Instituto Superior de Engenharia de Lisboa

LEETC – LEIRT

Programação II

2020/21 – 2.º semestre letivo

Terceira Série de Exercícios

1. Introdução

Nesta série de exercícios propõe-se o desenvolvimento de novas implementações de suporte às pesquisas especificadas na Segunda Série de Exercícios, com vista a melhorar a eficiência da sua execução. Além dos *arrays* dinâmicos já utilizados, são exploradas estruturas de dados em lista ligada, árvore binária e *hash table*.

2. Funcionalidades

Mantém-se o modelo geral do programa, as suas fases de operação e a interface com o utilizador, dispondo dos comandos anteriormente especificados.

Os comandos "t" e "s", embora produzindo o mesmo resultado, devem ter uma resposta mais eficiente. Para isso, usam novas estruturas de dados que agrupam subconjuntos de referências de modo a aceder mais rapidamente aos elementos pretendidos, sem necessitar de um varrimento exaustivo ao *array* de referências que contém o conjunto completo. As novas estruturas de dados são construídas após o preenchimento dos *arrays* de referências, aproveitando os dados preenchidos e ordenados.

3. Novas estruturas de dados

3.1. Agrupamento por terminação

Para suportar o comando "t" a nova estrutura de dados proposta é uma árvore de pesquisa binária, organizada pela *string* da terminação, que dá acesso a subconjuntos de referências, armazenadas em *arrays* parciais relacionados com cada terminação.

O tipo `TNode` seguinte carateriza os nós da árvore.

```
typedef struct tNode{
     struct tNode *left, *right; // ponteiros de ligação
     char termin[MAX_TERM];      // string - terminação representada
     RefArray *refArr;           // referencias associadas
} TNode;
```

Cada nó da árvore representa uma das terminações existentes, associando-a ao subconjunto dos descritores de ficheiro que têm essa terminação.

Os campos `left` e `right` são os ponteiros de ligação na árvore binária. O campo `termin` contém a *string* com a terminação representada. O campo `refArr` indica um *array* dinâmico de referências, idêntico aos definidos na série anterior, contendo o subconjunto das referências associadas à terminação contida em `termin`.

3.2. Agrupamento por nome

Para suportar o comando "s" a nova estrutura de dados proposta é uma *hash table* que usa listas ligadas para resolver as colisões e dá acesso a subconjuntos de referências, armazenadas em *arrays* parciais relacionados com cada nome.

O tipo `LNode` seguinte carateriza os nós das listas ligadas usadas pela *hash table*.

```
typedef struct lNode{
     struct lNode *next; // ponteiro de ligação
     char *name;       // string, aloj. dinâmico – nome representado
     RefArray *refArr; // referencias associadas
} LNode;
```

Cada nó destas listas representa um dos nomes existentes, excluindo as respetivas terminações, quando identificadas, e associa-o ao subconjunto dos descritores de ficheiro que têm esse nome.

O campo `next` é o ponteiro de ligação na lista. O campo `name` indica a *string*, alojada dinamicamente, com o nome representado. O campo `refArr` indica um *array* dinâmico de referências, idêntico aos definidos na série anterior, contendo o subconjunto das referências associadas ao nome indicado por `name`.

O tipo `HTable` seguinte representa o descritor de uma *hash table*.

```
typedef struct {
     int size;       // dimensão da tabela
     LNode **table;  // array de ponteiros alojado dinamicamente
} HTable;
```

O campo `size` é o número de elementos da tabela, configurável na criação. O campo `table` indica o *array* de ponteiros, alojado dinamicamente, cujos elementos apontam para as listas ligadas que representam os nomes e os respetivos subconjuntos de referências.

A indexação do *array* `table` deve feita com uma função de *hash* que tenha como argumentos o descritor da *hash table*, para identificar a dimensão, e a *string* com o nome a inserir ou a procurar.

4. Organização em módulos

Ao projeto anteriormente realizado deve adicionar, pelo menos, os seguintes módulos:

- Gestão da árvore de pesquisa binária;
- Gestão da *hash table*.

Além dos novos módulos fonte (.c) deve escrever os *header files* respetivos (.h). Atualize o *makefile*.

4.1. Escreva o módulo de gestão da árvore binária de modo a dispor das funções de interface seguintes. Pode criar outras funções, se achar conveniente, como auxiliares das especificadas. As funções que forem utilizadas exclusivamente no interior do módulo devem ter o qualificador *static*.

```
void tAdd( TNode **rootPtr, char *termin, FileInfo *ref );
```

Adiciona à árvore uma referência associada à terminação indicada por `termin`; se a terminação não existir, é inserida. O parâmetro `rootPtr` é o endereço do ponteiro raiz da árvore, o qual poderá ser modificado pela função. O parâmetro `ref` indica o descritor de ficheiro que se pretende referenciar, associado à *string* `termin`.

A árvore é ordenada alfabeticamente, com menores ligados pelo campo `left`. A inserção é sempre realizada nas folhas, com o propósito de simplificar o algoritmo. Para criar *arrays* de referências e para lhes adicionar elementos, deve usar as funções de interface do respetivo módulo.

```
void tBalance( TNode **rootPtr );
```

Reorganiza a árvore de modo que fique balanceada. O propósito desta função é ser executada após as inserções, de modo a melhorar a eficiência das pesquisas, já que a inserção simples nas folhas pode produzir árvores desbalanceadas.

```
RefArray *tSearch( TNode *root, char *termin);
```

Procura, na árvore identificada pela raiz `root`, a terminação indicada por `termin`. Retorna o endereço do *array* com o respetivo subconjunto de referências. No caso de a terminação não existir, retorna NULL.

```
void tFree( TNode *root );
```

Liberta o espaço de alojamento dinâmico ocupado pelos nós da árvore e por outros elementos que deles dependam.

4.2. Escreva o módulo de gestão da *hash table* de modo a dispor das funções de interface seguintes. Pode criar outras funções, se achar conveniente, como auxiliares das especificadas. As funções que forem utilizadas exclusivamente no interior do módulo devem ter o qualificador *static*. Espera-se nomeadamente que escreva funções internas para implementar a função de *hash* e a manipulação das listas ligadas.

```
HTable *hCreate( int size );
```

Cria o descritor para uma *hash table*, com alojamento dinâmico, e retorna o seu endereço. O parâmetro `size` representa o número de elementos atribuídos à tabela. Este valor, além de dimensionar o alojamento do *array* de ponteiros, deve ser registado para utilizar na função de *hash*, com vista ao cálculo do índice em cada acesso à tabela.

```
void hAdd( HTable *ht, char *name, FileInfo *ref );
```

Adiciona à *hash table* uma referência associada ao nome indicado por `name`; se este não existir, é inserido. O parâmetro `ht` indica o descritor da *hash table*. O parâmetro `ref` indica o descritor de ficheiro que se pretende referenciar, associado à *string* `name`. Para criar *arrays* de referências e para lhes adicionar elementos, deve usar as funções de interface do respetivo módulo.

```
RefArray *hSearch( HTable *ht, char * name);
```

Procura, na *hash table* indicada por `ht`, o nome indicado por `name`. Retorna o endereço do *array* com o respetivo subconjunto de referências. No caso de o nome não existir, retorna NULL.

```
void hFree( HTable *ht );
```

Liberta o espaço de alojamento dinâmico ocupado pela *hash table* e por outros elementos que dela dependam.

4.3. Adicione ao módulo aplicação o código necessário para a nova implementação dos comandos “t” e “s”.

É necessária a intervenção em duas fases:

- Após o preenchimento dos *arrays* de referências que representam o conjunto completo e a ordenação de um deles (`sortRef`), deve criar a árvore binária e a *hash table*;
- Ao interpretar os comandos, deve usar a árvore binária e a *hash table* para responder aos comandos “t” e “s”, respetivamente.

Para preencher as novas estruturas de dados, deve percorrer o *array* `sortRef` com a função `refArrScan`, usando uma função auxiliar no parâmetro `act` e passando o acesso à nova estrutura de dados no parâmetro `actParam`. Note que ao percorrer sequencialmente, vai adicionando elementos aos *arrays* que representam os subconjuntos, pelo que nestes as referências ficam com a mesma ordem relativa que têm no *array* que representa o conjunto completo.

No caso da árvore binária, a função auxiliar passada em `act`, ao tratar cada um dos elementos existentes, deve usar `tAdd` passando a terminação e a referência. O parâmetro `actParam` serve para passar o endereço do ponteiro raiz da árvore binária. Após a construção da árvore binária, recomenda-se o seu balanceamento.

No caso da *hash table*, a função auxiliar passada em `act`, ao tratar cada um dos elementos existentes, deve usar `hAdd` passando o nome (excluída a terminação) e a referência. O parâmetro `actParam` serve para passar o endereço do descritor da *hash table*.

Na fase de interpretação dos comandos, deve usar as funções *Search* das novas estruturas de dados para obter o acesso aos *arrays* com os subconjuntos de referências pretendidos; em seguida deve usar `refArrScan` para apresentar todos os elementos do subconjunto selecionado.

4.4. Realize o teste completo do código.

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, sendo este realizado através da função `tBalance`, no final. O código proposto abaixo considera o nó de árvore com o tipo `TNode` e 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 da função `treeToSortedList`. Nesta função, r é o ponteiro raiz da árvore a transformar; `link` é o endereço da parte a ligar à direita da árvore transformada, sendo NULL na chamada inicial; o valor de retorno é a raiz da árvore degenerada em lista.

```
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. Transformar a lista novamente numa árvore, agora balanceada, aplicando o algoritmo da função `sortedListToBalancedTree`. Nesta função, `listRoot` é o endereço do ponteiro raiz da árvore degenerada em lista; `n` é o número de nós existente; o retorno é a raiz da árvore reconstruída na forma balanceada. Usualmente recorre-se a uma função auxiliar para calcular o número de nós existente.

```
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;
}
```

ISEL, 2021.06.30