Algoritmos e Estrutura de Dados

Roselene Henrique Pereira Costa
Alfredo Johnson Rodríguez
Analeia Gomes
João Paulo Resende Rosa
Leandro Ferreira Monteiro
Simone Elza dos Santos Teodoro
Valeria Aparecida Mattar Vilas Boas

2023

Editora Científica

Dados Internacionais de Catalogação na Publicação (CIP)
Selma Alice Ferreira Ellwein – CRB 9/1558

C87a Costa, Roselene Henrique Pereira, et al.

Algoritmos e estruturas de dados. / Roselene Henrique Pereira Costa, Alfredo Johnson Rodríguez, Analéia Gomes, João Paulo Resende Rosa, Leandro Ferreira Monteiro, Simone Elza dos Santos Teodoro, Valeria Aparecida Mattar Vilas Boas. – Londrina: Editora Científica, 2023.

ISBN 978-65-00-68044-7

1. Listas Ligadas. 2. Pilhas. 3. Fitas. 4. Ensino Superior. I. Título. II. Autores.

CDD 660

SUMÁRIO

UNIDADE 1 – LISTAS LIGADAS

SEÇÃO 1 – DEFINIÇÃO E ELEMENTOS DE LISTAS LIGADAS

Estruturas de dados são formas de organizar e armazenar dados na memória do computador de maneira eficiente e acessível. Na linguagem C, as estruturas de dados são integradas por meio de estruturas (structs) e arrays.

Uma **estrutura** é uma variável composta que pode conter vários tipos de dados diferentes, incluindo variáveis primitivas, outras estruturas e ponteiros. Ela é definida usando a palavra-chave "struct" seguida do nome da estrutura e dos tipos de dados que ela contém, como no exemplo abaixo:

```
struct pessoa {
  char nome[50];
  int idade;
  float altura;
};
```

Nesse exemplo, a estrutura "pessoa" é composta por uma variável do tipo char para o nome, uma variável do tipo int para a idade e uma variável do tipo float para a altura.

Arrays, por outro lado, são coleções de elementos do mesmo tipo de dado, organizados em uma sequência na memória do computador. Eles são definidos usando a sintaxe abaixo:

```
<tipo> <nome_do_array>[<tamanho_do_array>];
```

Por exemplo, um array de inteiros com tamanho 10 pode ser definido assim:

```
int numeros[10];
```

As **estruturas de dados em C** são fundamentais para a criação de programas eficientes e organizados, permitindo que os dados sejam armazenados e manipulados de maneira mais fácil e intuitiva. Além disso, elas são uma ferramenta poderosa para a resolução de problemas computacionais complexos.

Conforme Rovai (2018) as estruturas de dados são formas de organização e distribuição de dados para tornar mais eficientes a busca e manipulação dos dados por algoritmos.

As estruturas de dados podem ser:

- **lineares** (ex. arrays) ou **não lineares** (ex. grafos);

- **homogêneas** (todos os dados que compõe a estrutura são do mesmo tipo) ou **heterogêneas** (podem conter dados de vários tipos);
- **estáticas** (têm tamanho/capacidade de memória fixa) ou **dinâmicas** (podem expandir).

Cada **estrutura de dados** tem um conjunto de métodos próprios para realizar operações como:

- Inserir ou excluir elementos;
- Buscar e localizar elementos;
- Ordenar (classificar) elementos de acordo com alguma ordem especificada.

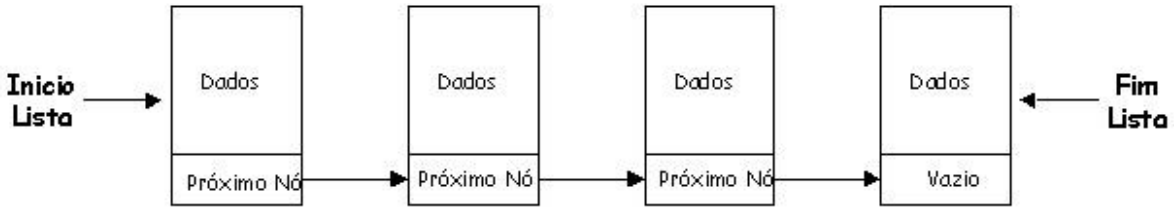
Segundo Celes et al. (2004), uma estrutura de dados que permite o armazenamento dinâmico de dados é **a lista encadeada**, ou também chamada de **lista ligada**, que pode ser integrada utilizando diversas outras estruturas de dados.

Uma **lista ligada** (ou encadeada) é uma estrutura de dados que consiste em uma coleção L:[a1, a2, ..., an], em que n > 0.

Na Figura 1 podemos visualizar o modelo de uma lista ligada, a estrutura da lista consiste em uma sequência de elementos encadeados, definida como nó de lista. O nó de lista é composto de duas informações:

- a informação armazenada (dados) e um ponteiro que indica o próximo elemento da lista (próximo nó).

Figura 1 – Lista Ligada



Fonte: os autores.

Segundo Rovai (2018), sua propriedade estrutural é baseada apenas na posição relativa dos elementos, que são dispostos linearmente. Além disso, de acordo com Silva (2007), ela é conhecida também como lista encadeada e é composta por um conjunto de dados programáveis em uma sequência de nós, onde a relação de

sucessão desses elementos é determinada por um ponteiro que indica a posição do próximo elemento. A lista pode ser ordenada ou não.

De acordo com Silva (2007), a terapia de uma lista encadeada envolve a definição de um ponto inicial ou ponteiro para o começo da lista. A partir disso, é possível realizar diversas operações na lista, como inserir, retirar e buscar de elementos.

Os procedimentos básicos de manipulação de uma lista, de acordo com Silva (2007), incluem:

- Criação ou definição da estrutura da lista.
- Inicialização da lista.
- Inserção de elementos com base em um endereço de referência.
- Alocação de um endereço de nó para inserir na lista.
- Remoção do nó com base em um endereço de referência.
- Deslocamento do nó removido da lista.

Conforme Rovai (2018), uma das vantagens das listas ligadas é que os elementos não precisam ser armazenados sequencialmente na memória, ao contrário dos vetores, que podem ser do tipo de organização. Na lista ligada, cada elemento pode ser alocado em diferentes regiões de memória, permitindo uma alocação dinâmica e mais flexível de espaço.

De acordo com Rovai (2018), uma lista que não possui nós é definida como vazia ou nula, e o valor do ponteiro para o próximo nó é considerado um ponteiro nulo.

Os elementos de uma lista não são armazenados em posições sequenciais de memória, mas sim de forma dinâmica em diferentes regiões de memória. Além disso, a lista permite percorrer os elementos em qualquer direção, graças à relação de sucessão entre eles determinados pelos ponteiros. Ao definir a estrutura de uma lista, é preciso especificar os campos que compõem cada nó, como o campo de informação e o campo de ponteiro para o próximo nó da lista. Esses campos podem ser de diferentes tipos de dados, como int, char ou float, dependendo das necessidades da aplicação, conforme Rovai (2018).

A Figura 2 ilustra um modelo de implementação de uma lista:

Figura 2 – Implementação lista

```
struct lista {
int info;
struct lista* prox;
};
typedef struct lista Lista;
```

Fonte: os autores.

A Figura 3 demonstra um exemplo de declaração para criar uma lista em C:

Figura 3 – Declaração para ciar uma lista

```
struct lista {
int info;
struct lista* prox;
};
typedef struct lista Lista;
```

Fonte: os autores.

É necessário inicializar a lista para ser utilizada após sua criação. Para isso, basta criar uma função em que inicializamos a lista como nula, a Figura 4 mostra uma das possíveis formas de inicialização.

Figura 4 – Inicializar a lista

```
/* Função para inicialização: retorna uma lista vazia */
    Lista* inicializa (void)
    {
        return NULL;
    }
```

Fonte: os autores.

Segundo Veloso (1996), em uma lista ligada, os elementos de conexão são os ponteiros. Um ponteiro é um tipo de variável que armazena um endereço de memória e não o conteúdo da posição de memória.

De acordo com Rovai (2018), a utilização de ponteiros é necessária em situações em que é preciso conhecer o endereço em que a variável está armazenada.

Para declarar um ponteiro, é possível utilizar a mesma palavra reservada do tipo da variável seguida do caractere * (asterisco). Veja o exemplo na Figura 5.

Figura 5 – Declaração de ponteiros

```
int *ptr; /* sendo um ponteiro do tipo inteiro*/
float *ptr; /* sendo um ponteiro do tipo ponto flutuante*/
char *ptr; /* sendo um ponteiro do tipo caracteres*/
```

Fonte: os autores.

A Figura 6 ilustra uma estrutura declarada, e é possível ver a utilização de ponteiro.

Figura 6 – Declaração lista com a utilização de ponteiro

```
typedef struct lista {
char *nome; //Declaração de um ponteiro do tipo char
int telefone;
struct lista *proximo;
} Dados;
```

Fonte: os autores.

Segundo Tanenbaum et al. (2007), um ponteiro é como qualquer outro tipo de variável e pode ser utilizado para armazenamento e manipulação de valores.

De acordo com Silva (2007) para obtermos o endereço de memória reservado a uma variável, utilizamos o operador & seguido do nome da variável. Já o operador * (asterisco), utilizado com a variável do tipo ponteiro, acessa o conteúdo armazenado no endereço de memória apontado pelo ponteiro, que pode ser visto na Figura 7.

Figura 7 – Ponteiro – operadores & e *

```
int x = 10; // variável
int *p; // ponteiro
p = &x; // ponteiro p aponta para o endereço da variável x
```

Fonte: os autores.

Conforme Rovai (2018), em listas, além do uso de ponteiros, utilizamos também alocações dinâmicas de memória, que são porções de memória reservadas para utilização das listas. Para compreendermos como funciona um ponteiro em uma lista, precisamos entender a função malloc(), Memory Allocation ou Alocação de Memória. Essa função é responsável pela reserva de espaços na memória principal. Sua finalidade é alocar uma faixa de bytes consecutivos na memória do computador e retornar o endereço dessa faixa ao sistema. O trecho de código a seguir apresenta um exemplo de utilização da função malloc() e do ponteiro:

```
int *ptr;
ptr = (int*) malloc(sizeof(int));
```

Nesse exemplo, o ponteiro "ptr" foi declarado como um ponteiro do tipo inteiro e, em seguida, utilizou-se a função malloc() para alocar espaço na memória para armazenar um valor inteiro. A função sizeof(int) retorna o tamanho em bytes do tipo inteiro e, por isso, é utilizada como argumento da função malloc(). O ponteiro "ptr" armazena o endereço de memória da área alocada pela função malloc().

Na Figura 8 temos um exemplo de código em C, com a utilização das funções malloc( ) e sizeof.

Figura 8 – Função malloc() e sizeof

```c
#include <stdio.h>
#include <stdlib.h>

int main(){
    int *p;
    p=(int *) malloc(sizeof(int));

    if(!p){
        printf("Erro de memoria insuficiente");
    }else{
        printf("Memoria alocada com sucesso");
    }
    return 0;
}
```

Fonte: Rovai (2018).

UNIDADE 1 – LISTAS LIGADAS

SEÇÃO 2 – OPERAÇÕES COM LISTAS LIGADAS

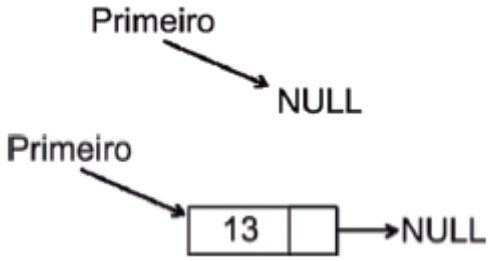
Conforme Tanembaum et. al. (2007) uma lista é uma estrutura de dados dinâmicos que permite que o número de nós varie consideravelmente conforme elementos são inseridos e removidos. Em contraste, uma natureza estática de um vetor implica que seu tamanho era constante.

De acordo com Celes et al. (2004), a inserção de um elemento em uma lista conectada requer a alocação dinâmica de espaço de memória para armazenar o novo elemento e conectá-lo à lista existente. Existem três situações distintas em que um elemento pode ser inserido em uma lista ligada.

As três situações em que um elemento pode ser inserido em uma lista conectada são:

1. **Inserção no início da lista**: O novo elemento é adicionado no começo da lista e passa a ser o novo primeiro nó, apontado para o antigo primeiro nó, ilustrado na Figura 9.

Figura 9 – Inserindo um novo elemento no início da lista vazia



Fonte: Rovai (2018).

A Figura 10 é um trecho de implementação do código para inserir um novo elemento no início da lista:

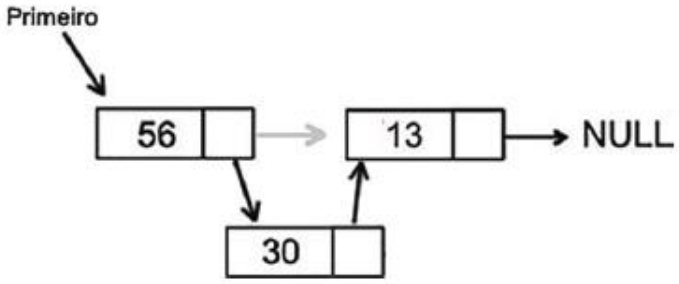
Figura 10 – Inserção no início da lista

```
Lista* inserir (Lista* l, int i) {
    Lista* novo = (Lista*) malloc(sizeof(Lista));
    novo -> info = i;
    novo -> prox = l;
return novo;
}
```

Fonte: os autores.

2. **Inserção no meio da lista**: O novo elemento é adicionado entre dois nós já existentes, apontado para o nó anterior e para o nó seguinte, ilustrado na Figura 11.

Figura 11 – Inserindo um novo elemento no meio da lista



Fonte: Rovai (2018).

A Figura 12 é um trecho de implementação do código para inserir um novo elemento no meio da lista:

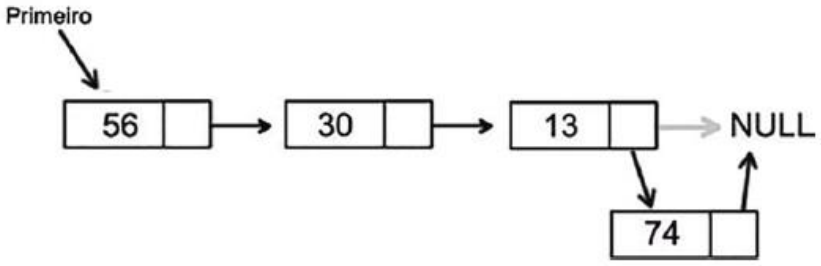
Figura 12 – Inserção no meio da lista

```
Lista* inserirPosicao(Lista* l, int pos, int v){
    int cont = 1;
    Lista *p = l;
    Lista* novo = (Lista*)malloc(sizeof(Lista));
    while (cont != pos){
        p = p -> prox;
        cont++;
    }
    novo -> info = v;
    novo -> prox = p -> prox;
    p -> prox = novo;
    return l;
}
```

Fonte: os autores.

3. **Inserção no final da lista**: O novo elemento é adicionado ao final da lista, ou seja, depois do último nó existente, e passa a ser o novo último nó, apontado para o nó anterior, ilustrado na Figura 13.

Figura 13 – Inserindo um novo elemento no final da lista



Fonte: Rovai (2018).

A Figura 14 é um trecho de implementação do código para inserir um novo elemento no final da lista:

Figura 14 – Inserção no final da lista

```
Lista* inserirFim(Lista* l, int v){
    int cont=0;
    Lista *p = l;
    Lista* novo = (Lista*)malloc(sizeof(Lista));
    while (p -> prox != NULL){
        p = p -> prox;
        cont++;
    }
    novo -> info = v;
    novo -> prox = p -> prox;
    p -> prox = novo;
    return l;
}
```

Fonte: dados da pesquisa.

Em qualquer uma dessas situações, é essencial que o valor do ponteiro da lista seja atualizado para apontar para o novo elemento adicionado.

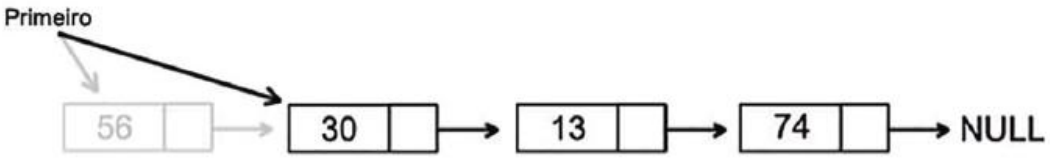
Para remover elementos de uma lista ligada, é necessário seguir os seguintes passos:

- Encontre o elemento a ser removido: percorra a lista a partir do primeiro elemento, encontre o elemento a ser removido.
- Atualizar os ponteiros: após encontrar o elemento a ser removido, atualizar os ponteiros do elemento anterior e posterior ao elemento removido para que apontem um para o outro, ignorando o elemento a ser removido.
- Desalocar o espaço de memória: liberar o espaço de memória ocupado pelo elemento removido.

Vale ressaltar que é importante verificar se a lista não está vazia antes de tentar remover elementos, e tratar caso em que o elemento a ser removido é o primeiro elemento da lista.

Conforme Rovai (2018), quando o primeiro elemento da lista é o elemento a ser removido, é necessário atualizar o valor da lista com o ponteiro para o segundo elemento, liberando o espaço alocado do elemento retirado, como podemos observar na Figura 15.

Figura 15 – Remover o primeiro elemento da lista

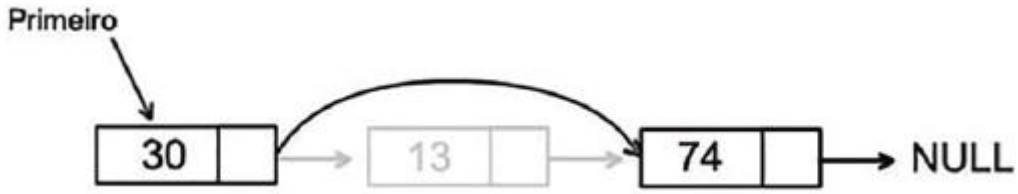


Fonte: Rovai (2018).

O processo envolve a atualização do ponteiro do primeiro nó para apontar para o segundo nó da lista e, em seguida, liberar o espaço de memória alocado para o primeiro nó removido utilizando a função free( ).

Conforme Rovai (2018), se a função de remoção de elementos de uma lista conectada precisar remover um elemento que esteja no meio da lista, o elemento anterior ao que será removido deve ter o seu ponteiro atualizado para o elemento seguinte ao que será removido. Depois disso, a alocação de memória do elemento removido deve ser liberada, como podemos observar na Figura 16.

Figura 16 – Remover elemento do meio da lista



Fonte: Rovai (2018).

De acordo com Rovai (2018), é possível criar uma função que execute a remoção de elementos da lista, tanto do início quanto de qualquer outra posição. Para isso, podemos utilizar um único trecho de código, que verificar se o elemento a ser removido é o primeiro da lista ou se está em outra posição. Em seguida, atualizamos os ponteiros da lista e liberamos a alocação de memória do elemento removido. A Figura 17 ilustra o código para remoção de elemento da lista.

Figura 17 – Remover elemento da lista

```c
//remover elemento da lista
Lista* remove (Lista* l, int v) {
    Lista* anterior = NULL;
    Lista* p = l;
    while (p != NULL && p -> info != v) {
        anterior = p;
        p = p -> prox;
    }
    if (p == NULL )
        return l;
    if (anterior == NULL) {
        l = p -> prox;
    } else {
        anterior -> prox = p -> prox;
    }
    return l;
}
```

Fonte: os autores.

De acordo com Rovai (2018), para verificar todos os elementos de uma lista conectada em determinado momento do sistema, é necessário percorrer toda a lista. Podemos imprimir todos os elementos da lista conectada utilizando o seguinte trecho de código da Figura 18.

Figura 18 – Imprimir a lista

```c
//função que percorrerá toda a lista e de impressão em tela
void imprimir (Lista* l) {
    Lista* p;
    printf("Elementos:\n");
    for (p = l; p != NULL; p = p -> prox) {
        printf("%d -> ", p -> info);
    }
}
```

Fonte: os autores.

Outra função bastante útil é verificar se um determinado elemento está presente na lista ou não, conforme explicado por Celes et al. (2004). Essa função pode ser criada para receber a informação do elemento que se deseja buscar e, caso o valor seja encontrado, a função retorna o ponteiro do nó da lista que representa o

elemento ou sua posição na lista. No entanto, em nosso exemplo, a função simplesmente informa se o elemento foi encontrado ou não. Podemos utilizar o seguinte trecho de código da Figura 19, para implementar a função de busca.

Figura 19 – Buscar elemento na lista

```
Lista* buscar(Lista* l, int v){
    Lista* p;
    for (p = l; p != NULL; p = p -> prox) {
        if (p -> info == v)
        return p;
    }
    return NULL;
}
```

Fonte: os autores.

UNIDADE 1 – LISTAS LIGADAS

SEÇÃO 3 – LISTAS DUPLAMENTE LIGADAS

De acordo com Celes et al. (2004), em estruturas de listas ligadas simples, não é possível abranger seus elementos de forma inversa, ou seja, do final até o início. Além disso, remover um elemento dessa lista pode ser mais trabalhado, pois é preciso percorrer toda a lista até encontrar o elemento anterior. Isso ocorre porque, dado o ponteiro para um elemento, é possível acessar apenas seu próximo elemento e não o anterior.

Uma **lista duplamente ligada** (ou lista duplamente encadeada) é uma estrutura de dados que consiste em uma sequência de elementos, chamados de nós, onde cada nó contém uma referência para o nó anterior e o próximo nó na lista. Isso significa que cada nó contém dois ponteiros, um para o nó anterior e outro para o próximo nó, permitindo percorrer a lista em ambas as direções, e pode ser vista a ilustração na Figura 20.

Figura 20 – Lista duplamente ligada

Fonte: Rovai (2018).

A lista duplamente ligada pode ser utilizada para armazenar e gerenciar uma coleção de dados de forma dinâmica, permitindo a inserção e remoção eficiente de elementos em qualquer posição da lista.

Na implementação de uma lista duplamente ligada, é necessário definir uma estrutura, que pode ser vista na Figura 21, com pelo menos três campos:

- um para armazenar o valor do elemento;
- um ponteiro que aponta para o próximo elemento;
- outro ponteiro que aponta para o elemento anterior.

Figura 21 – Estrutura para lista duplamente ligada

```
struct lista {
    int info;
    struct lista* ant;
    struct lista* prox;
};
typedef struct lista Lista;
```

Fonte: os autores.

Na Figura 22 é possível observar um exemplo de declaração para criação de uma lista duplamente ligada em linguagem C:

Figura 22 – Exemplo estrutura para lista duplamente ligada

```
struct pessoa {
    char nome[50];
    char sexo;
    int idade;
    struct pessoa* ant;
    struct pessoa* prox;
};
typedef struct pessoa Pessoa;
```

Fonte: os autores.

De acordo com Rovai (2018), também é necessário inicializar a lista duplamente encadeada para que possamos usá-la após sua declaração. Uma forma possível de inicialização é criar uma função que retorna a lista como nula, conforme mostrado na Figura 23.

Figura 23 – Função para retorna uma lista vazia

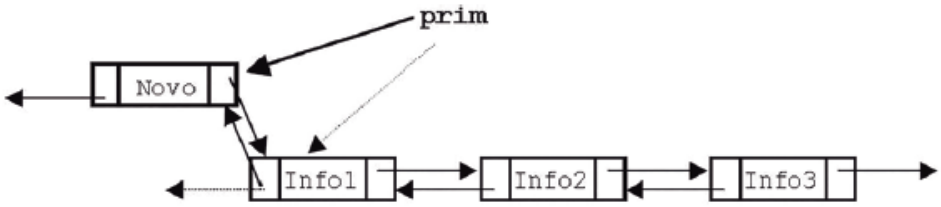
```
Lista* inicializar (void) {
    return NULL;
}
```

Fonte: os autores.

Essa função criará uma lista vazia que poderá ser utilizada posteriormente para a inserção de elementos.

No caso de a lista duplamente encadeada já possuir elementos inseridos, ao adicionar um novo elemento no início da lista, o elemento antigo se tornará o próximo elemento da lista, e o anterior receberá o valor NULL. A Figura 24, apresentada por Rovai (2018), ilustra a adição de um novo elemento no início da lista duplamente ligada.

Figura 24 – Inserir elemento no início da lista duplamente ligada



Fonte: os autores.

A Figura 25 apresenta um trecho de código que ilustra a inserção de um novo elemento no início da lista duplamente ligada.

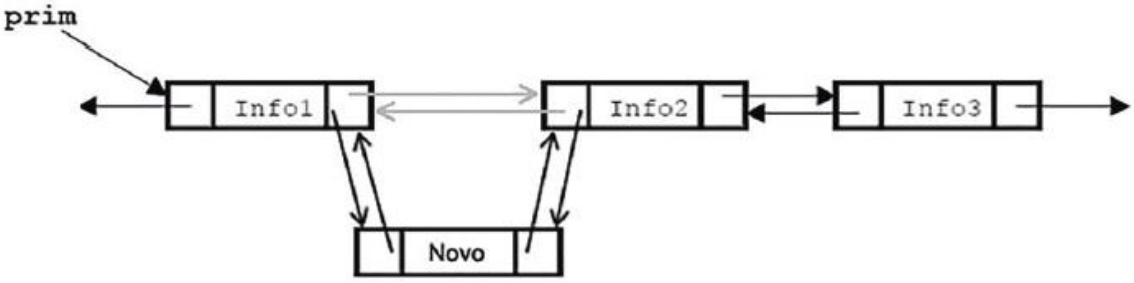
Figura 25 – Inserir elemento na lista duplamente ligada

```
Lista* inserir (Lista* l, int i) {
    Lista* novo = (Lista*) malloc(sizeof(Lista));
    novo -> info = i;
    novo -> prox = l;
    novo -> ant = NULL;
    //Verifica se a lista não está vazia
    if (l != NULL)
        l -> ant = novo;
    return novo;
}
```

Fonte: os autores.

Segundo Rovai (2018), a adição de um novo elemento em uma lista duplamente encadeada pode ser realizada tanto no início quanto no final da lista, assim como em uma posição específica. Na Figura 26, é possível observar a representação da adição de um elemento no meio da lista.

Figura 26 – Inserir elemento no meio da lista duplamente ligada



Fonte: Rovai (2018).

Para isso, é necessário criar um nó com o valor a ser adicionado e atualizar os ponteiros do nó anterior e do próximo nó na lista para apontarem para o novo nó. O novo nó, por sua vez, deve apontar para o nó anterior e para o próximo nó da lista. A Figura 27 apresenta um trecho de código referente à função de adição em uma posição da lista.

Figura 27 – Inserir elemento em uma posição da lista

```
//função de adição em uma posição da lista
Lista* inserirPosicao(Lista* l, int pos, int v){
    int i, cont = 1;
    Lista *p = l;
    Lista* novo = (Lista*)malloc(sizeof(Lista));
    Lista* temp = (Lista*)malloc(sizeof(Lista));
    while (cont != pos){
        p = p -> prox;
        cont++;
    }
    novo -> info = v;
    temp = p -> prox;
    p -> prox = novo;
    novo -> ant = p;
    novo -> prox = temp;
    temp -> ant = novo;
    return l;
}
```

Fonte: os autores.

A Figura 28 apresenta um trecho de código referente à função para adicionar um elemento no final da lista.

Figura 28 – Inserir elemento no final da lista

```
//função de adição no final da lista
Lista* inserirFim(Lista* l, int v){
    int cont = 1;
    Lista *p = l;
    Lista* novo = (Lista*)malloc(sizeof(Lista));
    while (p -> prox != NULL) {
        p = p -> prox;
        cont++;
    }
    novo -> info = v;
    novo -> prox = NULL;
    novo -> ant = p;
    p -> prox = novo;
    return l;
}
```

Fonte: os autores.

Segundo Celes et al. (2004), a função de remoção em uma lista duplamente conectada permite remover um elemento da lista conhecendo apenas o ponteiro para esse elemento. Para facilitar a localização de um elemento na lista, podemos utilizar a função de busca e, em seguida, ajustar o encadeamento da lista. Um trecho de código referente à função de busca pode ser visto na Figura 29.

Figura 29 – Buscar elemento na lista

```
Lista* buscar(Lista* l, int v){
    Lista* p;
    for (p = l; p != NULL; p = p -> prox) {
        if (p -> info == v)
        return p;
    }
    return NULL;
}
```

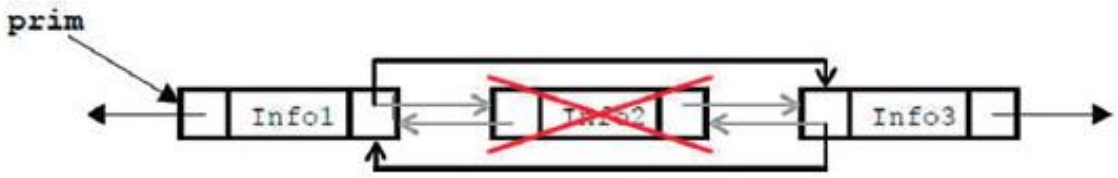
Fonte: os autores.

Encontrado o elemento que se deseja remover em uma lista duplamente ligado, basta ajustar os ponteiros do nó anterior e posterior ao nó a ser removido,

permitindo que o elemento no meio da lista possa ser removido do encadeamento, conforme ilustrado na Figura 30 por Rovai (2018).

Segundo Celes et al. (2004) se o elemento que se deseja remover estiver no início ou no final da lista, o apontamento para o anterior ou próximo será nulo, dependendo da posição do elemento.

Figura 30 – Remover elemento na lista



Fonte: Rovai (2018).

Para remover um elemento da lista duplamente ligada, basta atualizar os ponteiros do elemento anterior e posterior, fazendo com que eles apontem um para o outro, e em seguida, solte o espaço alocado do elemento removido. Um trecho de código referente à função pode ser visto na Figura 31.

Figura 31 – Remoção de um elemento da lista

```
Lista* retira (Lista* l, int v) {
    Lista* anterior = NULL;
    Lista* p = l;
    while (p != NULL && p -> info != v) {
        anterior = p;
        p = p -> prox;
    }
    if (p == NULL )
        return l;
    if (anterior == NULL) {
        l = p -> prox;
    } else {
        anterior -> prox = p -> prox;
    }
    return l;
}
```

Fonte: os autores.

De acordo com Rovai (2018), em uma lista duplamente ligada, é possível realizar a ordenação dos seus elementos de forma crescente ou decrescente por meio de uma simples comparação entre os elementos e, caso aceitem a condição de maior ou menor, realize a sua troca. Um trecho de código referente à função de ordenação pode ser visto na Figura 32.

Figura 32 – Ordenação da lista

```
void Ordena(Lista* l){
    Lista* p;
    Lista* aux;
    int temp;
    for (p = l; p != NULL; p = p -> prox) {
        for (aux = p -> prox; aux != NULL; aux = aux -> prox) {
            if ((p -> info) > (aux -> info)) {
                temp = p -> info;
                p -> info = aux -> info;
                aux -> info = temp;
            }
        }
    }
}
```

Fonte: os autores.

A Figura 33 ilustra a aplicação das funções mencionadas acima, utilizando a linguagem C, para isso foi criada a função principal (main).

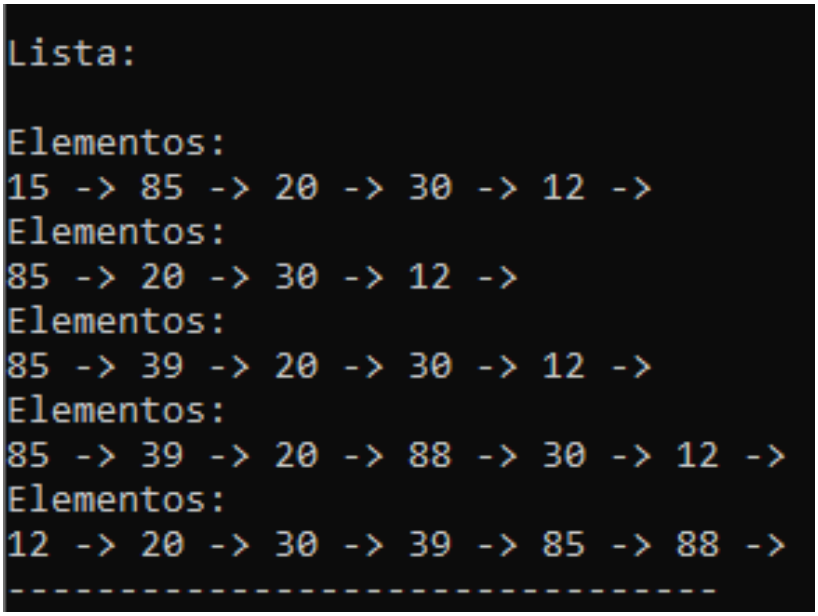
Figura 33 – Função main

```
int main(){
    Lista* listaFinal;
    listaFinal = inicializar();
    listaFinal = inserir(listaFinal, 30);
    listaFinal = inserir(listaFinal, 20);
    listaFinal = inserir(listaFinal, 85);
    listaFinal = inserir(listaFinal, 15);
    listaFinal = inserirFim(listaFinal, 12);
    printf("\nLista:\n");
    imprimir(listaFinal);
    listaFinal = retira(listaFinal, 15);
    imprimir(listaFinal);
    listaFinal = inserirPosicao(listaFinal, 1, 39);
    imprimir(listaFinal);
    listaFinal = inserirPosicao(listaFinal, 3, 88);
    imprimir(listaFinal);
    Ordena(listaFinal);
    imprimir(listaFinal);
    return 0;
}
```

Fonte: os autores.

A Figura 34 a saída da função principal (main) depois de compilada.

Figura 34 – Execução da função main()



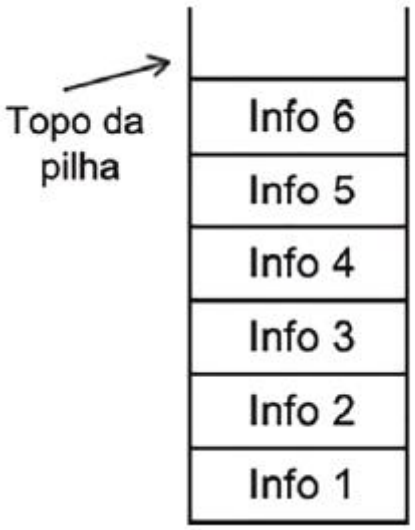
Fonte: os autores.

UNIDADE 2 – PILHAS E FILAS

SEÇÃO 1 – DEFINIÇÃO, ELEMENTOS E REGRAS DE PILHAS E FILAS

Segundo Tanenbaum et al. (2007), uma **pilha** é uma estrutura de dados que consiste em um conjunto de elementos ordenados que permite a inserção e remoção de elementos em apenas uma extremidade da estrutura, denominada topo da pilha. A Figura 35 ilustra essa definição básica.

Figura 35 – Estrutura da Pilha



Fonte: Rovai (2018).

Para acessar um elemento que esteja no meio ou no final da pilha, é necessário remover os elementos colocados acima dele até chegar ao elemento desejado. Como uma pilha é uma estrutura LIFO (Last-In, First-Out), os elementos são acessados apenas através do topo da pilha. Para acessar um elemento em uma posição específica da pilha, todos os elementos acima dele precisam ser removidos da pilha.

Um exemplo comum de uso de pilha é na implementação de algoritmos de expressões aritméticas infixas, onde é necessário converter uma expressão para uma forma postfix (ou polonesa reversa) antes de realizar sua avaliação.

Para isso, podemos utilizar uma pilha para armazenar os operadores e garantir a ordem correta de preferência na conversão da expressão. O processo pode ser realizado percorrendo a expressão de esquerda para direita e, a cada operando

ou operador encontrado, realizando a sua manipulação de acordo com as seguintes regras:

1. Se for um operando (número), adicione-se diretamente à saída.
2. Se for um operador, adicione-se à pilha de operadores.
3. Se for um parêntese de abertura, adiciona-se à pilha de operadores.
4. Se for um parêntese de fechamento, retire-se da pilha de operadores todos os operadores até encontrar o parêntese de abertura correspondente. Cada operador retirado é adicionado à saída.
5. Ao final da leitura da expressão, retire-se todos os operadores que ainda estiverem na pilha e adicione-se à saída.

Por exemplo, para a expressão aritmética infix (3 + 4) * 2 - 1, a sua conversão para postfix utilizando pilha seria:

1. Adiciona-se 3 à saída.
2. Adiciona-se + à pilha de operadores.
3. Adiciona-se 4 à saída.
4. Retira-se + da pilha e adiciona-se à saída.
5. Adiciona-se * à pilha de operadores.
6. Adiciona-se 2 à saída.
7. Retira-se * da pilha e adiciona-se à saída.
8. Adiciona-se 1 à saída.

A expressão postfix resultante seria: 3 4 + 2 * 1 -

Segundo Celes et al. (2004), os elementos da pilha são retirados na ordem inversa da ordem em que foram inseridos. Essa característica é conhecida como LIFO (last in, first out, ou seja, o último que entra é o primeiro a sair) ou FILO (first in, last out, ou seja, o primeiro que entra é o último a sair). É possível observar que a inserção de um elemento sempre ocorre no topo da pilha.

Ainda de acordo com Celes et al. (2004), para criar uma pilha é necessário seguir os seguintes passos:

- criar uma pilha vazia;
- insira elementos no topo da pilha;
- removedor de elementos do topo da pilha;
- verifique se a pilha está vazia;

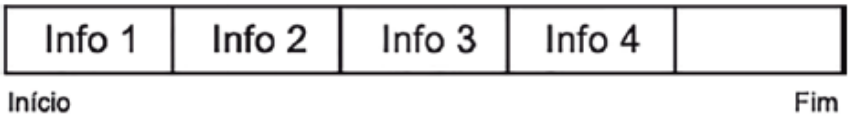
- liberar a estrutura de pilha quando não for mais necessária. É importante ressaltar que a pilha também pode estar no estado de pilha vazia quando não há nenhum elemento inserido.

Regras de Pilhas em linguagem C:

- O último elemento a ser inserido na pilha é o primeiro a ser removido, seguindo o princípio LIFO (Last In First Out).
- Ao inserir um elemento na pilha, o topo da pilha é incrementado em 1 e o elemento é inserido na posição do topo.
- Ao remover um elemento da pilha, o topo da pilha é decrementado em 1 e o elemento que estava no topo é removido.
- É necessário verificar se a pilha está vazia antes de remover um elemento, pois não é possível remover elementos de uma pilha vazia.
- É necessário verificar se a pilha está cheia antes de inserir um elemento, pois não é possível inserir elementos em uma pilha cheia.

Segundo Tanenbaum et al. (2007), uma **fila** é uma estrutura de dados que representa um conjunto de elementos onde a remoção desses elementos é feita por uma chamada de extremidade de início da fila, enquanto a inserção é feita na outra extremidade, conhecida como final da fila, e pode ser vista na Figura 36

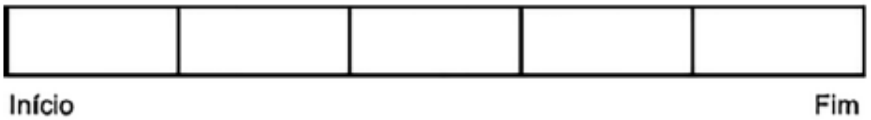
Figura 36 – Estrutura da Fila



Fonte: Rovai (2018).

Quando não houver elementos na fila, ela estará no estado de vazio, conforme demonstrado na Figura 37, de acordo com Rovai (2018).

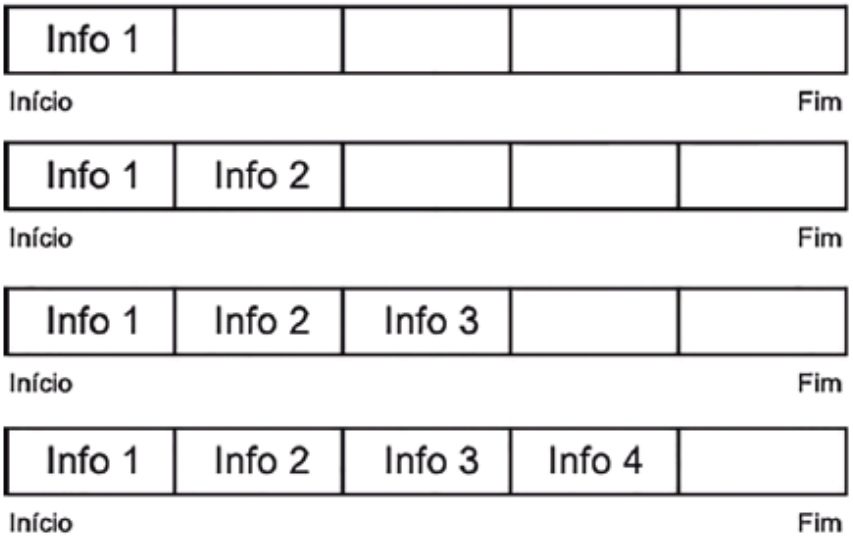
Figura 37 – Fila vazia



Fonte: Rovai (2018).

Segundo Rovai (2018), a fila é uma estrutura de dados que mantém a ordem de entrada dos elementos (fim da fila) e a ordem de saída dos elementos (início da fila), conforme podemos observar na Figura 38.

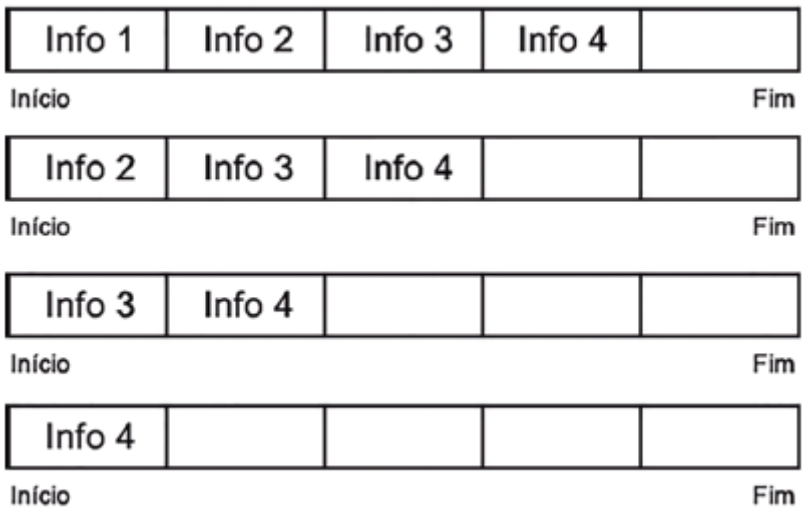
Figura 38 – Entrada de elemento na fila pelo seu final



Fonte: Rovai (2018).

A remoção dos elementos da fila é realizada pela extremidade oposta à entrada, ou seja, pelo início da fila, como exemplificado na Figura 39 de acordo com Rovai (2018).

Figura 39 – Saída de elemento pelo início da fila



Fonte: Rovai (2018).

Segundo Celes et al. (2004), para criar uma pilha é necessário seguir os seguintes passos:

- criar uma fila vazia;
- inserir elemento no final;
- retirar um elemento do início;
- verificar se a fila está vazia;
- liberar a fila.

Regras de Filas em linguagem C:

- A fila funciona seguindo uma regra FIFO (first in, first out), ou seja, o primeiro elemento a ser inserido é o primeiro a ser removido.
- A inserção de um elemento na fila é feita no final dela.
- A remoção de um elemento é feita no início da fila.
- Uma fila pode estar vazia quando não há elementos na estrutura.
- Para criar uma fila vazia, basta inicializar os ponteiros iniciais e fim como NULL.

Conforme Rovai (2018), as estruturas de lista, pilha e fila apresentam diferenças em suas operações. Enquanto a lista permite inserir e remover um elemento em qualquer posição, a pilha permite somente inserir e remover um elemento pelo topo. Já a fila, por sua vez, permite somente a remoção pelo seu início e a inserção pelo seu fim. Essas diferenças tornam essas estruturas mais adequadas para diferentes tipos de problemas e aplicações.

UNIDADE 2 – PILHAS E FILAS

SEÇÃO 2 – OPERAÇÕES E PROBLEMAS COM PILHAS

Conforme Celes et al. (2004), em uma estrutura de Pilha, devem ser integradas duas operações básicas: **push** e **pop**. A operação **push** tem como função inserir um novo elemento na pilha, enquanto a operação **pop** tem como função remover o elemento do topo da pilha. Em C, a função push() é utilizada para empilhar um elemento e a função pop() é utilizada para desempilhar um elemento.

Além disso, conforme Drozdek (2016), outras operações que podem ser executadas na pilha incluem a limpeza da pilha, utilizando a função **clear**(), e verificar se a pilha está vazia, utilizando a função **isEmpty**(). A função clear() remove todos os elementos da pilha, enquanto a função isEmpty() retorna verdadeiro se a pilha estiver vazia e falso caso contrário. Essas funções são úteis para gerenciar e utilizar uma pilha de forma eficiente em programas.

Para compreender as **operações** da estrutura **pilha** segue a solução para o problema de inverter os elementos de uma pilha. Para exemplificar são inseridos três elementos na pilha (10, 20 e 30) e, em seguida, são removidos e impressos os elementos (30, 20 e 10) na ordem inversa da inserção. Por fim, é liberada a memória alocada para a pilha.

Para isso, foi criado uma estrutura Pilha com um atributo **topo** (que indica o índice do elemento no topo da pilha) e um vetor itens (que armazena os elementos da pilha).

Algumas funções necessárias:

- A função **criaPilha()** é utilizada para alocar memória e inicializar a pilha vazia.
- As funções **pilhaVazia()** e **pilhaCheia()** verificam se a pilha está vazia ou cheia, respectivamente.
- As funções **push()** e **pop()** são responsáveis por inserir e remover elementos da pilha, respectivamente. A função **push()** inserir um elemento no topo da pilha, enquanto a função **pop()** remover e retornar o elemento do topo da pilha.

Segue a solução para esse exemplo simples de utilização de pilha em Linguagem C:

```
#include <stdio.h>
#include <stdlib.h>
```

```
#define MAX 10
// Definição da estrutura da pilha
typedef struct {
        int topo;
        int itens[MAX];
} Pilha;
// Função para criar uma pilha vazia
Pilha* criaPilha() {
        Pilha *p = (Pilha*) malloc(sizeof(Pilha));
        p->topo = -1;
        return p;
}
// Função para verificar se a pilha está vazia
int pilhaVazia(Pilha *p) {
        return (p->topo == -1);
}
// Função para verificar se a pilha está cheia
int pilhaCheia(Pilha *p) {
        return (p->topo == MAX-1);
}
// Função para inserir um elemento no topo da pilha
void push(Pilha *p, int item) {
        if (pilhaCheia(p)) {
                printf("Erro: pilha cheia\n");
                return;
        }
p->topo++;
p->itens[p->topo] = item;
}
// Função para remover um elemento do topo da pilha
int pop(Pilha *p) {
        int item;
        if (pilhaVazia(p)) {
```

```
            printf("Erro: pilha vazia\n");
            return -1;
        }
        item = p->itens[p->topo];
         p->topo--;
        return item;
}
// Função Principal
int main() {
        Pilha *p = criaPilha();
push(p, 10);
push(p, 20);
push(p, 30);
printf("%d\n", pop(p));
printf("%d\n", pop(p));
printf("%d\n", pop(p));
free(p);
return 0;
}
```

Segundo Rovai (2018), um dos problemas comuns que podem ser resolvidos utilizando pilhas são os labirintos. Os labirintos são desafios que foram criados para problematizar as estruturas de dados. Além disso, as pilhas podem ser utilizadas em algoritmos de Backtracking, os quais consistem em criar marcações que indicam onde o algoritmo pode retornar.

Ainda de acordo com Rovai (2018) em um labirinto, por exemplo, para encontrar um caminho correto, podemos caminhar pelo labirinto chegarmos em uma divisão no caminho. Dessa forma, adicionamos a posição da divisão, juntamente com o caminho escolhido, na pilha e seguimos por ele. Se o caminho escolhido não fornecer saída, removemos o ponto anterior da pilha, voltando para o último ponto em que o labirinto se divide, e tentamos outro caminho ainda não escolhido, adicionando-o na pilha.

O algoritmo de Backtracking também pode ser utilizado como uma operação de desfazer, presente em diversas aplicações de usuários, como, por

exemplo, em sistemas de GPS. Quando o motorista escolhe uma rota que não foi indicada pelo programa, o algoritmo de Backtracking é aplicado para redefinir a nova rota, conforme Rovai (2018).

UNIDADE 2 – PILHAS E FILAS

SEÇÃO 3 – OPERAÇÕES E PROBLEMAS COM FILAS

Segundo Drozdek (2016), a estrutura de dados de fila possui operações semelhantes às da estrutura de pilha. Para gerenciar uma fila, é necessário:

- criar uma fila vazia;
- inserir um elemento no fim da fila;
- remover o elemento do início da fila;
- verificar se a fila está vazia e liberar a fila.

De acordo com Celes et al. (2004), podemos utilizar um vetor para armazenar os elementos e implementar uma fila nessa estrutura de dados ou utilizar uma alocação dinâmica de memória para armazenar esses elementos.

Um exemplo de uso de fila em C pode ser o controle de impressão em um sistema operacional. Os trabalhos de impressão são adicionados à fila na ordem em que chegam e são processados em ordem de chegada, ou seja, o primeiro trabalho a chegar é o primeiro a ser impresso.

Para isso são necessárias algumas funções básicas de uma implementação de fila em C:

- enqueue: insira um elemento no final da fila.
- dequeue: remove um elemento do início da fila.
- isEmpty: verifica se a fila está vazia.
- isFull: verifique se a fila está cheia.
- front: retorna o elemento do início da fila sem removê-lo.
- rear: retorna o elemento do final da fila sem removê-lo.

Essas funções são comumente utilizadas em implementações de fila em C e são essenciais para a manipulação de elementos na fila.

O código abaixo ilustra um exemplo simples de implementação de fila em C para controle de impressão:

```
#include <stdio.h>
#include <stdlib.h>
#define MAX_QUEUE_SIZE 100
// definição da estrutura da fila
```

```
typedef struct {
    int items[MAX_QUEUE_SIZE];
    int front, rear;
} Queue;
// função para criar uma fila vazia
Queue* createQueue() {
    Queue* q = (Queue*) malloc(sizeof(Queue));
    q->front = -1;
    q->rear = -1;
    return q;
}
// função para verificar se a fila está vazia
int isEmpty(Queue* q) {
    return q->front == -1;
}
// função para verificar se a fila está cheia
int isFull(Queue* q) {
    return q->rear == MAX_QUEUE_SIZE - 1;
}
// função para adicionar um elemento no final da fila
void enqueue(Queue* q, int item) {
    if (isFull(q)) {
        printf("Erro: a fila está cheia\n");
        return;
    }
    if (isEmpty(q)) {
        q->front = 0;
    }
    q->rear++;
    q->items[q->rear] = item;
    printf("Trabalho de impressão %d adicionado à fila\n", item);
}
// função para remover o elemento do início da fila
```

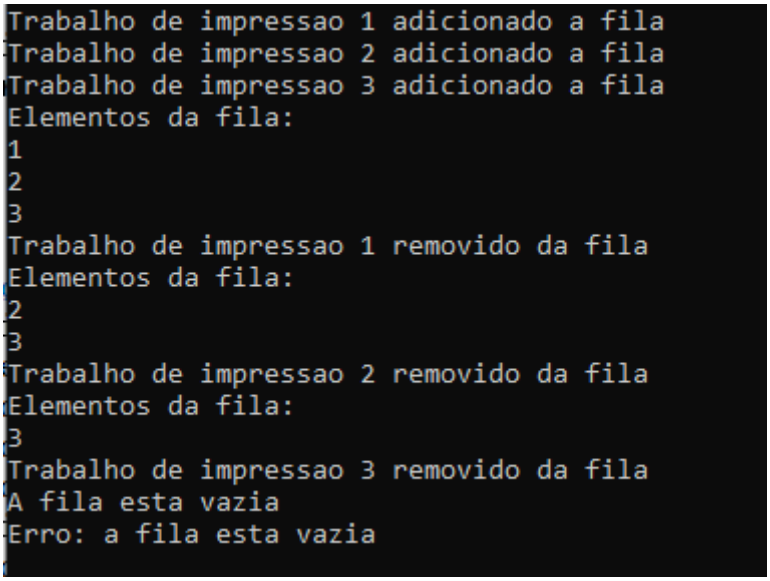
```
int dequeue(Queue* q) {
    if (isEmpty(q)) {
        printf("Erro: a fila está vazia\n");
        return -1;
    }
    int item = q->items[q->front];
    q->front++;
    if (q->front > q->rear) {
        q->front = q->rear = -1;
    }
    printf("Trabalho de impressão %d removido da fila\n", item);
    return item;
}
// função para imprimir os elementos da fila
void printQueue(Queue* q) {
    if (isEmpty(q)) {
        printf("A fila está vazia\n");
        return;
    }
    printf("Elementos da fila:\n");
    for (int i = q->front; i <= q->rear; i++) {
        printf("%d\n", q->items[i]);
    }
}
// função principal
int main() {
    Queue* q = createQueue();
    enqueue(q, 1);
    enqueue(q, 2);
    enqueue(q, 3);
    printQueue(q);
    dequeue(q);
    printQueue(q);
```

```
    dequeue(q);
    printQueue(q);
    dequeue(q);
    printQueue(q);
    dequeue(q);
    return 0;
}
```

Neste exemplo, os trabalhos de impressão são adicionados à fila através da função **enqueue**, e removidos da fila através da função **dequeue**. A função **printQueue** imprime os elementos da fila na ordem em que estão. A função **isEmpty** verifica se a fila está vazia, e a função **isFull** verifica se a fila está cheia (neste caso, a fila tem tamanho máximo definido pela constante MAX_QUEUE_SIZE).

A Figura 40 ilustra a do código do exemplo acima.

Figura 40 – Controle de impressão



Fonte: os autores.

UNIDADE 3 – TABELAS DE ESPALHAMENTO

SEÇÃO 1 – DEFINIÇÃO E USOS DE TABELA DE ESPALHAMENTO

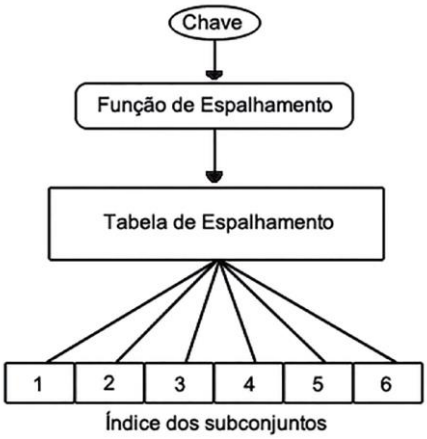
A **tabela de espalhamento**, também conhecida como **Tabela Hash**, é uma estrutura de dados que permite o armazenamento e recuperação eficiente de elementos por meio de uma chave. Essa estrutura utiliza uma função de espalhamento para converter a chave em um índice na tabela, onde o elemento é armazenado.

A principal vantagem da tabela de espalhamento é a sua capacidade de permitir a recuperação rápida de elementos com base na sua chave. Isso ocorre porque o tempo de acesso aos elementos é constante em média, independentemente do tamanho da tabela. Além disso, a tabela de espalhamento é uma estrutura de dados eficiente para a realização de operações de inserção, remoção e busca de elementos.

A tabela de espalhamento é amplamente utilizada em bancos de dados, compiladores, sistemas de busca, criptografia e outras aplicações que exigem acesso eficiente a dados por meio de uma chave. No entanto, é importante ressaltar que a escolha da função de espalhamento e o gerenciamento do espaço ocupado pela tabela são fatores críticos para a eficiência e segurança da estrutura de dados.

Segundo Rovai (2018), a Tabela de Espalhamento é responsável por armazenar os subconjuntos, cada um com um índice específico. Esses índices são gerados por meio da Função de Espalhamento, que deve determinar a qual subconjunto um elemento pertence, analisando suas características-chave. Além disso, a Função de Espalhamento é utilizada para recuperar rapidamente o subconjunto ao qual um elemento pertence, como ilustrado na Figura 41.

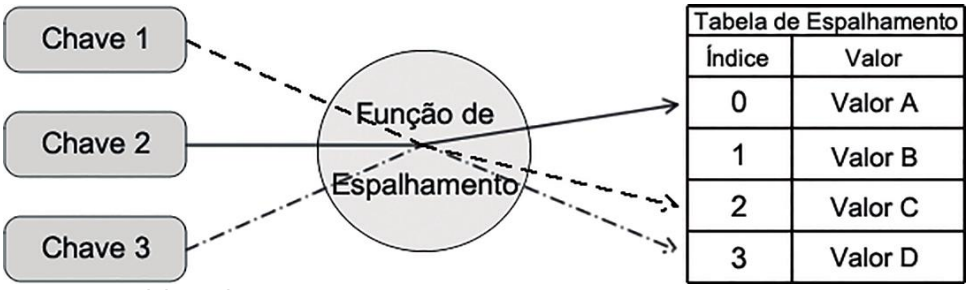
Figura 41 – Exemplo de Tabela de Espalhamento



Fonte: Rovai (2018).

Segundo Silva (2007), a função de espalhamento tem como objetivo converter o valor da chave de um elemento de dados em uma posição dentro de um subconjunto criado na estrutura. A ideia principal da Tabela de Espalhamento é separar os elementos em subconjuntos de acordo com suas características-chave. Dessa forma, a busca de um elemento é baseada na identificação do subconjunto ao qual ele pertence para eliminar os demais subconjuntos, como se fosse feito um filtro de pesquisas. Um exemplo dessa ideia pode ser observado na Figura 42.

Figura 42 – Esquema da Função e Tabela de Espalhamento



Fonte: Rovai (2018).

De acordo com Rovai (2018), o Espalhamento é uma estrutura de dados comum, sendo uma espécie de dicionário ou vocabulário que possibilita as operações de inserção, remoção e pesquisa.

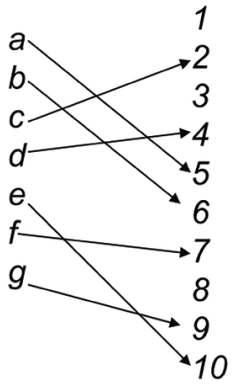
Segundo Drozdek (2016), a Tabela de Espalhamento calcula a posição da chave na tabela com base no valor da chave, permitindo acessar diretamente a posição na tabela quando a chave é conhecida, sem testes preliminares.

Silva (2007) destaca que a ideia geral do Espalhamento é dividir um conjunto de dados em subconjuntos baseados em critérios simples das chaves e gerenciar os subconjuntos menores com métodos simples. A Função de Espalhamento é utilizada para identificar em qual subconjunto podemos inserir ou procurar uma chave, permitindo que, dada uma chave, seja possível identificar em qual subconjunto a informação será armazenada ou procurada.

Segundo Celes et al. (2004), a Função de Espalhamento transforma uma chave em um endereço e associa-a ao endereço relativo do registro, com endereços que parecem aleatórios e podem gerar colisões.

Em resumo, o Espalhamento utiliza uma função que aplica parte da informação, chamada de chave, para retornar ao índice onde a informação deve ou deveria estar armazenada, buscando manter uma informação por índice, sem repetição. A Figura 43 apresenta um exemplo de como a técnica de Espalhamento funciona.

Figura 43 – Exemplo de Espalhamento ideal



Fonte: Rovai (2018).

UNIDADE 3 – TABELAS DE ESPALHAMENTO

SEÇÃO 2 – OPERAÇÕES EM TABELAS DE ESPALHAMENTO

Um exemplo de uso de tabela de espalhamento em C pode ser na implementação de um sistema de controle de estoque. Suponha que a empresa possui milhares de produtos, cada um identificado por um código numérico único.

Para armazenar as informações desses produtos, poderíamos utilizar uma tabela de espalhamento em que a chave seria o código do produto e o valor seria um ponteiro para uma estrutura que contém as informações do produto, como nome, descrição, quantidade em estoque, preço etc. Um exemplo de implementação seria:

```
#include <stdio.h>  #include <stdlib.h>  #include <string.h>
#define TAM_TABELA 10000
// Definição da estrutura do produto
typedef struct {
    int codigo;
    char nome[50];
    char descricao[100];
    int quantidade;
    float preco;
} Produto;
// Definição da tabela de espalhamento
typedef struct {
    Produto* tabela[TAM_TABELA];
} Tabela;
// Função que calcula o hash para uma determinada chave (código do produto)
int hash(int chave) {
    return chave % TAM_TABELA;
}
// Função que insere um produto na tabela
void inserir(Tabela* tabela, Produto* produto) {
    int indice = hash(produto->codigo);
    tabela->tabela[indice] = produto;
}
```

```
// Função que busca um produto na tabela pelo código
Produto* buscar(Tabela* tabela, int codigo) {
    int indice = hash(codigo);
    return tabela->tabela[indice];
}
// Função que remove um produto da tabela pelo código
void remover(Tabela* tabela, int codigo) {
    int indice = hash(codigo);
    tabela->tabela[indice] = NULL;
}
// Função principal
int main() {
    // Inicialização da tabela de espalhamento
    Tabela tabela;
    memset(&tabela, 0, sizeof(Tabela));
    // Criação de alguns produtos
    Produto* p1 = (Produto*) malloc(sizeof(Produto));
    p1->codigo = 123;
    strcpy(p1->nome, "Camisa");
    strcpy(p1->descricao, "Camisa de algodão");
    p1->quantidade = 10;
    p1->preco = 29.99;
    Produto* p2 = (Produto*) malloc(sizeof(Produto));
    p2->codigo = 456;
    strcpy(p2->nome, "Calça");
    strcpy(p2->descricao, "Calça jeans");
    p2->quantidade = 5;
    p2->preco = 59.99;
    // Inserção dos produtos na tabela
    inserir(&tabela, p1);
    inserir(&tabela, p2);
    // Busca de um produto pelo código
    Produto* p = buscar(&tabela, 123);
```

```
    printf("Produto: %s\nDescricao: %s\nQuantidade: %d\nPreco: R$ %.2f\n",
           p->nome, p->descricao, p->quantidade, p->preco);
    // Remoção de um produto da tabela
    remover(&tabela, 456);
    p = buscar(&tabela, 456);
    if (p == NULL) {
        printf("Produto nao encontrado.\n");
    }
    return 0;
}
```

O código apresentado é um exemplo de implementação de uma tabela de espalhamento, que é uma estrutura de dados que permite armazenar e buscar informações com base em uma chave.

- A constante TAM_TABELA define o tamanho da tabela de espalhamento.
- A estrutura Produto é definida com os campos código, nome, descrição, quantidade e preço, representando um item que será armazenado na tabela.
- A estrutura Tabela é definida como uma matriz de ponteiros para Produto, que será utilizada para armazenar os produtos na tabela de espalhamento.
- A função hash é responsável por calcular o índice da tabela a partir da chave (código do produto), utilizando o operador módulo.
- A função inserir insere um produto na tabela de espalhamento, utilizando a função hash para calcular o índice correspondente na tabela.
- A função buscar busca um produto na tabela de espalhamento pelo código, utilizando a função hash para calcular o índice correspondente na tabela.
- A função remover remove um produto da tabela de espalhamento pelo código, utilizando a função hash para calcular o índice correspondente na tabela e atribuindo NULL ao ponteiro correspondente na matriz de produtos.
- A função principal (main) inicializa a tabela de espalhamento, cria alguns produtos, insere os produtos na tabela, busca um produto pelo código, remove um produto da tabela e exibe as informações do produto buscado.

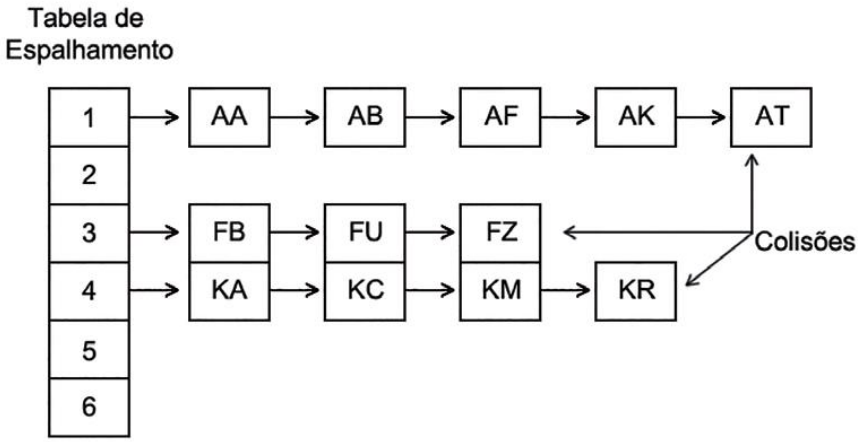
UNIDADE 3 – TABELAS DE ESPALHAMENTO

SEÇÃO 3 – OTIMIZAÇÃO DE TABELAS DE ESPALHAMENTO

De acordo com Rovai (2018), as Tabelas de Espalhamento podem ter seu desempenho comprometido devido ao aumento do número de colisões, que ocorrem quando dois ou mais elementos têm a mesma chave calculada pela função de espalhamento. Isso pode levar a um acúmulo de elementos em um mesmo índice da tabela, o que prejudica a eficiência da busca ou armazenamento de informações.

Segundo Celes et al. (2004) quanto maior o número de colisões, pior será o desempenho da tabela. No entanto, ainda é aceitável ter um certo grau de colisões, desde que a função de espalhamento consiga distribuir os elementos de forma razoavelmente uniforme, como pode ser visto na Figura 44.

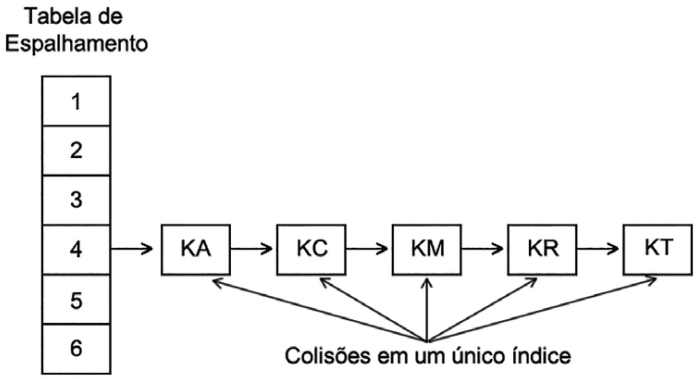
Figura 44 – Tabela de Espalhamento com colisões aceitáveis



Fonte: Rovai (2018).

Conforme Rovai (2018), há uma situação conhecida como "pior caso", em que todas as chaves da Função de Espalhamento são direcionadas para um único índice da tabela, o que pode causar um grande acúmulo de elementos nesse índice e, consequentemente, prejudicar o desempenho da Tabela de Espalhamento. Essa situação pode ser observada na Figura 45.

Figura 45 – Pior caso de uma Tabela de Espalhamento



Fonte: Rovai (2018).

Conforme Rovai (2018), é crucial que uma Tabela de Espalhamento tenha uma Função de Espalhamento bem definida para garantir um bom desempenho. Caso a função não seja adequada para as chaves, o desempenho da tabela será prejudicado, como no caso de utilizar uma função sem tratamento de colisões.

De acordo com Celes et al. (2004), para garantir um bom desempenho da função, um requisito básico é que ela forneça uma distribuição uniforme nos índices da tabela. Se houver não uniformidade na distribuição, o número de colisões aumentará, assim como o esforço para buscar ou armazenar os elementos na tabela. No entanto, é possível adotar algumas técnicas de otimização para garantir uma melhor distribuição dos elementos na Tabela de Espalhamento.

De acordo com Drozdek (2016), há várias técnicas de otimização para funções de espalhamento, mas as três mais comuns são:

- Endereçamento fechado;
- Endereçamento aberto;
- Encadeamento separado.

O **endereçamento fechado**, também conhecido como hashing com sondagem linear, consiste em utilizar todas as posições da tabela de espalhamento para armazenar os elementos. Quando ocorre uma colisão, ou seja, quando dois elementos tentam ser armazenados na mesma posição da tabela, é realizada uma sondagem linear, isto é, são verificadas as posições seguintes da tabela, em ordem, até que uma posição livre seja encontrada. A principal desvantagem dessa técnica é

que a tabela precisa ser grande o suficiente para armazenar todos os elementos sem que ocorra muita colisão.

O **endereçamento aberto**, ou hashing com sondagem quadrática, funciona de forma semelhante ao endereçamento fechado, mas em vez de utilizar todas as posições da tabela, apenas uma porcentagem delas é preenchida. Quando ocorre uma colisão, é realizada uma sondagem quadrática, isto é, as posições seguintes são verificadas de forma quadrática, utilizando uma fórmula matemática específica. Essa técnica permite uma maior ocupação da tabela e, consequentemente, uma menor quantidade de colisões, mas requer uma função de sondagem mais complexa.

O **encadeamento separado**, ou hashing com listas encadeadas, consiste em utilizar uma lista encadeada em cada posição da tabela. Quando ocorre uma colisão, o novo elemento é adicionado à lista correspondente. Essa técnica não requer uma tabela grande, mas pode gerar um maior consumo de memória, pois cada elemento precisa de um espaço adicional para armazenar o ponteiro que o liga à lista.

UNIDADE 4 – ARMAZENAMENTO ASSOCIATIVO

SEÇÃO 1 – DEFINIÇÃO E USOS DE MAPAS DE ARMAZENAMENTO

Mapas de Armazenamento (ou Storage Maps, em inglês) são estruturas de dados utilizadas para armazenar informações sobre a alocação de espaço em um dispositivo de armazenamento de dados, como discos rígidos ou SSDs.

Esses mapas mantêm informações sobre os blocos de armazenamento do dispositivo, indicando quais blocos estão livres e quais estão ocupados por dados. Eles permitem que o sistema operacional ou outras aplicações gerenciem o armazenamento, reservando blocos para novos arquivos ou liberando espaço ocupado por arquivos excluídos.

Os mapas de armazenamento são amplamente utilizados em sistemas operacionais e sistemas de arquivos, e são fundamentais para garantir um uso eficiente do espaço em disco e um bom desempenho de leitura e gravação de dados. Eles também podem ser utilizados em aplicações que envolvem o armazenamento de grandes quantidades de dados, como bancos de dados e sistemas de backup.

Existem várias técnicas para implementar mapas de armazenamento, incluindo tabelas de bits, listas de blocos livres e árvores de alocação de espaço. A escolha da técnica depende das características do dispositivo de armazenamento e das necessidades específicas do sistema ou aplicação que utiliza o mapa de armazenamento.

Conforme Goodrich e Tamassia (2013), o Armazenamento Associativo é uma estrutura que possibilita o acesso aos seus elementos por meio do seu valor, sem depender da sua posição na estrutura. Em algumas situações, utiliza-se somente uma parte do valor do elemento, denominada chave, em vez do valor completo do elemento.

Segundo Pereira (2008), mapas não podem ser definidos por fórmulas, ao contrário das funções. Um mapa é definido por meio de uma relação entre seus pares, que nem sempre é baseado em questões lógicas ou matemáticas. A função dos mapas associativos é controlar a associação entre elementos dentro da estrutura, realizando a associação entre uma chave e um valor recebido, permitindo a recuperação rápida de um valor associado a uma chave.

Conforme Rovai (2018), os mapas associativos são estruturas de dados que permitem implementar as seguintes funcionalidades:

- adicionar uma associação;
- verificar um valor de uma chave específica;
- remover uma associação de uma chave específica;
- verificar se existe uma associação para determinada chave;
- informar a quantidade de associações na estrutura.

Segue um exemplo de uso de mapa associativo em C usando a biblioteca padrão <map>: #include <stdio.h>:

```
#include <iostream>
#include <map>
using namespace std;
int main() {
    // Definindo o mapa associativo
    map<string, int> notas;
    // Adicionando elementos ao mapa
    notas["Joao"] = 9;
    notas["Maria"] = 10;
    notas["Jose"] = 8;
    // Recuperando valores do mapa
    cout << "Nota de Joao: " << notas["Joao"] << endl;
    cout << "Nota de Maria: " << notas["Maria"] << endl;
    cout << "Nota de Jose: " << notas["Jose"] << endl;
    // Verificando se uma chave existe no mapa
    if (notas.count("Joao") > 0) {
        cout << "Joao esta presente no mapa!" << endl;
    } else {
        cout << "Joao nao esta presente no mapa!" << endl;
    }
    // Removendo um elemento do mapa
    notas.erase("Maria");
    // Verificando o tamanho do mapa
```

```
    cout << "Quantidade de elementos no mapa: " << notas.size() << endl;
    return 0;
}
```

Nesse exemplo, o mapa associativo é utilizado para armazenar as notas de alunos, com as chaves sendo os nomes dos alunos e os valores sendo as notas. São realizadas operações de adição, recuperação, verificação de existência e remoção de elementos no mapa.

UNIDADE 4 – ARMAZENAMENTO ASSOCIATIVO

SEÇÃO 2 – MAPAS COM LISTA

A estrutura de Mapa com Lista em C é uma implementação de um mapa associativo utilizando uma lista encadeada. Em termos simples, um mapa associativo é uma estrutura de dados que permite associar um valor a uma chave. Essa estrutura é frequentemente utilizada para armazenar informações em pares de chave-valor, de modo que a busca de um valor a partir de uma chave seja feita de forma rápida.

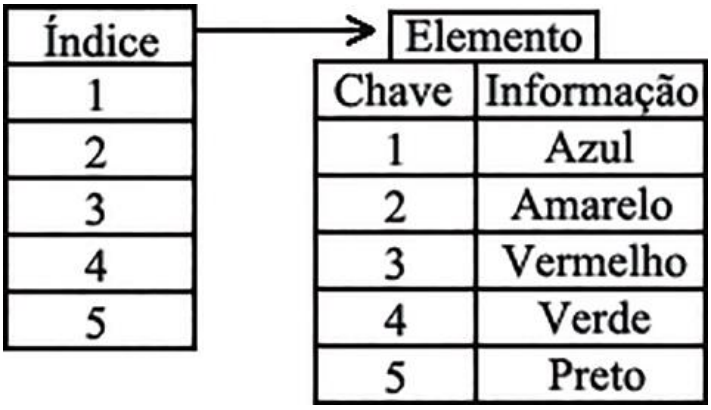
A implementação de um mapa com lista em C é feita através da criação de uma estrutura que armazena os pares chave-valor. Essa estrutura é composta por dois campos: um campo para a chave e outro para o valor. Em seguida, é criada uma lista encadeada que armazena essas estruturas de pares.

A inserção de um novo par chave-valor na lista é feita através da criação de uma nova estrutura de par e da adição dessa estrutura ao início da lista. A busca por um valor a partir de uma chave é feita percorrendo a lista e comparando a chave do par atual com a chave buscada. Quando a chave é encontrada, o valor correspondente é retornado.

A remoção de um par chave-valor é feita percorrendo a lista e localizando a estrutura de par correspondente à chave. Quando essa estrutura é encontrada, ela é removida da lista e a memória alocada para ela é liberada.

Conforme Rovai (2018), o índice é utilizado como uma referência associada a uma chave, com o objetivo de otimizar o desempenho e permitir uma localização mais rápida de um elemento quando solicitado, conforme ilustrado na Figura 46.

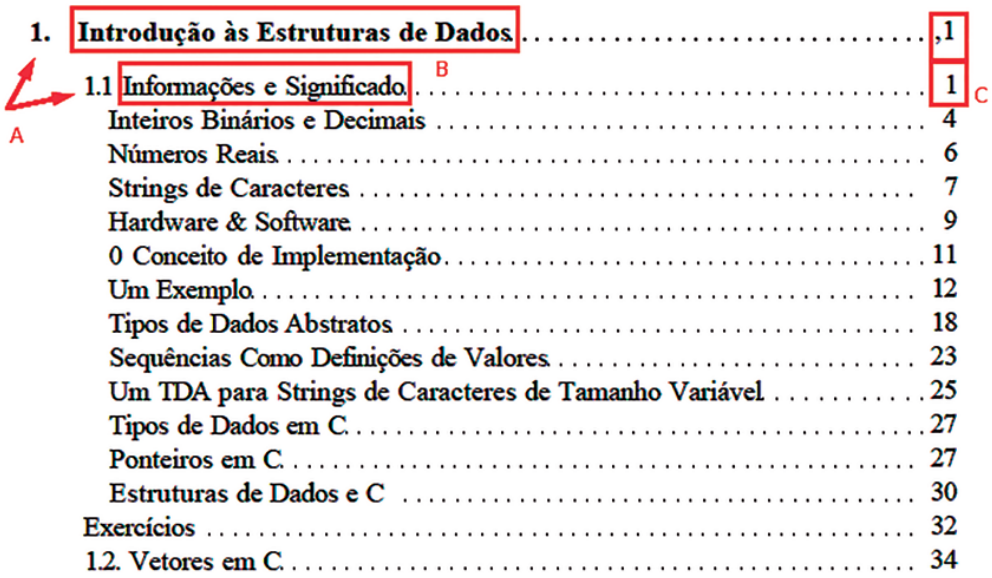
Figura 46 – Exemplo de um Mapa com Lista



Fonte: Rovai (2018).

Conforme Rovai (2018), um exemplo adicional de uso de índices é a criação de um resumo de um livro. O sumário é uma lista de informações referentes aos assuntos que serão vistos no livro, como pode ser observado na Figura 47.

Figura 47 – Exemplo de sumário



Fonte: Rovai (2018).

Um exemplo de implementação em C do mapa com a lista pode ser encontrado abaixo:

```
#include <iostream>
#include <map>
using namespace std;
// Cria a estrutura da lista
struct Restaurante{
string descprato;
string ingredprato;
};
int main(){
int i = 0, chave, opcao;
char descricao[15], ingr[50];
// Cria o Mapa com Lista
map<int, Restaurante> MapaPratos;
```

```
Restaurante pratos;
do {
system(“cls”);
printf(“Selecione uma opcao:\n”);
printf(“1 - Adicionar um Prato\n”);
printf(“2 - Remover um Prato\n”);
printf(“3 - Listar todos os Pratos\n”);
printf(“4 - Sair\n”);
scanf(“%d”, &opcao);
switch (opcao){
case 1:
system(“cls”);
printf(“Adicionar nova disciplina\n”);
printf(“Informe o nome do prato: “);
scanf(“%s”, &descricao);
pratos.descprato = descricao;
printf(“\nInforme os ingredientes do prato: “);
// A função gets recebe o texto todo da variável ingr, incluindo espaço
gets(ingr);
pratos.ingredprato = ingr;
MapaPratos[(MapaPratos.size()+1)] = pratos;
cout << “\nPrato adicionado com sucesso!” << endl;
cout << “\nPrato: “ + MapaPratos[MapaPratos.size()].
descprato + “\nIngredientes: “ + MapaPratos[MapaPratos.size()].
ingredprato + “\n” << endl;
system(“pause”);
opcao = 0;
break;
case 2:
system(“cls”);
printf(“Informe o numero do prato para remover: “);
scanf(“%d”, &chave);
// Verifica se o número do prato existe
```

```
if(MapaPratos.find(chave) == MapaPratos.end()){
cout << “Prato nao existente!\n” << endl;
} else {
MapaPratos.erase(MapaPratos.find(chave));
}
system(“pause”);
opcao = 0;
break;
case 3:
system(“cls”);
printf(“*** Listagem dos pratos! ***\n”);
for (i = 1; i <= MapaPratos.size(); i++) {
printf(“Prato %d: “, i);
// Se o prato pesquisado existir, escreve a posição e os dados da chave
if(MapaPratos.find(i) == MapaPratos.end())
cout << “Prato nao encontrada!\n” << endl;
else
// Caso exista o prato, retorna as informações na lista associada à chave
cout << “\nNome: “ + MapaPratos[i].descprato
+ “\nIngrediente: “ + MapaPratos[i].ingredprato + “\n” << endl;
}
system(“pause”);
opcao = 0;
break;
default:
break;
}
}while (opcao != 4);
return 0;
}
```

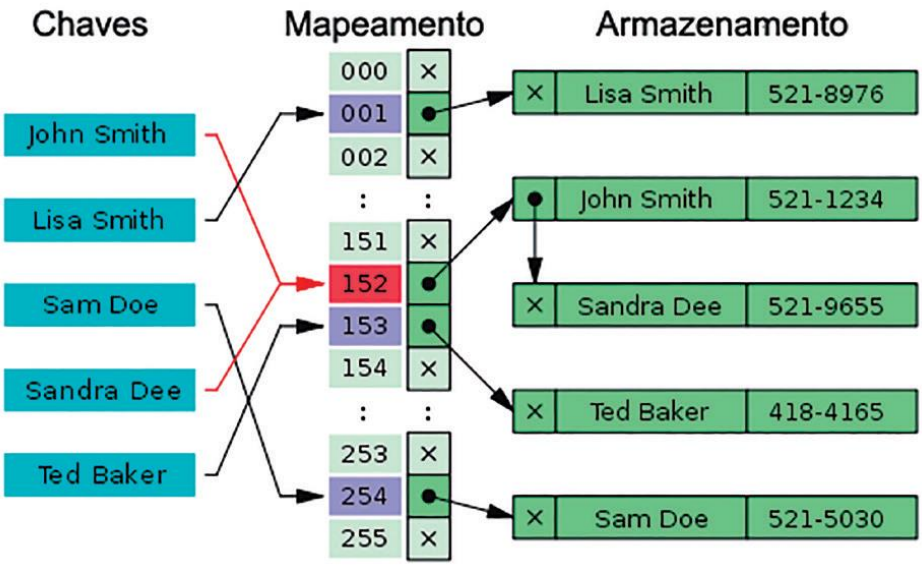
UNIDADE 4 – ARMAZENAMENTO ASSOCIATIVO

SEÇÃO 3 – MAPAS COM ESPALHAMENTO

Tenenbaum (1995) afirmou que, assim como a técnica de Espalhamento pode melhorar o desempenho de estruturas de dados para acesso rápido às informações, ela também pode ser aplicada em Mapas para suprir essa necessidade de melhoria no acesso aos dados. Goodrich (2013) acrescentou que, assim como em Mapas com Listas, as chaves em Mapas com Espalhamento também devem ser únicas.

O Mapa com Espalhamento apresenta algumas características, tais como a ausência de ordenação dos elementos, a rapidez na busca e inserção de dados, e a possibilidade de inserir valores e chaves nulas. De acordo com Rovai (2018), o Mapa com Espalhamento permite gerenciar uma sequência de elementos por meio de uma Tabela de Espalhamento, na qual cada entrada armazena uma lista de nós vinculados e cada nó armazena um elemento contendo uma chave e um valor mapeado. A Figura 48 ilustra esse processo.

Figura 48 – Exemplo de Mapa com Espalhamento



Fonte: Rovai (2018).

De acordo com Rovai (2018), um Mapa com Espalhamento é uma estrutura de dados associativa que utiliza uma função de espalhamento para agrupar seus

elementos com base na chave. Essa função gera uma associação entre elemento e chave, permitindo uma busca rápida e inserção de dados. Essa técnica é amplamente utilizada quando se trabalha com valores nomeados, em que a posição do elemento não é relevante, mas sim o valor da sua chave. Em resumo, um Mapa com Espalhamento é uma estrutura de armazenamento eficiente para acesso rápido às informações por meio de uma chave.

REFERÊNCIAS:

CELES, W.; CERQUEIRA, C.; RANGEL, J. L. Introdução a estrutura de dados: com técnicas de programação em C. Rio de Janeiro: Campus Elsevier, 2004.

DROZDEK, A. Estrutura de dados e algoritmos em C++. Trad. 4. ed. norte-americana. São Paulo: Cengage Learning, 2016.

FORBELLONE, A.L.V. Lógica de programação: a construção de algoritmos e estrutura de dados. São Paulo: Prentice Hall, 2005.

GOODRICH, M.T.; TAMASSIA, R. Estruturas de dados & algoritmos em Java. São Paulo: Bookman, 2013.

LORENZI, F.; MATTOS, P.N.; CARVALHO, T.P. Estruturas de dados. São Paulo: Cengage Learning, 2015.

PEREIRA, S.L. Estrutura de dados fundamentais: conceitos e aplicações. São Paulo: Érica, 2008.

ROVAI, K.R. Algoritmos e estrutura de dados. Londrina: Editora e Distribuidora Educacional, 2018.

SILVA, O.Q. Estrutura de dados e algoritmos usando C: fundamentos e aplicações. Rio de Janeiro: Ciência Moderna, 2007.

TANENBAUM, A.M.; LANGSAM, Y.; AUGENSTEIN, M.J. Estrutura de dados usando C. São Paulo: Pearson Prentice Hall, 2007.

VELOSO, P.A.S. Estruturas de dados. Rio de Janeiro: Campus-Elsevier, 1996.