Universidade Federal
de São João del-Rei

Bianca Rodrigues Guedes, João Lucas Vilas Boas Faria e João Victor Ottoni Garcia

TRABALHO PRÁTICO 1

São João Del Rei - MG
2023

Bianca Rodrigues Guedes, João Lucas Vilas Boas Faria e João Victor Ottoni Garcia

TRABALHO PRÁTICO 1

Trabalho de Algoritmos e Estrutura de Dados II relacionado ao desenvolvimento de um algoritmo que trabalhe como um servidor de *e-mails* através de listas encadeadas em linguagem *C*.

Professor: Rafael Sachetto Oliveira

São João Del Rei - MG
2023

SUMÁRIO

1. INTRODUÇÃO

Este trabalho tem como objetivo apresentar o funcionamento e implementação de um servidor de *e-mails* através de listas encadeadas em linguagem *C*. O uso de listas encadeadas para execução deste servidor é benéfico em sua escalabilidade, visto que tal estrutura de dados permite que o sistema funcione bem com diversos números de entradas.

Serão apresentados, neste documento, os principais conceitos relacionados às listas encadeadas e seu uso no desenvolvimento do trabalho prático. Ademais, serão tratadas as funcionalidades do servidor, tal qual cadastramento de usuários, envio de *e-mails*, remoção de usuários, entre outros recursos aplicados.

Foram utilizados registros e funções pré-solicitadas para cumprir as especificações do trabalho. Além disso, fez-se necessário o uso de funções auxiliares para busca de usuários. Portanto, esta documentação é de suma importância para compreensão do funcionamento do servidor de *e-mails* através de listas encadeadas em linguagem *C*.

2. CONCEITOS

2.1 LISTAS ENCADEADAS E SEUS COMPONENTES

Recebe o nome de lista encadeada uma sequência de itens na memória do computador. Cada item armazenado na sequência é chamado de “nó” ou “célula” desta lista, que não é necessariamente contígua, tornando a muito dinâmica, sendo possível remover e inserir itens de forma simplificada.

Esse item nada mais é do que *struct*, ou registro, que é um tipo de dado que possibilita guardar variáveis sob um mesmo tipo de dado. Assim, cada célula contém um item de determinado tipo e o endereço para a próxima célula. Desse modo, a estrutura das células pode ser escrita como:

```
1. typedef struct item {
2. tipo: celula;
3. struct item *prox;
4. … (continuação da struct)
5. } item_t;
```

No qual “tipo” refere-se aos tipos de dados em *C*, como *char*, *float*, *int* etc e “item” e “item_t” referem-se, respectivamente, ao nome e apelido do registro.

Desse modo, a célula passa a ser tratada como um novo tipo de dado, por exemplo:

```
1. typedef struct item_t: celula_t;
```

Assim, torna-se possível criar uma célula e um ponteiro da seguinte forma:

```
1. item_t celula_1;
2.  item_t *p;
```

Note que o tipo de dado declarado é denotado pelo registro “item_t”. Portanto, se “p” é o endereço da célula, “p->celula” aponta para o conteúdo da célula e *“p->prox”* é o endereço da célula seguinte a “p”. Além disso, quando “p” trata-se do último elemento da lista encadeada, diz-se que “p->prox = NULL”, ou seja, o próximo elemento de “p” é nulo. A partir disso, basta criar funções que operem com a lista encadeada criada com as *structs*.

Dessa forma, é notório o bom desempenho quanto à escalabilidade e atuação dinâmica, o que faz do uso de lista encadeada o método mais eficaz para solucionar o problema apresentado para criação de um servidor de *e-mails*.

3. IMPLEMENTAÇÃO

Até esta seção, foram apresentados a definição de lista encadeada e seus componentes. Conseguinte, serão introduzidas as funções utilizadas, seu funcionamento e os resultados obtidos.

3.1 *STRUCTS* UTILIZADAS

3.1.1 *EMAIL_T*

Esse registro será responsável para armazenar informações contidas nos *e-mails*, tais quais a prioridade da mensagem (tipo *int*), o *e-mail* (tipo *char*, tamanho máximo pré definido como 1000 caracteres) e um ponteiro *prox* (tipo *struct email*), que será utilizado para percorrer esta lista de *e-mails*.

Código utilizado:

```
1. typedef struct email {
2. int prioridade;
3. char mensagem[max];
4. struct email *prox;
5. } email_t;
```

3.1.2 USUÁRIO

A segunda *struct* criada armazena os dados do usuário e ponteiros para percorrer a lista. O primeiro elemento é o *ID* do usuário, do tipo *int*, seguido por um ponteiro para o primeiro *e-mail,* do tipo *struct email_t*, que será abordado mais profundamente na função de remover *e-mails*, possuindo sua utilidade na identificação do primeiro *e-mail* da caixa de entrada do usuário. Além disso, assim como na *struct* anterior, foi criado um ponteiro próximo, do tipo *struct user*, que trabalhará percorrendo a lista de *IDs*.

Código utilizado:

```
1. typedef struct usuario {
2. int id;
3. email_t *primeiro_email;
4. struct usuario *prox;
5. } user;
```

3.1.3 LISTA

A terceira e última *struct* é utilizada para criar a lista. Dentro dela são armazenados dois ponteiros do tipo *user*, chamados de *primeiro* e *ultimo*, que serão aplicados nas funções para trabalhar com os *IDs*, diretamente apontando o primeiro e/ou último elemento, ou também com ponteiros para o próximo do primeiro elemento, por exemplo.

Código utilizado:

```
1. typedef struct {
2. user *ultimo, *primeiro;
3. } lista;
```

3.2 FUNÇÕES UTILIZADAS

3.2.1 INICIALIZA LISTA

Inicialmente foi criada uma função *inicializa_lista(lista *lista)*, na qual o parâmetro é um ponteiro para lista do tipo *struct lista* previamente criada. O objetivo dessa função é inicializar uma lista, ou seja, criar uma lista vazia para possibilitar a inserção de itens necessários. Nessa etapa, o primeiro e último elemento da lista é apontado para *NULL*. Código desenvolvido:

```
1. void inicializa_lista(lista *lista) {
2. lista->primeiro = NULL;
3. lista->ultimo = NULL; }
```

3.2.2 BUSCAR USUÁRIO

Ademais, foi desenvolvida uma função auxiliar *busca_usuario(lista *lista, int ID)*, que tem como parâmetro a lista e o usuário do tipo *int* para encontrar o usuário a ser utilizado nas demais funções que serão apresentadas. Nessa função foi criado um ponteiro auxiliar "*p"* do tipo *user*, que será essencial para percorrer a lista e buscar o *ID* solicitado. Esse ponteiro recebe o primeiro elemento da lista e a percorre utilizando o comando *while*, cuja condição será: enquanto o próximo elemento apontado por "*p"* for diferente de *NULL*, examina a lista fazendo comparações entre o *ID* solicitado e os *IDs* pertencentes à lista. A função retorna *NULL* se o *ID* não for encontrado. Caso contrário, a função retorna um ponteiro para o próprio *ID* na lista encadeada.

Código desenvolvido:

```
1. user *busca_usuario(lista *l, int ID){
2. user *p = l->primeiro;
3. while (p != NULL) {
4. if (p->id == ID) {
5. return p;  }
6. p = p->prox; }
7. return NULL; }
```

3.2.3 CADASTRAR USUÁRIO

A partir desse momento, serão apresentadas as funções pré estabelecidas pelo professor para a resolução do problema. A função *cadastra_usuario(lista *l, int ID)* tem como parâmetro a lista já criada e o *ID* do usuário que será cadastrado.

Antes de mais nada, a função *busca_usuario* é chamada para verificar se o usuário já existe na lista. Para isso, usa-se o comando *if* para verificar se o retorno da função é diferente de *NULL,* que significa que o *ID* solicitado já se encontra cadastrado. Assim, é impressa uma mensagem de erro, informando ao usuário que o *ID* mencionado já está na lista e não pode ser cadastrado novamente.

No entanto, caso a função retorne que o usuário ainda não existe na lista, cria-se um ponteiro **novo* do tipo *user* e faz um alocamento de memória para inserção do usuário, empregando a ferramenta *malloc*. Em seguida, a lista é percorrida através dos ponteiros criados na *struct lista*. Quando o primeiro elemento da lista é igual a *NULL*, significa que a

lista está vazia, então o usuário cadastrado é inserido e o ponteiro para o primeiro e o último elemento são apontados para o novo *ID*. Caso contrário, a lista segue sendo percorrida e o usuário é cadastrado após o último elemento já inserido, passando, dessa maneira a ser o último item da lista. Após ser concluído, é impresso uma mensagem de sucesso no cadastramento do *ID*.

Código desenvolvido:

```
1.  if (busca_usuario(l, ID) != NULL) {
2.  printf("ERRO: CONTA %d JA EXISTE\n", ID);
3.  return;}
4.  user *novo = (user *)malloc(sizeof(user));
5.  novo->id = ID;
6.  novo->prox = NULL;
7.  novo->primeiro_email = NULL;
8.  if (l->primeiro == NULL) {
9.  l->primeiro = novo;
10. l->ultimo = novo;
11. } else {
12. l->ultimo->prox = novo;
13. l->ultimo = novo; }
14. printf("OK: CONTA %d CADASTRADA COM SUCESSO\n", ID); }
```

3.2.4 REMOVER USUÁRIO

Na função *remove_usuario(lista *lista, int ID)*, o primeiro passo é fazer a função *busca_usuario* percorrer a lista procurando pelo *ID* informado e verificando se ele existe na lista. Se não existir, uma mensagem de erro informando o usuário que o usuário solicitado não existe é impressa. Caso contrário, são utilizados ponteiros auxiliares para percorrer tanto a lista com os usuários quanto a lista com os *e-mails,* buscando o *ID* e sua caixa de entrada. A função trabalha com os ponteiros da lista encadeada a fim de liberar o espaço na memória através do comando *free,* que remove os ponteiros auxiliares que apontavam para o *ID* solicitado e seus *e-mails*. Desse modo, foram removidos com sucesso o usuário e todos os *e-mails* enviados à ele.

Código desenvolvido:

```
1.  void remove_usuario(lista *l, int ID){
```

```
2.  user *usuario = busca_usuario(l, ID);
3.  if (!usuario) {
4.  printf("ERRO: CONTA %d NAO EXISTE\n", ID);
5.  return; }
6.  user *aux = l->primeiro;
7.  if (l->primeiro->id == ID) {
8.  l->primeiro = l->primeiro->prox;
9.  while (usuario->primeiro_email != NULL) {
10. email_t *aux_t = usuario->primeiro_email;
11. usuario->primeiro_email = usuario->primeiro_email->prox;
12. free(aux_t);
13. }
14. if (l->primeiro == NULL) {
15. l->ultimo = NULL;
16. }
17. free(aux);
18. } else {
19. while (usuario->primeiro_email != NULL) {
20. email_t *aux_t = usuario->primeiro_email;
21. usuario->primeiro_email = usuario->primeiro_email->prox;
22. free(aux_t);}
23. while (aux->prox->id != ID) {
24. aux = aux->prox; }
25. user *aux2 = aux->prox;
26. aux->prox = aux->prox->prox;
27. free(aux2);}
28. printf("OK: CONTA %d REMOVIDA\n", ID);}
```

3.2.5 ENTREGAR *E-MAILS*

A função nomeada *enviar_email(lista *l, int ID, char msg, int prioridade)* trabalha diretamente com o envio dos *e-mails*. O parâmetro *prioridade* recebe um número inteiro de 0 a 9 que determina a prioridade da mensagem a ser enviada, de maneira que quanto maior o número, mais alta é a prioridade e, assim, o *e-mail* deve ser enviado primeiro.

Inicialmente, a função *busca_usuario* é chamada para analisar se o destinatário do *e-mail* está cadastrado no servidor. Do mesmo modo que as funções apresentadas anteriormente, caso o usuário não seja encontrado, uma mensagem informando o usuário é imprimida. No entanto, se o destinatário for encontrado, um novo espaço na memória é alocado para o *e-mail* através do comando *malloc*. A função recebe as informações e as salva na lista a mensagem e a prioridade recebida. Posteriormente, são feitas comparações para identificar a prioridade dos *e-mails* e organizá-los a partir disso. Ademais, a palavra reservada "FIM" declarada na função *main* indica que a mensagem foi finalizada. Assim, o *e-mail* será enviado corretamente ao usuário.

Código desenvolvido:

```
1. void enviar_email(lista *l, int ID, char *msg, int prioridade) {
2. user *usuario = busca_usuario(l, ID);
3. if (!usuario) {
4. printf("ERRO: CONTA %d NAO EXISTE\n", ID);
5. return; }
6. email_t *novo_email = (email_t *)malloc(sizeof(email_t));
7. novo_email->prioridade = prioridade;
8. strcpy(novo_email->mensagem, msg);
9. novo_email->prox = NULL;
10. email_t *p = usuario->primeiro_email;
11. if (p == NULL || p->prioridade < prioridade) {
12. novo_email->prox = usuario->primeiro_email;
13. usuario->primeiro_email = novo_email; }
14. else {
15. while (p->prox != NULL) {
16. if (p->prox->prioridade < prioridade) {
17. break; }
18. p = p->prox; }
19. novo_email->prox = p->prox;
20. p->prox = novo_email;}
21. printf("OK: MENSAGEM PARA %d ENTREGUE\n", ID);}
```

3.2.6 CONSULTAR *E-MAILS*

A função *consultar_email(lista l, int ID)* inicia-se tal qual as funções já apresentadas, chamando a função *busca_usuario* para verificar se o usuário que se deseja consultar e a respectiva caixa de entrada existe na lista. Caso não exista, a mensagem de erro informando ao usuário é impressa.

Quando o usuário está presente, a função primeiramente confere se sua caixa de entrada está vazia, verificando se o que o primeiro *e-mail* aponta é igual a *NULL*. Se não estiver, imprime o primeiro *e-mail* definido pela organização dos *e-mails* por sua prioridade na função anterior, *envia_email*.

Código desenvolvido:

```
1. void consultar_email(lista *l, int ID) {
2. user *usuario = busca_usuario(l, ID);
3. if (!usuario) {
4. printf("ERRO: CONTA %d NAO EXISTE\n", ID);
5. return; }
6. if (usuario->primeiro_email == NULL) {
7. printf("CONSULTA %d: CAIXA DE ENTRADA VAZIA\n", ID);
8. return;}
9. printf("CONSULTA %d: %s\n", ID, usuario->primeiro_email->mensagem);
10. email_t *p = usuario->primeiro_email;
11. usuario->primeiro_email = p->prox;
12. free(p);
13. return;}
```

3.2.7 IMPRIME LISTA

Além das funções fundamentais para o funcionamento do servidor de *e-mails*, foi desenvolvida uma função para testes nomeada *imprime_lista(lista *l)*, que percorre a lista e, enquanto o próximo elemento for diferente de *NULL*, imprime o item. Dessa forma, a lista de *users* é imprimida.

Código desenvolvido:

```
1. void imprime_lista(lista *l){
2. user *p = l->primeiro;
```

```
3. while (p != NULL) {
4. printf("|%d\n", p->id);
5. p = p->prox;}}
```

3.2.8 *MAIN*

Por fim, a função *int main()* é responsável por receber e ler o arquivo *.txt*, além de chamar as demais funções que compõem o programa. Como as palavras que definem os comandos para as funções não ultrapassam 20 caracteres, a função foi desenvolvida para pegar os 20 primeiros caracteres como instrução. Foi utilizada a ferramenta de comparação de *strings* '*strcmp'*, que compara o comando presente no texto com as funções disponíveis. Assim, a função recebe o comando e o encaminha para a função que o satisfará. Ademais, foi declarada palavra reservada "FIM", de modo que, quando lida, a função entende que o *e-mail* está completo.

Código que declara a palavra "FIM" como encerramento do *e-mail*:

```
1. while(1){{ fscanf(file, "%s", palavra);
2. if ((strcmp(palavra, "FIM") == 0)) {
3. break; }} (...)
```

Também fez-se necessário adicionar nessa função linhas de código que dessem espaço entre as palavras lidas do arquivo *.txt* enquanto não chegassem até "FIM", de tal maneira:

```
1. (...) else if ((strcmp(msg, "") != 0)) {
2. strcat(msg, " ");}
3. strcat(msg, palavra);}
```

Exemplos de comando para cada função apresentada:

Cadastrar: CADASTRA ID. Exemplo: CADASTRA 5.

Remover: REMOVE ID. Exemplo: REMOVE 5.

Enviar *e-mail*: ENTREGA ID PRIORIDADE MENSAGEM FIM. Exemplo: ENTREGA 5 8 bom dia! FIM;

Consulta: CONSULTAR ID. Exemplo: CONSULTAR 5.

4. CONCLUSÃO

Conclui-se, então, que o processo de resolução do problema proposto evidenciou a escalabilidade e dinamicidade das listas encadeadas, pois comportou-se de maneira eficaz independente do número de entrada de dados. Portanto, após a execução de diversos testes, comprovou-se que as funções foram eficientes para realizar os comandos necessários do servidor de *e-mails*.

5. BIBLIOGRAFIA

FEOFILOFF, Paulo. Algoritmos em linguagem C. Elsevier Brasil, 2009.

SACHETTO, Rafael. Estruturas de Dados Básicas - Listas Lineares. Slides. Universidade Federal de São João del Rei, São João del Rei. Disponível em: Portal Didático. Acesso em: Abril de 2023.