ITI 1121. Introduction to Computing II [*]

Marcel Turcotte
School of Electrical Engineering and Computer Science

Version of March 17, 2013

Abstract

- Linked List (Part 2)
  - Tail pointer
  - Doubly linked list
  - Dummy node

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

Time efficiency

Compare the time efficiency of the dynamic array (**ArrayList**) and linked list (**LinkedList**) implementations of the interface **List** (both allow to store an unlimited number of objects).

Let say that execution time of a method is **variable** (slow) if the number of operations depends on the number of elements currently stored in the data structure, and **constant** (fast) otherwise.

Time efficiency

Can you predict an overall winner beforehand?

	ArrayList	**LinkedList**
void addFirst(E o)	slow	fast
void addLast(E o)	slow	slow
void add(E o, int pos)	slow	slow
E get(int pos)	fast	slow
void removeFirst()	slow	fast
void removeLast()	fast	slow

- Based on the above table, when would you use an array? Applications that need a direct (random) access to the elements.
- Based on the above table, when would you use a singly-linked list? Applications that add or remove elements at the start of the list only.
- Which implementation is more memory efficient?

Speeding up addLast for SinglyLinkedList

There is a simple implementation technique that makes adding an element at the end of a list fast.

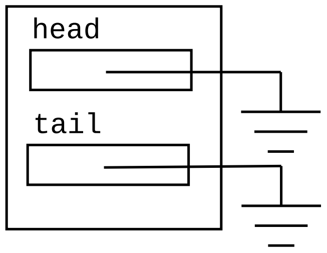
The problem with the singly linked list implementation is that one needs to traverse the data structure to access the last element.

What if we could always access the last element efficiently — as we do for the first element.

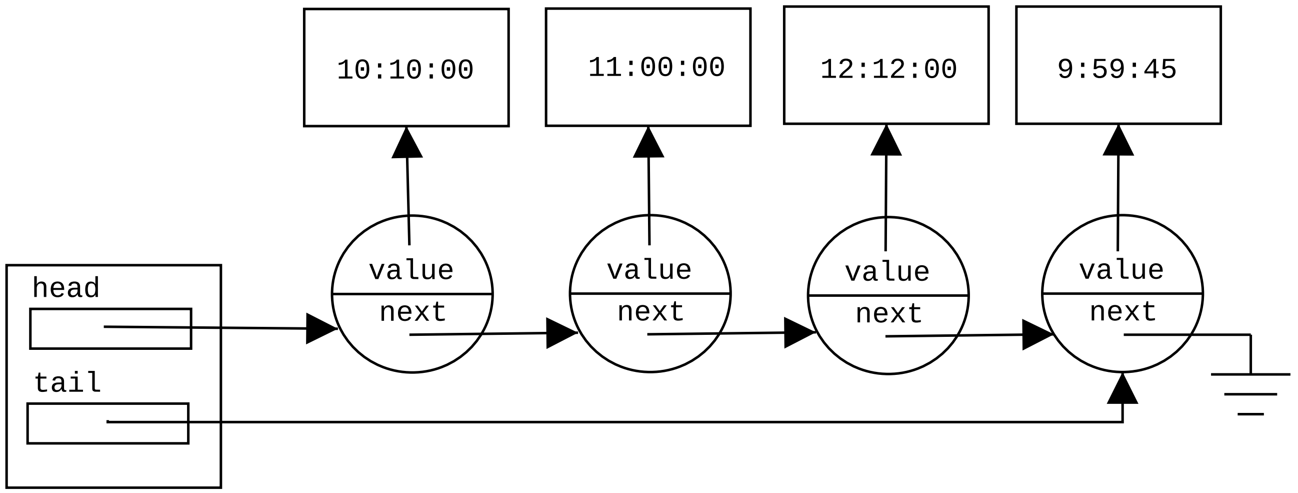
Got the idea?

Yes, adding an instance variable pointing to the **tail** element will solve our problem.

Representing an empty list:



General case:



```
public class SinglyLinkedList<E> implements List<E> {

    private static class Node<T> {

        private T value;
        private Node<T> next;

        private Node( T value, Node<T> next ) {
            this.value = value;
            this.next = next;
        }
    }

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

    // ...
}
```

⇒ This involves adding a new instance variable, **tail**.

```
public void addLast( E t ) {

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

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

```
public E removeFirst() {

    Node<E> nodeToDelete = head;
    E result = nodeToDelete.value;

    head = head.next;

    nodeToDelete.value = null; // ``scrubbing''
    nodeToDelete.next = null;

    if ( head == null ) {
        tail = null;
    }

    return result;
}
```

⇒ The methods need to be modified accordingly!

Time efficiency (revision 1)

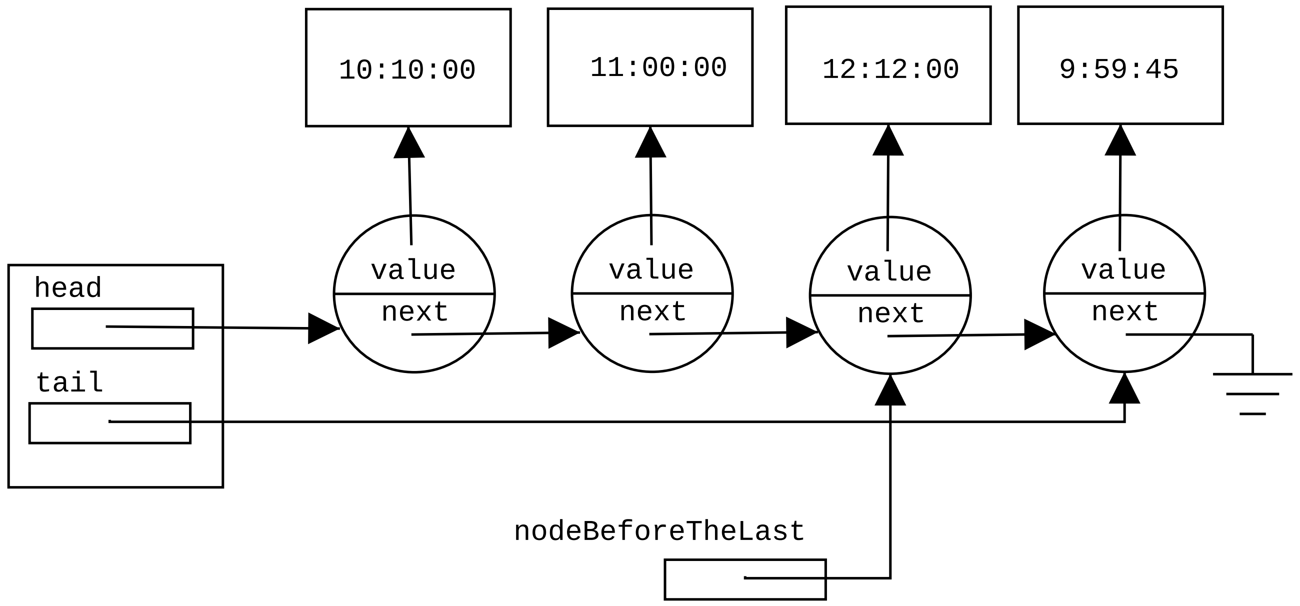
	ArrayList	**LinkedList**
void addFirst(E o)	slow	fast
void addLast(E o)	slow	**fast**
void add(E o, int pos)	slow	slow
E get(int pos)	fast	slow
void removeFirst()	slow	fast
void removeLast()	fast	slow

How about removing the last element of the list?

It's still slow.

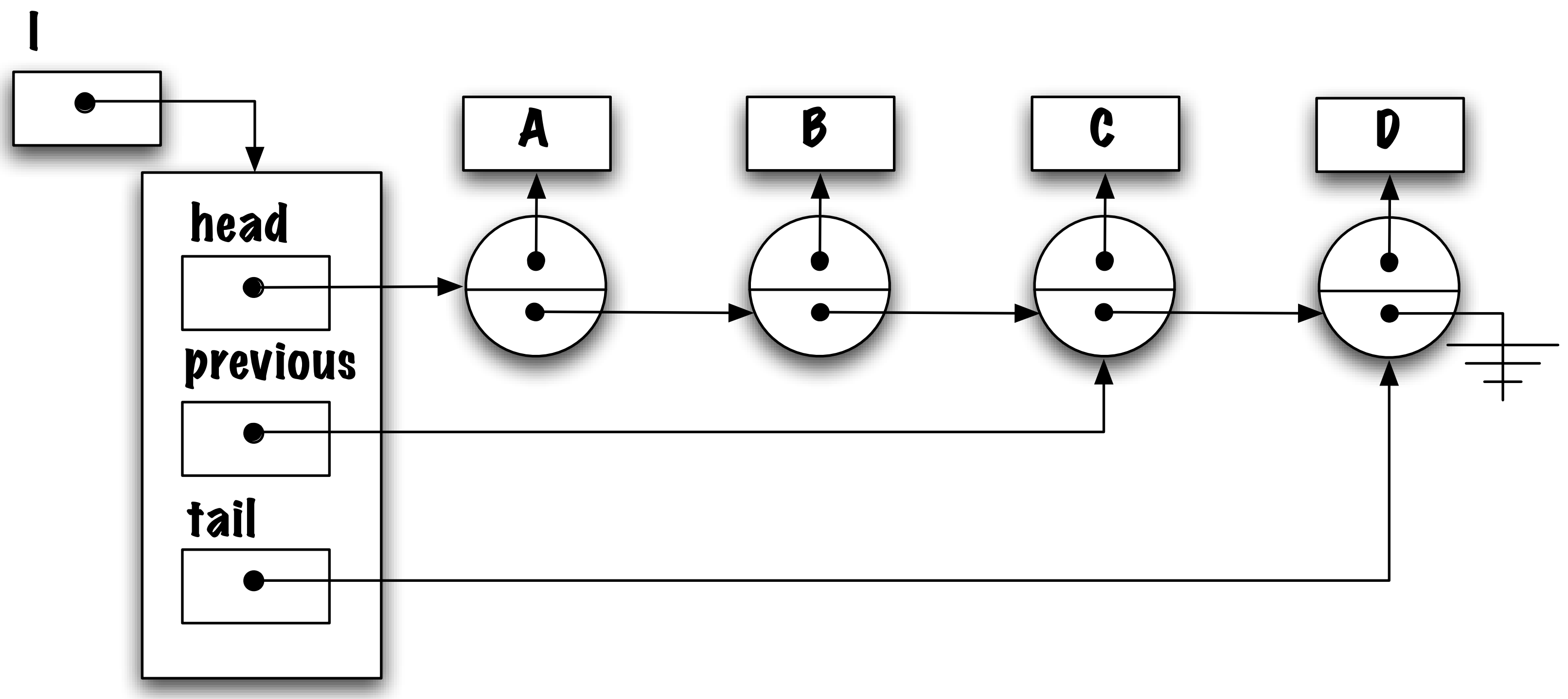
Speeding up removeLast()

Maintaining a reference to the last element of the list does not make the removal of the last element any faster, we still have to traverse the list:



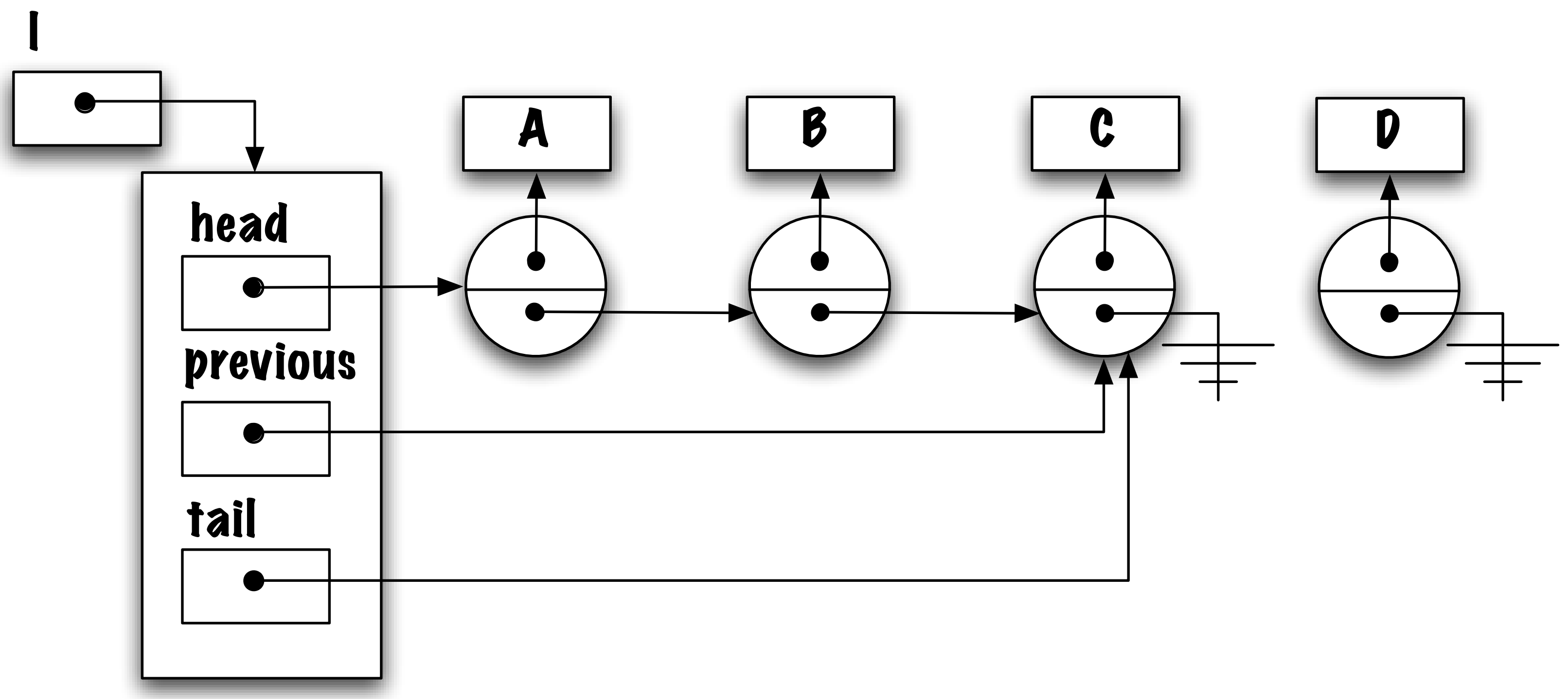
⇒ What's needed then? How about a new **instance** variable `previous`?

Speeding up removeLast()



What do you think?

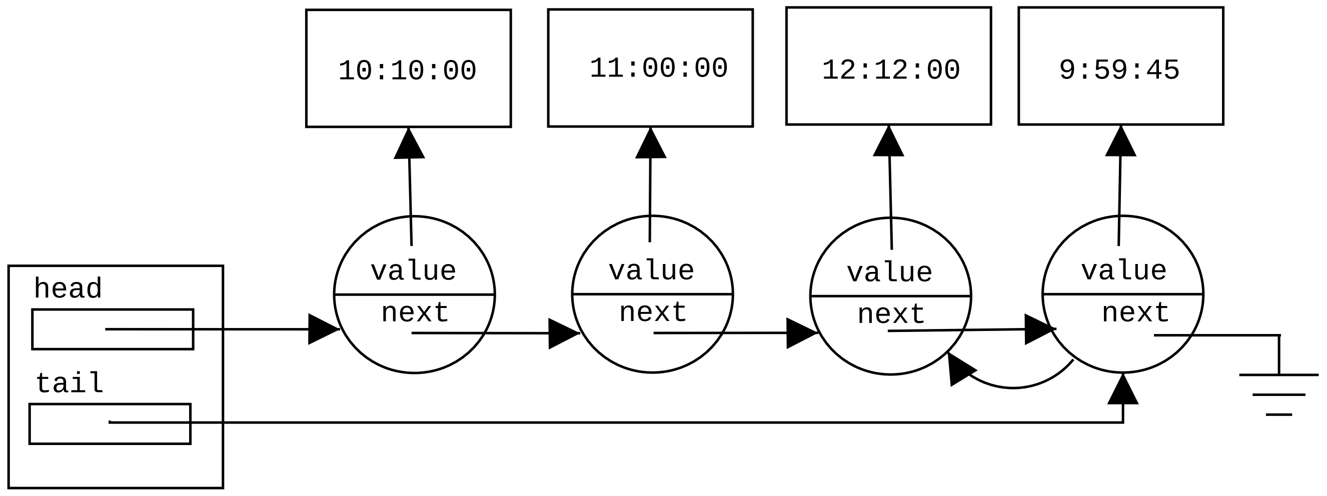
Speeding up removeLast()



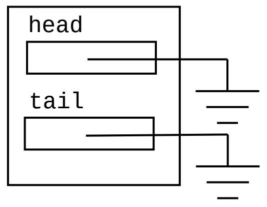
Moving the reference **tail** one position to left is now easy and fast!

But moving the reference **previous** one position to the left is now tedious and costly.

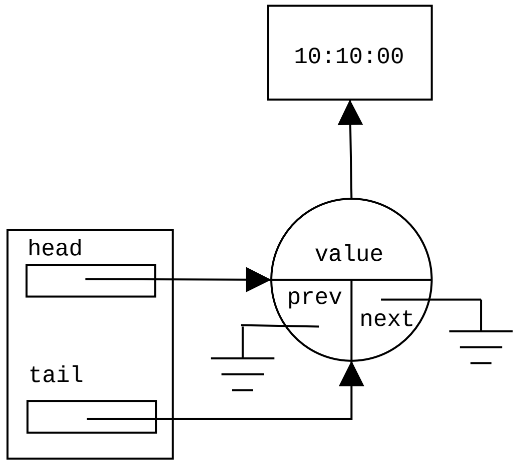
We'd need to access the previous element, the one before the last:



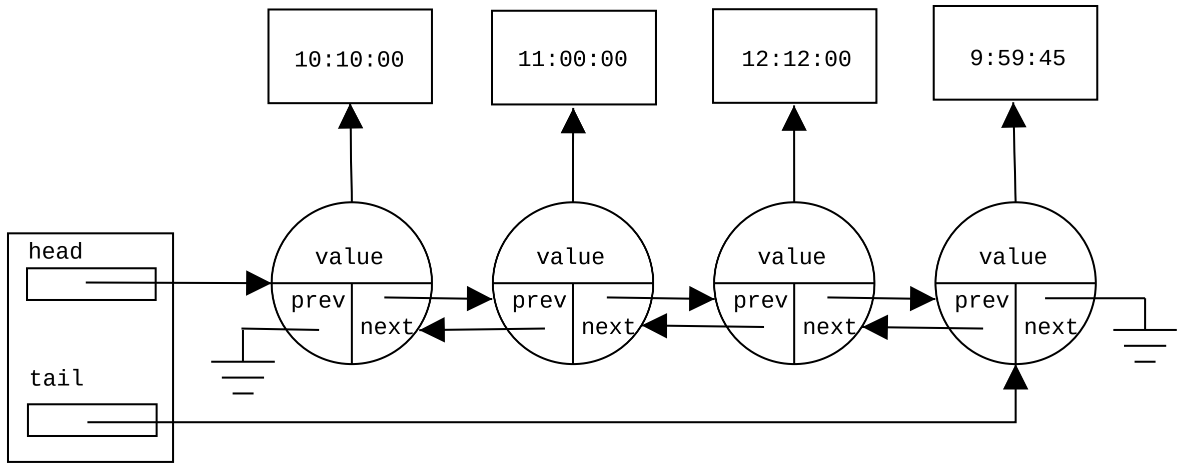
But also to all its predecessors!



Empty list:



Singleton:



General case:

```
public class DoublyLinkedList<E> implements List<E> {
    private static class Node<T> {
        private T value;
        private Node<T> previous; // <---
        private Node<T> next;
        private Node( T value, Node<T> previous, Node<T> next ) {
            this.value = value;
            this.previous = previous; // <---
            this.next = next;
        }
    }
    private Node<E> head;
    private Node<E> tail;
    public DoublyLinkedList() {
        head = null;
        tail = null;
    }
    // ...
}
```

removeLast() (special case: singleton)

removeLast() (general case)

```
public E removeLast() {
    // pre-condition: ?

    Node<E> toDelete = tail;
    E savedValue = toDelete.value;

    if ( head.next == null ) {
        head = null;
        tail = null;
    } else {
        tail = tail.previous;
        tail.next = null;
    }
    toDelete.value = null;
    toDelete.next = null;

    return savedValue;
}
```

$\Rightarrow$ `removeLast()` does not involve traversing the list anymore.

Time efficiency (revision 2)

	ArrayList	LinkedList
void addFirst(E o)	slow	fast
void addLast(E o)	slow	fast
void add(E o, int pos)	slow	slow
E get(int pos)	fast	slow
void removeFirst()	slow	fast
void removeLast()	fast	**fast**

Simple? Not so simple?

Whenever an operation changes the head pointer, a special case has to be made.

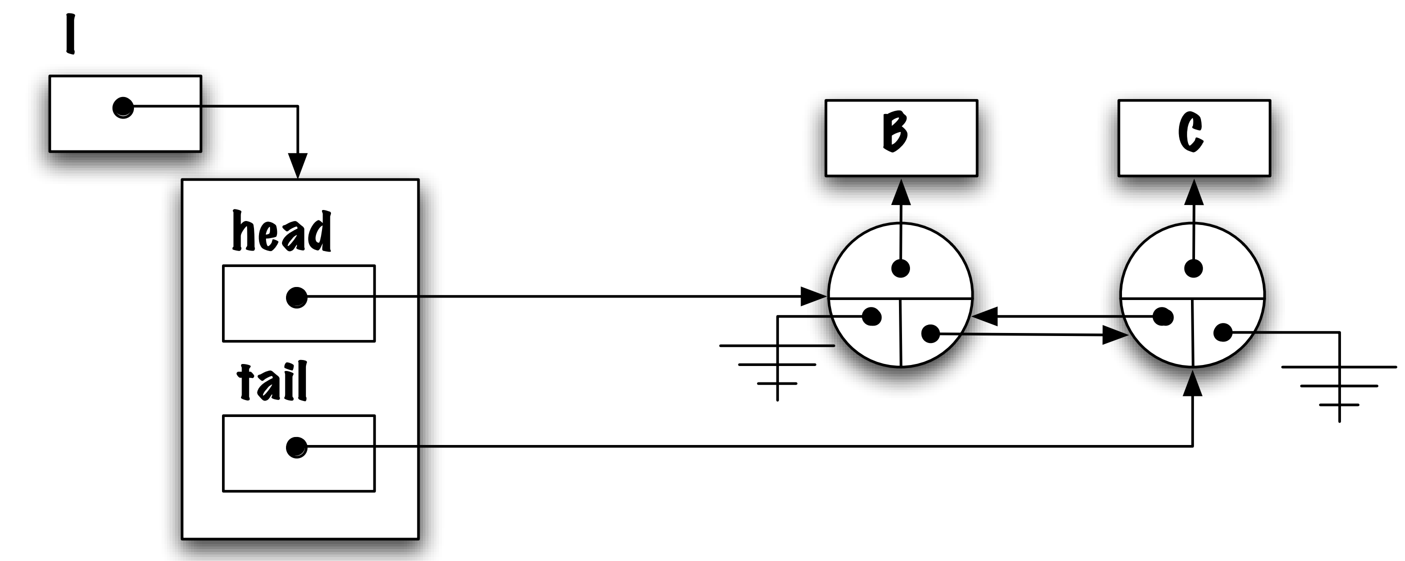
add(int pos, E o)

Pre-conditions?

```
if ( o == null ) {
    throw new IllegalArgumentException( "null" );
}
if ( pos < 0 ) {
    throw new IndexOutOfBoundsException( Integer.toString( pos ) );
}
```

add(int pos, E o)

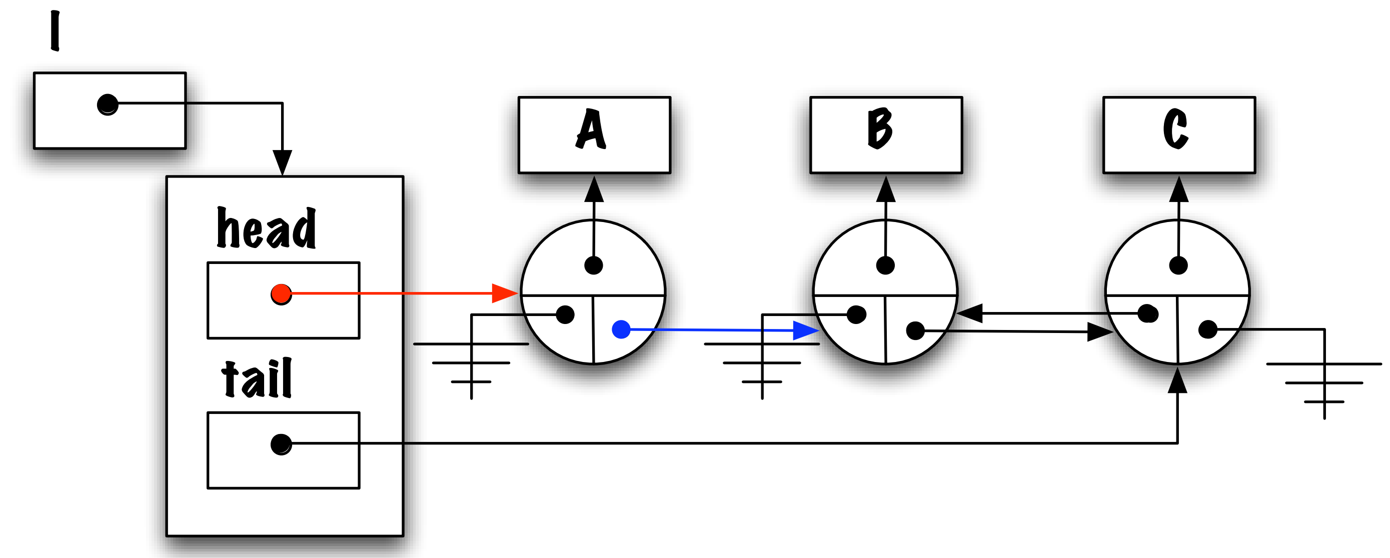
Special case(s)?



Adding an element a position 0.

add(int pos, E o)

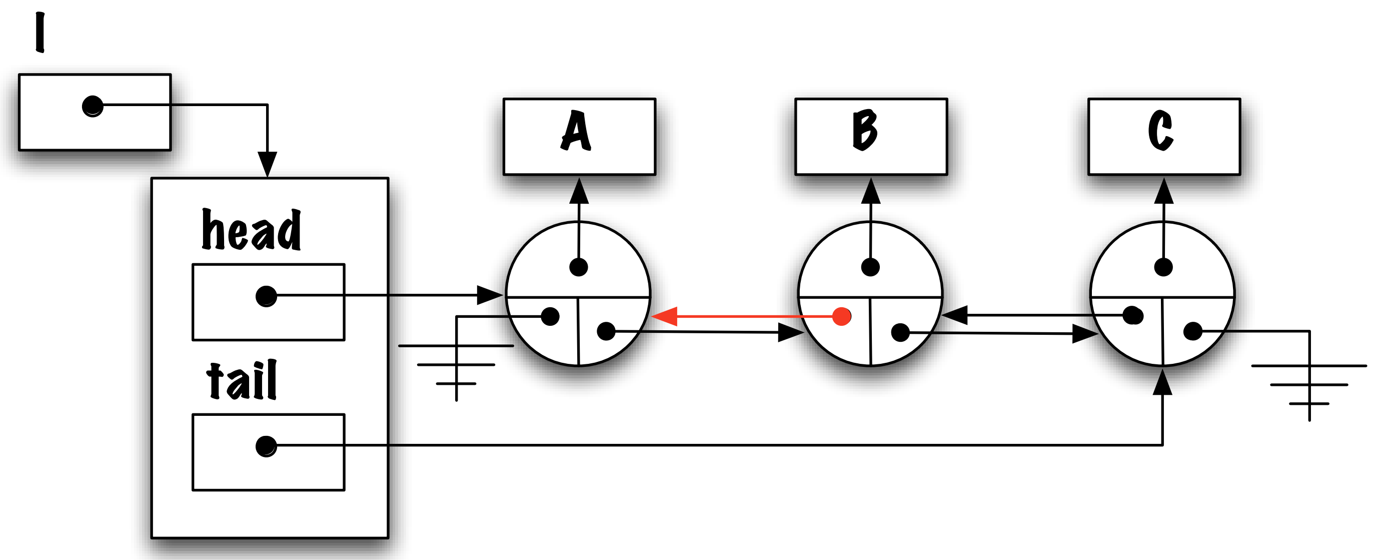
Special case: `head = new Node<E>( o, null, head )`



What is missing?

add(int pos, E o)

Special case: `head.next.previous = head`



add(int pos, E o)

Special case:

```
if ( pos == 0 ) {

    head = new Node<E>( o, null, head );
    head.next.previous = head;

}
```

Does cover all the cases?

What if the list was empty.

add(int pos, E o)

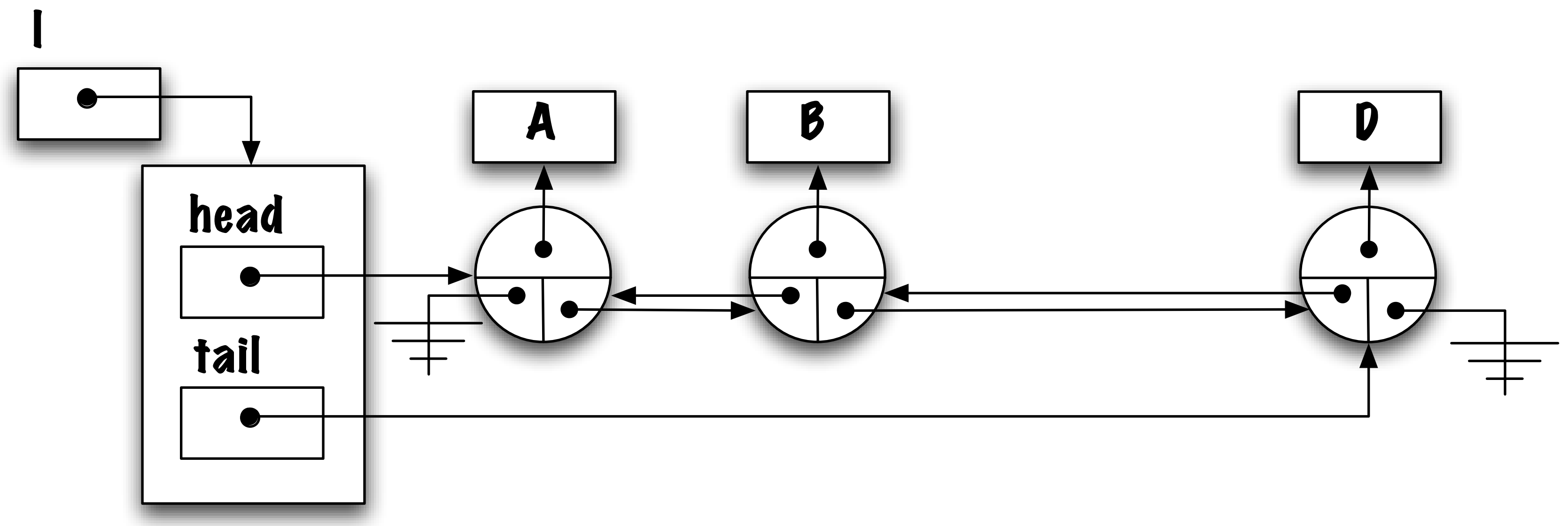
Special case:

```
if (pos == 0) {

    head = new Node<E>( o, null, head );
    if ( tail == null ) {
        tail = head;
    } else {
        head.next.previous = head;
    }
}
```

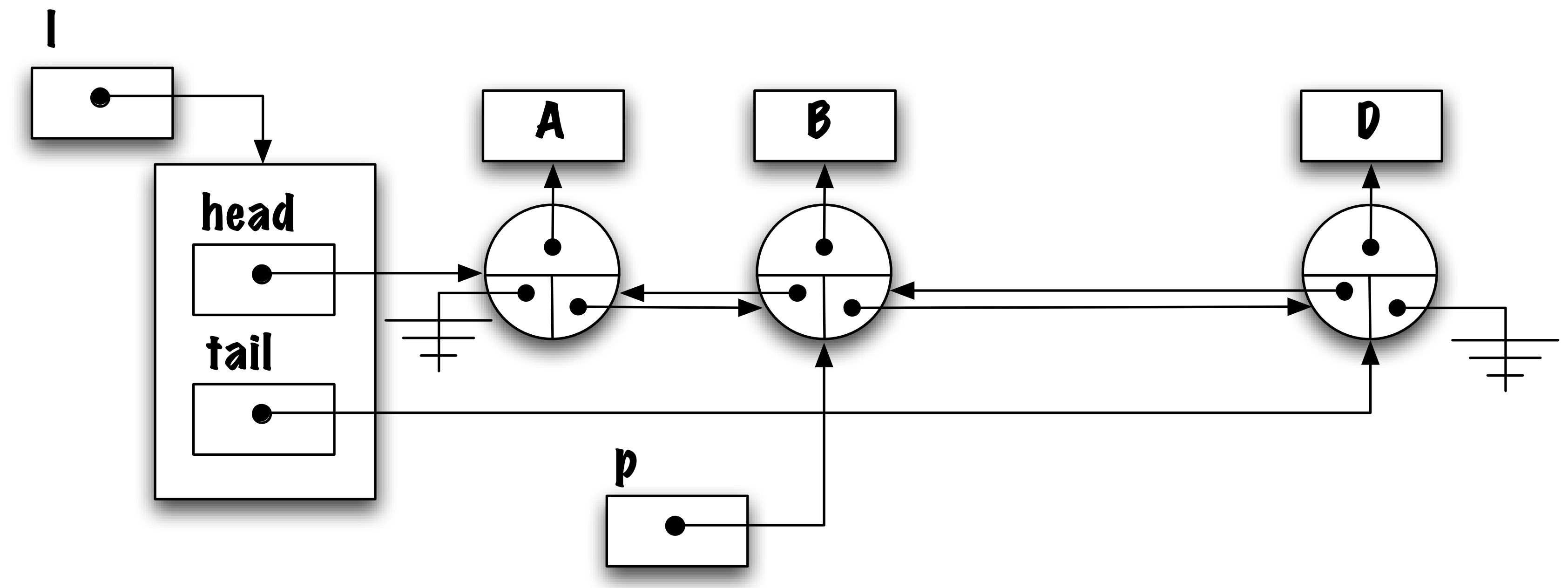
add(int pos, E o)

General case: adding an element at position 2.



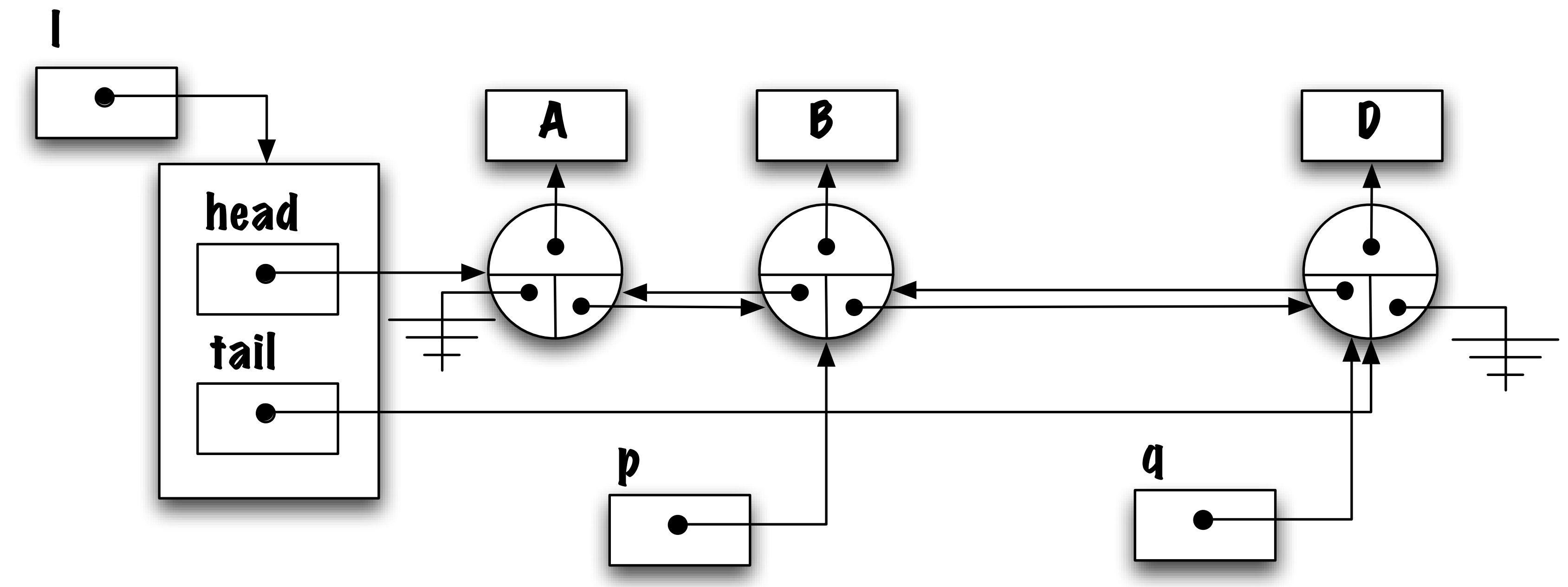
add(int pos, E o)

General case: traverse the list up to **pos-1**.



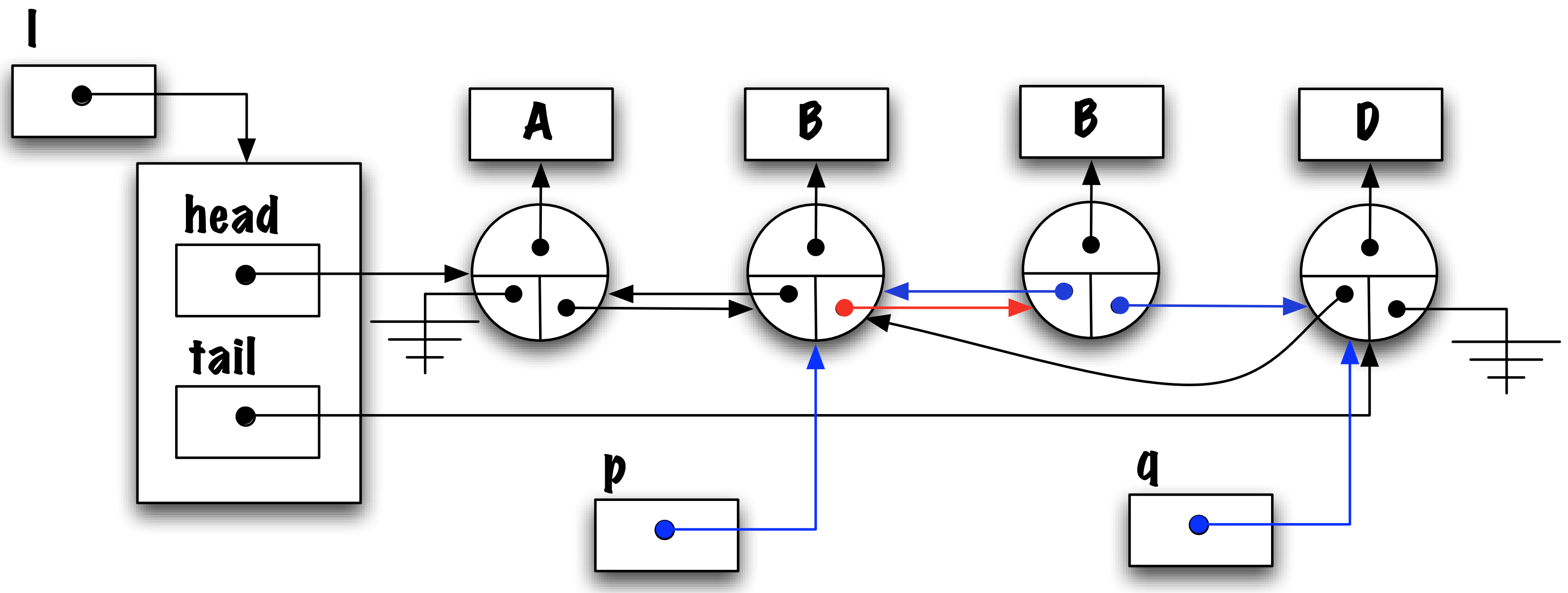
add(int pos, E o)

General case: **q = p.next**



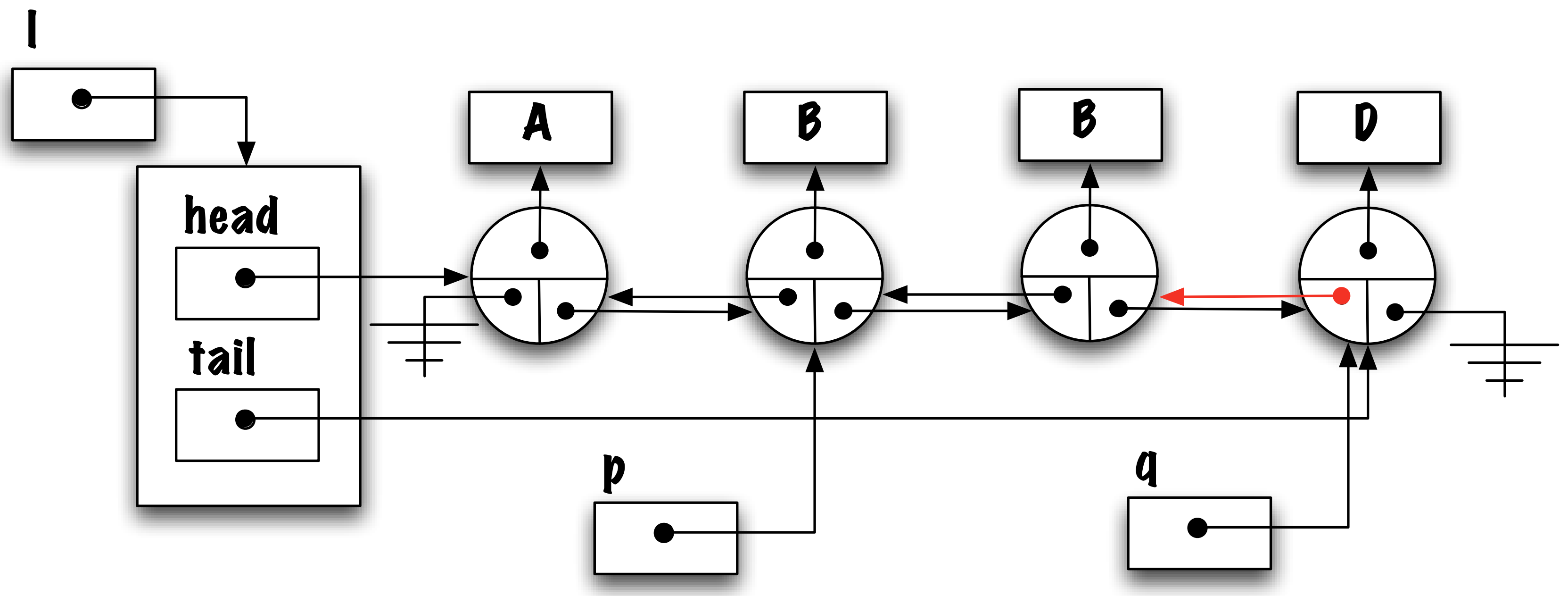
add(int pos, E o)

General case: **p.next = new Node<E>(o, p, q)**



add(int pos, E o)

General case: **q.previous = p.next**



add(int pos, E o)

General case:

```
Node<E> p = head;

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

p.next = new Node<E>( o, p, q );
q.previous = p.next;
```

Handles all the cases?

What if **pos** was too large?

add(int pos, E o)

General case:

```
Node<E> p = head;
for (int i = 0; i < (pos-1); i++) {
  if ( p == null ) {
      throw new IndexOutOfBoundsException( Integer.toString( pos ) );
  } else {
      p = p.next;
  }
}
Node<E> q = p.next;
p.next = new Node<E>( o, p, q );
q.previous = p.next;
```

Handles all the cases?

What about adding at the end of the list?

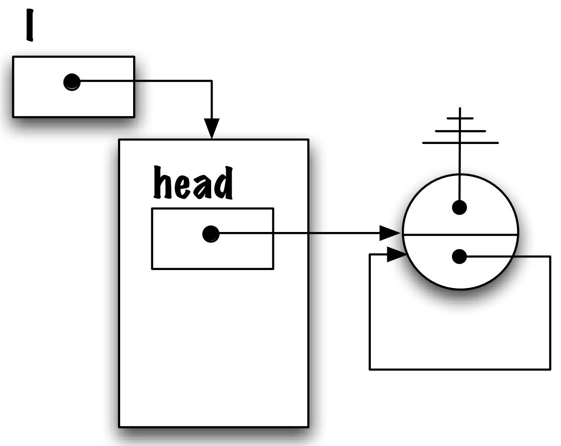
add(int pos, E o)

```
Node<E> p = head;
for (int i = 0; i < (pos-1); i++) {
  if ( p == null ) {
      throw new IndexOutOfBoundsException( Integer.toString( pos ) );
  } else {
      p = p.next;
  }
}
Node<E> q = p.next;
p.next = new Node<E>( o, p, q );
if ( p == tail ) {
  tail = p.next;
} else {
  q.previous = p.next;
}
```

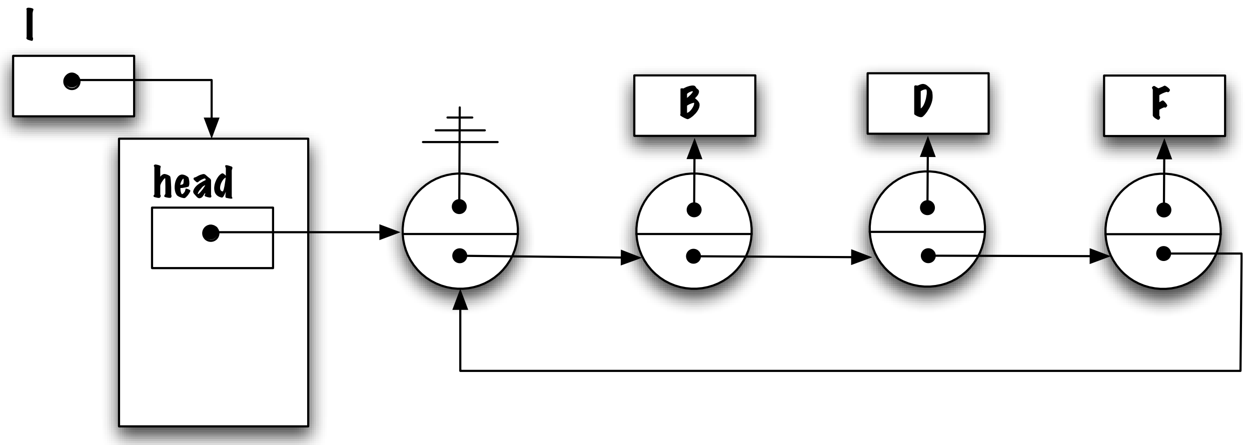
Dummy node

The following implementation techniques simplifies those cases. It consists in 1) using a dummy node (a node that contains no data) as the first element of the list and 2) creating a circular list.

The empty list consists of the dummy node pointing to itself.



General case:



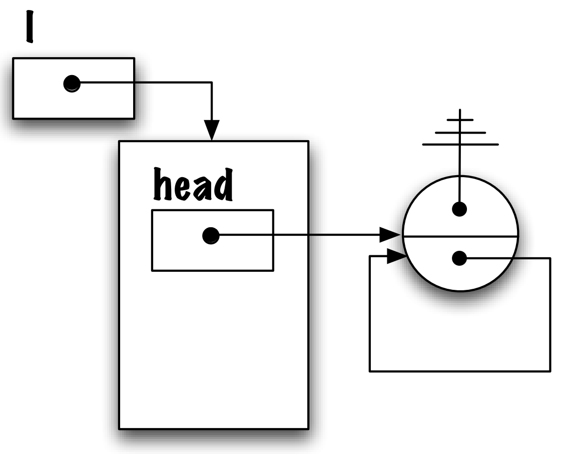
```
public class SinglyLinkedList<E> implements List<E> {
    private static class Node<T> {
        private T value;
        private Node<T> next;
        private Node( T value, Node<T> next ) {
            this.value = value;
            this.next = next;
        }
    }
    private Node<E> head;
    public SinglyLinkedList() {
        head = new Node<E>( null, null );
        head.next = head;
    }
    // ...
}
```

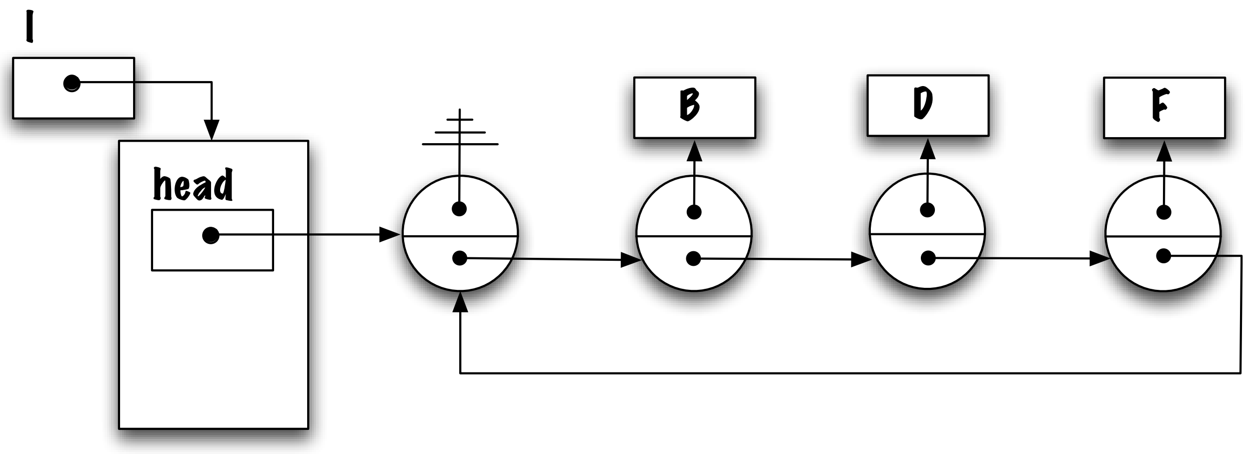
```
// Classic singly linked-list implementation

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

Dummy node (addLast)

The new element will be added after a node such that . . .





```
// Dummy node implementation

public void add( E t ) {
    Node<E> p = head;
    while ( p.next != head ) {
        p = p.next;
    }
    p.next = new Node<E>( t, head );
}
```

Remarks (dummy node)

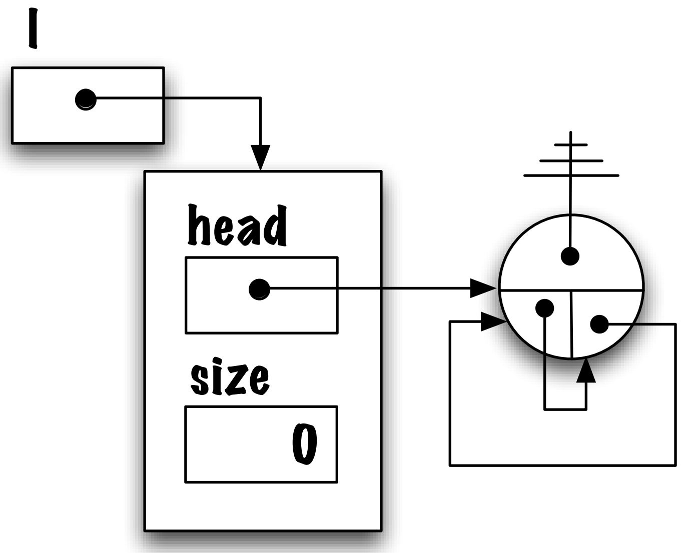
What makes the implementation of the methods more complex in the case of a linked list without dummy node?

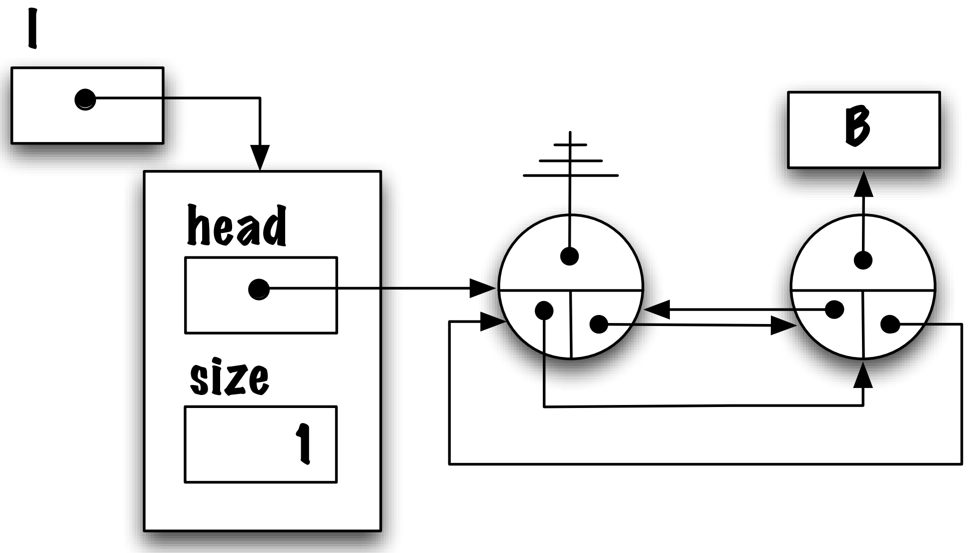
Modifications of the head of the list are special cases (remove, addFirst, addLast, . . .).

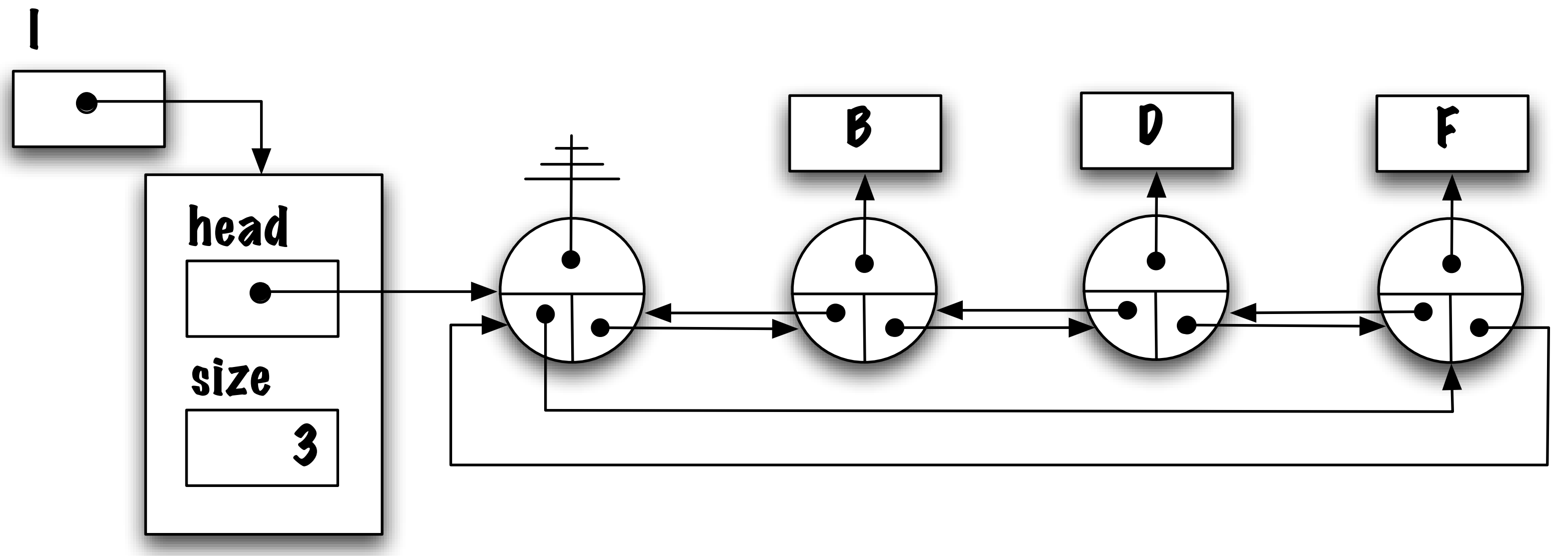
In the general case, the variable **next** of the previous node is changed, except if the modification occurs at the first position, then the **head** variable must be changed.

With the dummy node, it is always the variable **next** of the previous node that is changed.

The nodes could also be doubly linked, and there could be a counter in the header of the list.


l
head
size
0


l
head
size
1
B


l
head
size
3
B
D
F

Collection Framework

In Java the classes that are used to store objects are regrouped into a hierarchy of classes called Collection.

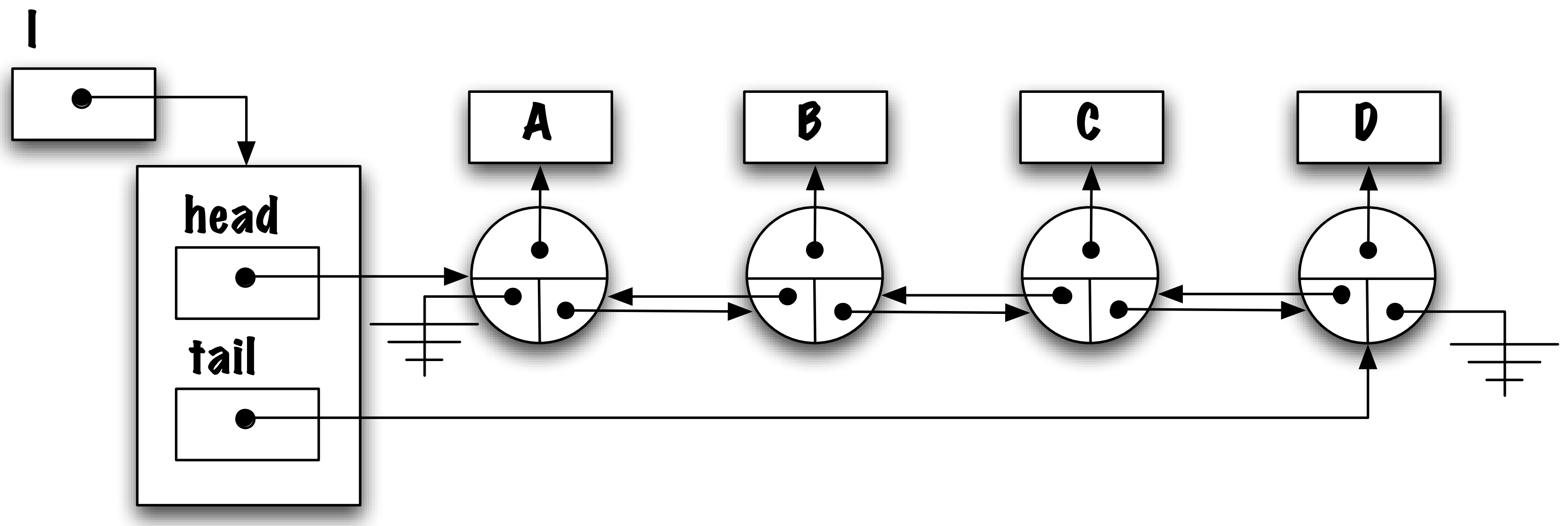
There are four broad categories of collections: linear, hierarchical, graph and unordered.

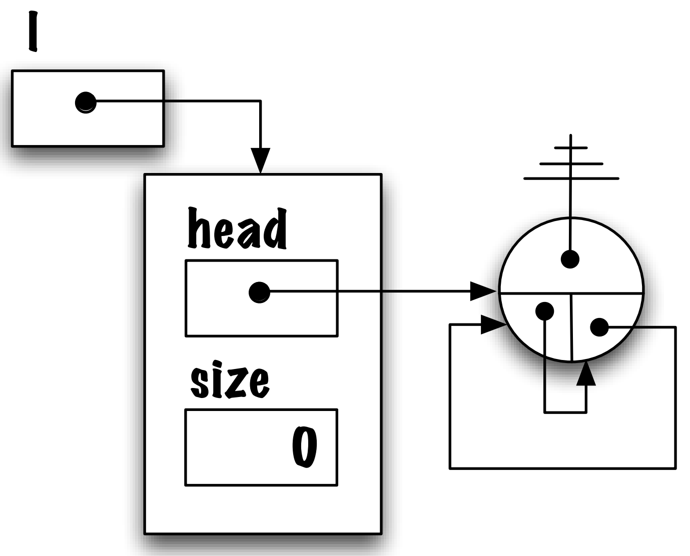
Linear collections comprise the lists, the stacks and the queues. Elements of a linear collection all have a specific predecessor and successor(except for the first and last element).

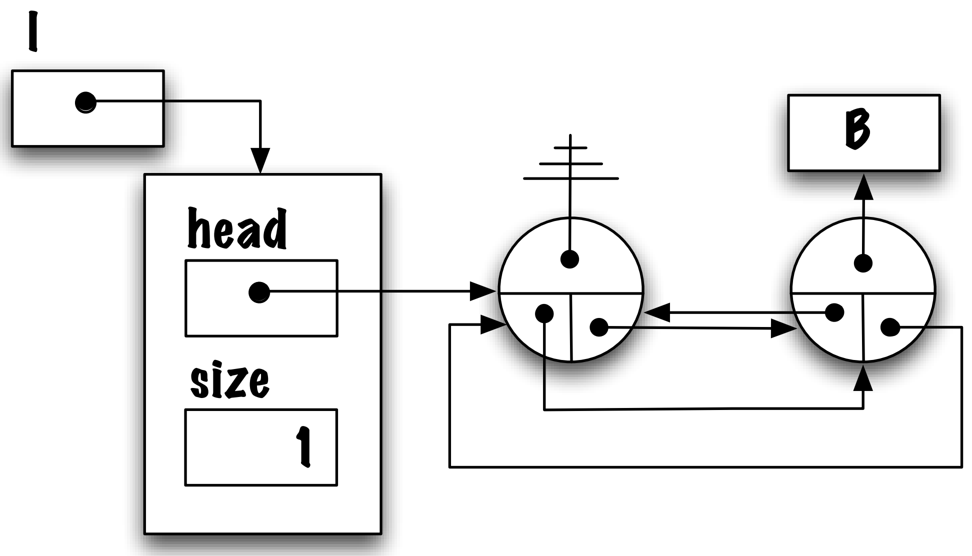
Hierarchical collections allow to represent various kinds of trees: e.g.: genealogical information.

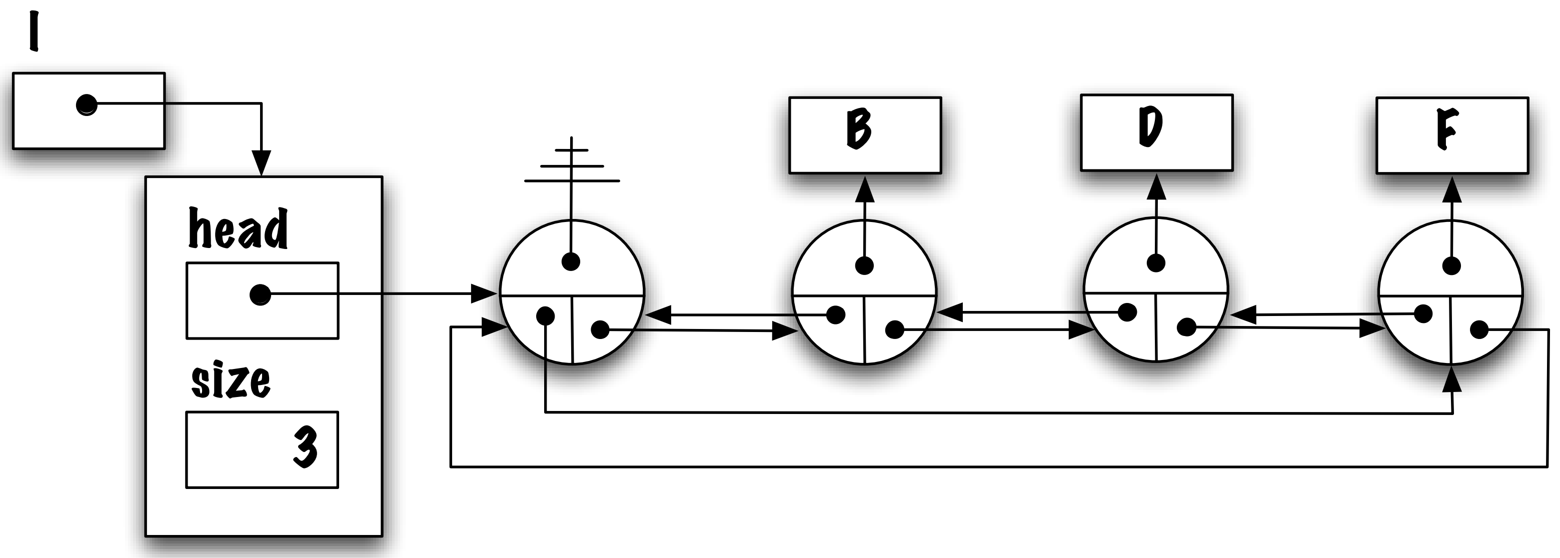
The graph collections are used to store directed, undirected, weighted and unweighted graphs: e.g.: a graph that represents all the cities in Canada and their distances.

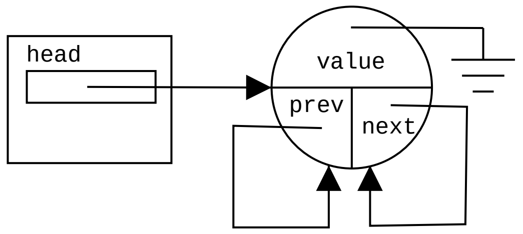
Unordered collections include sets, bags and maps.

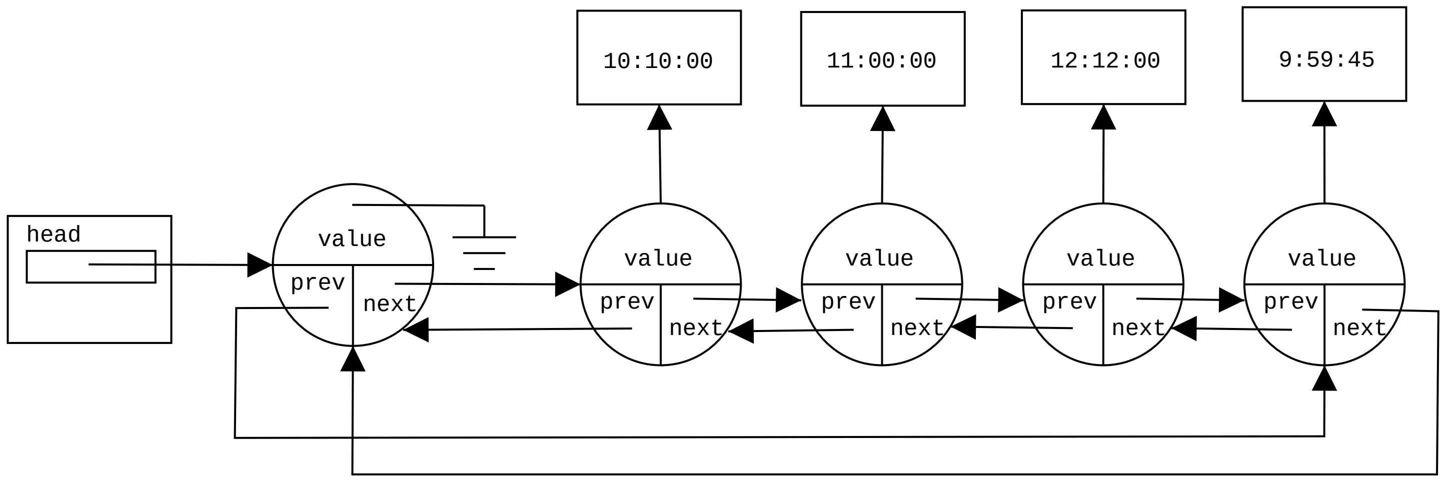

l
head
tail
A
B
C
D

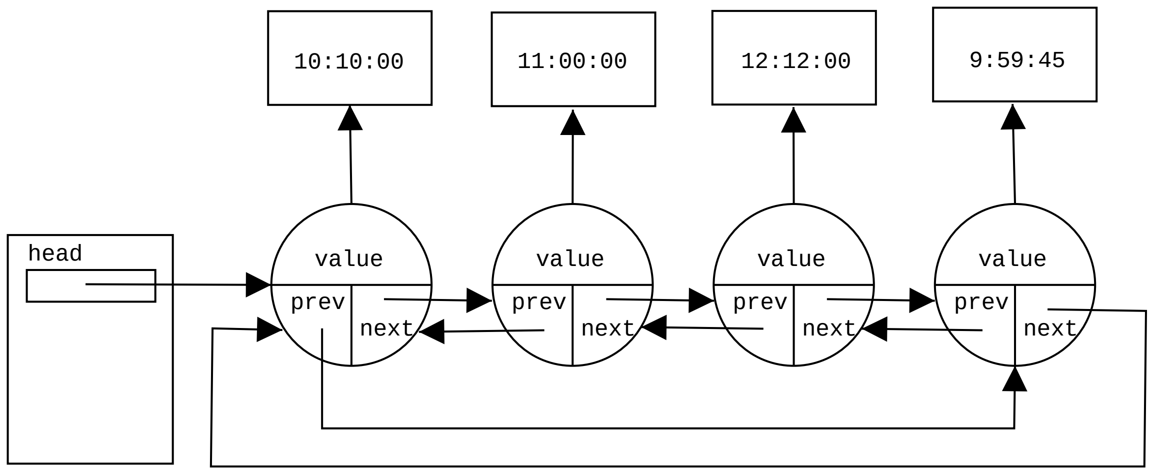

l
head
size
0


l
head
size
1
B


l
head
size
3
B
D
F


head
value
prev
next


head
value
prev
next
10:10:00
11:00:00
12:12:00
9:59:45


head
value
prev
next
10:10:00
11:00:00
12:12:00
9:59:45