UNIVERSIDADE FEDERAL DE SÃO JOÃO DEL REI
CIÊNCIA DA COMPUTAÇÃO
AEDS II

Trabalho Prático 1

Ana Julia Silva Ledo,
Lucas Eduardo Bernardes de Paula,
Messias Feres Curi Melo

São João Del Rei, Abril de 2023

Alunos:
Ana Julia Silva Ledo
Lucas Eduardo Bernardes de Paula
Messias Feres Curi Melo

Prof. Rafael Sachetto Oliveira

Palavras-chave: Lista por arranjo, linguagem c, arquivo TXT, alocação dinâmica, tipo abstrato de dados.

São João Del Rei, Abril de 2023

SUMÁRIO

INTRODUÇÃO

O objetivo do trabalho prático proposto é atender a empresa que busca trocar o servidor atual, que é fundamental fornecer suporte na entrega e consulta de mensagens de email para usuários cadastrados. Além disso, é necessário verificar se o funcionamento do sistema em diversas operações e diferentes situações, é correto.

Sendo assim, o sistema deve gerenciar o cadastro e a remoção de usuários, juntamente com a entrega e consulta do email. A linguagem de programação utilizada deverá ser em C e é indispensável a utilização de ferramentas importantes, os tipos abstratos de dados, além das estruturas de dados como fila, pilhas, listas encadeadas e listas por arranjo. Outrossim, o servidor será usado por milhões de pessoas, por isso, é fundamental que seja eficiente, com bom gerenciamento e aproveitamento da memória.

Para a realização dos testes, será utilizado o arquivo txt, possuindo um modelo padrão com as seguintes palavras “CADASTRA”, “REMOVE”, “ENTREGA” e “CONSULTA”. A função "cadastrar" cria uma caixa de entrada para o novo usuário, a "remover" retira o usuário do sistema, a “entrega” recebe um novo email destinado a um determinado usuário e a “consultar” lê o identificador de um usuário e retorna a mensagem de maior prioridade na caixa do usuário.

A utilização dos tipos abstratos de dados contribuem para que o código fique modular, flexível, seguro e de fácil entendimento, visto que a abstração permite que o programador não se preocupe com a implementação subjacente, o encapsulamento protege os dados de acesso não autorizado e permite a criação de interface mais simples e intuitiva, a modularidade facilita o gerenciamento de grandes programas e a reutilização do código em diversos projetos, e o polimorfismo proporciona o tratamento de maneira uniforme de objetos de diferentes tipos.

IMPLEMENTAÇÃO

Previamente no arquivo de cabeçalho são criadas as duas structs que atenderão o usuário durante a execução do código, a primeira struct contém o id do usuário, já a segunda contém a variável que vai conter o tamanho da lista e a que vai conter a capacidade.

Já na parte das structs do email, assim como o usuário, são criadas duas structs para conter suas informações, a primeira contém o id do email para sua localização, o id do destinatário para que ele possa ser enviado para a pessoa certa, sua prioridade perante os outros emails e o seu conteúdo, já a segunda struct contém todas as informações da primeira além de uma variável para conter o tamanho da lista de emails e uma para a capacidade.

Inicialmente na main são criadas duas listas por arranjo, uma para os usuários (listaU) e a segunda para os emails (listaE), onde também ficam contidos o id do destinatário do email, seu conteúdo e um número de 0 a 9 que indica sua prioridade perante as outras mensagens para o mesmo id.

Em seguida foi realizada a implementação do arquivo txt juntamente com as variáveis locais, as quais serão utilizadas durante a execução do código para o recebimento do conteúdo do arquivo.

Logo após é realizada a iniciação da leitura do arquivo txt, o qual é lido linha por linha, a primeira palavra da linha é lida como uma palavra de comando, a qual indica qual função será utilizada e, logo após, é lido qual o id que a função será realizada. O primeiro comando a ser verificado é o ‘CADASTRA’, onde é recebido o id que o usuário deseja cadastrar, esse id logo é atribuído a struct criada para o usuário e é chamada a função que o insere no fim da lista de usuários, a qual será explicada posteriormente na parte das funções. O segundo comando possível é o comando ‘REMOVE’, o qual recebe um id que o usuário deseja remover da lista, logo após, é chamada a função retira usuário para removê-lo.

O terceiro comando a ser verificado é o comando ‘CADASTRA’, o qual é utilizado para enviar os emails, seus parâmetros no txt são a palavra de comando, o id do destinatário, a prioridade da mensagem, seu conteúdo e a palavra ‘FIM’ para indicar o final da mensagem que é identificado por meio da comparação dos últimos

elementos da linha, logo após é chamada a função ‘caixa_nova_email()’ para criação da struct email com os dados da mensagem, em seguida é chamada a função que vai inserir o email no fim da lista dos emails.

O quarto e último comando a ser verificado é o comando ‘CONSULTA’, o qual é utilizado para consultar os email que existem na caixa de entrada do usuário, seus parâmetros são a palavra de comando e o id que a pessoa deseja checar, logo após a leitura, existe a chamada da função ‘consultar_email()’, e o email com maior prioridade é impresso.

Já na parte final do código, a leitura do arquivo txt é encerrada, são chamadas as funções ‘imprime_usuario() e ‘imprime_email()’ que imprimem os elementos remanescentes nas listas ao final da leitura do arquivo, logo após é realizada a liberação das variáveis alocadas dinamicamente, tanto com as funções ‘desaloca_lista_usuario()’ e ‘desaloca_lista_email()’, quanto com o ‘free()’ nas 3 variáveis que foram utilizadas na ‘main’.

No final é retornado ‘0’, indicando que o programa executou como esperado.

FUNÇÕES

-> `listaU *nova_lista_vazia_usuario()`

A função ‘nova_lista_vazia_usuario()’ não recebe parâmetros, seu papel principal é alocar o espaço necessário para a struct ‘listaU’, que no caso é a struct utilizada para a lista de usuários, em seguida, ela inicializa a lista de usuários como ‘NULL’, o tamanho e a capacidade como 0. Logo após, ela retorna um ponteiro ‘u’ para a nova lista de usuários vazia.

-> `listaE *nova_lista_vazia_email()`

A função ‘nova_lista_vazia_email()’, assim como a função acima, não recebe parâmetros, seu papel principal é alocar dinamicamente o espaço de memória necessário para struct ‘ListaE’ para criar uma lista vazia com um ponteiro ‘e’ apontando para ela, logo após é definido o conteúdo inicial das variáveis dessa struct, com a variável e mails recebendo NULL pois a lista está vazia, assim como as variáveis tamanho e capacidade recebendo o valor 0 pelo mesmo motivo, no final, a função retorna um ponteiro ‘e’.

-> usuario caixa_nova_usuario()

A função ‘caixa_nova_usuario()’ recebe como parâmetro os dados da struct ‘usuario’ e, assim, inicializa o item com os valores fornecidos pelo cliente, retornando o devido conjunto de dados para a ‘.main’.

-> email caixa_nova_email()

A função ‘caixa_nova_email()’, assim como a função acima, recebe como parâmetro os dados de sua respectiva struct, no caso a ‘email’ e inicializa o item com os valores fornecidos pelo cliente, retornando o devido conjunto de dados para a ‘.main’.

-> bool vazia_usuario(listaU *u)

A função ‘vazia_usuario()’ recebe como parâmetro a lista de usuários u, para verificar se ela está vazia ou não, caso ela esteja vazia retorna ‘TRUE’, caso ela não esteja vazia retorna falso.

-> bool vazia_email(listaE *e)

A função ‘vazia_email()’ é utilizada para verificar se a lista de email está vazia, ela recebe como parâmetro a lista de email e caso ela esteja vazia a função retorna ‘TRUE’, caso a lista não esteja vazia ela retorna falso.

-> void insere_no_fim_usuario(listaU *u, usuario x)

A função ‘insere_no_fim_usuario()’ recebe como parâmetro a lista u e o id do usuário contido na variável x, tal parâmetro representa o novo usuário a ser inserido na lista.

A função começa verificando se o id recebido já existe na lista de usuários. Para isso, ela percorre toda a lista comparando o id recebido com os ids que já estão presentes na lista. Caso encontre um id semelhante, a variável ‘cont’ recebe o valor 1, indicando que este id já foi cadastrado na lista e é impresso uma mensagem informando o usuário.

Caso não exista nenhum usuário com o id a ser inserido, o programa irá verificar se existe espaço na lista para o novo usuário. Se a lista ainda não foi alocada, o sistema irá alocar espaço para a lista, definindo a capacidade inicial "INICIAL". Caso

já existam usuários cadastrados, o sistema irá verificar se há espaço suficiente para adicionar o novo usuário na lista. Se não houver, o sistema irá realocar a lista, aumentando a capacidade para que haja memória disponível para o novo usuário.

Por fim, o novo usuário é adicionado no final da lista e uma mensagem de sucesso é exibida.

-> `void insere_no_fim_email(listaU *u, listaE *e, email x)`

A função ‘insere_no_fim_email()’ recebe como parâmetros a lista de usuários ‘u’, a lista de emails ‘e’ e também a variável x onde está contido o email.

Primeiramente, a função verifica se o id do destinatário existe na lista de usuários, caso ele não exista, uma mensagem é impressa para informar o usuário.

Se houver correspondência e o email do destinatário realmente estiver presente na lista de usuários, a função verificará se existe espaço na lista de emails para mais um, se não houver, a função realoca a lista, aumentando sua capacidade. Outrora, caso exista espaço na lista para o novo email, ele simplesmente é adicionado no final da lista e uma mensagem de sucesso é exibida.

A complexidade desse algoritmo é O(n), onde n é o número de usuários na lista de usuários (listaU). A complexidade se dá pelo fato de que a verificação da existência do ID destinatário envolve uma iteração pela lista de usuários, que pode ter no máximo n elementos. O restante do algoritmo é composto por operações de alocação de memória e atribuição de valores em tempo constante, por isso não afeta a complexidade geral.

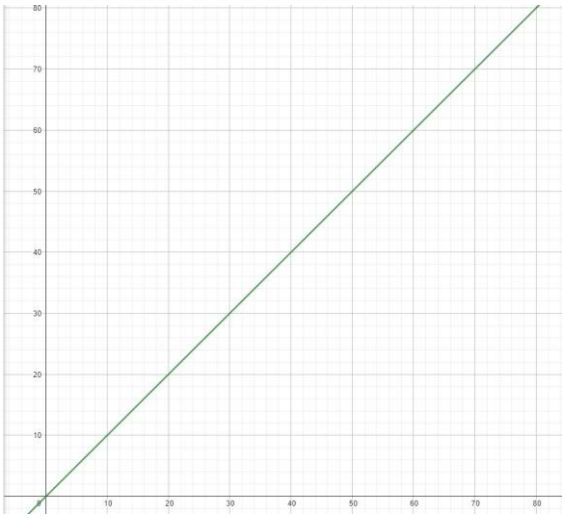

Gráfico de complexidade da função

-> `void consultar_email(listaE *e, int indice)`

A função ‘consultar_email()’ recebe como parâmetros a lista de emails e o id o qual o usuário deseja consultar a caixa de entrada.

A primeira coisa realizada pela função é a contagem de quantos emails não são destinados ao ID recebido, caso não exista nenhum email para o destinatário é exibido uma mensagem informando o usuário. Caso contrário, a função percorre a lista procurando pelo email com maior prioridade para o usuário, ao encontrar o email com maior prioridade, seu conteúdo é impresso na tela.

Em caso de houver dois emails com a prioridade mais alta, o algoritmo vai identificar qual deles foi recebido primeiro e imprimir o mesmo.

Por fim, a função remove o email lido da ‘listaE’, atualizando sua posição na lista e fazendo com que os emails abaixo subam uma posição.

-> `bool retira_usuario(listaU *u, listaE *e, int indice)`

A função ‘retira_usuário()’ recebe como parâmetros a lista de usuário ‘u’, a lista de emails ‘e’, o índice do usuário a ser removido. O papel dessa função é realizar a remoção do usuário da lista, assim como todos os emails em sua caixa de entrada.

Primeiramente, a lista é percorrida para verificar se o id passado como parâmetro realmente existe na lista de usuários, caso o usuário não exista uma mensagem de erro é exibida e a função retorna. Porém, caso o usuário exista, sua posição é salva para que ele seja removido, logo após, é verificado se existe algum email na caixa de entrada do usuário, para que caso exista eles sejam apagados pela função ‘retira_email()’.

Assim, o usuário é removido da lista, seus emails são apagados e uma mensagem informando quais emails foram apagados é impressa para o usuário.

Na função retira_usuario, existem três loops for aninhados e uma chamada de função retira_email dentro do segundo loop for, portanto a complexidade da função é O(n^4), onde n é o tamanho da lista de usuários ou de e-mails, dependendo de qual for maior.

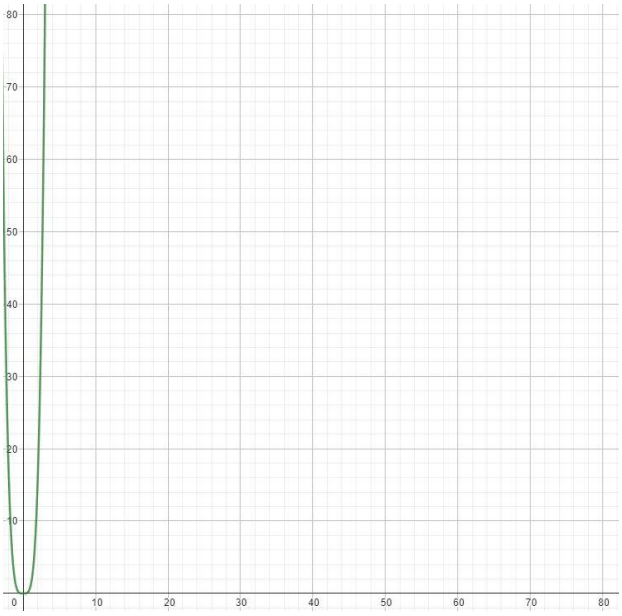

Gráfico de complexidade da função

-> `bool retira_email(listaE *e, int cont, int idIndice, email *it)`

A função ‘retira_email()’ é chamada unicamente dentro da função ‘retira_usuario()’, o seu papel dentro do programa é remover todos os emails na caixa de entrada de um usuário que está sendo removido.

Primeiramente, a função recebe como parâmetros a lista de email ‘e’, o contador que indica que existem emails para serem removidos, o índice do email para ser removido e um ponteiro para o email ‘it’.

Segundamente, a função verifica se o email solicitado existe na ‘listaE’, comparando o ‘idEmail’ do email na posição ‘idIndice’ com o ‘idEmail’ do email apontado pelo ponteiro "it". Se o email não existir ou a ‘listaE’ estiver vazia, a função exibe uma mensagem de erro e retorna ‘false’.

Caso o email exista, a função remove o email da ‘listaE’. Isso é feito por meio de um laço for que percorre a ‘listaE’ da posição do email a ser removido até a posição anterior à última, movendo cada email um índice para trás. Em seguida, o tamanho da ‘listaE’ é atualizado para refletir a remoção do email.

Se o valor de ‘cont’ for 0, a função exibe uma mensagem de sucesso indicando que o email foi removido. A função então retorna ‘true’.

-> `void imprime_usuario(listaU *u)`

A função ‘imprime_usuário()’ percorre toda a lista de usuários e imprime todos os id’s remanescentes

-> `void imprime_email(listaE *e)`

A função ‘imprime_email()’ percorre toda a lista de emails e imprime o id do email na lista, o id do destinatário, a prioridade do email e seu conteúdo.

-> `void desaloca_lista_usuario(listaU *u)`

A função ‘desaloca_lista_usuario()’ é a responsável por desalocar os dados da ‘listaU’, os quais foram alocados dinamicamente durante o código.

-> `void desaloca_lista_email(listaE *e)`

A função ‘desaloca_lista_email()’ é a responsável por desalocar os dados da ‘listaE’, os quais foram alocados dinamicamente durante o código.

RESULTADO E DISCUSSÕES

Ao decorrer do desenvolvimento do código, foram criados quatro conjuntos de dados, sendo eles os structs: ‘usuario’ que está conectado à struct ‘listaU’ e o ‘email’ que está conectado à struct ‘listaE’. Assim, em cada conjunto possui suas próprias variáveis para facilitar e ajudar na execução do sistema.

A ordenação dos dados é feita usando lista linear por meio de arranjos, sendo uma forma simples de interligar os elementos, é dinâmica (tamanho varia durante a execução) e pode ser percorrida em qualquer direção, além de não ser preciso alocar memória para saber o endereço anterior e/ou posterior.

Durante os testes foi utilizado dados diversos para verificar as funções de “CADASTRA”, “REMOVE”, “ENTREGA” e “CONSULTA”. Após as análises, todos os resultados obtidos foram satisfatórios, já que exibiram a saída esperada e foram capazes de executar seguindo o padrão desejado.

Em resumo, observamos que o tempo de processamento aumenta de forma linear com o número de dados inseridos, indicando que a implementação está seguindo um rumo esperado e não possui conflitos ou erros.

CONCLUSÃO

Após a criação da programação em C, podemos concluir que os tipos abstratos de dados são muito úteis para alcançar diversas tarefas, visto que é um modelo matemático que encapsula um modelo de dados e um conjunto de procedimentos.

Entretanto, um dos problemas encontrados durante a criação do código, envolve a modularização da programação, já que surgiram conflitos na leitura do arquivo txt com o uso da função ‘strtok’.

Junto a isso, a alocação dinâmica causou diversas complicações, pois no princípio não estávamos dando ‘free()’ em todos os ponteiros que haviam sidos criados, o que é necessário para liberação da memória, logo, foi utilizado o comando ‘-fsanitize=address’ para que fossem descobertos os “vazamentos” de memória.

A execução do trabalho foi essencial para aprimorar os conhecimentos sobre a alocação dinâmica, uma vez que tivemos que fazer pesquisa para poder encontrar os erros no código. Além disso, aplicar o conteúdo teórico aprendido na sala de aula, sobre os tipos abstratos de dados e algumas das estruturas de dados contribuiu para maior entendimento e fixação da matéria.

REFERÊNCIAS

- http://linguagemc.com.br/alocacao-dinamica-de-memoria-em-c/. Acesso em 20 de abr. de 2023.

- bit.ly/3AQIl5P. Acesso em 20 de abr. de 2023.

- **Slides e material didático** disponibilizado pelo professor Rafael Sachetto Oliveira. Acesso em 20 de abr. de 2023.