

Estruturas de Dados

Módulo 16 - Ordenação



Referências

Waldemar Celes, Renato Cerqueira, José Lucas Rangel,
Introdução a Estruturas de Dados, Editora Campus
(2004)

Capítulo 16 – Ordenação

Tópicos

- Introdução
- Ordenação bolha (*bubble sort*)
- Ordenação rápida (*quick sort*)

Introdução

- Ordenação de vetores:
 - entrada: vetor com os elementos a serem ordenados
 - saída: mesmo vetor com elementos na ordem especificada
 - ordenação:
 - pode ser aplicada a qualquer dado com ordem bem definida
 - vetores com dados complexos (structs)
 - chave da ordenação escolhida entre os campos
 - elemento do vetor contém apenas um ponteiro para os dados
 - troca da ordem entre dois elementos = troca de ponteiros

Ordenação Bolha

- Ordenação bolha:
 - processo básico:
 - quando **dois elementos** estão fora de ordem, **troque-os de posição** até que o **i-ésimo elemento de maior valor** do vetor seja levado para as posições finais do vetor
 - continue o processo até que todo o vetor esteja ordenado

Maior elemento

v0	4	2	5	1
v1	2	4	5	1
v2	2	4	5	1
v3	2	4	1	5
	0	1	2	3

2º maior elemento

v4	2	4	1	5
v5	2	4	1	5
	2	1	4	5
	0	1	2	3

3º maior elemento

v6	2	1	4	5
	1	2	4	5
	0	1	2	3

Ordenação Bolha

25	48	37	12	57	86	33	92	25x48
25	48	37	12	57	86	33	92	48x37 troca
25	37	48	12	57	86	33	92	48x12 troca
25	37	12	48	57	86	33	92	48x57
25	37	12	48	57	86	33	92	57x86
25	37	12	48	57	86	33	92	86x33 troca
25	37	12	48	57	33	86	92	86x92
25	37	12	48	57	33	86	<u>92</u>	final da primeira passada

o maior elemento, 92, já está na sua posição final

Ordenação Bolha

25	37	12	48	57	33	86	<u>92</u>	25x37
25	37	12	48	57	33	86	<u>92</u>	37x12 troca
25	12	37	48	57	33	86	<u>92</u>	37x48
25	12	37	48	57	33	86	<u>92</u>	48x57
25	12	37	48	57	33	86	<u>92</u>	57x33 troca
25	12	37	48	33	57	86	<u>92</u>	57x86
25	12	37	48	33	57	<u>86</u>	<u>92</u>	final da segunda passada

o segundo maior elemento, 86, já está na sua posição final

Ordenação Bolha

25	12	37	48	33	57	<u>86</u>	<u>92</u>	25x12 troca
12	25	37	48	33	57	<u>86</u>	<u>92</u>	25x37
12	25	37	48	33	57	<u>86</u>	<u>92</u>	37x48
12	25	37	48	33	57	<u>86</u>	<u>92</u>	48x33 troca
12	25	37	33	48	57	<u>86</u>	<u>92</u>	48x57
12	25	37	33	48	<u>57</u>	<u>86</u>	<u>92</u>	final da terceira passada

Idem para 57.

12	25	37	33	48	<u>57</u>	<u>86</u>	<u>92</u>	12x25
12	25	37	33	48	<u>57</u>	<u>86</u>	<u>92</u>	25x37
12	25	37	33	48	<u>57</u>	<u>86</u>	<u>92</u>	37x33 troca
12	25	33	37	48	<u>57</u>	<u>86</u>	<u>92</u>	37x48
12	25	33	37	<u>48</u>	<u>57</u>	<u>86</u>	<u>92</u>	final da quarta passada

Idem para 48.

12	25	33	37	<u>48</u>	57	86	92	<i>12x25</i>
12	25	33	37	<u>48</u>	57	86	92	<i>25x33</i>
12	25	33	37	<u>48</u>	57	86	92	<i>33x37</i>
12	25	33	<u>37</u>	<u>48</u>	57	86	92	<i>final da quinta passada</i>

Idem para 37.

12	25	33	<u>37</u>	48	57	86	92	<i>12x25</i>
12	25	33	<u>37</u>	48	57	86	92	<i>25x33</i>
12	25	<u>33</u>	<u>37</u>	48	57	86	92	<i>final da sexta passada</i>

Idem para 33.

12	25	<u>33</u>	<u>37</u>	48	57	86	92	<i>12x25</i>
12	<u>25</u>	<u>33</u>	<u>37</u>	48	57	86	92	<i>final da sétima passada</i>

Idem para 25 e, conseqüentemente, 12.

12	25	33	37	48	57	86	92	<i>final da ordenação</i>
----	----	----	----	----	----	----	----	---------------------------

Ordenação Bolha

- Implementação Iterativa(I):

```
/* Ordenação bolha */
void bolha (int n, int* v)
{
    int i,j;
    for (i=n-1; i>=1; i--)
        for (j=0; j<i; j++)
            if (v[j]>v[j+1]) {
                int temp = v[j];    /* troca */
                v[j] = v[j+1];
                v[j+1] = temp;
            }
}
```

maior elemento
(n=4; i=n-1=3)

4	2	5	1
2	4	5	1
2	4	5	1
2	4	1	5

2º maior elemento
(i=n-2=2)

2	4	1	5
2	4	1	5
2	1	4	5

3º maior elemento
(i=n-3=1)

2	1	4	5
1	2	4	5
0	1	2	3

Ordenação Bolha

- Implementação Iterativa (II):

```
/* Ordenação bolha (2a. versão) */  
void bolha (int n, int* v)  
{ int i, j;  
  for (i=n-1; i>0; i--) {  
    int troca = 0;  
    for (j=0; j<i; j++)  
      if (v[j]>v[j+1]) {  
        int temp = v[j]; /* troca */  
        v[j] = v[j+1];  
        v[j+1] = temp;  
        troca = 1;  
      }  
    if (troca == 0) return; /* não houve troca */  
  }  
}
```

pára quando há
uma passagem inteira
sem trocas

Ordenação Bolha

- Esforço computacional:
 - esforço computacional \cong número de comparações
 \cong número máximo de trocas
 - primeira passada: $n-1$ comparações
 - segunda passada: $n-2$ comparações
 - terceira passada: $n-3$ comparações
 - ...
 - tempo total gasto pelo algoritmo:
 - T proporcional a $(n-1) + (n-2) + \dots + 2 + 1 = (n-1+1) / 2 = n^2 / 2$
 - algoritmo de ordem quadrática: $O(n^2)$

Ordenação Bolha

- Implementação recursiva:

```
/* Ordenação bolha recursiva */  
void bolha_rec (int n, int* v)  
{ int j;  
  int troca = 0;  
  for (j=0; j<n-1; j++)  
    if (v[j]>v[j+1]) {  
      int temp = v[j]; /* troca */  
      v[j] = v[j+1];  
      v[j+1] = temp;  
      troca = 1;  
    }  
  if (troca != 0) /* houve troca */  
    bolha_rec(n-1,v);  
}
```

maior elemento
bolha_rec(4,v);

4	2	5	1
2	4	5	1
2	4	5	1
2	4	1	5

2º maior elemento
bolha_rec(3,v);

2	4	1	5
2	4	1	5
2	1	4	5

3º maior elemento
bolha_rec(2,v);

2	1	4	5
1	2	4	5

0 1 2 3

Ordenação Bolha

- Algoritmo genérico (I):
 - independente dos dados armazenados no vetor
 - usa uma função auxiliar para comparar elementos

```
/* Função auxiliar de comparação */  
static int compara (int a, int b)  
{  
    if (a > b)  
        return 1;  
    else  
        return 0;  
}
```

Ordenação Bolha

```
/* Ordenação bolha (3a. versão) */  
void bolha (int n, int* v)  
{ int i, j;  
  for (i=n-1; i>0; i--) {  
    int troca = 0;  
    for (j=0; j<i; j++)  
      if (compara(v[j],v[j+1])) {  
        int temp = v[j]; /* troca */  
        v[j] = v[j+1];  
        v[j+1] = temp;  
        troca = 1;  
      }  
    if (troca == 0) /* não houve troca */  
      return;  
  }  
}
```

Ordenação Bolha

- Algoritmo genérico (II):
 - função de ordenação e assinatura da função de comparação independentes do tipo do elemento
 - função de ordenação: `void bolha (int n, void* v, int tam);`
 - `v` ponteiro de qualquer tipo (definido como `void*`)
 - `tam` tamanho de cada elemento em bytes (para percorrer o vetor)
 - função de comparação: `int compara (void* a, void* b);`
 - `a e b` dois ponteiros genéricos
um para cada elemento que se deseja comparar

Ordenação Bolha

- Exemplo de função de comparação:
 - dados do aluno, com nome como chave de comparação

```
/* Dados do aluno */  
struct aluno {  
    char nome[81];  
    char mat[8];  
    char turma;  
    char email[41];  
};
```

```
/* função de comparação c/ ponteiros de alunos */  
static int compara (void* a, void* b)  
{  
    Aluno** p1 = (Aluno**) a;  
    Aluno** p2 = (Aluno**) b;  
    if (strcmp((*p1)->nome, (*p2)->nome) > 0)  
        return 1;  
    else  
        return 0;  
}
```

Ordenação Bolha

- Função auxiliar para caminhar no vetor:
 - endereço do elemento $i = i * \text{tam}$ bytes
 - para incrementar o endereço genérico de um determinado número de bytes, é necessário converter o ponteiro para ponteiro para caractere (pois um caractere ocupa um byte)

```
static void* acessa (void* v, int i, int tam)
{
    char* t = (char*)v;
    t += tam*i;
    return (void*)t;
}
```

Ordenação Bolha

- Função auxiliar para realizar a troca:
 - tipo de cada elemento não é conhecido => variável temporária para realizar a troca não pode ser declarada
 - troca dos valores feita byte a byte (ou caractere a caractere)

```
static void troca (void* a, void* b, int tam)
{
    char* v1 = (char*) a;
    char* v2 = (char*) b;
    int i;
    for (i=0; i<tam; i++) {
        char temp = v1[i];
        v1[i] = v2[i];
        v2[i] = temp;
    }
}
```

Ordenação Bolha

- Algoritmo genérico (III):

`void bolha_gen (int n, void* v, int tam, int(*cmp)(void*,void*))`

- não chama uma função de comparação específica
- recebe como parâmetro uma função *callback* de comparação com a assinatura

`int cmp (void*, void*)`

Ordenação Bolha

```
/* Ordenação bolha (genérica) */  
void bolha_gen (int n, void* v, int tam, int(*cmp)(void*,void*))  
{ int i, j;  
  for (i=n-1; i>0; i--) {  
    int fez_troca = 0;  
    for (j=0; j<i; j++) {  
      void* p1 = acessa(v,j,tam);  
      void* p2 = acessa(v,j+1,tam);  
      if (cmp(p1,p2)) {  
        troca(p1,p2,tam);  
        fez_troca = 1;  
      }  
    }  
    if (fez_troca == 0) /* nao houve troca */  
      return;  
  }  
}
```

Ordenação Bolha

```
static int compara_reais (void* a, void* b)
{
    float *p1 = (float *)a;
    float *p2 = (float *)b;
    if ((*p1) > (*p2)) return 1;
    else return 0;
}

/* na main */
.....
bolha_gen(n, v, sizeof(float), compara_reais);
.....
```

Ordenação Rápida

- Ordenação rápida (“*quick sort*”):
 - escolha um elemento arbitrário x , o *pivô*
 - rearrume o vetor de tal forma que x fique na posição correta $v[i]$
 - x deve ocupar a posição i do vetor sse
 - todos os elementos $v[0], \dots, v[i-1]$ são menores que x e
 - todos os elementos $v[i+1], \dots, v[n-1]$ são maiores que x
 - chame recursivamente o algoritmo para ordenar os (sub-)vetores $v[0], \dots, v[i-1]$ e $v[i+1], \dots, v[n-1]$
 - continue até que os vetores que devem ser ordenados tenham 0 ou 1 elemento

Ordenação Rápida

- Esforço computacional:
 - melhor caso:
 - pivô representa o valor mediano do conjunto dos elementos do vetor
 - após o mover o pivô em sua posição, restarão dois sub-vetores para serem ordenados, ambos com o número de elementos reduzido à metade, em relação ao vetor original
 - algoritmo é $O(n \log(n))$
 - pior caso:
 - pivô é o maior elemento e algoritmo recai em ordenação bolha
 - caso médio:
 - algoritmo é $O(n \log(n))$

Ordenação Rápida

- Rearruração do vetor para o pivô de $x=v[0]$:
 - do início para o final, compare x com $v[1]$, $v[2]$, ... até encontrar $v[a]>x$
 - do final para o início, compare x com $v[n-1]$, $v[n-2]$, ... até encontrar $v[b]\leq x$
 - troque $v[a]$ e $v[b]$
 - continue para o final a partir de $v[a+1]$ e para o início a partir de $v[b-1]$
 - termine quando os pontos de busca se encontram ($b<a$)
 - a posição correta de $x=v[0]$ é a posição b e $v[0]$ e $v[b]$ são trocados

Ordenação Rápida

- vetor inteiro de $v[0]$ a $v[7]$
(0-7) 25 48 37 12 57 86 33 92
- determine a posição correta de $x=v[0]=25$
 - de $a=1$ para o fim: $48 > 25$ ($a=1$)
 - de $b=7$ para o início: $25 < 92$, $25 < 33$, $25 < 86$, $25 < 57$ e $12 \leq 25$ ($b=3$)

(0-7) 25 48 37 12 57 86 33 92

$a \uparrow$ $b \uparrow$

- troque $v[a]=48$ e $v[b]=12$, incrementando a e decrementando b
- nova configuração do vetor:

(0-7) 25 12 37 48 57 86 33 92

$a, b \uparrow$

Ordenação Rápida

- configuração atual do vetor:
(0-7) 25 12 37 48 57 86 33 92
 $a, b \uparrow$
- determine a posição correta de $x=v[0]=25$
 - de $a=2$ para o final: $37 > 25$ ($a=2$)
 - de $b=2$ para o início: $37 > 25$ e $12 \leq 25$ ($b=1$)
- os índices a e b se cruzaram, com $b < a$
(0-7) 25 12 37 48 57 86 33 92
 $b \uparrow a \uparrow$
 - todos os elementos de 37 (inclusive) para o final são maiores que 25 e todos os elementos de 12 (inclusive) para o início são menores que 25 – com exceção de 25
- troque o pivô $v[0]=25$ com $v[b]=12$, o último dos valores menores que 25 encontrado
- nova configuração do vetor, com o pivô 25 na posição correta:
(0-7) 12 25 37 48 57 86 33 92

Ordenação Rápida

- dois vetores menores para ordenar:
 - valores menores que 25:
(0-0) 12
 - vetor já está ordenado pois possui apenas um elemento
 - valores maiores que 25:
(2-7) 37 48 57 86 33 92
 - vetor pode ser ordenado de forma semelhante, com 37 como pivô

```

/* Ordenação rápida */
void rapida (int n, int* v)
{
    if (n <= 1) return;
    else {
        int x = v[0];
        int a = 1;
        int b = n-1;
        do {
            while (a < n && v[a] <= x) a++;
            while (v[b] > x) b--;
            ...
        }
    }
}

```

/* ver (1) */

/* ver (2) */

Obs 1:

No deslocamento para a direita,
o teste “ $a < n$ ” é necessário porque
o pivô pode ser o elemento de maior valor,
nunca ocorrendo a situação $v[a] \leq x$,
o que faria acessar posições além dos
limites do vetor

Obs. 2:

No deslocamento para a esquerda,
um teste adicional do tipo $b \geq 0$
não é necessário, pois $v[0]$ é o pivô,
impedindo que b assuma valores negativos

```

...
do {
    while (a < n && v[a] <= x) a++;
    while (v[b] > x) b--;
    if (a < b) {                /* faz troca */
        int temp = v[a];
        v[a] = v[b];
        v[b] = temp;
        a++; b--;
    }
} while (a <= b);
/* troca pivô */
v[0] = v[b];
v[b] = x;

/* ordena sub-vetores restantes */
rapida(b,v);
rapida(n-a,&v[a]);
}
}

```

Ordenação Rápida

- Quick sort genérico da biblioteca padrão:
 - disponibilizado via a biblioteca [*stdlib.h*](#)
 - independe do tipo de dado armazenado no vetor
 - implementação segue os princípios discutidos na implementação do algoritmo de ordenação bolha genérico

Ordenação Rápida

- Protótipo do quick sort da biblioteca padrão:

`void qsort (void *v, int n, int tam, int (*cmp)(const void*, const void*));`

v: ponteiro para o primeiro elemento do vetor
ponteiro do tipo ponteiro genérico (void*) para acomodar qualquer tipo de elemento do vetor

n: número de elementos do vetor

tam: tamanho, em bytes, de cada elemento do vetor

cmp: ponteiro para a função de comparação

const: modificador de tipo para garantir que a função não modificará os valores dos elementos (devem ser tratados como constantes)

Ordenação Rápida

- Função de comparação:

`int nome (const void*, const void*);`

- definida pelo cliente do quick sort
- recebe dois ponteiros genéricos (do tipo `void*`)
 - apontam para os dois elementos a comparar
 - modificador de tipo `const` garante que a função não modificará os valores dos elementos (devem ser tratados como constantes)
- deve retornar `-1`, `0`, ou `1`, se o primeiro elemento for menor, igual, ou maior que o segundo, respectivamente, de acordo com o critério de ordenação adotado

Ordenação Rápida

- Exemplo 1:
 - ordenação de valores reais

Ordenação Rápida

- Função de comparação para float:
 - os dois ponteiros genéricos passados para a função de comparação representam ponteiros para **float**

```
/* função de comparação de reais */
static int comp_reais (const void* p1, const void* p2)
{
    /* converte ponteiros genéricos para ponteiros de float */
    float *f1 = (float*)p1;
    float *f2 = (float*)p2;
    /* dados os ponteiros de float, faz a comparação */
    if (*f1 < *f2) return -1;
    else if (*f1 > *f2) return 1;
    else return 0;
}
```

```
/* Ilustra uso do algoritmo qsort para vetor de float */
#include <stdio.h>
#include <stdlib.h>

/* função de comparação de reais - ver transparência anterior */
static int comp_reais (const void* p1, const void* p2)
{...}

/* ordenação de um vetor de float */
int main (void)
{
    int i;
    float v[8] = {25.6,48.3,37.7,12.1,57.4,86.6,33.3,92.8};
    qsort(v,8,sizeof(float),comp_reais);
    printf("Vetor ordenado: ");
    for (i=0; i<8; i++)
        printf("%g ",v[i]);
    printf("\n");
    return 0;
}
```

Ordenação Rápida

- Exemplo 2:
 - vetor de ponteiros para a estrutura aluno
 - chave de ordenação dada pelo nome do aluno

```
/* estrutura representando um aluno*/
struct aluno {
    char nome[81];          /* chave de ordenação */
    char mat[8];
    char turma;
    char email[41];
};
typedef struct aluno Aluno;

Aluno* vet[N];              /* vetor de ponteiros para Aluno */
```

Ordenação Rápida

- Função de comparação
 - os dois ponteiros genéricos passados para a função de comparação representam **ponteiros de ponteiros para Aluno**
 - função deve tratar **uma indireção a mais**

```
/* Função de comparação: elemento é do tipo Aluno* */  
static int comp_alunos (const void* p1, const void* p2)  
{  
    /* converte p/ ponteiros de ponteiros de Aluno */  
    Aluno **a1 = (Aluno**)p1;  
    Aluno **a2 = (Aluno**)p2;  
    /* dados os ponteiros de ponteiro de Aluno, faz a comparação */  
    return strcmp((*a1)->nome, (*a2)->nome);  
}
```

Resumo

- Bubble sort
 - quando dois elementos estão fora de ordem, troque-os de posição até que o i -ésimo elemento de maior valor do vetor seja levado para as posições finais do vetor
 - continue o processo até que todo o vetor esteja ordenado
- Quick sort
 - coloque um elemento arbitrário x , o *pivô*, em sua posição k
 - chame recursivamente o algoritmo para ordenar os (sub-)vetores $v[0], \dots, v[k-1]$ e $v[k+1], \dots, v[n-1]$
 - continue até que os vetores que devem ser ordenados tenham 0 ou 1 elemento
- Quick sort genérico da biblioteca padrão:
 - disponibilizado via [*stdlib.h*](#), com protótipo
`void qsort (void *v, int n, int tam, int (*cmp)(const void*, const void*))`