UNIVERSIDADE FEDERAL DO ESPÍRITO SANTO
DEPARTAMENTO DE INFORMÁTICA

Elaine Dias Pires
Filipe Gomes Arante de Souza

2º Relatório de Estrutura de Dados II
Busca em Memória Principal

Fevereiro / 2022

Sumário

1 Introdução

O objetivo desse trabalho é realizar uma busca em texto a partir de keywords. Para realizar a busca, implementou-se uma busca binária e como método de ordenação, foi utilizado a função padrão do C, quicksort. Também foi pedida a comparação entre quicksort e outro método de ordenação. O segundo método escolhido foi o heapsort, que já havia sido aprendido e implementado no primeiro trabalho da disciplina. Dessa forma, nesse relatório, será explicado a implementação da busca realizada, a estrutura de dados utilizada, bem como uma comparação entre quicksort e heap para algumas entradas.

2 Tipos Abstratos de Dados (TADs)

2.1 str

A interface e implementação deste TAD foram fornecidos pela professora com o intuito de facilitar a manipulação de strings. No contexto do trabalho, duas funções tiveram grande importância:

- **int compare(String *s, String *t)**: Usada para auxiliar na criação do comparador de sufixos;
- **bool equals_substring(String *text, int from, int to, String *query)**: Usada para auxiliar na busca binária;

Foram feitas pequenas alterações em duas funções:

- **String* create_string(char *a)**: O caractere \0 foi colocado no final do campo *c*;
- **bool equals_substring(String *text, int from, int to, String *query**: Foi adaptada para não diferenciar caracteres maiúsculos de minúsculos com auxílio da função **tolower** da biblioteca ctype.h;

2.2 suffix

Nesse TAD a professora forneceu a interface e deixou que os alunos realizassem a implementação. As principais funções desse TAD foram:

- **int binary_search(Suffix** a, int N, String* query)**: Retorna o índice do primeiro elemento encontrado em que query está contida. Retorna -1 caso não encontre;
- **int search_first_query(Suffix** a, int N, String* query)**: Retorna o índice da primeira ocorrência do elemento em que query está contida. Retorna -1 caso não encontre;

- **sort_suf_array(Suffix* *a, int N)**: Ordena o vetor com o algoritmo de ordenação Quicksort;
- **heap_sort_suf_array(Suffix* *a, int N)**: Ordena o vetor com o algoritmo de ordenação Heapsort;

Foi feita uma pequena alteração nesse TAD, colocou-se a assinatura da struct no .h e a implementação no .c , inicialmente a implementação estava na interface do TAD.

2.3 io

Neste TAD foram implementados as funções que tratam da entrada do usuário na linha de comando, efetuam a leitura do arquivo passado e geram o output do trabalho. As principais funções desse TAD são:

- **String* read_txt(char* fileName)**: Lê o arquivo e retorna uma String com todo o texto presente nele;
- **void main_memory_search(int argc, char** argv)**: Executa alguma funcionalidade de acordo com a linha de comando;

3 Ordenação

Além das entradas fornecidas pela professora, adicionamos alguns outros arquivos .txt, de tamanhos variados, para analisar melhor o desempenho de ambos algoritmos de ordenação.

Segue abaixo tabela com os dados referentes a ordenação do vetor de sufixos:

Entrada	Tam. Entrada	Tempo Quicksort (s)	Tempo Heapsort (s)
lorem_ipsum1.txt	7.223	0,031	0,094
lorem_ipsum2.txt	32.204	0,234	0,562
lorem_ipsum3.txt	92.164	0,656	1,672
lorem_ipsum4.txt	100.328	0,703	1,906
tale.txt	726.570	6,062	16,438
moby.txt	1.191.463	10,438	28,797
bible.txt	4.332.454	45,531	127,391

Tabela 1: Tempos para ordenar vetores de sufixos de acordo com a entrada.

Conforme as aulas e o primeiro trabalho desta disciplina, sabemos que as complexidades dos casos médios do Quicksort e Heapsort são $O(n * lg(n))$. Contudo, mesmo tendo o mesmo comportamento assintótico, é possível notar que o Quicksort teve desempenho superior cerca de 2,5 a 3 vezes mais rápido que o Heapsort.

Segue abaixo gráfico para melhor visualização:

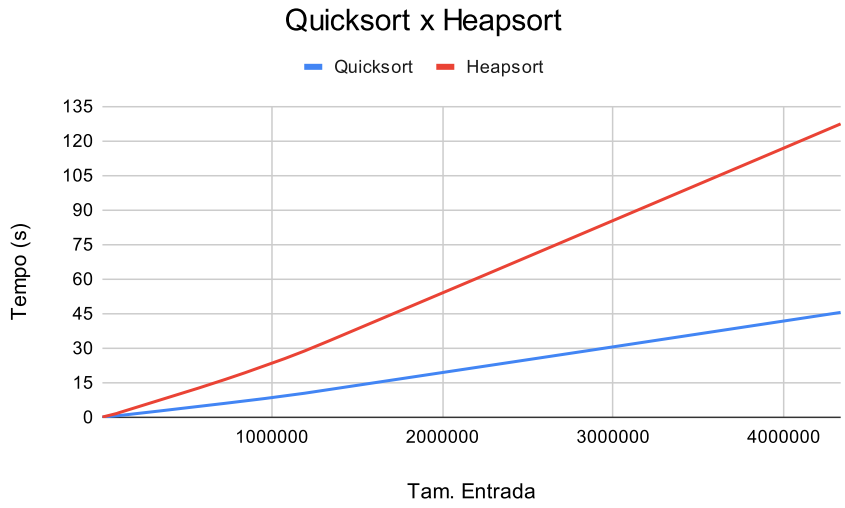

Figura 1: Comparação do desempenho entre os algoritmos Quicksort e Heapsort.

4 Busca Binária

A busca binária se baseia no princípio da Divisão e Conquista. Ou seja, dividir um problema em problemas menores e, neste caso, resolvê-los recursivamente até se ter a solução do problema total. Assim, dado um vetor *ordenado* e um elemento x que se queira encontrar no vetor, a busca binária divide o vetor ao meio e compara se x é igual ao elemento da posição central, se é menor ou maior. Se x for igual ao pivô, a função encerra. Se x for menor que o pivô, a função é chamada recursivamente para a parte esquerda e se x for maior, a função é chamada para a parte direita do vetor.

No contexto do trabalho, a comparação de ser maior, menor ou igual acontecia alfabeticamente por query ser uma string. Assim, a função **binary_search** chamava a função **partition** recursivamente até query ser encontrada. Uma vez que nosso vetor era um vetor de sufixos e não de queries, nós comparamos se query estava contida no sufixo da posição pivô, ou se estava à direita desse sufixo ou à esquerda.

A busca binária é um método de busca é muito eficiente, pois sua complexidade é $O(lg(n))$.

5 Conclusão

Neste trabalho foi feita uma busca em textos a partir de keywords. Pelo fato desse tipo de busca ser bastante utilizada em banco de dados, buscas na web, entre outras aplicações, a realização do trabalho foi bastante prazerosa por termos aprendido como essas buscas ocorrem. Ademais, foram utlizados dois métodos de ordenação, Quicksort e HeapSort. Pelo fato das entradas desse trabalho se assemelharem a entradas aleatórias, já sabíamos previamente que o Qucksort teria melhor desempenho, como de fato ocorreu.

6 Referências

6.1 Livros

- ZIVIANI, Nivio. **Projeto de Algoritmos com Implementações em Pascal e C**: 3ª edição revista e ampliada. 3. ed. Brasil: Cengage Learning, 2011.

6.2 Sites

- "C strcat() - C Standard Library. . Acessado 21 de fevereiro de 2022.