Relatório EP3 - MAC0121

João Gabriel Basi - N° USP: 9793801

1 O programa

O programa recebe, na linha de comando, um arquivo, um tipo de implemetação de lista ligada e uma ordem de impressão. O programa imprime na saída padrão as todas as palavras do arquivo e suas respectivas ocorrências, tanto em ordem alfabética, ou em ordem de ocorrência, dependendo do que for recebido na linha de comando.

2 As funções

2.1 No arquivo da main

- *show_usage*: Mostra uma mensagem de erro na saída de erro de acordo com o número fornecido;
- *executeOV*: Executa o programa utilizando uma tabela de símbolos ordenada e implementada com vetor;
- *executeUV*: Executa o programa utilizando uma tabela de símbolos desordenada e implementada com vetor;
- *executeOLL*: Executa o programa utilizando uma tabela de símbolos ordenada e implementada com lista ligada;
- *executeULL*: Executa o programa utilizando uma tabela de símbolos desordenada e implementada com lista ligada;
- *executeBST*: Executa o programa utilizando uma tabela de símbolos implementada com árvore de busca binária;

2.2 auxfuncs.h

Biblioteca com funções e uma struct que auxiliam o programa a manusear a memória:

- *InsertionResult*: Struct que auxilia no recebimento de dados sobre a inserção de um elemento na tabela de símbolos;
- *emalloc*: Aloca um espaço na memória, mostrando uma mensagem de erro se não houver espaço para a alocação;

- *estrdup*: Dulica uma string, retornando um ponteiro para a cópia e mostrando uma mensagem de erro se não houver espaço para a alocação;

2.3 buffer.h

Biblioteca que cria a struct Buffer e cria funções sobre ela:

- *Buffer*: Struct com um vetor de tamanho dinâmico, que auxilia no recebimento de strings da entrada;
- *BufferCreate*: Cria um buffer;
- *BufferDestroy*: Destroi um buffer;
- *BufferReset*: Reseta um buffer;
- *BufferPush*: Adiciona um caractere no fim do buffer;
- *readLine*: Lê uma linha da entrada e guarda no buffer.

2.4 vectorfuncs.h

Biblioteca com funções comuns entre as tabelas de símbolos implementadas com vetor:

- *Entry*: Struct que guarda uma chave e um valor associado à ela;
- *VectorSTable (VST)*: Struct que guarda uma tabela de símbolos com associações string-int por meio de um vetor de Entries;
- *VTableCreate*: Cria uma tabela de símbolos utilizando vetor;
- *VTableDestroy*: Destroi uma tabela de símbolos implemetada com vetor;
- *VTablePush*: Adiciona uma chave e um valor associado à ela no fim de uma tabela de símbolos feita com vetor;
- *valcompV*: Compara dois valores de uma tabela de símbolos implementda com vetor;
- *strcompV*: Compara duas chaves de uma tabela de símbolos implementda com vetor;
- *mergeSortV*: Organiza uma tabela de símbolos implementada com vetor utilizando a função fornecida.

2.5 linkedlistfuncs.h

Biblioteca com funções comuns entre as tabelas de símbolos implementadas com lista ligada:

- *LLNode*: Nó para lista ligada com uma chave, um valor associado à ela e um ponteiro para o próxomo nó;
- *LinkedListSTable (LLST)*: Cabeça para uma tabela de símbolos com associações string-int feita com uma lista ligada de LLNodes;
- *LLTableCreate*: Cria uma tablea de símbolos utilizando lista ligada;
- *LLTableDestroy*: Destroi uma tabela de símbolos que utiliza lista ligada.

2.6 tabelaSimbolo_VO.h

Biblioteca com funções sobre a tabela de símbolos implementada com vetor ordenado:

- *OVAdd*: Adiciona uma chave à uma tabela de símbolos ordenada e implementada com vetor;
- *OVPritVal*: Imprime os elementos de uma tabela de símbolos, ordenada e implementada com vetor, em ordem decrescente de valor;
- *OVPrintLexi*: Imprime os elementos de uma tabela de símbolos, ordenada e implementada com vetor, em ordem alfabética;

2.7 tabelaSimbolo_VD.h

Biblioteca com funções sobre a tabela de símbolos implementada com vetor desordenado;

- *UVAdd*: Adiciona uma chave à uma tabela de símbolos desordenada e implementada com vetor;
- *UVPrintVal*: Imprime os elementos de uma tabela de símbolos, desordenada e implementada com vetor, em ordem decrescente de valor;
- *UVPrintLexi*: Imprime os elementos de uma tabela de símbolos, desordenada e implementada com vetor, em ordem alfabética;

2.8 tabelaSimbolo_LO.h

Biblioteca com funções sobre a tabela de símbolos implementada com lista ligada ordenada;

- *OLLAdd*: Adiciona uma chave à uma tabela de símbolos ordenada e implementada com lista ligada;

- *OLLPrintVal*: Imprime os elementos de uma tabela de símbolos, ordenada e implementada com lista ligada, em ordem decrescente de valor;
- *OLLPrintLexi*: Imprime os elementos de uma tabela de símbolos, ordenada e implementada com lista ligada, em ordem alfabética;

2.9 tabelaSimbolo_LD.h

Biblioteca com funções sobre a tabela de símbolos implementada com lista ligada desordenada;

- *ULLAdd*: Adiciona uma chave à uma tabela de símbolos desordenada e implementada com lista ligada;
- *ULLPrintVal*: Imprime os elementos de uma tabela de símbolos, desordenada e implementada com lista ligada, em ordem decrescente de valor;
- *ULLPrintLexi*: Imprime os elementos de uma tabela de símbolos, desordenada e implementada com lista ligada, em ordem alfabética;

2.10 tabelaSimbolo_AB.h

Biblioteca com funções sobre a tabela de símbolos implementada com árvore de busca binária.

- *BTNode*: Nó para árvore binária com uma chave, um valor associado à ela, um ponteiro para o nó direito e um ponteiro para o nó esquerdo;
- *BinaryTreeSTable (BTST)*: Raiz para um tabela de símbolos com associações string-int feita com uma árvore binária de BTNodes;
- *BSTTableCreate*: Cria uma tabela de símbolos utilizando árvore binária;
- *BSTTableDestroy*: Destroi uma tabela de símbolos implemetada com árvore binária;
- *BSTAdd*: Adiciona uma chave à uma tabela de símbolos implementada com árvore de busca binária;
- *BSTPrintVal*: Imprime os elementos de uma tabela de símbolos, implementada com árvore de busca binária, em ordem decrescente de valor;
- *BSTPrintLexi*: Imprime os elementos de uma tabela de símbolos, implementada com árvore de busca binária, em ordem alfabética;

3 Análize dos algoritmos

3.1 Vetor desordenado

3.1.1 Inserção

A inserção na tabela foi feita de modo linear, comparando a chave a ser inserida com todas as que estão na tabela. Se for achada uma chave igual, a função para de comparar, caso contrário, o algoritmo insere a nova chave no final da lista.

No pior caso, temos que a palavra é inserida no fim da lista, fazendo n comparações.

No caso médio, temos que a média de comparações para uma inserção é:

$$E(x) = \sum_{i=1}^{n} \left(i \cdot \frac{1}{n} \right) = \frac{1}{n} \cdot \sum_{i=1}^{n} i \overset{\text{Soma de P.A.}}{\Longrightarrow} \frac{n(n+1)}{2n} = \frac{n+1}{2} \tag{1}$$

Já em se tratando de movimentação de elementos da lista, o algoritmo de inserção não movimenta nenhum elemento, já que os elementos novos são inseridos no final da lista.

3.1.2 Impressão

Para ambas as ordens de impressão, é preciso ordenar a lista. Para isso, o programa usa um merge sort, que faz $2n \log n$ movimentações e $n \log n$ comparações em todos os casos.

3.2 Vetor ordenado

3.2.1 Inserção

O programa faz uma busca binária para achar o lugar certo de inserção, então move todos os elementos, a partir desse lugar, uma posição à frente, para poder inserir a nova chave. Na busca binária, o algoritmo faz $\log n$ comparações, e colocando as chaves maiores para frente, se considerarmos que a probabilidade de uma palavra entrar em uma lugar da tabela é a mesma para qualquer posição, ele faz, em média, $\frac{n+1}{2}$ movimentações (a equação fica igual à equação (1) do item anterior) e no pior caso, a palavra é inserida no começo da lista, movendo os n elementos para frente.

3.2.2 Impressão

Como o vetor já está ordenado em ordem alfabética, para a impressão em ordem alfabética não são necessários comparações ou movimentações.

Já para a impressão em ordem de frequência, é preciso ordenar o vetor. Para isso foi utilizado um merge sort, que faz $2n \log n$ movimentações e $n \log n$ comparações.

3.3 Lista ligada desordenada

3.3.1 Inserção

O programa faz uma busca linear, que faz, em média, $\frac{n+1}{2}$ comparações por inserção se a chave estiver na lista (assim como calculado na equação (1) da inserção do vetor desordenado), e faz n comparações se ela não estiver; porém não faz nenhuma movimentação, já que o novo nó pode ser inserido trocando o ponteiro do nó anterior.

Se o algoritmo não achar uma chave igual na tabela, ele a insere no final da lista.

3.3.2 Impressão

Como a lista está desordenada, em ambos os casos programa passa as chaves da lista ligada para um vetor, fazendo n movimentações e n comparações, e ordena o vetor com um merge sort, que faz $2n \log n$ movimentações e $n \log n$ comparações, fazendo no total $n(1 + 2 \log n)$ movimentações e $n(1 + \log n)$ comparações.

3.4 Lista ligada ordenada

3.4.1 Inserção

O programa faz uma busca linear, que faz, em média, $\frac{n+1}{2}$ comparações (assim como calculado na equação (1) da inserção do vetor desordenado), porém, ele para ao achar uma chave maior ou igual à que será inserida e a insere nessa posição. O pior caso acontece se a nova chave for lexicograficamente maior que todas as da tabela, então o algoritmo faz n comparações (note que o pior caso aqui é mais difícil de acontecer que na lista ligada desordenada, já que aqui a palavra tem que ser lexicograficamente maior que todas as da tabela para ser inserida no final, diferente da desordenada que insere todas as palavras novas no fim da lista).

3.4.2 Impressão

Para a impressão em ordem alfabética, o programa só percorre a lista e imprime seus elementos, fazendo n comparações, já que ela já está ordenada por ordem alfabética. Já para a ordem de ocorrência, o programa faz o mesmo procedimento da impressão da lista ligada desordenada, resultando em $n(1 + 2 \log n)$ movimentações e $n(1 + \log n)$ comparações.

3.5 Árvore de busca binária

3.5.1 Inserção

O programa percorre a árvore fazendo, no melhor caso, $\log n$ comparações e, no pior caso, n comparações, porém a inserção nesse local é feita sem comparações ou movimentações, já que só é preciso mudar o ponteiro da folha em que será inserida o novo elemento.

O caso médio é mais complicadode se calcular, pois leva em conta a estrutura do texto utilizado, mas vamos dizer que é muito mais perto de $\log n$ do que n, ou seja $\log n + c$.

3.5.2 Impressão

Para a impressão em ordem alfabética, o programa utiliza um algoritmo recursivo que faz n comparações e nenhuma movimentação. Já para a impressão em ordem de ocorrência, o mesmo método da lista ligada desordenada é usado, resultando em $n(1 + 2\log n)$ movimentações e $n(1 + \log n)$ comparações.

4 Testes

Os testes foram feitos com uma versão do dicionário inglês de 1913 [1], em que os caracteres que não estão na tabela ASCII foram retirados, e com uma versão da bíblia em inglês também [2]. A versão ordenada da bíblia, citada na tabela, foi feita por mim, pegando as palavras da original e ordenando-as. Sendo assim, um arquivo com o mesmo número de palavras do original é gerado, porém as palavras estão organizadas em ordem alfabética. Esse caso é o pior caso de quase todas as tabelas, já que elas adicionam palavras novas e lexicograficamente maiores no final da lista, então isso obriga os algoritmos a checar todas as palavras da lista antes de adicioná-las. A outra versão, a ivertida, é o mesmo da versão ordenada, porém ordenada de "z"à "a", que seria o pior caso do vetor ordenado.

Inserção								
Implementação da tabela	Comparações		Movimentações		Tempos			
	médio	pior	médio	pior	Dicionário	Bíblia	ordenada	invertida
Vetor desordenado	$\frac{n+1}{2}$	n	0	0	273,37s	2,7s	21s	17s
Vetor ordenado	$\log n$	$\log n$	$\frac{n+1}{2}$	n	8,72s	0,2s	0,16s	0,23s
Lista ligada desordenada	$\frac{n+1}{2}$	n	0	0	427,5s	3,3s	27,2s	0,12s
Lista ligada ordenada	$\frac{n+1}{2}$	n	0	0	10900s (3h)	48s	27,5s	22,6s
Árvore de busca binária	$\log n + c$	n	0	0	1,78s	0,2s	32s	27,3s

Os tempos foram obtidos com a impressão em ordem alfabética. A impressão em ordem de ocorrência não mostrou uma diferença significante nos tempos, por isso não será citada. O tempo de impressão também não é citado pois também não mostrou muita diferença nos tempos.

Analizando os resultados, vemos que as maneiras mais eficientes são a árvore de busca binária e o vetor ordenado, e a mais ineficiente é a lista ligada ordenada. Nota-se também que o algoritmo dos dois tipos de lista ligada são bem similares, então eles fazem o mesmo tempo na bíblia ordenada.

Outro fato que se pode perceber olhando a tabela, é que a versão ordenada da lista ligada se saiu bem pior que a versão desordenada, o que vai contra a nossa intuição. Fazendo algumas observações, percebi que, na lista ligada desordenada, as palavras mais

comuns do vocabulário são adicionadas mais perto da cabeça da lista, dando uma vantagem sobre a lista ligada ordenada, já que palavras como "the"e "them", que são muito comuns no vocabulário inglês, ficam no final da versão ordenada, mas têm muita chance de aparecer no começo da desordenada.

Percebe-se também que quase todas as tabelas gastam todo o tempo comparando as chaves, a única em que isso não acontece é a implementada com vetor ordenado, que gasta a maior parte de seu tempo movimentando seu conteúdo para se manter ordenada. Porém essa troca ainda assim melhora sua performance, como é observado nos testes com a bíblia. Assim conseguimos ver que as comparações gastam muito mais tempo que as movimentações.

Referências

[1] Link para o dicionário `http://www.gutenberg.org/ebooks/29765`

[2] Link para a Bíblia `http://www.gutenberg.org/ebooks/10`