Estruturas de Dados

Colecções
Listas

2025/2026

Colecções

- Bibliotecas de estruturas de dados e algoritmos estão estão disponibilizadas para várias linguagens:
  - C++ : STL – Standard Template Library
  - C# : Collection Classes
  - Java: Collection API

API de Colecções

- Todas as estruturas de dados implementam o interface *Collection<T>.*
  - **Inserção/Remoção***: add, addAll, remove, removeAll, retainAll, clear.*
  - **Conversão para array**: *toArray*
  - **Suporte para iteradores**: *iterator*
  - **Informação e pesquisa:** *isEmpty, size, contains, containsAll*

API de Colecções

- Os algoritmos são disponibilizados como métodos estáticos da classe `java.util.Collections` ou `java.util.Arrays`
  - *Cópia, pesquisa binária, contagem de objectos, manipulação, substituição, ordenação e outros*

Retrocompatibilidade

- Apesar de todos os contentores de dados serem genéricos, podendo conter objectos de qualquer tipo, alguns métodos manipulam objectos, por motivos de compatibilidade com versões antigas da linguagem:
  - `boolean contains(Object x)` – Devolve verdadeiro se o Objecto está no contentor.
  - `boolean remove(Object x)`- Devolve verdadeiro se o objecto foi removido

Conversão para Array

- Exemplo

```
Collection<String> col=…;
String [] m=new String[col.size()];
Col.toArray(m);
```

Construção de Colecções

- Não é possível especificar os construtores através de um interface.
- No entanto, por convenção, todas as implementações de *Collection* deverão possuir os seguintes construtores:
  - *Construtor sem parâmetros*
  - *Construtor que cria uma colecção que referencia os mesmos elementos que outra colecção*

Construção de Colecções

```
class Pacote <X> implements Collection<X>
{
  Pacote(){
//cria colecção vazia;
  …
  }

  Pacote(Collection<? extends X> c){
  //cria uma colecção que referencia
  //os mesmos elementos que c
  …
  }
…
}
```

Métodos Opcionais

- O que acontece se um dos métodos requerido pelo interface não for suportado?
  - Por exemplo, remoção de elementos em colecções imutáveis?
- Está prevista a existência de métodos opcionais!
  - A execução de um método opcional que não é implementado resulta na geração de uma excepção *UnsupportedOperationException*.

Algoritmos Genéricos

- Os algoritmos são fornecidos como métodos estáticos de classes como *Collections e Arrays*.
  - A maior parte dos métodos é genérica, existindo também algumas versões *overloaded* que trabalham com tipos básicos de dados.
  - Alguns dos métodos usam *function objects* (Functors) para maior flexibilidade.

Functores

- Um *functor* (ou objecto-função) é um objecto que encapsula uma função/método.
  - São utilizados, no contexto do API de colecções, para variar a funcionalidade de alguns dos algoritmos.

Functor - Comparator

- Problemas:
  - Como ordenar objectos de classes que não são comparáveis ?
  - Como ordenar *strings* de acordo com o segundo caracter?
- Este tipo de problema é resolvido através da criação de um objecto *Comparador* adequado!

Functor - Comparator

```
interface Comparator<T>{
 public int compare(T o1, T o2);
 }
```

```
 class Comparador2Char implements
   Comparator<String>
 {
 Public int compare(String o1, String o2)
   {
   Character c1=new Character(o1.charAt(1));
   Character c2=new Character(o2.charAt(1));
   return o1.compareTo(o2);
   }
 }
```

Functores

- O objecto comparador é utilizado pelo algoritmo de ordenação para determinar a ordem relativa entre 2 objectos:

```
List<String> c= …
sort(c,new  Comparator2Char());
// c fica ordenado de acordo com o 2º
  caracter das Strings que armazena
```

Comparador por omissão

- O comparador por omissão de um determinado tipo E é definido como:

```
class DefaultComparator
<E extends Comparable<? super E> >
  implements Comparator<E>
  {
  public int compare( E lhs, E rhs )
   {
  return lhs.compareTo( rhs );
   }
  }
```

Algoritmos

- **Pesquisa Binária**
  - `Arrays.binarySearch` – versões *overloaded* para tipos básicos de dados, bem como duas versões para objectos (com e sem um functor *Comparator*)

  - `int binarySearch(Object[] a, Object key)`
  - `static<T> int binarySearch(T[] a, T key, Comparator<? super T> c)`

    - Caso o elemento exista, devolve a sua posição.
    - Caso o elemento não exista, devolve o inverso da primeira posição com um elemento de valor superior.
      - P.ex: se devolver -6, então o 6º elemento é o primeiro elemento maior do que o vaor procurado.

Algoritmos - Ordenação

- **Ordenação**
  - `Arrays.sort` – versões *overloaded* para tipos básicos de dados, bem como duas versões para objectos (com e sem um functor *Comparator*)

```
- static void sort(Object[] array)
- static <T> void sort(T[] a, Comparator<? super T> c)
```

Tipos de Colecções - List

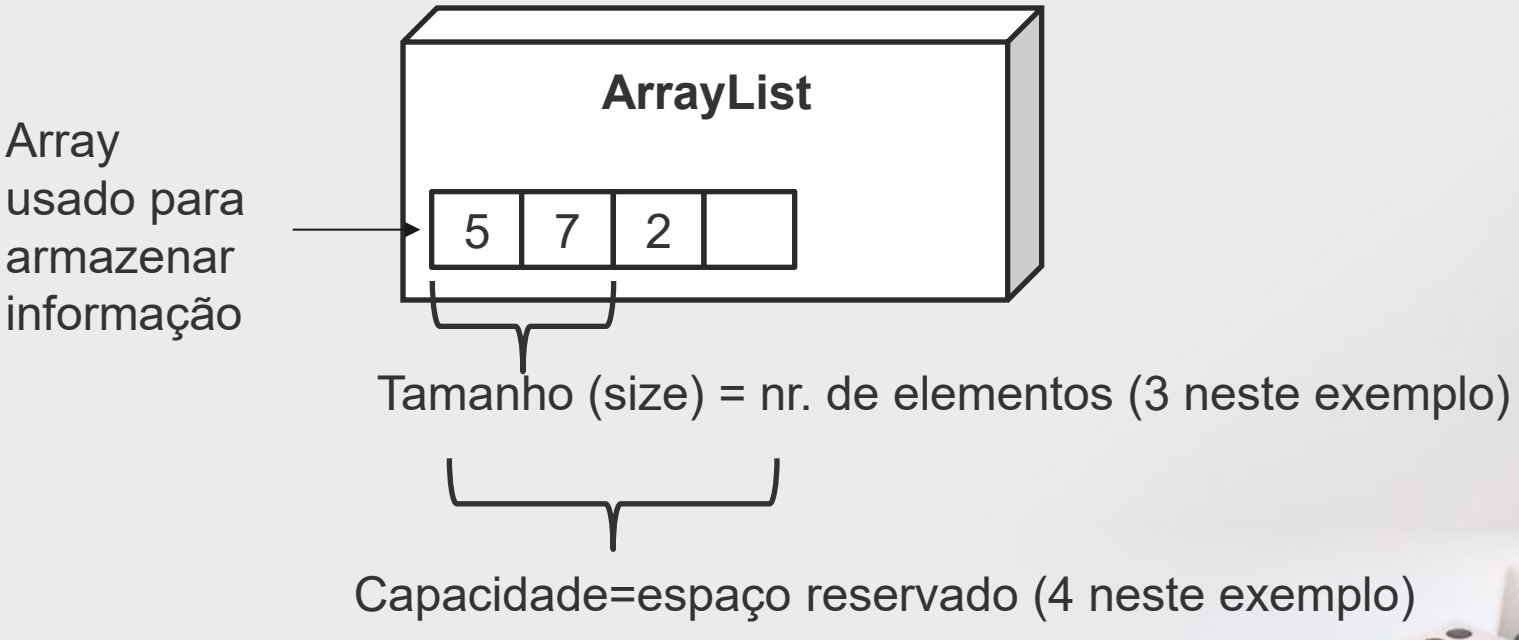
- **Há várias especializações do interface *Collection*:**
- ***List*** – Colecção na qual os items têm uma posição.Tem duas implementações básicas:
  - *LinkedList:*
  - *ArrayList* (ou Vector)
- Acrescenta métodos relacionados com o posicionamento dos elementos, p.ex:
  - indexOf(Object o), set(int index, E element), get(index), etc, …
- Acrescenta também um iterador bidireccional: *ListIterator*
  - *listIterator() e listIterator(int index)*

ListIterator

- void **add**(E e) – Insere Elemento na lista (opcional)
- Boolean **hasPrevious**() – Indica se há um elemento atrás do posição actual
- int **nextIndex**() – Retorna indice do próximo elemento
- E **previous**() - Retorna elemento anterior.
- void **set**(E e) – modifica o ultimo elemento devolvido (opcional)

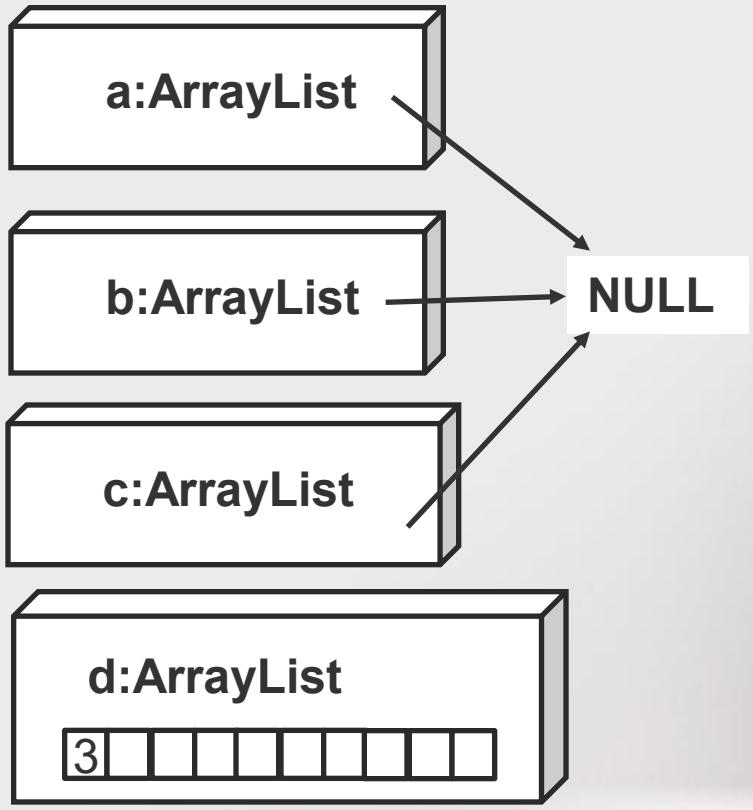
Operações sobre ArrayLists

- Organização interna



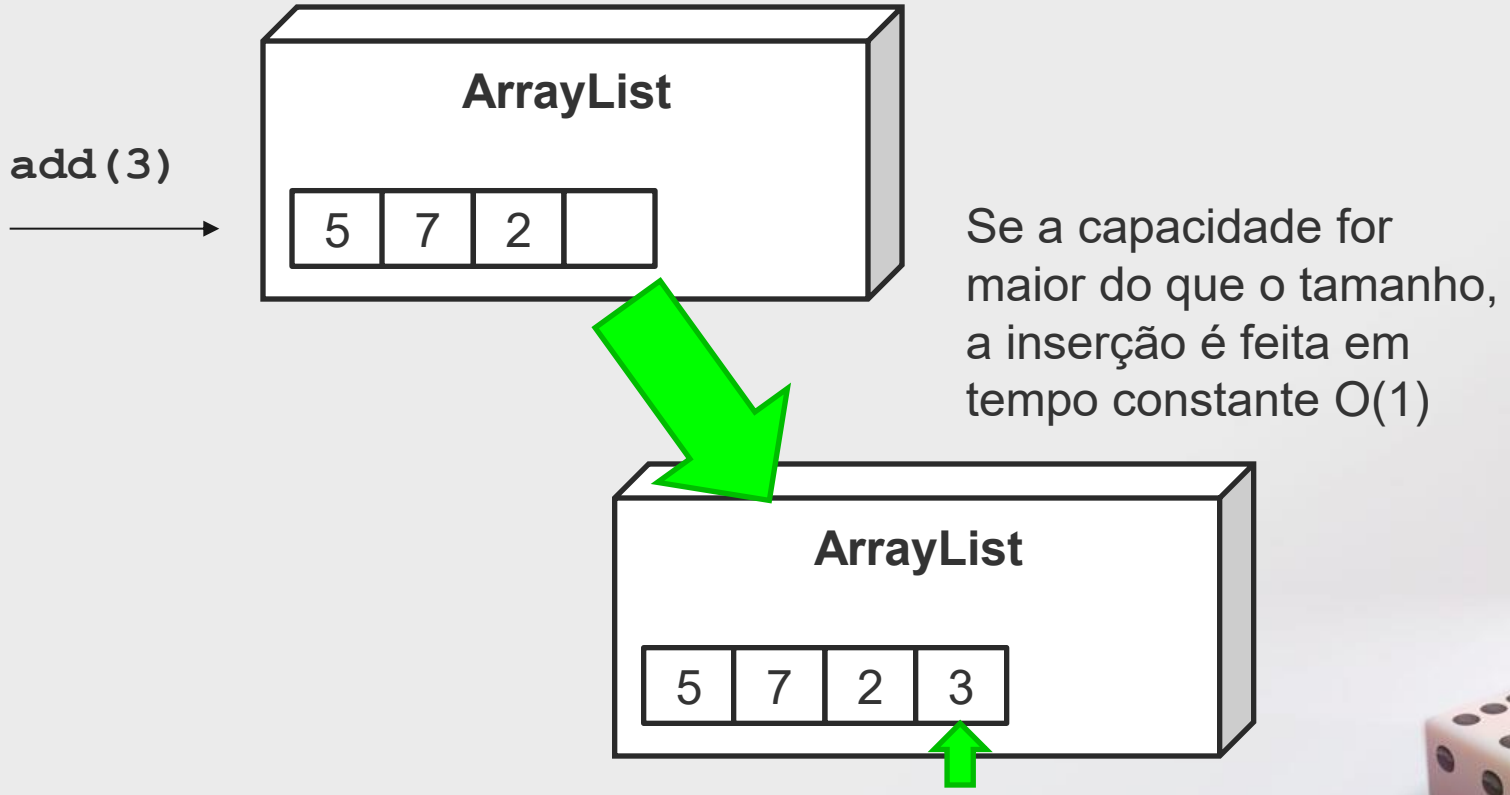
Operações sobre ArrayLists

- Inicialização (Java 8+):
  - A capacidade inicial é 10, mas o criador pode especificar outra capacidade inicial.
    - No entanto, para poupar memória, todas as instâncias de array list inicializadas vazias referem um array nulo.
    - É reservado um array com a capacidade inicial no momento de inserção do primeiro valor.



Operações sobre ArrayLists

- Acrescentar um valor

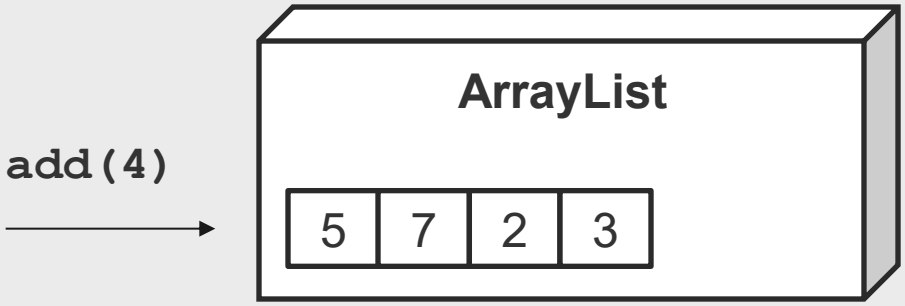


Se a capacidade for maior do que o tamanho, a inserção é feita em tempo constante O(1)

Operações sobre ArrayLists

- Acrescentar um valor

Se a capacidade for igual ao tamanho, a inserção é feita em tempo constante O(1)

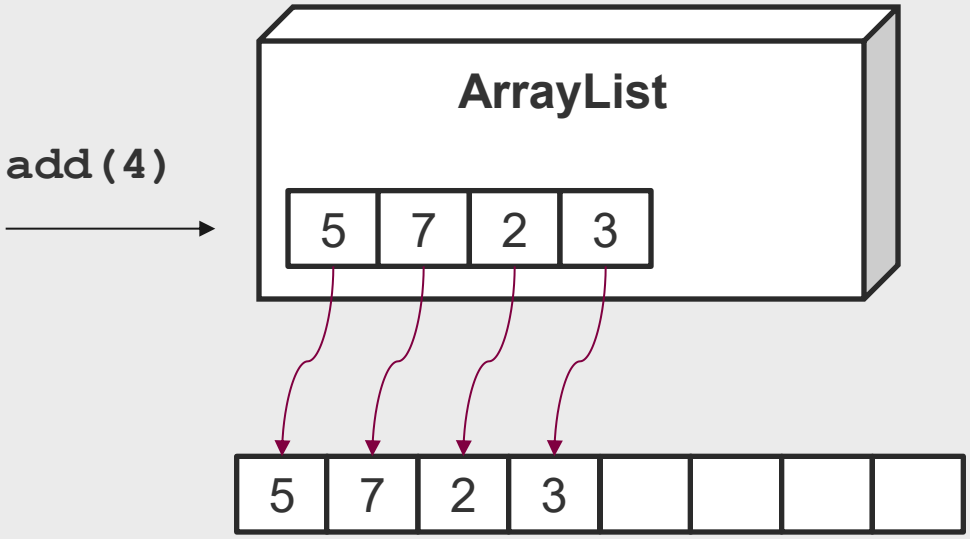


1ºpasso - Alocação de um array maior

Operações sobre ArrayLists

- Acrescentar um valor

Se a capacidade for igual ao tamanho, a inserção é feita em tempo linear O(N)

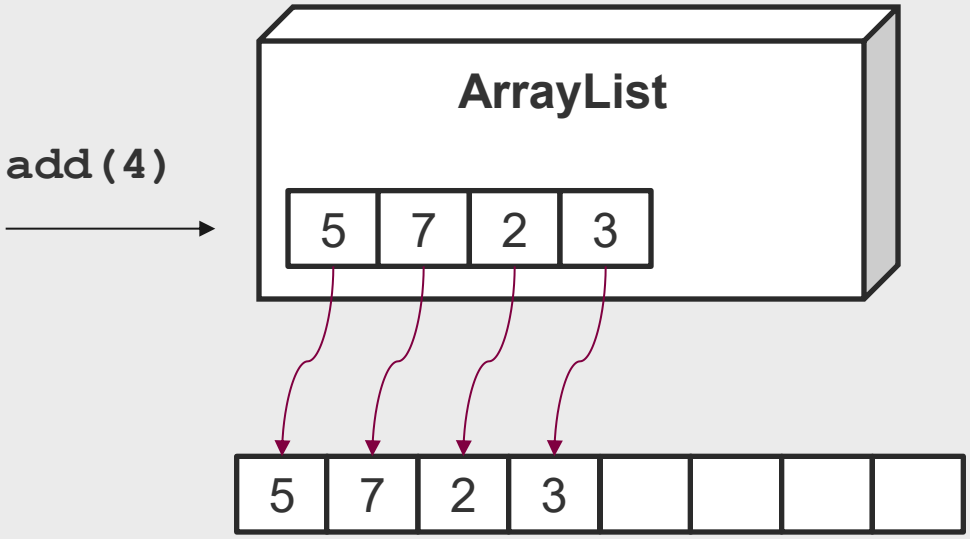


2ºpasso – Cópia de N valores

Operações sobre ArrayLists

- Acrescentar um valor

Se a capacidade for igual ao tamanho, a inserção é feita em tempo linear O(N)

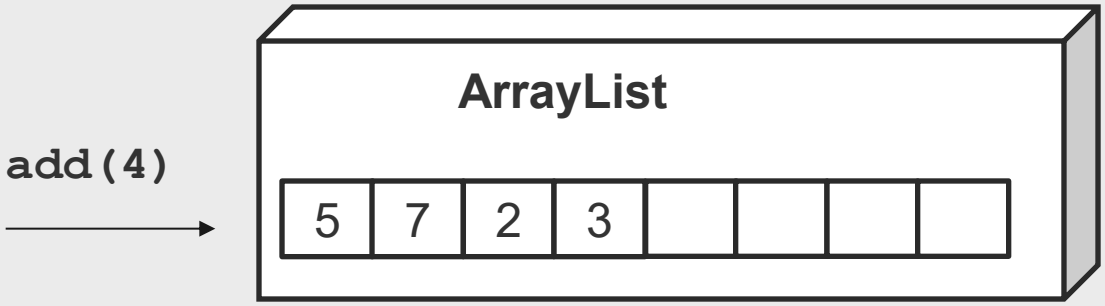


2ºpasso – Cópia de N valores. ***Demora tempo proporcional a N***

Operações sobre ArrayLists

- Acrescentar um valor

Se a capacidade for igual ao tamanho, a inserção é feita em tempo linear O(N)

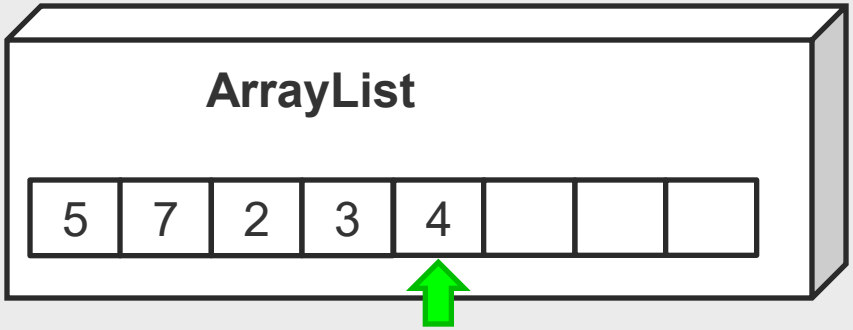


3ºpasso – Substituir array original por array novo

Operações sobre ArrayLists

- Acrescentar um valor

Se a capacidade for igual ao tamanho, a inserção é feita em tempo linear O(N)

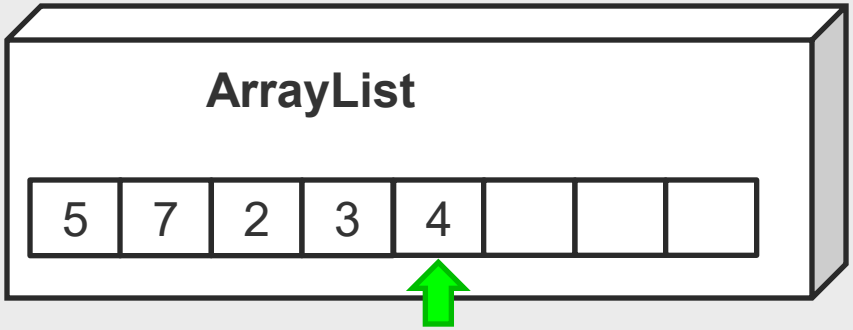


4ºpasso – colocar valor novo em ultima posição

Operações sobre ArrayLists

- Acrescentar um valor

Se a capacidade for igual ao tamanho, a inserção é feita em tempo linear O(N)



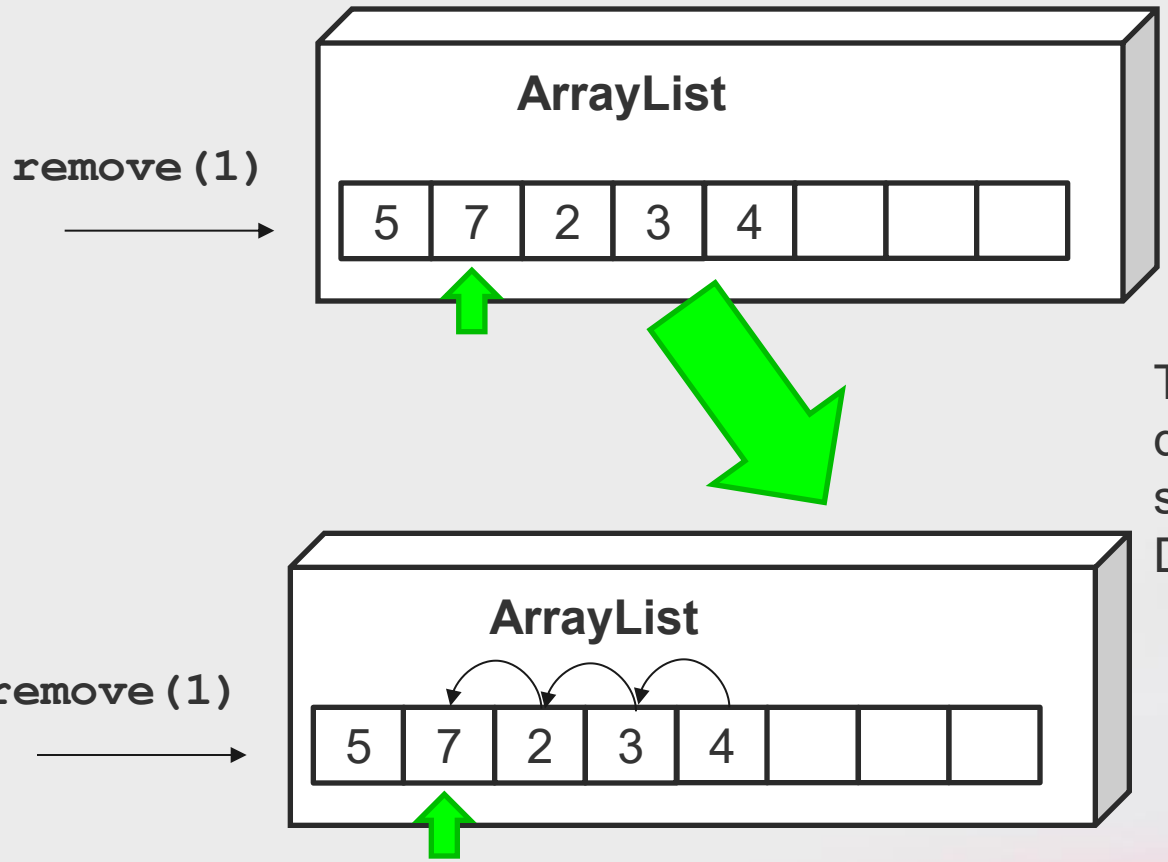
Se o aumento da capacidade for sempre proporcional à capacidade anterior (p.ex: 2x), então atinge-se comportamento constante amortizado.

Uma operação tem custo O(1) ***amortizado*** se o custo de a executar N vezes é O(N) *(mesmo que nem todas as execuções sejam O(1)).*

Isto deve-se à diluição do custo de inserções O(N) pelas inserções O(1).

Operações sobre ArrayLists

- Remover um valor em posição

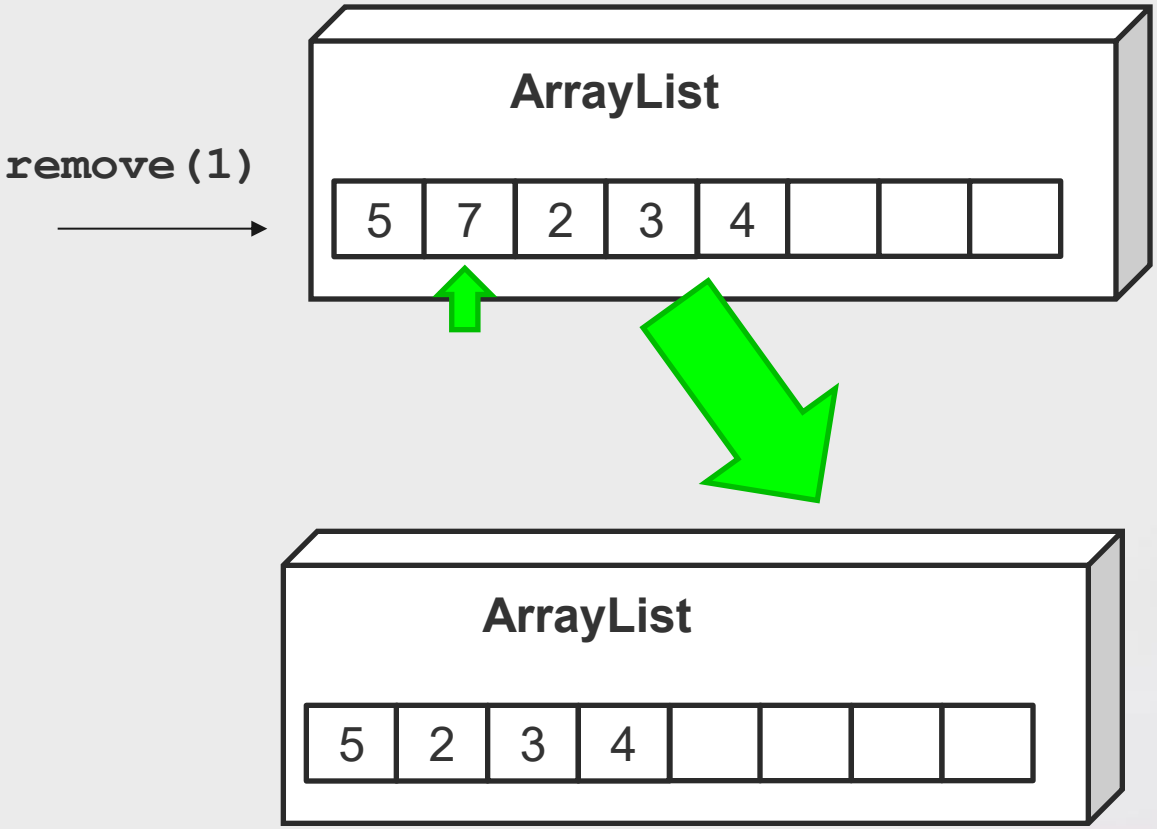


É feito em tempo linear O(N)

Todos os valores à frente de elemento removido são copiados para trás. Demora tempo linear

Operações sobre ArrayLists

- Remover um valor em posição



É feito em tempo linear O(N)

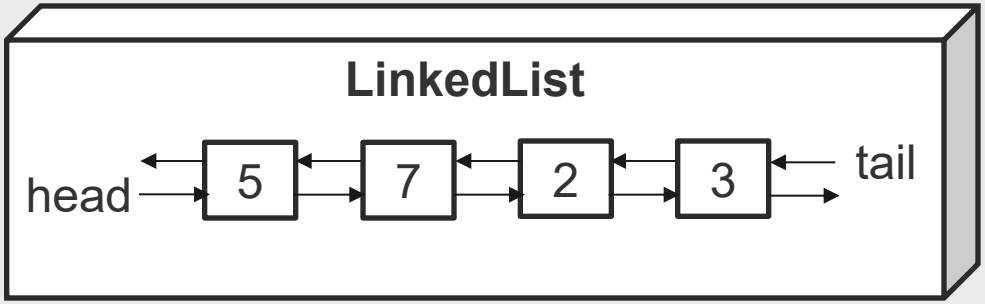
Todos os valores à frente de elemento removido são copiados para trás. Demora tempo linear.

Operações sobre ArrayLists

Operação	Complexidade
Add(obj)	O(1) amortizado. *O(N) se capacidade esgotada,* *O(1) caso contrário.*
Get(pos)	O(1)
Set(pos,obj)	O(1)
Remove(obj)	O(N) – É preciso procurar o obj
Add(pos,obj)	O(N)
Iterador.add(obj)	O(N)
Iterador.set(obj)	O(1)
Iterador.remove()	O(N)
Remove(ultima posição)	O(1) –Não há valores à frente...

Operações sobre LinkedLists

- Listas duplamente ligadas



O acesso a uma posição é feito iterando uma posição de cada vez a partir do fim ou início da lista…
É por isso de ordem linear

Qualquer acesso a uma posição explicita será linear

Operações sobre LinkedLists

Operação	Complexidade
Get(pos)	O(N) – acesso a posição explícita
Set(pos,obj)	O(N) – acesso a posição explícita
Remove(obj)	O(N) – É preciso procurar…
Add(pos,obj)	O(N) – acesso a posição explícita
Iterador.add(obj)	O(1)
Iterador.set(obj)	O(1)
Iterador.remove()	O(1)
Get, Set, Remove, Add (primeira ou ultima posição)	O(1) – Acesso direto a primeiro e ultimo

Tipos de Colecções

- **Há várias especializações do interface *Collection*:**
- ***Set*** – Conjunto: colecção que não contém duplicados.
  - *SortedSet: Conjunto de elementos ordenáveis*
  - *HashSet: Conjunto (não ordenado)*

Tipos de Colecções

- **Há várias especializações do interface *Collection*:**
- ***Queue*** – Fila: permite a inserção, extracção e inspecção e elementos.
  - PriorityQueue: fila na qual os elementos estão ordenados de acordo com um determinado critério.

Tipos de Colecções

- **Fora da hierarquia de colecções:**
- ***Map*** – Mapa: colecção de pares de *chaves* e respectivos *valores*.
  - *TreeMap: Mapa ordenado por chave*
  - *HashMap: Mapa não ordenado*