

# COM112 - ALGORITMO E ESTRUTURA DE DADOS II<sup>1</sup>

Pedro Henrique Del Bianco Hokama  
UNIFEI

---

<sup>1</sup>Baseado nos slides elaborados por Charles Ornelas Almeida, Israel Guerra e Nivio Ziviani

# Ordenação

# Ordenação

- Ordenar: processo de reorganizar um conjunto de objetos em uma ordem ascendente ou descendente.

# Ordenação

- Ordenar: processo de rearranjar um conjunto de objetos em uma ordem ascendente ou descendente.
- A ordenação visa facilitar a recuperação posterior de itens do conjunto ordenado.

# Ordenação

- Ordenar: processo de rearranjar um conjunto de objetos em uma ordem ascendente ou descendente.
- A ordenação visa facilitar a recuperação posterior de itens do conjunto ordenado.
  - ▶ Dificuldade de se utilizar um catálogo telefônico se os nomes das pessoas não estivessem listados em ordem alfabética.

# Ordenação

- Ordenar: processo de rearranjar um conjunto de objetos em uma ordem ascendente ou descendente.
- A ordenação visa facilitar a recuperação posterior de itens do conjunto ordenado.
  - ▶ Dificuldade de se utilizar um catálogo telefônico se os nomes das pessoas não estivessem listados em ordem alfabética.
- Notação utilizada nos algoritmos:

# Ordenação

- Ordenar: processo de rearranjar um conjunto de objetos em uma ordem ascendente ou descendente.
- A ordenação visa facilitar a recuperação posterior de itens do conjunto ordenado.
  - ▶ Dificuldade de se utilizar um catálogo telefônico se os nomes das pessoas não estivessem listados em ordem alfabética.
- Notação utilizada nos algoritmos:
  - ▶ Os algoritmos trabalham sobre os registros de um arquivo ou dados na memória.

# Ordenação

- Ordenar: processo de rearranjar um conjunto de objetos em uma ordem ascendente ou descendente.
- A ordenação visa facilitar a recuperação posterior de itens do conjunto ordenado.
  - ▶ Dificuldade de se utilizar um catálogo telefônico se os nomes das pessoas não estivessem listados em ordem alfabética.
- Notação utilizada nos algoritmos:
  - ▶ Os algoritmos trabalham sobre os registros de um arquivo ou dados na memória.
  - ▶ Cada registro possui uma chave utilizada para controlar a ordenação.



# Ordenação

- Ordenar: processo de rearranjar um conjunto de objetos em uma ordem ascendente ou descendente.
- A ordenação visa facilitar a recuperação posterior de itens do conjunto ordenado.
  - ▶ Dificuldade de se utilizar um catálogo telefônico se os nomes das pessoas não estivessem listados em ordem alfabética.
- Notação utilizada nos algoritmos:
  - ▶ Os algoritmos trabalham sobre os registros de um arquivo ou dados na memória.
  - ▶ Cada registro possui uma chave utilizada para controlar a ordenação.
  - ▶ Podem existir outros componentes em um registro.

# Ordenação

- Estrutura de um registro:

```
typedef long TipoChave;  
typedef struct Tipoltem{  
    TipoChave Chave;  
    /* Outros componentes */  
} Tipoltem;
```

# Ordenação

- Estrutura de um registro:

```
typedef long TipoChave;  
typedef struct Tipoltem{  
    TipoChave Chave;  
    /* Outros componentes */  
} Tipoltem;
```

- Qualquer tipo de chave sobre o qual exista uma regra de ordenação bem-definida pode ser utilizado.

# Ordenação

- Estrutura de um registro:

```
typedef long TipoChave;  
typedef struct Tipoltem{  
    TipoChave Chave;  
    /* Outros componentes */  
} Tipoltem;
```

- Qualquer tipo de chave sobre o qual exista uma regra de ordenação bem-definida pode ser utilizado.
- Um método de ordenação é estável se a ordem relativa dos itens com chaves iguais não se altera durante a ordenação.

# Ordenação

# Ordenação

- Alguns dos métodos de ordenação mais eficientes não são estáveis.

# Ordenação

- Alguns dos métodos de ordenação mais eficientes não são estáveis.
- A estabilidade pode ser forçada quando o método é não-estável.

# Ordenação

- Alguns dos métodos de ordenação mais eficientes não são estáveis.
- A estabilidade pode ser forçada quando o método é não-estável.
- Sedgewick (1988) sugere agregar um pequeno índice a cada chave antes de ordenar, ou então aumentar a chave de alguma outra forma



# Ordenação

# Ordenação

- Classificação dos métodos de ordenação:

# Ordenação

- Classificação dos métodos de ordenação:
  - ▶ Interna: arquivo a ser ordenado cabe todo na memória principal.

# Ordenação

- Classificação dos métodos de ordenação:
  - ▶ Interna: arquivo a ser ordenado cabe todo na memória principal.
  - ▶ Externa: arquivo a ser ordenado não cabe na memória principal.

# Ordenação

- Classificação dos métodos de ordenação:
  - ▶ Interna: arquivo a ser ordenado cabe todo na memória principal.
  - ▶ Externa: arquivo a ser ordenado não cabe na memória principal.
- Diferenças entre os métodos:

# Ordenação

- Classificação dos métodos de ordenação:
  - ▶ Interna: arquivo a ser ordenado cabe todo na memória principal.
  - ▶ Externa: arquivo a ser ordenado não cabe na memória principal.
- Diferenças entre os métodos:
  - ▶ Em um método de ordenação interna, qualquer registro pode ser imediatamente acessado.

# Ordenação

- Classificação dos métodos de ordenação:
  - ▶ Interna: arquivo a ser ordenado cabe todo na memória principal.
  - ▶ Externa: arquivo a ser ordenado não cabe na memória principal.
- Diferenças entre os métodos:
  - ▶ Em um método de ordenação interna, qualquer registro pode ser imediatamente acessado.
  - ▶ Em um método de ordenação externa, os registros são acessados sequencialmente ou em grandes blocos.

# Ordenação

- Classificação dos métodos de ordenação:
  - ▶ Interna: arquivo a ser ordenado cabe todo na memória principal.
  - ▶ Externa: arquivo a ser ordenado não cabe na memória principal.
- Diferenças entre os métodos:
  - ▶ Em um método de ordenação interna, qualquer registro pode ser imediatamente acessado.
  - ▶ Em um método de ordenação externa, os registros são acessados sequencialmente ou em grandes blocos.
- A maioria dos métodos de ordenação é baseada em comparações das chaves.



# Ordenação

- Classificação dos métodos de ordenação:
  - ▶ Interna: arquivo a ser ordenado cabe todo na memória principal.
  - ▶ Externa: arquivo a ser ordenado não cabe na memória principal.
- Diferenças entre os métodos:
  - ▶ Em um método de ordenação interna, qualquer registro pode ser imediatamente acessado.
  - ▶ Em um método de ordenação externa, os registros são acessados sequencialmente ou em grandes blocos.
- A maioria dos métodos de ordenação é baseada em comparações das chaves.
- Existem métodos de ordenação que utilizam o princípio da distribuição.

# Ordenação por distribuição

# Ordenação por distribuição

- Exemplo de ordenação por distribuição: considere o problema de ordenar um baralho com 52 cartas na ordem:

$\clubsuit < \diamondsuit < \heartsuit < \spadesuit$

desempatando por

$A < 2 < 3 < \dots < 10 < J < Q < K$

# Ordenação por distribuição

- Exemplo de ordenação por distribuição: considere o problema de ordenar um baralho com 52 cartas na ordem:

 <  <  < 

desempatando por

$A < 2 < 3 < \dots < 10 < J < Q < K$

- Algoritmo:
  - 1 Distribuir as cartas em treze montes: ases, dois, três, . . . , reis.
  - 2 Colete os montes na ordem contrária, de forma que o  $K$  fique em cima.
  - 3 Distribua novamente as cartas em quatro montes: paus, ouros, copas e espadas.
  - 4 Colete os montes na ordem especificada.

# Ordenação por distribuição

# Ordenação por distribuição

- Métodos como o ilustrado são também conhecidos como **ordenação digital**, **radixsort** ou **bucketsort**.

# Ordenação por distribuição

- Métodos como o ilustrado são também conhecidos como **ordenação digital**, **radixsort** ou **bucketsort**.
- O método não utiliza comparação entre chaves.

# Ordenação por distribuição

- Métodos como o ilustrado são também conhecidos como **ordenação digital**, **radixsort** ou **bucketsort**.
- O método não utiliza comparação entre chaves.
- Uma das dificuldades de implementar este método está relacionada com o problema de lidar com cada monte.



# Ordenação por distribuição

- Métodos como o ilustrado são também conhecidos como **ordenação digital**, **radixsort** ou **bucketsort**.
- O método não utiliza comparação entre chaves.
- Uma das dificuldades de implementar este método está relacionada com o problema de lidar com cada monte.
- Se para cada monte nós reservarmos uma área, então a demanda por memória extra pode tornar-se proibitiva.

# Ordenação por distribuição

- Métodos como o ilustrado são também conhecidos como **ordenação digital**, **radixsort** ou **bucketsort**.
- O método não utiliza comparação entre chaves.
- Uma das dificuldades de implementar este método está relacionada com o problema de lidar com cada monte.
- Se para cada monte nós reservarmos uma área, então a demanda por memória extra pode tornar-se proibitiva.
- Sabendo a priori a distribuição das cartas o custo para ordenar um arquivo com  $n$  elementos é da ordem de  $O(n)$ .

# Ordenação Interna

# Ordenação Interna

- Na escolha de um algoritmo de ordenação interna deve ser considerado o tempo gasto pela ordenação.

# Ordenação Interna

- Na escolha de um algoritmo de ordenação interna deve ser considerado o tempo gasto pela ordenação.
- Sendo  $n$  o número registros no arquivo, as medidas de complexidade relevantes são:

# Ordenação Interna

- Na escolha de um algoritmo de ordenação interna deve ser considerado o tempo gasto pela ordenação.
- Sendo  $n$  o número registros no arquivo, as medidas de complexidade relevantes são:
  - ▶ Número de comparações  $C(n)$  entre chaves.

# Ordenação Interna

- Na escolha de um algoritmo de ordenação interna deve ser considerado o tempo gasto pela ordenação.
- Sendo  $n$  o número registros no arquivo, as medidas de complexidade relevantes são:
  - ▶ Número de comparações  $C(n)$  entre chaves.
  - ▶ Número de movimentações  $M(n)$  de itens do arquivo.

# Ordenação Interna

- Na escolha de um algoritmo de ordenação interna deve ser considerado o tempo gasto pela ordenação.
- Sendo  $n$  o número registros no arquivo, as medidas de complexidade relevantes são:
  - ▶ Número de comparações  $C(n)$  entre chaves.
  - ▶ Número de movimentações  $M(n)$  de itens do arquivo.
- O uso econômico da memória disponível é um requisito primordial na ordenação interna.



# Ordenação Interna

- Na escolha de um algoritmo de ordenação interna deve ser considerado o tempo gasto pela ordenação.
- Sendo  $n$  o número registros no arquivo, as medidas de complexidade relevantes são:
  - ▶ Número de comparações  $C(n)$  entre chaves.
  - ▶ Número de movimentações  $M(n)$  de itens do arquivo.
- O uso econômico da memória disponível é um requisito primordial na ordenação interna.
- Métodos de ordenação **in situ** são os preferidos.

# Ordenação Interna

- Na escolha de um algoritmo de ordenação interna deve ser considerado o tempo gasto pela ordenação.
- Sendo  $n$  o número registros no arquivo, as medidas de complexidade relevantes são:
  - ▶ Número de comparações  $C(n)$  entre chaves.
  - ▶ Número de movimentações  $M(n)$  de itens do arquivo.
- O uso econômico da memória disponível é um requisito primordial na ordenação interna.
- Métodos de ordenação **in situ** são os preferidos.
- Métodos que utilizam listas encadeadas não são muito utilizados.

# Ordenação Interna

- Na escolha de um algoritmo de ordenação interna deve ser considerado o tempo gasto pela ordenação.
- Sendo  $n$  o número registros no arquivo, as medidas de complexidade relevantes são:
  - ▶ Número de comparações  $C(n)$  entre chaves.
  - ▶ Número de movimentações  $M(n)$  de itens do arquivo.
- O uso econômico da memória disponível é um requisito primordial na ordenação interna.
- Métodos de ordenação **in situ** são os preferidos.
- Métodos que utilizam listas encadeadas não são muito utilizados.
- Métodos que fazem cópias dos itens a serem ordenados possuem menor importância.

# Ordenação Interna

Ziviani classifica os métodos de ordenação interna:

# Ordenação Interna

Ziviani classifica os métodos de ordenação interna:

- Métodos simples:

# Ordenação Interna

Ziviani classifica os métodos de ordenação interna:

- Métodos simples:
  - ▶ Adequados para pequenos arquivos.

# Ordenação Interna

Ziviani classifica os métodos de ordenação interna:

- Métodos simples:
  - ▶ Adequados para pequenos arquivos.
  - ▶ Requerem  $O(n^2)$  comparações.

# Ordenação Interna

Ziviani classifica os métodos de ordenação interna:

- Métodos simples:
  - ▶ Adequados para pequenos arquivos.
  - ▶ Requerem  $O(n^2)$  comparações.
  - ▶ Produzem programas pequenos.



# Ordenação Interna

Ziviani classifica os métodos de ordenação interna:

- Métodos simples:
  - ▶ Adequados para pequenos arquivos.
  - ▶ Requerem  $O(n^2)$  comparações.
  - ▶ Produzem programas pequenos.
- Métodos eficientes:

# Ordenação Interna

Ziviani classifica os métodos de ordenação interna:

- Métodos simples:
  - ▶ Adequados para pequenos arquivos.
  - ▶ Requerem  $O(n^2)$  comparações.
  - ▶ Produzem programas pequenos.
- Métodos eficientes:
  - ▶ Adequados para arquivos maiores.

# Ordenação Interna

Ziviani classifica os métodos de ordenação interna:

- Métodos simples:
  - ▶ Adequados para pequenos arquivos.
  - ▶ Requerem  $O(n^2)$  comparações.
  - ▶ Produzem programas pequenos.
- Métodos eficientes:
  - ▶ Adequados para arquivos maiores.
  - ▶ Requerem  $O(n \log n)$  comparações.

# Ordenação Interna

Ziviani classifica os métodos de ordenação interna:

- Métodos simples:
  - ▶ Adequados para pequenos arquivos.
  - ▶ Requerem  $O(n^2)$  comparações.
  - ▶ Produzem programas pequenos.
- Métodos eficientes:
  - ▶ Adequados para arquivos maiores.
  - ▶ Requerem  $O(n \log n)$  comparações.
  - ▶ Usam menos comparações.

# Ordenação Interna

Ziviani classifica os métodos de ordenação interna:

- Métodos simples:
  - ▶ Adequados para pequenos arquivos.
  - ▶ Requerem  $O(n^2)$  comparações.
  - ▶ Produzem programas pequenos.
- Métodos eficientes:
  - ▶ Adequados para arquivos maiores.
  - ▶ Requerem  $O(n \log n)$  comparações.
  - ▶ Usam menos comparações.
  - ▶ As comparações são mais complexas nos detalhes.

# Ordenação Interna

Ziviani classifica os métodos de ordenação interna:

- Métodos simples:
  - ▶ Adequados para pequenos arquivos.
  - ▶ Requerem  $O(n^2)$  comparações.
  - ▶ Produzem programas pequenos.
- Métodos eficientes:
  - ▶ Adequados para arquivos maiores.
  - ▶ Requerem  $O(n \log n)$  comparações.
  - ▶ Usam menos comparações.
  - ▶ As comparações são mais complexas nos detalhes.
  - ▶ Métodos simples são mais eficientes para pequenos arquivos.

# Ordenação Interna

# Ordenação Interna

- Tipos de dados e variáveis utilizados nos algoritmos de ordenação interna:

```
typedef int TipoIndice;  
typedef TipoItem TipoVetor[MAXTAM + 1];  
/* MAXTAM + 1 por causa da sentinela */  
TipoVetor A;
```



# Ordenação Interna

- Tipos de dados e variáveis utilizados nos algoritmos de ordenação interna:

```
typedef int TipoIndice;  
typedef TipoItem TipoVetor[MAXTAM + 1];  
/* MAXTAM + 1 por causa da sentinela */  
TipoVetor A;
```

- O índice do vetor vai de 0 até *MaxTam*, devido às chaves sentinelas.

# Ordenação Interna

- Tipos de dados e variáveis utilizados nos algoritmos de ordenação interna:

```
typedef int TipoIndice;  
typedef TipoItem TipoVetor[MAXTAM + 1];  
/* MAXTAM + 1 por causa da sentinela */  
TipoVetor A;
```

- O índice do vetor vai de 0 até *MaxTam*, devido às chaves sentinelas.
- O vetor a ser ordenado contém chaves nas posições de 1 até *n*.

# Ordenação por Seleção

# Ordenação por Seleção

- Um dos algoritmos mais simples de ordenação.

# Ordenação por Seleção

- Um dos algoritmos mais simples de ordenação.
- Algoritmo:

# Ordenação por Seleção

- Um dos algoritmos mais simples de ordenação.
- Algoritmo:
  - ▶ Selecione o menor item do vetor.

# Ordenação por Seleção

- Um dos algoritmos mais simples de ordenação.
- Algoritmo:
  - ▶ Selecione o menor item do vetor.
  - ▶ Troque-o com o item da primeira posição do vetor.

# Ordenação por Seleção

- Um dos algoritmos mais simples de ordenação.
- Algoritmo:
  - ▶ Selecione o menor item do vetor.
  - ▶ Troque-o com o item da primeira posição do vetor.
  - ▶ Repita essas duas operações com os  $n - 1$  itens restantes, depois com os  $n - 2$  itens, até que reste apenas um elemento.



# Ordenação por Seleção

# Ordenação por Seleção

- O método é ilustrado abaixo:

	1	2	3	4	5	6
Chaves iniciais:	<i>O</i>	<i>R</i>	<i>D</i>	<i>E</i>	<i>N</i>	<i>A</i>
i = 1	<b>A</b>	<i>R</i>	<i>D</i>	<i>E</i>	<i>N</i>	<b>O</b>
i = 2	<i>A</i>	<b>D</b>	<b>R</b>	<i>E</i>	<i>N</i>	<i>O</i>
i = 3	<i>A</i>	<i>D</i>	<b>E</b>	<b>R</b>	<i>N</i>	<i>O</i>
i = 4	<i>A</i>	<i>D</i>	<i>E</i>	<b>N</b>	<b>R</b>	<i>O</i>
i = 5	<i>A</i>	<i>D</i>	<i>E</i>	<i>N</i>	<b>O</b>	<b>R</b>

# Ordenação por Seleção

- O método é ilustrado abaixo:

	1	2	3	4	5	6
Chaves iniciais:	<i>O</i>	<i>R</i>	<i>D</i>	<i>E</i>	<i>N</i>	<i>A</i>
i = 1	<b>A</b>	<i>R</i>	<i>D</i>	<i>E</i>	<i>N</i>	<b>O</b>
i = 2	<i>A</i>	<b>D</b>	<b>R</b>	<i>E</i>	<i>N</i>	<i>O</i>
i = 3	<i>A</i>	<i>D</i>	<b>E</b>	<b>R</b>	<i>N</i>	<i>O</i>
i = 4	<i>A</i>	<i>D</i>	<i>E</i>	<b>N</b>	<b>R</b>	<i>O</i>
i = 5	<i>A</i>	<i>D</i>	<i>E</i>	<i>N</i>	<b>O</b>	<b>R</b>

- As chaves em negrito sofreram uma troca entre si.

# Ordenação por Seleção

```
1 void Selecao(Tipoltem *A, Tipolndice n){
2     Tipolndice i, j, Min;
3     Tipoltem x;
4     for (i = 0; i < n - 1; i++){
5         Min = i;
6         for (j = i + 1; j < n; j++){
7             if (A[j].Chave < A[Min].Chave){
8                 Min = j;
9             }
10        }
11        x = A[Min];
12        A[Min] = A[i];
13        A[i] = x;
14    }
15 }
```

# Ordenação por Seleção

# Ordenação por Seleção

- $C(n) = \frac{n^2}{2} - \frac{n}{2}$

# Ordenação por Seleção

- $C(n) = \frac{n^2}{2} - \frac{n}{2}$
- $M(n) = 3(n - 1)$

# Ordenação por Seleção

- $C(n) = \frac{n^2}{2} - \frac{n}{2}$
- $M(n) = 3(n - 1)$
- A atribuição  $Min = j$  da linha 8 é executada em média  $n \log n$  vezes, Knuth (1973)



# Ordenação por Seleção

- $C(n) = \frac{n^2}{2} - \frac{n}{2}$
- $M(n) = 3(n - 1)$
- A atribuição  $Min = j$  da linha 8 é executada em média  $n \log n$  vezes, Knuth (1973)
- A complexidade do algoritmo de ordenação por seleção é portanto  $O(n^2)$

# Ordenação por Seleção

Vantagens:

# Ordenação por Seleção

Vantagens:

- Custo linear para o número de movimentos de registros.

# Ordenação por Seleção

## Vantagens:

- Custo linear para o número de movimentos de registros.
- É o algoritmo a ser utilizado para arquivos com registros muito grandes.

# Ordenação por Seleção

## Vantagens:

- Custo linear para o número de movimentos de registros.
- É o algoritmo a ser utilizado para arquivos com registros muito grandes.
- É muito interessante para arquivos pequenos.

## Desvantagens:

# Ordenação por Seleção

## Vantagens:

- Custo linear para o número de movimentos de registros.
- É o algoritmo a ser utilizado para arquivos com registros muito grandes.
- É muito interessante para arquivos pequenos.

## Desvantagens:

- O fato de o arquivo já estar ordenado não ajuda em nada, pois o custo continua quadrático.

# Ordenação por Seleção

## Vantagens:

- Custo linear para o número de movimentos de registros.
- É o algoritmo a ser utilizado para arquivos com registros muito grandes.
- É muito interessante para arquivos pequenos.

## Desvantagens:

- O fato de o arquivo já estar ordenado não ajuda em nada, pois o custo continua quadrático.
- O algoritmo não é **estável**.

# Ordenação por Inserção



# Ordenação por Inserção

- Método preferido dos jogadores de cartas.

# Ordenação por Inserção

- Método preferido dos jogadores de cartas.
- Algoritmo:

# Ordenação por Inserção

- Método preferido dos jogadores de cartas.
- Algoritmo:
  - ▶ Em cada passo a partir de  $i=2$  faça:

# Ordenação por Inserção

- Método preferido dos jogadores de cartas.
- Algoritmo:
  - ▶ Em cada passo a partir de  $i=2$  faça:
    - ★ Selecione o  $i$ -ésimo item da sequência fonte.

# Ordenação por Inserção

- Método preferido dos jogadores de cartas.
- Algoritmo:
  - ▶ Em cada passo a partir de  $i=2$  faça:
    - ★ Selecione o  $i$ -ésimo item da sequência fonte.
    - ★ Coloque-o no lugar apropriado na sequência destino de acordo com o critério de ordenação.

# Ordenação por Inserção

# Ordenação por Inserção

- O método é ilustrado abaixo:

	1	2	3	4	5	6
Chaves iniciais:	<b>O</b>	<i>R</i>	<i>D</i>	<i>E</i>	<i>N</i>	<i>A</i>
i = 2	<b>O</b>	<b>R</b>	<i>D</i>	<i>E</i>	<i>N</i>	<i>A</i>
i = 3	<b>D</b>	<b>O</b>	<b>R</b>	<i>E</i>	<i>N</i>	<i>A</i>
i = 4	<b>D</b>	<b>E</b>	<b>O</b>	<b>R</b>	<i>N</i>	<i>A</i>
i = 5	<b>D</b>	<b>E</b>	<b>N</b>	<b>O</b>	<b>R</b>	<i>A</i>
i = 6	<b>A</b>	<b>D</b>	<b>E</b>	<b>N</b>	<b>O</b>	<b>R</b>

- As chaves em negrito representam a sequência destino.

# Ordenação por Inserção

```
1 void Insercao(Tipoltem *A, Tipolndice n){
2     Tipolndice i, j;
3     Tipoltem x;
4     for ( i = 2; i <= n ; i++){
5         x = A[i];
6         j = i - 1;
7         A[0] = x; /* sentinela */
8         while ( x.Chave < A[j].Chave){
9             A[j+1] = A[j];
10            j--;
11        }
12        A[j+1] = x;
13    }
14 }
```



# Ordenação por Inserção

Considerações sobre o algoritmo:

# Ordenação por Inserção

Considerações sobre o algoritmo:

- O processo de ordenação pode ser terminado pelas condições:

# Ordenação por Inserção

Considerações sobre o algoritmo:

- O processo de ordenação pode ser terminado pelas condições:
  - ▶ Um item com chave menor que o item em consideração é encontrado.

# Ordenação por Inserção

Considerações sobre o algoritmo:

- O processo de ordenação pode ser terminado pelas condições:
  - ▶ Um item com chave menor que o item em consideração é encontrado.
  - ▶ O final da sequência destino é atingido à esquerda.

# Ordenação por Inserção

## Considerações sobre o algoritmo:

- O processo de ordenação pode ser terminado pelas condições:
  - ▶ Um item com chave menor que o item em consideração é encontrado.
  - ▶ O final da sequência destino é atingido à esquerda.
- Solução:
  - ▶ Utilizar um registro sentinela na posição zero do vetor.

# Ordenação por Inserção - Complexidade

# Ordenação por Inserção - Complexidade

- Seja  $C(n)$  a função que conta o número de comparações.

# Ordenação por Inserção - Complexidade

- Seja  $C(n)$  a função que conta o número de comparações.
- No laço mais interno, na  $i$ -ésima iteração, o valor de  $C_i$  é:
  - ▶ Melhor caso:  $C_i(n) = 1$
  - ▶ Pior caso:  $C_i(n) = i$
  - ▶ Caso médio:  $C_i(n) = \frac{1}{i}(1 + 2 + \dots + i) = \frac{i+1}{2}$



# Ordenação por Inserção - Complexidade

- Seja  $C(n)$  a função que conta o número de comparações.
- No laço mais interno, na  $i$ -ésima iteração, o valor de  $C_i$  é:
  - ▶ Melhor caso:  $C_i(n) = 1$
  - ▶ Pior caso:  $C_i(n) = i$
  - ▶ Caso médio:  $C_i(n) = \frac{1}{i}(1 + 2 + \dots + i) = \frac{i+1}{2}$
- Assumindo que todas as permutações de  $n$  são igualmente prováveis no caso médio, temos:
  - ▶  $C(n) = (1 + 1 + \dots + 1) = n - 1$
  - ▶ Pior caso:  $C(n) = (2 + 3 + \dots + n) = \frac{n^2}{2} + \frac{n}{2} - 1$
  - ▶ Caso médio:  $\frac{1}{2}(3 + 4 + \dots + n + 1) = \frac{n^2}{4} + \frac{3n}{4} - 1$

# Ordenação por Inserção - Complexidade

# Ordenação por Inserção - Complexidade

- Seja  $M(n)$  a função que conta o número de movimentações de registros.

# Ordenação por Inserção - Complexidade

- Seja  $M(n)$  a função que conta o número de movimentações de registros.
- O número de movimentações na  $i$ -ésima iteração é:

$$M_i(n) = C_i(n) - 1 + 3 = C_i(n) + 2$$

# Ordenação por Inserção - Complexidade

- Seja  $M(n)$  a função que conta o número de movimentações de registros.
- O número de movimentações na  $i$ -ésima iteração é:

$$M_i(n) = C_i(n) - 1 + 3 = C_i(n) + 2$$

- Logo, o número de movimentos é:

# Ordenação por Inserção - Complexidade

- Seja  $M(n)$  a função que conta o número de movimentações de registros.
- O número de movimentações na  $i$ -ésima iteração é:

$$M_i(n) = C_i(n) - 1 + 3 = C_i(n) + 2$$

- Logo, o número de movimentos é:
  - ▶ Melhor caso:  $M(n) = (3 + 3 + \dots + 3) = 3(n - 1)$

# Ordenação por Inserção - Complexidade

- Seja  $M(n)$  a função que conta o número de movimentações de registros.
- O número de movimentações na  $i$ -ésima iteração é:

$$M_i(n) = C_i(n) - 1 + 3 = C_i(n) + 2$$

- Logo, o número de movimentos é:
  - ▶ Melhor caso:  $M(n) = (3 + 3 + \dots + 3) = 3(n - 1)$
  - ▶ Pior caso:  $M(n) = (4 + 5 + \dots + n + 2) = \frac{n^2}{2} + \frac{5n}{2} - 3$

# Ordenação por Inserção - Complexidade

- Seja  $M(n)$  a função que conta o número de movimentações de registros.
- O número de movimentações na  $i$ -ésima iteração é:

$$M_i(n) = C_i(n) - 1 + 3 = C_i(n) + 2$$

- Logo, o número de movimentos é:
  - ▶ Melhor caso:  $M(n) = (3 + 3 + \dots + 3) = 3(n - 1)$
  - ▶ Pior caso:  $M(n) = (4 + 5 + \dots + n + 2) = \frac{n^2}{2} + \frac{5n}{2} - 3$
  - ▶ Caso médio:  $M(n) = \frac{1}{2}(5 + 6 + \dots + n + 3) = \frac{n^2}{4} + \frac{11n}{4} - 3$



# Ordenação por Inserção - Considerações

# Ordenação por Inserção - Considerações

- O número mínimo de comparações e movimentos ocorre quando os itens estão originalmente em ordem.

# Ordenação por Inserção - Considerações

- O número mínimo de comparações e movimentos ocorre quando os itens estão originalmente em ordem.
- O número máximo ocorre quando os itens estão originalmente na ordem reversa.

# Ordenação por Inserção - Considerações

- O número mínimo de comparações e movimentos ocorre quando os itens estão originalmente em ordem.
- O número máximo ocorre quando os itens estão originalmente na ordem reversa.
- É o método a ser utilizado quando o arquivo está “quase” ordenado.

# Ordenação por Inserção - Considerações

- O número mínimo de comparações e movimentos ocorre quando os itens estão originalmente em ordem.
- O número máximo ocorre quando os itens estão originalmente na ordem reversa.
- É o método a ser utilizado quando o arquivo está “quase” ordenado.
- É um bom método quando se deseja adicionar uns poucos itens a um arquivo ordenado, pois o custo é linear.

# Ordenação por Inserção - Considerações

- O número mínimo de comparações e movimentos ocorre quando os itens estão originalmente em ordem.
- O número máximo ocorre quando os itens estão originalmente na ordem reversa.
- É o método a ser utilizado quando o arquivo está “quase” ordenado.
- É um bom método quando se deseja adicionar uns poucos itens a um arquivo ordenado, pois o custo é linear.
- O algoritmo de ordenação por inserção é estável.

# Quicksort

# Quicksort

- Proposto por Hoare em 1960 e publicado em 1962.



# Quicksort

- Proposto por Hoare em 1960 e publicado em 1962.
- É o algoritmo de ordenação interna mais rápido que se conhece para uma ampla variedade de situações.

# Quicksort

- Proposto por Hoare em 1960 e publicado em 1962.
- É o algoritmo de ordenação interna mais rápido que se conhece para uma ampla variedade de situações.
- Provavelmente é o mais utilizado.

# Quicksort

- Proposto por Hoare em 1960 e publicado em 1962.
- É o algoritmo de ordenação interna mais rápido que se conhece para uma ampla variedade de situações.
- Provavelmente é o mais utilizado.
- A idéia básica é dividir o problema de ordenar um conjunto com  $n$  itens em dois problemas menores.

# Quicksort

- Proposto por Hoare em 1960 e publicado em 1962.
- É o algoritmo de ordenação interna mais rápido que se conhece para uma ampla variedade de situações.
- Provavelmente é o mais utilizado.
- A idéia básica é dividir o problema de ordenar um conjunto com  $n$  itens em dois problemas menores.
- Os problemas menores são ordenados independentemente.

# Quicksort

- Proposto por Hoare em 1960 e publicado em 1962.
- É o algoritmo de ordenação interna mais rápido que se conhece para uma ampla variedade de situações.
- Provavelmente é o mais utilizado.
- A idéia básica é dividir o problema de ordenar um conjunto com  $n$  itens em dois problemas menores.
- Os problemas menores são ordenados independentemente.
- Os resultados são combinados para produzir a solução final.

# Quicksort

# Quicksort

- A parte mais delicada do método é o processo de partição.

# Quicksort

- A parte mais delicada do método é o processo de partição.
- O vetor  $A[Esq \dots Dir]$  é rearranjado por meio da escolha arbitrária de um **pivô**  $x$ .



# Quicksort

- A parte mais delicada do método é o processo de partição.
- O vetor  $A[Esq \dots Dir]$  é rearranjado por meio da escolha arbitrária de um **pivô**  $x$ .
- O vetor  $A$  é particionado em duas partes:

# Quicksort

- A parte mais delicada do método é o processo de partição.
- O vetor  $A[Esq \dots Dir]$  é rearranjado por meio da escolha arbitrária de um **pivô**  $x$ .
- O vetor  $A$  é particionado em duas partes:
  - ▶ A parte esquerda com chaves menores ou iguais a  $x$ .

# Quicksort

- A parte mais delicada do método é o processo de partição.
- O vetor  $A[Esq \dots Dir]$  é rearranjado por meio da escolha arbitrária de um **pivô**  $x$ .
- O vetor  $A$  é particionado em duas partes:
  - ▶ A parte esquerda com chaves menores ou iguais a  $x$ .
  - ▶ A parte direita com chaves maiores ou iguais a  $x$ .

# Quicksort

# Quicksort

- Algoritmo para o particionamento:

# Quicksort

- Algoritmo para o particionamento:
  - 1 Escolha arbitrariamente um **pivô**  $x$ .

# Quicksort

- Algoritmo para o particionamento:
  - 1 Escolha arbitrariamente um **pivô**  $x$ .
  - 2 Percorra o vetor a partir da esquerda até que  $A[i] \geq x$ .

# Quicksort

- Algoritmo para o particionamento:
  - 1 Escolha arbitrariamente um **pivô**  $x$ .
  - 2 Percorra o vetor a partir da esquerda até que  $A[i] \geq x$ .
  - 3 Percorra o vetor a partir da direita até que  $A[j] \leq x$ .



# Quicksort

- Algoritmo para o particionamento:
  - 1 Escolha arbitrariamente um **pivô**  $x$ .
  - 2 Percorra o vetor a partir da esquerda até que  $A[i] \geq x$ .
  - 3 Percorra o vetor a partir da direita até que  $A[j] \leq x$ .
  - 4 Troque  $A[i]$  com  $A[j]$ .

# Quicksort

- Algoritmo para o particionamento:
  - 1 Escolha arbitrariamente um **pivô**  $x$ .
  - 2 Percorra o vetor a partir da esquerda até que  $A[i] \geq x$ .
  - 3 Percorra o vetor a partir da direita até que  $A[j] \leq x$ .
  - 4 Troque  $A[i]$  com  $A[j]$ .
  - 5 Continue este processo até os apontadores  $i$  e  $j$  se cruzarem.

# Quicksort

- Algoritmo para o particionamento:
  - 1 Escolha arbitrariamente um **pivô**  $x$ .
  - 2 Percorra o vetor a partir da esquerda até que  $A[i] \geq x$ .
  - 3 Percorra o vetor a partir da direita até que  $A[j] \leq x$ .
  - 4 Troque  $A[i]$  com  $A[j]$ .
  - 5 Continue este processo até os apontadores  $i$  e  $j$  se cruzarem.
- Ao final, o vetor  $A[Esq..Dir]$  está particionado de tal forma que:

# Quicksort

- Algoritmo para o particionamento:
  - 1 Escolha arbitrariamente um **pivô**  $x$ .
  - 2 Percorra o vetor a partir da esquerda até que  $A[i] \geq x$ .
  - 3 Percorra o vetor a partir da direita até que  $A[j] \leq x$ .
  - 4 Troque  $A[i]$  com  $A[j]$ .
  - 5 Continue este processo até os apontadores  $i$  e  $j$  se cruzarem.
- Ao final, o vetor  $A[Esq..Dir]$  está particionado de tal forma que:
  - ▶ Os itens em  $A[Esq], A[Esq + 1], \dots, A[j]$  são menores ou iguais a  $x$ .

# Quicksort

- Algoritmo para o particionamento:
  - 1 Escolha arbitrariamente um **pivô**  $x$ .
  - 2 Percorra o vetor a partir da esquerda até que  $A[i] \geq x$ .
  - 3 Percorra o vetor a partir da direita até que  $A[j] \leq x$ .
  - 4 Troque  $A[i]$  com  $A[j]$ .
  - 5 Continue este processo até os apontadores  $i$  e  $j$  se cruzarem.
- Ao final, o vetor  $A[Esq..Dir]$  está particionado de tal forma que:
  - ▶ Os itens em  $A[Esq], A[Esq + 1], \dots, A[j]$  são menores ou iguais a  $x$ .
  - ▶ Os itens em  $A[i], A[i + 1], \dots, A[Dir]$  são maiores ou iguais a  $x$ .

# Quicksort

# Quicksort

- Ilustração do processo de partição:

1	2	3	4	5	6
<i>O</i>	<i>R</i>	<b><i>D</i></b>	<i>E</i>	<i>N</i>	<i>A</i>
<i>A</i>	<i>R</i>	<b><i>D</i></b>	<i>E</i>	<i>N</i>	<i>O</i>
<i>A</i>	<b><i>D</i></b>	<i>R</i>	<i>E</i>	<i>N</i>	<i>O</i>

# Quicksort

- Ilustração do processo de partição:

1	2	3	4	5	6
<i>O</i>	<i>R</i>	<b><i>D</i></b>	<i>E</i>	<i>N</i>	<i>A</i>
<i>A</i>	<i>R</i>	<b><i>D</i></b>	<i>E</i>	<i>N</i>	<i>O</i>
<i>A</i>	<b><i>D</i></b>	<i>R</i>	<i>E</i>	<i>N</i>	<i>O</i>

- O pivô  $x$  é escolhido como sendo  $A[(i + j) \text{div} 2]$ .



# Quicksort

- Ilustração do processo de partição:

1	2	3	4	5	6
<i>O</i>	<i>R</i>	<b><i>D</i></b>	<i>E</i>	<i>N</i>	<i>A</i>
<i>A</i>	<i>R</i>	<b><i>D</i></b>	<i>E</i>	<i>N</i>	<i>O</i>
<i>A</i>	<b><i>D</i></b>	<i>R</i>	<i>E</i>	<i>N</i>	<i>O</i>

- O pivô  $x$  é escolhido como sendo  $A[(i + j) \text{div} 2]$ .
- Como inicialmente  $i = 1$  e  $j = 6$ , então  $x = A[3] = D$ .

# Quicksort

- Ilustração do processo de partição:

1	2	3	4	5	6
<i>O</i>	<i>R</i>	<b><i>D</i></b>	<i>E</i>	<i>N</i>	<i>A</i>
<i>A</i>	<i>R</i>	<b><i>D</i></b>	<i>E</i>	<i>N</i>	<i>O</i>
<i>A</i>	<b><i>D</i></b>	<i>R</i>	<i>E</i>	<i>N</i>	<i>O</i>

- O pivô  $x$  é escolhido como sendo  $A[(i + j) \text{div} 2]$ .
- Como inicialmente  $i = 1$  e  $j = 6$ , então  $x = A[3] = D$ .
- Ao final do processo de partição  $i$  e  $j$  se cruzam em  $i = 3$  e  $j = 2$ .

# Partição

```
1 void Particao (Tipolndice Esq, Tipolndice Dir,  
2 Tipolndice *i, Tipolndice *j, Tipoltem *A){  
3     Tipoltem x, w;  
4     *i = Esq;  
5     *j = Dir;  
6     x = A[( *i + *j ) / 2 ] ; /* obtem o pivo x */  
7     do{  
8         while (x.Chave > A[*i].Chave) (*i)++;  
9         while (x.Chave < A[*j].Chave) (*j)--;  
10        if (*i <= *j){  
11            w = A[*i]; A[*i] = A[*j]; A[*j] = w;  
12            (*i)++; (*j)--;  
13        }  
14    } while (*i <= *j);  
15 }
```

# Partição

# Partição

- O anel interno do procedimento Particao é extremamente simples.

# Partição

- O anel interno do procedimento Particao é extremamente simples.
- Razão pela qual o algoritmo Quicksort é tão rápido.

# Quicksort

```
1 void Ordena(TipoIndice Esq, TipoIndice Dir,
2             TipoItem *A){
3     TipoIndice i, j;
4     Particao (Esq, Dir, &i, &j, A);
5     if (Esq < j) Ordena(Esq, j, A);
6     if (i < Dir) Ordena(i, Dir, A);
7 }
8
9 void QuickSort(TipoItem *A, TipoIndice n){
10     Ordena(1, n, A);
11 }
```

# Quicksort: Análise



# Quicksort: Análise

- Seja  $C(n)$  a função que conta o número de comparações.

# Quicksort: Análise

- Seja  $C(n)$  a função que conta o número de comparações.
- Pior caso:

$$C(n) = O(n^2)$$

# Quicksort: Análise

- Seja  $C(n)$  a função que conta o número de comparações.
- Pior caso:

$$C(n) = O(n^2)$$

- O pior caso ocorre quando, sistematicamente, o pivô é escolhido como sendo um dos extremos de um arquivo já ordenado.

# Quicksort: Análise

- Seja  $C(n)$  a função que conta o número de comparações.
- Pior caso:

$$C(n) = O(n^2)$$

- O pior caso ocorre quando, sistematicamente, o pivô é escolhido como sendo um dos extremos de um arquivo já ordenado.
- Isto faz com que o procedimento Ordena seja chamado recursivamente  $n$  vezes, eliminando apenas um item em cada chamada.

## Quicksort: Análise

- Seja  $C(n)$  a função que conta o número de comparações.
- Pior caso:

$$C(n) = O(n^2)$$

- O pior caso ocorre quando, sistematicamente, o pivô é escolhido como sendo um dos extremos de um arquivo já ordenado.
- Isto faz com que o procedimento Ordena seja chamado recursivamente  $n$  vezes, eliminando apenas um item em cada chamada.
- O pior caso pode ser evitado empregando pequenas modificações no algoritmo.

## Quicksort: Análise

- Seja  $C(n)$  a função que conta o número de comparações.
- Pior caso:

$$C(n) = O(n^2)$$

- O pior caso ocorre quando, sistematicamente, o pivô é escolhido como sendo um dos extremos de um arquivo já ordenado.
- Isto faz com que o procedimento Ordena seja chamado recursivamente  $n$  vezes, eliminando apenas um item em cada chamada.
- O pior caso pode ser evitado empregando pequenas modificações no algoritmo.
- Para isso basta escolher três itens quaisquer do vetor e usar a **mediana dos três** como pivô.

# Quicksort: Análise

# Quicksort: Análise

- Melhor caso:

$$C(n) = 2C(n/2) + n = n \log n - n + 1$$



# Quicksort: Análise

- Melhor caso:

$$C(n) = 2C(n/2) + n = n \log n - n + 1$$

- Esta situação ocorre quando cada partição divide o arquivo em duas partes iguais.

# Quicksort: Análise

- Melhor caso:

$$C(n) = 2C(n/2) + n = n \log n - n + 1$$

- Esta situação ocorre quando cada partição divide o arquivo em duas partes iguais.
- Caso médio de acordo com Sedgewick e Flajolet (1996, p. 17):

$$C(n) \approx 1,386n \log n - 0,846n$$

- Isso significa que em média o tempo de execução do Quicksort é  $O(n \log n)$ .

# Quicksort

# Quicksort

- Vantagens:

# Quicksort

- Vantagens:
  - ▶ É extremamente eficiente para ordenar arquivos de dados.

# Quicksort

- Vantagens:
  - ▶ É extremamente eficiente para ordenar arquivos de dados.
  - ▶ Necessita de apenas uma pequena pilha como memória auxiliar.

# Quicksort

- Vantagens:

- ▶ É extremamente eficiente para ordenar arquivos de dados.
- ▶ Necessita de apenas uma pequena pilha como memória auxiliar.
- ▶ Requer cerca de  $n \log n$  comparações em média para ordenar  $n$  itens.

# Quicksort

- Vantagens:
  - ▶ É extremamente eficiente para ordenar arquivos de dados.
  - ▶ Necessita de apenas uma pequena pilha como memória auxiliar.
  - ▶ Requer cerca de  $n \log n$  comparações em média para ordenar  $n$  itens.
- Desvantagens:



# Quicksort

- Vantagens:

- ▶ É extremamente eficiente para ordenar arquivos de dados.
- ▶ Necessita de apenas uma pequena pilha como memória auxiliar.
- ▶ Requer cerca de  $n \log n$  comparações em média para ordenar  $n$  itens.

- Desvantagens:

- ▶ Tem um pior caso  $O(n^2)$  comparações.

# Quicksort

- Vantagens:

- ▶ É extremamente eficiente para ordenar arquivos de dados.
- ▶ Necessita de apenas uma pequena pilha como memória auxiliar.
- ▶ Requer cerca de  $n \log n$  comparações em média para ordenar  $n$  itens.

- Desvantagens:

- ▶ Tem um pior caso  $O(n^2)$  comparações.
- ▶ Sua implementação é muito delicada e difícil:

# Quicksort

- Vantagens:

- ▶ É extremamente eficiente para ordenar arquivos de dados.
- ▶ Necessita de apenas uma pequena pilha como memória auxiliar.
- ▶ Requer cerca de  $n \log n$  comparações em média para ordenar  $n$  itens.

- Desvantagens:

- ▶ Tem um pior caso  $O(n^2)$  comparações.
- ▶ Sua implementação é muito delicada e difícil:
  - ★ Um pequeno engano pode levar a efeitos inesperados para algumas entradas de dados.

# Quicksort

- Vantagens:

- ▶ É extremamente eficiente para ordenar arquivos de dados.
- ▶ Necessita de apenas uma pequena pilha como memória auxiliar.
- ▶ Requer cerca de  $n \log n$  comparações em média para ordenar  $n$  itens.

- Desvantagens:

- ▶ Tem um pior caso  $O(n^2)$  comparações.
- ▶ Sua implementação é muito delicada e difícil:
  - ★ Um pequeno engano pode levar a efeitos inesperados para algumas entradas de dados.
- ▶ O método não é estável.