Instituto Superior de Engenharia de Lisboa

LEETC

Programação II

2021/22 – 1.º semestre letivo

Segunda Série de Exercícios

Nesta série de exercícios especifica-se o desenvolvimento de um programa para processamento de palavras a partir de ficheiros de texto codificados em UTF-8. A funcionalidade principal é identificar a repetição de palavras e a respetiva frequência (número de ocorrências). Para isso, o programa deve isolar sequencialmente as palavras e comparar cada uma com as anteriores, de modo a incrementar a frequência, se já existe, ou registar a palavra, se ainda não existe. Com vista a comparar corretamente as palavras, é necessário uniformizar a sua representação e descartar sinais de pontuação ou outros símbolos que não façam parte das mesmas.

1. Funções para isolamento e uniformização das palavras

Nesta secção são especificadas as funções básicas para separar as palavras existentes numa linha de texto e aplicar a normalização que permita compará-las com outras, já armazenadas numa estrutura de dados.

Além das funções indicadas, pode escrever outras que considere convenientes para o funcionamento daquelas.

1.1. Escreva as funções

```
char *lineSplitFirst( char line[] );
char *lineSplitNext( void );
```

A função `lineSplitFirst` recebe em `line` a *string* com uma linha de texto e inicia a separação das palavras que ela contém; identifica a primeira palavra, coloca um terminador de *string* após a mesma e retorna o endereço onde ela se inicia; se não houver palavras, retorna NULL.

A função `lineSplitNext` continua a separação das palavras; identifica a próxima palavra, coloca um terminador de *string* após a mesma e retorna o endereço onde ela se inicia; se não houver mais palavras, retorna NULL.

Para efeitos de separação de palavras, devem ser considerados os separadores normais (espaço, tabulação ou mudança de linha) e os sinais de pontuação, de modo que estes sejam descartados.

Propõe-se que utilize a função `strtok` da biblioteca normalizada, passando no seu segundo argumento uma *string* com o conjunto de delimitadores referido. Note que nesta *string* não é válida a utilização de símbolos compostos UTF-8.

1.2. Escreva a função

```
int utf8SetLower( char *symb );
```

que coloca em minúscula o símbolo iniciado no endereço indicado por `symb`, o qual pode ser do subconjunto ASCII básico ou um símbolo composto UTF-8, iniciado por 0xc3. Nesta gama, o segundo byte do símbolo representa uma maiúscula se for inferior a 0xa0; caso contrário, representa minúscula; a diferença entra as duas representações da mesma letra é 0x20.

No caso de o símbolo ser composto e não pertencer à gama 0xc3, deve permanecer inalterado.

A função retorna o número de bytes do símbolo, podendo ser: 1, se for ASCII; maior que 1, se for composto.

1.3. Escreva a função

```
char *normalize( char *word );
```

que normaliza a palavra passada no parâmetro `word`, aplicando os seguintes critérios:

- Se existirem, no início ou no final da *string*, carateres que não algarismos nem letras, devem ser descartados. Nomeadamente, pretende-se eliminar a ocorrência de hífen ("-") antes de uma palavra ou aspas a envolvê-la. Propõe-se que filtre, nas extremidades da palavra, o hífen e quaisquer símbolos compostos UTF-8 não pertencentes à gama 0xc3;
- Se existir hífen no interior da palavra, a ligar partes da mesma, deve ser mantido;
- Os símbolos de letras permanecem na palavra e devem ser passados para minúsculas.

Para descartar carateres: no início da *string*, podem ser ignorados avançando o ponteiro para a primeira letra; no final, devem ser excluídos inserindo um terminador de *string*. A função `normalize` retorna o endereço do início efetivo da palavra normalizada.

2. Programa de aplicação

Escreva e teste um programa de aplicação, designado por `wfr` (*word frequency report*) para identificar a frequência das palavras em ficheiros de texto codificados em UTF-8. O ficheiro de texto é indicado como argumento de linha de comando.

2.1. Fases da atividade do programa

A atividade do programa decorre em duas fases: (1) processamento das palavras; (2) interação com o utilizador.

Na fase de processamento das palavras, o programa lê o ficheiro de texto, isola as palavras, normaliza-as e armazena-as numa estrutura de dados preparada para registar as respetivas frequências. Quando é processada uma palavra, se já estiver registada, a sua frequência é incrementada; se não, a palavra é adicionada ao registo. Quando terminar esta fase, devem ficar disponíveis os dados das palavras em forma conveniente para realizar a fase seguinte. Nomeadamente, devem ser criados acessos ordenados com dois critérios: alfabeticamente e por frequência.

Na fase de interação, o programa permanece em ciclo, recebendo do utilizador os comandos seguintes:

- `a` (*alphabetic*) Apresenta a lista de todas as palavras, por ordem alfabética crescente, e a respetiva frequência;
- `w` *`word`* (*word frequency*) Apresenta a frequência da palavra indicada por *`word`*; no caso de 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.

2.2. Implementação do programa

Escreva o programa `wfr`. Propõe-se que faça o processamento linha a linha, usando a função `fgets` de biblioteca, para ler o texto de entrada, e as funções indicadas na secção anterior; pode criar outras que considere convenientes.

Para o alojamento da linha, na leitura, propõe-se que crie um *array* de `char` com dimensão fixa, superior à da maior linha. Sugere-se que use o comando "`wc`" (*word count*), usando a opção "`-L`" para obter a dimensão da maior linha dos ficheiros que irá utilizar no ensaio.

Quanto à estrutura de dados principal para armazenar as palavras, sugere-se, como simplificação, que crie um *array* de dimensão fixa, com elementos do tipo `WordFreq`. Note que, se os ficheiros a processar tiverem muitas palavras, poderá necessitar de definir este *array* com uma dimensão muito elevada; por isso, recomenda-se que o defina como variável global.

```
typedef struct{
    char word[MAX_WORD_SIZE];
    int freq;
} WordFreq;
```

No dimensionamento do campo `word` e do próprio *array* principal, deve ter em conta o tamanho e a quantidade das palavras existentes nos ficheiros que pretende ensaiar. Pode utilizar o comando "`wc`" para obter a contagem de palavras num ficheiro.

Inicialmente o *array* principal está vazio, sendo sucessivamente preenchido quando são encontradas palavras que ainda não estejam armazenadas. Quando processar cada palavra do texto, deve verificar se ela já existe: se sim, incrementa a frequência; se não, adiciona a palavra ao *array*.

Para implementação do preenchimento sugere-se, como abordagem simplificada, adicionar a palavra sempre no final do *array*, portanto sem ordenação, pelo que a pesquisa necessitará de ser exaustiva. (Adiante é proposto um refinamento, para manter o *array* sempre ordenado e poder usar pesquisa dicotómica, mais eficiente.)

No final do preenchimento, este *array* deve ser ordenado alfabeticamente, de modo a permitir a execução do comando "`a`" e facilitar a execução eficiente do comando "`w`". Pretende-se que utilize as funções `qsort` e `bsearch` de biblioteca para, respetivamente, ordenar o *array* e implementar a pesquisa do comando "`w`".

Com vista à execução dos comandos "+" e "-", é necessário dispor de um acesso ordenado pela frequência. No entanto, não deve perder a ordenação alfabética do *array* principal. Por isso, propõe-se a criação de um novo *array*, auxiliar, cujos elementos são de ponteiros para `WordFreq`. Estes são inicialmente apontados para os elementos do *array* principal e depois são ordenados com o critério pretendido. Para isso, deve utilizar a função `qsort` de biblioteca.

Para testar o programa, os alunos devem criar ou adaptar ficheiros de texto com diversas quantidades de palavras diferentes e das respetivas repetições.

Com o propósito de promover o ensaio do programa com muitas palavras, é disponibilizado em anexo o texto integral do romance "A Capital" de Eça de Queirós, obtido do repositório com as obras do mesmo autor, no endereço: http://figaro.fis.uc.pt/queiros/lista_obras.html.

3. Exercício opcional – Melhoria de eficiência no preenchimento do *array* principal

O método de preenchimento sugerido anteriormente tem eficiência reduzida, devido a não se poder utilizar pesquisa dicotómica, inviável pela inexistência de ordenação na fase de preenchimento.

Propõe-se nesta secção, como exercício opcional, a implementação da inserção de modo a manter o *array* principal sempre ordenado alfabeticamente. Assim, para verificar se a palavra existe pode utilizar pesquisa dicotómica. Sugere-se o ensaio de dois métodos: (1) Usar `bsearch` para pesquisar e `qsort` para aplicar a ordenação, após cada inserção; (2) Escrever código vocacionado para pesquisa binária e inserção ordenada no *array*, sem reordenar integralmente.

Note que, com ambos os métodos, o *array* principal permanece ordenado alfabeticamente, pelo que é desnecessário aplicar a ordenação após o preenchimento.

3.1. Utilização de `bsearch` e `qsort`

O primeiro método é simples de implementar com as funções de biblioteca, mas pode ter alguma perda de eficiência devido à reordenação integral do *array* a cada inserção.

3.2. Desenvolvimento de inserção ordenada

Para o segundo método, é necessário escrever código alternativo, mas tem a possibilidade de ser mais eficiente. Uma forma viável é criar uma função de pesquisa binária que, contrariamente à `bsearch`, permita conhecer o local onde a palavra deve ser inserida, quando ainda não existe. Isto permite deslocar o resto do *array* principal, criando espaço para inserir o novo elemento na posição ordenada. O deslocamento do conteúdo pode se realizado com a função `memmove` da biblioteca.

3.3. Avaliação de desempenho

Para comparar o desempenho de diferentes implementações, pode executar o utilitário "`time`" do Linux, seguido da linha de comando que executa o programa a ensaiar. Sugere-se ainda que crie ficheiros de texto com comandos para o programa a ensaiar e que os utilize em redirecionamento de *standard input*, de modo a evitar diferenças de tempo motivadas pela interação através do teclado.

4. Anexo – Ordenação de *strings* com carateres acentuados

Para comparar palavras por ordem alfabética, respeitando a acentuação, a função `strcmp` não produz a ordem mais adequada, devido à distância entre os códigos numéricos usados na codificação das letras com os diversos acentos. Propõe-se o uso da função `strcoll` de biblioteca, que está preparada para reconhecer a ordem correta, atribuindo ordem adjacente às variantes de cada letra com os diversos acentos, e nas formas de minúscula ou maiúscula. Esta função necessita de definições relacionadas com a língua, selecionadas pela função `setlocale`.

De modo a usar a função `strcoll` é necessário assegurar as seguintes condições:

- Dispor do *locale* para a língua pretendida. Pode verificar as variantes existentes com o comando "`locale -a`" na linha de comando.

  O *locale* "`pt_PT.UTF-8`" representa as definições específicas de Portugal. Se não estiver instalado é necessário gerá-lo, executando o comando "`sudo locale-gen pt_PT.UTF-8`".

- No programa, incluir o *header file* "`locale.h`".

- No início da atividade, executar "`setlocale(LC_ALL,"pt_PT.UTF-8");`".

- Nas comparações, usar "`strcoll( string1, string 2 );`"

ISEL, 2021.11.16