



INSTITUTO FEDERAL DE EDUCAÇÃO, CIÊNCIA E TECNOLOGIA BAIANO
TECNOLOGIA EM ANÁLISE E DESENVOLVIMENTO DE SISTEMAS

ANDREAS GUNTHER MATOS LEAL

DAVI MONTEIRO CARVALHO

MARCOS HENRIQUE BRITO SOUZA

TRABALHO - LISTAS

CAMPUS GUANAMBI

2025

ANDREAS GUNTHER MATOS LEAL
DAVI MONTEIRO CARVALHO
MARCOS HENRIQUE BRITO SOUZA

TRABALHO - LISTAS

Trabalho apresentado como requisito parcial de avaliação do componente curricular Estrutura de Dados do Curso Superior de Tecnologia em Análise e Desenvolvimento de Sistemas, 2 ° período, do Instituto Federal de Educação, Ciências e Tecnologia - Campus Guanambi.

Orientador: Prof. Reinaldo Monteiro Cotrim

CAMPUS GUANAMBI

2025

ÍNDICE DE FIGURAS

Figura 1 – Exemplo de inserção em listas encadeadas.....	10
Figura 2 – Exemplo de busca em lista encadeada.....	11
Figura 3 – Processo de remoção de nós em listas encadeadas.....	12
Figura 4 – Exemplo de exibição em lista encadeada.....	13
Figura 5 – Comparativo entre os tipos de listas.....	14
Figura 6 – Struct No para a lista de entregas.....	15
Figura 7 – Inserção de elementos na lista - início.....	16
Figura 8 – Diagrama de nós e ligações de inserção na lista.....	16
Figura 9 – Inserção de elementos na lista - final.....	17
Figura 10 – Remoção de elementos no início e no final da lista.....	18
Figura 11 – Diagrama de nós e ligações para remoção na lista.....	19
Figura 12 – Função para buscar na lista.....	20
Figura 13 – Função para exibir a lista.....	21
Figura 14 – Diagrama de nós e ligações para impressão da playlist.....	22
Figura 15 – Struct Musica para a playlist.....	24
Figura 16 – Inserção de elementos na playlist - início.....	25
Figura 17 – Inserção de elementos na playlist - final.....	26
Figura 18 – Diagrama de nós e ligações de inserção.....	26
Figura 19 – Remoção de elementos no início da playlist.....	27
Figura 20 – Diagrama de nós e ligações para remoção na playlist.....	27
Figura 21 – Função para buscar uma música na playlist.....	28
Figura 22 – Função para exibir a playlist.....	29
Figura 23 – Diagrama de nós e ligações para impressão da playlist.....	29
Figura 24 – Função para passar a música na playlist.....	30
Figura 25 – Struct para a playlist.....	32
Figura 26 – Inserção de elementos na lista - início.....	33
Figura 27 – Inserção de elementos na lista - final.....	34
Figura 28 – Diagrama de nós e ligações de inserção na lista.....	35
Figura 29 – Remoção de elementos no início da lista.....	36
Figura 30 – Diagrama de nós e ligações para remoção da lista.....	37
Figura 31 – Função para buscar na lista.....	37
Figura 32 – Função para exibir a lista.....	38
Figura 33 – Diagrama de nós e ligações para impressão da lista.....	38
Figura 34 – Função para exibir as informações com animação.....	39
Figura 35 – Struct “Node” para o Programa.....	41
Figura 36 – Criação da Lista - Adição do Primeiro Elemento.....	42
Figura 37 – Inserção de Elementos no Início do Carrossel.....	43
Figura 38 – Inserção de Elementos no Final do Carrossel.....	44
Figura 39 – Diagrama de Nós e Ligações da Lista.....	45

Figura 40 – Remoção de Elementos no Início do Carrossel.....	45
Figura 41 – Diagrama de Nós e Ligações para Remoção na Lista.....	46
Figura 42 – Função para Buscar uma Imagem no Carrossel.....	46
Figura 43 – Função para Exibir o Carrossel - Parte 1/3.....	47
Figura 44 – Saída da Função para Exibir o Carrossel - Parte 1/3.....	47
Figura 45 – Função para Exibir o Carrossel - Parte 2/3.....	48
Figura 46 – Saída da Função para Exibir o Carrossel - Parte 2/3.....	49
Figura 47 – Função para Exibir o Carrossel - Parte 3/3.....	50
Figura 48 – Saída da Função para Exibir o Carrossel - Parte 3/3.....	51

SUMÁRIO

1. INTRODUÇÃO AO TEMA.....	6
2. TIPOLOGIA DAS LISTAS ENCADEADAS.....	7
3. IMPORTÂNCIA E APLICAÇÕES.....	8
4. OPERAÇÕES FEITAS COM LISTAS.....	9
5. COMPARAÇÃO ENTRE OS TIPOS DE LISTAS.....	14
6. LISTA SIMPLEMENTE ENCADEADA.....	15
7. LISTA DUPLAMENTE ENCADEADA.....	24
8. LISTA CIRCULAR SIMPLES.....	32
9. LISTA DUPLAMENTE CIRCULAR.....	41
10. CONCLUSÃO.....	53
11. REFERÊNCIAS.....	54

1. INTRODUÇÃO AO TEMA

A Lista Encadeada (Linked List) é uma das estruturas de dados mais flexíveis e essenciais na ciência da computação, ela é utilizada para armazenar uma coleção de elementos de uma forma dinâmica na memória do computador. Diferente dos arrays (vetores), que armazenam os dados em blocos contíguos de memórias, uma lista encadeada organiza os seus elementos em nós, onde cada nó contém o dado e um ponteiro para o próximo nó da sequência. E essa característica a torna comumente utilizada em situações que exigem inserções, remoções eficientes de elementos, sem a necessidade de realocar totalmente a estrutura **(CORMEN et al., 2012)**.

Conceitos Básicos:

- **Nó (Node):** A unidade fundamental de uma lista encadeada. Cada nó é composto por duas partes: um campo para armazenar o dado e um ou mais ponteiros que o conectam a outros nós na lista.
- **Ponteiro (Pointer):** Uma referência que armazena o endereço de memória do próximo nó (e, em alguns casos, do nó anterior), formando a "ligação" entre os elementos da lista.
- **Cabeça (Head):** O ponteiro que aponta para o primeiro nó da lista. É o ponto de partida para percorrer a estrutura. Se a cabeça for nula (NULL), a lista está vazia.
- **Cauda (Tail):** O último nó da lista. Em listas simplesmente encadeadas, seu ponteiro para o próximo nó aponta para NULL, indicando o fim da sequência.

Esses conceitos formam a base para toda a manipulação de listas encadeadas em linguagens como C, em que o gerenciamento de memória através de ponteiros é uma característica essencial e central **(TENENBAUM; LANGSAM; AUGENSTEIN, 1995)**.

2. TIPOLOGIA DAS LISTAS ENCADEADAS

As listas encadeadas podem ser classificadas em diferentes tipos, variando conforme a quantidade e a organização de seus ponteiros. A escolha de uma tipologia específica está associada ao porquê da devida aplicação (**SEDGEWICK; WAYNE, 2011**). As principais variações são:

- Lista Simplesmente Encadeada (Singly Linked List): Modelo mais fundamental, no qual cada nó possui um único ponteiro que referencia o nó seguinte. A navegação pela lista é unidirecional.
- Lista Duplamente Encadeada (Doubly Linked List): Nesta variação, cada nó contém dois ponteiros: um que aponta para o próximo nó e outro que aponta para o nó anterior. Essa configuração permite a travessia da lista em ambos os sentidos, otimizando certas operações, como a remoção de um nó específico.
- Lista Circular Simples (Circular Singly Linked List): Derivação da lista simplesmente encadeada em que o ponteiro do último nó, em vez de apontar para NULL, vai apontar para o primeiro nó da lista (cabeça), formando uma estrutura em ciclos.
- Lista Duplamente Circular (Doubly Circular Linked List): Combina as características das listas duplamente encadeadas e circular. O ponteiro "próximo" do último nó aponta para a cabeça, e o ponteiro "anterior" da cabeça aponta para o último nó, criando um ciclo bidirecional contínuo.

3. IMPORTÂNCIA E APLICAÇÕES

A importância das listas encadeadas fica clara quando se observam as suas vantagens em relação a estruturas de dados mais rígidas, como os vetores (*arrays*). Os principais benefícios práticos são:

- **Memória Dinâmica:** Diferentemente de um array, que precisa ter seu tamanho já definido anteriormente, uma lista encadeada pode aumentar e diminuir de tamanho durante a execução do código. A memória é alocada apenas quando um novo elemento/valor é adicionado, o que evita o desperdício de memória que ficaria basicamente como “lixo”, sem um uso devido. Diferente de um vetor, que precisa ter seu tamanho definido previamente, uma lista encadeada cresce e diminui durante a execução do programa. **(TENENBAUM; LANGSAM; AUGENSTEIN, 1995).**
- **Facilidade de Inserção e Remoção:** Em vez de ter que deslocar múltiplos dados (como ocorreria em um vetor), na lista encadeada ele terá apenas que atualizar os ponteiros dos nós vizinhos, tornando o processo muito mais rápido e ideal para dados que mudam constantemente e de forma ágil **(CORMEN et al., 2012).**
- **Implementação de Funcionalidades de Navegação:** A flexibilidade das listas é essencial para funcionalidades de software que exigem navegação sequencial. Uma aplicação seria o sistema de "avançar" e "voltar" de um navegador de internet. Cada página visitada pode ser representada como um nó em uma lista duplamente encadeada, permitindo que o usuário navegue para frente e para trás em seu histórico de forma eficiente **(GEEKSFORGEEKS).**

4. OPERAÇÕES FEITAS COM LISTAS

Inserção

O processo de inserção em listas encadeadas consiste na criação dinâmica de um novo nó e após isso ajustar os ponteiros da estrutura (struct) para incluí-lo. Nas listas simplesmente encadeadas, o novo nó aponta para o próximo elemento, e o anterior sempre precisa atualizar seu ponteiro. Na lista duplamente encadeada, temos dois ponteiros, o que vai apontar para o próximo elemento e um que vai apontar para o elemento anterior, ligando assim os dois sentidos da lista.

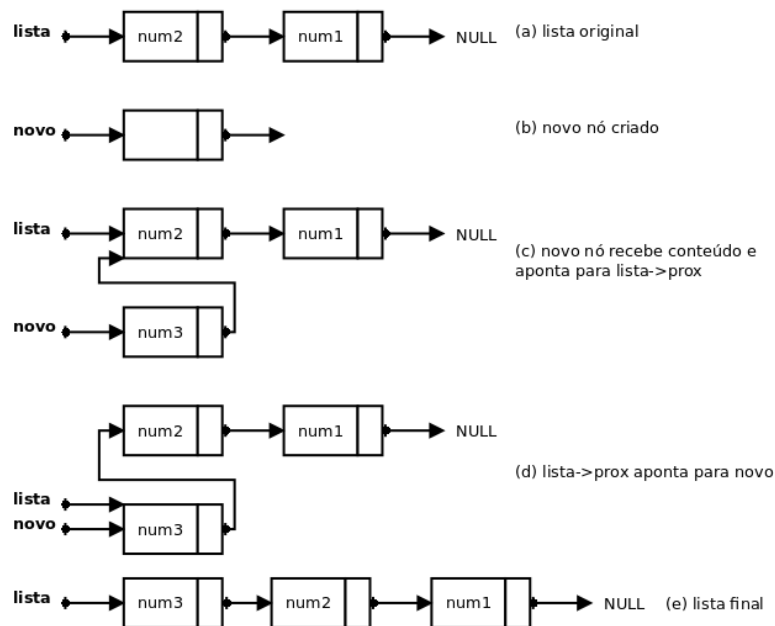
Ademais, nas listas circulares, a inserção envolve ajustar o último nó para sempre apontar para o primeiro, garantindo um ciclo dentro da lista. E por fim, a lista duplamente circular oferece os mesmos procedimentos da lista duplamente encadeada e a lista circular.

De modo geral, o processo de inserção deve respeitar três casos principais:

1. **Lista vazia:** o novo nó se torna a cabeça (head).
2. **Inserção no início:** o novo nó passa a ser a cabeça e aponta para o antigo primeiro elemento.
3. **Inserção no final:** o novo nó é anexado após o último elemento, atualizando o ponteiro do nó anterior.

Figura 1 – Exemplo de inserção em listas encadeadas

Inserir nó no início da lista



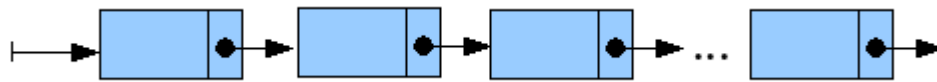
Fonte: Instituto de Computação (IC) - Unicamp

Esta figura demonstra o processo da alocação de um novo nó e o reajuste dos ponteiros. O diagrama mostra como o ponteiro do nó anterior é atualizado para apontar para o novo nó, e como o ponteiro do novo nó é ajustado para apontar para o elemento seguinte, garantindo que a sequência da lista seja mantida corretamente após a inserção.

Busca

A busca em listas encadeadas percorre os nós sequencialmente até encontrar o valor que o usuário deseja, na lista simplesmente encadeada a busca ocorre respeitando o seu limite, ou seja, vai percorrer em um único sentido, do primeiro nó até o último. Já na lista duplamente encadeada e na circular, é possível percorrer em ambos os sentidos, tornando a busca mais rápida em alguns casos. Nesse caso, o algoritmo vai comparar o valor armazenado em cada nó com o valor procurado, caso o nó não exista, o algoritmo percorre toda a lista e encerra ao atingir o final.

Figura 2 – Exemplo de busca em lista encadeada



Fonte: Instituto de Matemática e Estatística (IME) - USP

A imagem ilustra o processo de percorrer uma lista encadeada, que é a operação utilizada tanto para buscar um valor quanto para exibir todos os elementos.

O processo inicia com um ponteiro auxiliar apontando para o início da lista, logo em seguida o algoritmo avança pelos elementos, movendo o ponteiro auxiliar para o próximo nó, até que o valor desejado seja encontrado, ou até chegar em NULL, o que quer dizer que aquele valor não existe na lista.

Remoção

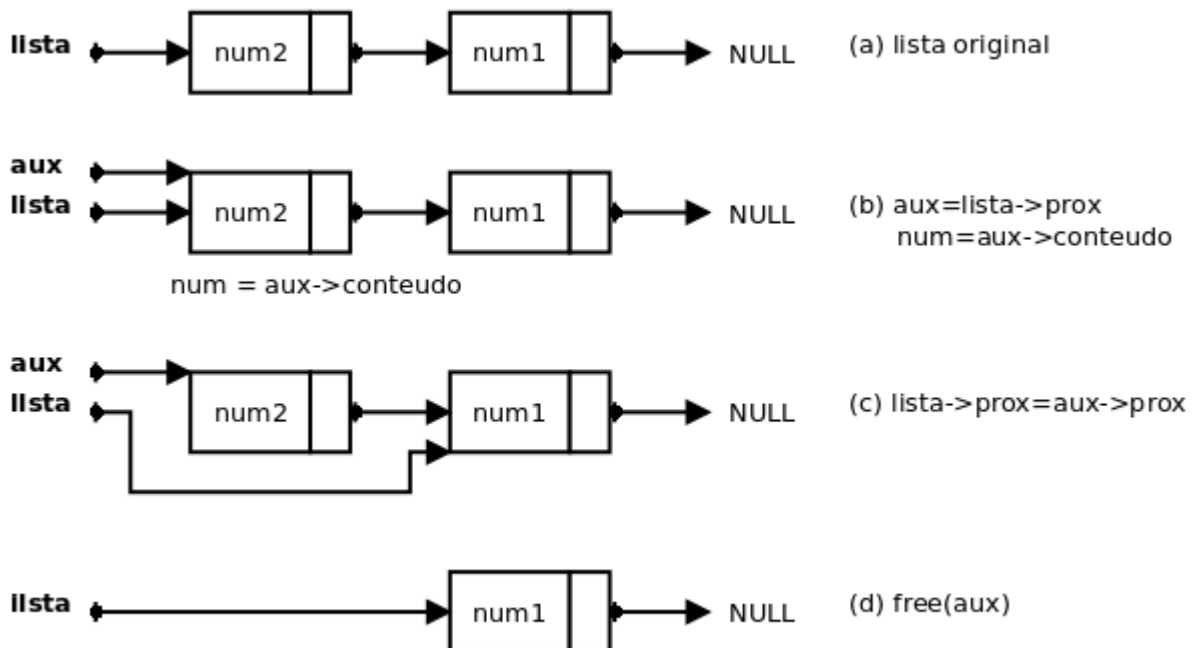
A remoção de um elemento em uma lista encadeada requer o ajuste dos ponteiros dos nós vizinhos para “pular” o nó removido, mantendo a continuidade da sequência. Em listas simplesmente encadeadas, apenas o ponteiro do nó anterior precisa ser ajustado, nas duplamente encadeadas, tanto o próximo do nó anterior quanto o anterior do nó seguinte são atualizados, e nas circulares, é necessário garantir que o encadeamento se mantenha fechado e cíclico após a exclusão.

Casos típicos:

- Remoção da cabeça: a cabeça passa a ser o próximo nó.
- Remoção do último elemento: o penúltimo passa a apontar para NULL ou para a cabeça (no caso circular).
- Remoção do único elemento: a lista passa a ser vazia (NULL).

Figura 3 – Processo de remoção de nós em listas encadeadas

Remover do Início da lista



Fonte: Instituto de Computação (IC) - Unicamp

Este diagrama ilustra o caso específico da remoção da cabeça (o primeiro nó) de uma lista simplesmente encadeada. E ele é feito da seguinte forma:

1. Criar um ponteiro auxiliar para guardar a referência do primeiro nó, que será removido.
2. Atualizar o ponteiro principal da lista para que ele pule o primeiro nó e aponte diretamente para o segundo elemento.
3. Com a lista já reestruturada, o nó original é liberado da memória com `free()`.

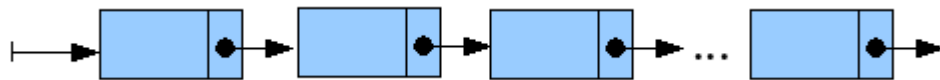
Exibição

A exibição dos elementos nas listas mostra os elementos na ordem em que estão ligados dentro das listas e em todos os tipos a lógica básica é começar a cabeça da lista e imprimir cada nó somente uma vez, imprimindo o elemento e avançando pelos ponteiros.

Casos Típicos:

- Lista vazia: imprimir “Lista vazia” e encerrar.
- Travessia: visitar nós sequencialmente, evitando laços infinitos.
- Formatação visual: usar setas para deixar claro o encadeamento, por ex.: A -> B -> C -> NULL ou em casos como a lista duplamente encadeada: NULL <--> A <--> B <--> C <--> NULL.

Figura 4 – Exemplo de exibição em lista encadeada



Fonte: Instituto de Matemática e Estatística (IME) - USP

Assim, como na parte de busca, aqui podemos utilizar a mesma imagem para representar a exibição, nele utilizamos um ponteiro auxiliar para ajudar a percorrer, assim ele vai fazer a impressão de todos os elementos da lista, e quando o ponteiro auxiliar apontar para NULL, significa que a lista chegou ao fim e a exibição é finalizada.

5. COMPARAÇÃO ENTRE OS TIPOS DE LISTAS

A tabela abaixo destaca as principais diferenças estruturais e funcionais entre os quatro tipos de listas encadeadas, a lista encadeada é a forma mais básica possuindo apenas um ponteiro para o próximo elemento, por isso ela é de fácil implementação, mas limita o acesso a apenas uma direção. Já a lista duplamente encadeada vai melhorar essa limitação que a lista encadeada possui ao incluir dois ponteiros, um que aponta para o próximo nó e outro que aponta para o anterior e essa estrutura permite percorrer a lista nos dois sentidos.

Figura 5 – Comparativo entre os tipos de listas

Tipo de Lista	Direção do Encadeamento	Ponteiros por Nó	Características Principais
Simplesmente Encadeada	Somente próxima	1	Estrutura básica, leve, unidirecional.
Duplamente Encadeada	Próxima e anterior	2	Permite percorrer nos dois sentidos.
Circular Simples	Somente próxima (cíclica)	1	Evita NULL, navegação contínua.
Duplamente Circular	Próxima e anterior (cíclica)	2	Navegação bidirecional e cíclica.

Fonte: Leal, Andreas et al, outubro de 2025

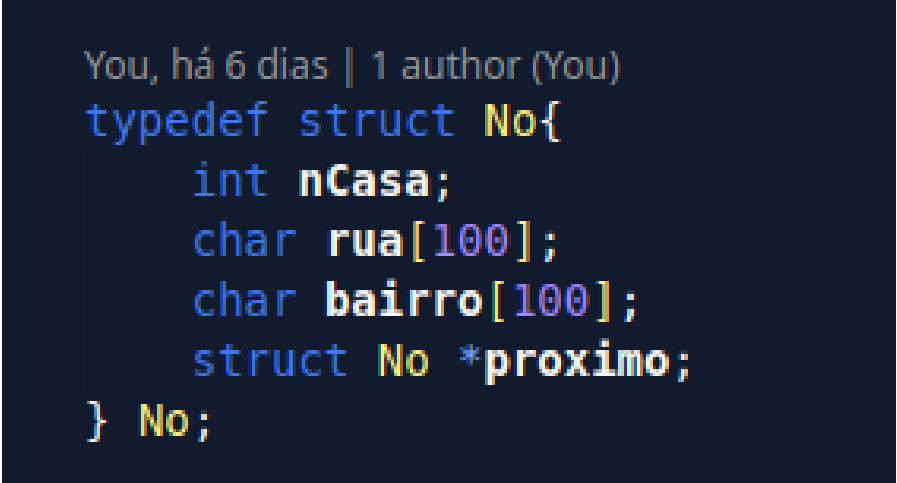
A lista circular simples elimina o contexto de fim ao fazer com que o último nó aponte novamente para o primeiro isso possibilita uma navegação cíclica. Por fim, a lista duplamente circular combina as vantagens da circular e dá duplamente encadeada, ela permite percorrer a lista dos dois lados e de forma cíclica.

6. LISTA SIMPLEMENTE ENCADEADA

Neste tópico será abordado a parte prática da lista encadeada, que tem como exemplo um sistema de cadastro de encomendas, o código destaca as operações principais, como: inserção, remoção, busca, exibição e navegação.

Escolha da prática: Implementamos um sistema para ajudar entregadores usando lista simplesmente encadeada porque o fluxo de trabalho é essencialmente linear: percorre-se a rota sempre para a frente, endereço após endereço. Essa estrutura é leve e direta, facilita inserções e remoções durante o percurso (por exemplo, adicionar um novo ponto ou retirar um já entregue) e mantém o código simples para demonstrar o básico de navegação em listas. Embora versões circulares ou duplamente encadeadas ofereçam retorno automático ou navegação reversa, a lista simples cumpre bem o objetivo didático e operacional de listar e seguir os endereços em sequência.

Figura 6 – Struct No para a lista de entregas



```
You, há 6 dias | 1 author (You)
typedef struct No{
    int nCasa;
    char rua[100];
    char bairro[100];
    struct No *proximo;
} No;
```

Fonte: Leal, Andreas et al, outubro de 2025

Cada nó da lista encadeada é representado por esse struct, que possui:

nCasa: Campo que armazena o número na casa que constar no endereço da encomenda.

rua: Campo que armazena a rua da casa que constar no endereço da encomenda.

bairro: Campo que armazena o bairro da casa que constar no endereço da encomenda.

proximo: Ponteiro que guarda o endereço da próxima próxima No.

Figura 7 – Inserção de elementos na lista - início

```
int inicial_lista(No **head){

    No *aux, *novoHead = *head;

    *head = malloc(sizeof(No*));
    if(*head == NULL){
        printf("Falha na alocação de memória. Encerrando...\n");
        return 1;
    }

    aux = (*head);

    printf("
    |_____| \n");
    printf("
    |          ENDEREÇO Encomenda %d          | \n", 1);
    printf("
    |_____| \n");
    printf("
    | Número: ");
    scanf("%d%c", &aux->nCasa);

    printf("
    | Rua: ");
    fgets(aux->rua, sizeof(aux->rua), stdin);
    aux->rua[strcspn(aux->rua, "\n")] = '\0'; // remove o '\n' se existir

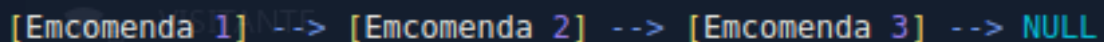
    printf("
    | Bairro: ");
    fgets(aux->bairro, sizeof(aux->bairro), stdin);
    aux->bairro[strcspn(aux->bairro, "\n")] = '\0';
    printf("
    |_____| \n");

    aux->proximo = novoHead;
    return 0;
}
```

Fonte: Leal, Andreas et al, outubro de 2025

A inserção no início da lista ocorre quando o novo nó se torna a nova cabeça (head). O ponteiro próximo do novo nó passa a apontar para o antigo primeiro nó, e o ponteiro anterior é definido como NULL, pois não há elementos antes dele.

Figura 8 – Diagrama de nós e ligações de inserção na lista



Fonte: Leal, Andreas et al, outubro de 2025

Figura 9 – Inserção de elementos na lista - final

```
4
5 int final_lista(No *head){
6     No *aux = head;
7
8     while (aux->proximo != NULL) {
9         aux = aux->proximo;
10    }
11
12    aux->proximo = malloc(sizeof(No));
13    if(aux->proximo == NULL){
14        printf("Falha na alocação de memória. Encerrando...\n");
15        return 1;
16    }
17
18    aux = aux->proximo;
19
20    printf("
21    printf("| ENDEREÇO Encomenda |
22    printf("|
23    printf("| Número: ");
24    scanf("%d%c", &aux->nCasa);
25
26    printf("| Rua: ");
27    fgets(aux->rua, sizeof(aux->rua), stdin);
28    aux->rua[strcspn(aux->rua, "\n")] = '\0'; // remove o '\n' se existir
29
30    printf("| Bairro: ");
31    fgets(aux->bairro, sizeof(aux->bairro), stdin);
32    aux->bairro[strcspn(aux->bairro, "\n")] = '\0';
33
34    aux->proximo = NULL;
35
36    return 0;
37 }
```

Fonte: Leal, Andreas et al, outubro de 2025

A inserção no final adiciona o novo nó após o último elemento da lista. Nesse caso, o campo aux nó aponta para o último elemento, e seu próximo recebe NULL.

Figura 10 – Remoção de elementos no início e no final da lista

```
void remocao_final(No **head){
    No *aux = *head;
    No *prev = NULL;

    if (aux->proximo == NULL) {
        printf("\n");
        printf("        Ação inválida\n");
        printf("        Não é possível. Use a opção\n");
        printf("        \\\"4) Remover entrega(Inicio)\\\" \n");
        printf("\n");
        printf("\nPressione ENTER para voltar ao menu.\n");
        getchar();
        getchar();
        system("clear");
        return ;
    }

    while (aux->proximo != NULL) {
        prev = aux;
        aux = aux->proximo;
    }
    prev->proximo = NULL;
    free(aux);
    aux = NULL;

    printf("\n");
    printf("        Exclusão bem-sucedida\n");
    printf("\n");
    printf("\nPressione ENTER para voltar ao menu.\n");
    getchar();
    getchar();
    system("clear");
}

void remocao_inicil(No **head){
    No *headAtual = *head;
    (*head) = (*head)->proximo;
    free(headAtual);
    headAtual = NULL;
    printf("\n");
    printf("        Memória liberada com sucesso.\n");
    printf("        Itens excluídos.\n");
    printf("\n");
    printf("\nPressione ENTER para voltar ao menu.\n");
    getchar();
    getchar();
    system("clear");

    if(*head == NULL){
        printf("\n");
        printf("        LISTA VAZIA\n");
        printf("\n");
        printf("        Você será redirecionado ao início\n");
        printf("        do sistema.\n");
        printf("\n");
        printf("\nPressione ENTER para voltar ao menu.\n");
        getchar();
        system("clear");
    }
}
```

Fonte: Leal, Andreas et al, outubro de 2025

No código de lista encadeada a remoção pode ser feita no início ou no final. O código ajusta os ponteiros dos nós vizinhos para garantir que o encadeamento continue correto após a exclusão, além de que quando não tiver nenhum elemento da lista o

sistema irá direcionar o usuário para o menu inicial, para criar uma nova lista. E quando a remoção ocorre no final o elemento que era o penúltimo recebe o endereço como *NULL* e se torna o último, caso só sobre a cabeça (head) o sistema imprime uma mensagem falando que a operação não é possível e mostra qual opção o usuário pode usar para completar a operação.

Figura 11 – Diagrama de nós e ligações para remoção na lista

```
Antes: [Encomenda 1] --> [Encomenda 2] --> [Encomenda 3] --> NULL  
Remove: 'Encomenda 2'  
Depois: [Encomenda 1] --> [Encomenda 3] --> NULL
```

Fonte: Leal, Andreas et al, outubro de 2025

Figura 12 – Função para buscar na lista

```
void busca_casa(No **head){
    system("clear");
    int a = 1;
    No *aux = *head;

    printf("Número da casa: ");
    scanf("%d%c", &a);

    while (aux != NULL){
        if(aux->nCasa == a){
            printf("\n");
            printf("ENDEREÇO Encomenda %d\n", a);
            printf("\n");
            printf("Número: %d\n", aux->nCasa);
            printf("Rua: %s\n", aux->rua);
            printf("Bairro: %s (char [39])\nPressione ENTER para voltar ao menu.\n");
            printf("\nPressione ENTER para voltar ao menu.\n");
            getchar();
            return;
        }
        aux = aux->proximo;
    }

    printf("\n");
    printf("NENHUMA ENTREGA CADASTRADA\n");
    printf("\n");
    printf("\nPressione ENTER para voltar ao menu.\n");
    getchar();
    return;
}
```

Fonte: Leal, Andreas et al, outubro de 2025

A busca percorre os nós da lista e comparando o número informado com cada endereço, caso o número não esteja na lista o sistema retorna uma mensagem indicando que o número não foi encontrado. E caso for encontrado ele retorna as informações referente ao número da casa informado.

Figura 13 – Função para exibir a lista

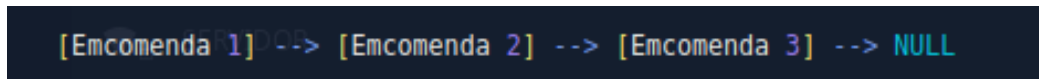
```
void busca_casa(No **head){  
  
    system("clear");  
    int a = 1;  
    int b = 0;  
    No *aux = *head;  
  
    printf("Número do Casa: ");  
    scanf("%d%c", &a);  
  
    do{  
        if(aux->nCasa == a){  
            printf("ENDEREÇO Encomenda %d\n", a);  
            printf("Número: %d\n", aux->nCasa);  
            printf("Rua: %s\n", aux->rua);  
            printf("Bairro: %s\n", aux->bairro);  
            b++;  
        }  
        aux = aux->proximo;  
    } while (aux != NULL);  
  
    if(b == 0){  
        printf("NENHUMA ENTREGA CADASTRADA\n");  
        printf("\nPressione ENTER para voltar ao menu.\n");  
        getchar();  
        return;  
    }  
  
    printf("\n");  
    printf("\nPressione ENTER para voltar ao menu.\n");  
    getchar();  
}
```

Fonte: Leal, Andreas et al, outubro de 2025

A função de listar(imprimir) percorre a lista do início ao fim e mostra todos os endereços encadeados, exibindo também o relacionamento entre nós.

Saída esperada:

Figura 14 – Diagrama de nós e ligações para impressão da playlist



Fonte: Leal, Andreas et al, outubro de 2025

Esse diagrama de nós e ligações se refere a como será a saída(print) da lista que contém 3 encomendas, ela terá como visualização as encomendas registradas e o NULL no final da lista.

Casos de Borda:

Lista vazia: o sistema retorna para o início.

Lista com único elemento: NULL -> [A] -> NULL.

Inserção múltipla: cada novo nó ajusta apenas o ponteiro do nó anterior.

Remoção da última música: o ponteiro anterior torna-se o novo final da lista.

Vantagens:

Menos memória já que é 1 ponteiro por nó.

Implementação mais simples.

Boa para filas/pilhas já que dá para enfileirar e desenfileirar de maneira eficiente.

Desvantagens:

Sem navegação para trás porque não dá para ir ao anterior sem percorrer desde o início.

Algumas operações ficam como achar o penúltimo, remover último, ou inserir “antes de X” exigem varredura.

7. LISTA DUPLAMENTE ENCADEADA

Este tópico irá representar a parte prática da lista duplamente encadeada, em que foi utilizado um exemplo de uma playlist de música, o código destaca as operações principais, como: inserção, remoção, busca, exibição e navegação.

Escolha da prática: Foi escolhida a implementação de uma playlist de música utilizando lista duplamente encadeada porque demonstra o funcionamento de navegação comum, como player de música, e com essa lista é possível avançar e voltar entre as músicas. A escolha dessa estrutura foi feita para facilitar o entendimento dos conceitos básicos de encadeamento duplo, e também porque hoje em dia a música se tornou algo habitual na sociedade e quase todas as pessoas do mundo passaram a escutá-la.

Apesar de que a lista duplamente circular possa ser mais adequada para essa aplicação, pois permite uma navegação que quando se atingisse o final da playlist, o sistema voltaria automaticamente para o início, mas ainda assim essa versão com a lista duplamente encadeada utilizada no projeto oferece uma clareza bem didática para explicar a questão de retroceder e avançar dentro de uma lista.

Figura 15 – Struct Musica para a playlist

```
struct Musica {  
    char nome[100]; //armazena o nome da música  
    struct Musica *prox; //ponteiro para o próximo nó (música seguinte)  
    struct Musica *ante; //ponteiro para o nó anterior (música anterior)  
};
```

Fonte: Leal, Andreas et al, outubro de 2025

Cada nó da lista duplamente encadeada é representado por esse struct, que possui:

nome: campo que armazena o nome da música.

prox: ponteiro que guarda o endereço da próxima música na playlist.

ante: ponteiro que guarda o endereço da música anterior.

Figura 16 – Inserção de elementos na playlist - início

```
void inserir_inicio(struct Musica **playlist){
    struct Musica *novo = malloc(sizeof(struct Musica));
    if(novo == NULL){
        printf("Erro de alocação!\n");
        printf("Pressione Enter para continuar\n");
        getchar(); getchar();
        return;
    }
    printf("Digite o nome da música: \n");
    getchar(); //limpa o buffer, retirando o \n do enter
    fgets(novo->nome, 100, stdin); //fgets faz a leitura da string e stdin é a entrada padrão que nesse caso é o teclado
    novo->nome[strcspn(novo->nome, "\n")] = '\0'; //strcspn encontra o \n(enter do usuário) no cod e substitui por \0(finaliza a string)
    novo->prox = *playlist; //novo nó aponta para o antigo primeiro
    novo->ante = NULL; //não tem valor anterior
    if (*playlist != NULL){
        (*playlist)->ante = novo; //caso a cabeça seja diferente de NULL o ante(agenda a antiga cabeça) aponta para o novo
    }
    *playlist = novo; //cabeça vira novo valor
    printf("Música adicionada!\n");
    printf("Pressione Enter para continuar!\n");
    getchar(); //limpa o buffer
}
```

Fonte: Leal, Andreas et al, outubro de 2025

A inserção no início da lista ocorre quando o novo nó se torna a nova cabeça (head). O ponteiro prox do novo nó passa a apontar para o antigo primeiro nó, e o ponteiro ante é definido como NULL, pois não há elementos antes dele.

Figura 17 – Inserção de elementos na playlist - final

```
void inserir_final(struct Musica **playlist){
    struct Musica *aux, *novo = malloc(sizeof(struct Musica));
    if(novo == NULL){
        printf("Erro de alocação!\n");
        printf("Pressione Enter para continuar\n");
        getchar(); getchar();
        return;
    }
    printf("Digite o nome da música: \n");
    getchar();
    fgets(novo->nome, 100, stdin);
    novo->nome[strcspn(novo->nome, "\n")] = '\0';
    novo->prox = NULL;
    if(*playlist == NULL){
        *playlist = novo;
        novo->ante = NULL;
    }
    else {
        aux = *playlist; //aux recebe a cabeça da lista
        while(aux->prox != NULL){ //percorre até o ultimo
            aux = aux->prox;
        }
        aux->prox = novo; //ultimo valor vira o novo
        novo->ante = aux; //antigo ultimo valor vira o penúltimo
    }
    printf("Música adicionada!\n");
    printf("Pressione Enter para continuar!\n");
    getchar();
}
```

Fonte: Leal, Andreas et al, outubro de 2025

A inserção no final adiciona o novo nó após o último elemento da lista. Nesse caso, o campo ante do novo nó aponta para o último elemento, e seu prox recebe NULL. Se a lista estiver vazia, o novo nó será a cabeça. Senão, o código percorre até o último nó e insere o novo após ele. O ponteiro ante do novo nó aponta para o antigo último elemento.

Após a execução da função, as músicas ficam encadeadas da seguinte forma:

Figura 18 – Diagrama de nós e ligações de inserção

```
NULL <--> [Música 1] <--> [Música 2] <--> [Música 3] <--> NULL
```

Fonte: Leal, Andreas et al, outubro de 2025

Figura 19 – Remoção de elementos no início da playlist

```
void remover_inicio(struct Musica **playlist){
    struct Musica *remover;
    if (*playlist == NULL){
        printf("Playlist vazia!\n");
        printf("Pressione Enter para continuar!\n");
        getchar(); getchar();
        return;
    }
    remover = *playlist;
    *playlist = remover->prox;
    if (*playlist != NULL){
        (*playlist)->ante = NULL;
    }
    free(remover);
    printf("Primeira música removida!\n");
    printf("Pressione Enter para continuar\n");
    getchar(); getchar();
}
```

Fonte: Leal, Andreas et al, outubro de 2025

No código de lista duplamente encadeada a remoção pode ser feita no início, no final ou na música atual. O código ajusta os ponteiros dos nós vizinhos para garantir que o encadeamento continue correto após a exclusão.

Figura 20 – Diagrama de nós e ligações para remoção na playlist

```
ANTES: NULL <--> [Música 1] <--> [Música 2] <--> [Música 3] <--> NULL
REMOVE 'Música 1' //vai remover essa musica
DEPOIS: NULL <--> [Música 2] <--> [Música 3] <--> NULL
```

Fonte: Leal, Andreas et al, outubro de 2025

Essa figura representa uma lista com 3 músicas ligadas entre si, cada nó contém o nome da música e os seus 2 ponteiros que apontam para o anterior e para o próximo, após a remoção da 1º música, o 2º nó passa a ser a nova cabeça da lista.

Figura 21 – Função para buscar uma música na playlist

```
struct Musica* buscar(struct Musica **playlist){
    struct Musica *aux;
    aux = *playlist;
    if(*playlist == NULL){
        printf("Playlist vazia!\n");
        printf("Pressione Enter para continuar\n");
        getchar(); getchar();
        return NULL;
    }
    char nome[100];
    printf("Digite o nome da música que quer buscar:\n");
    getchar();
    fgets(nome, 100, stdin);
    nome[strcspn(nome, "\n")] = '\0';
    while(aux != NULL){
        if(strcmp(aux->nome, nome) == 0){ //compara 2 strings caractere por caractere, se for igual retorna 0
            printf("Essa música existe na playlist!\n");
            printf("Pressione Enter para continuar\n");
            getchar();
            return aux;
        }
        aux = aux->prox;
    }

    printf("Essa música não existe na playlist!\n");
    printf("Pressione Enter para continuar\n");
    getchar();
    return NULL;
}
```

Fonte: Leal, Andreas et al, outubro de 2025

A busca percorre os nós da lista e compara o nome informado com cada música, utilizando a função strcmp() para verificar se há alguma música igual ao que o usuário digitou dentro da string, se o nome da música for encontrado, a função retorna o nó correspondente.

Figura 22 – Função para exibir a playlist

```
void imprimir(struct Musica *playlist){
    struct Musica *aux = playlist;
    printf("NULL <--> "); //inicio
    while (aux != NULL){
        printf("%s <--> ", aux->nome);
        aux = aux->prox;
    }
    printf("NULL\n"); //final
    printf("Pressione Enter para continuar\n");
    getchar(); getchar();
}
```

Fonte: Leal, Andreas et al, outubro de 2025

A função de exibição(imprimir) percorre a lista do início ao fim e mostra todas as músicas encadeadas, exibindo também o relacionamento entre os nós.

Saída esperada:

Figura 23 – Diagrama de nós e ligações para impressão da playlist

```
NULL <--> Música 1 <--> Música 2 <--> Música 3 <--> NULL
```

Fonte: Leal, Andreas et al, outubro de 2025

Esse diagrama de nós e ligações se refere a como será a saída(print) da lista que contém 3 músicas, ela terá como visualização as músicas e os NULL em cada ponta da lista, a head e a tail, e cada setinha (<-->) representa os ponteiros ante para o elemento anterior e o prox para o proximo elemento, ligando assim todos os elementos e permitindo uma navegação entre eles.

Figura 24 – Função para passar a música na playlist

```
void passar_musica(struct Musica **atual){
    if(*atual == NULL){
        printf("Playlist vazia!\n");
    }
    else if((*atual)->prox == NULL){
        printf("Você já está na última música da playlist!\n");
    }
    else {
        *atual = (*atual)->prox;
        printf("Música atual: %s\n", (*atual)->nome);
    }
    printf("Pressione Enter para continuar\n");
    getchar(); getchar();
}
```

O algoritmo também permite navegar entre as músicas usando as funções de passar, voltar e exibir a música atual, demonstrando o uso dos ponteiros prox e ante. Assim, o usuário pode avançar para a próxima música ou voltar para a anterior, como em um reprodutor de mídia real.

Casos de Borda:

Lista vazia: o sistema exibe Playlist vazia!

Lista com único elemento: a exibição mostra NULL <--> [A] <--> NULL.

Inserção múltipla: os ponteiros são atualizados corretamente, mantendo a integridade da estrutura.

Remoção da última música: o ponteiro anterior torna-se o novo final da lista.

Vantagens:

Permite percorrer nos dois sentidos (frente e verso).

Facilita remoção e inserção em qualquer posição da lista.

Boa para aplicações com navegação bidirecional (ex.: playlist, histórico de páginas).

Desvantagens:

Requer mais memória (dois ponteiros por nó).

Implementação mais complexa.

Maior tempo de inserção e remoção devido à atualização de dois ponteiros.

8.LISTA CIRCULAR SIMPLES

Este tópico irá representar a parte prática da lista circular, em que foi utilizado um exemplo de um sistema de passagens aéreas, o código destaca as operações principais, como: inserção, remoção, busca, exibição e navegação.

Escolha da prática: Foi escolhida a implementação de um sistema de cadastro de passagens aéreas utilizando lista circular porque essa estrutura permite percorrer todos os voos de forma contínua, retornando automaticamente ao início quando se chega ao fim da lista. Essa característica reflete o funcionamento real de sistemas que exibem passagens em sequência, facilitando a consulta e atualização dos dados.

Figura 25 – Struct para a playlist

```
typedef struct No{  
    int Naviao;  
    char saida[100];  
    char destino[100];  
    int preco;  
    char data[12];  
    struct No *proximo;  
} No;
```

Fonte: Leal, Andreas et al, outubro de 2025

Cada nó da lista circular é representado por esse struct, que possui:

Naviao: Campo que armazena o número na do avião.

saida: Campo que armazena o local de saída do avião.

destino: Campo que armazena o local de destino do avião.

preco: Campo que armazena o valor da passagem do avião.

data: Campo que armazena a data do voo do avião.

proximo: Ponteiro que guarda o endereço da próxima próxima No.

Figura 26 – Inserção de elementos na lista - início

```
int inicial_lista(No **head){

    No *aux, *ahead = *head;

    *head = malloc(sizeof(No));
    if (*head == NULL) {
        printf("Falha na alocação de memória. Encerrando...\n");
        return 1;
    }

    aux = *head;

    printf("\n\n");
    printf("| PASSAGEM AÉREA | \n");
    printf("| \n");
    printf(| Número do avião: ");
    scanf("%d%c", &aux->Naviao);

    printf(| Cidade de saída: ");
    fgets(aux->saida, sizeof(aux->saida), stdin);
    aux->saida[strcspn(aux->saida, "\n")] = '\0';

    printf(| Destino: ");
    fgets(aux->destino, sizeof(aux->destino), stdin);
    aux->destino[strcspn(aux->destino, "\n")] = '\0';

    printf(| Preço (R$ inteiro): ");
    scanf("%d%c", &aux->preco);

    printf(| Data (DD/MM/AAAA): ");
    fgets(aux->data, sizeof(aux->data), stdin);
    aux->data[strcspn(aux->data, "\n")] = '\0';

    printf(| \n");

    No *ultimo = ahead;
    while (ultimo->proximo != ahead) {
        ultimo = ultimo->proximo;
    }

    ultimo->proximo = aux;
    aux->proximo = ahead;

    *head = aux;

    return 0;
}
```

A inserção no início da lista ocorre quando o novo nó se torna a nova cabeça (head). O ponteiro prox do novo nó passa a apontar para o antigo primeiro nó, e o último nó recebe o endereço da nova cabeça (head), assim fechando a lista circular novamente.

Figura 27 – Inserção de elementos na lista - final

```
int final_lista(No *head){  
    No *aux = head, *haux = head;  
  
    while (aux->proximo != haux) {  
        aux = aux->proximo;  
    }  
  
    aux->proximo = malloc(sizeof(No));  
    if(aux->proximo == NULL){  
        printf("Falha na alocação de memória. Encerrando...\n");  
        return 1;  
    }  
  
    aux = aux->proximo;  
  
    printf("┌───────────────────────────────────┐\n");  
    printf("│                                │\n");  
    printf("│                                │\n");  
    printf("│ Número do avião:             │\n");  
    scanf("%d%c", &aux->Naviao);  
  
    printf("│ Cidade de saída:            │\n");  
    fgets(aux->saida , sizeof(aux->saida), stdin);  
    aux->saida[strcspn(aux->saida,"\\n")] = '\\0';  
  
    printf("│ Destino:                     │\n");  
    fgets(aux->destino, sizeof(aux->destino), stdin);  
    aux->destino[strcspn(aux->destino, "\\n")] = '\\0';  
  
    printf("│ Preço (R$ inteiro):         │\n");  
    scanf("%d%c", &aux->preco);  
  
    printf("│ Data (DD/MM/AAAA):          │\n");  
    fgets(aux->data, sizeof(aux->data), stdin);  
    aux->data[strcspn(aux->data, "\\n")] = '\\0';  
  
    printf("└───────────────────────────────────┘\n");  
  
    aux->proximo = haux;  
  
    return 0;
```

A inserção no final adiciona o novo nó após o último elemento da lista. Nesse caso, o campo do novo nó aponta para o último elemento, e seu proximo recebe a cabeça (head).

Figura 28 – Diagrama de nós e ligações de inserção na lista

--> [passagem 1] --> [passagem 2] --> [passagem 3] --> [passagem 1]

Fonte: Leal, Andreas et al, outubro de 2025

Figura 29 – Remoção de elementos no início da lista

```
void remocao_inicil(No **head){

    No *headAtual = *head;
    No *aux = headAtual;
    if (headAtual->proximo == headAtual) {
        free(headAtual);
        *head = NULL;

        printf("\n");
        printf("    Memória liberada com sucesso.    \n");
        printf("    Itens excluídos.                  \n");
        printf("\n");
        printf("\nPressione ENTER para voltar ao menu.\n");
        getchar();
        system("clear");

        printf("\n");
        printf("    LISTA VAZIA                        \n");
        printf("\n");
        printf("    Você será redirecionado ao início  \n");
        printf("    do sistema.                       \n");
        printf("\n");
        printf("\nPressione ENTER para voltar ao menu.\n");
        getchar();
        system("clear");

        return ;
    }

    while (aux->proximo != headAtual) {
        aux = aux->proximo;
    }
    *head = headAtual->proximo;
    aux->proximo = *head;

    free(headAtual);
    headAtual = NULL;

    printf("\n");
    printf("    Memória liberada com sucesso.    \n");
    printf("    Nó inicial removido.             \n");
    printf("\n");
    printf("\nPressione ENTER para voltar ao menu.\n");
    getchar();
    system("clear");
}
```

Fonte: Leal, Andreas et al, outubro de 2025

No código de lista circular a remoção pode ser feita no início e no final. O código ajusta os ponteiros dos nós vizinhos para garantir que o encadeamento continue correto após a exclusão.

Figura 30 – Diagrama de nós e ligações para remoção da lista

```
Antes: --> [passagem 1] --> [passagem 2] --> [passagem 3] --> [passagem 1]
Remove: 'passagem 2'
Depois: --> [passagem 1] --> [passagem 2] --> [passagem 1]
```

Fonte: Leal, Andreas et al, outubro de 2025

Figura 31 – Função para buscar na lista

```
void busca_aviao(No **head){
    system("clear");
    int a = 1;
    int b = 0;
    No *aux = *head;

    printf("Número do avião: ");
    scanf("%d%c", &a);

    do{
        if(aux->Naviao == a){
            printf("
            PASSAGEM AÉREA %d
            \n", a);
            printf("
            Número do avião: %d \n", aux->Naviao);
            printf("
            Cidade de saída: %s \n", aux->saida);
            printf("
            Destino: %s \n", aux->destino);
            printf("
            Preço (R$ inteiro): %d \n", aux->preco);
            printf("
            Data (DD/MM/AAAA): %s \n", aux->data);
            printf("
            \n");
            b++;
        }
        aux = aux->proximo;
    } while (aux
        != *head);

    if(b == 0){
        printf("
        \n");
        printf("
        ESTE AVIÃO NÃO CONSTA NA LISTA DE VOOS
        \n");
        printf("
        \n");
        printf("\nPressione ENTER para voltar ao menu.\n");
        getchar();
        return;
    }
    printf("\n");
    printf("\nPressione ENTER para voltar ao menu.\n");
    getchar();
}
```

Fonte: Leal, Andreas et al, outubro de 2025

A busca percorre os nós da lista e comparando com a informação que o usuário indicou, utilizando a função `busca_aviao()` para verificar se há algum avião com essa numeração.

Figura 32 – Função para exibir a lista

```
void listar(No **head){
    system("clear");
    int a = 1;
    No *aux = *head, *haux = *head;

    if (*head == NULL) {
        printf("\n");
        printf("NENHUMA ENTREGA CADASTRADA\n");
        printf("\n");
        return;
    }

    do{
        printf("PASSAGEM AÉREA %d\n", a);
        printf("Número do avião: %d\n", aux->Naviao);
        printf("Cidade de saída: %s\n", aux->saida);
        printf("Destino: %s\n", aux->destino);
        printf("Preço (R$ inteiro): %d\n", aux->preco);
        printf("Data (DD/MM/AAAA): %s\n", aux->data);
        printf("\n");
        a++;
        aux = aux->proximo;

    } while (aux != haux && aux != NULL);
    printf("\nPressione ENTER para voltar ao menu.\n");
    getchar();
    getchar();
}
```

Fonte: Leal, Andreas et al, outubro de 2025

A função de `listar(imprimir)` percorre a lista do início ao fim e mostra todos os nós.
Saída esperada:

Figura 33 – Diagrama de nós e ligações para impressão da lista

```
--> [passagem 1] --> [passagem 2] --> [passagem 3] -->
```

Fonte: Leal, Andreas et al, outubro de 2025

Figura 34 – Função para exibir as informações com animação

```
void exibir(No *head){
    system("clear");

    if (!head){
        printf("\n");
        printf("ESTE AVIAO NAO CONSTA NA LISTA DE VOOS \n");
        printf("\n");
        printf("\nPressione ENTER para voltar ao menu.\n");
        while (getchar() != '\n');
        return;
    }

    drain_stdin();

    No *cur = head;
    int a = 1;
    const useconds_t delay_us = 4000000;

    printf("Carrossel simples - pressione ENTER para parar.\n");
    usleep(3000000);

    for(;;){
        system("clear");

        printf(" \n");
        printf(" |          PASSAGEM AEREA %d          | \n", a);
        printf(" | \n");
        printf(" | Numero do aviao: %d \n", cur->Naviao);
        printf(" | Cidade de saida: %s \n", cur->saida);
        printf(" | Destino: %s \n", cur->destino);
        printf(" | Preco (R$ inteiro): %d \n", cur->preco);
        printf(" | Data (DD/MM/AAAA): %s \n", cur->data);
        printf(" | \n");
        printf("\n(Aperte ENTER para encerrar o carrossel)\n");

        if (enter_pressed_now()){
            printf("\nEncerrado pelo usuario.\n");
            break;
        }

        usleep(delay_us);
        cur = (cur->proximo ? cur->proximo : head);
        a = (cur == head) ? 1 : (a + 1);
    }
}
```

Fonte: Leal, Andreas et al, outubro de 2025

O algoritmo exibe as informações na tela e as mantém atualizadas no mesmo lugar, a cada intervalo configurado, substitui os dados sem limpar a tela. Para encerrar a atualização, basta pressionar Enter.

Casos de Borda

Lista com único elemento: o nó aponta para ele mesmo, formando um ciclo de um único voo.

Inserção múltipla: os ponteiros são atualizados mantendo o elo entre o último e o primeiro voo, preservando a circularidade.

Remoção da última passagem: ao excluir o único nó, a lista volta para a primeira tela.

Vantagens

Permite percorrer todos os voos de forma contínua, sem precisar reiniciar a busca.

Facilita a visualização cíclica de passagens (volta ao início automaticamente).

Ideal para sistemas que repetem listagens ou simulam carrosséis de voos.

Desvantagens

Requer atenção para evitar loops infinitos ao percorrer.

Inserções e remoções exigem cuidado para manter a ligação circular.

Depuração é mais difícil, pois não há ponteiros NULL indicando fim da lista.

9. LISTA DUPLAMENTE CIRCULAR

A fim de aplicar os conceitos de lista duplamente circular, um carrossel de imagens foi desenvolvido. A escolha foi motivada pelo caráter cíclico e bidirecional da estrutura, que reflete o comportamento esperado de um carrossel, permitindo que o usuário navegue tanto para frente quanto para trás de forma contínua, sem um ponto inicial ou final fixo.

O código implementa as **operações fundamentais** dessa estrutura de dados, como **inserção**, **remoção**, **busca**, **exibição** e **navegação**, possibilitando a manipulação dinâmica dos elementos da lista. O código possui operações fundamentais, como: **inserção**, **remoção**, **busca**, **exibição** e **navegação**.

Figura 35 – Struct “Node” para o Programa

```
typedef struct Node {  
    char label[SIZE];  
    struct Node *prev;  
    struct Node *next;  
} Node;
```

Fonte: Leal, Andreas et al, outubro de 2025

Cada nó da **lista duplamente circular** é criado a partir do **struct “Node”**, que possui:

- **label**: campo que armazena o nome da imagem;
- **prev**: ponteiro que guarda o endereço da música anterior;
- **next**: ponteiro que guarda o endereço da próxima música na playlist.

Figura 36 – Criação da Lista - Adição do Primeiro Elemento

```
void create(Node **head, char label[]) {
    Node *tmp;
    tmp = malloc(sizeof (Node *));
    if (tmp == NULL) {
        printf("Erro de alocação...\n");
        return;
    }
    (*head) = tmp;
    strcpy((*head)→label, label);
    (*head)→prev = (*head);
    (*head)→next = (*head);
    printf("Adicionado com sucesso!\n");
}
```

Fonte: Leal, Andreas et al, outubro de 2025

A criação da lista ocorre quando o ponteiro **head** deixa de apontar a **NULL** e passa a abrigar o primeiro **node**. Um ponteiro **tmp** é utilizado para receber o endereço da memória alocada e passá-la com segurança para o ponteiro **head**. Caso a alocação falhe, o programa é encerrado com uma mensagem de erro, este trecho de código se repete ao longo das funções que serão posteriormente apresentadas, a fim de garantir a consistência dos dados da lista. Os ponteiros **prev** e **next** da cabeça recebem **NULL**.

Figura 37 – Inserção de Elementos no Início do Carrossel

```
void addBeg(Node **head, char label[]) {
    Node *node, *tmp, *aux;
    if ((*head) == NULL) {
        create(&(*head), label);
        return;
    }
    tmp = malloc(sizeof (Node *));
    if (tmp == NULL) {
        printf("Erro de alocação...\n");
        return;
    }
    node = tmp;
    strcpy(node->label, label);
    node->next = (*head);
    node->prev = (*head)->prev;
    aux = node->prev;
    aux->next = node;
    (*head)->prev = node;
    (*head) = node;
    printf("Adicionado com sucesso!\n");
}
```

Fonte: Leal, Andreas et al, outubro de 2025

A inserção no início da lista ocorre da seguinte maneira:

1. Se não existir lista criada, é chamada a função **create**. Logo após, a função de adicionar ao início é encerrada.
2. Caso exista lista, o ponteiro local **node** recebe o endereço do novo espaço alocado, **prev** aponta para o nó anterior à atual cabeça, **next** aponta para a cabeça. O **label** também é recebido através da função **strcpy**.
3. O ponteiro **next** do último nó da lista aponta para **node**.
4. O ponteiro **prev** da atual cabeça passa a apontar para **node**.
5. Ao apontar **node** em **head**, a nova cabeça é definida.

Figura 38 – Inserção de Elementos no Final do Carrossel

```
void addEnd(Node **head, char label[]) {
    Node *node, *tmp, *aux;
    if ((*head) == NULL) {
        create(&(*head), label);
        return;
    }
    tmp = malloc(sizeof (Node *));
    if (tmp == NULL) {
        printf("Erro de alocacao...\n");
        return;
    }
    node = tmp;
    strcpy(node->label, label);
    node->next = (*head);
    node->prev = (*head)->prev;
    aux = (*head)->prev;
    aux->next = node;
    (*head)->prev = node;
    printf("Adicionado com sucesso!\n");
}
```

Fonte: Leal, Andreas et al, outubro de 2025

A inserção no final adiciona o novo nó após o último nó (anterior à cabeça). Se a lista estiver vazia, a função **create** será chamada. Senão, a execução da tarefa se dá seguinte maneira:

1. O campo **label** do novo nó (**node**) recebe o label passado para a função.
2. O ponteiro **next** de **node** aponta para **head**.
3. O ponteiro **prev** de **node** aponta ao **prev** de **head**.
4. O atual último elemento da lista aponta para **node** em **next**.
5. A cabeça passa a apontar para **node** em **prev**.

Após a execução da função, as músicas ficam encadeadas da seguinte forma:

Figura 39 – Diagrama de Nós e Ligações da Lista

... <—> [Imagem 1] <—> [Imagem 2] <—> [Imagem 3] <—> ...

Fonte: Leal, Andreas et al, outubro de 2025

Figura 40 – Remoção de Elementos no Início do Carrossel

```
void rmBeg(Node **head) {
    Node *node;
    if ((*head)→next == (*head)) {
        free((*head));
        (*head) = NULL;
        printf("Espaco liberado na memoria...\n");
        printf("Lista deletada com sucesso!\n");
        return;
    }
    node = (*head)→next;
    node→prev = (*head)→prev;
    node = (*head)→prev;
    node→next = (*head)→next;
    free((*head));
    (*head) = node→next;
    printf("Removido com sucesso!\n");
}
```

Fonte: Leal, Andreas et al, outubro de 2025

A remoção pode ser feita no início ou no final do carrossel. Os seguintes passos são executados para executar a remoção no início:

1. Se existir apenas um nó (cabeça), é chamada a função **free()** e o ponteiro **head** recebe **NULL**, a função de remoção no começo finaliza.
2. Ajusta os ponteiros dos nós vizinhos para garantir que o encadeamento continue correto após a exclusão.
3. O espaço antes alocado pela cabeça é liberado.
4. A cabeça aponta para o nó agora correspondente.

Figura 41 – Diagrama de Nós e Ligações para Remoção na Lista

```
ANTES: ... <—> [Imagem 1] <—> [Imagem 2] <—> [Imagem 3] <—> ...  
REMOVE 'Imagem 1'  
DEPOIS: ... <—> [Imagem 2] <—> [Imagem 3] <—> ...
```

Fonte: Leal, Andreas et al, outubro de 2025

Logo acima, a figura representa uma lista com 3 imagens ligadas entre si. Cada nó contém o nome da imagem e seus 2 ponteiros que apontam para os nós: anterior e próximo. Após a remoção do 1º nó, o 2º nó passa a ser a nova cabeça da lista.

Figura 42 – Função para Buscar uma Imagem no Carrossel

```
void search(Node *head, char label[]) {  
    Node *node = head;  
    int cnt = 0;  
    do {  
        if (strcmp(node->label, label) == 0) {  
            cnt++;  
        }  
        node = node->next;  
    } while (node != head);  
  
    if (cnt > 0) {  
        printf("%s.png esta na lista!\n", label);  
        printf("Aparece %d vezes...\n", cnt);  
    } else {  
        printf("%s.png nao esta na lista!\n", label);  
    }  
}
```

Fonte: Leal, Andreas et al, outubro de 2025

A função **search()** percorre a lista através de um ponteiro local **node**. Caso o resultado seja positivo entre a comparação da string passada para a função **search()** e a string presente no nó atual, um contador local, denominado **cnt** será auto-incrementado, a fim de entregar a quantidade de vezes na qual a imagem se repete na lista. Se ao final da inspeção o valor de **cnt** for 0, o programa retorna que a imagem **não** se encontra no carrossel.

Figura 43 – Função para Exibir o Carrossel - Parte 1/3

```
void show(Node *head) {  
    Node *node = head;  
    int choice, simbol, no = 0;  
    partialFrame();  
    printf("Selecione um modo de exibicao: \n");  
    printf("[1] - Navegacao\n");  
    printf("[2] - Automatico\n");  
    printf("+=====+\n");  
    printf("Digite: ");  
    scanf("%d", &choice);  
}
```

Fonte: Leal, Andreas et al, outubro de 2025

A função **show()** se inicia com uma solicitação pelo modo de exibição do carrossel ao usuário, oferecendo as opções de **navegação** [1] ou **automático** [2].

Saída esperada:

Figura 44 – Saída da Função para Exibir o Carrossel - Parte 1/3

```
+=====+  
|°°°°°°| Carrossel de Imagem |°°°°°°|  
+=====+  
Selecione um modo de exibicao:  
[1] - Navegacao  
[2] - Automatico  
+=====+  
Digite: █
```

Fonte: Leal, Andreas et al, outubro de 2025

O programa espera até que uma opção seja selecionada. Para escolher, basta que o usuário digite o número correspondente à opção desejada e tecle **[ENTER]**.

Figura 45 – Função para Exibir o Carrossel - Parte 2/3

```
if (choice == 1) {
    do {
        partialFrame();
        printf("Arquivo: %s.png\n", node->label);
        switch (no){
            case 0:
                print01();
                break;
            case 1:
                print02();
                break;
            default:
                print03();
                break;
        }
        printf("[1] Retroceder\n");
        printf("[2] Avancar\n");
        printf("[0] Parar\n\n");
        printf("Digite: ");
        scanf("%d", &simbol);
        if (simbol == 2){
            node = node->next;
            if (no == 2) {
                no = 0;
            } else {
                no++;
            }
        } else if (simbol == 1) {
            node = node->prev;
            if (no == 0) {
                no = 2;
            } else {
                no--;
            }
        } else {
            choice = 0;
        }
    } while (choice != 0);

    return;
}
```

Fonte: Leal, Andreas et al, outubro de 2025

Será verificado o valor de **simbol** para decidir se **node** receberá o endereço do próximo nó, ou do anterior. Para ilustrar melhor o exemplo de carrossel, imagens feitas com prints e caracteres são exibidas a partir das funções **print01**, **print02** e **print03**. O valor da variável **no** é manipulado a fim de simular a navegação entre imagens reais de um carrossel.

Saída esperada:

Figura 46 – Saída da Função para Exibir o Carrossel - Parte 2/3

```
+=====+
|°°°°°°| Carrossel de Imagem |°°°°°°|
+=====+
Arquivo: teste.png

  .--.
  |o_o |
  |:_/ |
 //     \ \
 (|     |)
 /'\_   _/'\
 \___)=(___/_

[1] Retroceder
[2] Avancar
[0] Parar

Digite:
```

Fonte: Leal, Andreas et al, outubro de 2025

O programa permite que o usuário avance e retroceda entre as imagens. Ao chegar no último elemento e selecionar na opção **avancar**, o programa retorna o primeiro elemento. Caso selecione **retroceder** a partir deste, o programa retornará ao último elemento.

Figura 47 – Função para Exibir o Carrossel - Parte 3/3

```
do {
    partialFrame();
    printf("Arquivo: %s.png\n", node->label);
    if (no == 0) {
        print01();
        no++;
    } else if (no == 1) {
        print02();
        no++;
    } else {
        print03();
        no = 0;
    }
    fflush(stdout);
    sleep(TIME);
    node = node->next;
} while (node != head);
}
```

Fonte: Leal, Andreas et al, outubro de 2025

Se a opção selecionada for 2, o programa executa a função **partialFrame()** para exibir o layout, o nome do arquivo atual e a imagem ilustrativa. Por fim, através da função **sleep()** o programa espera 2 segundos. O nó apontado passa a ser o próximo e o fluxo se repete até que se chegue à última imagem do carrossel.

Saída esperada:

Figura 48 – Saída da Função para Exibir o Carrossel - Parte 3/3

```
+=====+
|°°°°°°| Carrossel de Imagem |°°°°°°|
+=====+
Arquivo: teste.png

      .--.
      |o_o|
      |:~/|
      //  \ \
      (|    |)
      /'\_  _/'\
      \___)=(___/

Pressione [ENTER] para continuar
```

Fonte: Leal, Andreas et al, outubro de 2025

O programa passa pelos nós e os exibe até que chegue ao fim da lista. As imagens são trocadas de forma sequencial, simulando a movimentação do carrossel.

Casos de Borda:

1. **Lista vazia:** o sistema exibe “**lista vazia!**” e solicita a inserção de novas imagens a partir da opção “**adicionar**”.
2. **Remover quando há apenas um elemento:** o programa verifica com antecedência se há apenas um elemento e chama a função correspondente para deletar a lista.
3. **Selecionar opções que não estão no menu:** o programa entende qualquer outra opção como “sair” e é finalizado.

Vantagens:

1. **Navegação cíclica:** Por ser circular, o último elemento aponta para o primeiro, e o primeiro aponta para o último. Isso é o que um carrossel faz. Quando o usuário está na última imagem e clica em “próximo”, ele é direcionado para a primeira.

2. **Navegação Bidirecional:** Por ser duplamente encadeada, cada nó (imagem) tem um ponteiro próximo e um ponteiro anterior. Isso torna a navegação para frente e para trás mais eficiente.

Desvantagens:

1. **Uso de Memória:** Cada nó na lista precisa armazenar dois ponteiros, além do dado (a imagem). Uma lista simplesmente encadeada ou um array usariam menos memória para os ponteiros (um ou nenhum, respectivamente).
2. **Complexidade de Implementação:** Gerenciar dois ponteiros por nó, e ainda garantir que a circularidade seja mantida, é mais complexo de programar do que um simples array.
3. **Falta de Acesso Direto (Aleatório):** Se a intenção for pular diretamente para a 5ª imagem, não será possível simplesmente acessar o índice [4]. Seria necessário percorrer a lista a partir da imagem atual (ou do início) até chegar à quinta. Para um carrossel com muitas imagens, isso pode ser menos eficiente do que um array.

10. CONCLUSÃO

Portanto, neste trabalho foi possível compreender o funcionamento e as aplicações das listas encadeadas que são estruturas fundamentais na área da programação. Nesse documento, foi abordado sobre 4 principais tipos de listas encadeadas, a **lista simplesmente encadeada**, **lista duplamente encadeada**, **lista circular simples** e **lista duplamente circular**, e embora cada uma compartilhe algo em comum como os nós conectados por ponteiros, cada um possui uma **característica principal** que determina o seu uso.

A **lista simplesmente encadeada** é ideal para implementações mais simples com um menor custo de memória, já a **lista duplamente encadeada** se mostra útil por causa da sua flexibilidade de navegação em seus 2 sentidos, para o próximo e para o anterior, sendo utilizada em aplicações como playlist de música, assim como no código que foi apresentado. As **listas circulares** trazem o conceito de uma lista cíclica, útil para quando se exige uma repetição dentro do sistema, e por fim a **lista duplamente circular** traz o conceito tanto de lista duplamente encadeada quanto de lista circular.

Dessa forma, o estudo mostrou que utilizar as listas encadeadas é essencial para fazer com que um algoritmo rode bem e se torne bem adaptável, especialmente em C que faz com que o desenvolvedor cuide da memória manualmente.

11. REFERÊNCIAS

CORMEN, Thomas H.; LEISERSON, Charles E.; RIVEST, Ronald L.; STEIN, Clifford. *Algoritmos: teoria e prática*. 3. ed. Rio de Janeiro: Elsevier, 2012.

GEEKSFORGEEKS. *Linked List Data Structure*. [S. l.]: GeeksforGeeks, [s.d.]. Disponível em: <https://www.geeksforgeeks.org/data-structures/linked-list/>. Acesso em: 15 out. 2025.

SEDGEWICK, Robert; WAYNE, Kevin. *Algorithms*. 4. ed. Boston: Addison-Wesley Professional, 2011.

TENENBAUM, Aaron M.; LANGSAM, Yedidyah; AUGENSTEIN, Moshe J. *Estruturas de dados usando C*. São Paulo: Makron Books, 1995.