Estruturas de Dados

P3 – 29/06/01

J. L. Rangel

1. (a) Descreva com suas palavras o funcionamento do algoritmo de ordenação conhecido por Bubble Sort, explicando em particular por que se diz que a complexidade do algoritmo é O(n^2).

(b) Mostre <u>todos</u> os passos intermediários do Bubble Sort na ordenação do vetor

```
[3 9 7 5 11 1 ]
```

em ordem crescente.

(a) a idéia básica do algoritmo de ordenação por bolha é a seguinte (caso da ordenação em ordem crescente):
Em cada passagem, cada elemento do vetor é comparado com o seguinte, e, caso seja maior que o seguinte, os dois elementos são trocados. Como resultado desta fase, o último elemento do vetor será o maior de todos os elementos do vetor, e, portanto, já está em sua posição correta, e não precisa mais ser considerado.
O processo é repetido, sem incluir os elementos do fim do vetor, que já se encontram sem suas posições corretas, de maneira que a cada vez menos elementos são comparados.
Assim, se o vetor tem n elementos, da primeira vez fazemos n-1 comparações (com ou sem troca), da segunda n-2, e assim por diante, até que na última passagem apenas uma comparação será feita. O tempo gasto pelo algoritmo portanto é proporcional a (n-1)+(n-2)+ ... +1, ou seja, é O(n^2).

(b) Temos:

vetor	comparações	
[3 9 7 5 11 1]	3:9	
[3 9 7 5 11 1]	9:7	troca
[3 7 9 5 11 1]	9:5	troca
[3 7 5 9 11 1]	9:11	
[3 7 5 9 11 1]	11:1	troca
[3 7 5 9 1 <u>11</u>]		11 está em posição
[3 7 5 9 1 <u>11</u>]	3:7	
[3 7 5 9 1 <u>11</u>]	7:5	troca
[3 5 7 9 1 <u>11</u>]	7:9	
[3 5 7 9 1 <u>11</u>]	9:1	troca
[3 5 7 1 <u>9 11</u>]		9 e 11 estão em posição
[3 5 7 1 <u>9 11</u>]	3:5	
[3 5 7 1 <u>9 11</u>]	5:7	
[3 5 7 1 <u>9 11</u>]	7:1	troca
[3 5 1 <u>7 9 11</u>]		7, 9 e 11 estão em posição

```
[3 5 1 7 9 11 ]  3:5
[3 5 1 7 9 11 ]  5:1   troca
[3 1 5 7 9 11 ]  5:1   troca
[3 1 5 7 9 11 ]        5, 7, 9 e 11 estão em posição

[3 1 5 7 9 11 ]  3:1   troca
[1 3 5 7 9 11 ]        todos estão em posição
```

2. Conjuntos de inteiros que variam de 0 (zero) a 127 podem ser implementados com vetores de bits usando a estrutura de dados definida pelo tipo

```
typedef int BITSET[4];
```

desde que o tipo `int` *tenha 4 bytes. Implemente duas funções* `exclui` *e* `complemento` *cujos protótipos são, respectivamente:*

```
void exclui(BITSET s, int i); /* s = s – {i} */
void complemento(BITSET s1, BITSET s2);
                          /* s1 = complemento de s2 */
```

- *A função* `exclui` *remove o elemento* `i` *do conjunto* `s`.
- *A função* `complemento` *calcula o complemento do conjunto* `s2`*, que passa a ser o novo valor de* `s1`*. Ou seja, na saída da função* `complemento` *o conjunto* `s1` *contém exatamente os elementos do universo (números de 0 a 127) que não estavam no conjunto* `s2`.

```
void exclui(BITSET s, int i) {
   int n=i/32;         /* int do i-esimo bit */
   int k=i%32;         /* posição do i-esimo bit no int */
   int m=~(1<<k) /* mascara: tudo 1 exceto bit k */
   s[n]=s[n] & m;/* and com m apaga k-esimo bit */
}
```

ou a versão mais curta:

```
void exclui(BITSET s, int i) {
   s[i/32] &= ~(1<<(i%32));
}

void complemento(BITSET s1, BITSET s2) {
   int i;
   for (i=0; i<=3; i++)
      s1[i]=~s2[i];
}
```

3. Suponha que queremos implementar uma representação para matrizes 5×5 *triangulares inferiores, com elementos reais (tipo* `double`*). As colunas e as linhas da matriz são numeradas de 1 a 5.*

(a) defina um tipo `TR` *apropriado para representar estas matrizes, que não gaste espaço desnecessário com os elementos da matriz que sempre têm valor zero.*

(b) escreva funções para dar valor a um elemento da matriz, e para ler um valor de uma matriz, com protótipos

```
void set(TR m, int i, int j, double v);
                          /* faz m[i, j]=v, se i>=j */
double get(TR m, int i, int j);
                          /* recupera m[i, j] */
```

(a)
```
typedef double TR[15];   /* espaço para 1+2+3+4+5 reais */
```

(b) Os elementos de `m` vão ficar na ordem

```
 0:(1,1)  1:(2,1)  2:(2,2)  3:(3,1)  4:(3,2)  5:(3,3)
 6:(4,1)  7:(4,2)  8:(4,3)  9:(4,4) 10:(5,1) 11:(5,2)
12:(5,3) 13:(5,4) 14:(5,5)
```

Note que linhas e colunas são numeradas a partir de 1. A função `pos` indica onde fica o elemento `(i,j)`:

```
int pos(int i, int j) {
   return i*(i-1)/2 + j - 1);
}
```

Portanto,

```
void set(TR m, int i, int j, double v) {
   m[pos(i,j)=v;
}
double get(TR m, int i, int j) {
   if i<j
      return 0;
   return m[pos(i,j)];
}
```

4. A forma mais simples de tratar colisões em tabelas hash consiste em usar a próxima posição vazia da tabela. Assim, para inserir um elemento x na tabela, usando uma função de hash h, buscamos a primeira das posições livres entre h(x), h(x)+1, h(x)+2, ... (se o final da tabela for atingido, continuamos circularmente a partir do início). Com este tipo de tratamento de colisões, não é permitida a remoção de elementos da tabela.

Suponha as seguintes declarações, relativas a um cadastro de alunos armazenado numa tabela hash que usa como chave de busca o nome do aluno.

```
typedef struct info *PINFO;
struct info {
   int matric;       /* número de matrícula do aluno */
   char nome[81];    /* nome do aluno */
};

#define N 256
typedef PINFO HASH[N];
```

Considerando o tratamento de colisões descrito acima e que posições não preenchidas da tabela de hash armazenam o valor `NULL`, escreva uma função de busca que recebe uma tabela hash representando um cadastro e o número de matrícula de um aluno. Esta função deve verificar se o aluno está armazenado no cadastro, retornando `1` se o aluno estiver no cadastro e `0` se não estiver. O protótipo da função deve ser:

```
int busca (HASH cadastro, int mat);
```

A função de hash a ser usada é

```
int hash (int m) {
   return m%N;
}

int busca (HASH cadastro, int mat) {
   int h=hash(mat);      /* posição inicial */
   int p=h;

   do {
      PINFO pinf=cadastro[p];
      if (pinf==NULL)
         return 0;       /* não achou */
      if (pinf->matric==mat)
         return 1;       /* achou */
      p=(p+1)%N;         /* avança circular */
   } while (p!=h);       /* deu a volta? */
   return 0;
}
```

5. Escreva um programa completo cuja entrada é o arquivo "`entrada.txt`" do diretório corrente, e cuja saída é um arquivo "`saida.txt`" também no diretório corrente. O arquivo de saída difere do arquivo de entrada apenas por ter suas linhas numeradas.

Por exemplo, se o arquivo de entrada contém as três linhas

```
A diferença entre software e hardware:
Software é o que você xinga.
Hardware é o que você chuta.
```

O arquivo de saída correspondente deve conter

```
1 A diferença entre software e hardware:
2 Software é o que você xinga.
3 Hardware é o que você chuta.
```

Uma solução possível é

```
#include <stdio.h>
int main(void) {
   FILE* ent=fopen("entrada.txt","rt");
   FILE* sai=fopen("saida.txt","wt");

   char linha[121];   /* chega? */
   int i=1;           /* contador */

   if ((ent==NULL) || (sai==NULL)) {
      printf("problemas com os arquivos\n");
      return 1;       /* qualquer coisa != 0 */
   }

   while(!feof(ent)){
      fscanf(ent,"%[^\n]",linha);
      fprintf(sai,"%d %s\n", i++, linha);
   }

   fclose(ent);
   fclose(sai);

   return 0;
}
```

Em vez de `fscanf`, pode ser usada `fgets`; outra possibilidade para o formato de saída pode ter um “tab” (`\t`), em vez do espaço.