



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

GEOVANNI MARTINS DE SOUZA
JEOVANA MIRANDA SOUZA
LÍVIA ALKIMIM DOS SANTOS

RELATÓRIO TÉCNICO SOBRE LISTAS DINÂMICAS

CAMPUS GUANAMBI

2025

GEOVANNI MARTINS DE SOUZA
JEOVANA MIRANDA SOUZA
LÍVIA ALKIMIM DOS SANTOS

RELATÓRIO TÉCNICO SOBRE LISTAS DINÂMICAS

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 didático.....	
Figura 2: Estrutura básica da lista simplesmente encadeada.....	
Figura 3: Inserção de elementos na lista.....	
Figura 4: Remoção do último elemento da lista.....	
Figura 5: Exibição da lista.....	
Figura 6: Exemplo didático de lista duplamente encadeada.....	
Figura 7: Estrutura básica da lista duplamente encadeada.....	
Figura 8: Inserção de elemento na lista duplamente encadeada.....	
Figura 9: Exemplo didático de Lista Circular.....	
Figura 10: Função para iniciar a lista circular.....	
Figura 11: Função de busca em uma lista circular.....	
Figura 12: Exemplo didático de uma lista duplamente ligada circular.....	
Figura 13: Inserção de novo nó.....	
Figura 14: Busca por um valor.....	
Figura 15: Remoção de um nó específico em lista duplamente circular.....	
Figura 16: Comparação entre os tipos de lista.....	
Figura 17: Função voidInserirArquivo.....	
Figura 18: função removerArquivo.....	
Figura 19: Continuação da função removerArquivo.....	
Figura 20: função exibirFila.....	
Figura 21: Função imprimirArquivo.....	
Figura 22: Função inserirUrl.....	
Figura 23: Função removeUrl.....	
Figura 24: Função navegar.....	
Figura 25: Função exibirHistorico.....	
Figura 26: Função criar.....	
Figura 27: Função imprimir.....	
Figura 28: função remover.....	
Figura 29: Função buscar.....	
Figura 30: criar lista.....	
Figura 31: Métodos de busca distintos.....	
Figura 32: Função playMusic.....	
Figura 33: Função de reprodução de músicas.....	

SUMÁRIO

1. Introdução ao Tema.....	
2. Explicação teórica.....	
2.1.Comparação entre os tipos de lista.....	
3. Aplicação Prática	
3.1 Aplicação prática de lista simples.....	
3.2 Aplicação prática de lista dupla.....	
3.3 Aplicação prática de Lista Circular.....	
3.4 Aplicação Prática de lista duplamente circular.....	
4. Conclusão	
5. Bibliografia	

1. Introdução ao Tema

Uma lista linear é uma sequência de $n \geq 0$ nós $X[1]$, $X[2]$, ..., $X[n]$, cujas propriedades estruturais essenciais envolvem apenas as posições relativas entre os itens, conforme aparecem em uma linha. Quando $n = 0$ a lista é chamada nula ou vazia. (KNUTH, 1997, p. 239, tradução nossa).

Conceitos Básicos:

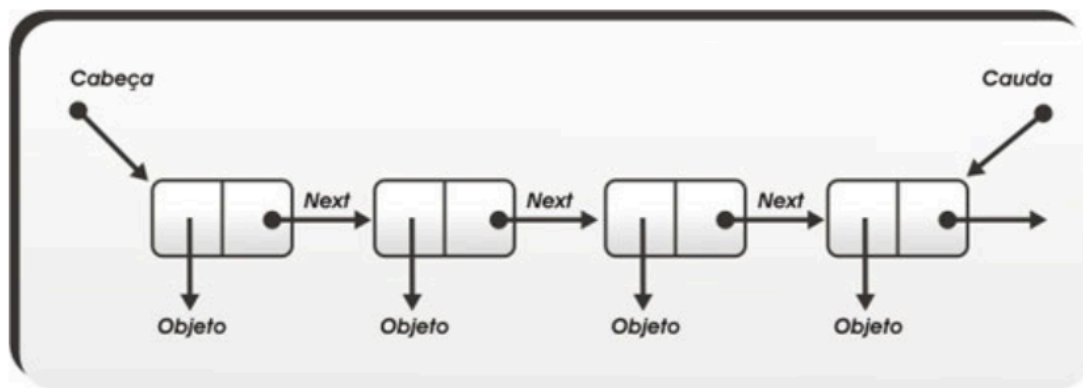
- **Nós (Nodes):** É a unidade básica de uma lista, responsável por armazenar um valor e, quando aplicável, um ponteiro para outro nó.
- **Ponteiro (Pointer):** É uma referência que indica a posição de outro elemento na memória, permitindo que os nós fiquem conectados.
- **Inserção (Insertion):** adiciona um novo nó em uma posição específica.
- **Remoção (Remove):** Faz a desalocação do nó existente.
- **Percurso (Traversal):** Acessa sequencialmente todos os nós.
- **Busca (Search):** Localiza nós com valores específicos.

Ela armazena uma sequência finita de elementos (geralmente do mesmo tipo) que é manipulada por operações fundamentais como: acessar elementos, inserir, excluir e percorrer a sequência é chamada de fila. São discutidas implementações concretas (listas estáticas em vetores/arrays e listas dinâmicas encadeadas), questões de organização de memória e análise do custo das operações (complexidade).

2. Explicação teórica

Lista Simplesmente Encadeada: Estrutura de dados linear em que cada nó contém dois componentes, sendo eles um campo de dado e um ponteiro/referência para o próximo nó na sequência. O primeiro nó é chamado de head; o último nó aponta para NULL (ou equivalente), indicando o fim da lista. A travessia (percurso) é possível somente em um sentido, do head até o nó final. Operações típicas: inserção, remoção, pesquisa, percurso.

Figura 1: Exemplo didático.



Fonte: Exemplo de funcionamento de uma lista encadeada STACK OVERFLOW EM PORTUGUÊS (2025).

A estrutura é simples, possuindo um campo do tipo inteiro chamado valor (poderia ser qualquer tipo, inclusive outro tipo mais complexo) e um ponteiro para um próximo nó.

Figura 2: Estrutura básica da lista simplesmente encadeada.

```
1. typedef struct No {
2.     int valor;
3.     struct No *proximo;
4. } No;
```

Fonte: Martins, Geovanni et al, outubro de 2025.

Na sequência, faremos a inserção de elementos na lista encadeada. A seguir, é apresentado o procedimento para inserir um novo elemento no início da lista.

Figura 3: Inserção de elementos na lista.

```
70 void iniciarLista(struct cabecaLista *ptr) {
71     struct no *novo = malloc(sizeof *novo);
72     printf("Insira o valor inicial: ");
73     scanf("%d", &novo->valor);
74     novo->proximo = NULL;
75     ptr->cabeca = novo;
76 }
```

Fonte: Martins, Geovanni et al, outubro de 2025.

Após, o próximo passo é a remoção do último elemento da lista

Figura 4: Remoção do último elemento da lista.

```
110 int removeUltimo(struct cabecalista *ptr) {
111     if (ptr->cabeca == NULL) return 0;
112
113     struct no *atual = ptr->cabeca;
114     struct no *anterior = NULL;
115
116     while (atual->proximo != NULL) {
117         anterior = atual;
118         atual = atual->proximo;
119     }
120
121     if (anterior == NULL) {
122         free(ptr->cabeca);
123         ptr->cabeca = NULL;
124     } else {
125         anterior->proximo = NULL;
126         free(atual);
127     }
128
129     return 1;
130 }
```

Fonte: Martins, Geovanni et al, outubro de 2025.

Para finalizar, exibimos a lista na tela

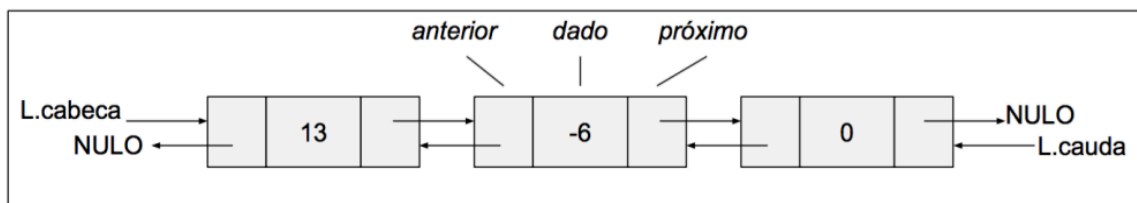
Figura 5: Exibição da lista.

```
96 void exibirLista(struct cabecalista *ptr) {
97     struct no *atual = ptr->cabeca;
98     if (atual == NULL) {
99         printf("Lista vazia!");
100         return;
101     }
102
103     while (atual != NULL) {
104         printf("%d ", atual->valor);
105         atual = atual->proximo;
106     }
107     printf("\n");
108 }
```

Fonte: Martins, Geovanni et al, outubro de 2025.

Lista Duplamente Encadeada: A lista duplamente encadeada é formada por nós similares ao da lista encadeada simples, porém com um atributo adicional: um apontador para o nó anterior. Dessa forma, é possível percorrer esse tipo de lista em ambas as direções.

Figura 6: Exemplo didático de lista duplamente encadeada.



Fonte: Estrutura de uma lista duplamente encadeada GRÉGIO (2025).

O nó representado na figura acima pode ser criado da seguinte forma:

Figura 7: Estrutura básica da lista duplamente encadeada.

```

2.      Estrutura nó para a lista duplamente encadeada
3.      */
4.      typedef struct no{
5.          int valor;
6.          struct no *proximo;
7.          struct no *anterior;
8.      }No;

```

Fonte: Martins, Geovanni et al, outubro de 2025.

Como mencionado, na lista duplamente encadeada cada alteração na lista precisa atualizar os dois ponteiros de cada nó envolvido na operação. A seguir é apresentado um procedimento para inserir um novo nó no início da lista dupla. Pode-se analisar que os dois ponteiros são atualizados de acordo com a operação realizada, neste caso uma inserção no início.

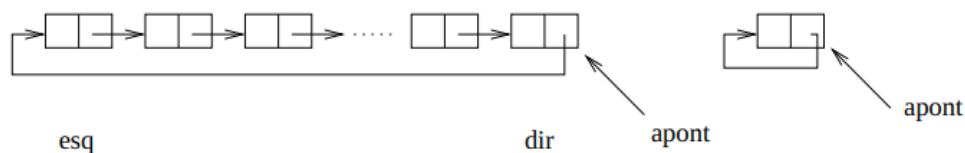
Figura 8: Inserção de elemento na lista duplamente encadeada

```
1.  /*  
2.      procedimento para inserir um novo nó no início da lista  
3.  */  
4.  void inserir_no_inicio(No **lista, int num){  
5.      No *novo = malloc(sizeof(No));  
6.  
7.      if(novo){  
8.          novo->valor = num;  
9.          // próximo do novo nó aponta para o início da lista  
10.         novo->proximo = *lista;  
11.         // o anterior é nulo pois é o primeiro nó  
12.         novo->anterior = NULL;  
13.         // se a lista não estiver vazia, o anterior do primeiro nó aponta para o novo nó  
14.         if(*lista)  
15.             (*lista)->anterior = novo;  
16.         // o novo nó passa a ser o início da lista (o primeiro nó da lista)  
17.         *lista = novo;  
18.     }  
19.     else  
20.         printf("Erro ao alocar memoria!\n");  
21. }
```

Fonte: Martins, Geovanni et al, outubro de 2025.

Lista Circular Simples: Uma lista simples tem a propriedade que o último nó aponta sempre para o primeiro nó. Então, de qualquer nó da lista pode-se atingir qualquer outro nó da lista.

Figura 9: Exemplo didático de Lista Circular



Fonte: Estrutura de uma lista circular GOLD (2001).

A principal complexidade da lista circular está no fato de que, em qualquer operação na lista, precisamos manipular vários ponteiros, mantendo sempre o último nó apontando para o primeiro nó. Nos trechos de código seguintes, são apresentadas as funções para inserção no início da lista e busca. Ao inserir no início, o primeiro nó é alterado fazendo com que o ponteiro do último nó seja alterado consequentemente visto que, o último ponteiro aponta para o primeiro nó da lista. Já na função de busca em uma lista circular consiste em percorrer os

elementos da lista até encontrar o valor desejado ou retornar ao início, caso o elemento não exista. Diferente de uma lista linear, a lista circular não possui um “fim” explícito já que o último nó aponta novamente para o primeiro, formando um ciclo. Por isso, a busca é implementada com um laço do-while, que garante que o primeiro nó seja verificado antes de comparar com o início. Se o valor buscado for encontrado, a função retorna o nó correspondente; caso contrário, retorna “NULL”.

Figura 10: Função para iniciar a lista circular

```
4. void inserir_no_inicio(Lista *lista, int num){
5.     No *novo = malloc(sizeof(No));
6.
7.     if(novo){
8.         novo->valor = num;
9.         // o próximo aponta para o início da lista
10.        novo->proximo = lista->inicio;
11.        // novo se torna o início da lista
12.        lista->inicio = novo;
13.        // se fim for nulo (lista vazia), fim aponta para novo nó
14.        if(lista->fim == NULL)
15.            lista->fim = novo;
16.        // fim aponta para início
17.        lista->fim->proximo = lista->inicio;
18.        lista->tam++;
19.    }
20.    else
21.        printf("Erro ao alocar memoria!\n");
22. }
```

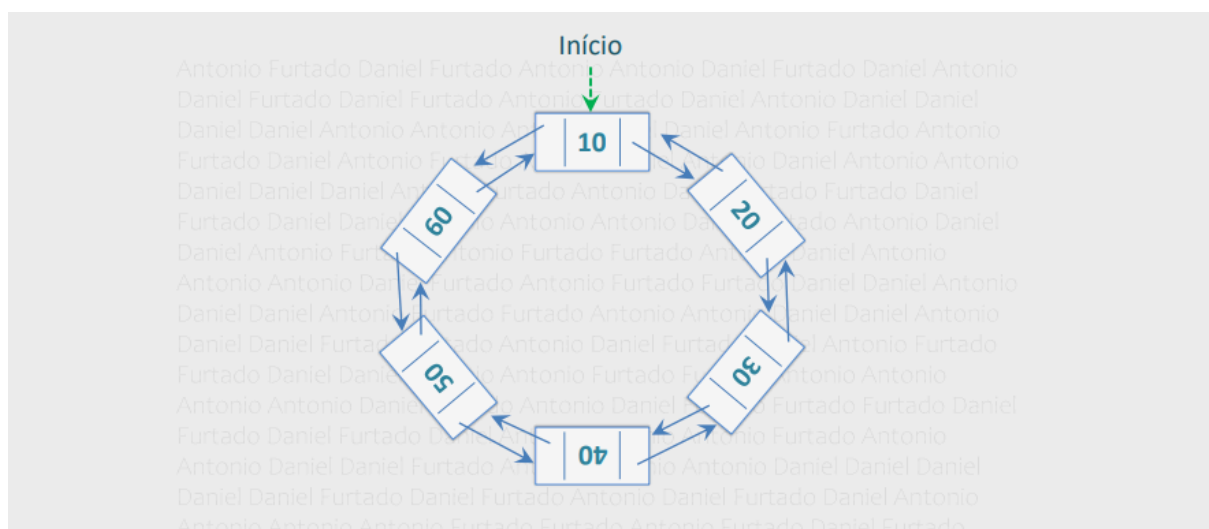
Fonte: Martins, Geovanni et al, outubro de 2025.

Figura 11: Função de busca em uma lista circular

```
136. // função para buscar um valor
137. No* buscar(Lista *lista, int num){
138.     No *aux = lista->inicio;
139.
140.     if(aux){
141.         do{
142.             if(aux->valor == num)
143.                 return aux;
144.             aux = aux->proximo;
145.         }while(aux != lista->inicio);
146.     }
147.     return NULL;
148. }
```

Fonte: Martins, Geovanni et al, outubro de 2025.

Lista Duplamente Circular: A lista duplamente circular é uma variação da lista duplamente encadeada, diferenciando-se pelo fato de o último nó apontar para o primeiro, formando assim um ciclo contínuo. Esse tipo de estrutura permite percorrer a lista em ambas as direções de forma ininterrupta, conforme ilustrado na imagem a seguir.



Fonte: FURTADO, Daniel A. Listas Duplamente Ligadas Circulares.

O último nó se liga ao primeiro pelo next, e o primeiro se liga ao último pelo prev, formando um ciclo. Existe apenas uma referência externa para o primeiro nó, mas a partir de qualquer nó é possível percorrer a lista tanto para frente quanto para

trás. As inserções e remoções podem ser feitas de forma eficiente no início ou no final da lista, sem depender da referência externa para o nó anterior.

Figura 12: Estrutura do nó da lista duplamente circular

```
4
5 // Estrutura do nó da lista duplamente circular
6 typedef struct no {
7     int valor;
8     struct no *next;
9     struct no *prev;
10 } No;
```

Fonte: Martins, Giovanni et al, outubro de 2025.

Figura 13: Inserção de novo nó

```
12 // Função para criar um novo nó
13 No* criarNo(int valor) {
14     No* novo = (No*)malloc(sizeof(No));
15     if (!novo) {
16         printf("Erro de alocação!\n");
17         exit(1);
18     }
19     novo->valor = valor;
20     novo->next = novo->prev = novo;
21     return novo;
22 }
```

Fonte: Martins, Giovanni et al, outubro de 2025.

Figura 14: Busca por um valor

```

46 // Busca por valor
47 No* buscar(No* inicio, int chave) {
48     if (!inicio) return NULL;
49
50     No* temp = inicio;
51     do {
52         if (temp->valor == chave)
53             return temp;
54         temp = temp->next;
55     } while (temp != inicio);
56
57     return NULL; // não encontrado
58 }

```

Fonte: Martins, Giovanni et al, outubro de 2025.

Para a remoção de um nó específico, seja por posição ou valor, é necessário o apontamento de um nó para outro e atualização do ponteiro para manter a integridade da lista circular. Ou seja, O nó anterior (prev) do nó removido deve apontar para o próximo nó (prox) do nó removido. O nó seguinte (prox) deve apontar de volta para o nó anterior (prev) do nó removido. Se a cabeça da lista for removida, é necessário atualizar o ponteiro da cabeça.

Figura 15: Remoção de um nó específico em lista duplamente circular

```

// Remoção de um nó por valor
No* remover(No* inicio, int chave) {
    if (!inicio) return NULL;

    No* temp = inicio;
    do {
        if (temp->valor == chave) {
            if (temp->next == temp) { // único nó
                free(temp);
                return NULL;
            }
            temp->prev->next = temp->next;
            temp->next->prev = temp->prev;
            if (temp == inicio)
                inicio = temp->next; // atualiza início se necessário
            free(temp);
            return inicio;
        }
        temp = temp->next;
    } while (temp != inicio);

    return inicio; // valor não encontrado
}

```

Fonte: Martins, Giovanni et al, outubro de 2025.

2.1. Comparação entre os tipos de lista

Como já discutido, listas encadeadas são estruturas de dados lineares cujos elementos (nós) não ocupam posições contíguas na memória. Cada nó contém um ponteiro que referencia o próximo elemento e, em alguns casos, também o nó anterior (KNUTH, 1999). Essa característica proporciona maior flexibilidade para inserções e remoções em tempo de execução, diferentemente dos arrays, que exigem realocação de memória. A seguir, apresenta-se uma comparação entre os principais tipos de listas encadeadas, destacando suas características, navegações e aplicações.

Figura 16: Comparação entre os tipos de lista

Tipo de lista	Ponteiros	Navegação	Aplicações comuns
Simplesmente encadeada	1 (próximo)	Apenas frente	Pilhas, listas lineares simples
Duplamente encadeada	2 (próximo e anterior)	Frente e trás	Editores de texto, histórico
Circular	1 (próximo)	Ciclo contínuo	Filas circulares, escalonamento
Duplamente circular	2 (próximo e anterior)	Ciclo contínuo bidirecional	Buffers, jogos, filas de prioridade

Fonte: Martins, Geovanni et al, outubro de 2025.

3. Aplicação Prática

Nesta seção, serão apresentados exemplos práticos que utilizam os quatro tipos de lista estudados na disciplina de Estrutura de Dados, acompanhados de suas respectivas justificativas e do passo a passo de implementação dos códigos.

3.1 Aplicação prática de lista simples: Fila de impressão seguindo o princípio FIFO (First In, First Out)

O objetivo desta prática foi aplicar os conhecimentos sobre a estrutura de lista simples, incluindo a criação de funções para sua implementação e a manipulação de ponteiros e endereços de memória. Para facilitar a compreensão da proposta didática, foram desenvolvidas as operações de inserir, remover, atualizar e exibir, utilizando um exemplo hipotético de simulação de uma fila de impressora. Nessa simulação, os arquivos são adicionados e processados conforme o escalonamento de processo FIFO (*First In, First Out*), no qual os arquivos são inseridos em uma fila e, ao final, é impresso aquele que foi adicionado primeiro.

Conforme ilustrado na Figura 17, a função `inserirArquivo` tem como objetivo inserir o primeiro nó, defini-lo como a cabeça da lista e alocar a quantidade de nós especificada pelo usuário por meio da função `malloc`. Como boa prática, caso a variável que armazena a cabeça da lista esteja vazia, a função deve retornar `NULL`.

Observação: a utilização da função `getchar` demonstra o domínio de boas práticas, pois evita resíduos deixados na entrada padrão. Além disso, o uso da função `fgets` permite a leitura de cadeias de caracteres que contêm espaços.

Figura 17: Função `voidInserirArquivo`

```
81 void inserirArquivo(struct no **ptrCabeca) {
82     // auxiliar para não modificar o principal
83     struct no *atual = *ptrCabeca;
84     // armazena o espaço do novo arquivo
85     struct no *novoArquivo = malloc(sizeof *novoArquivo);
86     novoArquivo->prox = NULL;
87
88     // cancela resíduos que podem ter ficado na entrada
89     getchar();
90
91     printf("Qual o nome do arquivo (+ extensão)? \n");
92     // lê o arquivo, com permissão para espaços
93     fgets(novoArquivo->arquivo, sizeof(novoArquivo->arquivo), stdin);
94
95     // remove o caractere que simboliza o 'enter' ao final da string
96     novoArquivo->arquivo[strcspn(novoArquivo->arquivo, "\n")] = '\0';
97
98     // significa que a lista ainda não possui arquivos
99     if (*ptrCabeca == NULL) {
100         *ptrCabeca = novoArquivo;
101
102         printf("%s adicionado com sucesso\n", (*ptrCabeca)->arquivo);
103         return;
104     } else {
105         // looping até achar o nó que aponta para NULL como próximo
106         while (atual->prox != NULL) {
107             atual = atual->prox;
108         }
109
110         // quem era o último começa a apontar para o novo;
111         atual->prox = novoArquivo;
112         printf("%s adicionado com sucesso\n", novoArquivo->arquivo);
113     }
114 }
115 }
```


Fonte: Martins, Geovanni et al, outubro de 2025.

Conforme ilustrado na Figura 18 e 19, a função `removerArquivo` tem como objetivo armazenar o nó que será removido, definido pelo usuário, e o próximo nó que assumirá a posição do elemento excluído. Foi utilizada a estrutura `switch case` para realizar perguntas ao usuário sobre o modo de remoção desejado, oferecendo duas opções: por nome ou por índice. Um laço `while` é empregado para percorrer a lista enquanto houver nós e o arquivo não for encontrado. No caso da remoção por índice, o contador inicia em 1 e é comparado com o tamanho total da lista. Caso o número informado pelo usuário seja maior que o tamanho da lista, uma mensagem de aviso é exibida. Se o valor informado for 0, a função retorna imediatamente. Para localizar o índice solicitado, é executado um loop que incrementa a variável `i` até que seu valor seja igual ao índice desejado (`i++` a cada iteração). Além disso, se o nó a ser removido for a cabeça da lista, a variável temporária passa a apontar para `NULL`, e o ponteiro principal é atualizado para o próximo nó, que se torna a nova cabeça da lista.

Figura 18: função `removerArquivo`

```

117 void removerArquivo(struct no **ptrCabeca) {
118     // para deletar o arquivo é o que ele deletou
119     struct no *tempAntecessor = NULL, *atual = *ptrCabeca;
120     // cancela resíduos que podem ter ficado na entrada
121     getchar();
122
123     int escolhaUser;
124     printf("Como você quer remover? [1] Pelo Nome; [2] Pelo Índice\n");
125     scanf("%d", &escolhaUser);
126
127     switch (escolhaUser) {
128         // por nome
129         case 1:
130             char nomeArquivoDeletar[80];
131             // cancela resíduos que podem ter ficado na entrada
132             getchar();
133
134             // pergunto ao usuário o nome do arquivo e após isso salvo esse nome
135             printf("Qual o nome do arquivo que você gostaria de remover? Escreva o nome correto, junto com sua extensão\n");
136
137             fgets(nomeArquivoDeletar, sizeof(nomeArquivoDeletar), stdin);
138             // remove o 'enter' que acaba sendo salvo no final
139             nomeArquivoDeletar[strcspn(nomeArquivoDeletar, "\n")] = '\0';
140
141             // looping até encontrar o arquivo (enquanto tiver nós e enquanto não tiver encontrado o arquivo)
142             while (atual != NULL && strcmp(atual->arquivo, nomeArquivoDeletar) != 0) {
143                 tempAntecessor = atual;
144                 atual = atual->prox;
145             }
146
147             // vê se o arquivo foi encontrado
148             if (atual == NULL) {
149                 printf("Arquivo não encontrado\n");
150                 return;
151             }
152
153             // se for o primeiro nó
154             if (tempAntecessor == NULL) {
155                 *ptrCabeca = atual->prox;
156             } else {
157                 tempAntecessor->prox = atual->prox;
158             }
159
160             // libera o nó deletado
161             free(atual);
162             atual = NULL;
163
164             printf("Removido com sucesso!\n");
165             break;
166
167         // por índice
168         case 2:
169             int i = 1, indiceEscolhaUser, tamanhoDaLista = tamanhoLista(ptrCabeca);
170
171             // pede o índice ao usuário
172             printf("Qual o índice que você gostaria de remover? \n");
173             scanf("%d", &indiceEscolhaUser);

```

Fonte: Martins, Geovanni et al, outubro de 2025.

Figura 19: Continuação da função removerArquivo

```
174
175 // vê se o número que o usuário escolheu é possível de estar na lista
176 if (tamanhoDaLista == 0) {
177     printf("A lista não possui nós\n");
178     return;
179 } else if (indiceEscolhaUser > tamanhoDaLista || indiceEscolhaUser < 0) {
180     printf("O número escolhido está fora do tamanho da lista\n");
181     return;
182 }
183
184 // fazer um looping para ir contabilizando até encontrar o índice correto
185 while (i != indiceEscolhaUser) {
186     tempAntecessor = atual;
187     atual = atual->prox;
188     i++;
189 }
190
191 // verifica se é o primeiro
192 if (tempAntecessor == NULL) {
193     *ptrCabeca = atual->prox;
194 } else {
195     tempAntecessor->prox = atual->prox;
196 }
197
198 // libera o nó deletado
199 free(atual);
200 atual = NULL;
201
202 printf("Removido com sucesso!\n");
203 break;
204 default:
205     printf("Opção Inexistente\n");
206 }
```

Fonte: Martins, Giovanni et al, outubro de 2025.

Como ilustrado na Figura 20, a função `exibirFila` utiliza um nó denominado `ptrl`, que armazena o endereço da cabeça da fila. Em seguida, é executado um laço que, enquanto essa variável for diferente de `NULL`, imprime o conteúdo armazenado e avança para o próximo nó.

Figura 20: função exibirFila

```
209 void exibirFila(struct no **ptrCabeca) {
210     // define um ponteiro "iterador" que começa pela cabeça
211     struct no *ptrI;
212     ptrI = (*ptrCabeca);
213
214     // enquanto o endereço dele não for NULL
215     while(ptrI != NULL){
216         // printa o arquivo e passa para o próximo
217         printf("%s ", ptrI->arquivo);
218         ptrI = ptrI->prox;
219     }
220     printf("\n");
221 }
222
```

Fonte: Martins, Geovanni et al, outubro de 2025.

Como ilustrado na Figura 21, a função imprimirArquivo armazena o endereço do primeiro nó e recebe o ponteiro da cabeça da lista. Caso ptr seja igual a NULL, a função retorna imediatamente. Caso contrário, o conteúdo da cabeça é impresso e ptr passa a apontar para o próximo nó. Como boa prática de programação, o espaço de memória ocupado pelo nó atual, ou seja, pelo arquivo já impresso é liberado por meio da função free.

Figura 21: Função imprimirArquivo

```
234 void imprimirArquivo(struct no **ptrCabeca) {
235     // guarda a cabeça
236     struct no *atual = *ptrCabeca;
237
238     // verifica se dá para imprimir
239     if (tamanhoLista(ptrCabeca) == 0) {
240         printf("Não tem arquivo para imprimir\n");
241         return;
242     }
243
244     // imprime a cabeça
245     printf("%s\n", (*ptrCabeca)->arquivo);
246
247     // cabeça passa a apontar para o próximo
248     *ptrCabeca = atual->prox;
249
250     // libera o arquivo impresso
251     free(atual);
252     atual = NULL;
253 }
```

Fonte: Martins, Geovanni et al, outubro de 2025.

3.2 Aplicação prática de lista dupla: Simulação de navegação entre páginas O objetivo desta prática foi aplicar os conhecimentos sobre a estrutura de lista duplamente encadeada, incluindo a criação de funções para sua implementação e a manipulação de ponteiros e endereços de memória. Em essência, a função de um histórico de navegador é armazenar, navegar, inserir e remover as abas visitadas. Essa aplicação foi desenvolvida com o propósito de simular o funcionamento de uma lista duplamente encadeada nesse contexto, implementando operações como adicionar novas abas (somente ao final, já que não é possível inserir abas no passado), excluir abas do histórico, exibir todas as abas e navegar entre elas, avançando para a próxima ou retornando à anterior.

Conforme ilustrado na Figura 22, a função inserirUrl possui três parâmetros: struct página **cabeca, que representa a primeira página (histórico); struct página **final, que indica a última aba acessada; e struct página **atual, que corresponde à aba atual do usuário. O comando sizeof *NovaUrl é utilizado para alocar espaço na

memória conforme o tamanho da estrutura. Em seguida, NovaUrl aponta para esse novo espaço, e a função fgets realiza a leitura da URL. Caso a lista esteja vazia, o novo nó torna-se a cabeça, o atual e o final, apontando NULL para os lados. Quando já existe histórico, o novo nó aponta para o anterior, o antigo final aponta para ele e o novo final passa a apontar para NULL.

Figura 22: Função inserirUrl

```
70 void inserirUrl(struct pagina **cabeca, struct pagina **final, struct pagina **atual) {
71     struct pagina *NovaUrl = malloc(sizeof *NovaUrl);
72
73     printf("Digite a URL que quer acessar:\n");
74
75     // lê a URL
76     fgets(NovaUrl->url, sizeof(NovaUrl->url), stdin);
77
78     // remove o caracter que simboliza o 'enter' ao final da string
79     NovaUrl->url[strcspn(NovaUrl->url, "\n")] = '\0';
80
81     if (*cabeca == NULL) {
82         *cabeca = NovaUrl;
83         *atual = NovaUrl;
84         *final = NovaUrl;
85
86         // por ser a cabeça ele aponta NULL para os outros lados
87         NovaUrl->ant = NULL;
88         NovaUrl->prox = NULL;
89
90     } else {
91         // só é possível adicionar no final
92         // faz o final caminhar para o próximo e conecta os ponteiros
93
94         // o novo nó vai apontar para o anterior
95         NovaUrl->ant = *final;
96         // o antigo final aponta para o próximo
97         (*final)->prox = NovaUrl;
98         // novo final aponta para NULL
99         NovaUrl->prox = NULL;
100        // o novo nó agora é o final
101        *final = NovaUrl;
102    }
103 }
```

Fonte: Martins, Geovanni et al, outubro de 2025.

Conforme ilustrado na Figura 23, a função removeUrl utiliza os mesmos três parâmetros da função anterior e acrescenta duas novas variáveis: urlRemovida, que armazena a URL digitada pelo usuário (ou seja, a URL que ele deseja remover), e *ptrl, que representa o ponteiro para a cabeça da lista. Enquanto ptrl for diferente de NULL, o laço é executado para verificar diferentes condições. A primeira delas

consiste em checar se a URL informada corresponde à cabeça da lista. Caso seja, o ponteiro é atualizado para o próximo nó. Se a URL estiver em um nó intermediário, o ponteiro da anterior passa a apontar para o próximo elemento. Por fim, se a URL estiver no último nó, ele também é removido e o ponteiro é ajustado para manter a integridade da lista. Como boa prática, a função `free` é utilizada para liberar a memória do nó removido. Caso existam duas URLs idênticas, elas permanecem conectadas corretamente na estrutura.

Figura 23: Função `removeUrl`

```
105 void removerUrl(struct pagina **cabeca, struct pagina **final, struct pagina **atual) {
113
114     while (ptrI != NULL) {
115         // verifica se é a cabeça que tem que ser removida
116         if (strcmp((*cabeca)->url, urlRemovida) == 0) {
117             // salva o que vai ser removido
118             struct pagina *remover = ptrI;
119
120             // a cabeça aponta para a nova cabeça
121             *cabeca = ptrI->prox;
122             // se a cabeça tiver algum valor
123             if (*cabeca != NULL) {
124                 (*cabeca)->ant = NULL;
125             }
126
127             // se a atual for a que vai ser removida, ele aponta para a cabeça
128             if (*atual == ptrI) {
129                 *atual = *cabeca;
130             }
131
132             // se a final também for removida, ele aponta para a cabeça
133             if (*final == ptrI) {
134                 *final = *cabeca;
135             }
136
137             // libera
138             free(remover);
139             remover = NULL;
140
141             printf("Apagado com sucesso!\n");
142             return;
143         }
144
145         // se a url do próximo for igual a url que vai ser removida
146         if (ptrI->prox != NULL && strcmp(ptrI->prox->url, urlRemovida) == 0) {
147             // pega o anterior e o próximo para conecta-los
148             struct pagina *anterior = ptrI, *proximo = ptrI->prox->prox;
149
150             // se a atual for a que vai ser removida
151             if (*atual == ptrI->prox) {
152                 *atual = anterior;
153             }
154
155             // se o final for a que vai ser removida
156             if (*final == ptrI->prox) {
157                 *final = anterior;
158             }
159
160             // o antecessor vai começar a apontar para o novo próximo
161             anterior->prox = proximo;
162             if (proximo != NULL) {
163                 proximo->ant = anterior;
164             }
165
166             // libera
167             free(ptrI->prox);
168             ptrI->prox = NULL;
169
170             printf("Apagado com sucesso!\n");
171         }
172     }
173 }
```

Fonte: Martins, Geovanni et al, outubro de 2025.

Conforme ilustrado na Figura 24, a função `navegar` tem como objetivo permitir a navegação entre as páginas acessadas pelo usuário. Ela utiliza os mesmos

parâmetros das funções anteriores e, inicialmente, verifica se o parâmetro atual é nulo; caso seja, a função é encerrada com o comando return. Em seguida, um menu de opções é apresentado ao usuário, oferecendo as ações de próximo, voltar e sair da navegação. O controle das escolhas é realizado por meio da estrutura switch case, pertencente à biblioteca stdio.h. Há dois casos principais: ao atingir o limite de navegação, uma mensagem é exibida; caso contrário, o ponteiro é atualizado para a página correspondente, seja a anterior ou a próxima.

Figura 24: Função navegar

```
201 void navegar(struct pagina **atual, struct pagina **cabeca, struct pagina **final) {
202     // verifica se o atual é válido
203     if (*atual == NULL) {
204         return;
205     }
206
207     int continuar = 1, opcaoUser;
208
209     while (continuar == 1) {
210         // limpa o terminal
211         system("clear");
212         printf("Você está em: %s\n", (*atual)->url);
213         printf("\n");
214         // opções para o usuário
215         printf("Escolha: \n [1] Próximo;\n [2] Voltar;\n [0] Sair da Navegação;\n");
216
217         scanf("%d", &opcaoUser);
218
219         switch (opcaoUser) {
220             case 1:
221                 // se o usuário estiver no limite ele mostrará
222                 if (*atual == *final) {
223                     printf("você chegou ao limite\n");
224                     getchar(); // limpa o caractere do \n que sobrou ao digitar
225                     getchar(); // só continua quando o usuário apertar 'enter'
226                 } else {
227                     *atual = (*atual)->prox;
228                 }
229                 break;
230
231             case 2:
232                 // se o usuário estiver no limite ele mostrará
233                 if (*atual == *cabeca) {
234                     printf("você chegou ao limite\n");
235                     getchar(); // limpa o caractere do \n que sobrou ao digitar
236                     getchar(); // só continua quando o usuário apertar 'enter'
237                 } else {
238                     *atual = (*atual)->ant;
239                 }
240                 break;
241             case 0:
242                 continuar = 0;
243                 break;
244         }
245     }
246
247 }
```


Fonte: Martins, Geovanni et al, outubro de 2025.

Conforme ilustrado na Figura 25, a função `exibirHistorico` tem como objetivo percorrer todo o histórico de páginas inseridas pelo usuário por meio de um ponteiro iterador denominado `*ptrI`. Esse ponteiro inicia na cabeça da lista e, enquanto o endereço apontado for diferente de `NULL`, exibe na tela todas as páginas armazenadas no histórico de navegação.

Figura 25: Função `exibirHistorico`

```
180 void exibirHistorico(struct pagina **cabeca) {
181     // é um ponteiro iterador para percorrer o histórico
182     struct pagina *ptrI = *cabeca;
183
184     while (ptrI != NULL) {
185         printf("%s ", ptrI->url);
186         ptrI = ptrI->prox;
187     }
188     printf("\n");
189 }
```

Fonte: Martins, Geovanni et al, outubro de 2025.

3.3 Aplicação prática de Lista Circular: O objetivo desta prática foi aplicar os conhecimentos sobre a estrutura de lista circular, explorando sua lógica de encadeamento contínuo e o uso de ponteiros para percorrer os elementos. A lista de tarefas foi escolhida como exemplo por representar de forma clara o funcionamento desse tipo de estrutura, em que os elementos seguem uma ordem sequencial e, ao atingir o último, o encadeamento retorna ao primeiro, formando um ciclo. Essa característica reflete situações comuns no cotidiano, como agendas de compromissos ou sistemas que necessitam de repetição contínua das operações.

Conforme ilustrado na Figura 26, a função `criar` possui como parâmetro a cabeça da lista, que é passada por referência para permitir que a função altere seu valor caso a lista esteja vazia. A função `malloc` é utilizada para alocar dinamicamente memória para um novo nó. Quando a cabeça aponta para `NULL`, significa que a lista ainda não possui elementos; nesse caso, o novo nó é inserido e, por se tratar de uma lista circular, seu ponteiro `prox` é definido para apontar para ele mesmo, garantindo o encadeamento contínuo da estrutura.

Figura 26: Função criar

```
72 void criar(tarefa **cabeca, char *nome, char *esforco)
73 {
74     tarefa *novo = malloc(sizeof(tarefa));
75     strcpy(novo->nome, nome);
76     strcpy(novo->esforco, esforco);
77
78     if(*cabeca == NULL)
79     {
80         *cabeca = novo;
81         novo->prox = *cabeca;
82     }
83
84     else
85     {
86         tarefa *aux = *cabeca;
87         while (aux->prox != *cabeca)
88         {
89             aux = aux->prox;
90         }
91         aux->prox = novo;
92         novo->prox = *cabeca;
93     }
94 }
95 }
```

Fonte: Martins, Giovanni et al, outubro de 2025.

Conforme ilustrado na Figura 27, a função imprimir inicia com um ponteiro apontando para a cabeça da lista, que é utilizada para acessar seus elementos. Para percorrer a lista, é criado um ponteiro auxiliar que começa na cabeça e avança pelos nós sequencialmente. Caso o ponteiro auxiliar seja NULL, significa que a lista está vazia, e uma mensagem informativa é exibida. Um laço é utilizado para percorrer todos os elementos, sendo encerrado apenas quando o ponteiro retorna à cabeça da lista, garantindo que todos os nós da estrutura circular sejam visitados.

Figura 27: Função imprimir

```
96 void imprimir(tarefa **cabeca)
97 {
98     tarefa *aux = *cabeca;
99     if(aux == NULL)
100     {
101         printf("lista vazia");
102         return;
103     }
104     do
105     {
106         printf("%s - %s\n", aux->nome, aux->esforco);
107         aux = aux->prox;
108     } while (aux != *cabeca);
109 }
110
111 }
```

Fonte: Martins, Giovanni et al, outubro de 2025.

Conforme ilustrado na figura 28 função remover tem como objetivo excluir uma tarefa da lista circular com base no seu nome, garantindo que a estrutura permaneça consistente após a remoção. Inicialmente, verifica-se se a lista está vazia, exibindo uma mensagem caso não existam elementos. Em seguida, a função percorre a lista com um ponteiro auxiliar, mantendo também um ponteiro para o nó anterior, necessário para ajustar os encadeamentos. São tratados os diferentes casos: remoção de um único nó, remoção da cabeça e remoção de nós intermediários ou finais. Em todos os casos, a memória do nó removido é liberada, e os ponteiros são atualizados de forma a manter a circularidade da lista.

Figura 28: função remover

```
112 void remover(tarefa **cabeca, char *nome)
113 {
114     if (*cabeca == NULL)
115     {
116         printf("lista vazia");
117         return;
118     }
119     tarefa *aux = *cabeca;
120     tarefa *anterior = NULL;
121     char nomeRecebido[50];
122     strcpy(nomeRecebido, nome);
123
124     do {
125         if (strcmp(aux->nome, nomeRecebido) == 0) {
126             if (aux == aux->prox) {
127                 free(aux);
128                 *cabeca = NULL;
129                 return;
130             }
131
132             if (aux == *cabeca) {
133                 tarefa *ultimo = *cabeca;
134                 while (ultimo->prox != *cabeca)
135                     ultimo = ultimo->prox;
136                 *cabeca = aux->prox;
137                 ultimo->prox = *cabeca;
138                 free(aux);
139                 return;
140             }
141             anterior->prox = aux->prox;
142             free(aux);
143             return;
144         }
145
146         anterior = aux;
147         aux = aux->prox;
148     } while (aux != *cabeca);
149 }
```

Fonte: Martins, Geovanni et al, outubro de 2025.

Conforme ilustrado na figura 29, a função buscar tem como objetivo localizar tarefas na lista circular com base no nome fornecido. Para isso, é criado um ponteiro auxiliar que percorre todos os elementos da lista, iniciando pela cabeça. A cada nó, é realizada uma comparação entre o nome armazenado e o nome desejado; caso

haja correspondência, o nome da tarefa e seu esforço são exibidos na tela. O laço de percorrimento se mantém ativo até que o ponteiro retorne à cabeça, garantindo que todos os elementos da lista sejam verificados e que todas as tarefas correspondentes sejam apresentadas.

Figura 29: Função buscar

```
150 void buscar(tarefa **cabeca, char *nome)
151 {
152     tarefa *aux = *cabeca;
153
154     do
155     {
156         if(strcmp(aux->nome, nome) == 0)
157         {
158             printf("%s - %s\n", aux->nome, aux->esforco);
159         }
160         aux = aux->prox;
161     }while(aux != *cabeca);
162 }
163
```

Fonte: Martins, Geovanni et al, outubro de 2025.

3.4 Aplicação Prática de lista duplamente circular: O objetivo desta prática foi aplicar os conhecimentos sobre a estrutura de lista duplamente circular, explorando sua lógica de encadeamento contínuo e o uso de ponteiros para percorrer os elementos. Essa estrutura foi escolhida por representar bem o funcionamento de uma lista circular, pois permite avançar e retroceder entre os elementos de forma contínua, semelhante a uma playlist de músicas em que a reprodução segue em ciclo, interrompendo apenas quando o usuário decide parar.

Para melhor compreensão, serão apresentados a seguir exemplos das principais operações realizadas nessa estrutura: **criação, exclusão, busca e exibição.**

Conforme ilustrado o trecho do código na figura 30, uma música (nó) é adicionada à playlist, implementada como uma lista duplamente encadeada circular. A função recebe como parâmetros a cabeça da lista (primeiro nó), o nome da música a ser adicionada, o nome do cantor e o ano de lançamento.

Inicialmente, é criado um novo nó por meio da função `malloc()`, responsável pela alocação dinâmica de memória com o tamanho correspondente à estrutura `song`. Em seguida, os campos do novo nó recebem os valores passados como parâmetros: `name`, `singer` e `releaseYear`. O próximo passo consiste em verificar se a lista está vazia. Caso esteja, o novo nó criado passa a ser a nova cabeça da lista. Para garantir que a estrutura mantenha o comportamento de uma lista duplamente circular, as linhas 130 e 131 do código são fundamentais: nelas, o ponteiro `next` aponta para a própria cabeça da lista, enquanto o ponteiro `back` aponta para o próprio nó, fechando o ciclo circular. Se a lista já possuir elementos, é criado um ponteiro auxiliar do tipo `song`, que recebe o valor de `head`. Nesse caso, é necessário percorrer a lista até o último elemento, o que é feito por meio de um laço de repetição que continua enquanto o ponteiro `next` da variável auxiliar for diferente de `head`, isso ocorre porque, em uma lista circular, o último elemento sempre aponta de volta para o primeiro. Após encontrar o último elemento, o código define que o ponteiro `next` desse nó passará a apontar para o novo nó criado. O novo nó, por sua vez, terá seu ponteiro `back` apontando para o nó anterior (variável auxiliar) e seu ponteiro `next` apontando para a cabeça da lista. Por fim, o ponteiro `back` da cabeça (`head->back`) é atualizado para apontar para o novo nó, mantendo assim a circularidade e o encadeamento duplo da estrutura.

Figura 30: criar lista

```
116 //-----
117
118 void addToPlaylist(song **head, char *name, char *singer, int releaseYear)
119 {
120     song *new = malloc(sizeof(song));
121     strcpy(new->name, name);
122     strcpy(new->singer, singer);
123     new->releaseYear = releaseYear;
124
125     if(*head == NULL)
126     {
127         *head = new;
128         new->next = *head;
129         new->back = new;
130     }
131     else
132     {
133         song *aux = *head;
134         while(aux->next != *head)
135         {
136             aux = aux->next;
137         }
138         aux->next = new;
139         new->next = *head;
140         new->back = aux;
141         (*head)->back = new;
142     }
143 }
144
145
146
```

Fonte: Martins, Geovanni et al, outubro de 2025.

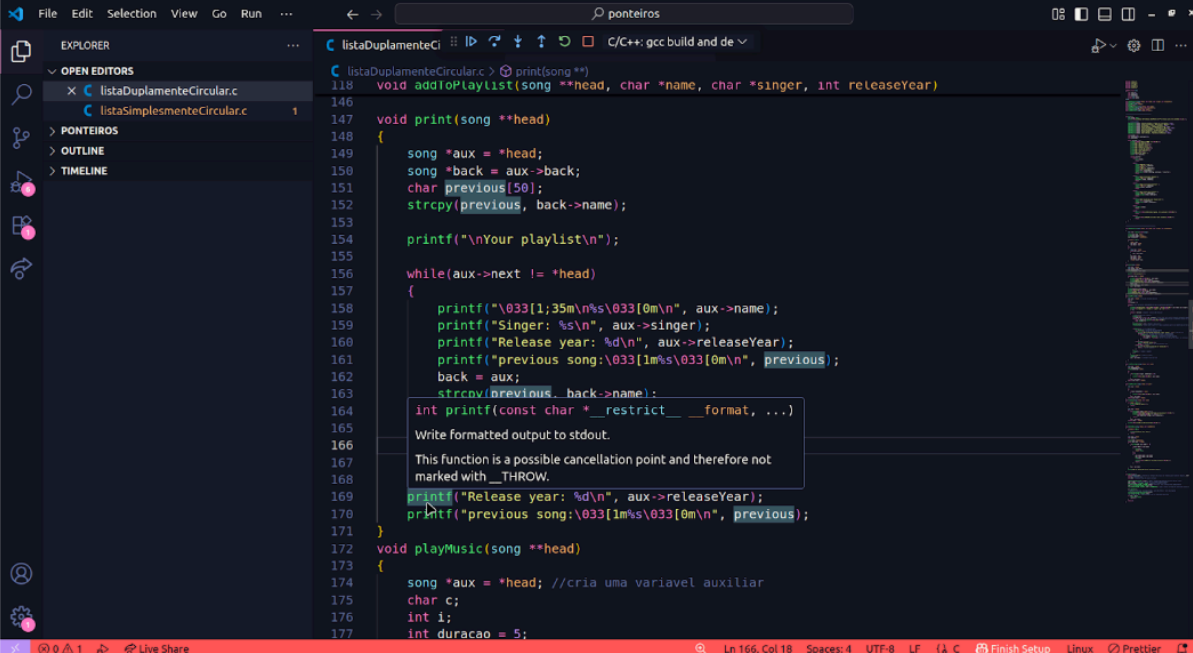
Como ilustrado na imagem 31, foram implementados três métodos de busca distintos. O primeiro realiza a pesquisa pelo nome da música e retorna apenas um resultado. Os outros dois funcionam como filtros, permitindo a busca por cantor ou por ano de lançamento. O funcionamento dessas funções é simples: cada uma recebe como parâmetro o início da lista e a chave de busca (nome, cantor ou ano de lançamento). Em seguida, a função percorre toda a lista até encontrar os elementos que correspondem à chave informada. Nos casos de busca por cantor e por ano, todas as ocorrências encontradas são exibidas na tela.

3- remover

Para remover foi usada a função deleteMusic(). Ela recebe como parâmetros o ponteiro para a cabeça da lista e o nome da música que se deseja excluir.

Inicialmente, a função verifica se a lista está vazia. Caso esteja, é exibida uma mensagem informando essa condição, e a execução é encerrada. Em seguida, é criado um ponteiro auxiliar do tipo song, que recebe a cabeça da lista, e uma variável name que armazena o nome da música a ser removida. O próximo passo é percorrer a lista utilizando um laço que continua enquanto o ponteiro next do nó atual for diferente da cabeça, garantindo que toda a estrutura seja percorrida. Durante o percurso, cada nó é comparado com o nome informado. Se a música for encontrada, a função verifica se há apenas um nó na lista — ou seja, se o ponteiro back e o next apontam para o próprio nó. Nesse caso, o nó é liberado da memória e a cabeça da lista é definida como NULL, indicando que a playlist ficou vazia. Caso existam outros nós, o encadeamento é ajustado: o ponteiro next do nó anterior passa a apontar para o próximo nó, e o ponteiro back do próximo nó passa a apontar para o nó anterior, removendo assim o elemento da sequência. Após o ajuste, o nó é desalocado com a função free(), liberando a memória ocupada. Por fim, é exibida uma mensagem informando que a música foi removida com sucesso. Se o nome fornecido não for encontrado em nenhum nó, a função exibe uma mensagem indicando que não há música com o título especificado.

Figura 31: Métodos de busca distintos



```

118 void addIoPlaylist(song **head, char *name, char *singer, int releaseYear)
146
147 void print(song **head)
148 {
149     song *aux = *head;
150     song *back = aux->back;
151     char previous[50];
152     strcpy(previous, back->name);
153
154     printf("\nYour playlist\n");
155
156     while(aux->next != *head)
157     {
158         printf("\033[1;35m\n%s\033[0m\n", aux->name);
159         printf("Singer: %s\n", aux->singer);
160         printf("Release year: %d\n", aux->releaseYear);
161         printf("previous song:\033[1m%s\033[0m\n", previous);
162         back = aux;
163         strncpy(previous, back->name);
164         int printf(const char *__restrict __format, ...)
165             Write formatted output to stdout.
166             This function is a possible cancellation point and therefore not
167             marked with __THROW.
168         printf("Release year: %d\n", aux->releaseYear);
169         printf("previous song:\033[1m%s\033[0m\n", previous);
170     }
171
172 void playMusic(song **head)
173 {
174     song *aux = *head; //cria uma variavel auxiliar
175     char c;
176     int i;
177     int duracao = 5;

```

Fonte: Martins, Giovanni et al, outubro de 2025.

A função responsável por exibir os dados na tela é a `print()`, que recebe como parâmetro apenas o ponteiro para a cabeça da lista. Assim como nas outras funções, foi criada uma variável auxiliar para percorrer toda a lista. O laço segue o mesmo padrão das demais operações, realizando a navegação até que o ponteiro volte ao início, já que se trata de uma lista duplamente circular. Dentro do loop, são impressas todas as informações da música, como nome, cantor e ano de lançamento, além do nome da música anterior, que é acessado por meio do ponteiro `back`. Também foram usadas cores no terminal para deixar a exibição mais organizada e facilitar a identificação das informações.

Conforme ilustrado na figura 32, para mostrar como funcionaria um aplicativo de música, criamos a função `playMusic()`. Ela recebe como parâmetro a cabeça da lista e, dentro da função, usamos uma variável auxiliar do tipo `song` (`aux`) para percorrer as músicas da playlist.

Figura 32: Função `playMusic`

```
172 void playMusic(song **head)
173 {
174     song *aux = *head; //cria uma variavel auxiliar
175     char c;
176     int i;
177     int duracao = 5;
178
179     while(1) //toca as musicas eternamente até que o usuario peça para parar
180     {
181         printf("\nTocando agora: \033[1;35m%s\033[0m - \033[1m%s\033[0m\n", aux->name, aux->singer);
182         printf("\np = pause | r = retome | s = pular | q = sair\n\n");
183         i = 0;
184
185         while(i < duracao) //enquanto a musica não terminar
186         {
187             // mostrar barra de progresso
188             printf("\r");
189             int progresso = (i+1) * 10 / duracao; //essa linha aqui calcula de quanto em quantos quadradinhos deve progredir
190             for(int j = 0; j < 10; j++) { //imprime os quadradinhos até que o progresso concluir os 10 espaços
191                 if(j < progresso) printf("\033[1;35m█\033[0m");
192                 else printf(" ");
193             }
194             printf("] %ds ", i+1); //Fecha a barra com ]
195             fflush(stdout); //essa linha é para forçar que o quadrado apareça no momento em que for preenchido
196         }
```

Fonte: Martins, Geovanni et al, outubro de 2025.

A função roda dentro de um laço infinito, garantindo que as músicas sejam tocadas continuamente até que o usuário decida parar. Dentro desse laço, há um loop interno (`while(i < duracao)`) que simula a execução de cada música, segundo a variável `duracao`, que define o tempo de reprodução em segundos. A ideia é que as músicas não apareçam todas de uma vez na tela, mas que dê a sensação de que cada uma está realmente sendo tocada.

Durante a execução de cada música, é exibida uma barra de progresso, que se atualiza a cada segundo. O progresso é calculado com a linha:

```
int progresso = (i+1) * 10 / duracao;
```

e os quadradinhos preenchidos são impressos com:

```
for(int j = 0; j < 10; j++) {  
    if(j < progresso) printf("\033[1;35m█\033[0m");  
    else printf(" ");  
}
```

O `fflush(stdout)` garante que a barra seja atualizada imediatamente no terminal.

Para permitir a interação do usuário sem travar o programa, usamos a função `kbhit()`, que detecta se alguma tecla foi pressionada. Dependendo da tecla digitada (`p`, `r`, `s` ou `q`), o programa pode pausar, retomar, pular, voltar ou encerrar a reprodução. Por exemplo, ao pausar, o programa entra em um laço que só termina quando o usuário pressiona `r`. Ao final de cada música, o terminal é limpo com `system("clear")`, e o ponteiro `aux` avança para o próximo nó da lista (`aux = aux->next`), garantindo a reprodução contínua e circular das músicas.

Figura 33: Função de reprodução de músicas

```
196
197
198     if(kbhit()) { //se foi digitado algo
199         c = getchar(); //armazena oq foi digitado
200         if(c == 'p') { //se oq foi digitado foi p
201             printf("\nMúsica pausada! Pressione r para retomar...\n"); //imprime isso
202             while(1) { //cria um loop eterno até que r seja digitado
203                 if(kbhit()) { //mesma coisa, verifica se user digitou
204                     char resume = getchar(); //armazena o caractr digitado
205                     if(resume == 'r') break; //se for r ele vai sair do loop e vai continuar o progresso
206                 }
207             }
208         } else if(c == 's') {
209             printf("\nPróxima música\n");
210             break; // sai do loop e vai pra próxima música
211         } else if(c == 'q') {
212             printf("\nA playlist parou de ser reproduzida\n");
213             return; // sai da playlist
214         }
215         else if(c == 'b'){
216             printf("\nMúsica anterior\n");
217             voltar = 1;
218             break;
219         }
220     }
221     i++;
222     sleep(1);
223 }
224
225 sleep(0.1);
226 system("clear");
227
228 if (voltar == 1)
229 {
230     aux = aux->back;
231     voltar = 0;
232 }
233
234 else
235     aux = aux->next;
236
237 }
238
```

Fonte: Martins, Giovanni et al, outubro de 2025.

4. Conclusão

Ao analisar todo o conteúdo apresentado, foi possível compreender e identificar os principais tipos de estruturas de listas: simples, duplamente encadeada, circular e duplamente circular. Cada uma delas foi abordada tanto em sua parte teórica quanto por meio de exemplos práticos e anagramas ilustrativos, permitindo uma melhor visualização de seu funcionamento. Além disso, a comparação entre as diferentes estruturas possibilitou entender suas aplicações, vantagens e particularidades, evidenciando a importância dessas listas na organização e manipulação eficiente de dados em programação.

Bibliografia

FURTADO, Daniel A. *Listas Duplamente Ligadas Circulares*. 2025. 10 p. Universidade Federal de Uberlândia. Disponível em: <https://furtado.prof.ufu.br/site/teaching/ED1/ED1-06-TAD-Listas-Dupla-Circular.pdf>. Acesso em: 17 out. 2025.

GOLD, R. *Lista circular*. São Paulo: Instituto de Matemática e Estatística da Universidade de São Paulo, 2001. Disponível em: <https://www.ime.usp.br/~gold/cursos/2001/mac2301/lista-circular.pdf>. Acesso em: 17 out. 2025. Imagem extraída do documento.

GRÉGIO, André. *Listas Duplamente Encadeadas*. Disponível em: <https://www.inf.ufpr.br/gregio/CI1001/ListaDuplamenteEncadeada.pdf>. Acesso em: 15 out. 2025.

KNUTH, Donald E. *The Art of Computer Programming: Volume 1 – Fundamental Algorithms*. 3. ed. Reading, MA: Addison-Wesley, 1997. acesso em 14 de outubro de 2025.

KNUTH, Donald E. *The Art of Computer Programming: Volume 1 – Fundamental Algorithms*. 3. ed. Reading: Addison-Wesley, 1999. Acesso em 17 de outubro de 2025.

STACK OVERFLOW EM PORTUGUÊS. Qual o funcionamento de uma lista encadeada em C? Disponível em: <https://pt.stackoverflow.com/questions/222303/qual-o-funcionamento-de-uma-lista-encadeada-em-c>. Acesso em 15 outubro de 2025.