

# Ordenação de estruturas lineares

## Definições e algoritmos

Prof. Marcelo de Souza

45EST – Algoritmos e Estruturas de Dados  
Universidade do Estado de Santa Catarina



## Leitura principal:

- ▶ Capítulo 4 de **Ziviani2010** (**Ziviani2010**)<sup>1</sup> – Ordenação.

## Leitura complementar:

- ▶ Capítulo 13 de **Pereira2008** (**Pereira2008**)<sup>2</sup> – Ordenação e busca.
- ▶ Capítulo 15 de **Preiss2001** (**Preiss2001**)<sup>3</sup> – Algoritmos de ordenação e ordenadores.

---

<sup>1</sup>Ziviani2010.

<sup>2</sup>Pereira2008.

<sup>3</sup>Preiss2001.

# Ordenação de estruturas lineares

## Conceitos básicos



**Ordenar** uma estrutura consiste em rearranjar seus elementos, respeitando uma dada ordem.

- ▶ Geralmente, usamos ordem crescente ou ordem decrescente.



# Ordenação de estruturas lineares

## Conceitos básicos



**Ordenar** uma estrutura consiste em rearranjar seus elementos, respeitando uma dada ordem.

- ▶ Geralmente, usamos ordem crescente ou ordem decrescente.

Podemos ordenar qualquer coleção de itens, desde que sejam comparáveis uns aos outros.

- ▶ **Números** (ordem crescente/decrescente) ou **strings** (ordem alfabética), já comparáveis;
- ▶ **Livros** (ordem dada pelo título, autor, ano ou páginas, por exemplo).
  - ▶ Necessário implementar a interface `Comparable` e o método `compareTo`.



# Ordenação de estruturas lineares

## Conceitos básicos



**Ordenar** uma estrutura consiste em rearranjar seus elementos, respeitando uma dada ordem.

- ▶ Geralmente, usamos ordem crescente ou ordem decrescente.

Podemos ordenar qualquer coleção de itens, desde que sejam comparáveis uns aos outros.

- ▶ **Números** (ordem crescente/decrescente) ou **strings** (ordem alfabética), já comparáveis;
- ▶ **Livros** (ordem dada pelo título, autor, ano ou páginas, por exemplo).
  - ▶ Necessário implementar a interface `Comparable` e o método `compareTo`.

A eficiência de um algoritmo de ordenação é muito importante, especialmente quando tratamos estruturas com grandes volumes de dados.



Alguns dos muitos algoritmos de ordenação, com suas complexidades assintóticas de tempo.

Algoritmo	Caso médio	Melhor caso	Pior caso
Insertion sort	$\mathcal{O}(n^2)$	$\mathcal{O}(n)$	$\mathcal{O}(n^2)$
Selection sort	$\mathcal{O}(n^2)$	$\mathcal{O}(n^2)$	$\mathcal{O}(n^2)$
Shell sort	$\mathcal{O}(n^{1.5})$	$\mathcal{O}(n)$	$\mathcal{O}(n^2)$
Merge sort	$\mathcal{O}(n \log n)$	$\mathcal{O}(n \log n)$	$\mathcal{O}(n \log n)$
Quick sort	$\mathcal{O}(n \log n)$	$\mathcal{O}(n \log n)$	$\mathcal{O}(n^2)$
Radix sort	$\mathcal{O}(n)$	$\mathcal{O}(n)$	$\mathcal{O}(n)$

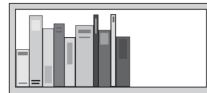
Algumas observações:

- ▶ Apesar do método **radix sort** ter complexidade linear, ele não é aplicável em muitos casos. Logo, o melhor desempenho para o caso geral é  $\mathcal{O}(n \log n)$ .
- ▶ O pior caso do método **quick sort** é facilmente evitado escolhendo pivôs apropriados. Por ser mais rápido que o merge sort na prática, esse é o algoritmo de ordenação mais usado.

# Insertion sort

## Ideia geral

**Cenário:** queremos ordenar livros, de modo que o menor fique à esquerda.

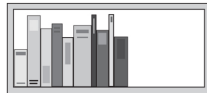


# Insertion sort

## Ideia geral

**Cenário:** queremos ordenar livros, de modo que o menor fique à esquerda.

- ▶ O primeiro livro está na sua posição inicial.
- ▶ Analisamos o segundo livro (da esquerda para a direita).
  1. Se ele for maior que o primeiro livro, os dois livros já estão ordenados.



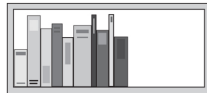


# Insertion sort

## Ideia geral

**Cenário:** queremos ordenar livros, de modo que o menor fique à esquerda.

- ▶ O primeiro livro está na sua posição inicial.
- ▶ Analisamos o segundo livro (da esquerda para a direita).
  1. Se ele for maior que o primeiro livro, os dois livros já estão ordenados.
  2. Caso contrário, removemos o segundo livro, deslocamos o primeiro livro para a direita, e inserimos o livro removido na primeira posição.

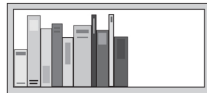


# Insertion sort

## Ideia geral

**Cenário:** queremos ordenar livros, de modo que o menor fique à esquerda.

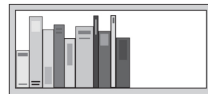
- ▶ O primeiro livro está na sua posição inicial.
- ▶ Analisamos o segundo livro (da esquerda para a direita).
  1. Se ele for maior que o primeiro livro, os dois livros já estão ordenados.
  2. Caso contrário, removemos o segundo livro, deslocamos o primeiro livro para a direita, e inserimos o livro removido na primeira posição.
  3. Os dois primeiros livros estão ordenados.



# Insertion sort

## Ideia geral

**Cenário:** queremos ordenar livros, de modo que o menor fique à esquerda.



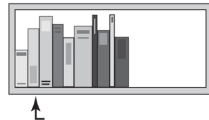
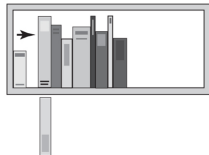
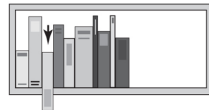
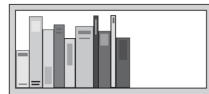
- ▶ O primeiro livro está na sua posição inicial.
- ▶ Analisamos o segundo livro (da esquerda para a direita).
  1. Se ele for maior que o primeiro livro, os dois livros já estão ordenados.
  2. Caso contrário, removemos o segundo livro, deslocamos o primeiro livro para a direita, e inserimos o livro removido na primeira posição.
  3. Os dois primeiros livros estão ordenados.
- ▶ Agora analisamos o terceiro livro.
  1. Se ele for maior que o segundo livro, os três livros já estão ordenados.

# Insertion sort

## Ideia geral

**Cenário:** queremos ordenar livros, de modo que o menor fique à esquerda.

- ▶ O primeiro livro está na sua posição inicial.
- ▶ Analisamos o segundo livro (da esquerda para a direita).
  1. Se ele for maior que o primeiro livro, os dois livros já estão ordenados.
  2. Caso contrário, removemos o segundo livro, deslocamos o primeiro livro para a direita, e inserimos o livro removido na primeira posição.
  3. Os dois primeiros livros estão ordenados.
- ▶ Agora analisamos o terceiro livro.
  1. Se ele for maior que o segundo livro, os três livros já estão ordenados.
  2. Caso contrário, removemos o terceiro livro, deslocamos o segundo livro para a direita, e verificamos se o livro removido é maior que o primeiro. Caso seja, o inserimos na segunda posição.

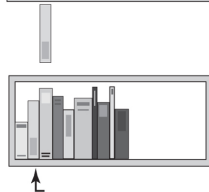
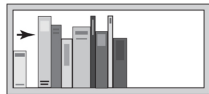
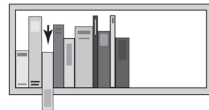
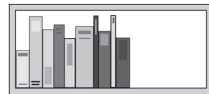


# Insertion sort

## Ideia geral

**Cenário:** queremos ordenar livros, de modo que o menor fique à esquerda.

- ▶ O primeiro livro está na sua posição inicial.
- ▶ Analisamos o segundo livro (da esquerda para a direita).
  1. Se ele for maior que o primeiro livro, os dois livros já estão ordenados.
  2. Caso contrário, removemos o segundo livro, deslocamos o primeiro livro para a direita, e inserimos o livro removido na primeira posição.
  3. Os dois primeiros livros estão ordenados.
- ▶ Agora analisamos o terceiro livro.
  1. Se ele for maior que o segundo livro, os três livros já estão ordenados.
  2. Caso contrário, removemos o terceiro livro, deslocamos o segundo livro para a direita, e verificamos se o livro removido é maior que o primeiro. Caso seja, o inserimos na segunda posição.
  3. Caso contrário, deslocamos o primeiro livro para a direita, e inserimos o livro removido na primeira posição.

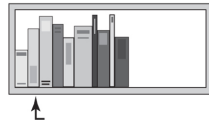
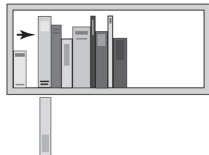
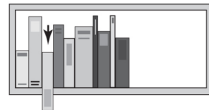
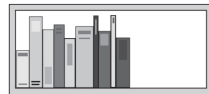


# Insertion sort

## Ideia geral

**Cenário:** queremos ordenar livros, de modo que o menor fique à esquerda.

- ▶ O primeiro livro está na sua posição inicial.
- ▶ Analisamos o segundo livro (da esquerda para a direita).
  1. Se ele for maior que o primeiro livro, os dois livros já estão ordenados.
  2. Caso contrário, removemos o segundo livro, deslocamos o primeiro livro para a direita, e inserimos o livro removido na primeira posição.
  3. Os dois primeiros livros estão ordenados.
- ▶ Agora analisamos o terceiro livro.
  1. Se ele for maior que o segundo livro, os três livros já estão ordenados.
  2. Caso contrário, removemos o terceiro livro, deslocamos o segundo livro para a direita, e verificamos se o livro removido é maior que o primeiro. Caso seja, o inserimos na segunda posição.
  3. Caso contrário, deslocamos o primeiro livro para a direita, e inserimos o livro removido na primeira posição.
  4. Os três primeiros livros estão ordenados.



# Insertion sort

## Ideia geral

**Cenário:** queremos ordenar livros, de modo que o menor fique à esquerda.

- ▶ Repetimos esses passos para todos os demais livros.

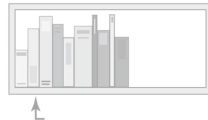
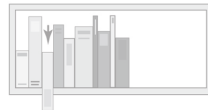
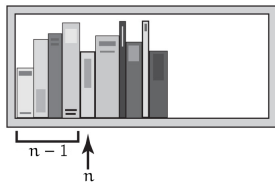


# Insertion sort

## Ideia geral

**Cenário:** queremos ordenar livros, de modo que o menor fique à esquerda.

- ▶ Repetimos esses passos para todos os demais livros.
  1. Ao analisar o  $n$ -ésimo livro, os  $n - 1$  primeiros livros estão ordenados.
  2. Removemos o  $n$ -ésimo livro (ainda não ordenado).
  3. Deslocamos os livros ordenados para a direita, um a um, até encontrar a posição correta para o  $n$ -ésimo livro.
  4. Inserimos o livro na sua nova posição.

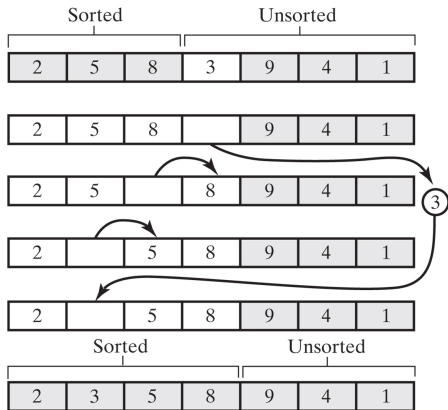




# Insertion sort

## Exemplo de funcionamento

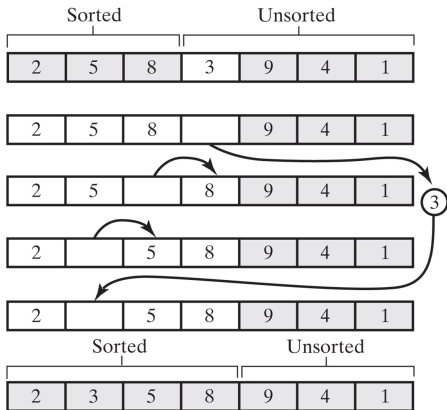
A mesma ideia se aplica para qualquer coleção de objetos comparáveis, como números inteiros.



# Insertion sort

## Exemplo de funcionamento

A mesma ideia se aplica para qualquer coleção de objetos comparáveis, como números inteiros.



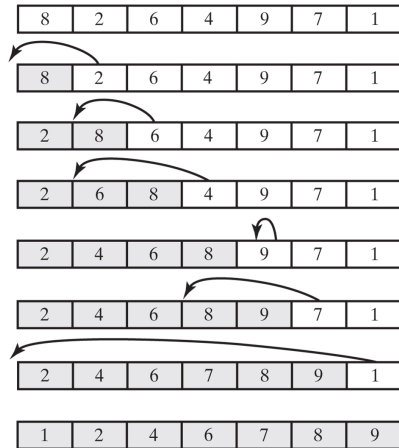
**Exemplo:** processamento do elemento 3.

1. Os três primeiros elementos estão ordenados.
2. Remove o quarto elemento (3) da sua posição.
3. Desloca o 8, uma vez que  $8 > 3$ .
4. Desloca o 5, uma vez que  $5 > 3$ .
5. Insere na posição vaga, uma vez que  $3 > 2$ .
6. Ao final, os quatro primeiros elementos estão ordenados.

# Insertion sort

## Exemplo de funcionamento

A mesma ideia se aplica para qualquer coleção de objetos comparáveis, como números inteiros.



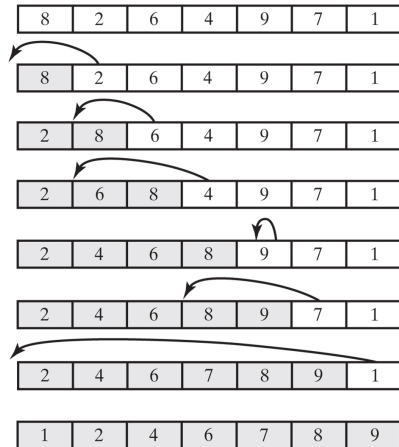
# Insertion sort

## Exemplo de funcionamento

A mesma ideia se aplica para qualquer coleção de objetos comparáveis, como números inteiros.

**Exemplo:** execução completa do algoritmo.

- ▶ A cada iteração, o primeiro elemento é removido da sub-lista não ordenada e inserido na posição correta da sub-lista ordenada, i.e. mantendo sua ordenação.
- ▶ Iterações:
  1. Insere o elemento 2 na primeira posição;
  2. Insere o elemento 6 na segunda posição;
  3. Insere o elemento 4 na segunda posição;
  4. Insere o elemento 9 na quinta posição;
  5. Insere o elemento 7 na quarta posição;
  6. Insere o elemento 1 na primeira posição.
- ▶ Não é usada nenhuma estrutura auxiliar.





# Insertion sort

## Implementação

Insertion sort **simples** para *arrays*:

```
1  public void insertionSort(int[] array) {
2      for (int i = 1; i < array.length; i++)
3          insertInOrder(array[i], array, 0, i - 1);
4  }
5
6  private void insertInOrder(int element, int[] array, int begin, int end) {
7      int index = end;
8      while ((index >= begin) && (element < array[index])) {
9          array[index + 1] = array[index];
10         index--;
11     }
12     array[index + 1] = element;
13 }
```



# Insertion sort

## Implementação

Insertion sort **genérico** para *arrays*:

```
1  public <E extends Comparable<? super E>> void insertionSort(E[] array) {
2      for (int i = 1; i < array.length; i++)
3          insertInOrder(array[i], array, 0, i - 1);
4  }
5
6  private <E extends Comparable<? super E>> void insertInOrder(E element, E[] array, int
   ↳ begin, int end) {
7      int index = end;
8      while ((index >= begin) && (element.compareTo(array[index]) < 0)) {
9          array[index + 1] = array[index];
10         index--;
11     }
12     array[index + 1] = element;
13 }
```



# Insertion sort

## Análise de complexidade

Considerando a ordenação de *arrays*:

- ▶ O laço de repetição principal percorre todos os elementos da estrutura, com exceção do primeiro. Logo, esse laço executa  $n$  vezes.
- ▶ Em cada iteração, o elemento correspondente é selecionado e o método `insertInOrder` busca sua posição no subconjunto dos elementos já ordenados.
- ▶ No **pior caso** (estrutura em ordem decrescente), esse método percorre todos os elementos já ordenados para encontrar a posição correta do elemento a ser inserido. Logo, o algoritmo completo executará  $1 + 2 + \dots + (n - 1) = n(n - 1)/2$  iterações.
- ▶ No **melhor caso** (estrutura ordenada), nenhuma iteração do laço de repetição é executada.

# Insertion sort

## Análise de complexidade



Considerando a ordenação de *arrays*:

- ▶ O laço de repetição principal percorre todos os elementos da estrutura, com exceção do primeiro. Logo, esse laço executa  $n$  vezes.
- ▶ Em cada iteração, o elemento correspondente é selecionado e o método `insertInOrder` busca sua posição no subconjunto dos elementos já ordenados.
- ▶ No **pior caso** (estrutura em ordem decrescente), esse método percorre todos os elementos já ordenados para encontrar a posição correta do elemento a ser inserido. Logo, o algoritmo completo executará  $1 + 2 + \dots + (n - 1) = n(n - 1)/2$  iterações.
- ▶ No **melhor caso** (estrutura ordenada), nenhuma iteração do laço de repetição é executada.

**Portanto:**

- ▶ Pior caso:  $\mathcal{O}(n^2)$ .
- ▶ Melhor caso:  $\mathcal{O}(n)$ .
- ▶ Quanto mais ordenada a estrutura estiver, menor a complexidade do método na prática.



