Instituto Superior de Engenharia de Lisboa

LEIRT

Programação II

2024/25 – 2.º semestre letivo

Terceira Série de Exercícios

Esta série de exercícios tem como objetivo principal a exploração de estruturas de dados dinâmicas com diversas topologias: *array* dinâmico, lista ligada; árvore binária de pesquisa; *hash-table*.

São aproveitadas algumas das funcionalidades desenvolvidas nas séries anteriores. Propõe-se a construção de novas funcionalidades, usando as estruturas de dados referidas, nomeadamente para memorizar as palavras existentes e a sua localização nos textos, de modo a permitir a pesquisa das palavras e a exibição das linhas onde se encontram.

Tal como na série anterior, pretende-se o desenvolvimento em módulos separados, acompanhados dos *header files* (.h) adequados e dos ficheiros *makefile* para gerar os executáveis especificados.

1. Módulo de leitura de texto.

   Pretende-se estruturar, num módulo, o código de leitura de ficheiros de texto, usando a função `getline` da biblioteca normalizada (em substituição da função `fgets` usada nas séries anteriores). A função `getline` apresenta a vantagem de usar alojamento dinâmico, alojando ou redimensionando, de forma automática, o espaço necessário para depositar a linha. Aproveita-se este exercício também para identificar e utilizar a localização de cada linha no ficheiro.

1.1. Escreva o módulo `textread.c` e o respetivo *header file* `textread.h`, disponibilizando as funções seguintes:

```
int textStart( char *fileName );
```
abre o ficheiro indicado para leitura, regista o seu acesso, para uso das funções seguintes, numa variável de estado com alojamento permanente, pertencente ao módulo. Retorna 1, em caso de sucesso, ou 0, em caso de erro.

```
char *textSequenceLine( long *storeOffset );
```
lê, sequencialmente, uma linha do ficheiro anteriormente aberto e deposita o seu conteúdo num espaço de memória alojado dinamicamente. Retorna o endereço da memória onde o conteúdo da linha foi depositado ou NULL, para indicar fim de leitura. O código utilizador desta função fica responsável pela gestão da memória que armazena a linha. O parâmetro `storeOffset` permite registar, por afetação da variável apontada, o *offset* do ficheiro onde se localiza a linha lida; se `storeOffset` for NULL indica que não se pretende registar o *offset*. Para ler a linha, deve usar a função `getline`; Para obter o seu *offset* deve utilizar a função `ftell` da biblioteca normalizada, antes da leitura.

```
char *textLocatedLine( long offset );
```
lê uma linha, localizada na posição `offset` do ficheiro anteriormente aberto, e deposita-a num espaço de memória alojado dinamicamente. Retorna o endereço da memória onde o conteúdo da linha está depositado, ou NULL em caso de insucesso, devido a *end-of-file* ou a qualquer erro. O código utilizador desta função fica responsável pela gestão da memória que armazena a linha. O posicionamento para leitura no *offset* indicado deve ser realizado com a função `fseek` da biblioteca normalizada.
A função `textLocatedLine` modifica a localização selecionada para leitura do ficheiro, pelo que não deve ser usada durante a fase em que se usa `textSequenceLine` para leitura sequencial.

`void textEnd( void );`
termina o acesso ao ficheiro anteriormente aberto; se houver memória de alojamento dinâmico sob o controlo do módulo, deve libertá-la.

Por exemplo, para realizar a leitura sequencial de textos deve usar estas funções numa sequência por cada ficheiro, começando por `textStart`, prosseguindo repetitivamente com `textSequenceLine` e terminando com `textEnd`; Noutra fase, para realizar a leitura arbitrária de linhas isoladas, é igualmente necessário acionar `textStart`, prosseguir com uma ou várias leituras através de `textLocatedLine`, nas posições pretendidas, e terminar com `textEnd`.

A especificação das funções propostas pode ser consultada inserindo, na consola de Linux ou num motor de busca, "`man 3 getline`", "`man 3 ftell`" e "`man 3 fseek`".

1.2. Prepare o *makefile*; escreva e ensaie o programa de teste `prog31.c`, sendo o executável `prog31`. O programa deve usar as funções do módulo para ler, sequencialmente, todas as linhas de texto de um ficheiro indicado em argumento de linha de comando. Deve reproduzir as linhas em *standard output*, apresentando antes de cada uma a respetiva localização, entre chavetas. Após a reprodução do texto, deve permanecer em ciclo; em cada iteração recebe de *standard input* um número que representa o *offset* de uma linhas e apresenta-a em *standard output*. Termina quando o utilizador inserir *enf-of-file* (Ctrl-D).

Por exemplo, se o conteúdo do ficheiro for o seguinte,
```
abc
123456
ABC
```
Deve reproduzir:
```
{0}abc
{4}123456
{11}ABC
```
No final, se o utilizador inserir, por exemplo, 4, 11 e 0, responde da forma seguinte:
```
4
123456
11
ABC
0
abc
```

2. Módulo para armazenamento da localização de linhas.

Pretende-se armazenar a localização de linhas de texto arbitrárias, pertencentes a vários ficheiros. Assume-se que os nomes dos ficheiros estão armazenados em *strings* acessíveis, durante todo o tempo de execução do programa, através de um *array* de ponteiros. O tipo `Location` seguinte destina-se a registar a localização de uma linha; o significado dos campos é descrito nos comentários. Os elementos deste tipo são agrupados em *arrays* dinâmicos (vetores, com o tipo `VecLoc`) que representam as localizações de conjuntos de linhas.

```
typedef struct{ // Descritor de uma localização de linha
   int fileIdx; //    índice no array de ponteiros para os nomes dos ficheiros
   int line;    //    número da linha no texto
   long offset; //    posição do início da linha no ficheiro
} Location;

typedef struct{     // Descritor de um vetor de localizações de linhas
   int space;       //    quantidade de elementos alojados no array dinâmico
   int count;       //    quantidade de elementos ocupados no array dinâmico
   Location *data; //    aponta array alojado dinamicamente
} VecLoc;
```

2.1. Escreva o novo módulo `vecloc.c` e o respetivo *header file* `vecloc.h`, com base nos tipos anteriores, disponibilizando as funções seguintes:

`VecLoc *vlCreate( void );`
cria, em alojamento dinâmico, o descritor de um vetor de localizações e inicia-o no estado vazio. Retorna o endereço do descritor criado.

`int vlAdd( VecLoc *vec, Location *loc );`
adiciona, ao vetor indicado por `vec`, um elemento `Location` preenchido por cópia da localização indicada por `loc`. No caso de a linha indicada já estar referenciada no *array*, não deve ser adicionada; note que a leitura das linhas será sequencial, pelo que se a linha já existir ela está referenciada na última posição do *array*. Quando ocorre inserção, o novo elemento é depositado na posição adjacente ao último anteriormente existente. Deve ser assegurado espaço, no *array* alojado dinamicamente, para o novo elemento; se necessário, deve redimensionar o espaço, fazendo-o em blocos de vários elementos, com recurso à função `realloc` da biblioteca normalizada. A função `vlAdd` retorna: 1, se inseriu dados; 0, se não, por exemplo se a linha já estiver referenciada no *array* ou se o realojamento falhar.

`int vlSize( VecLoc *vec );`
retorna a quantidade de elementos armazenados no vetor.

`Location *vlGet( VecLoc *vec, int idx );`
retorna o endereço do elemento localizado no índice `idx` do vetor indicado por `vec`. Os índices dos elementos são numerados a partir de 0.

`void vlDelete( VecLoc *vec );`
elimina o descritor do vetor indicado por `vec` e liberta o espaço de alojamento dinâmico que lhe estiver atribuído.

2.2. Escreva e ensaie o programa de teste `prog32.c`, sendo o executável `prog32`. O programa deve usar as funções do módulo especificado, para criar um vetor de localizações e registar as localizações de todas as linhas de vários ficheiros de texto. Os nomes dos ficheiros são indicados nos argumentos de linha de comando. Tendo em conta que os nomes dos ficheiros são acessíveis através dos parâmetros da função `main`, o índice de ficheiro nas localizações é relacionado com o índice a usar em `argv[]`.

A primeira fase deste programa é a leitura sequencial dos textos, usando a função `textSequenceLine`, essencialmente para obter os respetivos *offsets*. Após esta fase, o programa deve permanecer em ciclo; em cada iteração recebe de *standard input* um par de números que representa o índice de ficheiro e o número de uma linha e apresenta-a em *standard output*. Termina quando o utilizador inserir *enf-of-file*. Para obter o conteúdo de cada linha a apresentar, deve usar a função `textLocatedLine`.

3. Nova versão da aplicação para pesquisa de palavras, baseada em árvore binária de pesquisa.

Pretende-se o desenvolvimento de uma nova versão do programa de aplicação especificado no exercício 2.3 da SE2. Com o propósito de ser mais eficiente, o programa passa a armazenar as palavras existentes e a localização das linhas que as contêm, de modo a aceder rapidamente às linhas de texto para responder às pesquisas.

A nova versão, designada por `prog33`, usa uma árvore binária de pesquisa para armazenar as palavras e a localização das respetivas linhas. Cada nó da árvore, dispõe de uma *string* com a palavra representada e de um vetor com a localização das linhas que contêm essa palavra. Propõe-se o tipo `TNode` seguinte para os nós da árvore.

```
typedef struct tNode{       // Nó da árvore binária de pesquisa
   struct tNode *left, *right; // ponteiros de ligação na árvore
   char *word;              //    string alojada dinamicamente, com a palavra associada
   VecLoc *vec;             //    vetor com a localização das linhas que contêm a palavra
} TNode;
```

Dado que as pesquisas são insensíveis ao uso de maiúsculas ou minúsculas, as palavras são armazenadas apenas em minúsculas, independentemente da sua escrita original nos textos.

Tal como na versão anterior, o programa deve receber, nos argumentos de linha de comando, os nomes dos ficheiros de texto para pesquisa. O ciclo de vida do programa organiza-se em duas fases: (1) construção da estrutura de dados; (2) interação com o utilizador para execução das pesquisas. Na primeira fase, o programa lê todos os ficheiros, normaliza as linhas lidas, separa as palavras e armazena-as na árvore, em conjunto com a sua informação de localização. Na segunda fase, recebe as palavras a pesquisar e responde ao utilizador, usando a informação armazenada para ler apenas as linhas necessárias. Por cada linha de texto onde a palavra existe, deve apresentar, através de *standard output*:

- O nome do ficheiro;
- O número da linha que contém a palavra;
- O texto original dessa linha (sem o efeito da normalização).

Na construção deste programa, os alunos, além de utilizar módulos fonte com algumas funções desenvolvidas anteriormente, nas séries de exercícios anteriores e na atual, devem desenvolver o módulo de árvores binárias e a componente de aplicação.

3.1. Escreva o novo módulo de gestão da árvore binária de pesquisa, `tree.c`, e o respetivo *header file* `tree.h`, com base nos tipos anteriores, disponibilizando as funções seguintes:

`void tAddWord( TNode **rootPtr, char *data, Location *loc );`
adiciona a palavra indicada por `data` e a sua ocorrência indicada por `loc`, a uma árvore cujo ponteiro raiz é indicado por `rootPtr`. Se a palavra já existir, adiciona apenas a respetiva ocorrência. Ao adicionar a palavra, deve criar uma réplica da mesma, em memória alojada dinamicamente. Tendo em conta a existência de acentuação, a comparação de palavras deve ser realizada com a função `strcoll` da biblioteca normalizada. Para adicionar a ocorrência, deve utilizar o módulo de gestão dos vetores de localização das linhas.

`void tBalance(TNode **rootPtr);`
realiza o balanceamento de uma árvore binária, cujo ponteiro raiz é indicado por `rootPtr`. Em anexo ao enunciado é proposto um algoritmo de balanceamento.

`VecLoc *tFindWord( TNode *root, char *data );`
procura numa árvore, cuja raiz é `root`, a palavra da *string* `data` e retorna o endereço do seu vetor de localizações. Se a palavra não existir, retorna NULL.

`void tDelete( TNode *root );`
elimina uma árvore, cuja raiz é `root`, libertando toda a memória de alojamento dinâmico que está sob o seu controlo.

3.2. Escreva a função
`int wordStoreTree( char *word, void *context );`
destinada a ser passada no parâmetro `action` da função `wordProcess` da SE2 para adicionar a palavra passada no parâmetro `word` à árvore binária de pesquisa.

O parâmetro `context`, neste caso, é usado como ponteiro para uma estrutura auxiliar, com o tipo `ContextTree` seguinte, contendo o acesso ao ponteiro raiz da árvore binária e os dados de localização da linha que está a ser processada. A função `wordStoreTree` retorna sempre 1.

```
typedef struct {      // Contexto para registar ocorrência de uma palavra
   TNode **rootPtr;   //    endereço do ponteiro raiz da árvore binária
   Location locData; //    localização da linha, para registar nas ocorrências
} ContextTree;
```

3.3. Prepare o *makefile* para esta versão; escreva e teste o módulo da aplicação `prog33.c`.

Para a orgânica do programa, propõe-se a estratégia seguinte:

- Executa a primeira fase para construir a estrutura de dados, usando as funções de normalização e de separação de palavras, das séries anteriores, bem como como as da série atual para leitura sequencial de texto, inserção na árvore e nos vetores de localização das linhas;
- Realiza o balanceamento da árvore binária, para melhorar a eficiência das pesquisas subsequentes;
- Executa a segunda fase, permanecendo em ciclo a receber, do utilizador, as palavras a pesquisar;
- Após receber e normalizar cada palavra a pesquisar, procura-a na estrutura de dados, obtendo o acesso ao respetivo vetor de localizações;
- Por cada localização encontrada, lê a linha individualmente e apresenta-a na forma especificada; Pode agrupar o conjunto de localizações pertencentes ao mesmo ficheiro, de modo a abrir e fechar o ficheiro apenas uma vez para ler esse conjunto de linhas;
- A atividade termina quando o utilizador acionar o fim de inserção. Antes de terminar, o programa deve libertar toda a memória de alojamento dinâmico na sua posse.

4. Nova versão da aplicação para pesquisa de palavras, baseada em *hash-table*.

Pretende-se o desenvolvimento de uma terceira versão da aplicação de pesquisa, baseada em *hash-table*, com o propósito ensaiar este modelo de estrutura de dados e de comparar a sua eficiência com a versão baseada em árvore binária de pesquisa.

A nova versão, designada por `prog34`, usa uma *hash-table* para armazenar as palavras e a respetiva localização. A indexação na *hash-table* é realizada por uma função de *hash* que usa como chave a própria palavra a armazenar ou pesquisar. A gestão de colisões é realizada por listas ligadas cujos nós dispõem de uma *string* com a palavra representada e de um vetor com a localização das linhas que contêm essa palavra. Propõe-se os tipos seguintes para os nós das listas ligadas pertencentes à *hash-table* e para o seu descritor.

```
typedef struct hlNode{  // Nó de uma lista de palavras no mesmo índice da tabela
   struct hlNode *next; //    ponteiros de ligação na árvore
   char *word;          //    string alojada dinamicamente, com a palavra associada
   VecLoc *vec;         //    vetor com a localização das linhas que contêm a palavra
} HLNode;

typedef struct {    // Descritor de uma hash-table
   HLNode **table; //    acesso à tabela de ponteiros, alojada dinamicamente
   int size;       //    dimensão da tabela, a definir na criação
} HTable;
```

O programa comporta-se exatamente como na versão anterior e é organizado nas mesmas duas fases. A única diferença é que agora usa a *hash-table*, em vez da árvore binária, para armazenar e pesquisar a informação.

Na construção deste programa, os alunos devem igualmente utilizar módulos fonte com algumas funções desenvolvidas anteriormente, assim como desenvolver o módulo de *hash-table* e a componente de aplicação. Sugere-se que tome como base a versão com árvore binária e execute as adaptações necessárias para usar a nova estrutura de dados.

4.1. Escreva o novo módulo de gestão da *hash-table*, `hashtable.c`, e o respetivo *header file* `hashtable.h`, com base nos tipos anteriores, disponibilizando as funções seguintes:

`HTable *hCreate( int size );`
aloja o descritor para a *hash-table*, inicia-o no estado vazio e retorna o seu endereço.

`void hAddWord( HTable *ht, char *data, Location *loc );`
adiciona a palavra indicada por `data`, com a ocorrência indicada por `locPtr` a uma *hash-table*, cujo descritor é indicado por `ht`; se a palavra já existir, adiciona apenas a respetiva ocorrência. Ao adicionar a palavra, deve criar uma réplica da mesma, em memória alojada dinamicamente. Para adicionar a ocorrência, deve utilizar o módulo de gestão dos vetores de localização das linhas.

`VecLoc *hFindWord( HTable *ht, char *data );`
procura numa *hash-table*, cujo descritor é indicado por `ht`, a palavra da *string* `data` e retorna o endereço do seu vetor de localizações. Se a palavra não existir, retorna NULL.

`void hDelete( HTable *ht );`
elimina uma *hash-table*, cujo descritor é indicado por `ht`, libertando toda a memória de alojamento dinâmico que está sob o seu controlo.

4.2. Escreva a função
`int wordStoreHash( char *word, void *context );`
destinada a ser passada no parâmetro `action` da função `wordProcess` da SE2 para adicionar a palavra passada no parâmetro `word` à *hash-table*.

O parâmetro `context`, neste caso, é usado como ponteiro para uma estrutura auxiliar, com o tipo `ContextHash` seguinte, contendo o acesso ao descritor da tabela e os dados de localização da linha que está a ser processada. A função `wordStoreHash` retorna sempre 1.

```
typedef struct {      // Contexto para registar ocorrência de uma palavra
   HTable *tab;       //    endereço do descritor da tabela
   Location locData; //    localização da linha, para registar nas ocorrências
} ContextHash;
```

4.3. Prepare o *makefile* para esta versão; escreva e teste o código da aplicação `prog34.c`.

Para a orgânica do programa, propõe-se a estratégia seguinte:

- Executa a primeira fase para construir a estrutura de dados, usando as funções de normalização e de separação de palavras, das séries anteriores, bem como como as da série atual para leitura sequencial de texto, inserção na árvore e nos vetores de localização das linhas;
- Executa a segunda fase, permanecendo em ciclo a receber, do utilizador, as palavras a pesquisar;
- Após receber e normalizar cada palavra a pesquisar, procura-a na estrutura de dados, obtendo o acesso ao respetivo vetor de localizações;
- Por cada localização encontrada, lê a linha individualmente e apresenta-a na forma especificada; Pode agrupar o conjunto de localizações pertencentes ao mesmo ficheiro, de modo a abrir e fechar o ficheiro apenas uma vez para ler esse conjunto de linhas;
- A atividade termina quando o utilizador acionar o fim de inserção. Antes de terminar, o programa deve libertar toda a memória de alojamento dinâmico na sua posse.

5. Ensaio comparativo de desempenho.

Pretende-se um ensaio comparativo das diversas versões do programa, `prog22` (da SE2), `prog33` e `prog34`, usando o comando `time` para obter os tempos de execução. Propõe-se que, além dos executáveis e ficheiros de texto a processar, prepare ficheiros com palavras a pesquisar, para usar com redireccionamento de *standard input*, de modo a facilitar a execução das mesmas pesquisas com os diferentes executáveis. Sugere-se também que redirecione o *standard output* para outro ficheiro, de modo a exibir na consola apenas os tempos pretendidos.

Exemplo de comando:

```
time prog22 texto1 texto2 texto3 < ficheiro_palavras > resultados
```

Pretende-se ainda distinguir o tempo de preparação do índice face ao tempo total, de modo a determinar o tempo relativo às pesquisas. Para isso, propõe-se que cada executável seja executado em, pelo menos, dois cenários:

a) Usando um ficheiro de palavras vazio (tempo nulo dedicado a pesquisas);
b) Usando um ficheiro de palavras com conteúdo, de preferência extenso.

Tendo em conta a possibilidade de repetir algumas vezes este ensaio, é conveniente escrever um *shell-script* com os comandos de execução das diversas versões nos dois cenários indicados. Um *shell-script* é um ficheiro de texto, preparado no editor, cujas linhas são comandos reconhecidos pelo interpretador (*shell*). Para que possa ser executado, o ficheiro de *script* tem de ser marcado com atributo de execução, através do comando `chmod`.

Para que as comparações de desempenho sejam expressivas, é importante processar textos com dimensão significativa e com grande diversidade de palavras. Propõe-se a utilização de textos indicados em anexo na série anterior.

A. **Anexo** – Balanceamento da árvore binária

Para uma utilização eficiente, as árvores binárias devem ser balanceadas. Propõe-se, para simplificar, que as crie sem manter permanentemente o balanceamento, realizando-o através da função `tBalance`, no final. O código proposto abaixo considera o nó de árvore com o tipo `TNode` os campos de ligação com os nomes `left` e `right`.

Para implementar a função `tBalance`, propõe-se a técnica de balanceamento em dois passos:

1. Transformar a árvore binária numa árvore degenerada em lista ordenada, ligada pelo campo `right`, usando o algoritmo seguinte.

```
TNode *treeToSortedList( TNode *r, TNode *link ){
     TNode * p;
     if( r == NULL )
           return link;
     p = treeToSortedList( r->left, r );
     r->left = NULL;
     r->right = treeToSortedList( r->right, link );
     return p;
}
```

2. Conhecido o número de elementos, transformar a lista numa árvore, usando o algoritmo seguinte.

```
TNode* sortedListToBalancedTree(TNode **listRoot, int n) {
     if( n == 0 )
           return NULL;
     TNode *leftChild = sortedListToBalancedTree(listRoot, n/2);
     TNode *parent = *listRoot;
     parent->left = leftChild;
     *listRoot = (*listRoot)->right;
     parent->right = sortedListToBalancedTree(listRoot, n-(n/2 + 1) );
     return parent;
}
```

O código fonte destas funções está disponível no ficheiro "`se3-annex-functions.c`" distribuído em conjunto com o enunciado.