Instituto Superior de Engenharia de Lisboa

LEETC

Programação II

2021/22 – 1.º semestre letivo

Quarta Série de Exercícios

Pretende-se uma nova implementação do programa especificado na terceira série de exercícios, com as modificações seguintes:

- Exploração de novas estruturas de dados dinâmicas – árvore binária de pesquisa e lista ligada;
- Adição de funcionalidade – registar os números das linhas em que ocorre cada palavra.

A árvore binária de pesquisa é usada para registar as palavras do texto, em substituição do *array* dinâmico usado anteriormente. O objetivo principal é exercitar e demostrar o funcionamento desta estrutura, mas tem também a vantagem de ser mais flexível, já que evita necessidade de alojar um grande bloco de memória contígua (para o *array* dinâmico) e de o redimensionar durante o registo das palavras.

A utilização de listas ligadas destina-se a associar a cada palavra a lista das linhas em que ela se encontra no texto original. Para este registo, considera-se linha uma sequência de carateres terminada pelo símbolo ASCII de mudança de linha '\n'. Deve considerar a primeira linha com o número 1. Se uma palavra ocorrer mais do que uma vez na mesma linha, deve ficar registada apenas uma ocorrência nessa linha.

Tal como especificado nas séries de exercícios anteriores, o programa realiza a contagem – e agora também o registo de linha – das ocorrências de palavras num ficheiro de texto codificados em UTF-8, armazenando a respetiva informação e suportando comandos de listagem e pesquisa por palavra ou por quantidade relativa de ocorrências. Tem igualmente a operação em duas fases, processamento das palavras e interação com o utilizador. Reproduz-se a especificação dos comandos, com alteração do comando "`w`" para reproduzir a lista de ocorrências das palavras:

- `a` (*alphabetic*) Apresenta a lista de todas as palavras, por ordem alfabética crescente, e a respetiva frequência;
- `w` *`word`* (*word frequency and occurrences*) Apresenta a frequência da palavra indicada por *`word`*, seguida da lista das linhas do texto onde esta ocorre; no caso de a palavra não existir, apresenta a informação de insucesso;
- `+` *`number`* (*most frequent words*) Apresenta um subconjunto das palavras mais frequentes, por ordem decrescente de frequência; o parâmetro *`number`* indica a quantidade de palavras;
- `-` *`number`* (*least frequent words*) Apresenta um subconjunto das palavras menos frequentes, por ordem crescente de frequência; o parâmetro *`number`* indica a quantidade de palavras;
- `q` (*quit*) Termina.

1. Novos módulos

Propõe-se que desenvolva dois novos módulos para implementar a árvore binária e as listas ligadas.

O módulo de listas ligadas deve disponibilizar, pelo menos, funções para:

Iniciar o descritor de lista no estado vazio;

Adicionar cada ocorrência de uma palavra, verificando se já existe, de modo a registar apenas uma ocorrência em cada linha;

Realizar o varrimento completo da lista, para reproduzir as ocorrências de uma palavra;

Destruir a lista, libertando a memória ocupada.

O módulo de árvore binária deve disponibilizar, pelo menos, funções para:

Iniciar a raiz da árvore no estado vazio;

Adicionar cada palavra, fazendo a respetiva contagem, e associando as linhas de ocorrência;

Balancear a árvore, com vista a melhorar a eficiência das pesquisas;

Realizar a pesquisa por palavra, para permitir o acesso à sua frequência e ocorrências;

Realizar o varrimento completo da árvore, por exemplo para reproduzir a lista alfabética das palavras ou para aplicar algum processamento a todas as palavras;
Destruir a árvore, libertando a memória ocupada.

Dos módulos anteriormente desenvolvidos, apenas devem sofrer alterações aqueles em que isso seja necessário para interagir com os novos módulos. Nomeadamente: o módulo de aplicação necessitará de alterações para indicar os números das linhas onde ocorre cada palavra; o módulo de gestão de dados passa a utilizar os serviços do módulo de árvore binária, em substituição de algumas das suas estruturas internas.

2. Desenvolvimento

2.1. Módulo de listas ligadas

Defina os tipos para este módulo: tipo `List`, descritores de acesso às listas; tipo `LNode`, representa os respetivos elementos.

```
typedef struct lNode{     // Nó de lista ligada
    struct lNode *next;   //   ligação na lista
    int num;              //   número de linha da ocorrência
} LNode;

typedef struct{           // Descritor de uma lista ligada
    LNode *head, *tail;   //   ponteiros para o início e fim da lista
} List;
```

As estruturas do tipo proposto para o descritor de lista dispõem de ponteiros para o início e fim, de modo a facilitar a inserção à cauda. Os alunos devem ter em conta que o texto é lido sequencialmente, portanto com número de linha crescente, e explorar esta estrutura de dados de modo que os números de linha sejam colocados por ordem crescente, de forma eficiente. Recordamos que em caso de várias ocorrências na mesma linha apenas uma é registada.

Escreva o código do módulo e o respetivo *header file*, contendo as declarações necessárias para a sua utilização. Deve conter, nomeadamente, as assinaturas das funções de interface.

Verifique o funcionamento do módulo, criando uma aplicação de teste e o correspondente *makefile*.

2.2. Alterações ao descritor de palavra

Redefina o tipo `WordFreq`, adicionando o campo `occur`.

```
typedef struct{ // Descritor de uma palavra
    char *word; //   aponta palavra armazenada em memória alojada dinamicamente
    int freq;   //   número de ocorrências
    List occur; //   descritor da lista com o os números de linha onde há ocorrência
} WordFreq;
```

Este descritor mantém os campos especificados na série anterior, sendo adicionado um novo campo, descritor da lista associada à palavra para registar as respetivas ocorrências. Note-se que a quantidade de elementos da lista pode ser inferior ao valor do campo `freq`, se houver linhas com mais do que uma ocorrência.

2.3. Módulo de árvore binária de pesquisa

Defina o tipo `TNode` que representa os nós para construção da árvore.

```
typedef struct tNode{                   // Nó de árvore binária de pesquisa
    struct tNode *left, *right;         //  ligações às subárvores
    WordFreq *ref;                      //  acesso ao descritor de uma palavra
} TNode;
```

O objetivo principal dos nós de árvore é representar as palavras, dispondo para isso do campo ponteiro `ref` (*reference*) que dá acesso ao descritor de uma palavra. Este descritor é criado quando se cria o respetivo nó da árvore, mas é alojado em separado com o propósito de ter alguma independência. Por exemplo, os descritores de palavra podem ser referenciados também por outras estruturas de dados, nomeadamente para aceso organizado por frequência.

Para a construção da árvore, propõe-se o algoritmo simplificado de inserção nas folhas, o qual apresenta como inconveniente a possibilidade de causar desbalanceamento. Este é um aspeto sensível para a utilização de uma árvore binária, dado que a melhor eficiência no acesso é conseguida se a árvore estiver balanceada. Por isso, deve ser criada uma função de balanceamento que seja executada, pelo menos, no final da inserção das palavras, de modo que as pesquisas sejam eficientes. O balanceamento também pode ser realizado a intervalos de várias inserções na árvore, para melhorar a eficiência da própria construção; não se recomenda balancear a cada inserção porque o balanceamento também consome tempo, o que pode não compensar. Em anexo é proposto um algoritmo para balancear a árvore.

As opções quanto aos parâmetros da maioria das funções, e as respetivas assinaturas, devem ser definidos pelos alunos. No entanto especifica-se, concretamente, a função destinada ao varrimento completo da árvore, de modo que ela possa servir diversos propósitos, recebendo uma função através de parâmetro.

```
int tScan( TNode *root, int(*action)(WordFreq *wf, void *context),
           void *context );
```

A função `tScan` deve percorrer toda a árvore, por ordem alfabética, e chamar, em cada nó, a função `action`, passando-lhe a palavra associada ao nó e o parâmetro `context` que recebeu. O valor de retorno é o somatório dos valores retornados pela função `action`.

Esta função pode servir, por exemplo, para:
- Contar as palavras, recebendo em `action` uma função que retorna sempre 1.
- Apresentar as palavras em *standard output*, recebendo em `action` uma função que apresenta os dados de uma palavra;
- Aplicar um processamento a todas as palavras, recebendo em `action` a função específica de processamento e em `context` o acesso a dados acessórios para esse processamento.

Escreva o código do módulo e o respetivo *header file*, contendo as declarações necessárias para a sua utilização. Deve conter, nomeadamente, as assinaturas das funções de interface.

Verifique o funcionamento do módulo, criando uma aplicação de teste e o correspondente *makefile*.

2.4. Alterações ao módulo de gestão de dados

Nesta versão do projeto é usada a árvore binária de pesquisa para registar as palavras por ordem alfabética, bem como as respetivas frequências e ocorrências, para suportar os comandos "`a`" e "`w`". A árvore, assim, usada em vez do primeiro *array* dinâmico de ponteiros, que era ordenado alfabeticamente, pelo que este não deve ser criado.

Mantém-se o *array* dinâmico de ponteiros ordenado por frequência, usado para suportar os comandos "+" e "−". Na atual especificação do projeto, propõe-se que este *array* seja criado somente após a construção total da árvore binária de pesquisa, com o conjunto das palavras. Para isso, é necessário

realizar um varrimento completo da árvore, o qual, no processamento de cada nó, adiciona ao *array* o ponteiro para o descritor da palava associada ao nó da árvore. Deve utilizar a função `tScan`, especificada no ponto anterior, passando-lhe: em `action`, uma função capaz de adicionar uma palavra ao *array* dinâmico de ponteiros; em `context`, o aceso ao descritor do *array* de ponteiros. Propõe-se também que o *array* seja alojado, à partida, com a dimensão necessária, para evitar o redimensionamento.

Realize as modificações ao código do módulo. Além das diferenças na implementação, tenha em conta a necessidade de receber, como parâmetro, os números de linha em que as palavras ocorrem, com vista ao respetivo registo. Atualize o *header file* associado ao módulo.

Verifique o funcionamento do módulo, criando uma aplicação de teste e o correspondente *makefile*.

2.5. Atualize a implementação do módulo de aplicação final

Realize as modificações necessárias para adequar a aplicação final à especificação atual e às alterações de interface do módulo de gestão de dados.

Atualize o *makefile* e teste o programa completo.

Anexo – Balanceamento da árvore binária

Com vista uma utilização eficiente, as árvores binárias de pesquisa devem ser balanceadas. Propõe-se, para simplificar, que construa 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;
}
```