# ITI 1121. Introduction to Computing II [*]

Marcel Turcotte
School of Electrical Engineering and Computer Science

Version of March 17, 2013

**Abstract**

• Abstract Data Types (ADTs):
  – List

[*]These lecture notes are meant to be looked at on a computer screen. Do not print them unless it is necessary.

## Definitions

A **List** is a linear **abstract data type** that places no restrictions on accessing the data; inspections, insertions and deletions can occur at any position.

The basic operations of a **List** are:

**int size():** returns the number of elements, the length of an empty list is 0;

**E get( int index ):** an access method that allows to inspect the content of any position. What is the index of the first element, 0 or 1? Similarly to arrays, the first element is found at position 0;

**add( int index, E o ):** an element can be added at any position of a list;

**remove( int index ):** similarly, an element can be removed by position or by content.

**List**s are more general than **Stack**s and **Queue**s; which can be implemented with help of a **List**.

```java
public interface List<E> {

    public abstract void add( int index, E elem );
    public abstract boolean add( E elem );

    public abstract E remove( int index );
    public abstract boolean remove( E o );

    public abstract E get( int index );
    public abstract E set( int index, E element );

    public abstract int indexOf( E o );
    public abstract int lastIndexOf( E o );

    public abstract boolean contains( E o );

    public abstract int size();
    public abstract boolean isEmpty();
}
```

$\Rightarrow$ This interface declares only a subset of the methods listed by java.util.List;

## Implementations

- ArrayList;
- LinkedList;
  - Singly linked list;
  - Doubly linked list;
  - Dummy node;
  - Iterative list processing (Iterator);
  - Recursive list processing.

New concepts will be introduced as needed in order to improve the efficiency of the implementation.

**Efficiency** is measured in terms of **execution speed** or **memory usage**, we will mainly focus on execution speed.

## Singly Linked List

The simplest of the linked list implementations is the singly linked list (**SinglyLinkedList**). Our implementation will use a static nested class, called **Node** here. Obviously, the type of the reference **value** is **E**.

Each node of the list holds a value and is connected to its successor.

```
private static class Node<E> {
    private E value;
    private Node<E> next;
    private Node( E value, Node<E> next ) {
        this.value = value;
        this.next = next;
    }
}
```

The class **SinglyLinkedList** has an instance variable that designates the first element of the list, we'll call this variable **head**.

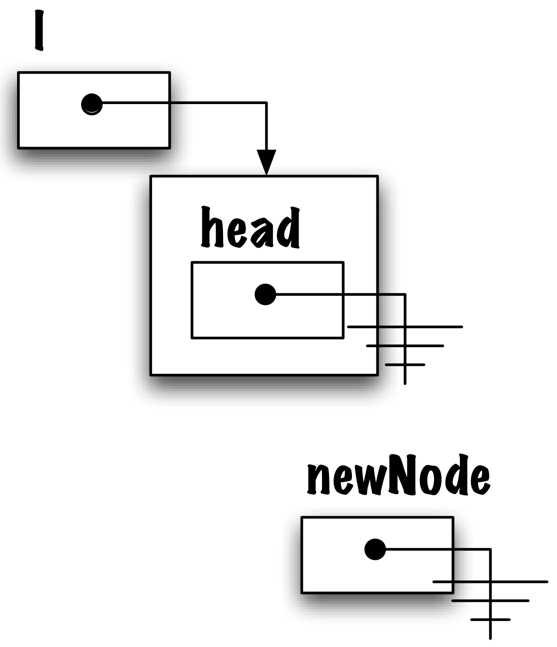
$\Rightarrow$ This nested class is sometimes called **Elem** or **Entry**.

## addFirst( E o ) (1/2)

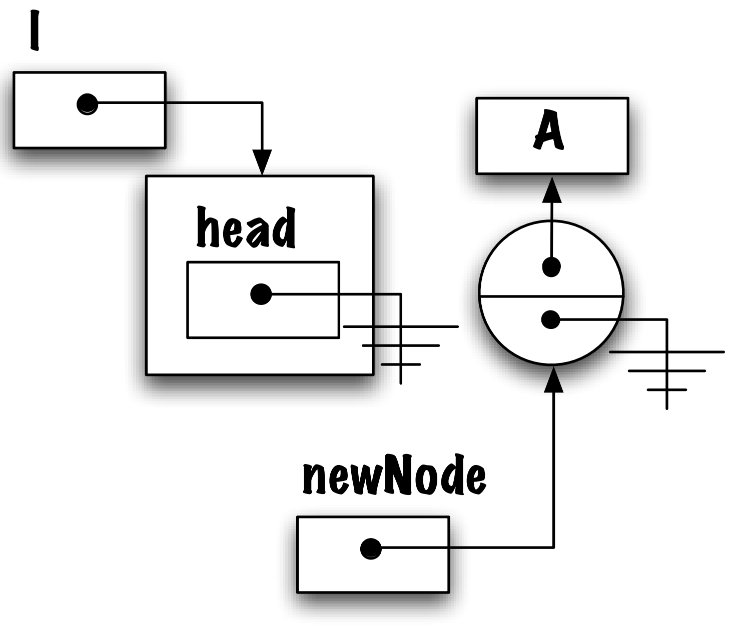
Inserting an element at the front of a list involves 1) creating a new node and 2) adding the node to the list. We distinguish two cases, the list is empty or not.

```
public void addFirst( E o ) {

    Node<E> newNode = new Node<E>( o, null );

    if ( head == null ) {
        head = newNode;
    } else {
        newNode.next = head;
        head = newNode;
    }
}
```

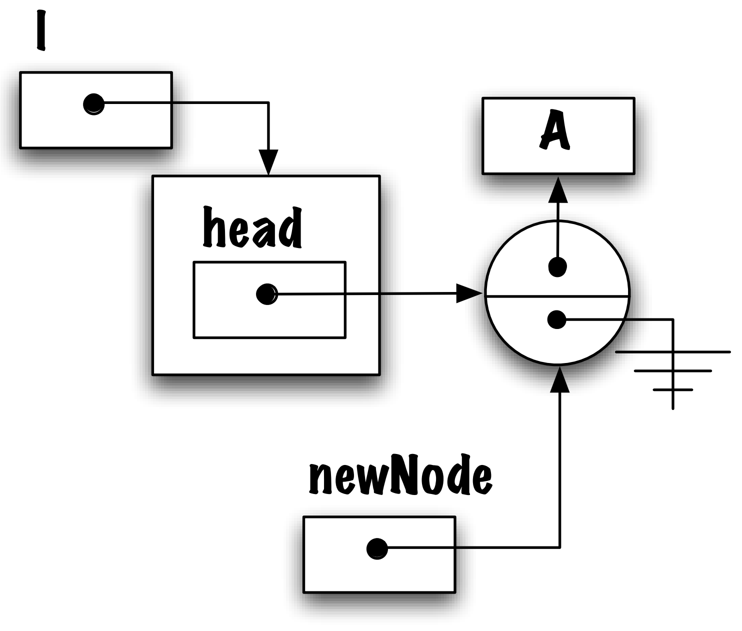
## Special case: addFirst( E o ) (1/2)



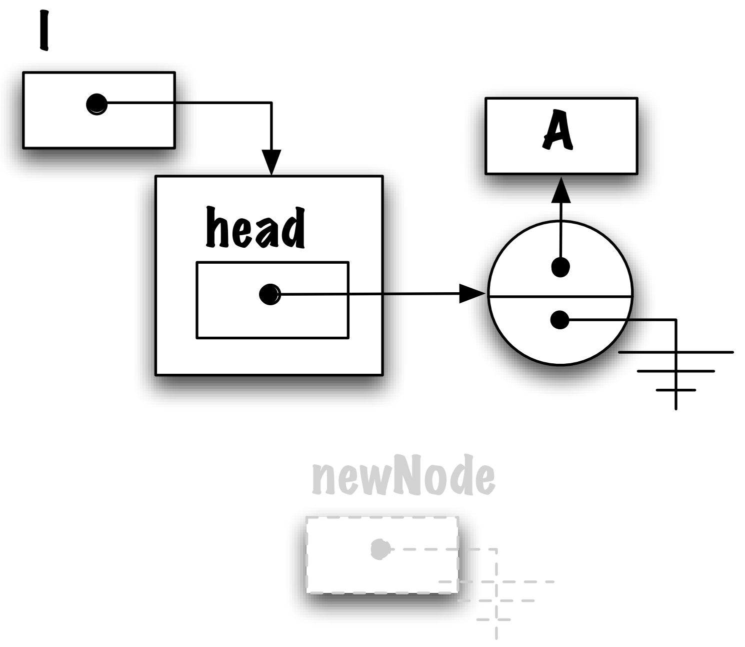
## Special case: addFirst( E o ) (1/2)



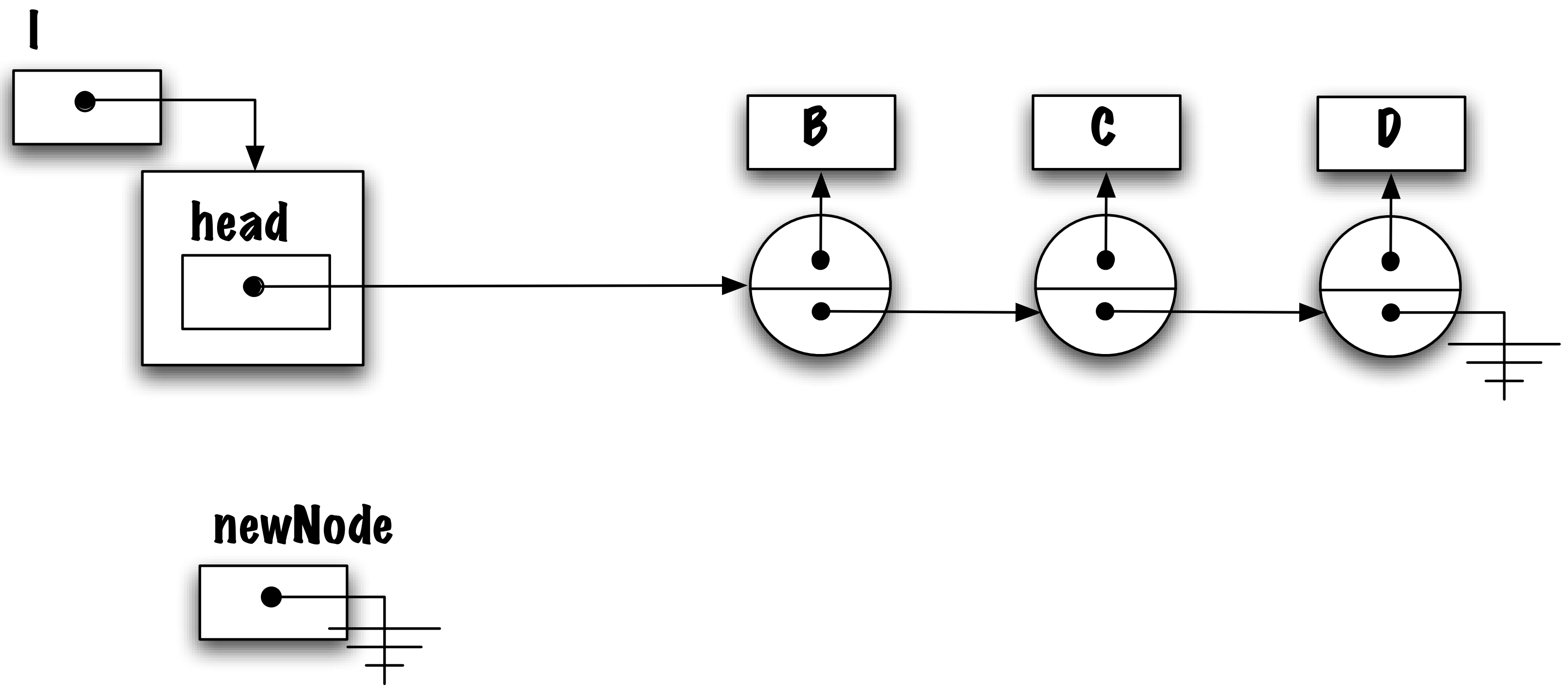
## Special case: addFirst( E o ) (1/2)



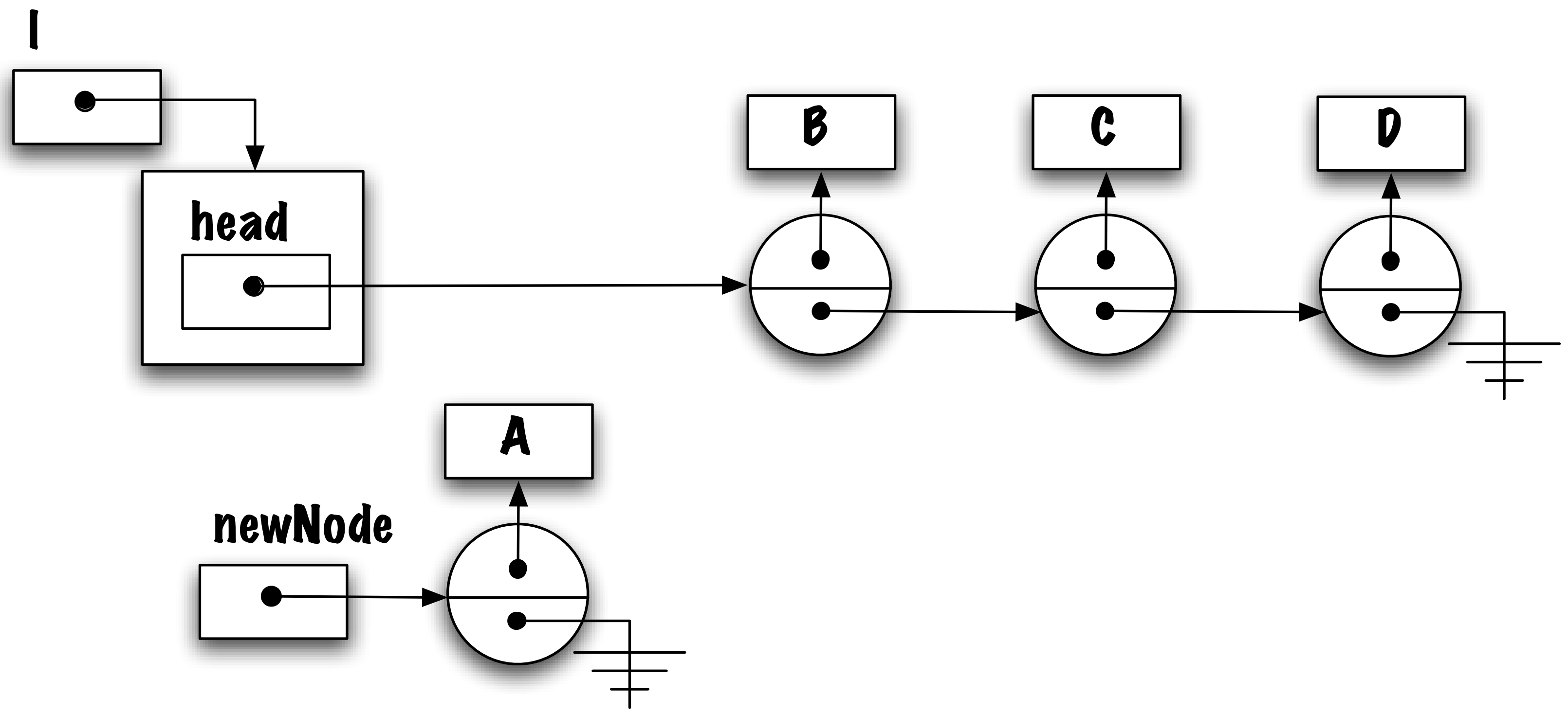
## Special case: addFirst( E o ) (1/2)



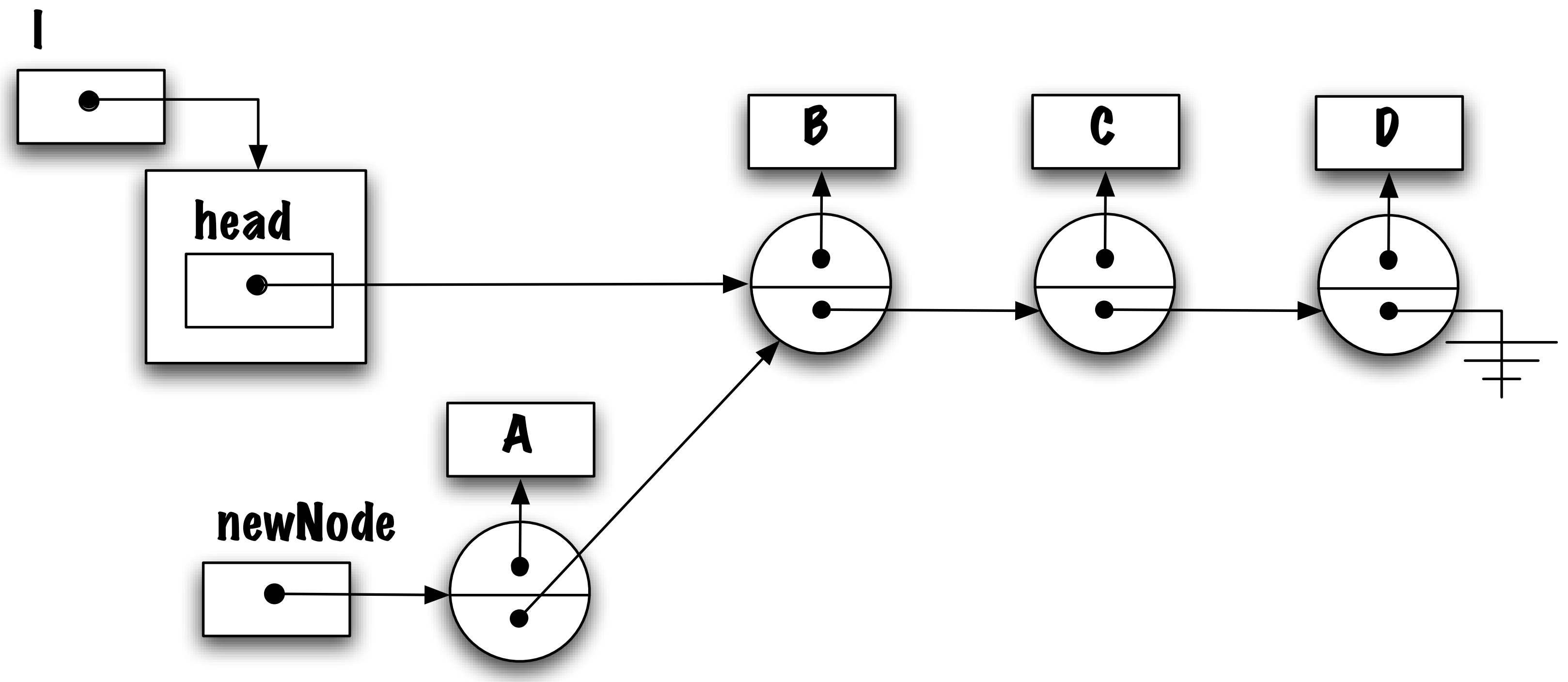
## General case: addFirst( E o ) (1/2)



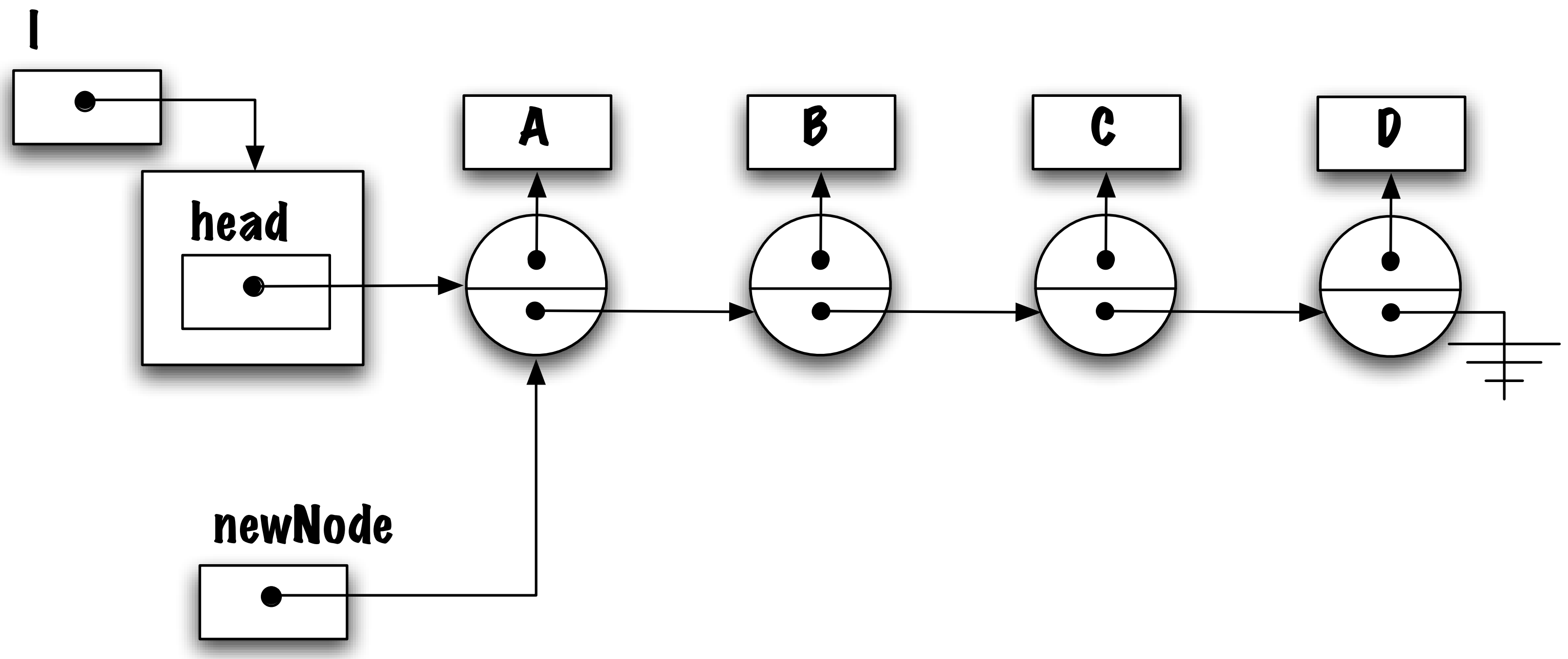
# General case: addFirst( E o ) (1/2)



## General case: addFirst( E o ) (1/2)



## General case: addFirst( E o ) (1/2)



Do we need to make a distinction between the an empty list and a list that contains some elements?

How about this?

```
public void addFirst( E o ) {
    head = new Node<E>( o, head );
}
```

Does it work for both cases (special and general)?

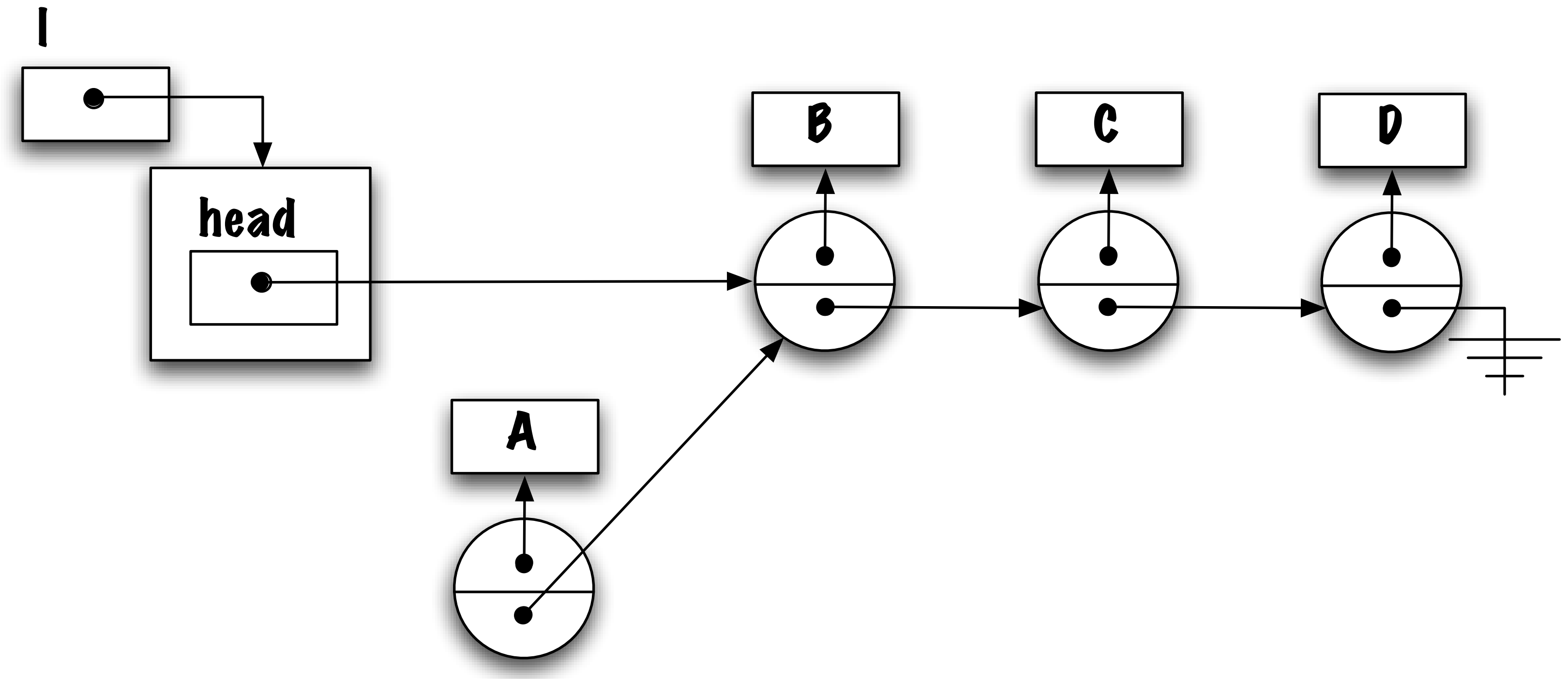
Yes, it does. Why?

It works because Java evaluates the right-hand side of the "=" first.

## addFirst( E o ) (2/2)

Evaluating the right-hand side first.

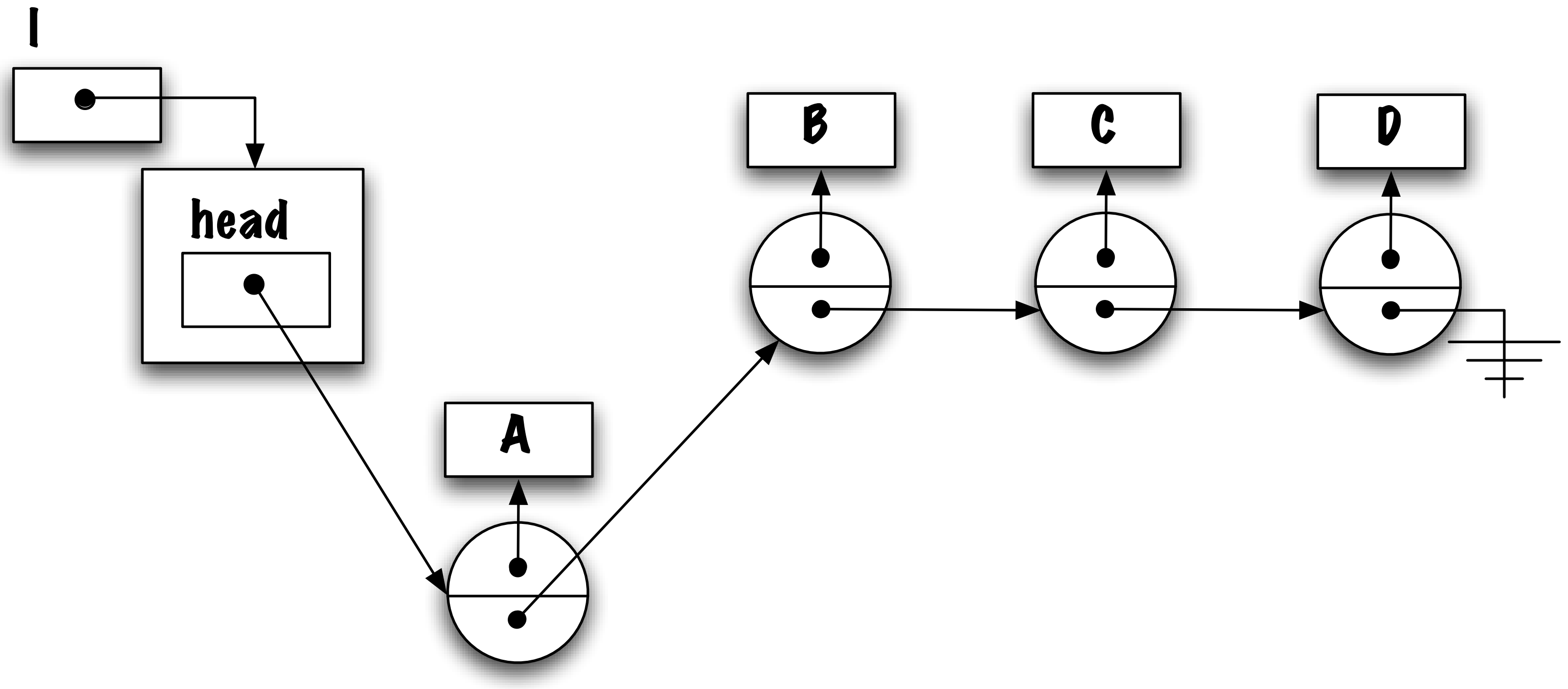
```
head = new Node<E>( o, head );
```



## addFirst( E o ) (2/2)

The result (a reference to the newly created **Node**) is assigned to the variable **head**.

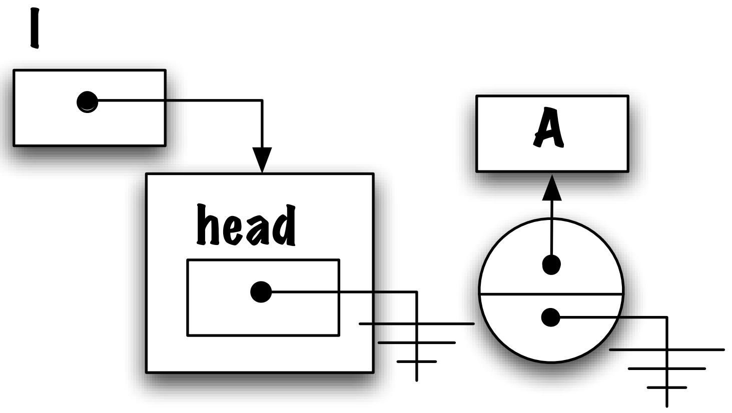
```
head = new Node<E>( o, head );
```



## addFirst( E o ) (2/2)

Similarly, evaluating the right-hand side first, here **head** is **null**, **null** is assigned to the instance variable **next** of the **Node**.

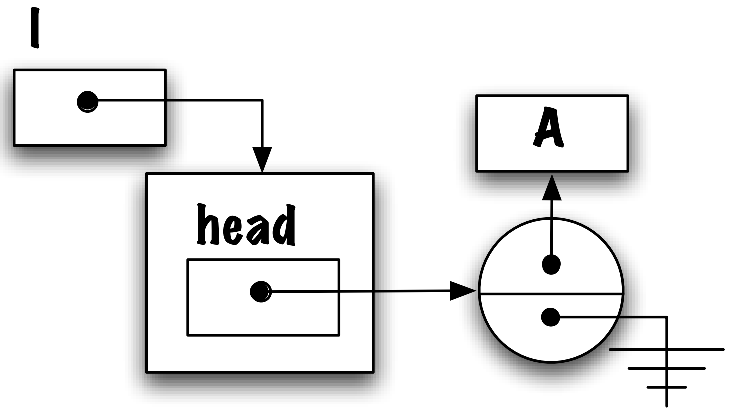
```
head = new Node<E>( o, head );
```



## addFirst( E o ) (2/2)

The result (a reference to the newly created **Node**) is assigned to the variable **head**.

```
head = new Node<E>( o, head );
```



## add( E o ) (1/3)

Adding an element at the end requires locating the end of the list (traversal) and inserting an element as its successor.

```
public void add( E o ) {

    Node<E> newNode = new Node<E>( o, null );

    Node<E> p = head;
    while ( p != null ) {
        p = p.next;
    }
    p.next = newNode;
}
```

⇒ Something is wrong with the method, what is it?

## add( E o ) (1/3)

The method **add** exists the while loop when **p** becomes **null**. Hence, a **NullPointerException** will be thrown when executing **p.next = newNode**.

```
public void add( E o ) {

    Node<E> newNode = new Node<E>( o, null );

    Node<E> p = head;
    while ( p != null ) {
        p = p.next;
    }
    p.next = newNode;
}
```

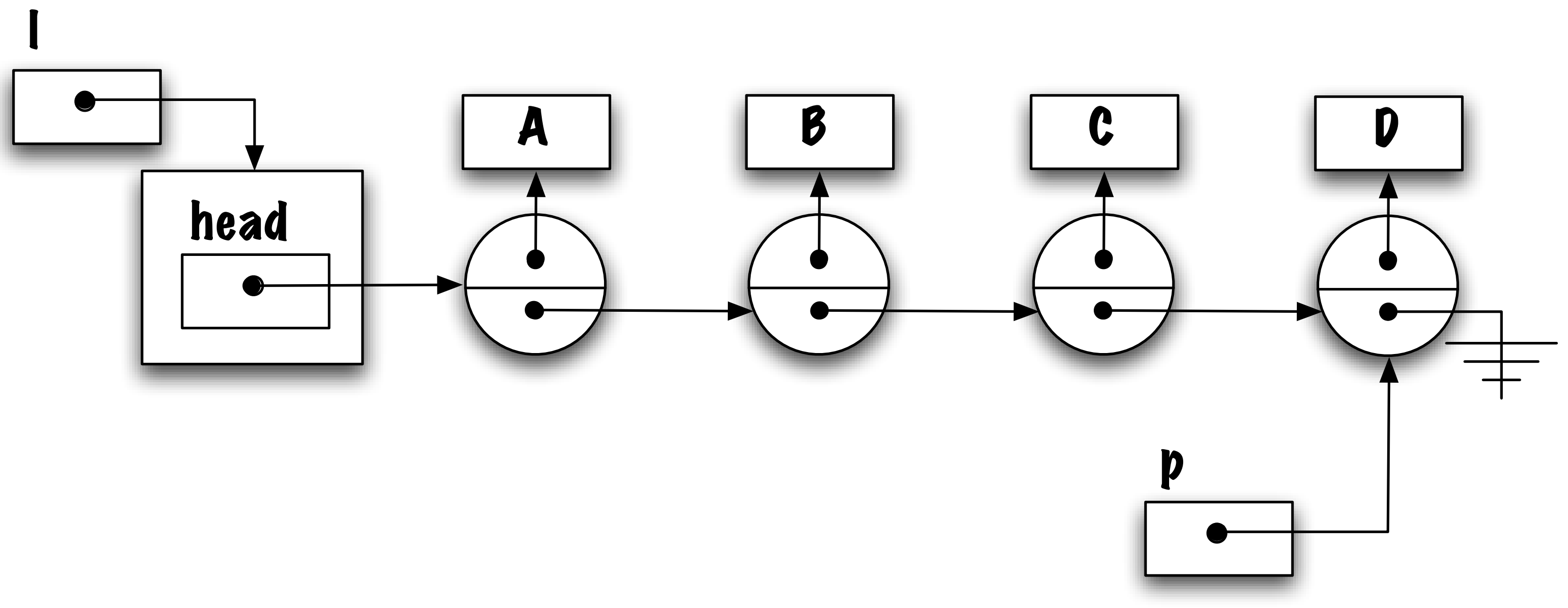
## add( E o ) (1/3) (take 2)

The method **add** exists the while loop when **p** becomes **null**. Hence, a **NullPointerException** will be thrown when executing **p.next = newNode**.

```
public void add( E o ) {

    Node<E> newNode = new Node<E>( o, null );

    Node<E> p = head;
    while ( p != null ) {
        p = p.next;
    }
    p = newNode;
}
```

⇒ What do you think?

## add( E o ) (2/3)



How can the iteration be stopped on the last element?

It stops when `p.next == null` is true.

## add( E o ) (2/3)

```
public void add( E o ) {

    Node<E> newNode = new Node<E>( o, null );

    Node<E> p = head;
    while ( p.next != null )
        p = p.next;

    p.next = newNode;
}
```

⇒ Something is still wrong, what is it?

## add( E o ) (2/3)

What would occur if the list was empty?

```
public void add( E o ) {

    Node<E> newNode = new Node<E>( o, null );

    Node<E> p = head;
    while ( p.next != null )
        p = p.next;

    p.next = newNode;
}
```

1) what would occur for `p.next`? 2) How about `head`? `head` has to be modified as well.

## add( E o ) (3/3)

```
public void add( E o ) {

    Node<E> newNode = new Node<E>( o, null );
    if ( head == null ) {
        head = newNode;
    } else {
        Node<E> p = head;
        while ( p.next != null )
            p = p.next;
        p.next = newNode;
    }
}
```

⇒ The empty list is often a special case!

## removeFirst()

```
public E removeFirst() {

    // pre-condition?

    E savedValue = head.value;

    Node<E> first = head;
    head = head.next;

    first.next = null;  // memory ‘‘scrubbing’’
    first.value = null;

    return savedValue;
}
```

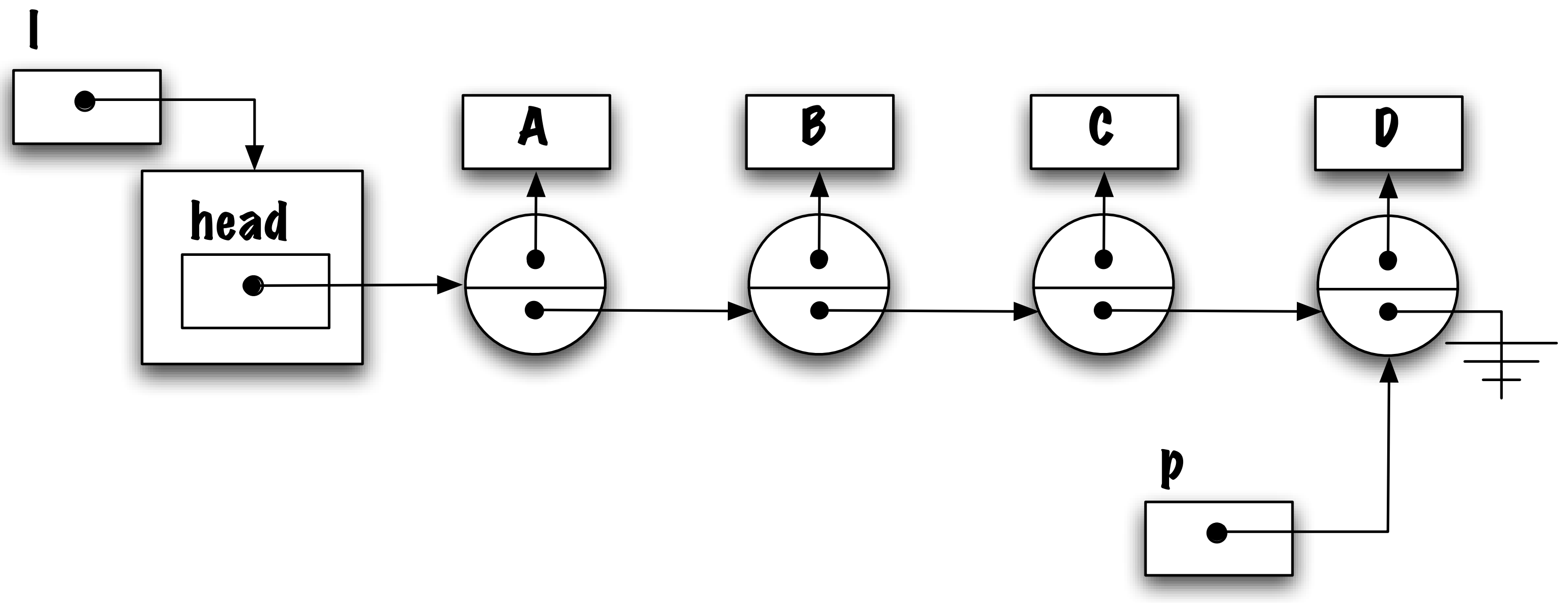
## removeLast() (1/4)

This will certainly involve a traversal and, as we have seen with the method **addLast()**, we have to be careful about where to stop!

```
Node<E> p = head;
while ( p.next != null ) {
    p = p.next;
}
```

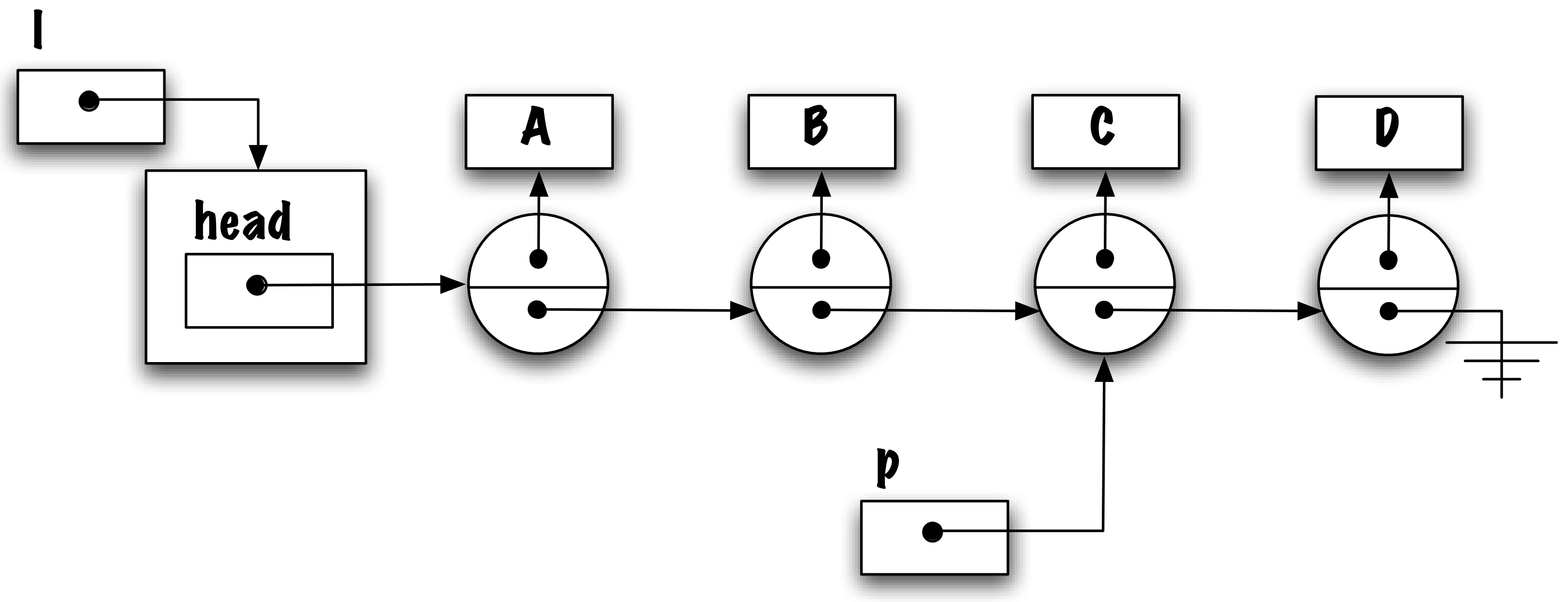
$\Rightarrow$ How is that?

# removeLast() (1/4)



The local variable **p** refers to the last element of the list, how can one change the pointer **next** of the previous element?

## removeLast() (1/4)



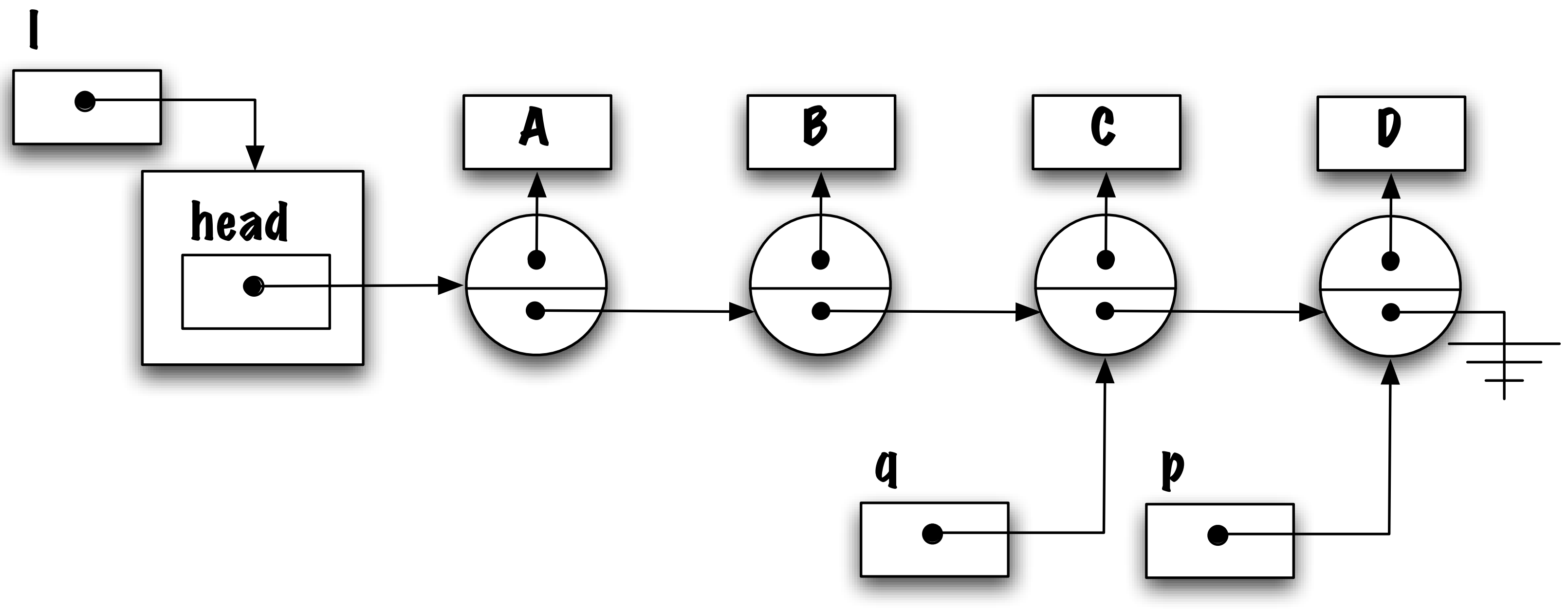
The iteration should be stopped when **p** designates the second element from the end of the list

## removeLast() (2/4)

We could have a second loop that would stop when its **q.next** would be equal to **p**.

```
Node<E> p = head;
while ( p.next != null ) {
    p = p.next;
}


Node<E> q = head;
while ( q.next != p ) {
    q = q.next;
}
```

## removeLast() (2/4)



⇒ What do you think?

## removeLast() (3/4)

Instead, we would have stopped the first loop earlier. How?

```
Node<E> p = head;
while ( p.next.next != null ) {
    p = p.next;
}
res = p.next.value;
p.next = null;
```

⇒ Are there special cases?

## removeLast() (4/4)

```
public E removeLast() {
    // pre-condition?
    E savedValue;
    if ( head.next == null ) {
        savedValue = head.value;
        head = null;
    } else {
        Node<E> p = head;
        while ( p.next.next != null ) {
            p = p.next;
        }
        Node<E> last = p.next;
        savedValue = last.value;
        last.value = null;
        p.next = null;
    }
    return savedValue;
}
```

## **remove( E o )**

Accessing elements by content.

Returns **true** if **o** was successfully removed and **false** otherwise.

Give an outline of the implementation.

1. Traversing the list;
2. Stopping criteria?
3. Removal.

## remove( E o )

How about this?

```
public boolean remove( E o ) {

        Node<E> p = head;

        while ( p != null && ! p.value.equals( o ) ) {
            p = p.next;
        }
        Node<E> toDelete = p;

        ...

    return true;
}
```

```
public boolean remove( E o ) {

    Node<E> toDelete = null;
    Node<E> p = head;

    while ( p.next != null && ! p.next.value.equals( o ) ) {
      p = p.next;
    }

    toDelete = p.next;
    p.next = toDelete.next;
    toDelete.value = null;
    toDelete.next = null;

    return true;
}
```

Problems? What if the element has not been found?

```
public boolean remove( E o ) {
    Node<E> toDelete = null;
    Node<E> p = head;

    while ( p.next != null && ! p.next.value.equals( o ) ) {
        p = p.next;
    }
    if ( p.next == null ) {
        return false;
    }
    toDelete = p.next;
    p.next = toDelete.next;
    toDelete.value = null;
    toDelete.next = null;
    return true;
}
```

## What should be done with the empty list?

```
ublic boolean remove( E o ) {
    if ( head == null )
        return false;
    Node<E> toDelete = null;
    if ( head.value.equals( o ) ) {
        toDelete = head;
        head = head.next;
    } else {
        Node<E> p = head;
        while ( p.next != null && ! p.next.value.equals( o ) ) {
        p = p.next;
        }
        if ( p.next == null ) {
        return false;
        }
        toDelete = p.next;
        p.next = toDelete.next;
    }
    toDelete.value = toDelete.next = null;
    return true;
}
```

## get( int pos )

Accessing the elements by position.

To be consistent with the array-based implementation, let's designate the first element 0 — from the point of view of a user of the class, it does not matter if the class uses an array or a linked list to store the elements.

This certainly involves traversing the list!

But requires stopping early.

How to determine when to stop?

Yes, we simply count the number of nodes that have been visited.

Therefore, we need a counter.

## get( int pos )

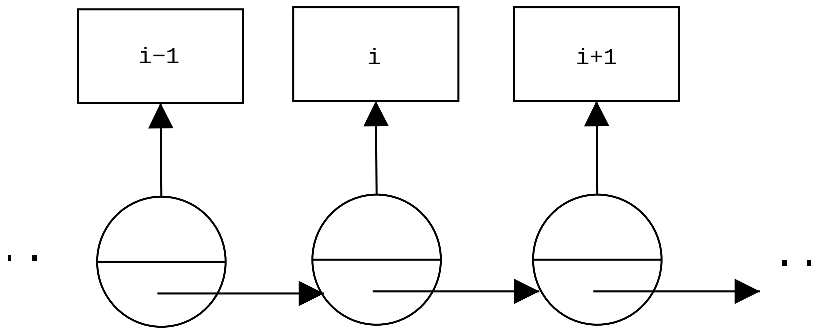
```
public E get( int pos ) {

    Node<E> p = head;

    for ( int i=0; i<pos; i++ ) {
        p = p.next;
    }

    return p.value;
}
```

What is missing? Exercise, make the necessary changes to handle the cases where the value of **pos** is not valid.

## remove( int pos )

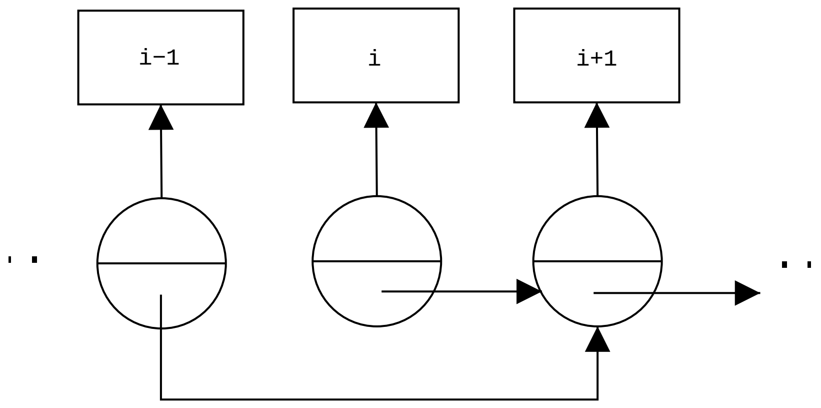
Removing the element at position **pos** has a lot in common with accessing an element by position; this involves traversing the list up to a certain element.

Let's consider the general case first. Imagine that element $i$ needs to be removed. Draw the memory diagram that represents this situation.



⇒ Which .next needs to be modified?

The instance variable **next** of the previous node needs to be changed!



Assuming that **p** designates the previous node, the following statement would implement the required change,

```
p.next = p.next.next;
```

⇒ we should also take care of "scrubbing" the memory.

## remove( int pos ) 1/2

```
public E remove( int pos ) {

    E savedValue;

    Node<E> p = head;
    for ( int i=0; i<( pos-1 ); i++ ) {
        p = p.next;
    }
    Node<E> toBeRemoved = p.next;
    savedValue = toBeRemoved.value;

    p.next = p.next.next;

    toBeRemoved.value = null;
    toBeRemoved.next = null;
    return savedValue;
}
```

⇒ is the method general enough? is it working for all cases?

## remove( int pos ) 2/2

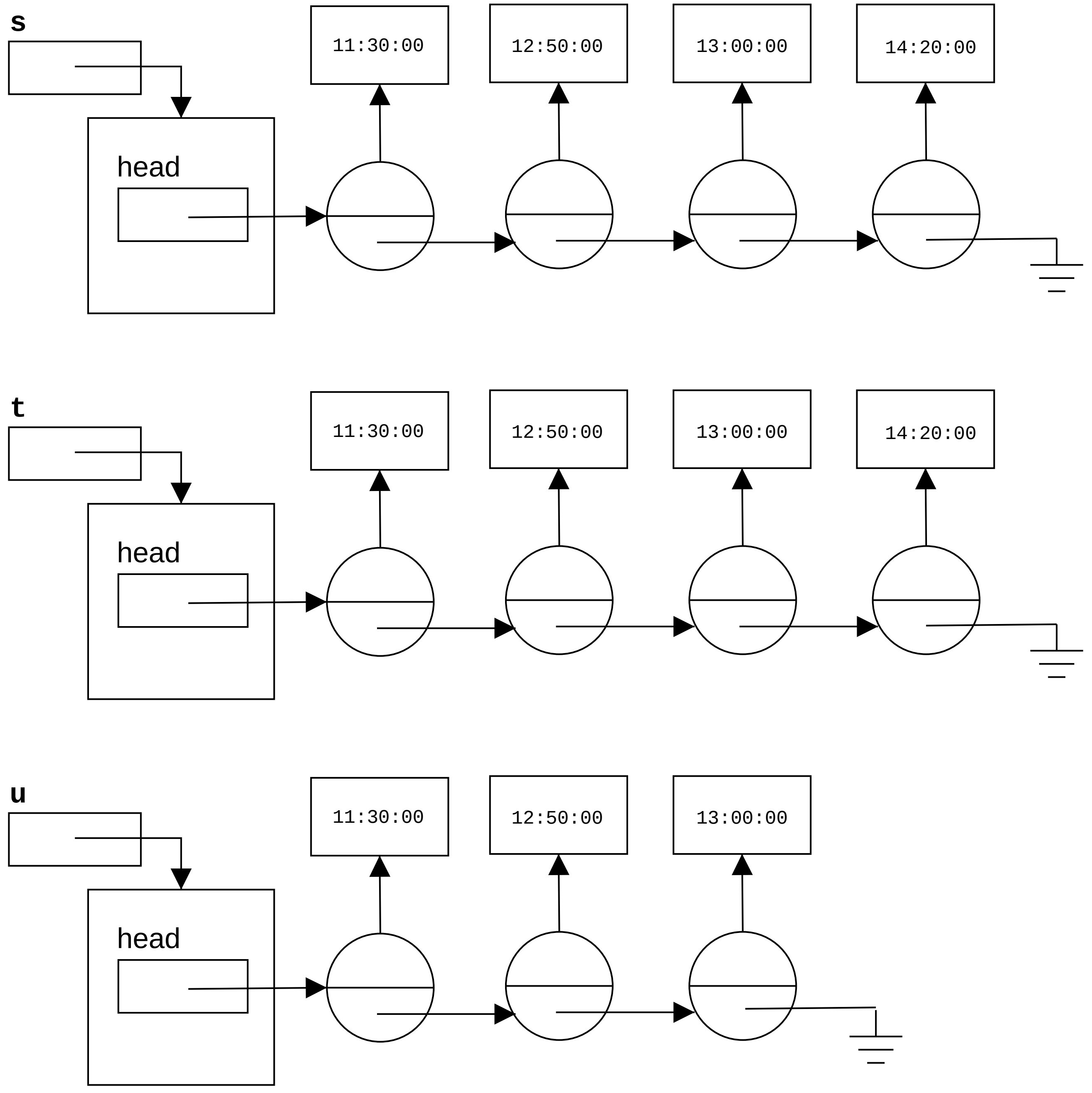
```
public E remove( int pos ) {
    E savedValue;
    Node<E> toBeRemoved;
    if ( pos == 0 ) {
        toBeRemoved = head;
        head = head.next;
    } else {
        Node<E> p = head;
        for ( int i=0; i<( pos-1 ); i++ ) {
            p = p.next;
        }
        toBeRemoved = p.next;
        p.next = p.next.next;
    }
    savedValue = toBeRemoved.value;
    toBeRemoved.value = null;
    toBeRemoved.next = null;
    return savedValue;
}
```

$\Rightarrow$ Removing the element at position 0 is a special case, it involves modifying the **head** reference (instead of **next** of the previous element).

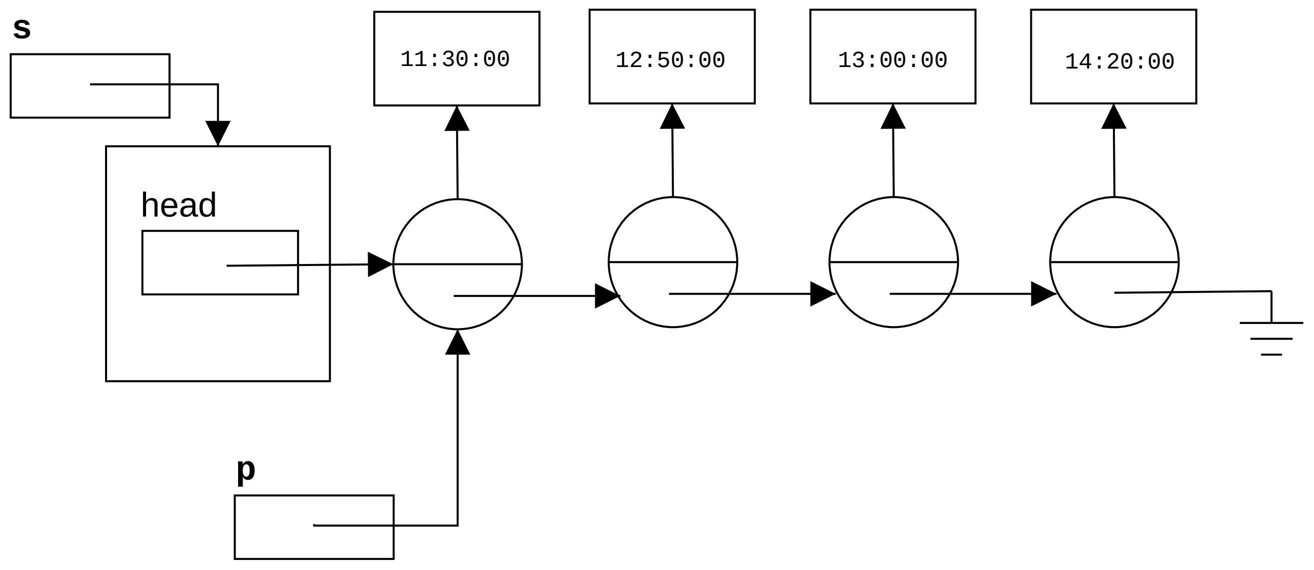
## Implementing the method equals

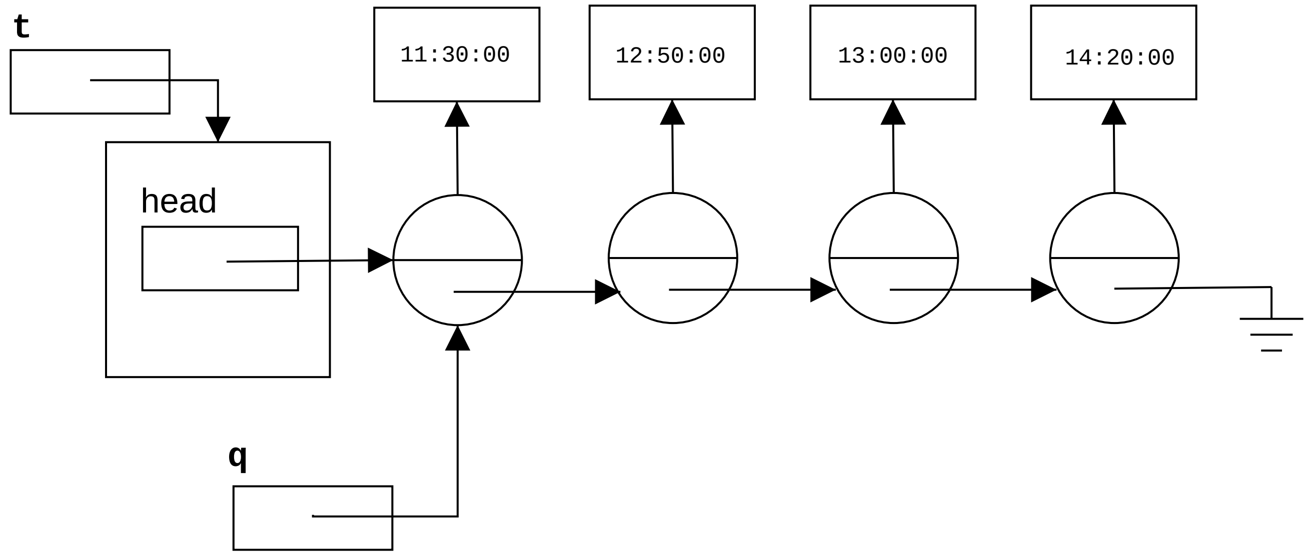
We would like to implement a method that would return **true** if two lists have an equivalent content, i.e. contain equivalent objects, in the same order.

This also means that the lists must be of the same length.

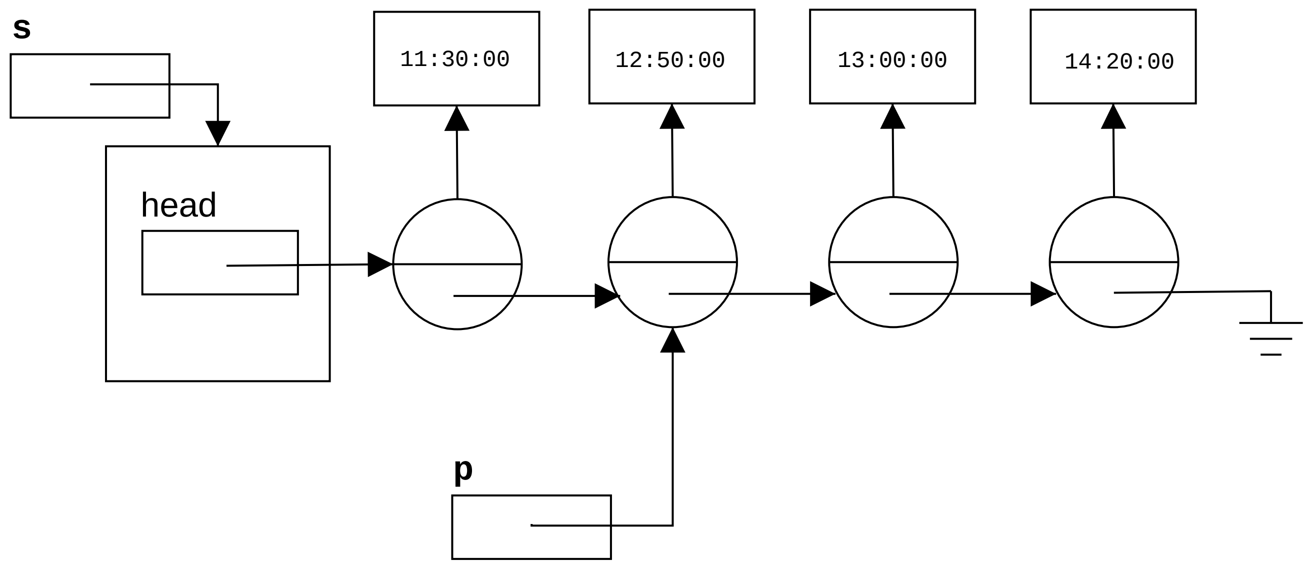

s
11:30:00
12:50:00
13:00:00
14:20:00
head
t
11:30:00
12:50:00
13:00:00
14:20:00
head
u
11:30:00
12:50:00
13:00:00
head

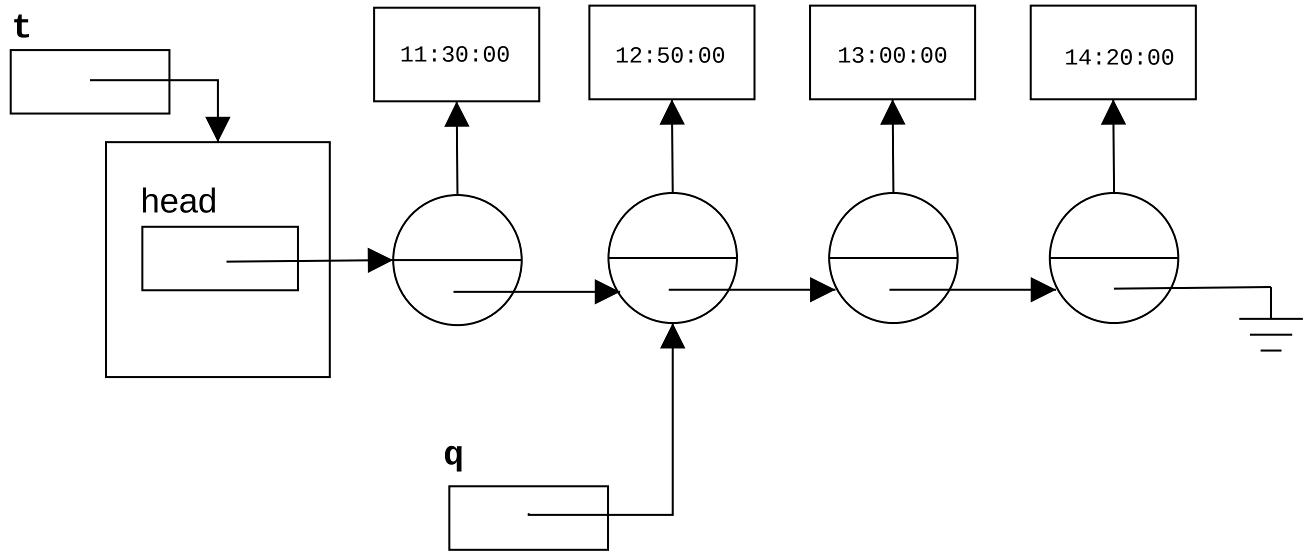
```
s.equals( t ); // true!

s.equals( u ); // false!
```



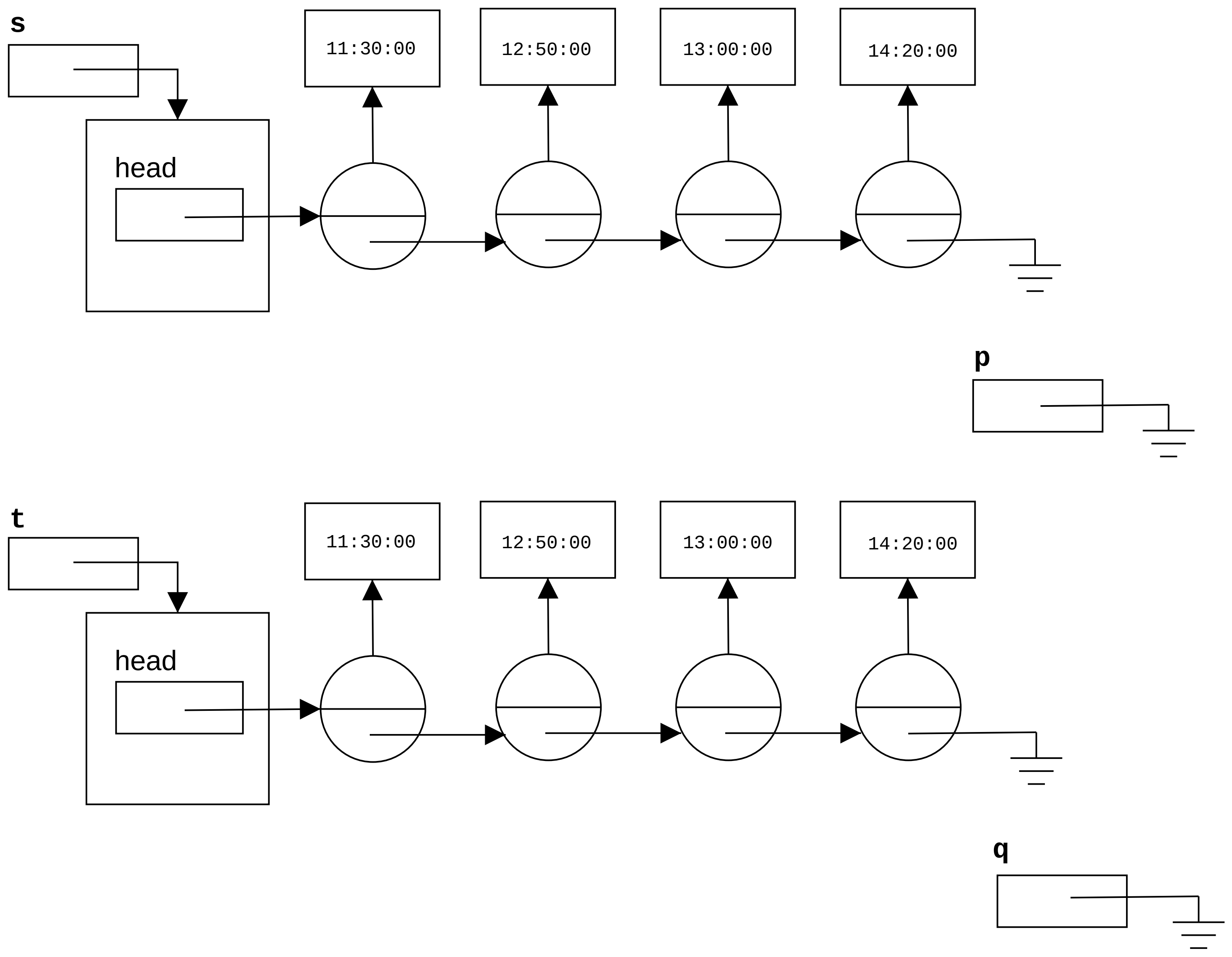


$\Rightarrow$ What's needed?





$\Rightarrow$ `p = p.next` and `q = q.next`



⇒ Eventually, the end of one or both lists is reached.

## The 9 steps approach

Manipulating references requires a meticulous approach — otherwise elements are lost, sometimes the whole of the list, or null pointer exceptions are thrown!

1. identify inputs/outputs
2. BEFORE/AFTER memory diagrams
3. Generalization
4. Find all specific cases
5. Handling special cases
6. Writing test cases
7. Writing blocks of code
8. Writing the method(unoptimized)
9. Optimization

To illustrate this approach let's implement the following method:

```
public E removeFirst();
```

This method removes the first element of the list and returns it to the caller method.

## Step 1: identify inputs/outputs

Input: this is an instance method that changes the state of the object, here **SinglyLinkedList**.

Output: it returns the E element that was remove.

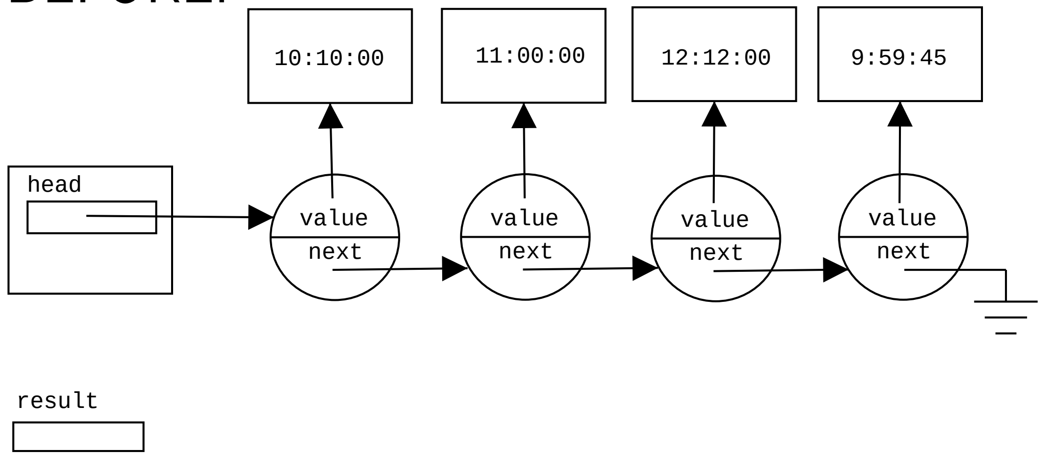
## Step 2: BEFORE/AFTER memory diagrams

Draw the memory diagrams BEFORE and AFTER for a typical/general case.

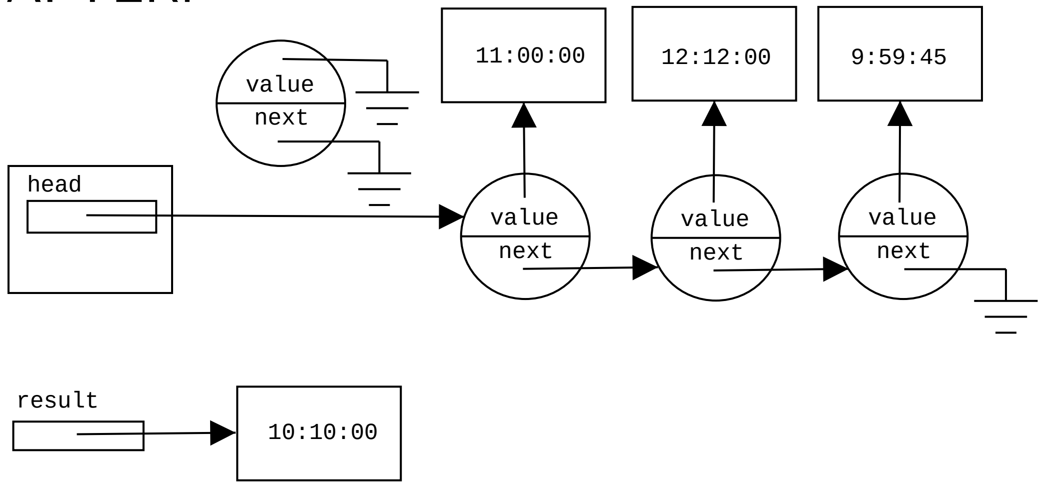
The BEFORE diagram illustrates the state of the object (and all input variables) just before the method is called.

Similarly, the AFTER diagram illustrates the state of the object (and output variables immediately after the method was called).

BEFORE:



AFTER:



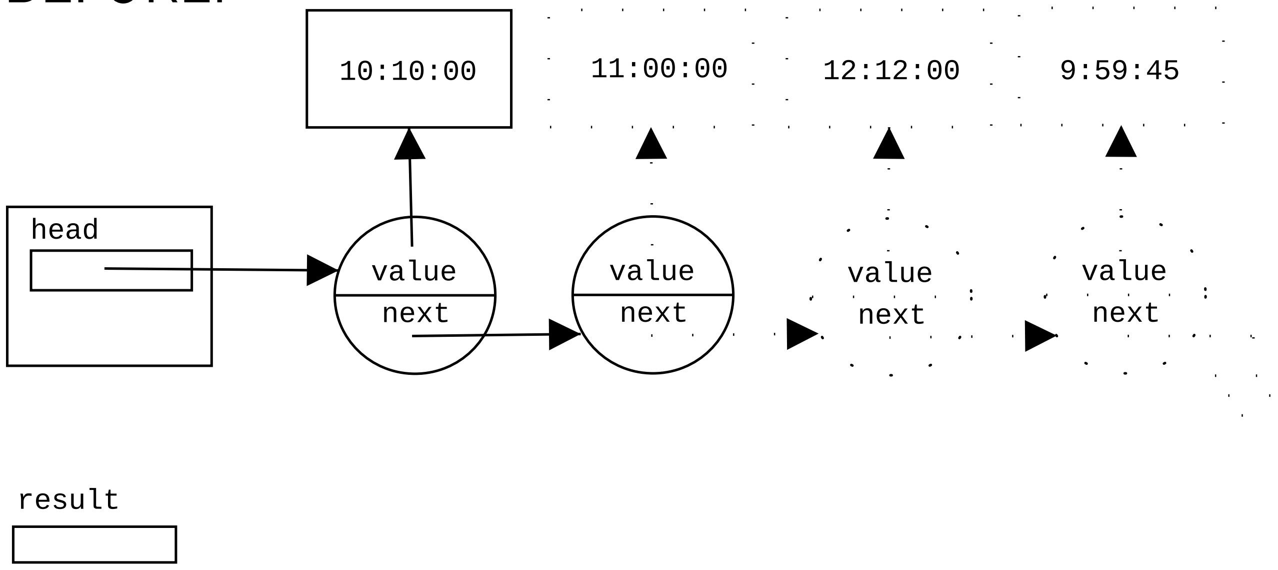
## Step 3: Generalization

Generalization of the BEFORE/AFTER diagrams.

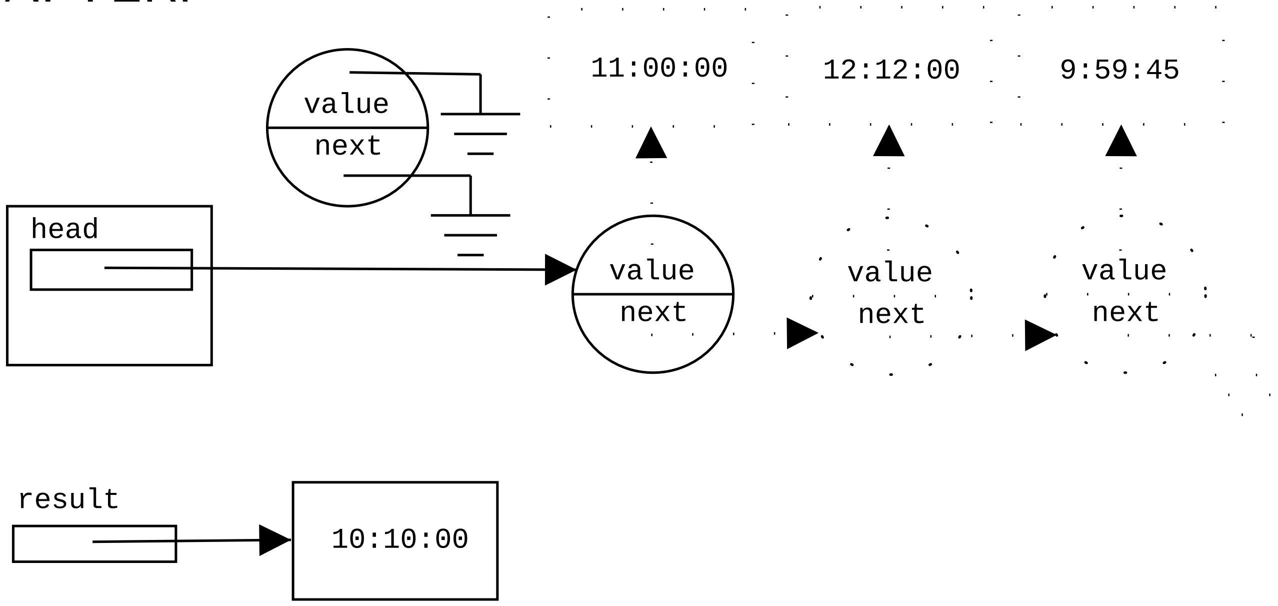
**(a)** Remove from the diagrams everything that either has not been changed or has not been used by the method.

**(b)** Associate a variable with each constant item.

BEFORE:

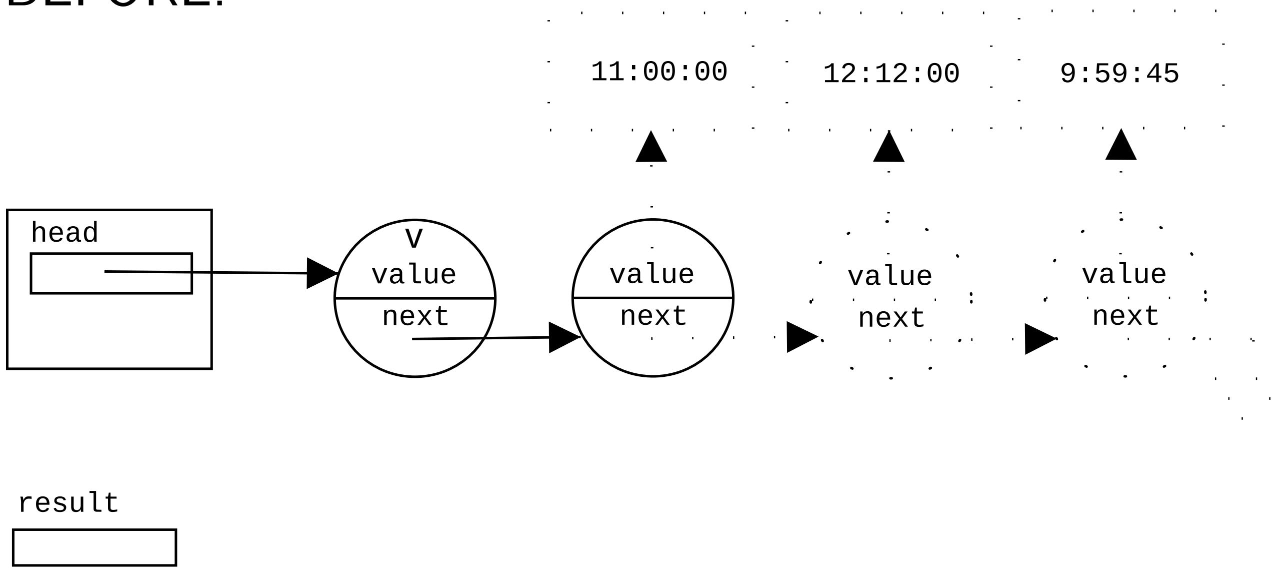


AFTER:

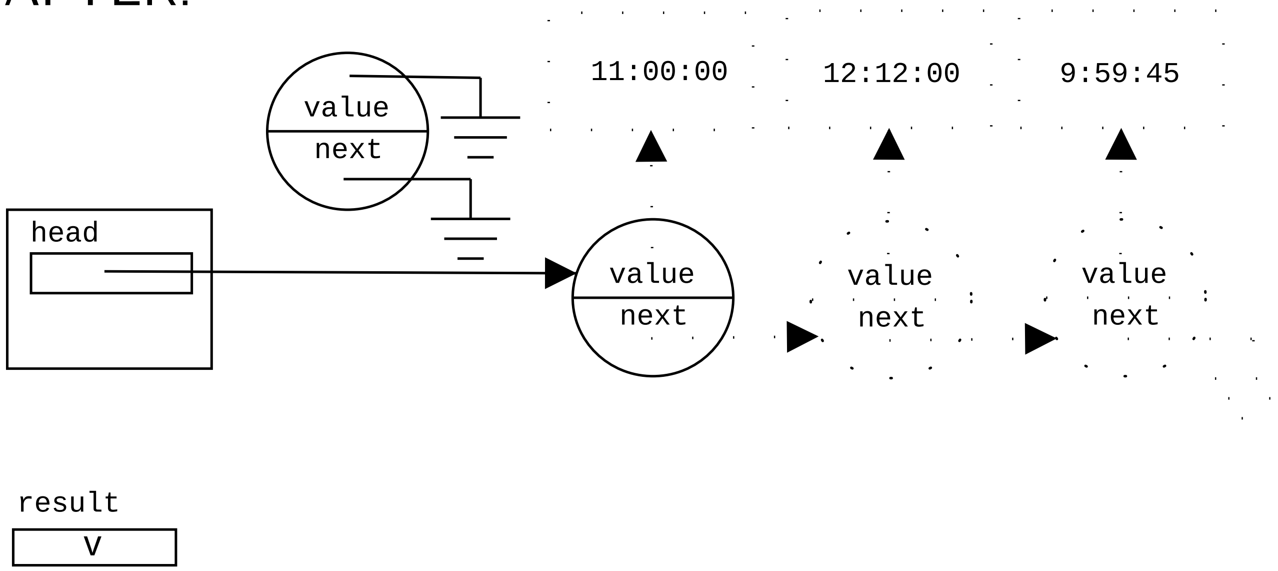


$\Rightarrow$ Find all elements that are not directly involved.

BEFORE:

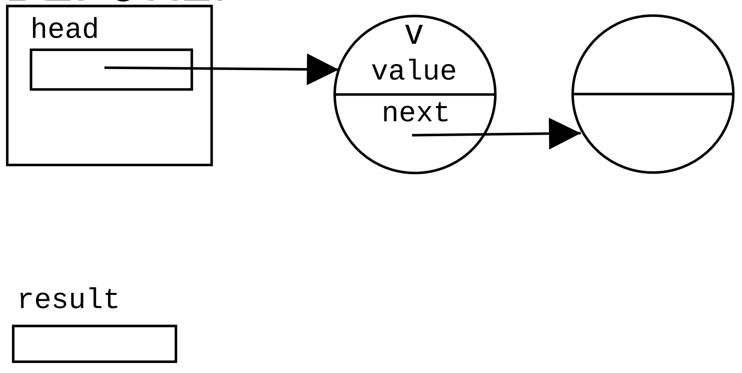


AFTER:

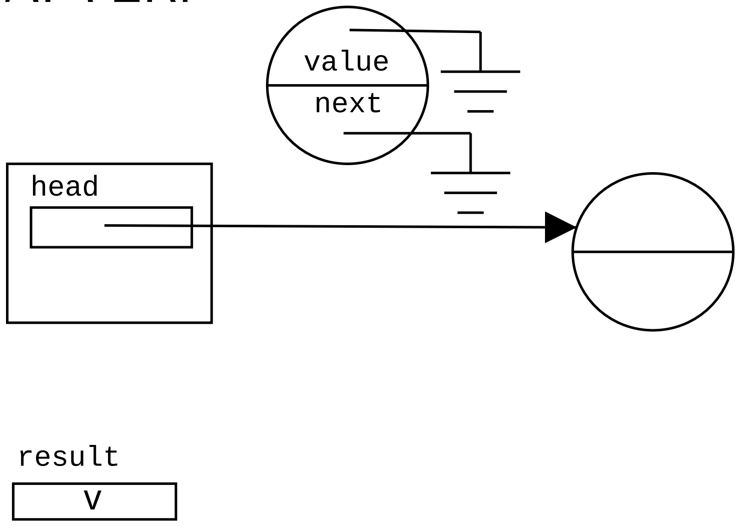


$\Rightarrow$ replacing constant items by variables

BEFORE:



AFTER:



⇒ BEFORE/AFTER diagrams for the general case.

## Step 4: Find all specific cases

There are always one or two exceptions to the general case; in particular with linked structures.

What input does not correspond to the resulting BEFORE/AFTER diagrams for the general case?

The cases for zero and one nodes.

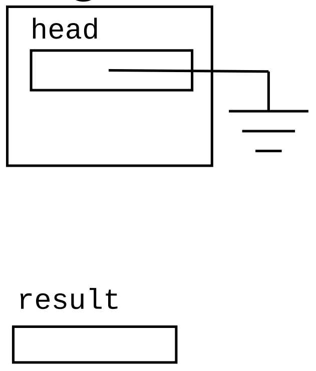
Specific cases can be further divided in two categories: illegal and special cases.

An **illegal** case represents a situation that should not occur, such as removing an element when the list is empty.

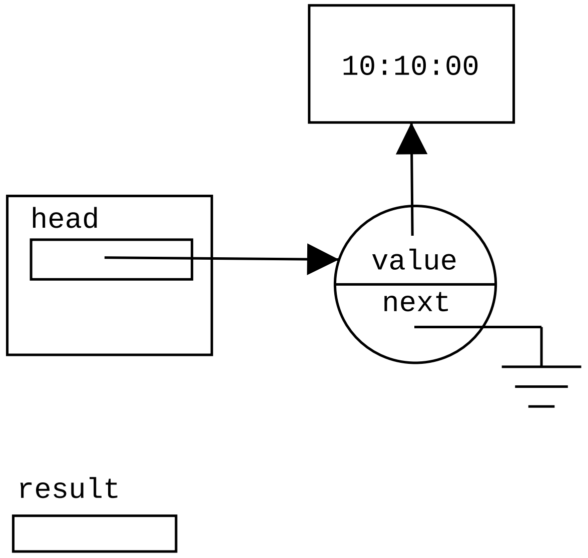
You must document these cases properly, adding a comment to this effect in your program. The proper treatment for illegal cases involves throwing an exception.

A **special** case is a valid situation that is not handled by the general case. Here the singleton list is a valid situation but it handled properly by the general case.

## Illegal case:



## Special case:



## Step 5: Handling special cases

There are two ways to handle special cases:

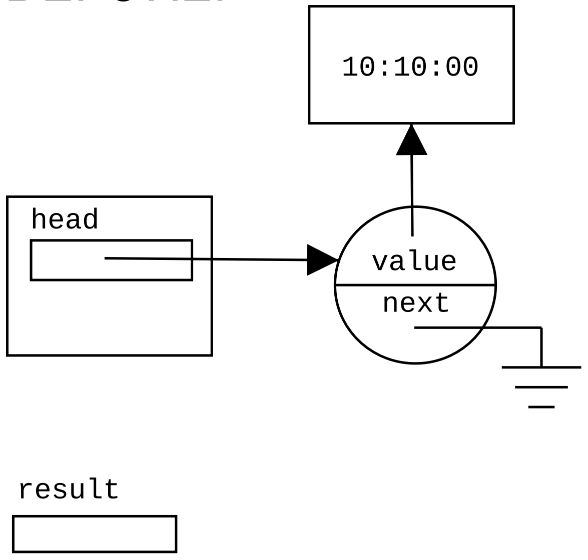
**a)** Modify the diagrams for the general case so that the special cases are also handled.

Whenever possible, this is the best course of action since this will also simplify writing the methods.

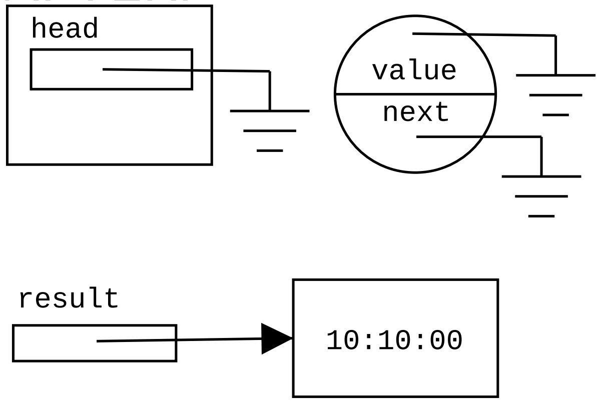
**b)** Repeat steps 2 (BEFORE/AFTER diagrams) and 3 (generalization) for each special case.

Try to create diagrams that can handle the most special cases.

BEFORE:

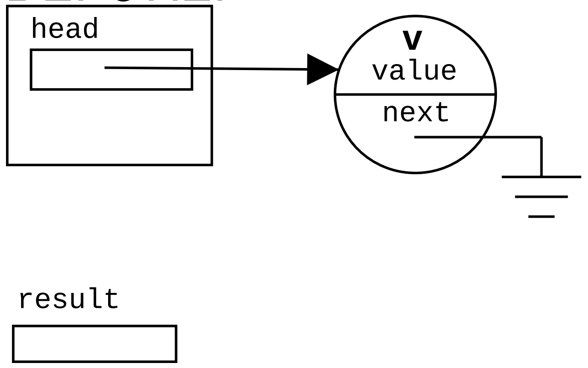


AFTER:

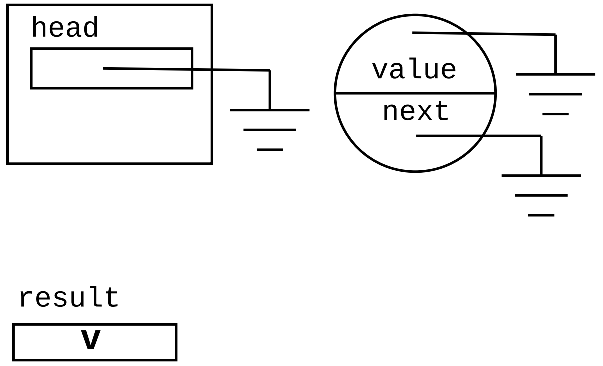


⇒ BEFORE/AFTER diagrams for a list of one element.

BEFORE:



AFTER:



⇒ General BEFORE/AFTER diagrams for the special case where the list contains a single element.

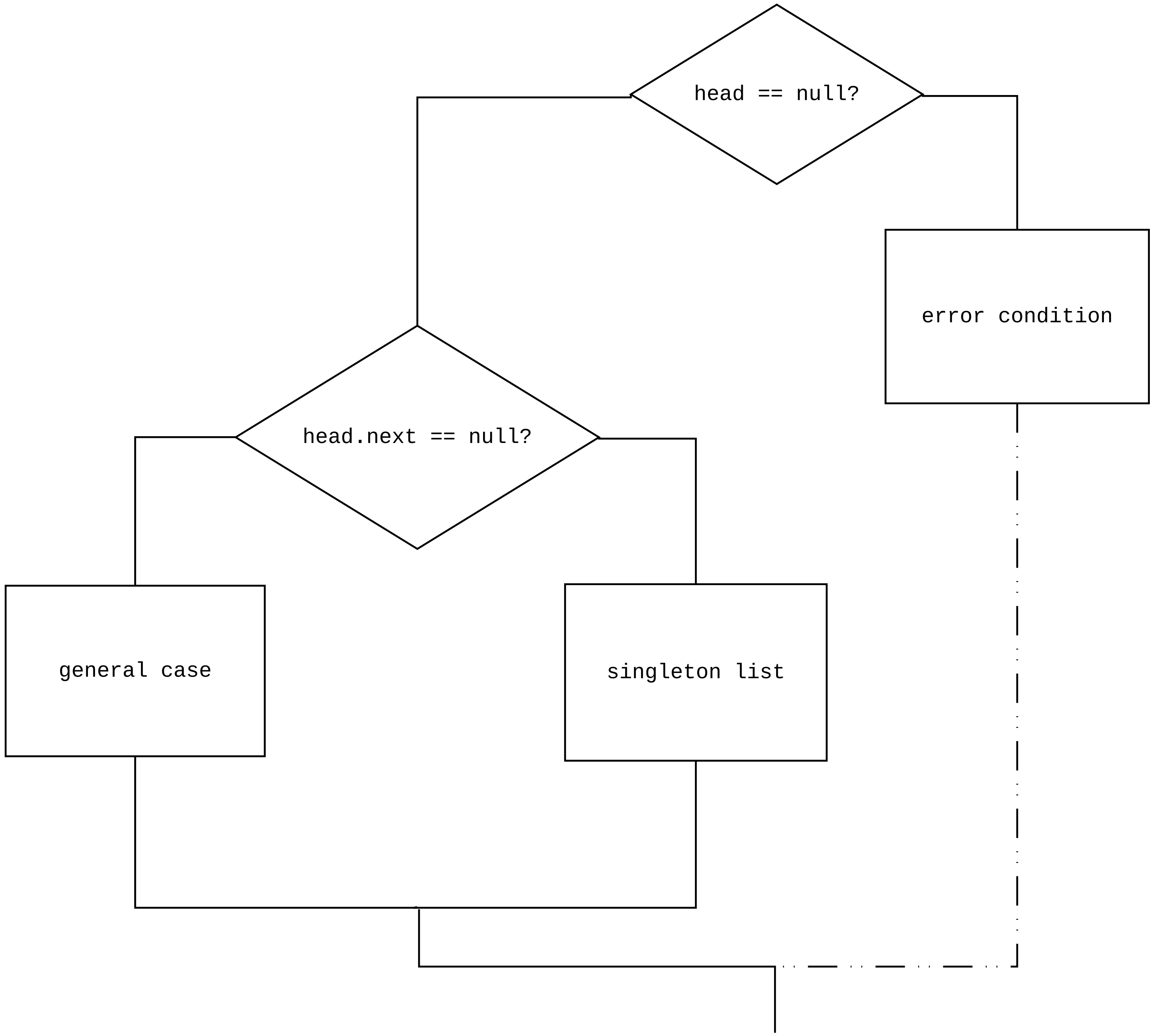
## Step 6: Writing test cases

Looking at the diagrams find expressions allowing to distinguish all cases:

**empty list:** `head == null`?

**singleton list:** `head.next == null`?

**general case:** `else`


head == null?
error condition
head.next == null?
general case
singleton list

## Step 7: Writing blocks of code

For each case, write a block of statements that implements the case — step 8 consists in putting all the blocks together while step 9 consists in code simplification (optimization).

How?

**a)** Declare a variable for the return value:

**b)** **i)** Identify all "objects" involved with a logical name — for example, here all the nodes that are accessed, nodeToDelete, newHead.

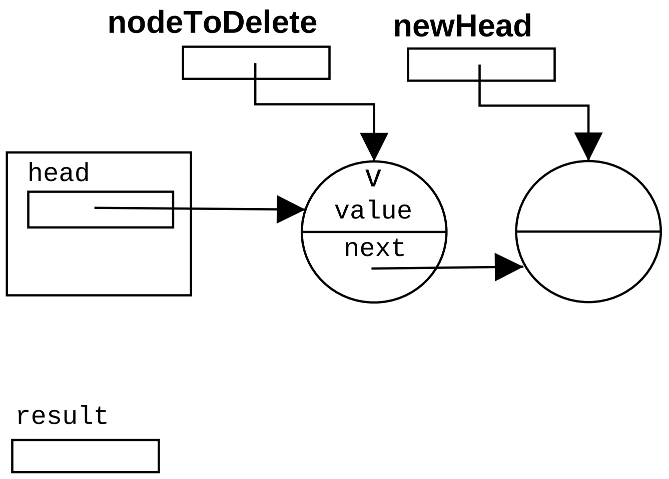
**ii)** For each logical name declare a local variable and initialize the variable appropriately looking at the before diagram.

**c)** Assign the variable for the return value.

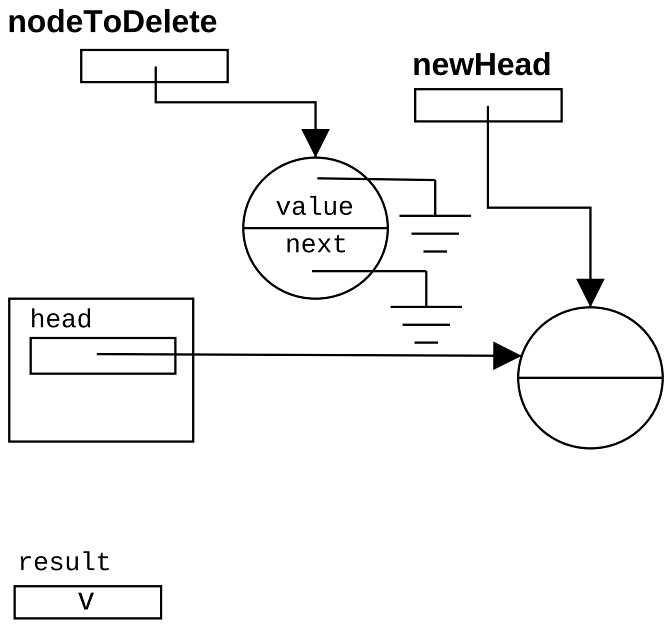
**d)** **i)** Identify all the changes between BEFORE and AFTER

**ii)** For each difference, write the statements that implement the change.
If step (b) has been done systematically then the order to process the changes is not important.

BEFORE:

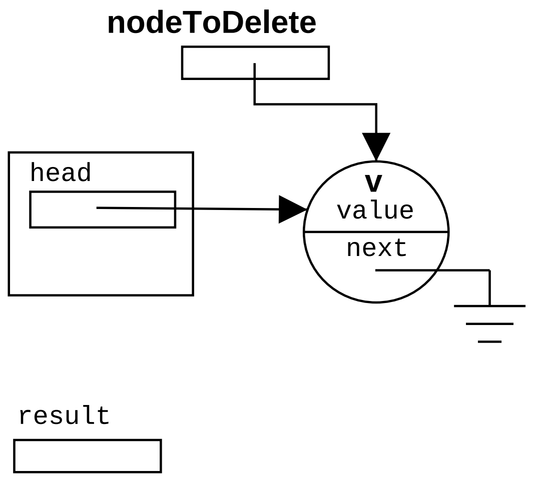


AFTER:

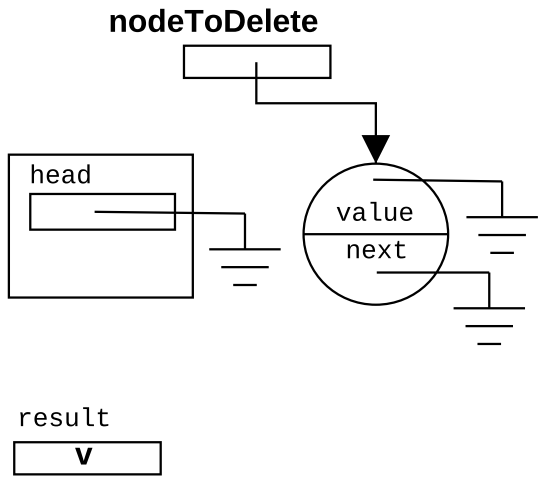


```
// Step a)
E result;

// Step b)
Node<E> nodeToDelete = head;
Node<E> newHead = head.next;

// Step c)
result = nodeToDelete.value;

// Step d)
head = newHead;
nodeToDelete.value = null;
nodeToDelete.next = null;
```

BEFORE:



AFTER:



```
// Step b)
Node<E> nodeToDelete = head;

// Step c)
result = nodeToDelete.value;

// Step d)
head = null;
nodeToDelete.value = null;
nodeToDelete.next = null;
```

## Step 8: Writing the method (unoptimized)

The test structures and blocks of code are put together.

```
public E removeFirst() {
    E result;
    if (head == null) {
        // illegal case
    } else if (head.next == null) {
        Node<E> nodeToDelete = head;      // Step b
        result = nodeToDelete.value;  // Step c
        head = null;                  // Step d
        nodeToDelete.value = null;
        nodeToDelete.next = null;
    } else {
        Node<E> nodeToDelete = head;      // Step b
        Node<E> newHead = head.next;
        result = nodeToDelete.value;  // Step c
        head = newHead;               // Step d
        nodeToDelete.value = null;
        nodeToDelete.next = null;
    }
    return result;
}
```

## Step 9: Optimization

Statements that are common to two branches of an `if` can be put in front of the test.

Variables that used once can be eliminated.

Look for further statements that could be equivalent and therefore factored out.

```
public E removeFirst() {
    E result;
    if (head == null) {
        // illegal case
    } else {
        Node<E> nodeToDelete = head;
        result = nodeToDelete.value;
        if (head.next == null) {
            head = null;
        } else {
            Node<E> newHead = head.next;
            head = newHead;
        }
        nodeToDelete.value = null;
        nodeToDelete.next = null;
    }
    return result;
}
```

$\Rightarrow$ Statements that are common to both branches of the second if statement are put together outside of the test.

```
public E removeFirst() {
    E result;
    if (head == null) {
        // illegal case
    } else {
        Node<E> nodeToDelete = head;
        result = nodeToDelete.value;
        if (head.next == null) {
            head = null;
        } else {
            head = head.next;
        }
        nodeToDelete.value = null;
        nodeToDelete.next = null;
    }
    return result;
}
```

$\Rightarrow$ `newHead` can be eliminated because used only once.

```
public E removeFirst() {
    E result;
    if (head == null) {
        // illegal case
    } else {
        Node<E> nodeToDelete = head;
        result = nodeToDelete.value;
        head = head.next;
        nodeToDelete.next = null;
        nodeToDelete.value = null;
    }
    return result;
}
```

$\Rightarrow$ since `head = null` is equivalent to `head = head.next` the branches are equivalent.

## Summary

What has been achieved?

- a systematic approach that can help avoiding pitfalls;
- the method assists the programmer in defining the necessary local variables;
- identifying all nodes involved in the change of state (using local variables) means that operations modifying the links structure can now be done in any order without the risk of loosing any node.