# P4 Programação II 2012.2 Departamento de Informática/PUC-Rio

Aluno: _____

Matrícula: _____ Turma: _____

| | Valor | Nota |
|---|---|---|
| Q1 | 3.0 | |
| Q2 | 2.5 | |
| Q3 | 2.5 | |
| Q4 | 2.0 | |
| Total | 10.0 | |

1. A prova é sem consulta e **sem perguntas**. A interpretação do enunciado faz parte da prova.
2. A prova deve ser completamente resolvida nas folhas que constam deste caderno, utilizando-se frente e/ou verso;
3. A prova pode ser feita utilizando-se lápis ou caneta (azul ou preta);
4. Dispositivos eletrônicos (celulares, tablets, …) devem ser desligados.

**ATENÇÃO**: nesta prova não há necessidade de escrever comandos `#include` nas respostas.

**Q1**. **(3.0 pts).** Um programa trabalha com palavras codificadas. Uma palavra é armazenada utilizando-se duas cadeias de caracteres da seguinte forma: na primeira cadeia cada vogal da palavra original é substituída por um caractere em branco; e a segunda cadeia tem apenas a sequência de vogais que foram omitidas na primeira cadeia. Por exemplo: a palavra ESPIRITUAL é armazenada através das seguintes cadeias de caracteres:

```
char w2[] = " SP R T  L";
char vogais2[] = "EIIUA";.
```

Escreva a função `remonta` que recebe as duas cadeias e retorna uma nova cadeia de caracteres em um novo espaço de memória do tamanho exato necessário contendo a palavra completa. Durante a montagem da nova cadeia, a função deve testar se há falha na codificação por falta de espaços em branco ou de vogais. **ATENÇÃO: por questões de eficiência, a função não deve percorrer as cadeias mais do que uma vez.** Nos casos de erro, a função deve retornar NULL, deve liberar qualquer área alocada e imprimir uma das seguintes mensagens: “Erro: falta vogal”, “Erro: falta espaço em branco”, “Erro: memória insuficiente”. Como exemplos de erro, temos as cadeias `char w4[] = " SPR T L";` e `char vogais4[] = "EIIUA";` gerando a mensagem “Erro: falta espaco em branco”; e as cadeias `char w2[] = " SP R T  L";` e `char vogais2[] = "EIUA";` gerando a mensagem “Erro: falta vogal”.

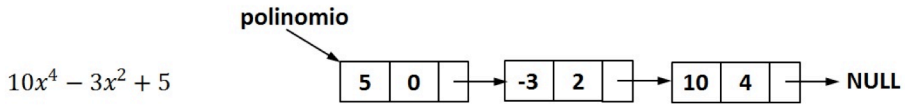
**Q2 (2.5 pts).** Considere um sistema que armazena o cadastro das consultas mensais de um médico como um vetor de ponteiros para o tipo `Consulta` descrito a seguir:

```
struct consulta
{
   int  dia;          /* dia da consulta  */
   int  hora;         /* hora da consulta */
   char paciente[81]; /* nome do paciente */
};
typedef struct consulta Consulta;
```

Este vetor está ordenado por dia em ordem crescente e, para um mesmo dia, ordenado crescentemente por hora. Suponha que não há repetições de dia e hora. Primeiro complete a função `buscaConsulta` que implementa a técnica de busca binária para encontrar o índice do vetor v de tamanho n que corresponde a um determinado dia e uma determinada hora. Depois escreva a função auxiliar `compConsulta`.

```
int buscaConsulta(Consulta ** v, int n, int dia, int hora)
{
  int ini=_________________;
  int fim= _________________;
  int meio, cmp;
  while (ini ____________________________)
  {
    meio= ____________________________________;
    cmp=compConsulta(_____________________________________________________);
    if (_____________________________)
      ___________________________________________
    else
      if (_____________________________)
        ___________________________________________
      else
        ___________________________________________;
  }
  return -1;
}
```

**Q3 (2.5 pts).** Considere que polinômios são representados por listas simplesmente encadeadas, onde cada termo do polinômio é um elemento da lista contendo o valor do coeficiente (float) e o valor do expoente (int), conforme o exemplo da figura abaixo:




Note que o primeiro elemento é o de menor expoente (no caso, o termo constante de expoente 0). Primeiro apresente a estrutura do elemento desta lista usando `struct` e depois escreva a seguinte função que avalia o polinômio para um valor dado de $x$:

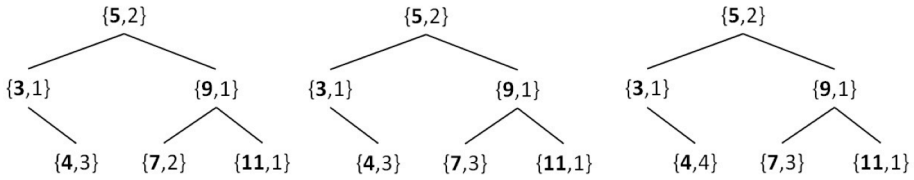
```
double avalia(Elemento * polinomio, float x)
```

Por exemplo, para $x$ = 2 e o polinômio da figura acima, a função retorna 153.00.

**Q4 (2.0 pts).** Considere que uma fábrica mantém os códigos de seus produtos organizados numa **árvore binária de busca** onde cada nó contém o código e o número de ocorrências de consulta a este código:

```
struct noArv
{
   int codigo;
   int numOcorrencia;
   struct noArv *esq, *dir;
};
typedef struct noArv NoArv;
```

A árvore está ordenada por ordem crescente de código. Por exemplo, na Fig. 2a, o produto de código 7 teve 2 ocorrências de consulta. Tirando proveito da ordenação da árvore, escreva a função recursiva `consulta` que, dado um código x, percorre a árvore e, se encontrar o código x, atualiza o número de ocorrência e retorna o endereço do nó encontrado. A função retorna NULL se não encontrar o código. Por exemplo, para a árvore da Fig. 2a, se x = 7, então a função incrementa o número de ocorrência e retorna o endereço do nó {7,3} . Se nova consulta for feita com x = 4, a função retorna o endereço de {4,4}, como mostra a Fig.2c.




**(a)** **(b)** **(c)**

Fig. 2

**stdio.h:**

```
int scanf (char* formato, ...);
int printf (char* formato, ...);
FILE* fopen (char* nome, char* modo);
int fclose (FILE* fp);
int fscanf (FILE* fp, char* formato, ...);
int fprintf (FILE* fp, char* formato, ...);
char*fgets(char* str, int size, FILE* fp);
int sscanf(char* str, char* formato, ...);
```

**stdlib.h:**

```
void* malloc (int nbytes);
void free (void* p);
```

**math.h:**

```
double sqrt (double x);
double pow (double x, double exp);
double cos (double radianos);
double sin (double radianos);
```

**string.h:**

```
int strlen (char* s);
int strcmp (char* s, char *t);
char* strcpy (char* destino, char* fonte);
char* strcat (char* destino, char* fonte);
char* strdup (char* s);
```