Relatório do Primeiro Trabalho Prático de Algoritmos e Estrutura de Dados 2

Getúlio Coimbra Regis[1], Igor Lara de Oliveira[2]

Universidade Tecnológica Federal do Paraná – UTFPR
Campo Mourão, Paraná, Brasil

[1]getulioregis@alunos.utfpr.edu.br
[2]igooli@alunos.utfpr.edu.br

1. Introdução

Este relatório é referente ao desenvolvimento do primeiro trabalho prático da disciplina de Algoritmos e Estrutura de Dados 2 (BCC33A), onde deveríamos desenvolver uma algoritmo para ordenar arquivos grandes, onde este arquivo por limitações não poderá ser lido totalmente na memória.

Para melhor compreensão deste relatório ele foi dividido três partes, sendo elas:

- **Requisitos:** para o desenvolvimento deste trabalho o professor requisitou que fossem criados dois Tipos Abstratos de Dados (TAD) e que fosse seguido um fluxograma.
- **Implementação:** como foi feito o desenvolvimento dos TADs e a utilização deles.
- **Análise:** os resultados das tabelas pedidas no trabalho.

2. Requisitos

Para o desenvolvimento do trabalho foram propostos a criação de dois TAD e duas implementações, sendo elas:

- ***Buffer* de Entrada:** neste *buffer* será usado para a leitura dos arquivos, tanto do arquivo principal, quanto dos arquivos temporários.
- ***Buffer* de Saída:** neste *buffer* será armazenado os registros para serem salvos nos arquivos temporários e no arquivo ordenado final.
- **Ordenação externa:** leitura do arquivo principal de entrada e ordenação em arquivos temporários.
- **Intercalação de K-Vias:** leitura dos arquivos temporários e depois intercalá-los para gerar um arquivo principal de saída ordenado

2. Implementação

Todos os códigos foram escritos utilizando a linguagem C, e para facilitar a compilação utilizamos de um *Makefile*, que iria compilar cada módulo independentemente gerando um arquivo ".o", e depois fazer um *link* com o arquivo principal, o *main.c*, e após a compilação deste deletar os arquivos ".o", além destes módulos e do arquivo principal também escrevemos um programa de teste que nos auxiliaria com o *debug*.

Este *Makefile* também nos possibilita compilar cada módulo separadamente, sem necessitar de *linkar* com o arquivo principal, gerando apenas o ".o" do módulo requisitado e não o deletando como normalmente seria feito, e um programa de teste para isso utilizamos do comando de terminal: *make nomeDoModulo*, como por exemplo: *make BufferSaida* ou o programa de teste *make teste*.

A divisão do trabalho pelos membros foi igualitária, pois utilizamos chamadas usando o aplicativo *Discord* para o desenvolvimento, e para facilitar o desenvolvimento em conjunto e versionamento do trabalho, criamos um repositório no *GitHub*, a figura 1 mostra como se encontra o repositório após a conclusão do trabalho.

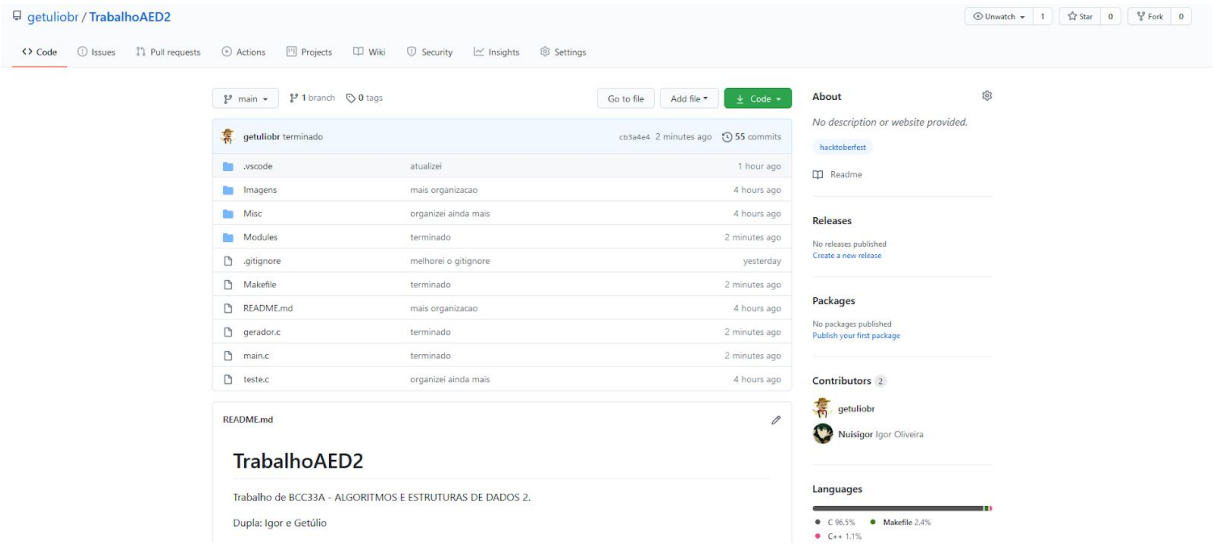

Figura 01. Como se encontra o repositório após a conclusão do trabalho.
Fonte: https://github.com/getuliobr/TrabalhoAED2

Para o desenvolvimento do programa principal foram feitas três bibliotecas, dentre elas: *Utils.h*, *BufferEntrada.h* e *BufferSaida.h.*

2.1 Buffer de Entrada

Dentro da TAD do *Buffer* de Entrada fizemos a implementação de seis funções principais para o funcionamento do programa principal, dentre elas:

- ***iv_Criar_E*** **:** função que retorna um *buffer* de entrada através da passagem de três parâmetros, o nome do arquivo, a quantidade de registros para serem armazenados no *buffer* de entrada e o ponteiro do arquivo de entrada.
- ***iv_Consumir*** **:** função que retorna o próximo registro do *buffer* atualizando sua posição e verifica se já está no final do buffer, caso esteja, a função Lê os próximos registros do arquivo e adiciona no buffer.
- ***iv_Proximo*** **:** função que retorna o próximo registro do buffer sem atualizar sua posição.
- ***iv_Vazio*** **:** função que retorna 1 caso o arquivo do buffer já foi lido por completo e 0 caso contrário.
- ***iv_Destruir_E*** **:** função que destrói o buffer de entrada por completo.
- ***compare*** **:** função que recebe e compara o ID de dois registros, *a* e *b*, retornando um valor 0 caso a comparação dê igual, maior que 0 caso *a* seja maior que *b*, e menor que 0 caso *b* é maior que *a*.

2.2 Buffer de Saída

Dentro da TAD do *Buffer* de Saída fizemos a implementação de quatro funções principais para o funcionamento do programa principal, dentre elas:

- ***iv_Criar_S*** **:** função que retorna um buffer de saída através da passagem de três parâmetros, o nome do arquivo, a quantidade de registros para ser armazenada no buffer de saída e o ponteiro do arquivo de saída.
- ***iv_Despejar*** **:** função que escreve todos os registros do buffer de saída no arquivo e limpa o buffer de saída.
- ***iv_Insere*** **:** função que insere um registro dentro do buffer de saída e verifica se já está no final do buffer, caso esteja, a função despeja os registros do buffer no arquivo.
- ***iv_Destruir_S*** **:** função que destrói o buffer saída por completo.

2.3 Utilidades (*Utils*)

Dentro desta biblioteca foram criados os tipos de estruturas que seriam utilizados nas próximas bibliotecas e também duas funções: uma que retorna o tamanho do arquivo e outra que é a principal do trabalho, a que controla os buffers de entrada e saída para gerar o arquivo ordenado final.

Sendo estas as estruturas:

- ***ITEM_VENDA*** **:**

```
typedef struct ITEM_VENDA {
        uint32_t id;
        uint32_t id_venda;
        uint32_t data;
        float desconto;
        char obs[1008];
    } ITEM_VENDA
```

- ***BUFF*** **:**

```
typedef struct BUFF{
        ITEM_VENDA* iv;
        unsigned int totalIv;
        unsigned int tam;
        unsigned int pos;
        FILE** arq;
    }BUFF;
```

2.3.1 Função Ordena

Esta é a principal função do trabalho, para o melhor entendimento desta, separamos em duas partes, a ordenação externa e a intercalação de k-vias.

1ª parte:

Abre o arquivo principal desordenado e cria vários arquivos temporários, menores, de B *bytes* ordenados, guardando o nome de cada arquivo temporário dentro de um vetor de *strings*.

```
41    for(i = 0; i < quantidadeArquivos; i++) {
42        if(arq_principal != NULL && fsize(arq_principal) <= ftell(arq_principal)) break;
43        BUFF* entrada = iv_Criar_E(arquivoentrada, quantidade_elementos_max, &arq_principal);
44        if(feof(arq_principal)) break;                                    // Condição de parada
45        qsort(entrada->iv, quantidade_elementos_max, sizeof(ITEM_VENDA), compare);    // QuickSort pra ordenar o vetor do buffer
46
47        char* arqsaida = calloc(32, sizeof(char));                         // Nomeação de arquivo de saida
48        strcpy(arqsaida, "s");
49        char snum[10];
50        sprintf(snum, "%d", i);
51        strcat(arqsaida, snum);
52        strcat(arqsaida, ".dat");
```

```
53
54      FILE* novo = NULL;                                                  // Referencia novo ponteiro de file
55      BUFF* saida = iv_Criar_S(arqsaida, quantidade_elementos_max, &novo);// Cria um BUFFER de saida com o ponteiro e nome criados
56      arq_ordenados[i] = arqsaida;
57
58      arq_ordenados_count = i;
59
60      free(saida->iv);                                                    // Copia o Buffer de entrada já ordenado, o tamanho e posição
61      saida->iv = entrada->iv;                                            // ^
62      saida->tam = entrada->tam;
63      iv_Despejar(saida);                                                 // Despeja
64
65      iv_Destruir_S(saida);                                              // Destroi a saida
66      free(entrada);                                                     // Limpa a entrada
67  };
```

Figura 02. Implementação da primeira parte da função ordena

.

2ª parte:

Cria o arquivo de saída e abre um vetor de *buffers* com todos os arquivos temporários dentro, e faz a inserção do menor elemento de todos os arquivos temporários por vez até escrever todos os arquivos dentro do arquivo de saída, após isso limpa todas as estruturas utilizadas na função.

```
73  int qtdeElementos = tamanhoDoArquivo / 1024;                                    // Quantidade de elementos que existem no arquivo;
74  int tamanho_buffer_saida = S/1024;                                              // Tamanho do buffer de saida
75  int tamanho_buffer_entrada_ordenado = floor(((B-S)/quantidadeArquivos)/1024);   // Tamanho buffer da entrada dos ordenados
76
77  FILE* arq_saida_principal = NULL;                                               // FILE do arquivo de saida principal(Todos os elementos ordenados)
78  BUFF* arq_saida_p = iv_Criar_S(saida, tamanho_buffer_saida, &arq_saida_principal);  // Criação do Buffer de saida principal
79
80  FILE** arq_entrada_ordenados = malloc(sizeof(FILE*)*arq_ordenados_count);       // Vetor de FILES dos arquivos já ordenados
81  BUFF** arq_entrada_ord = malloc(sizeof(BUFF*)*arq_ordenados_count);             // Vetor de Buffers dos arquivos já ordenados
82
83  for( i = 0; i < arq_ordenados_count; i++){                                      // Abertura de arquivos ordenados
84      arq_entrada_ordenados[i] = NULL;
85      arq_entrada_ord[i] = iv_Criar_E(arq_ordenados[i], tamanho_buffer_entrada_ordenado, &arq_entrada_ordenados[i]);
86  }
87
88  for(i = 0; i < qtdeElementos; i++){                         // Coloca os itens no arquivo
89      int menor;                                              // Guarda o maior id de N Buffer lidos
90      ITEM_VENDA menor_iv;
91      menor_iv.id = qtdeElementos+42;
92      for(k = 0; k < arq_ordenados_count; k++){
93          if(iv_Vazio(arq_entrada_ord[k])) continue;
94          ITEM_VENDA a = iv_Proximo(arq_entrada_ord[k]);
95          if(a.id < menor_iv.id){
96              menor = k;
97              menor_iv = a;
98          }
99      }
100     menor_iv = iv_Consumir(arq_entrada_ord[menor]);     // Consome o menor arquivo
101     iv_Inserir(arq_saida_p, menor_iv);                  // Insere o menor arquivo no buffer de saida
102 }
103 iv_Destruir_S(arq_saida_p);                             // Destruição da Saida
104
105 for(i = 0; i < arq_ordenados_count; i++){               // Destruição dos arquivos de entrada
106     iv_Destruir_E(arq_entrada_ord[i]);
107     remove(arq_ordenados[i]);
```

Figura 03. Implementação da segunda parte da função ordena

3. Análise

Para a análise do algoritmo completaremos as tabelas de testes pedidas pelo professor.

		S		
		B/8	B/4	B/2
B	8388608 (8MB)	1.499 s	1.429 s	1.465 s
	16777216 (16MB)	1.298 s	1.248 s	1.249 s
	33554432 (32MB)	1.322 s	1.349 s	1.317 s

Figura 04. Tempo de Execução Para Ordenar Arquivo com 256000 Registros

		S		
		B/8	B/4	B/2
B	16777216 (16MB)	2.725 s	2.820 s	2.745 s
	33554432 (32MB)	2.812 s	3.283 s	3.130 s
	67108864 (64MB)	3.566 s	3.086 s	2.919 s

Figura 05. Tempo de Execução Para Ordenar Arquivo com 512000 Registros

		S		
		B/8	B/4	B/2
B	67108864 (64MB)	5.508 s	5.562 s	6.177 s
	134217728 (128MB)	5.262 s	6.476 s	6.650 s
	268435456 (256MB)	6.327 s	6.309 s	7.480 s

Figura 06. Tempo de Execução Para Ordenar Arquivo com 921600 Registros

		S		
		B/8	B/4	B/2
B	67108864 (64MB)	9.622 s	10.489 s	11.034 s
	134217728 (128MB)	9.227 s	10.970 s	11.683 s
	268435456 (256MB)	10.872 s	11.218 s	10.355 s

Figura 07. Tempo de Execução Para Ordenar Arquivo com 1572864 Registros

3.1 Discussão de resultados

Os arquivos do algoritmo estavam sendo salvos e lidos a partir de um SSD, o que possibilitou uma rápida execução, pois quase não tinha que esperar a escrita/leitura dos arquivos.

No entanto, a partir de 921600 registros notamos que algumas vezes o programa não executaria de forma correta, o que acabou fazendo o codigo ter que ser revisado, mas mesmo fazendo várias alterações não conseguimos rodar o código 100% das vezes com 1572864 registros, o *debugger* do *Visual Studio Code* sempre apresentava um *Segmentation Fault* em locais diferentes.

Contudo o algoritmo nunca passava mais de um mega da quantidade máxima de bytes exigidas como podemos ver na próxima figura, isso se dá a variáveis que são alocadas estaticamente.

Figura 08. Quantidade de memória gasta com 1572864 registros e B valendo 256 MB