Instituto Superior de Engenharia de Lisboa

LEIRT – LEETC

Programação II

2021/22 – 2.º semestre letivo

Quarta Série de Exercícios

Nesta série de exercícios complementa-se a funcionalidade do programa especificado na série de exercícios anterior, adicionando-se um comando de pesquisa por palavras dos títulos, suportado em estruturas de dados dinâmicas – árvore binária de pesquisa e listas ligadas.

Tal como especificado nas séries de exercícios anteriores, o programa recebe, como argumento de linha de comando, o nome de um ficheiro de texto contendo a tabela de dados. Na sua atividade há duas fases distintas – preparação de dados e execução de comandos. Na fase de preparação de dados, lê a tabela e preenche a estrutura de dados. Na fase de execução de comandos, além dos comandos anteriormente especificados, é agora admitido o novo comando "`f`", pelo que apresenta ciclicamente um *prompt* e espera, de *standard input*, um dos comandos seguintes.

- `a` – (*artists*) apresenta a lista de faixas áudio ordenada por *Artist*. Se houver várias faixas do mesmo intérprete, aplica, sucessivamente, a ordem por *Album* e *Title*;
- `t` – (*titles*) apresenta a lista de faixas áudio ordenada por *Title*. Se houver várias faixas com título idêntico, aplica, sucessivamente, a ordem por *Artist* e *Album*;
- `s` *`title`* – (*search*) apresenta a faixa áudio com o título indicado pelo parâmetro. Se o título não existir, apresenta uma mensagem de erro;
- `f` *`word1 [word2 word3 …]`* – (*find*) apresenta a lista das faixas áudio cujo título contém todas as palavras indicadas. A ordenação é idêntica à do comando `t`;
- `q` – (*quit*) termina.

As listas produzidas pelos comandos "`a`", "`t`" e "`f`" em *standard output*, são por ordem alfabética crescente. Cada linha de texto em *standard output*, bem como a linha de resposta ao comando "`s`", exibe a informação de uma faixa áudio, contendo os campos da *tag* separados por ponto-e-vírgula.

1. Novas estruturas de dados dinâmicas

Para implementar o comando de pesquisa por palavras isoladas, "`f`", valorizando a eficiência, propõe-se a utilização de uma árvore binária de pesquisa. Cada nó da árvore identifica uma palavra e referencia, com uma lista ligada, o conjunto de faixas que contêm essa palavra no título.

Para representar os nós das listas e da árvore, propõe-se a criação dos tipos `LNode` e `TNode` seguintes.

```
typedef struct lNode{      // Nó de lista ligada.
    struct lNode *next;    // Ligação na lista.
    MP3Tag_t *ref;         // Ponteiro para a tag referenciada.
} LNode;

typedef struct tNode{      // Nó de árvore binária de pesquisa.
    struct tNode *left, *right;     // Ligação na árvore.
    char *word;        // Palavra associada ao nó – string alojada dinamicamente.
    LNode *list;       // Lista ligada com as referências da palavra.
} TNode;
```

O tipo `Manage_t` é modificado para conter o ponteiro raiz da árvore binária de pesquisa.

```
typedef struct{
    DinRef_t *refA;   // Referências para o comando "a".
    DinRef_t *refT;   // Referências para os comandos "t" e "s".
    TNode *bst;       // Árvore binária para pesquisar títulos por palavras isoladas
} Manage_t;
```

Sugere-se que a árvore seja iniciada no estado vazio, na função `manCreate`, e que a sua construção seja realizada gradualmente, aquando da referenciação das tags, na função `manAddTag`. Para isso, é necessário identificar, isoladamente, as palavras existentes em cada título e adicioná-las à árvore, associando a referência para a *tag* a que pertencem. No processamento de cada palavra, deve verificar-se a sua localização na árvore, ou criá-la se necessário, e adicionar a referência da *tag* à respetiva lista. Note-se que há uma ordem especificada para apresentar as *tags* no comando "`f`", pelo que é necessário manter cada lista de referências ordenada pelo critério referido.

Na execução do comando "`f`", se for indicada apenas uma palavra basta procurá-la na árvore e apresentar as faixas da respetiva lista de referências; no caso de várias palavras, pode procurar a primeira e verificar se as outras existem em cada um dos títulos referenciados, apresentando apenas as faixas cujo título contém todas as palavras.

Para melhorar a eficiência das pesquisas, é conveniente o balanceamento da árvore. Propõe-se que este seja realizado na função `manSort`. Em anexo é indicado um algoritmo para esse fim.

2. Novos módulos

Propõe-se o desenvolvimento de dois novos módulos, para implementar as listas ligadas, que referenciam conjuntos de *tags*, e a árvore binária de pesquisa. Também para estes módulos deve caraterizar a funcionalidade e definir as assinaturas das funções de interface, com vista à escrita do respetivo *header file*.

Deve manter a relação do módulo de aplicação com o módulo de gestão de dados, sendo este a utilizar as funções de interface dos novos módulos. Apresenta-se de seguida o resumo da intervenção a realizar, para os módulos existentes e para os novos.

2.1. Módulo de *arrays* dinâmicos de referências; não se prevê a necessidade de alterações.

2.2. Módulo de gestão dos dados; prevê-se a alteração das funções:

```
Manage_t *manCreate( void );
```

Além de utilizar o módulo de *arrays* de referências para criar os respetivos descritores, deve iniciar a raiz da árvore binária de pesquisa, no estado de árvore vazia.

```
void manDelete( Manage_t *man );
```

Além de utilizar o módulo de *arrays* de referências para eliminar os respetivos descritores, deve utilizar o módulo de árvore binária de pesquisa para eliminar toda a respetiva memória alojada dinamicamente.

```
void manAddTag( Manage_t *man, MP3Tag_t *tag );
```

Além de adicionar a referência para a *tag* aos dois *arrays* dinâmicos de referências, deve adicionar à árvore binária de pesquisa as palavras existentes em cada título, associando a respetiva *tag*. Poderá criar funções auxiliares para a identificação das palavras a colocar na árvore binária. Em anexo são indicados exemplos de separação de palavras, usando em alternativa uma das funções `strtok` ou `sscanf` da biblioteca normalizada.

```
void manSort( Manage_t *man );
```

Além de ordenar os *arrays* dinâmicos de referências, deve balancear a árvore binária de pesquisa.

```
void manCommand( Manage_t *man, char *cmdLine );
```

Nesta função é necessário adicionar o processamento do comando "`f`".

2.3. Módulo de leitura da tabela de dados; não se prevê a necessidade de alterações.

2.4. Módulo de listas ligadas; deve conter, pelo menos, as funções:

```
void lAddRef( LNode **hp, MP3Tag_t *tag );
```

Esta função adiciona, a uma lista ligada, um nó, no qual regista o ponteiro para a estrutura indicada por `tag`. A lista é identificada, no parâmetro `hp` (*head pointer*), pelo endereço do ponteiro para o seu primeiro elemento. A inserção na lista dever ser ordenada, com o critério adequado para a sua utilização no comando "`f`", de modo a apresentar os dados com a ordem especificada.

```
void lDelete( LNode *h );
```

Esta função elimina a lista identificada pelo parâmetro `h`. Deve libertar a memória dinamicamente alojada para os nós da lista, mas não a das *tags* referenciadas.

```
void lScan( LNode *h, void (*action)( MP3Tag_t * ) );
```

Esta função percorre a lista ligada, identificada pelo parâmetro `h` (*head*), aplicando a função passada em `action` a cada uma das *tags* referenciadas. Esta função é útil, nomeadamente, para apresentar os dados das *tags* em *standard output*.

2.5. Módulo de árvore binária de pesquisa; deve conter, pelo menos, as funções:

```
void tAddWordRef( TNode **rp, char *w, MP3Tag_t *tag );
```

Esta função adiciona uma referência, para a estrutura indicada por `tag`, à palavra indicada por `w`, localizada numa árvore binária de pesquisa. Se a palavra não existir deve ser criada, fazendo-se o alojamento dinâmico da respetiva *string*. A árvore é identificada, no parâmetro `rp` (*root pointer*), pelo endereço do ponteiro para o nó da sua raiz. A inserção na lista de referências, associada à palavra, dever ser realizada pela função `lAddRef`.

```
void tDelete( TNode *r );
```

Esta função elimina a árvore identificada pelo parâmetro `r` (*root*). Deve libertar a memória dinamicamente alojada para os nós da lista, para as palavras que eles representam e para as respetivas listas ligadas de referências, utilizando a função `lDelete`.

```
TNode *tSearch( TNode *r, char *w );
```

Esta função procura, na árvore binária de pesquisa identificada por `r` (*root*), a palavra indicada por `w`, e retorna o endereço do respetivo nó ou NULL se não existir.

2.6. Módulo de aplicação, responsável por gerir a obtenção e armazenamento dos dados, bem como a interação com o utilizador. Não se prevê a necessidade de alterações significativas; poderá implicar ajustes mínimos devido à existência do novo comando "f". Recorda-se as atividades da controladas por este módulo:

- Iniciar o funcionamento da gestão dos dados;
- Percorrer a tabela recebida por parâmetro de linha de comando, obter os dados de cada *tag*, criar a sua representação e referenciá-la na gestão de dados;
- Acionar a passagem da fase de preparação à fase de comandos, organizando os acessos ordenados e a árvore binária de pesquisa;
- Aceitar os comandos do utilizador e promover as respostas correspondentes.
- Finalizar a atividade de forma ordenada, promovendo nomeadamente a libertação da memória alojada dinamicamente, antes de terminar a execução.

3. Desenvolvimento

3.1. Escreva os novos módulos a respetiva integração no programa por fases, realizando sucessivos testes parciais para verificar o funcionamento de cada módulo na respetiva fase de desenvolvimento.

Tendo em conta a dependência de funcionalidades, o desenvolvimento deve ter a ordem seguinte:

1. Módulo de listas ligadas;
2. Módulo de árvore binária de pesquisa;
3. Adição de funcionalidades ao módulo de leitura da tabela de dados;
   Sugere-se que comece por testar o comando "`f`" com apenas uma palavra, adicionado depois o processamento de mais palavras.
4. Ajustes eventuais ao módulo de aplicação.

Em conjunto com cada módulo utilitário deve escrever o respetivo *header file*, contendo as declarações necessárias para a utilização do módulo, constando nomeadamente as assinaturas das funções de interface; no caso de definir funções auxiliares, estas devem ter *scope* privado.

Os testes parciais devem ser realizados com aplicações de teste específicas, produzidas por *makefiles* em versões adequadas a cada fase de desenvolvimento.

3.2. Organize o programa completo

Adicione ao *makefile* as regras para utilizar os novos módulos.

Reveja o código da aplicação final, verificando se há necessidade de ajustes e realizando-os se for o caso.

Teste o programa completo, reutilizando o ficheiro de dados usado para ensaio das séries anteriores; pode criar outros ficheiros se considerar conveniente.

Anexo

Identificação das palavras nos títulos

Para construir a árvore binária de pesquisa é necessário identificar, e copiar para alojamento dinâmico, cada uma das palavras existentes nos títulos das faixas. Propõe-se que use, em alternativa, uma das funções `strtok` ou `sscanf` da biblioteca normalizada. Os exemplos de código seguintes ilustram o uso destas funções.

```
void exampleSplit1( const char str[] ){
    char sc[MAX_STR];
    strcpy( sc, str );
    char *p = strtok( sc, " \t\n" );
    while( p != NULL ){
          printf( "%s\n", p );
          p = strtok( NULL, " \t\n" );
    }
}

void exampleSplit2( const char str[] ){
    char word[MAX_WORD];
    int i = 0, n;
    while( sscanf( str + i, "%s%n", word, &n ) == 1 ){
          printf( "%s\n", word );
          i += n;
    }
}
```

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`, a intervalos de várias inserções e, principalmente, 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;
}
```