AULAS 9 E 10 - ÁRVORES BINÁRIAS DE PESQUISA

O tipo de dados abstrato ABP é constituído pelos ficheiros de interface **abp.h** e de implementação **abp.c** e permite a manipulação de árvores binárias de pesquisa que representam **conjuntos de frações**, com capacidade de múltipla instanciação e controlo centralizado de erros. A estrutura de dados de suporte da árvore binária é uma estrutura constituída pelo campo **`Size`**, para armazenar o número de nós existentes na árvore, e pelo campo **`Root`**, que armazena o ponteiro para a raiz da árvore.

Para implementar e simular toda a sua funcionalidade vai precisar dos tipos de dados abstratos Fraction (**fraction.h** e **fraction.c**) – na sua versão simplificada e que deve assegurar que está totalmente operacional –, Queue (**queue.h** e **queue.c**) e Stack (**stack.h** e **stack.c**).

Acrescente ao tipo de dados abstrato a seguinte funcionalidade:

- Implemente o **construtor** para criar uma árvore vazia, que devolve a referência da nova árvore ou NULL, caso não consiga criar a árvore por falta de memória. A função deve ter o seguinte protótipo:

```
PtABP ABPCreate (void);
```

- Complete o **destrutor** implementado a **função recursiva** que atravessa a árvore e liberta toda a memória dinâmica ocupada pelos nós e pelos elementos (i.e., frações) referenciados. A função deve utilizar a função interna `ABPNodeDestroy` e deve ter o seguinte protótipo:

```
static void DestroyTree (PtABPNode *proot);
```

- Uma função para procurar um dado elemento na árvore. A função devolve 1 em caso afirmativo e 0 em caso contrário. A função deve ter o seguinte protótipo:

```
int ABPSearch (PtABP ptree, PtFraction pelem);
```

  A função tem os seguintes valores de erro: OK, NO_ABP, ABP_EMPTY, NULL_PTR (se `pelem` for um ponteiro nulo) ou NO_ELEM (se não existir na árvore), e deve utilizar uma **função interna recursiva**, que atravessa a árvore e procura o elemento dado, com o seguinte protótipo:

```
static PtFraction SearchTree (PtABPNode proot, PtFraction pelem);
```

- Uma **função repetitiva** para inserir um novo elemento na árvore. A função deve ter o seguinte protótipo:

```
void ABPInsert (PtABP ptree, PtFraction pelem);
```

  A função tem os seguintes valores de erro OK, NO_ABP, NULL_PTR, NO_MEM ou REP_ELE (se o elemento já existir na árvore). Esta operação deve incrementar o campo **`Size`** quando a inserção ocorre com sucesso.

- Uma **função repetitiva** para devolver o menor elemento (i.e., fração) armazenado na árvore. A função deve ter o seguinte protótipo:

```
PtFraction ABPMin (PtABP ptree);
```

  A função tem os seguintes valores de erro OK, NO_ABP ou ABP_EMPTY.

- Uma função para devolver o maior elemento (i.e., fração) armazenado na árvore. A função deve ter o seguinte protótipo:

```
PtFraction ABPMax (PtABP ptree);
```

  A função tem os seguintes valores de erro OK, NO_ABP ou ABP_EMPTY e deve utilizar uma **função interna recursiva**, que atravessa a árvore e encontra o elemento pretendido, com o seguinte protótipo:

```
static PtFraction MaxTree (PtABPNode proot);
```

- Uma **função repetitiva** para fazer uma cópia da árvore e que devolve a referência da nova árvore ou NULL, caso não consiga criar a árvore por falta de memória. A função deve ter o seguinte protótipo:

```
PtABP ABPCopy (PtABP ptree);
```

  A função tem os seguintes valores de erro OK ou NO_MEM. Para realizar esta operação tem de percorrer a árvore com uma **travessia em pré-ordem usando uma pilha** para inserir as frações na nova árvore.

- Uma **função repetitiva** para devolver uma **fila contendo referências (i.e., ponteiros) para as frações existentes na árvore**. A função deve ter o seguinte protótipo:

```
PtQueue ABPFillQueue (PtABP ptree);
```

  A função tem os seguintes valores de erro OK, NO_ABP ou NO_MEM e se a árvore estiver vazia deve devolver uma fila vazia. Para realizar esta operação, deve percorrer a árvore com uma **travessia por níveis usando uma fila** para inserir as referências para as frações na fila de saída.

As próximas funções que fazem, respetivamente, a união, a diferença e a intersecção de duas árvores (i.e., conjuntos) têm de **utilizar a função** anterior **`ABPFillQueue`** usando a seguinte estratégia: após validar a existência das duas árvores, criam uma fila que referencia as frações de uma das árvores (a árvore em questão depende do objetivo pretendido) e, depois de testar a sua existência na outra árvore, inserem/removem as frações na primeira árvore. **A segunda árvore não deve ser modificada**.

- Uma **função repetitiva** para adicionar à primeira árvore todos os elementos existentes na segunda árvore. A função deve ter o seguinte protótipo:

```
void ABPReunion (PtABP ptree1, PtABP ptree2);
```

  A função tem os seguintes valores de erro OK, NO_ABP ou NO_MEM.

- Uma **função repetitiva** para remover da primeira árvore todos os elementos existentes na segunda árvore. A função deve ter o seguinte protótipo:

```
void ABPDifference (PtABP ptree1, PtABP ptree2);
```

  A função tem os seguintes valores de erro OK, NO_ABP ou NO_MEM.

- Uma **função repetitiva** para reter na primeira árvore todos os elementos existentes na segunda árvore, ou seja, removendo todos os que não lhe pertencem. A função deve ter o seguinte protótipo:

```
void ABPIntersection (PtABP ptree1, PtABP ptree2);
```

  A função tem os seguintes valores de erro OK, NO_ABP ou NO_MEM.

Tem à sua disposição um programa de simulação **trababp.c** e a *makefile* **mkabp**. Tem ainda ficheiros de texto com frações – que podem ser carregados em árvores através da operação **`ABPCreateFile`**. O ficheiro **abp0.txt** não armazena qualquer fração pelo que permite a criação de uma árvore vazia. O ficheiro **abp1.txt** armazena 7 frações – sendo que uma é uma fração inválida e outra é uma fração repetida – pelo que permite a criação de uma árvore com 5 elementos.

Nota muito importante: As funções devem ser implementadas de modo eficiente, sem visitar nós desnecessários. As funções devem funcionar corretamente mesmo para árvores vazias. As funções devem verificar as situações de erro indicadas e atualizar o erro de acordo com as mesmas usando os códigos de erro apropriados de forma consistente. Todas as operações que fazem alocação de memória dinâmica (criação ou inserção de elementos nas árvores, filas e pilhas) devem ser validadas e, em caso de erro, devem executar as medidas mais adequadas em cada caso. Todos as estruturas de dados dinâmicas (filas e pilhas) criadas para realizar as travessias das árvores devem ser eliminadas – a fim de libertar a memória dinâmica ainda ocupada por elas – no fim da execução dos algoritmos, onde quer que o algoritmo termine.