Trabalho Prático 2
Analisador de Texto

Aluno: Enzo Roiz
Matrícula: 2010049017
Professores: Gisele e Clodoveu.

Descrição do problema

O analisador de texto, que desejamos implementar, deve analisar um arquivo de texto passado para o programa. Aspectos considerados na “correção” do arquivo para a análise serão: converter os acentos presentes no texto para o respectivo par sem acento, juntar palavras terminadas com hífen à palavra subsequente e remover os caracteres de pontuação do início, meio e fim das palavras. Isto feito, o analisador deve ainda imprimir as palavras corrigidas do texto segundo a ordem lexicográfica definida e o número de ocorrências desta palavra no arquivo.

Implementação

Estrutura de dados:

Para a implementação do trabalho foi utilizada o tipo abstrato de dados “listas encadeadas”, que são listas que contém células. Cada célula é encadeada com uma próxima, por meio de endereçamento. A célula final tem **NULL** como próxima célula. Dentro de cada célula, temos um tipo chamado de **TipoItem**, onde estão armazenadas informações necessárias como **palavra[MAXTAM]**, onde será armazenada uma palavra de tamanho máximo MAXTAM e **numVezes**, onde são registradas o número de vezes que a palavra ocorreu. Além disso a célula contém o endereço da próxima célula da lista.

Funções

void PegaArgumentos(int , char **, char **, char **):
Recebe como parâmetros argc: número de argumentos com os quais a função main foi chamada na linha de comando. **argv endereço da matriz de strings onde estão armazenados o nome do programa e os argumentos passados pela linha de comando. ** entrada: endereço da string com o nome do arquivo de entrada. **saída: endereço da string com o nome do arquivo de saída. Nesta função, entrada receberá o nome do arquivo de entrada saída receberá o nome do arquivo de saída, ambos passados pela linha de comando.

void FLVazia(TipoLista *);
Recebe como parâmetro um ponteiro para uma lista. Nesta função, é criada a lista. É feita uma célula cabeça para facilitar as operações na lista com os apontadores “Primeiro” e “Ultimo” apontando para ela própria.

int Vazia(TipoLista);
Recebe como parâmetro uma lista. Verifica se a lista está vazia e retorna non-zero caso esteja vazia e zero caso não esteja vazia.

void Insere(TipoItem, TipoLista *);
Recebe como parâmetros o item a ser inserido na lista e um ponteiro para uma lista. Cria uma célula nova na lista, inserindo nela o item passado por parâmetro na última posição da lista.

void Imprime(char *, TipoLista);
Recebe uma string contendo o nome do arquivo de saída onde serão impressos os resultados e a lista de palavras ordenadas. Esta função imprime as palavras em ordem e o número de ocorrências dela. No final do arquivo, a palavra “FIM”.

char * StrLower(char *);
Recebe como parâmetro uma palavra e converte as letras maiúsculas para letras minúsculas, retornando a palavra recebida na função com todas as letras minúsculas.

char ConverteAcento(char);
Recebe como parâmetro um determinado caractere e o converte para o seu par sem acento, retornando o caractere corrigido.

void Tokeniza(char *, char *);
Recebe como parâmetro a palavra a ser corrigida e uma string onde será armazenada a palavra após a correção. A palavra é então “normalizada”, corrigindo seus acentos para o par sem acento e removendo a pontuação não permitida pela especificação.

void TabelaASC(int *);
Recebe um vetor de inteiros de tamanho 256 (tamanho da tabela ASCII estendida), e o preenche com os valores de cada caracter segundo a ordem da tabela ASCII.

int StrCmpOrdem (const char *, const char *, int *);
Uma adaptação da função strcmp que recebe como parâmetro duas strings e retorna zero caso sejam iguais e non-zero, caso sejam diferentes. A adaptação da função strcmp para a StrCmpOrdem(), fica por conta da passagem da tabela ASCII, que define a nova ordem lexicográfica que será usada para comparar as palavras.

int LeArquivo(char *, int *, TipoLista *);
Recebe como parâmetros o nome do arquivo de entrada, a tabela ASCII, e uma lista. Nesta função, será lido o arquivo, e a partir das informações lidas será definida a nova ordem lexicográfica e serão inseridas na lista todas as palavras lidas no texto. A função retorna o número de palavras do texto.

void QuickSort(Apontador *, int *, int *);
void Ordena(int , int, Apontador *, int *);
void Particao(int, int, int *, int *, Apontador *, int *);
O algoritmo QuickSort foi escolhido para a implementação do programa por ser o algoritmo de ordenação interna mais rápido que se conhece. Todas as funções acima citadas fazem parte dele. Em todas estas funções são passados como parâmetro, um vetor com os apontadores para a lista de palavras e a tabela ASCII. O algoritmo do QuickSort é recursivo, chamando as próprias funções para a ordenação. Em Particao() ele divide o vetor de apontadores para as células da lista em vetores cada vez menores de acordo com os critérios para esta divisão. A função Ordena() é faz a chamada da função Particao() e avalia como será feita a sua própria chamada recursiva. Por fim o algoritmo QuickSort() recebe como parâmetros as posições inicial e final do vetor que se deseja ordenar e a tabela ASCII para poder fazer as comparações necessárias, e faz a primeira chamada da função Ordena().

int ConverteCharAsc(char *);
Recebe como parâmetro um caractere e retorna o valor inteiro, ou seja, o número correspondente a este caractere.

void OrdenaLista(Apontador *, TipoLista , int *, TipoLista *);
Recebe como parâmetros o vetor de apontadores para as células da lista, a lista contendo as palavras lidas, a tabela ASCII e a lista onde serão armazenadas as palavras já ordenadas. Nesta função é preenchido o vetor de apontadores para as células da lista de palavras, é chamada a função de ordenação QuickSort(), para ordenar o vetor de apontadores e consequentemente as palavras, as palavras são inseridas de forma ordenada na lista e contadas quantas vezes ocorrem.

Programa principal

Chama as funções PegaArgumentos(), para receber os argumentos passados por linha de comando, TabelaAsc(), para o preencher vetor com a tabela ASCII, LeArquivo() para ler o arquivo desejado. Armazena o numero de palavras lidas, retornado pela função, em numPalavras, declara o vetor de apontadores ponteiros[numPalavras], com o tamanho numPalavras, como visto. Chama as funções OrdenaLista(), ordenando as palavras lidas no arquivo e contando a ocorrência delas no texto, e por fim, a função Imprime() para escrever os resultados encontrados no arquivo de saída desejado. Para a resolução do problema de forma a lidar com um número de palavras consideravelmente grande, foi criado um vetor de apontadores para as células da lista que continham as palavras. Assim, para ordenar o vetor, foram comparadas as palavras apontadas utilizando o algoritmo QuickSort(). Assim os apontadores foram ordenados de forma que as palavras apontadas por eles se encontrassem de forma ordenada segundo a nova ordem lexicográfica.

Análise de complexidade

A análise de complexidade utilizará a notação O(Big-O) e será medida abordando o **número de palavras, "*n*"**.

void PegaArgumentos(int , char **, char **, char **);
void FLVazia(TipoLista *);
int Vazia(TipoLista);
void Insere(TipoItem, TipoLista *);
char * StrLower(char *);
char ConverteAcento(char);
void Tokeniza(char *, char *);
void TabelaASC(int *);
int StrCmpOrdem (const char *, const char *, int *);
int ConverteCharAsc(char *);
A complexidade das funções é **O(1)** já que estas funções independem do número de palavras contidas no arquivo texto. Insere() é **O(1)** pois estamos inserindo sempre na última posição da lista, não percorrendo-a. StrLower(), Tokeniza() e StrCmpOrdem() são **O(1),** pois independem do número de palavras e sim do número de caracteres de cada palavra, pois no pior caso percorrem a palavra toda. Mas como o tamanho máximo da palavra é MAXTAM, teremos as condições impostas nestas funções satisfeitas MAXTAM vezes, realizando algumas operações de complexidade O(1), o que nos dá O(MAXTAM) = **O(1).**

void Imprime(char *, TipoLista);
Nesta função é percorrida a lista de palavras ordenadas. No pior caso, não há repetição de palavras sendo a lista final igual a lista de palavras inicial. Portanto:

O(n).

int LeArquivo(char *, int *, TipoLista *);
Nesta função há um loop while que lê todas as palavras contidas no arquivo e as insere na lista. Então, a complexidade desta função é.

O(n).

void Particao(int, int, int *, int *, Apontador *, int *); / void Ordena(int , int, Apontador *, int *); / void QuickSort(Apontador *, int *, int *);
Como sabemos o algoritmo ser do tipo dividir para conquistar, no melhor caso, teremos então **O(n log n)**, mas no pior caso, onde as partições serão feitas sempre de modo a separar o vetor com 1 e (n-1) palavras, o algoritmo tem complexidade quadrática **O(n²)**, porém são muito raros os casos em que isto ocorre, e consideraremos para a análise o caso geral, então a complexidade considerada será:

O(n log n)

void OrdenaLista(Apontador *, TipoLista , int *, TipoLista *);
Nesta função, temos dois loops while, não aninhados, que percorrem, separadamente, todas as palavras da lista de palavras. Assim, teremos então:

O(n) + O(n) = 2O(n) = **O(n)**

Como a complexidade de um programa é determinada pela operação mais relevante, que contém maior complexidade dentre as outras, neste caso a ordenação utilizando QuickSort(), podemos verificar então que a complexidade do programa será:

O(max(O(1), O(n))) = O(n), O(max(O(n), O(n log n))) = **O(n log n)**

Testes

Para a verificação dos resultados do TP foram utilizados o teste disponibilizado no moodle pelo monitor da disciplina, testes disponibilizados pelos próprios alunos, também no moodle, e arquivos contendo Lorem Ipsum, conferindo manualmente a saída. Em todos casos a saída foi certa, passando em todos os vinte testes do Prático.

Conclusão

O trabalho foi interessante, aprimorou e fez que utilizasse na prática os meus conhecimentos adquiridos em sala sobre estruturas de dados, especialmente listas encadeadas. A implementação ocorreu em várias etapas. Na primeira implementação, estava lendo o arquivo caractere por caractere e tratando-o, ocasionando lentidão no programa. Na segunda, passei a utilizar fscanf(), lendo o arquivo por palavras, porém armazenando estas palavras em um vetor de palavras. Como teria de ser definido o número máximo de palavras, esta implementação limitou o trabalho e fez com que ele não passasse em alguns testes. Por fim a implementação que melhor solucionou o problema. Na qual, passei a armazenar os apontadores para as células da lista, assim, teria de mesmo modo acesso as palavras e poderia armazenar uma quantidade maior de palavras. A dificuldade maior encontrada foi achar a solução ótima para o problema, tendo que repensar e refazer diversas vezes a implementação.

Referências:

- Slides passados em sala de aula.
- Curso de Linguagem C online: http://www.mtm.ufsc.br/~azeredo/cursoC/
- Como implementar a função strlwr(): http://www.cprogramming.com/snippets/source-code/how-to-implement-strlwr-in-c
- Código da função strcmp(), utilizado como base para a implementação da função StrCmpOrdem(): http://compsci.ca/v3/viewtopic.php?t=24383
- Tabela ASCII http://www.asciitable.com/
- Gerador de Lorem Ipsum para testes. http://br.lipsum.com/