



Algoritmos e Estruturas de Dados

Estruturas de Dados – Capítulo 3
2019/20

Estruturas de Dados

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

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

Classe **Array** (1)

```
package dataStructures;
```

Esta classe foi feita em POO

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

```
    private static final int DEFAULT_SIZE = 50;
```

```
    /** 0 array generico
     */
```

```
    protected E [] array;
```

Atributos acessíveis no package

```
    /** 0 numero de elementos actual
     */
```

```
    protected int counter;
```

```
    ...
```

```
}
```

Classe Array (2)

```
package dataStructures;
```

array



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

```
...
```

counter

0

```
/**Construtor que define um array com uma dada dimensao inicial */
```

```
@SuppressWarnings("unchecked")
```

```
public Array(int capacity) {  
    array = (E []) new Object[capacity];  
    counter = 0;  
}
```

```
public Array() {  
    this(DEFAULT_SIZE);  
}
```

```
...  
}
```

O aviso vem do facto de não ser possível verificar, em tempo de compilação, os tipos dos dados que vão ser inseridos no vetor.

Para submissão no Mooshak, utilizar @SuppressWarnings

Compiler gives a warning.

Classe Array (3)

4	6	78	12				
---	---	----	----	--	--	--	--

```
package dataStructures;
```

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

```
...  
public void addLast(E elem) {  
    if (isFull()) resize();  
    array[counter++] = elem;  
}
```

insertLast(45)

4	6	78	12	45			
---	---	----	----	----	--	--	--

```
public void add(int pos, E elem) throws InvalidPositionException{  
    if (pos < 0 || pos > counter)  
        throw new InvalidPositionException("Invalid Position.");  
    if (isFull()) resize();  
    for(int i = counter-1; i >= pos; i--)  
        array[i+1] = array[i];  
    array[pos] = elem;  
    counter++;  
}
```

insertAt(42,1)

	4	6	78	12	45		
i=4	4	6	78	12		45	
i=3	4	6	78		12	45	
i=2	4	6		78	12	45	
i=1	4		6	78	12	45	
	4	42	6	78	12	45	

```
public void addFirst(E elem) {  
    if (isFull()) resize();  
    add(0, elem);  
}  
...  
}
```

Classe Array (4)

```
package dataStructures;
public class Array<E> implements List<E>{
    ...
    /** Metodo auxiliar para duplicar o tamanho do vector. */
    @SuppressWarnings("unchecked")
    private void resize() {
        E[] tmp = (E []) new Object[counter * 2];
        for (int i = 0; i < counter; i++) tmp[i] = array[i];
        array = tmp;
    }
    @Override
    public int size() {
        return counter;
    }
    @Override
    public boolean isEmpty() {
        return counter==0;
    }
    public boolean isFull() {
        return counter == array.length;
    }
    public int capacity() {
        return array.length;
    }
    ...
}
```

Métodos (*isFull* e *capacity*) da classe Array<E>, Mas não da interface List<E>

Classe **Array** (5)

```
package dataStructures;
```

4	6	78	12	45	56	60	
---	---	----	----	----	----	----	--

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

```
...
```

```
    public E removeLast() throws NoSuchElementException{  
        if (counter==0)  
            throw new NoSuchElementException("No such element.");  
        return array[--counter];
```

```
    }
```

```
...
```

```
}
```

removeLast()

4	6	78	12	45	56		
---	---	----	----	----	----	--	--

Classe Array (6)

```
package dataStructures;
```

4	6	78	12	45	56	60	
---	---	----	----	----	----	----	--

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

```
...
```

```
public E remove(int pos) throws InvalidPositionException{  
    if (pos<0 || pos >=counter)  
        throw new InvalidPositionException("Invalid position.");
```

```
    E elem = array[pos];
```

```
    for(int i = pos; i < counter-1; i++)
```

```
        array[i] = array[i+1];
```

```
    counter--;
```

```
    return elem;
```

```
}
```

removeAt(1) → 6

	4	6	78	12	45	56		
i=1	4	78	78	12	45	56		
i=2	4	78	12	12	45	56		
i=3	4	78	12	45	45	56		
i=4	4	78	12	45	56	56		
	4	78	12	45	56			

```
public E removeFirst() throws NoSuchElementException{
```

```
    if (counter==0)
```

```
        throw new NoSuchElementException("No such element.");
```

```
    return remove(0);
```

```
}
```

```
...
```

```
}
```


Classe **Array** (7)

```
package dataStructures;
```

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

```
    ...
```

```
    public E get(int pos) throws InvalidPositionException{  
        if (pos<0 || pos >=counter)  
            throw new InvalidPositionException("Invalid position.");  
        return array[pos];  
    }
```

```
    public E getFirst() throws NoElementException {  
        if (counter==0)  
            throw new NoElementException("No such element.");  
        return get(0);  
    }
```

```
    public E getLast() throws NoElementException {  
        if (counter==0)  
            throw new NoElementException("No such element.");  
        return get(counter-1);  
    }
```

```
    ...
```

```
}
```

Classe **Array** (8)

```
package dataStructures;
```

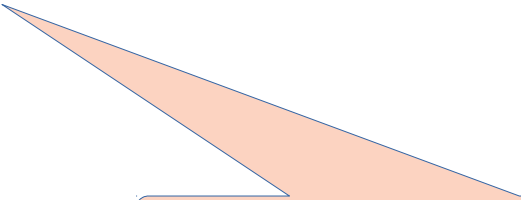
```
public class Array<E> implements List<E>{
```

```
...
```

```
    public Iterator<E> iterator() throws NoSuchElementException{  
        if (counter==0)  
            throw new NoSuchElementException("Array is empty.");  
        return new ArrayIterator<E>(array,counter);  
    }
```

```
...
```

```
}
```



Classe a implementar

Classe Array (9)

Pesquisa Sequencial

```
package dataStructures;

public class Array<E> implements List<E>{
    ...

    public int find(E elem) {
        boolean found = false;
        int i=0;
        while(i < counter && !found)
            if (array[i].equals(elem))
                found = true;
            else i++;
        if (found) return i;
        else return -1;
    }

    ...
}
```

Quando encontramos o elemento paramos.

Classe **ArrayIterator** (1)

```
package dataStructures;
```

Classe também realizada em POO

```
public class ArrayIterator<E> implements Iterator<E> {
```

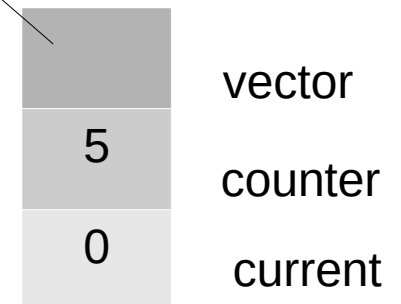
```
    private E[] vector;  
    private int counter;  
    private int current;
```

```
    public ArrayIterator(E[] vector, int counter) {  
        this.vector = vector;  
        this.counter = counter;  
        rewind();  
    }
```

```
    @Override  
    public void rewind() {  
        current=0;  
    }
```

```
    ...
```

```
}
```



Classe **ArrayIterator** (2)

```
package dataStructures;
```

```
public class ArrayIterator<E> implements Iterator<E> {
```

```
    ...
```

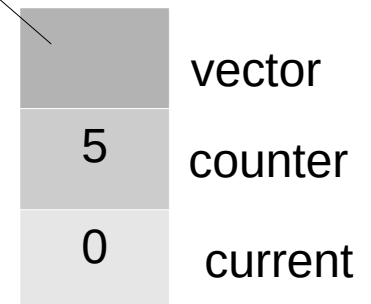
```
    @Override
```


```
    public boolean hasNext() {  
        return current < counter;  
    }
```

```
    @Override
```

```
    public E next() throws NoSuchElementException {  
        if (!hasNext())  
            throw new NoSuchElementException("No more elements.");  
        return vector[current++];  
    }
```

```
}
```





Estruturas de Dados estáticas vs dinâmicas

- **E. D. estáticas:**

- A capacidade tem que ser definida quando a estrutura de dados é criada;
- Inserções e eliminações em posições interiores, consome muito tempo de execução pois os elementos tem que ser mudados de posição;
- Aumentar a capacidade da estrutura de dados, implica criar novo espaço e mover todos os elementos.

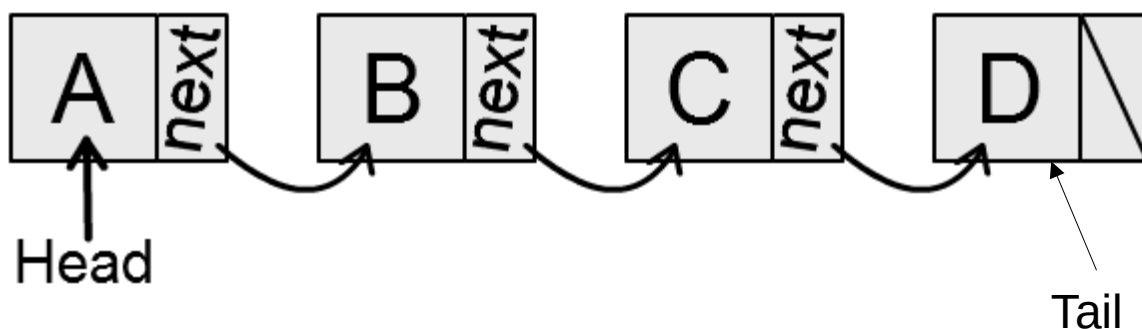
- **E. D. dinâmicas:**

- Colecção de nós que em conjunto formam uma sequência linear;
- Cada nó tem uma referência para o próximo elemento;
- A estrutura de dados só necessita de guardar o primeiro nó.

Listas ligadas

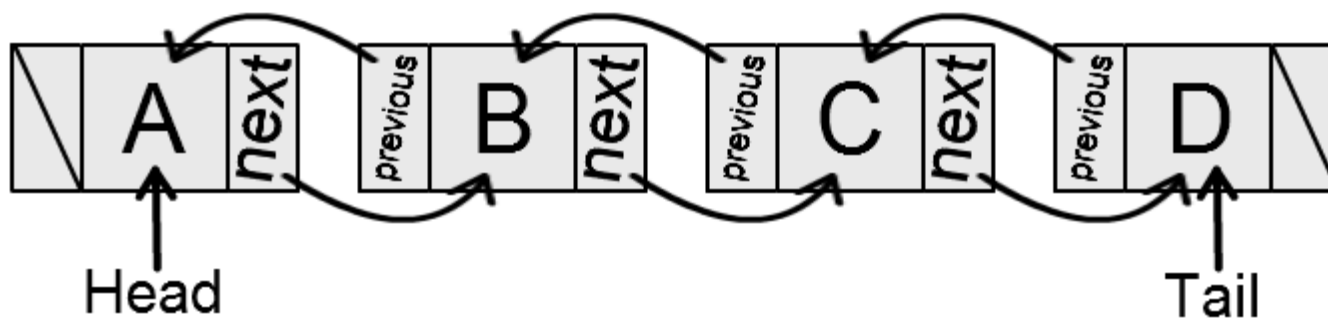
Linked list

$A \rightarrow B \rightarrow C \rightarrow D$



Doubly linked list

$A \rightleftarrows B \rightleftarrows C \rightleftarrows D$



Classe SListNode (1)

Reference to an element	next
----------------------------	------

```
package dataStructures;
```

```
class SListNode<E>{
```

```
    // Element stored in the node.
```

```
    protected E element;
```

```
    // (Pointer to) the next node.
```

```
    protected SListNode<E> next;
```

```
    public SListNode( E elem, SListNode<E> theNext ){
```

```
        element = elem;
```

```
        next = theNext;
```

```
    }
```

```
    public SListNode( E theElement ){
```

```
        this(theElement, null);
```

```
    }
```

```
    ...
```

```
}
```

Atenção à visibilidade da classe

Classe SListNode (2)

```
package dataStructures;
```

Reference to an element

next

```
class SListNode<E>{
```

```
    ...
```

```
    public E getElement( ){  
        return element;  
    }
```

```
    public SListNode<E> getNext( ){  
        return next;  
    }
```

```
    public void setElement( E newElement ){  
        element = newElement;  
    }
```

```
    public void setNext( SListNode<E> newNext ){  
        next = newNext;  
    }
```

```
}
```

Classe DListNode (1)

prev	Reference to an element	next
------	----------------------------	------

```
package dataStructures;
```

```
class DListNode<E>{  
    // Element stored in the node.  
    protected E element;  
    // (Pointer to) the next node.  
    protected DListNode<E> next;  
    // (Pointer to) the previous node.  
    protected DListNode<E> previous;
```

Atenção à visibilidade da classe

```
    public DListNode( E elem, DListNode<E> thePrev, DListNode<E> theNext ){  
        element = elem;  
        previous = thePrev;  
        next = theNext;  
    }
```

```
    public DListNode( E theElement ){  
        this(theElement, null, null);  
    }
```

```
    ...  
}
```

Classe **DListNode** (2)

prev	Reference to an element	next
------	----------------------------	------

```
package dataStructures;
```

```
class DListNode<E>{
```

```
    ...
```

```
    public E getElement( ){  
        return element;  
    }
```

```
    public DListNode<E> getNext( ){  
        return next;  
    }
```

```
    public DListNode<E> getPrevious( ){  
        return previous;  
    }
```

```
    ...
```

```
}
```

Classe **DListNode** (3)

prev	Reference to an element	next
------	----------------------------	------

```
package dataStructures;
```

```
class DListNode<E>{
```

```
    ...
```

```
    public void setElement( E newElement ){  
        element = newElement;  
    }
```

```
    public void setNext( DListNode<E> newNext ){  
        next = newNext;  
    }
```

```
    public void setPrevious( DListNode<E> newPrevious ){  
        previous = newPrevious;  
    }
```

```
}
```

Classe SinglyLinkedList<E>

- Estrutura dinâmica que implementa o TAD List<E>, e permite o percurso num só sentido.

```
package dataStructures;
```

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

```
// Node at the head of the list.
```

```
protected SListNode<E> head;
```

```
// Node at the tail of the list.
```

```
protected SListNode<E> tail;
```

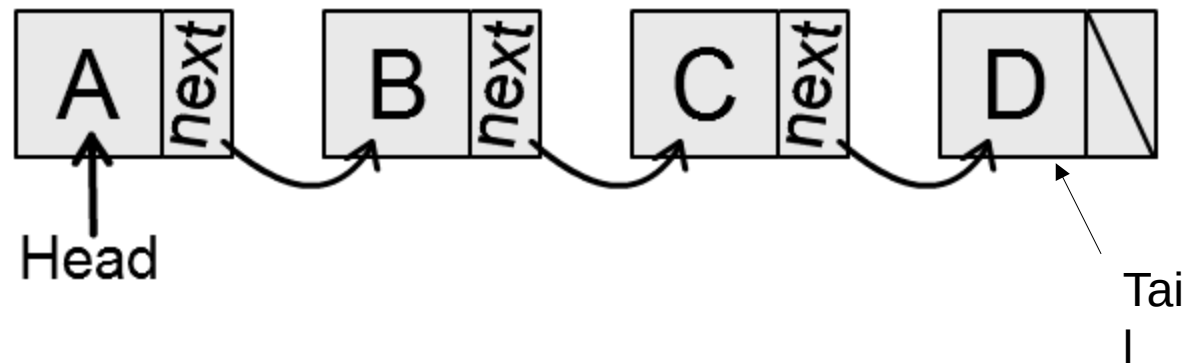
```
// Number of elements in the list.
```

```
protected int currentSize;
```

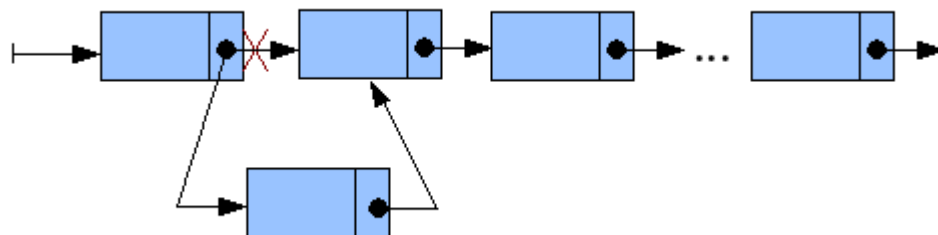
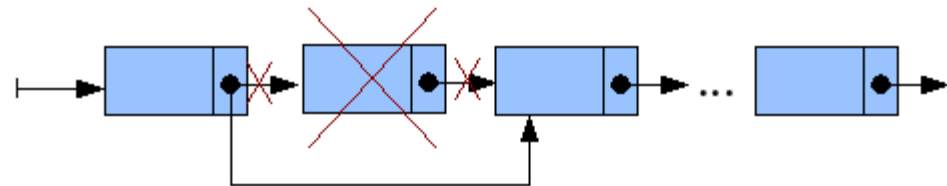
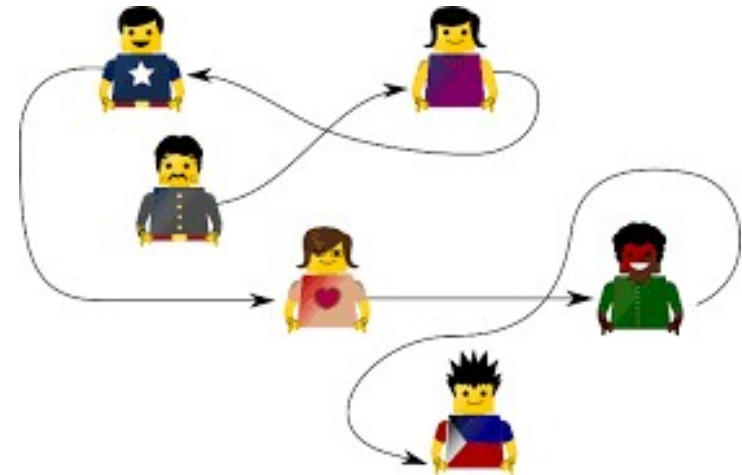
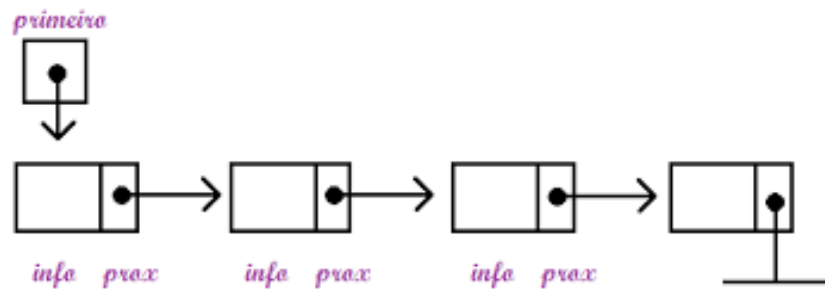
```
// TODO
```

```
...  
}
```

Atributos acessíveis no package



Classe SinglyLinkedList<E>



Classe **DoublyLinkedList** (1)

- Estrutura dinâmica que implementa o TAD `TwoWayList<E>`, e permite percursos bidirecionais.

```
package dataStructures;
```

```
public class DoublyLinkedList<E> implements TwoWayList<E> {
```

```
// Node at the head of the list.
```

```
protected DListNode<E> head;
```

```
// Node at the tail of the list.
```

```
protected DListNode<E> tail;
```

```
// Number of elements in the list.
```

```
protected int currentSize;
```

```
public DoublyLinkedList( ){
```

```
    head = null;
```

```
    tail = null;
```

```
    currentSize = 0;
```

```
}
```

```
...
```

```
}
```

Atributos acessíveis no package

Lista vazia

null

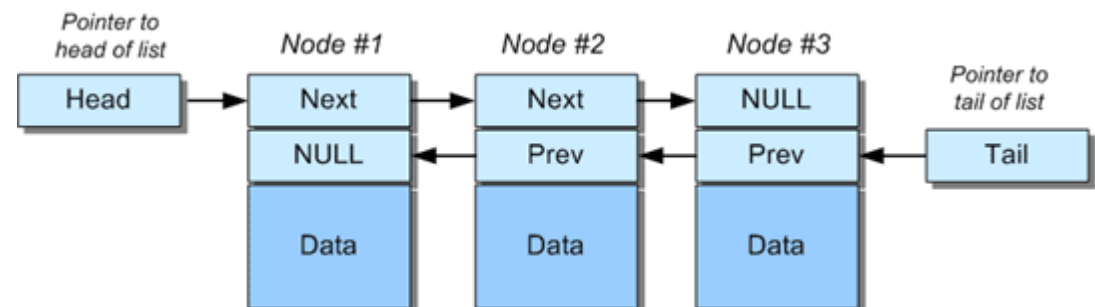
head

null

tail

0

currentSize



Classe DoublyLinkedList (2)

```
package dataStructures;
```

```
public class DoublyLinkedList<E> implements TwoWayList<E> {
```

```
...
```

```
@Override
```

```
public boolean isEmpty() {
```

```
    //TODO
```

```
    return false;
```

```
}
```

```
@Override
```

```
public int size() {
```

```
    //TODO
```

```
    return 0;
```

```
}
```

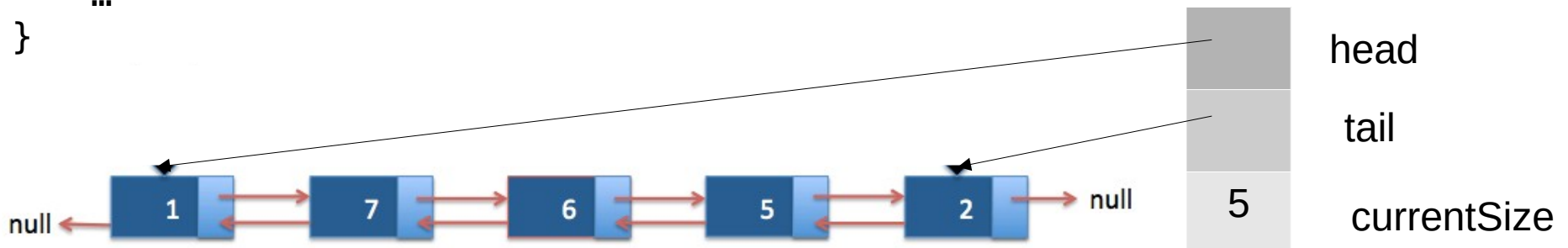
```
...
```

```
}
```

Lista vazia

null	head
null	tail
0	currentSize

Lista com elementos



Classe DoublyLinkedList (3)

```
package dataStructures;
```

```
public class DoublyLinkedList<E> implements TwoWayList<E> {
```

```
    ...
```

```
    @Override
```

```
    public int find(E element) {
```

```
        int pos=0;
```

```
        DListNode<E> auxNo=head;
```

```
        boolean found=false;
```

```
        //TODO
```

```
        //percurso nos nos, começando na cabeca
```

```
        // ate encontrar elemento ou chegar a null
```

```
        if (found)
```

```
            return pos;
```

```
        return -1;
```

```
    }
```

```
    ...
```

```
}
```

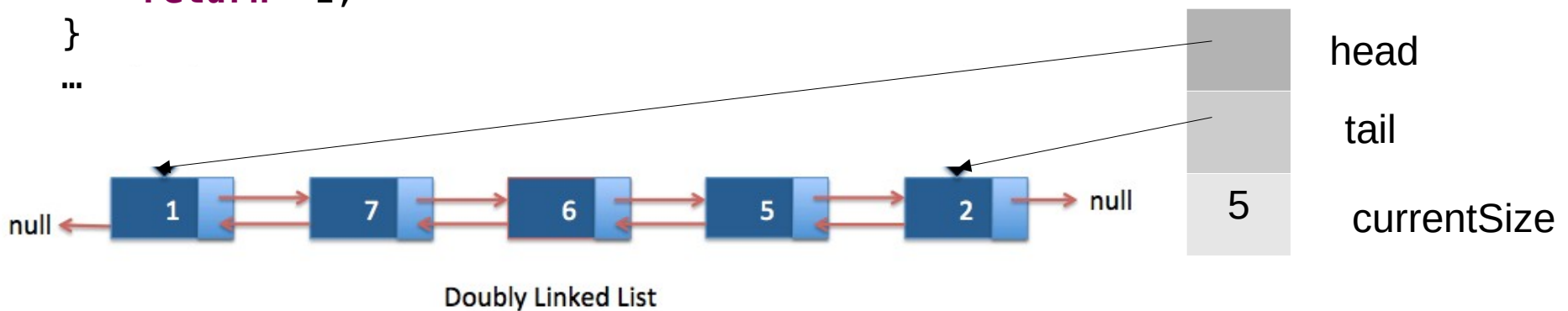
Percurso em Lista

Início → ListNode<E> auxNo=head;

Avanço → auxNo = auxNo.getNext();

Condição de fim → auxNo==null

Lista com elementos



Classe DoublyLinkedList (4)

```
package dataStructures;
```

```
public class DoublyLinkedList<E> implements TwoWayList<E> {
```

```
    @Override
```

```
    public void addFirst(E element) {
```

```
        //TODO
```

```
        //Criar nó sendo o seguinte a cabeça actual e o anterior null
```

```
        //Caso lista não vazia actualizar anterior da cabeça actual
```

```
        //Colocar a cabeça igual ao novo nó
```

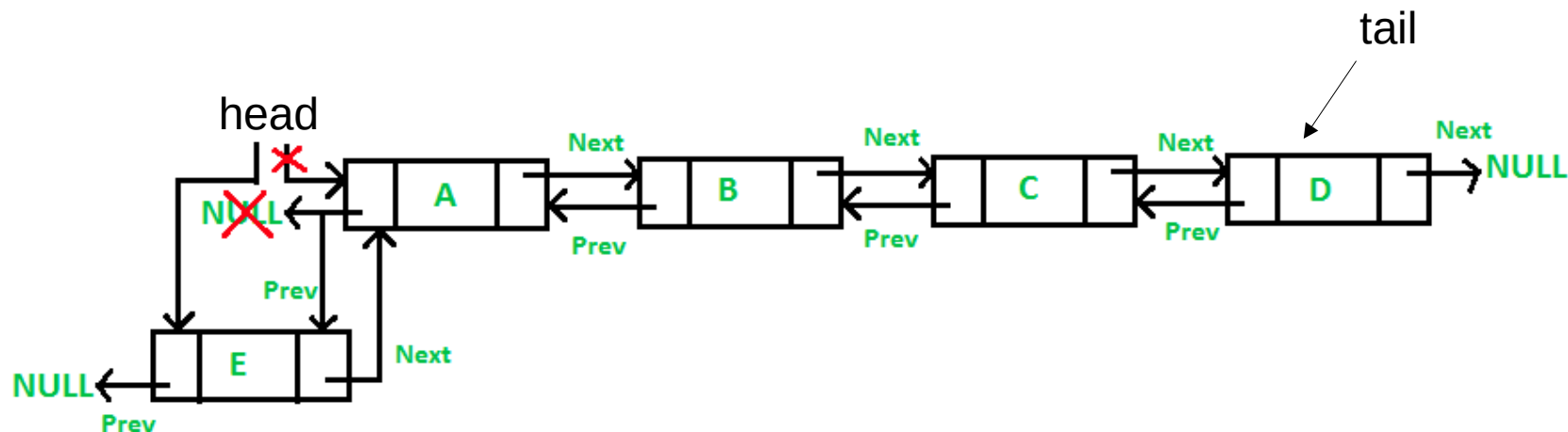
```
        //Caso lista vazia, colocar a cauda igual ao novo nó
```

```
        //Incrementar número de elementos
```

```
    }
```

```
    ...
```

```
}
```



Classe DoublyLinkedList (5)

```
package dataStructures;
```

```
public class DoublyLinkedList<E> implements TwoWayList<E> {
```

```
    ...
```

```
    @Override
```

```
    public void addLast(E element) {
```

```
        // TODO
```

```
        //Criar nó sendo o seguinte null e o anterior a cauda actual
```

```
        //Caso lista não vazia actualizar seguinte da cauda actual
```

```
        //Colocar a cauda igual ao novo nó
```

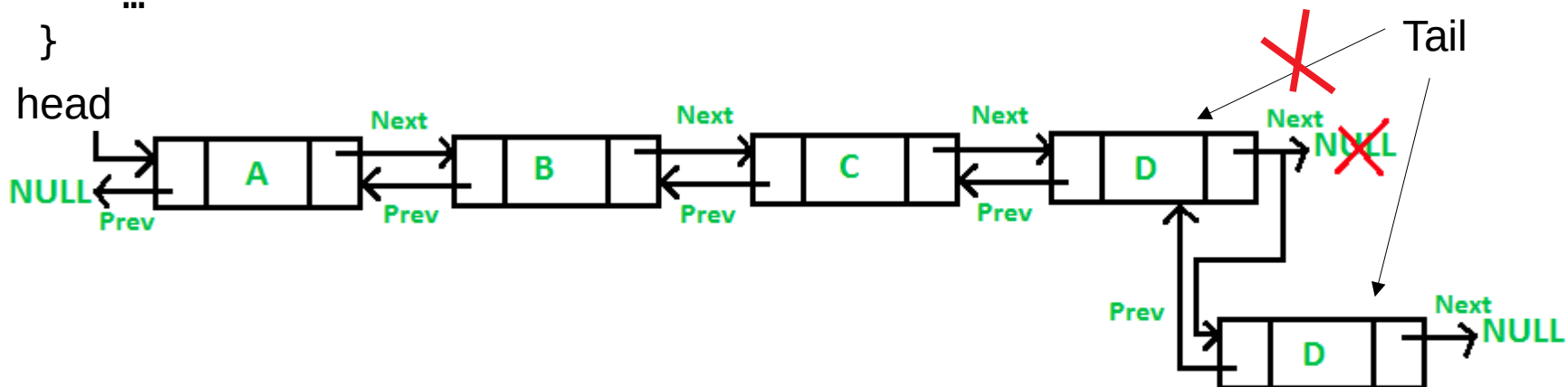
```
        //Caso lista vazia colocar a cabeça igual ao novo nó
```

```
        //Incrementar número de elementos
```

```
    }
```

```
    ...
```

```
}
```



Classe DoublyLinkedList (6)

```
package dataStructures;
```

```
public class DoublyLinkedList<E> implements TwoWayList<E> {
```

```
...
```

```
@Override
```

```
public void add(int position, E element) throws InvalidPositionException {
```

```
    if (position < 0 || position > currentSize)
```

```
        throw new InvalidPositionException("Invalid Position.");
```

```
    if (position == 0)
```

```
        addFirst(element);
```

```
    else
```

```
        if (position == currentSize)
```

```
            addLast(element);
```

```
        else
```

```
            addMiddle(position, element);
```

```
}
```

```
...
```

```
}
```

Método a implementar.

Classe DoublyLinkedList (7)

```
package dataStructures;
```

```
public class DoublyLinkedList<E> implements TwoWayList<E> {  
    ...
```

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

```
        // TODO
```

```
        //Percorrer até à posição requerida
```

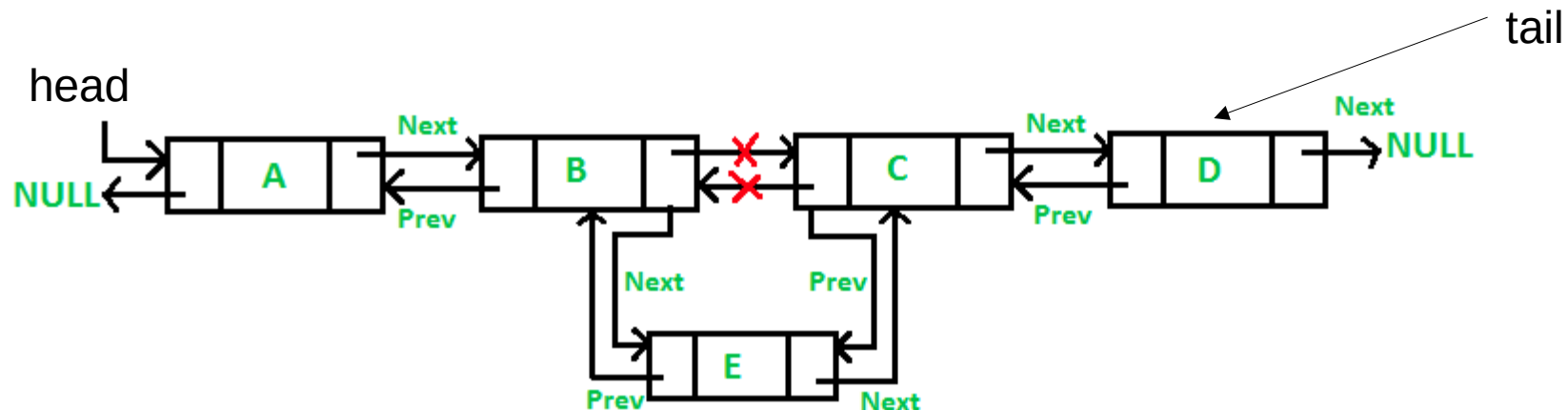
```
        //Criar nó e actualizar apontadores
```

```
        //Incrementar número de elementos
```

```
    }
```

```
    ...
```

```
}
```



Classe DoublyLinkedList (8)

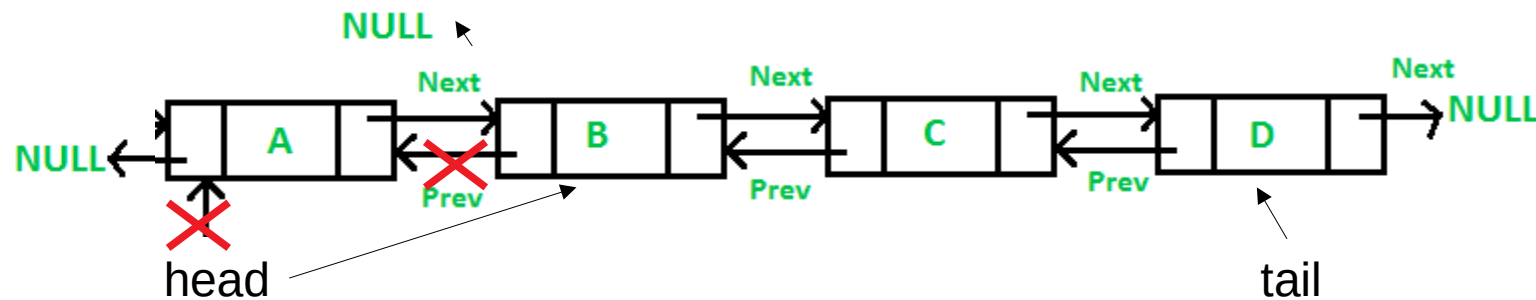
```
package dataStructures;
```

```
public class DoublyLinkedList<E> implements TwoWayList<E> {  
    ...
```

```
@Override
```

```
public E removeFirst() throws NoSuchElementException {  
    if (currentSize==0)  
        throw new NoSuchElementException("No such element.");  
    // TODO  
    // Cuidado: lista com 1 elemento  
    return null;  
}
```

```
...  
}
```



Classe DoublyLinkedList (9)

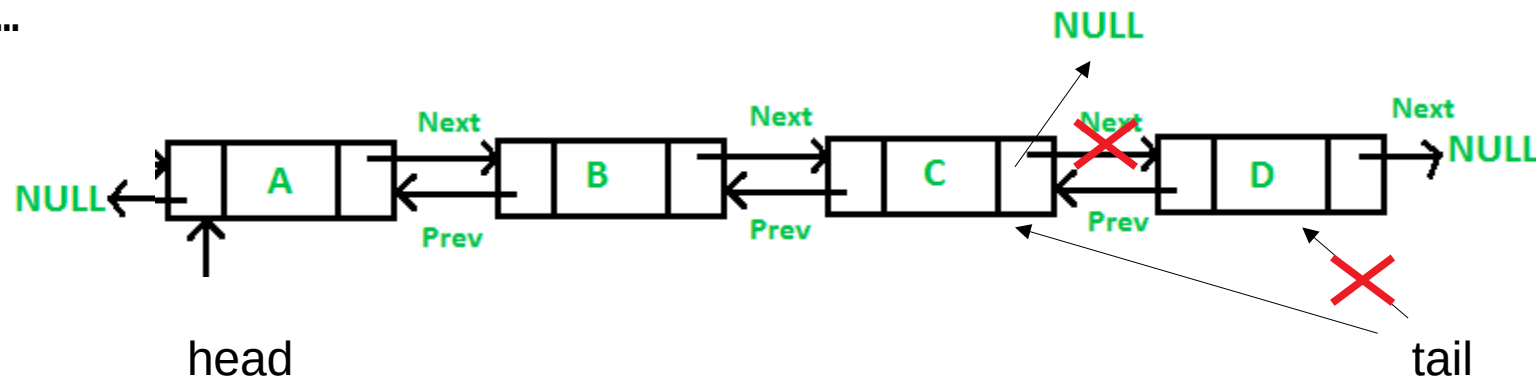
```
package dataStructures;
```

```
public class DoublyLinkedList<E> implements TwoWayList<E> {  
    ...
```

```
@Override
```

```
public E removeLast() throws NoSuchElementException {  
    if (currentSize==0)  
        throw new NoSuchElementException("No such element.");  
    // TODO  
    // Cuidado: lista com 1 elemento  
    return null;  
}
```

```
...  
}
```



Classe **DoublyLinkedList** (10)

```
package dataStructures;
```

```
public class DoublyLinkedList<E> implements TwoWayList<E> {
```

```
    ...
```

```
    @Override
```

```
    public E remove(int position) throws InvalidPositionException {
```

```
        if (position < 0 || position >= currentSize)
```

```
            throw new InvalidPositionException("Invalid position.");
```

```
        if (position == 0)
```

```
            return removeFirst();
```

```
        if (position == currentSize - 1)
```

```
            return removeLast();
```

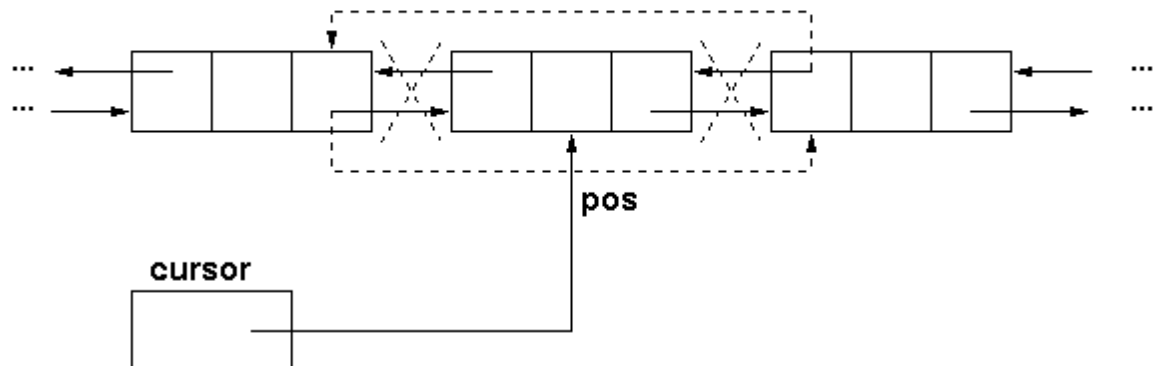
```
        return removeMiddle(position);
```

```
    }
```

```
}
```

Método a implementar

Removal of an element of a doubly-linked list



Classe **DoublyLinkedList** (11)

```
package dataStructures;
```

```
public class DoublyLinkedList<E> implements TwoWayList<E> {
```

```
    private void addMiddle(int position, E element) {  
        DListNode<E> aux=getNode(position);  
        //TODO  
    }
```

```
    private E removeMiddle(int position) {  
        DListNode<E> aux=getNode(position);  
        //TODO  
        return null;  
    }
```

```
    private DListNode<E> getNode(int position){  
        DListNode<E> aux=blue;  
        for(int i=1;i<=position;i++)  
            aux=aux.getNext();  
        return aux;  
    }  
    ...
```

```
}
```

Método a implementar.

Classe **DoublyLinkedList** (12)

```
package dataStructures;
```

```
public class DoublyLinkedList<E> implements TwoWayList<E> {
```

```
...
```

```
@Override
```

```
public E getFirst() throws NoSuchElementException {  
    if (currentSize==0) throw new NoSuchElementException("No such element.");  
    return getNode(0).getElement();  
}
```

```
@Override
```

```
public E getLast() throws NoSuchElementException {  
    if (currentSize==0) throw new NoSuchElementException("No such element.");  
    return getNode(currentSize-1).getElement();  
}
```

```
@Override
```

```
public E get(int position) throws InvalidPositionException {  
    if (position<0 || position>=currentSize)  
        throw new InvalidPositionException("Invalid position.");  
    return getNode(position).getElement();  
}
```

Classe **DoublyLinkedList** (13)

```
package dataStructures;
```

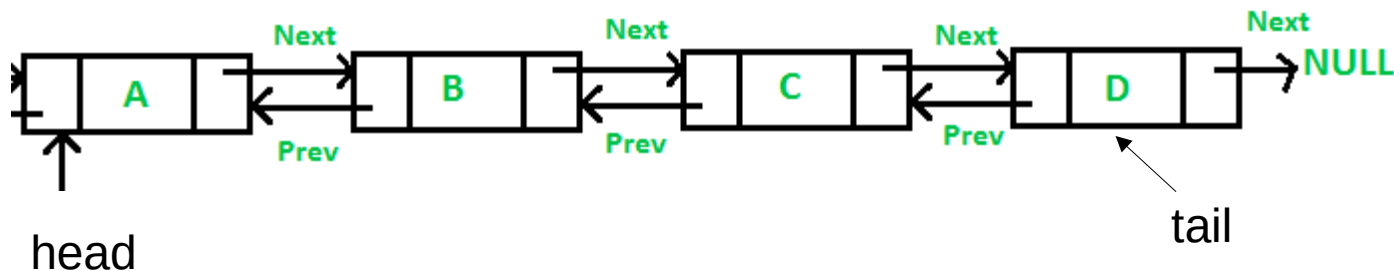
```
public class DoublyLinkedList<E> implements TwoWayList<E> {  
    ...
```

```
@Override
```

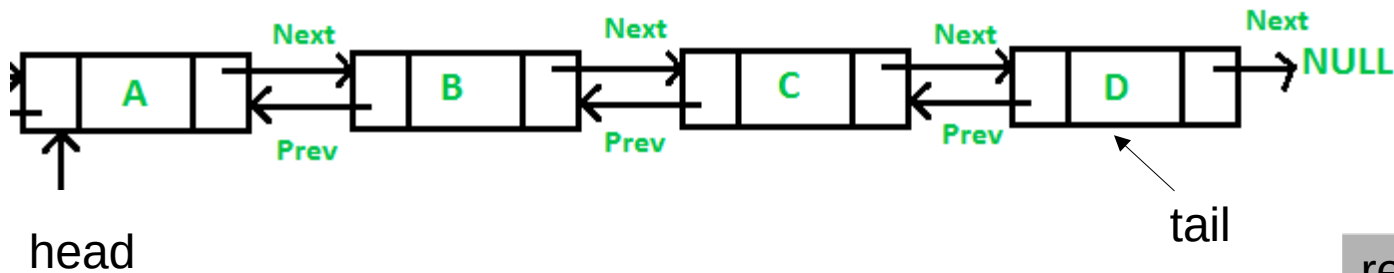
```
public TwoWayIterator<E> iterator() throws NoSuchElementException {  
    if (currentSize==0)  
        throw new NoSuchElementException("List is empty.");  
    return new DoublyLLIterator<E>(head,tail);  
}
```

```
...  
}
```

Classe a implementar



Classe DoublyLLIterator (1)



rewind()	
next()	A
next()	B
previous()	A
next()	B
previous()	A
fullForward()	
previous()	D

Classe **DoublyLLIterator** (2)

```
package dataStructures;

public class DoublyLLIterator<E> implements TwoWayIterator<E> {

    // Node with the first element in the iteration.
    private DListNode<E> firstNode;

    // Node with the last element in the iteration.
    private DListNode<E> lastNode;

    // Node with the next element in the iteration.
    private DListNode<E> nextToReturn;

    // Node with the previous element in the iteration.
    private DListNode<E> prevToReturn;

    ...
}
```

Classe DoublyLLIterator (3)

```
package dataStructures;
```

```
public class DoublyLLIterator<E> implements TwoWayIterator<E> {
```

```
...
```

```
public DoublyLLIterator(DListNode<E> head, DListNode<E> tail) {
```

```
    firstNode=head;
```

```
    lastNode=tail;
```

```
    rewind();
```

```
}
```

```
@Override
```

```
public void rewind() {
```

```
    nextToReturn = firstNode;
```

```
    prevToReturn = null;
```

```
}
```

```
@Override
```

```
public void fullForward() {
```

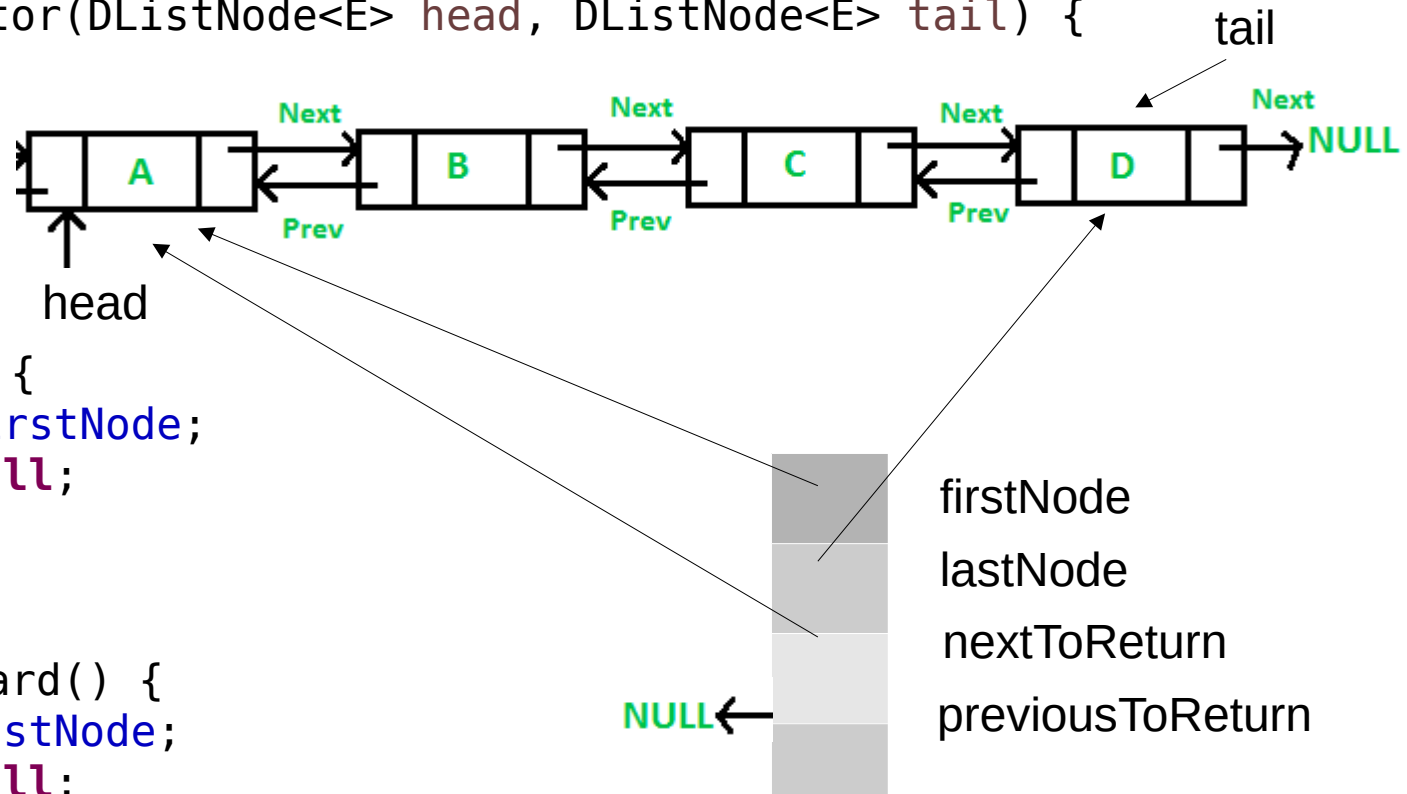
```
    prevToReturn = lastNode;
```

```
    nextToReturn = null;
```

```
}
```

```
...
```

```
}
```



Classe DoublyLLIterator (4)

```
package dataStructures;
```

```
public class DoublyLLIterator<E> implements TwoWayIterator<E> {  
...  
@Override  
public boolean hasNext() {  
    return nextToReturn != null;  
}  
  
@Override  
public E next() throws NoSuchElementException {  
    if ( !this.hasNext() )  
        throw new NoSuchElementException("No more elements.");  
    E element = nextToReturn.getElement();  
    //TODO  
    //Actualizar nextToReturn e previousToReturn  
    return element;  
}  
...  
}
```

Diagram illustrating the DoublyLLIterator structure and state:

- The doubly linked list contains nodes A, B, C, and D.
- Arrows labeled "Next" connect nodes A → B → C → D → NULL.
- Arrows labeled "Prev" connect nodes D → C → B → A.
- The "head" pointer points to the first node (A).
- The "tail" pointer points to the last node (D).
- The iterator state variables (firstNode, lastNode, nextToReturn, previousToReturn) are shown as a vertical stack of boxes, with arrows indicating their current values: firstNode points to A, lastNode points to D, nextToReturn points to A, and previousToReturn points to D.

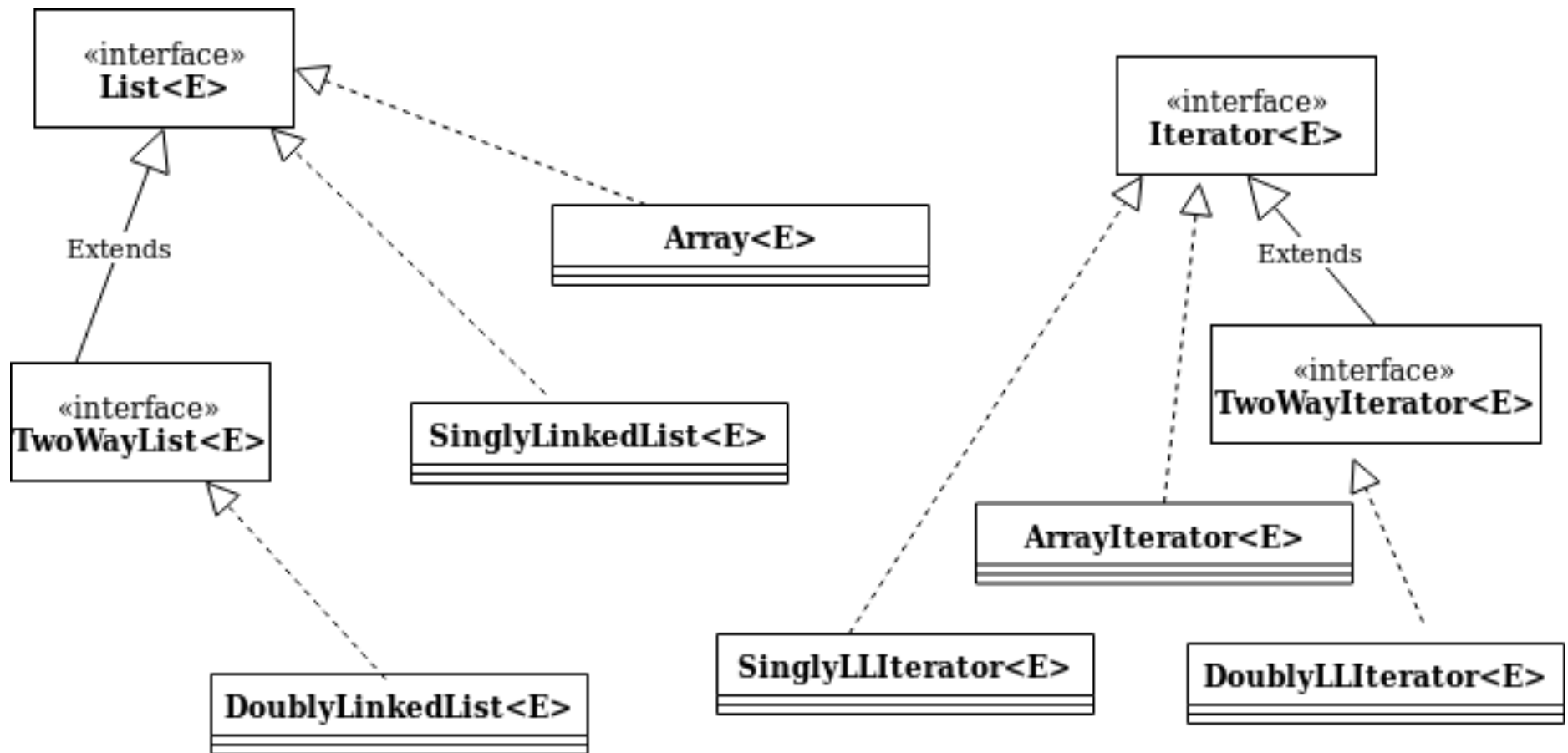
Classe DoublyLLIterator (5)

```
package dataStructures;
```

```
public class DoublyLLIterator<E> implements TwoWayIterator<E> {  
...  
@Override  
public boolean hasNextPrevious() {  
    return previousToReturn != null;  
}  
  
@Override  
public E previous() throws NoSuchElementException {  
    if ( !this.hasNextPrevious() )  
        throw new NoSuchElementException("No more elements.");  
    E element = previousToReturn.getElement();  
    //TODO  
    //Atualizar nextToReturn e previousToReturn  
    return element;  
}  
...  
}
```

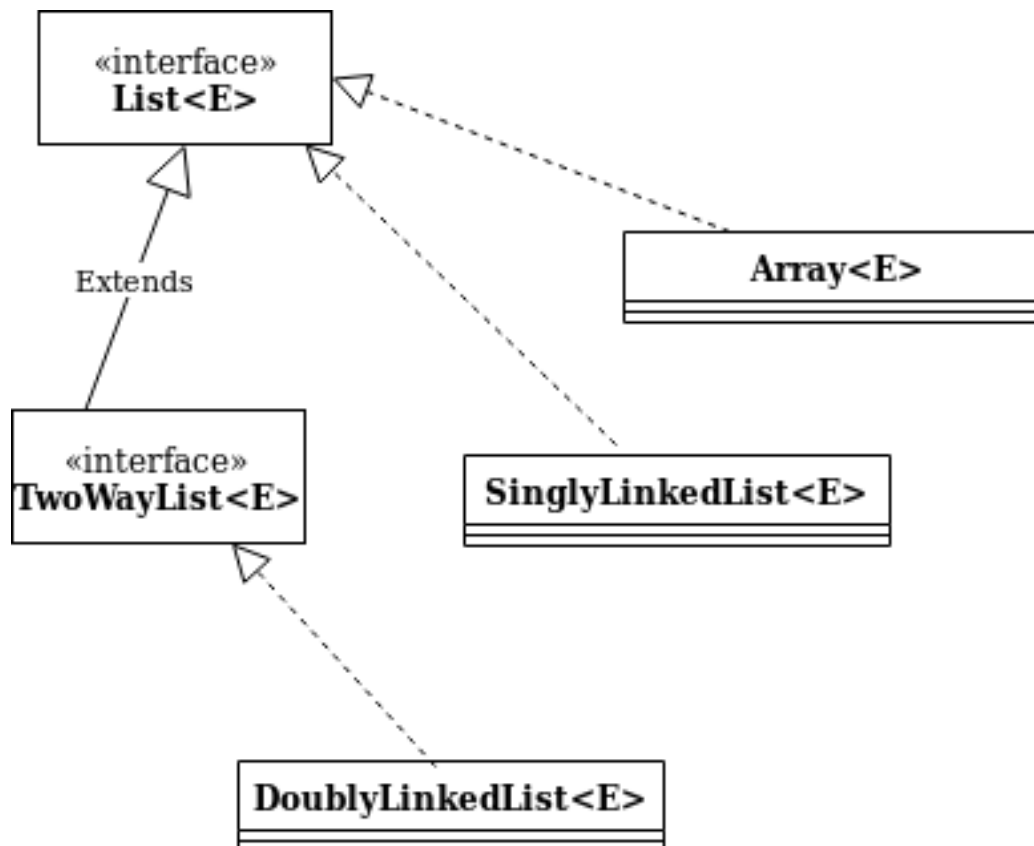

Package dataStructures

TAD List<E>



Interface **List<E>**

- Qual das implementações escolher?
 - Vector ou Lista simplesmente ligada ou Lista duplamente ligada



Classe **Array**

- Queremos um método de ordenação. Qual usar?
 - bubbleSort ou insertionSort
 - Mas existem mais ...



```
package dataStructures;
```

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

```
...
```

```
    public static <E> void xSort( E[] vec, int vecSize, Comparator<E> c)){
```

```
        //TODO
```

```
    }
```

```
}
```

Interface a implementar com método:
int compare(E obj1, E obj2);



Exercícios propostos

- ✓ 2ª aula prática → Completar a implementação da classe DoublyLinkedList, usar no exercício do parque de diversões (versão A) e submeter no mooshak (problema B).
- ✓ TPC 2ª semana → Implementar a classe SinglyLinkedList, usar no exercício do parque de diversões (assuma que não existe percurso invertido).