**Algoritmo e Estrutura de Dados**
**Relatório do Trabalho 1 - Biblioteca**
**Autores:**

Alex Galhardo
Numero USP: 10408344

Ian Castilho Caldeira Brant
Numero USP: 10133967

**ICMC USP**

**São Carlos, 14 de Outubro de 2017**

## ● Introducao

Esse relatório tem como objetivo descrever e analisar o desenvolvimento, em C, de um sistema computacional para uma biblioteca, utilizando conceitos de TAD e estruturas de dados.

O objetivo do código é fornecer assistência a um(a) bibliotecário(a), que será o seu único usuário. Dessa forma, o sistema desenvolvido deveria realizar funções como cadastrar e descadastrar livros e alunos, buscar livros e alunos, realizar empréstimos de livros para alunos de modo que, caso não haja cópias disponíveis do livro que será emprestado, o aluno será inserido em uma fila de espera, tendo o livro à sua disposição quando outro aluno o devolvesse. O sistema deve também ter um processo de envio de mensagens ao bibliotecário, alertando-o quando um livro fica disponível para um determinado aluno, por exemplo.

Um dos principais desafios nesse trabalho foi desenvolver dois diferentes TADs que realizam a mesma função, mas se diferem em seu método de alocação de memória, sendo um estático e o outro dinâmico. Ambos TADs deveriam poder ser utilizados em um mesmo main code. Além disso, o programa deveria ser econômico com relação à memória utilizada, não desperdiçando capacidade, tornando necessário o uso de métodos de alocação estática encadeada (banco de memória) e alocação dinâmica encadeada.

## ● Estrutura

No TAD Estático, foram utilizadas as estruturas de dados (structs,em C) descritas abaixo:

- **LIVRO:** Struct que contém os dados cadastrais de um livro (Titulo, Autor, Edicao, Copias, etc) bem como variáveis importantes para o funcionamento dos bancos de memória, como a posição do primeiro e do ultimo aluno para qual o livro está emprestado no banco de memória de alunos para os quais o livro está emprestado (**iniemp** e **fimemp**), a posicão do primeiro e do ultimo aluno que estão na fila de espera no banco de memória das filas de espera (**iniesp**, **fimesp**), posição do livro no banco de memoria de livros (pos), quantidade de pessoas na fila de espera (espera) e quantidade de alunos para os quais o livro está emprestado.

- **NO_LIVROS**: Struct que contém os dados de um determinado nó da lista de livros cadastrados no sistema. Cada nó contém como informação uma struct **LIVRO**, com os dados do livro que está naquele nó, bem como um inteiro **prox**, que indica o índice do próximo nó no banco de memória onde estará a lista de livros.

- **LISTA_LIVROS**: Struct que contém a lista de livros em modelo de banco de memória. Ela possui um vetor de elementos **NO_LIVROS ,** do tamanho que se deseja (através do #define **tamLivros**), bem como os inteiros **ini, fim** e **pv,** que indicam a posição do primeiro e último nós na lista e a posição do primeiro item vazio da lista..

As estruturas **ALUNO, NO_ALUNOS** e **LISTA_ALUNOS,** seguem o mesmo modelo das estruturas de Livros, adaptadas para conter informações de alunos.

- **LISTA_ESPERA:** Essa estrutura é um banco de memória para armazenar as filas de espera dos livros. É uma lista de alunos, portanto possui como informação o vetor **NO_ALUNOS.** Possui também o inteiro **pv** que indica a posição do primeiro vazio da lista. Como essa lista é um banco de memória para armazenar várias filas de espera, as informações de início e fim das filas não são armazenadas aqui, pois cada livro possuirá suas próprias posições de inicio e de fim

- **LISTA_EMPRESTADOS:** Essa lista é idêntica à lista de filas de espera, mas é utilizada para armazenar a lista de alunos para os quais um determinado livro está emprestado.
- **LISTA_EMPRESTIMOS:** Com lógica semelhantes às listas de espera e emprestados, essa lista é utilizada para armazenar a informação de quais livros um determinado aluno tem emprestados.
- **BIBLIOTECA:** Essa estrutura centraliza, em uma biblioteca, todos os bancos de memória utilizados para armazenar informações de livros, alunos, filas de espera, lista de emprestados e emprestimos.

No TAD Dinâmico, foram utilizadas as estruturas de dados (structs,em C) descritas abaixo (descritos brevemente devido à semelhança, em vários pontos, com o estático):

- **LIVRO:** Struct que contém os dados cadastrais de um livro, semelhantes ao TAD estático, bem como um ponteiro ***prox** que aponta para o próximo no de livros (contido dentro da própria struct).
- **LISTA_LIVROS:** Struct que define a existência da lista de livros, contendo um ponteiro ***inicio** que aponta para o primeiro livro da lista. A partir daí, cada livro aponta para um próximo. Como é apenas uma lista, não são necessários ponteiros que apontem para o início e para o fim da lista.

As estruturas **ALUNO** e **NO_ALUNOS** seguem o mesmo modelo das estruturas de Livros, adaptadas para conter informações de alunos.

- **DADOS:** Essa struct armazena os dados de livros e alunos que entraram em uma determinada fila de espera, ou em uma determinada mensagem.
- **LISTA_ESPERA:** Struct que define a criação de uma lista de espera, que será usada para armazenar as filas de espera de todos os livros da biblioteca.
- **PILHA_EMAIL:** Pilha que irá armazenar as mensagens enviadas para o usuário do sistema.
- **BIBLIOTECA:** Essa estrutura centraliza, em uma biblioteca, todas as estruturas utilizadas para armazenar informações de livros, alunos, filas de espera e mensagens.

## ● Desafios Técnicos Encontrados

- Unificar as Mains: esse foi um dos principais desafios encontrados na elaboração do trabalho. Devido às diferente técnicas pensadas para o funcionamento das operações nas versões estática e dinâmica, não conseguimos criar uma main que funcionasse com ambas versões. Isso se deu, por exemplo, por termos visto como necessário que na versão estática, um determinado elemento como livro ou aluno fosse encontrado e modificado a partir de um inteiro que indicaria a sua posição no vetor onde o mesmo se encontra armazenado. Isso não acontece no dinâmico, onde a identificação se da por ponteiros. Dessa forma, as mesmas funções precisariam ter, em alguns casos, diferentes tipos de entrada, fato que não conseguimos contornar, impossibilitando a unificação das mains.

- Utilização de Tad: a utilização de conceitos de TAD se mostrou um desafio na elaboração do trabalho devido à necessidade de respeitar suas regras, e também ao fato de nós mesmos (desenvolvedores do código) sermos os usuários dos TADs que criamos. Isso tornou o desenvolvimento do sistema mais trabalhoso, pois algumas funções, como as de cadastro, poderiam ser mais facilmente desenvolvidas caso pudessem conter diretamente as entradas e saídas de informações, na interface com o usuário do sistema.

- Interface com o Usuário: a interface com o usuário foi um desafio pois, cada elemento adicionado nessa interface adicionava alguma complexidade no desenvolvimento do código. A opção que demos ao usuário para que confirme ou

não uma operação, por exemplo, sempre acabava bifurcando o código, sendo necessário desenvolver o caso em que há confirmação bem como o caso em que não há confirmação. Isso não aumenta a complexidade computacional do código, mas torna o seu desenvolvimento um pouco menos direto e intuitivo.

- **Operações do TAD e suas complexidades computacionais**

## Estático:

void criaBiblioteca(BIBLIOTECA *b): Função que cria uma biblioteca. Essa função tem complexidade O(**tamEmprestimo** + **tamEmprestados** + **tamEspera** + **tamAlunos** + **tamLivros**) devido aos fors utilizados para preencher os vazios dos bancos de memória com a informação do próximo de cada nó vazio.

void insereAluno(BIBLIOTECA *b, ALUNO al, int *erro): Funcao que insere um aluno na biblioteca. Essa função possui complexidade O(6) nos piores casos (banco de memoria não está cheio) e O(1) no pior caso, em que o banco de memória da lista de alunos está cheio.

void removeAluno(BIBLIOTECA *b, ALUNO aluno, int *erro): Funcao que remove um aluno da biblioteca. Essa funcao tera complexidade O(n) no pior caso, sendo n o número de alunos já cadastrados. Isso acontece porque a função percorre a lista de alunos até encontrar o aluno a ser removido, sendo o pior caso quando o aluno estiver no fim da lista ou quando o aluno nao for encontrado.

void insereLivro(BIBLIOTECA *b, LIVRO li, int *erro): Função que insere um livro na biblioteca. Mesmo caso da funcao insereAluno, adaptada para informações de livros.

void removeLivro(BIBLIOTECA *b, LIVRO x, int *erro): Função que remove um livro da biblioteca. Mesmo caso da funcao removeAluno, adaptada para informações de livros.

void insereEspera(BIBLIOTECA *b, LIVRO *li, ALUNO *al, int *erro): Funcao que insere um aluno na fila de espera de um livro. Possui complexidade equivalente às demais funções de inserção.

void RemoveEspera(BIBLIOTECA *b, LIVRO *li, ALUNO al, int *erro): Função que remove um aluno da fila de espera de um livro. Possui complexidade equivalente às demais funções de remoção.

int EstaNaEspera(BIBLIOTECA *b, LIVRO *livro, ALUNO aluno): verifica se um determinado aluno esta na fila de espera de um livro. No pior caso possui complexidade O(n), sendo n o número de alunos na fila de espera. Isso acontece devido à necessidade de percorrer a fila de espera do livro.
void InsereEmprestimos(BIBLIOTECA *b, LIVRO *li, ALUNO *al, int *erro): Funcao que insere um livro na lista de livros que um aluno possui emprsetados. Possui complexidade equivalente às demais funções de inserção.

void InsereEmprestados(BIBLIOTECA *b, LIVRO *li, ALUNO *al, int *erro): Funcao que insere um aluno na lista de alunos que estao com um determinado livro empresatado. Possui complexidade equivalente às demais funções de inserção.

void RemoveEmprestado(BIBLIOTECA *b, LIVRO *li, ALUNO al, int *erro): Função que remove um aluno da lista de alunos que estao com um determinado livro empresatado. Possui complexidade equivalente às demais funções de remoção.

int EstaEmprestado(BIBLIOTECA *b, LIVRO *livro, ALUNO aluno): verifica se um determinado aluno esta na lista de alunso para os quais um livro está emprseatado. No pior caso possui complexidade O(n), sendo n o número de alunos para os quais o livro está emprestado.

void verificaISBN(BIBLIOTECA *b, LIVRO *livro, int *ver): verifica se um determinado ISBN possui 4 dígitos e se o mesmo já está cadastrado. Possui complexidade O(n) no pior caso, sendo n a quantidade de livros cadastrados na biblioteca.

void verificaTitulo(BIBLIOTECA *b, LIVRO livro, int *ver): verifica se um determinado Titulo já está cadastrado na biblioteca. Possui complexidade O(n) no pior caso, sendo n a quantidade de livros cadastrados na biblioteca.

void verificaNumUSP(BIBLIOTECA *b, ALUNO *aluno, int *ver): verifica se um determinado número USP possui 8 dígitos e se o mesmo já está cadastrado. Possui complexidade O(n) no pior caso, sendo n a quantidade de alunos cadastrados na biblioteca.

void verificaNome(BIBLIOTECA *b, ALUNO aluno , int *ver): verifica se um determinado Nome já está cadastrado na biblioteca. Possui complexidade O(n) no pior caso, sendo n a quantidade de alunos cadastrados na biblioteca.

void verificaEmail(BIBLIOTECA *b, ALUNO aluno, int *ver): verifica se um determinado email é terminado em @usp.br e se o mesmo já se encontra cadastrado na biblioteca. Possui complexide O (k + n), sendo k o tamanho do email e n o número de alunos cadastrados na biblioteca.

## Dinâmico:

oid criaBiblioteca(BIBLIOTECA *b): Função que cria uma biblioteca, contendo uma lista de alunos, uma lista de livros, uma lista de espera e uma pilha para os emails. Essa função tem complexidade O(**1**) já que não tem nenhum loop em sua execução, que também não depende do tamanho de nenhum tipo de entrada.

void insereLivro(BIBLIOTECA *b, LIVRO *livro, int *erro): Função que insere um livro na biblioteca criada. Essa função tem complexidade O(**1**) já que não tem nenhum loop em sua execução, que também não depende do tamanho de nenhum tipo de entrada.

void removeLivro(BIBLIOTECA *b, LIVRO *livro, int *erro): Função que remove um livro da biblioteca. No pior caso, essa função tem complexidade O(n), sendo n a quantidade de livros cadastrados na biblioteca. Esse pior caso aconteceria caso o livro a ser removido fosse o ultimo na lista de livros.

void insereAluno(BIBLIOTECA *b, ALUNO *aluno, int *erro): Função que insere um aluno na biblioteca criada. Essa função tem complexidade O(**1**) já que não tem nenhum loop em sua execução, que também não depende do tamanho de nenhum tipo de entrada.

void removeAluno(BIBLIOTECA *b, ALUNO *aluno, int *erro): Semelhante à função de remoção de livros, adaptada para alunos.

void verificaISBN(BIBLIOTECA *b, LIVRO *livro, int *ver): verifica se um determinado ISBN possui 4 dígitos e se o mesmo já está cadastrado. Possui complexidade O(n) no pior caso, sendo n a quantidade de livros cadastrados na biblioteca.

void verificaTitulo(BIBLIOTECA *b, LIVRO *livro, int *ver): verifica se um determinado Titulo já está cadastrado na biblioteca. Possui complexidade O(n) no pior caso, sendo n a quantidade de livros cadastrados na biblioteca.

void verificaNumUSP(BIBLIOTECA *b, ALUNO *aluno, int *ver): verifica se um determinado número USP possui 8 dígitos e se o mesmo já está cadastrado. Possui complexidade O(n) no pior caso, sendo n a quantidade de alunos cadastrados na biblioteca.

verificaEmail(BIBLIOTECA *b, ALUNO *aluno, int *ver): verifica se um determinado email é terminado em @usp.br e se o mesmo já se encontra cadastrado na biblioteca. Possui complexide O (k + n), sendo k o tamanho do email e n o número de alunos cadastrados na biblioteca.

Pudemos concluir, dessa forma, que a versão estática possui maior complexidade de execução, pois possui funções que terão complexidade proporcional ao número máximo de elementos que podem cadastrados na biblioteca, enquanto no dinâmico os piores casos terão complexidade proporcional à quantidade de elementos já cadastrados. Dessa forma, a versão estática sempre terá complexidade maior ou igual à dinâmica.