Instituto Superior de Engenharia de Lisboa

Programação II

2020/21 – 1.º semestre letivo

Teste de época normal

2021.02.22

Grupo1
[5 valores]
Considere as definições das funções `f1`, e `f2`. Na linha 1 do troço de código seguinte deverá inserir o algarismo das unidades do seu número de aluno no local indicado. Por exemplo: se o seu número de aluno fosse o 12345, aquela linha ficaria assim: `#define ALG 5`
Nota importante: a utilização do algarismo errado penalizará significativamente a avaliação da alínea b) deste grupo.

```
1  #define ALG algarismo_das_unidades_do_seu_número_de_aluno
2  #include <stdio.h>
3  #include <stdlib.h>
4  #include <string.h>
5
6  int f2(char * s1, char * s2) {
7    return strlen(s1) == strlen(s2);
8  }
9
10 char ** f1(char * a[], size_t *size, int (user_f1)(char *, char *),
11 char * key, void user_f2(char *)) {
12   size_t i = *size;
13   while(i--)
14     if(user_f1(a[i], key)) {
15       if(user_f2) user_f2(a[i]);
16       memmove(&a[i], &a[i+1], (--*size - i) * sizeof(a[0]));
17       printf("%zu %zu\n", i, *size);
18     }
19   return realloc(a, *size * sizeof (a[0]));
20 }
```

Na caixa seguinte consta um extrato do *output* do comando `man memmove` executado em consola.

```
(…)
  void *memmove(void *dest, const void *src, size_t n);


DESCRIPTION
  The memmove() function copies n bytes from memory area src to memory area
dest. The memory areas may overlap: copying takes place as though the bytes in
src are first copied into a temporary array that does not overlap src or dest,
and the bytes are then copied from the temporary array to dest.
(…)
```

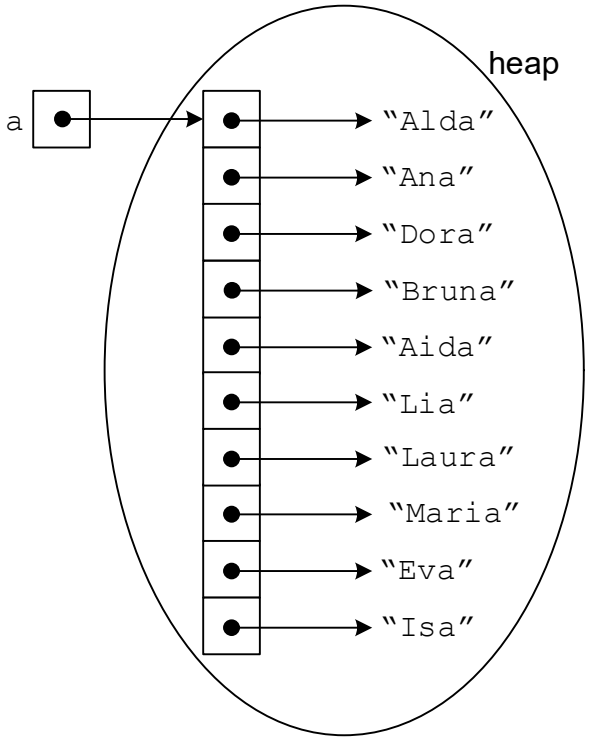
No troço de programa seguinte consta um exemplo de utilização das funções `f1` e `f2`.

```
...   char ** a;


...   (…)//Preenchimento do array dinâmico (alojado no heap) referenciado por a.
30    size_t i, size=10;
31    char **b = f1(a, &size, f2, a[ALG], NULL);
32    for(i=0; i<size; i++){
33      printf("%s\n", b[i]);
34      free(b[i]);
35    }
36    free(b);
```

a) [1] Comente o código da função `f1` e apresente, correta e completamente preenchido, o respetivo cabeçalho descritivo seguindo o formato do que se apresenta na caixa seguinte, que está preenchido (como exemplo) para a função `memmove`.

```
/*--------------------------------------------------------------------------------
Nome da função: memmove


Descrição: Esta função copia n bytes da memória apontada por src para a memória
apontada por dest. As áreas de memória podem sobrepor-se: a cópia ocorre como se
os bytes a copiar fossem primeiro copiados para um array temporário, não
sobreposto às áreas de memória apontadas por src e dst, e fossem depois copiados
para a memória apontada por dest.


Parâmetros:

void *dest: referência para a memória para onde é realizada a cópia.

const void *src: referência para a memória de onde é realizada a cópia.

size_t n: número de bytes a copiar.


Retorno:

void: não tem tipo de retorno.

--------------------------------------------------------------------------------
------*/
```

b) [1] Apresente, justificando, os valores produzidos em *standard output* resultantes da execução do troço de código anterior, admitindo que (imediatamente antes da execução da linha 30) o ponteiro **a** referencia um *array* de ponteiros para *strings* cujo conteúdo é o apresentado na figura anexa. Considere que o *array* de ponteiros e as *strings* têm alojamento no *heap*.



Nota: a ausência de justificação dos valores produzidos em *standard output* penalizará significativamente a avaliação desta alínea.

c) [1] Para o caso exemplificado no troço de código anterior:

1. Considerando que, de acordo com a figura, foram requisitados/alocados 11 blocos de memória dinâmica, um para o *array* e dez para as *strings*, indique qual o tamanho esperado, em `bytes`, de cada bloco.
2. Indique, justificando, quantas vezes é chamada a função `free` e explique que bloco de memória é libertado em cada chamada.
3. Indique, justificando, se toda a memória dinâmica requisitada/alocada foi libertada. Se foi, refira se podia ter sido libertada com menos chamadas à função `free`. Se não foi, refira, sem adicionar mais instruções ao programa, como podia ter sido libertada

d) [2] Considere que se pretende agora realizar uma função `f1a` com a mesma funcionalidade da função `f1` mas com maior generalidade, que possa operar sobre *arrays* de ponteiros para outros tipos e não apenas *arrays* de ponteiros para *strings*. Sendo uma versão da função `f1`, a função `f1a` deve seguir **de muito perto** o algoritmo da função `f1`.

Escreva a definição da função `f1a`, adicionando ou removendo parâmetros se necessário. Escreva a definição da função com uma indentação correta, usando obrigatoriamente comentários.

Rescreva a linha onde é usada a função `f1`, no exemplo de utilização anterior, substituindo-a pela utilização da função `f1a`, igualmente aplicada ao *array* referenciado por **a**.

Nota: desaconselha-se veementemente a escrita da definição desta função sem a utilização de comentários porque penalizará significativamente a avaliação desta alínea.

Grupo 2

[6 valores] Os tipos seguintes formam uma estrutura de dados para representar um livro de endereços. O tipo **Person** representa os dados de uma pessoa. O tipo **AddressBook** representa a coleção de elementos **Person** na forma de um *array* alojado dinamicamente e apontado pelo campo **people**; campo **peopleSize** identifica a quantidade de elementos preenchidos no *array*; o campo **bookSize** representa a quantidade total de elementos alojados para o *array* (podendo ser superior a **peopleSize**).

```
typedef struct {    // Descritor de um elemento do livro de endereços
    char *firstName;        // nome - string alojada dinamicamente
    char *lastName;         // apelido - string alojada dinamicamente
    char *address;// endereço - string alojada dinamicamente
    char phone[MAX_PHONE];  // algarismos do telefone, em ASCII
} Person;

typedef struct {    // Descritor de um livro de endereços
    Person *people;         // elementos de informação - array alojado dinamicamente
    int peopleSize;// quantidade de elementos preenchidos
    int bookSize;   // quantidade de elementos alojados
} AddressBook;
```

a) [1] Admita que estas estruturas de dados já foram criadas e preenchidas. Pretende-se ordenar o conteúdo do *array*, alfabeticamente, considerando: 1.º critério, campo **firstName**; 2.º critério (em caso de empate do 1.º), campo **lastName**.

Escreva a função

void sortFirstLast(AddressBook *book);

que ordena o *array* identificado pelo campo **people**, com o critério especificado. Deve utilizar a função **qsort** de biblioteca e escrever a função de comparação necessária.

b) [2] Pretende-se criar referências para subconjuntos dos elementos existentes. Estas são organizadas em listas ligadas, formadas por nós do tipo **List** seguinte.

```
typedef struct list {       // Descritor de um nó de lista ligada
    struct list *link;      // ligação na lista
    Person *elem;  // aponta o elemento referenciado
} List;
```

As listas podem ter utilização permanente ou temporária. Para o caso de utilização temporária, prevê-se a possibilidade de criar réplicas de listas, eliminar seletivamente referências ou eliminar a lista por inteiro.

Escreva a função

void listInsert(List **headAddr, Person *p);

que insere uma nova referencia, indicada por **p**, na lista identificada pelo ponteiro cujo endereço é passado em **headAddr**.

A lista é ordenada pelo conteúdo dos elementos referenciados, alfabeticamente, considerando: 1.º critério, campo **lastName**; 2.º critério (em caso de empate do 1.º), campo **firstName**.

c) [1] Escreva a função

List *listClone(List *head);

que cria uma réplica integral da lista de referências indicada por **head**. A função retorna o endereço do primeiro elemento da nova lista.

d) [2] Escreva a função

void listFilter(List **headAddr, char *name);

que elimina alguns elementos da lista identificada pelo ponteiro cujo endereço é passado em **headAddr**. São mantidas na lista as referências para elementos que têm pelo menos um dos nomes (**firstName** ou **lastName**) idêntico ao indicado por **name**. São eliminadas as restantes; o seu espaço de alojamento dinâmico deve ser reciclado.

Grupo 3 (6 val.)

[6 valores] No grupo anterior foram especificadas as estruturas de dados para representar um livro de endereços e listas de referências para subconjuntos dos seus elementos.

```
typedef struct {    // Descritor de um elemento do livro de endereços
    char *firstName;        // nome - string alojada dinamicamente
    char *lastName;         // apelido - string alojada dinamicamente
    char *address;// endereço - string alojada dinamicamente
    char phone[MAX_PHONE];  // algarismos do telfone, em ASCII
} Person;

typedef struct {    // Descritor de um livro de endereços
    Person *people;         // elementos de informação - array alojado dinamicamente
    int peopleSize;// quantidade de elementos preenchidos
    int bookSize;   // quantidade de elementos alojados
} AddressBook;

typedef struct list {       // Descritor de um nó de lista ligada
    struct list *link;      // ligação na lista
    Person *elem;  // aponta o elemento referenciado
} List;
```

Foram também especificadas as seguintes funções

void listInsert(List **headAddr, Person *p); Insere, na lista indicada por ***headAddr**,.uma nova referência indicada por **p**.

List *listClone(List *head); Produz uma réplica integral da lista indicada por **head**.

void listFilter(List **headAddr,, char *name); Elimina, da lista indicada por ***headAddr**, as referências que não contêm o nome indicado por **name**.

Considera-se ainda a existência das funções:

void listDelete(List *head); Elimina integralmente a lista indicada por **head**.

void listPrint(List *head); Apresenta em *stdout* os dados das pessoas referenciadas na lista indicada por **head**.

Pretende-se criar uma estrutura de dados para aceder de forma eficiente aos elementos que têm um determinado nome num dos seus campos (**firstName** ou **lastName**).

Para isso, cria-se uma árvore binária formada pelos nós com o tipo **Tree** seguinte. Cada nó, além das ligações na árvore, representa uma palavra correspondente a um dos nomes e dispõe de uma lista de referências para as pessoas que têm esse nome num dos seus campos.

```
typedef struct tree {       // Descritor de um nó de árvore binária
    struct tree *left, *right;      // ligação na árvore
    char *name;    // uma parte do nome dos elementos referenciados
```

```
    List *people;   // lista com os elementos que contêm "name"
} Tree;
```

Admita que existe a função

void treeInsertName(Tree **rootAddr, char *name, Person *elem);

que insere na árvore uma referência, indicada por **elem**, associada ao nome indicado por **name**. A ordenação da árvore é alfabética pelo campo **name**, com menores ligados pelo campo **left**.

a) [1] Escreva a função

void treeReferElem(Tree **rootAddr, Person *elem);

que, utilizando a anterior, adiciona à árvore, indicada por ***rootAddr**, duas referências para o elemento indicado por **elem**, respetivamente associadas aos seus campos de nome - **firstName** e **lastName**.

b) [1] Escreva a função

Tree *treeReferBook(AddressBook *book);

que, utilizando a anterior, constrói a árvore para referenciar todos os nomes (**firstName** e **lastName** de todos os elementos) existentes no livro de endereços.

c) [2] Escreva a função

List *treeSearch(Tree *root, char *name);

que procura, na árvore indicada por **root**, e retorna o acesso à lista de referências associada ao nome indicado por **name**.

d) [2] Escreva a função

void printSelected(Tree *root, char *name1, char *name2);

que, utilizando a árvore e as funções convenientes, apresenta os dados dos elementos que contêm simultaneamente os dois nomes indicados por **name1** e **name2**. Não importa se cada um dos parâmetros **name1** e **name2** é **firstName** ou **lastName**.

Se o parâmetro **name2** for NULL, a função apresenta todos os elementos referenciados por **name1**.

A estratégia proposta é usar listas temporárias de referências: criar uma réplica da lista de referências associada a **name1**; eliminar seletivamente os elementos que não contêm o nome indicado por **name2**; apresentar o conteúdo das referências válidas; eliminar a lista temporária.

Grupo 4 [3 valores] Considere um conjunto de módulos escritos em linguagem C, compilados individualmente com o comando "**gcc -c *.c**". Na caixa abaixo apresenta-se, dos módulos compilados, as respetivas listas de símbolos, resultantes do comando "**nm *.o**", classificados com as abreviaturas: `T`, *public text*; `t`, *private text*; `U`, *undefined*; `d`, *private BSS data*.

```
copy.o:
0000000000000000     T copy

data.o:
0000000000000015     T dAdd
000000000000004d     T dEnd
000000000000003f     T dGetIdx
0000000000000034     T dSize
0000000000000000     T dStart
                     U fopen
                     U malloc
                     U realloc
norm.o:
0000000000000000     T normalize

print.o:
0000000000000000     T print
                     U printf

read.o:
0000000000000000 b f
                     U fclose
                     U fgets
                     U fopen
                     U malloc
                     U normalize
0000000000000087     T rEnd
0000000000000048     T rItem
0000000000000000     T rStart
0000000000000026     t split
                     U strtok
```

Considere também que há um módulo de aplicação, **demo.c**, contendo a função **main** seguinte

```
int main( int arc, char **argv ){
        rStart( argv[1] );
        Item * d;
        while( ( d = rItem() )!= NULL ){
                dAdd( d );
        }
        rEnd();
        for( int i = 0; i < dSize(); ++i ){
                print( dGetIdx( i ) );
        }
        return 0;
}
```

Admita que são criados e usados no processo de compilação os *header files* **data.h**, **norm.h**, **print.h** e **read.h**, cada um contendo as assinaturas das funções públicas do módulo fonte (**.c**) com o respetivo prefixo. Existe também o *header file* **item.h**, com a definição do tipo **Item**, e é incluído em todos os módulos fonte.

a) [0,5] Apresente a lista de símbolos produzida por "**nm demo.c**" e a respetiva classificação (t, T, U, etc.).

b) [0,5] Diga, justificando, se há funções definidas com o atributo **static**.

c) [1] Escreva o *header file* **read.h**, tendo em conta o controlo de inclusão múltipla e considerando assinaturas das funções compatíveis com a utilização na função **main**. Indique, justificando, quais os módulos de código fonte (**.c**) onde o *header file* **read.h** deve ser incluído.

d) [1] Com base nos símbolos listados, identifique os módulos que devem ser ligados ao módulo de aplicação **demo.o** para gerar o executável. Escreva um *makefile* que produza, de forma eficiente, o executável com o nome "**demo**" a partir dos módulos fonte (**.c**).