Linked lists in Java

Table of contents

1. Creating a list of elements
2. Iteration over the elements in a list
3. Counting the number of elements in a list
4. Getting an element given the index into a list
5. Setting an element given the index into a list
6. Searching for an element in a list
7. Inserting an element into a list
8. Removing an element from a list
9. Copying a list
10. Reversing a list
11. Rotating a list left
12. Rotating a list right

Introduction

When studying Java's built-in array type, we were quite annoyed by the limitations of these arrays:

- *fixed capacity*: once created, an array cannot be resized. The only way to "resize" is to create a larger new array, copy the elements from the original array into the new one, and then change the reference to the new one. Very inefficient, but of course we can use a bit of intelligence to make it a bit better (eg., double the space each time instead of adding just the required amount, etc).
- *shifting elements in insert/remove*: since we do not allow *gaps* in the array, inserting a new element requires that we shift all the subsequent elements right to create a *hole* where the new element is placed. In the worst case, we "disturb" every slot in the array when we insert in the beginning of the array!

  Removing may also require shifting to *plug* the hole left by the removed element. If the ordering of the elements does not not matter, we can avoid shifting by replacing the removed element with the element in the last slot (and then treating the last slot as empty).

The answer to these problems is the *linked list* data structure, which is a sequence of *nodes* connected by *links*. The links allow inserting new nodes anywhere in the list or to remove an existing node from the list without having to disturb ther rest of the list (the only nodes affected are the ones adjacent to the node being inserted or removed). Since we can extend the list one node at a time, we can also resize a list until we run out of resources. However, we'll find out soon enough that what we lose in the process — random access, and space! Linked list is a sequence container that supports sequential access only, and requires additional space for at least `n` references for the links.

We maintain a linked list by referring to the first node in the list, conventionally called the *head* reference. All the other nodes can then be reached by traversing the list, starting from the *head*. An empty list is represented by setting the head reference to the null. Given a head reference to a list, how do you count the number of nodes in the list? You have to iterate over the nodes, starting from head, and count until you reach the end.

Figure 1 shows an empty, a single-element, and a 4-element **head-referenced singly-linked list**.

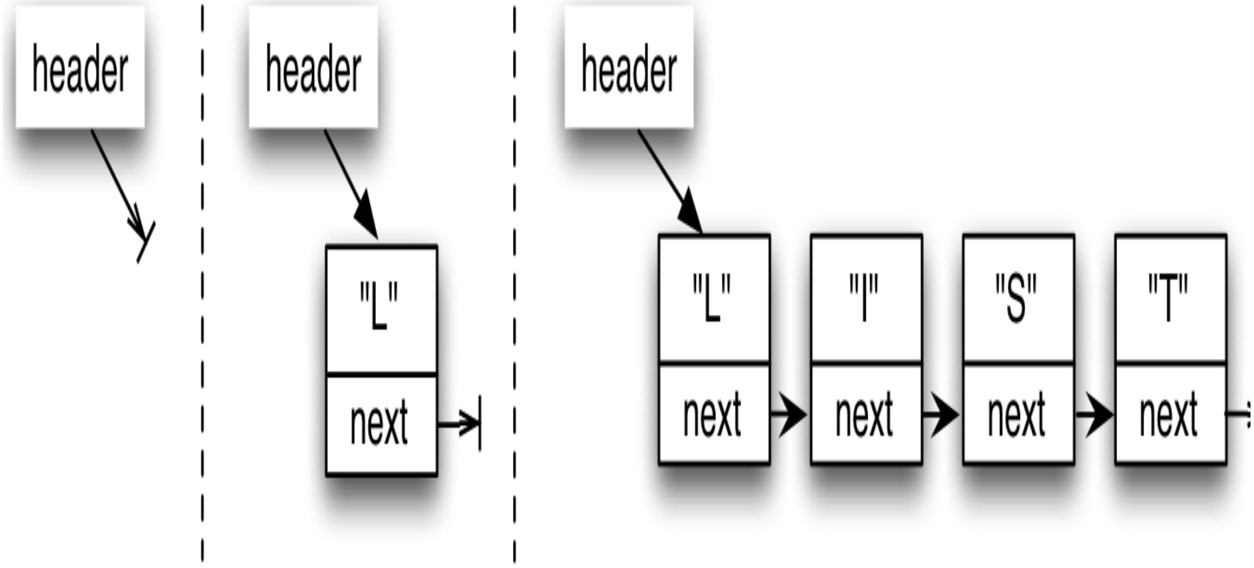


Figure 1: Example of head-referenced lists: empty, 1-element, and 4-element lists

As I mentioned earlier, a linked list is a sequence of nodes. To put anything (primitive type or object reference) in the list, we must first put this "thing" in a node, and then put the node in the list. Compare this with an array – we simply put the "thing" (our primitive type or object reference) directly into the array slot, without the need for any extra scaffolding. Our `Node` class is quite simple — it contains element (an object reference if we have a list of objects), and a reference to the *next* node in the list:

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

This allows you a create a *singly-linked* list, and as a result, we can only move *forward* in the list by following the *next* links. To be able to move *backward* as well, we'll have to add another reference to the *previous* node, increasing the space usage even more.

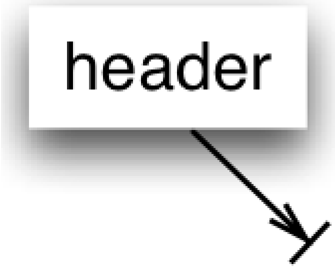
Now we look at the typical list manipulations that we're discussing in class this week. See the link to a complete implementation for you to play with at the end of this document.

Back to Table of contents

Creating a list of elements

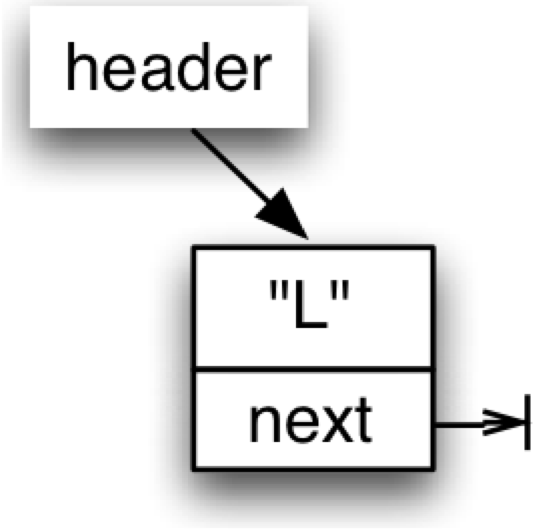
To create an empty list, referenced by *header*:

```
    Node header = null;      // empty list
```



A list with just one String element ("L"):

```
    Node header = new Node("L", null);
```



Back to Table of contents

Iterating over the elements

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

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

You can use a for loop as well of course.

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

Back to Table of contents

Counting the number of elements in a list

Given a head reference to a list, how can you count the number of elements in the list? You have to iterate over the nodes, starting from head, and count until you reach the end.

```
/**
 * Counts the number of nodes in the given list.
 *
 * @param head reference to the header node of the list
 * @return the number of nodes in the list
 */
public static int count(Node head) {
    int count = 0;
    for (Node n = head; n != null; n = n.next)
        count++;

    return count;
}
```

Back to Table of contents

Getting an element given the index into a list

It's useful to be able to extract an element given the index into a linked list (the index works the same way as for arrays – it goes from `0` to `size-1`). Since a linked list does not support random access, we have to scan the list to find the node at the given index, and then get the element within that node. Since we may not know the length (same as what we call size) of the list, we have to check if the index is within bounds.

```
/**
 * Returns the element at the given index in the given list.
 *
 * @param head  the reference to the header node of the list
 * @param index the index of the element to get
 * @return the element at the given index, or null if the index is out
 *         of bounds.
 */
public static Object get(Node head, int index) {
    if (index < 0)
        return null;                    // invalid index.

    int i = 0;
    for (Node n = head; n != null; n = n.next) {
        if (i == index)
            return n.element;
        else
           i++;
    }
    return null;                        // index out of bounds.
}
```

If we know the number of elements in the list (ie., the *size*), then it's much easier.

```
/**
 * Returns the element at the given index in the given list.
 *
 * @param head  the reference to the header node of the list
 * @param size  the number of elements in the list
 * @param index the index of the element to get
 * @return the element at the given index, or null if the index is out
 *         of bounds.
 */
public static Object get(Node head, int size, int index) {
    if (index < 0 || index >= size)
        return null;                    // invalid index.
    Node n = head;
    for (int i = 0; i < index; i++, n = n.next)
        ;

    return n.element;
}
```

It's typical in most applications to write a method to get the node at a given index, so that we can get the element by first getting the node, and the return the element within. We define a `nodeAt` method to return the node at the given index.

```
/**
 * Returns the node at the given index in the given list.
 *
 * @param head  the reference to the header node of the list
 * @param size  the number of elements in the list
 * @param index the index of the node to get
 * @return the node at the given index, or null if the index is out
 *         of bounds.
 */
public static Node nodeAt(Node head, int size, int index) {
    if (index < 0 || index >= size)
        return null;                    // invalid index.

    Node n = head;
    for (int i = 0; i < index; i++, n = n.next)
        ;

    return n;
}
```

With this method, it's trivial to write the `get` method.

```
/**
 * Returns the element at the given index in the given list.
 *
 * @param head  the reference to the header node of the list
 * @param size  the number of elements in the list
 * @param index the index of the element to get
 * @return the element at the given index, or null if the index is out
 *         of bounds.
 */
public static Object get(Node head, int size, int index) {
    Node node = nodeAt(head, size, index);
    if (node == null)
        return null;                    // invalid index.
    else
        return node.element;
}
```

Back to Table of contents

Setting an element given the index into a list

Setting the element at the given index (also called *updating* the element) is as simple as `get` using `nodeAt` method.

```
/**
 * Sets the element to the given one at the given index in the given list.
 *
 * @param head  the reference to the header node of the list
 * @param size  the number of elements in the list
 * @param index the index of the element to update
 * @param elem  the element to update the value with
 * @return the old element at index if valid, or null if the index is out
 *         of bounds.
 */
public static Object set(Node head, int size, int index, Object elem) {
    Node node = nodeAt(head, size, index);
    if (node == null)
        return null;                    // invalid index.
    else {
        Object oldElem = node.element;
        node.element = elem;
        return oldElem;
    }
}
```

Back to Table of contents

Searching for an element in a list

Searching for an element in a list can be done by sequentially searching through the list. There are two typical variants: return the index of the given element (`indexOf`), or return true if the element exists (`contains`). Both of these require that we walk through the list and check for the existence of the given element.

```
/**
 * Returns the index of the element in the list.
 *
 * @param head  the reference to the header node of the list
 * @param size  the number of elements in the list
 * @param elem  the element to find the index of
 * @return the index of the element is found, or -1 otherwise
 */
public static int indexOf(Node head, int size, Object elem) {
    int i = 0;
    for (Node n = head; n != null; n = n.next, i++)
        if (n.element.equals(elem))
            return i;

    return -1;                          // Not found
}
```

The other variant returns true if the element is found, or false otherwise.

```
/**
```

```
 * Returns true if the element exists in the list.
 *
 * @param head  the reference to the header node of the list
 * @param size  the number of elements in the list
 * @param elem  the element to find the index of
 * @return true if the element is found, or false otherwise
 */
public static boolean contains(Node head, int size, Object elem) {
    int i = 0;
    for (Node n = head; n != null; n = n.next, i++)
        if (n.element.equals(elem))
            return true;

    return false;                               // Not found
}
```

You should note that we can easily use one of the two methods to define the other instead of writing both! We can rewrite `contains` using `indexOf` or vice-versa quite trivally.

```
/**
 * Returns true if the element exists in the list.
 *
 * @param head  the reference to the header node of the list
 * @param size  the number of elements in the list
 * @param elem  the element to find the index of
 * @return true if the element is found, or false otherwise
 */
public static boolean contains(Node head, int size, Object elem) {
    int index = indexOf(head, size, elem);
    if (index >= 0)
        return true;
    else
        return false;

    // The "if ... else" block can also be written as:
    // return (index >= 0) ? true : false;
    // or even simpler as
    // return index >= 0;
}
```

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 ("head" changes), 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. There is one *special case* however — inserting a new node in the beginning of the list, because the *head* node does not have a predecessor. Inserting in the beginning has an important *side-effect* — it changes the head reference! Inserting in the middle or at the end are the same — we first find the predecessor node, and link in the new node with the given element. This *special case* of inserting in the beginning of the list can be avoided by the use of *dummy header* node (also called a *sentinel* node). More on this later (see the notes on *dummy head-referenced doubly-linked circular list*).

```
/**
 * Inserts the new element at the given index into the list.
 *
 * @param head  the reference to the header node of the list
 * @param size  the number of elements in the list
 * @param index the index of the newly inserted element
 * @param elem  the new element to insert at the given index
 * @return the reference to the header node, which will change when
 *         inserting in the beginning.
 * @exception IndexOutOfBoundsException if the index is out-of-bounds.
 */
public static Node insert(Node head, int size, Object elem, int index) {
    // Check for invalid index first. Should be between 0 and size (note:
    // note size-1, since we can insert it *after* the tail node (tail
    // node has an index of size-1).
    if (index < 0 || index > size)
        throw new IndexOutOfBoundsExceptionException();

    // Create the new node to hold the element in the list
    Node newNode = new Node(elem, null);

    // Remember the special case -- in the beginning of the list, when
    // the head reference changes.
    if (index == 0) {
        newNode.next = head;
        head = newNode;
    } else {
        // get the predecessor node first
        Node pred = nodeAt(index - 1);
        newNode.next = pred.next;
        pred.next = newNode;
    }
    return head;
}
```

If there was also a *tail* reference, in addition to *head*, *appending* to the list is very fast — it avoids having to scan the entire list until the *tail* node (the predecessor in this case) is found.

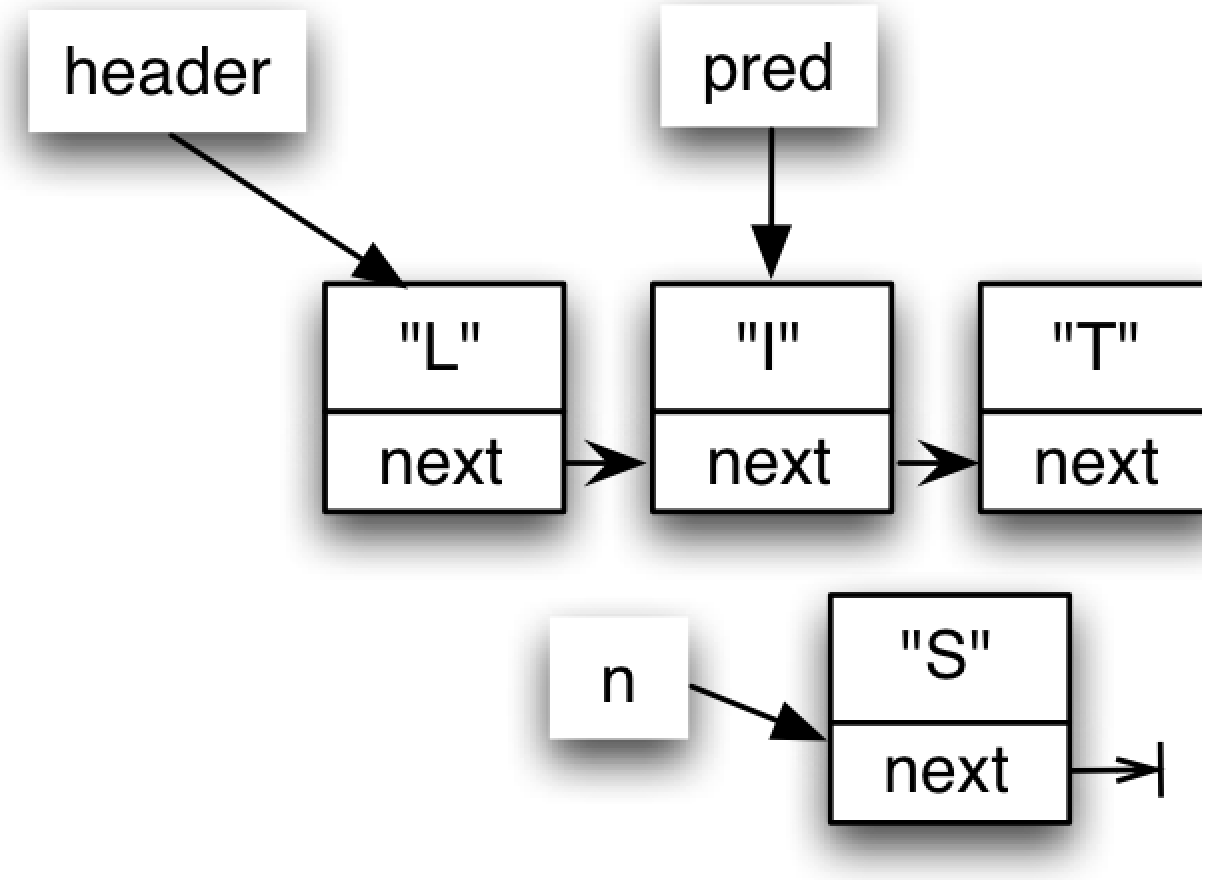
One thing to note is that we must have a reference to the *predecessor* node (unless we're adding to the beginning of the list, in which case the *head* reference changes), so it's often easier to write a method `insertAfter` that takes a new element to add after a given predecessor node.

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

    // Done!
    return n;
}
```

Except for the case of inserting a new element in the beginning of the list (a *special case*), we can simply find the predecessor node, and add the new element using `insertAfter` method.

Figure 2 shows the steps in inserting a new node in a list, given a predecessor node.


header
pred
"L"
next
"I"
next
"T"
next
n
"S"
next

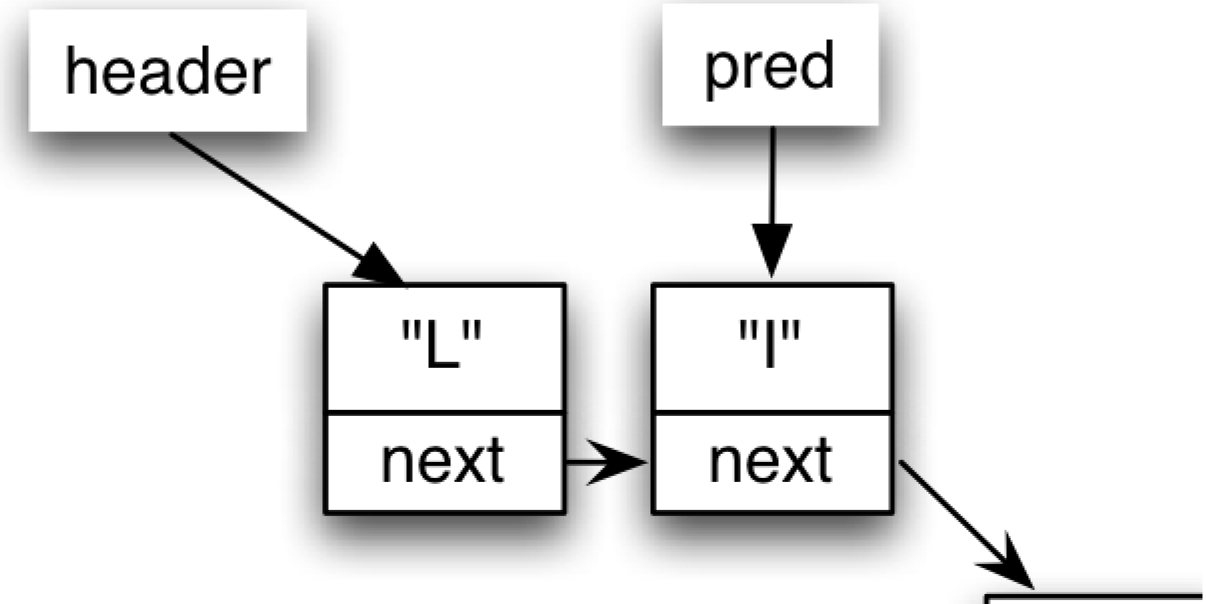

header
pred
"L"
next
"I"
next

Figure 2: Inserting a new element "S" into the list

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. And just like in insertion, removing the 1st node in the list is a *special case* — it does not have a predecessor, and removing it has the side-effect of changing head! We've already seen this particular special case — when inserting a new element in the beginning of the list. And, the special *special case* of removing the last node from a single-element list can also be avoided by the use of *dummy header* node.

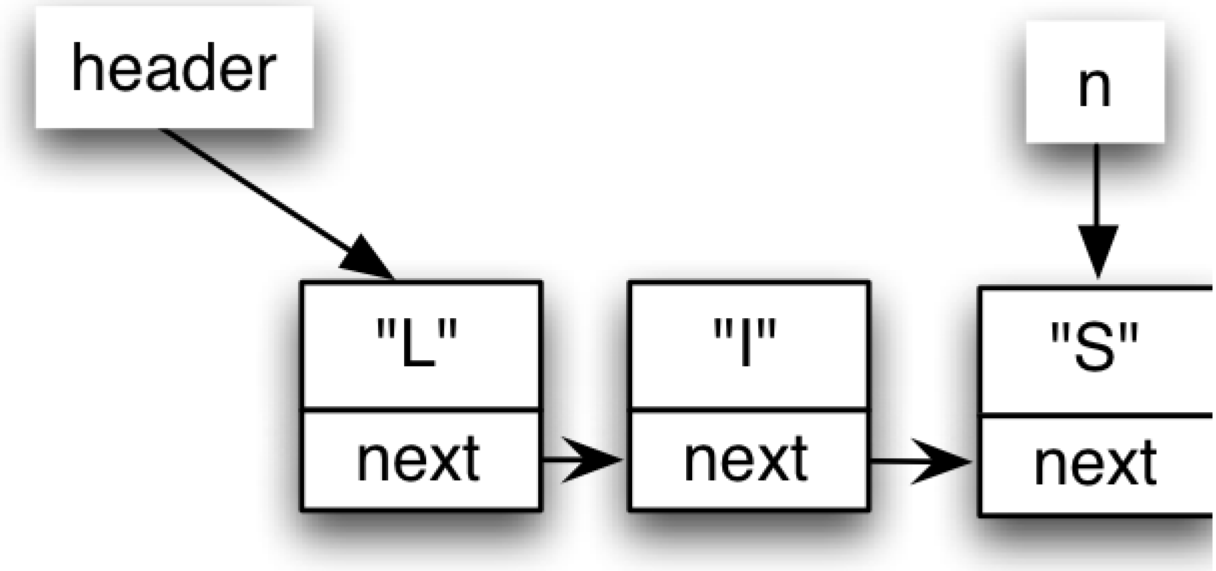
```
/**
 * Removes the element at the given index in the list.
 *
 * @param head  the reference to the header node of the list
 * @param size  the number of elements in the list
 * @param index the index of the element to remove
 * @return the reference to the header node, which will change when
 *         removing the element at the beginning.
 * @exception IndexOutOfBoundsException if the index is out-of-bounds.
 */
public static Node remove(Node head, int size, int index) {
    // Check for invalid index first. Should be between 0 and size-1.
    if (index < 0 || index >= size)
        throw new IndexOutOfBoundsException();

    // Reference to the removed node.
    Node removedNode = null;

    // Remember the special case -- from the beginning of the list, when
    // the head reference changes.
    if (index == 0) {
        removedNode = head;
        head = head.next;
    } else {
        // get the predecessor node first
        Node pred = nodeAt(index - 1);
        removedNode = pred.next;
        pred.next = removedNode.next;
    }

    // Help the GC
    removedNode.element = null;
    removedNode.next = null;
    return head;
}
```

Figure 3 shows the steps in removing a specific node a list, given its reference.


header
n
"L"
next
"I"
next
"S"
next

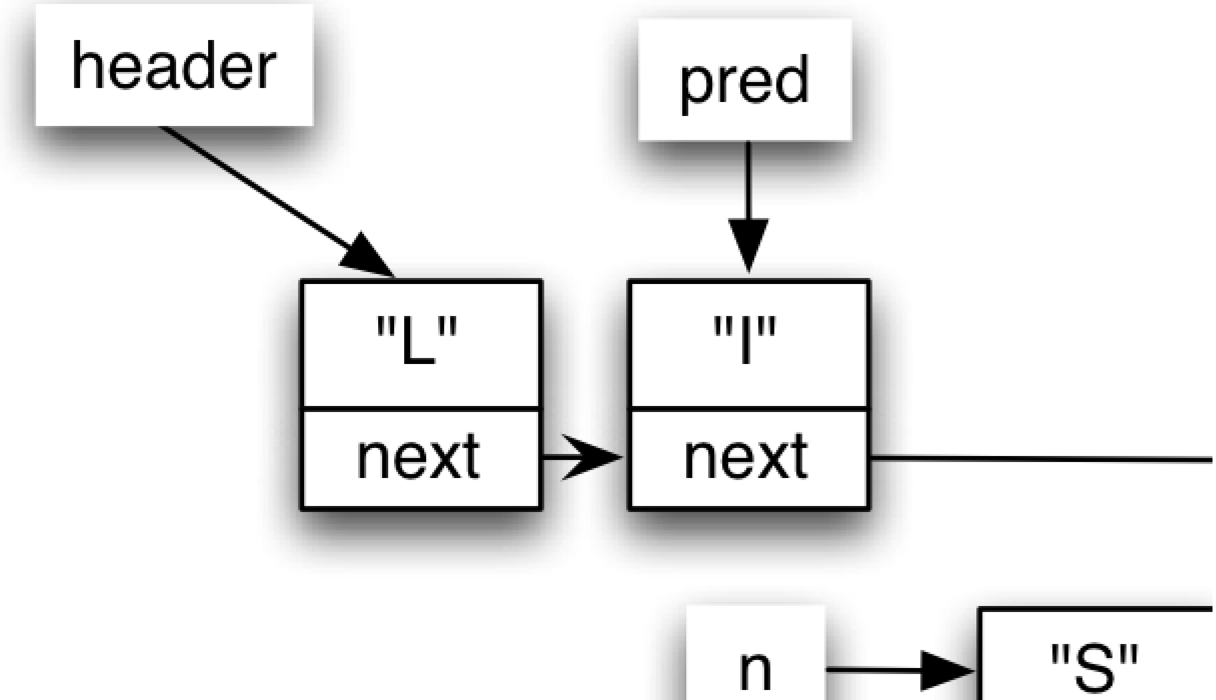

header
pred
"L"
next
"I"
next
n
"S"

Figure 3: Removing the node containing element "S" from the list

Back to Table of contents

Copying a list

Copying the elements of a *source* list to *destination* list is simply a matter of iterating over the elements of the source list, and inserting these elements at the end of the destination list.

```
/**
 * Copy the source list and return the reference to the copy.
 *
 * @param source the head reference of the source list
 * @return reference to the head of the copy
 */
public static Node copyList(Node source) {
    Node copyHead = null;
    Node copyTail = null;
    for (Node n = source; n != null; n = n.next) {
        Node newNode = new Node(n.element, null);
        if (copyHead == null) {
            // First node is special case - head and tail are the same
            copyHead = newNode;
            copyTail = copyHead;
        } else {
            copyTail.next = newNode;
            copyTail = copyTail.next;   // same as: copyTail = newNode;
        }
    }
    return copyHead;
}
```

Back to Table of contents

Reversing a list

Since a linked list does not support random access, it is difficult to reverse a list *in-place* without changing the head reference. Instead we'll create a new list with its own head reference, and copy the elements in the reverse order. This method does not modify the original list, so we can call it an *out-of-place* method.

```
/**
 * Reverses the list and returns the reference to the header node.
 *
 * @param list the list to reverse
 * @param size the number of elements in the list
 * @return reference to the head of the reversed list
 */
public static Node reverse(Node head, int size) {
    Node newHead = null;
    // We iterate over the nodes in the original list, and add a copy
    // of each node to the *beginning* of the new list, which reverses
    // the order.
    for (Node n = head; n != null; n = n.next) {
        Node newNode = new Node(n.next, null);

        // Add the node's copy to the beginning of the reversed list.
        newNode.next = newHead;
        newHead = newNode;
    }
    return newHead;
}
```

The problem with this approach is that it creates a copy of the whole list, just in the reverse order. We would like an *in-place* approach instead — re-order the links instead of copying the nodes! That would of course change the original list, and would have a new head reference (which is the reference to the tail node in the original list).

```
/**
 * Reverses the list and returns the reference to the header node.
 *
 * @param list the list to reverse
 * @param size the number of elements in the list
 * @return reference to the head of the reversed list
 */
public static Node reverse(Node head, int size) {
    Node newHead = null;

    // We iterate over the nodes in the original list, and add each
    // node to the *beginning* of the new list, which reverses the
    // order. Have to be careful when we iterate -- have to save the
    // reference to the *next* node before changing the next link.
    Node n = head;
    while (n != null) {
        Node nextNode = n.next;         // need to do this!

        // now change the link so that node 'n' is placed in the
        // beginning of the list
        n.next = newHead;
        newHead = n;

        // Can't do n = n.next because the link is changed. That's why
        // we saved the reference to the next node earlier, so we can
        // use that now.
        n = nextNode;
    }

    return newHead;
}
```

Back to Table of contents

Rotating a list left

Rotating a list left is much simpler than rotating an array — the 2nd node becomes the new *head*, and the 1st becomes the new *tail* node. We don't have to actually *move* the elements, it's just a matter of re-arranging a few links!

```
/**
 * Rotates the given list left by one element
 *
 * @param head the list to rotate left
 * @param size the number of elements in the list
 * @return reference to the head of the rotated list
 */
public static Node rotateLeft(Node head, int size) {
    // Bug alert - does it work for empty lists?
    Node oldHead = head;
    head = head.next;

    // Now append the old head to the end of this list. Find the tail,
    // and add the old head after the tail.
    Node tail = head;
    while (tail.next != null)
        tail = tail.next;

    // Now add after tail
    tail.next = oldHead;
    oldHead.next = null;

    return head;
}
```

We can also use *circular* or *cyclic* lists.

Back to Table of contents

Rotating a list right

Rotating a list right is almost the same as rotating left — the current *tail* becomes the new *head*.

```
/**
 * Rotates the given list right by one element
 *
 * @param head the list to rotate right
 * @param size the number of elements in the list
 * @return reference to the head of the rotated list
 */
public static Node rotateRight(Node head, int size) {
    // Bug alert - does it work for empty lists?

    // Need to find tail and it's predecessor (do you see why?)
    Node p = null;
    Node q = head;
    while (q.next != null) {
        p = q;
        q = q.next;
    }

    // Now q points to the tail node, and p points to it's predecessor
    // First make tail the new head
    q.next = head;
    head = q;

    // make p the tail
    p.next = null;

    return head;
}
```

We can also use *circular* or *cyclic* lists.

Back to Table of contents

You can look at the example LinkedList.java class to see how these can be implemented. Run the `LinkedList.main()` to see the output.