Universidade de São Paulo

Instituto de Ciências Matemáticas e de Computação

Disciplina de Estrutura de Dados 3

Engenharia de Computação

Profa. Dra. Cristina Dutra de Aguiar Ciferri

Trabalho Prático : Ordenação de arquivos de dados, *merging*, *matching*, e *multiway merging.*

Alunos:

Luis Eduardo Prado Santini	nº USP: 9065750
Rafael Guilherme da Silva Tiburcio	nº USP: 9760151
Samuel Libardi Godoy	nº USP: 9805891

08/11/2018

Índice

Introdução e função principal:

O trabalho prático possui um caráter bastante modularizado, portanto decidimos dividi-lo em tais módulos funcionais. A função principal tem como papel de tratamento dos parâmetros de entrada do programa para decidir quais módulos ou funções serão utilizadas.

Portanto na função principal encontraremos inicialmente as declarações de inclusão das bibliotecas utilizadas, sendo elas feitas por terceiros como, *stdlib.h*, *stdio.h* e *math.h*, bem como dos nossos próprios módulos:

- ***registro.h*:** módulo que contém nossa estrutura de dados definida, com os 4 campos utilizados no trabalho em seus respectivos tamanhos de 4, 20, 30 e 10 bytes. Tal arquivo possui apenas a definição da estrutura.

- ***geraDados.h*:** módulo que contém a função de geração de dados que será utilizada na funcionalidade 1.

- ***listaDados.h*:** módulo que contem a função de listagem dos dados utilizada na funcionalidade 2.

- ***ordenaInterna.h*:** trata-se do módulo de ordenação interna, utilizado na funcionalidade 3.

- ***merge.h*:** módulo de *merging* e *multiway merging* que será utilizado nas funcionalidades 4 e 6.

- ***ordenaExterna.h*** e ***split.h*:** são módulos que utilizaremos na funcionalidade 7.

  Todos esses módulos serão posteriormente melhor detalhados.

Além disso, a função principal basicamente compara o argumento de qual funcionalidade foi passado na inicialização do programa e chama a função específica para tal, passando os devidos argumentos.

Posteriormente a execução da determinada funcionalidade, cabe a função principal escrever na tela se o processamento ocorreu com sucesso ou se houve algum erro.

O trabalho foi testado e compilado no sistema operacional Ubuntu 18.10 com o GCC 7.3.

Funcionalidade 1 e geração de dados:

Após a função principal processar a entrada, se for o caso da funcionalidade 1, será acionado o módulo de geração de dados.

- ***geraDados.h:*** tal arquivo serve apenas como declaração da função *genDataFile* para ser vista e utilizada pelo programa principal.

A função *genDataFile* recebe como parâmetros o número de registros que serão gerados no arquivo, bem como o nome do arquivo que será gerado.

Nela encontramos a declaração das bibliotecas padrão como *stdlib.h* e *stdio.h*, bem como da *time.h* que será útil com suas funções para geração aleatória dos dados, *registro.h* pois manipularemos nossos registros de dados e *string.h* pois utilizaremos funções de manipulação de strings.

Funcionamento:

Inicialmente a função gera o *seed* aleatório, para uso posterior e abre os arquivos de dados binários que usamos de base para a geração dos nossos registros, bem como também abre o arquivo de saída em que tais registros gerados serão escritos. Além disso o arquivo de saída tem seu status de integridade setado em 0.

Então fizemos um loop de N repetições, sendo esse N o número de registros que serão gerados, no loop há a chamada das funções de geração de cada campo e posterior escrita do registro no arquivo.

Ao final do loop o cabeçalho do arquivo de saída é alterado para 1 e todos os arquivos são devidamente fechados.

Funções de geração de campos:

- ***genField1()****:* Gera um número aleatório para ser colocado no primeiro campo do registro, e determina se o campo precisa ser repetido ou não para atender as especificações de repetição do projeto.
- ***genField2():*** Lê um nome aleatório de rua do arquivo de base de dados “fruas” que será colocado no segundo campo do registro, e determina se o campo precisa ser repetido ou não para atender as especificações do projeto.
- ***genField3():*** Lê um nome aleatório de rua do arquivo de base de dados “fbairros” que será colocado no terceiro campo do registro, e determina se o campo precisa ser repetido ou não para atender as especificações do projeto.
- ***genField4():*** Gera a partir da *seed* aleatória um tempo aleatório que é convertido para *string* de data de acordo com o requerido no projeto, preenchendo o quarto campo do registro, também decide se é necessário haver repetições e copia um registro já criado nesse caso.

Funcionalidade 2 e listagem de dados:

Após a função principal processar a entrada, se for o caso da funcionalidade 2, será acionado o módulo de listagem de dados.

- ***listaDados.h*:** esse arquivo serve apenas para declaração da função *listaDados()* para ser reconhecida e acionada pelo programa principal.

A função *listaDados()* recebe como parâmetros somente o arquivo que será lido para realizar a listagem.

Temos as declarações das bibliotecas padrão *stdlib.h* e *stdio.h* e também *registro.h* pois utilizaremos a estrutura de registros para manipular os dados.

Funcionamento:

Trata-se de uma função bem simples que inicia na tentativa de abrir o arquivo, se isso ocorre com sucesso, ela checa o cabeçalho para saber se o arquivo está íntegro.

Se esses passos ocorreram com sucesso, inicia-se um loop de leitura de um registro no arquivo e escrita na tela, tal loop dura enquanto o arquivo não se encerrar.

Ao final do loop, o arquivo é fechado e o procedimento se encerra, retornando à função principal.

Funcionalidade 3, Ordenação Interna:

Após a função principal processar a entrada, se for o caso da funcionalidade 3, será acionado o módulo de ordenação interna.

- ***ordenaInterna.h*:** trata-se apenas da declaração da função *ordenaInterna()* para poder ser usada na função principal.

O procedimento *ordenaInterna()* recebe como parâmetros o arquivo que será ordenado e o arquivo de saída que conterá os dados ordenados.

Encontramos as declarações de bibliotecas padrão *stdlib.h* e *stdio.h*, *registro.h* pois manipularemos dados nesse formato, e de um novo *header compares.h*.

- ***compares.h*:** contem a declaração de funções que serão usadas na execução da ordenação interna para comparar os registros. Elas são *regcmp()* e *datecmp().*

- ***datecmp()*:** Procedimento que compara datas no formato dd/mm/aa, será utilizada na função *regcmp().*

- ***regcmp():*** Compara os 4 campos de cada registros na ordem de prioridade especificada no projeto, e retorna se são diferentes.

Funcionamento:

A ordenação dos dados ocorre na função *ordenaInterna()* que inicialmente abre o arquivo de leitura e checa sua integridade por meio do cabeçalho, em caso de sucesso entra em um loop que aloca memória principal até atingir um tamanho suficiente para conter todos os dados do arquivo. Após o término, fecha-se o arquivo de leitura.

Então abre o arquivo de destino no modo de escrita, colocando o status em 0. Por meio da função *qsort()* realiza o procedimento quicksort ordenando os dados do vetor criado em memória principal.

Após tal ordenação, escreve todos os registros no arquivo de destino de uma só vez, tratando-se da forma mais rápida de escrita sequencial. Por fim, altera o status de integridade para 1, fecha o arquivo de destino , libera a memória alocada e retorna a função principal.

Funcionalidades 4 e 6, operação cosequencial de *merging e multiway merging:*

Após a função principal processar a entrada, se for o caso da funcionalidade 4 ou 6, será acionado o módulo de merging.

- ***merge.h:*** Como a maioria dos headers nesse programa, trata-se apenas da declaração da função merge para ser invocada na função principal.

O módulo *merge()* recebe como parâmetros um vetor de nomes de arquivos, além do nome do arquivo de destino, saída final do processo.

Temos nesse procedimento o uso das bibliotecas padrão *stdlib.h* e *stdio.h,* utilizamos também *registro.h* e *compares.h*, já explicada na funcionalidade 3.

Funcionamento:

Começamos criando um vetor que conterá o primeiro registro de cada arquivo, para uso posterior. Iniciamos um loop que durará até todos os arquivos passados como parâmetros terem sido abertos, checando seus status. Conforme tais arquivos vão sendo abertos, o vetor com o primeiro registro de cada um vai sendo preenchido.

Abrimos o arquivo de destino como escrita, e definimos o status de integridade como 0. Então se inicia um loop que, enquanto houverem arquivos de origem abertos, encontra o registro de menor valor, no vetor de primeiros registros e escreve no arquivo de saída. Tentamos ler o próximo registro desse arquivo que continha o registro de menor valor, caso não hajam mais registros, o arquivo é fechado.

Ao final do loop todos os arquivos estarão fechados e o arquivo de destino preenchido com todos os registros ordenados. Então o status do arquivo destino é alterado para 1, garantindo a integridade.

Por fim o arquivo é fechado de saída é fechado, e a memória alocada para manipulação dos dados em memória principal é liberada, terminando o procedimento e retornando à função principal.

Vale ressaltar que como a função recebe K arquivos de entrada, ela funciona para resolver tanto a funcionalidade 4, como a funcionalidade 6, uma vez que recebe um vetor de K arquivos de entrada e realiza o merging nesses K arquivos, no caso específico da funcionalidade 4, K = 2.

Funcionalidade 5, operação cosequencial de matching:

Após a função principal processar a entrada, se for o caso da funcionalidade 5 ou 6, será acionado o módulo de matching.

- ***match.h***: O header desse módulo também tem apenas a função de declarar a função para utilização no programa principal.

A função *match()* recebe como parâmetros um inteiro que informa o número de arquivos, bem como um vetor com os arquivos de origem e o nome do arquivo de destino.

Temos presentes as bibliotecas para operações padrão *stdlib.h* e *stdio.h* e utilizamos também *registro.h* e *compares.h*, ambas já apresentadas.

Funcionamento:

A função inicia-se com a declaração das variáveis utilizadas, abertura dos arquivos recebidos como parâmetro e checagem dos status de integridade. Os registros iniciais de cada arquivo são copiados para um vetor de registros que será usado nas comparações do matching.

Abrimos o arquivo de destino e atribuímos status 0 até a conclusão do procedimento com sucesso.

Então inicia-se um loop que enquanto houverem arquivos de entrada abertos, busca pelo vetor de primeiros registros pelo de menor valor e verifica se houve matching com algum deles. Se não houver avança na leitura do arquivo com o registro de menor valor.

Eventualmente algum arquivo irá chegar ao fim durante os avanços, nesse caso o arquivo é fechado. Quando não houver mais arquivos abertos o loop se encerra, o procedimento de matching está concluído, portanto o status do arquivo de destino é alterado para 1, a memória utilizada é liberada e há o retorno para a função principal.

Funcionalidade 7, Ordenação Externa:

Após a função principal processar a entrada, se for o caso da funcionalidade 7, será acionado o módulo de ordenação externa.

- ***ordenaExterna.h***: é um header file apenas com a declaração de função de ordenação externa, *ordenaExterna(),* para ser chamada a partir da função principal.

A função *ordenaExterna()* recebe como parâmetros o nome do arquivo de entrada e também do arquivo de saída do procedimento.

Nela encontramos a declaração de bibliotecas padrão *stdlib.h* e *stdio.h*, bem como da biblioteca *math.h* pois usaremos a função *ceilf()* de arredondamento no cálculo do número de subarquivos gerados.

Além disso temos *merge.h* que será usada no merging de arquivos, *asprintf.h* para declaração da função *asprintf()* que utilizaremos e não vem por padrão com a linguagem C. Por fim temos *split.h* que será descrita em seguida.

- ***split.h:*** server apenas para a declaração da função *split()* usada em outras rotinas.

Função *split():* trata-se de uma sub-rotina para a realização do split de arquivos na ordenação externa. Inicialmente abre o arquivo de origem recebido como parâmetro e checa o status de integridade, se isso ocorrer com sucesso seguimos para a alocação de memória no buffer para o conteúdo de um sub-arquivo completo.

Então se inicia um loop que enquanto houver registros no arquivo de origem irá lê-los, ordená-los e escrevê-los adequadamente num subarquivo, conforme tais subarquivos forem totalmente preenchidos, são fechados e um novo é criado. Ao final a memória utilizada é liberada e a função retorna o número de arquivos gerados pelo split.

Funcionamento:

O procedimento inicia-se pela realização do split, quebrando o arquivo em vários subarquivos e ordenando-os internamente. Então inicia-se um loop que abre MAX_OPEN_FILES arquivos de cada vez, e realiza o merge em um único. No total em cada nível existem “srcsbubc” sub-arquivos de origem que serão unidos em “srcsbubc”/MAX_OPEN_FILES arquivos de destino, e o loop dura até que reste apenas 1 arquivo de destino, que será a saída final do procedimento.

Quando o loop termina, é realizado o renomeio do arquivo resultante para o nome passado por parâmetro como arquivo de destino, e retorna à função principal.

Referências Bibliográficas:

Implementação multiplataforma da função asprintf(), obtida de *https://stackoverflow.com/questions/40159892/using-asprintf-on-windows*, acessado em 06/12/2018.