Universidade Federal de Uberlândia
Instituto de Matemática e Estatística
Disciplina: Computação Avançada
Profa. Dra. Christiane R. S. Brasil

LISTAS ENCADEADAS EM C

Daniel Barreto de Oliveira
Henrique Tomaz Gonzaga
Matheus de Moraes Neves

14 de novembro de 2024

Sumário

1 Introdução **2**

2 Metodologia **3**

3 Resultados e Discussão **5**
3.1 Definição de células 5
3.2 Cabeça de uma lista encadeada 5
3.3 Funções para manipulação de células 6
3.4 Funções para busca e manipulação de células 8
3.5 Resolução dos Exercícios 11
3.5.1 Exercício 2.1: Contagem de Células 11
3.5.2 Exercício 3.1: Correção Função de Busca 11
3.5.3 Exercício 3.6: Verificação de Igualdade Entre Listas 12

4 Conclusão **13**

A Impressão de Listas Encadeadas **14**

B Limpeza de Listas Encadeadas **15**

Capítulo 1

Introdução

As listas encadeadas constituem uma das estruturas de dados lineares e dinâmicas. Assim como a pilha e a fila, são fundamentais em Ciência da Computação, por serem amplamente utilizadas para a organização e manipulação eficiente de um conjuntos de dados.

Uma lista encadeada é composta por uma sequência de células, também chamadas de "nós", cada uma contendo o seu conteúdo e um ponteiro apontando para o endereço da próxima célula da lista. Neste estudo dirigido, baseado em: Paulo Feofiloff (2018), acessado em novembro de 2024, serão apresentados e explorados os conceitos fundamentais das listas encadeadas, incluindo sua estrutura, a definição da cabeça da lista e operações básicas como inserção, remoção e busca.

Serão abordadas diferentes formas de implementação, ilustradas pelos códigos de busca e remoção, busca e inserção, além da resolução dos exercícios propostos. O objetivo deste relatório é fornecer uma compreensão do funcionamento das listas encadeadas e demonstrar como essas operações podem ser aplicadas de maneira eficiente em alguns cenários.

Capítulo 2

Metodologia

Para exemplificar as funções e procedimentos, as seguintes bibliotecas terão funções suas utilizadas dentro da `int main()`:

Código 2.1: Bibliotecas Utilizadas

```
1 #include <stdio.h>
2 #include <stdlib.h>
3 #include <locale.h>
```

As bibliotecas declaradas são suficientes para o funcionamento dos códigos que serão apresentados.

Primeiramente, é necessário criar uma estrutura para as células de uma lista encadeada e outra para a cabeça de uma lista encadeada. Para essa implementação, as estruturas utilizadas serão as seguintes:

Código 2.2: Apresetação das Estruturas Utilizadas

```
1 typedef struct reg celula;
2 typedef struct ListaEncadeada;
```

A partir dessas estruturas é criada uma função que inicializa a cabeça de uma lista encadeada.

Código 2.3: Apresentação da Função para Criar uma Lista Encadeada

```
1 ListaEncadeada* cria_lista();
```

No presente relatório serão utilizadas células do tipo `int`, como visto anteriormente (Código 2.2). Dessa forma, ainda que a estrutura `ListaEncadeada` e a função `cria_lista` Código 2.3) dependam apenas do tipo declarado na estrutura `celula`, o mesmo não pode ser dito das funções e procedimentos seguintes que serão apresentados. Existem funções/procedimentos análogos aos aqui presentes, nos quais utilizam outros tipos de células, como por exemplo o tipo `char`.

Em uma lista encadeada é importante que se tenham meios de adicionar e remover valores. Para isso, existem procedimentos para a manipulação de células:

Código 2.4: Apresentação de Procedimentos para a Manipulação de Células

```
void insere (int valor, ListaEncadeada *lista);
void remove (celula *endereco);
```

Outro item fundamental é um meio para a busca de valores na lista. Além disso, à fim de facilitar alguns processos, podem ser implementados procedimentos que combinem busca com manipulação de célula (Código 2.4). Nesse relatório, estes serão os respectivos procedimentos:

Código 2.5: Apresentação de Funções e Procedimentos para Busca e Manipulação de células

```
celula* busca (int valor, ListaEncadeada *lista);
void busca_e_remove (int valor, ListaEncadeada *lista);
void busca_e_insere(int x, int y, ListaEncadeada *lista);
```

Por fim, as seguintes funções fazem parte das respostas dos exercícios 2.1 e 3.6:

Código 2.6: Apresentação das Funções dos Exercícios 2.1 e 3.6

```
int contar_celulas_it (ListaEncadeada *lista);
int contar_celulas_rec (celula *p);

int listas_iguais_it (ListaEncadeada *lista1, ListaEncadeada *
   lista2);
int listas_iguais_rec (celula *p1, celula *p2);
```

Todas essas estruturas, funções e procedimentos serão definidas no Capítulo 2. Assim como, exemplos de utilização dos itens mencionados serão apresentados no decorrer do capítulo. Para isso, será considerado em todos os exemplos a utilização das bibliotecas referidas (Código 2.1) e da seguinte declaração dentro da `main`:

Código 2.7: Declaração Padrão para Exemplos

```
int main() {
    ListaEncadeada *lista = cria_lista();

    setlocale (LC_ALL, "pt_BR.UTF-8");

    return 0;
}
```

Os os exemplos estarão entre a função `setlocale` e `return 0;`

Além dessa definição padrão para exemplos (Código 2.7), foram disponibilizados nos Apêndices A e B dois procedimentos para a facilitação da verificação desses exemplos. O primeiro dos apêndices contém um procedimento para a impressão de uma lista encadeada, já o segundo apêndice contém um procedimento para a limpeza de uma lista encadeada. Suas utilizações foram omitidas dos exemplos por questões de didática.

Capítulo 3

Resultados e Discussão

Prosseguindo para os objetivos deste estudo dirigido, serão definidos as estruturas de uma lista encadeada, juntamente à construção de funções que efetuem as operações básicas desta lista e, em seguida, sua aplicação nos exercícios propostos.

3.1 Definição de células

Para a criação de uma lista encadeada, em primeiro lugar, uma estrutura de célula precisa ser feita. Para isso, é preciso criar a estrutura de um nó da lista encadeada, representado cada elemento dela, ondem contenha seu respectivo valor e um ponteiro que possa ser direcionado para um próximo elemento. A implementação utilizada foi a seguinte:

Código 3.1: Definição da Estrutura de Células

```
1 typedef struct reg {
2     int          conteudo;    // Armazena o valor do elemento.
3     struct reg *prox;       // Ponteiro para o próximo nó da lista
          .
4 } celula;
```

Nesse código o valor ficará presente na variável `conteudo`, do tipo `int` e o ponteiro que pode direcionar para uma próxima célula foi nomeado de `prox`.

3.2 Cabeça de uma lista encadeada

Após a definição da estrutura de células, é criada uma estrutura para representar a cabeça da lista encadeada, que contém o ponteiro para o primeiro elemento da lista, sem conteúdo próprio. Segue abaixo sua implementação:

Código 3.2: Definição da Estrutura de Listas Encadeadas

```
1 typedef struct {
2     celula *inicio;     // Ponteiro para o primeiro nó da lista.
3 } ListaEncadeada;
```

Para uma melhor utilização de uma lista encadeada é interessante implementar facilitadores, tendo em vista alguns cuidados que precisam ser tomados quando uma nova lista

é feita. Uma função que cria e inicializa uma lista encadeada com cabeça, retornando um ponteiro para a lista recém-criada é algo recomendável. Para isso, foi utilizado a seguinte função:

Código 3.3: Definição da Função de Criação de Listas Encadeadas

```
1 ListaEncadeada* cria_lista () {
2     ListaEncadeada *lista = (ListaEncadeada*) malloc (sizeof (
          ListaEncadeada));
3     lista->inicio = NULL;
4     return lista;
5 }
```

Nessa implementação, a linha 2 do código é responsável por criar corretamente uma `ListaEncadeada`. Utilizando-se a função `malloc` para ter um ponteiro que aponte para um bloco na memória que caiba seu argumento, no caso `sizeof (ListaEncadeada)`. Já a função `sizeof` retorna o tamanho da estrutura `ListaEncadeada`. Combinando esses itens, cria-se uma lista encadeada e, após isso, é atribuído `NULL` ao `inicio`, retornando o ponteiro criado.

Tanto os códigos dessa seção quanto da seção de 'Definição de células' (Seção 3.1) foram exemplificados pelo código de declaração padrão para exemplos (Código 2.7). Desta forma, não será necessário exemplificá-los novamente, visto que já foram evidenciadas as utilizações desses itens nos Códigos 3.1, 3.2 e 3.3.

3.3 Funções para manipulação de células

Após definir a estrutura de uma lista montada, podemos propor uma função que insere um novo nó no início da lista encadeada, recebendo o ponteiro para a lista e o valor a ser inserido.

Código 3.4: Definição da Função Insere

```
1 void insere (int valor, ListaEncadeada *lista) {
2     celula *nova = (celula*)malloc(sizeof(celula));
3     nova->conteudo = valor;
4     nova->prox = lista->inicio;
5     lista->inicio = nova;
6 }
```

Seu funcionamento é de fácil compreensão: a célula `nova` recebe em seu `conteudo` o `valor` informado. Em seguida, sua célula sucessora se torna o `inicio` da lista, por consequência, é definida a célula `nova` como o `inicio` da lista. Sua utilização é exemplificada a seguir:

```
1     insere (10, lista);
2     insere (20, lista);
3     insere (30, lista);
```

Nesse código, foi criada uma lista com a sequência $(30, 20, 10)$.

Além de um método para inserir novos nós, é igualmente importante contar com uma função para sua remoção. Este procedimento remove uma célula sucessora, recebendo

como parâmetro o ponteiro para a célula anterior àquela que será removida. Essa abordagem garante que a cabeça da lista nunca seja removida. A implementação do procedimento está apresentada logo abaixo:

Código 3.5: Definição da Função Remove

```
void remove_cel (celula *p) {
    celula *lixo;
    lixo = p->prox;
    p->prox = lixo->prox;
    free (lixo);
}
```

Seu funcionamento segue o mesmo princípio do procedimento `insere`. Primeiramente, o procedimento atribui à variável `lixo` a próxima célula daquela recebida como parâmetro `p`. Em seguida, o ponteiro `proximo` da célula `p` é sobrescrito com o valor do ponteiro `proximo` da célula `lixo`; ou seja, o próximo do próximo é atribuído.

No entanto esse procedimento não é prático para todos os casos, pois exige que o usuário forneça um endereço ao invés de um valor. Para resolver isso, na Seção 3.4, será apresentada uma maneira de localizar o endereço de uma célula que contenha um valor especificado. A seguir, é mostrado um exemplo de utilização desse procedimento para remover o penúltimo valor adicionado:

```
    insere (10, lista);
    insere (20, lista);
    insere (30, lista);

    remove_cel (lista->inicio);
```

Apesar das limitações do procedimento `remove_cel`, pode-se criar um código funcional ao combiná-lo com o procedimento de inserção de valores (`insere`). O seguinte programa oferece três opções: adicionar um valor, sobrescrever o último valor adicionado e parar de inserir novos valores. O código é o seguinte:

```
    char op = 'N';
    int numero;

    while (op == 'N' || op == 'R') {
        printf ("Digite um inteiro para ser adicionada à lista: "
           );
        scanf ("%d", &numero);

        insere (numero, lista);

        if (op == 'R')
            remove_cel (lista->inicio);

        printf ("\nEscolha uma das seguite opções:\n");
        printf ("N) Adcionar novo valor.\n");
```

```
15          printf ("R) Reescrever último valor adcionado.\n");
16          printf ("Digite qualquer outra tecla para não inserir
                mais valores.\n");
17
18          scanf (" %c", &op);
19      }
```

3.4 Funções para busca e manipulação de células

Tendo em vista as limitações práticas de se utilizar apenas os procedimentos `insere` (Código 3.4) e `remove_cel` (Código 3.5), torna-se interessante implementar meios para realizar outras tarefas. Assim, o procedimento a seguir busca um valor em uma lista encadeada: ele recebe o ponteiro para a lista e o valor a ser buscado, retornando o ponteiro para o nó que contém o valor ou `NULL` caso o valor não seja encontrado.

Código 3.6: Definição da Função Busca

```
1  celula* busca (int valor, ListaEncadeada *lista) {
2      celula *p = lista->inicio;
3      while (p != NULL && p->conteudo != valor) {
4          p = p->prox;
5      }
6      return p; // Retorna o nó ou NULL se não encontrar
7  }
```

Esse procedimento é simples e eficiente. Seu funcionamento é o seguinte: um ponteiro é criado para o início da lista e, por meio do laço de repetição `while`, ele percorre toda a lista até encontrar o valor especificado; ou até o fim da lista, caso o valor não seja encontrado. Ao final do processo, o ponteiro retornado estará apontando para a célula que contém o valor ou `NULL`, caso o valor não exista. Embora evidente, é importante destacar que esse processo identifica apenas a ocorrência mais recente do valor, caso ele apareça mais de uma vez na lista.

Com a implementação do procedimento de busca, é possível combiná-lo com o procedimento `remove_cel` (Código 3.5) para permitir a remoção de um valor específico. O código para esse processo é o seguinte:

Código 3.7: Exemplo do Uso da Função Busca

```
1      insere (10, lista);
2      insere (20, lista);
3      insere (30, lista);
4
5      celula *removerProx = busca (lista, 20);
6      remove_cel (removerProx);
```

Neste código, será removida a célula com o conteúdo 10, pois ela é a que vem logo após a célula que contém o valor 20. Portanto, para remover uma célula, é necessário informar a célula predecessora. Vale ressaltar que as linhas 5 e 6 do Código 3.7 poderiam ser

substituídas pela seguinte linha: `remove_cel (busca (20, lista));`, mas essa alteração não foi feita por questões didáticas.

Algo mais interessante seria um procedimento que buscasse um valor e removesse sua célula da lista, pois isso permitiria retirar o último valor adicionado ou até mesmo a cabeça da lista. Além disso, não seria necessário informar a localização do valor anterior. Assim, este procedimento busca um valor na lista e o remove, caso seja encontrado. Ele recebe o `valor` a ser procurado e a `lista` onde a busca será realizada.

Código 3.8: Definição da função de busca e remoção

```
void busca_e_remove (int valor, ListaEncadeada *lista) {
    celula *p = lista->inicio;
    celula *anterior = NULL;

    while (p != NULL && p->conteudo != valor) {
        anterior = p;
        p = p->prox;
    }

    if (p != NULL) { // Nó encontrado
        if (anterior == NULL) { // Primeiro nó é o nó a ser
           removido
            lista->inicio = p->prox;
        } else {
            anterior->prox = p->prox;
        }
        free (p);
    }
}
```

Por se tratar de um procedimento que realiza duas tarefas (busca e remoção), sua explicação será dividida em duas partes.

Primeiramente, são criados dois novos ponteiros, `p` e `anterior`, uma vez que a estrutura utilizada (Código 3.1) não possui um ponteiro para a célula antecessora. Em seguida, ambos os ponteiros percorrem a lista da mesma forma que o procedimento 3.6. Após a busca, se o valor informado for encontrado, caso ele esteja no início da lista, o ponteiro `inicio` da estrutura `ListaEncadeada` (Código 3.2) passará a apontar para o "nó"seguinte. Caso contrário, o ponteiro `anterior` passará a apontar para a célula sucessora da célula `p`.

Por ser um procedimento bastante prático, ele apresenta demonstrações claras, como o seguinte código:

```
   insere (10, lista);
   insere (20, lista);
   insere (30, lista);

```

```
5     busca_e_remove (30, lista);
```

Com ele é possível remover o valor mais recente adicionado à lista.

Outro procedimento útil é o de adicionar um valor em um local específico de uma lista encadeada. Este procedimento recebe uma lista e insere o valor `x` imediatamente antes da célula que contém o valor `y`. Se nenhuma célula contém `y`, o valor `x` é inserido no final da lista. Sua implementação é a seguinte:

Código 3.9: Definição da Função de Busca e Inserção

```
1  void busca_e_insere(int x, int y, ListaEncadeada *lista) {
2      celula *p, *q, *nova;
3      nova = malloc(sizeof(celula));
4      nova->conteudo = x;
5
6      p = lista->inicio;  // Começa no início da lista
7      q = p->prox;        //define o ponteiro para o próximo
8
9      while (q != NULL && q->conteudo != y) {
10         p = q;
11         q = q->prox;
12     }
13
14     nova->prox = q;
15     p->prox = nova;
16 }
```

De maneira análoga à explicação do procedimento `busca_e_remove` (Código 3.8), o procedimento de busca e inserção pode ser explicado em duas partes.

A primeira parte do funcionamento do procedimento `busca_e_insere` é a criação de três novos ponteiros: `p`, `q` e `nova`, que serão usados para manipular as células. O ponteiro `nova`, definido na linha 3, segue o mesmo princípio da função `cria_lista` (Código 3.3), linha 2; e é na célula apontada por `nova` que o valor `x` será armazenado. Quanto aos ponteiros `p` e `q`, tanto sua definição quanto sua utilização no laço de repetição `while` (linhas 6-7 e 9-12, respectivamente) são análogas aos procedimentos de busca mencionados anteriormente, como nos Códigos 3.6 e 3.8. Em resumo, os ponteiros `p` e `q` irão percorrer toda a lista até encontrar o valor `y` ou até o final da lista, para então inserir a célula `nova` entre eles.

Apesar de sua certa engenhosidade, o procedimento `busca_e_insere` possui sua utilização bem simples. Segue abaixo uma aplicação:

```
1     insere(10, lista);
2     insere(20, lista);
3     insere(40, lista);
4
5     busca_e_insere(30, 20, lista);
```

Na aplicação mostrada é criada uma lista com a seguinte sequência $(40, 20, 10)$ e por meio do procedimento `busca_e_insere` ela é transformada em uma lista com a sequência $(40, 30, 20, 10)$.

3.5 Resolução dos Exercícios

3.5.1 Exercício 2.1: Contagem de Células

O primeiro exercício propõe criar uma função que conte o número de elementos em uma lista encadeada, implementando duas versões: uma iterativa e outra recursiva.

Esta é uma função iterativa que conta o número de células (nós) na lista encadeada. Recebendo o ponteiro para a lista e retornando o número total de células.

Código 3.10: Função Iterativa para Contagem de Células

```
int contar_celulas_it (ListaEncadeada *lista) {
    int contador = 0;
    celula *p = lista->inicio;
    while (p != NULL) {
        contador++;
        p = p->prox;
    }
    return contador;
}
```

Esta é uma função recursiva que conta o número de células (nós) na lista encadeada. Recebendo um ponteiro para o primeiro nó e retornando o número total de células.

Código 3.11: Função Recursiva para Contagem de Células

```
int contar_celulas_rec (celula *p) {
    if (p == NULL) return 0;
    return 1 + contar_celulas_rec (p->prox);
}
```

3.5.2 Exercício 3.1: Correção Função de Busca

Agora, para o segundo exercício, é apresentada a função abaixo que promete ter o objetivo de verificar se um inteiro x pertence a uma lista encadeada. O exercício pede uma crítica ao código.

```
celula *busca (int x, celula *le) {
   celula *p = le;
   int achou = 0; //Variável boolenada de sinalização
   while (p != NULL && !achou) { //Sugestão: while(p!=NULL)
      if (p->conteudo == x) achou = 1; //Sugestão: if (p->
         conteudo == x) return p;
      p = p->prox; }
   if (achou) return p; //Sugestão: retrun NULL;
   else return NULL;
}
```

Contudo, é possível observar o uso de variáveis booleanas de sinalização "achou"no código. Como resultado, o código acaba ficando mais complexo do que o necessário, com o laço `while` realizando uma operação adicional, mesmo após o objeto "x"já ter sido encontrado na lista, ou seja, na linha 7, a função não irá retornar o valor x buscado.

3.5.3 Exercício 3.6: Verificação de Igualdade Entre Listas

Por fim, o terceiro exercício solicita uma função que verifique se duas listas encadeadas são iguais com uma versão recursiva e outra iterativa.

Esta é uma função iterativa que verifica se duas listas encadeadas são iguais. Recebendo ponteiros para duas listas e retornando 1 se forem iguais, ou 0 se não forem.

Código 3.12: Função Iterativa para Ver Igualdade de Listas

```
1  int listas_iguais_it (ListaEncadeada *lista1, ListaEncadeada *
       lista2) {
2      celula *p1 = lista1->inicio;
3      celula *p2 = lista2->inicio;
4      while (p1 != NULL && p2 != NULL) {
5          if (p1->conteudo != p2->conteudo) return 0;
6          p1 = p1->prox;
7          p2 = p2->prox;
8      }
9      return p1 == NULL && p2 == NULL;
10 }
```

Esta é uma função recursiva que verifica se duas listas encadeadas são iguais. Recebendo ponteiros para os primeiros nós das duas listas e retornando 1 se forem iguais, ou 0 se não forem.

Código 3.13: Função Recursiva para Ver Igualdade de Listas

```
1  int listas_iguais_rec (celula *p1, celula *p2) {
2      if (p1 == NULL && p2 == NULL) return 1;  // Ambas listas
           terminaram
3      if (p1 == NULL || p2 == NULL) return 0;  // Uma terminou
           antes da outra
4      if (p1->conteudo != p2->conteudo) return 0;
5      return listas_iguais_rec (p1->prox, p2->prox);
6  }
```

Capítulo 4

Conclusão

Em resumo, é possível concluir que listas encadeadas ofereceram flexibilidade e eficiência em operações de inserção e remoção de elementos, especialmente em cenários em que o tamanho da estrutura não é conhecido previamente ou varia com frequência. A criação da cabeça da lista é uma etapa importante que facilita o procedimento dessas operações básicas. Apesar de apresentar alguns empecilhos para o usuário - como no código de remover uma célula `remove_cel` (Código 3.5) - , as funções associadas a esse tipo de estrutura de dados demonstraram uma característica positiva de interdependência. É possível integrar forma eficiente e superar essas dificuldades iniciais, como é o caso da função busca e remoção `buesca_e_remove_cel` (Código 3.8).

De maneira adicional, foi criado diversas funções pelos exercícios propostos, sendo testados os métodos iterativos e recursivos, além de observar a ineficiência de um código utilizando variáveis booleanas de sinalização.

Por fim, as listas encadeadas apresentam algumas desvantagens, como o maior consumo de memória devido à criação da cabeça da lista e ao armazenamento de ponteiros adicionais; além da limitação na busca de elementos que aparecem mais de uma vez em uma mesma lista, como apontado na função busca `busca_cel` (Código 3.6). No entanto, em contextos onde operações de inserção e remoção são mais frequentes e o espaço de memória não é uma restrição crítica, as listas encadeadas oferecem vantagens consideráveis.

Apêndice A

Impressão de Listas Encadeadas

Segue abaixo a implementação de um procedimento para impressão de uma lista encadeada (Código 3.2):

```
1 void imprime_lista (ListaEncadeada *lista) {
2     celula *p = lista->inicio;
3     while (p != NULL) {
4         printf("%d -> ", p->conteudo);
5         p = p->prox;
6     }
7     printf("NULL\n");
8 }
```

Apêndice B

Limpeza de Listas Encadeadas

Segue abaixo a implementação de um procedimento para liberar todas as células de uma lista encadeadas (Código 3.2):

```
void limpa_lista(ListaEncadeada *lista) {
    celula *p = lista->inicio;
    while (p != NULL) {
        celula *temp = p;
        p = p->prox;
        free(temp);
    }
    lista->inicio = NULL;
}
```

Referências Bibliográficas

Paulo Feofiloff. *Site Projetos de Algoritmos*. 2018. Disponível em: <https://www.ime.usp.br/~pf/algoritmos/aulas/lista.html#cells>. Acesso em: 09/11/2024.