Algoritmos e Estruturas de Dados

TAD List
Implementações com estruturas de dados dinâmicas

LEI - Licenciatura em Eng. Informática

2025/26

FACULDADE DE
CIÊNCIAS E TECNOLOGIA

Uma sequência
TAD List

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

- Representa uma sequência de elementos a que podemos aceder através:
  - da posição de um elemento;
  - de um elemento igual.

- As operações incluem:
  - aceder a um elemento por posição;
  - inserir um elemento numa posição;
  - remover o elemento numa posição;
  - devolver a posição de um dado elemento..

TAD List (1)

```
package dataStructures;
import dataStructures.exceptions.*;
import java.io.Serializable;
/**
 * List (sequence) Abstract Data Type
 * Includes description of general methods to be implemented by lists.
 * @author AED  Team
 * @version 1.0
 * @param <E> Generic Element
 */
public interface List<E> extends Serializable {
    int NOT_FOUND=-1;
    /**
     * Returns true iff the list contains no elements.
     * @return true if list is empty
     */
    boolean isEmpty();
    /**
     * Returns the number of elements in the list.
     * @return number of elements in the list
     */
    int size();
    /**
     * Returns an iterator of the elements in the list (in proper sequence).
     * @return Iterator of the elements in the list
     */
    Iterator<E> iterator();
```

TAD List (2)

```
/**
 * Returns the first element of the list.
 * @return first element in the list
 * @throws NoSuchElementException - if size() == 0
 */
E getFirst();

/**
 * Returns the last element of the list.
 * @return last element in the list
 * @throws NoSuchElementException - if size() == 0
 */
E getLast();

/**
 * 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
 */
E get(int position);
```

TAD List (3)

```
/**
 * Inserts the specified element at the first position in the list.
 *
 * @param element to be inserted
 */
void addFirst(E element);

/**
 * Inserts the specified element at the last position in the list.
 *
 * @param element to be inserted
 */
void addLast(E element);

/**
 * 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);
```

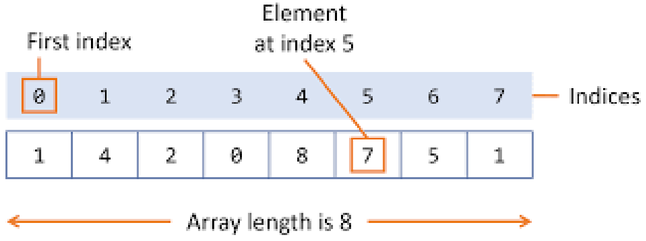
TAD List (4)

```
/**
 * 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
 */
E removeFirst();

/**
 * 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
 */
E removeLast();

/**
 * 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
 */
E remove(int position);
```

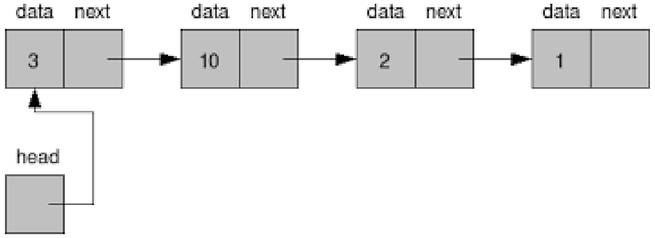
TAD List (5)

```
/**
    * Returns the position of the first occurrence of the specified element
    * in the list, if the list contains the element.
    * Otherwise, returns NOT_FOUND.
    *
    * @param element - element to be searched in list
    * @return position of the first occurrence of the element in the list (or -1)
    */
   int indexOf(E element);
}
```

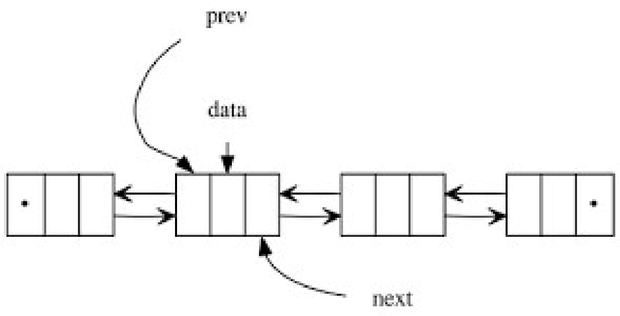
Possíveis estruturas de dados



Vetores



Lista simplesmente ligada



Lista duplamente ligada

Estruturas de Dados

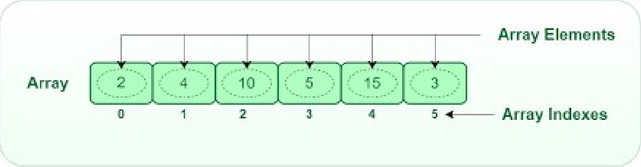
Estrutura de Dados : é uma forma concreta de organizar informação na memória dum computador.

Algumas estruturas de dados:

- **Vetores**: estruturas de dados estáticas (capacidade definida no momento da criação – memória continua para guardar os elementos).
- **Listas ligadas**: estruturas de dados dinâmicas (sem capacidade pré-definida – alocação de memória quando necessária).

Classe ListInArray<E>

```java
/**
 * List in Array
 * @author AED  Team
 * @version 1.0
 * @param <E> Generic Element
 *
 */
public class ListInArray<E> implements List<E> {

    private static final int FACTOR = 2;
    /**
     * Array of generic elements E.
     */
    private E[] elems;

    /**
     * Number of elements in array.
     */
    private int counter;
```



Implementação já realizada na cadeira de POO

```java
/**
 * Construtor with capacity.
 * @param dimension - initial capacity of array.
 */
@SuppressWarnings("unchecked")
public ListInArray(int dimension) {
    elems = (E[]) new Object[dimension];
    counter = 0;
}
```

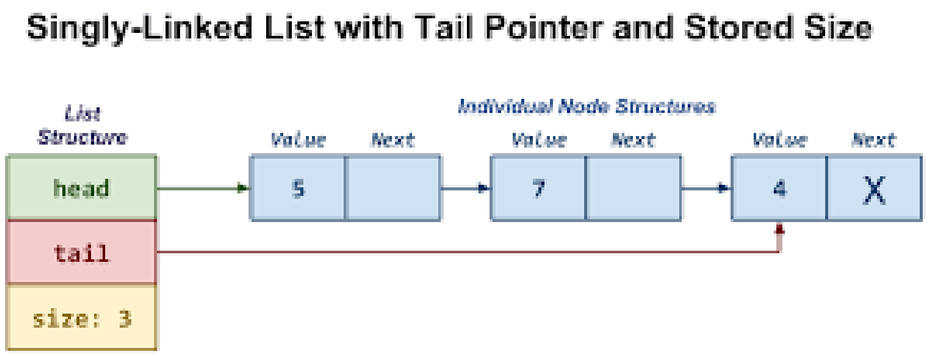
Lista em Vetor

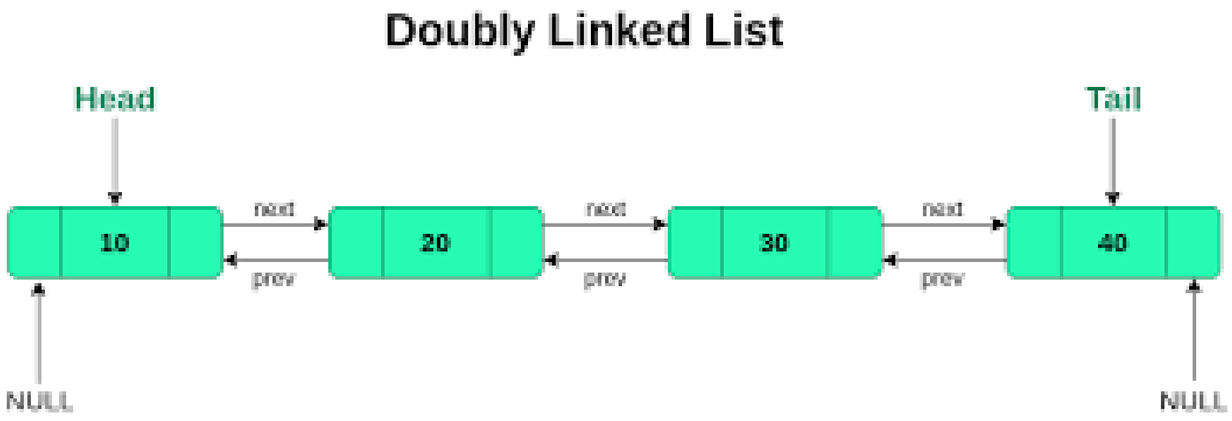
Operação	Melhor Caso	Pior Caso	Caso Médio
isEmpty, size			
getFirst, getLast			
get			
addFirst			
addLast, add			
removeLast			
removeFirst			
remove			
indexOf (por elemento)			
iterator			

Lista em Vetor

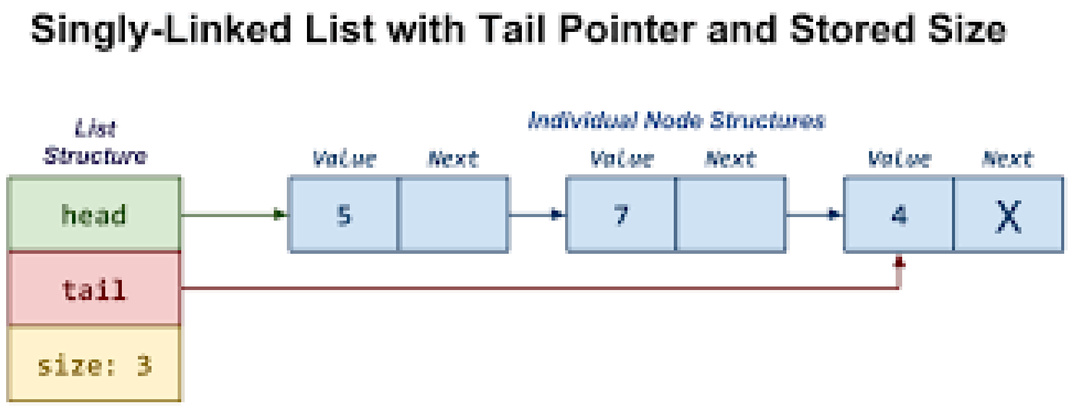
Operação	Melhor Caso	Pior Caso	Caso Médio
`isEmpty, size`	$O(1)$	$O(1)$	$O(1)$
`getFirst, getLast`	$O(1)$	$O(1)$	$O(1)$
`get`	$O(1)$	$O(1)$	$O(1)$
`addFirst`	$O(n)$	$O(n)$	$O(n)$
`removelast`	$O(1)$	$O(1)$	$O(1)$
`addLast, add`	$O(1)$	$O(n)$	$O(n)$
`removeFirst`	$O(n)$	$O(n)$	$O(n)$
`remove`	$O(1)$	$O(n)$	$O(n)$
`indexOf (por elemento)`	$O(1)$	$O(n)$	$O(n)$
`iterator`	$O(1)$	$O(1)$	$O(1)$

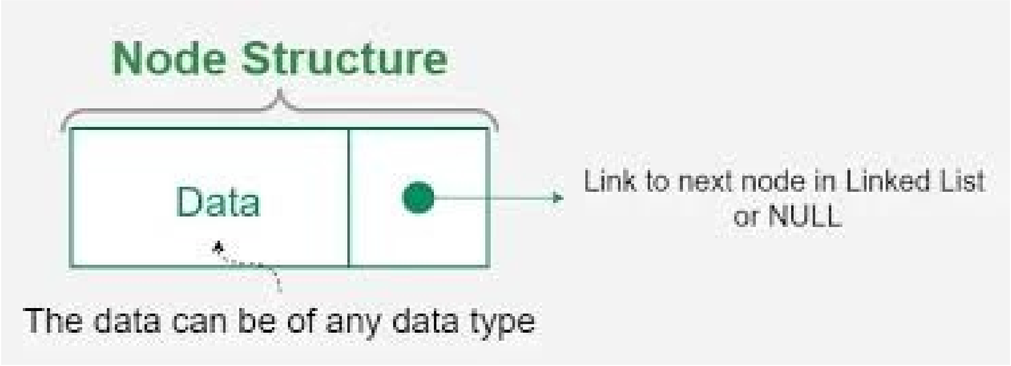
Lista Ligada





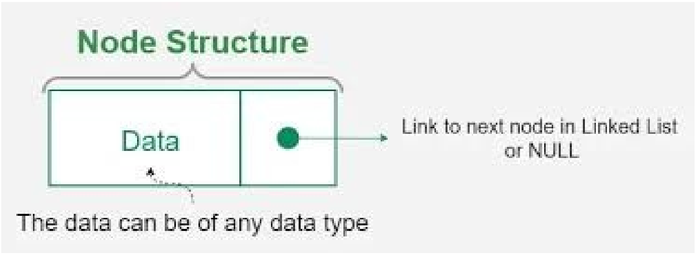
Lista Simplesmente Ligada

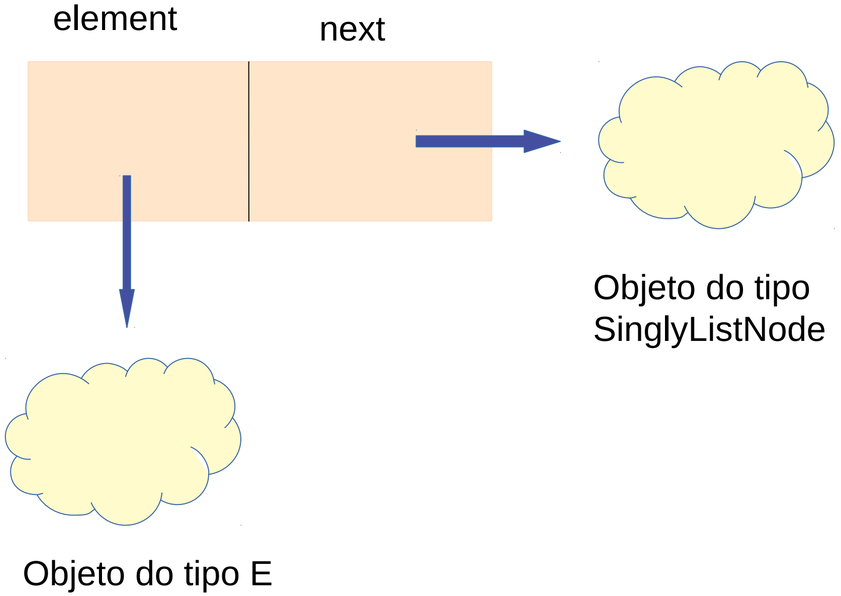




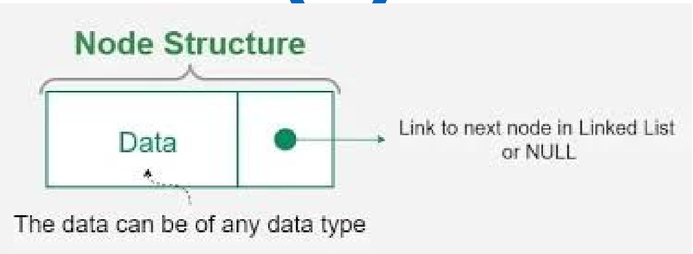
Classe SinglyListNode<E> (1)

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

class SinglyListNode<E> implements Serializable {
    /**
     * Element stored in the node.
     */
    private E element;
    /**
     * (Pointer to) the next node.
     */
    private SinglyListNode<E> next;
    /**
     *
     * @param theElement - The element to be contained in the node
     * @param theNext - the next node
     */
    public SinglyListNode( E theElement, SinglyListNode<E> theNext ){
        element = theElement;
        next = theNext;
    }
    /**
     *
     * @param theElement to be contained in the node
     */
    public SinglyListNode( E theElement ) {
        this(theElement, null);
    }
```

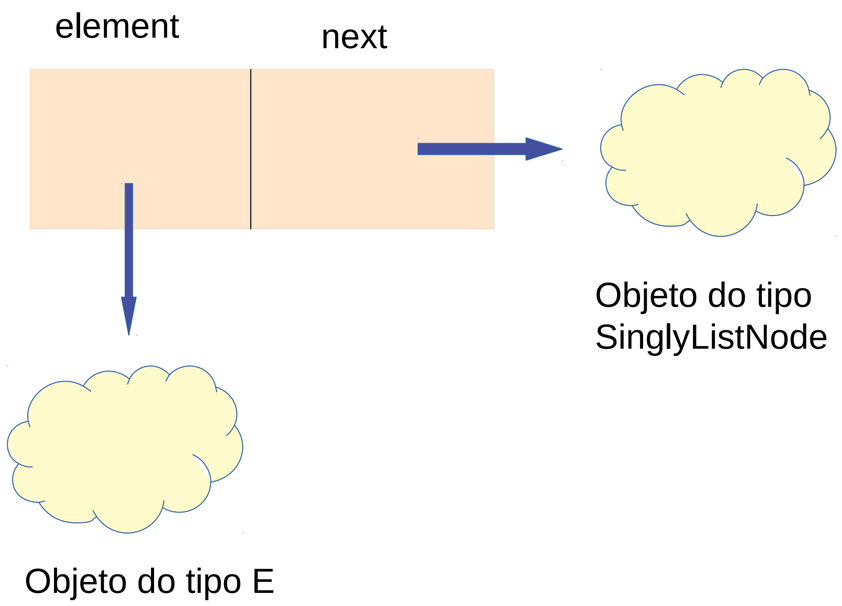




Classe SinglyListNode<E> (2)



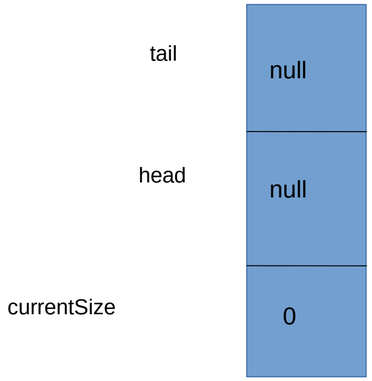

```
    /**
     *
     * @return the element contained in the node
     */
    public E getElement( ) {
        return element;
    }
    /**
     *
     * @return the next node
     */
    public SinglyListNode<E> getNext( ) {
        return next;
    }
    /**
     *
     * @param newElement - New element to replace the current element
     */
    public void setElement( E newElement ) {
        element = newElement;
    }
    /**
     *
     * @param newNext - node to replace the next node
     */
    public void setNext( SinglyListNode<E> newNext ) {
        next = newNext;
    }
}
```



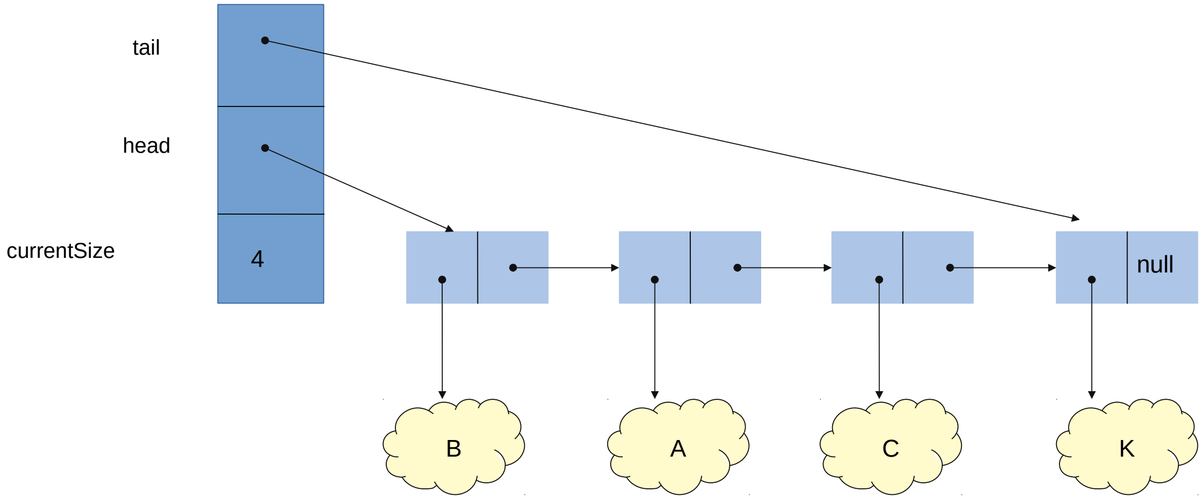
Classe SinglyListNode

Operação	Melhor Caso	Pior Caso	Caso Médio
`getElement`	O(1)	O(1)	O(1)
`getNext`	O(1)	O(1)	O(1)
`setElement`	O(1)	O(1)	O(1)
`setNext`	O(1)	O(1)	O(1)

Lista Simplesmente Ligada



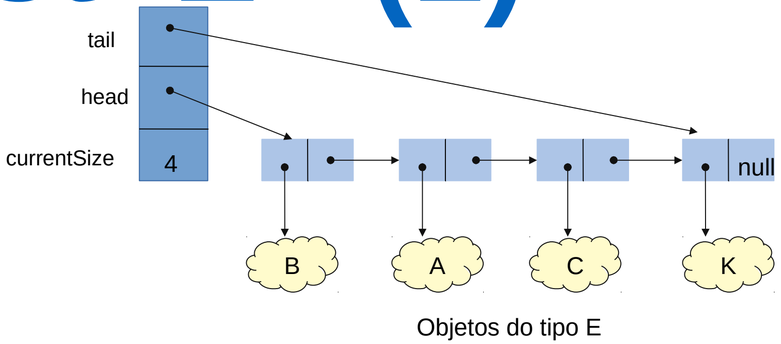
Lista vazia – zero elementos



Objetos do tipo E

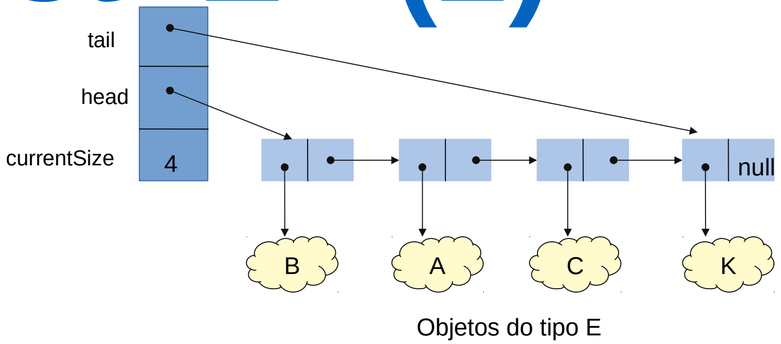
Lista com 4 elementos

Classe SinglyLinkedList<E> (1)




```
package dataStructures;

import dataStructures.exceptions.*;

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

Classe SinglyLinkedList<E> (2)

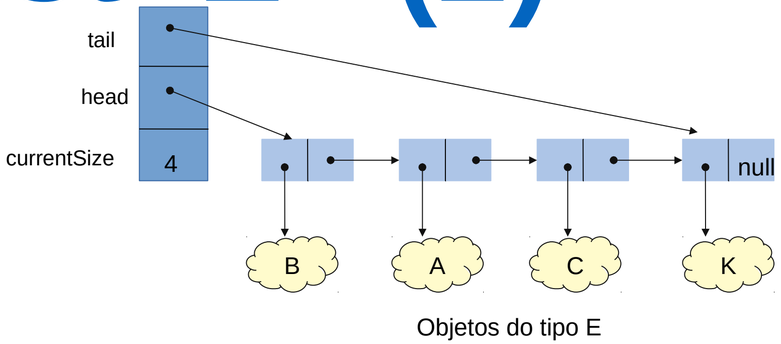


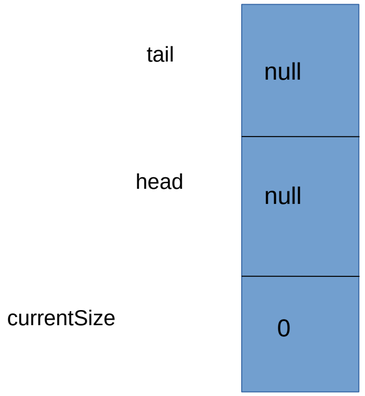

```
/**
 * Constructor of an empty singly linked list.
 * head and tail are initialized as null.
 * currentSize is initialized as 0.
 */
public SinglyLinkedList( ) {



}
```

Lista vazia – zero elementos

Classe SinglyLinkedList<E> (2)

```
/**
 * Constructor of an empty singly linked list.
 * head and tail are initialized as null.
 * currentSize is initialized as 0.
 */
public SinglyLinkedList( ) {
    head = null;
    tail = null;
    currentSize = 0;
}
/**
 * Returns true iff the list contains no elements.
 * @return true if list is empty
 */
public boolean isEmpty() {

}
```

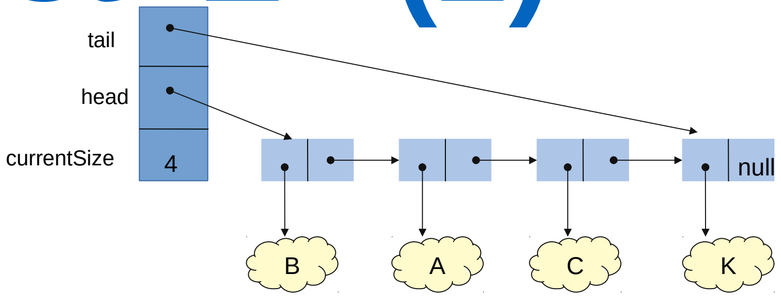




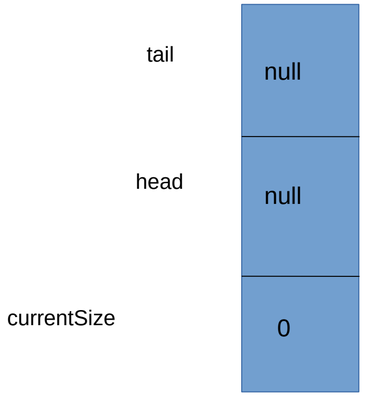
Lista vazia – zero elementos

Classe SinglyLinkedList<E> (2)

```
/**
 * Constructor of an empty singly linked list.
 * head and tail are initialized as null.
 * currentSize is initialized as 0.
 */
public SinglyLinkedList( ) {
    head = null;
    tail = null;
    currentSize = 0;
}
/**
 * Returns true iff the list contains no elements.
 * @return true if list is empty
 */
public boolean isEmpty() {
    return currentSize==0;
}
/**
 * Returns the number of elements in the list.
 * @return number of elements in the list
 */

public int size() {

}
```



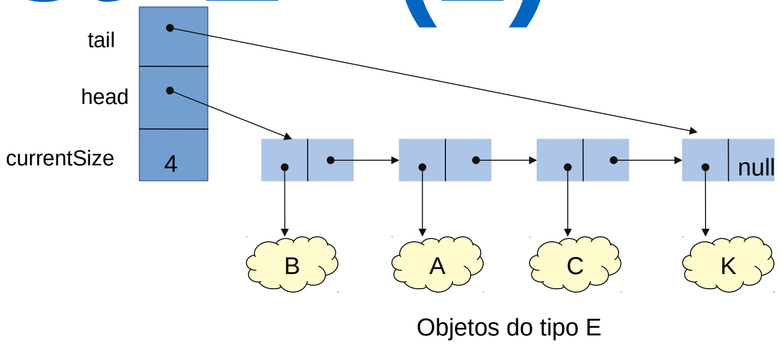
Objetos do tipo E

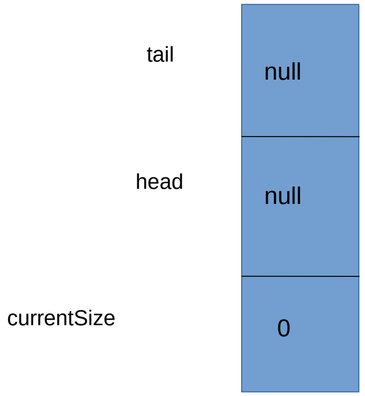


Lista vazia – zero elementos

Classe SinglyLinkedList<E> (2)

```
/**
 * Constructor of an empty singly linked list.
 * head and tail are initialized as null.
 * currentSize is initialized as 0.
 */
public SinglyLinkedList( ) {
    head = null;
    tail = null;
    currentSize = 0;
}
/**
 * Returns true iff the list contains no elements.
 * @return true if list is empty
 */
public boolean isEmpty() {
    return currentSize==0;
}
/**
 * Returns the number of elements in the list.
 * @return number of elements in the list
 */

public int size() {
    return currentSize;
}
```



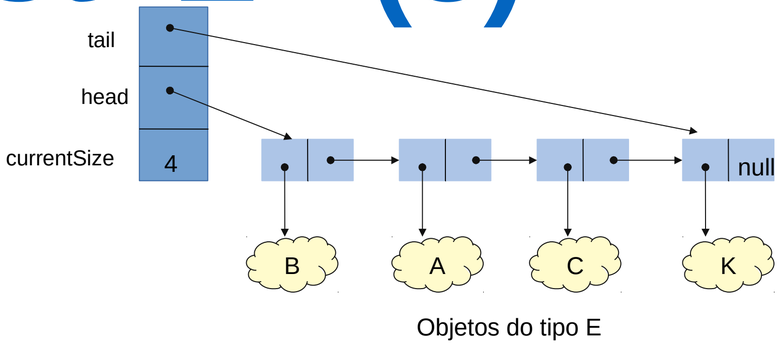


Lista vazia – zero elementos

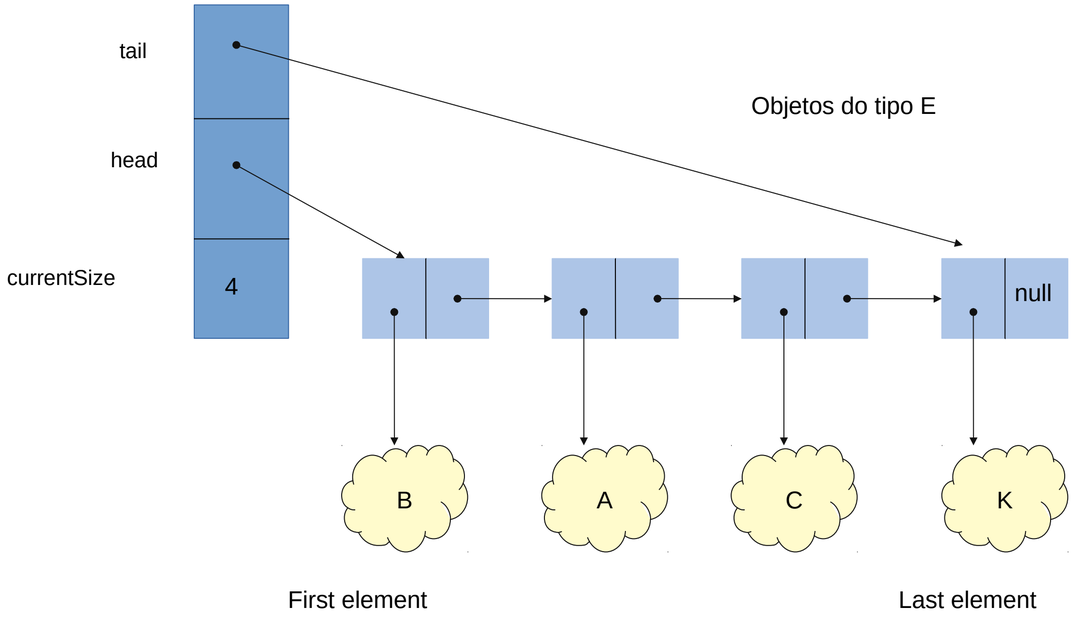
Lista Simplesmente Ligada

Operação	Melhor Caso	Pior Caso	Caso Médio
`isEmpty, size`	$O(1)$	$O(1)$	$O(1)$
`getFirst, getLast`			
`get`			
`addFirst, addLast`			
`add`			
`removeFirst`			
`removeLast`			
`remove`			
`indexOf (por elemento)`			
`iterator`			

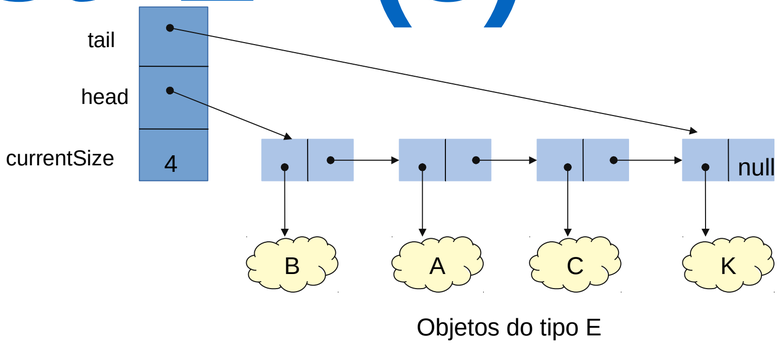
Classe SinglyLinkedList<E> (3)



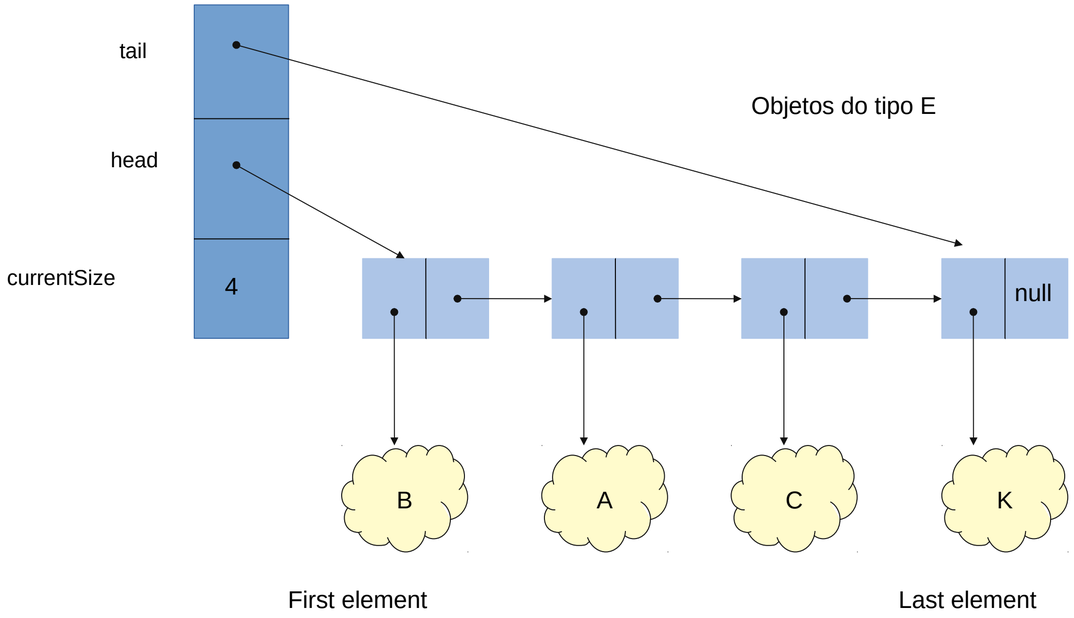

```
/**
 * Returns the first element of the list.
 *
 * @return first element in the list
 * @throws NoSuchElementException - if size() == 0
 */
public E getFirst() {


}
```



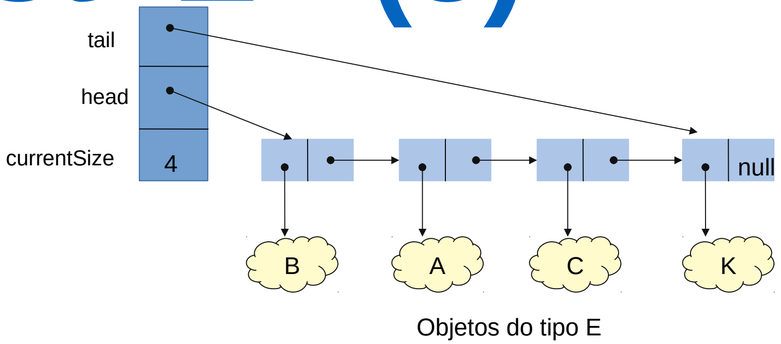
Classe SinglyLinkedList<E> (3)



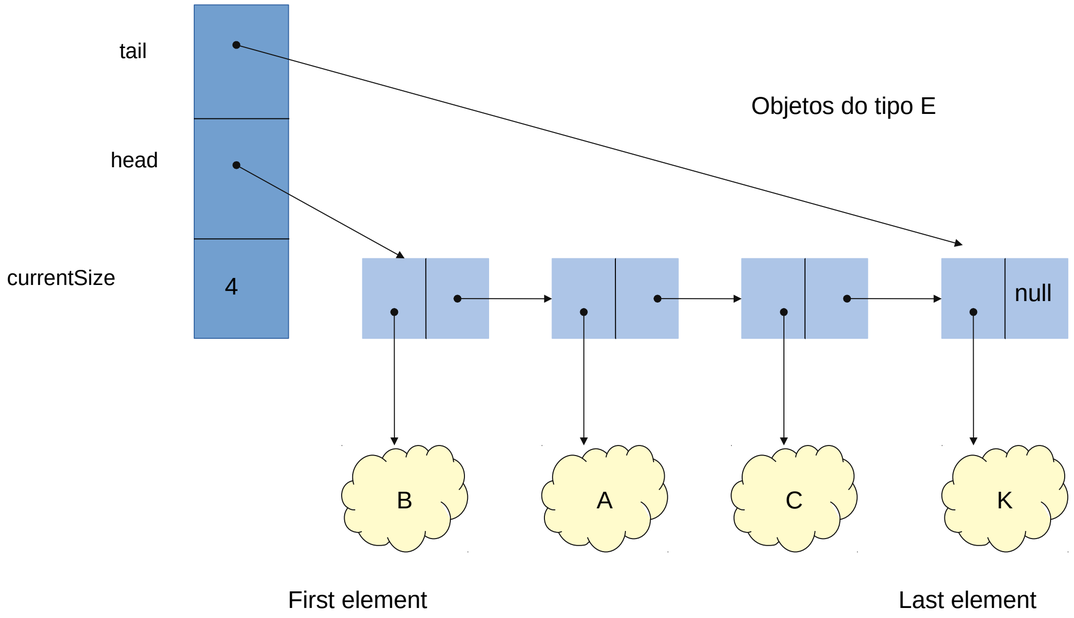

```
/**
 * Returns the first element of the list.
 *
 * @return first element in the list
 * @throws NoSuchElementException - if size() == 0
 */
public E getFirst() {
    if ( this.isEmpty() )
        throw new NoSuchElementException();
    return head.getElement();
}

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



}
```



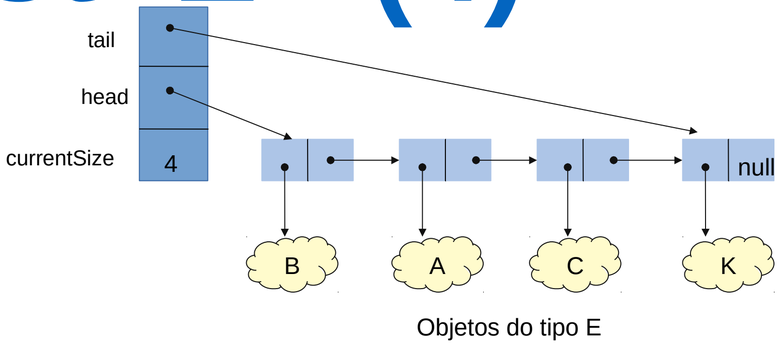
Classe SinglyLinkedList<E> (3)



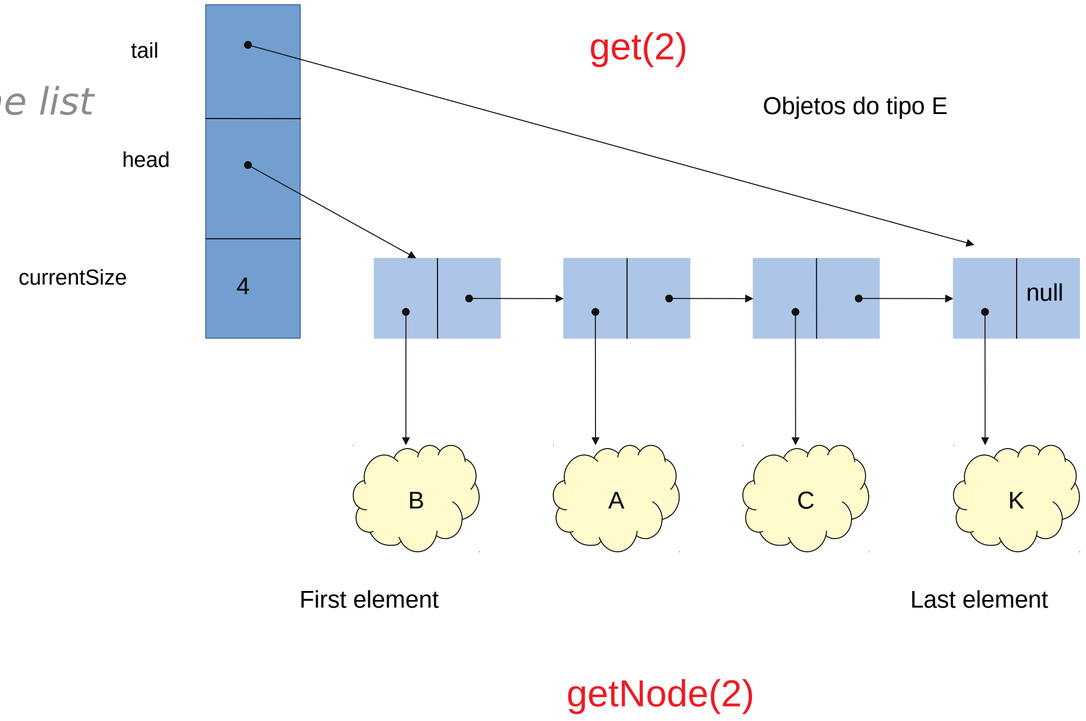

```
/**
 * Returns the first element of the list.
 *
 * @return first element in the list
 * @throws NoSuchElementException - if size() == 0
 */
public E getFirst() {
    if ( this.isEmpty() )
        throw new NoSuchElementException();
    return head.getElement();
}

/**
 * Returns the last element of the list.
 *
 * @return last element in the list
 * @throws NoSuchElementException - if size() == 0
 */
public E getLast() {
    if ( this.isEmpty() )
        throw new NoSuchElementException();
    return tail.getElement();
}
```



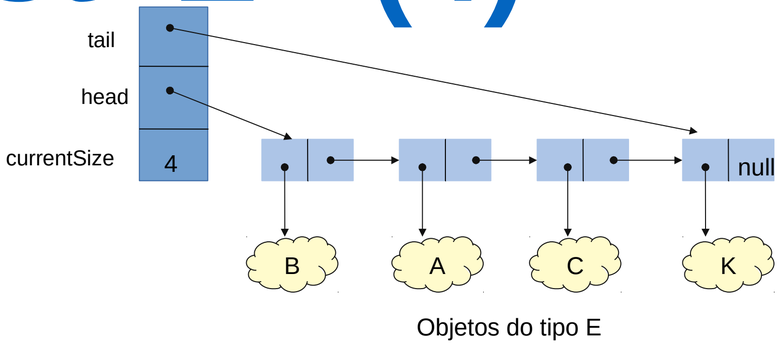
Classe SinglyLinkedList<E> (4)



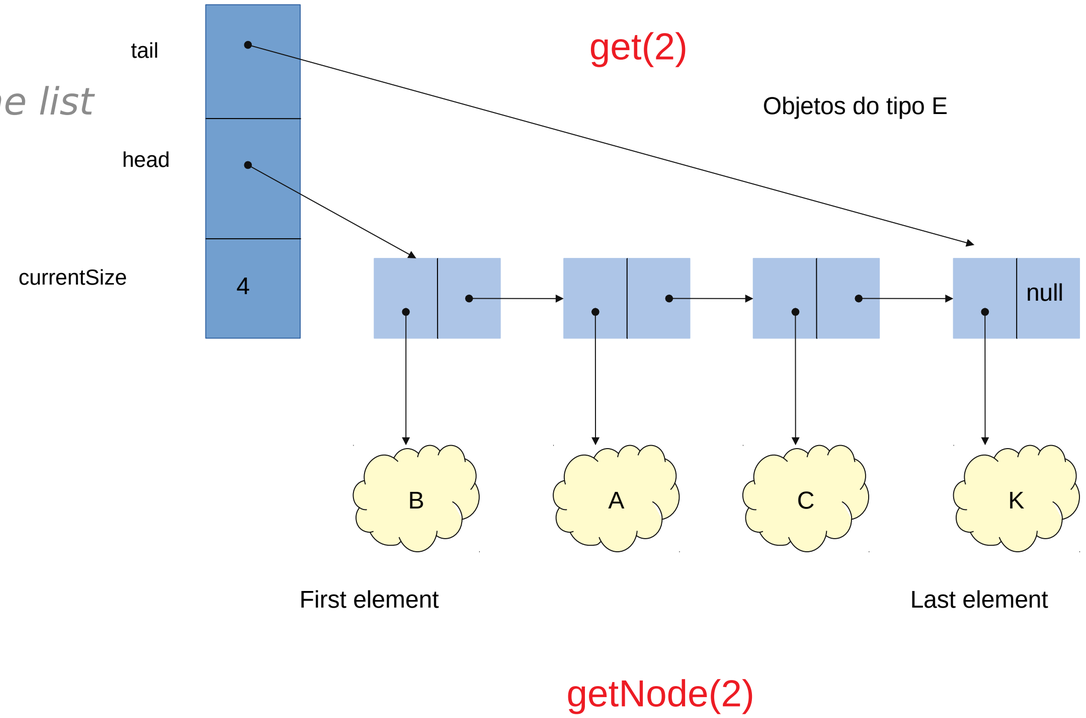

```
/**
 * Returns the element at the specified position in the list.
 * Range of valid positions: 0, ..., size()-1.
 * If the specified position is 0, get corresponds to getFirst.
 * If the specified position is size()-1, get corresponds to getLast.
 *
 * @param position - position of element to be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

public E get(int position) {

 }
```



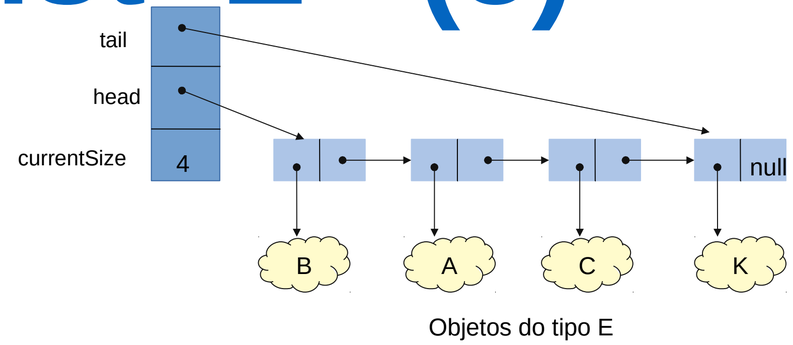
Classe SinglyLinkedList<E> (4)




```
/**
 * 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) {

    if ( position < 0 || position >= currentSize )
        throw new InvalidPositionException();
    if (position == 0)
        return getFirst();
    if (position == currentSize-1)
        return getLast();

    return getNode(position).getElement();
 }
```



Classe SinglyLinkedList<E> (5)

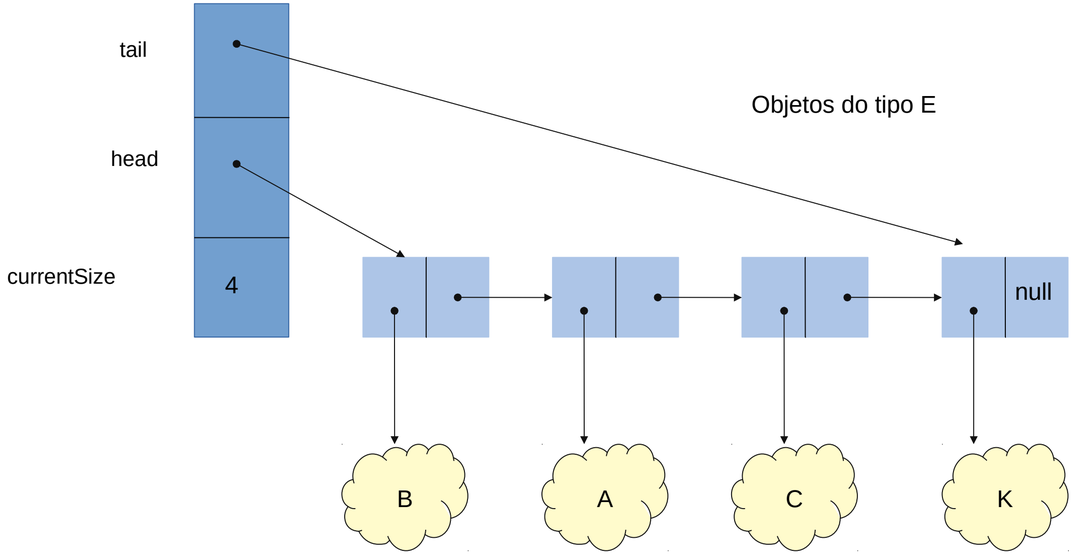


getNode(2)

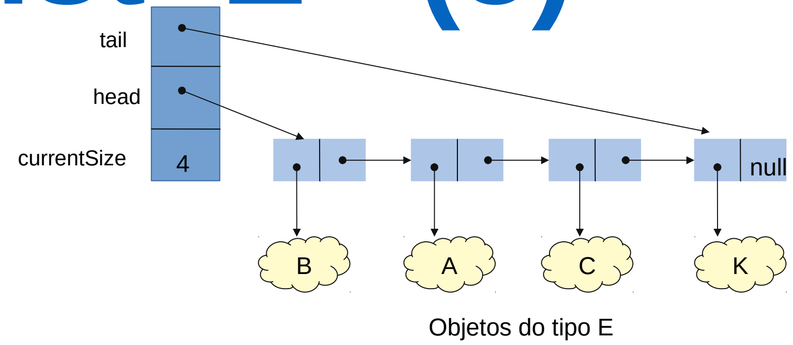
```
private SinglyListNode<E> getNode(int position) {

    return node;
}
```

Percurso em Lista

Início →

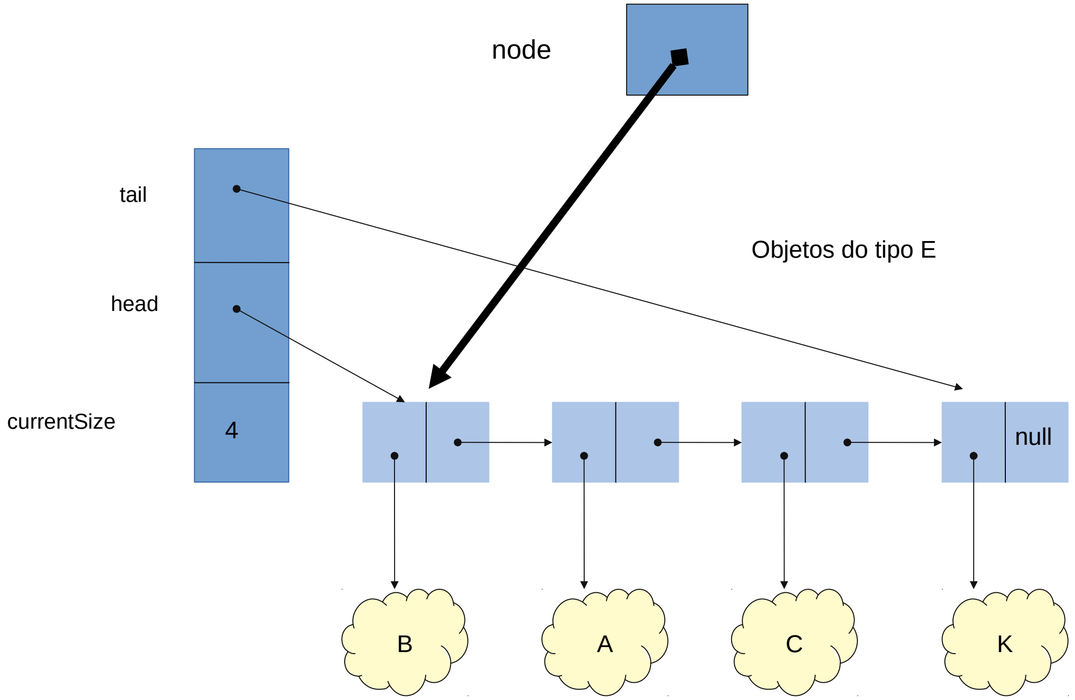


Classe SinglyLinkedList<E> (5)



getNode(2)

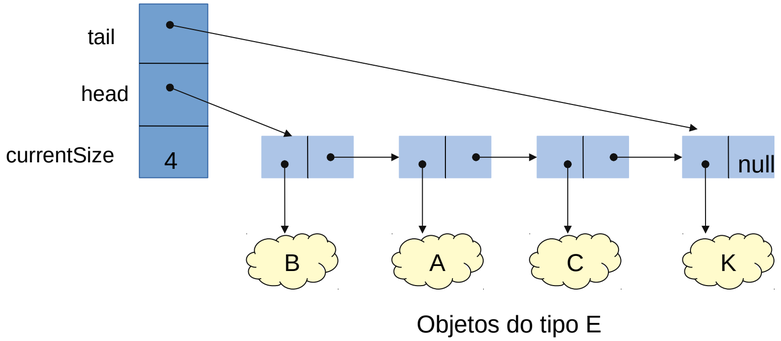
```
private SinglyListNode<E> getNode(int position) {
    SinglyListNode<E> node = head;

    return node;
}
```



Percurso em Lista

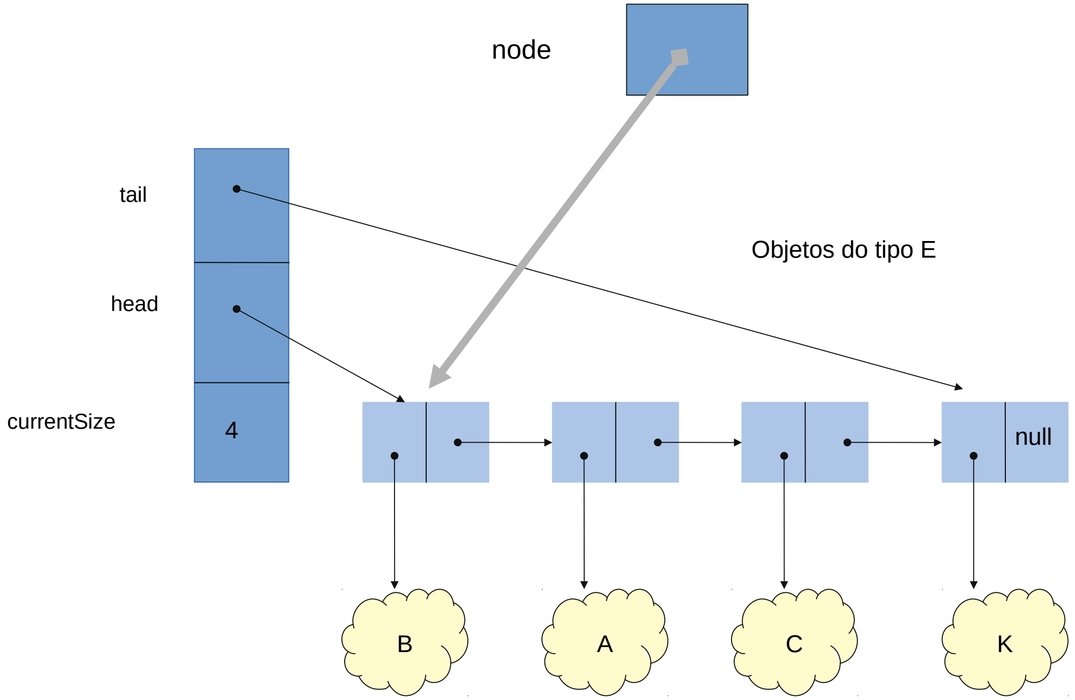
Início → singlyListNode<E> node=head;

Classe SinglyLinkedList<E> (6)



getNode(2)

```
private SinglyListNode<E> getNode(int position) {
   SinglyListNode<E> node = head;

   for ( int i = 0;          ; i++)


      return node;
   }
```

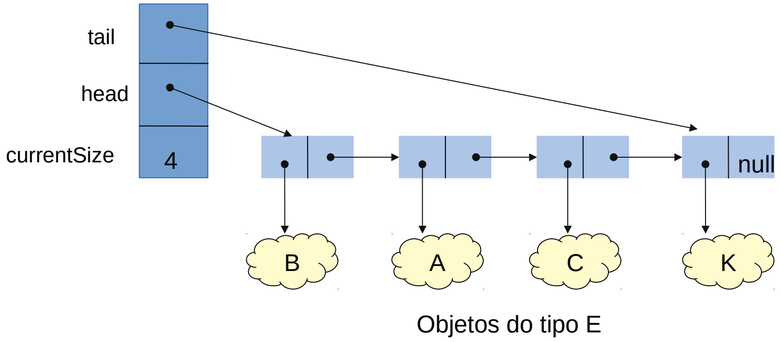


Percurso em Lista

Início → singlyListNode<E> node=head;

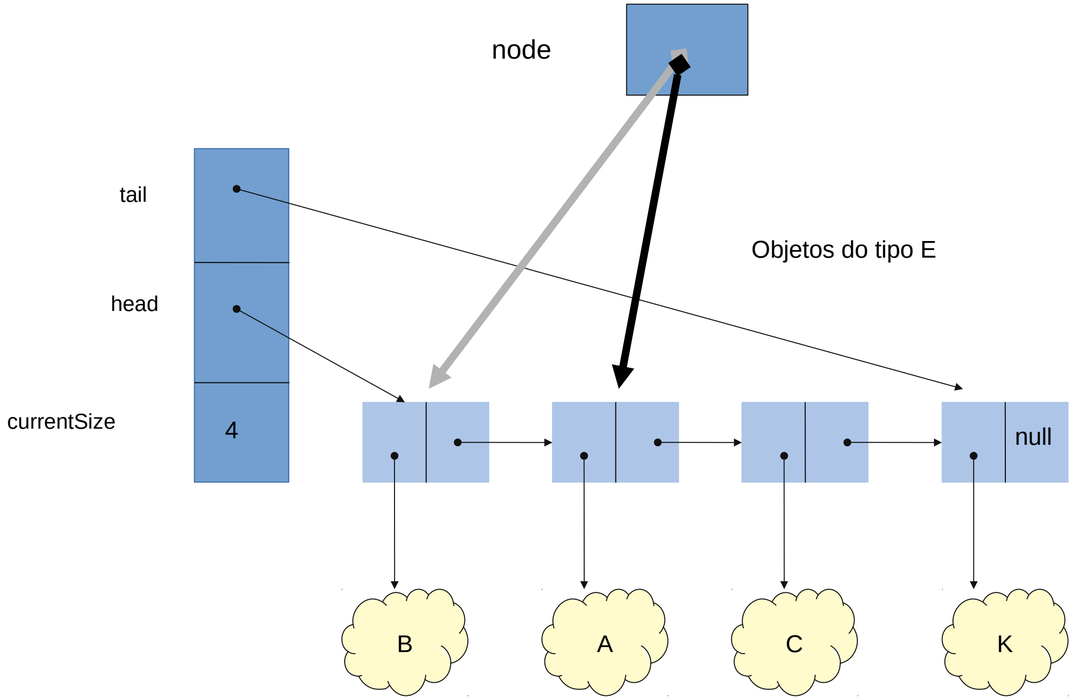
Avanço →

Classe SinglyLinkedList<E> (6)



getNode(2)

```
private SinglyListNode<E> getNode(int position) {
   SinglyListNode<E> node = head;

   for ( int i = 0;          ; i++)

        node = node.getNext();

        return node;
    }
```

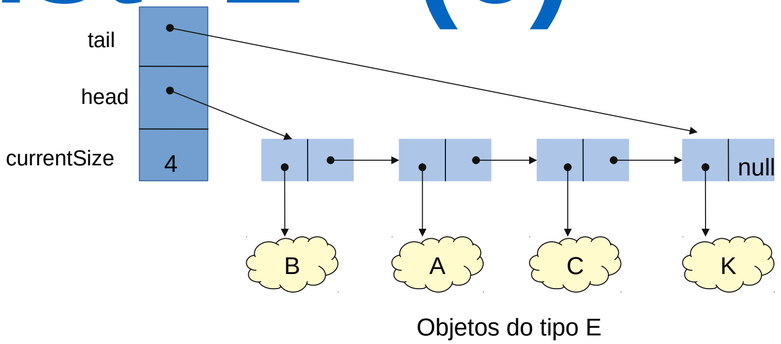


Percurso em Lista

Início → singlyListNode<E> node=head;

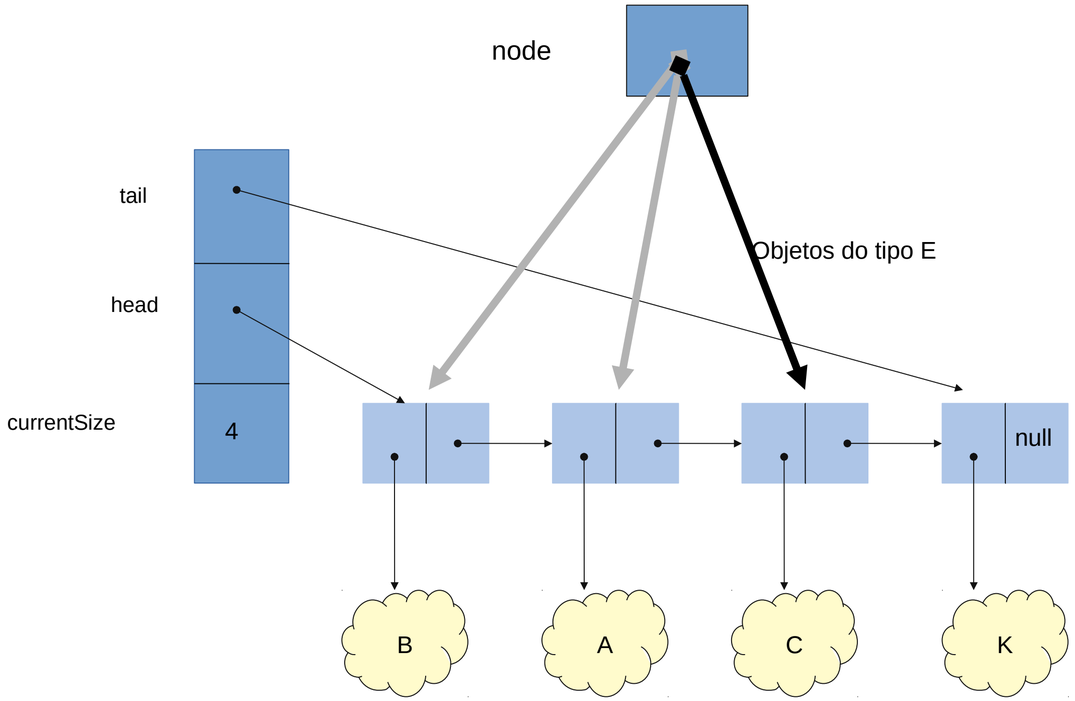
Avanço → node = node.getNext();

Classe SinglyLinkedList<E> (6)



getNode(2)

```
private SinglyListNode<E> getNode(int position) {
   SinglyListNode<E> node = head;

   for ( int i = 0;          ; i++)

        node = node.getNext();

        return node;
     }
```

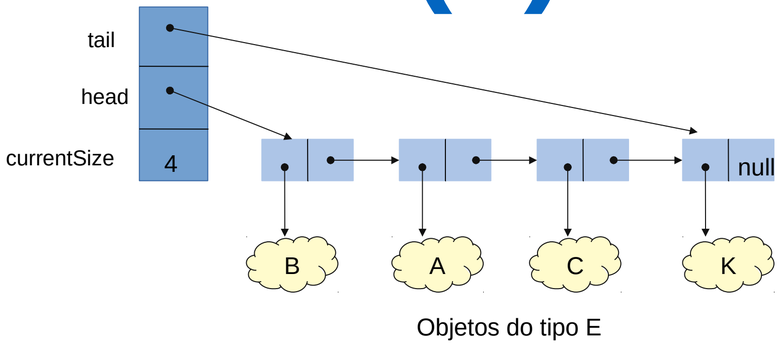


Percurso em Lista

Início → singlyListNode<E> node=head;

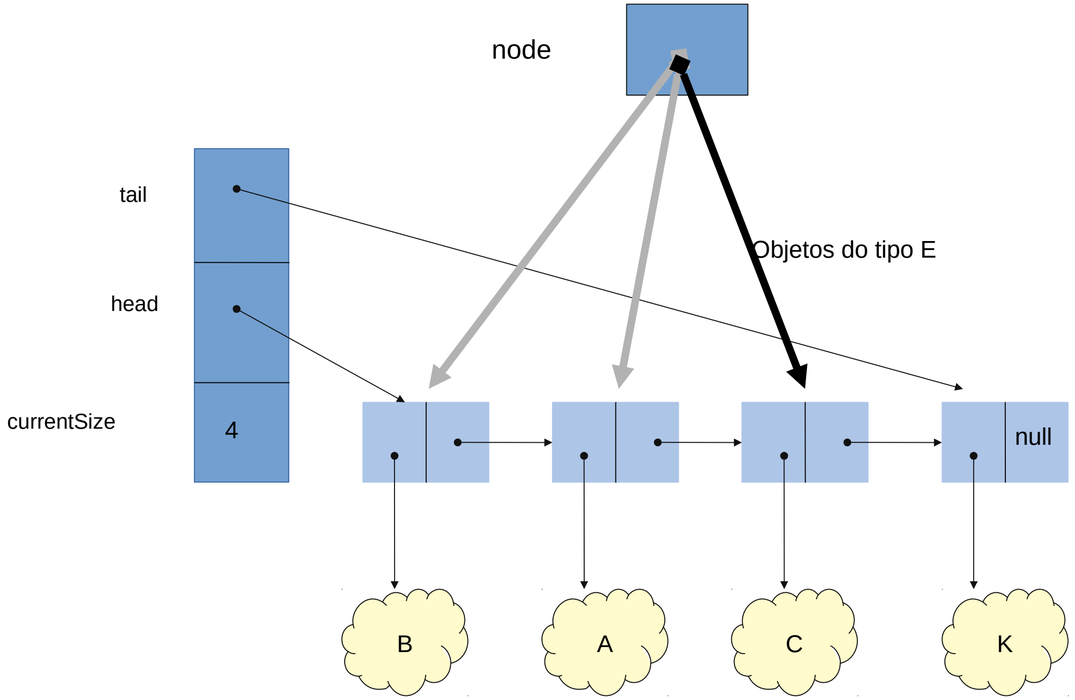
Avanço → node = node.getNext();

Classe SinglyLinkedList<E> (7)



getNode(2)

```
private SinglyListNode<E> getNode(int position) {
   SinglyListNode<E> node = head;

   for ( int i = 0; i < position; i++)
      node = node.getNext();

   return node;
}
```



Percurso em Lista

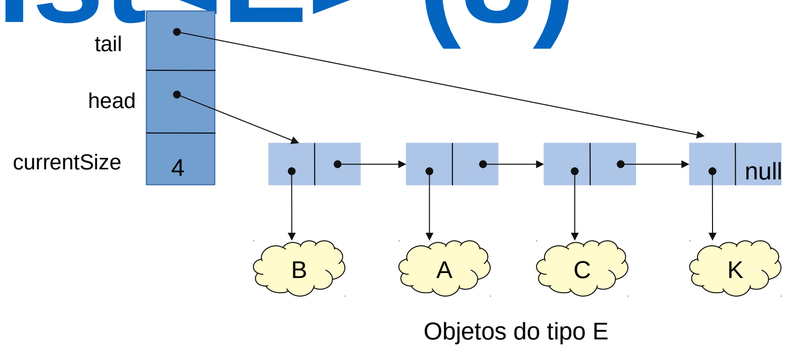
Início → singlyListNode<E> node=head;

Avanço → node = node.getNext();

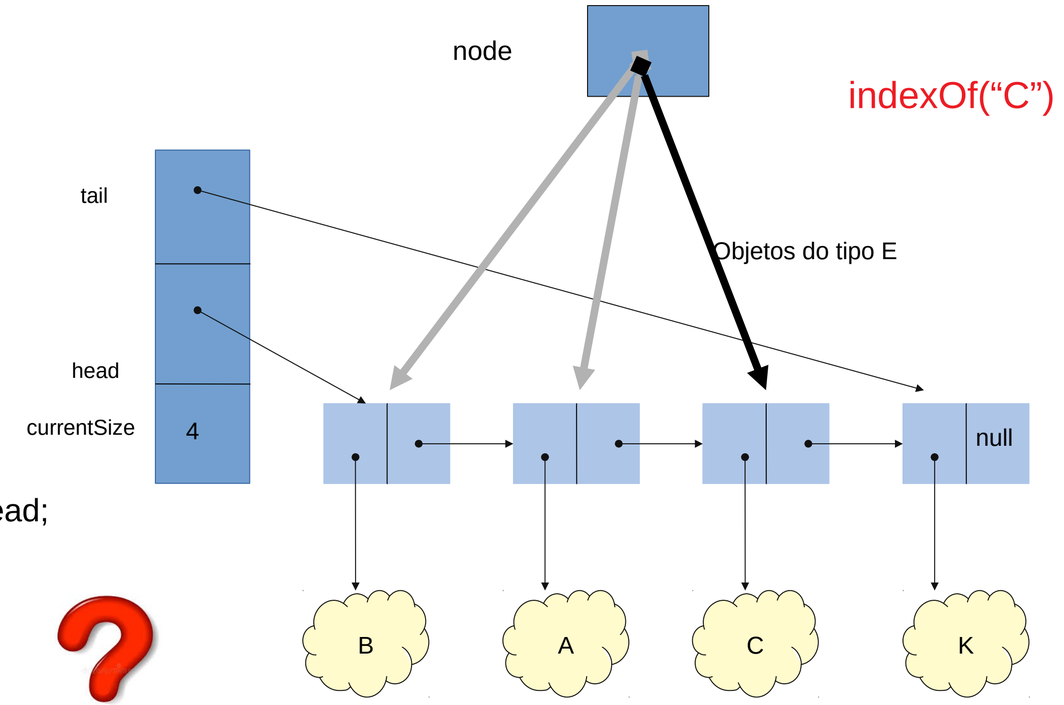
Lista Simplesmente Ligada

Operação	Melhor Caso	Pior Caso	Caso Médio
`isEmpty, size`	$O(1)$	$O(1)$	$O(1)$
`getFirst, getLast`	$O(1)$	$O(1)$	$O(1)$
`get`	$O(1)$	$O(n)$	$O(n)$
`addFirst, addLast`			
`add`			
`removeFirst`			
`removeLast`			
`remove`			
`indexOf (por elemento)`			
`iterator`			

Classe SinglyLinkedList<E> (8)




```
/**
 * 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) {
    SinglyListNode<E> node = head;
    int position = 0;

    while (                                    ) {
        node = node.getNext();
        position++;
    }
    if (                )
        return NOT_FOUND;
    return position;
}
```



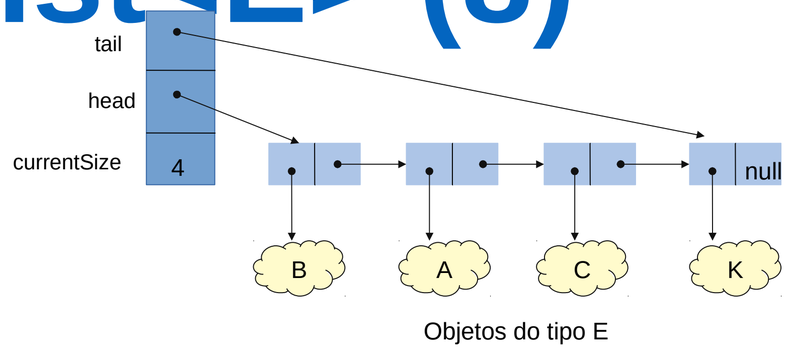
Percurso em Lista

Início → singlyListNode<E> node=head;

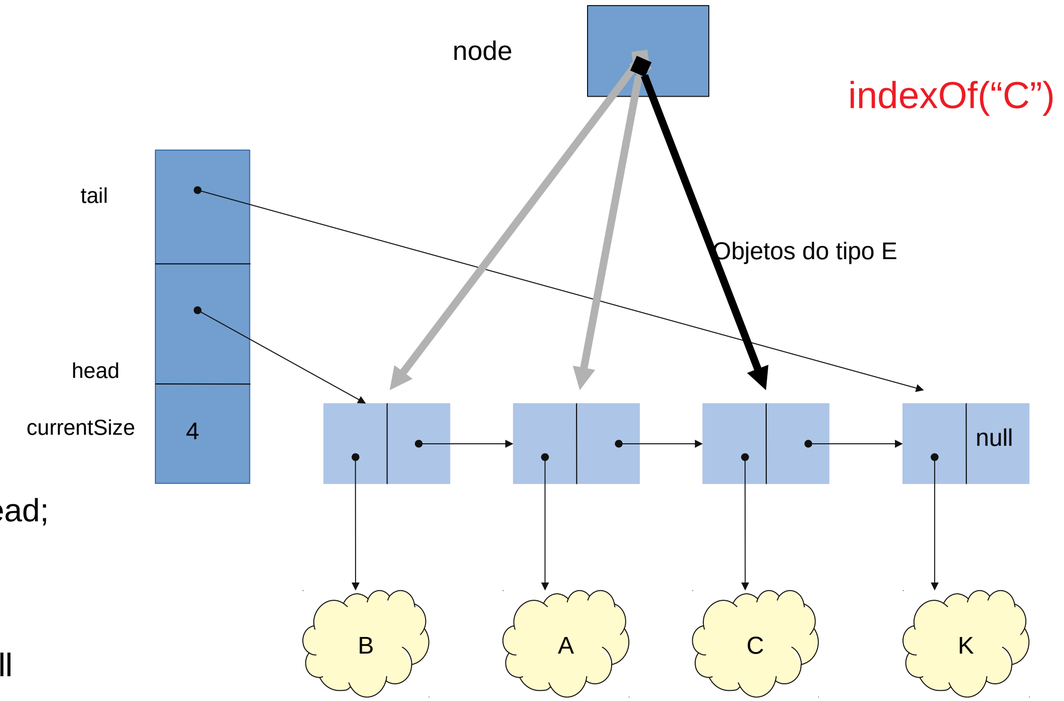
Avanço → node = node.getNext();

Condição de paragem →

Classe SinglyLinkedList<E> (8)




```
/**
 * 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) {
    SinglyListNode<E> node = head;
    int position = 0;

    while ( node != null && !node.getElement().equals(element) ) {
        node = node.getNext();
        position++;
    }
    if ( node == null )
        return NOT_FOUND;
    return position;
}
```



Percurso em Lista

Início → singlyListNode<E> node=head;

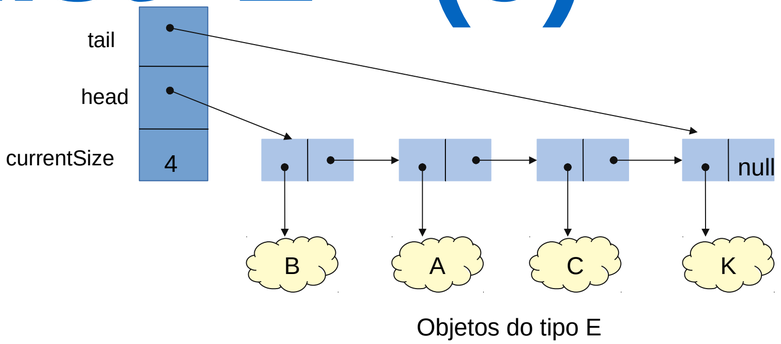
Avanço → node = node.getNext();

Condição de paragem → node==null

Lista Simplesmente Ligada

Operação	Melhor Caso	Pior Caso	Caso Médio
`isEmpty, size`	$O(1)$	$O(1)$	$O(1)$
`getFirst, getLast`	$O(1)$	$O(1)$	$O(1)$
`get`	$O(1)$	$O(n)$	$O(n)$
`addFirst, addLast`			
`add`			
`removeFirst`			
`removeLast`			
`remove`			
`indexOf (por elemento)`	$O(1)$	$O(n)$	$O(n)$
`iterator`			

Classe SinglyLinkedList<E> (9)

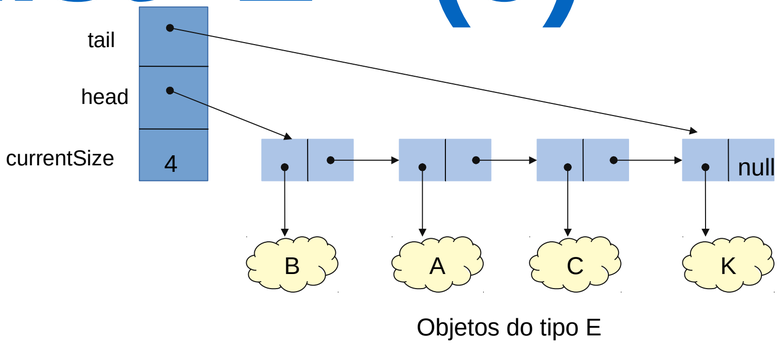



```
/**
 * Inserts the specified element at the first position in the list.
 *
 * @param element to be inserted
 */
public void addFirst(E element) {

}
```

addFirst(“Z”)

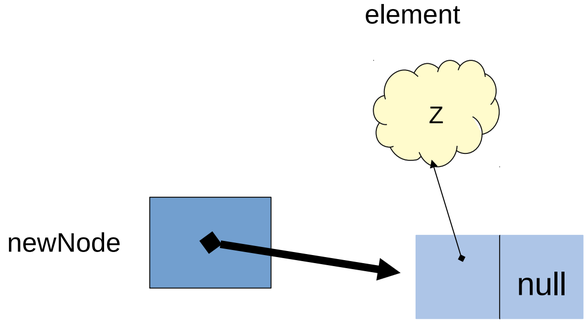
element

Classe SinglyLinkedList<E> (9)

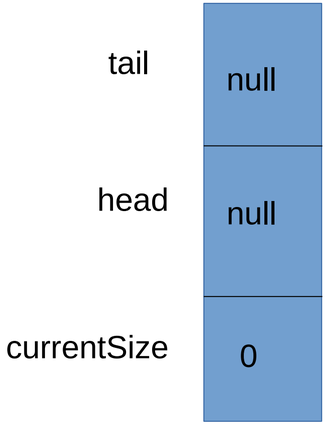



```
/**
 * Inserts the specified element at the first position in the list.
 *
 * @param element to be inserted
 */
public void addFirst(E element) {
    SinglyListNode<E> newNode = new SinglyListNode<>(element, head);
    if ( this.isEmpty() )

}
```

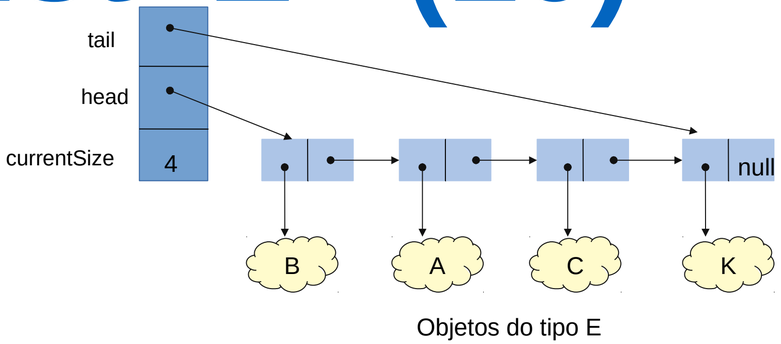
addFirst("Z")



Lista vazia



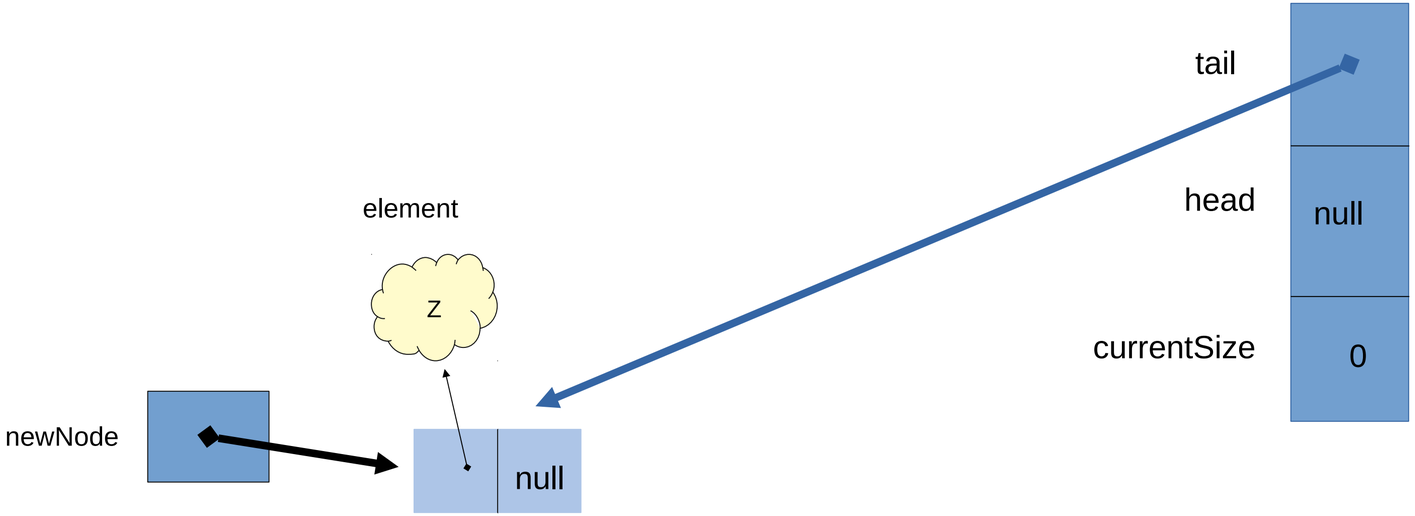
Classe SinglyLinkedList<E> (10)



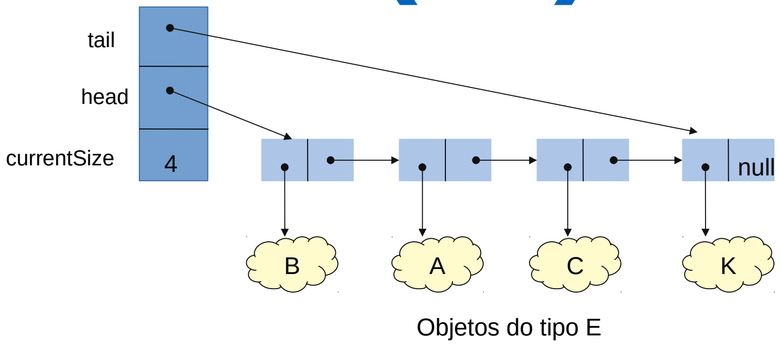

```
/**
 * Inserts the specified element at the first position in the list.
 *
 * @param element to be inserted
 */
public void addFirst(E element) {

    SinglyListNode<E> newNode = new SinglyListNode<>(element, head);

    if ( this.isEmpty() )
        tail = newNode;

}
```

addFirst("Z")

Lista vazia



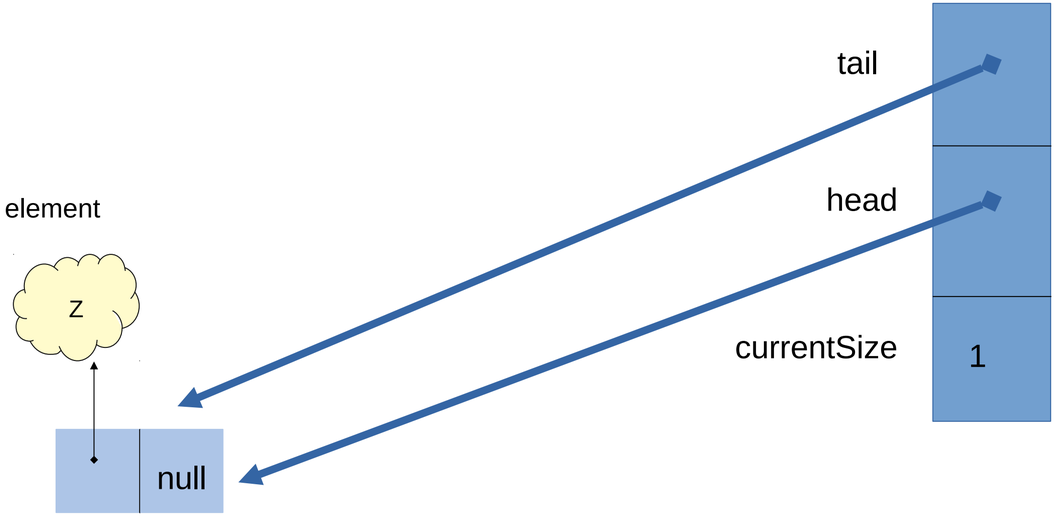
Classe SinglyLinkedList<E> (11)



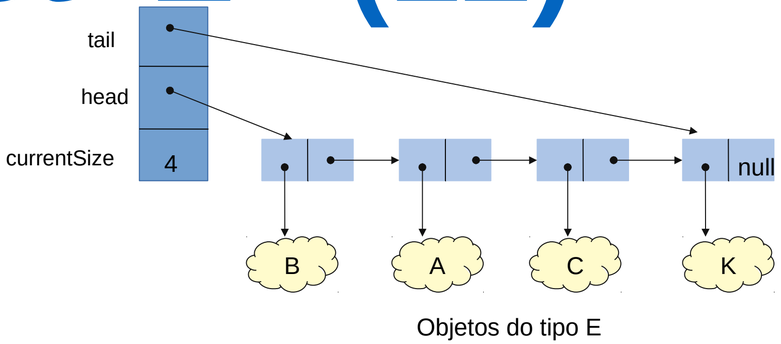

```
/**
 * Inserts the specified element at the first position in the list.
 *
 * @param element to be inserted
 */
public void addFirst(E element) {

    SinglyListNode<E> newNode = new SinglyListNode<>(element, head);

    if ( this.isEmpty() )
        tail = newNode;

   head = newNode;
   currentSize++;

}
```

addFirst(“Z”)

Lista vazia



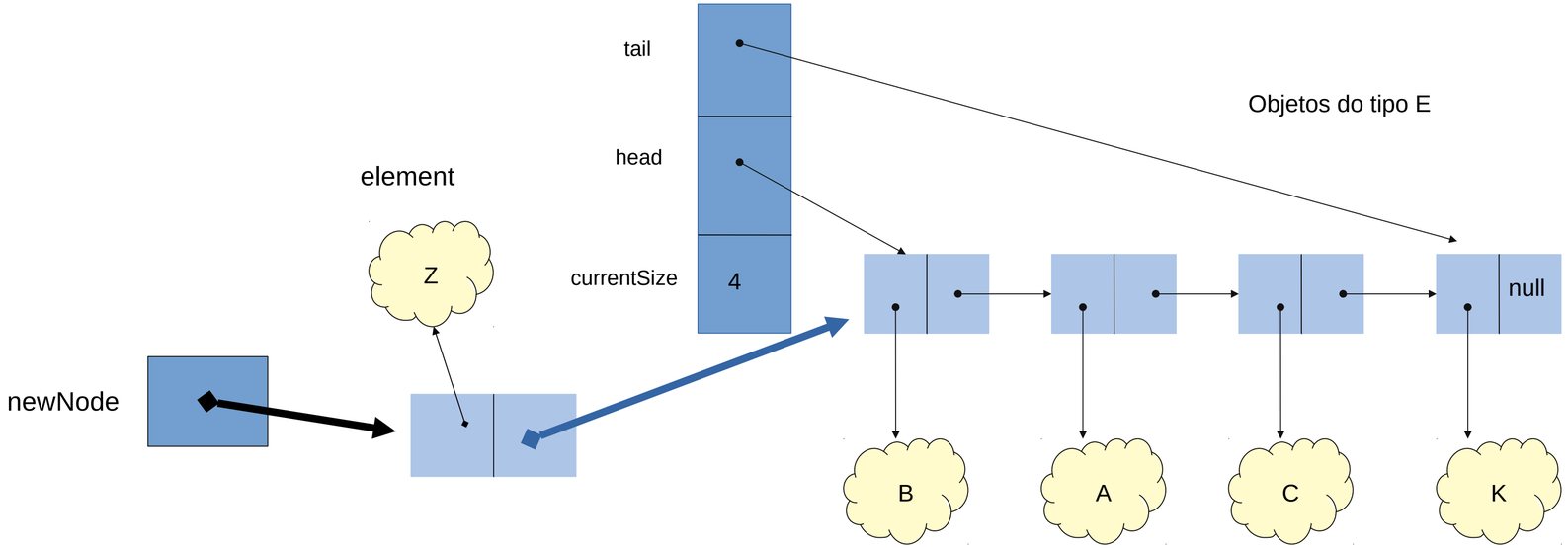
Classe SinglyLinkedList<E> (12)



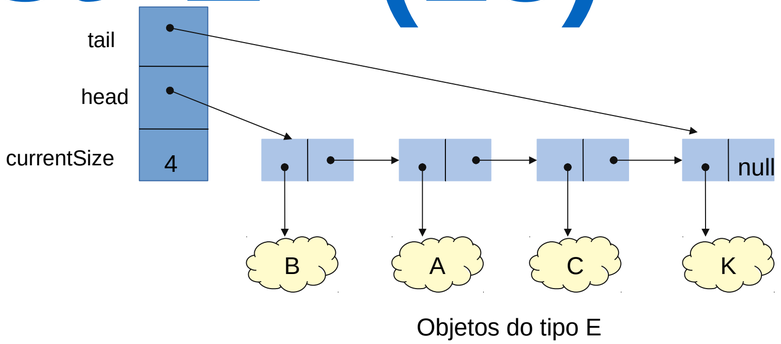

```
/**
 * Inserts the specified element at the first position in the list.
 *
 * @param element to be inserted
 */
public void addFirst(E element) {

    SinglyListNode<E> newNode = new SinglyListNode<>(element, head);

    if ( this.isEmpty() )
        tail = newNode;


}
```

addFirst("Z")

Lista não vazia



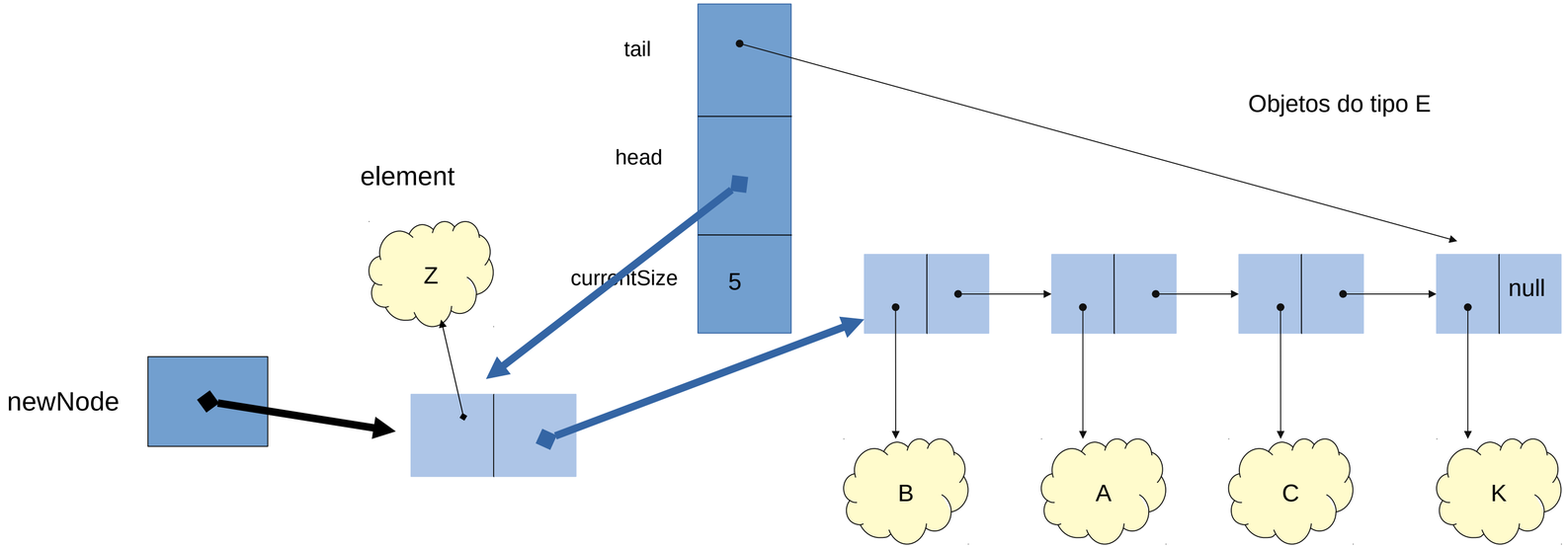
Classe SinglyLinkedList<E> (13)



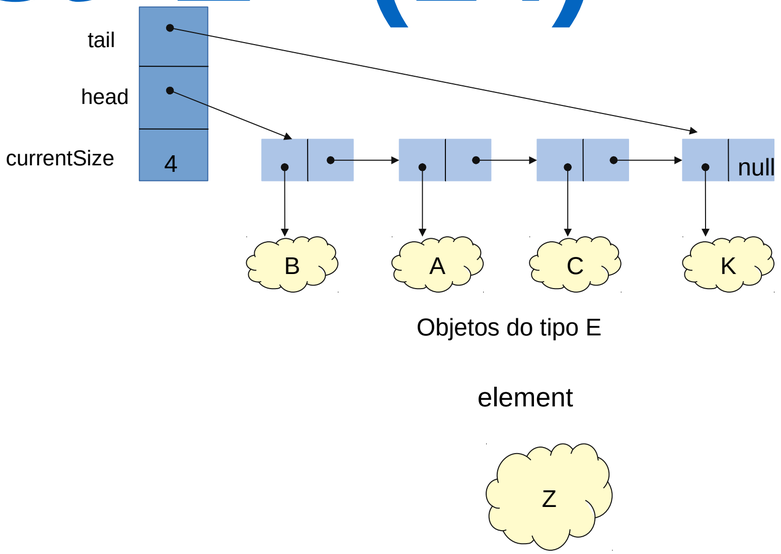

```
/**
 * Inserts the specified element at the first position in the list.
 *
 * @param element to be inserted
 */
public void addFirst(E element) {

    SinglyListNode<E> newNode = new SinglyListNode<>(element, head);

    if ( this.isEmpty() )
        tail = newNode;

    head = newNode;
    currentSize++;

}
```

addFirst(“Z”)

Lista não vazia



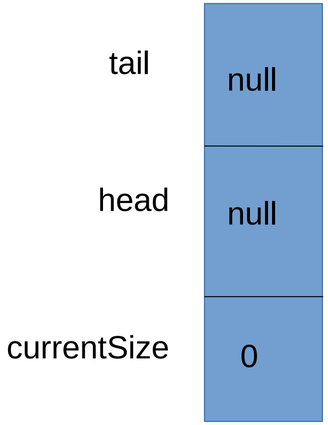
Classe SinglyLinkedList<E> (14)



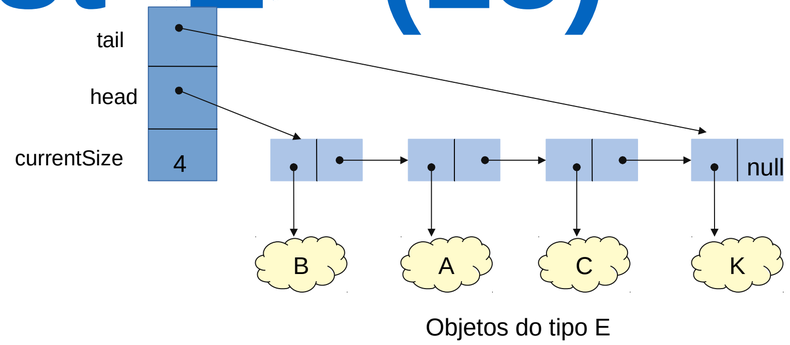

```
/**
 * Inserts the specified element at the last position in the list.
 *
 * @param element to be inserted
 */
public void addLast(E element) {
    if ( this.isEmpty() ) addFirst(element);

}
```

addLast(“Z”)

Lista vazia



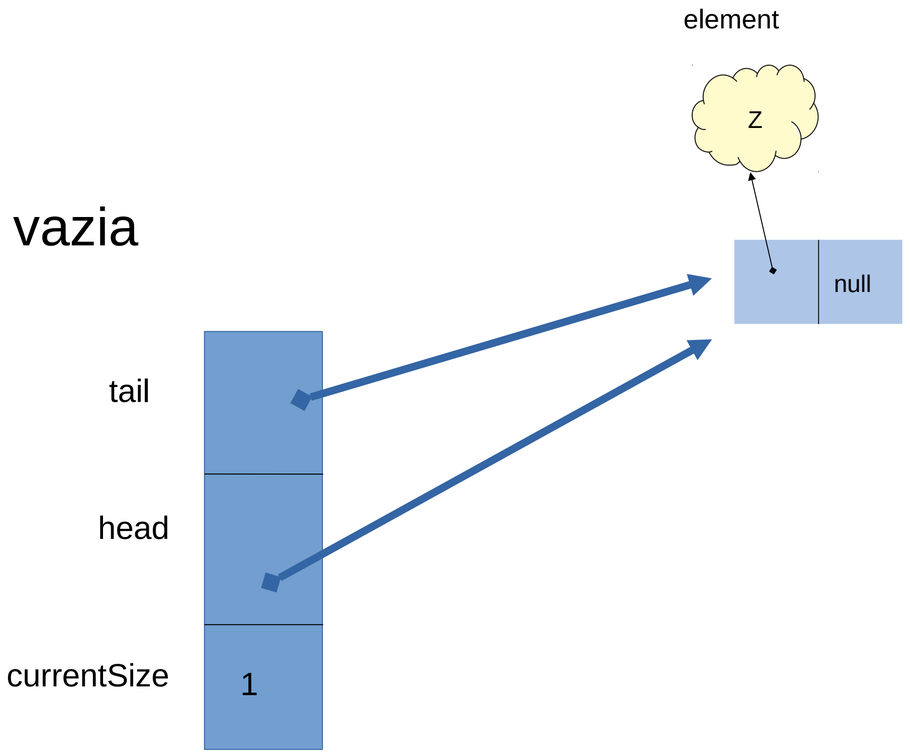
Classe SinglyLinkedList<E> (15)



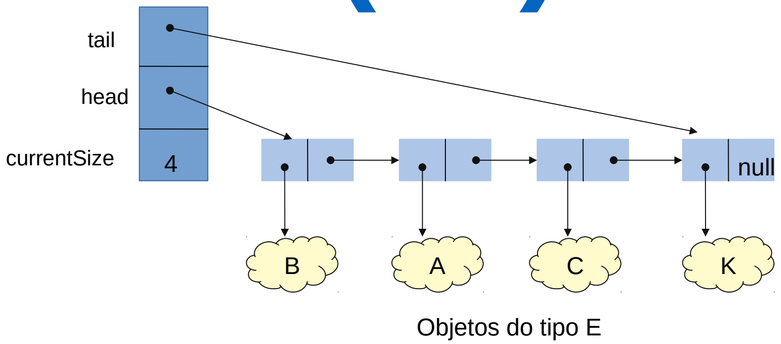

```
/**
 * Inserts the specified element at the last position in the list.
 *
 * @param element to be inserted
 */
public void addLast(E element) {

    if ( this.isEmpty() ) addFirst(element);



}
```

addLast(“Z”)

Lista vazia



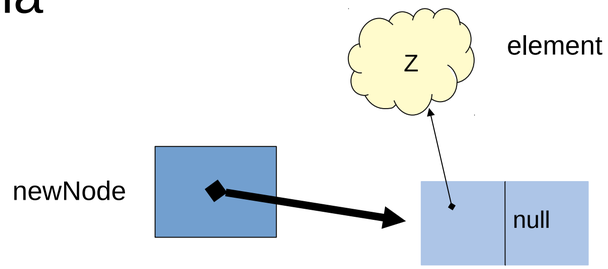
Classe SinglyLinkedList<E> (16)

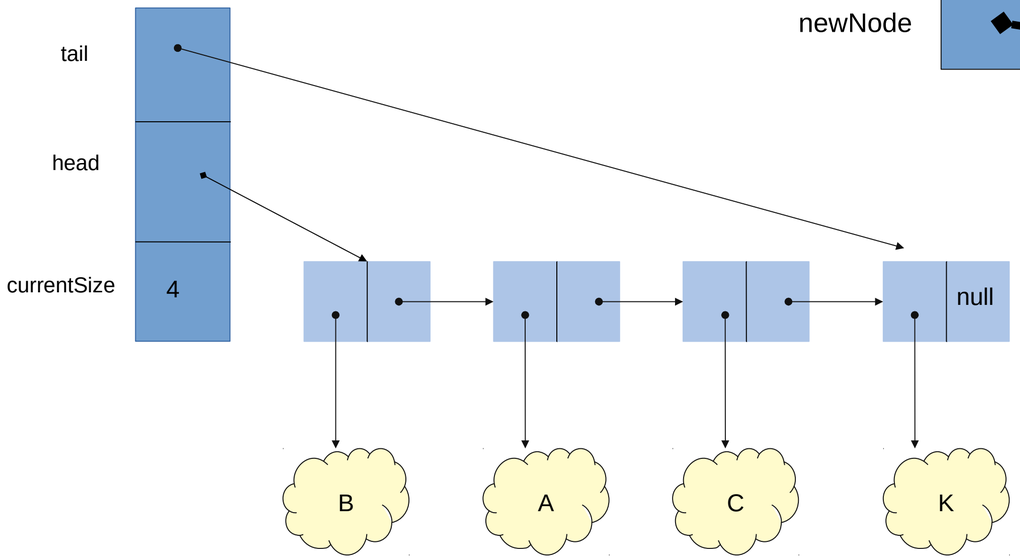



```
/**
 * Inserts the specified element at the last position in the list.
 *
 * @param element to be inserted
 */
public void addLast(E element) {
    if ( this.isEmpty() ) addFirst(element);
    SinglyListNode<E> newNode = new SinglyListNode<>(element, null);

}
```

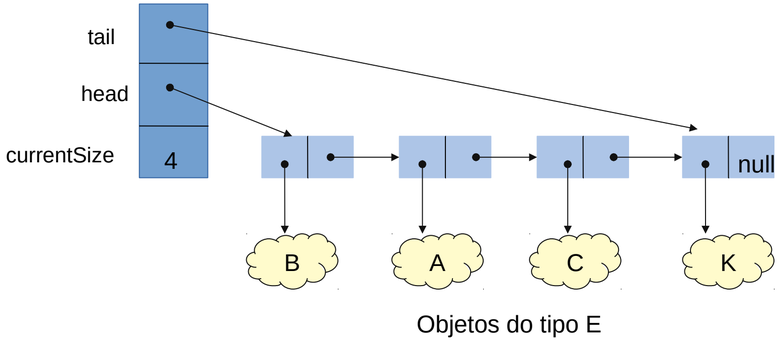
addLast("Z")

Lista não vazia





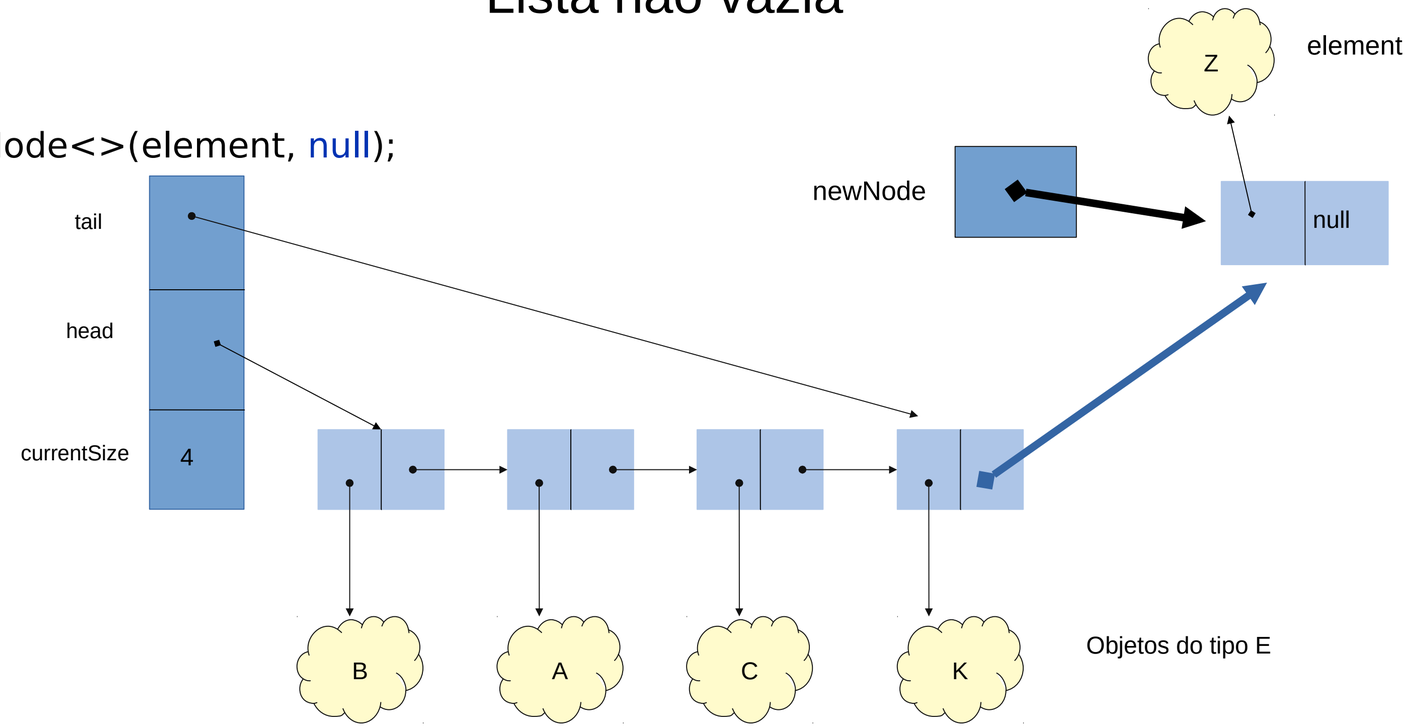
Classe SinglyLinkedList<E> (17)



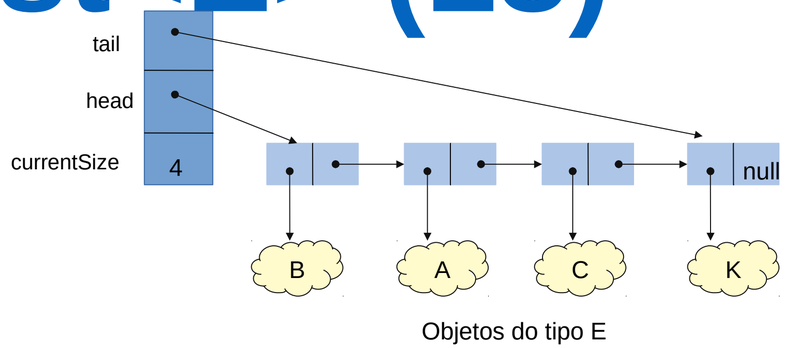

```
/**
 * Inserts the specified element at the last position in the list.
 *
 * @param element to be inserted
 */

public void addLast(E element) {
    if ( this.isEmpty() ) addFirst(element);

    SinglyListNode<E> newNode = new SinglyListNode<>(element, null);

    tail.setNext(newNode);


}
```

addLast(“Z”)

Lista não vazia



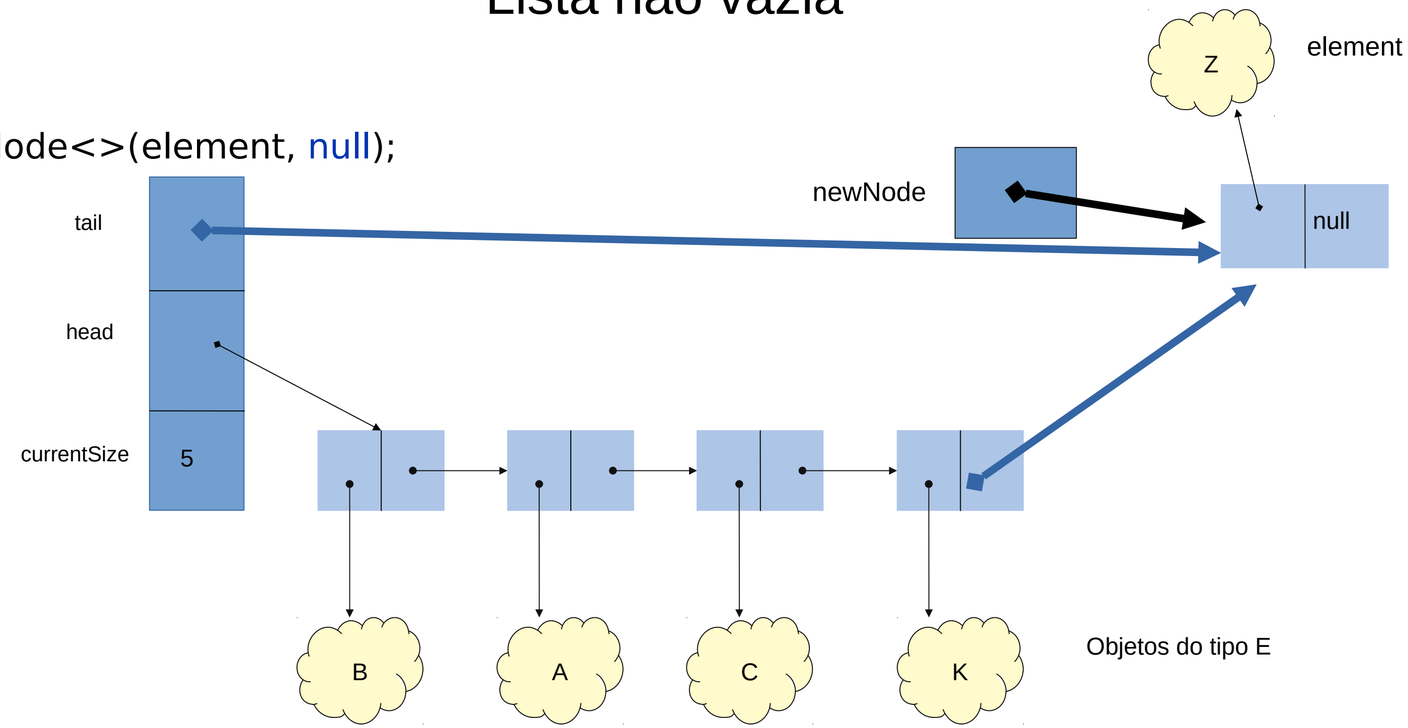
Classe SinglyLinkedList<E> (18)



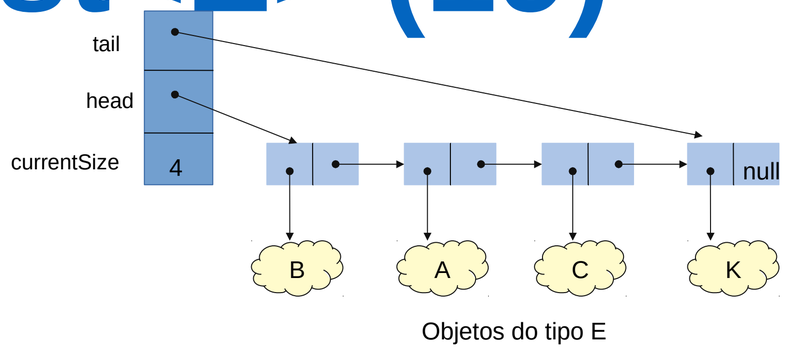

```
/**
 * Inserts the specified element at the last position in the list.
 *
 * @param element to be inserted
 */

public void addLast(E element) {
    if ( this.isEmpty() ) addFirst(element);

    SinglyListNode<E> newNode = new SinglyListNode<>(element, null);

    tail.setNext(newNode);

    tail = newNode;
    currentSize++;
}
```

addLast(“Z”)

Lista não vazia



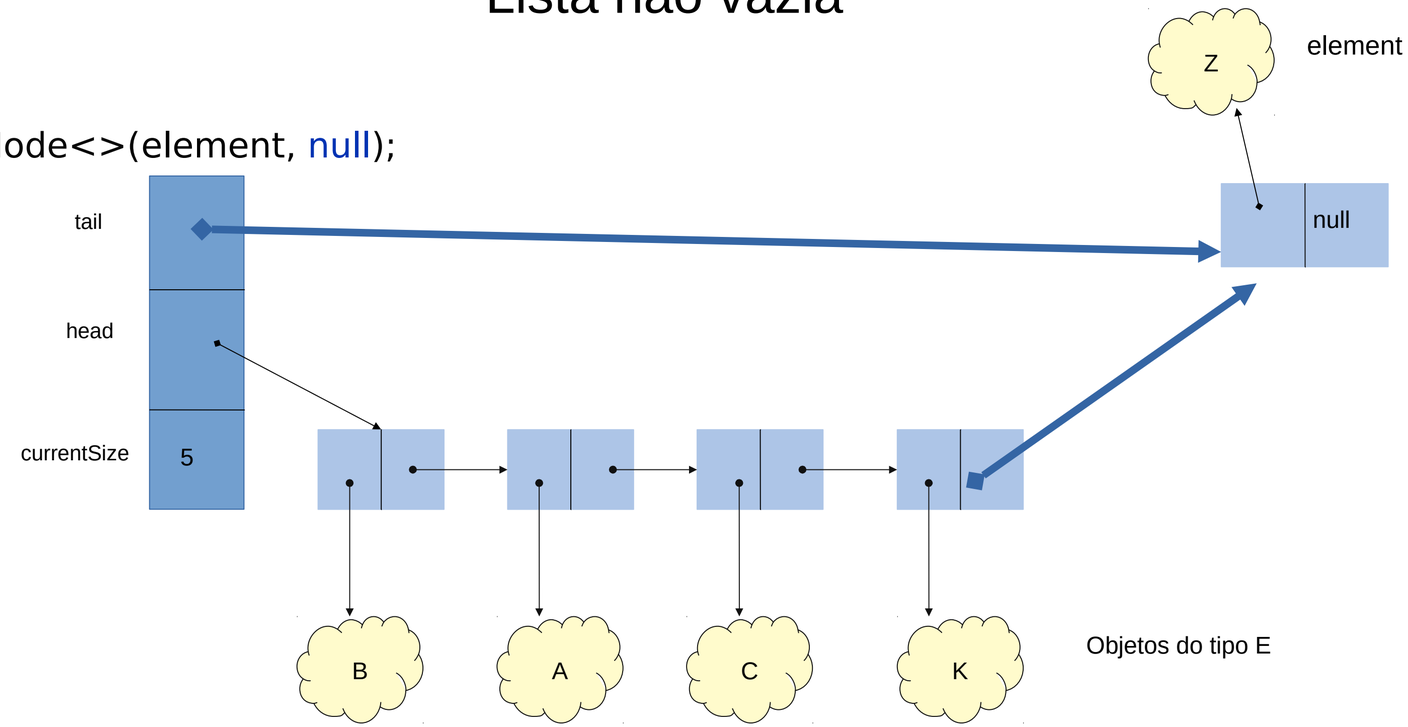
Classe SinglyLinkedList<E> (19)



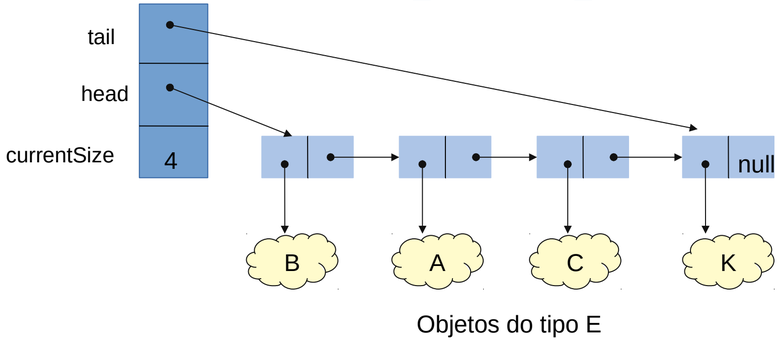

```
/**
 * Inserts the specified element at the last position in the list.
 *
 * @param element to be inserted
 */

public void addLast(E element) {
    if ( this.isEmpty() ) addFirst(element);

    SinglyListNode<E> newNode = new SinglyListNode<>(element, null);

    tail.setNext(newNode);

    tail = newNode;
    currentSize++;
}
```

addLast(“Z”)

Lista não vazia



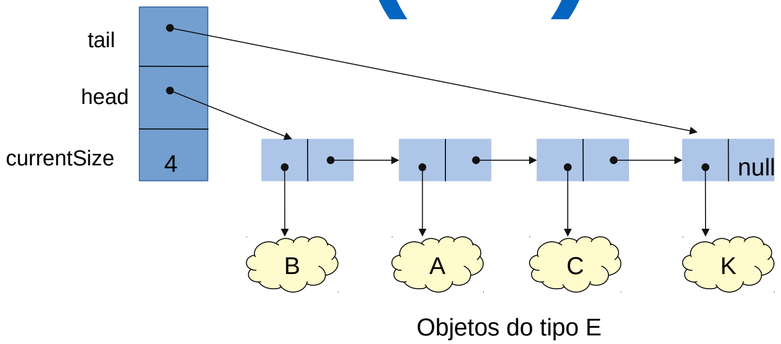
Classe SinglyLinkedList<E> (20)



add(2, “Z”)

```
/**
 * 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
 */

public void add(int position, E element) {
    if ( position < 0 || position > currentSize )
        throw new InvalidPositionException();
    if ( position == 0 )
        this.addFirst(element);
    else if ( position == currentSize )
        this.addLast(element);
    else
        this.addMiddle(position, element);
}
```

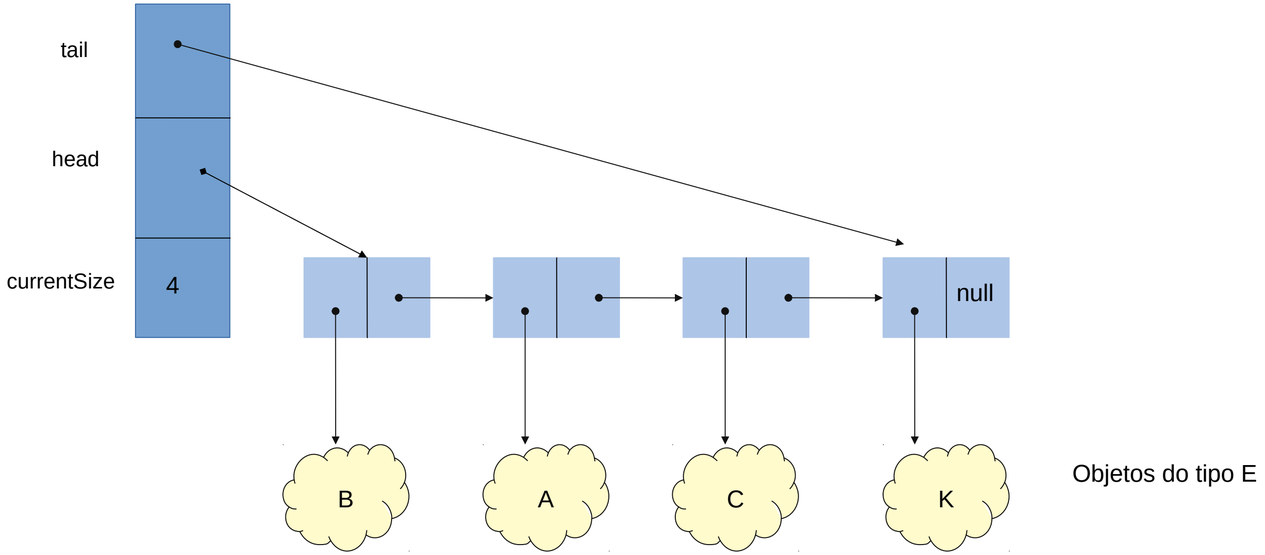
Classe SinglyLinkedList<E> (21)



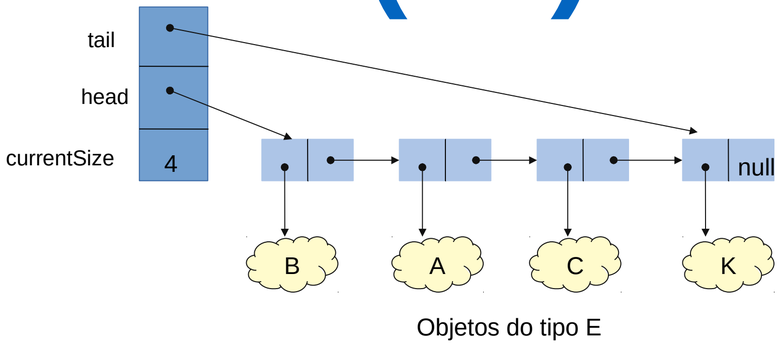
add(2, “Z”)

```
private void addMiddle(int position, E element) {
```

```
}
```

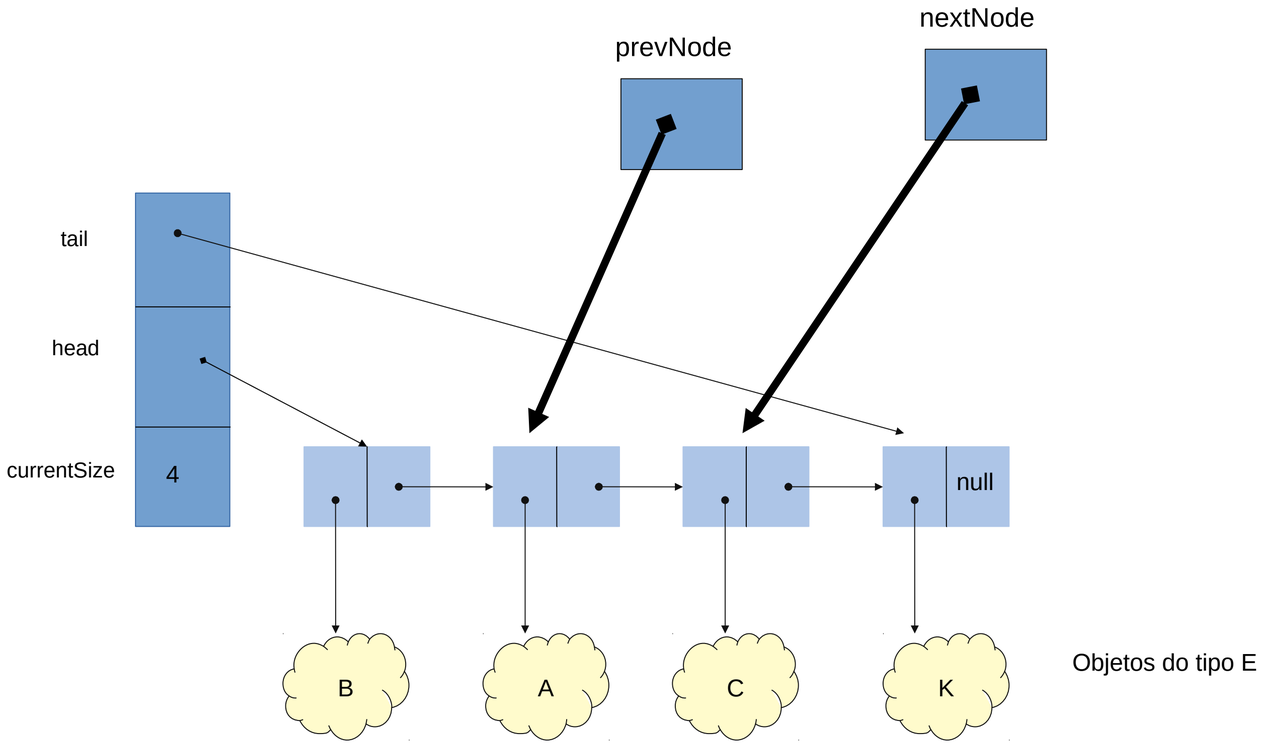


Classe SinglyLinkedList<E> (21)

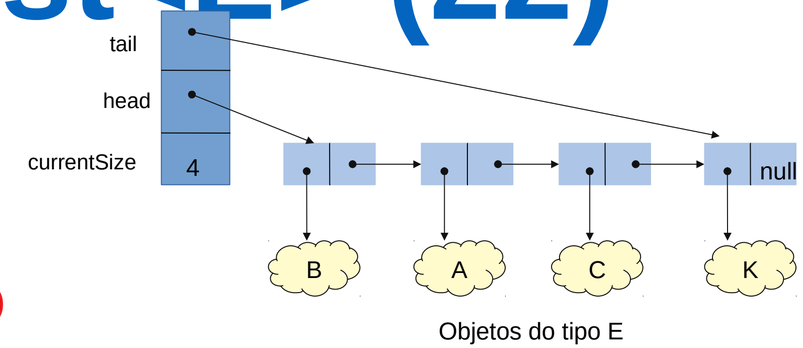


add(2, “Z”)

```
private void addMiddle(int position, E element) {
    SinglyListNode<E> prevNode = this.getNode(position – 1);
    SinglyListNode<E> nextNode = prevNode.getNext();


}
```

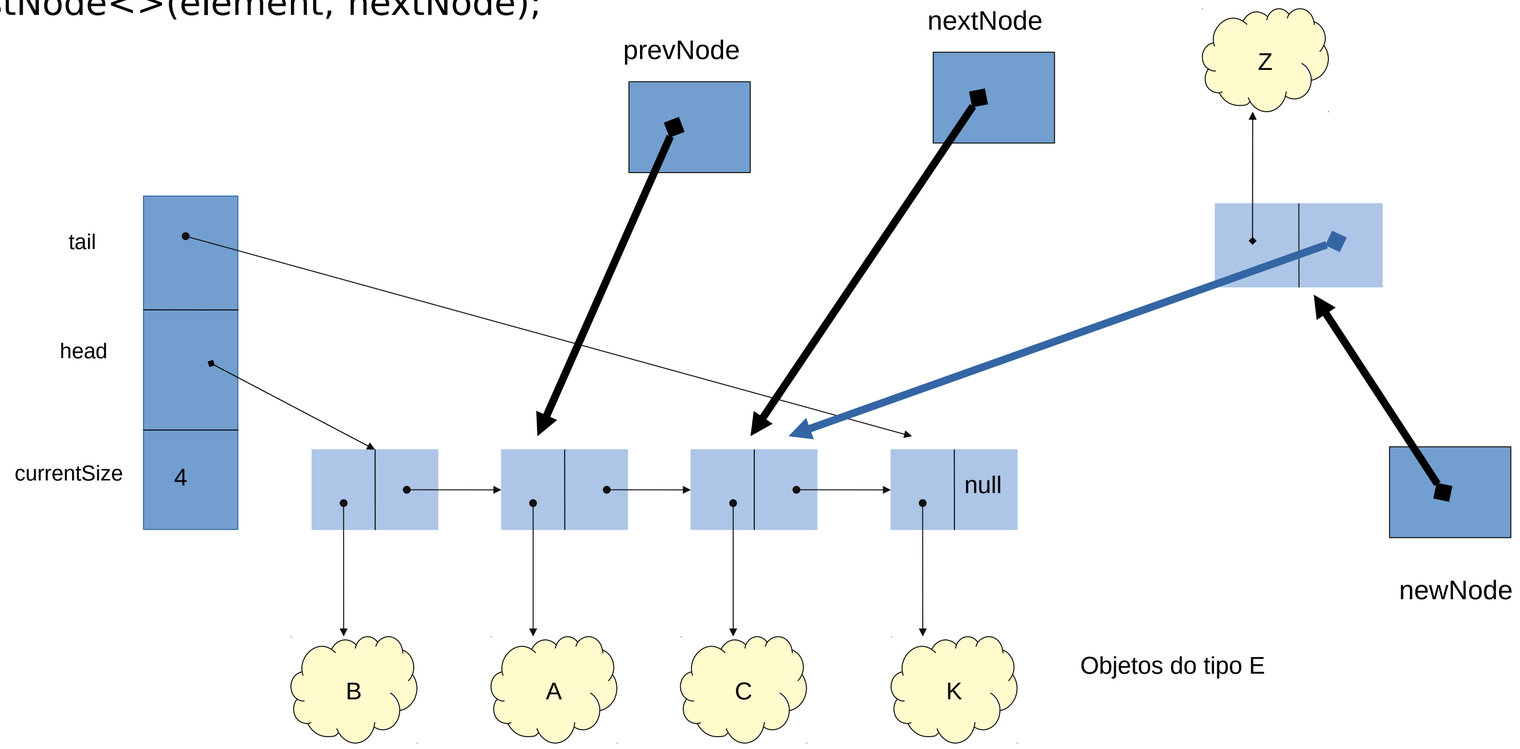


Classe SinglyLinkedList<E> (22)

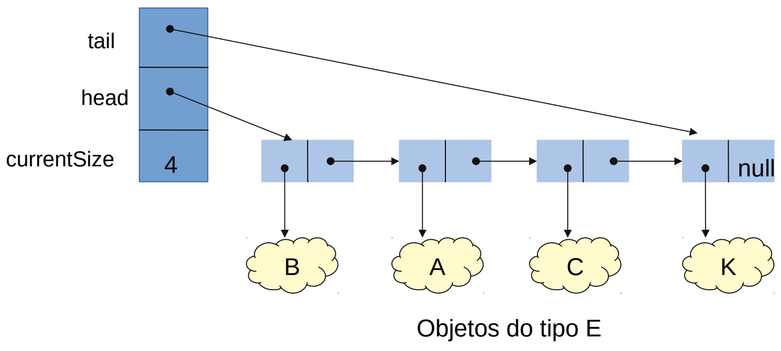


add(2, “Z”)

```
private void addMiddle(int position, E element) {
    SinglyListNode<E> prevNode = this.getNode(position – 1);
    SinglyListNode<E> nextNode = prevNode.getNext();
    SinglyListNode<E> newNode = new SinglyListNode<>(element, nextNode);

}
```

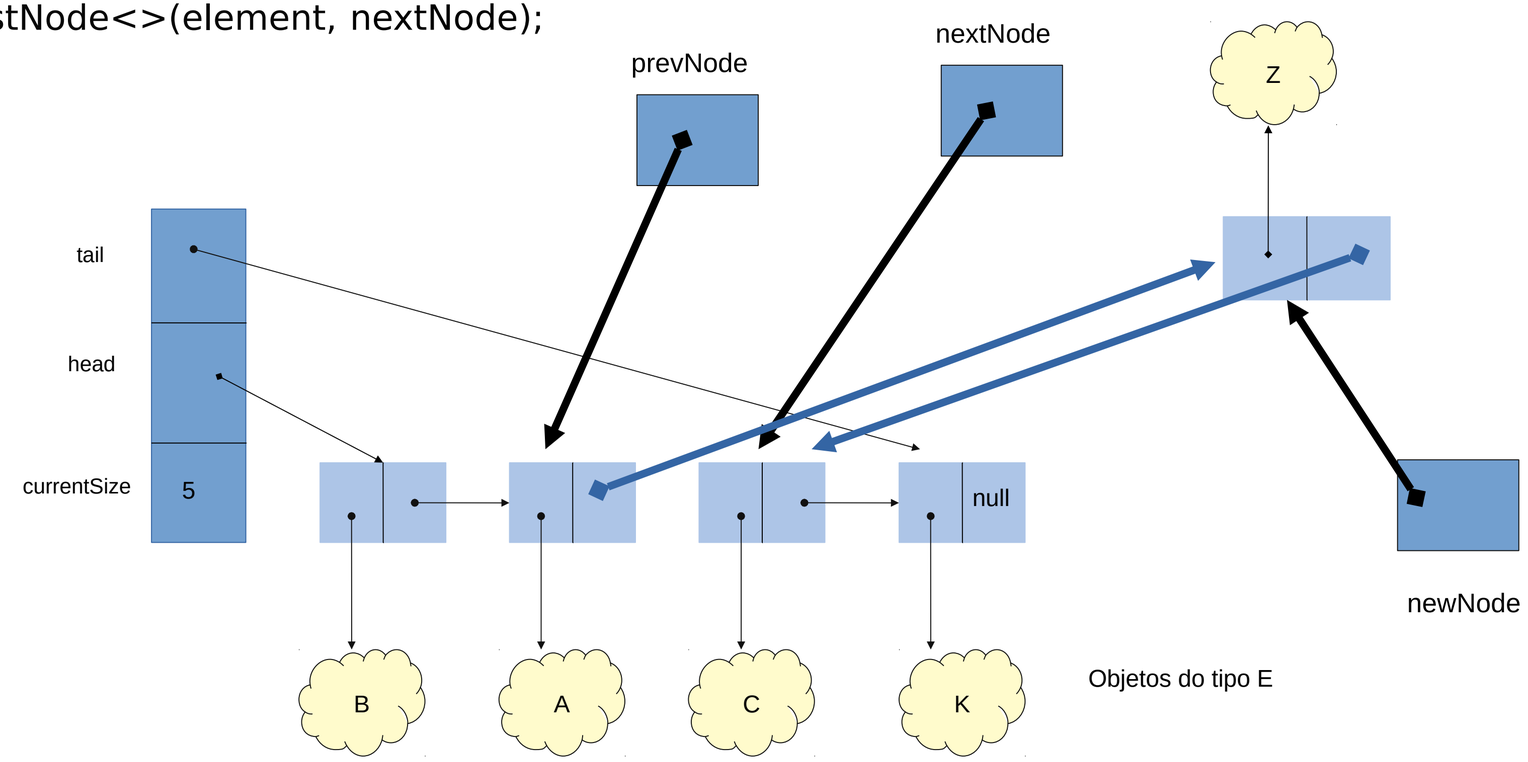


Classe SinglyLinkedList<E> (23)

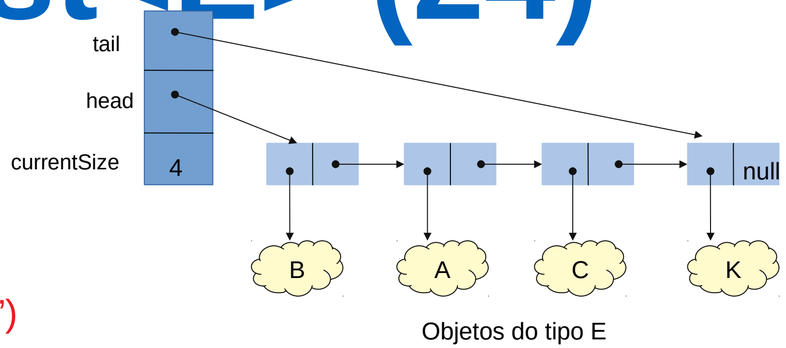



```
private void addMiddle(int position, E element) {
    SinglyListNode<E> prevNode = this.getNode(position – 1);
    SinglyListNode<E> nextNode = prevNode.getNext();
    SinglyListNode<E> newNode = new SinglyListNode<>(element, nextNode);
    prevNode.setNext(newNode);
    currentSize++;

}
```

add(2, “Z”)

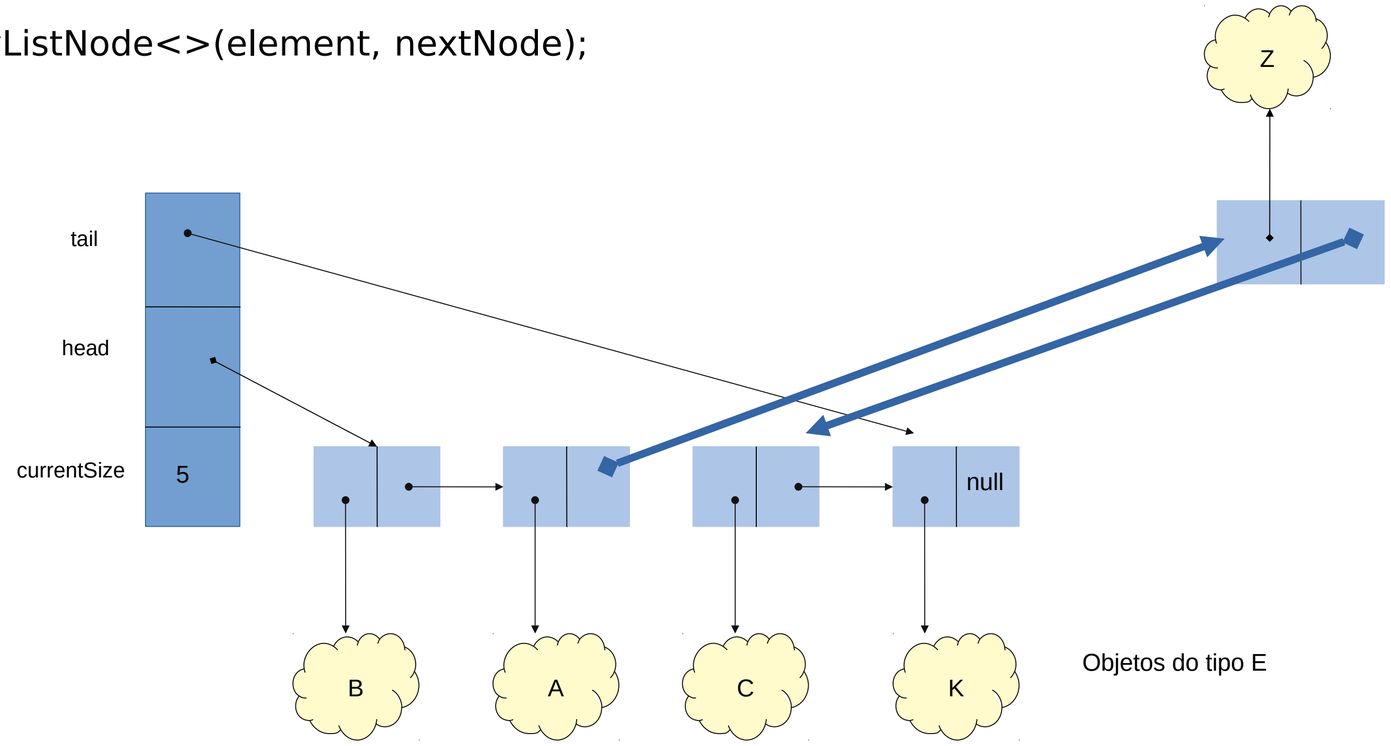


Classe SinglyLinkedList<E> (24)




```
private void addMiddle(int position, E element) {
    SinglyListNode<E> prevNode = this.getNode(position – 1);
    SinglyListNode<E> nextNode = prevNode.getNext();
    SinglyListNode<E> newNode = new SinglyListNode<>(element, nextNode);
    prevNode.setNext(newNode);
    currentSize++;

}
```

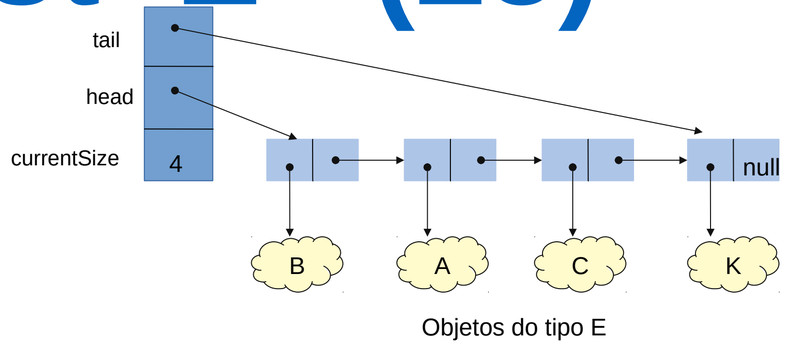
add(2, “Z”)



Lista Simplesmente Ligada

Operação	Melhor Caso	Pior Caso	Caso Médio
`isEmpty, size`	$O(1)$	$O(1)$	$O(1)$
`getFirst, getLast`	$O(1)$	$O(1)$	$O(1)$
`get`	$O(1)$	$O(n)$	$O(n)$
`addFirst, addLast`	$O(1)$	$O(1)$	$O(1)$
`add`	$O(1)$	$O(n)$	$O(n)$
`removeFirst`			
`removeLast`			
`remove`			
`indexOf (por elemento)`	$O(1)$	$O(n)$	$O(n)$
`iterator`			

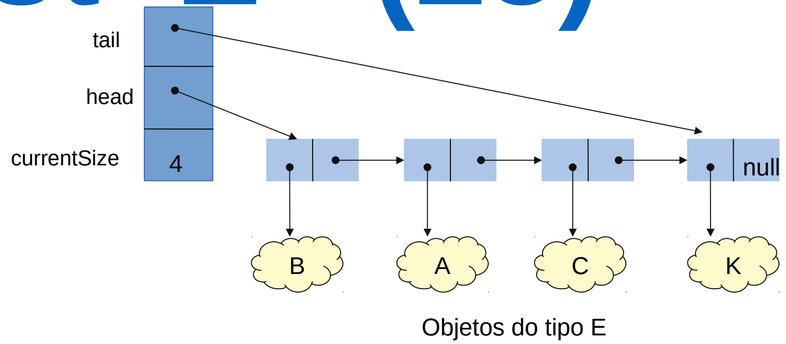
Classe SinglyLinkedList<E> (25)



removeFirst()

```
/**
 * 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() {
    if ( this.isEmpty() )
        throw new NoSuchElementException();



    return element;
}
```

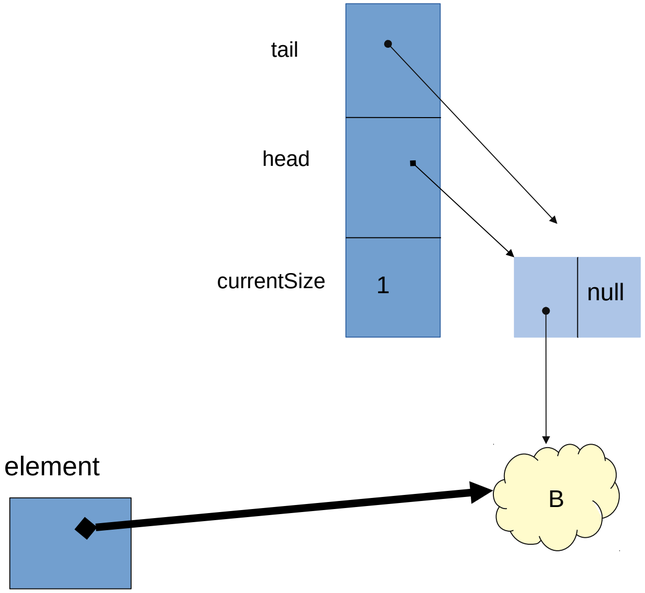
Classe SinglyLinkedList<E> (25)




```
/**
 * 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() {
    if ( this.isEmpty() )
        throw new NoSuchElementException();

    E element=head.getElement();



    return element;
}
```

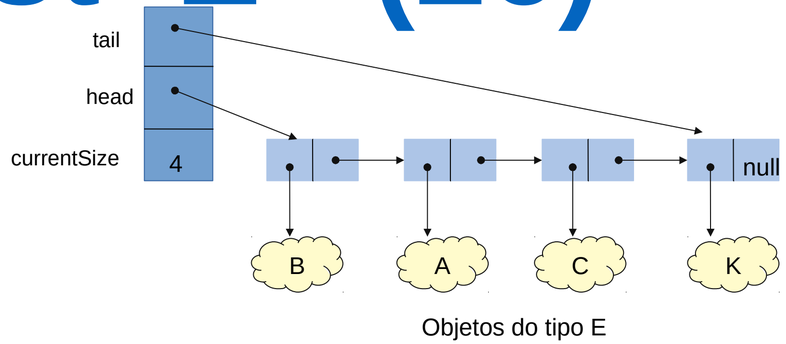
removeFirst()

Lista com um elemento



Objetos do tipo E

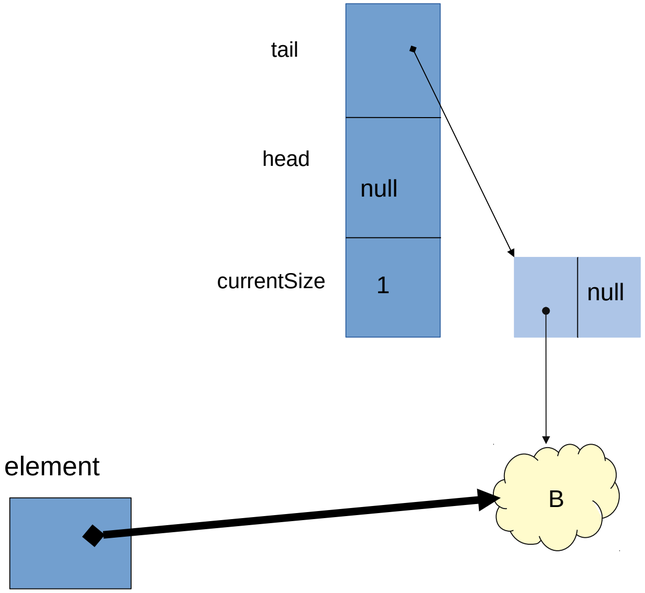
Classe SinglyLinkedList<E> (26)




```
/**
 * 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() {
    if ( this.isEmpty() )
        throw new NoSuchElementException();

    E element=head.getElement();
    head = head.getNext();



    return element;
}
```

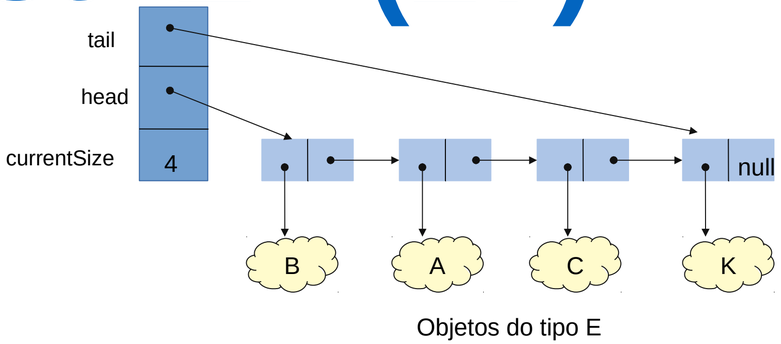
removeFirst()

Lista com um elemento



Objetos do tipo E

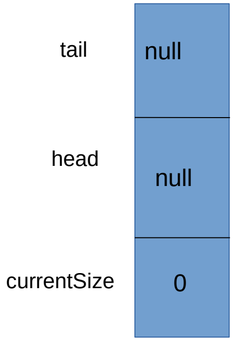
Classe SinglyLinkedList<E> (27)




```
/**
 * 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() {
    if ( this.isEmpty() )
        throw new NoSuchElementException();

    E element=head.getElement();
    head = head.getNext();

    if ( head == null )
        tail = null;

    currentSize--;

    return element;
}
```

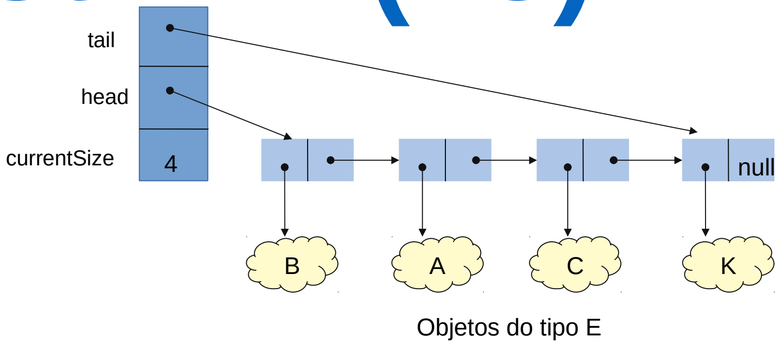
removeFirst()

Lista com um elemento



Objetos do tipo E

Classe SinglyLinkedList<E> (28)

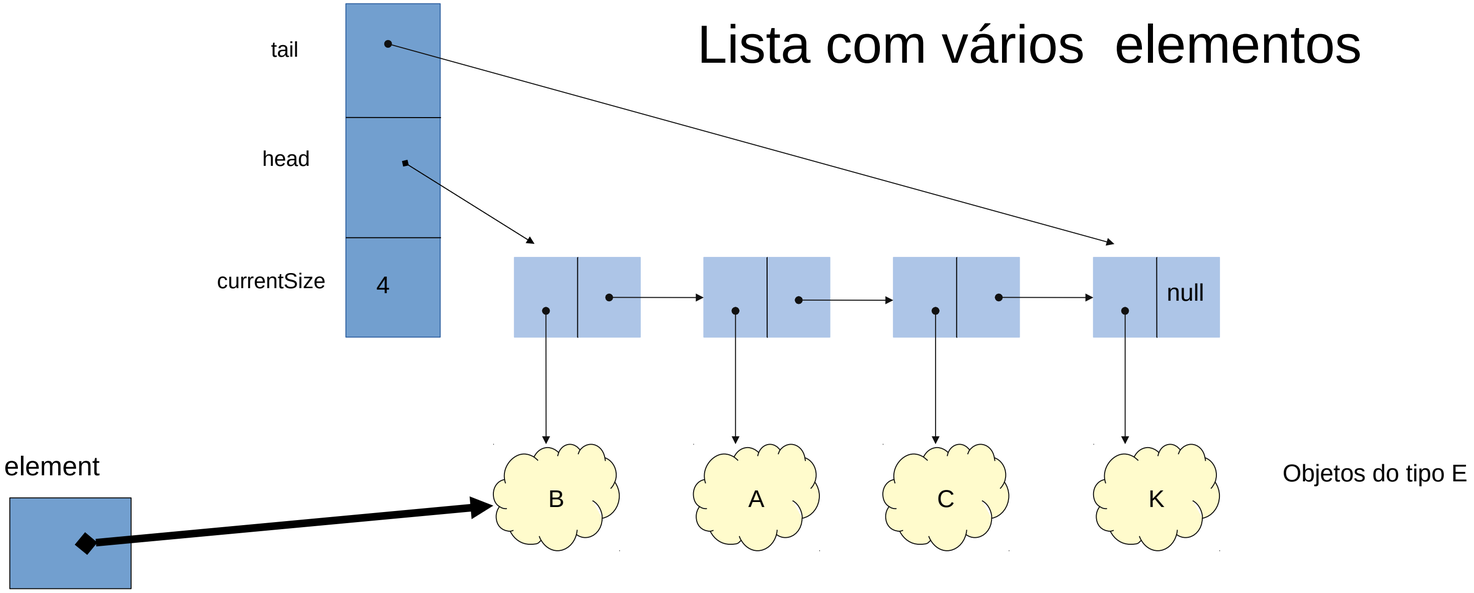



```
/**
 * Removes and returns the element at the first position in the list.
 *
 * @return element removed from the first position of the list
 * @throws NoSuchElementException - if size() == 0
 */

public E removeFirst() {
    if ( this.isEmpty() )
        throw new NoSuchElementException();

    E element=head.getElement();
```

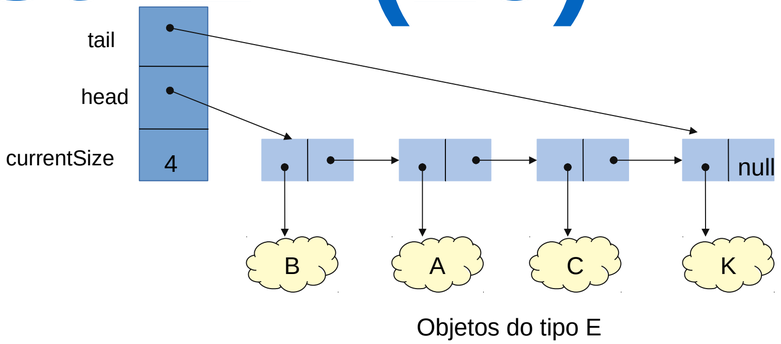
```
}
```

removeFirst()

Lista com vários elementos

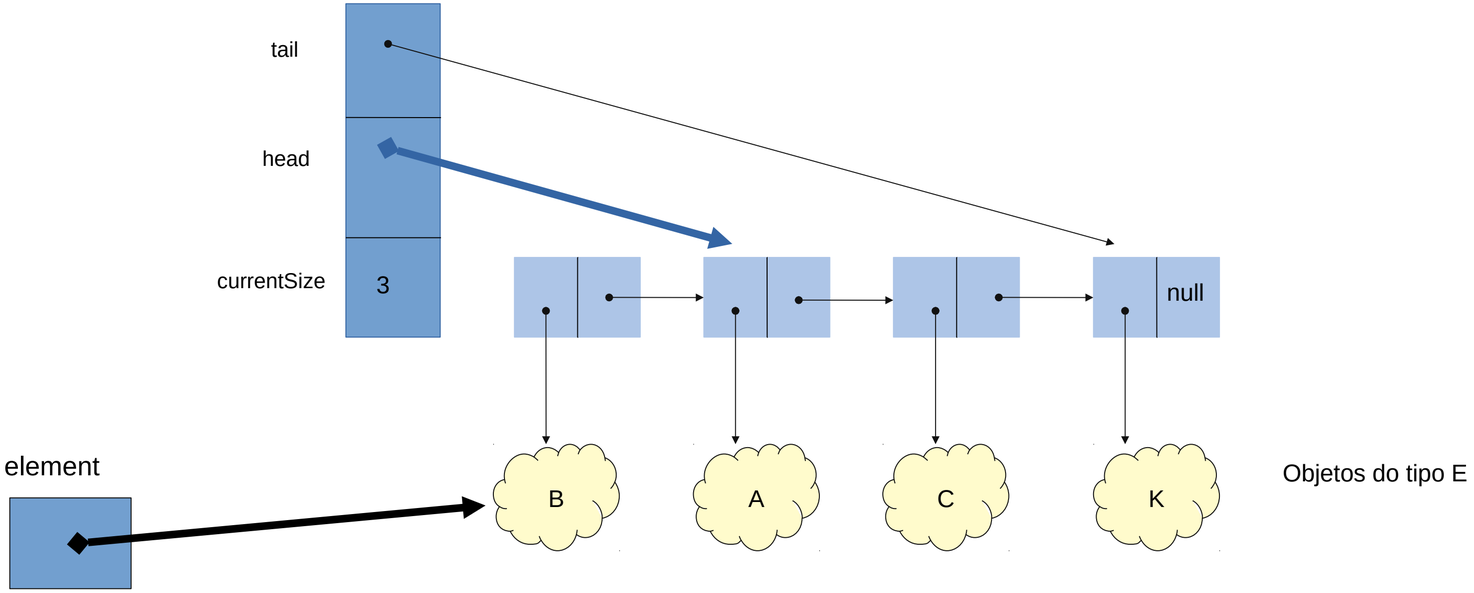


Classe SinglyLinkedList<E> (29)

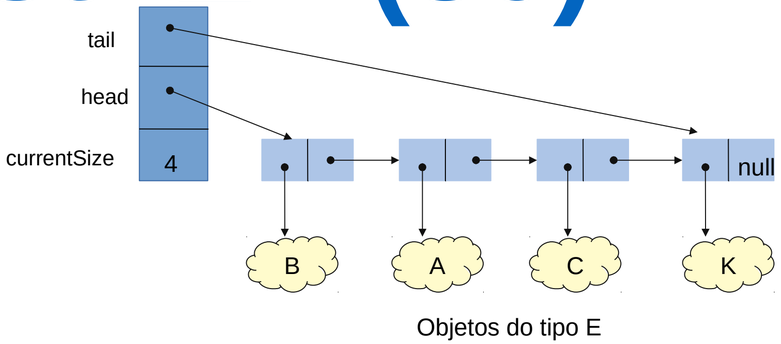



```
/**
 * Removes and returns the element at the first position in the list.
 *
 * @return element removed from the first position of the list
 * @throws NoSuchElementException - if size() == 0
 */

public E removeFirst() {
    if ( this.isEmpty() )
        throw new NoSuchElementException();

    E element=head.getElement();
    head = head.getNext();

    if ( head == null )
        tail = null;

}
```

removeFirst()

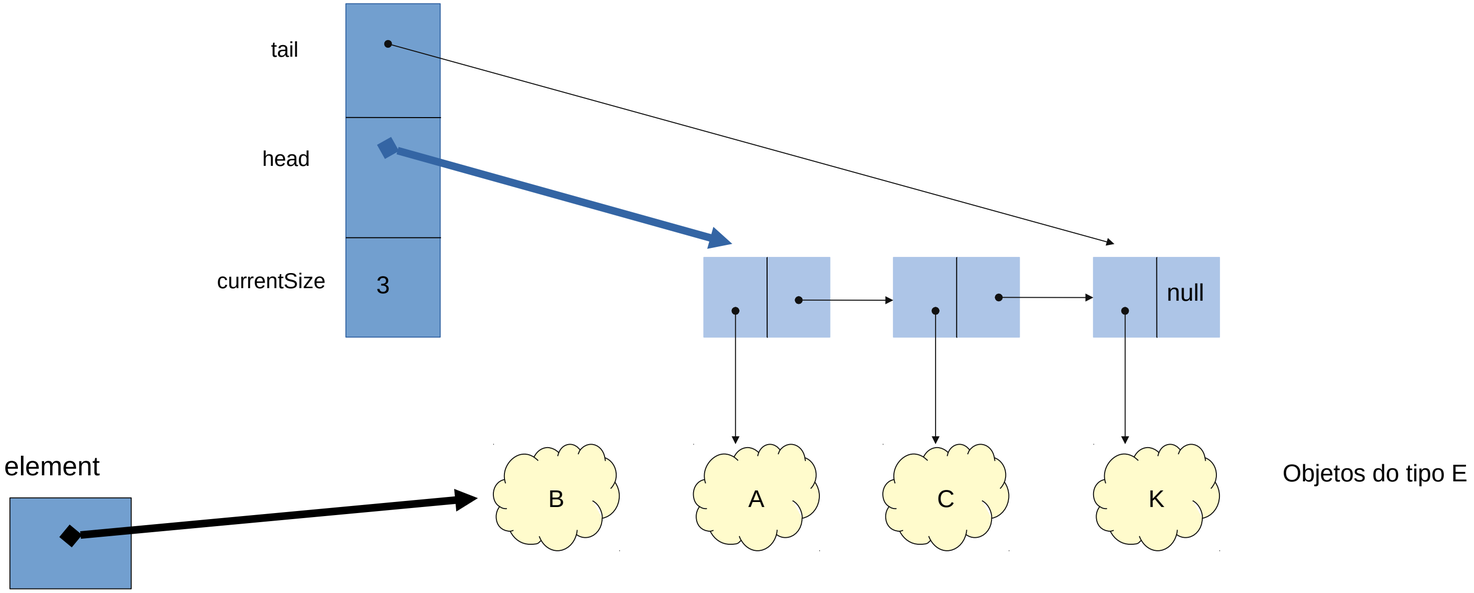


Classe SinglyLinkedList<E> (30)

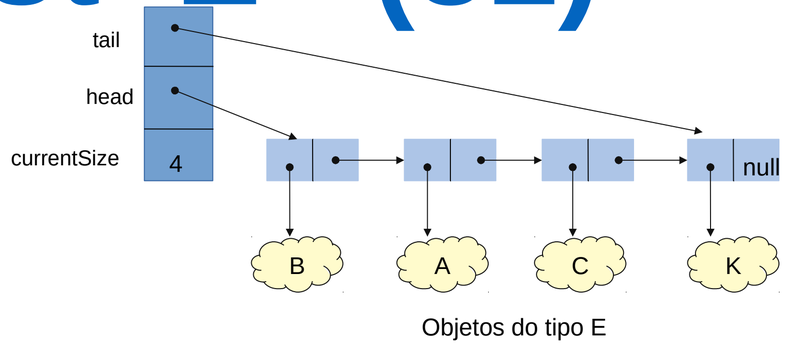



```
/**
 * Removes and returns the element at the first position in the list.
 *
 * @return element removed from the first position of the list
 * @throws NoSuchElementException - if size() == 0
 */

public E removeFirst() {
    if ( this.isEmpty() )
        throw new NoSuchElementException();

    E element=head.getElement();
    head = head.getNext();

    if ( head == null )
        tail = null;

    currentSize--;
    return element;
}
```

removeFirst()



Classe SinglyLinkedList<E> (31)

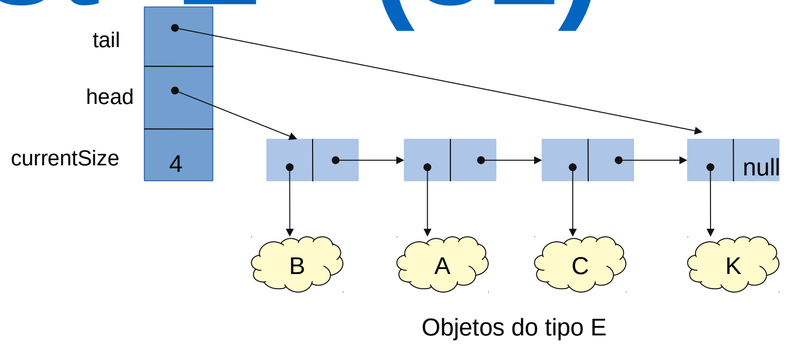


removeLast()

```
/**
 * 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() {
    if ( this.isEmpty() )
        throw new NoSuchElementException();
```

```
    return element;
}
```

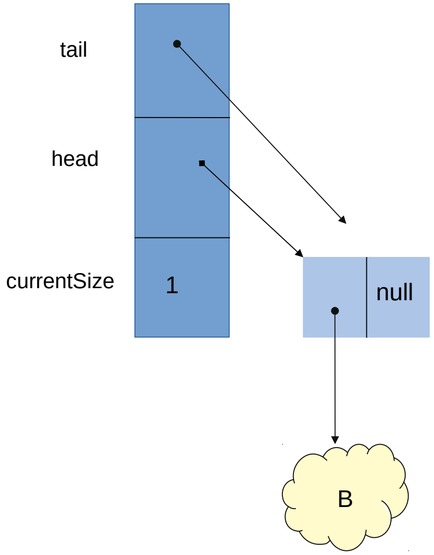
Classe SinglyLinkedList<E> (31)




```
/**
 * 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() {
    if ( this.isEmpty() )
        throw new NoSuchElementException();
    if ( size() == 1 )
        return removeFirst();



    return element;
}
```

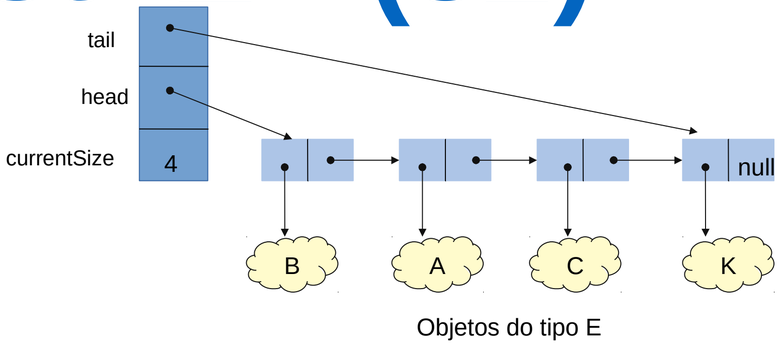
removeLast()

Lista com um elemento



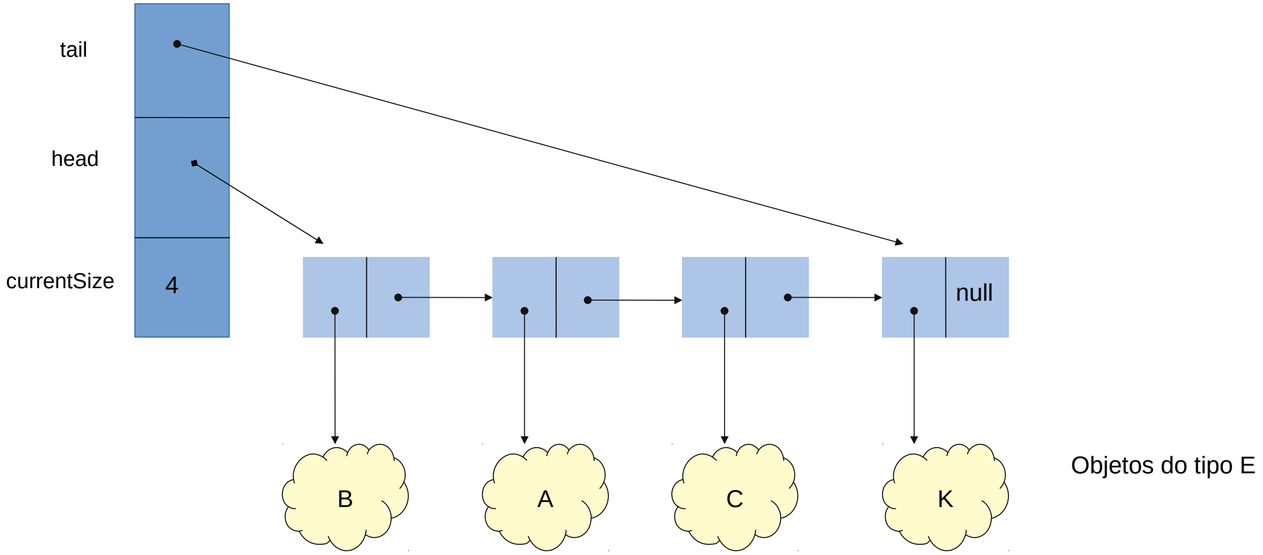
Objetos do tipo E

Classe SinglyLinkedList<E> (32)



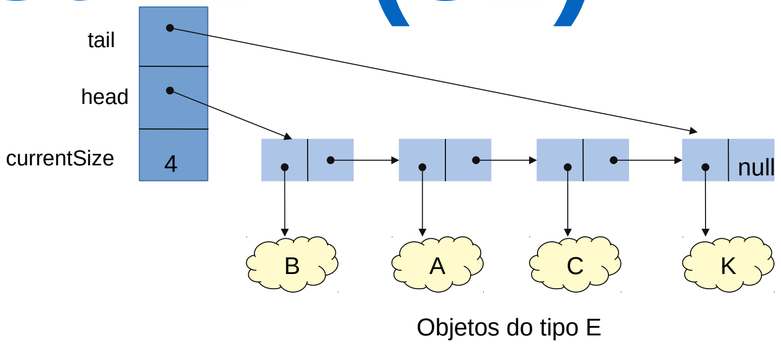

```
/**
 * Removes and returns the element at the last position in the list.
 *
 * @return element removed from the last position of the list
 * @throws NoSuchElementException - if size() == 0
 */

public E removeLast() {
    if ( this.isEmpty() )
        throw new NoSuchElementException();
    if ( size() == 1 )
        return removeFirst();



    return element;
}
```

removeLast()

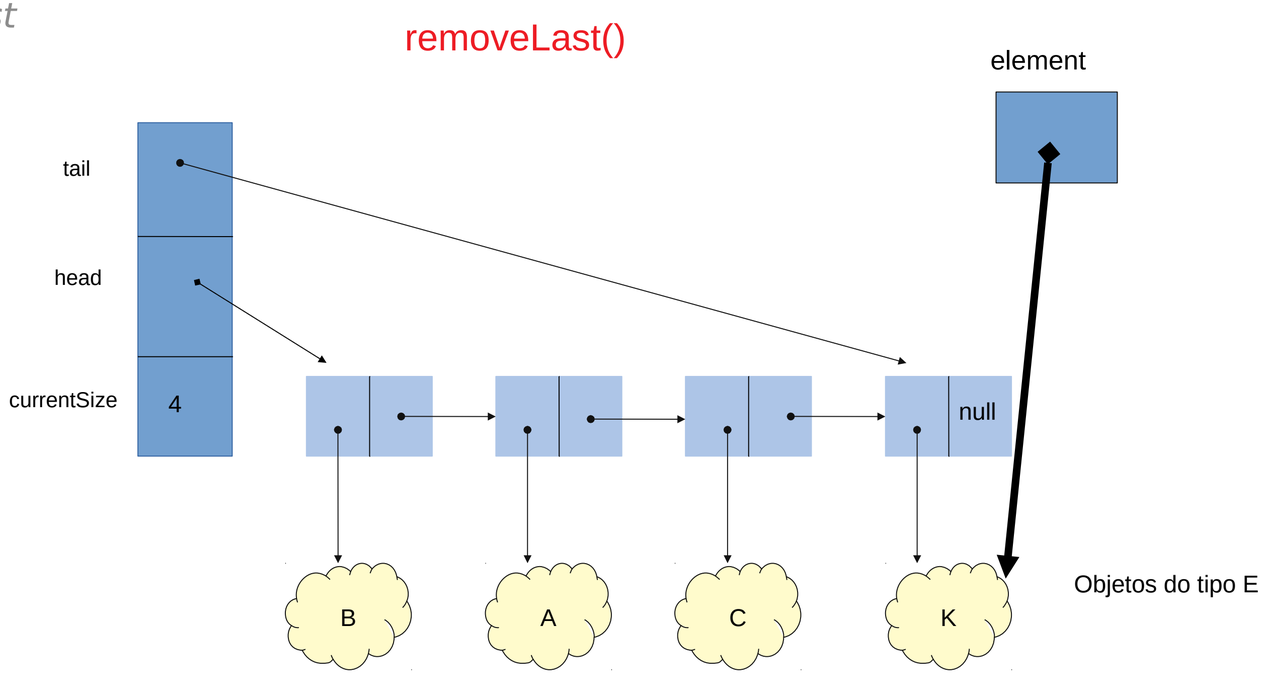


Lista com vários elementos

Classe SinglyLinkedList<E> (32)

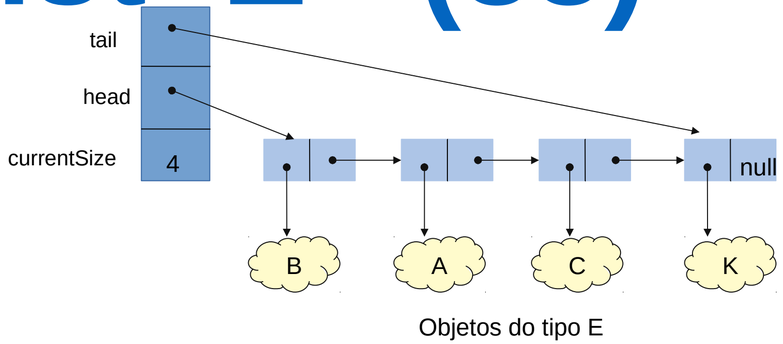



```
/**
 * Removes and returns the element at the last position in the list.
 *
 * @return element removed from the last position of the list
 * @throws NoSuchElementException - if size() == 0
 */

public E removeLast() {
    if ( this.isEmpty() )
        throw new NoSuchElementException();
    if ( size() == 1 )
        return removeFirst();

    E element=tail.getElement();



    return element;
}
```

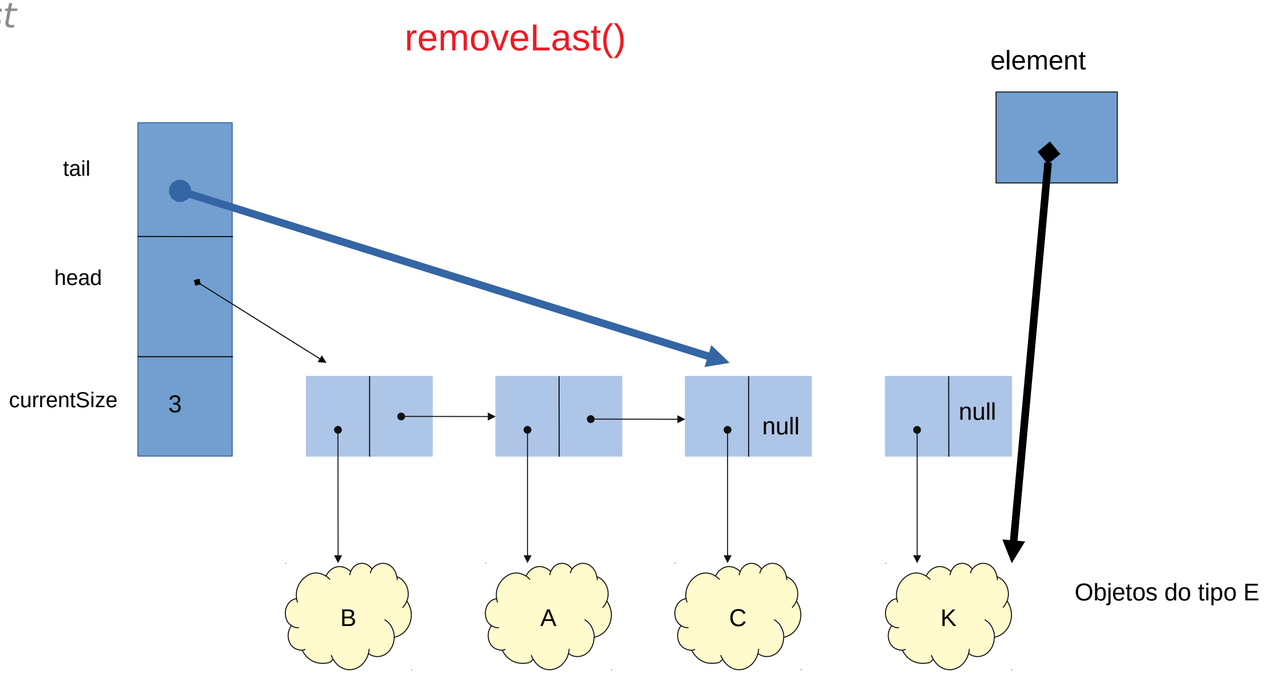


Lista com vários elementos

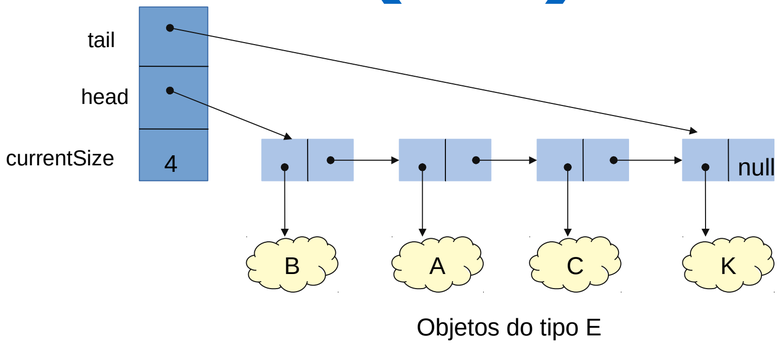
Classe SinglyLinkedList<E> (33)



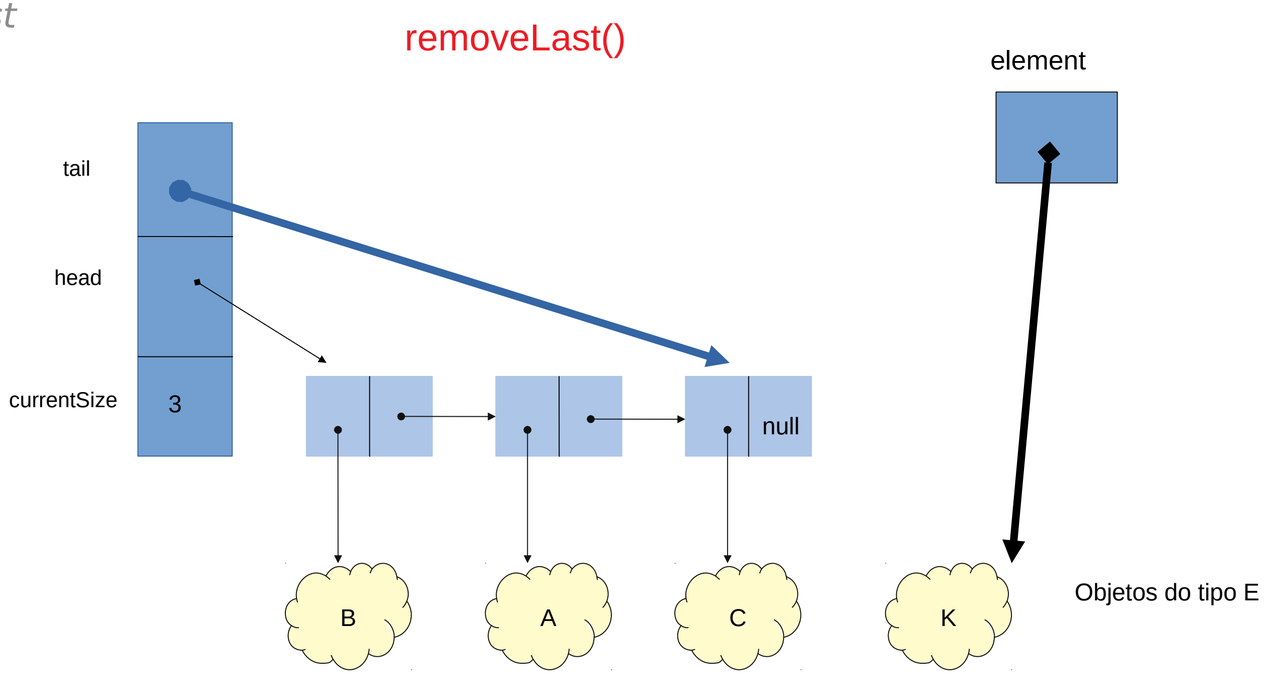

```
/**
 * Removes and returns the element at the last position in the list.
 *
 * @return element removed from the last position of the list
 * @throws NoSuchElementException - if size() == 0
 */

public E removeLast() {
    if ( this.isEmpty() )
        throw new NoSuchElementException();
    if ( size() == 1 )
        return removeFirst();

    E element=tail.getElement();
    tail = getNode(size() - 2);
    tail.setNext(null);
    currentSize--;

    return element;
}
```



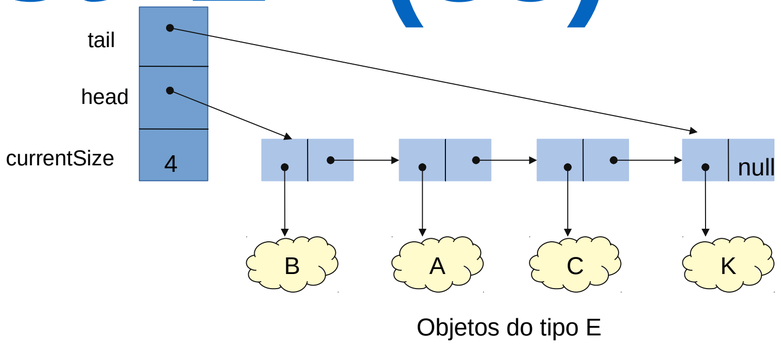
Classe SinglyLinkedList<E> (34)




```
/**
 * 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() {
    if ( this.isEmpty() )
        throw new NoSuchElementException();
    if ( size() == 1 )
        return removeFirst();

    E element=tail.getElement();
    tail = getNode(size() - 2);
    tail.setNext(null);
    currentSize--;

    return element;
}
```



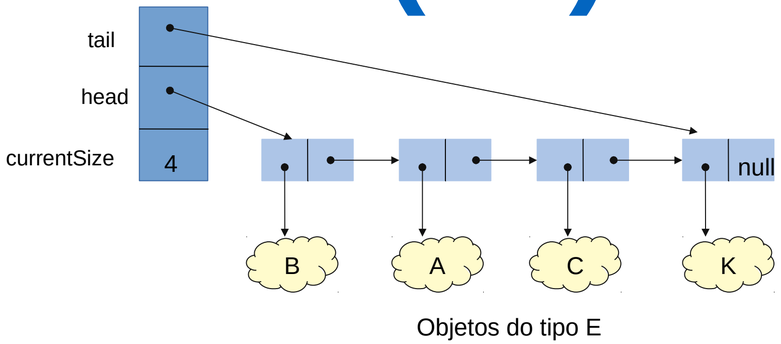
Classe SinglyLinkedList<E> (35)



remove(2)

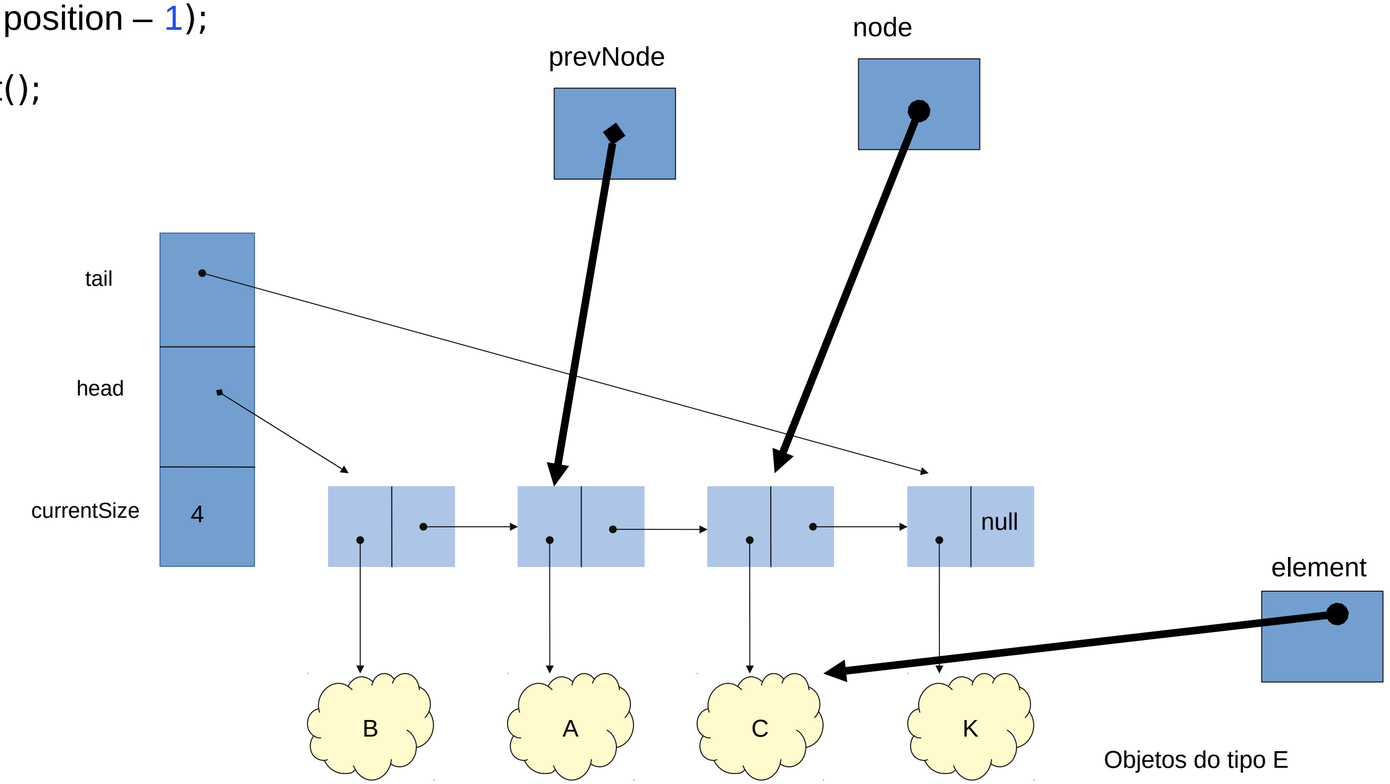
```
/**
 * Removes and returns the element at the specified position in the list.
 * Range of valid positions: 0, ..., size()-1.
 * If the specified position is 0, remove corresponds to removeFirst.
 * If the specified position is size()-1, remove corresponds to removeLast.
 *
 * @param position - position of element to be removed
 * @return element removed at position
 * @throws InvalidPositionException - if position is not valid in the list
 */
@Override
public E remove(int position) {
    if ( position < 0 || position >= currentSize )
        throw new InvalidPositionException();
    if ( position == 0 )
        return this.removeFirst();
    if ( position == currentSize - 1 )
        return this.removeLast();
    return this.removeMiddle(position);
}
```

Classe SinglyLinkedList<E> (36)

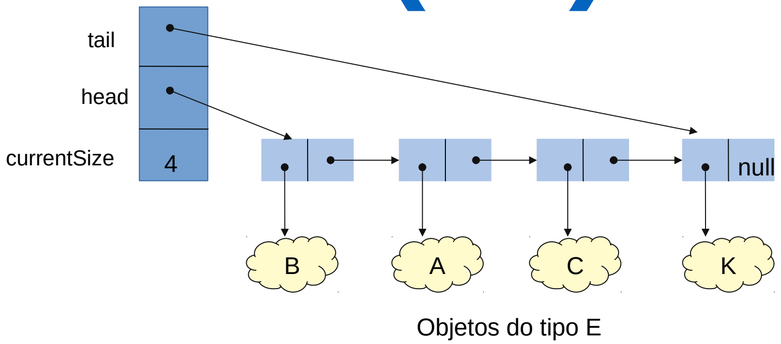



```
private E removeMiddle(int position) {
    SinglyListNode<E> prevNode = this.getNode(position – 1);
    SinglyListNode<E> node = prevNode.getNext();


    return node.getElement();
}
```

remove(2)

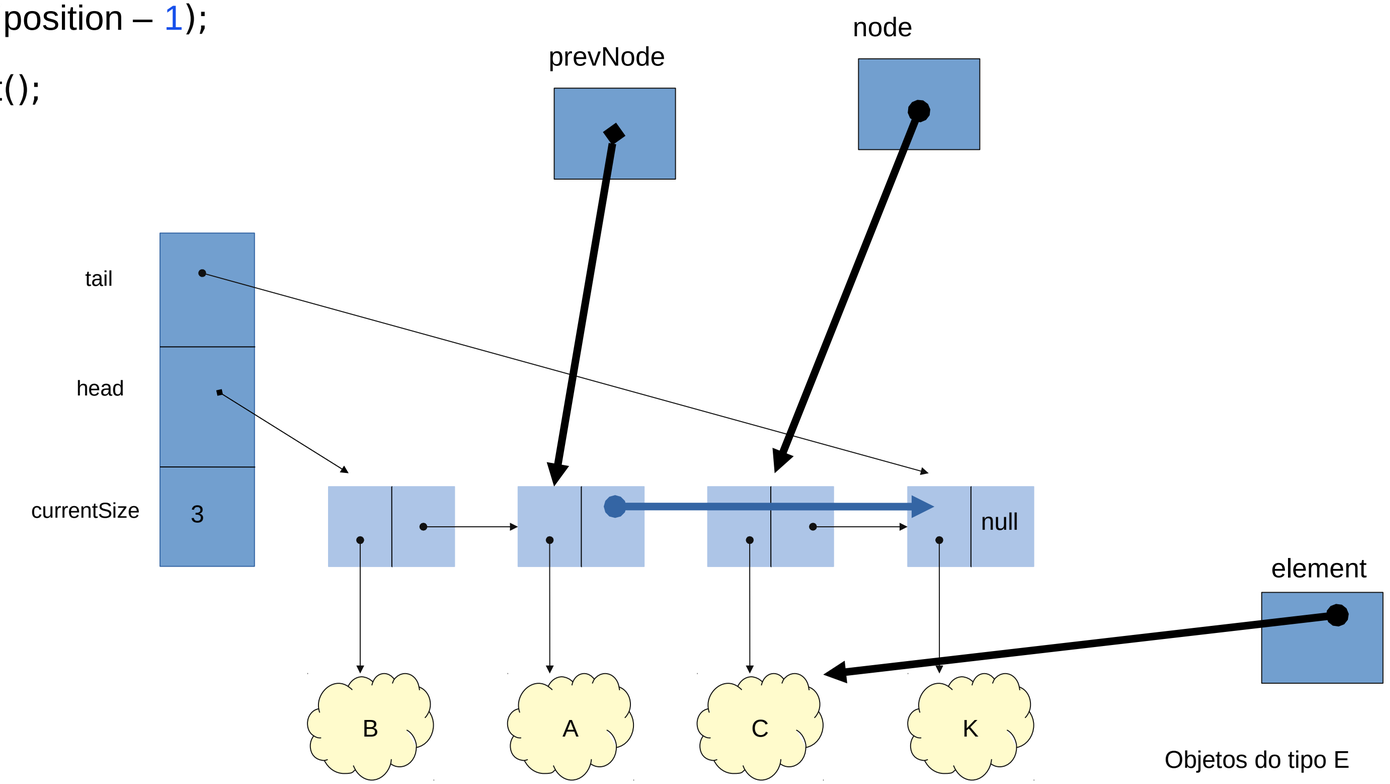


Classe SinglyLinkedList<E> (37)

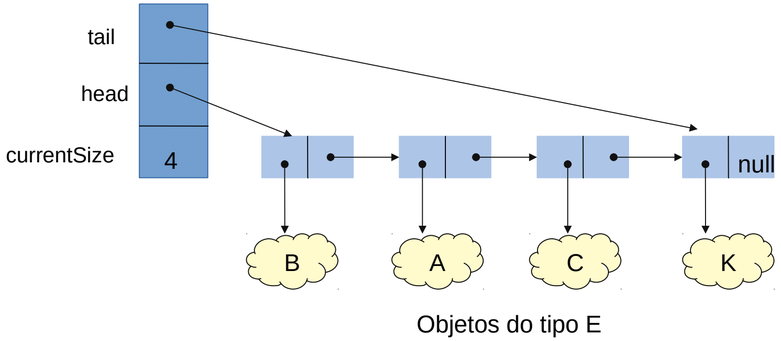



```
private E removeMiddle(int position) {
    SinglyListNode<E> prevNode = this.getNode(position – 1);
    SinglyListNode<E> node = prevNode.getNext();
    prevNode.setNext(node.getNext());
    currentSize--;
    return node.getElement();
}
```

remove(2)

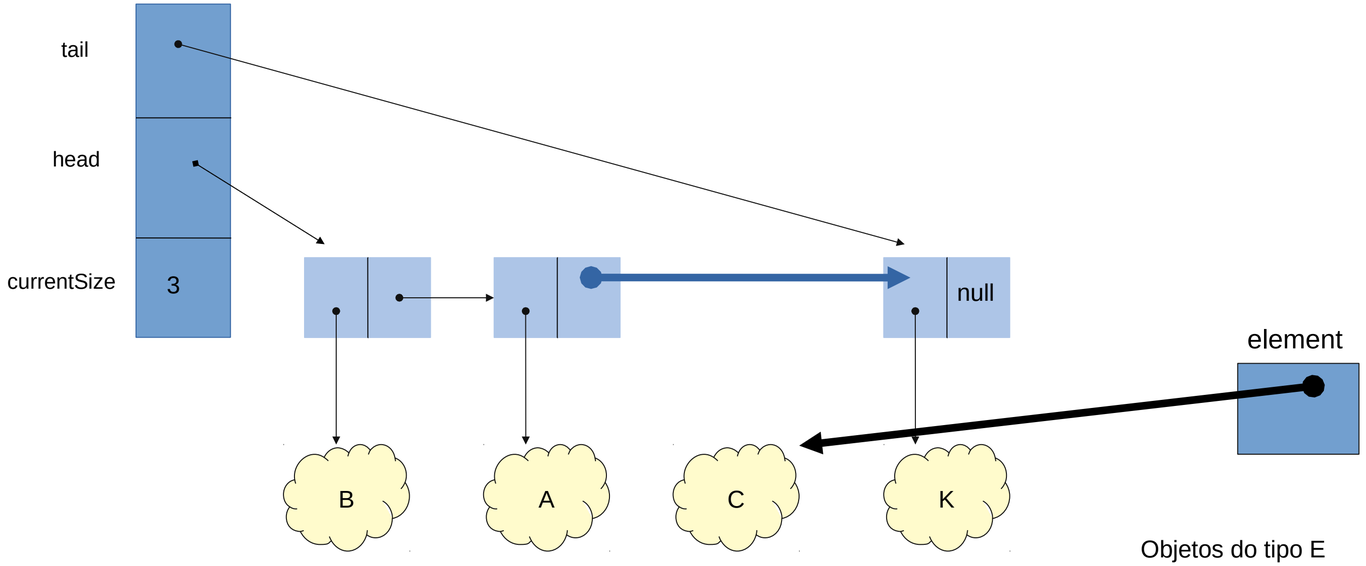


Classe SinglyLinkedList<E> (38)




```
private E removeMiddle(int position) {
    SinglyListNode<E> prevNode = this.getNode(position – 1);
    SinglyListNode<E> node = prevNode.getNext();
    prevNode.setNext(node.getNext());
    currentSize--;
    return node.getElement();
}
```

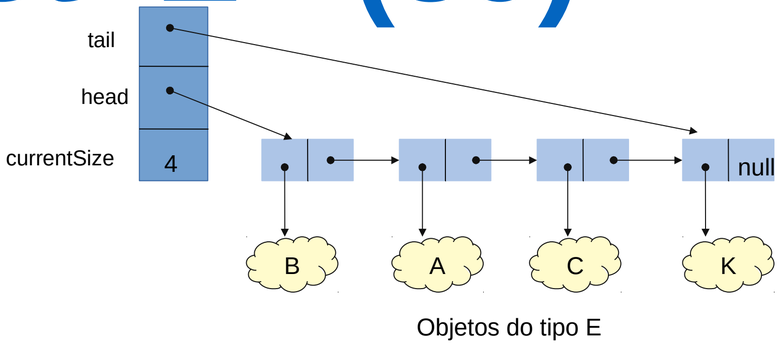
remove(2)



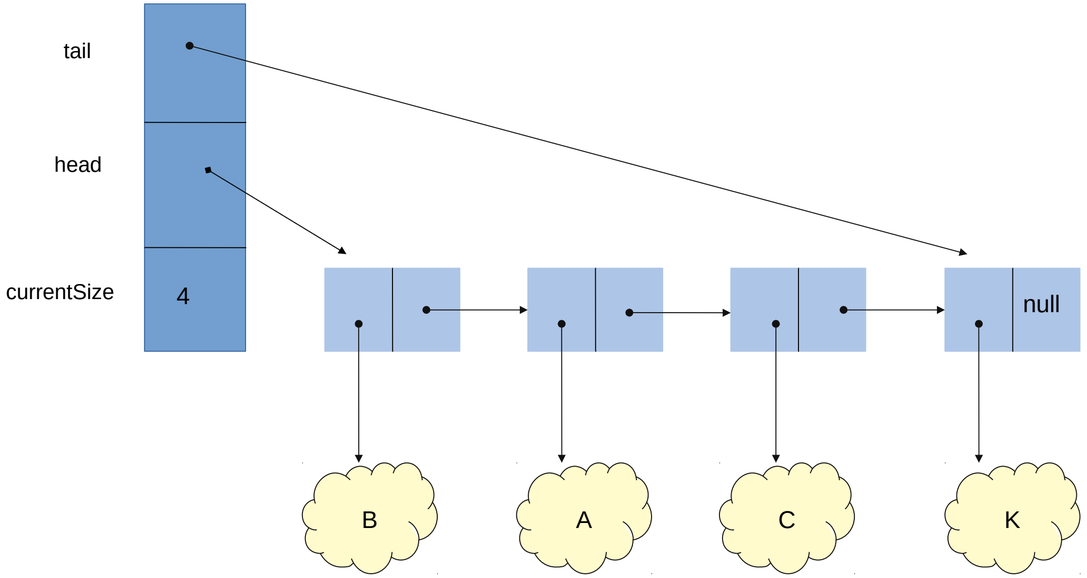
Lista Simplesmente Ligada

Operação	Melhor Caso	Pior Caso	Caso Médio
`isEmpty, size`	$O(1)$	$O(1)$	$O(1)$
`getFirst, getLast`	$O(1)$	$O(1)$	$O(1)$
`get`	$O(1)$	$O(n)$	$O(n)$
`addFirst, addLast`	$O(1)$	$O(1)$	$O(1)$
`add`	$O(1)$	$O(n)$	$O(n)$
`removeFirst`	$O(1)$	$O(1)$	$O(1)$
`removeLast`	$O(n)$	$O(n)$	$O(n)$
`remove`	$O(1)$	$O(n)$	$O(n)$
`indexOf (por elemento)`	$O(1)$	$O(n)$	$O(n)$
`iterator`			

Classe SinglyLinkedList<E> (39)




```
/**
 * 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 SinglyIterator<>(head);
}
```



Classe SinglyLinkedList

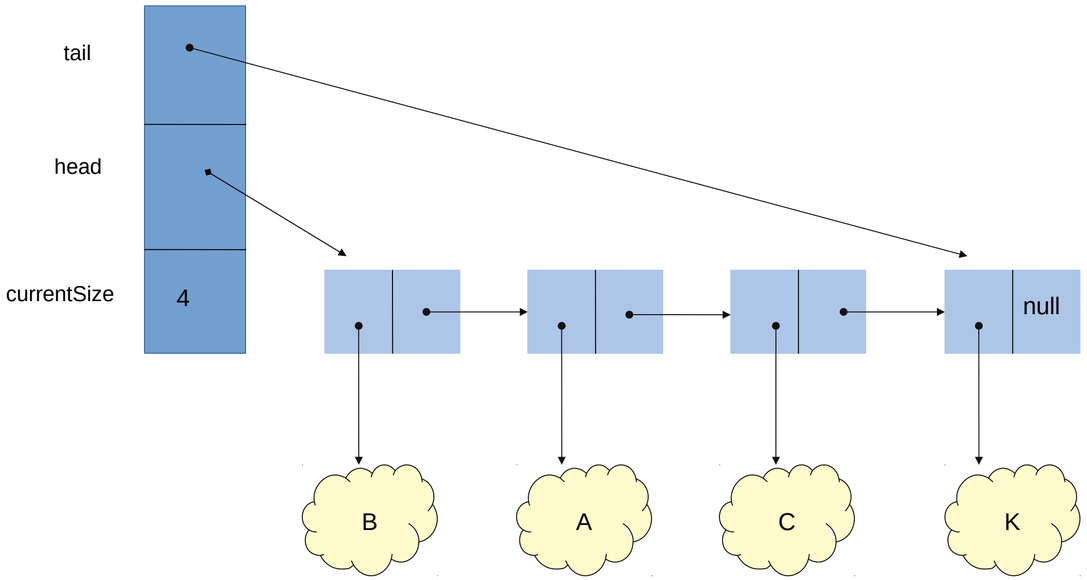
Operação	Melhor Caso	Pior Caso	Caso Médio
`isEmpty, size`	$O(1)$	$O(1)$	$O(1)$
`getFirst, getLast`	$O(1)$	$O(1)$	$O(1)$
`get`	$O(1)$	$O(n)$	$O(n)$
`addFirst, addLast`	$O(1)$	$O(1)$	$O(1)$
`add`	$O(1)$	$O(n)$	$O(n)$
`removeFirst`	$O(1)$	$O(1)$	$O(1)$
`removeLast`	$O(n)$	$O(n)$	$O(n)$
`remove`	$O(1)$	$O(n)$	$O(n)$
`indexOf (por elemento)`	$O(1)$	$O(n)$	$O(n)$
`iterator`	$O(1)$	$O(1)$	$O(1)$

A complexidade espacial da lista ligada simples é linear O(n)

Como funciona o iterador (1)

rewind()	
next()	
hasNext()	true

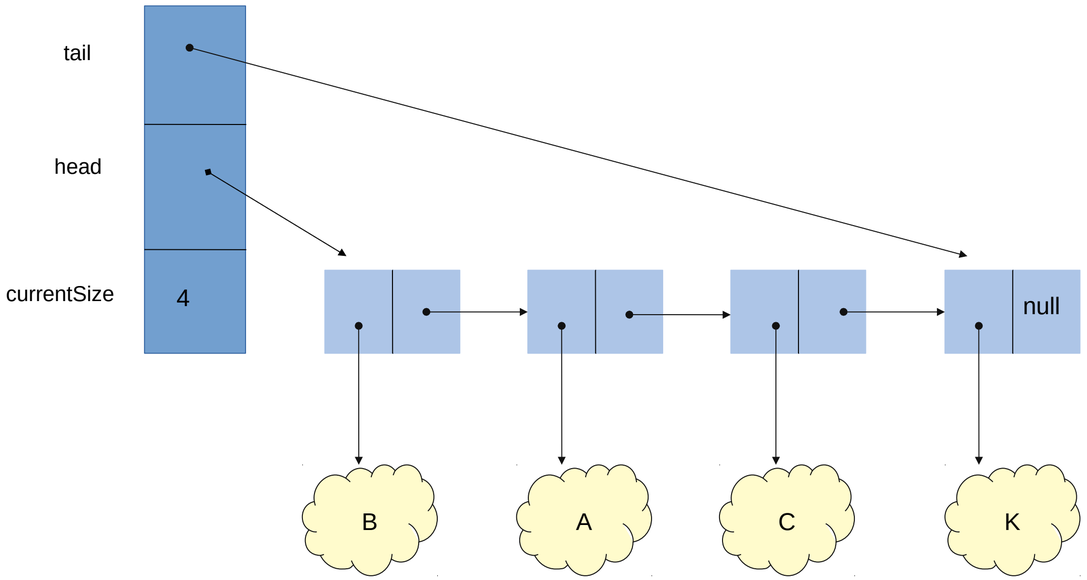
rewind()	
next()	B
hasNext()	



Como funciona o iterador (2)

rewind()	
next()	
hasNext()	true

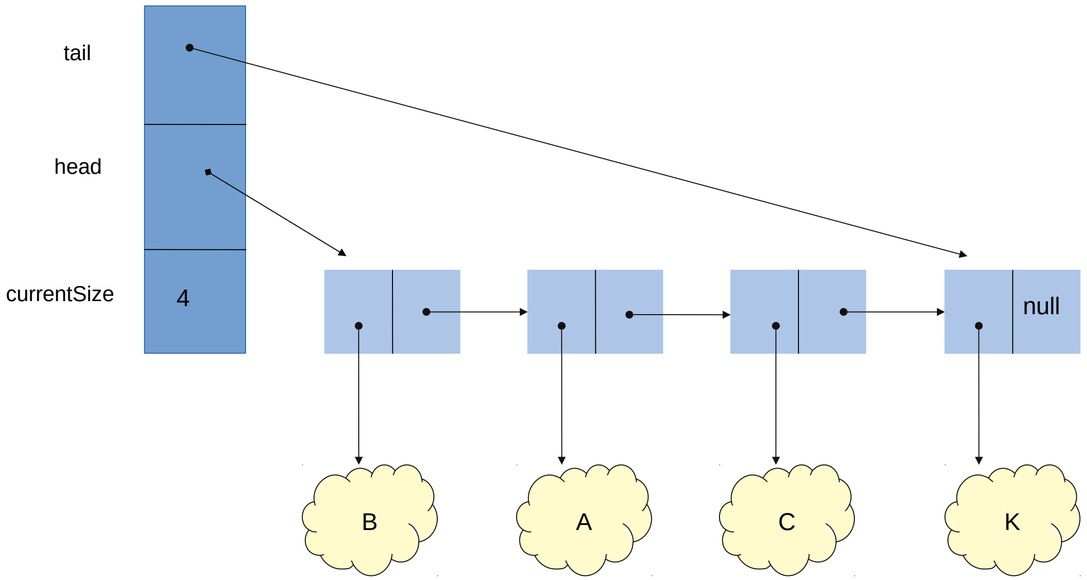
rewind()	
next()	A
hasNext()	



Como funciona o iterador (3)

rewind()	
next()	
hasNext()	true

rewind()	
next()	C
hasNext()	

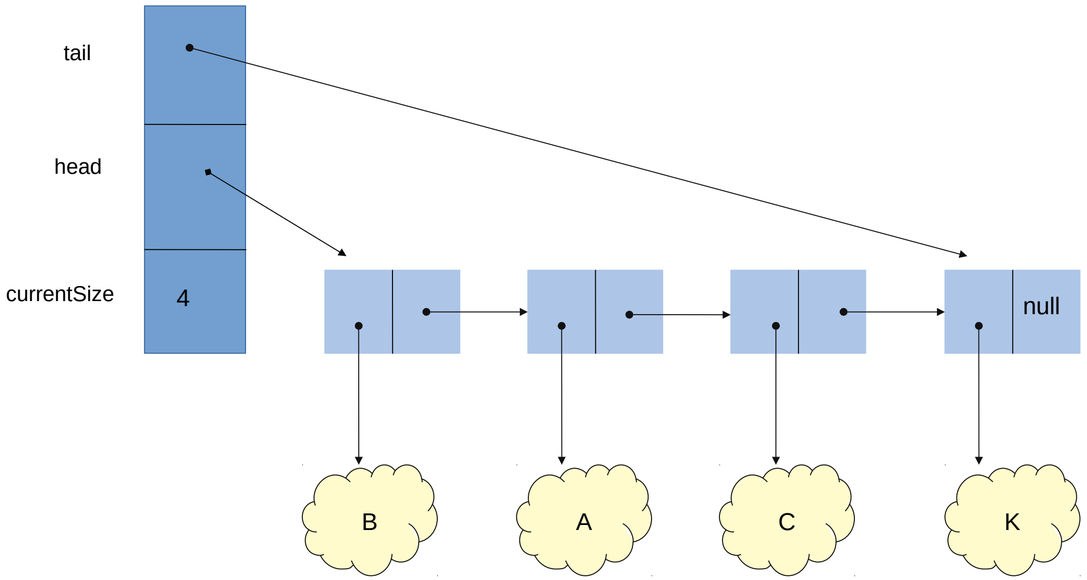


Como funciona o iterador (4)

rewind()	
next()	
hasNext()	true

rewind()	
next()	K
hasNext()	

rewind()	
next()	
hasNext()	false

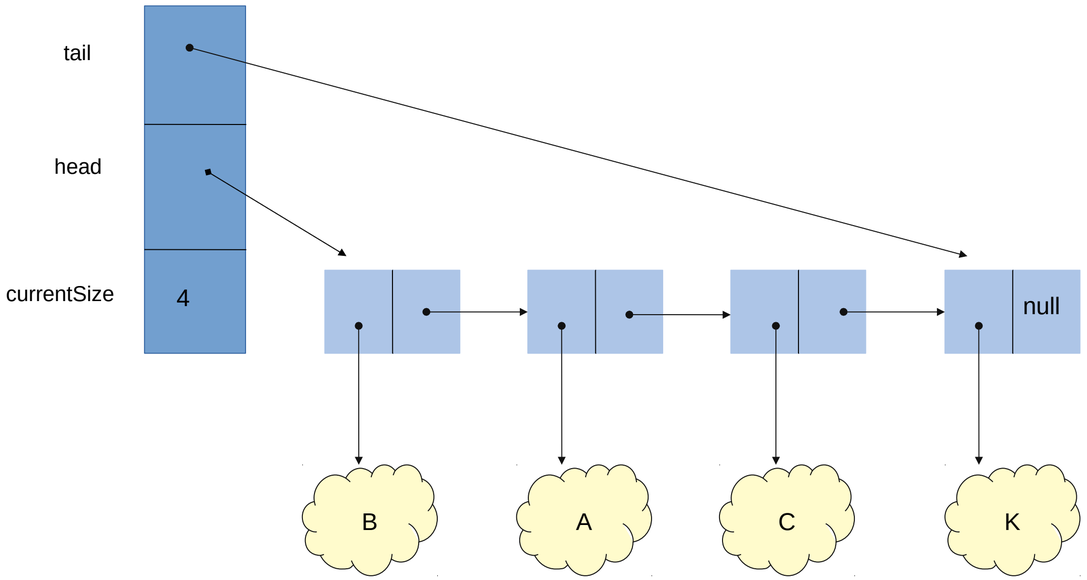


Como funciona o iterador (5)

rewind()	
next()	
hasNext()	

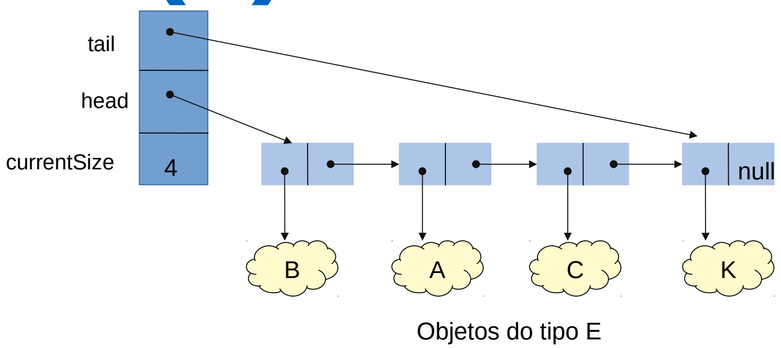
rewind()	
next()	
hasNext()	true

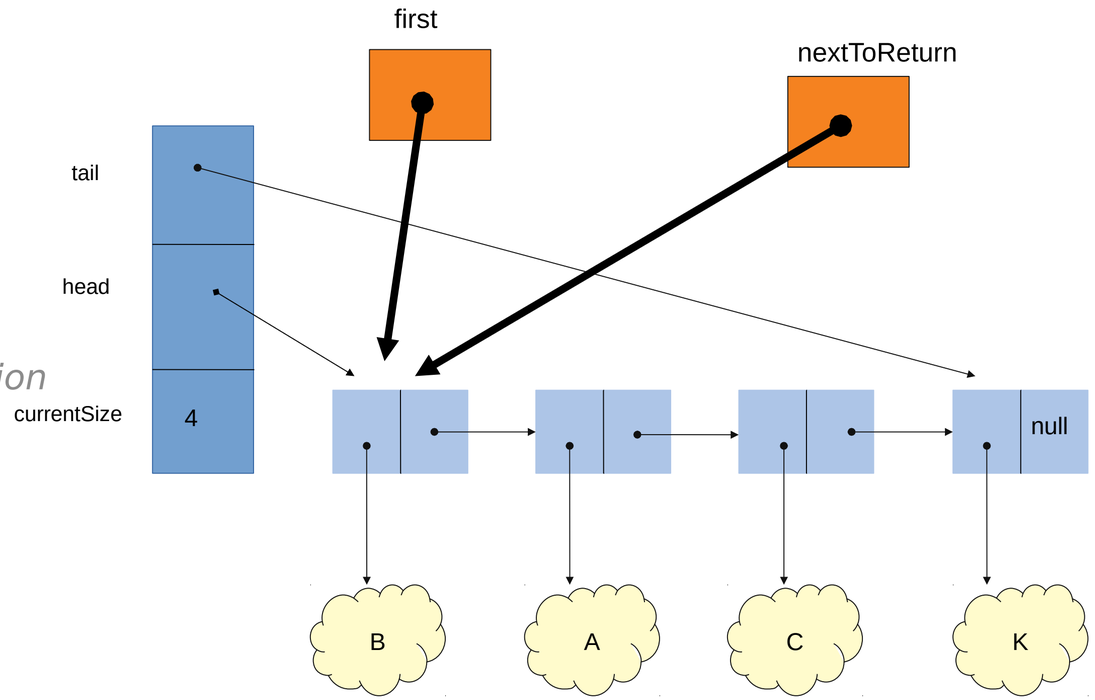
rewind()	
next()	B
hasNext()	



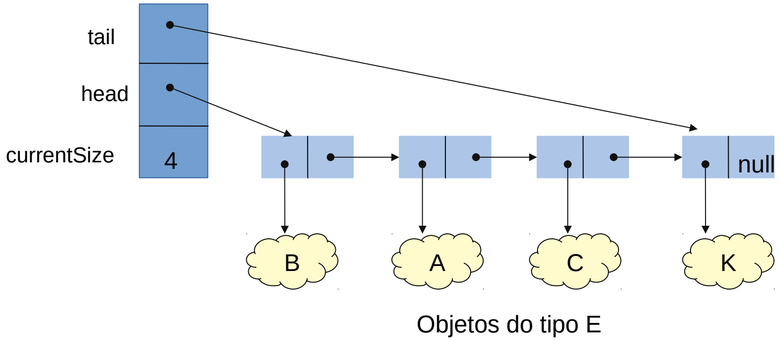
Classe SinglyIterator<E> (1)

```
package dataStructures;
import dataStructures.exceptions.NoSuchElementException;

class SinglyIterator<E>  implements Iterator<E> {
    /**
     * First node of the list.
     */
    private final SinglyListNode<E> first;
    /**
     * Node with the next element in the iteration.
     */
    private SinglyListNode<E> nextToReturn;

    /**
     * SinglyIterator constructor
     * @param first - Node with the first element of the iteration
     */
    public SinglyIterator(SinglyListNode<E> first) {
        this.first=first;
        nextToReturn=first;
    }

    public boolean hasNext( ) {
        return nextToReturn != null;
    }
```



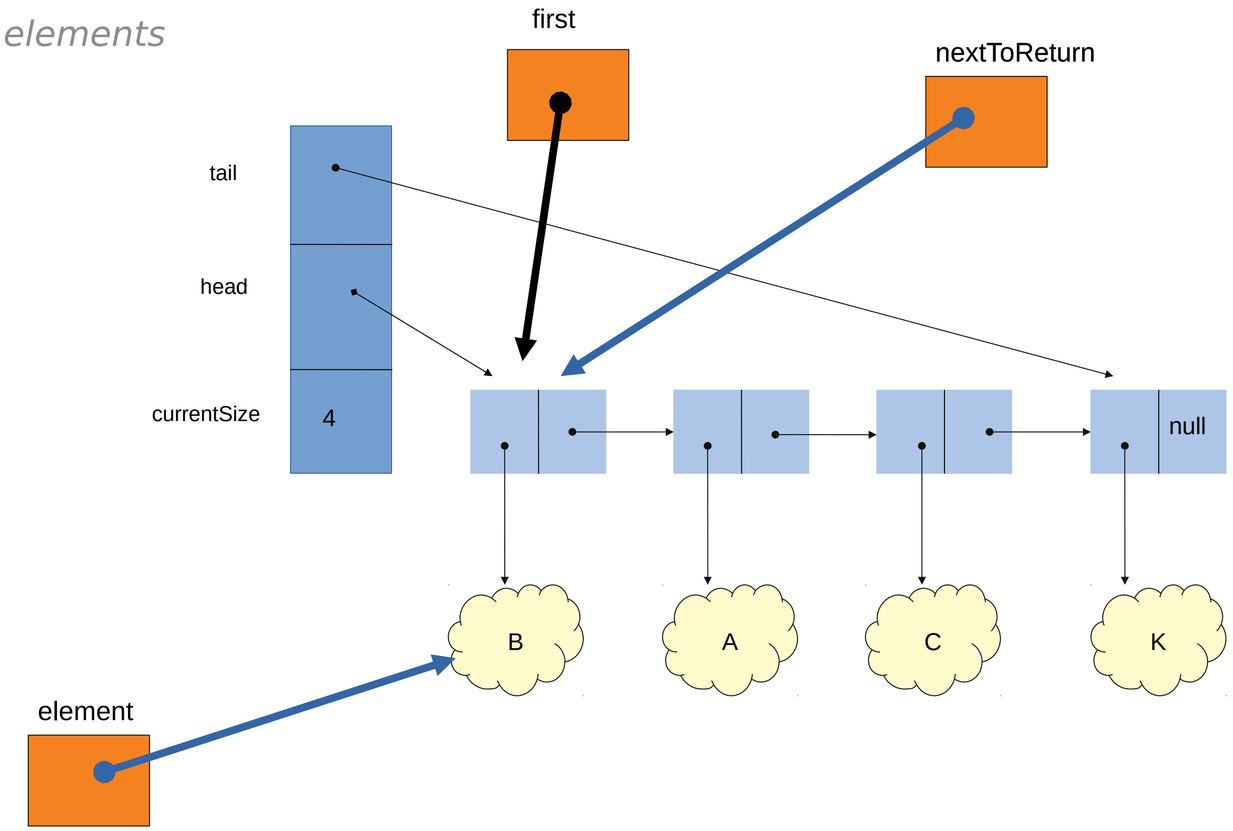


Classe SinglyIterator<E> (2)

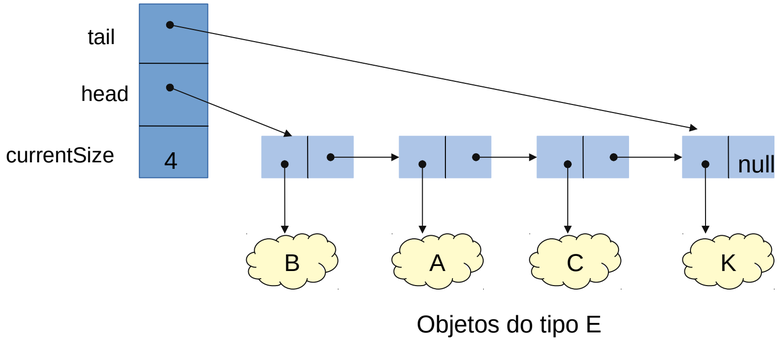



```
/**
    * Returns the next element in the iteration.
    * @return the next element in the iteration
    * @throws NoSuchElementException - if no more elements
    */

   public E next( ){

      if ( !this.hasNext() )
         throw new NoSuchElementException();

      E element = nextToReturn.getElement();

      return element;
   }

}
```

next()

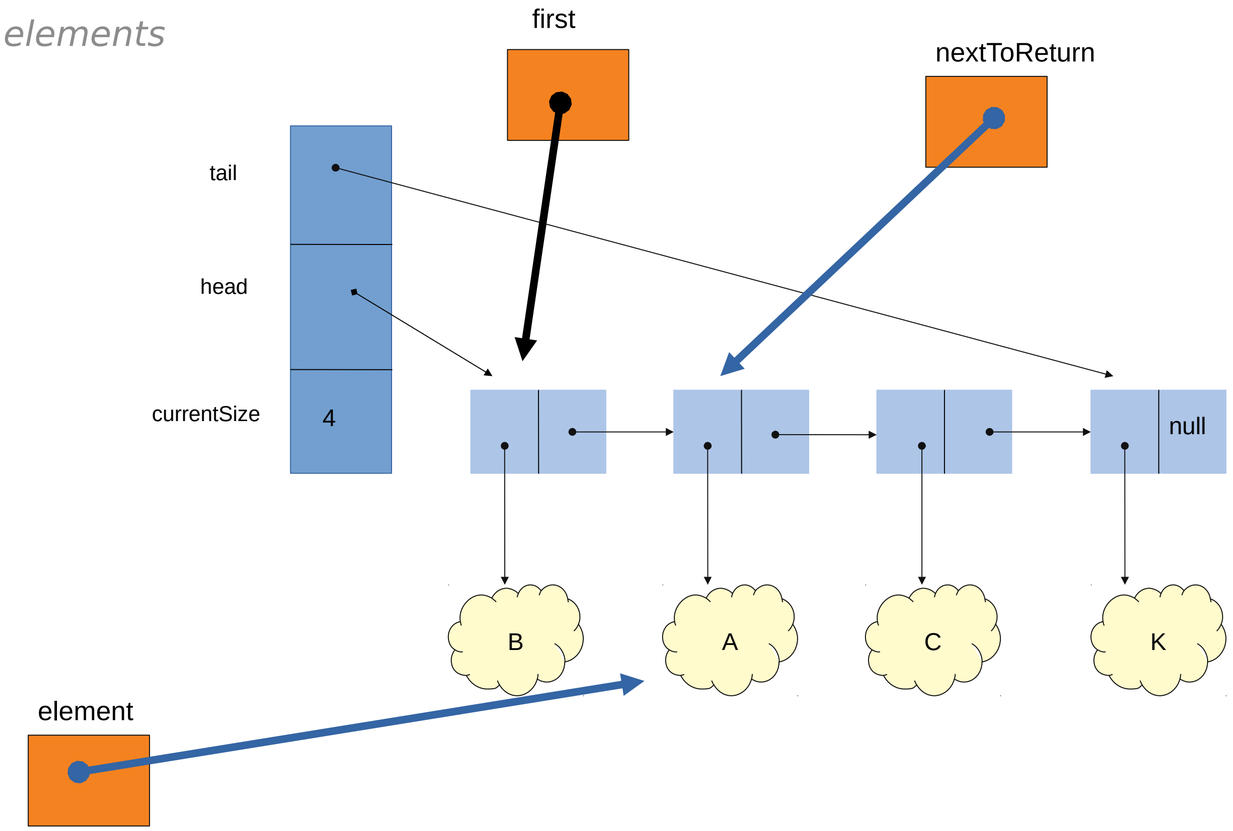


Classe SinglyIterator<E> (3)

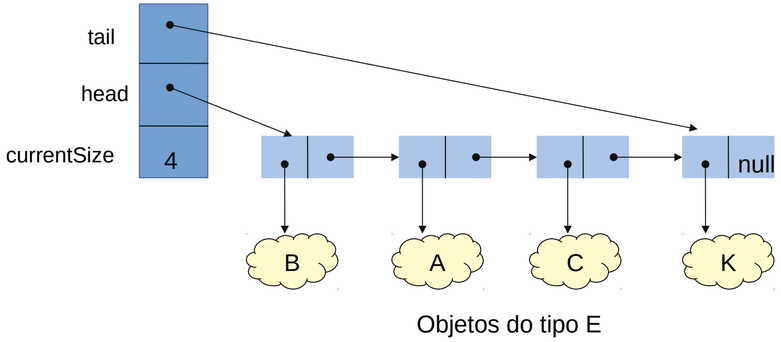



```
/**
    * Returns the next element in the iteration.
    * @return the next element in the iteration
    * @throws NoSuchElementException - if no more elements
    */

   public E next( ){

      if ( !this.hasNext() )
         throw new NoSuchElementException();

      E element = nextToReturn.getElement();
      nextToReturn = nextToReturn.getNext();
      return element;
   }

}
```

next()

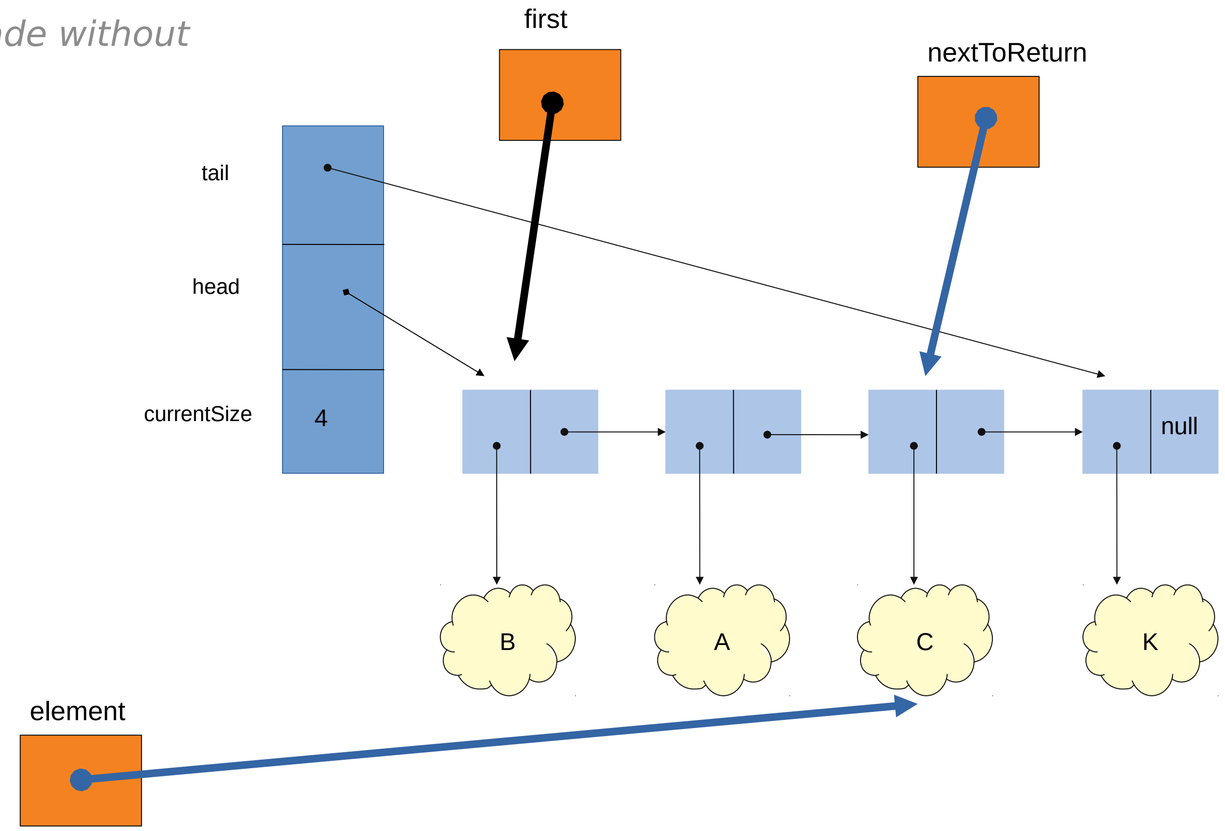


Classe SinglyIterator<E> (4)

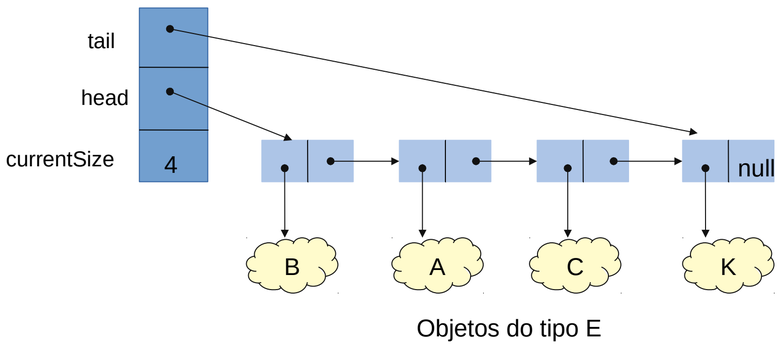



```
/**
    * Returns the next element in the iteration.
    * @return the next element in the iteration
    * @throws NoSuchElementException - if call is made without
    * verifying precondition
    */

   public E next( ){

      if ( !this.hasNext() )
         throw new NoSuchElementException();

      E element = nextToReturn.getElement();
      nextToReturn = nextToReturn.getNext();
      return element;
   }

}
```

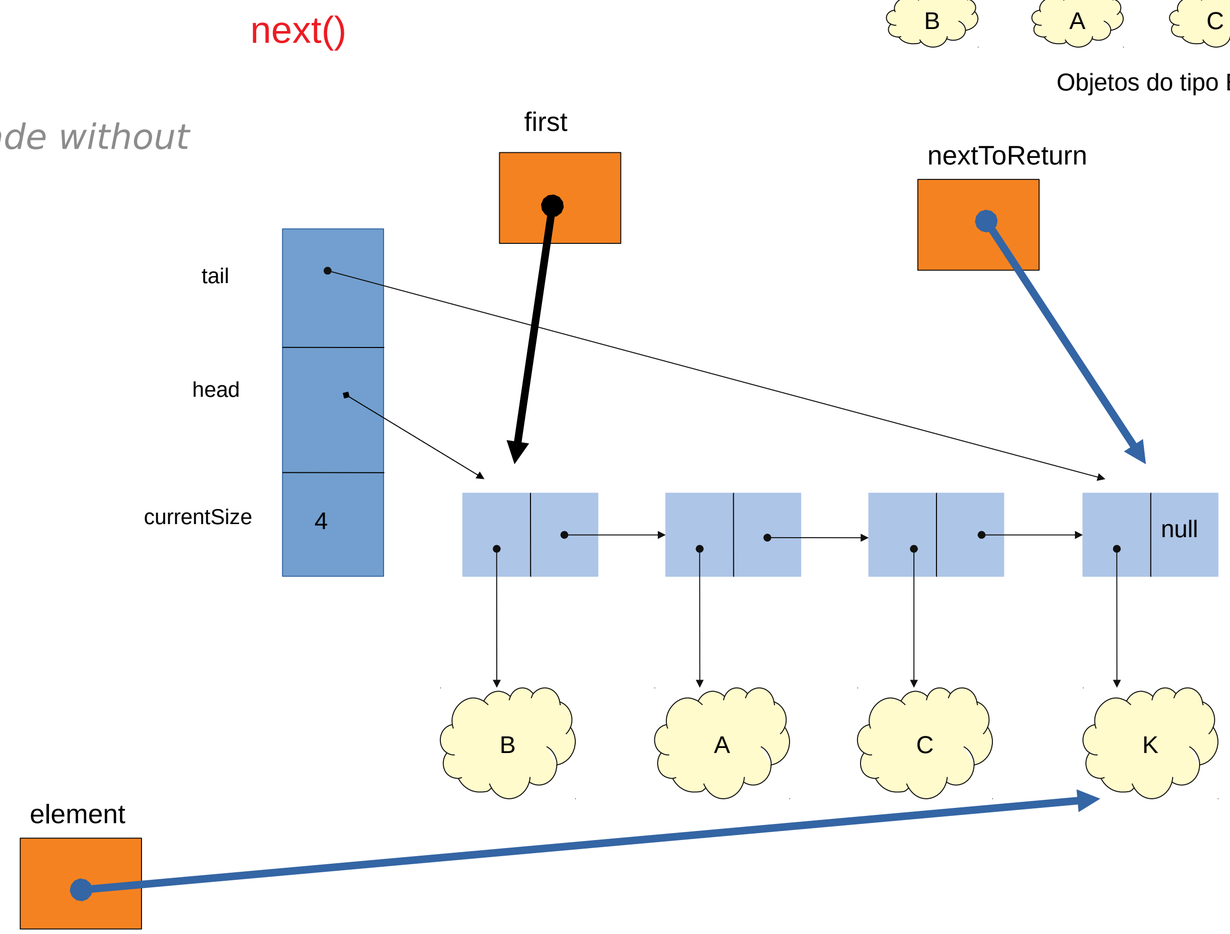
next()



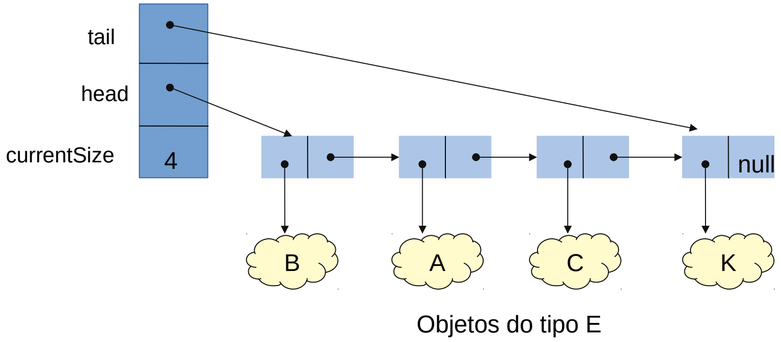
Classe SinglyIterator<E> (5)



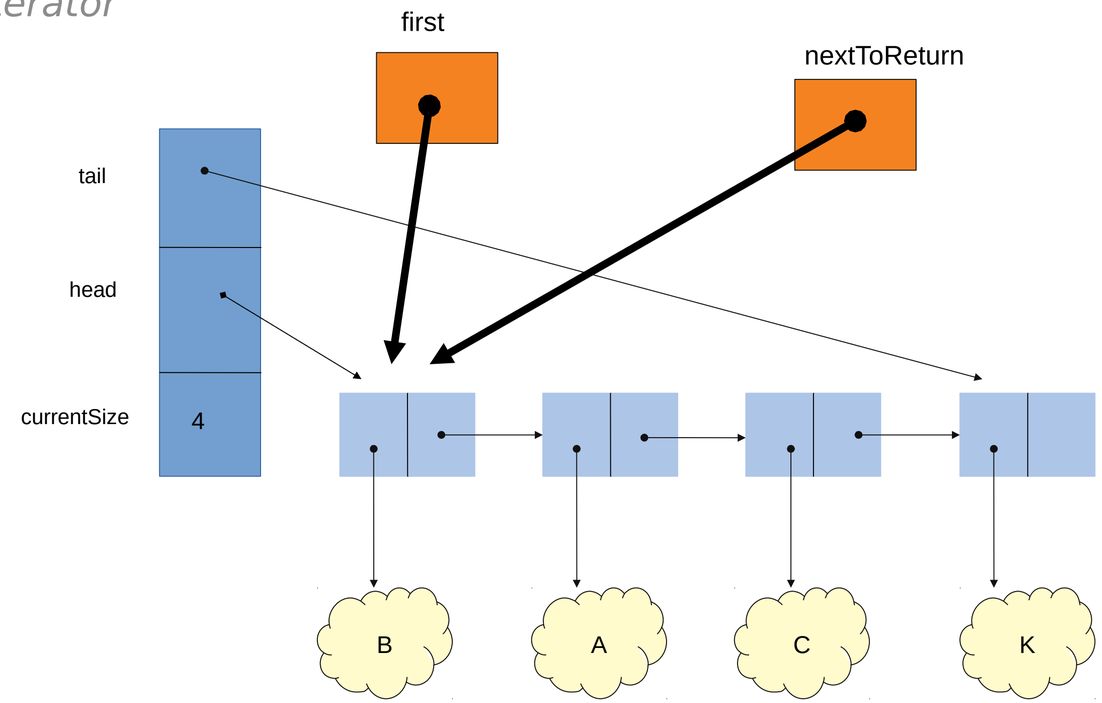

```
/**
    * Returns the next element in the iteration.
    * @return the next element in the iteration
    * @throws NoSuchElementException - if call is made without
    * verifying precondition
    */

   public E next( ){

      if ( !this.hasNext() )
         throw new NoSuchElementException();

      E element = nextToReturn.getElement();
      nextToReturn = nextToReturn.getNext();
      return element;
   }

}
```



Classe SinglyIterator<E> (6)




```
    /**
     * Restarts the iteration.
    * After rewind, next will return the first element the iterator
     */

    public void rewind() {

        nextToReturn=first;
    }

}
```



Classe SinglyIterator

Operação	Melhor Caso	Pior Caso	Caso Médio
`hasNext`	O(1)	O(1)	O(1)
`next`	O(1)	O(1)	O(1)
`rewind`	O(1)	O(1)	O(1)

A complexidade espacial do iterador da lista ligada simples é constante O(1)

Diagrama de Classes