



SCC-602 Algoritmos e Estruturas de Dados I - EC

Profa. Graça Nunes  
2º. Semestre de 2010

Prova 1(Gabarito)  
16/09/2010

Nome: \_\_\_\_\_ Nro USP: \_\_\_\_\_

1) Considere o TAD Lista Linear Sequencial, onde:

```
#define MAX 100

typedef struct{ /*tipo elemento*/

    int chave;

    char info;

}tipo_elem;

typedef struct{

    int nelem; /*número atual de elementos*/

    tipo_elem A[MAX];

}Lista;
```

As operações do TAD incluem: inicialização da lista, inserção e eliminação ordenadas, busca sequencial e binária, verificação de lista vazia ou cheia, etc.

Pede-se: defina uma nova função do TAD, **Lista\* Merge(Lista \*L1, Lista \*L2)**, que faça a intercalação (ou *merging*) de duas listas **ordenadas**, L1 e L2, retornando a lista resultante da intercalação. Por exemplo, se

L1 = (12, 35, 46) e L2 = (1, 7, 54, 100), então Resultado do Merging de L1 e L2 = (1, 7, 12, 35, 46, 54, 100)

(a) **(1.0)** Escreva o algoritmo da função

*Se L1 for vazia, retorna L2;*

*Se L2 for vazia, retorna L1;*

*Comece inspecionando o 1º. elemento de L1 e o primeiro de L2.*

*Enquanto tiver elemento em L1 e L2 faça:*

*Se o elemento atual de L1 for menor que o atual de L2*

*Então insira-o em L3 (no fim) e passe para o próximo de L1*

*Senão insira o de L2 em L3 (no fim) e passe para o próximo de L2;*

*Se L1 ou L2 ainda tiver elementos, insira-os em L3 (no fim);*

*Retorne L3.*

(b) **(2.0)** Escreva o código C da função.

```
Lista* Merge (Lista *L1, Lista *L2) {  
Lista *L3; int i=0, j=0;  
if (vazia(L1)) return L2;  
if (vazia(L2)) return L1;  
while (i <= L1->nelem & j<= L2->nelem) {  
if L1->A[i].chave < L2->A[j].chave {  
    inserir_fim( L3, L1->A[i]); i++; }  
else { inserir_fim( L3, L2->A[j]); j++; }  
};  
if (i<= L1->nelem) {  
    for (j=i; j<= L1->nelem; j++)  
        inserir_fim (L3, L1->A[j]);  
else for (i=j; i<= L2->nelem; i++)  
        inserir_fim( L3, L2->A[i]);  
return L3; }  
}
```

(c) **(1.0)** Faça a análise da complexidade de tempo de seu algoritmo em função dos tamanhos das listas.

*Resp.: Seja n e m os tamanhos de L1 e L2. Seja qual for o algoritmo de merging, deverá haver uma cópia de todos os elementos de L1 e L2 em L3, o que resulta em um número de cópias igual a n+m. Assim, o tempo crescerá linearmente em função dos tamanhos de L1 e*

L2. Se, por outro lado, considerássemos a comparação entre chaves a operação mais relevante, teríamos um número total de  $k = \min(n,m)$  comparações. No entanto, cada comparação resulta numa cópia em L3, e depois das comparações, o restante dos elementos da maior lista ainda têm de ser copiados em L3. Logo, o que predomina é o tempo de cópia dos elementos em L3.

2) (1.5) Considere a lista linear sequencial de inteiros, no array a seguir:

2	4	6	8	10	12	14	16
---	---	---	---	----	----	----	----

(a) Considerando o método de **Busca Binária**:

(i) (0.25) Quantas comparações (e com quais chaves) são necessárias na busca pelo valor 15?

Resp.: **Compara-se com as chaves: 8, 12, 14 e 16 ( 4 comparações)**

(ii) (0.25) Idem para o inteiro 6.

Resp.: **Compara-se com as chaves: 8, 4 e 6 ( 3 comparações)**

(b) Considerando a **Busca Sequencial**:

(i) (0.25) Quantas comparações (e com quais chaves) são necessárias na busca pelo valor 15?

Resp.: **Compara-se com as chaves: 2, 4, 6, 8, 10, 12, 14 e 16 ( 8 comparações)**

(ii) (0.25) Idem para o inteiro 6.

Resp.: **Compara-se com as chaves: 2, 4 e 6 ( 3 comparações)**

(c) (0.5) Generalize o número máximo de comparações na Busca Binária e na Busca Sequencial, considerando o tamanho do array igual a **n**.

Resp.: **Busca Binária:  $\sim \log_2 n$ . Busca Sequencial:  $n$**

3. (2.0) Escreva um algoritmo para determinar se uma string de caracteres está na forma:

$$x \# y$$

onde  $x$  é uma string que consiste de letras 'A' e 'B', e onde  $y$  é o reverso de  $x$  (isto é, se  $x =$  "ABABBA",  $y$  deve ser igual a "ABBABA"). Você pode somente ler um caracter por vez. Requisito: Armazene os caracteres em um array, com funcionamento de uma **pilha**.

Resp.:

Inicio

leia caracter  $c$ ;

enquanto  $c \neq \#$  faça

```

    { empilha c; leia c };

leia c;

enquanto c = topo_pilha faça

    { desempilha; leia c };

se a pilha ficou vazia então retorna TRUE

senão retorna FALSE

```

Fim

4) (1.0) Considere um TAD com o conjunto de funções (apenas seus cabeçalhos) a seguir, sobre **conjuntos de inteiros**. Complete o TAD com a definição completa de uma função booleana (retorna TRUE ou FALSE) que verifique se um inteiro **n** **pertence** a um conjunto **C**. Mas, **atenção**: você só pode usar as funções abaixo (uma ou mais), e a operação não pode alterar o conteúdo do conjunto. Repare que não é preciso saber qual é a estrutura de dados do tipo **conjunto**.

```

boolean insere_conjunto (conjunto C, int n)
/* insere n em C, se não estiver lá, e retorna TRUE; caso contrário, retorna FALSE */

```

```

boolean elimina_conjunto (conjunto C, int n)
/* elimina n de C, se n estiver lá, e retorna TRUE; caso contrário, retorna FALSE */

```

```

boolean vazio (conjunto C)
/* retorna TRUE, se conjunto C for vazio; FALSE, caso contrário */

```

Resp.:

```

boolean pertence (conjunto C, int n) {
/* n pertence a C se for possível eliminar n de C. Para manter o conjunto original, se eliminar, insere novamente */
    if vazio (C)
        return FALSE;
    else
        if elimina_conjunto(C, n) {
            return insere_conjunto(C, n);
        }
        else return FALSE
}

```

5) (1.0) Considere o seguinte tipo de dado Lista Encadeada:

```

struct Rec {
    char elem;
    struct Rec *lig;
};

```

```

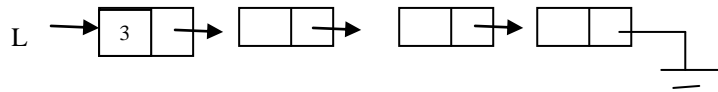
typedef struct {
    int nelem;
    struct Rec *head;

```

} Lista;

Considere também a seguinte configuração de uma lista L:

Lista \*L;



Mostre, redesenhando a estrutura (incluindo p e q), o resultado após os seguintes comandos:

```
struct Rec *p; *q;  
p = L->head;  
p->elem = 'a';  
q = p->lig;  
q->lig->elem = 'b';  
p->lig = q->lig;  
free(q);  
L->nelem--;
```

