Ficha 4

Programação Imperativa

1 Estruturas Ligadas em Array

Considere que se pretende implementar estruturas ligadas usando memória automática (i.e., sem usar a *heap* nem as habituais e pré-definidas operações de `malloc` e `free`). Vamos supor que queremos usar estas estruturas ligadas para implementar conjuntos de inteiros. Em exercícios anteriores fizemos isso usando listas ligadas (ordenadas) de inteiros ou árvores binárias de procura.
Para estes dois casos, os tipos de dados que definimos foram:

```
typedef struct lnode {
     int value;
     struct lnode *next;
} Lnode, *List;

typedef struct tnode {
     int value;
     struct tnode *left, *right;
} Tnode, *Tree;
```

A operação de acrescentar um elemento ao fim de uma lista pode ser definida como:

```
List snoc (List l, int x) {
   List new = (List) malloc (sizeof (Lnode));
   new->value = x; new->next = NULL;
   if (l == Null) r = new;
   else { r = l;
          while (l->next != NULL)
                l = l->next;
          l->next = new;
        }
   return r;
}
```

Pretendemos implementar estas mesmas estruturas dinâmicas num array.

```
#define MAX ...
typedef struct cell {
    int value;
    int next;
} Cell, Heap [MAX];
```

Em cada célula da heap guardaremos o valor e o índice onde está guardada a próxima célula. Precisamos, tal como na implementação acima, de ter uma forma de nos referirmos à lista vazia. Vamos ainda assumir que existe definida uma variável global `HEAP` que vai conter a memória.

```
#define VAZIA -1
Heap HEAP;
```

Finalmente, o nosso tipo de listas ligadas vai passar a ser definido como o índice do array `HEAP` onde está armazenada o primeiro elemento da lista.

```
typedef int LList;
```

Vejamos como definir a função `snoc` para esta nova implementação de listas.

```
LList Lsnoc (LList l, int x) {
   LList new = memAlloc();
   HEAP[new].value = x; HEAP[new].next = VAZIA;
   if (l == VAZIA) r = new;
   else { r = l;
          while (HEAP[l].next != VAZIA)
                l = HEAP[l].next;
          HEAP[l].next = new;
        }
   return r;
}
```

1. Defina as funções de gestão de memória necessárias:
   (a) `int memAlloc ()` que retorna um índice do array `HEAP` livre (marcando-o como ocupado).
   (b) `void memFree (int i)` que recebe um endereço ocupado do array `HEAP` e o marca como livre.
   (c) `void memInit ()` que inicializa o array `HEAP` com os endereços todos livres.
2. Adapte as funções de consulta e remoção de um elemento da lista para esta implementação de listas ligadas de inteiros.
3. Esta estratégia para implementar listas ligadas em array pode ser adaptada para implementar outras estruturas ligadas. Por exemplo, se quisermos implementar árvores binárias em array, podemos usar as seguintes definições:

   ```
   #define MAX ...
   typedef struct treecell {
       int value;
       int esq, dir;
   } TreeCell, Heap [MAX];
   ```

   Adapte as definições de manuseamento de árvores binárias para esta representação.

2 Ficheiros

1. Considere o seguinte tipo para representar uma lista ligada de inteiros:

```
typedef struct slint {
    int valor;
    struct slint *prox;
} NLint, *Lint;
```

   Defina funções `writeLint` e `readLint` de escrita e leitura de uma destas listas em ficheiro de forma a garantir que a leitura de uma lista previamente escrita, preserva a ordem dos elementos na lista.

```
int writeLint (FILE *f, Lint l);
int readLint (FILE *f, Lint *l);
```

2. Suponha agora que pretendemos armazenar listas de *strings* (com tamanho variável).

```
typedef struct slstr {
    char  *valor;
    struct slstr *prox;
} NLstr, *Lstr;
```

   Repita o exercício anterior para este tipo de listas.

3. Considere a seguinte definição de uma árvore binária em que cada nodo contem um inteiro e uma *string* de tamanho variável.

```
typedef struct spares {
   int n; char *s;
   struct spares *esq, *dir;
} NPar, *ABPares;
```

   Defina funções de escrita e leitura de uma destas árvores em ficheiro. Note que, para que estas operações preservem a forma da árvore é necessário que os nodos da árvore sejam escritos seguindo uma travessia *preorder*.

4. Considere o seguinte tipo para representar um contacto numa lista telefónica:

```
typedef struct contacto {
   char *nome;
   char *email;
   int numeros; // tamanho do array telefones
   char **telefone; //array com os contactos (posicoes nao ocupadas a NULL)
} Contacto;
```

   (a) Considere que esta lista telefónica está organizada numa lista ligada ordenada por ordem crescente do nome:

```
typedef struct ltelf {
    Contacto c;
    struct ltelf *seg;
} Nodo, *ListaT;
```

i. Defina funções de gestão da lista (acrescentar/remover um contacto, acrescentar/remover um número de telefone de um contacto existente) e de consulta da informação de um dado nome.

ii. Defina funções de escrita/leitura da informação em ficheiro (de acesso sequencial).

(b) Considere agora que esta lista telefónica está organizada numa árvore binária de procura ordenada por ordem crescente do nome:

```
typedef struct ltelfB {
    Contacto c;
    struct ltelfB *esq, *dir;
} NodoB, *ListaB;
```

i. Defina funções de gestão da lista telefónica (acrescentar/remover um contacto, acrescentar/remover um número de telefone de um contacto existente) e de consulta da informação de um dado nome.

ii. Defina funções de escrita/leitura da informação em ficheiro que deverão preservem a forma da árvore.

3 Estruturas ligadas em ficheiro

Nas alíneas anteriores, o problema da persistência dos dados de um programa foi resolvida providenciando funções que armazenam os dados num ficheiro e que os lêem de ficheiro. Esta solução é viável (e provavelmente indicada) quando se trata de estruturas de pequena dimensão que podem ser totalmente armazenadas em memória e cujo tempo de leitura ou escrita não a torne inviável.
Uma solução alternativa consiste em armazenar os dados apenas em ficheiro (de leitura/escrita). Aqui torna-se indispensável que continuemos a ter os dados organizados nessas mesmas estruturas ligadas de forma a melhorar o comportamento das várias operações.
Podemos aplicar os mecanismos usados na secção 1, vendo um ficheiro como um array de *bytes*.
As operações de selecção e escrita numa determinada posição de um ficheiro são conseguidas à custa das seguintes operações elementares:

- `long ftell (FILE *f)` que devolve a actual posição no ficheiro `f`
- `int fseek (FILE *f, long offset, int base)` que coloca o ficheiro posicionado na posição `offset` relativamente a `base`. Esta última (`base`) pode ser uma das seguintes constantes pré-definidas
  - `SEEK_SET` significando o início do ficheiro
  - `SEEK_END` signifcando o fim do ficheiro
  - `SEEK_CUR` significando a posição actual

Esta função retorna `0` em caso de sucesso.

- `size_t fread (void *mem, size_t tam, size_t n, FILE *f)` que lê do ficheiro `f` (a partir da posição actual) `n` items, cada um com tamanho `tam` (bytes) e coloca-os no endereço `mem`. A função retorna o número de items lidos.
- `size_t fwrite (void *mem, size_t tam, size_t n, FILE *f)` que escreve no ficheiro `f` (a partir da posição actual) `n` items, cada um com tamanho `tam` (bytes) a partir do endereço `mem`. A função retorna o número de items escritos.

Considere então o seguinte tipo de dados para representar um contacto:

```
typedef struct contacto {
   char nome[80];
   char email[80];
   char telefone[10];
} Contacto;
```

1. Para armazenar os contactos num ficheiro, organizados numa lista ordenada pelo nome, vamos estabelecer o seguinte:
   - No início do ficheiro vamos armazenar o endereço onde está armazenado o primeiro contacto.
   - Para cada contacto, armazenamos também o endereço onde está armazenado o contacto seguinte. Se se tratar do último contacto, esse valor será `0L`. Para isso vamos definir o seguinte tipo:

     ```
     typedef struct registo {
         Contaco c;
         long proximo;
     } Registo;
     ```

   - Um novo registo será sempre escrito no final do ficheiro (e ligado convenientemente na lista).

   Defina as seguintes funções:

   (a) `FILE *abreFich (char *nome)` que abre o ficheiro `fich`. Caso ele não exista, deverá ser criado de forma a obedecer aos pressupostos apresentados.

   (b) `long novo (FILE *f)` que devolve o endereço onde deve ser escrito um novo registo no ficheiro.

   (c) `long primeiro (FILE *f)` que devolve o endereço onde está armazenado o primeiro contacto (`0L` se não houver nenhum).

   (d) `void dumpLista (FILE *f)` que escreve no ecran a lista (ordenada) dos contactos.

   (e) `int lookup (Contacto *dest, FILE *f, char *nome)` que procura a informação relatva ao nome `nome`. A função deverá retornar `0` em caso de sucesso (preenchendo `dest`).

   (f) `int acrescenta (FILE *f, char *nome, char *email, char *telef)` que acrescenta um novo contacto. Se o nome já existir, a função deve actualizar os dados (email e telfone). Nesse caso deve retornar `1`, caso contrário deverá retornar `0`.

2. A solução descrita acima pressupõe que não se fazem remoções. Se tal não for o caso, e tal como fizemos na secção 1, devemos organizar as posições apagadas numa lista de forma a serem reaproveitadas em futuras inserções. Para isso vamos armazenar no início do ficheiro a informação sobre o endereço da primeira posição apagada, bem como o do primeiro contacto.

```
typedef struct controlo {
    long primeiro;
    long apagados;
} Controlo;
```

   (a) Redefina a função `FILE *abreFich (char *nome)` que abre o ficheiro `fich`. Caso ele não exista, deverá ser criado de forma a obedecer aos pressupostos apresentados.

   (b) Redefina a função `long novo (FILE *f)` que devolve o endereço onde deve ser escrito um novo registo no ficheiro. Se o endereço devolvido for de um dos registos previamente apagados, esse endereço deve passar a ser considerado usado!

   (c) Defina a função `void liberta (FILE *f, long end)` que marca como livre o endereço `end` do ficheiro `f`.

   (d) Defina a função `int remove (FILE *f, char *nome)` que remove da lista de contactos a informação sobre `nome`. A função devolve um código de erro (`0` em caso de sucesso).

3. Considere agora que se pretende armazenar os contactos num ficheiro, organizados numa árvore binária de procura (ordenada pelo nome).

```
typedef struct registo {
    Contaco c;
    long esq, dir;
} Registo;
```

   Defina as seguintes funções:

   (a) `void dumpArvore (FILE *f)` que escreve no ecran a lista (ordenada) dos contactos (travessia *inorder* da árvore).

   (b) `int lookup (Contacto *dest, FILE *f, char *nome)` que procura a informação relativa ao nome `nome`. A função deverá retornar `0` em caso de sucesso (preenchendo `dest`).