Relatório de Implementação
Trabalho 2 - Estrutura de Dados II
Juliana Camilo Repossi - 2020100631

1) Implementação:

Seguindo os passos para a realização desse trabalho, de forma inicial foi preciso limpar o texto de entrada, assim achei que faria mais sentido, do ponto de vista de TAD, fazer um novo .c e .h para isso, sendo ele o TAD texto. Esse tipo de dado contém apenas uma função que executa todo o passo 1 da descrição do problema, ou seja, faz a limpeza do texto descartando os duplos espaços e trocando as quebras de linhas por espaço, e por fim, aloca todo esse conteúdo em uma string, que é retornada pela função.

Depois já é criado o array de sufixos e as demais operações serão feitas sob demanda da opção de menu digitada. A opção “a” tem a saída esperada sem vazamentos de memória. A opção “o” também, vale lembrar que nessa opção foi utilizada a ordenação de sistema qsort, que foi implementada pela função “compara sufixos”. Essa função foi elaborada utilizando como função auxiliar a “compara_char” que atua de forma semelhante a “compare_from” como o proposto, com o “plus” de não fazer distinção de letras maiúsculas e minúsculas, no qual uma implementação mais simples baseada em tabela ascii não faria. Dessa forma, no momento de comparar duas substrings, é testado se pode estar sendo comparado uma letra maiúscula e minúscula ou vice-versa, em caso afirmativo é subtraído da minúscula 32, lógica da tabela ascii, para equiparar suas posições e apenas ordenar por ordem alfabética, nesse caso. Nos demais, a comparação simples é feita.

A seguir temos a opção “r” para medir o tempo de execução de dois ou mais algoritmos, sendo um deles o quickSort de sistema. Foi a parte mais interessante do projeto, pois escolhi para o segundo método o insertionSort, devido aos resultados do primeiro trabalho terem sido mais próximos do quick no quesito tempo, exceto no caso invertido. Porém, tive uma surpresa da diferença enorme de tempo nos testes com arquivos maiores como “moby” e ”tale” para executar essa parte, ficando extremamente inviável nessa ocasião utilizar um algoritmo simples de ordenação como o insertionSort. Penso que, a maior parte desse gargalo para esse método foi devido a alta quantidade de comparações que esse algoritmo faz, e devido às comparações, nesse caso, não serem tão imediatas e triviais, isso com certeza contribuiu para torná-lo ainda mais lento e aumentar consideravelmente a diferença entre os dois. É importante ressaltar que para entradas bem pequenas o insertion não apresenta diferença em relação ao quick.

Por fim, quanto às opções “c” e “s” que são semelhantes, vale a pena destacar o “up” dado na função rank, no qual, como o esperado, ela se aproveita do vetor de sufixos estar ordenado e procura até encontrar a primeira letra da query. Todavia há duas possibilidades a serem cogitadas:

- Achar a primeira letra, e começar a comparar as substrings que começam com aquela letra com a query e obter sucesso ou não;
- E, não encontrar nenhuma ocorrência sequer da letra;

Partindo desse ponto de vista é possível aprimorar essa função de forma que, assim que a execução saia da área possível de ser encontrada aquela primeira letra, ou seja, que passar da sua posição na ordem alfabética ou na ordem ASCII, sendo encontrada ou não ocorrências, não é mais necessário que o algoritmo continue a procurar aquela primeira letra, pois não há mais possibilidade de encontrá-la no vetor ordenado. Sendo assim, fazendo o tratamento adequado de letras maiúsculas e minúsculas com a “compara_char”, se a letra da vez do array for maior que a primeira letra da query já não é possível encontrar a string desejada e pode-se encerar a função, aumentando consideravelmente a eficiência da aplicação.

2) Testes de execução:

- Opção ‘a’: Foi impossível testar o tale.txt e o moby.txt nessa opção, pois o tempo de execução passou de 15 min. Logo, para esses casos testei com o abra.txt e um outro arquivo de teste.txt simples. E os resultados saíram como o esperado.
- Opção ‘o’: também neste caso, foi inviável testar os arquivos maiores disponibilizados, contudo os testes foram feitos com os arquivos abra.txt e teste.txt, e o resultado foi como o esperado, inclusive o “plus” proposto de não diferenciar letras maiúsculas de minúsculas.
- Opção ‘r’: Nessa opção fiz questão de esperar o tempo que fosse preciso para ter uma noção maior da diferença, então além de testar os arquivos pequenos já mencionados, também criei um de tamanho médio com a música “Photograph” do Ed Sheeran com alguns espaços e quebras de linhas a mais, resultando em 1717 caracteres. E os resultados foram os apresentados abaixo:

Como podemos ver nos exemplos, entradas muito pequenas podem mascarar, porém as médias começam a indicar disparidades de forma expressiva nos tempos de ordenação. Em ed.txt de 1717 caracteres, a diferença entre o quicksort e o insertionsort já é da ordem de 5 vezes. E, em entradas maiores, que é um caso comum nessa aplicação, a preocupação com um método eficiente de ordenação fica escancarada. O tale.txt e o moby.txt assustam com a diferença dos tempos entre métodos supracitados, variando de 1298 à 1620 vezes mais rápido usando o quick como método de ordenação.

- Opção ‘c’ e ‘s’: Por fim, essas opções produzem os resultados esperados e de forma veloz, assim como o proposto nos testes de especificação do trabalho.

3) Conclusão

A conclusão que tiramos deste projeto, de forma ainda mais clara que no último, é de como é importante escolher um método de busca e ordenação eficiente e condizente com o problema para utilizar em uma aplicação. Pois, uma má decisão de projeto nessa fase, negligenciando informações como tempo de execução, complexidade, número de comparações e até mesmo de trocas que seus candidatos a ordenadores fazem, podem te levar a tornar um projeto simples inviável pelo seu tempo de execução.