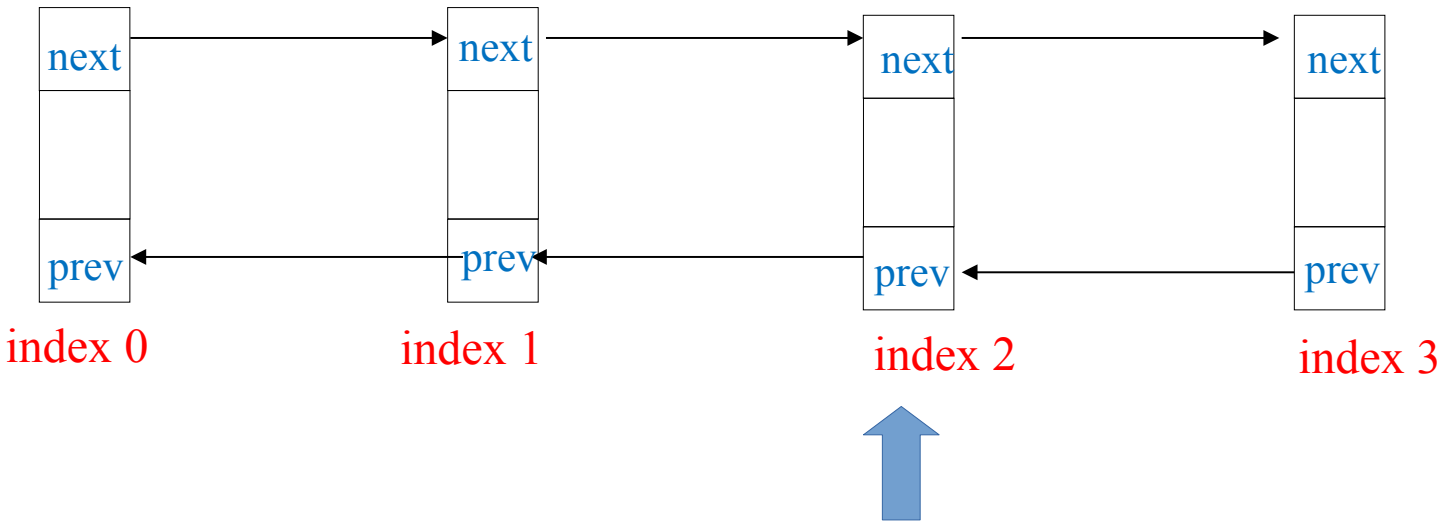
- Position node right pointer points to the new node and the *next node* of position nodes left pointer points to new node.



Inserting a Node in the Middle

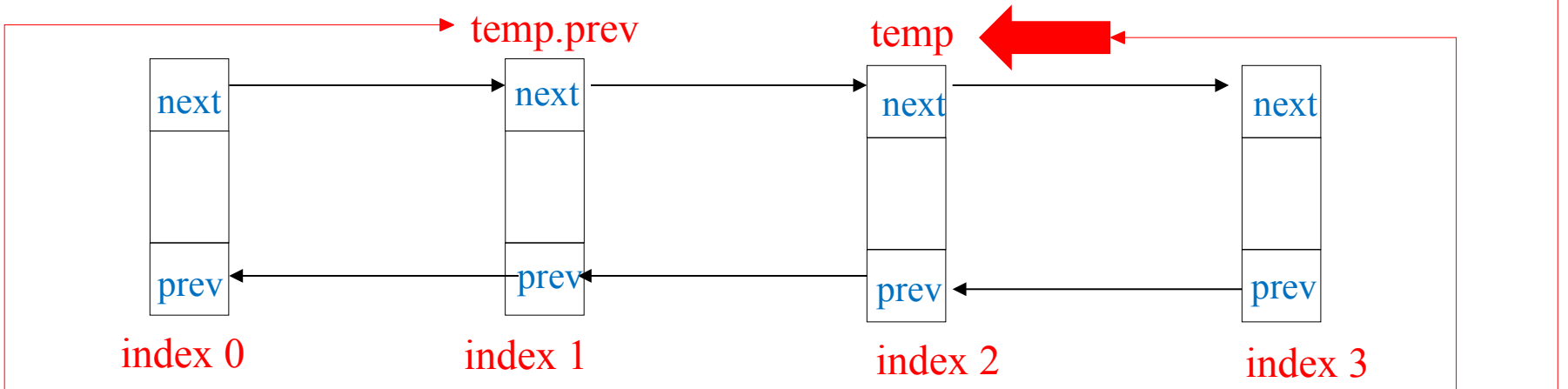
```
public void add(int index, E element){
        //index can only be 0 ~ size()
        if(index < 0 || index > size)
                throw new IndexOutOfBoundsException();
        if(index == 0)
                addFirst(element);
        else if(index == size)
                addLast(element);
        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 ++;
        }
    }
```

```
Node<E> temp = head;
for(int i=0; i<index; i++){
    temp = temp.next;
}

Node<E> insert = new Node(element, temp, temp.prev);
temp.prev.next = insert;
temp.prev = insert;
size ++;
```



Assume that we want to add a node at index=2

```
Node<E> temp = head;
for(int i=0; i<index; i++){
    temp = temp.next;
}

Node<E> insert = new Node(element, temp, temp.prev);
temp.prev.next = insert;
temp.prev = insert;
size ++;
```



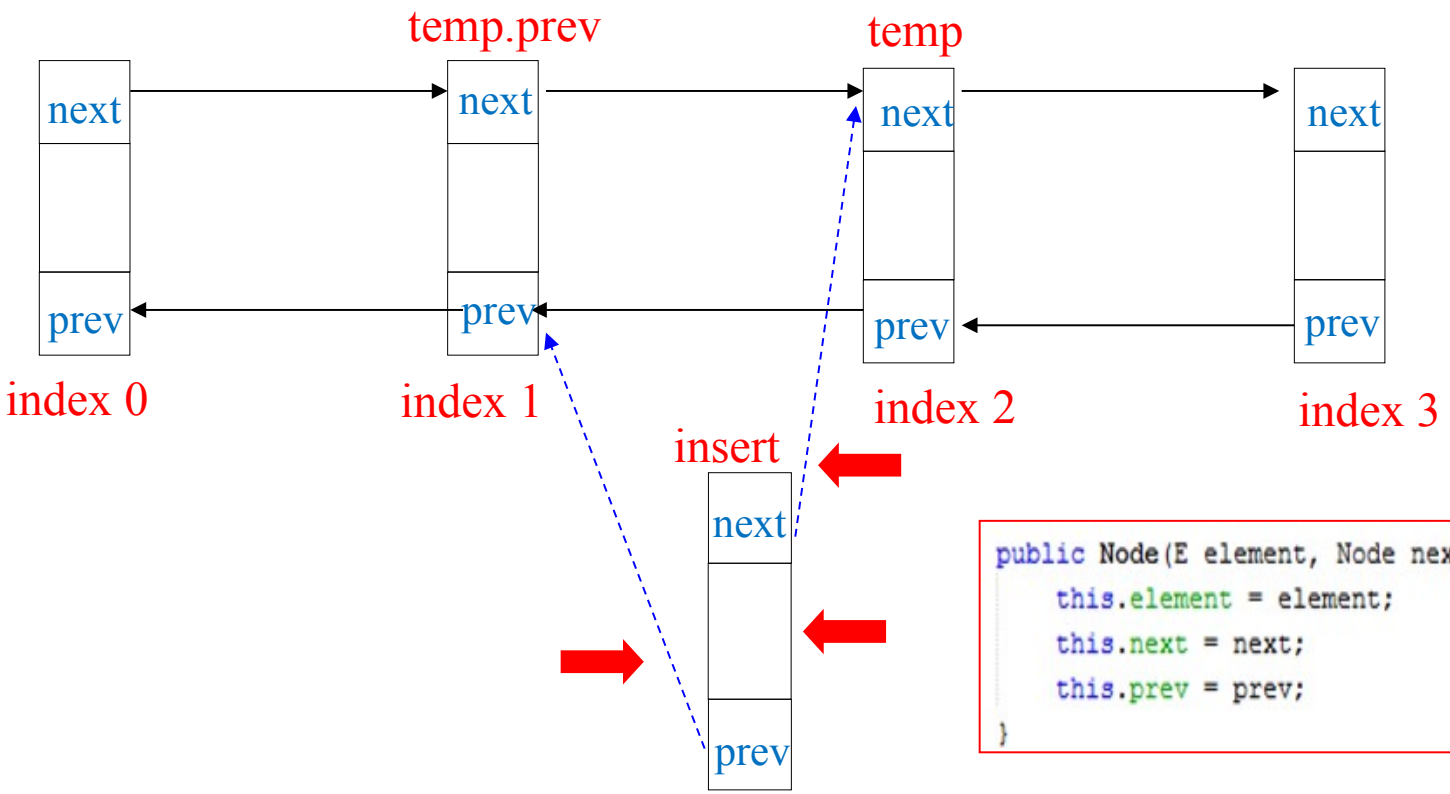
2

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

1

These lines allowed temp to traverse & stop at requested index (in this case index 2)

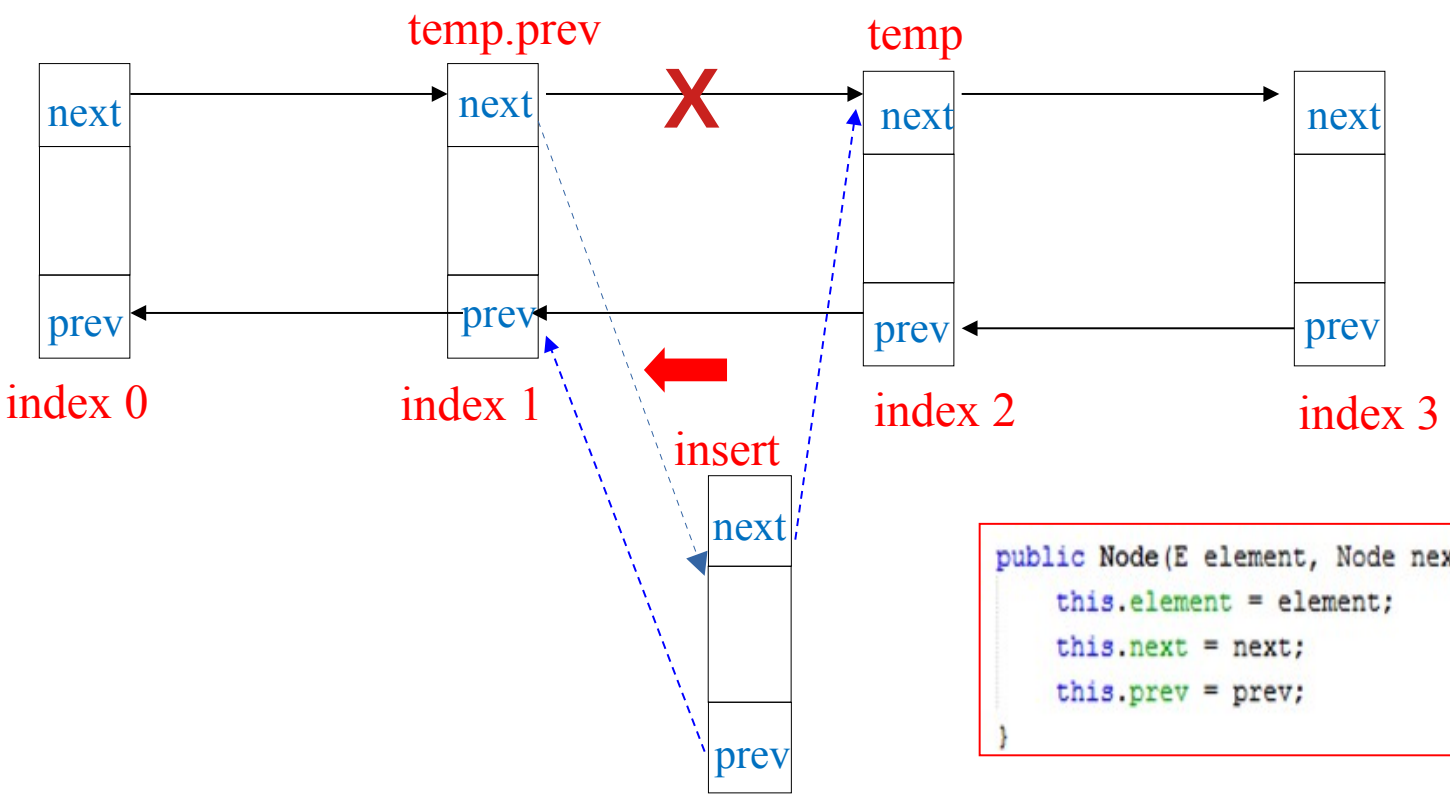
```
Node<E> temp = head;
for(int i=0; i<index; i++){
    temp = temp.next;
}

Node<E> insert = new Node(element, temp, temp.prev);
temp.prev.next = insert;
temp.prev = insert;
size ++;
```




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

new Node introduced at index 2

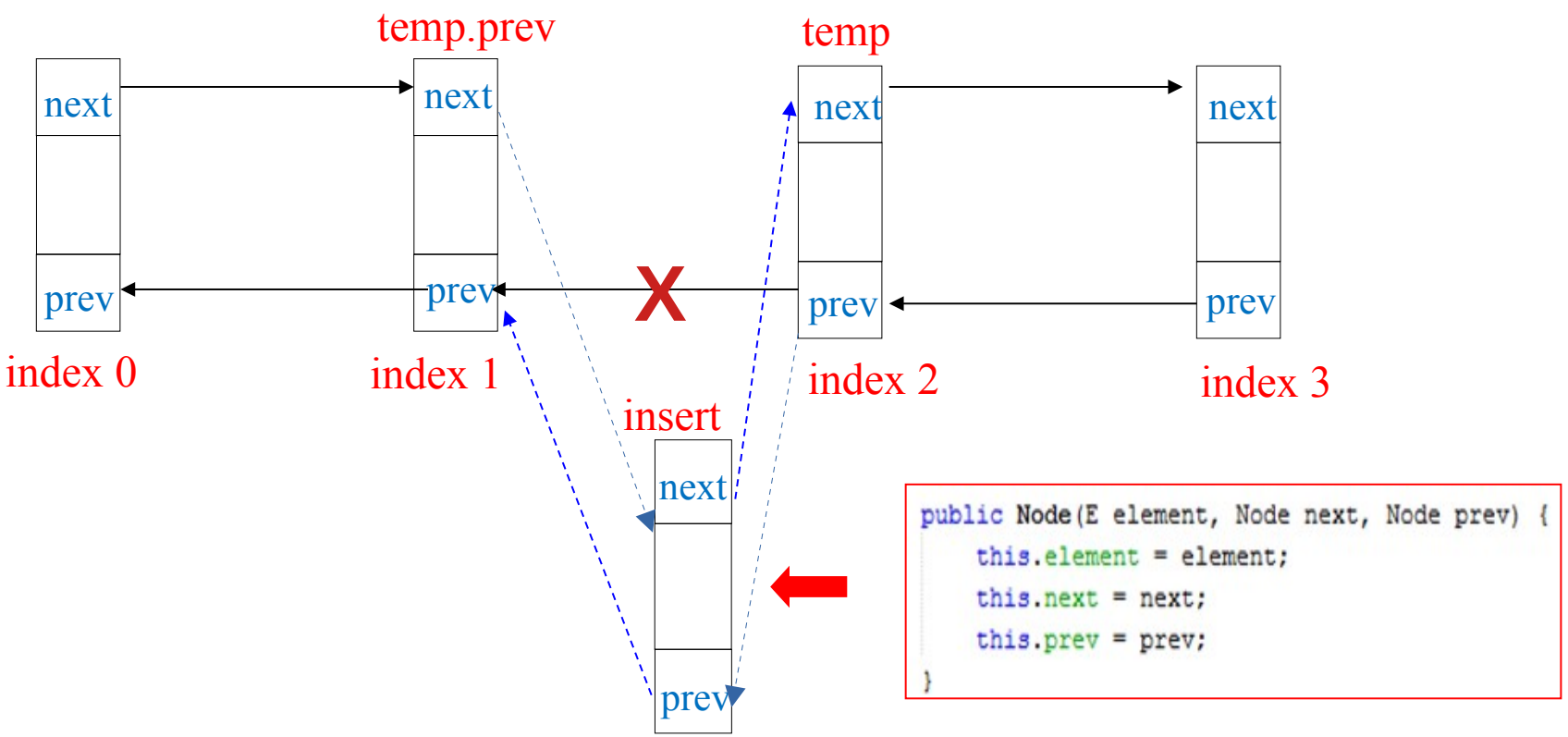
```
Node<E> temp = head;
for(int i=0; i<index; i++){
    temp = temp.next;
}

Node<E> insert = new Node(element, temp, temp.prev);
temp.prev.next = insert;
temp.prev = insert;
size ++;
```




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

new Node introduced at index 2

```
Node<E> temp = head;
for(int i=0; i<index; i++){
    temp = temp.next;
}

Node<E> insert = new Node(element, temp, temp.prev);
temp.prev.next = insert;
temp.prev = insert;
size ++;
```




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

new Node introduced at index 2

Traversing Forward

- Similar with Singly Linked List

```
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!

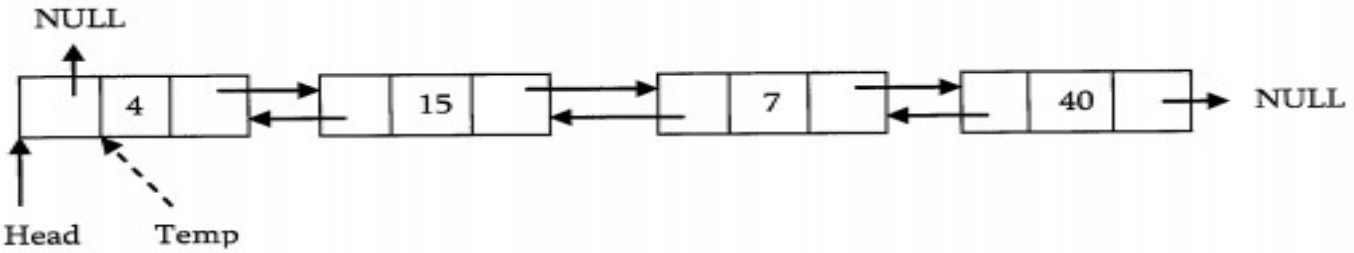
Try to code Traversing Backward

```
public void iterateBackward(){

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

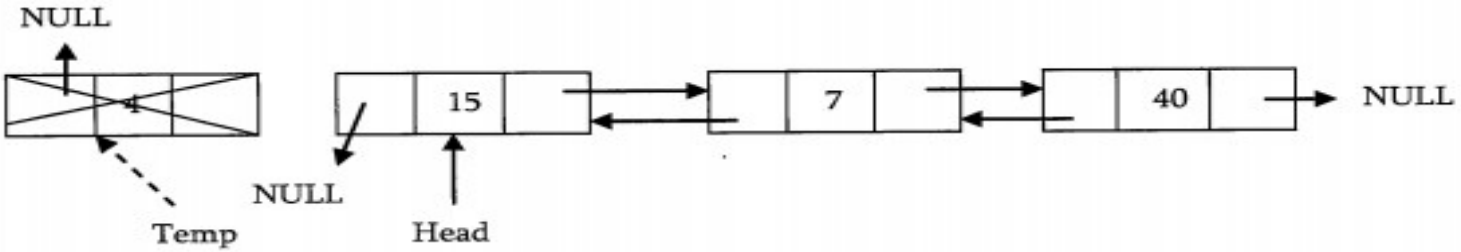
Deleting the First Node

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.



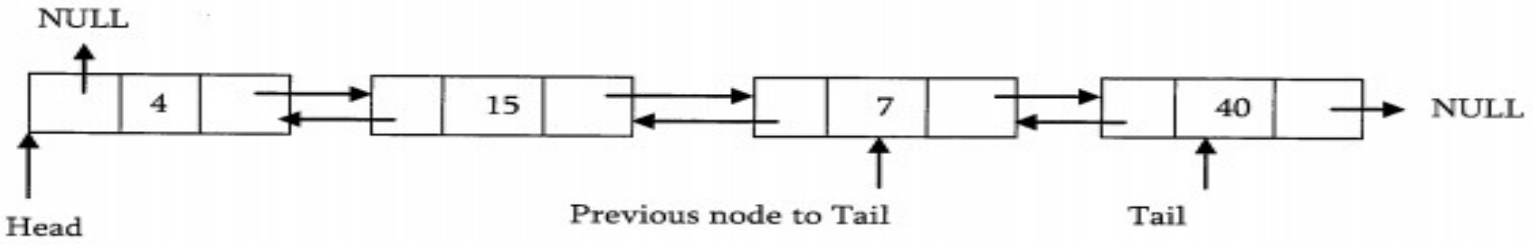

```
public E removeFirst() {
    if (size == 0) throw new NoSuchElementException();
   //copy head to node tmp
    Node<E> tmp = head;
    //head.next become a head
    head = head.next;
    //set pointer of prev of new head to be null
    head.prev = null;
    //reduce number of node
    size--;
  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

Deleting the Last Node

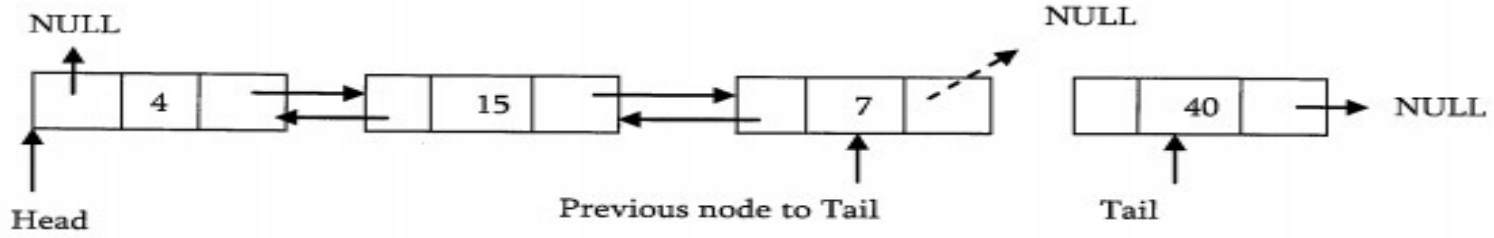
Deleting the Last Node in Doubly Linked List

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:

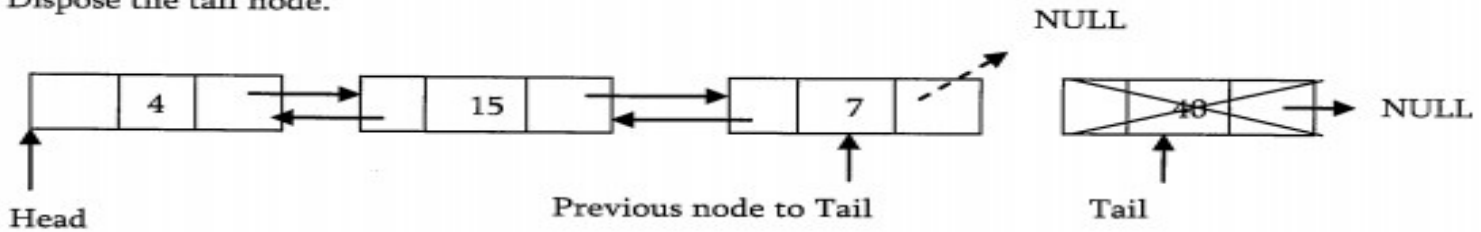
- Traverse the list and while traversing maintain the previous node address also. By the time we reach the end of list, we will have two pointers one pointing to the NULL (tail) and other pointing to the node before tail node.



- Update tail nodes previous nodes next pointer with NULL.



- Dispose the tail node.



Deleting the Last Node

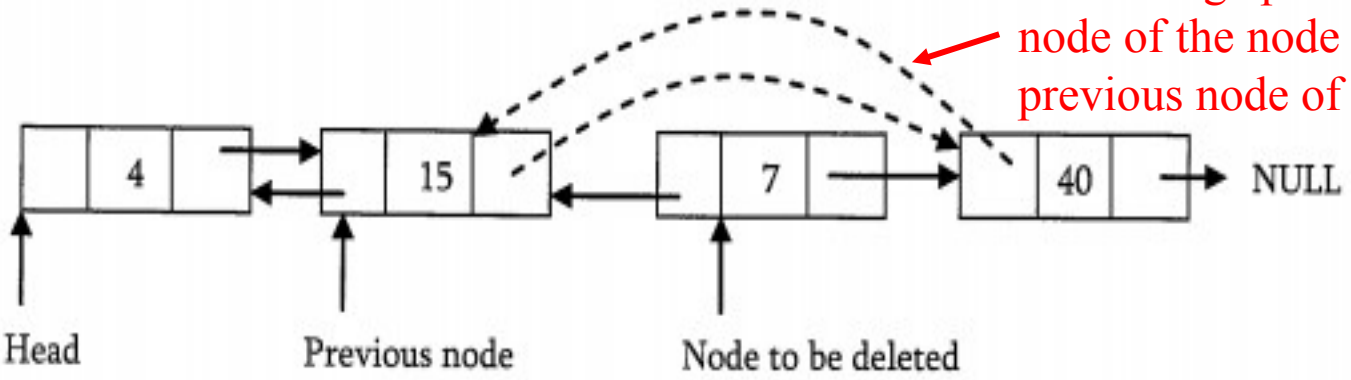
```
public E removeLast() {
    if (size == 0) throw new NoSuchElementException();
     //copy tail to node tmp
    Node<E> tmp = tail;
    //tail.prev become a tail
    tail = tail.prev;
    //set pointer of next of new tail to be null
    tail.next = null;
   //reduce number of node
    size--;
  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

Deleting an Intermediate Node

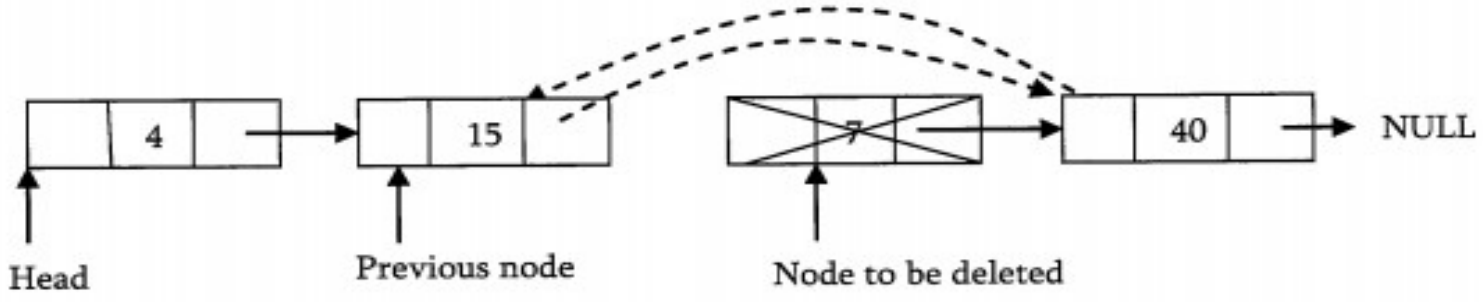
In this case, node to be removed is *always located between* two nodes. Head and tail links are not updated in this case. Such a removal can be done in two steps:

- 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.

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.



Try to code!

Answer : Deleting an 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;
}
```