ESTRUTURAS DE DADOS

2024/2025

ESCOLA SUPERIOR DE TECNOLOGIA E GESTÃO

Aula 06

- Listas
- Listas Ordenadas
- Listas Não Ordenadas
- Implementação em *array*
- Implementação em Lista Duplamente Ligada
- Padrão *Iterator* (e aplicação)

Ricardo Santos | rjs@estg.ipp.pt

Listas

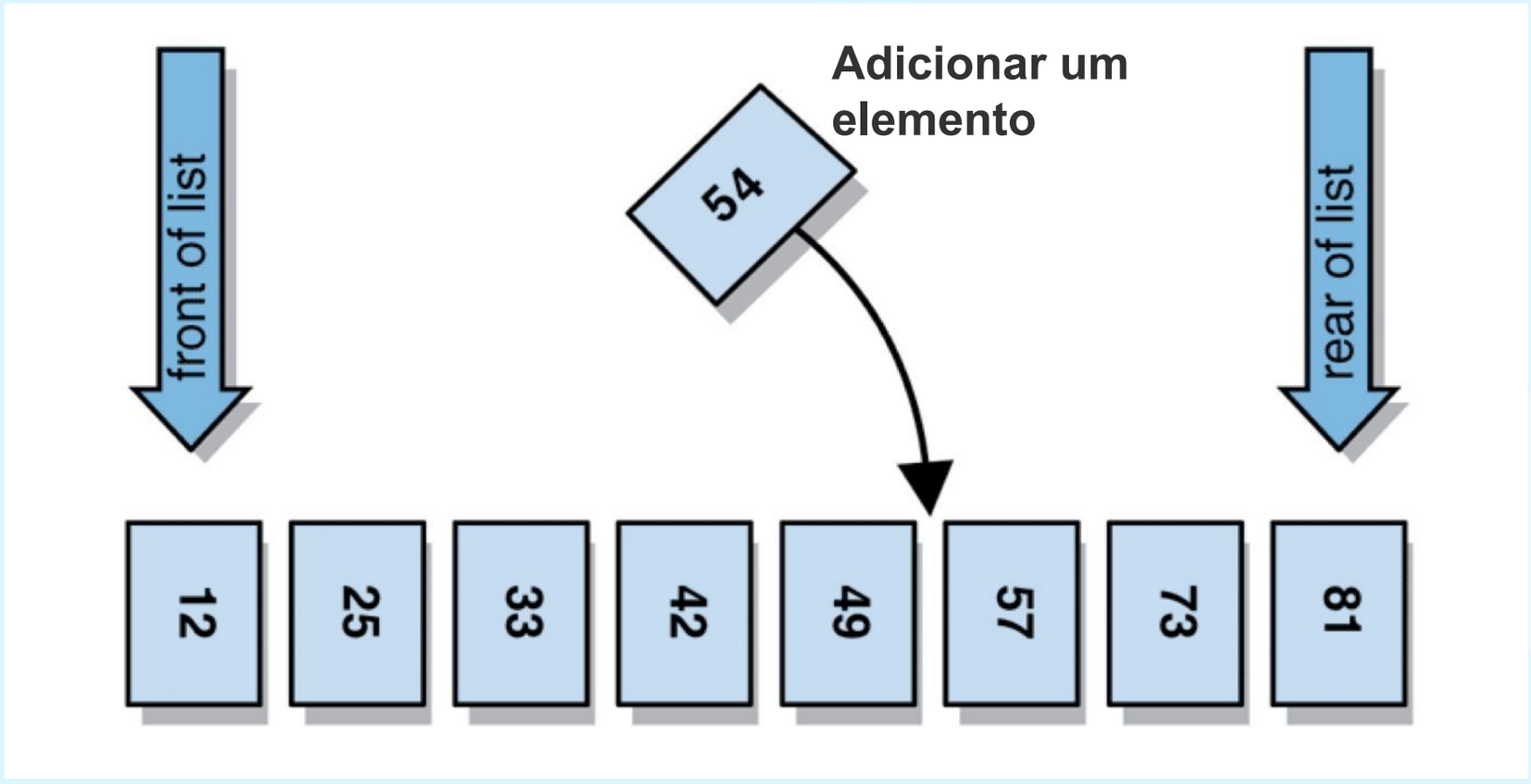
- As listas são colecções lineares, como pilhas e filas, mas são mais flexíveis
- A adição e remoção de elementos em listas não são limitadas pela estrutura da colecção
- Ou seja, não têm de operar numa ou noutra ponta (cabeça ou cauda)

- Vamos examinar três tipos de coleções do tipo lista:
  - listas ordenadas
  - listas não ordenadas
  - listas indexadas

Listas Ordenadas

- Os elementos numa lista ordenada estão ordenados por alguma característica intrínseca dos elementos
  - nomes em ordem alfabética
  - valores por ordem ascendente
- Portanto, terão de ser os próprios elementos a determinar onde são armazenados na lista

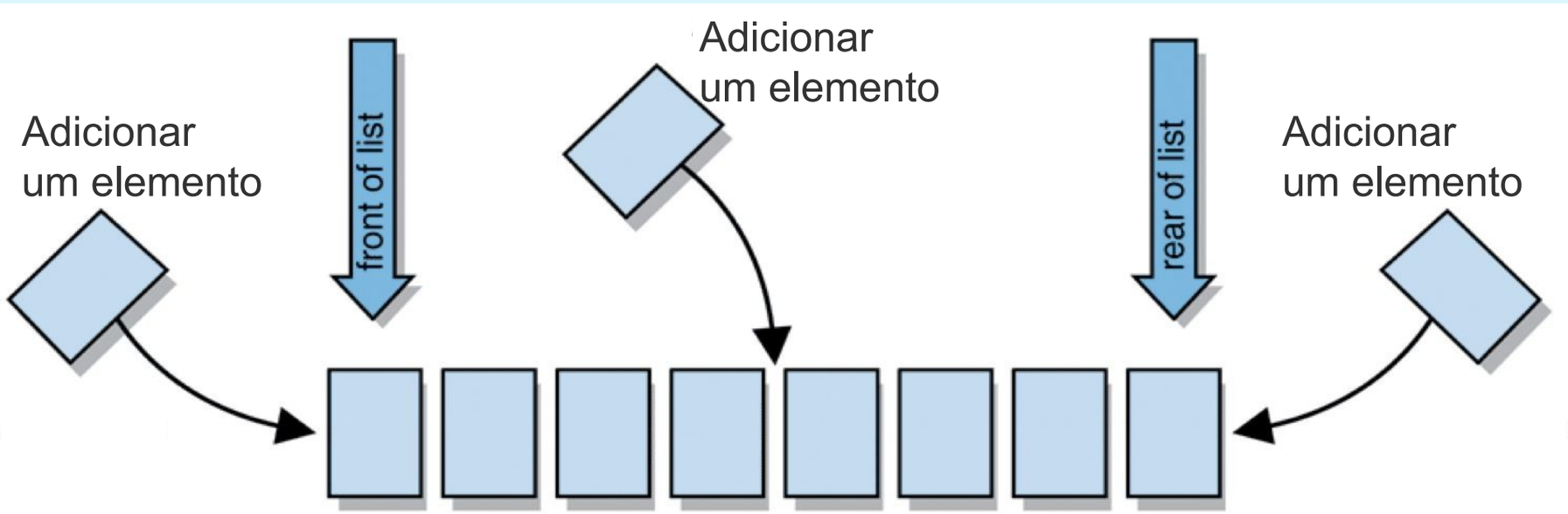
Vista Conceptual de uma Lista



Listas Não Ordenadas

- Há uma ordem para os elementos numa lista não ordenada, mas a ordem não é baseada em características do elemento
- O utilizador da lista determina a ordem dos elementos
- Um novo elemento pode ser colocado na parte dianteira ou traseira da lista, ou pode ser inserido após um determinado elemento já na lista

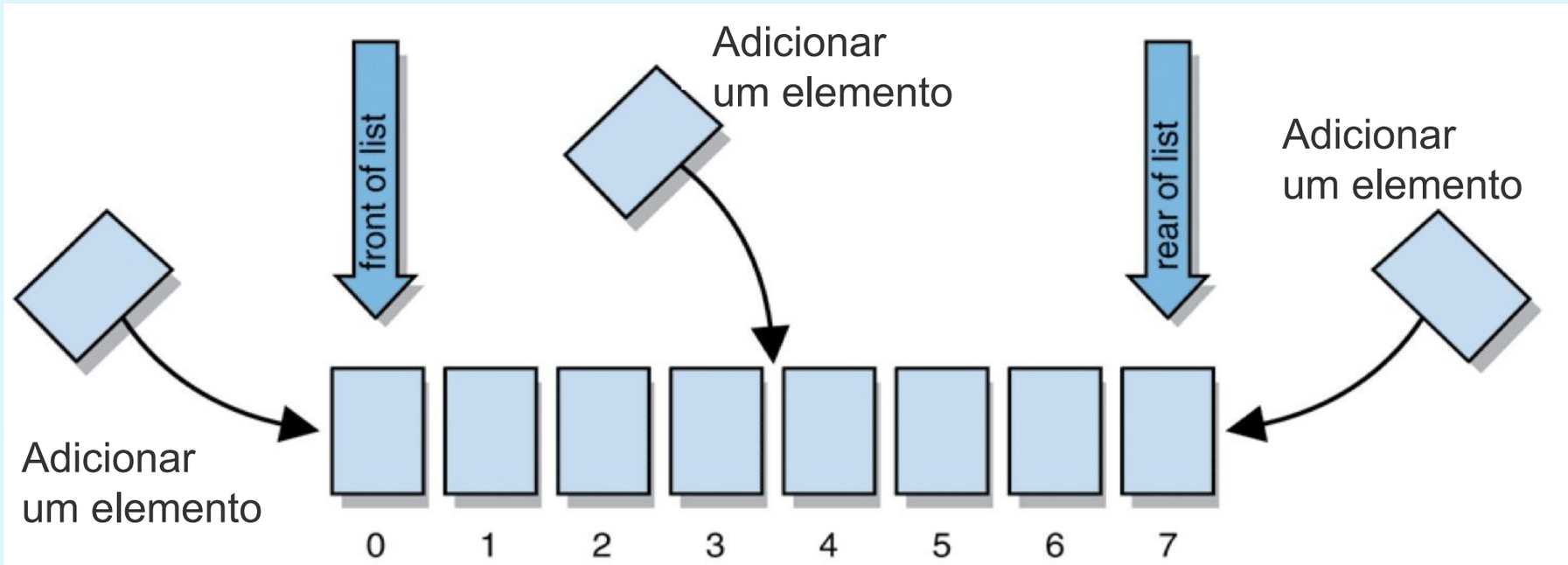
Vista Conceptual de uma Lista Não Ordenada



Listas Indexadas

- Numa lista indexada, os elementos são referenciados pela sua posição numérica na lista
- Como numa lista não ordenada, não existe uma relação intrínseca entre os elementos
- O utilizador pode determinar a ordem
- De cada vez que a lista é alterada, os índices são actualizados

Vista Conceptual de uma Lista Indexada



Operações da Lista

- Há muitas operações comuns aos três tipos de lista
- Estas incluem a remoção de elementos de várias maneiras e verificar o estado da lista
- As principais diferenças entre os tipos da lista envolvem a forma como são adicionados os elementos

Operações comuns da Lista

Operação	Descrição
`removeFirst`	Remove o primeiro elemento da lista
`removeLast`	Remove o último elemento da lista
`remove`	Remove um determinado elemento da lista
`first`	Examina o elemento na frente da lista
`last`	Examina o elemento na traseira da lista
`contains`	Determina se a lista contém um determinado elemento
`isEmpty`	Determina se a lista está vazia
`size`	Determina o número de elementos da lista
`iterator`	Retorna um iterador para os elementos da lista
`toString`	Representação da lista em string

Operações particulares de uma Lista Ordenada

Operação	Descrição
`add`	Adicionar um elemento à lista

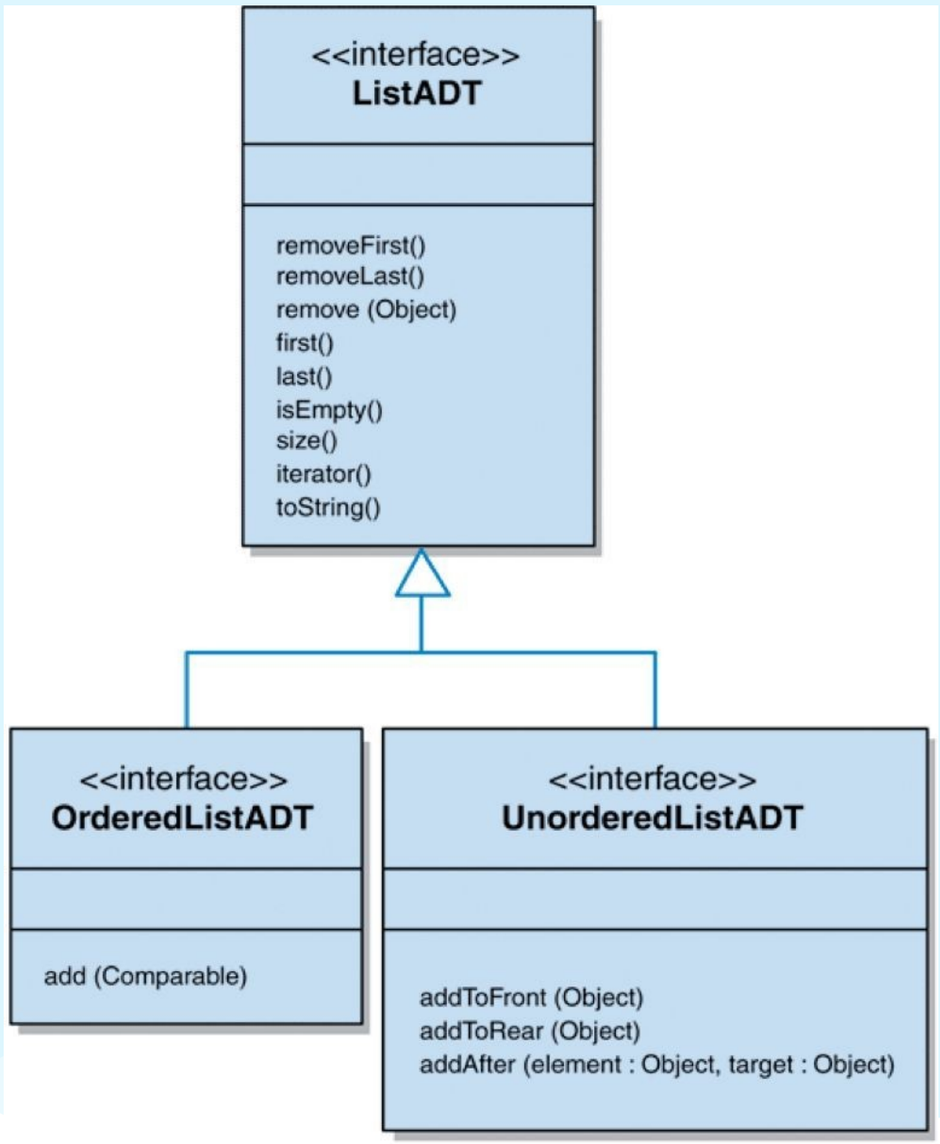
Operações particulares de uma Lista Não Ordenada

Operação	Descrição
`addToFront`	Adiciona um elemento à frente da lista
`addToRear`	Adiciona um elemento à traseira da lista
`addAfter`	Adiciona um elemento depois de um outro existente na lista

Operações da Lista

- Tal como acontece com outras colecções, são usados interfaces *Java* para definir as operações de colecção
- Lembrem-se que as interfaces podem ser definidas através de herança (derivado de outras interfaces)
- É definida a lista comum de operações numa única interface, depois são derivadas outras interfaces para os outros tipos de lista

As várias interfaces de *List*



Interface `ListADT`

```
import java.util.Iterator;

public interface ListADT<T> extends Iterable<T> {

   /**
    * Removes and returns the first element from this list.
    *
    * @return  the first element from this list
    */
   public T removeFirst ();

   /**
    * Removes and returns the last element from this list.
    *
    * @return  the last element from this list
    */
   public T removeLast ();

   /**
    * Removes and returns the specified element from this list.
    *
    * @param element  the element to be removed from the list
    */
   public T remove (T element);
```

```
/**
 * Returns a reference to the first element in this list.
 * @return  a reference to the first element in this list
 */
public T first ();

/**
 * Returns a reference to the last element in this list.
 * @return  a reference to the last element in this list
 */
public T last ();

/**
 * Returns true if this list contains the specified target
 * element.
 * @param target  the target that is being sought in the list
 * @return        true if the list contains this element
 */
public boolean contains (T target);

/**
 * Returns true if this list contains no elements.
 * @return  true if this list contains no elements
 */
public boolean isEmpty();
```

```
/**
 * Returns the number of elements in this list.
 *
 * @return  the integer representation of number of
 * elements in this list
 */
public int size();

/**
 * Returns an iterator for the elements in this list.
 *
 * @return  an iterator over the elements in this list
 */
public Iterator<T> iterator();

/**
 * Returns a string representation of this list.
 *
 * @return  a string representation of this list
 */
@Override
public String toString();
}
```

Interface `OrderedListADT`

```
public interface OrderedListADT<T> extends ListADT<T> {

    /**
     * Adds the specified element to this list at
     * the proper location
     *
     * @param element  the element to be added to this list
     */
    public void add (T element);
}
```

Interface UnorderedListADT

```
public interface UnorderedListADT<T> extends ListADT<T> {

    /**
     * Adds the specified element to the front of this list.
     *
     * @param element the element to be added to the front of this list
     */
    public void addToFront(T element);

    /**
     * Adds the specified element to the rear of this list.
     *
     * @param element the element to be added to the rear of this list
     */
    public void addToRear(T element);

    /**
     * Adds the specified element after the specified target.
     *
     * @param element the element to be added after the target
     * @param target the target is the item that the element will be added after
     */
    public void addAfter(T element, T target);
}
```

Implementar uma Lista recorrendo a um *array*

Implementar uma Lista com um *array*

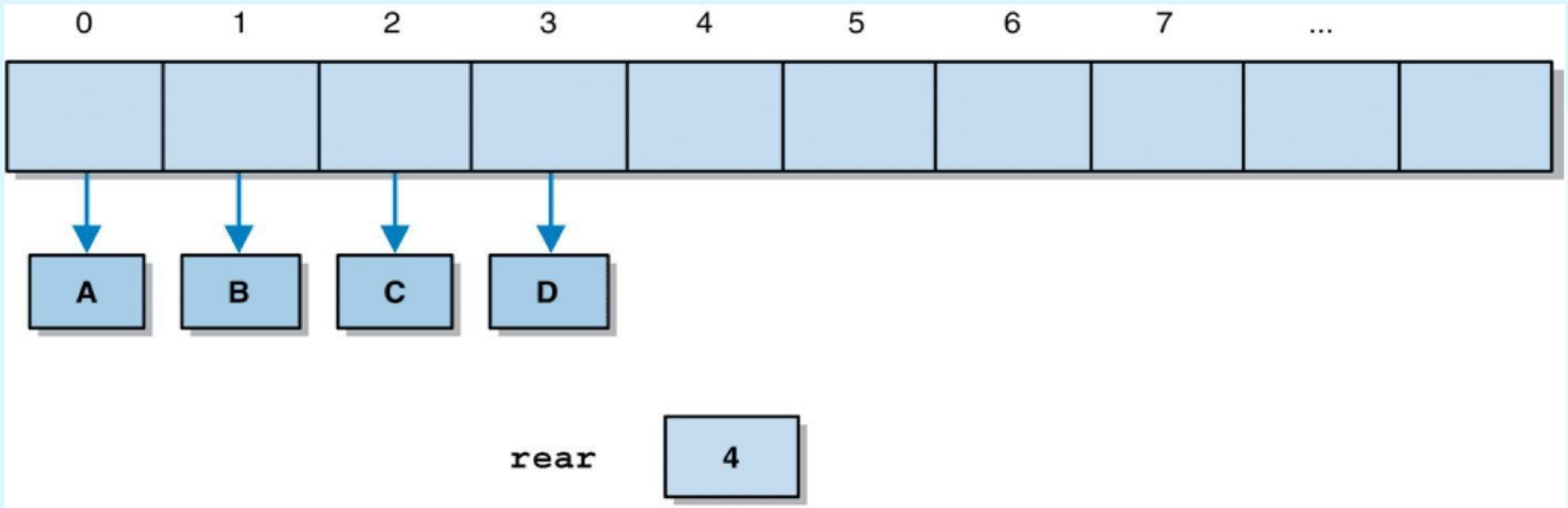
- Uma implementação de lista com base num *array* pode seguir estratégias semelhantes às usadas para uma fila (***Queue***)

- Podemos fixar uma ponta da lista no índice `0` e fazer um shift quando necessário quando um elemento é adicionado ou removido

- Ou poderíamos usar um *array* circular para evitar a mudança numa extremidade

- No entanto, não há como evitar uma mudança quando um elemento no meio é adicionado ou removido
- Vamos primeiro examinar a versão fixa

Implementação em *array* de uma Lista



Implementar uma Lista recorrendo a uma Lista Simplesmente Ligada

LinkedList - **Operação** remove

```
/**
 * Removes the first instance of the specified element from this
 * list and returns a reference to it.
 * Throws an EmptyListException
 * if the list is empty.  Throws a NoSuchElementException if the
 * specified element is not found in the list.
 */
public T remove (T targetElement)
     throws EmptyCollectionException, ElementNotFoundException {
   if (isEmpty())
      throw new EmptyCollectionException ("List");

   boolean found = false;
   LinearNode<T> previous = null;
   LinearNode<T> current = head;

   while (current != null && !found)
      if (targetElement.equals (current.getElement()))
         found = true;
      else
```

```
      {
         previous = current;
         current = current.getNext();
       }

   if (!found)
      throw new ElementNotFoundException ("List");

   if (size() == 1)
      head = tail = null;
   else if (current.equals (head))
      head = current.getNext();
   else if (current.equals (tail))
   {
      tail = previous;
      tail.setNext(null);
   }
   else
      previous.setNext(current.getNext());

   count--;

   return current.getElement();
}
```

Implementar uma Lista recorrendo a uma Lista Duplamente Ligada

Listas Duplamente Ligadas

- Uma lista duplamente ligada tem duas referências para cada nó, uma para o `next` elemento na lista e outra para o `previous` elemento
- Desta forma andar num sentido ou noutro na lista fica muito mais fácil e elimina a necessidade para guardar a referência anterior em alguns algoritmos
- Existe no entanto uma maior sobrecarga na gestão da lista

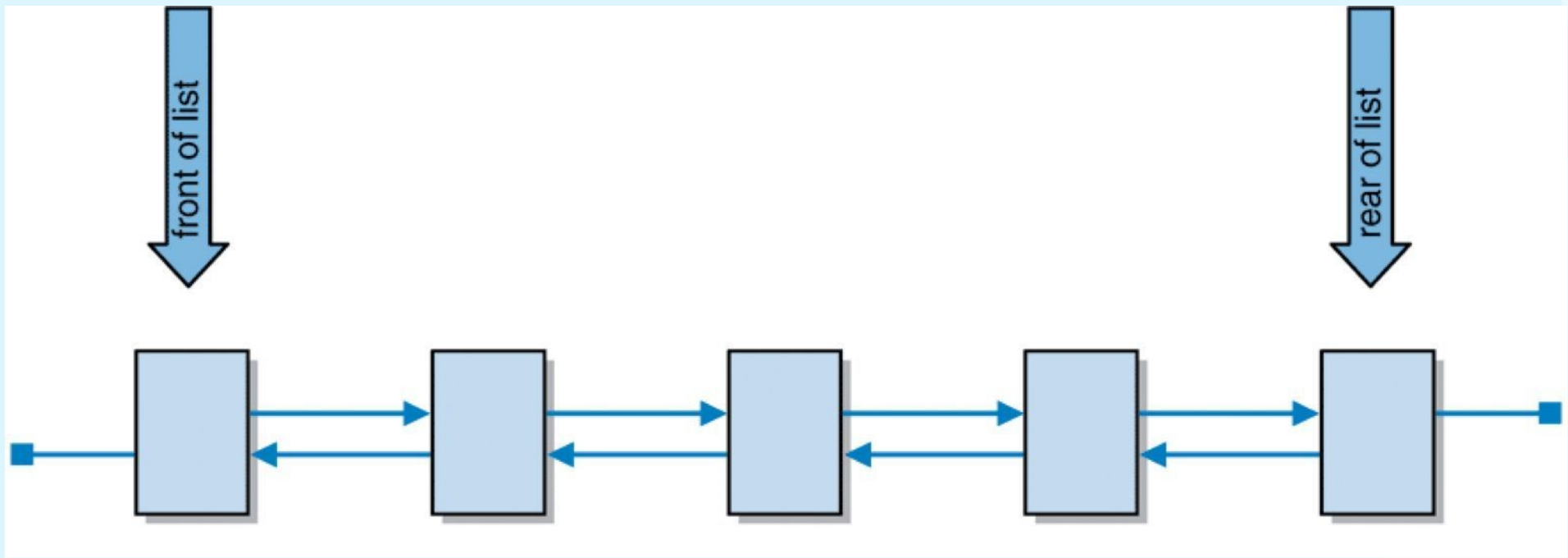
Classe DoubleNode

```
/**
 * DoubleNode represents a node in a doubly linked list.
 *
 */

public class DoubleNode<E>
{
    private DoubleNode<E> next;
    private E element;
    private DoubleNode<E> previous;
    /**
     * Creates an empty node.
     */
    public DoubleNode()
    {
       next = null;
       element = null;
       previous = null;
    }
```

```
/**
 * Creates a node storing the specified element.
 *
 * @param elem  the element to be stored into the new node
 */
public DoubleNode (E elem)
{
   next = null;
   element = elem;
   previous = null;
}

/**
 * Returns the node that follows this one.
 *
 * @return  the node that follows the current one
 */
public DoubleNode<E> getNext()
{
   return next;
}
```

```
    /**
     * Sets the node that precedes this one.
     * @param dnode  the node to be set as the one to precede
     * the current one */
    public void setPrevious (DoubleNode<E> dnode) {
       previous = dnode;
    }

    /**
     * Returns the element stored in this node.
     * @return  the element stored in this node */
    public E getElement() {
       return element;
    }

    /**
     * Sets the element stored in this node.
     * @param elem  the element to be stored in this node */
    public void setElement (E elem){
       element = elem;
    }
}
```

Lista Duplamente Ligada



Listas Duplamente Ligadas

- Vamos rever o método `remove` para a implementação em lista ligada, desta vez usando uma **lista duplamente ligada**
  - **Exercício:** Realizar as operações `remove` e `add` para a lista duplamente ligada
- Observe como o código desta versão é muito mais elegante comparada com a versão anterior
- O custo dessa elegância é a sobrecarga associada ao armazenamento e gestão de *links* adicionais

Análise das Implementações da Lista

- Em ambas implementações de *array* e listas ligadas a maior parte das operações são similares em termos de eficiência

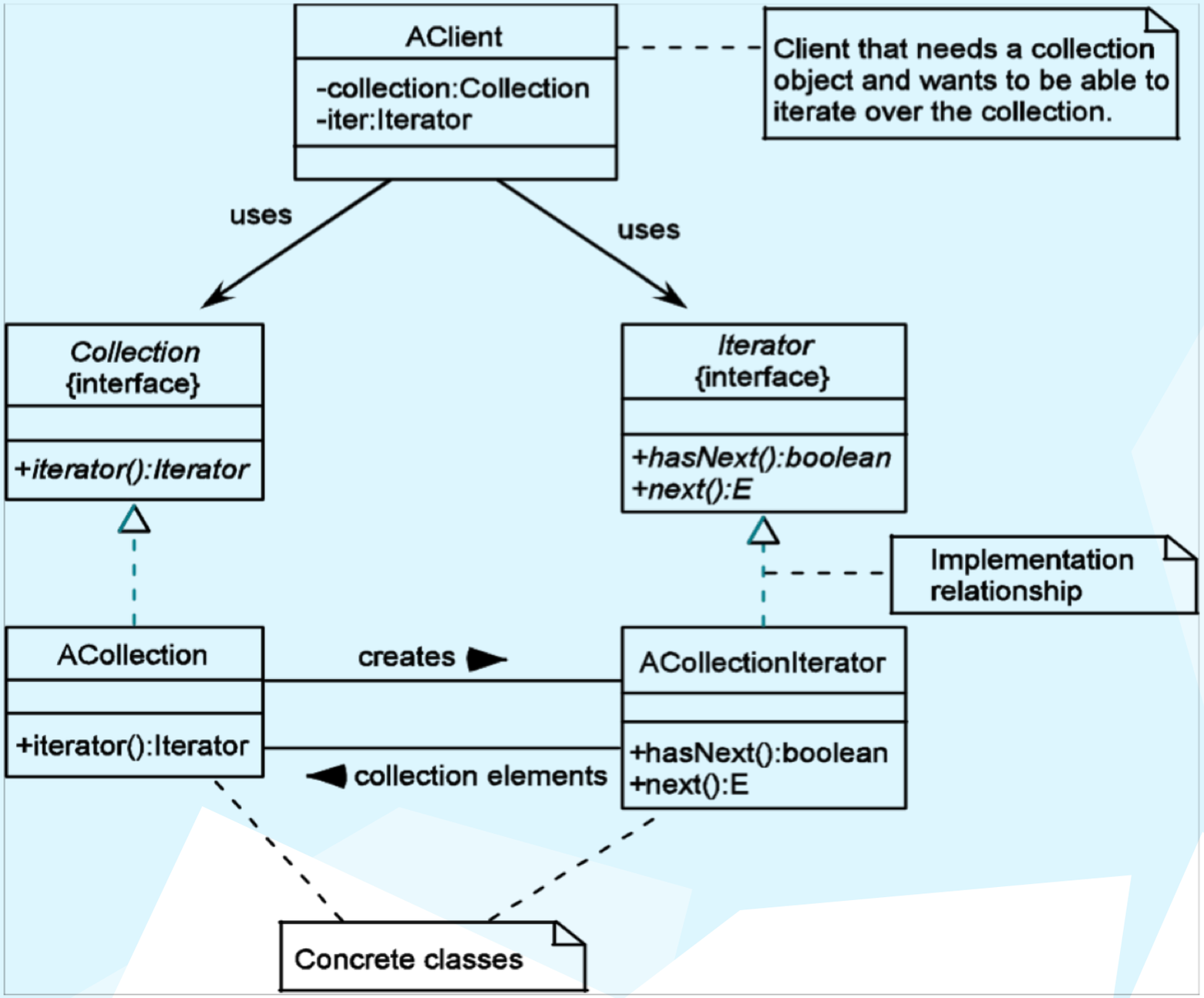
- A maioria são **O(1)**, excepto no *shift* ou na pesquisa, caso em que são **O(n)**

- Em certos casos, a frequência da necessidade de determinadas operações podem orientar o uso de uma abordagem sobre a outra

Padrão de *Software Iterator*

- **Problema:** A necessidade de percorrer os elementos de uma colecção de uma forma comum para todos os tipos de colecções
- **Solução:** Fornecer uma interface `iterator` (iterador) que regule como os iteradores se irão comportar e obrigar a que um objecto da colecção forneça um objecto `iterator` capaz de percorrer todos os elementos da colecção

Estrutura do Padrão *Iterator*

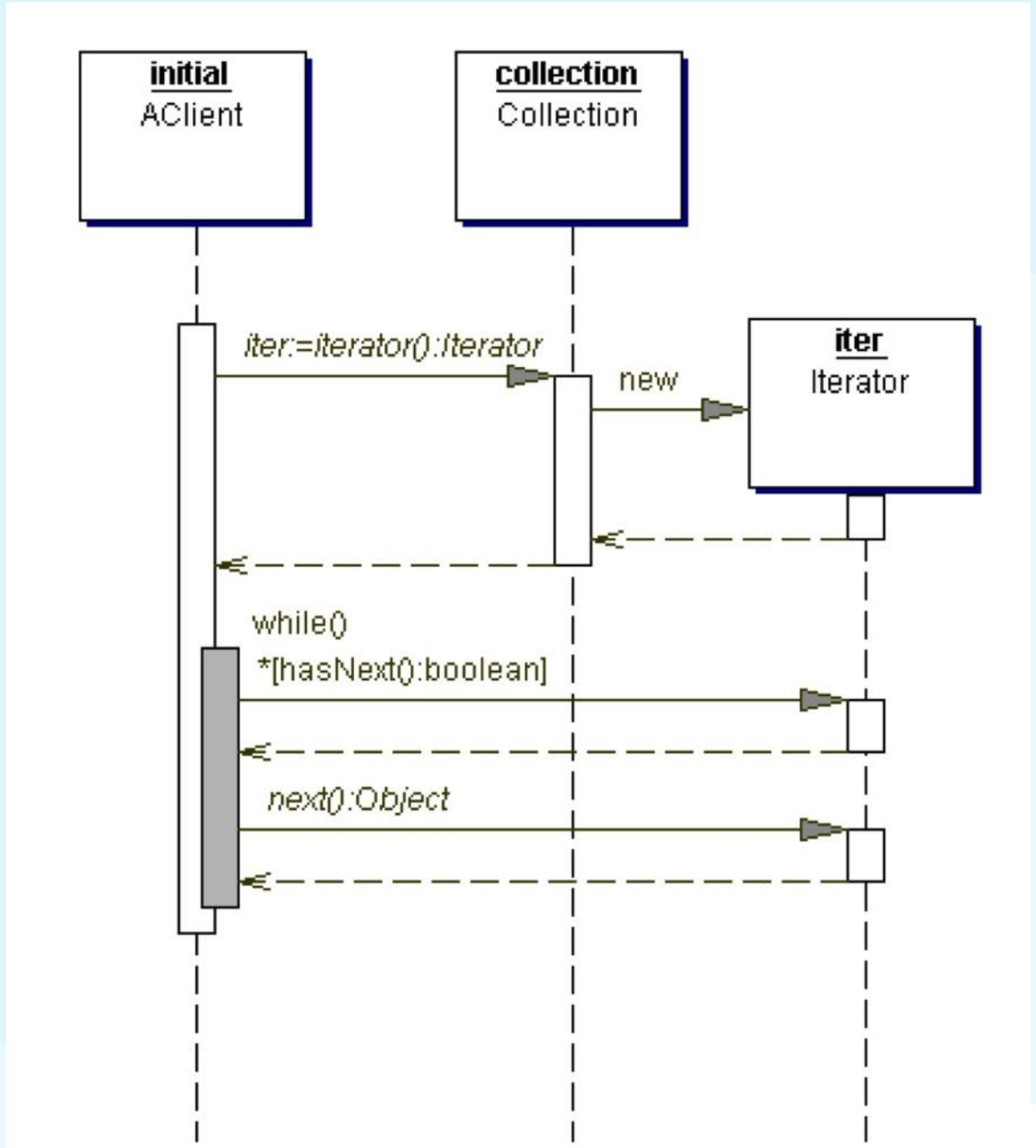


Comportamento do *Iterator*

1. o cliente pergunta ao objecto `collection` pelo objecto `iterator`

2. o cliente usa o objecto `iterator` para percorrer os elementos da `collection`

Atenção: Podem existir múltiplos iteradores simultaneamente activos numa colecção

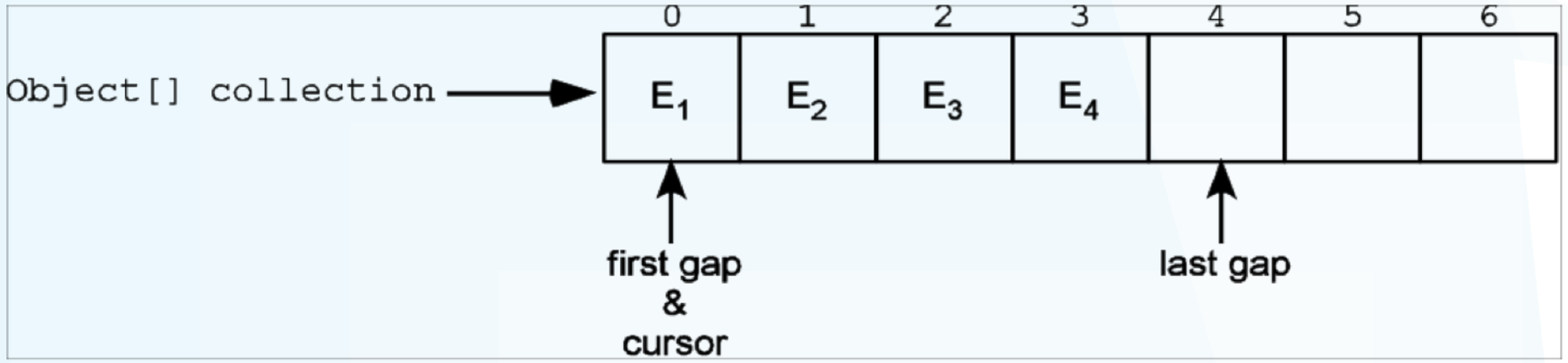


Interface `Iterator<E>`

Required Methods	
`boolean hasNext()`	Returns true if the iteration has more elements.
`E next()`	Returns the next element in the iteration.
Optional Methods All optional methods not fully supported by an implementing class must throw an `UnsupportedOperationException`.	
`void remove()`	Removes from the underlying collection the element returned by the last call to `next()`. Once an element as been removed, `remove()` cannot be called again until another call to `next()` has been made.

Definir um `BasicIterator<E>`

- Para ter acesso ao elementos da coleção, o iterador precisa ter conhecimento interno da implementação da **colecção**
  - criar o `BasicIterator` como uma inner class
- Mapear o modelo lógico de um iterador para a estrutura que armazena os elementos da **coleção**



- o *cursor* armazena o *index* do elemento a ser retornado pelo `next()`

`Iterator`: Modificações concorrentes

- O que acontece se a colecção for modificada enquanto uma iteração está em curso?
- Este problema é denominado de modificação concorrente e ocorre quando a colecção é modificada estruturalmente a partir do objecto `collection` enquanto uma interacção está em curso

- Alguns cenários:
  - Um elemento é adicionado à colecção e é colocado antes da posição do cursor da sequência de iteração em curso - Neste caso a alteração será passada despercebida
  - O ultimo elemento da sequência de iteração é removido de forma a que o cursor está agora na ultima posição, talvez confundindo o método `hasNext()`
    - `throw a ConcurrentModificationException`

Modificações concorrentes: Dois problemas

- **Problema 1:** Detectar modificação concorrente
- **Solução:** adicionar uma variável `modcount` à Colecção para manter um registo das modificações estruturais (*adds*/*deletes*)
  - `modcount` é incrementado cada vez que o objecto da colecção é estruturalmente modificado a partir dos métodos `add` ou `remove`
  - Quando um objecto `BasicIterator` é criado, copia a variável `modcount` da colecção para uma variável local, `expectedModCount`

- Os métodos do iterador procuram por modificações concorrentes verificando se a variável `expectedModCount` é igual a `modCount`

- Se não existirem alterações na estrutura da colecção as duas variáveis serão iguais, caso contrário será lançada uma excepção

- **Problema 2:** `Iterator.remove()` tem duas cláusulas
  - o elemento a remover é o ultimo elemento a ser retornado pelo `next()`
  - `Iterator.remove()` apenas pode ser invocado uma vez por chamada ao `next()` (já que o `next()` fornece o elemento ao `remove`)

- **Solução:** O problema pode ser resolvido com recurso a uma flag `okToRemove` na classe `BasicIterator` que é
  - Inicializada a `false`
  - Colocada a `true` pelo `next()`
  - colocada a `false` pelo `Iterator.remove()`
  - Desta forma irá garantir que cada chamada ao `remove()` estará coordenada com o método `next()`