Algoritmos e Estruturas de Dados

TAD TwoWayList
Implementações com estruturas de dados dinâmicas

LEI - Licenciatura em Eng. Informática

2025/26

FACULDADE DE CIÊNCIAS E TECNOLOGIA

Uma sequência
TAD TwoWayList

- O TAD ***TwoWayList*** é uma coleção de elementos, em que cada elemento está associado a uma dada posição (e.g. uma playlist,...)

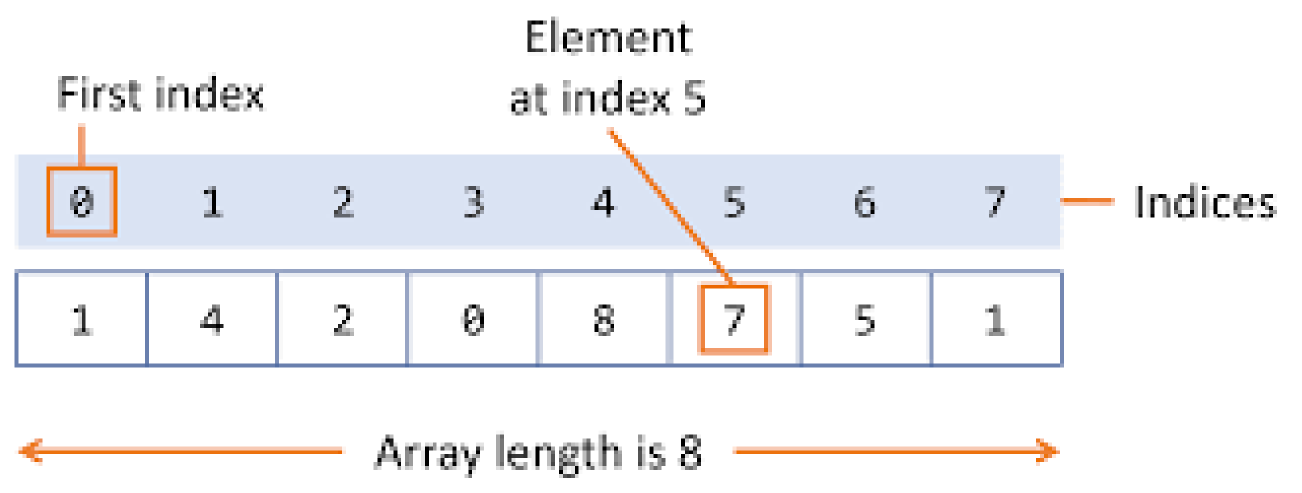
- Representa uma sequência de elementos a que podemos aceder através da posição, como a lista.

- As operações incluem todas as da lista, mais:
  - “twoWayIterator”, retorna um iterador que permite percorrer a sequência nos dois sentidos.

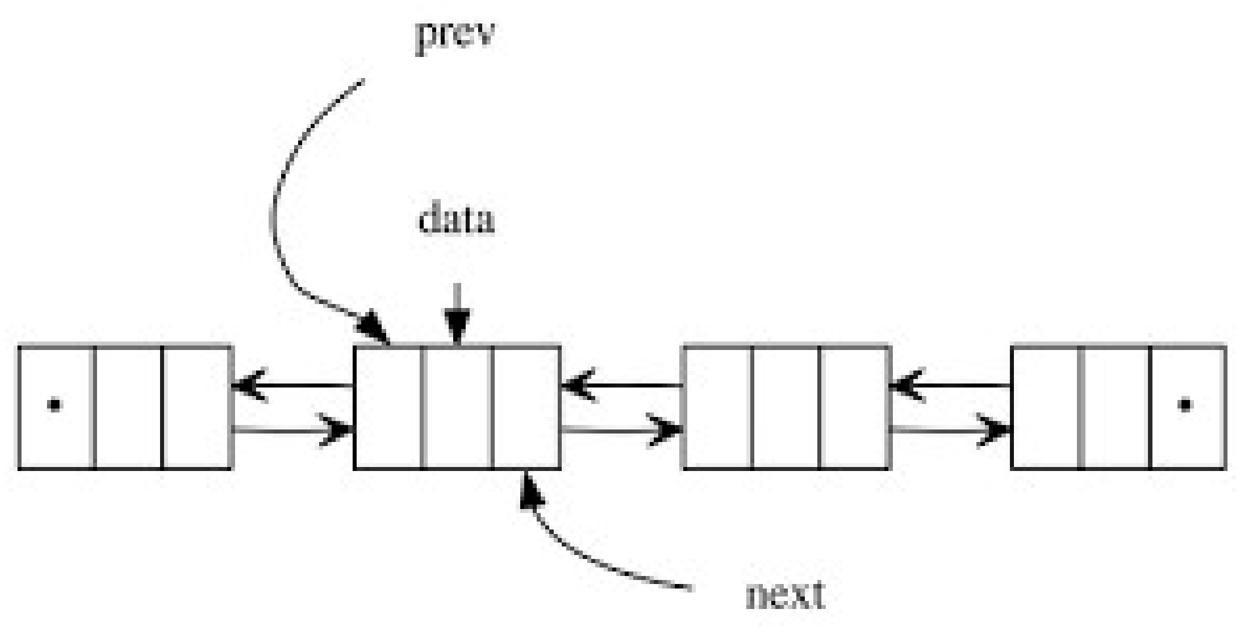
TAD TwoWayList

```java
package dataStructures;
/**
 * Two-Way List
 *
 * @author AED team
 * @version 1.0
 *
 * @param <E> Generic Element
 */
public interface TwoWayList<E> extends List<E> {
   /**
    * Returns a two-way iterator of the elements in the list.
    *
    * @return Itwo-Way terator of the elements in the list
    */
   TwoWayIterator<E> twoWayIterator();
}
```

Possíveis estruturas de dados

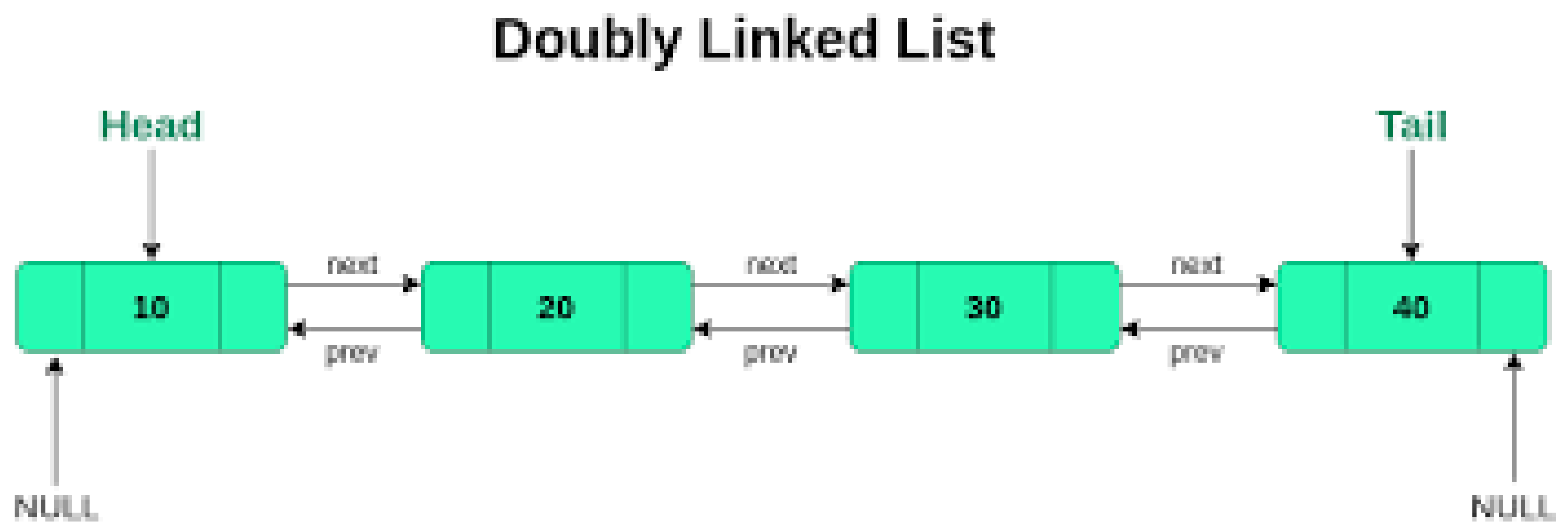


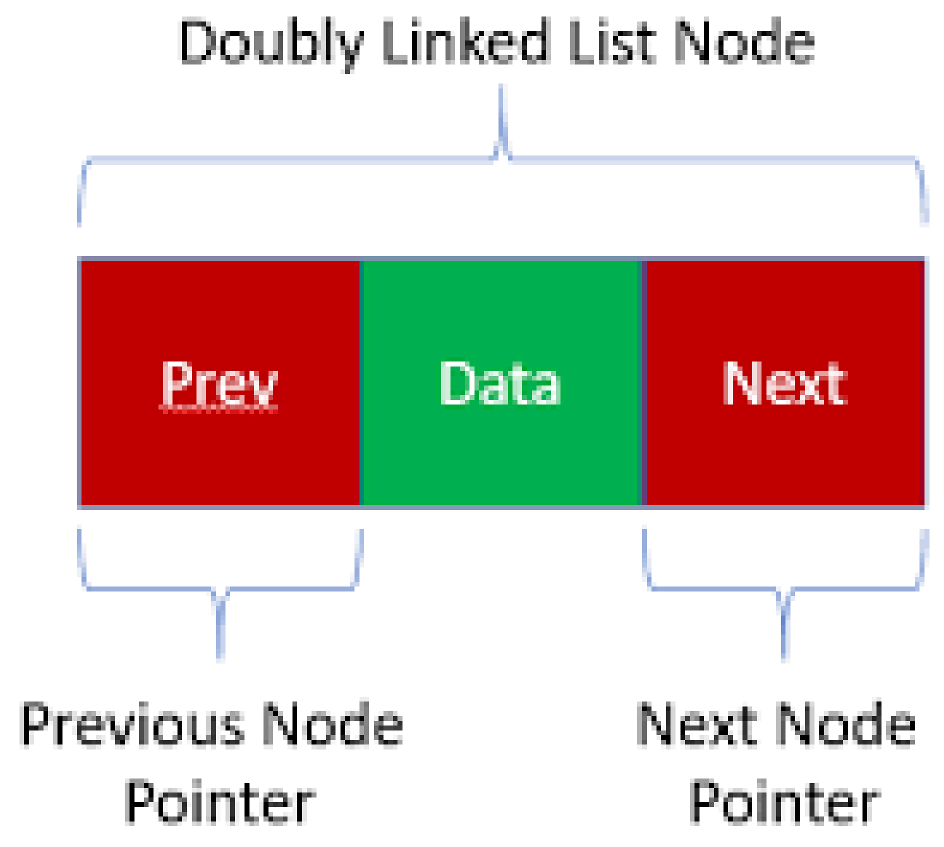
Vetores



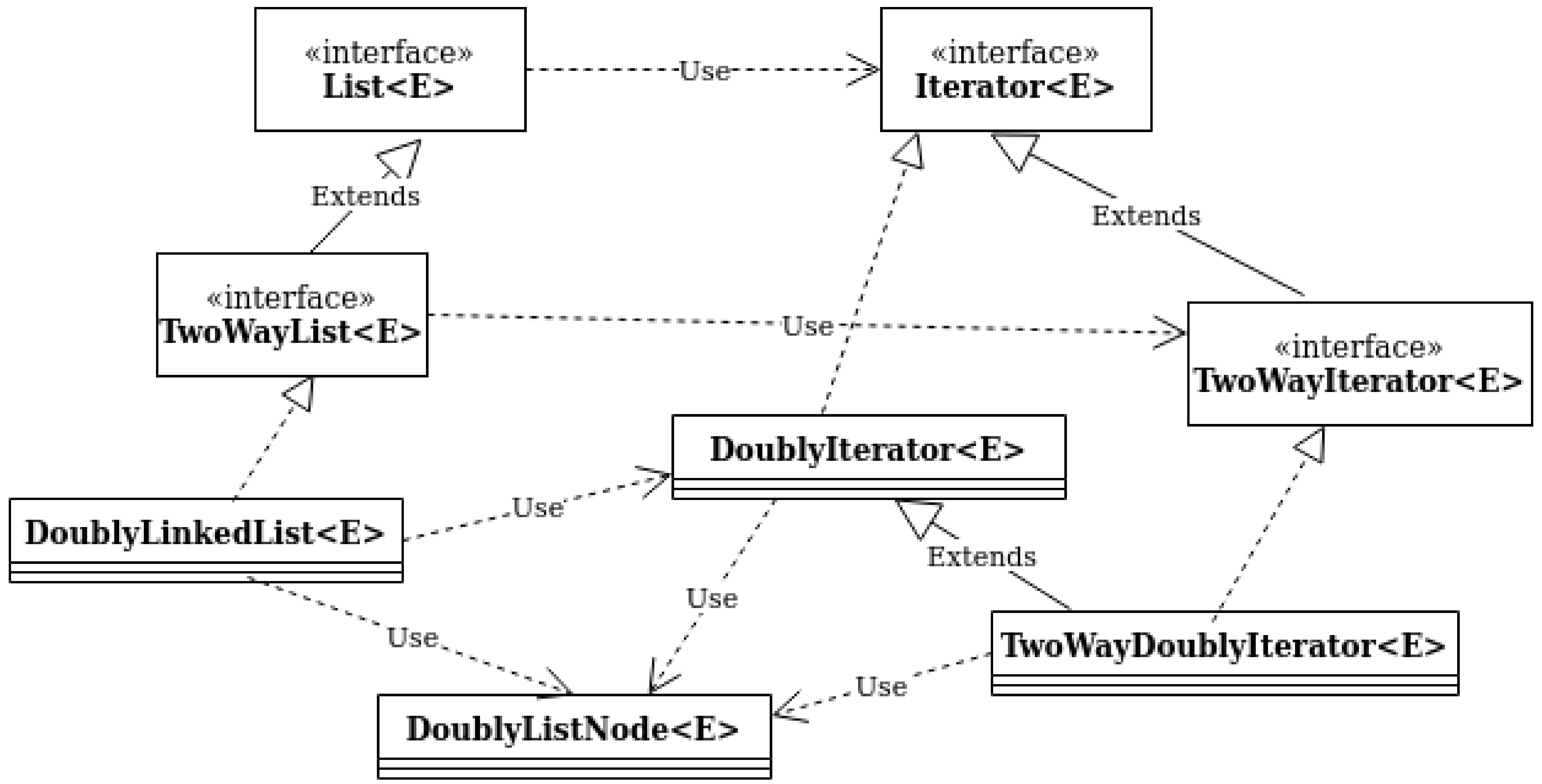
Lista duplamente ligada

Lista Duplamente Ligada



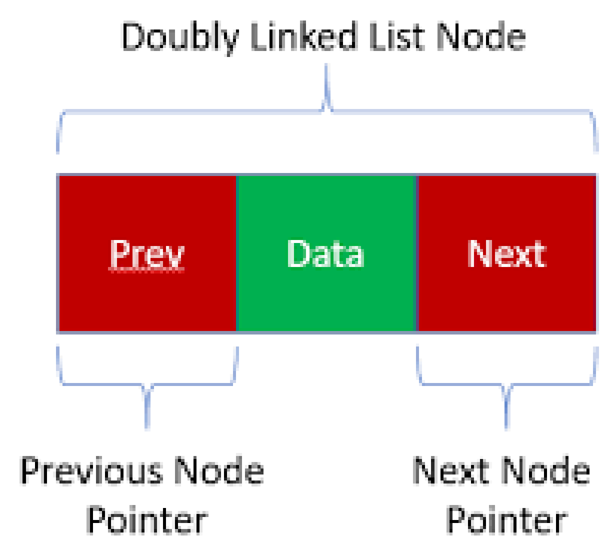


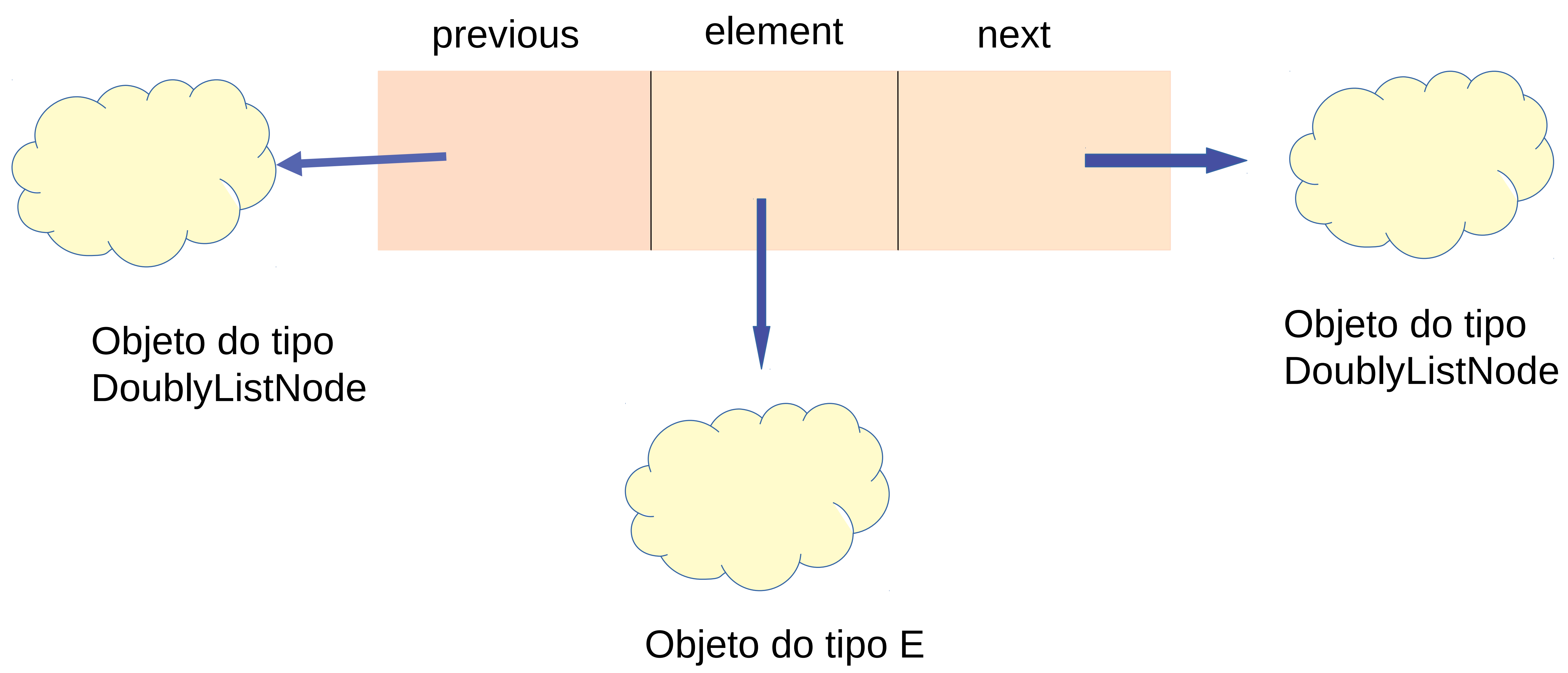
Classes a implementar



Classe DoublyListNode<E> (1)

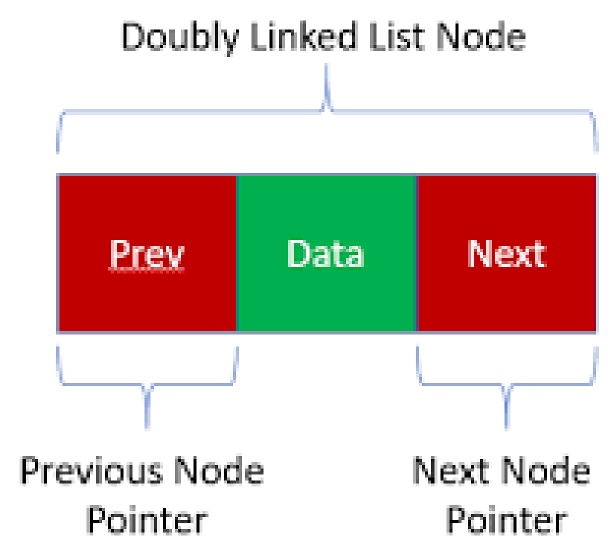
```
package dataStructures;
import java.io.Serializable;

/**
    * Double List Node Implementation
    * @author AED  Team
    * @version 1.0
    * @param <E> Generic Element
    *
    */
class DoublyListNode<E> implements Serializable {
  /**
     * Element stored in the node.
     */
     private E element;

     /**
     * (Pointer to) the previous node.
     */
     private DoublyListNode<E> previous;

     /**
     * (Pointer to) the next node.
     */
     private DoublyListNode<E> next;
```

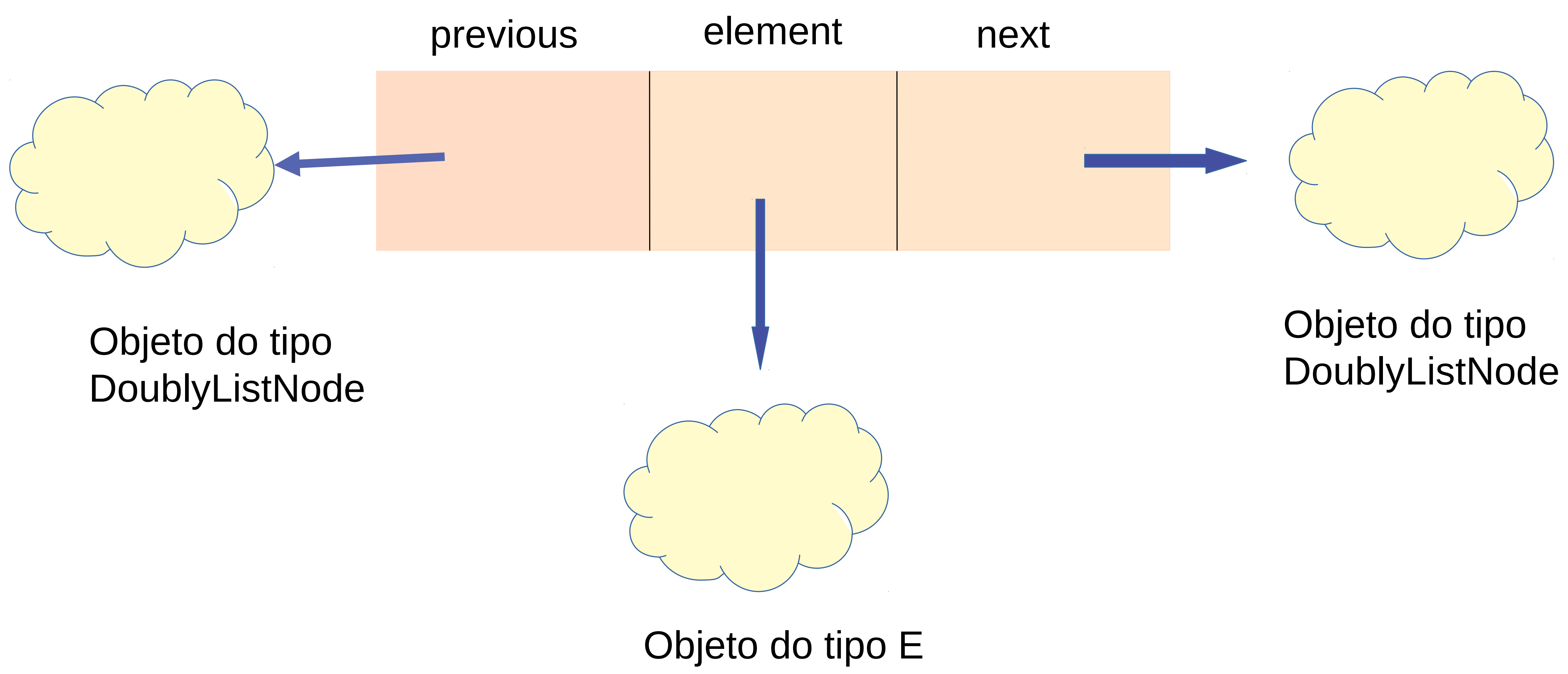




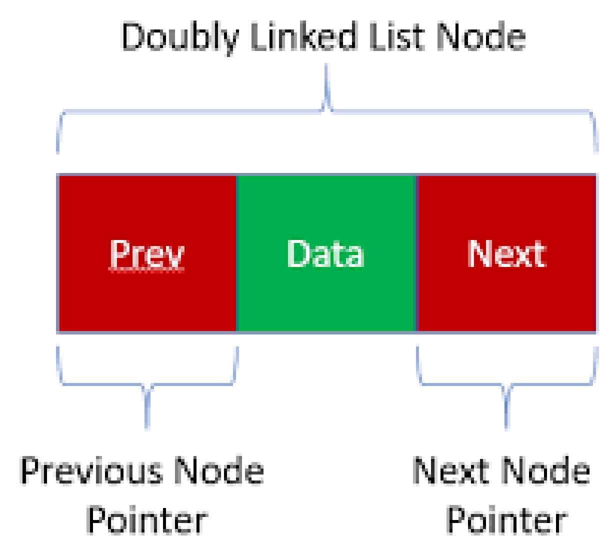
Classe DoublyListNode<E> (2)

```
/**
 * @param theElement - The element to be contained in the node
 * @param thePrevious - the previous node
 * @param theNext - the next node
 */
public DoublyListNode(E theElement, DoublyListNode<E> thePrevious,
                 DoublyListNode<E> theNext ) {
   //TODO: Left as an exercise.
}
/**
 * @param theElement to be contained in the node
 */
public DoublyListNode(E theElement ) {
   //TODO: Left as an exercise.
}
/**
 * @return the element contained in the node
 */
public E getElement( ) {
   return element;
}
```

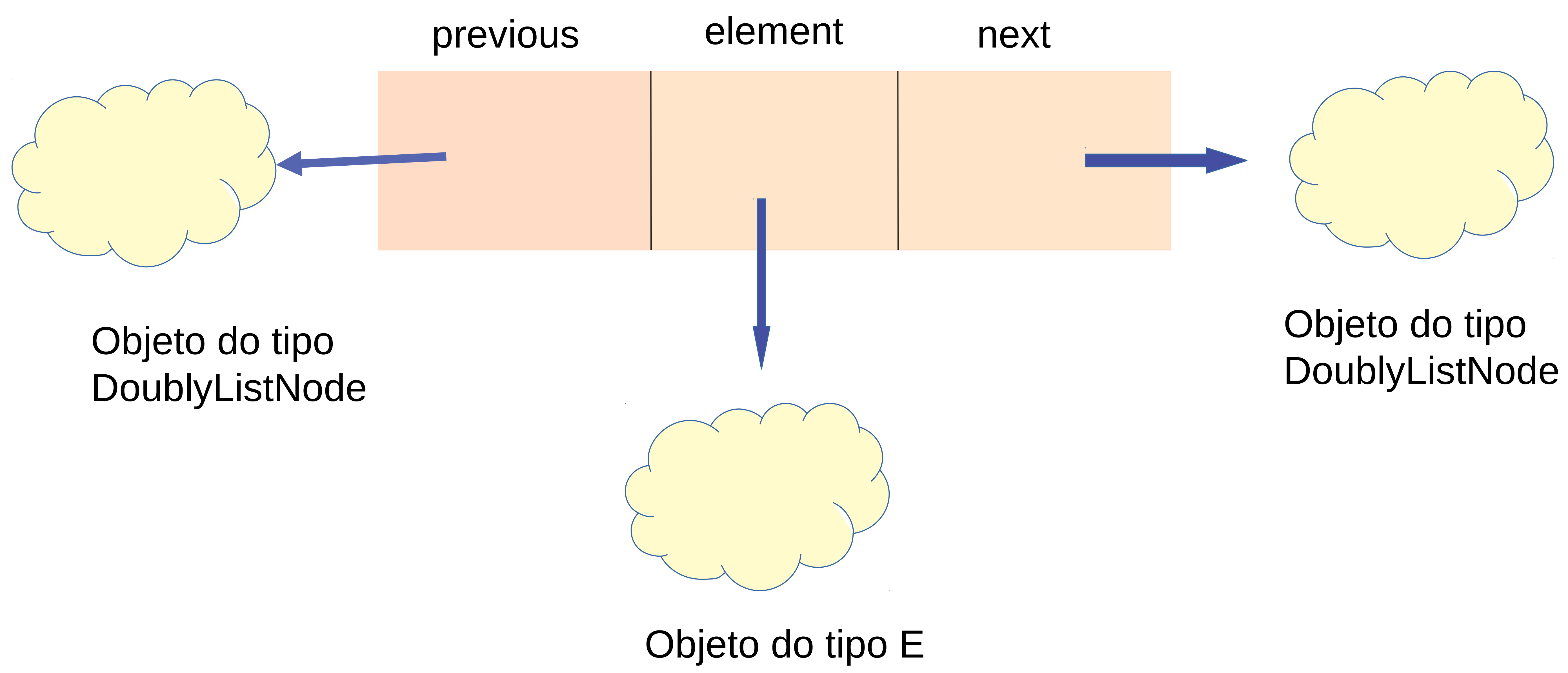




Classe DoublyListNode<E> (4)

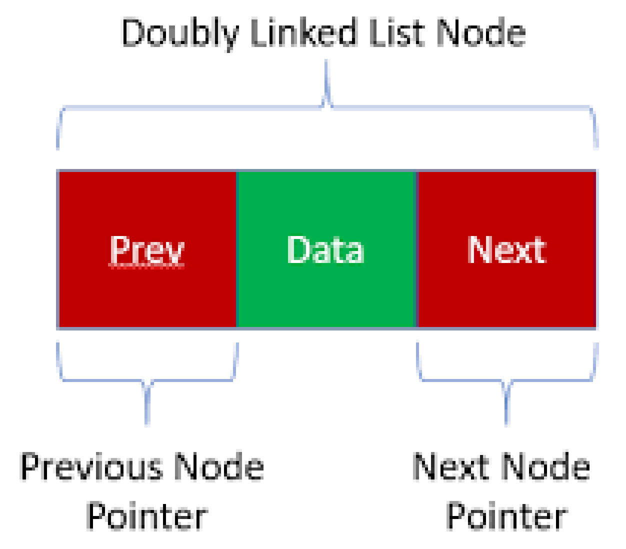


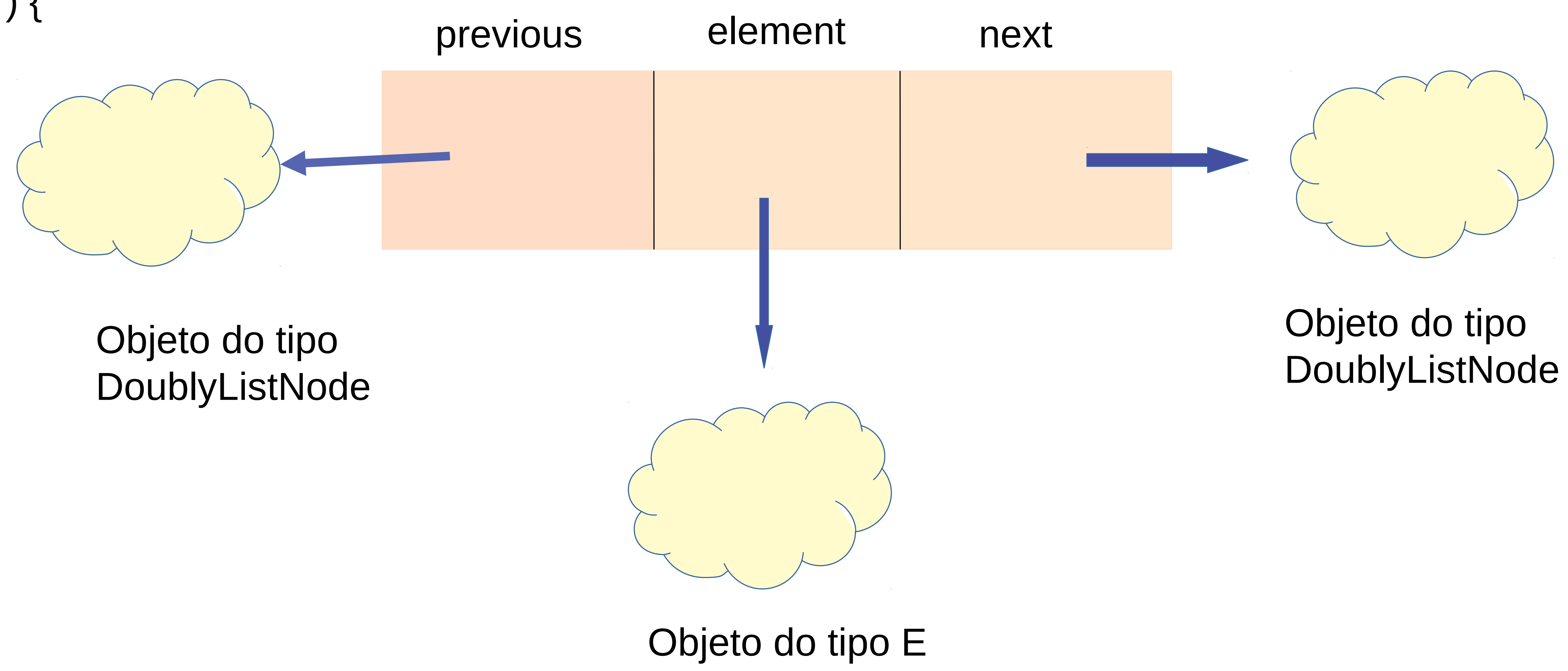
```
/**
 * @return the previous node
 */
public DoublyListNode<E> getPrevious( ) {
    return previous;
}

/**
 * @return the next node
 */
public DoublyListNode<E> getNext( ) {
    return next;
}
```



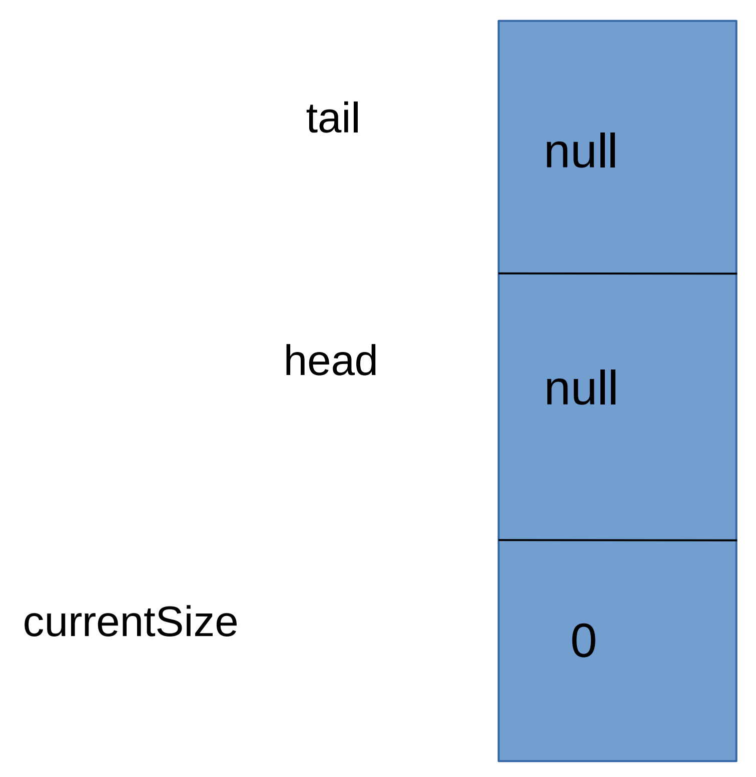
Classe DoublyListNode<E> (5)

```
/**
 * @param newElement - New element to replace the current element
 */
public void setElement( E newElement ) {
   //TODO: Left as an exercise.
}

/**
 * @param newPrevious - node to replace the current previous node
 */
public void setPrevious( DoublyListNode<E> newPrevious ) {
   //TODO: Left as an exercise.

}

/**
 *
 * @param newNext - node to replace the next node
 */
public void setNext( DoublyListNode<E> newNext ) {
   //TODO: Left as an exercise.

}
}
```

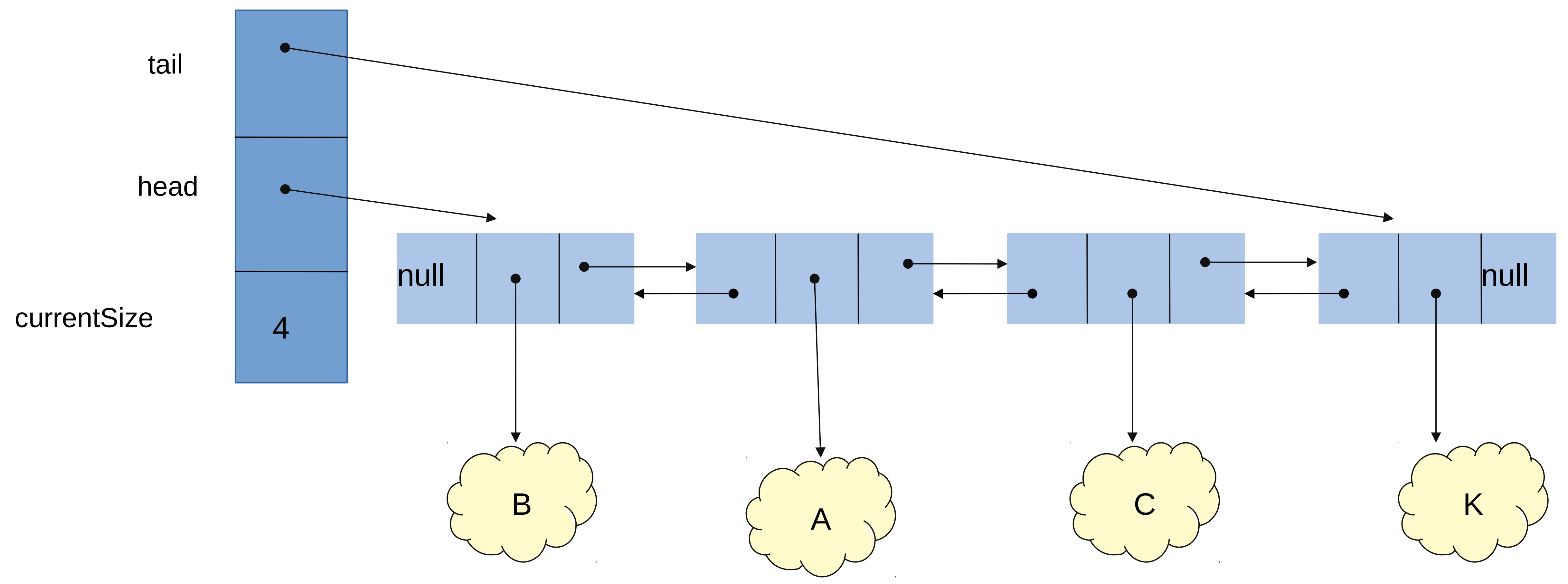




Lista Duplamente Ligada



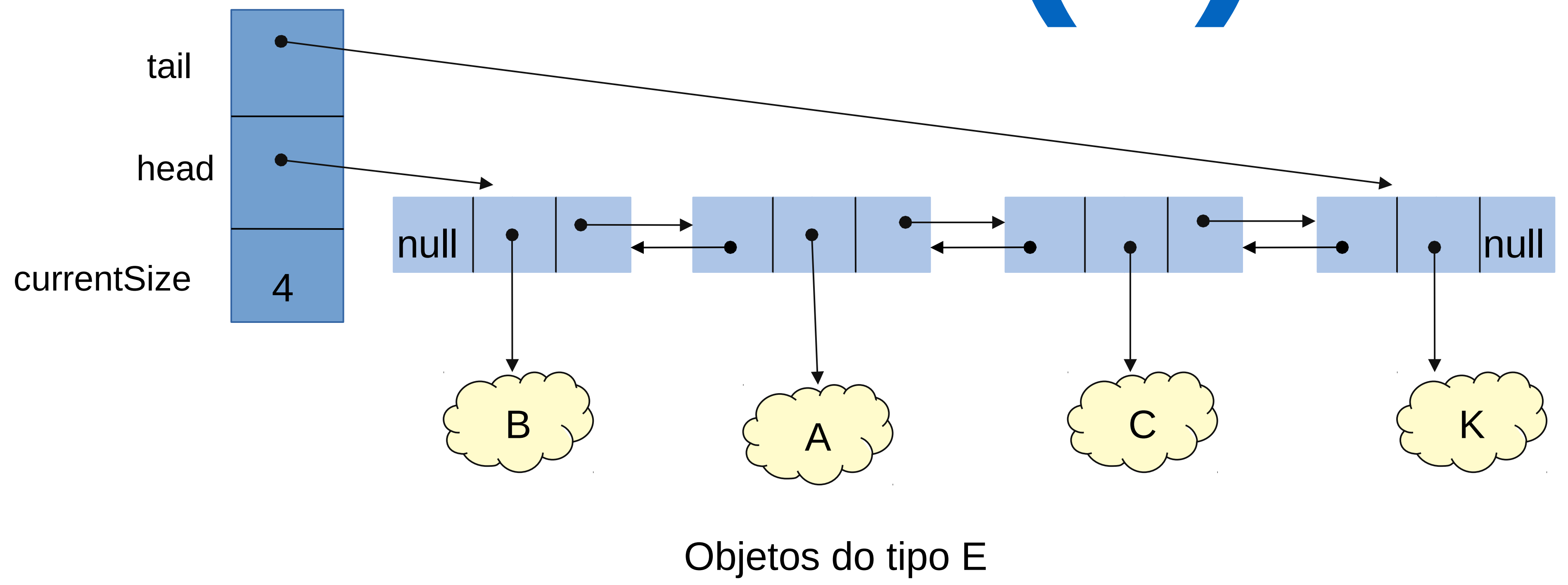
Lista vazia – zero elementos



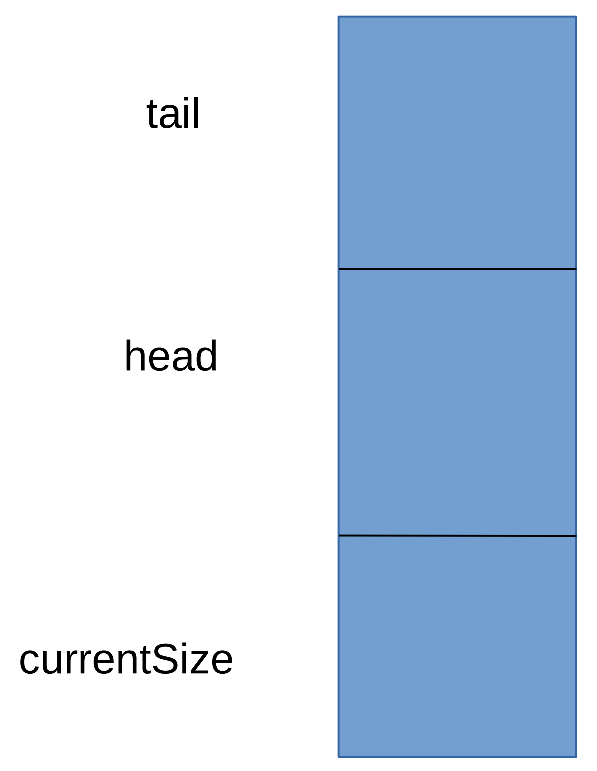
Objetos do tipo E

Lista com 4 elementos

Classe DoublyLinkedList<E> (1)

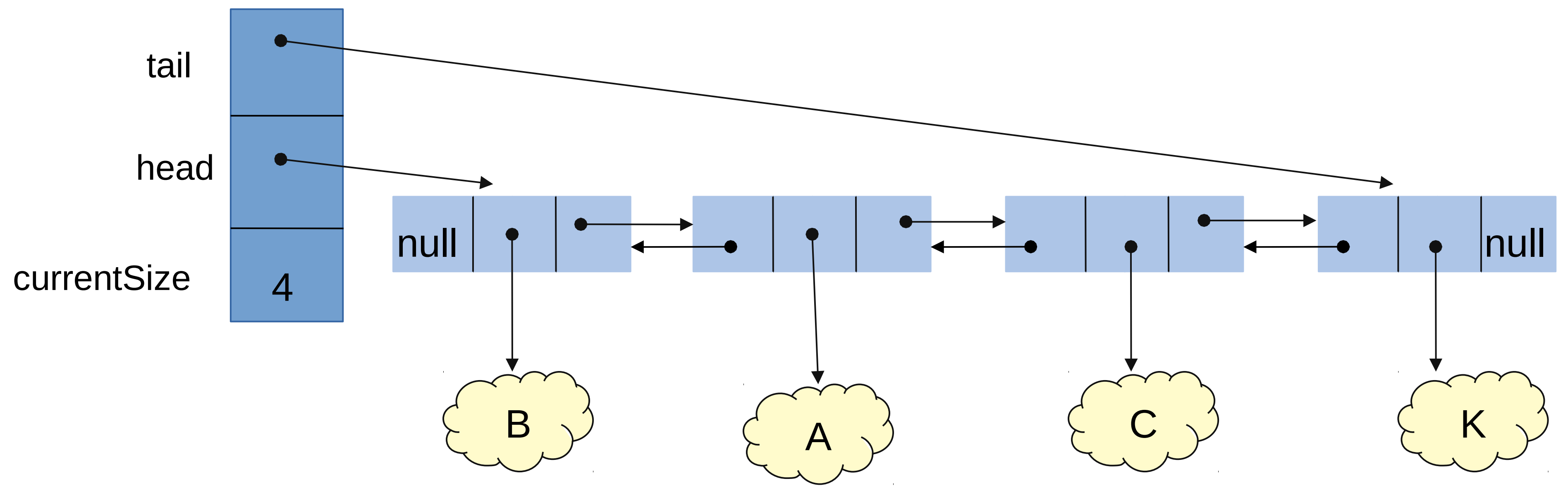


```
package dataStructures;

import dataStructures.exceptions.InvalidPositionException;
import dataStructures.exceptions.NoSuchElementException;

public class DoublyLinkedList<E> implements TwoWayList<E> {
   /**
    *  Node at the head of the list.
    */
   private DoublyListNode<E> head;
   /**
    * Node at the tail of the list.
    */
   private DoublyListNode<E> tail;
   /**
    * Number of elements in the list.
    */
   private int currentSize;
```

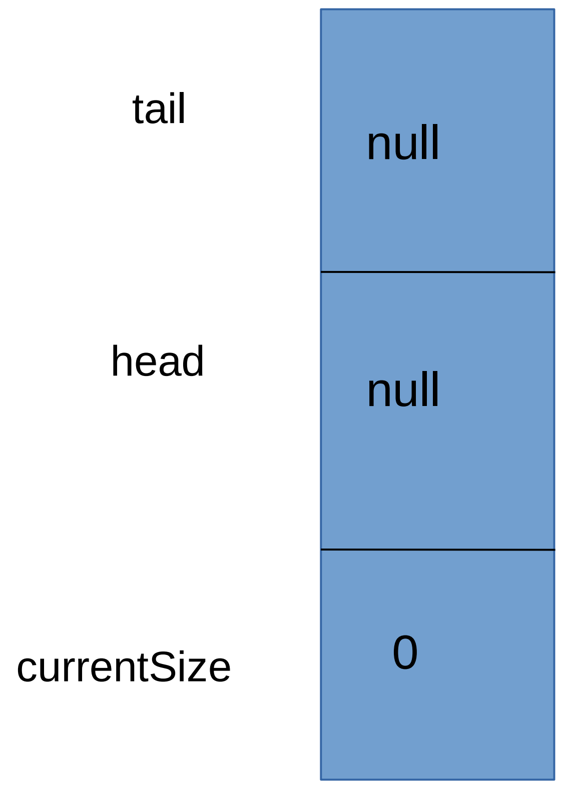


Classe DoublyLinkedList<E> (2)

```
/**
 * Constructor of an empty double linked list.
 * head and tail are initialized as null.
 * currentSize is initialized as 0.
 */
public DoublyLinkedList( ) {
   //TODO: Left as an exercise.
}

/**
 * Returns true iff the list contains no elements.
 * @return true if list is empty
 */
public boolean isEmpty() {
   //TODO: Left as an exercise.
}

/**
 * Returns the number of elements in the list.
 * @return number of elements in the list
 */
public int size() {
   //TODO: Left as an exercise.
}
```

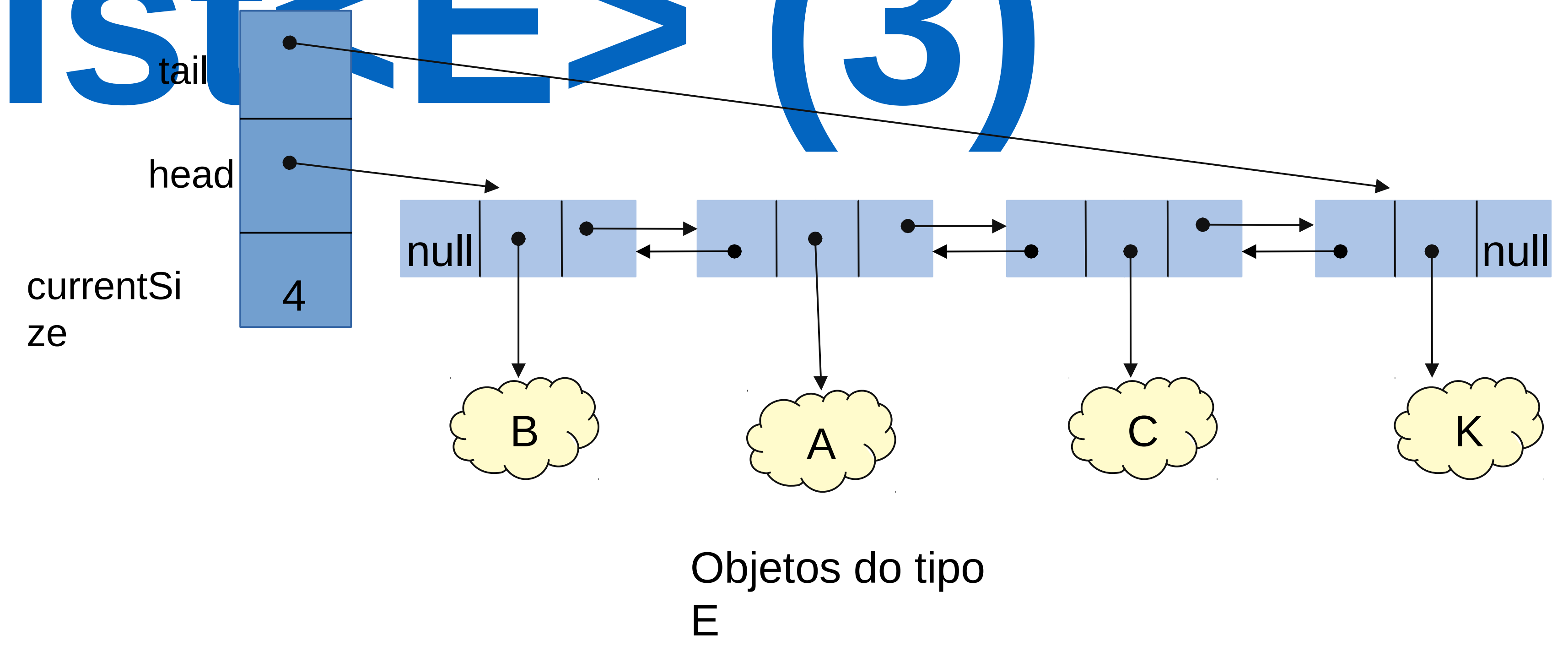


Objetos do tipo E



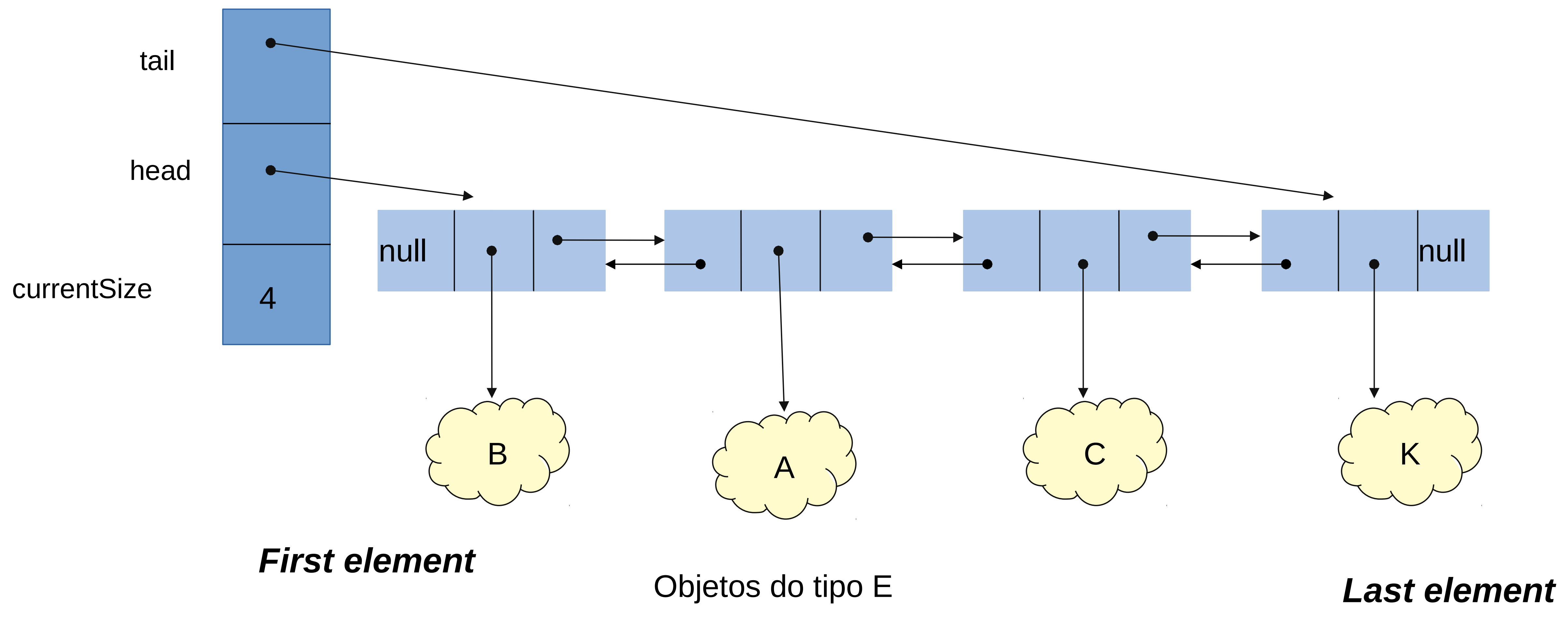
Lista vazia – zero elementos

Classe DoublyLinkedList<E> (3)

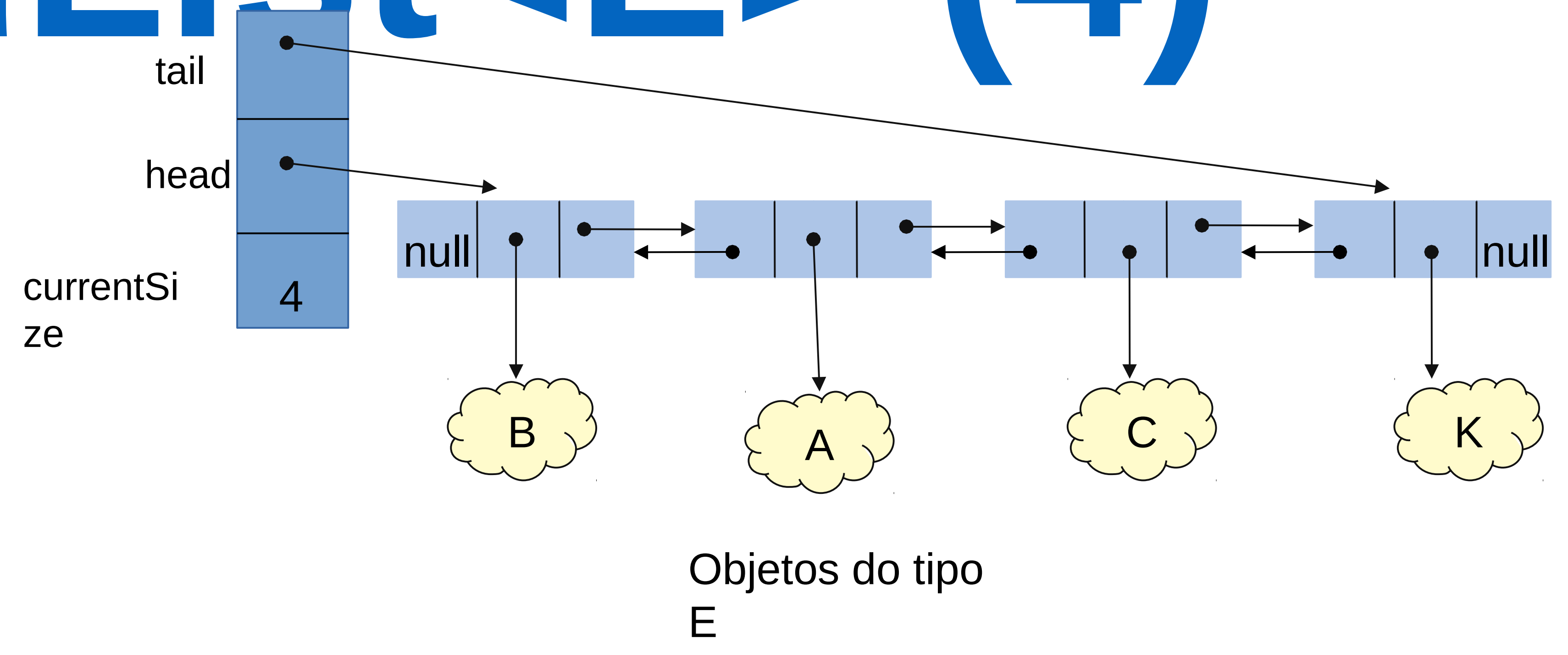


```
/**
 * Returns the first element of the list.
 * @return first element in the list
 * @throws NoSuchElementException - if size() == 0
 */
public E getFirst( ) {
   //TODO: Left as an exercise


}

/**
 * Returns the last element of the list.
 * @return last element in the list
 * @throws NoSuchElementException - if size() == 0
 */
public E getLast( ) {
   //TODO: Left as an exercise.


}
```

Lista com 4 elementos

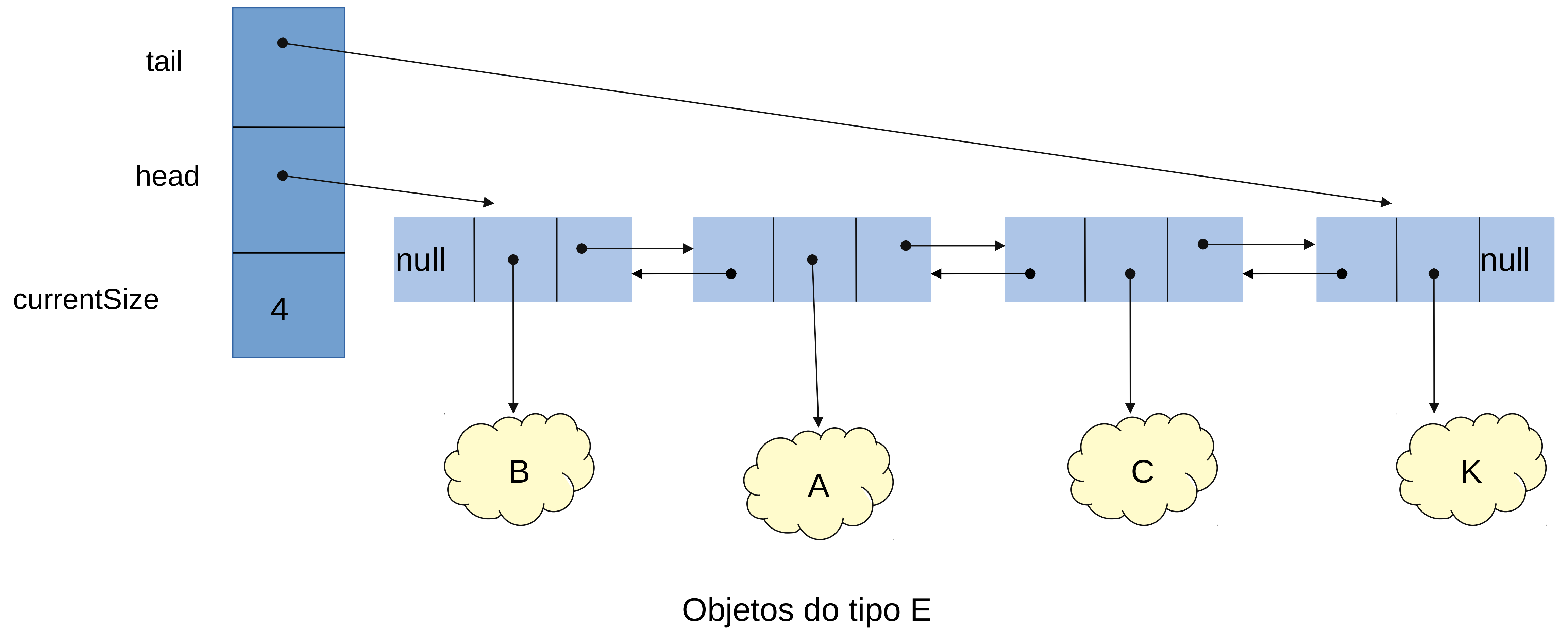


Classe DoublyLinkedList<E> (4)



```
/**
 * Returns the element at the specified position in the list.
 * Range of valid positions: 0, ..., size()-1.
 * If the specified position is 0, get corresponds to getFirst.
 * If the specified position is size()-1, get corresponds to getLast.
 * @param position - position of element to be returned
 * @return element at position
 * @throws InvalidPositionException if position is not valid in the list
 */
public E get( int position ) {
   //TODO: Left as an exercise.


}
```

Lista com 4 elementos

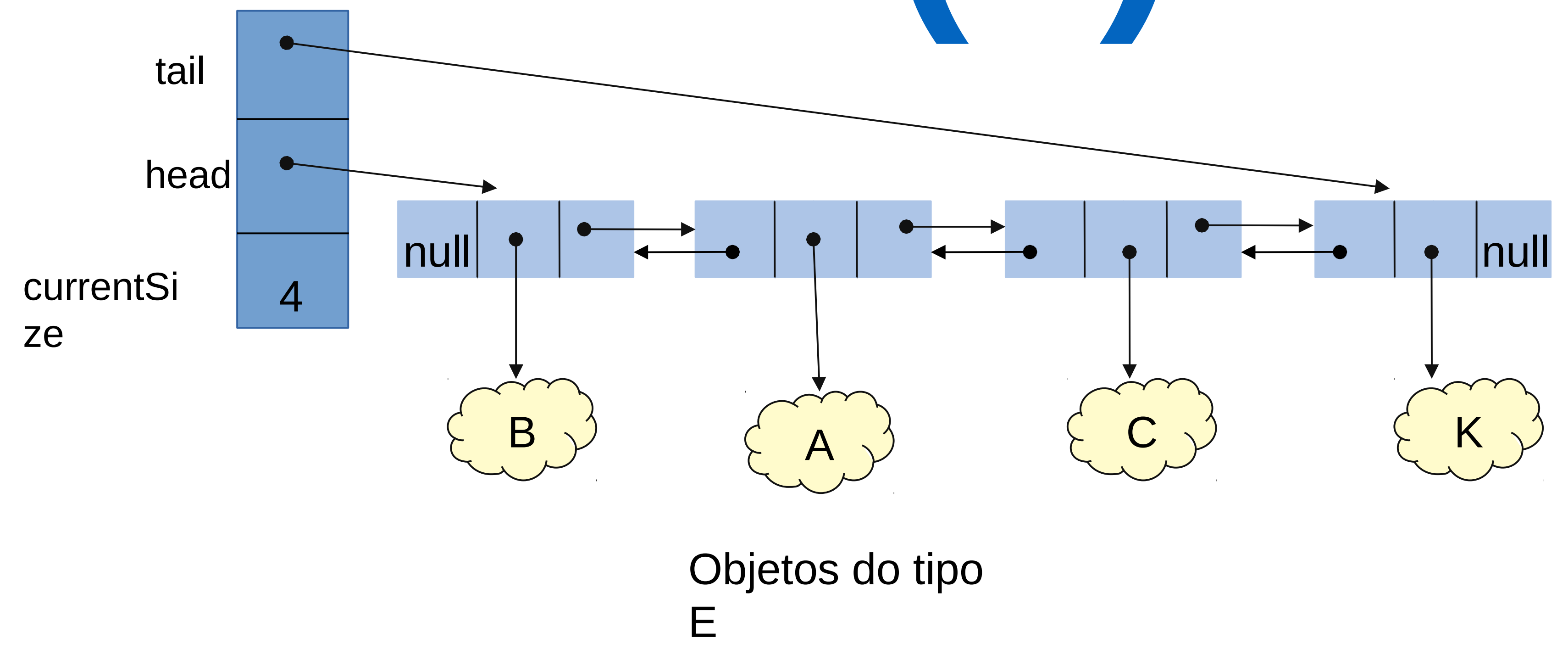


Percurso em Lista

Início → doublyListNode<E> node=head;

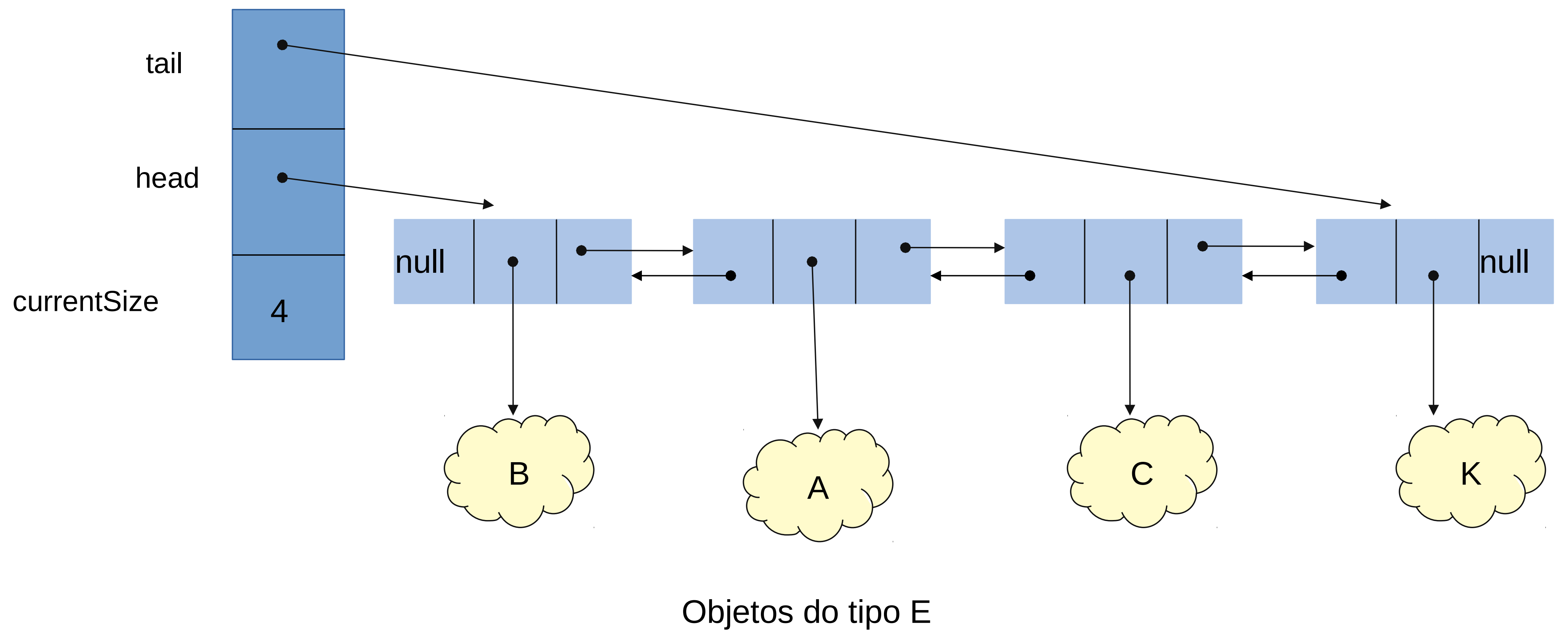
Avanço → node = node.getNext();

Classe DoublyLinkedList<E> (5)



Lista com 4 elementos

```
/**
 * Returns the position of the first occurrence of the specified element
 * in the list, if the list contains the element.
 * Otherwise, returns -1.
 * @param element - element to be searched in list
 * @return position of the first occurrence of the element in the list (or -1)
 */
public int indexOf( E element ) {
   //TODO: Left as an exercise.

}
```



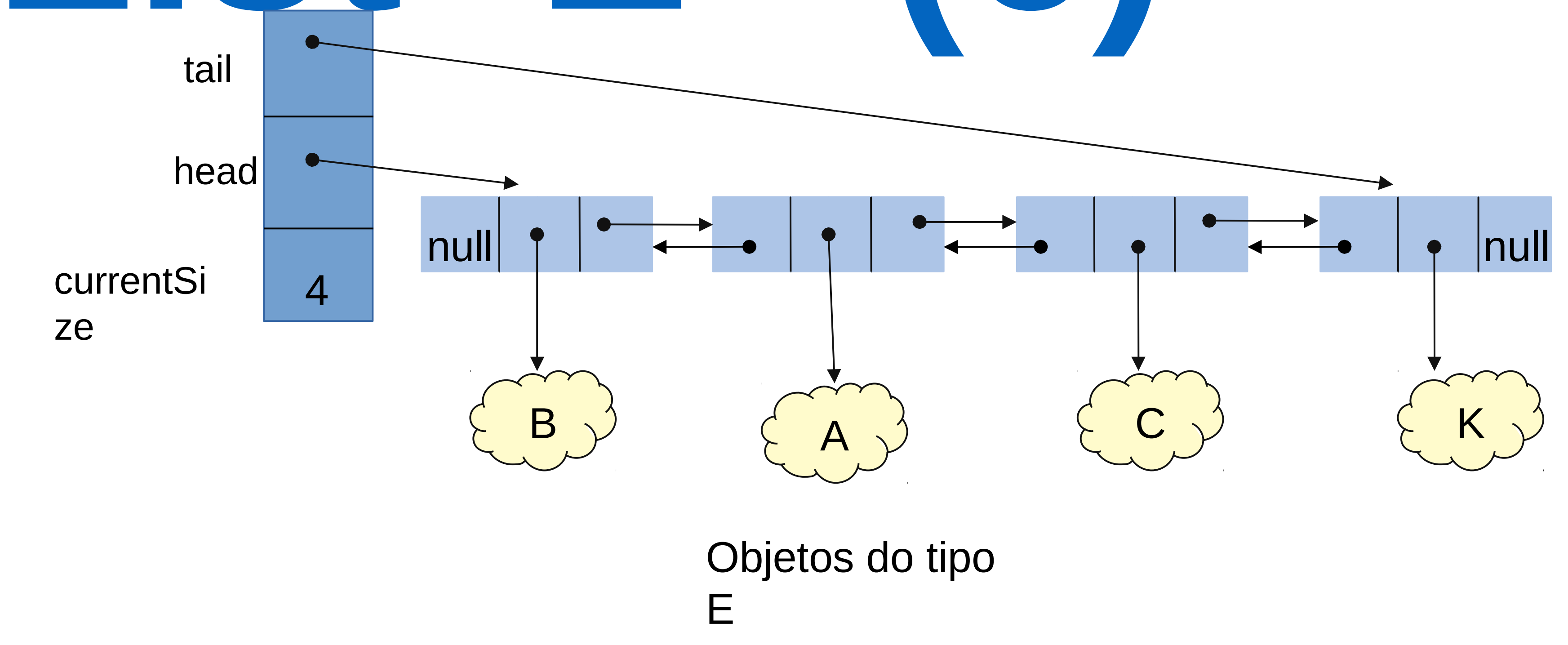
Percurso em Lista

Início → doublyListNode<E> node=head;

Avanço → node = node.getNext();

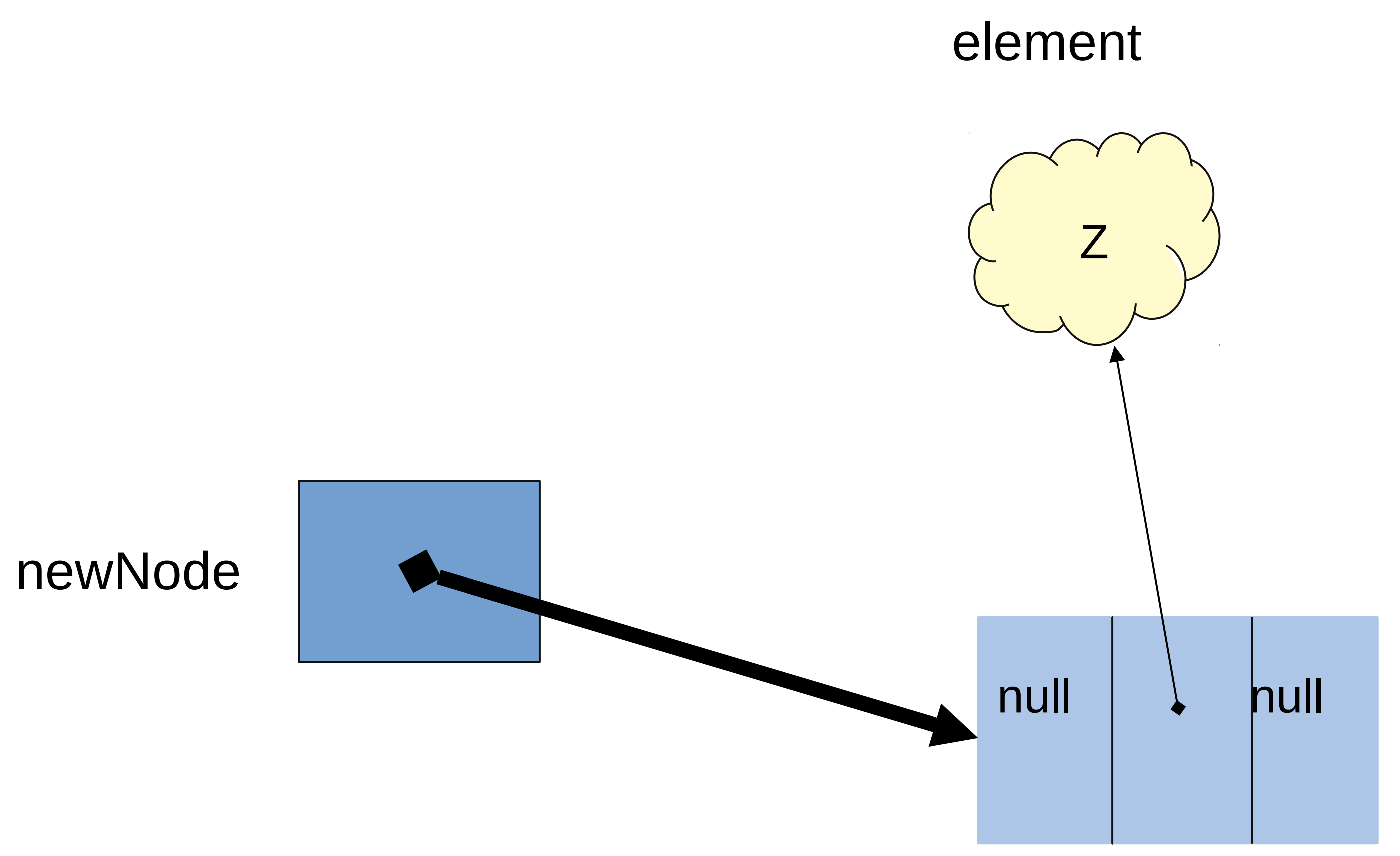
Condição de paragem → node==null

Classe DoublyLinkedList<E> (6)

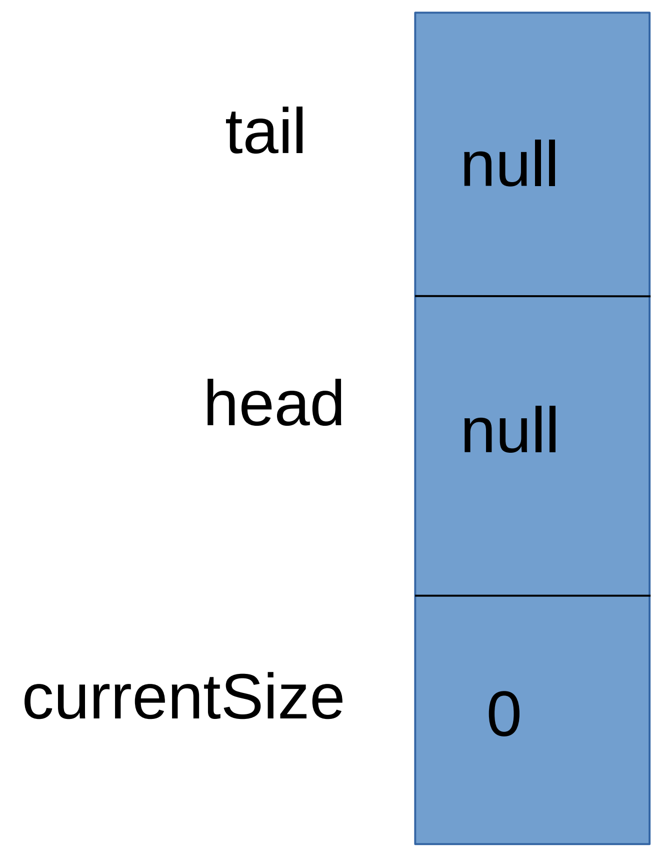


```
/**
 * Inserts the element at the first position in the list.
 * @param element - Element to be inserted
 */
public void addFirst( E element ) {
   //TODO: Left as an exercise.

}
```

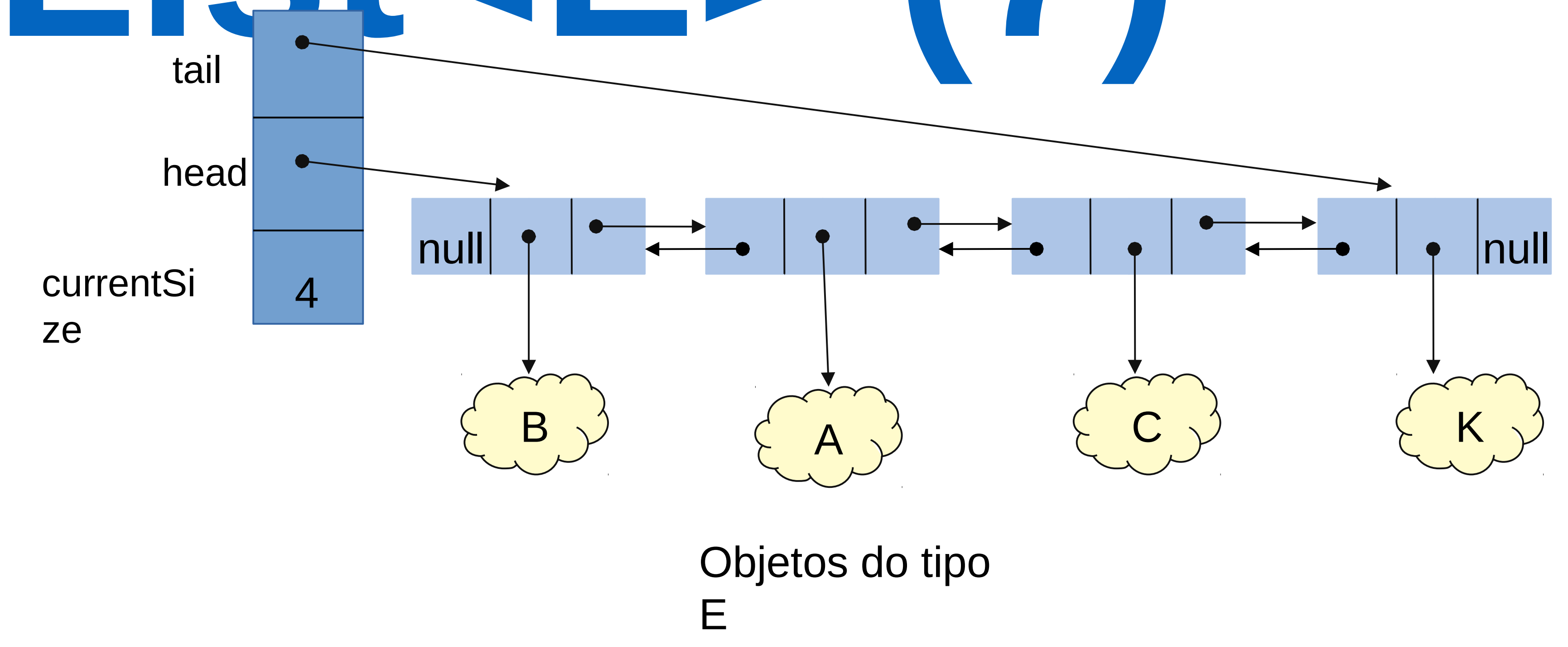
addFirst("Z")



Lista vazia

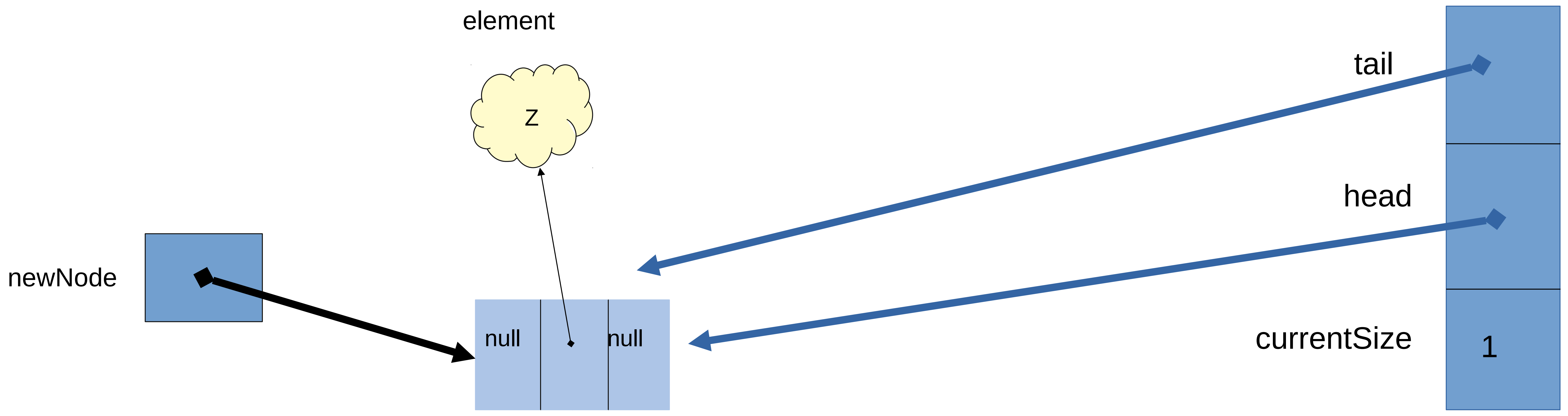


Classe DoublyLinkedList<E> (7)

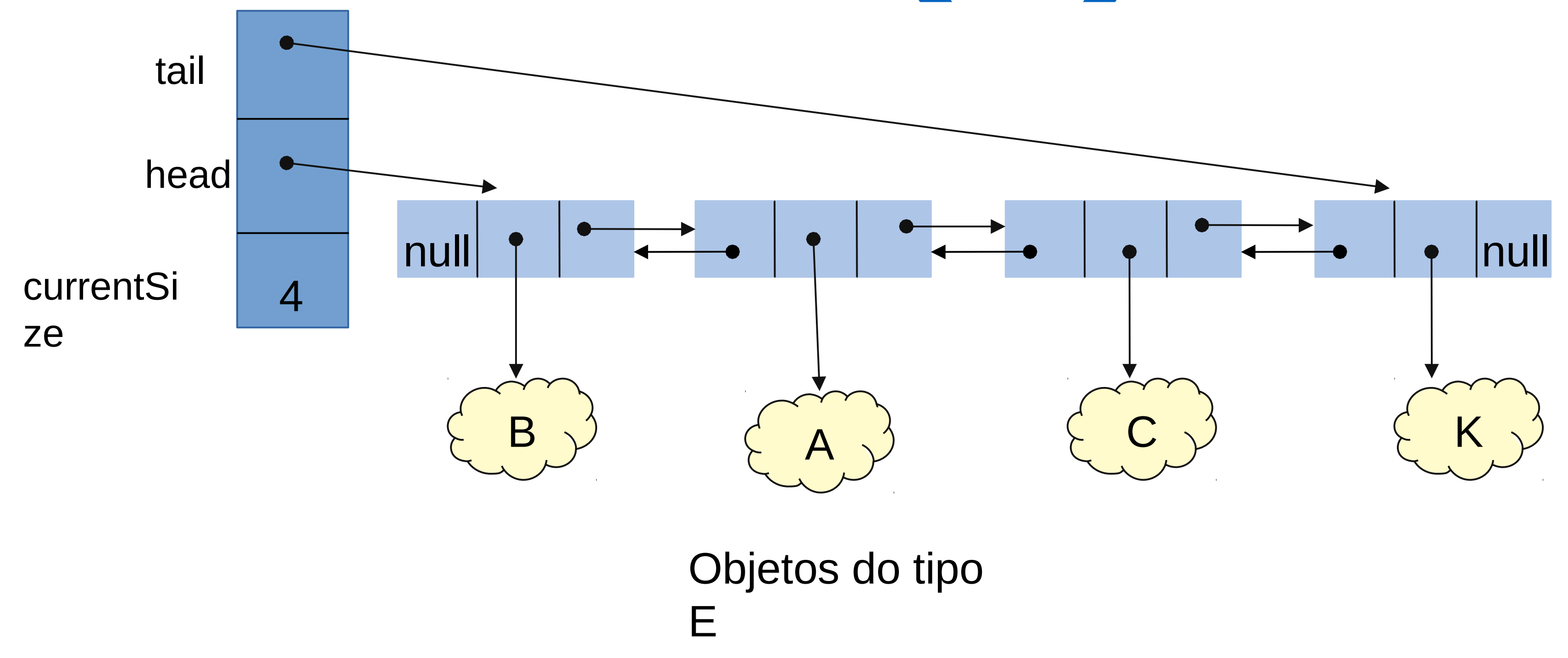


```
/**
 * Inserts the element at the first position in the list.
 * @param element - Element to be inserted
 */
public void addFirst( E element ) {
   //TODO: Left as an exercise.

}
```

addFirst("Z")



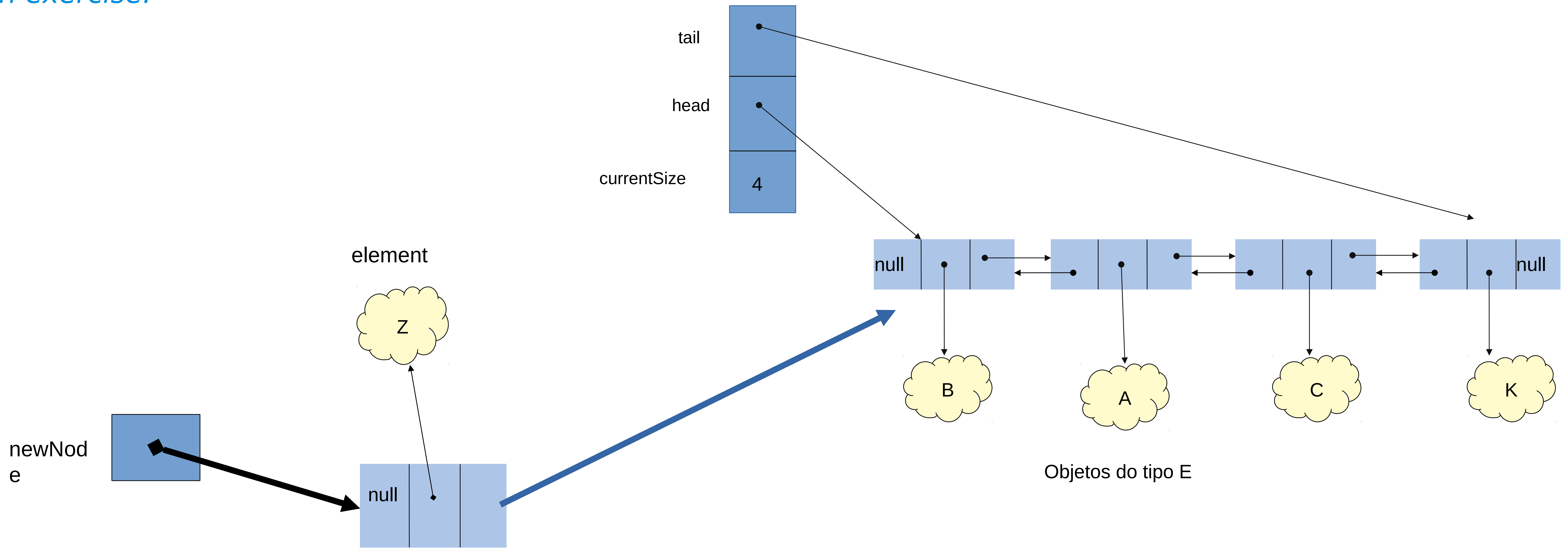
Classe DoublyLinkedList<E> (8)



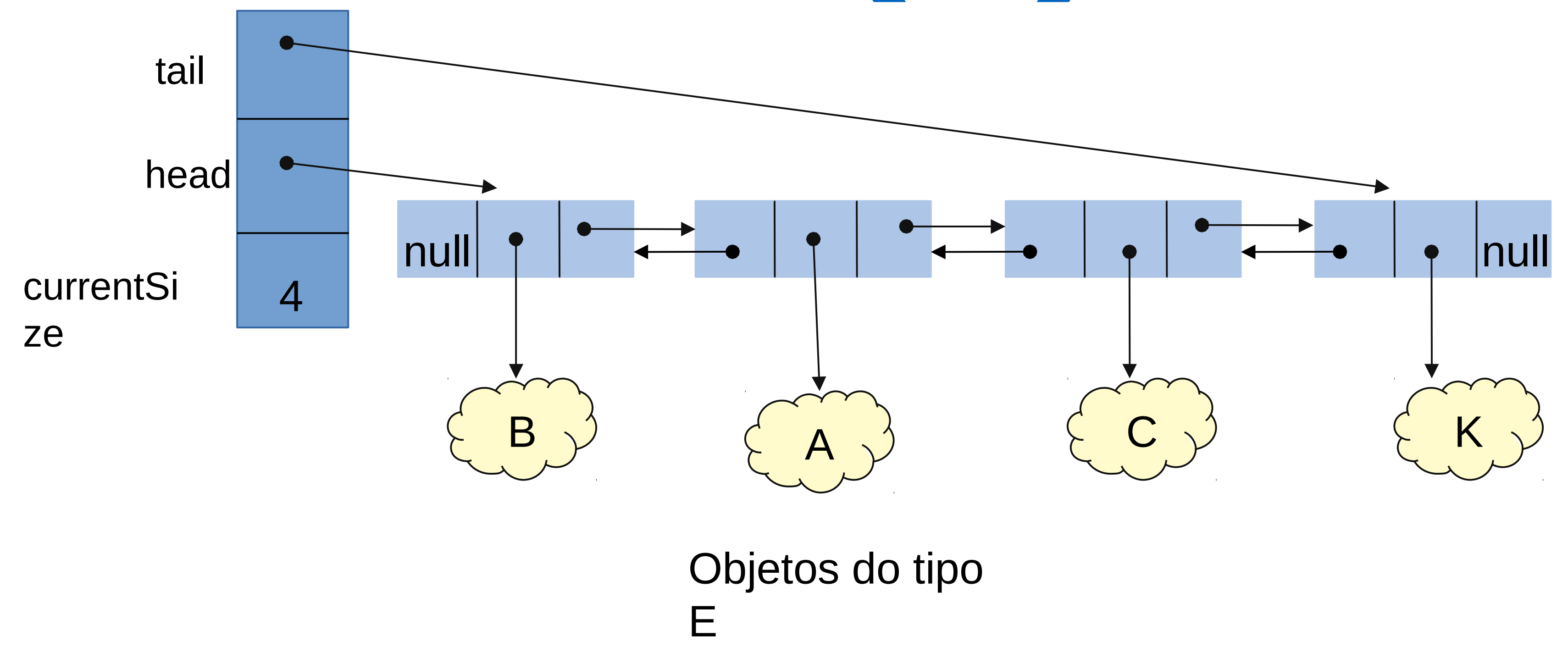
Lista com 4 elementos

```
/**
 * Inserts the element at the first position in the list.
 * @param element - Element to be inserted
 */
public void addFirst( E element ) {
   //TODO: Left as an exercise.

}
```

addFirst("Z")

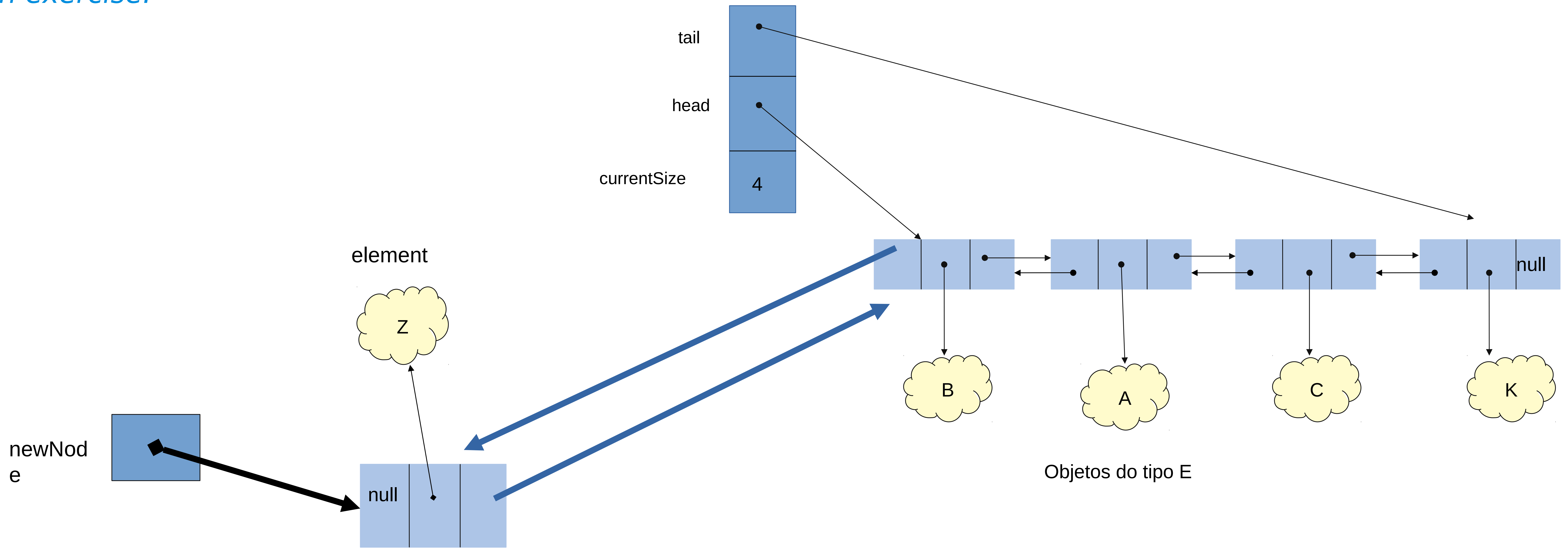


Classe DoublyLinkedList<E> (9)

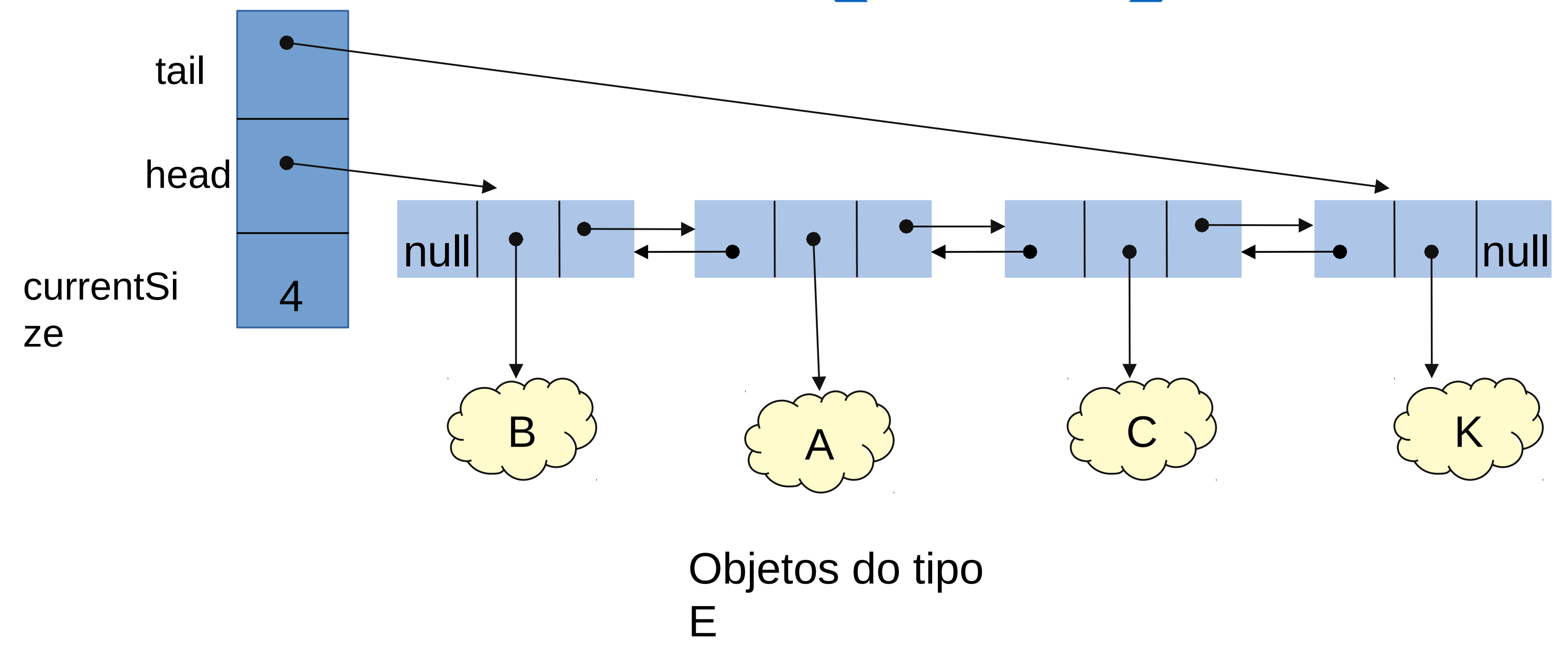


```
/**
 * Inserts the element at the first position in the list.
 * @param element - Element to be inserted
 */
public void addFirst( E element ) {
   //TODO: Left as an exercise.

}
```

addFirst("Z")

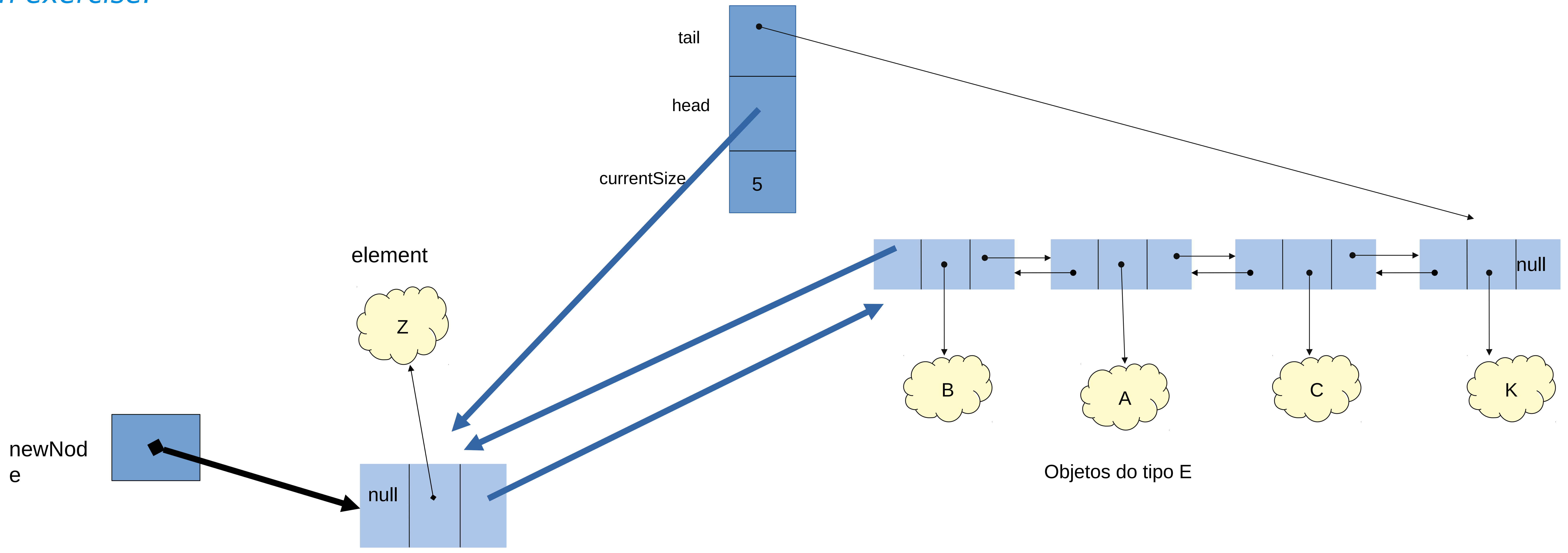


Classe DoublyLinkedList<E> (10)

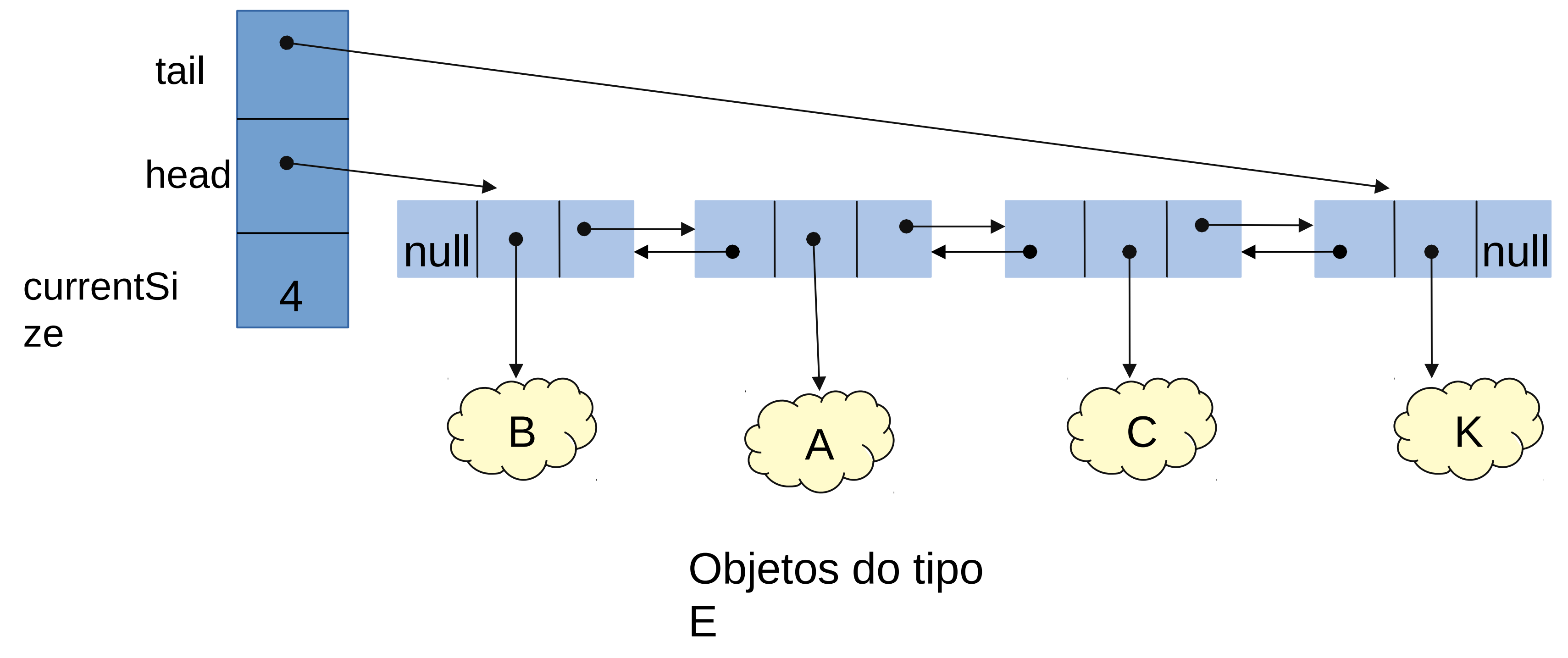


```
/**
 * Inserts the element at the first position in the list.
 * @param element - Element to be inserted
 */
public void addFirst( E element ) {
   //TODO: Left as an exercise.

}
```

addFirst("Z")



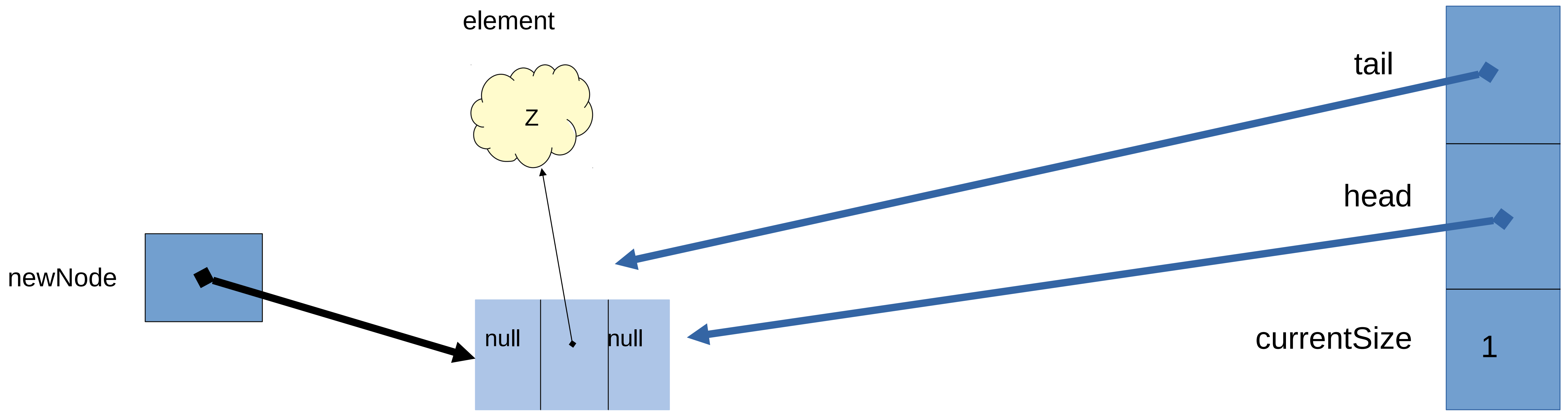
Classe DoublyLinkedList<E> (11)



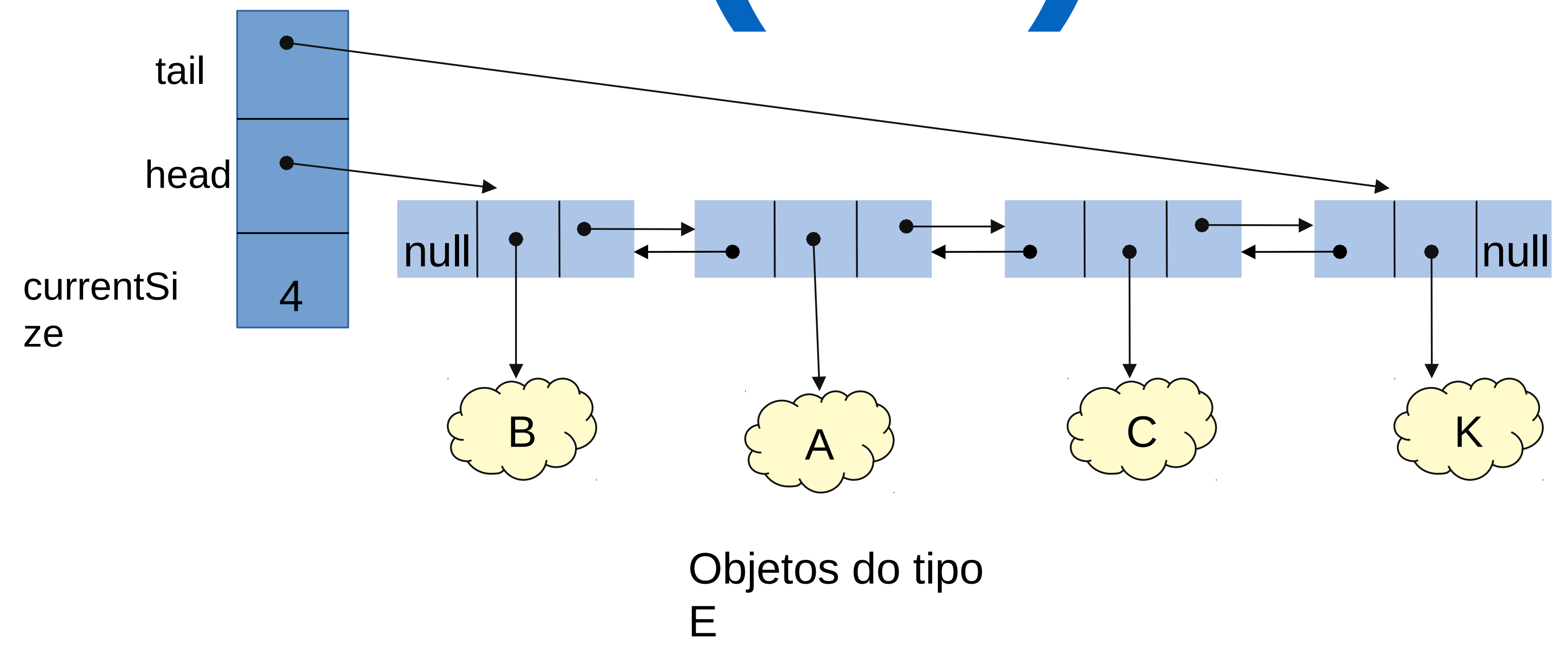
```
/**
 * Inserts the element at the last position in the list.
 * @param element - Element to be inserted
 */
public void addLast( E element ) {
   //TODO: Left as an exercise.

}
```

addLast(“Z”)

Lista vazia

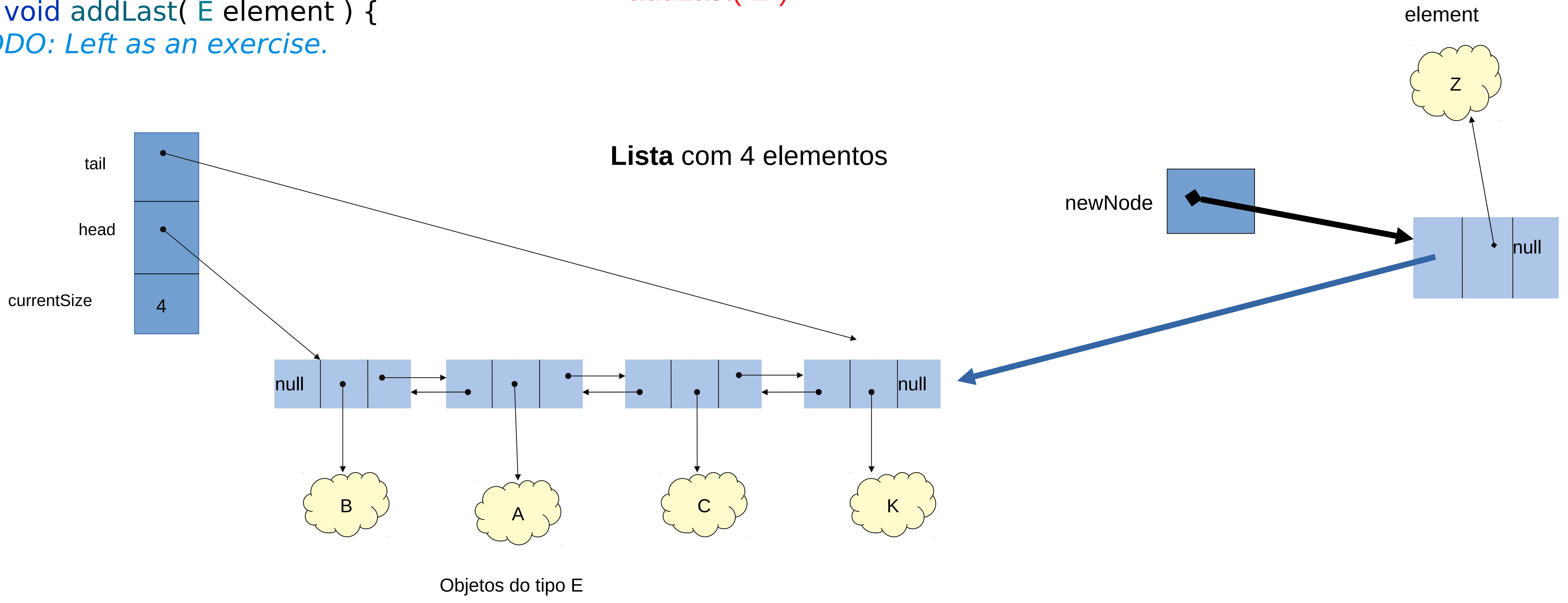


Classe DoublyLinkedList<E> (12)

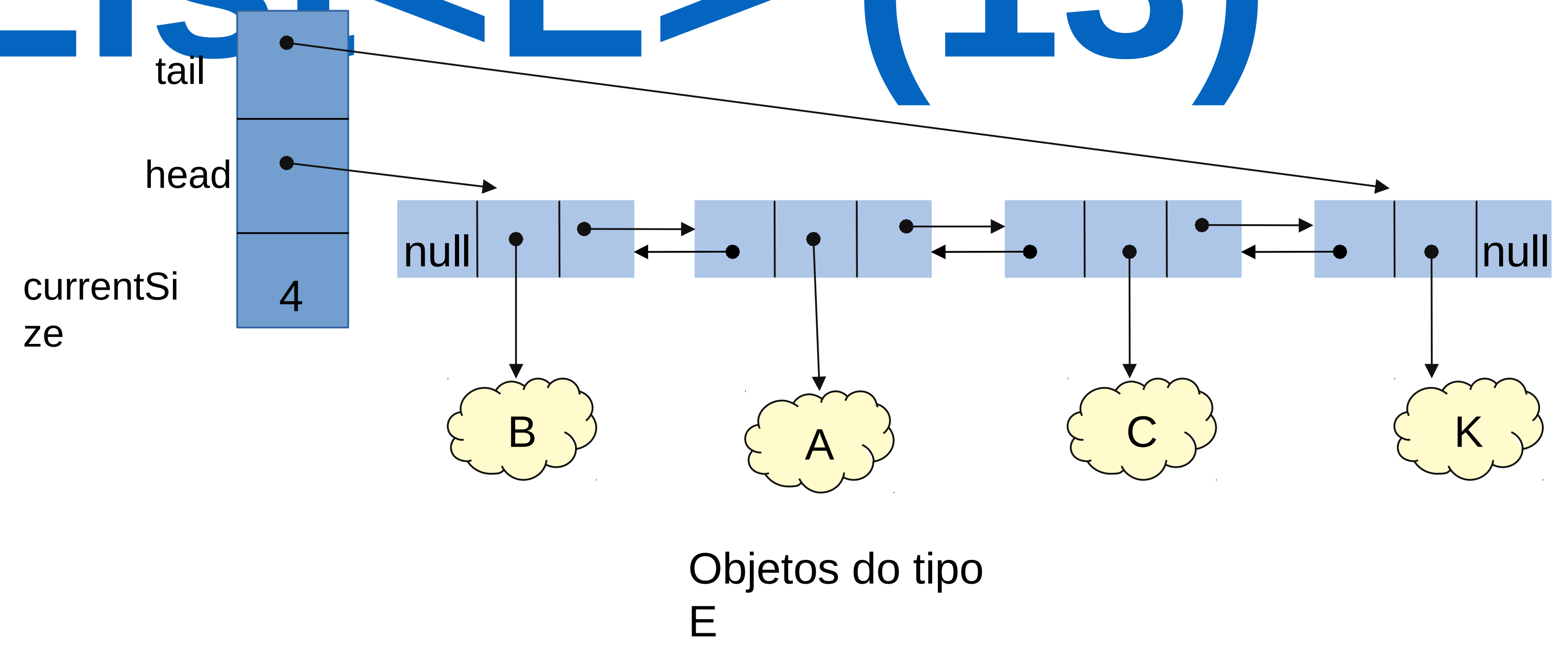


```
/**
 * Inserts the element at the last position in the list.
 * @param element - Element to be inserted
 */
public void addLast( E element ) {
   //TODO: Left as an exercise.

}
```

addLast(“Z”)

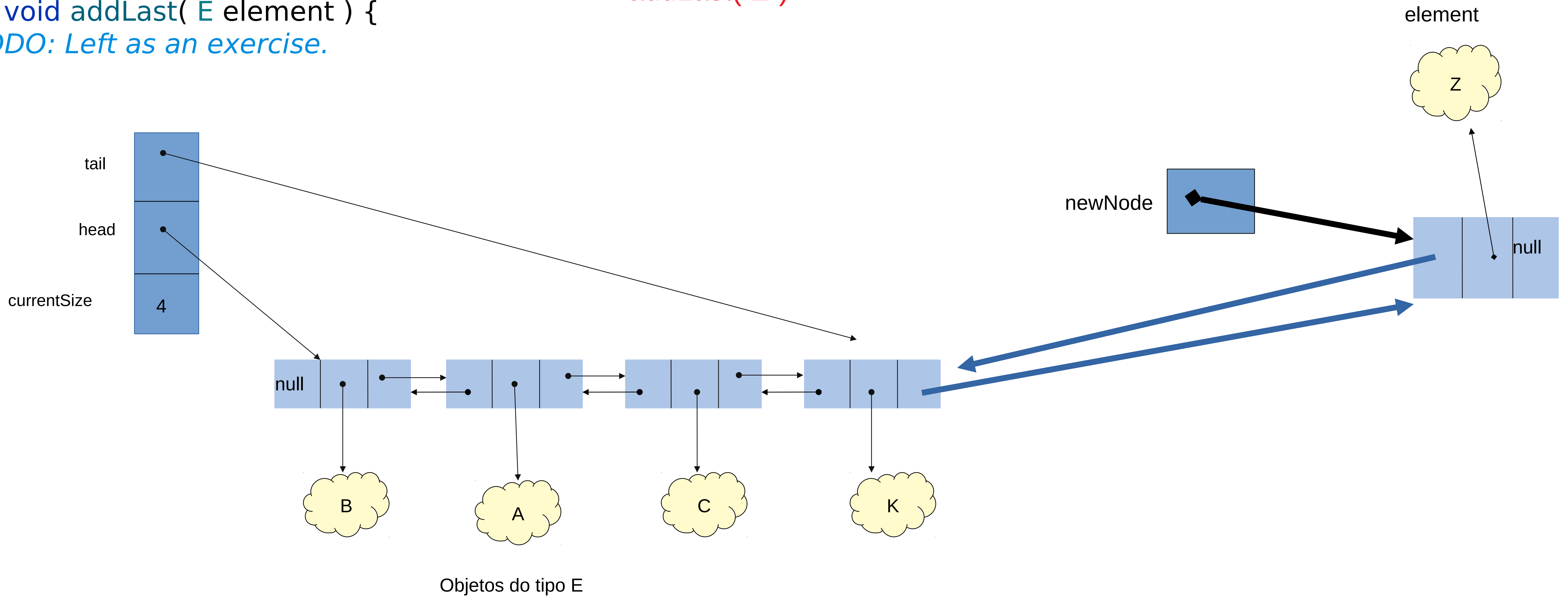


Classe DoublyLinkedList<E> (13)

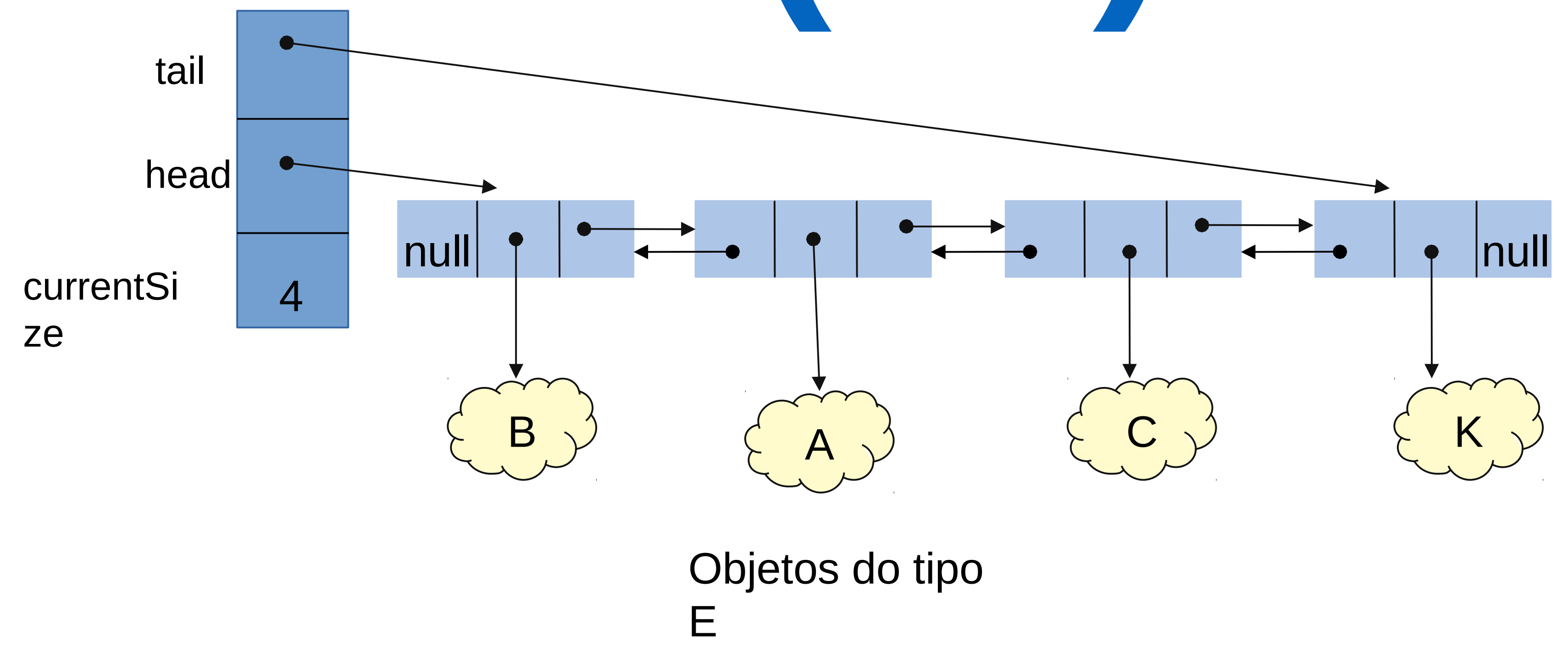


```
/**
 * Inserts the element at the last position in the list.
 * @param element - Element to be inserted
 */
public void addLast( E element ) {
   //TODO: Left as an exercise.

}
```

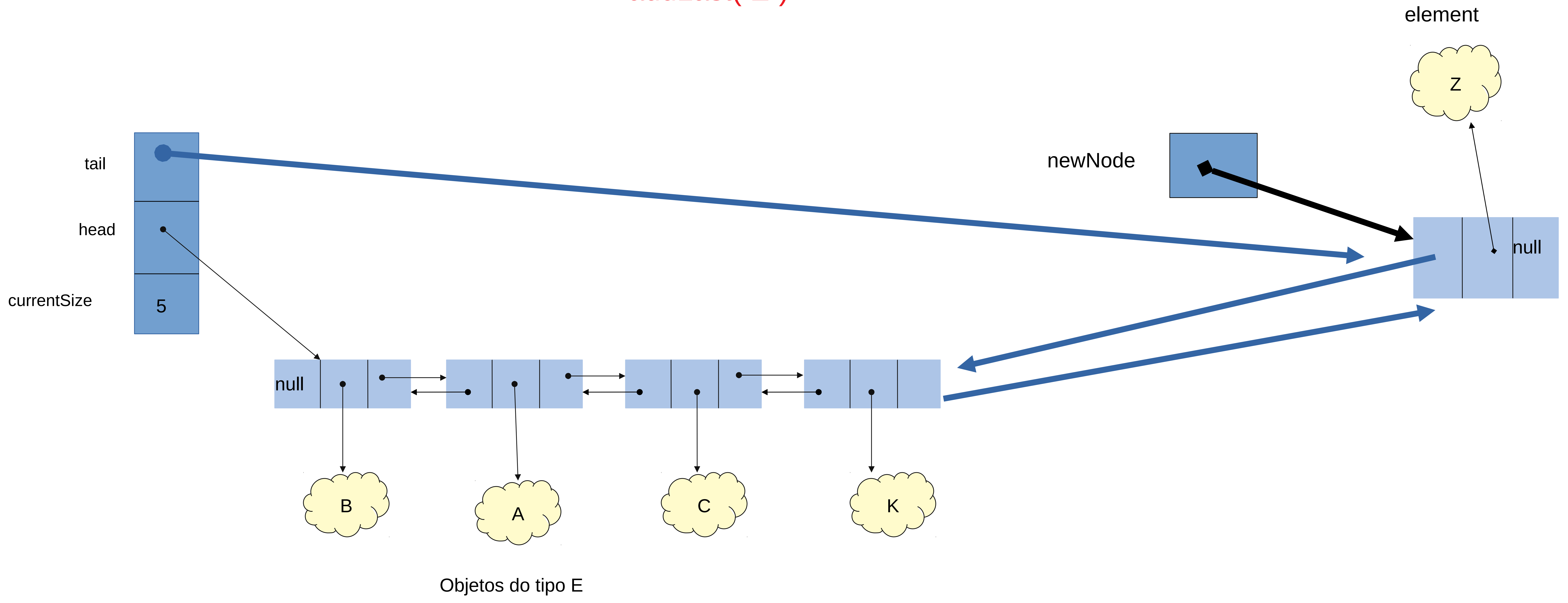
addLast("Z")



Classe DoublyLinkedList<E> (14)

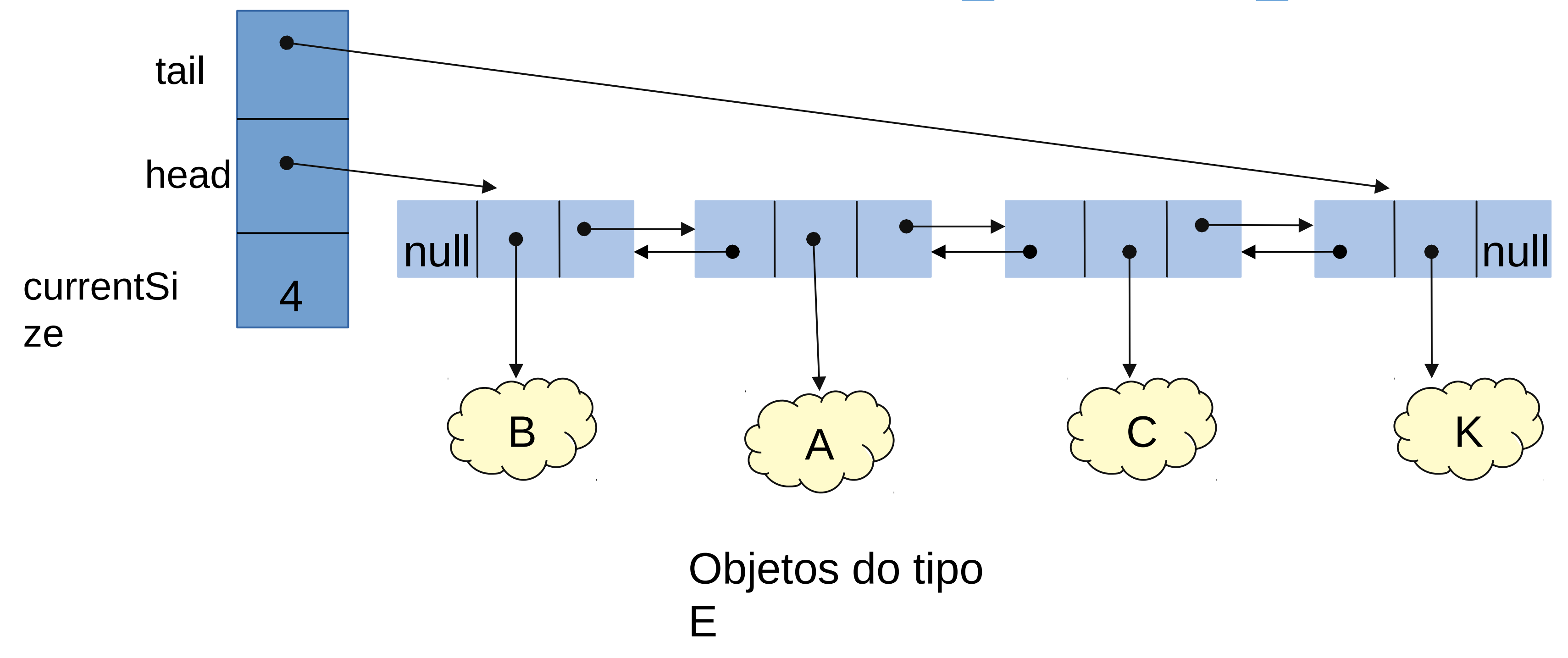


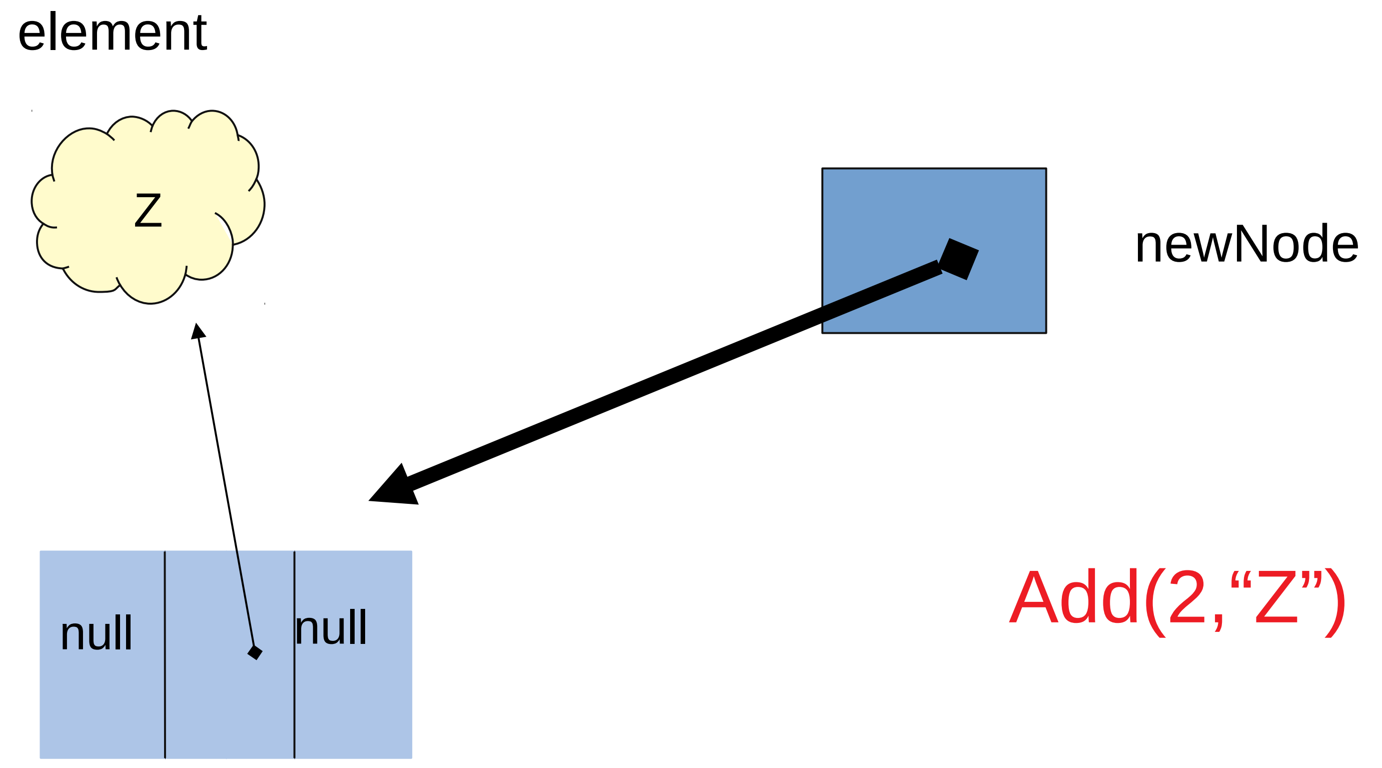
addLast("Z")



Classe DoublyLinkedList<E> (15)

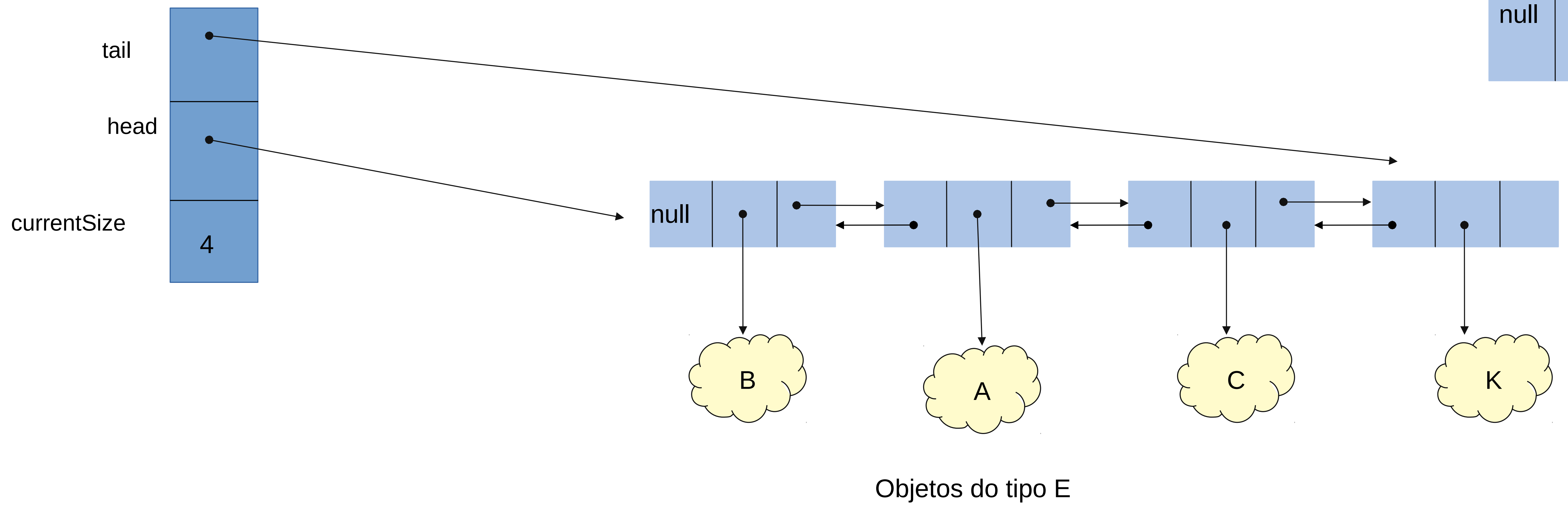
```
/**
 * Inserts the specified element at the specified position in the list.
 * Range of valid positions: 0, ..., size().
 * If the specified position is 0, add corresponds to addFirst.
 * If the specified position is size(), add corresponds to addLast.
 * @param position - position where to insert element
 * @param element - element to be inserted
 * @throws InvalidPositionException - if position is not valid in the list
 */
void add( int position, E element );
```





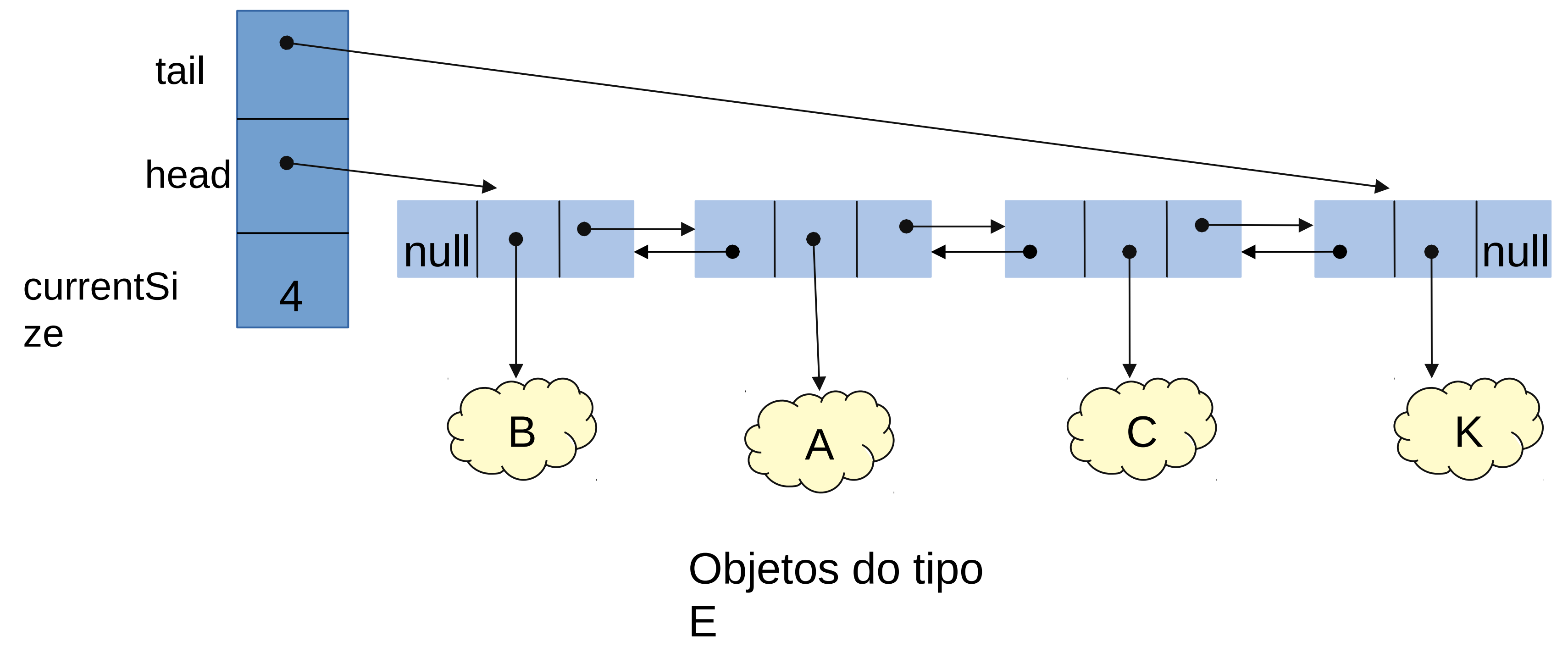
Add(2,“Z”)

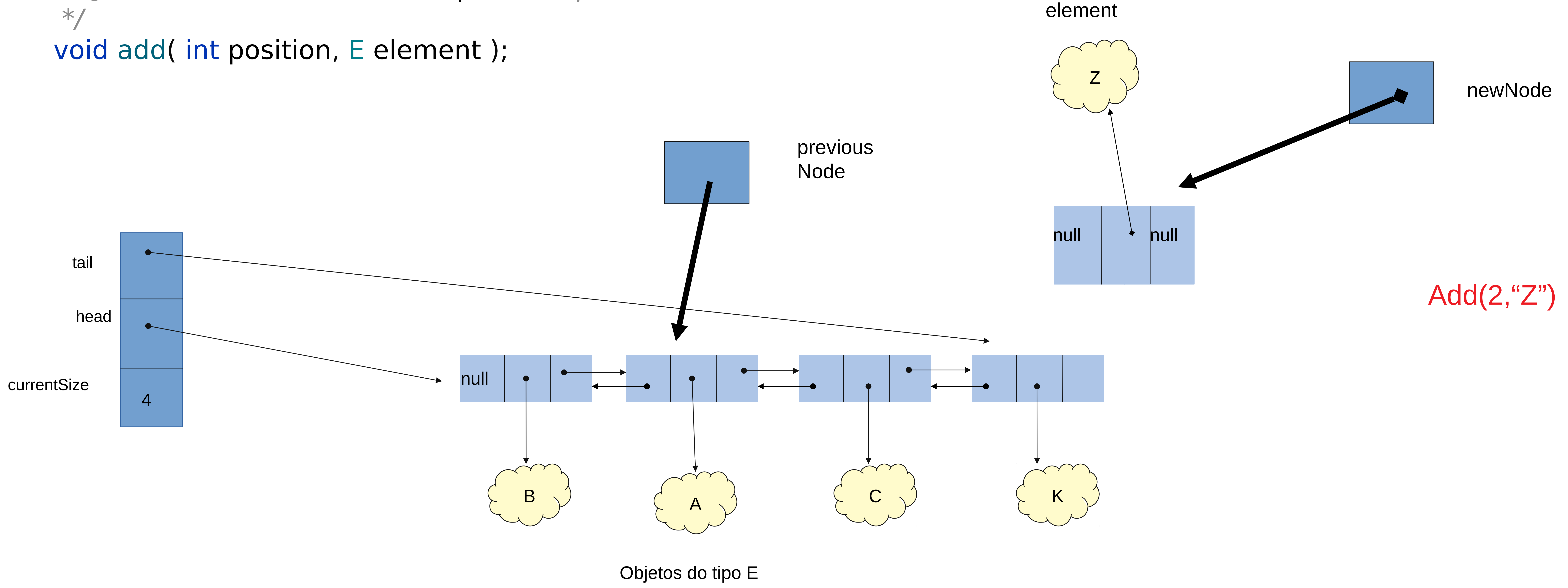
Lista com 4 elementos



Classe DoublyLinkedList<E> (16)

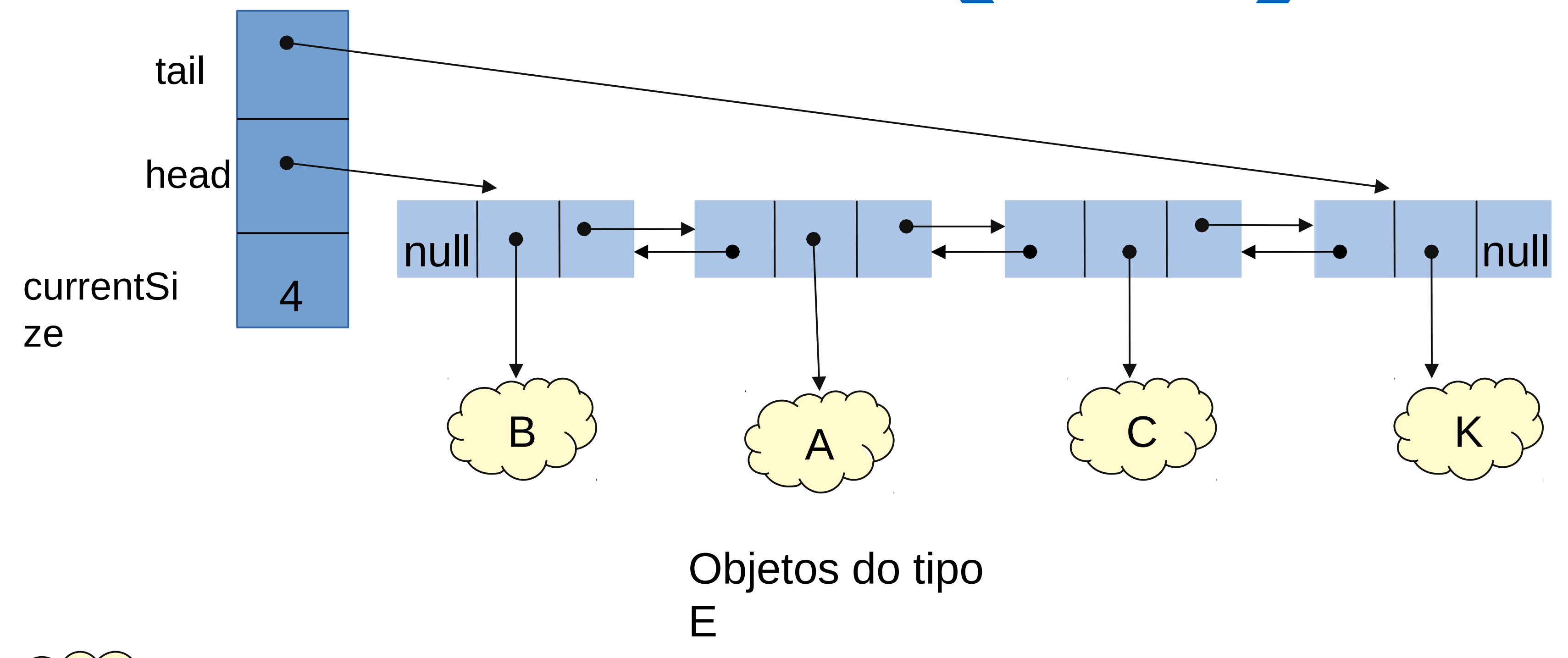
```
/**
 * Inserts the specified element at the specified position in the list.
 * Range of valid positions: 0, ..., size().
 * If the specified position is 0, add corresponds to addFirst.
 * If the specified position is size(), add corresponds to addLast.
 * @param position - position where to insert element
 * @param element - element to be inserted
 * @throws InvalidPositionException - if position is not valid in the list
 */
void add( int position, E element );
```

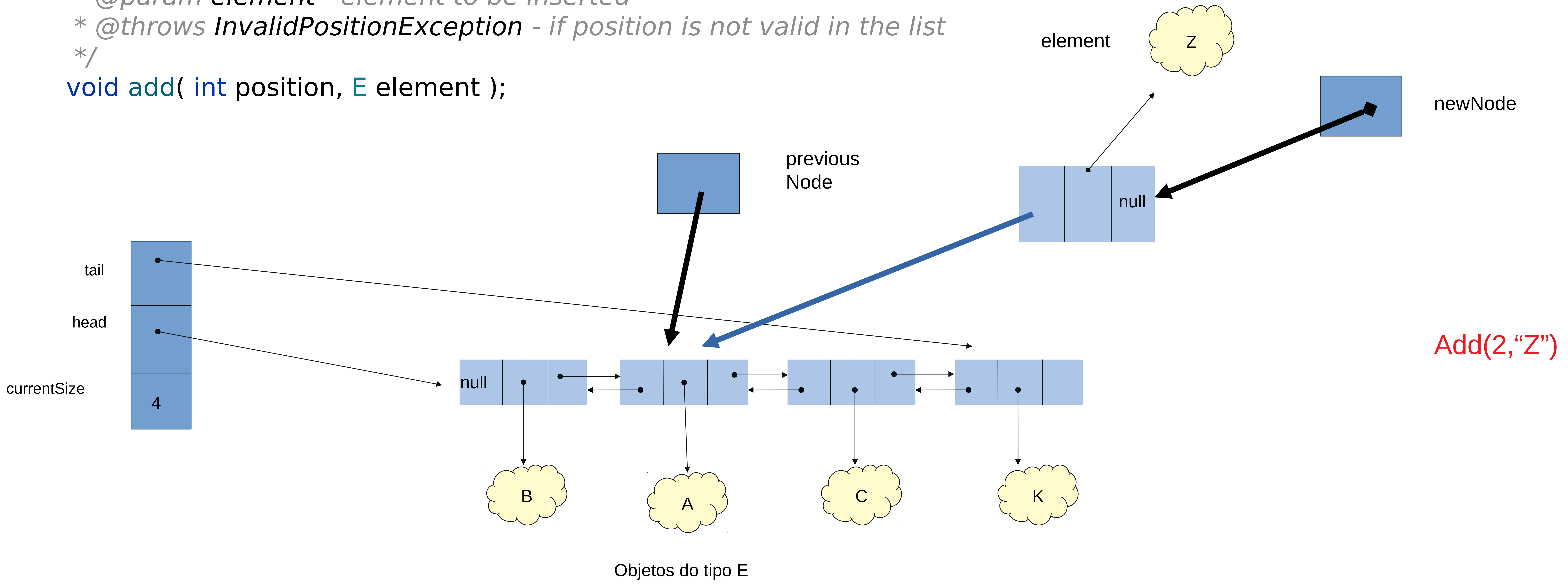




Classe DoublyLinkedList<E> (17)

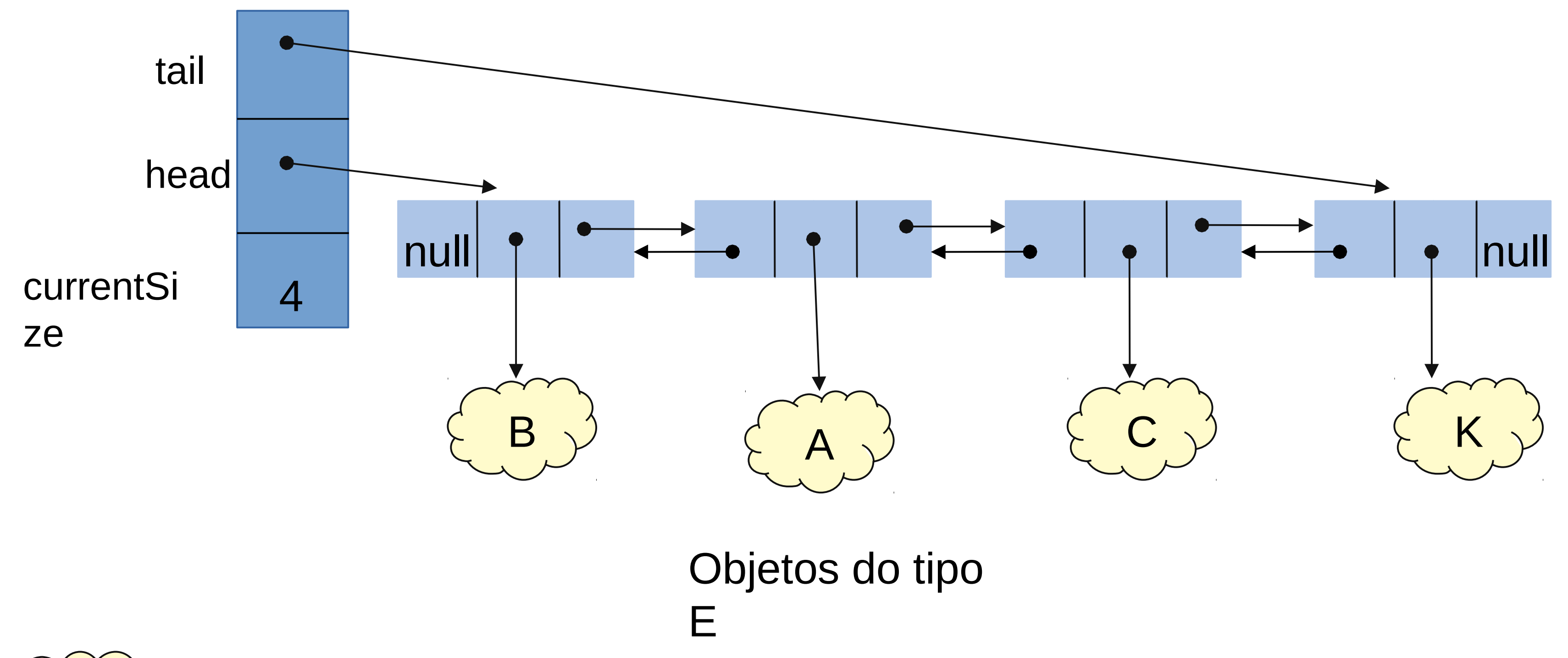
```
/**
 * Inserts the specified element at the specified position in the list.
 * Range of valid positions: 0, ..., size().
 * If the specified position is 0, add corresponds to addFirst.
 * If the specified position is size(), add corresponds to addLast.
 * @param position - position where to insert element
 * @param element - element to be inserted
 * @throws InvalidPositionException - if position is not valid in the list
 */
void add( int position, E element );
```

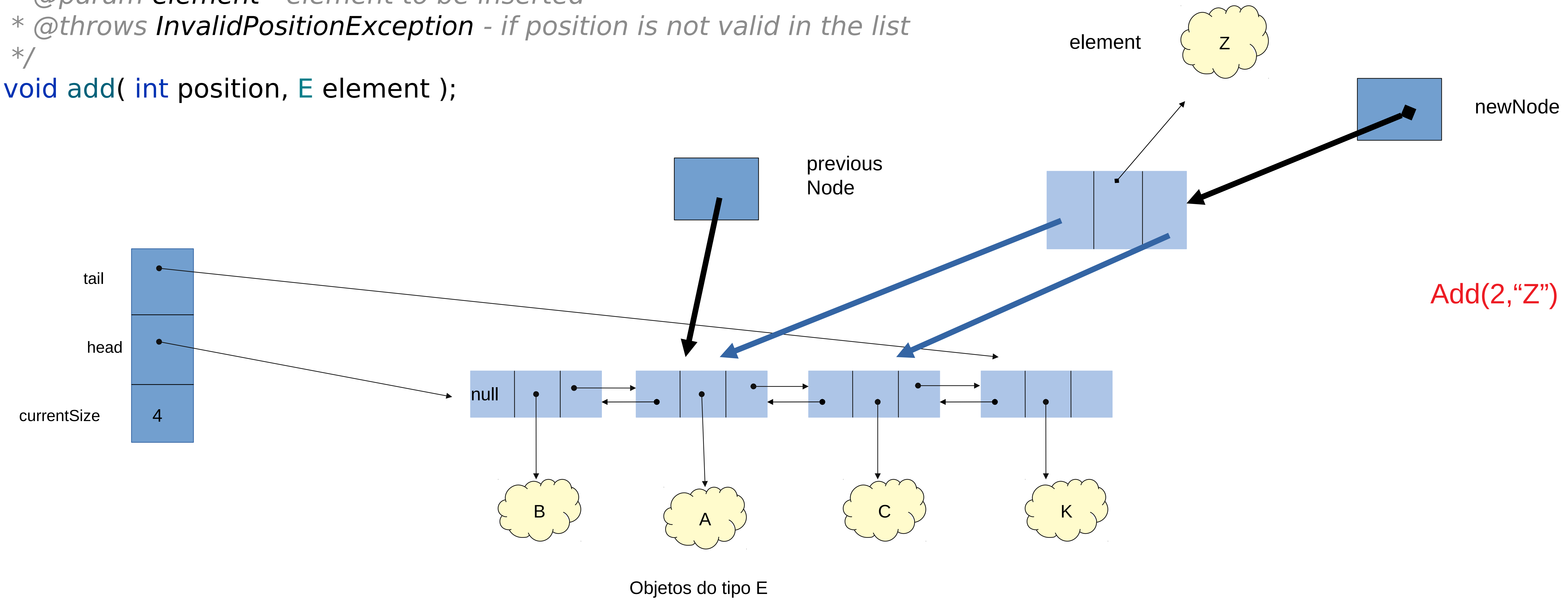




Classe DoublyLinkedList<E> (18)

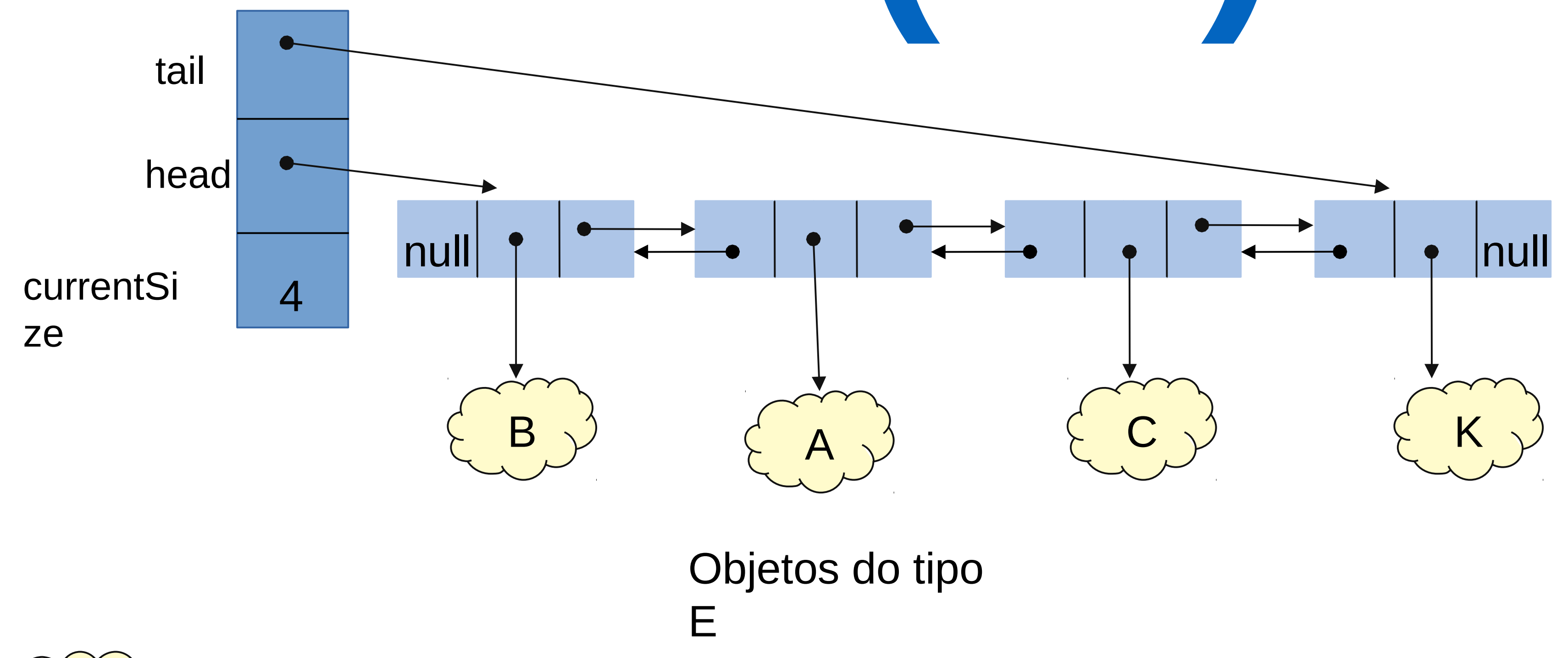
```
/**
 * Inserts the specified element at the specified position in the list.
 * Range of valid positions: 0, ..., size().
 * If the specified position is 0, add corresponds to addFirst.
 * If the specified position is size(), add corresponds to addLast.
 * @param position - position where to insert element
 * @param element - element to be inserted
 * @throws InvalidPositionException - if position is not valid in the list
 */
void add( int position, E element );
```

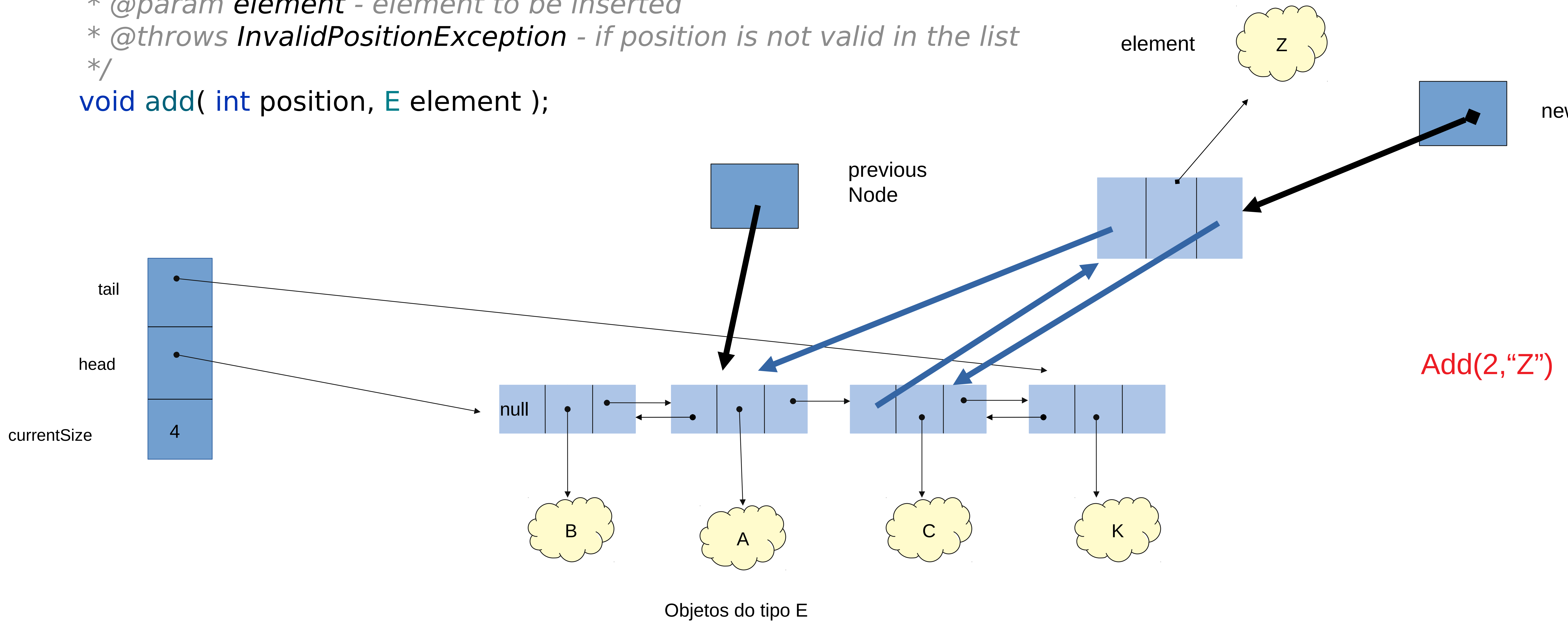




Classe DoublyLinkedList<E> (19)

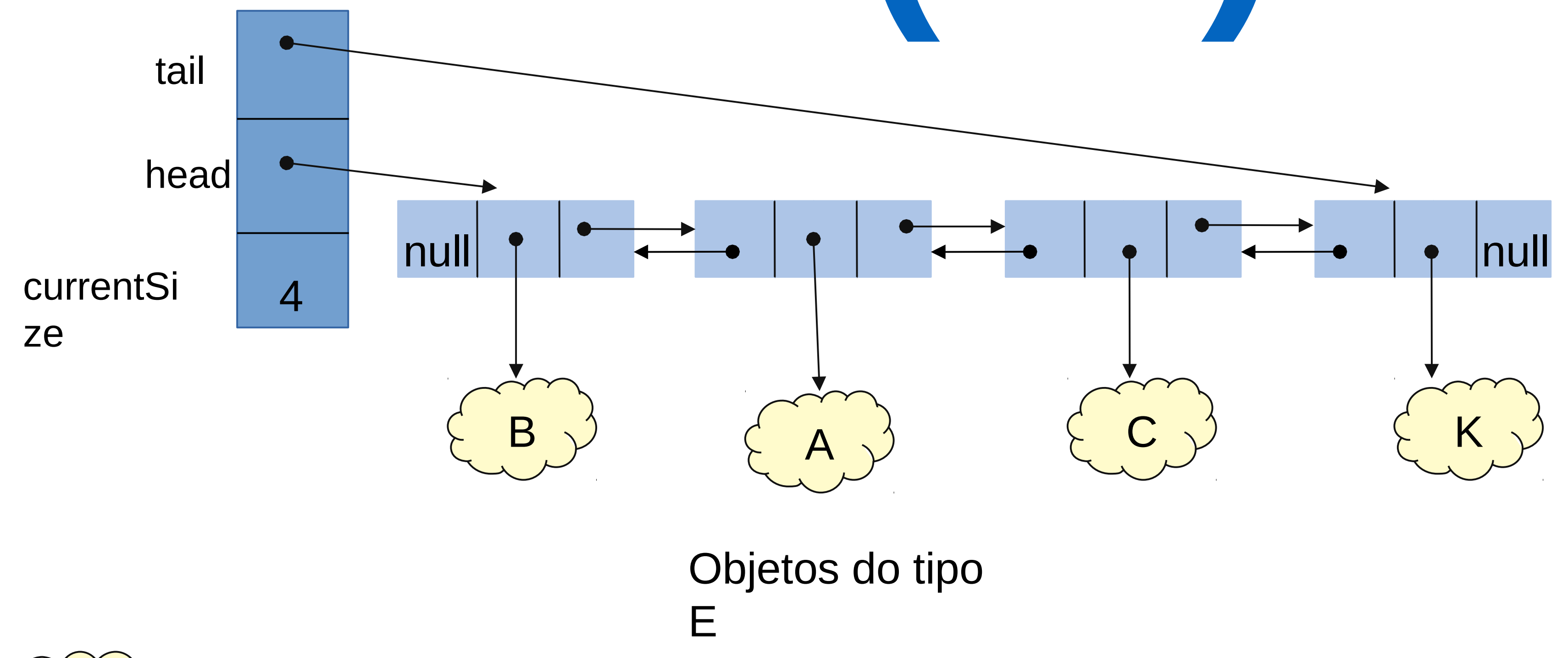
```
/**
 * Inserts the specified element at the specified position in the list.
 * Range of valid positions: 0, ..., size().
 * If the specified position is 0, add corresponds to addFirst.
 * If the specified position is size(), add corresponds to addLast.
 * @param position - position where to insert element
 * @param element - element to be inserted
 * @throws InvalidPositionException - if position is not valid in the list
 */
void add( int position, E element );
```

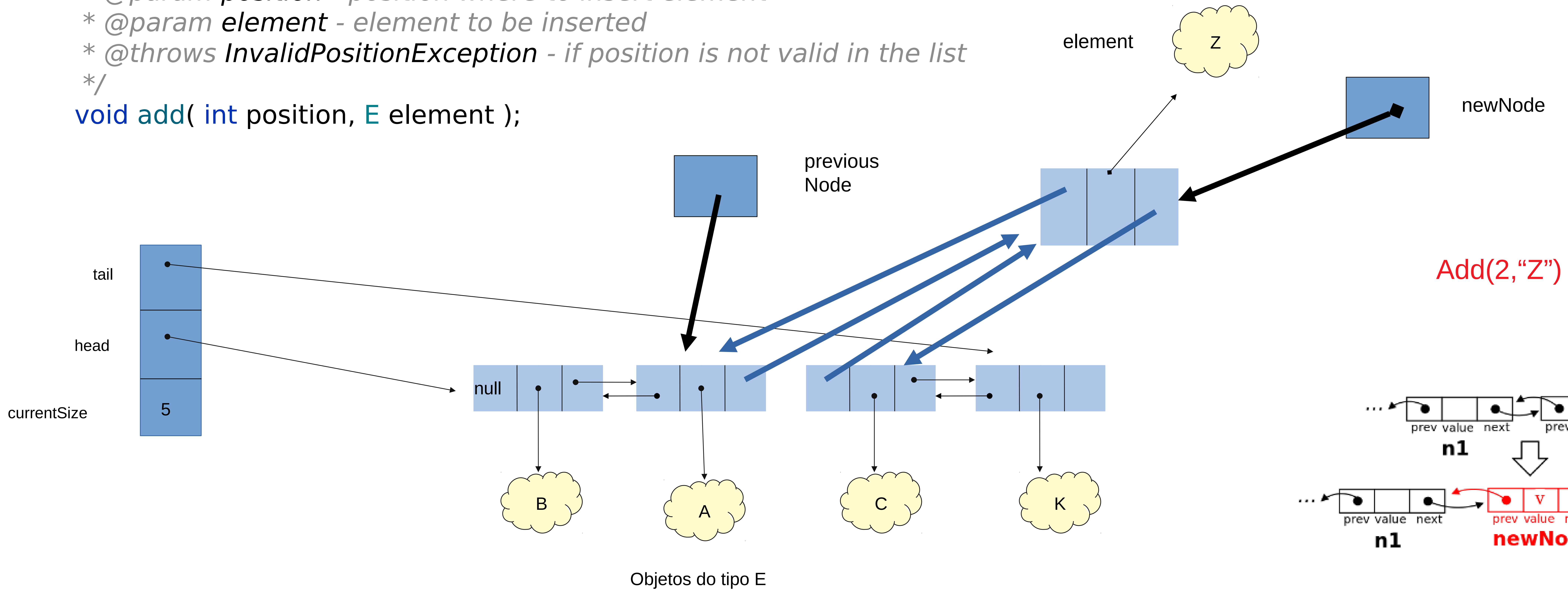


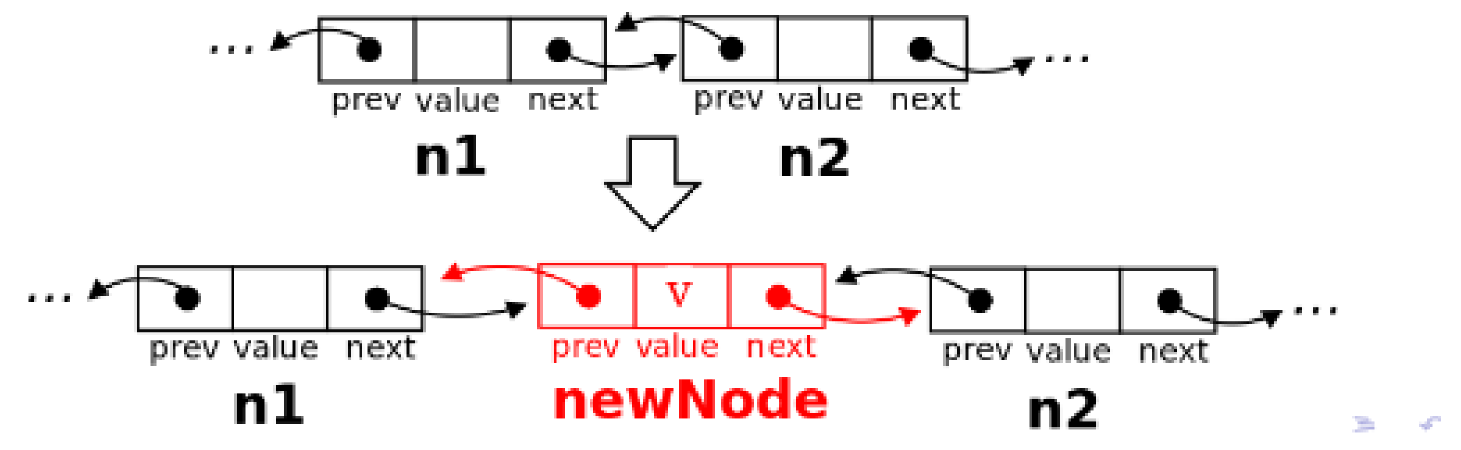


Classe DoublyLinkedList<E> (20)

```
/**
 * Inserts the specified element at the specified position in the list.
 * Range of valid positions: 0, ..., size().
 * If the specified position is 0, add corresponds to addFirst.
 * If the specified position is size(), add corresponds to addLast.
 * @param position - position where to insert element
 * @param element - element to be inserted
 * @throws InvalidPositionException - if position is not valid in the list
 */
void add( int position, E element );
```

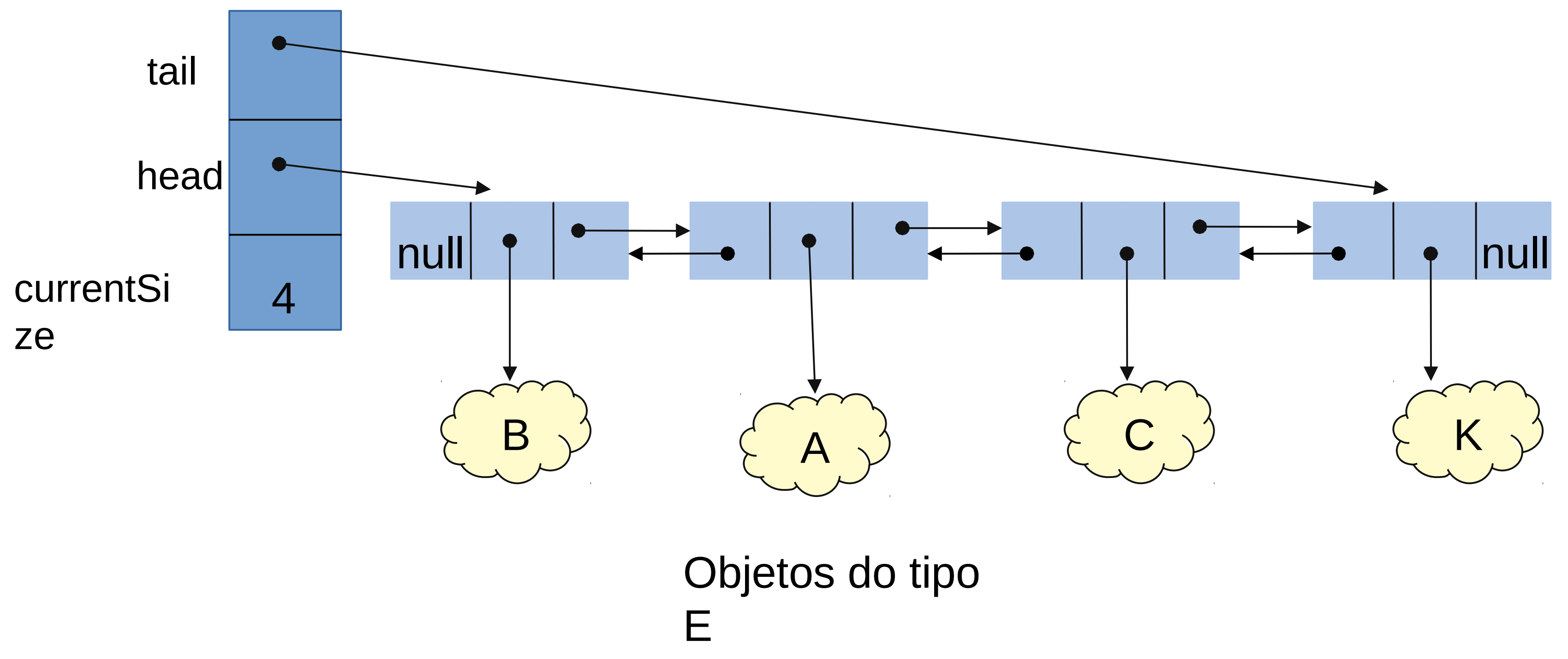






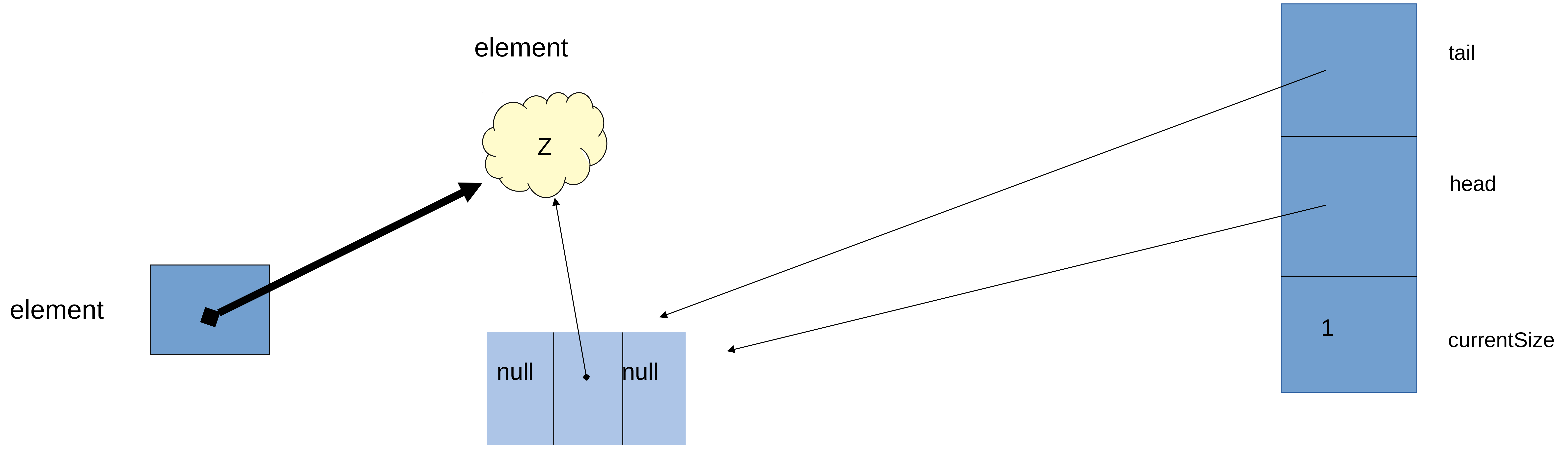
Classe DoublyLinkedList<E> (21)

```
/**
 * Removes and returns the element at the first position in the list.
 * @return element removed from the first position of the list
 * @throws NoSuchElementException - if size() == 0
 */
public E removeFirst( ) {
   //TODO: Left as an exercise.


}
```

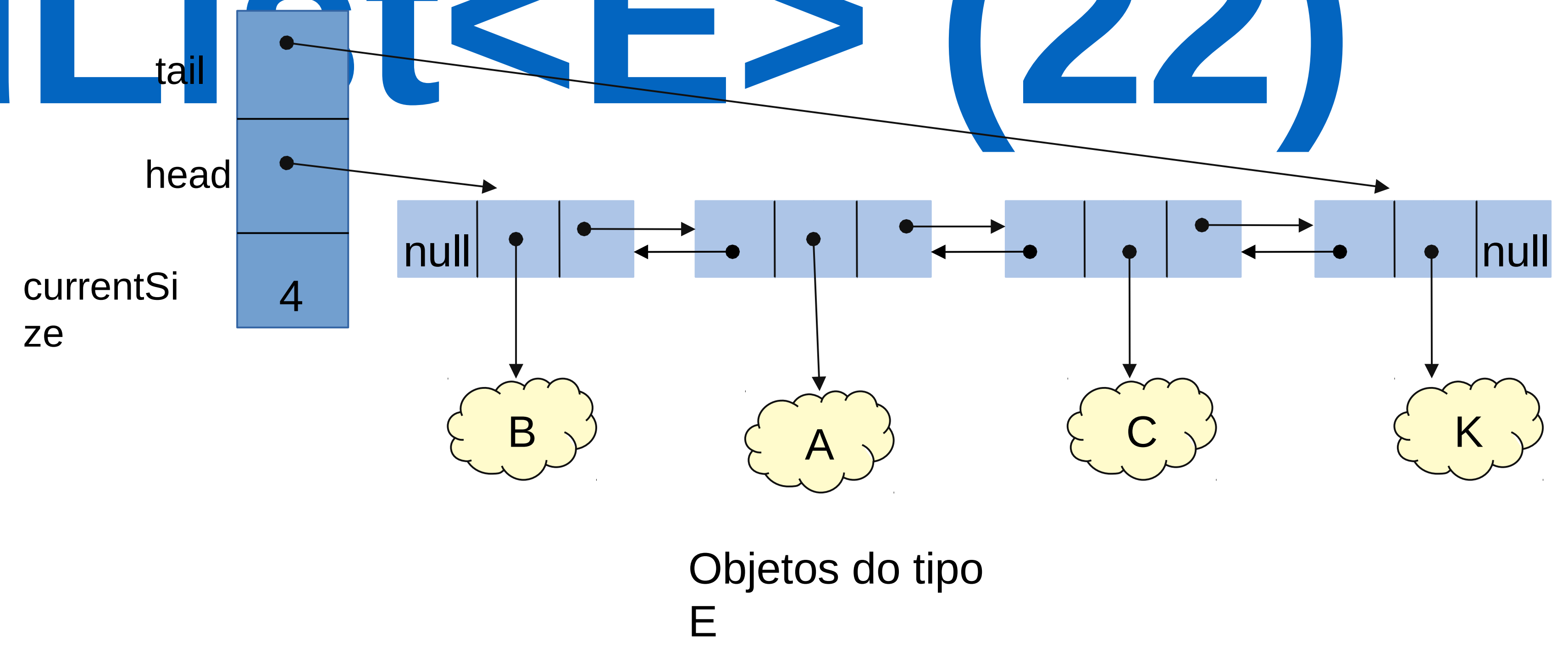


removeFirst()

Lista com um elemento

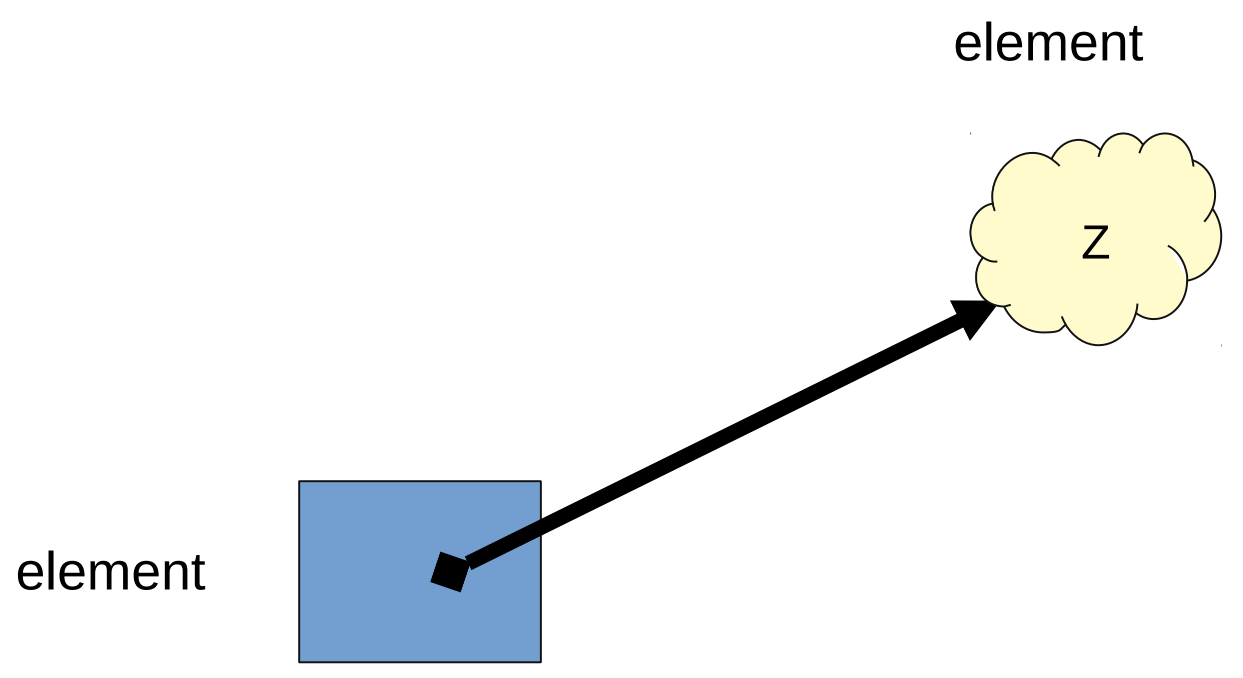


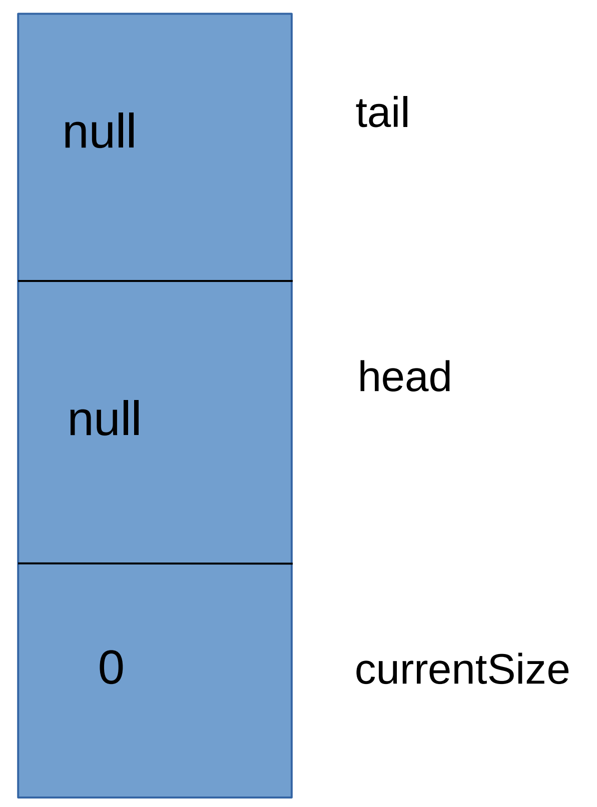
Classe DoublyLinkedList<E> (22)



```
/**
 * Removes and returns the element at the first position in the list.
 * @return element removed from the first position of the list
 * @throws NoSuchElementException - if size() == 0
 */
public E removeFirst( ) {
   //TODO: Left as an exercise.


}
```

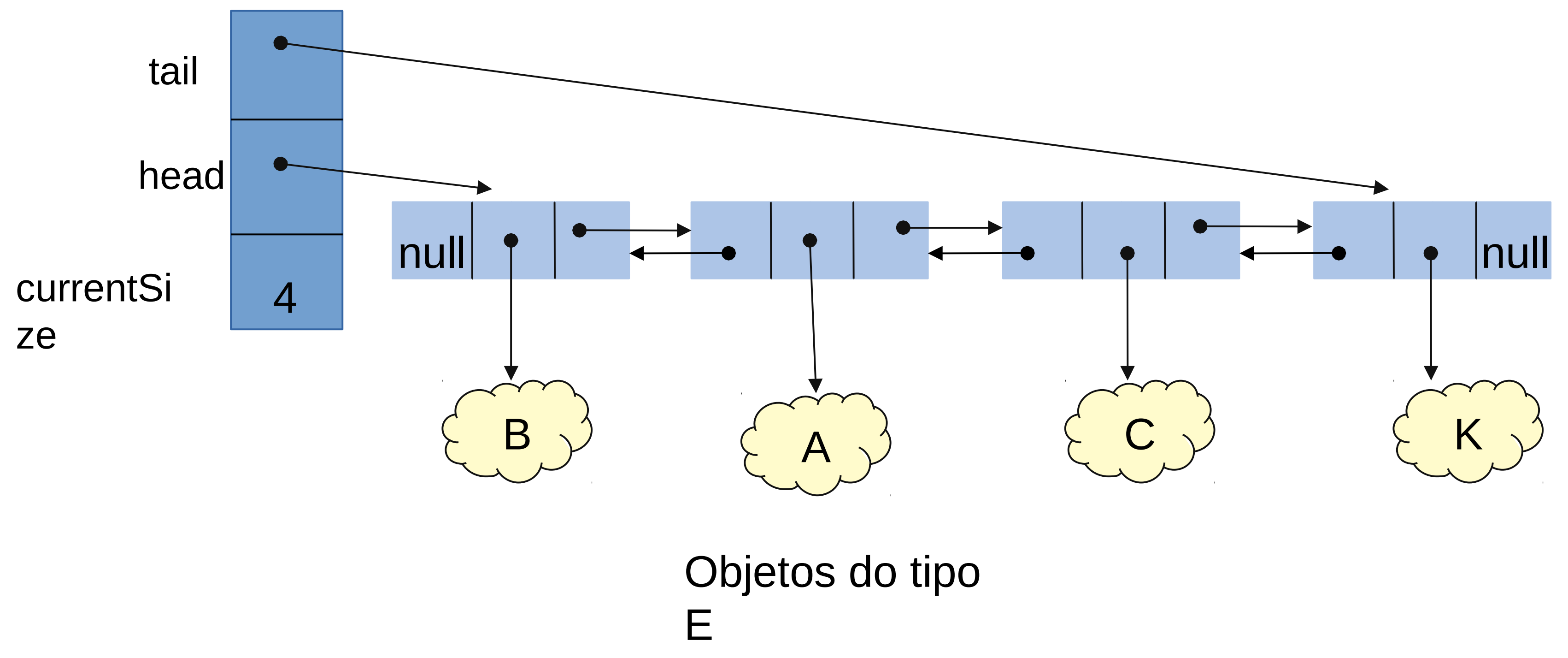
removeFirst()





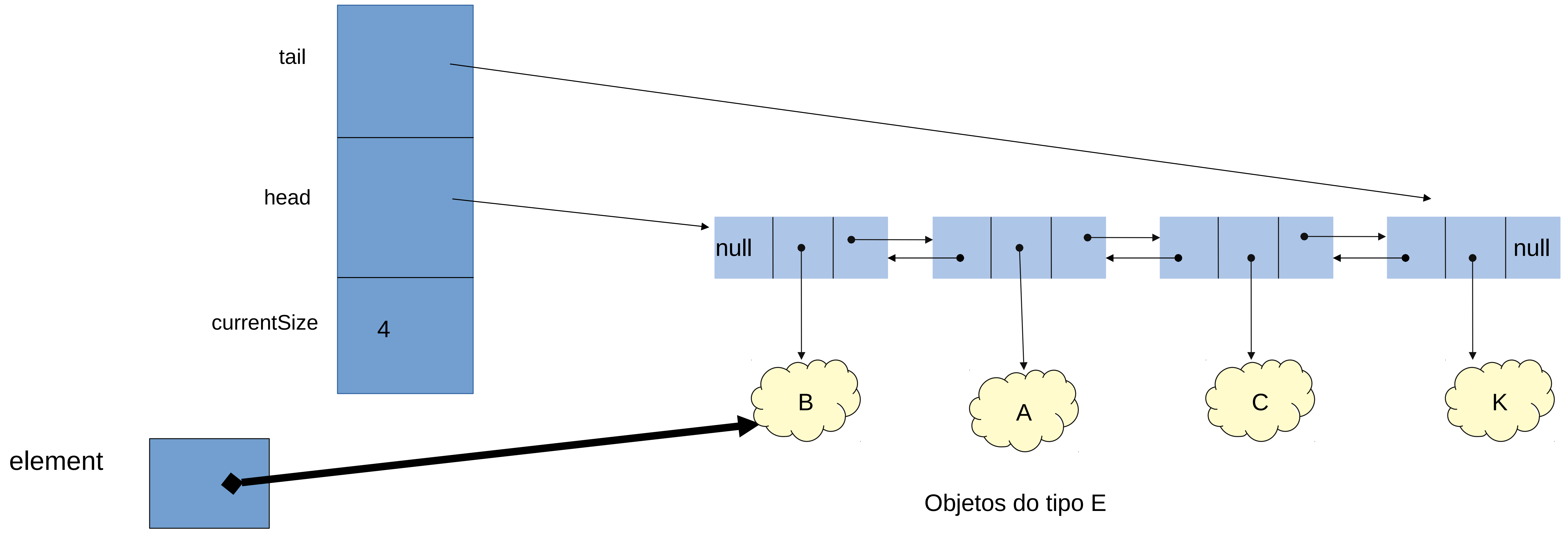
Classe DoublyLinkedList<E> (23)

```
/**
 * Removes and returns the element at the first position in the list.
 * @return element removed from the first position of the list
 * @throws NoSuchElementException - if size() == 0
 */
public E removeFirst( ) {
   //TODO: Left as an exercise.


}
```

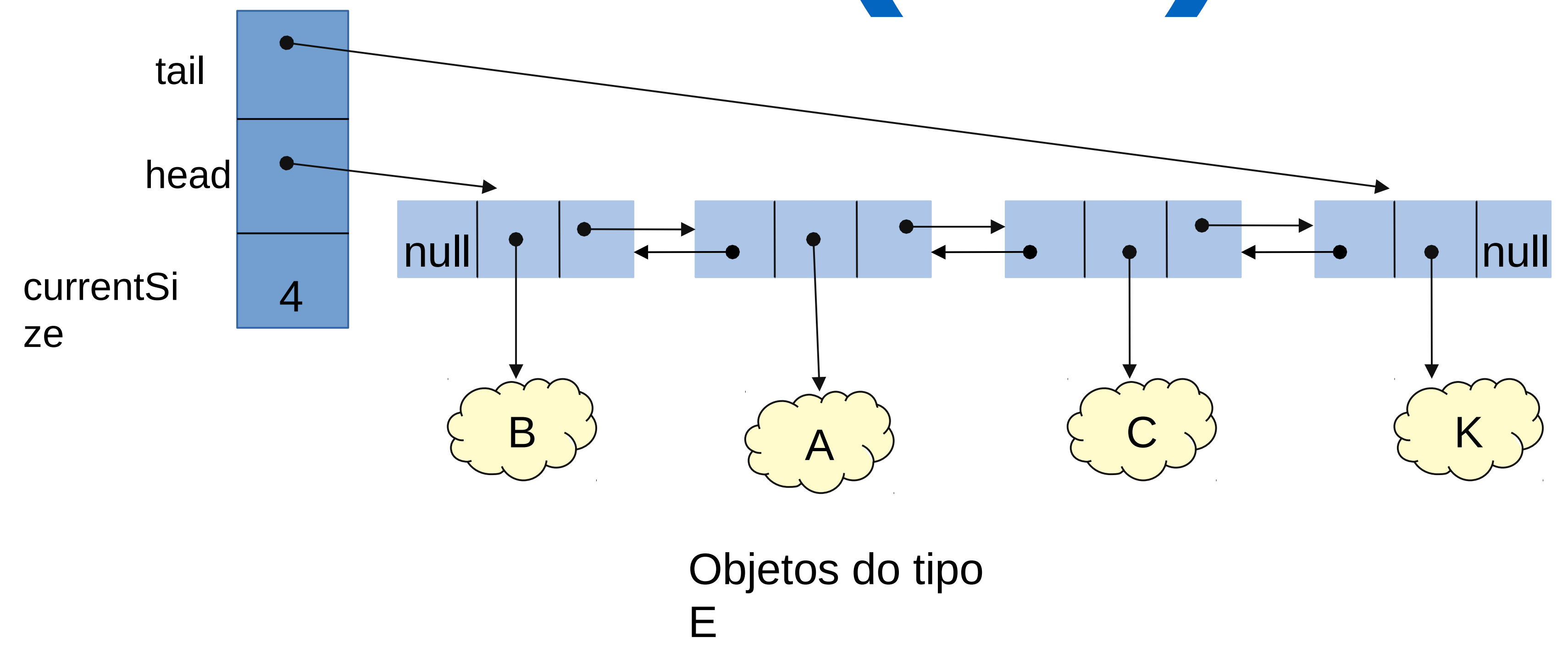


removeFirst()

Lista com vários elementos

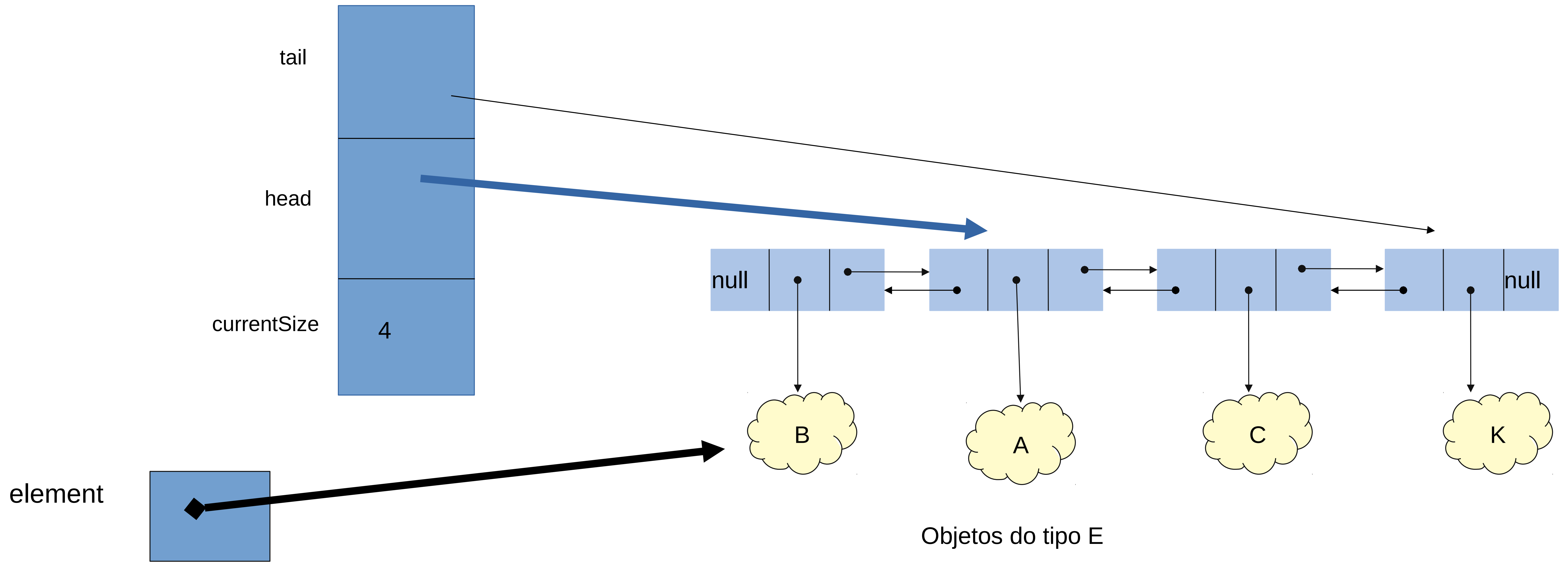


Classe DoublyLinkedList<E> (24)

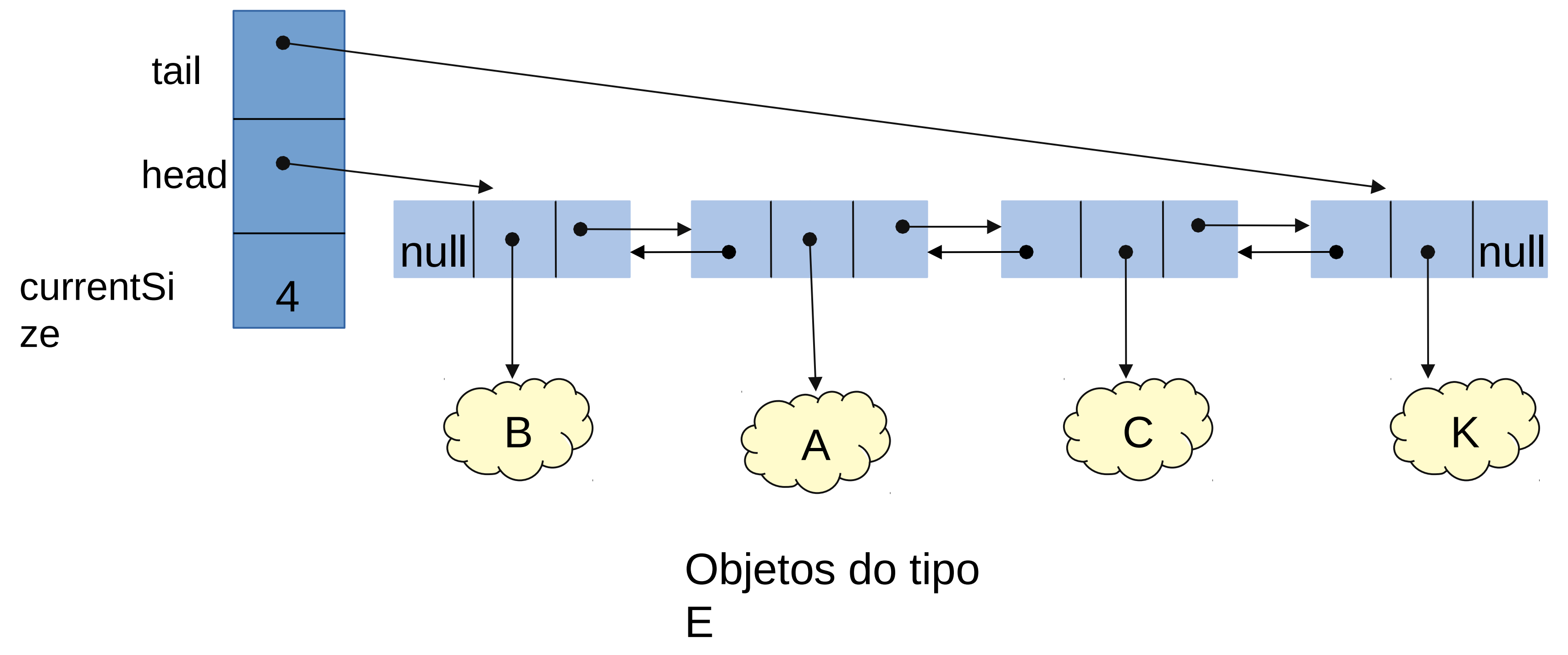


```
/**
 * Removes and returns the element at the first position in the list.
 * @return element removed from the first position of the list
 * @throws NoSuchElementException - if size() == 0
 */
public E removeFirst( ) {
   //TODO: Left as an exercise.


}
```

removeFirst()

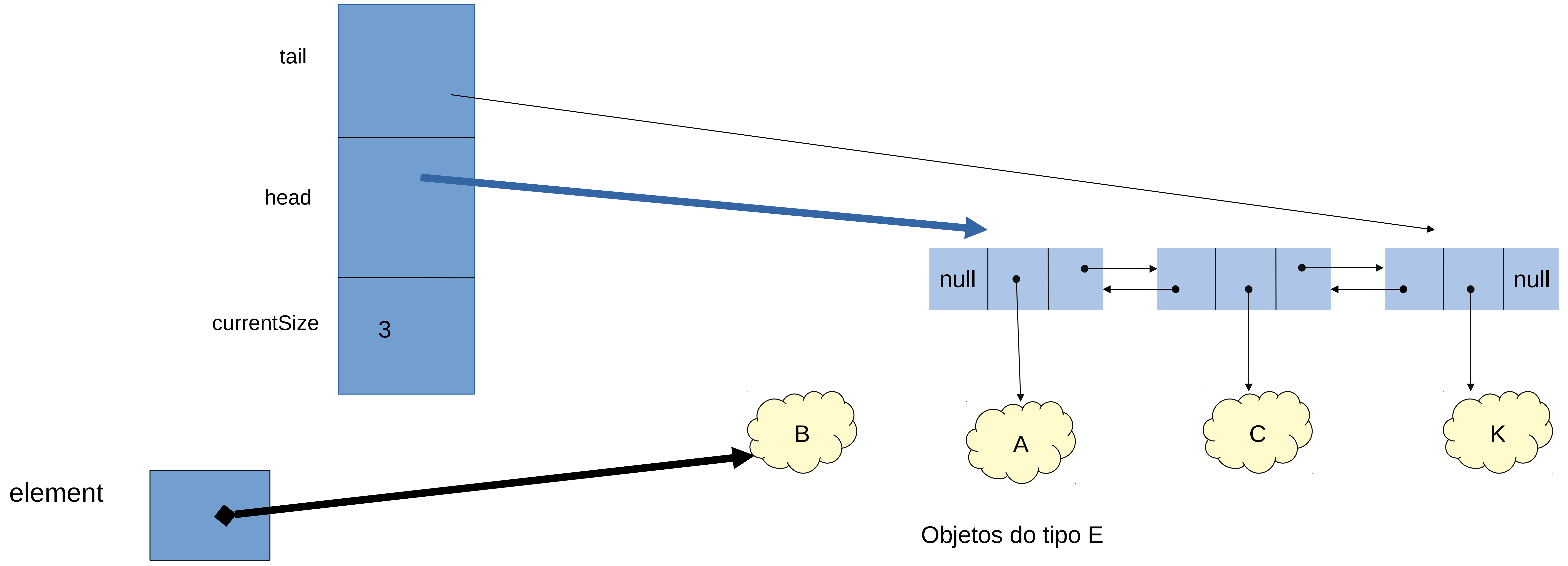


Classe DoublyLinkedList<E> (25)

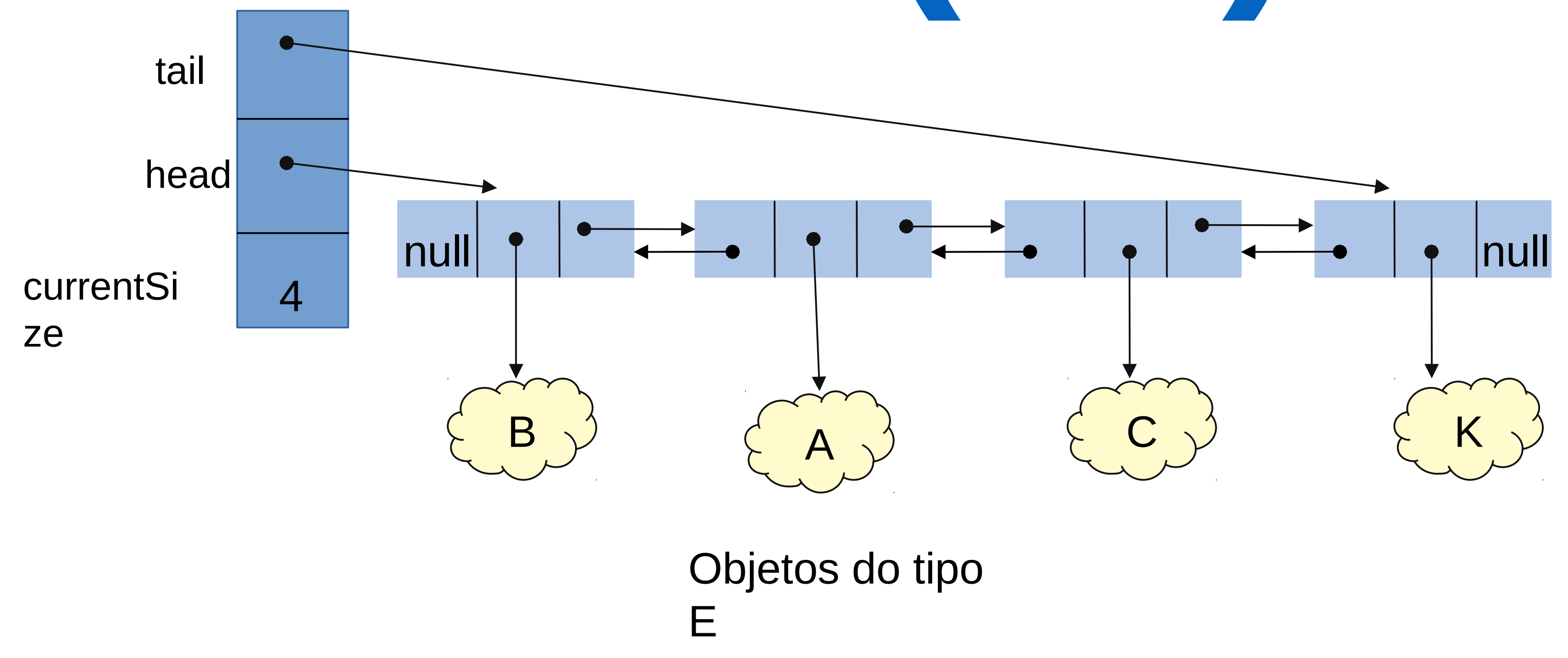


```
/**
 * Removes and returns the element at the first position in the list.
 * @return element removed from the first position of the list
 * @throws NoSuchElementException - if size() == 0
 */
public E removeFirst( ) {
   //TODO: Left as an exercise.


}
```

removeFirst()



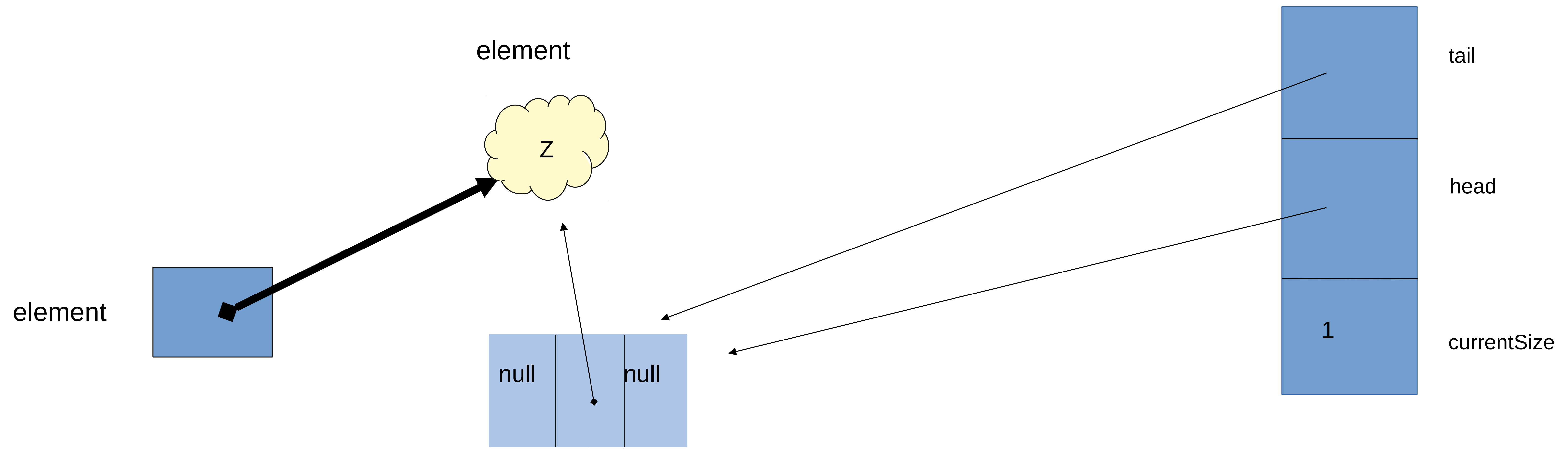
Classe DoublyLinkedList<E> (26)



```
/**
 * Removes and returns the element at the last position in the list.
 * @return element removed from the last position of the list
 * @throws NoSuchElementException - if size() == 0
 */
public E removeLast( ) {
   //TODO: Left as an exercise.


}
```

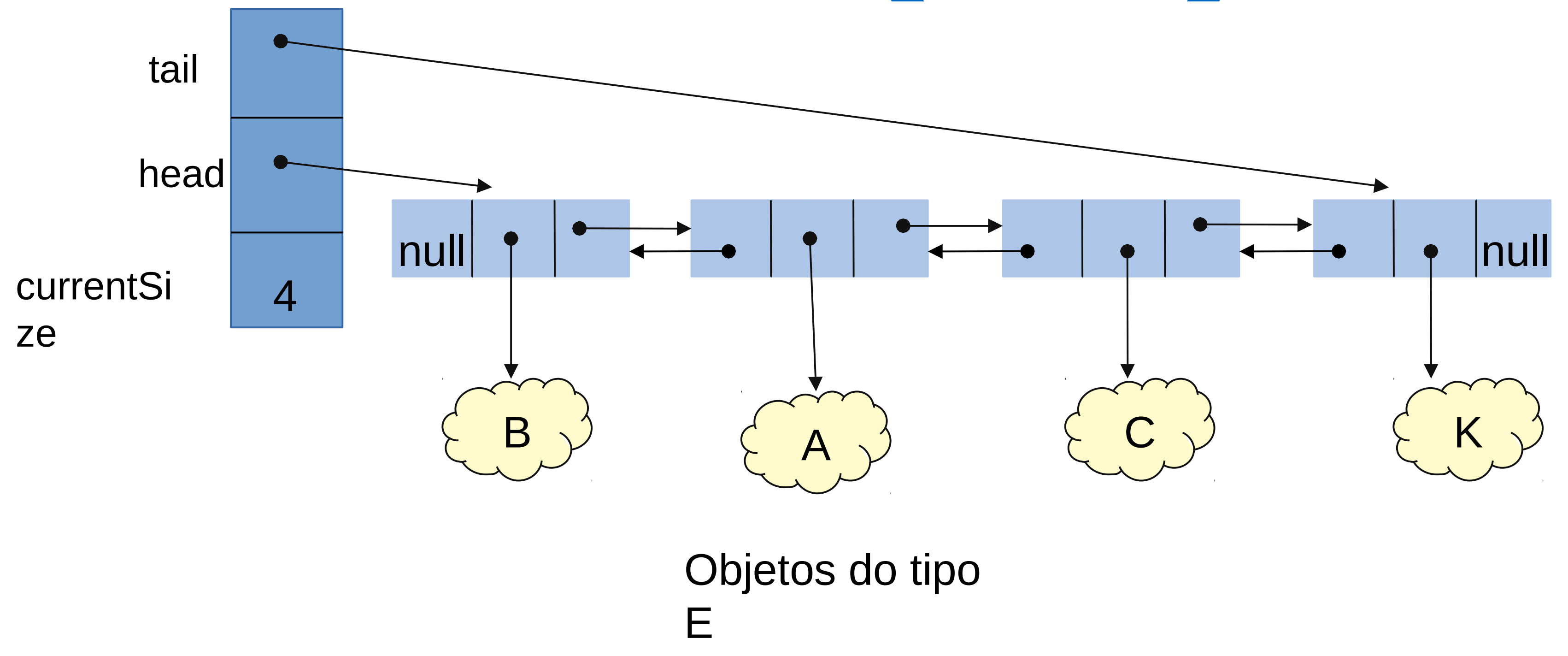
removeLast()

Lista com um elemento

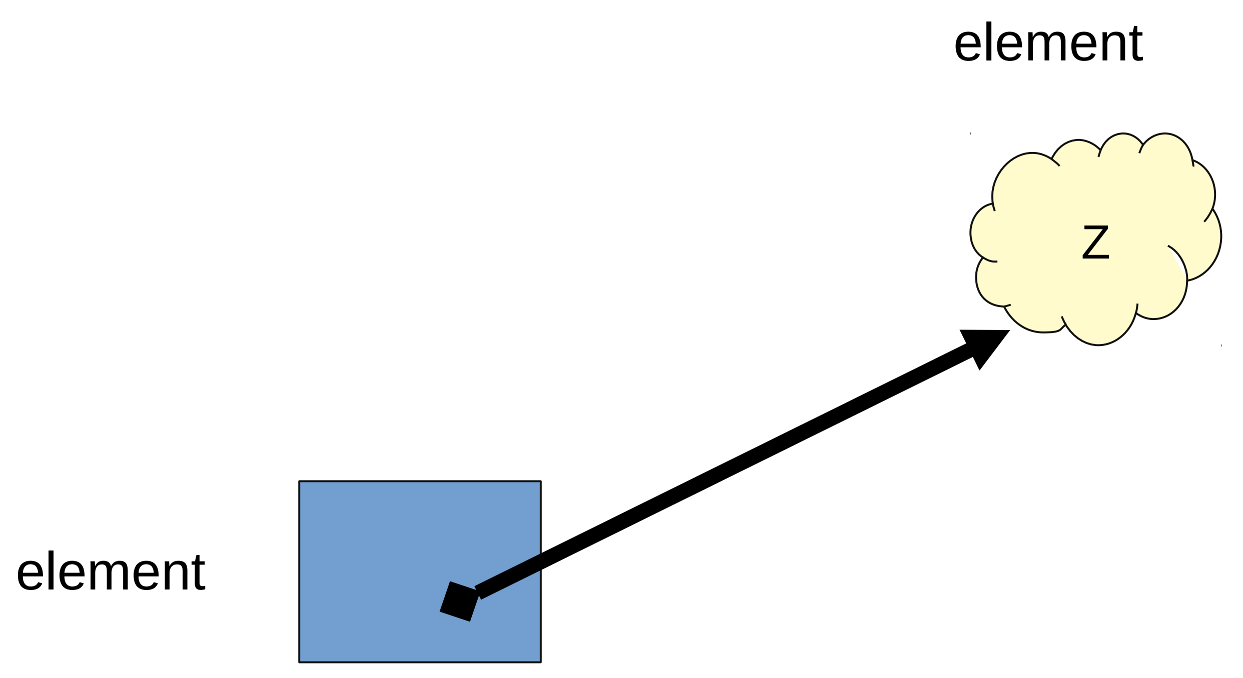


Classe DoublyLinkedList<E> (27)

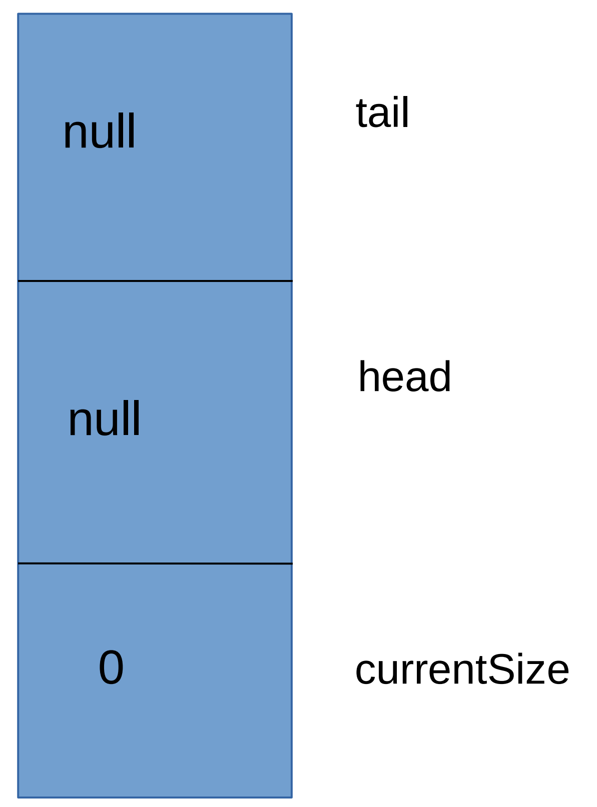
```
/**
 * Removes and returns the element at the last position in the list.
 * @return element removed from the last position of the list
 * @throws NoSuchElementException - if size() == 0
 */
public E removeLast( ) {
   //TODO: Left as an exercise.

}
```



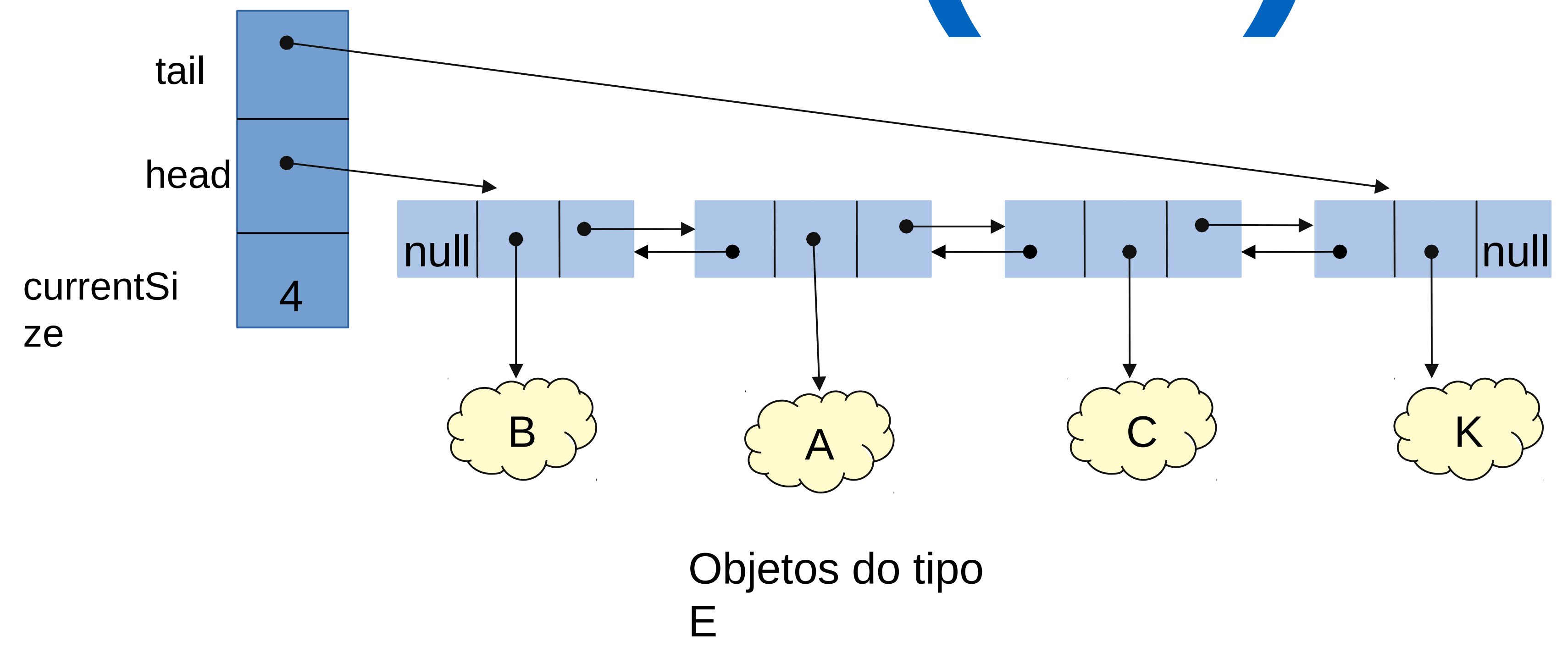
removeFirst()



Lista com um elemento

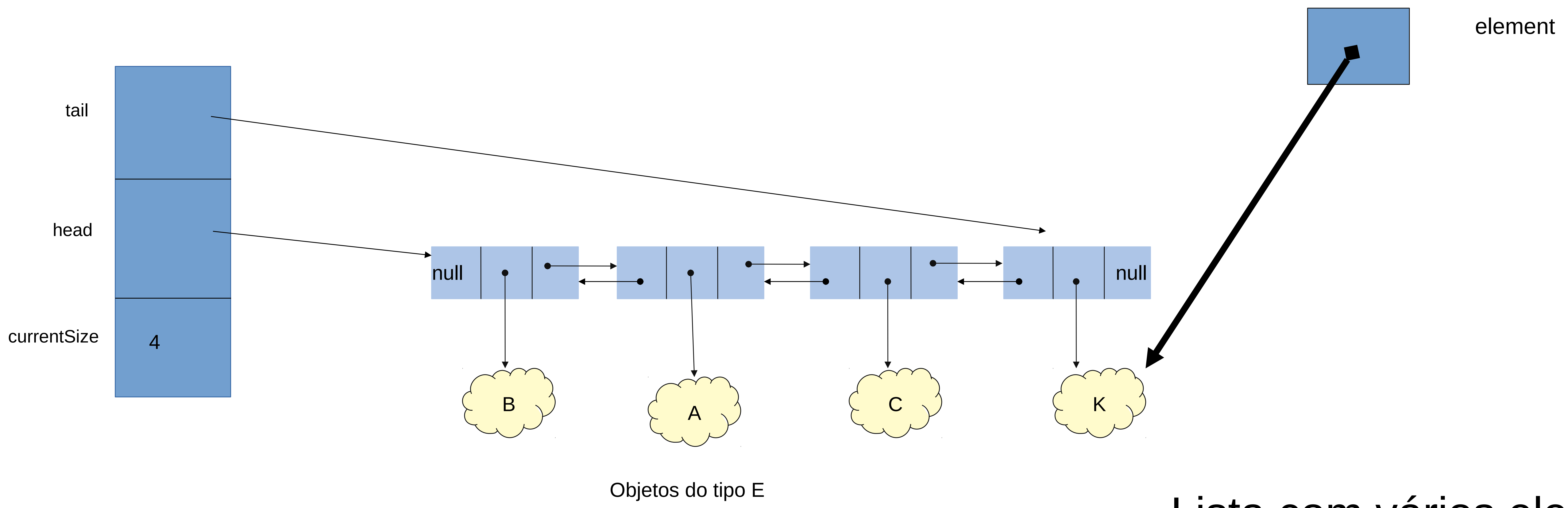


Classe DoublyLinkedList<E> (28)



```
/**
 * Removes and returns the element at the last position in the list.
 * @return element removed from the last position of the list
 * @throws NoSuchElementException - if size() == 0
 */
public E removeLast( ) {
   //TODO: Left as an exercise.


}
```

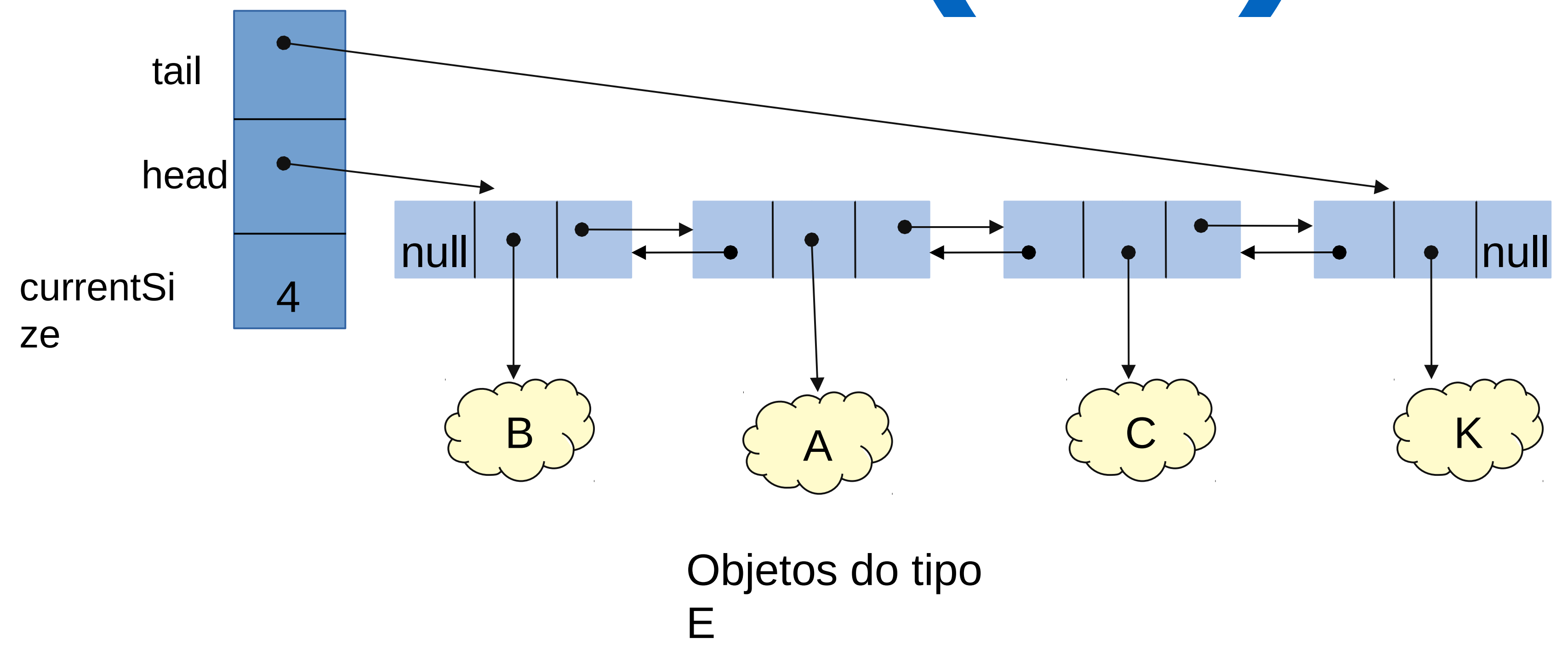
removeLast()



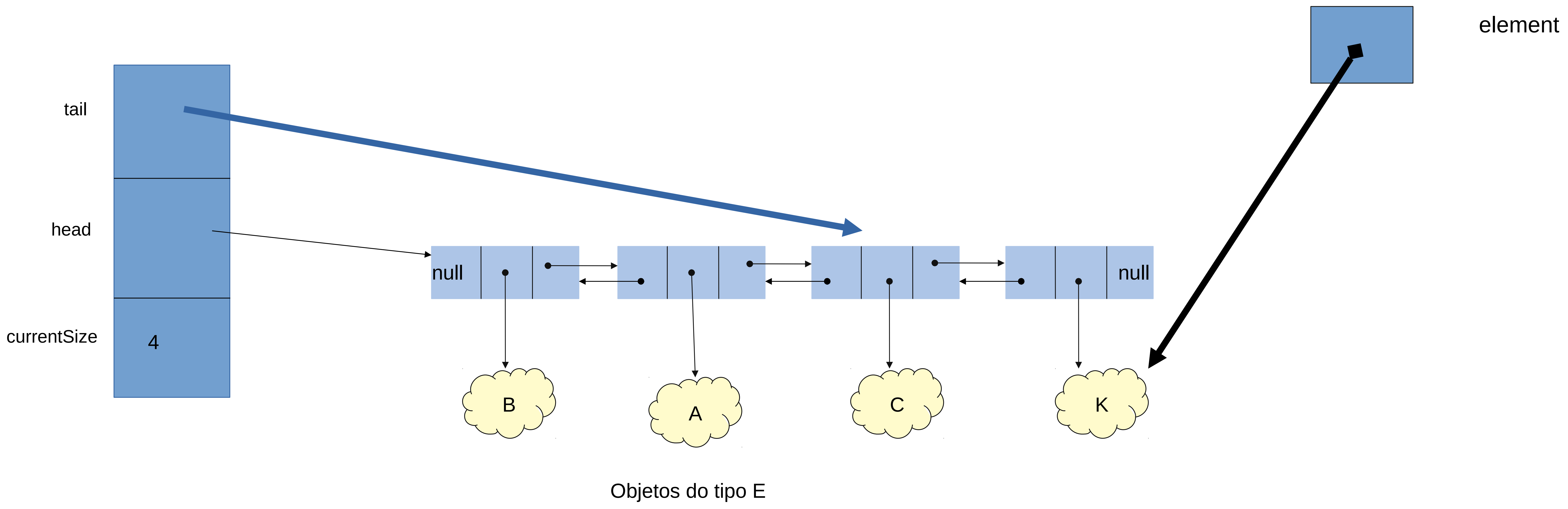
Lista com vários elementos

Classe DoublyLinkedList<E> (29)

```
/**
 * Removes and returns the element at the last position in the list.
 * @return element removed from the last position of the list
 * @throws NoSuchElementException - if size() == 0
 */
public E removeLast( ) {
   //TODO: Left as an exercise.


}
```

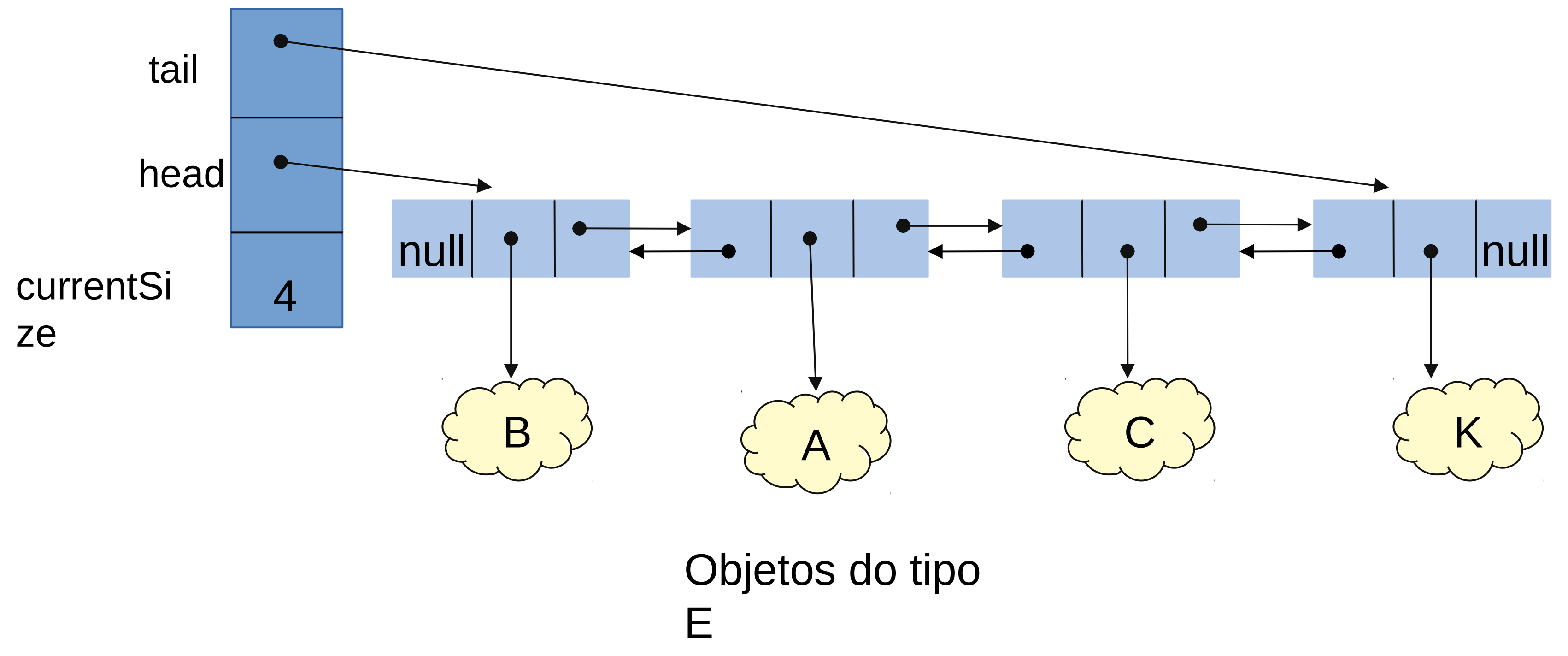


removeLast()

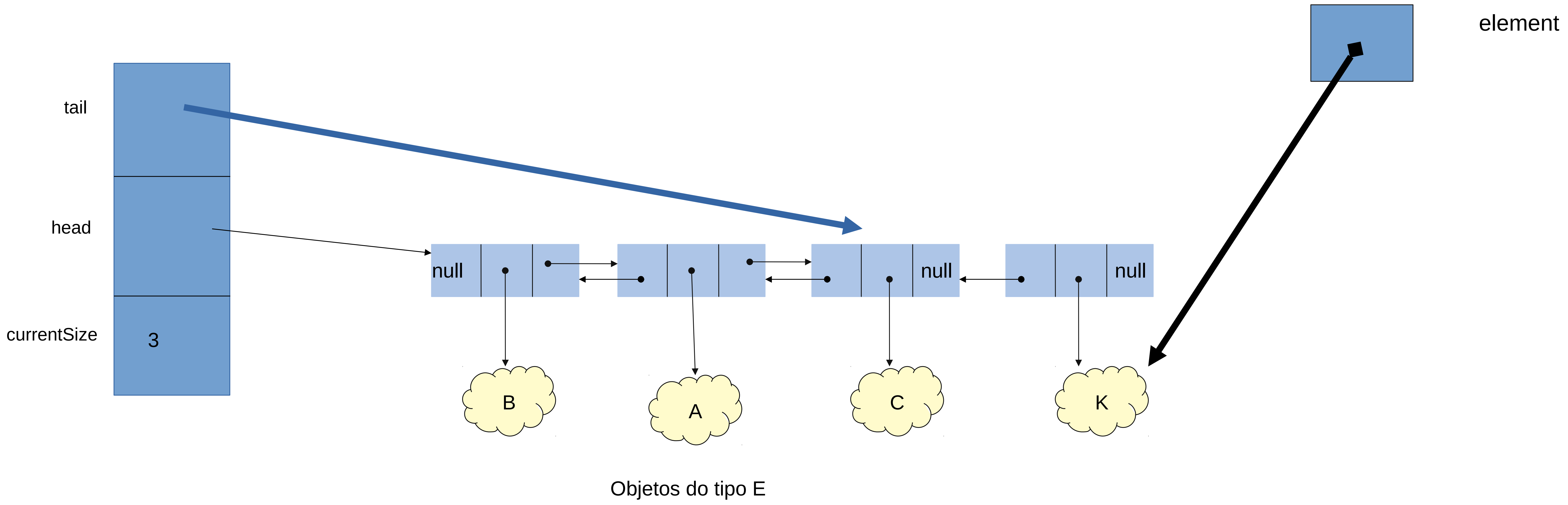


Classe DoublyLinkedList<E> (30)

```
/**
 * Removes and returns the element at the last position in the list.
 * @return element removed from the last position of the list
 * @throws NoSuchElementException - if size() == 0
 */
public E removeLast( ) {
   //TODO: Left as an exercise.


}
```

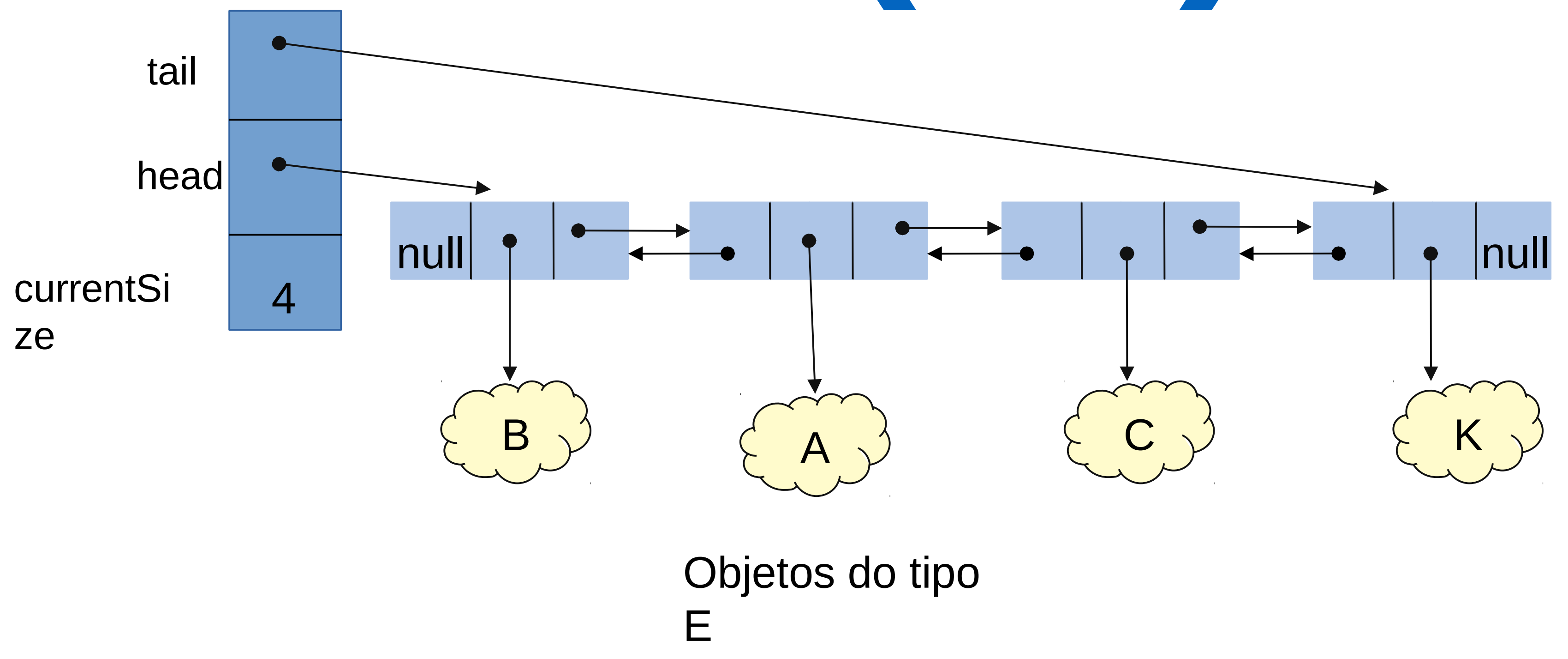


removeLast()

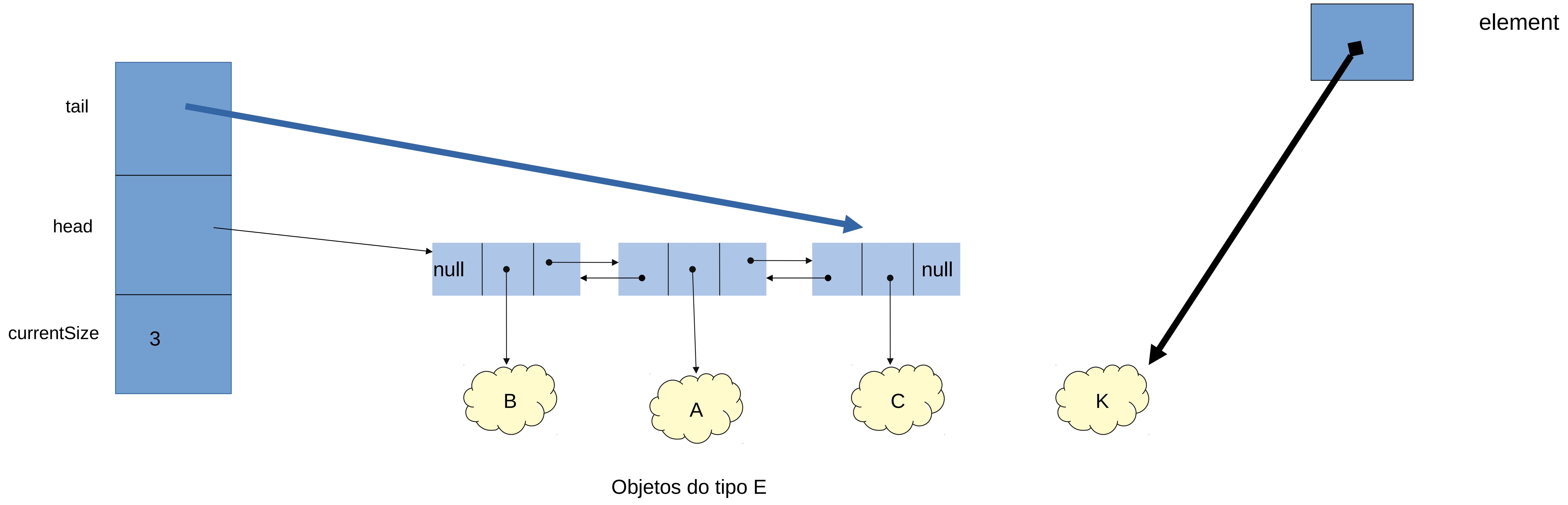


Classe DoublyLinkedList<E> (31)

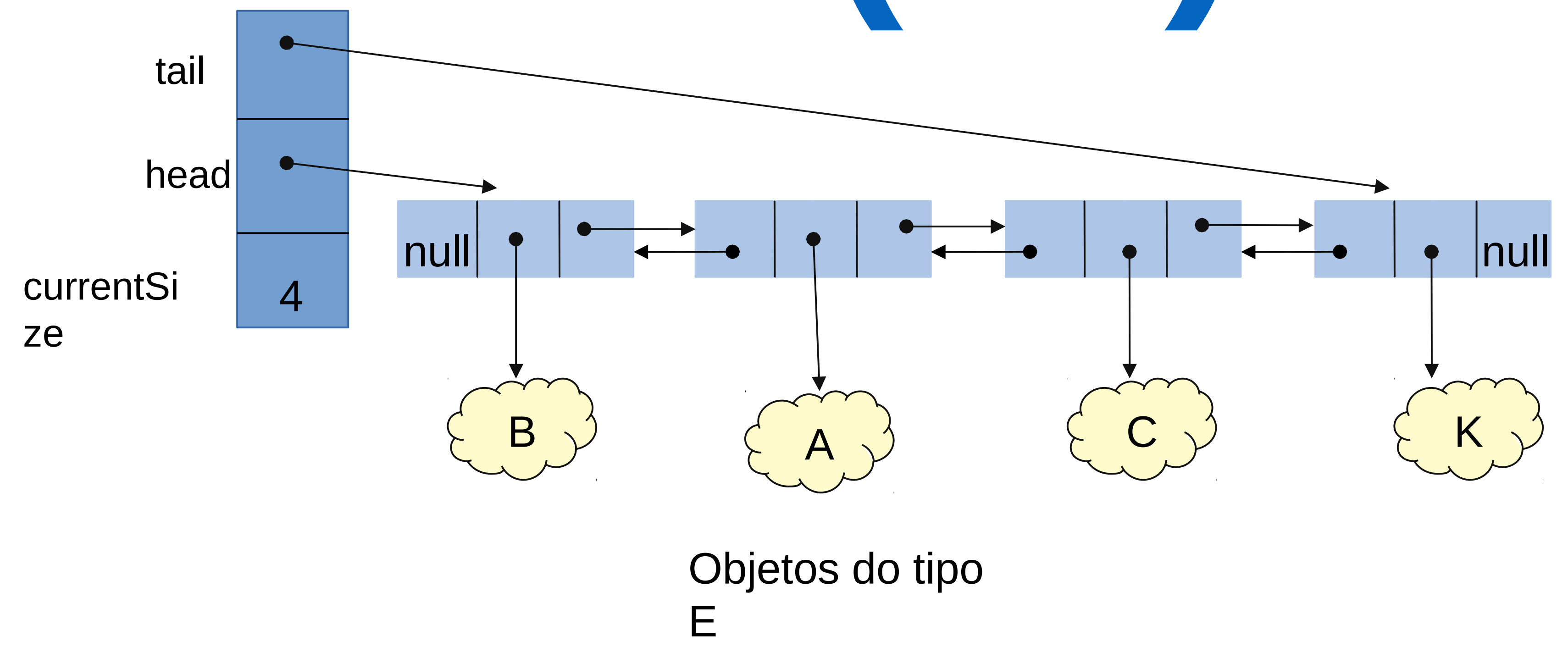
```
/**
 * Removes and returns the element at the last position in the list.
 * @return element removed from the last position of the list
 * @throws NoSuchElementException - if size() == 0
 */
public E removeLast( ) {
   //TODO: Left as an exercise.


}
```



removeLast()

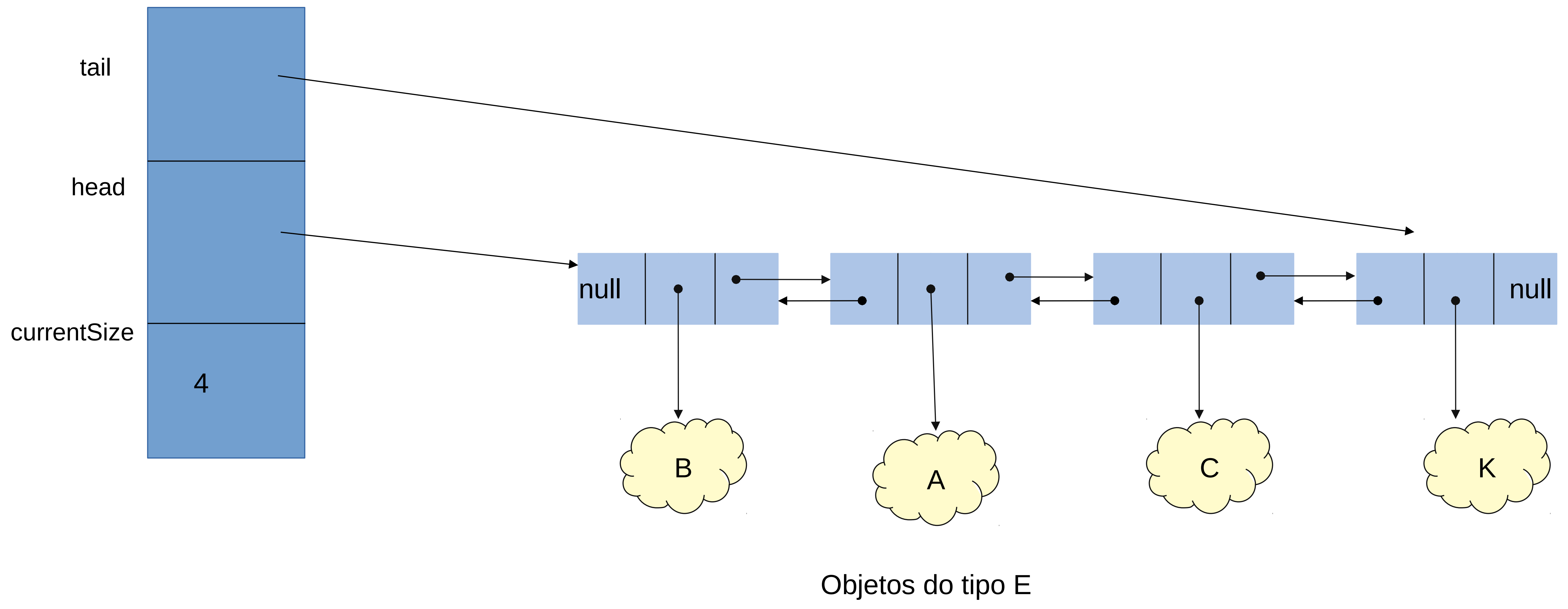


Classe DoublyLinkedList<E> (32)

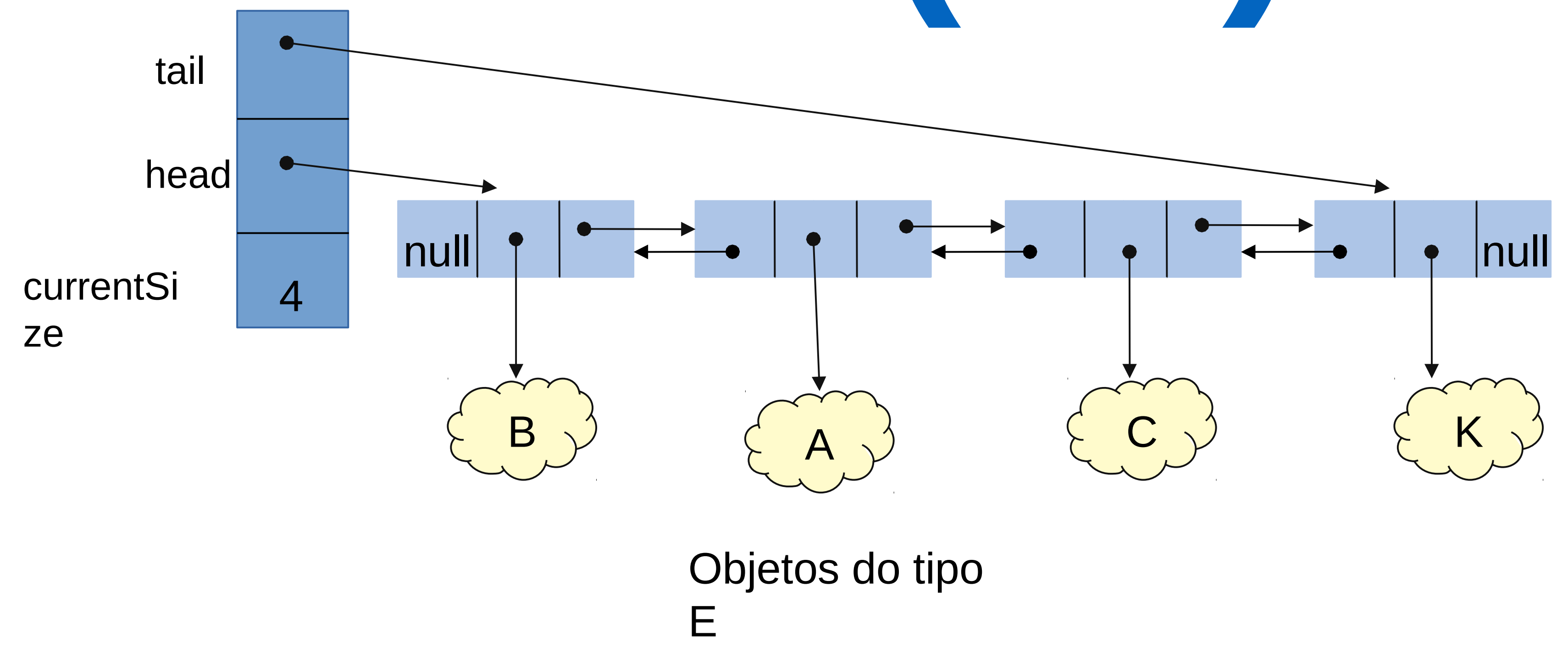


```
/**
 *  Removes and returns the element at the specified position in the list.
 * Range of valid positions: 0, ..., size()-1.
 * If the specified position is 0, remove corresponds to removeFirst.
 * If the specified position is size()-1, remove corresponds to removeLast.
 * @param position - position of element to be removed
 * @return element removed at position
 * @throws InvalidPositionException - if position is not valid in the list
 */
public E remove( int position ) {
   //TODO: Left as an exercise.

}
```

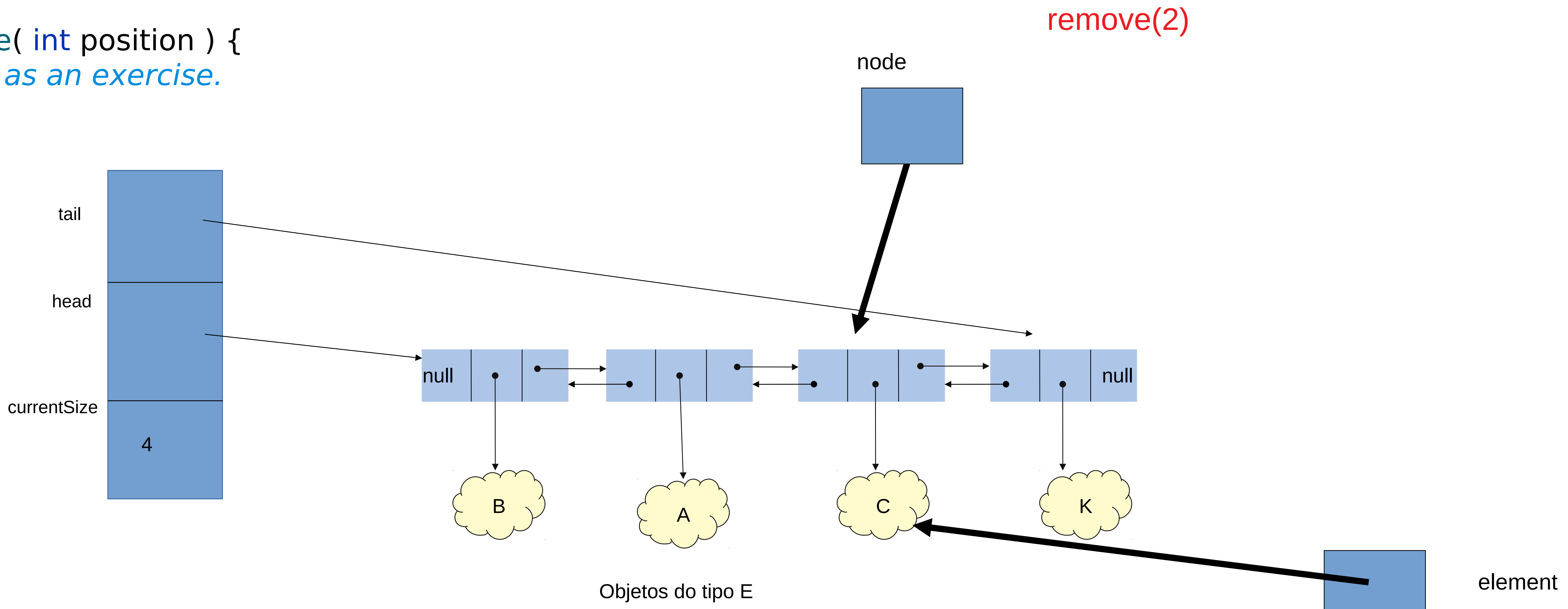
remove(2)



Classe DoublyLinkedList<E> (33)

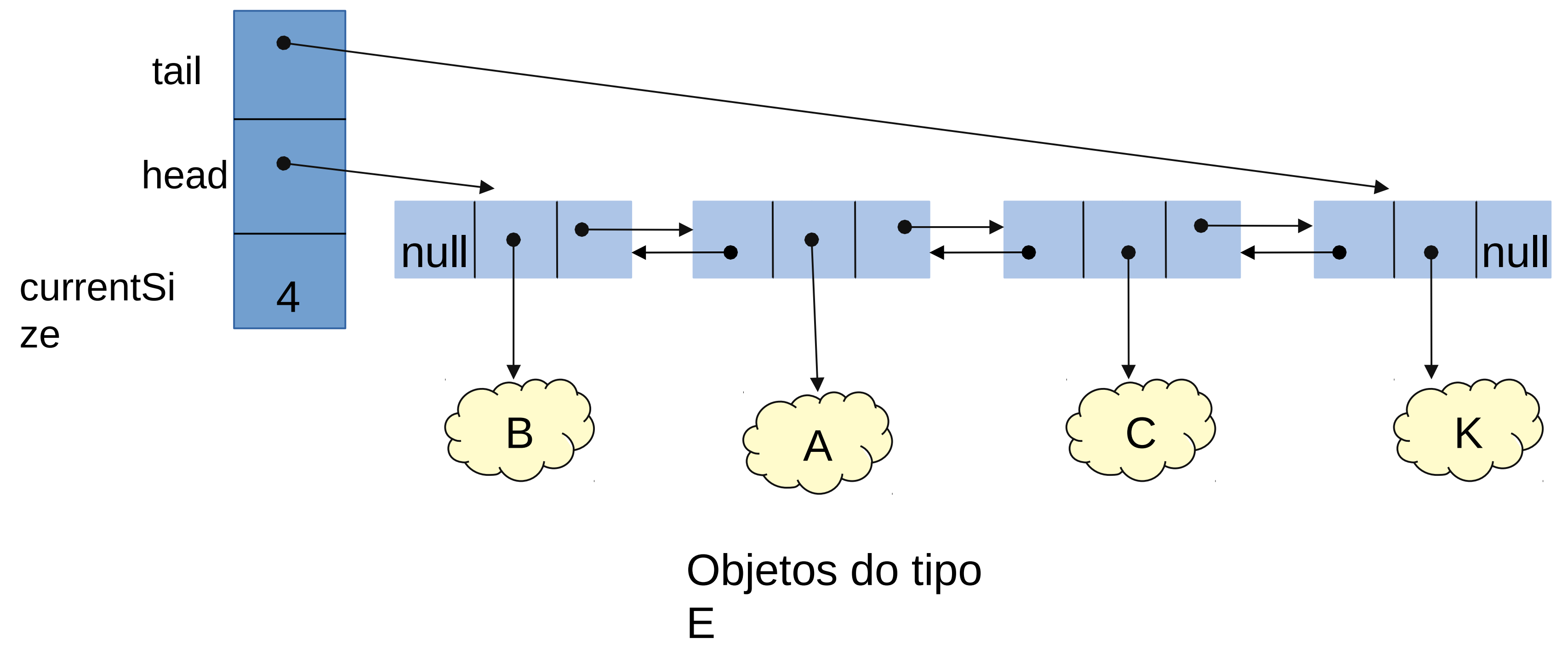


```
/**
 *  Removes and returns the element at the specified position in the list.
 * Range of valid positions: 0, ..., size()-1.
 * If the specified position is 0, remove corresponds to removeFirst.
 * If the specified position is size()-1, remove corresponds to removeLast.
 * @param position - position of element to be removed
 * @return element removed at position
 * @throws InvalidPositionException - if position is not valid in the list
 */
public E remove( int position ) {
   //TODO: Left as an exercise.

}
```

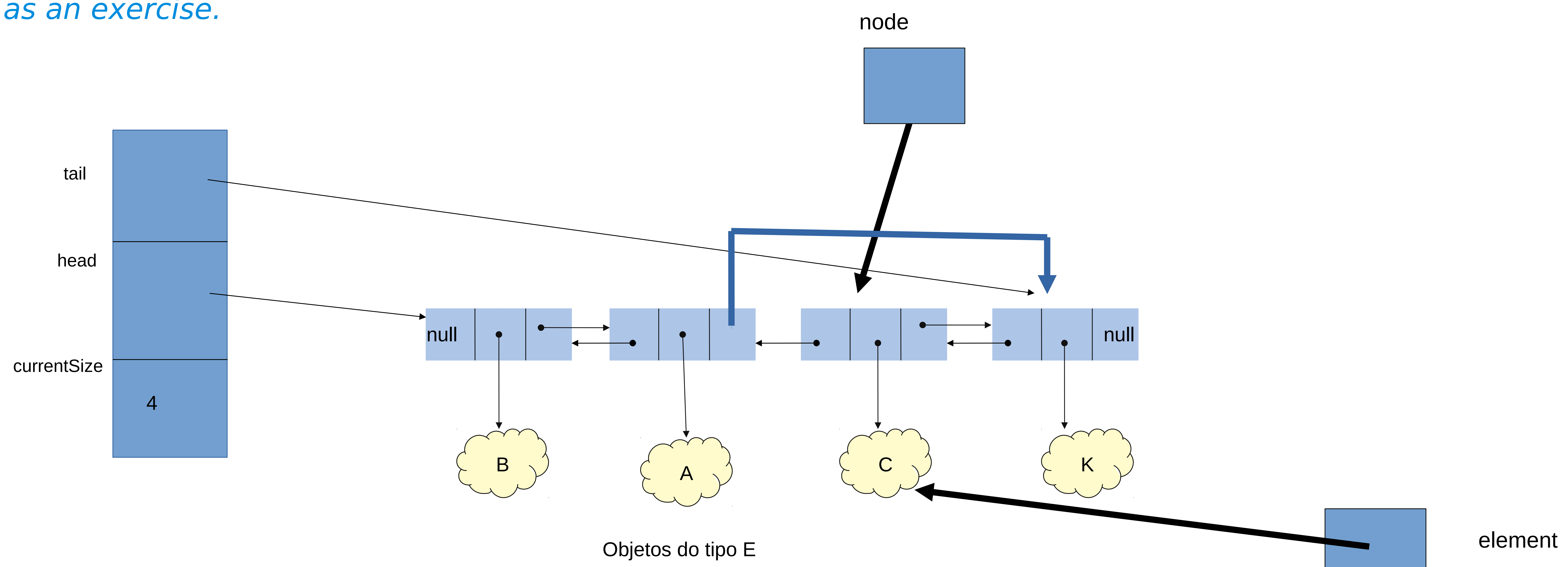


Classe DoublyLinkedList<E> (34)

```
/**
 *  Removes and returns the element at the specified position in the list.
 * Range of valid positions: 0, ..., size()-1.
 * If the specified position is 0, remove corresponds to removeFirst.
 * If the specified position is size()-1, remove corresponds to removeLast.
 * @param position - position of element to be removed
 * @return element removed at position
 * @throws InvalidPositionException - if position is not valid in the list
 */
public E remove( int position ) {
   //TODO: Left as an exercise.

}
```

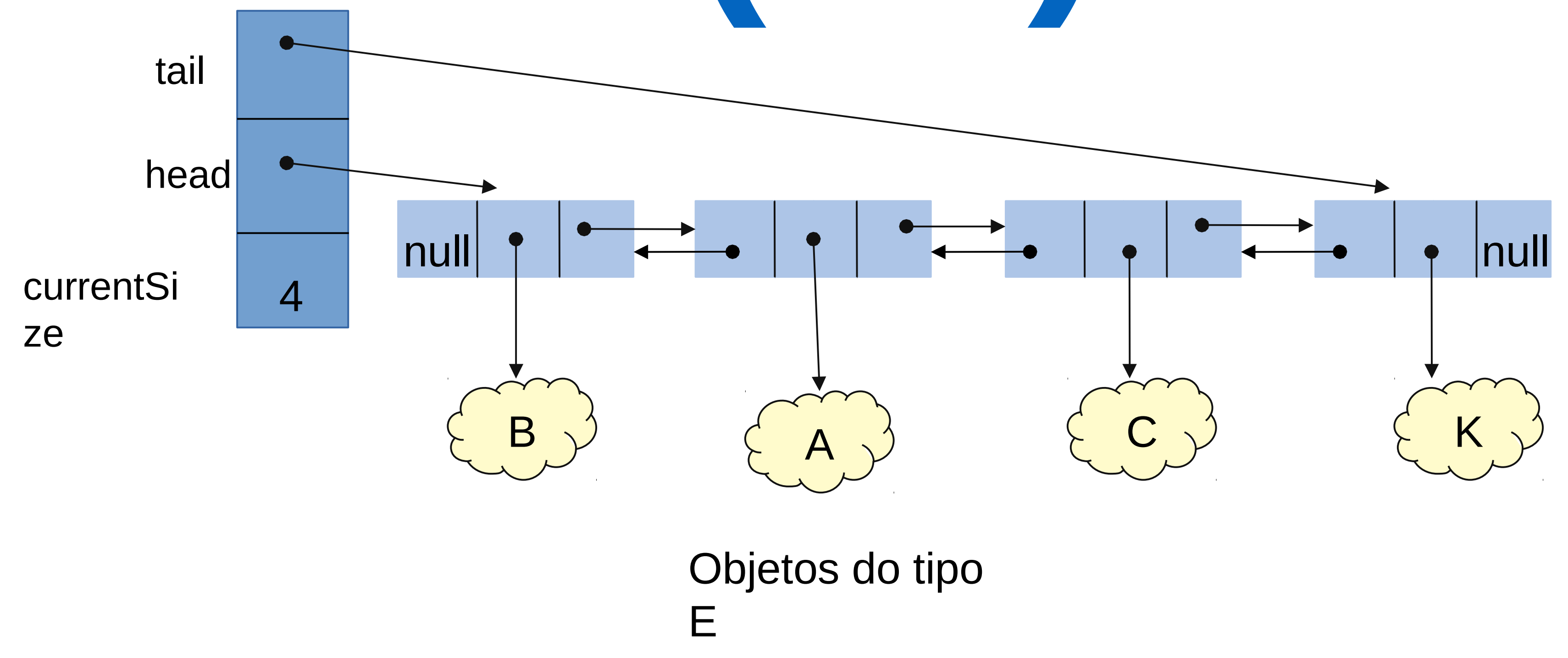


remove(2)

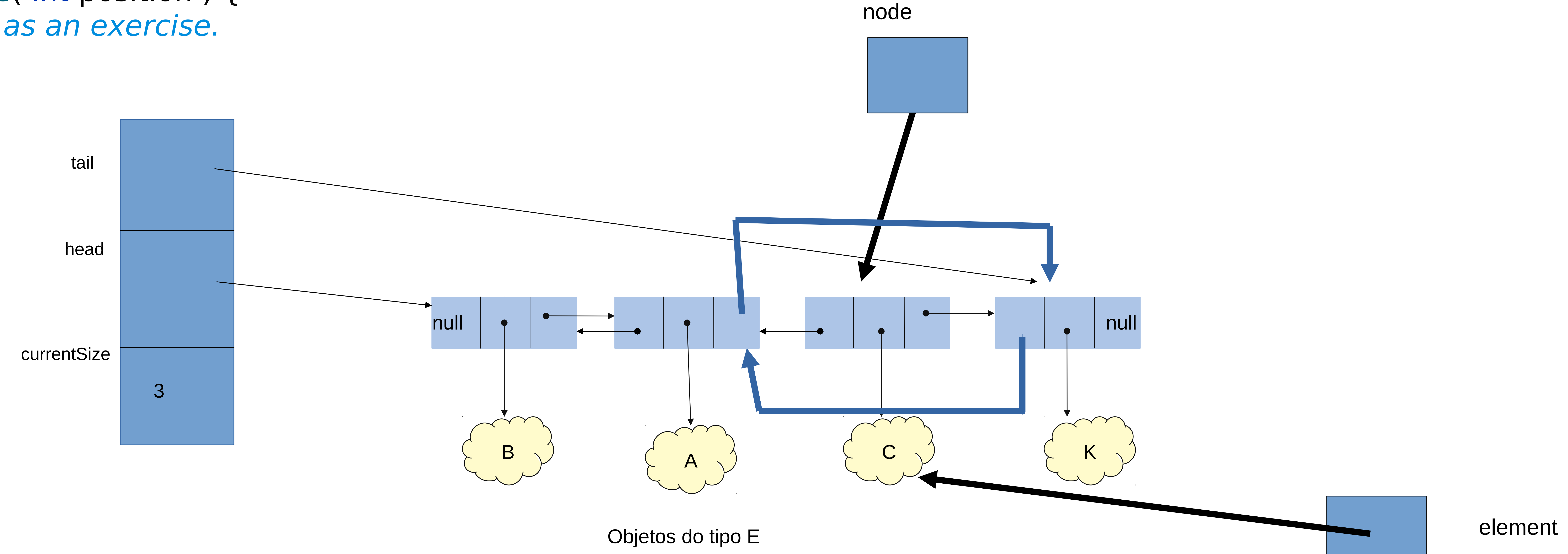


Classe DoublyLinkedList<E> (35)

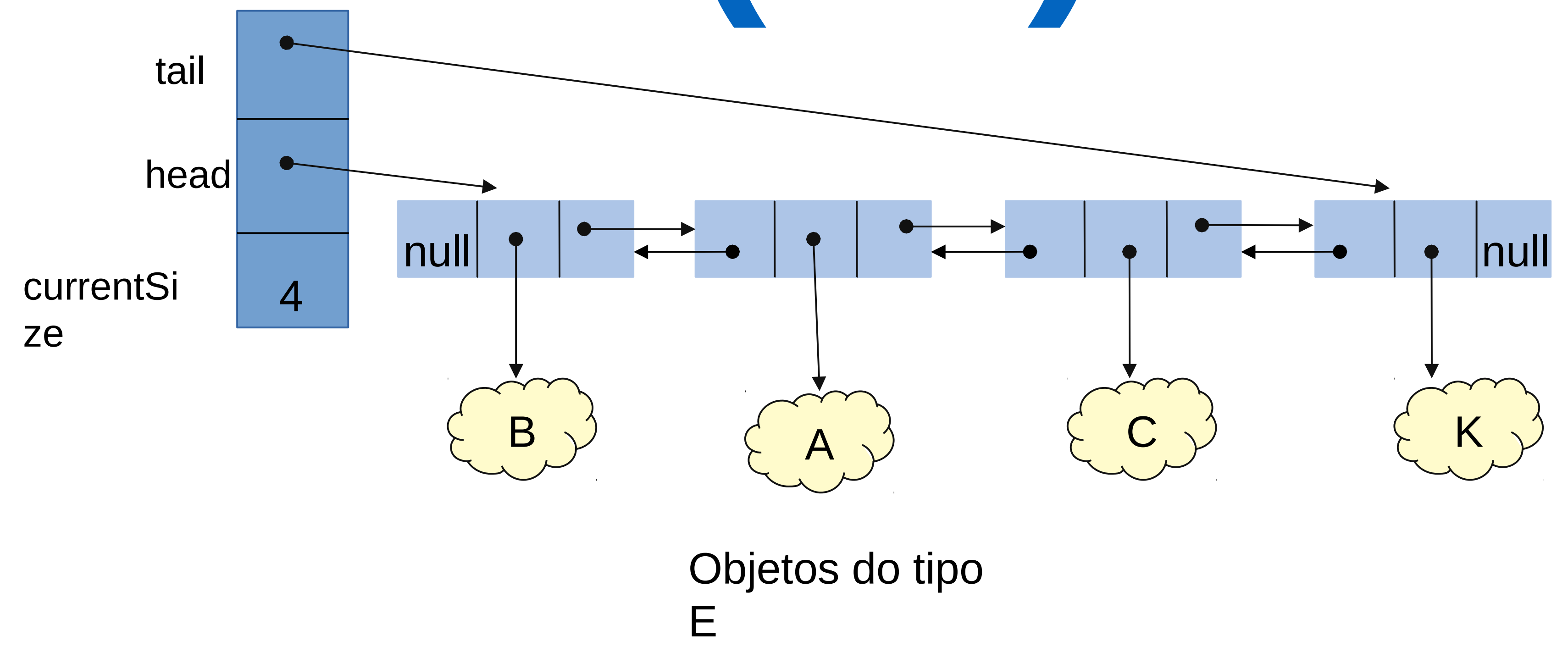
```
/**
 *  Removes and returns the element at the specified position in the list.
 * Range of valid positions: 0, ..., size()-1.
 * If the specified position is 0, remove corresponds to removeFirst.
 * If the specified position is size()-1, remove corresponds to removeLast.
 * @param position - position of element to be removed
 * @return element removed at position
 * @throws InvalidPositionException - if position is not valid in the list
 */
public E remove( int position ) {
   //TODO: Left as an exercise.

}
```



remove(2)

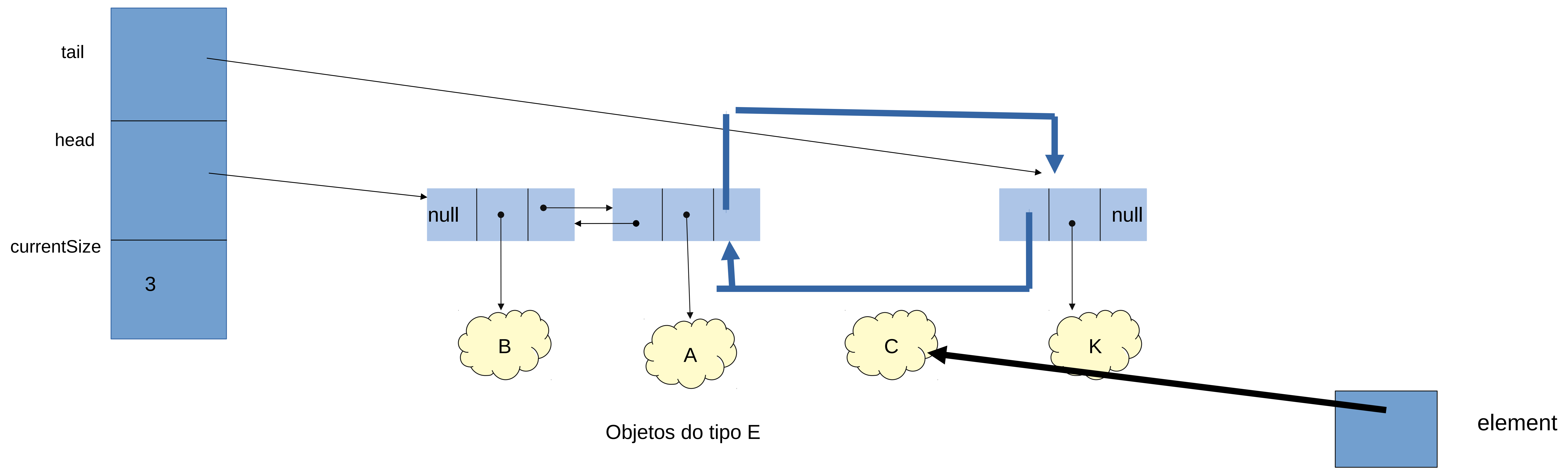


Classe DoublyLinkedList<E> (36)



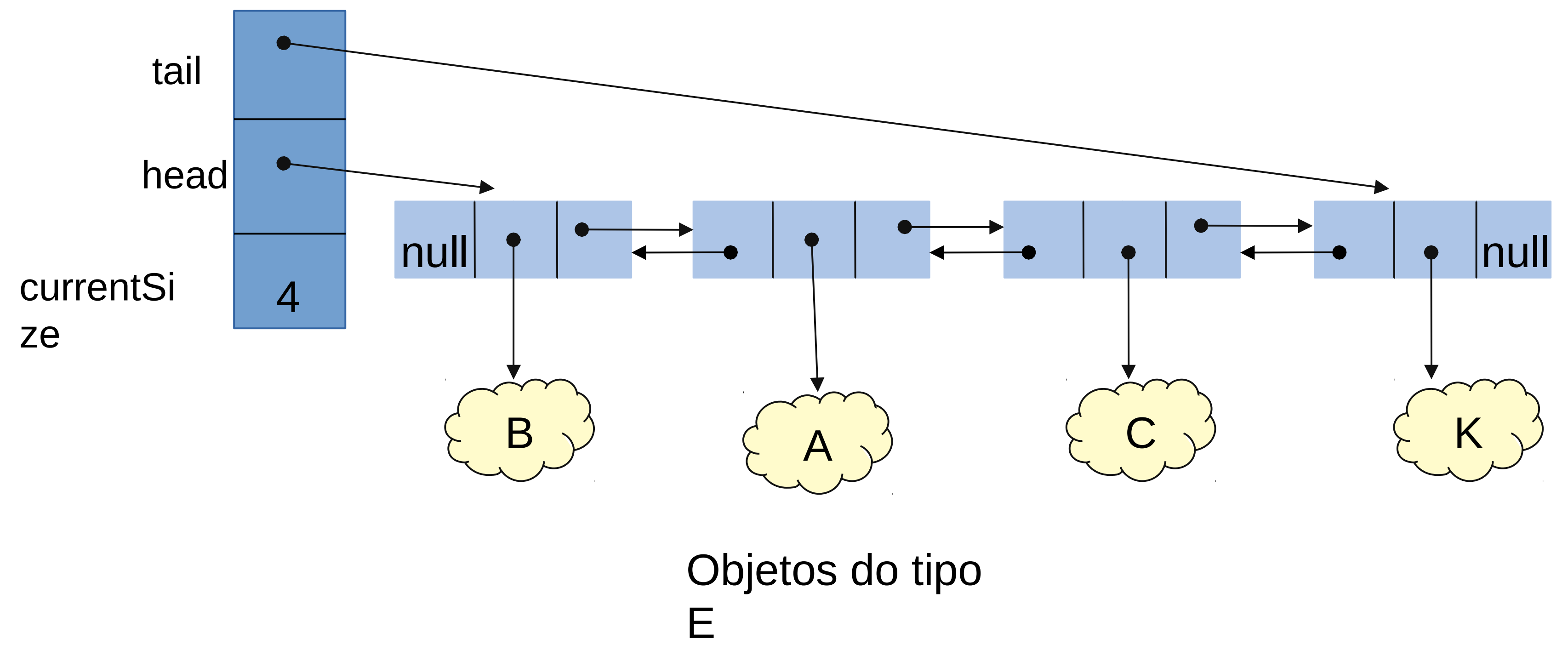
```
/**
 *  Removes and returns the element at the specified position in the list.
 * Range of valid positions: 0, ..., size()-1.
 * If the specified position is 0, remove corresponds to removeFirst.
 * If the specified position is size()-1, remove corresponds to removeLast.
 * @param position - position of element to be removed
 * @return element removed at position
 * @throws InvalidPositionException - if position is not valid in the list
 */
public E remove( int position ) {
   //TODO: Left as an exercise.

}
```

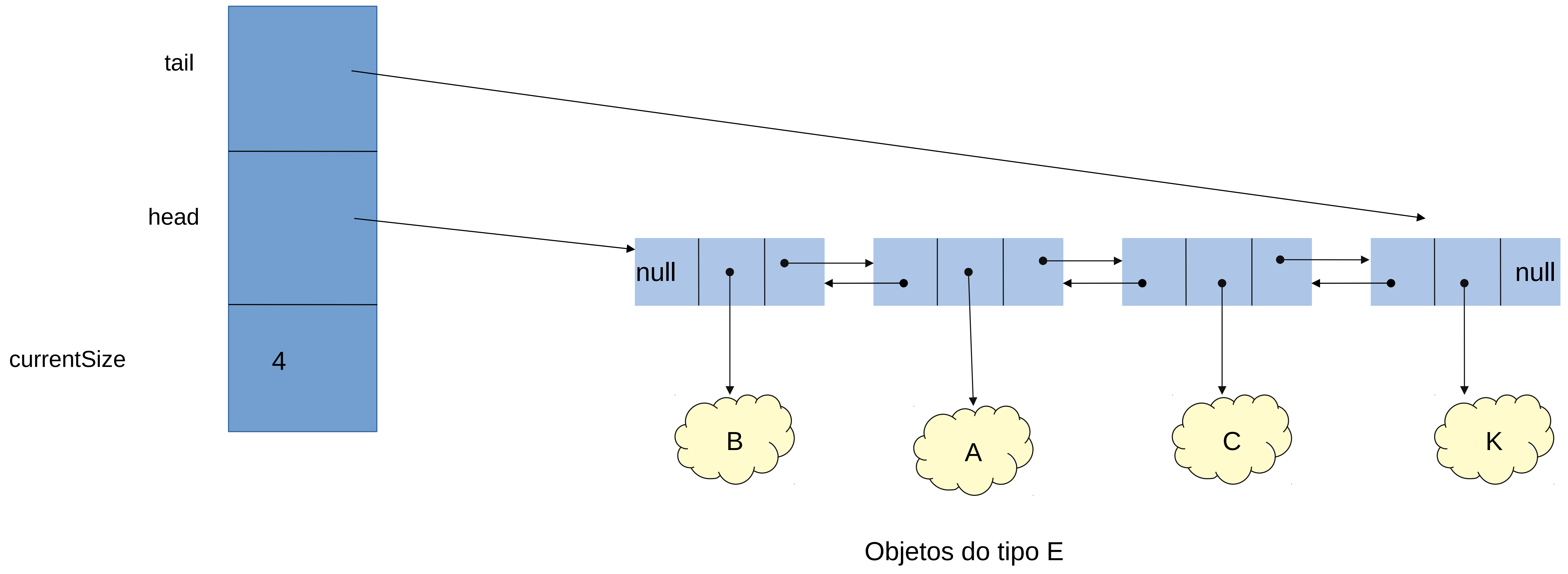
remove(2)



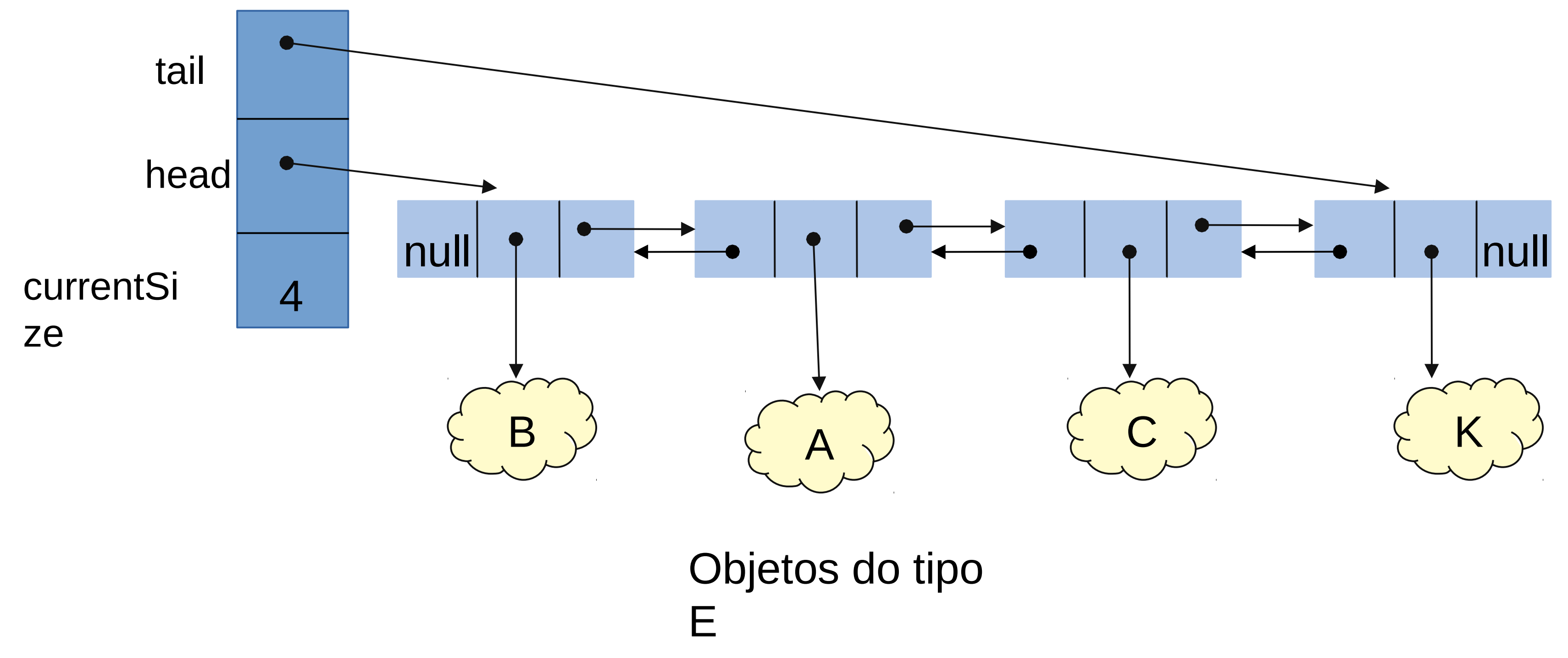
Classe DoublyLinkedList<E> (37)

```
/**
 * Returns an iterator of the elements in the list (in proper sequence).
 * @return Iterator of the elements in the list
 */
public Iterator<E> iterator() {
    return new DoublyIterator<>(head);
}
```





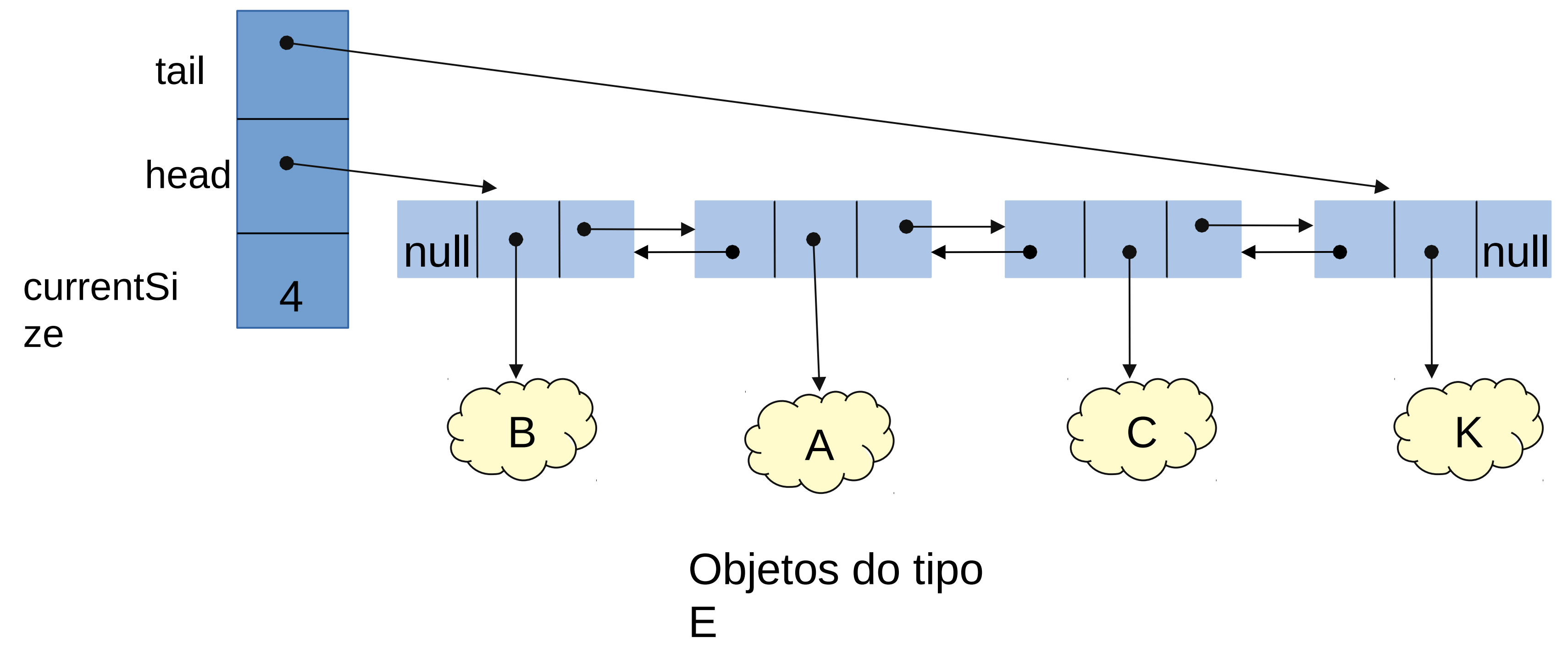
Classe DoublyLinkedList<E> (38)



```java
/**
 * Returns an iterator of the elements in the list (in proper sequence).
 * @return Iterator of the elements in the list
 */
public Iterator<E> iterator() {
    return new DoublyIterator<>(head);
}
```

```java
class DoublyIterator<E> implements Iterator<E> {
   /**
    * Node with the first element in the iteration.
    */
   private DoublyListNode<E> firstNode;

   /**
    * Node with the next element in the iteration.
    */
   DoublyListNode<E> nextToReturn;


      //TODO: Left as an exercise.

 }
```

Classe DoublyLinkedList<E> (39)

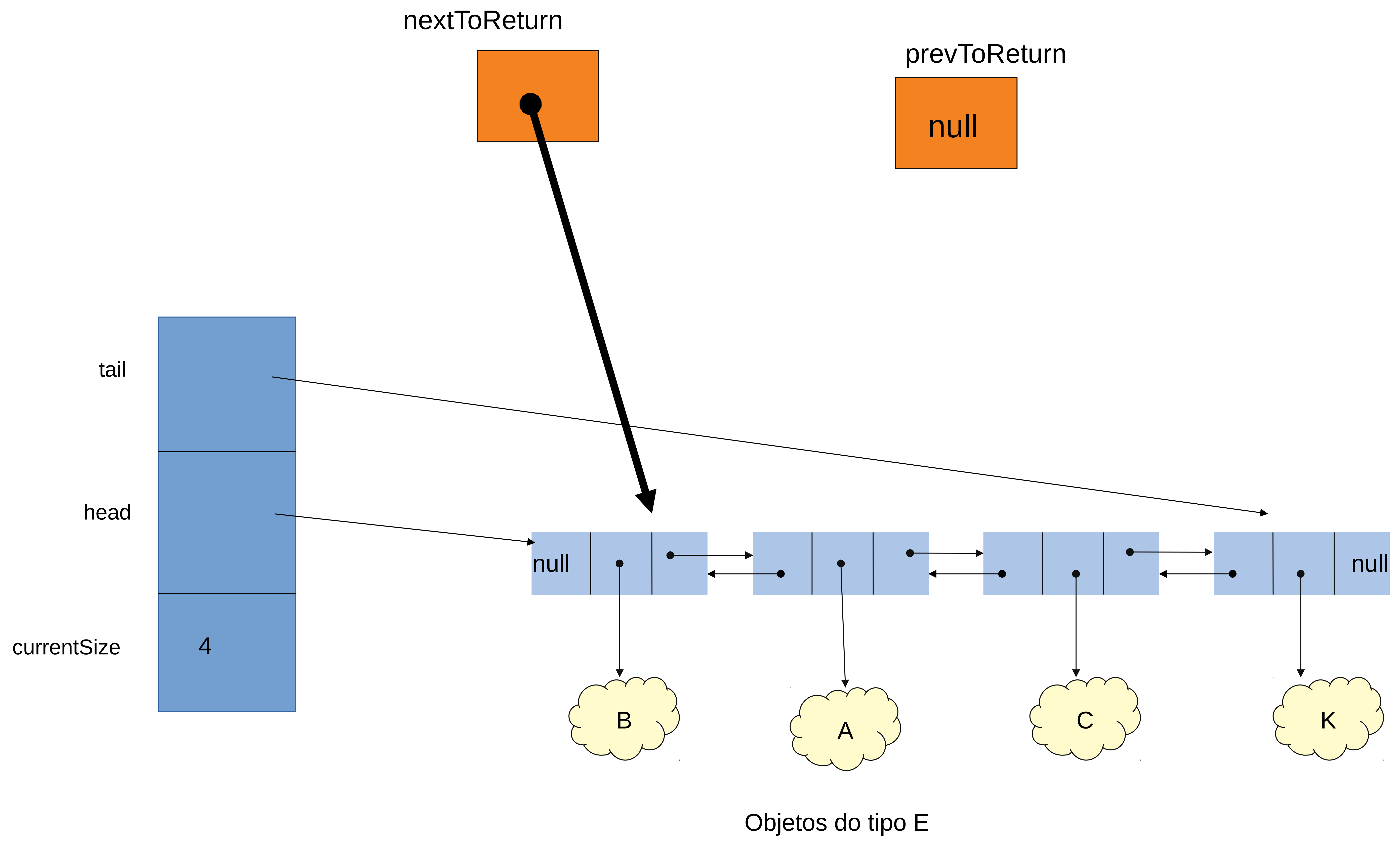


```java
/**
 * Returns a two-way iterator of the elements in the list.
 *
 * @return Two-Way Iterator of the elements in the list
 */

public TwoWayIterator<E> twoWayiterator() {
    return new TwoWayDoublyIterator<>(head, tail);
}
```

```java
class TwoWayDoublyIterator<E> extends DoublyIterator<E>
        implements TwoWayIterator<E> {

    /**
     * Node with the first element in the iteration.
     */
    private DoublyListNode<E> lastNode;
    /**
     * Node with the previous element in the iteration.
     */
    private DoublyListNode<E> prevToReturn;


        //TODO: Left as an exercise.

}
```

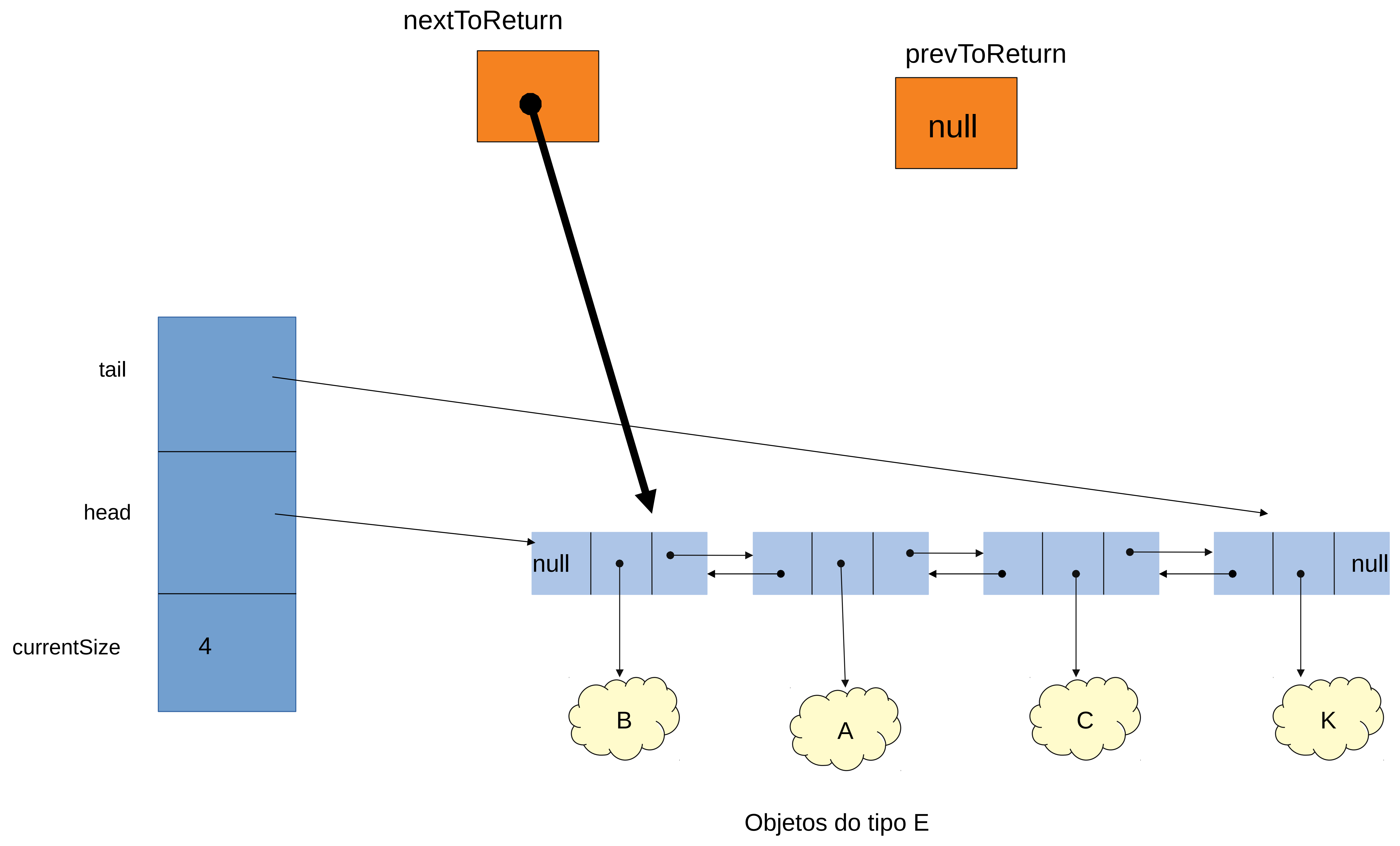
Como funciona o iterador (1)

rewind()	
next()	
hasNext()	true
fullforward()	
previous()	
hasPrevious()	false



Como funciona o iterador (2)

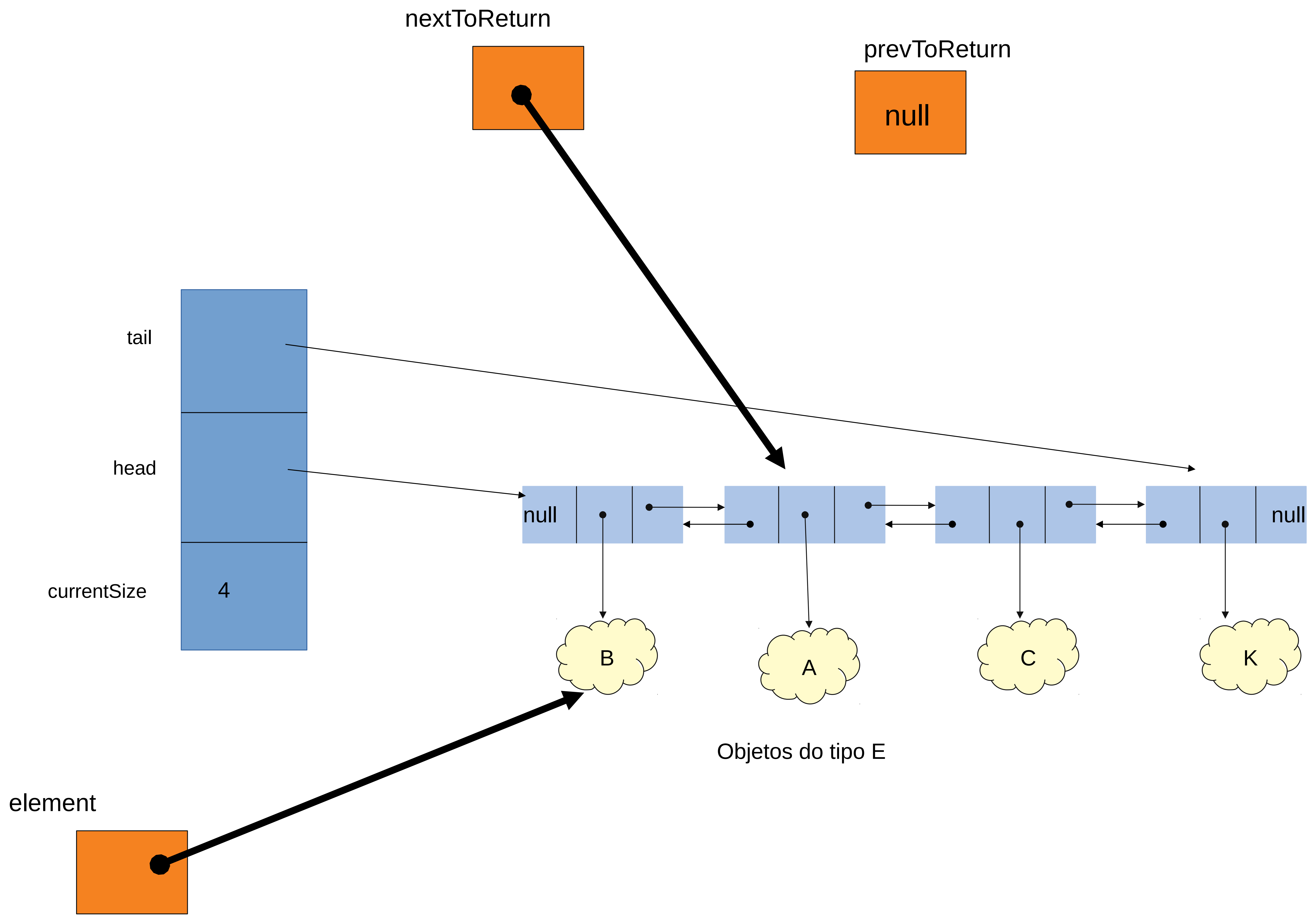
rewind()	
next()	
hasNext()	
fullforward()	
previous()	
hasPrevious()	



Como funciona o iterador (3)

rewind()	
next()	B
hasNext()	
fullforward	
previous()	
hasPrevious()	

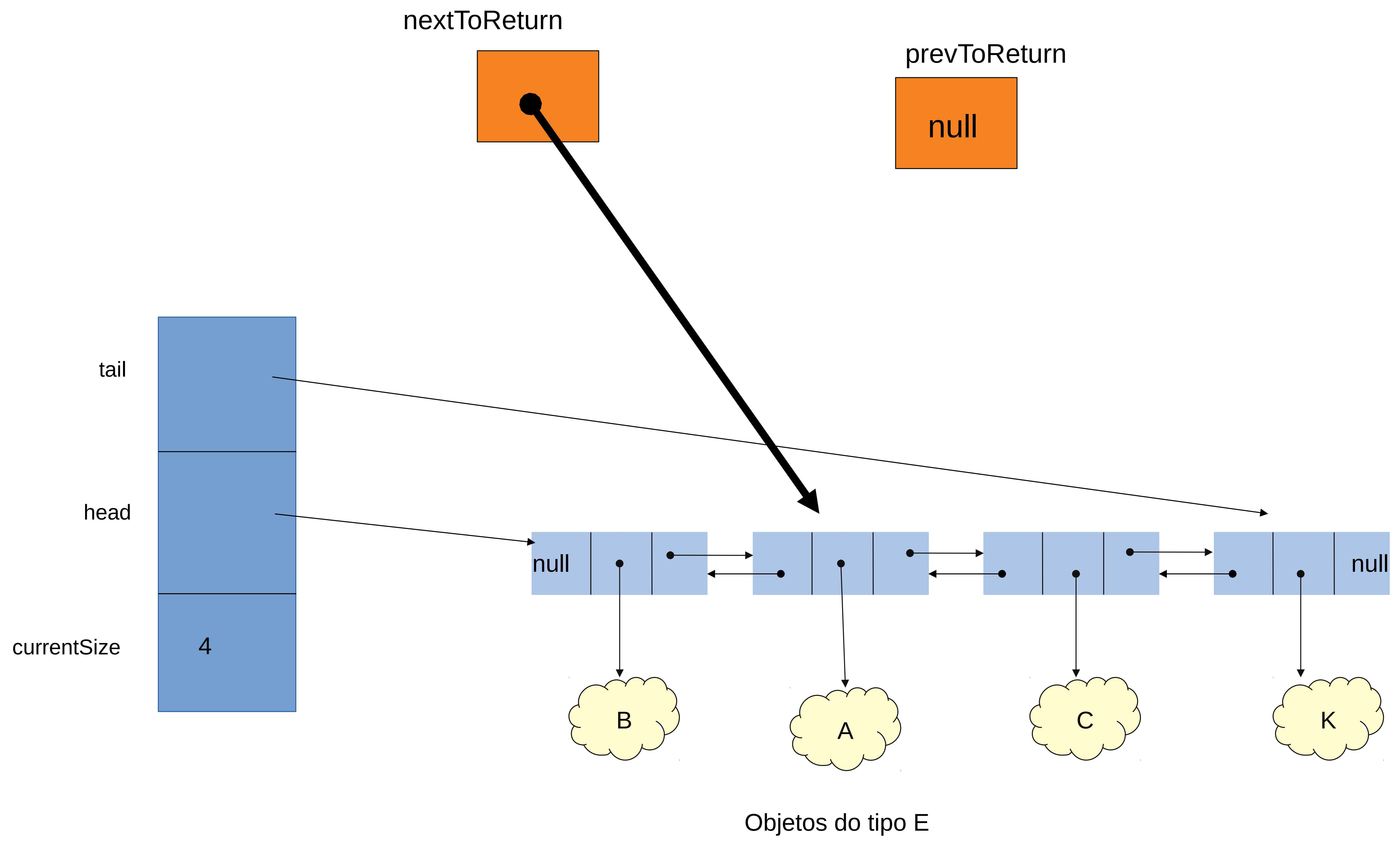
B



Como funciona o iterador (4)

rewind()	
next()	
hasNext()	true
fullforward()	
previous()	
hasPrevious()	false

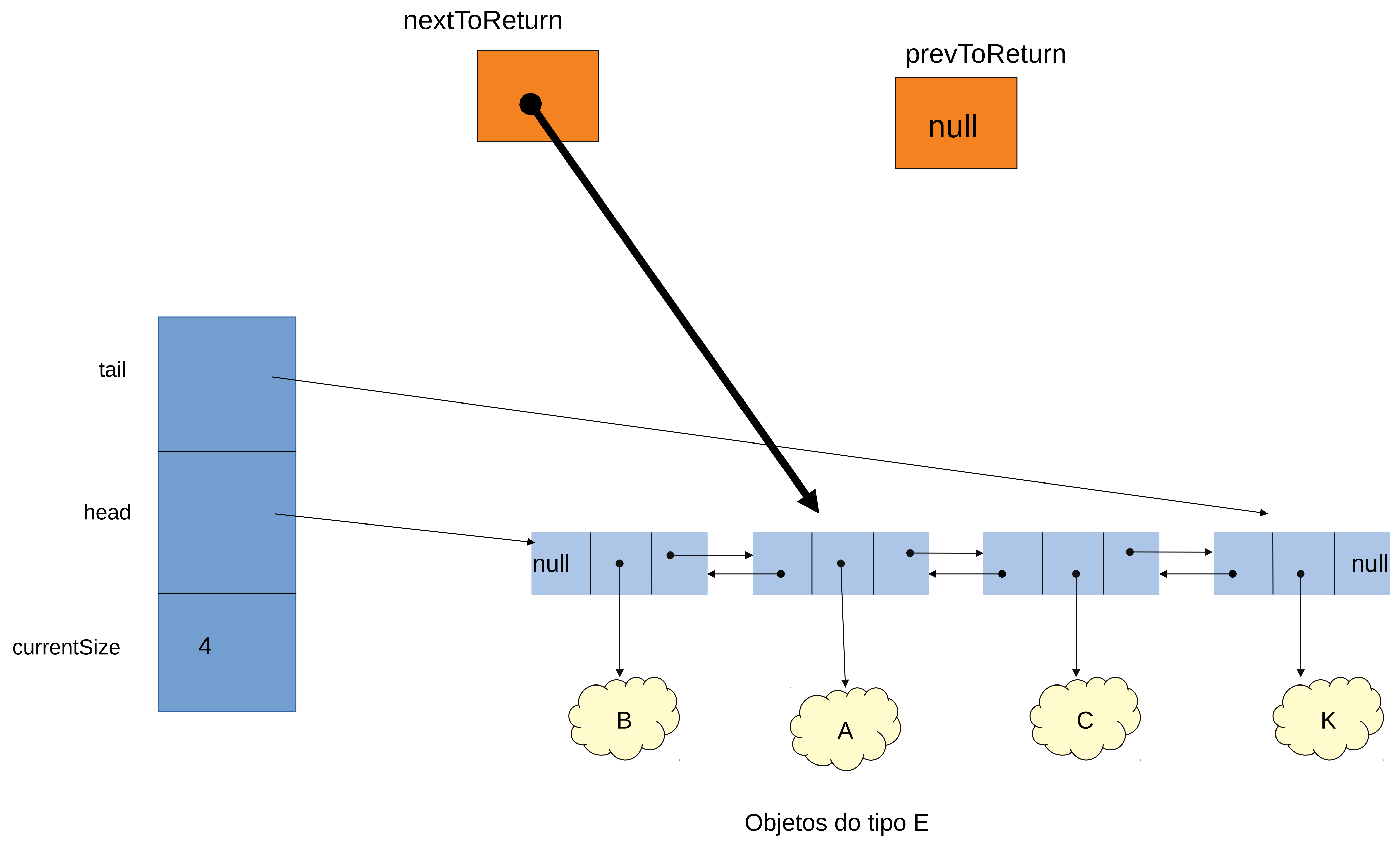
B



Como funciona o iterador (5)

rewind()	
next()	
hasNext()	
fullforward()	
previous()	
hasPrevious()	

B

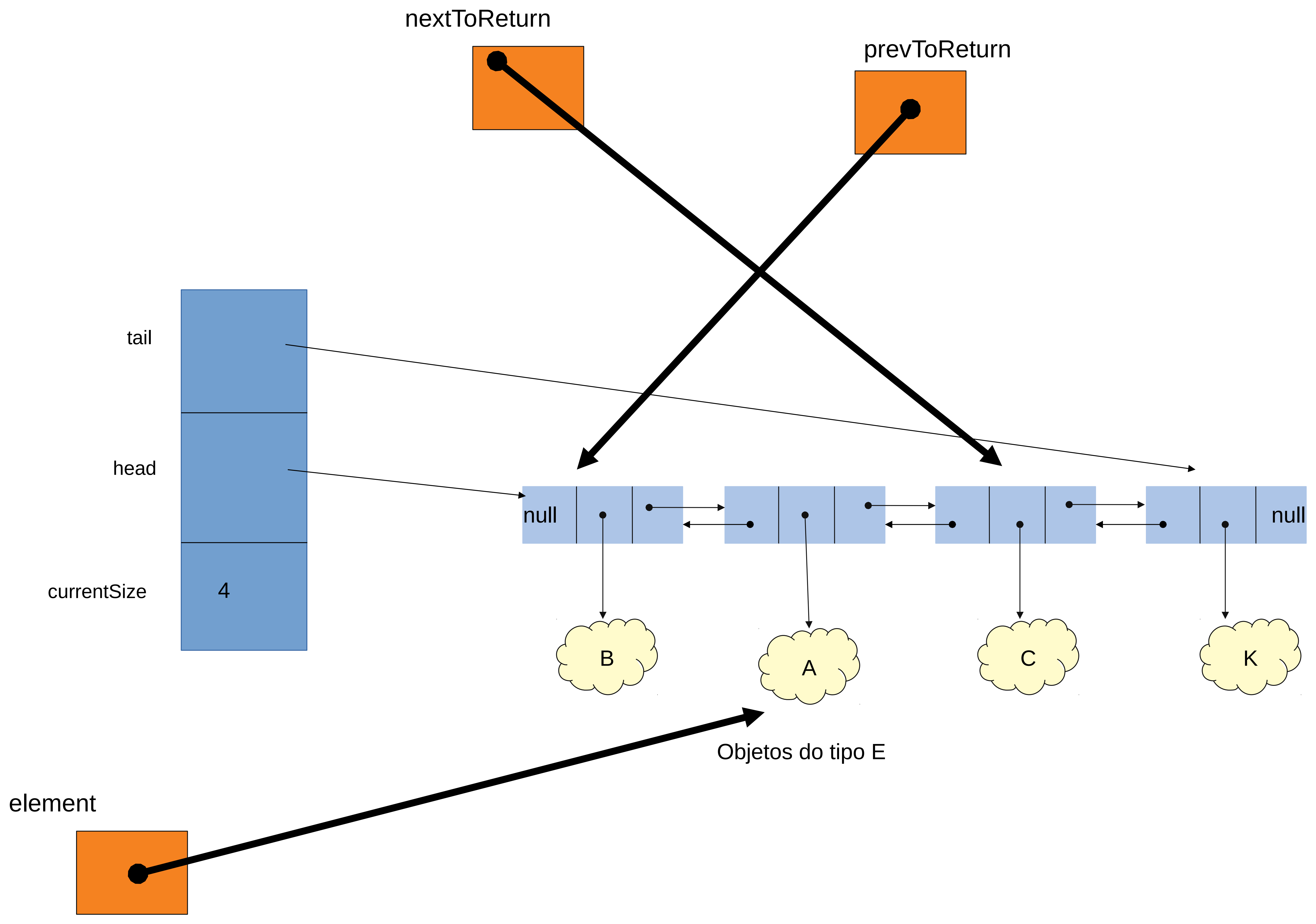


Como funciona o iterador (6)

rewind()	
next()	A
hasNext()	

fullforward	
previous()	
hasPrevious()	

B; A

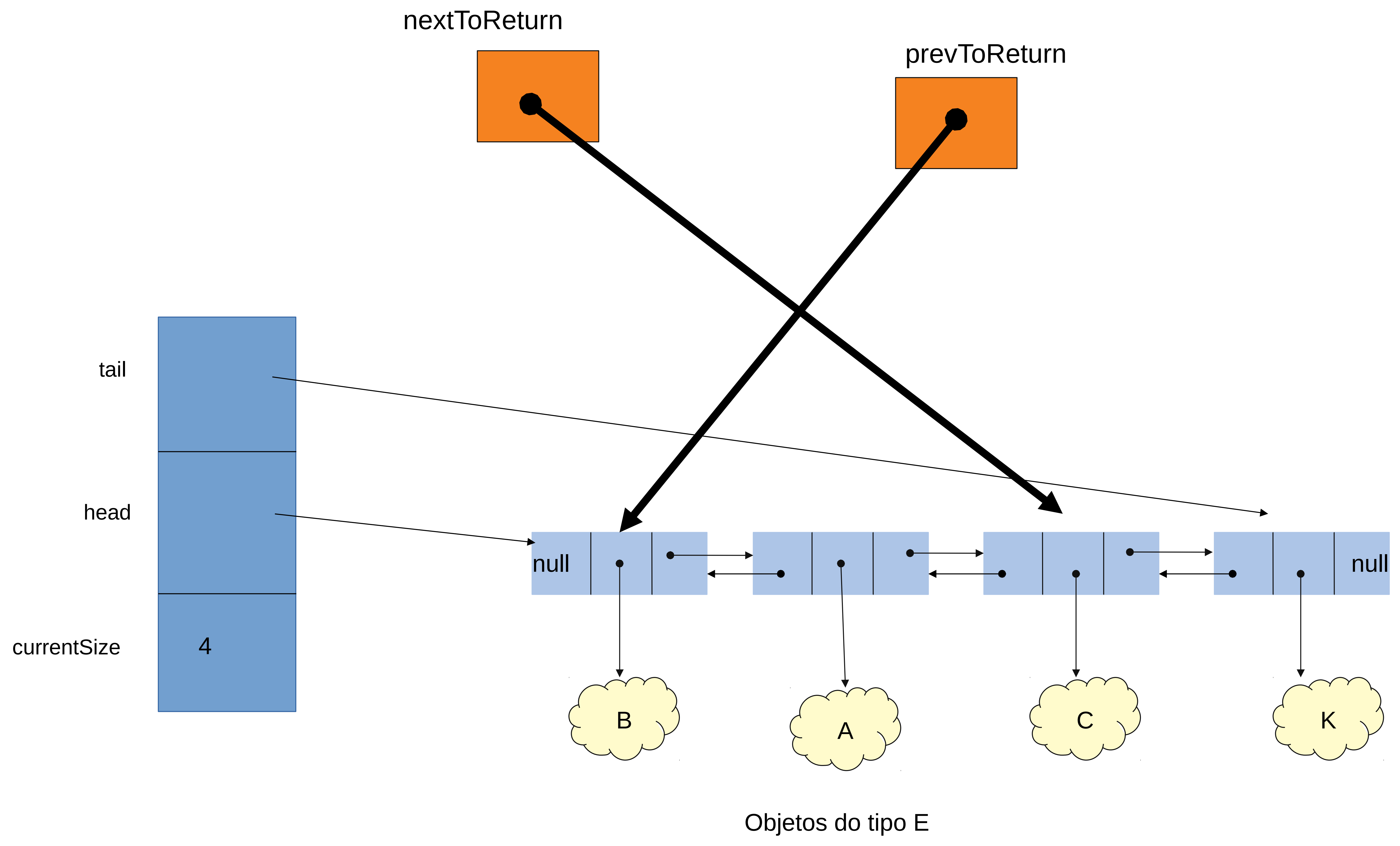


Como funciona o iterador (7)

rewind()	
next()	
hasNext()	true

fullforward()	
previous()	
hasPrevious()	true

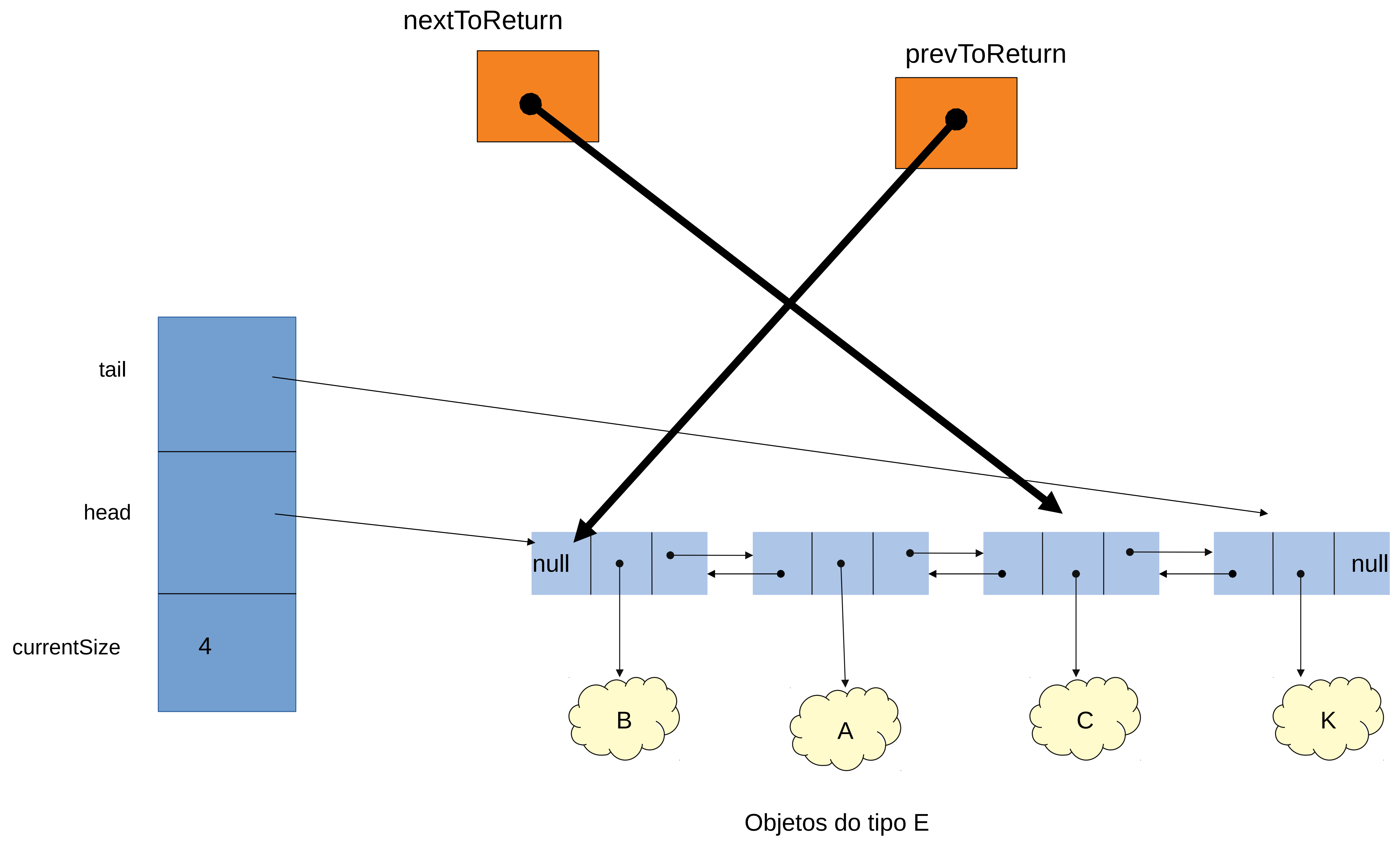
B; A



Como funciona o iterador (8)

rewind()	
next()	
hasNext()	
fullforward	
previous()	
hasPrevious()	

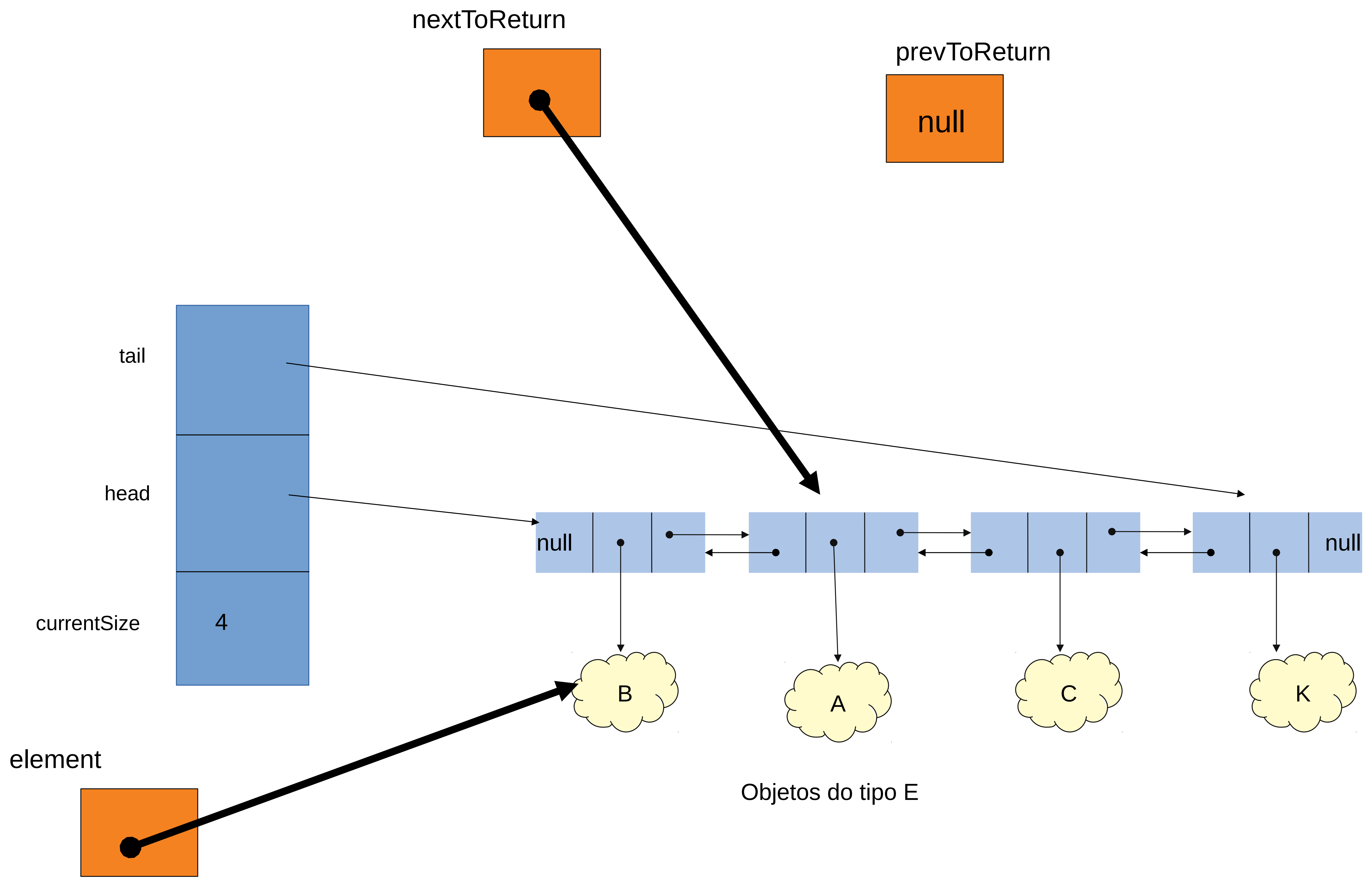
B; A



Como funciona o iterador (9)

rewind()	
next()	
hasNext()	
fullforward	
previous()	B
hasPrevious()	

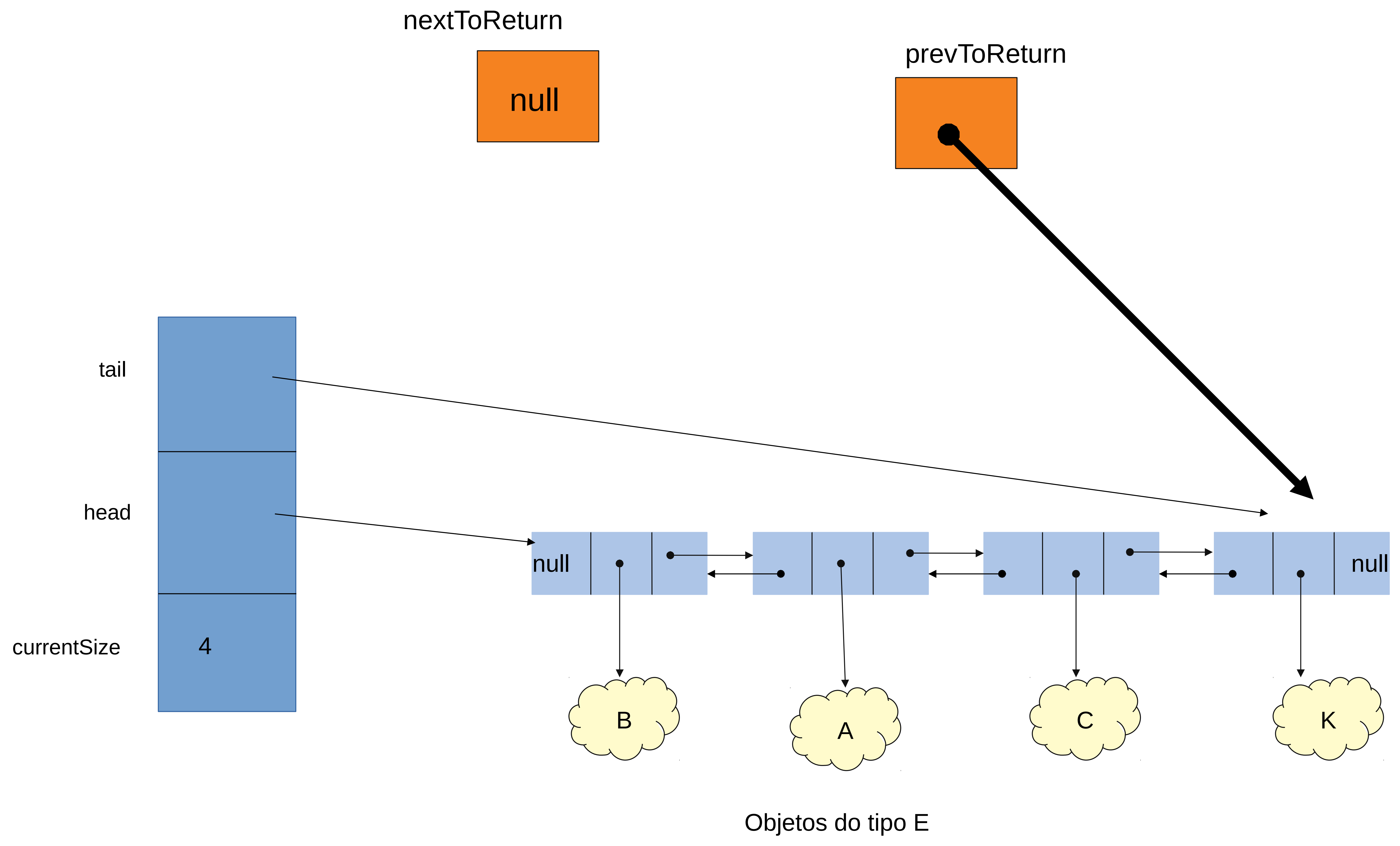
B; A; B



Como funciona o iterador (10)

rewind()	
next()	
hasNext()	
fullforward()	
previous()	
hasPrevious()	

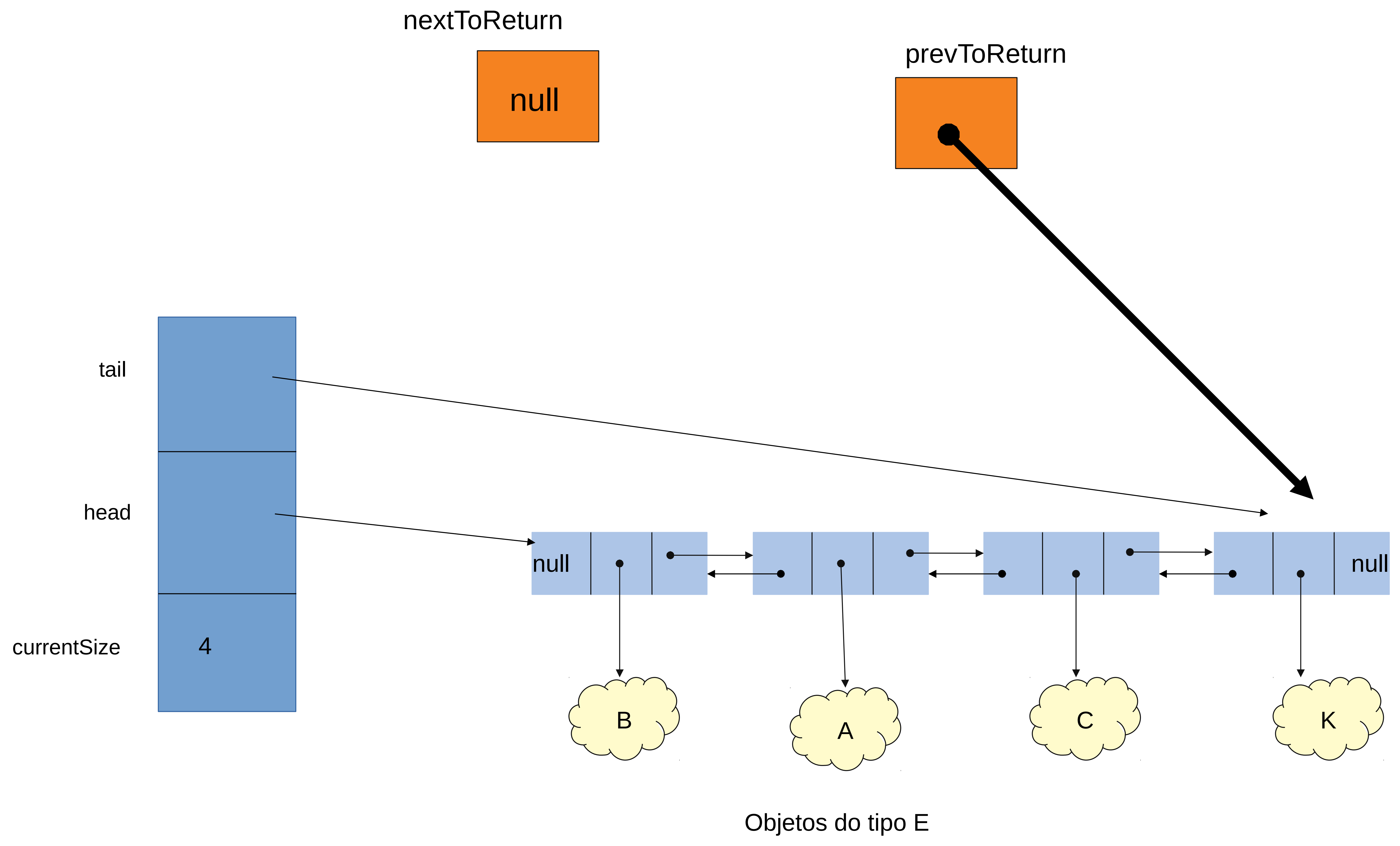
B; A; B



Como funciona o iterador (11)

rewind()	
next()	
hasNext()	false
fullforward()	
previous()	
hasPrevious()	true

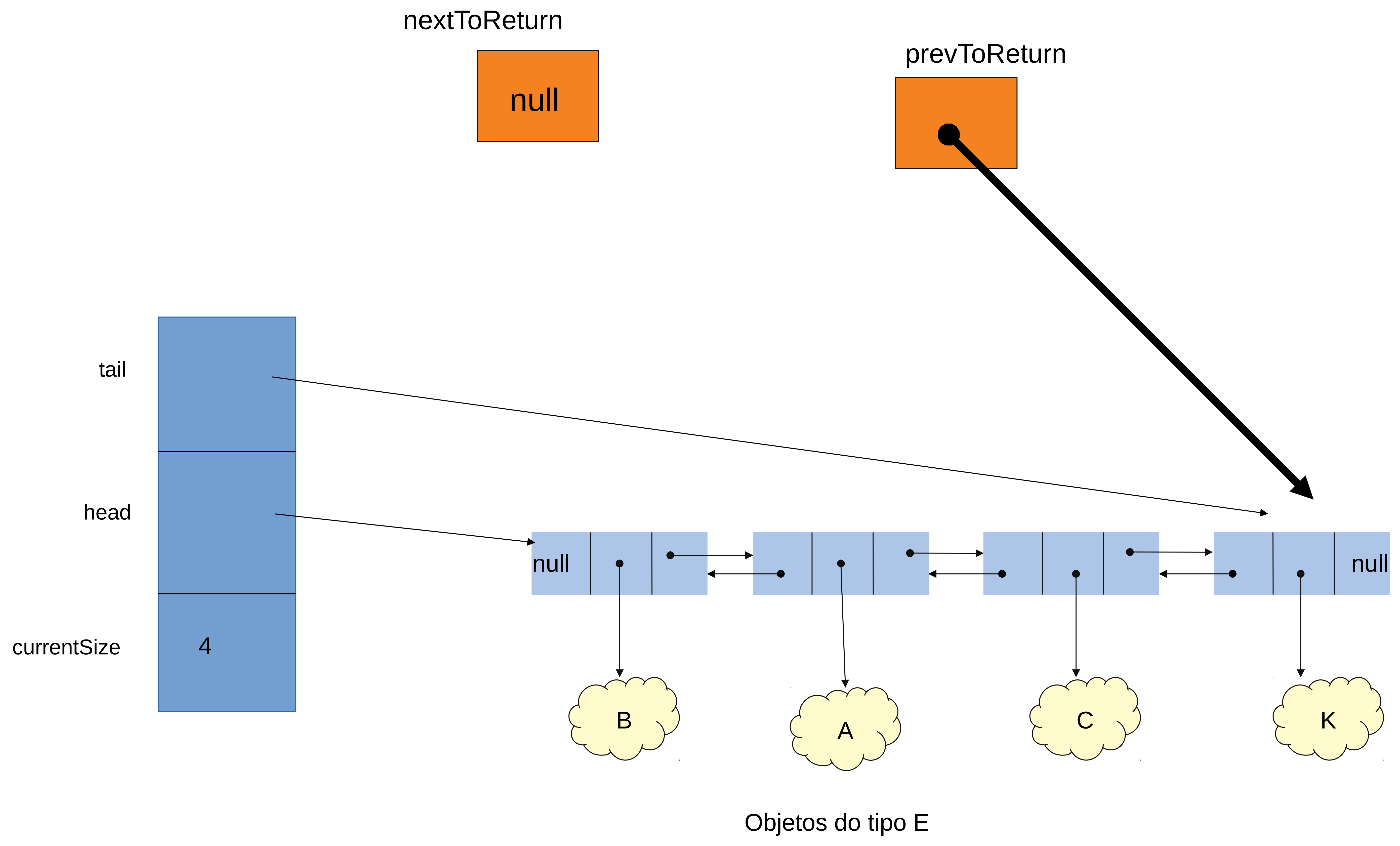
B; A; B



Como funciona o iterador (12)

rewind()	
next()	
hasNext()	
fullforward()	
previous()	
hasPrevious()	

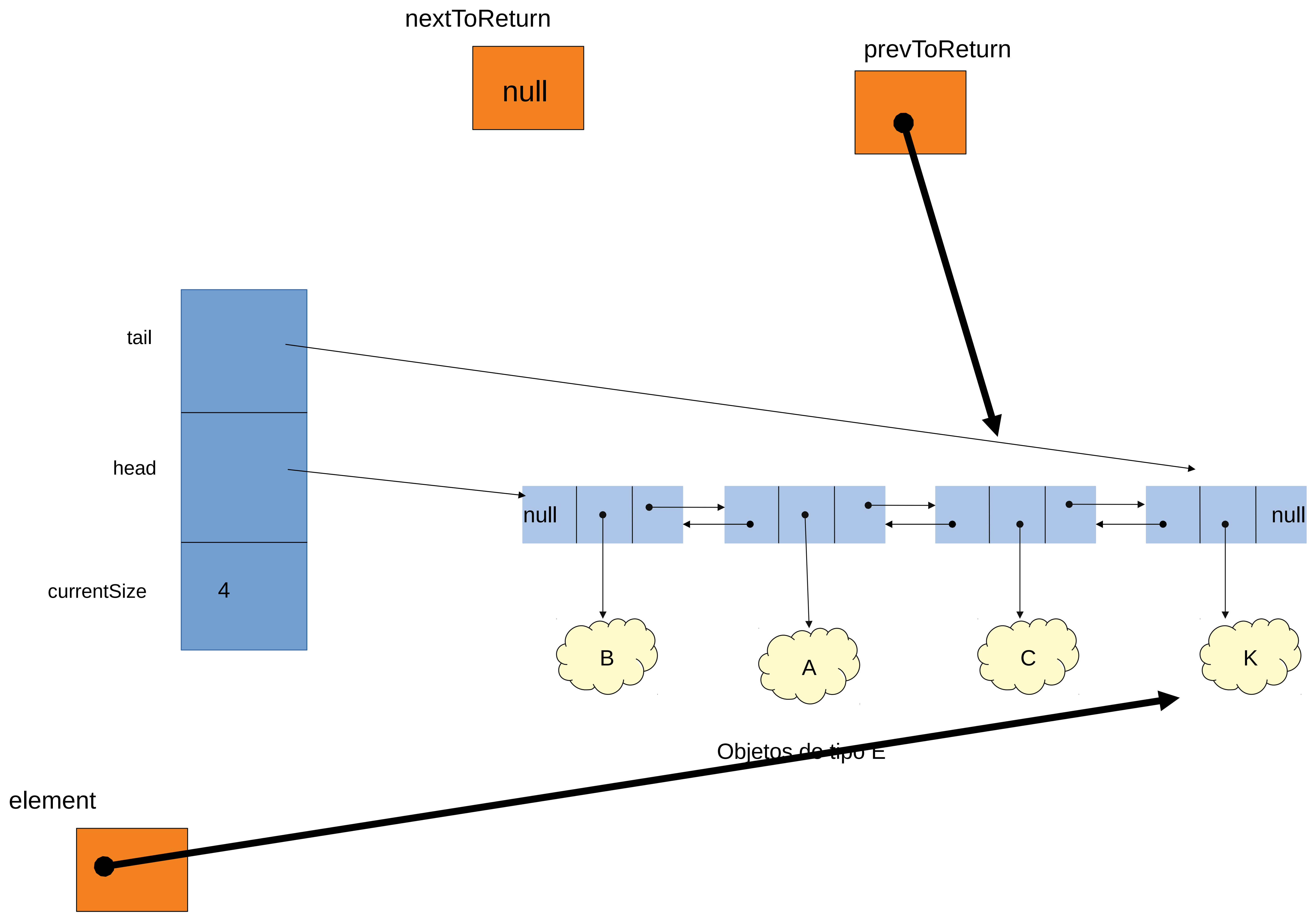
B; A; B



Como funciona o iterador (13)

rewind()	
next()	
hasNext()	
fullforward()	
previous()	K
hasPrevious()	

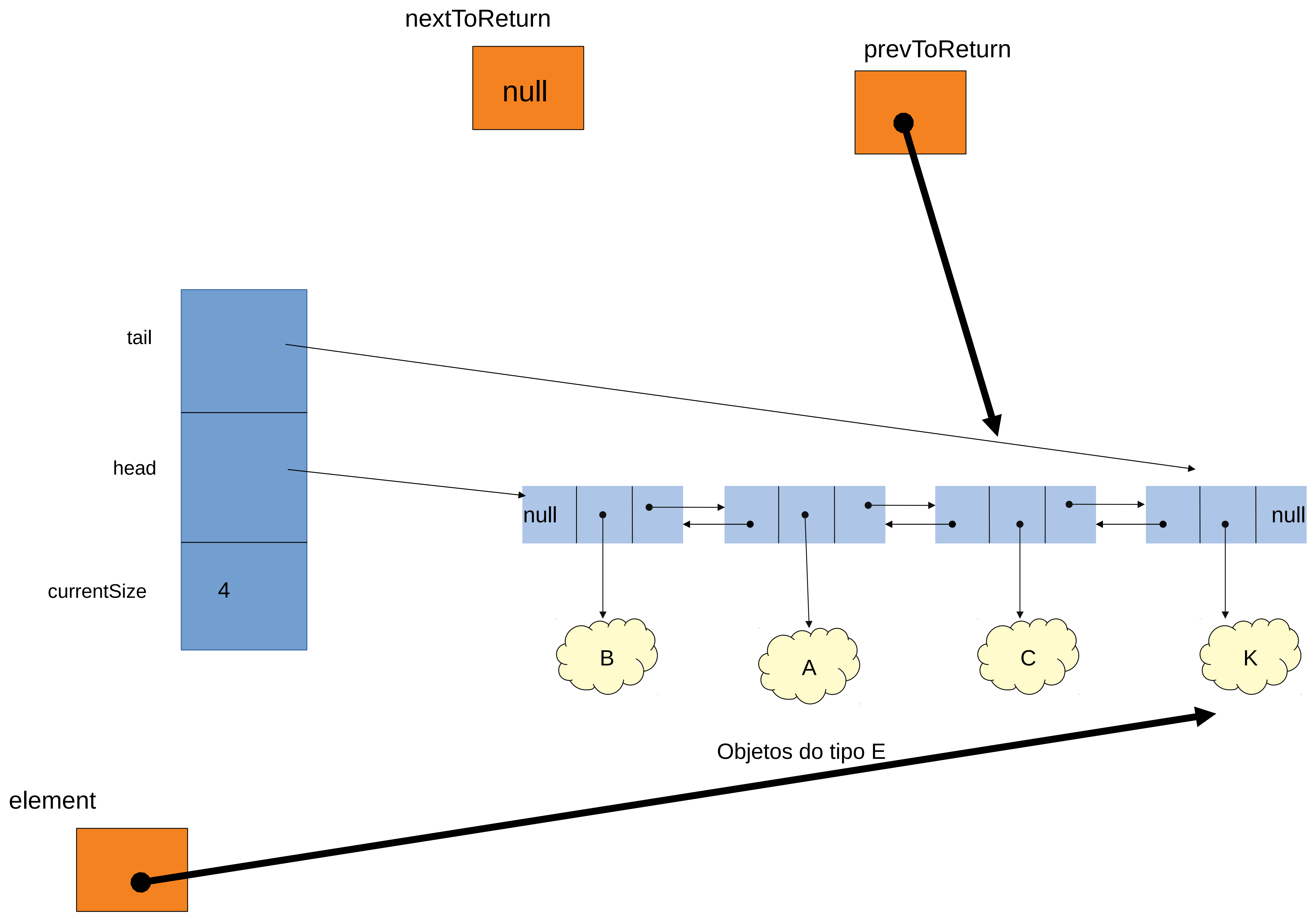
B; A; B; K



Como funciona o iterador (14)

rewind()	
next()	
hasNext()	false
fullforward()	
previous()	
hasPrevious()	true

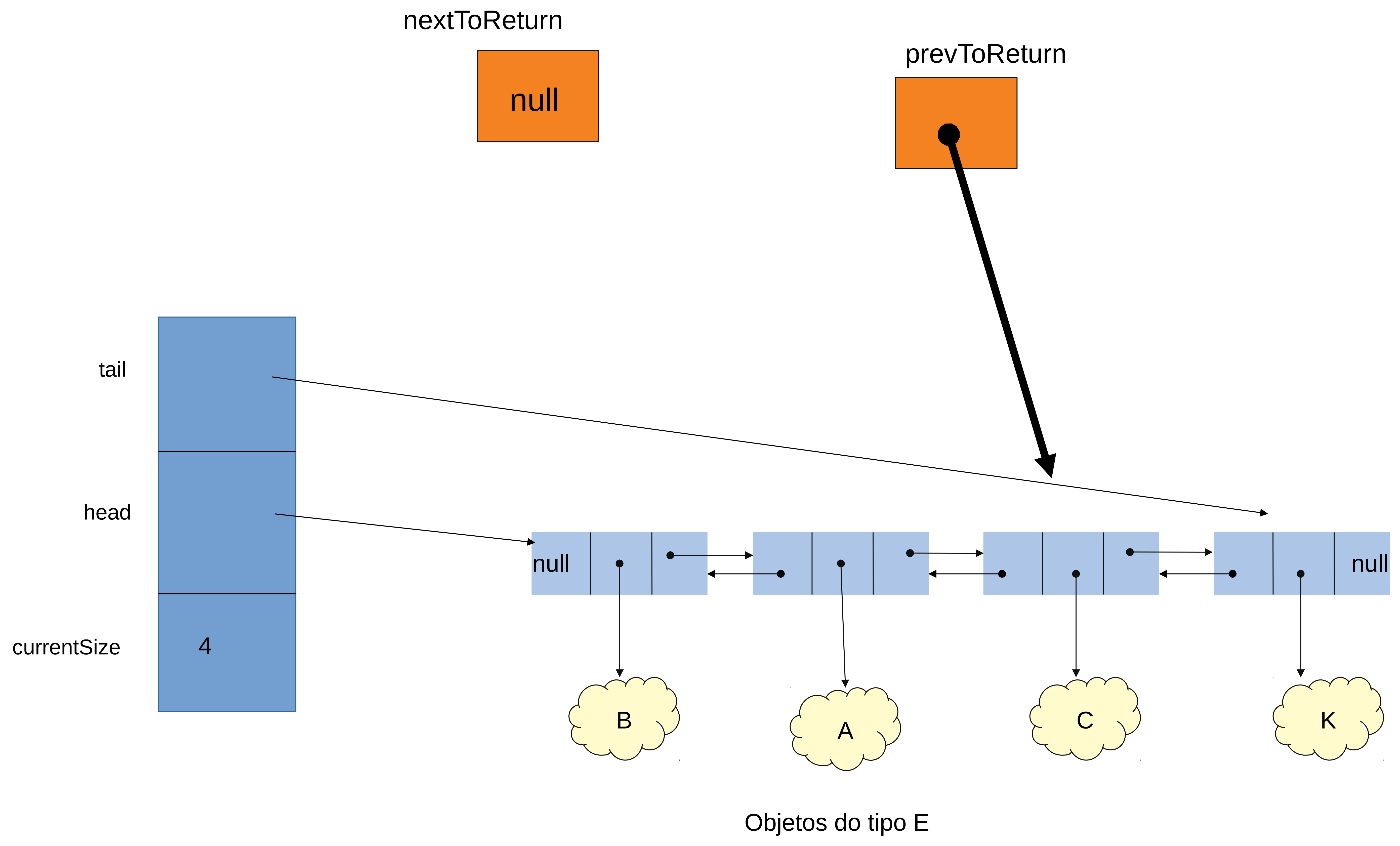
B; A; B; K



Como funciona o iterador (15)

rewind()	
next()	
hasNext()	
fullforward()	
previous()	
hasPrevious()	

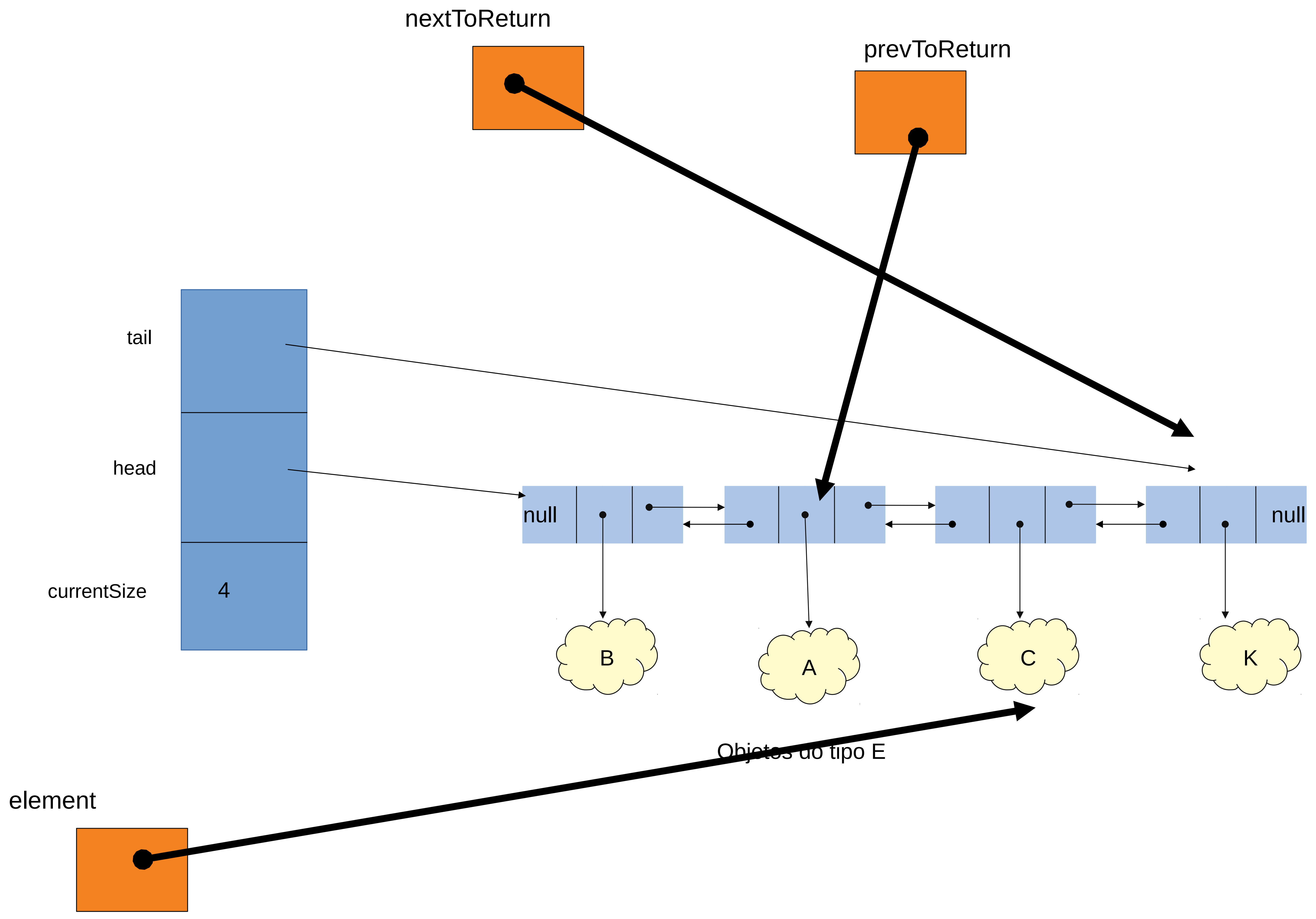
B; A; B; K



Como funciona o iterador (16)

rewind()	
next()	
hasNext()	
fullforward()	
previous()	C
hasPrevious()	

B; A; B; K; C



Lista Duplamente Ligada

Operação	Melhor Caso	Pior Caso	Caso Esperado
isEmpty, size			
getFirst, getLast			
get			
addFirst, addLast			
add			
removeFirst, removeLast			
remove			
indexOf (por elemento)			
iterator, twoWayIterator			

A complexidade espacial da lista duplamente ligada é