Mestrado Integrado em Engenharia Informática (FCT/UNL)
Ano Lectivo 2015/2016
Linguagens e Ambientes Programação – Teste 2 – Parte ❶

09 de junho de 2016 às 14:00

Teste com consulta com 1 hora e 40 minutos de duração + 15 minutos de tolerância

Nome: Num:

Notas: *Este enunciado é constituído por 4 grupos de perguntas. Responda no próprio enunciado, usando a frente e o verso.*
Pode definir funções e métodos auxiliares sempre que quiser.
Normalmente, respostas imperfeitas merecem alguma pontuação.
Fraude implica reprovação na cadeira.

1. [6 valores] Neste problema, para resolver em C, vamos trabalhar com listas ligadas de inteiros não ordenadas e onde podem ocorrer valores repetidos. Como habitualmente, cada nó contém um apontador **`next`** que indica o nó seguinte (toma o valor NULL se não existir nó seguinte). Mas agora, para acelerar algumas operações, cada nó contém um novo apontador **`jump`** que indica o nó situado 10 posições a seguir (toma o valor NULL se não existir nó situado 10 posições a seguir). Os vários apontadores **`jump`** constituem atalhos que permitem navegar mais rapidamente dentro da lista.

Na representação abaixo, cada nó tem três campos: (1) valor do nó, (2) apontador para o nó a seguir, (3) apontador para 10 nós a seguir. A função `newNode` cria nós inicializados, mas com o apontador **`jump`** ainda a valer NULL.

```
#define JUMP      10

typedef struct Node {
    int value ;
    struct Node *next; // nó seguinte
    struct Node *jump; // salta 10 nós
} Node, *List;
```

```
List newNode(int value, List next)
{
    List n = malloc(sizeof(Node));
    if( n == NULL ) return NULL;
    n->value = value;
    n->next = next;
    n->jump = NULL;
```

Importante: Programe soluções iterativas, portanto sem uso de recursão. Sempre que possível, percorra a lista-argumento apenas uma vez para que a complexidade dos algoritmos seja linear. Sempre que possível, aproveite os apontadores **`jump`** para acelerar as operações.

a) [2 valores] Escreva em C uma função para determinar o comprimento duma lista (com os atalhos **`jump`** já preenchidos).

```
int length(List l) {
```

b) [2 valores] Escreva em C uma função para obter o apontador para o último elemento duma lista (com os atalhos **`jump`** já preenchidos). Se a lista for vazia, o resultado é NULL.

```
List last(List l) {
```

c) [2 valores] Escreva em C uma função para preencher todos os apontadores **`jump`** duma lista (assumindo que isso não foi feito na altura da criação da lista). Use um algoritmo com complexidade linear.

```
List fillJump(List l) {
```

2. [4 valores] Escolha múltipla. As respostas erradas não descontam. Indique as respostas mais correctas aqui:

A	B	C

A) Relativamente a um programa escrito em C, qual das seguintes propriedades não pode ser verificada estaticamente, (i.e. não pode ser verificada em tempo de compilação):

a) Há acessos a variáveis intermédias.

b) Há situações em que dois apontadores apontam para uma mesma célula de memória (*aliasing*).

c) Todos os literais de tipo string estão corretamente delimitados por aspas.

d) Há variáveis que, apesar de definidas, não são usadas no código, sendo portanto inúteis.

B) Existem apologistas da tipificação estática e apologistas da tipificação dinâmica. Qual das seguintes alternativas corresponde a um argumento em defesa da tipificação estática.

a) No ciclo de desenvolvimento de software, quanto mais cedo se apanharem erros, mais económica será a correção desses erros.

b) É preciso que o comportamento dum programa ilegal seja encaminhado para ações controladas e bem definidas, tais como o lançamento de exceções.

c) É bom poder-se ir correndo e testando algumas partes dum programa, mesmo com outras partes ainda insuficientemente desenvolvidas e tecnicamente inválidas.

d) O objetivo principal de um sistema de tipos é detetar erros em programas de computador.

C) Na prática atual da programação para a Web, o estilo de programação assíncrona, baseada em eventos…

a) Não se usa de todo, nem do lado do cliente, nem do lado do servidor.

b) É o estilo omnipresente, sendo usado tanto do lado do cliente como do lado do servidor.

c) É o estilo omnipresente do lado do cliente, sendo menos usado do lado do servidor.

d) É o estilo omnipresente do lado do servidor, sendo menos usado do lado do cliente.

Mestrado Integrado em Engenharia Informática (FCT/UNL)
Ano Lectivo 2015/2016
Linguagens e Ambientes Programação – Teste 2 – Parte ❷

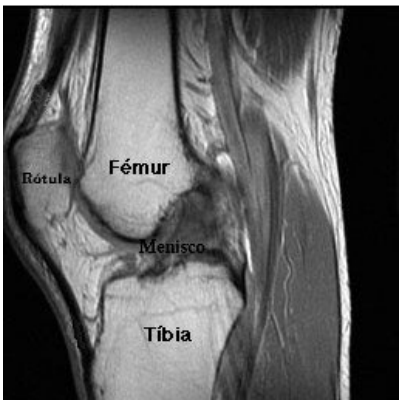
09 de junho de 2016 às 14:00

Teste com consulta com 1 hora e 40 minutos de duração + 15 minutos de tolerância

Nome:	Num:

3. [6 valores] **Esqueletos.** Não se assuste, mas esta pergunta tem a ver com esqueletos: esqueletos humanos e de outros animais vertebrados. O objectivo é definir em JavaScript um sistema de classes abstratas e concretas, bem factorizado e extensível, adequado à representação de esqueletos. Um esqueleto é constituído por ossos e cartilagens (e mais coisas de que não falamos, para simplificar).

A seguinte imagem mostra uma parte do esqueleto humano, concretamente a *articulação do joelho*, que se define como um agrupamento de quatro elementos básicos: três ossos - Fémur, Tíbia e Rótula - e uma cartilagem - Menisco.



Articulação do joelho.

Para representar esqueletos, consideramos duas grandes variedades de componentes: **COMPONENTES BÁSICAS** - **ossos** e **cartilagens** - e **COMPONENTES COMPOSTAS** - **articulações** e **partes lógicas**. Relativamente às componentes básicas, os **ossos** são elementos duros e rígidos (e.g. tíbia, perónio, clavícula) e as **cartilagens** são elementos macios e flexíveis (e.g. meniscos dos joelhos, cartilagens da caixa torácica). Quanto às componentes compostas, uma **articulação** é um grupo dois ou mais ossos e cartilagens conectados (e.g. joelho, cotovelo, inserção dum dente num maxilar); uma **parte lógica** é um qualquer agrupamento de articulações e de partes lógicas (e.g. dedo, braço, cabeça, tronco).

Para descrever completamente um esqueleto humano bastaria usar uma única parte lógica (**raiz**) e dezenas de articulações avulso. No entanto, essa representação teria falta de estrutura. Para adicionar estrutura, inserimos mais partes lógicas. Por exemplo, podemos inserir as tradicionais partes lógicas: *cabeça*, *tronco* e quatro *membros*. Neste caso, a representação do esqueleto humano passa a consistir numa parte lógica principal (**raiz**), que armazena seis partes lógicas mais simples, cada uma das quais, por sua vez, armazena articulações. Para aperfeiçoar a representação, podemos ainda acrescentar partes lógicas ainda mais detalhadas, e.g. braço, antebraço, mão, etc. Que fique claro: exceptuando a **raiz** (que é uma parte lógica obrigatória), as restantes partes lógicas são de utilização facultativa, servindo apenas para estruturar a representação.

O que foi escrito implica que um esqueleto é representado, de facto, usando uma árvore. Esta árvore é mais ou menos profunda, dependendo do número de partes lógicas consideradas. Repare também que se trata duma árvore especial, na medida em que duas ou mais articulações diferentes podem referir o mesmo osso ou cartilagem, e também duas ou mais partes lógicas podem referir a mesma articulação – nesse caso temos de reconhecer que a árvore é afinal um grafo dirigido sem ciclos.

Problema

Usando um sistema de classes abstratas e concretas, defina em JavaScript um tipo soma **`SkeletonElement`** para representar esqueletos. Preocupe-se com a extensibilidade do sistema e pense que, no futuro, podemos querer introduzir, sem muito esforço, mais tipos de componentes, digamos: ligamentos, tendões, próteses, etc.

Atributos das componentes concretas (pode introduzir mais atributos, se necessitar).

```
Bone (osso) - name, weight, strength
Cartilage (cartilagem)  - name, weight
Joint (articulação)  - name, flexibility, components (array)
LogicPart (parte lógica)  - name, components (array)
```

Operações disponíveis para todas as componentes concretas (pode definir métodos auxiliares, se precisar).

> **INIT(…)** – Métodos de inicialização.
>
> **basics()** – Retorna um array com todas as componentes básicas constituintes e apenas estas. As componentes compostas não aparecem no resultado. No resultado não interessa a ordem e podem ocorrer objetos repetidos.
>
> **weight()** – Calcula o peso total. Como num esqueleto podem aparecer referências repetidas, você tem de se preocupar em evitar contar o peso de cada componente mais do que uma vez.
>
> **validate()** - verifica que: (1) cada articulação só contém ossos e cartilagens, (2) cada parte lógica, só contém partes lógicas e articulações, (3) não existem ciclos no grafo. Escreva código extensível, por exemplo baseado nos conceitos abstratos de *básico* e de *composto*.
>
> **NOTA**: Assume-se que as operações **basics** and **weight** são aplicadas à raiz de esqueletos já validados.

Ajudas: Para representar coleções em JavaScript, use arrays.
Para acrescentar um objeto a um array: **`arr.push(obj)`**.
Para concatenar arrays: **`arr1.concat(arr2)`**.
Para testar se um objeto não faz parte dum array: **`arr.indexOf(obj) == -1`**.

A primeira linha das várias classes já foi escrita. Se tiver falta de espaço, aproveite as costas da folha.

```
var SkeletonElement = EXTENDS(JSRoot, {
```

```
var BasicElement = EXTENDS(SkeletonElement, {
```

```
var CompositeElement = EXTENDS(SkeletonElement, {
```

```
var Bone = EXTENDS(BasicElement, {
```

```
var Cartilage = EXTENDS(BasicElement, {
```

```
var Joint = EXTENDS(CompositeElement, {
```

```
var LogicPart = EXTENDS(CompositeElement, {
```

4. [4 valores] Considere o seguinte programa escrito em GCC, uma variante do C que suporta aninhamento de funções:

```
#include <stdio.h>

typedef int Fun(int);

int ret(int n) { return n; }

int fact(int n, Fun k) {
    int cont(int a) { return k(n*a); }

    if( n == 0 ) return k(1);
    else return fact(n-1, cont);
}

int main(void) {
    int res = fact(2, ret);
    printf("%d\n", res);
    return 0 ;
}
```

Mostre qual o estado da pilha de execução **no momento em que ela atinge a altura máxima**. Não se esqueça dos registos de ativação das funções `start` e `main`.

Use as convenções habituais das aulas: Para efeito da criação do registo de activação inicial, imagine que cada programa em GCC está embebido numa função sem argumentos chamada `start`. Depois trate todas as entidades globais do programa como sendo locais à função imaginária `start`. Assuma também que a primeira célula da pilha de execução é identificada como posição 00, a segunda célula como posição 01, etc.

15	31	47
14	30	46
13	29	45
12	28	44
11	27	43
10	26	42
09	25	41
08	24	40
07	23	39
06	22	38
05	21	37
04	20	36
03	19	35
02	18	34
01	17	33
00	16	32