UNIVERSIDADE TECNOLÓGICA FEDERAL DO PARANÁ
CIÊNCIA DA COMPUTAÇÃO

FELIPE ARCHANJO DA CUNHA MENDES, 2252740
BRENO FARIAS DA SILVA, 2300516

TRABALHO - ALGORITMO E ESTRUTURA DE DADOS II

CAMPO MOURÃO
2021

Resumo

Com o objetivo de atender ao requisito de limitação do uso de memória, o qual é frequentemente imposto no cotidiano dos programadores, este projeto então tem como objetivo fazer a ordenação de um arquivo binários usando as seguintes técnicas: A primeira parte consiste na geração do arquivo binário. Em seguida, é preciso dividi-lo em n partes. Sendo isso, seguindo o diagrama proposto pelo professor, temos um arquivo original de 900mb, o qual foi dividido em 9 arquivos de 100mb cada. Esses arquivos são guardados no disco, porém vamos carregar um por um e ordená-los. Dessa forma, vamos carregando um arquivo por vez, ordenando-o e colocando de volta no disco. Ao final desse processo, temos todos os N arquivos ordenados internamente, mas não entre eles. Após esse processo, vamos carregando um registro por vez, de modo a colocá-lo em um buffer de saída, que será esvaziado após atingir a sua capacidade máxima de registros. Os registros nele contidos serão guardados em um arquivo final, o que estará totalmente ordenado. Por fim, vale ressaltar que devemos apagar os N arquivos gerados no começo.

Funções utilizadas

Para a melhor organização de nossos projetos, criamos alguns tads para encapsular funções específicas para determinada ação.

Ordenação.h

Este arquivo contém a implementação de funções de ordenação como, por exemplo, quickSort, partition, troca e merge_k_vias. Algumas dessas funções já foram estudadas previamente e estão sendo reaproveitadas, contudo o merge_k_vias é a novidade. Esta função trata da leitura dos buffers de entrada, de modo a procurar o menor elemento entre eles e então guardá-lo no buffer de saída. Esse processo persiste até o momento em que o buffer de saída estiver cheio, e nesse momento os seus dados serão guardados na memória. Porque não guardamos um registro por vez? Pois a leitura e escrita em disco geram lentidão pelo código, sendo assim o foco está em minimizar comunicação com o disco.

QuickSort, Partition & troca: Responsável por fazer a ordenação de vetores de item de venda
Merge_K_Vias: Responsável por pegar os elementos do buffer e colocar de forma ordenada no arquivo de saída.

Arquivo.h

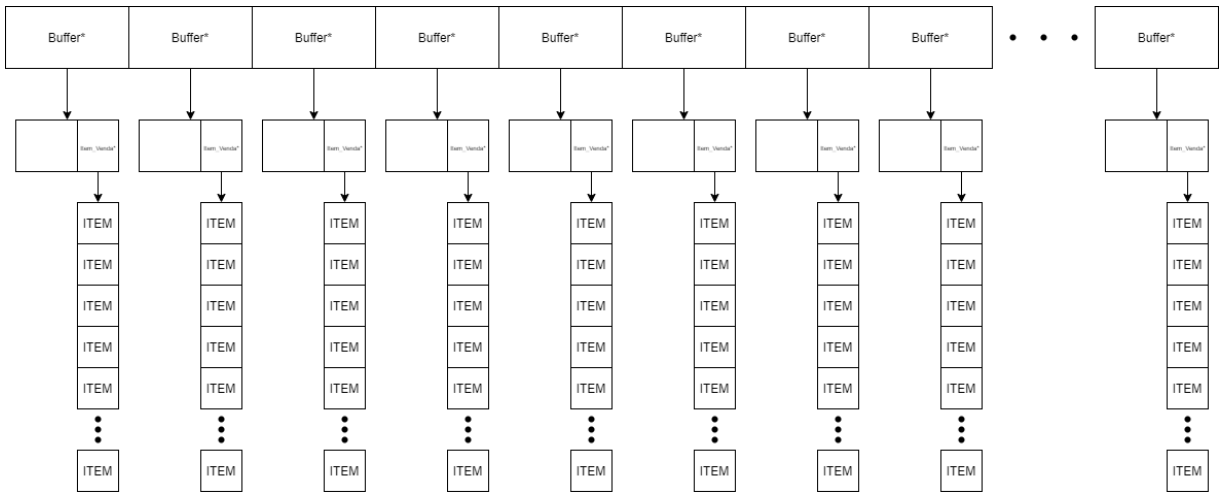
O arquivo.h contém apenas protótipos de duas funções e alguns includes. referentes a manipulação de arquivos em c.

Arquivo_Dividir: Responsável por pegar os elementos do arquivo de entrada, ordenar com o quicksort e distribuir em vários arquivos temporários.
Arquivos_Temporarios_Remover: Remove todos os arquivos temporários.

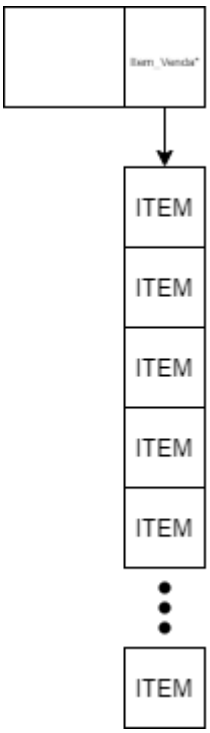
Buffer.h

O Buffer.h contém o TAD do buffer. Temos os protótipos de funções para o buffer de entrada e para o buffer de saída. As funções não tem muito segredo.

BE_CriarVetor: Responsável por criar um vetor de ponteiros de buffers de entrada.
BE_Criar: Responsável por criar buffers de entrada.
BE_Reencher: Responsável por reencher o buffer de entrada conforme os itens de venda forem retirados dele. Quando um buffer fica vazio ele tenta reencher, pegando os dados dos arquivos temporários até esvaziá-los também.
BE_Proximo: Responsável por deslocar o vetor de itens do buffer de entrada conforme o item do buffer for sendo consumido.
BE_Consumir: Responsável por consumir o item do topo do vetor de itens do buffer.
BE_Destruir: Responsável por desalocar os buffers de entrada.
BE_Vazio: Responsável por retornar true se o buffer de entrada estiver vazio e false se o contrário.
BE_VetorVazio: Responsável por retornar true se o vetor de buffers de entrada estiver vazio e false se o contrário.



BS_Criar: Responsável por criar o buffer de saída.
BS_Inserir: Responsável por inserir os itens de venda consumidos na função Merge_K_Vias no buffer de saída.
BS_Despejar: Responsável por despejar os itens de venda acumulados no buffer saída no arquivo de saída, conforme esse buffer atingir seu nível máximo ou quando os buffers de entrada estiverem completamente vazios.
BS_Destruir: Responsável por desalocar o buffer de saída.



Análise de desempenho

Considerando que a reprodução do código foi feita em uma máquina com HDD.

TABELA 1:

		S	**S**	**S**
		B/8	**B/4**	**B/2**
B	**8388608(8MB)**	0.71800	0.836000	0.699000
B	**16777216(16MB)**	1.122000	0.998000	0.777000
B	**33554432(32MB)**	6.730000	5.669000	3.139000

Figure 1: Tempo De Execução Para Ordenar Arquivo com 256000 Registro

TABELA 2:

		S	S	S
		B/8	**B/4**	**B/2**
B	**16777216(16MB)**	5.674000	5.464000	6.179000
B	**33554432(32MB)**	5.445000	5.715000	4.925000
B	**67108864(64MB)**	26.970000	20.478000	10932000

Figure 2: Tempo De Execução Para Ordenar Arquivo com 512000 Registro

TABELA 3:

		S	S	S
		B/8	**B/4**	**B/2**
B	**67108864(64MB)**	22.803000	20.650000	16.880000
B	**134217728(128MB)**	135.361000	101.268000	62.280000
B	**268435456(256MB)**	1238.045000	923.420000	465.541000

Figure 3: Tempo De Execução Para Ordenar Arquivo com 921600 Registros

TABELA 4:

		S	S	S
		B/8	**B/4**	**B/2**
B	**67108864(64MB)**	57.828000	64.328000	60.611000
B	**134217728(128MB)**	171.839000	146.060000	109.732000
B	**268435456(256MB)**	2372.732000	1346.96000	524.098000

Figure 4: Tempo De Execução Para Ordenar Arquivo com 1572864 Registros

Conclusão: Dado que quando temos mais vias no QuickSort teremos mais complexidade em colocá-los o menor elemento dos n-vetores no buffer de saída, logo é esperado que a eficiência possa cair, pois a ordenação tende a ser mais eficiente, contudo as comparações para guardar no vetor final ficam mais complexas. Com isso, vemos que o pior caso é quando temos o maior número de buffers com cada buffer tendo o maior valor. Mesmo quando aumentamos o tamanho dos buffers, o que mais atrapalha não é o ciclo de ordenação, mas a leitura/escrita do disco. Outro fator relevante é a velocidade de leitura e escrita de arquivos sequenciais e aleatórios. A leitura sequencial é ótima, pois os blocos estão alocados de forma sequencial no disco e na leitura aleatória depende da facilidade de localização do deslocamento do registro dentro do arquivo, o que nem sempre é favorável. Além disso, o disco também pode ficar fragmentado, mas vamos tentar ignorar isso dado que os testes foram executados usando o sistema operacional windows 11. Dado que os processos de leitura e escrita no disco param os processos lógico-aritméticos, pois as operações seguintes dependem do resultado dessas operações de leitura de escrita, o algoritmo também encontra-se limitado por este fator

inevitável (aparentemente). Por mais que seja mais fácil ordenar um vetor menor do que um vetor grande, perdemos eficiência na hora armazenar no vetor/arquivo final. Técnicas como o paralelismo provavelmente seriam bem-vindas nesse momento.

Divisão do trabalho

A solução do trabalho foi um esforço conjunto de ambas as partes. Em todo o processo de desenvolvimento nós utilizamos uma extensão do VScode chamada LiveShare, que transforma os arquivos do editor de texto em um ambiente colaborativo de trabalho onde ambos editam o mesmo código.