**Victor Pereira Lima** **NUSP 10737028**

# Relatório EP4

## *1 - O código*

Por uma questão de organização o meu código do EP4 foi dividido em 4 arquivos .C (e seus respectivos .h) : um *main.c*, que trata das questões mais centrais do código, e 3 “módulos” (*vetor.c*, *lista.c*, *arvore.c*) que realizam as operações básicas de cada estrutura específica e serão explicados abaixo.

### *1.a - vetor.c*

O “módulo” vetor.c apresenta duas funções de inserção:

- uma função de inserção desordenada com complexidade O(N), visto que no pior caso o vetor inteiro é percorrido procurando se aquela palavra já está inserida, e só então a palavra é inserida na última posição do vetor;
- uma função de inserção ordenada com complexidade O(N), visto que primeiro a função realiza x <= N comparações para verificar em que lugar do vetor a palavra será inserida, e depois realiza N - x deslocamentos para a direita nas casas a direita da posição da palavra inserida.

Duas funções de busca:

- uma função de busca “desordenada” com complexidade O(n), visto que no pior caso percorre o vetor inteiro para então retornar o resultado;
- uma função de busca binária com complexidade O(logN), visto que enquanto a palavra não é achada uma metade do vetor é descartada na busca.

Três funções que em conjunto realizam o algoritmo de ordenação quickSort:

- o algoritmo quickSort é implementado com o “separa do Sedgewick” e assim como em sua versão normal tem complexidade O(N * logN), visto que é virtualmente impossível que uma obra literária esteja inteiramente em ordem alfabética.

E por último uma função que imprime as palavras e suas ocorrências e remove os elementos do vetor, com complexidade exatamente O(N).

### *1.b - lista.c*

O “módulo” lista.c apresenta duas funções de inserção:

- uma função de inserção desordenada com complexidade O(N), visto que no pior caso a lista inteira é percorrida verificando se a palavra pertence a lista e só então a palavra é inserida no final da lista;
- uma função de inserção ordenada muito parecida com a desordenada, também com complexidade O(N). Uma vantagem da inserção ordenada em listas se comparada com a inserção ordenada em vetores é que não é preciso deslocar todos os elementos a direita da palavra inserida, mas sim somente mudar o valor de dois apontadores.

Uma função de busca com complexidade O(N), visto que no pior caso a lista inteira é percorrida para então retornar o apontador resultante.

Duas funções que em conjunto realizam o algoritmo mergeSort:

- o algoritmo mergeSort é implementado com uma função que intercala listas ordenadas vista em sala de aula em conjunto com o mergeSortList do Sedgewick. A vantagem desse algoritmo é que diferentemente da maioria

dos algoritmos de ordenação de listas, ele possui uma complexidade O(N * logN).

E por último uma função que imprime as palavras e suas ocorrências e limpa o espaço alocado para os itens da lista, com complexidade exatamente O(N).

*1.c - arvore.c*

O “módulo” arvore.c apresenta uma função de inserção:

- a função de inserção em árvore binária de busca tem complexidade O(altura). Porém devido a aleatoriedade da ordem lexicográfica das palavras que vão aparecendo ao decorrer de um texto literário, podemos afirmar que a altura da árvore vai se aproximar muito mais de logN do que de N. Logo podemos dizer que a inserção em ABB tem complexidade aproximada(logN).

Uma função de busca:

- a função de busca em ABB também tem complexidade O(altura), com pior caso O(N). Porém como visto acima, a altura da árvore provavelmente será algo parecido com logN, e portanto a busca também terá complexidade O(logN).

Uma função de impressão:

- a função de impressão utilizada no programa se utiliza do percurso inOrdem. Esse percurso é implementado de forma recursiva da forma visita subABB esq - visita Raiz - visita subABB dir, e tem a propriedade de imprimir a árvore exatamente em ordem alfabética, com complexidade O(N), visto que todos os nós são visitados.

E por fim uma função que deleta a ABB também com complexidade O(N).

*1.d - main.c*

Por fim temos o arquivo *main.c*, que junta os outros “módulos” para dar funcionamento ao programa. Primeiro os argumentos da linha de comando e o arquivo txt em questão são verificados. Caso sejam válidos, o programa entra em um loop que consiste de inserir cada palavra lida com a função fscanf e tratada com a strtok(que vai tratar a “expressão” obtida da fscanf, retirando os caracteres não alfanuméricos) na estrutura escolhida pelo usuário até que se chegue ao final do arquivo.

## *2 - Testes*

*2.a - Números*

Primeiro realizei testes em um texto com apenas 5 parágrafos e cerca de 200 palavras diferentes. Como o número era pequeno observei resultados semelhantes em todas estruturas, beirando os 0.001 s para a execução do programa todo.

Após isso, realizei os testes acerca de alguns capítulos do livro “O cortiço”, totalizando cerca de 6000 palavras diferentes e algo em torno de 30000 ocorrências. Os resultados obtidos estão apresentados na tabela abaixo:

| | **Inserção + Ordenação + Impressão** | **1500 Buscas** |
|---|---|---|
| **VD A** | 1.571 s | 2.016 s |
| **VD O** | 1.806 s | |
| **VO A** | 2.636 s | 1.254 s |
| **VO O** | 2.875 s | |
| **LD A** | 2.060 s | 2.808 s |
| **LD O** | 2.006 s | |
| **LO A** | 3.710 s | |
| **LO O** | 3.605 s | |
| **AB A** | 0.064 s | 0.046 s |
| **AB O** | 0.205 s | |

Para finalizar, fiz testes em cima do livro todo “Memórias póstumas de Brás Cubas”, com cerca de 11000 palavras diferentes e 70000 ocorrências. Os resultados obtidos estão na tabela abaixo:

| | **Inserção + Ordenação + Impressão** | **1500 Buscas** |
|---|---|---|
| **VD A** | 2.463 s | 0.469 s |
| **VD O** | 2.695 s | |
| **VO A** | 4.285 s | 0.004 s |
| **VO O** | 4.599 s | |
| **LD A** | 3.066 s | 0.765 s |
| **LD O** | 3.039 s | |
| **LO A** | 6.046 s | |
| **LO O** | 6.366 s | |

| | | |
|---|---|---|
| **AB A** | 0.085 s | 0.041 s |
| **AB O** | 0.285 s | |

*2.b - Conclusões*

- para uma amostra pequena, a estrutura não importa muito, logo acredito que a mais indicada seja aquela com que a pessoa seja mais familiar e tenha mais facilidade em usar  e implementar;
- no caso dos vetores, confesso que fiquei um pouco surpreso com a diferença no tempo de execução do programa com vetores desordenados em relação aos vetores ordenados, visto que a complexidade da inserção é a mesma nos dois casos, e os vetores desordenados ainda deveriam ser ordenados antes de serem impressos. Porém pensando bem isso se deve ao fato de que apesar da inserção ordenada ser O(N), a operação de deslocar os itens do vetor para a direita é bem mais custosa se compara com as simples comparações da inserção desordenada. Provavelmente essa é a causa da grande vantagem para os vetores desordenados.
- no caso das listas, elas foram provavelmente as “derrotadas” nesta tabela de símbolos. Apesar de gostar muito da estrutura delas, pela economia de espaço em relação aos vetores e pela maneira que o código escrito com listas é geralmente bem intuitivo, em questão de performance elas deixaram um pouco a desejar, provavelmente pela natureza quase sempre linear nas operações com listas. Apesar disso, as buscas em lista ocorreram de forma estranhamente rápida, sendo esse o único ponto positivo a se destacar na implementação da tabela com listas.
- por fim, as árvores binárias de busca, que na minha opinião foram as grandes vencedoras. Seja pelo código simples e de fácil manutenção (pode se dizer até bonito), seja principalmente pela questão perfomance. Elas foram em média 32 vezes mais rápidas do que as outras estruturas, isso levando se em conta as implementações mais rápidas das outras estruturas.
- olhando como um todo, acredito que os testes refletiram bem as complexidades esperadas das estruturas, visto que as operações com complexidade logarítmica começaram a se distanciar das operações com complexidade linear, comparando principalmente listas ligadas X árvores binárias.