Relatório do Exercício 3 (Parte 2)

SCC0223 - ESTRUTURAS DE DADOS I

Francisco Rosa Dias de Miranda

Setembro 2020

Listas Ligadas Não Ordenadas

Problema: implementar, em linguagem C, o acervo de filmes do sistema Netflix. Para isso, utilizaremos o Tipo Abstrado de Dado conhecido como Lista Ligada com Cabeça.

Tipos

Um dos parâmetros para a atividade, definido no arquivo `lista.h` , era o protótipo da estrutura de dados a ser utilizada:

```
typedef struct lista LISTA;
```

Este elemento do tipo **LISTA** é chamado *cabeça* da lista ligada, que é responsável por armazenar o ponteiro apontando para o primeiro elemento (nó) da lista, que será vinculado aos seguintes através do ponteiro ***prox**, presente na struct **NO**:

```
typedef struct no_{
    ITEM filme;
    NO *prox;
} NO;
```

A escolha desse tipo de lista nos favorecerá na realização de determinadas operações, assim como também nos obrigará a tomar algumas decisões de implementação, como veremos adiante.

Funções

Para este exercício, foi solicitado implementar os seguintes protótipos:

```
LISTA *lista_criar(void);
boolean lista_apagar(LISTA **l);
void lista_imprimir(LISTA *l);
boolean lista_cheia(LISTA *l);
boolean lista_vazia(LISTA *l);
boolean lista_inserir_pos(LISTA *l, ITEM filme, int pos);
boolean lista_remover(LISTA *l, int chave);
int lista_buscar(LISTA *l, int chave);
int lista_tamanho(LISTA *l);
```

A seguir, explicaremos melhor cada uma dessas funções.

1. lista_criar

Essa função aloca dinamicamente e retorna o endereço de memória de um elemento do tipo **LISTA**, e inicializa seu respectivo apontador de início com o valor `NULL`. Caso haja algum problema com a alocação, retorna `NULL`.

Como estamos utilizando somente uma lista por vez neste programa, para evitar vazamento de memória foi definido na `main.c` que, se caso já exista uma lista, apague-a antes uma nova ser criada.

2. lista_apagar

Essa função recebe como parâmetro o endereço de um apontador de um elemento do tipo **LISTA** e desaloca-o da memória, junto a todos possíveis nós a ele ligados.

Caso a exclusão seja feita com sucesso, a função retorna `TRUE`; caso a lista já esteja vazia, é retornado `FALSE`.

3. lista_inserir_pos

Função que recebe uma lista, e insere um **ITEM** na posição **pos** dela. Caso a lista não exista, é retornado `FALSE`; caso a posição não exista, é retornado `ERRO`; e se a inserção for feita corretamente, retorna `TRUE`.

Quando a inserção ocorre no início da lista, é necessário alterar o apontador **LISTA->inicio**, já em qualquer outra posição é o valor de um **NO->prox** que mudamos, por isso é necessário um tratamento especial no primeiro caso.

É interessante notar que, caso não estivéssemos trabalhando com um nó cabeça na lista, seria necessário uma estratégia diferente de inserir na primeira posição para que este método funcionasse, como passagem da lista por referência.

4 lista_remover_chave

Função que remove um **ITEM** da lista a partir de sua chave. Retorna `TRUE` se a remoção for feita; `FALSE` caso a chave não seja encontrada.

Este método percorre toda a lista, do primeiro até o último elemento, e, caso encontre a chave desejada, libera a memória do **NO** que a armazena, faz seu antecessor apontar para o sucessor do nó removido.

Novamente, é necessário aqui um tratamento especial para a remoção na primeira posição, por nesse caso envolver o apontador **LISTA->inicio**.

5. lista_buscar

Função que procura por uma chave dentro da lista, caso a encontre retorna sua posição; caso contrário retorna `ERRO`.

A maneira com a qual a chave é procurada dentro da lista é similar a utilizada na função **6.**, de complexidade linear.

6. lista_imprimir

Esse método recebe um elemento do tipo **LISTA** e imprime na saída padrão os itens armazenados em seus respectivos nós. Caso a lista esteja vazia, nada é impresso.

7. lista_tamanho

Função que retorna o tamanho da lista passada como argumento. Assim como as funções anteriores, esta também percorre linearmente a lista e vai contando quantos nós até o valor `NULL`, que indica o final.

8. lista_cheia

Não implementado, pois seriam necessários ***muitos*** elementos na lista para que o computador ficasse sem memória suficiente para alocar mais.

9. lista_vazia

Função lógica que recebe uma lista e retorna `TRUE` caso a lista esteja alocada, porém vazia (ou seja, se o apontador **LISTA->inicio** for `NULL`; `ERRO` caso a lista ainda não tenha sido criada pela função **1.**; e `FALSE` caso já hajam elementos na lista.

Interface do programa

A execução deste programa é relativamente simples. Ao iniciar, é solicitado um *input* da entrada padrão de **1-9**, sendo cada um desses números uma das funções acima descritas.

Após a execução de cada comando, é exibido na saída padrão o respectivo valor de retorno da função solicitada. Para os casos **3.**, **4.** e **5.**, é também solicitado o número de operações a serem feitas.

Casos de teste

Criar e remover lista

Ação	Retorno Esperado
Criação de uma nova lista	-
Criação de uma lista quando já existe uma	Apaga a lista antiga e cria uma nova
Apagar uma lista inexistente	FALSE
Apagar uma lista vazia	TRUE
Apagar uma lista existente	TRUE
Apaga a lista ao sair da aplicação	-

Entrada	Saída Esperada
2	0
1 2	1
1 3 1 111 Dinos 1 2	1 1

É interessante ressaltar que essa função poderia ocasionar um vazamento de memória se uma lista fosse criada em cima de uma já existente.

Como perderíamos o apontador para a lista antiga, a mesma ficaria inacessível, e após sucessivas iterações em um projeto de larga escala poderíamos esgotar a memória.

Para evitar esse problema, além dessa decisão a nível de implementação, utilizamos o *Valgrind* para verificar se há vazamentos em tempo de execução:

```
==14717== HEAP SUMMARY:
==14717==     in use at exit: 0 bytes in 0 blocks
==14717==   total heap usage: 12 allocs, 12 frees, 5,416 bytes allocated
==14717==
==14717== All heap blocks were freed -- no leaks are possible
```

Inserção na lista

Os casos de teste para inserção foram elaborados a partir da tabela abaixo:

Ação	Retorno esperado
Inserção no início da lista	TRUE
Inserção no final da lista	TRUE
Inserção em uma posição existente	TRUE
Inserção em uma posição inexistente	ERRO
Inserção em uma lista inexistente	FALSE

A partir daí, quando aplicados:

Entrada	Saída
3 2 111 Titanic 1 112 Mulan 2	0 0
1 3 5 114 Dragon 1 11512 Panic 2 11122 Honey 1 115 Saw 4 12312 Muppets 10	 1 1 1 1 -32000

Remoção a partir de uma chave

Para os testes de remoção, a seguinte tabela foi elaborada:

Ação	Retorno Esperado
Remoção no início da lista	TRUE
Remoção no final da lista	TRUE
Remoção de uma lista vazia	ERRO
Remoção de uma lista inexistente	ERRO
Remoção de uma chave inexistente	FALSE

Resultando em:

Entrada	Saída
4 1 123123	-32000
1 4 1 112	-32000
3 4 111 Titanic 1 112 Mulan 2 123 Darling 1 113 Annabelle 4 4 3 123 111 1233223 6	1 1 1 1 1 1 0 112 Mulan 113 Annabelle

Busca de uma chave

Elaboramos os testes de busca conforme ações descritas a seguir:

Ação	Retorno Esperado
Busca de uma chave na primeira posição	1
Busca de uma chave na posição n	n

Ação	Retorno Esperado
Busca em uma lista vazia	`ERRO`
Busca em uma lista inexistente	`ERRO`
Busca de uma chave inexistente	`FALSE`

Os casos de teste elaborados foram:

Entrada	Saída
1 3 11 111 Titanic 1 112 Mulan 2 123 Darling 3 113 Annabelle 4 114 Dragon 5 11512 Panic 6 11122 Honey 7 115 Saw 8 1156 Hunter 9 135677 Tango 10 112311 Joker 11 5 10 111 112 135677 115 112311 1444 1000 10 23 0	 1 1 1 1 1 1 1 1 1 1 11 2 10 8 11 0 0 0 0 0
5 1 10	-32000

Entrada	Saída
1 5 1 100 0	-32000

Impressão da lista

Os casos de teste para a impressão foram levantados a partir das seguintes ações:

Ação	Retorno Esperado
Impressão de uma lista inexistente	-
Impressão de uma lista vazia	-
Impressão de uma lista populada	Elementos da lista

Os casos de teste realizados foram:

Entrada	Saída
6	
1 6	
1 3 4 111 Titanic 1 112 Fast 2 115 Girls 3 113 Spider 4 6	1 1 1 1 111 Titanic 112 Fast 115 Girls 113 Spider

Tamanho de uma lista

Para o tamanho da lista, propomos as seguintes ações a serem testadas:

Ação	Retorno Esperado

Ação	Retorno Esperado
Verificar o tamanho de uma lista inexistente	`FALSO`
Verificar o tamanho de uma lista vazia	`0`
Verificar o tamanho de uma lista com `n` elementos	`n`

A partir daí, rodamos os seguintes testes:

Entrada	Saída
7	0
1 7	0
1 3 4 111 Avatar 1 112 Tango 2 124 Nemesis 3 1134 Tarzan 4 7	 1 1 1 1 4

Verificação de lista vazia

As ações que direcionaremos os casos de teste são:

Ação	Retorno Esperado
Verificar se uma lista inexistente está vazia	`ERRO`
Verificar se uma lista vazia está vazia	`TRUE`
Verificar se uma lista cheia está vazia	`FALSE`

Os casos de teste propostos foram:

Entrada	Saída
9	-32000

Entrada	Saída
1 9	1
1 3 4 111 Avatar 1 112 Tango 2 124 Nemesis 3 1134 Tarzan 4 9	 1 1 1 1 0

Conclusão

O programa apresentado teve o desempenho esperado em todos os casos de teste aqui apresentados. Agora, discutiremos brevemente algumas caracteristicas do tipo de implementação de TAD aqui utilizado.

O melhor caso para as funções de inserção, remoção e busca é quando o conteúdo está nas primeiras posições da lista. Contudo, é necessária a passagem da lista por referência para a função de exclusão.

O pior caso é quando temos uma lista com muitos elementos e queremos efetuar uma operação no final dela, aumentando o custo computacional, pois é necessário percorrer linearmente toda a lista para efetuar tais operações.

É interessante comentar também que a utilização de um nó cabeça faz com que a recursão só seja possível através de funções auxiliares, por isso não decidimos utilizás-la nesse projeto, mesmo com o caráter eminentemente recursivo das listas ligadas.

Um outro ganho poderia ser conseguido com a utilização de listas duplamente ligadas - neste caso, as operações que percorrem a lista não necessitariam de um ponteiro auxiliar para guardar o nó que foi percorrido anteriormente.