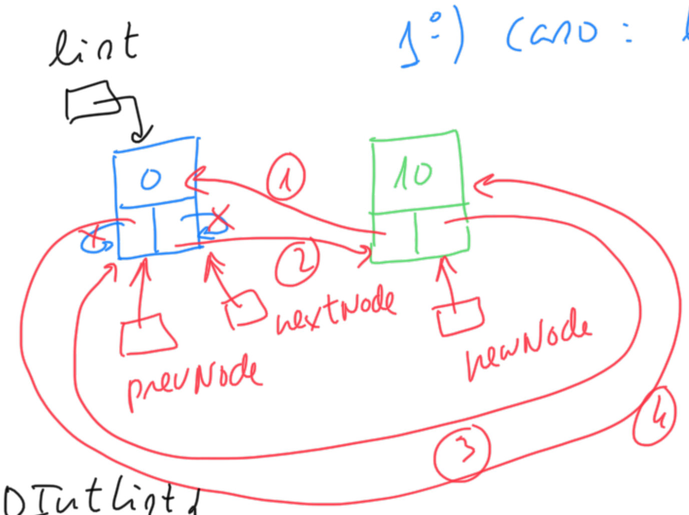
DList_addFront

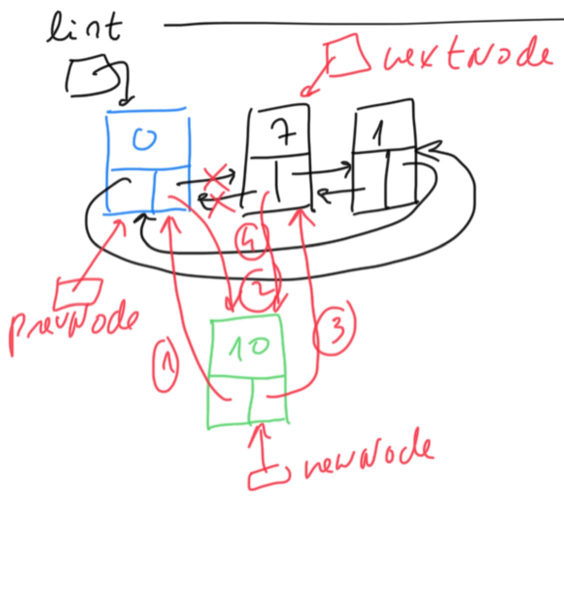
1º) caso: lista vazia



Novo nó a inserir com valor $\boxed{10}$

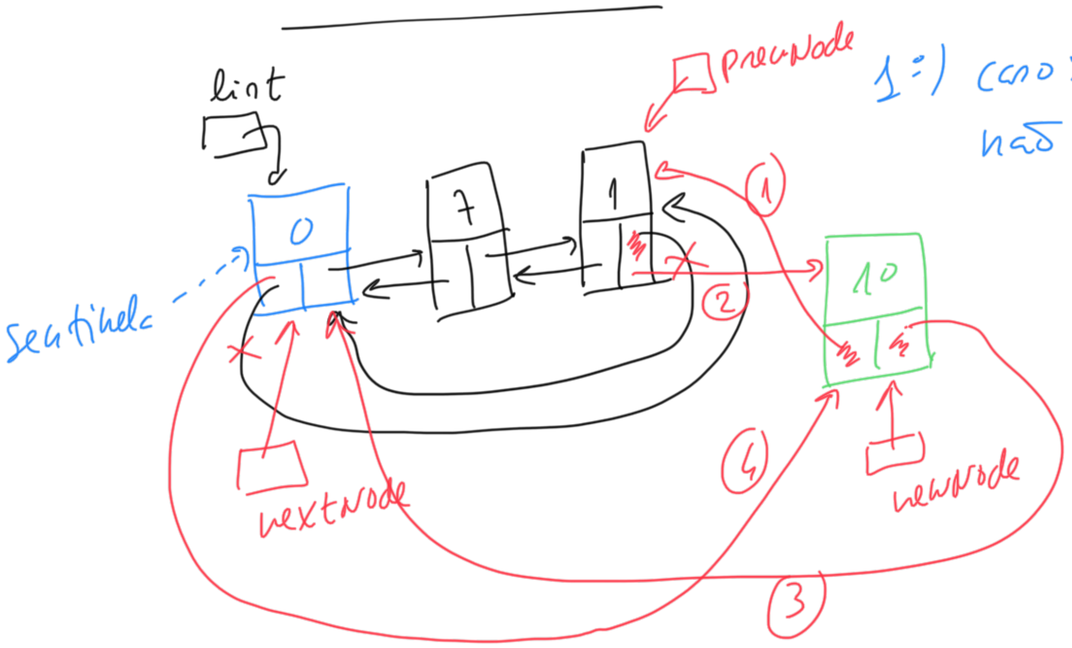
```
class DIntList {
    fun addFront(v: Int) {
        var prevNode : IntNode? = null
        "   nextNode :   "        "
        val newNode = IntNode(v)

        prevNode = list          // sentinela
        nextNode = list.next     // 1º elemento (se existir)
                                 // ou sentinela (se vazio)
  (1)   newNode.prev = prevNode
  (2)   prevNode.next = newNode
  (3)   newNode.next = nextNode
  (4)   nextNode.prev = newNode

        ++size
    }
```

2º caso: lista não vazia



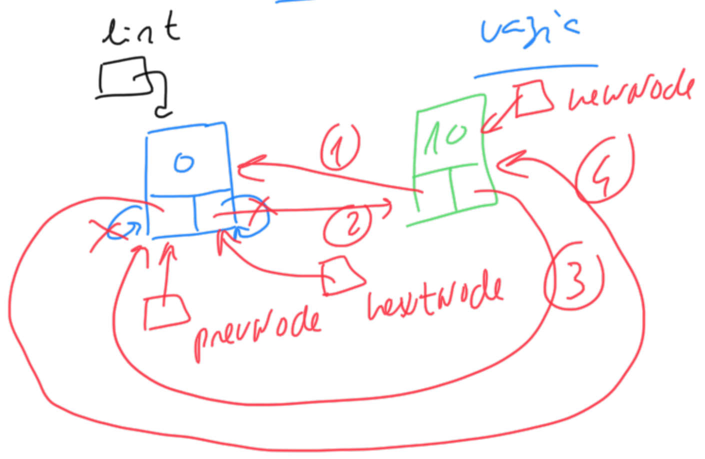
DList - addLast

1º) caso: lista não vazia



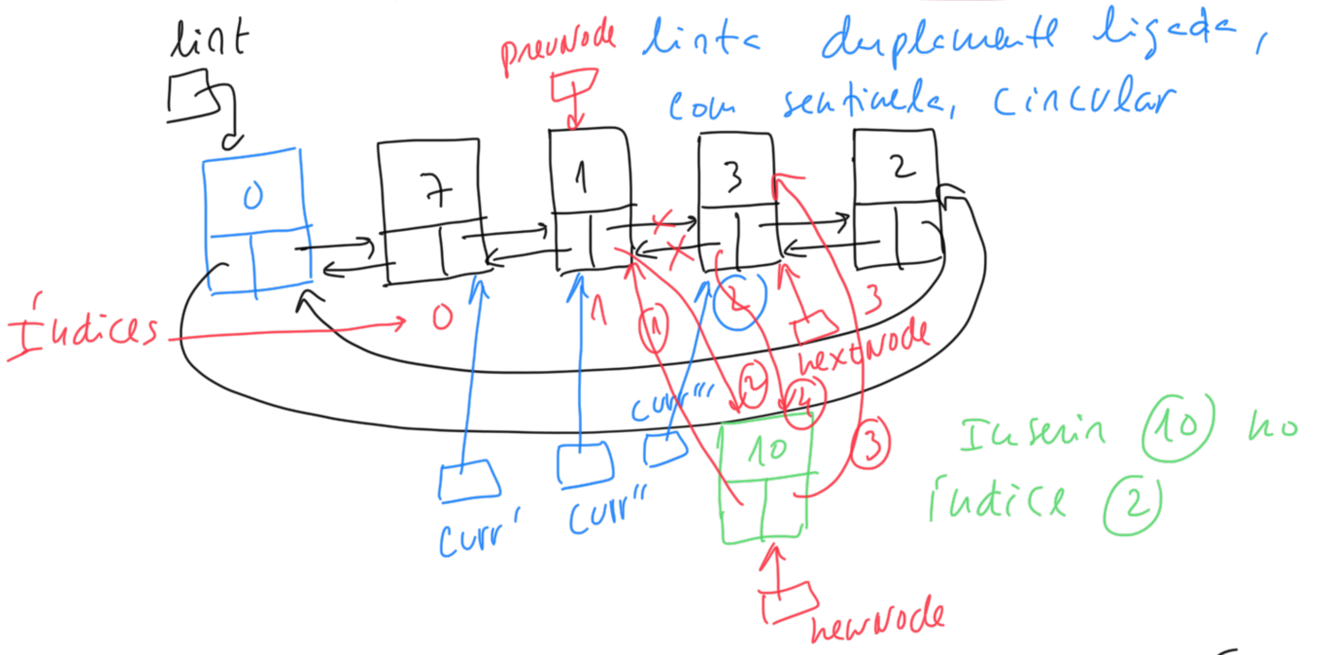
```
fun addLast(v: Int) {
    var prevNode : IntNode ? = null
     "  nextNode :   "       "
    val newNode = IntNode(v)
    prevNode = list.previous      // último nó
    nextNode = list               // a seguir ao último
 ① newNode.previous = prevNode
 ② prevNode.next = newNode
 ③ newNode.next = nextNode
 ④ nextNode.next = newNode
    ++size
}
```

2º) caso: lista vazia



DList - insert (v, index)

lista duplamente ligada, com sentinela, circular



```
fun insert(v: Int, index: Int) { → Pré-condições:
    var prevNode : IntNode? = null         index ∈ [0, size]
    "   nextNode : "      "
    "   curr     : "      "
    val newNode = IntNode(v)

    curr = list.next      // 1º elem.

    var i = 0
    while ( i < index ) {
        curr = curr.next
        ++i
    }
    prevNode = curr.previous
    nextNode = curr
(1) newNode.previous = prevNode
(2) prevNode.next = newNode
(3) newNode.next = nextNode
(4) nextNode.previous = newNode

    ++size
}
```

T.P.C.

```
<E> insertInOrder(
    v: E,
    cmp: Comparator<E>
)
```

↓

```
interface
  Comparator<E> {
    fun compare(
        e1: E,
        e2: E): Int
}
```

}

$\swarrow$

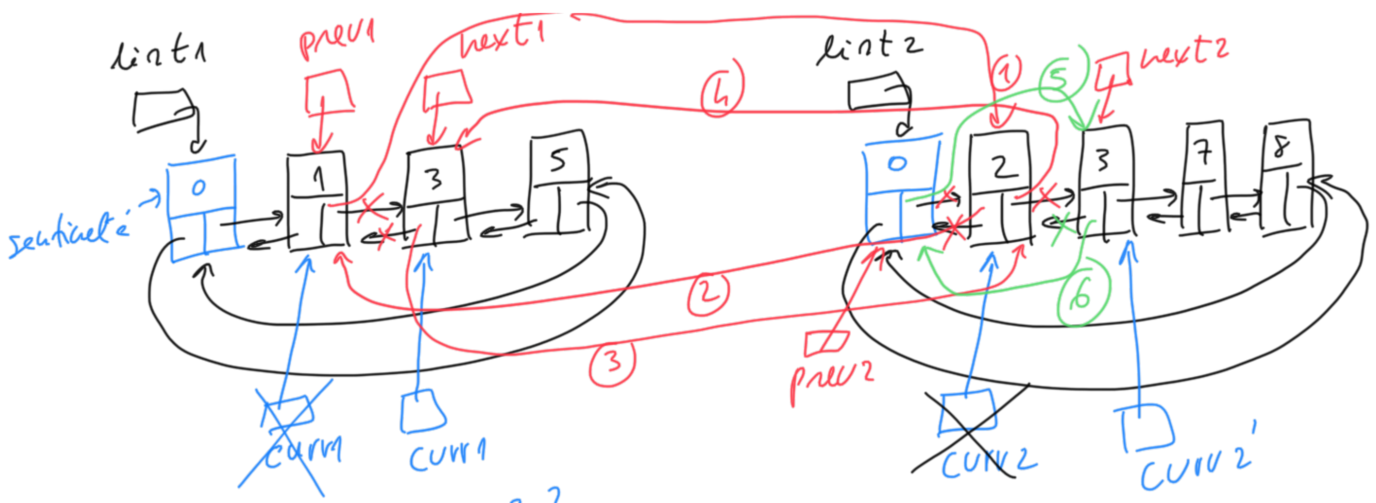
< 0, se $l_1 < l_2$

$= 0$, se $l_1 = l_2$

> 0, se $l_1 > l_2$

DList - merge

```
fun <E> merge( list1 : Node<E>, list2: Node<E>,
                    cmp: Comparator<E>) : Node<E> {
    // listas duplamente ligadas, circulares, com
    // sentinela, e ordenadas crescentemente pelo
    // comparador cmp.
    // o método deve fazer merge das duas listas,
    // de forma ordenada, movendo os nós de list2
    // para list1. No final, retorne a list1, e a
    // list2 deve ficar vazia
```



1<2?
Sim

3<2?
Não
⇓
mover nó ② para trás de nó ③

```
var  prev1 = Node<E>? = null
 "   next1 =  "          "
 "   prev2 =  "          "
 "   next2 =  "          "
 "   curr1 =  "          "
 "   curr2 =  "          "

curr1 = list1.next
curr2 = list2.next
while (curr1 != list1 && curr2 != list2) {
    if (cmp.compare(curr1.value, curr2.value)
                           <= 0) { // value1 <=
                                          value2
            curr1 = curr1.next
    }
    else { // value1 > value2
           // ligar nó com value2 na list1

        prev1 = curr1.previous
        next1 = curr1
        prev2 = curr2.previous
        next2 = curr2.next
```

5.

```
        (1) prev1.next = curr2
        (2) curr2.previous = prev1
        (3) next1.previous = curr2
        (4) curr2.next = next1
        (5) prev2.next = next2
        (6) next2.previous = prev2
        curr2 = next2
    }
}
if (curr2 != list2) { // list2 ainda não acabou
    prev1 = curr1.previous //último de list1
    next1 = curr1 //sentinela de list1
    // ligar list2 no fim de list1
    val last2 = list2.previous
    curr2.previous = prev1
    prev1.next = curr2
    next1.previous = last2
    last2.next = next1
}
// colocar list2 → vazia
list2.previous = list2
list2.next = list2.previous
return list1
}
```