

Trabalho de Estrutura de Dados (GCC216)

Sistema de cadastro de deuses nórdicos

Andrew Takeshi Tanaka de Vita

Lucas Antônio Lopes Neves

Vinicius Tavares Pimenta

(Ciência da Computação)

Lavras, 31 de Maio de 2018

**Sumário**

1. Introdução
2. Como foi realizado o projeto
3. Estruturas de dados utilizadas
4. Explicação do código
   1. Uso do Registro
   2. Função main
   3. Inserção de um novo objeto no arquivo
   4. Remoção de um objeto do arquivo
   5. Consulta de um objeto no arquivo
   6. Impressão todo o conteúdo do arquivo
   7. Impressão dos registros de um bloco de modo ordenado
5. Considerações finais
6. **Introdução**

Este trabalho tem como finalidade colocar em prática todo o conhecimento adquirido nas aulas de Estruturas de Dados (GCC216), tanto nas aulas práticas quanto nas teóricas.

Os requisitos principais para o trabalho era desenvolver um sistema para cadastro, com operações para:

* inserção de um novo objeto no arquivo;
* remoção de um objeto do arquivo;
* consulta de um objeto no arquivo;
* impressão todo o conteúdo do arquivo;
* impressão dos registros de um bloco de modo ordenado.

O tema escolhido foi *Deuses Egípcios e Nórdicos*.

1. **Como foi realizado o projeto**

O grupo decidiu utilizar a plataforma G*itHub* para gerenciar o projeto. Com ela, o grupo foi capaz de ter um controle além de trabalhar em equipe vendo cada alteração feita no arquivo do trabalho.

O *link* para o acesso do projeto no *GitHub* está disponível em:

<https://github.com/lucas54neves/projetoED>

3) **Estruturas de dados utilizadas**

BlocosDados e TabelaH são estruturas de dados utilizadas no programa. O primeiro tem a finalidade de manipular os objetos de deuses . E por fim, o segundo tem o intuito de informar a posição dos blocos dentro do arquivo binário.

4) **Explicação do código**

* 1. **Uso do Registro**

Como o grupo ficou responsável pelo tema dos deuses egípcios e nórdicos, foi montado um registro (*struct*) chamado *deuses* utilizando suas subcategorias (id, nome, domínio e biografia) como campos*.*

* 1. **Função main**

Foi criado um laço *(while)* que é o menu principal. Ele funciona enquanto a opção escolhida pelo usuário não for *0* (que é a opção de fechar o programa). Para que o laço fosse inicializado, a variável opção foi iniciada com o valor -1.

O *system(“clear”)* foi utilizado para limpar a tela toda vez que se abre uma opção do menu.

No menu, foi dado ao usuário cinco opções e solicitado a ele que digite um número referente à opção desejada. De acordo com a opção selecionada pelo usuário, um *switch case* foi utilizado para chamar as funções da opção desejada.

* 1. **Inserção de um novo objeto no arquivo**

A inserção de dados tem início pelo método global: *void InsereDados (TabelaH\* tabelaCadastro)*. Este tem por finalidade fazer com que o usuário digite os valores dos dados do deus (deus.id, deus.nome, deus.dominio e deus.biografia). Após isso, um outro método global global, *int FuncaoHash(deus.id)*, é usada para gerar uma posição (h) na tabela hash através da passagem de parâmetro do valor de deus.id. O valor dessa posição é gerada em um valor em decimal que, posteriormente, é convertida em um valor binário através através da utilização do método global *ConverteBinario(h)*, o qual é passado por parâmetro o valor da posicao (h) em decimal. Com o valor da posição da tabela hash em binário em mãos, pode-se passar por parâmetro essa posição no método da classe da TabelaH, *int PosicaoBytes (int h)*, a fim de armazenar essa posição no atributo vetor mElementos[h] da classe da TabelaH. Tendo feito isso, agora pode-se armazenar na variável posVal a tabelaCadastro, a qual irá apontar para o retorno desse método. A variável posVal vai ser utilizada para identificar se a posição da tabela hash está preenchida com algum valor, caso esteja a posVal vai ser diferente de -1, caso contrário, o valor será -1, identificando que aquela posição está vazia. Caso o usuário for fazer a primeira inserção no programa, ele entrará na condição que a posição se encontra vazia, ou seja, o valor do posVal será -1. Desse modo, a tabelaCadastro irá apontar para o método da tabela, *insereTabela(deus)*, passando por referência o registro deus. Nesse método iremos armazenar o valor da posição da tabela hash em binário na variável pos utilizando os métodos globais *int ConverteDecimal* e *int FuncaoHash(deus.id)*. Com o valor da posição em binário armazendo na variável pos, pode-se chamar o método da classe TabelaH, *bool PosOcupada (int pos)*, a fim de verificar se a posição tem algum bloco existente ou não. Caso não exista nenhum bloco nessa posição, criaremos um ponteiro do tipo BlocosDados chamado novoBloco, o qual iremos instanciar. Tendo feito isso, chamamos o método da classe BlocosDados, *void InsereBloco(deus)*, passando o registro por parâmetro. Nesse método será realizado uma verificação se o bloco está vazio, em uso ou cheio. Caso ele venha estar vazio, o atributo vetor mBloco[mTamBloco] irá receber o registro deus, o mTamBloco irá ser incrementado e o mCabecalho vai receber a posição na tabela hash. Após isso, pode-mos escrever no arquivo utilizando o método global *EscreveArquivoNovo(novoBloco)*. Após isso, iremos encontrar a posição em bytes do bloco do arquivo utilizando o método da classe BlocosDados, *int PosicaoArquivo(BlocosDados\* novoBloco),* passando o novoBloco por parâmetro. Nesse método o processo funciona do seguinte modo: primeiro abre-se o arquivo em modo de leitura, em sequência direciona-se o ponteiro no final do arquivo usando *Leitura.seekg(0, Leitura.end).* Tendo feito isso, eu capturo o tamanho do arquivo e armazeno na variável tamArq usando *int tamArq = Leitura.tellg().* Após isso, direciono o ponteiro para o início do arquivo. Depois, pego o tamanho do arquivo e divido pelo tamanho do bloco obtendo a quantidade de blocos existente no arquivo binário usando a seguinte operação: *int qntBloc = tamArq / sizeof(BlocosDados).* Em seguida, aloco um objeto de BlocosDados para receber as informações dentro do arquivo binário dos blocos salvo nele. Após isso, cria-se uma condição para capturar a posição daquele bloco em representação em bytes. Obtida a posição, armazeno ela na variável pos, a qual será retornada pelo método, e, posteriormente, armazeno na variável bytes. Tendo o valor da posição do vetor (*pos*) e o valor bytes do bloco (*bytes*), armazeno no atributo vetor mElementos[pos] o valor em bytes do bloco no arquivo binário. Esse foi o primeiro exemplo quando a posição na tabela hash (*posVal*) está vazia. Caso a posição esteja preenchida, ou seja, posVal diferente de -1, iremos alocar um objeto de BlocosDados que será usada no método global, *void CarregaBloco (blocIns, posVal),* a fim de transferir as informações do arquivo para o bloco usando *Carregar.read((char\*)(auxBloco), sizeof(BlocosDados)).* Tendo as informações do bloco em mãos, verificaremos se o id é repetido usando o método da classe BlocosDados, bool ProcuraIdRepetido (int id), passsando por referência o deus.id. Esse método irá retorna uma valor em booleano que será armazenado na variável existeRep. Após isso, passamos uma condição para verificar se a variável existeRep é verdadeira. Caso ela for, o programa irá informar ao usuário que o Id é repetido e vai perguntar se ele deseja tentar outro id. Agora caso a variável existeRep for falsa, iremos apontar a tabelaCadastro para o método void InsereTabela(Dado deus), passando o registro deus por parâmetro. Em seguida, no método da classe TabelaH, *void InsereTabela (Dado deus),* iremos armazenar a posição na variável pos utilizando os métodos globais, *int funcaoHash (deus.id)* e *int ConverteDecimal.* Tendo feito isso, podemos utilizar o método da classe TabelaH, *bool PosOcupada (int pos),* a fim de verificar se a posição tem um bloco existente. Caso sim, alocaremos um objeto do tipo BlocosDados que será usado no método global, void CarregaBloco (BlocosDados\* auxBloco, int posBytes), com intuito de transferir as informações do arquivo para o bloco usando *Carregar.read((char\*)(auxBloco), sizeof(BlocosDados)).* Tendo feito isso, apontaremos o auxBloco no método da classe BlocosDados, *void insereBloco (Dado deus)*. Nesse método, iremos armazenar o registro deus no atributo vetor mBloco[mTamBloco] e, posteriormente, incrementa-se o mTamBloco. Em sequência, usamos o método global, void EscreveArquivoVelho (BlocosDados\* auxBloco, int posBytes), para escrever no arquivo binário. Esse foi o segundo exemplo quando a posição na tabela hash está em uso, porém o bloco ainda não está cheio. E por fim, quando o bloco está cheio, verificado através do método da classe BlocosDados, *bool BlocoCheio (),* em que é identificado bloco cheio quando mTamBloco é igual a 4. Após identificar que o bloco está cheio, iremos informar o usuário a situação do bloco e, posteriormente, usa-se exit(EXIT\_FAILURE) para interrompe a execução do programa.

d) **Remoção de um objeto do arquivo**

A remoção de um objeto do arquivo tem início através do método global *void RemoveDados (TabelaH\* tabelaCadatro)*. Nesse método o usuário irá entrar com um id. Esse id será passado por parâmetro no método global, int FuncaoHash (int i), e, posteriormente, convertido em binário pelo método global int ConverteDecimal (int binario). Esse resultado é a posição referente a tabelaHash que será armazenado na variável numH. Tendo feito isso, iremos usar uma condição para verificar se a posição está ocupada. A condição será verificada através do returno do método da classe Tabela H, bool PosOcupada (int pos). Caso esteja ocupado, iremos alocar um objeto chamado blocRem do tipo BlocosDados. Em sequência, iremos obter a posição em bytes da tabela hash em que se encontra o bloco através do método da classe TabelaH, inline int PosicaoBytes (int h). Tem esse valor, iremos armazenar na variável posBytes, a qual será usada no método global, void CarregaBloco (BlocosDados\* auxBloco, int posBytes). Esse método irá transferir as informações do arquivo para o bloco usando *Carregar.read((char\*)(auxBloco), sizeof(BlocosDados)).* Após isso, iremos armazenar na variável posId o retorno do método da classe BlocosDados, int PosDeus (int id), o qual refere-se a posição do deus que está no bloco. Tendo feito isso, chamo o método void RemoveDeus (int posId). Nela iremos primeiramente atribuir o valor -1 ao id do bloco que será removido. Em sequência, usamos a condicional para verificar se a posId é menor que o mTamBloco-1. Caso ela for, apenas eu direciono a armazeno o mBloco posterior ao bloco atual. Mas, caso o mTamBloco for igual a 1, eu apenas atribuo a variável booleana mUse o valor false, indicando que agora em diante o bloco não está em uso. Ao final desse processo, decrementamos o mTamBloco. Esse foi o primeiro exemplo de remoção de blocos tendo em vista que a posição está ocupada e que o posId que o usuário digitou existe. Caso não exista, o programa irá informar ao usuário que não existe deus com esse id, além de aproveitar e perguntar ao usuário se ele desejar sair da posição. E por fim, o último caso trata-se de quando a posição está ocupada e o posId não existe, ou seja, posId é igual a -1. Dessa forma, iremos informar ao usuário que não existe deus com esse id, além perguntar ao usuário se ele desejar sair da posição.

e) **Consulta de um objeto no arquivo**

A consulta de um objeto no arquivo tem início pelo método global *void ConsultaDados (TabelaH\* tabelaCadastro).* Nesse método o usuário irá entrar com um id. Esse id será passado por parâmetro no método global, int FuncaoHash (int i), e, posteriormente, convertido em binário pelo método global int ConverteDecimal (int binario). Esse resultado é a posição referente a tabelaHash que será armazenado na variável numH. Tendo feito isso, iremos usar uma condição para verificar se a posição está ocupada. A condição será verificada através do returno do método da classe Tabela H, bool PosOcupada (int pos). Caso esteja ocupado, iremos alocar um objeto chamado blocoCons do tipo BlocosDados. Em sequência, iremos obter a posição em bytes da tabela hash em que se encontra o bloco através do método da classe TabelaH, inline int PosicaoBytes (int h). Tem esse valor, iremos armazenar na variável posBytes, a qual será usada no método global, void CarregaBloco (BlocosDados\* auxBloco, int posBytes). Esse método irá transferir as informações do arquivo para o bloco usando *Carregar.read((char\*)(auxBloco), sizeof(BlocosDados)).* Após isso, iremos usar o método da classe BlocosDados, void ImpressaoConsulta (int id), a fim que imprimir apenas os dados do deus consultado. Caso o exitir um mBloco[i].id igual ao id será impresso os dados, caso contrário irá informar ao usuário que o id não está cadastrado.

f) **Impressão todo o conteúdo do arquivo**

A impressão de todo o conteúdo do arquivo tem início pelo método global *void ImprimeArquivoOrdem ().* Nesse método iremos começar abrindo o arquivo em modo de leitura e alocando um objeto chamado impTodo do tipo BlocosDados. Em sequência direciona-se o ponteiro no final do arquivo usando *Leitura.seekg(0, Leitura.end).* Tendo feito isso, eu capturo o tamanho do arquivo e armazeno na variável tamArq usando *int tamArq = Leitura.tellg().* Após isso, direciono o ponteiro para o início do arquivo. Depois, armazeno o tamanho do bloco na variável var. Em sequência, pego o tamanho do arquivo e divido pela var com intuito de obter a quantidade de blocos existente no arquivo binário. Após, eu crio uma repetição indo até a qntBloco. Cada repetição eu transfero as informações do arquivo para o bloco usando *leitura.read((char\*)(impTodo), sizeof(BlocosDados)) e chamo o método da classe BlocosDados, void ImprimeBloco ().* Nesse método iremos primeiramente verificar se o bloco está em uso, caso sim, iremos imprimir seus dados. Em sequência, alocamos um registro chamado deus do tipo Dado, e passamos esse registro e o tamanho do bloco por parâmetro no método global *void InsertionSort (Dado\* deus, int tam),* com intuito de ordenar os registros.Tendo feito isso, fazemos uma repetição indo até mTamBloco, a fim de imprimir seus dados.

g) **Impressão dos registros de um bloco de modo ordenado**

A impressão de todo o conteúdo do arquivo tem início pelo método global *void ImprimeBlocoOrdem ().* Nesse método iremos chamar o método da classe TabelaH, *void ImprimeTabela ()*, a fim de mostrar a tabela hash. Em sequência, o usuário deve digitar a posição em número binário correspondente a tabela que será armazenado na variável numBin. Esse variável será usada para procurar a posição em bytes do posição em bytes do bloco usando o método da classe TabelaH, inline int PosicaoBytes (int h). Com essa posição em bytes, podemos verificar se o valor do resultado é -1, ou seja, posição desocupada. Caso sim, iremos informar ao usuário a situação atual da posição. Caso contrário, iremos alocar um objeto chamado blocOrd do tipo BlocosDados e passar esse objeto juntamente com a posição em bytes do bloco no método global *void CarregaBloco (BlocosDados\* auxBloco, int posBytes)*, com intuito de transferir as informações do arquivo para o bloco usando *Carregar.read((char\*)(auxBloco), sizeof(BlocosDados)).* Em sequência, uso o método da classe BlocosDados, *void ImprimeBloco ()*, para realizar a impressão dos dados do bloco.

**5) Considerações finais**

Este trabalho consolidou os conhecimentos adquiridos durante as aulas, uma vez que foi possível colocar a matéria em prática.

Além disso, o trabalho foi essencial para que todos os membros do grupo possam aprimorar suas habilidades de trabalho em equipe. Durante o processo de criação do programa, alguns erros no código foram cometidos devido à quantidade de pessoas codificando o mesmo código e por erro de lógica. Porém, esses erros foram corrigidos com a ajuda mútua dos membros.

Por fim, o tema do trabalho foi essencial para motivar o grupo para realizar o projeto, uma vez que esse tema era de interesse de cada um dos membros.