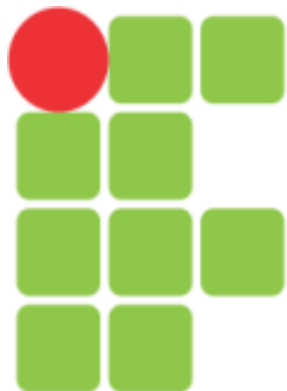




INSTITUTO FEDERAL
CEARÁ



INSTITUTO FEDERAL
CEARÁ



Ernani Andrade Leite

ernani@ifce.edu.br

Ordenação

Ordenar corresponde ao processo de rearranjar um conjunto de objetos em uma ordem ascendente ou descendente.

O objetivo principal da ordenação é facilitar a recuperação posterior de itens do conjunto ordenado.

Os algoritmos de ordenação trabalham sobre os registros de um arquivo e somente uma parte, ou campo, do registro, chamada **chave**, é utilizada para controlar a ordenação. Além da chave podem existir outros componentes em um registro, os quais não têm influência no processo de ordenar, a não ser pelo fato de que permanecem com a mesma chave.

Terminologia básica

A ordenação pode ser por : **contiguidade física, vetor indireto de ordenação e encadeamento**

Os algoritmos podem operar sobre o elemento ou sobre uma tabela de ponteiros

- registro é muito grande: ordenação **indireta**
- os registros não são rearrumados e sim os índices

Contiguidade física:

Existe uma forte característica física no processo de ordenação,

Os elementos de um registro são movimentados de sua posição física original para sua nova posição

Contigüidade Física de Tabelas

- Dois registros consecutivos R_i e $R_{i+1} \rightarrow Ch_i < Ch_{i+1}$

	campo1	dados satélites
1	10	
2	19	
3	13	
4	12	
5	7	

Antes da classificação

	campo1	dados satélites
1	7	
2	10	
3	12	
4	13	
5	19	

Após a classificação

Acesso aos registros ordenados através de pesquisa **seqüencial** e **binária**!

Vetor indireto de ordenação:

Não há movimentação das entradas das posições originais.

A classificação é realizada com o apoio de um arquivo-índice.

É esse arquivo que contém o endereço para os registros reais, os quais são mantidos classificados

Vetor Indireto de Ordenação (VIO)

	campo1	dados satélites
1	10	
2	19	
3	13	
4	12	
5	7	

	campo1	
1	7	5
2	10	1
3	12	4
4	13	3
5	19	2

Acesso aos registros ordenados através de pesquisa **seqüencial**, **binária** e **direta** (mas sempre por via indireta)

Encadeamento:

Também não há a movimentação dos registros das posições originais.

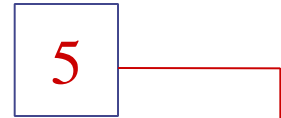
A ordem desejada é conseguida pelo uso de uma lista ligada.

Coloca-se um campo a mais na tabela para permitir que o próximo registro seja identificado.

É necessário utilizar um ponteiro para o primeiro elemento da lista ligada

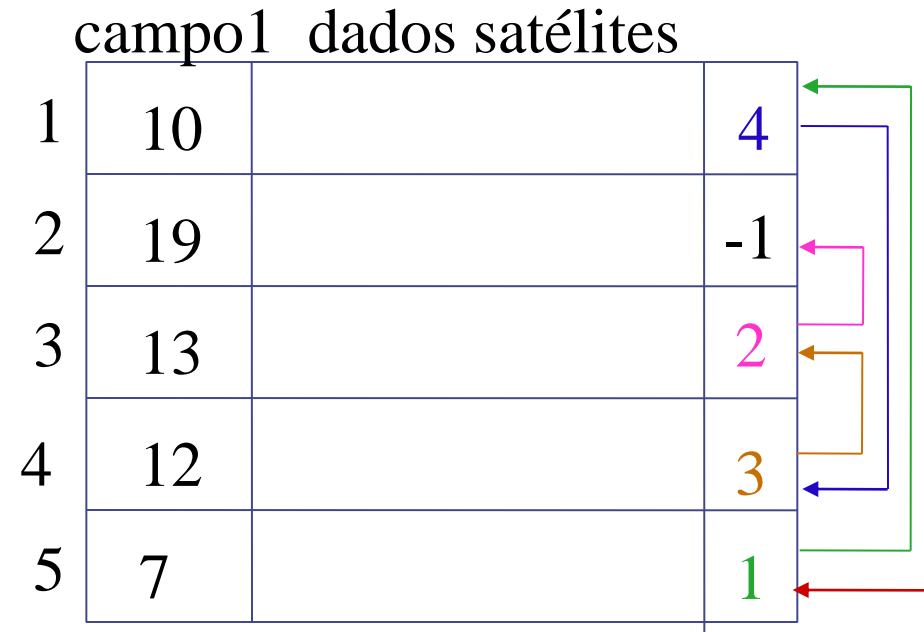
Encadeamento de Registros

cabeça da lista



	campo1	dados satélites
1	10	
2	19	
3	13	
4	12	
5	7	

Antes da classificação



Após a classificação

Acesso aos registros ordenados através de pesquisa **seqüencial**!

Estrutura de um item do arquivo

A estrutura de dados registro é a indicada para representar os itens componentes de um conjunto, ou arquivo, a ser ordenado:

```
struct TipoItem {  
    int Chave;  
  
    // outros componentes do registro  
};
```

A escolha do tipo para a chave é arbitrária. Qualquer tipo sobre o qual exista uma regra de ordenação bem definida pode ser utilizado.

As ordens numérica (*int*) e alfabética (*string*) são as usuais.

Tipos de Ordenação

Os métodos de ordenação são classificados em dois grandes grupos.

Se o arquivo a ser ordenado cabe todo na memória principal, então o método de ordenação é chamado de **ordenação interna**. Neste caso o número de registros a ser ordenado é pequeno o bastante para caber em um array do C.

Se o arquivo a ser ordenado não cabe na memória principal e, por isso, tem que ser armazenado em fita ou disco, então o método de ordenação é chamado de **ordenação externa**.

A principal diferença entre os dois métodos é que, em um método de ordenação interna, qualquer registro pode ser imediatamente acessado, enquanto, em um método de ordenação externa, os registros são acessados sequencialmente ou em grandes blocos.

Ordenação Interna

Os algoritmos de ordenação constituem bons exemplos de como resolver problemas utilizando computadores. As técnicas de ordenação permitem apresentar um conjunto amplo de algoritmos distintos para resolver uma mesma tarefa. Dependendo da aplicação, cada algoritmo considerado possui uma vantagem particular sobre os outros.

O aspecto predominante na **escolha de um algoritmo** de ordenação é o tempo gasto para ordenar um arquivo.

Para algoritmos de ordenação interna as medidas de complexidade relevantes contam o número de comparações entre chaves e o número de movimentações (ou trocas) de itens do arquivo. A quantidade extra de memória auxiliar utilizada pelo algoritmo é também um aspecto importante.

Tipos utilizados

Na implementação dos algoritmos de ordenação interna, serão utilizados:

1. a constante "n" que representa a quantidade de itens do arquivo que serão ordenados.
2. o tipo de dado "TipoVetor", representa o conjunto de elementos do TipoItem, apresentado anteriormente, a ser ordenado. Observe que o índice do vetor é um subintervalo de 0 até n , para poder armazenar chaves especiais na posição 0 chamadas de **sentinelas**.
3. a variável "A", indica o conjunto a ser ordenado propriamente dito.

```
#define n 10 // tamanho do vetor

typedef struct TipoItem TipoVetor[n+1];

struct TipoVetor A;
```

Procedimento Auxiliar

Os algoritmos de ordenação que serão descritos a seguir, na sua grande maioria, fazem uso do procedimento "Swap" para trocar dois elementos de posição.

```
void Swap (TipoItem *x, TipoItem *y) {  
    TipoItem aux;  
    aux = *x;  
    *x = *y;  
    *y = aux;  
}
```

Troca o valor das variáveis “x” e “y” entre si, usando uma posição de memória auxiliar (variável “aux”).

1. Ordenação por Seleção (Selection Sort)

Um dos algoritmos mais simples de ordenação. Neste método a classificação é efetivada por seleção sucessiva do menor valor de chave contido no vetor "A". A cada passo é feita uma varredura do segmento do vetor com os elementos ainda não selecionados e determinado aquele elemento de menor valor, o qual é colocado, por troca, em sua posição definitiva no vetor classificado e o processo repetido para o segmento que contém os elementos ainda não selecionados.

Na Prática

- 1º passo:** procura a posição do **menor** elemento do vetor e em seguida realiza a troca deste elemento com aquele que está na primeira posição;
- 2º passo:** procura o **menor** elemento no subvetor que começa a partir da segunda posição e troca com o segundo elemento e assim por diante;

O método deverá repetir sucessivamente o processo de seleção durante "**n-1**" passos.

```
// Ordenação por seleção  
// (Selection Sort)
```

```
void Selecao(TipoVetor A) {  
    int i, j, menor;  
    for (i=1; i<=(n-1); i++) {
```

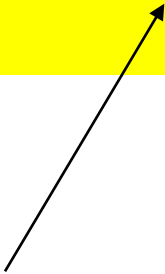
```
        menor = i;  
        for (j=(i+1); j<=n; j++)  
            if (A[j].Chave < A[menor].Chave)  
                menor = j;
```

```
        Swap (&A[i], &A[menor]);
```

```
    }
```

```
}
```

seleciona a posição do
menor elemento



// Ordenação por seleção
// (Selection Sort) - EXEMPLO

A = [5|3|2|7|1]

Selecao(**TipoVetor** A)

int i, j, **menor**;

PARA i <- 1 ATÉ 4 **FAÇA**

menor <- i;

PARA j <- i+1 ATÉ 5 **FAÇA**

SE (A[j] < A[**menor**])

menor <- j;

FIM-SE;

FIM-PARA;

Troca (A[i] <-> A[**menor**]) ;

FIM-PARA;

2. Ordenação por Troca ou Método da Bolha (Bubble Sort)

A filosofia básica deste método consiste em:

1. Varrer o vetor, comparando os elementos vizinhos entre si: if ($A[j] > A[j+1]$).
2. Caso estejam fora de ordem, os mesmos trocam de posição entre si (Swap).
3. Procede-se assim até o final do vetor. Na primeira varredura verifica-se que o último elemento do vetor já está no seu devido lugar (no caso de ordenação crescente, ele é o maior de todos). A segunda varredura é análoga à primeira e vai até o penúltimo elemento.
4. Este processo é repetido até que seja feito um número de varreduras igual ao número de elementos a serem ordenados menos um. Ao final do processo o vetor está classificado segundo o critério escolhido.

// Ordenação por Troca ou
// Método da Bolha (Bubble Sort)

void Bolha (TipoVetor A) {

int i, j;

for (i=1; i<=(n-1); i++) 4

for (j=1; j<=(n-i); j++) 3

 1 if (A[j].Chave > A[j+1].Chave)
 Swap (&A[j], &A[j+1]); 2

}

// Ordenação por Troca ou

// Método da Bolha (Bubble Sort) - EXEMPLO

A = [5|3|2|7|1]

Bolha (**TipoVetor** A)

int i, troca, **aux**;

troca ← 1; n ← 1;

ENQTO° (n ≤ 5) **E** (troca = 1) FAÇA

troca ← 0;

PARA i ← 1 ATÉ 4 FAÇA

SE (A[i] < A[i+1])

troca ← 1;

aux ← A[i];

A[i] ← A[i+1];

A[i+1] ← **aux**

FIM-SE;

FIM-PARA;

n ← n + 1;

FIM-ENQUANTO;

3. Ordenação por Inserção (Insertion Sort)

Em cada passo, a partir de $i=2$, o i -ésimo item da sequência fonte é apanhado e transferido para a sequência destino, sendo inserido no seu lugar apropriado. A colocação do item no seu lugar apropriado na sequência destino é realizada movendo-se itens com chaves maiores para a direita ($A[j+1]:=A[j];$) e então inserindo o item na posição deixada vazia ($A[j+1]:=x;$). Neste processo de alternar comparações e movimentos de registros existem duas condições distintas que podem causar a terminação do processo:

1. um item com chave menor ou igual que o item em consideração é encontrado;
2. o final da sequência destino é atingido à esquerda (sentinela).

A melhor solução para a situação de um vetor com duas condições de terminação é a utilização de um registro **sentinela**: na posição zero do vetor colocamos o próprio registro em consideração. Para tal o índice do vetor tem que ser estendido para $n+1$, ou seja, de 0 até n .

```
// Ordenação por inserção  
// (Insertion Sort)
```

```
Insercao(TipoVetor A, n) {  
    TipoItem x;  
    int i, j;  
    for (j=2; j<=n; j++) {  
        x = A[j];  
        i = j - 1;
```

```
        while (i>0) && (A[i]> x) {  
            A[i+1] = A[i];  
            i = i - 1;  
        }  
        A[i+1] = x;
```

```
    }
```

```
}
```

insere o i-ésimo item (x) no seu lugar
apropriado na sequência destino²²

4. QuickSort

QuickSort é o algoritmo de ordenação interna mais rápido que se conhece para uma ampla variedade de situações, sendo provavelmente mais utilizado do que qualquer outro algoritmo. O algoritmo foi inventado por Hoare em 1960, quando visitava a Universidade de Moscou como estudante, e foi publicado mais tarde por Hoare (1962), após uma série de refinamentos.

A idéia básica é a de partir o problema de ordenar um conjunto com n itens em dois problemas menores. A seguir, os problemas menores são ordenados independentemente e depois os resultados são combinados para produzir a solução do problema maior.

A parte mais delicada deste método é relativa ao procedimento partição, o qual tem que reorganizar o vetor $A[Esq..Dir]$ através da escolha arbitrária de um item x do vetor chamado **pivô**, de tal forma que ao final o vetor A está particionado em uma parte esquerda com chaves menores ou iguais a x e uma parte direita com chaves maiores ou iguais a x .

Procedimento “Partição”:

1. escolha arbitrariamente um item do vetor e coloque-o em x ;
2. percorra o vetor a partir da esquerda até que um item $A[i] \geq x$ é encontrado; da mesma forma percorra o vetor a partir da direita até que um item $A[j] \leq x$ é encontrado;
3. como os dois itens $A[i]$ e $A[j]$ estão fora de lugar no vetor final então troque-os de lugar;
4. continue este processo até que os apontadores i e j se cruzem em algum ponto do vetor.

Ao final o vetor $A[Esq..Dir]$ está particionado de tal forma que:

- a. os itens $A[Esq]$, $A[Esq + 1]$, ..., $A[j]$ são menores ou iguais a x ;
- b. os itens $A[i]$, $A[i + 1]$, ..., $A[Dir]$ são maiores ou iguais a x ;


```

void Particao(TipoVetor A, int Esq, int Dir, int *i, int *j) {
    TipoItem x;
    *i = Esq;
    *j = Dir;
    x = A[(*i + *j) / 2]; // Obtém o Pivô 1
    do {
        while (A[*i].Chave < x.Chave) 2
            *i = *i + 1;

        while (x.Chave < A[*j].Chave) 3
            *j = *j - 1;

        if (*i <= *j) {
            Swap(&A[*i], &A[*j]);
            *i = *i + 1;
            *j = *j - 1;
        }
    } while (*i <= *j); 4
}

```

Nota: o anel interno do procedimento partição consiste apenas em incrementar um apontador e comparar um item do vetor contra um valor fixo em x . Este anel é extremamente simples, razão pela qual o algoritmo Quicksort é tão rápido²⁵

Particionando o Vetor

O método é ilustrado usando como chaves os caracteres da palavra "ORDENA", ou seja, $n = 6$.

O item x (pivô) é escolhido como sendo: $A[(i + j) \text{ div } 2]$. Como inicialmente $i = 1$ e $j = 6$, então $x = A[3] = D$, o qual aparece em negrito na primeira linha. A varredura a partir da posição 1 pára no item O e a varredura a partir da posição 6 pára no item A, sendo os dois itens trocados, como mostrado na segunda linha. A seguir a varredura a partir da posição 2 para no item R e a varredura a partir da posição 5 pára no item D, e então os dois itens são trocados, como mostrado na terceira linha. Neste momento i e j se cruzam ($i = 3$ e $j = 2$), o que encerra o processo de "partição".

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

```
void Ordena (TipoVetor A, int Esq, int Dir) {  
    int i, j;  
    Particao (A, Esq, Dir, &i, &j);  
  
    if (Esq < j)  
        Ordena (A, Esq, j);  
  
    if (i < Dir)  
        Ordena (A, i, Dir);  
}
```

```
void QuickSort (TipoVetor A) {  
    Ordena (A, 1, n);  
}
```

Notas: observar que o procedimento "Ordena" é recursivo.

Procedimento “QuickSort”

Após obter os dois pedaços do vetor através do procedimento partição, cada pedaço é ordenado recursivamente.

A Figura a seguir ilustra o que acontece com o vetor exemplo em cada chamada recursiva do procedimento ordena. Cada linha mostra o resultado do procedimento partição, onde o **pivô** é mostrado em negrito.

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

Comparação entre algoritmos de ordenação:

Considerando-se entradas com vetores com tamanhos 500, 5.000, 10.000 e 30.000, temos:

- a) Se o tamanho do conjunto de dados for ≤ 50 , recomenda-se o método de *inserção*;
- b) Se o tamanho do conjunto de dados for > 5.000 , recomenda-se o método de *quick sort*. Entretanto, esse método necessita de memória adicional por ser recursivo. Recomenda-se evitar chamadas recursivas para pequenos intervalos. Deve-se, então, testar antes da recursividade (se $N \leq 50$, *inserção*)
- c) Outro critério a ser considerado é se o vetor de entrada está ordenado (crescente ou decrescente) ou em ordem aleatória. O método *inserção* é mais vantajoso para entradas em ordem crescente. O *quick sort* é mais vantajoso para ordem aleatória e decrescente.

Referências

- Projeto de Algoritmos: Com Implementações em Pascal e C.
 - Nivio Ziviani.
 - 4^a ed.- São Paulo: Pioneira, 1999.
- Estrutura de Dados.
 - Paulo Veloso, Clesio dos Santos, Paulo Azeredo e Antonio Furtado.
 - 13^a ed.- Rio de Janeiro: Editora Campus, 1991.