# Dummy Headed Doubly Linked Circular lists (DHDLC) in Java

---

## Table of contents



---

## Introduction

We have so far focused on *head-referenced singly-linked linear* lists. Now we move on to a much more versatile variation that is often used in practice — *dummy head-referenced doubly linked circular* lists (DHDLC) — which is in fact the current implementation of the java.util.LinkedList class in JDK. It uses a *dummy* head node to avoid special cases of inserting into an empty list, or removing the last node from a list of unit size; and, it uses double links to allow iterating in both directions. The cost of course is the extra space needed to hold the dummy node (minimal cost), and the extra *previous* link in addition to the usual *next* link for each node (much more significant cost).

Before going ahead with our discussion on DHDLC, let's quickly review the various forms of doubly-linked lists. Figures 1, 2, and 3 show four of the common variations of a doubly-linked list: **linear** and **circular**, **head-referenced** and **dummy-head-referenced**. There are other possible variations as well (such as **tail-referenced** and **dummy-tail-referenced**), but these four are sufficient to illustrate the issues involved. In the figures below, **`header`** is the reference to the head or initial node of the list, which may be `null`; and the element in a dummy head node is marked with an × to indicate that the element value is irrelevant (and typically set to `null`).

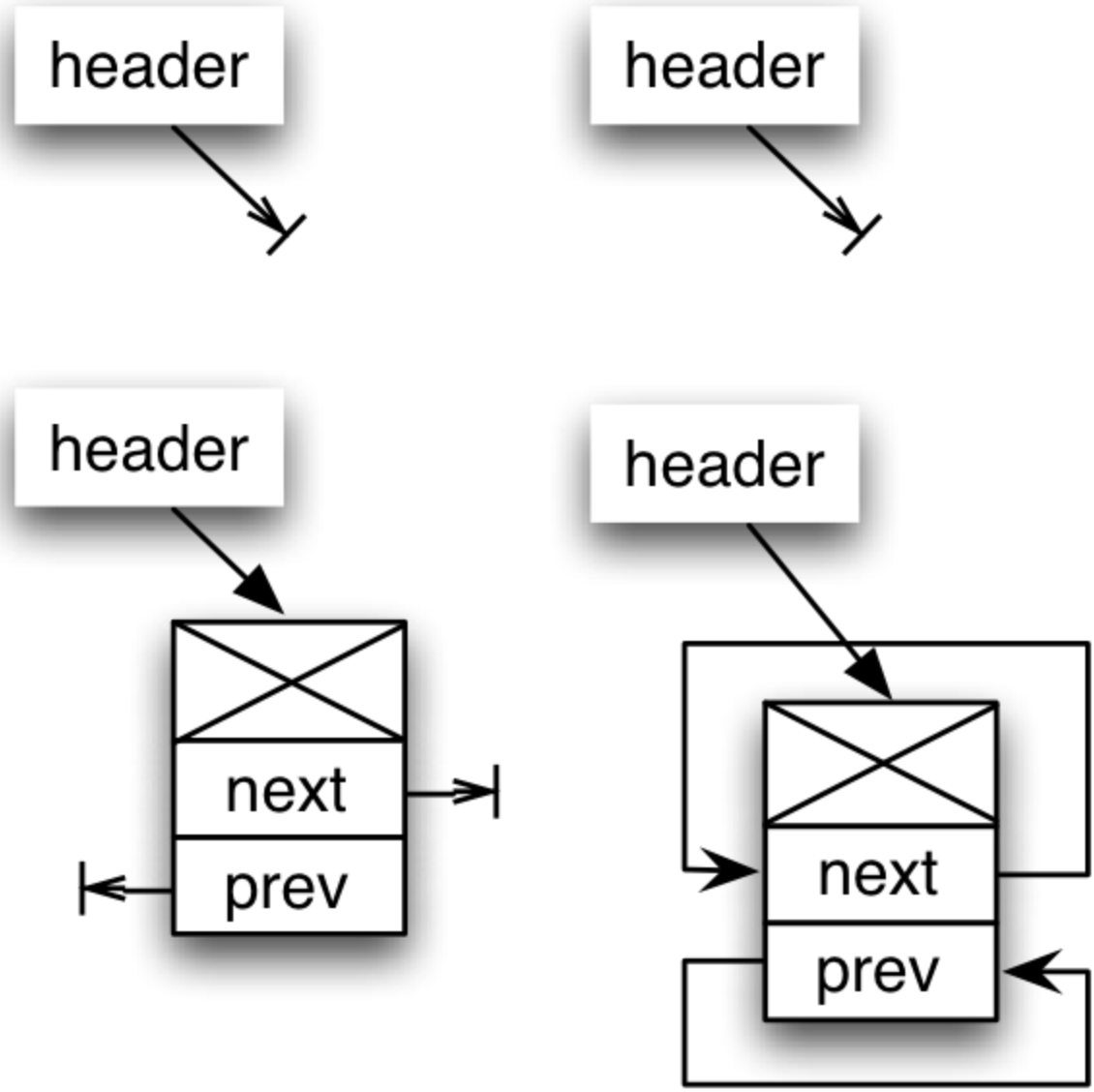



**Figure 1**: Head-referenced linear and circular empty lists (top row), and dummy head-referenced linear and circular empty lists (bottom row)

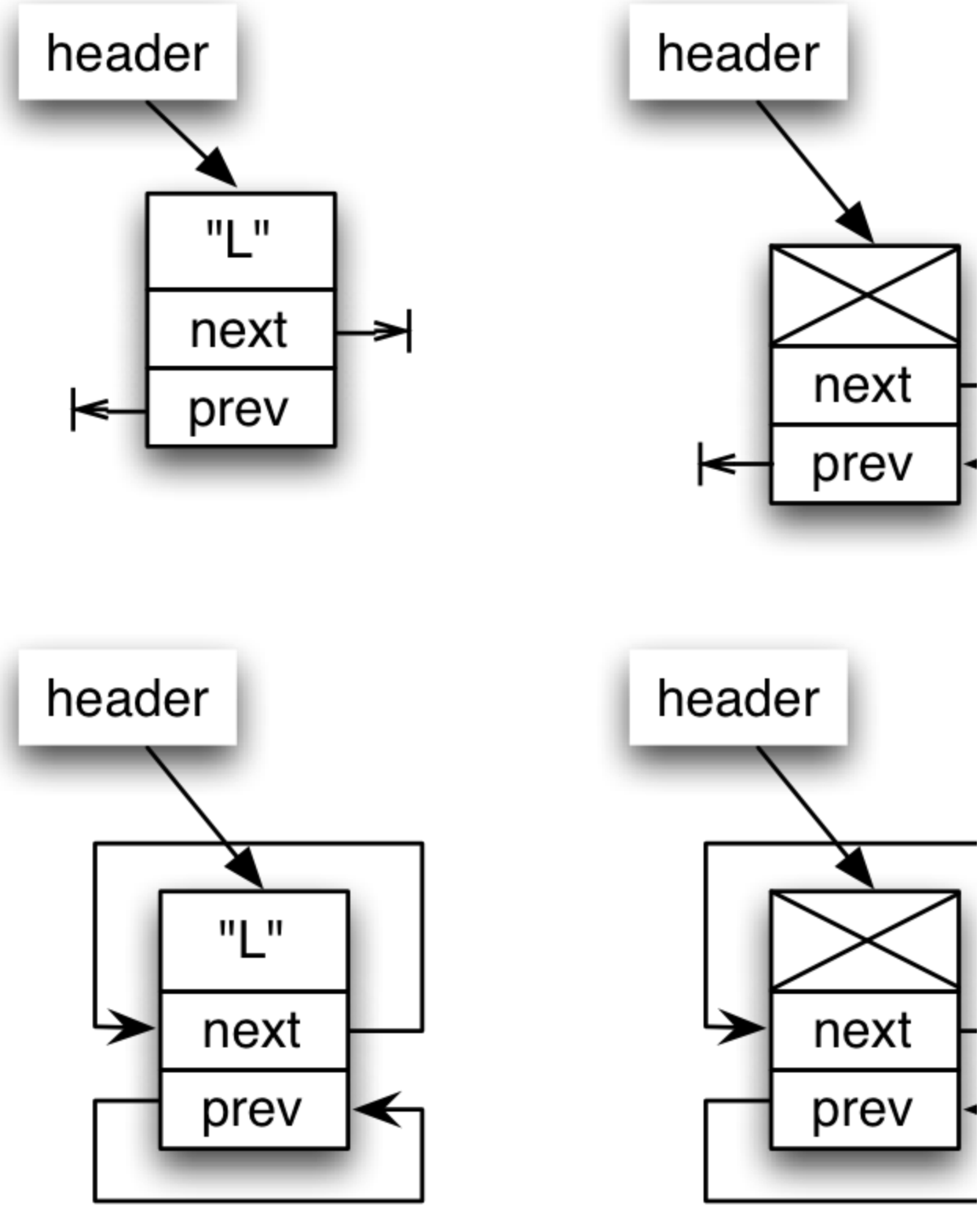



**Figure 2**: Head-referenced linear and circular single-element lists (top row), and dummy head-referenced linear and circular single-element lists (bottom row)

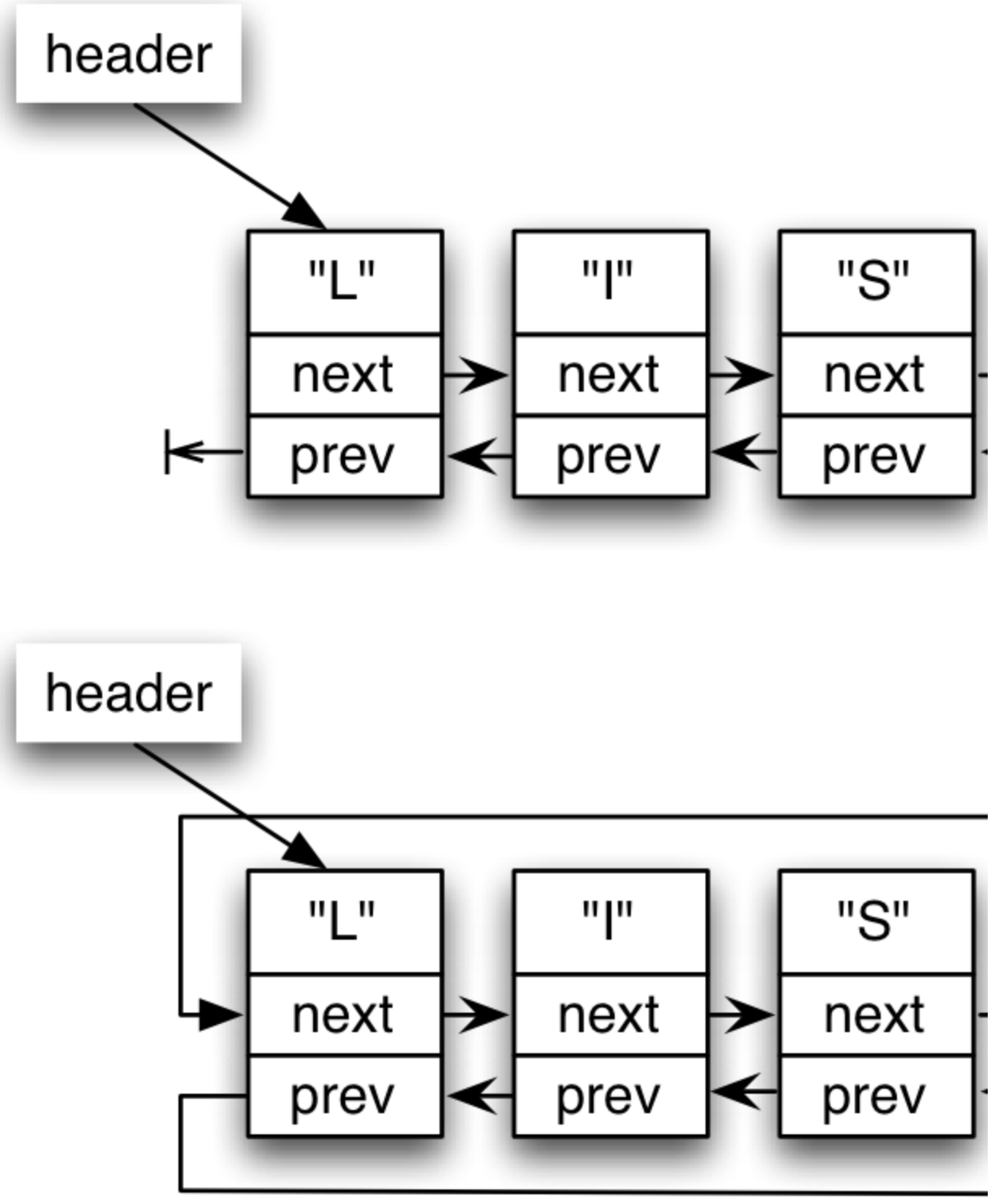



**Figure 3**: Head-referenced linear and circular 4-element lists (top row), and dummy head-referenced linear and circular 4-element lists (bottom row)

In the rest of this document, we are going focus only on dummy-headed circular doubly-linked lists, which require the following:

1. Each node in the list has references to both the *next* and *previous* nodes. To allow for links in both directions, we need to augment our `Node` class that we have used so far to add the extra `prev` link.

```
    public class Node {
        // The element that we want to put in the list
        public Object element;
        // The reference to the next node in the list
        public Node next;
        // The reference to the previous node in the list
        public Node prev;

        /**
         * Creates a new node with given element and next/prev
references.
         *
         * @param e the element to put in the node
         * @param n the next node in the list, which may be null
         * @param p the previous node in the list, which may be null
         */
        public Node(Object e, Node n, Node p) {
            element = e;
            next = n;
            prev = p;
        }
    }
```

2. The *tail* node's next refers to the *dummy head* node, and *dummy head* node's previous refers to the *tail* node.

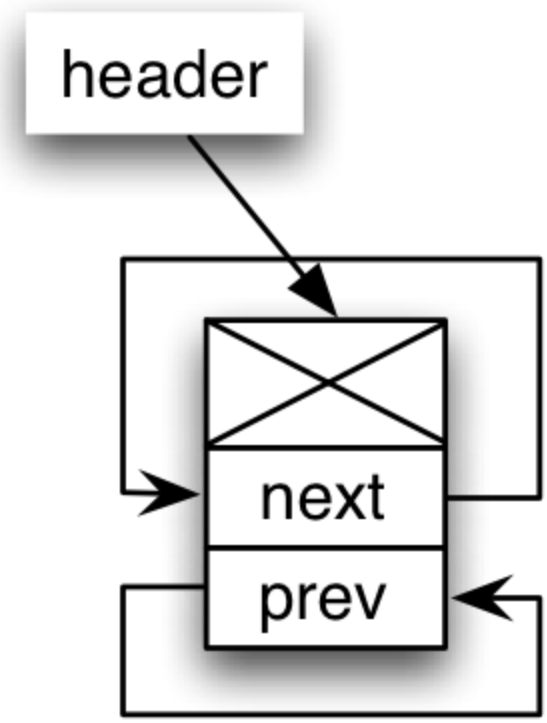
This allows you to create a *doubly-linked* list, and as a result, we can move *forward* and *backward* in the list by following the *next* and *prev* links. In the next few sections, we'll see how to create and manipulate a DHDLC linked list.

Back to Table of contents

# Creating a list of elements

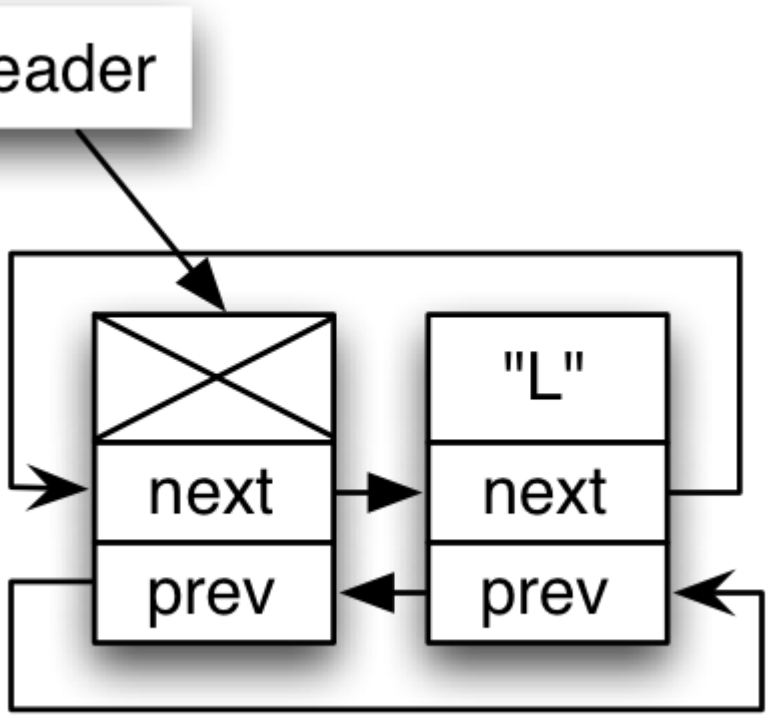
To create an empty list, referenced by a *dummy head* node:

```
// Create the dummy head node
Node header = new Node(null, null,
null);
// And then make it circular
header.next = header.prev = header;
```




Now, let's add a single String element ("L") after the dummy head:

```
Node n =
new
Node("L",
null,
null);
n.next =
header.next
;
n.prev =
header;
header.next
= n;
n.next.prev
= n;
```




You should note that the reference to the *tail* node is simply `header.prev`, which gives us the ability to *append* to the list in constant time (without having to iterate to find the tail reference, or having to maintain a separate tail reference).

Back to Table of contents

# Iterating over the elements

To iterate over the elements of a list, we start *after* the *dummy head* node, and then advance one node at a time until we reach the *dummy head* node again. For example, the following prints out each element (stored within the nodes) in the list.

```
Node n = header.next;
while (n != header) {
    System.out.println(n.element);
    n = n.next;
}
```

You can use a for loop as well of course.

```
for (Node n = header.next; n != header; n = n.next)
    System.out.println(n.element);
```

Thanks to the *previous* links, we can iterate backwards as well. We start from the *tail* node, which is `header.prev`, and move backwards.

```
Node n = header.prev;             // n now refers to the tail node
while (n != header) {
    System.out.println(n.element);
    n = n.prev;
}
```

You can use a for loop as well of course.

```
for (Node n = header.prev; n != header; n = n.prev)
    System.out.println(n.element);
```

Back to Table of contents

# Inserting an element into a list

There are three places in a list where we can insert a new element: in the beginning ("header" does not change since we're using a dummy), in the middle, and at the end. To insert a new node in the list, you need the reference to the *predecessor* to link in the new node. Unlike for a singly-linked linear list, there is no "special" case here, since there is always a valid *predecessor* node available, thanks to the dummy head. Let's write a static method that inserts a new element *after* the specified predecessor node in a list.

```
/**
 * Inserts the new element after the specified node in the list.
 *
 * @param p    the node after which the new element is to be added
 * @param elem the new element to insert after the specified node
 * @return the reference to the newly added node
 */
public static Node insertAfter(Node p, Object elem) {
    // Create the new node to hold the element in the list
    Node n = new Node(elem, null, null);

    // Now add it after the predecessor `p'
    Node q = p.next;            // q will refer to the next node
    n.next = q;
    n.prev = p;
    p.next = n;
    q.prev = n;

    // Done -- no special cases whatsoever!
    return n;
}
```

We can now use it to insert a new element after any existing node in the list. For example, the following adds the String element "Foo" in the beginning of the list (note that the dummy head is the predecessor in this case):

```
// Insert the String element "Foo" after the dummy head.
Node n = insertAfter(header, "Foo");
```

And the following appends the String element "Foo" to the end of the list:

```
// Appends the String element "Foo" after the tail node.
Node n = insertAfter(header.prev, "Foo"); // header.prev is the tail
```

Figure 4 shows the steps in inserting a new node in a list, given a predecessor node.

**Figure 4**: Inserting a new element "S" into the list

We're often asked to insert a new element at a given *position* (or equivalently, index or location). For example, the case in Fig. 4 may be written as:

```
insert(header, 2, "S"); // insert "S" at index 2
```

This is quite trivial — all we need to do is to get the reference to the predecessor node (basically, we find the node at `index-1` position by iterating), and then use the `insertAfter` method we've already written and used. There is a *special case* however — when the new element is to be inserted in the *beginning* of the list at index 0. The predecessor in this case is simply the dummy head node!

```
/**
 * Inserts the new element at the specified position.

 * @param header reference to the dummy head node
 * @param size   the size of the list
 * @param elem   the new element to be inserted
 * @param index  the position where it should be inserted
 *
 * @exception IndexOutOfBoundsException if index < 0 || index ≥ size
 */
public static void insert(Node header, int size, Object elem, int index)
    throws IndexOutOfBoundsException {

    if (index < 0 || index ≥ size)
        throw new IndexOutOfBoundsException();

    // Find the predecessor, keeping in mind the special case (index = 0)
    Node pred = null;
    if (index == 0)
        pred = header;
    else {
        pred = header.next;
        for (int i = 0; i < index - 1; i++)
            pred = pred.next;
    }

    // Now simply insert the new element after the predecessor!
    insertAfter(pred, elem);
}
```

Back to Table of contents

# Removing an element from a list

Removing an element from the list is done by removing the node that contains the element. If we only know the element that we want to remove, then we have to sequentially search the list to find the node which contains the element (if it exists in the list that is); or else, we may already

have a reference to the node which contains the element to remove; or, we have an index of the element in the list. Just like inserting a new node in a list, removing requires that you have the reference to the predecessor node. Since we're using a doubly-linked list, finding a predecessor of a node `n` is trivial — it's `n.prev`. And thanks to the dummy head, there is no "special" case here as well. Let's write a static method that takes a reference to a node to remove from the specified list.

```
/**
 * Removes the specified node from the list.
 *
 * @param n the specified node to remove
 */
public static void removeNode(Node n) {
    // References to the predecessor and successor
    Node p = n.prev;
    Node q = n.next;

    // Remove node n.
    p.next = q;
    q.prev = p;

    // Unlink 'n' from the list.
    n.next = n.prev = null;

    // Help GC
    n.element = null;
}
```

That's it! We can remove the first node in the list as follows:

```
removeNode(header.next);
```

And the last/tail node in the list as follows:

```
removeNode(header.prev);
```

If we have only a reference to the element to remove, we have to first find the node, and then remove it using `removeNode`. Let's write that method as well.

```
/**
 * Removes the specified element from the list.
 *
 * @param header the reference to the dummy head of the list
 * @param elem the specified element to remove from the list
 * @return true if the element was removed, or false otherwise
 */
public static void remove(Node header, Object elem) {
    // Sequentially scan to find the node. You should probably write a
    // a findNode method to return the Node that contains a given element.
    Node n = header.next;
    while (n != header) {
        if (n.element.equals(elem))
            break;
        n = n.next;
    }
```

```
    if (n == header)
        return false;           // did not find it!
    else {
        removeNode(n);
        return true;
    }
}
```

Figure 5 shows the steps in removing a specific node in a list, given its reference.

**Figure 5**: Removing the node containing element "S" from the list

Back to Table of contents