LISTA LINEAR ENCADEADA

Prof. Dr. Juliano Varella de Carvalho

FEEVALE DIGITAL

LISTA LINEAR ENCADEADA

Vamos imaginar que você precisa armazenar uma lista de diversos itens na memória do seu computador. Esses itens podem ser informações sobre clientes, detalhes de produtos, livros, carros, etc. Basicamente, existem duas alternativas para que isso ocorra: guardar esses itens em um vetor sequencial (alocação estática), ou salvar os itens usando alocação dinâmica. No primeiro caso denominaremos a estrutura de dados de Lista Linear Sequencial, e no segundo de Lista Linear Encadeada.

A Lista Linear Sequencial utiliza espaço contíguo (adjacente) na memória, ou seja, os itens (nodos) armazenados são organizados em blocos de memória sequenciais. Desta forma, é possível acessar os itens diretamente por intermédio dos índices associados a cada um deles. No entanto, como já vimos, o tamanho de memória alocado é dimensionado em tempo de compilação, exigindo, muitas vezes, que se faça a previsão de um número grande de itens que acabam não sendo utilizados durante a execução do programa.

A Lista Linear Encadeada, por sua vez, armazena os nodos em porções de memória não contíguas. Dessa maneira, os nodos não podem ser acessados diretamente, usando índices. No entanto, esse tipo de lista permite que os nodos sejam adicionados ou removidos à medida que seja necessário. Portanto, não é preciso alocar espaço de memória em tempo de compilação.

Nesse material focaremos nossos estudos em uma Lista Linear Encadeada. Trabalharemos com dois tipos de Listas Lineares Encadeadas, as Listas Simplesmente Encadeadas e as Listas Duplamente Encadeadas.

Lista Simplesmente Encadeada

Uma lista simplesmente encadeada é uma sequência conectada de elementos denominados nodos da lista. Ou seja, um nodo (ou nó) é o elemento que será manipulável em uma lista. Se trabalharemos com uma lista de produtos, o nodo é o produto; se temos uma lista de clientes, o nodo é um cliente; se precisamos armazenar uma lista de pontos turísticos, o nodo é um ponto turístico. O nodo é composto de dois campos: a **informação** que ele armazena e um **ponteiro** para o próximo nodo da lista, conforme Figura I.

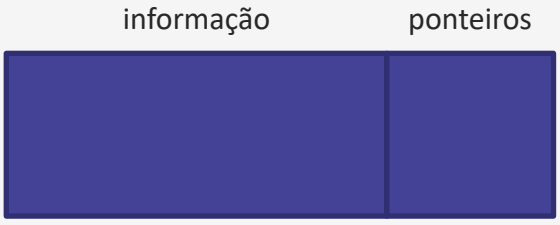

Figura I – Representação de um nodo.

Um nodo, participante de uma lista, pode ser responsável por guardar informações sobre um ponto turístico. Imagine que o ponto turístico contenha as seguintes informações necessárias: descrição, latitude (lat) e longitude (long), de acordo com a Figura II.

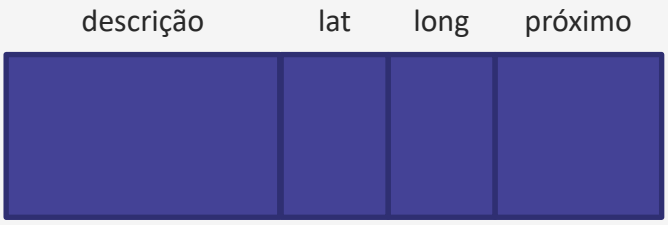

Figura II – Exemplo de um nodo que representa um ponto turístico.

Logo, se precisarmos manter vários nodos em uma lista simplesmente encadeada, ela pode ser representada como ilustrado na Figura III.

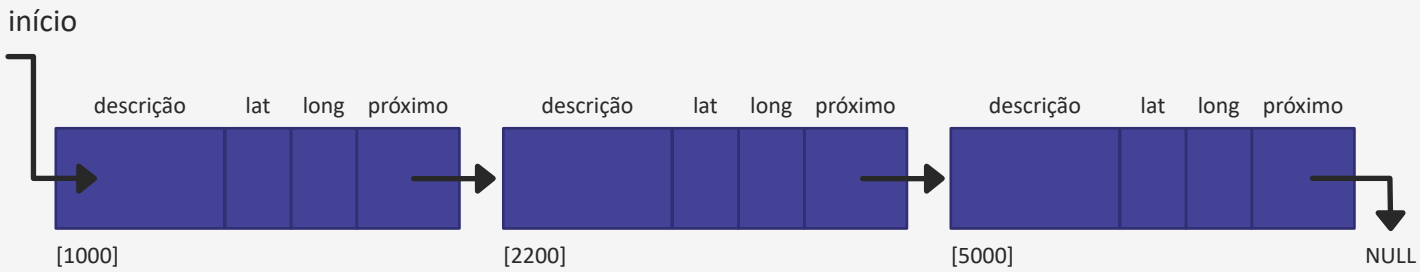

Figura III – Lista simplesmente encadeada com três nodos.

Na lista simplesmente encadeada, temos a presença de um ponteiro denominado início, responsável por guardar o endereço de memória onde está localizado o primeiro nodo da lista, ou seja, onde a lista inicia. Esse ponteiro início é comumente denominado de cabeça da lista. Note que o ponteiro próximo, de cada nodo, aponta para um endereço de memória onde estará o nodo subsequente. O primeiro nodo inicia no endereço [1000], o segundo nodo começa no endereço [2200] e o terceiro e último nodo inicia no endereço [5000]. O ponteiro do último nodo da lista aponta para NULL, indicando que a lista termina nesse nodo. Se pensarmos em valores para os nodos, a lista poderia ser representada como na Figura IV.

Figura IV – Lista simplesmente encadeada composta por três pontos turísticos.

Logo, a nossa TAD (Tipo Abstrato de Dados) Lista Simplesmente Encadeada necessitará de um ponteiro para guardar o endereço inicial da lista (cabeça da lista), um tipo de dado para representar o nodo e, por fim, um conjunto de operações para manipular a lista. Dentre essas operações destacaremos a criação da lista; a inserção de um nodo no início da lista; a busca de um nodo específico na lista; a remoção de um nodo específico da lista; e, por fim, a destruição de toda a lista.

Tipo de dado para representar o nodo

Utilizaremos em nosso exemplo de Lista Simplesmente Encadeada o armazenamento de Pontos Turísticos em memória. Para isto, os nodos da lista terão a estrutura demonstrada na Figura V.

```
struct PontoTuristico {
    char descricao[41];
    float latitude;
    float longitude;
    struct PontoTuristico *proximo;
};
```

Figura V – Struct PontoTuristico que representa um tipo definido pelo programador.

Note que cada ponto turístico será armazenado em 53 Bytes: 41 Bytes para a descrição do ponto turístico, 4 Bytes para sua latitude, mais 4 Bytes para a longitude e outros 4 Bytes para armazenar o ponteiro para o próximo nodo na memória.

Criação de um ponteiro para representar o cabeça da lista

Na função *main()* será necessário criar o ponteiro responsável por apontar para o início da lista simplesmente encadeada, ele se será denominado de cabeça da lista, ilustrado pela Figura VI.

```
…
int main() {
    struct PontoTuristico *cabeca;
…
```

Figura VI – Definição de um ponteiro (cabeça), responsável por apontar para início da lista.

Imagine que o seu programa deseja fazer diversas manipulações sobre a lista simplesmente encadeada: criar a lista, inserir novos pontos turísticos na lista, buscar nodos da lista, remover nodos da lista, dentre outras operações. Tais chamadas a essas funções (operações) estarão logo a seguir a criação do cabeça da lista. Veremos a partir de agora como implementar tais operações (Figura VII).

```
…
int main() {
    struct PontoTuristico *cabeca;
    cabeca = criar_lista();
    cabeca = inserir_inicio(cabeca);
    cabeca = inserir_final(cabeca);
…
```

Figura VII – Exemplo de chamadas às funções (operações) da lista.

Criação de uma lista

A função responsável por criar a lista não parâmetro de entrada e simplesmente deve retornar NULL. Esse retorno fará com que o ponteiro que faz a chamada dessa função aponte para NULL, ou seja, o cabeça da lista é inicializado com NULL (Figura VIII).

```
struct PontoTuristico* criar_lista(void) {
    return NULL;
};
```

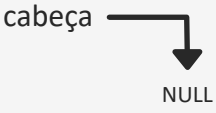

Figura VIII – Função de criação da lista e representação do estado do ponteiro cabeça da lista.

Inserção de um nodo no início da lista

A operação para inserir um novo nodo na lista é composto de quatro instruções, conforme exibido na Figura IX.

```
struct PontoTuristico* inserir_inicio(struct PontoTuristico* cabeca) {
    struct PontoTuristico* novo = (struct PontoTuristico*) malloc(sizeof(struct
PontoTuristico));
    lePontoTuristico(novo);
    novo->proximo = cabeca;
    return(novo);
}
```

Figura IX – Código da função responsável por criar novos nodos no início da lista.

A função *inserir_inicio()* recebe como parâmetro de entrada o cabeça da lista. A primeira instrução aloca um bloco de memória para inserir as informações do novo nodo (Figura X (a)). Logo em seguida, os valores relacionados ao novo ponto turístico são inseridos dentro dele (Figura X (b)). Para inserção dos valores utiliza-se uma função denominada *lerPontoTuristico()*. A terceira instrução faz com que o ponteiro próximo do novo nodo (que será o nodo inicial) aponte para o primeiro nodo do estado atual da lista (Figura X (c)). A última instrução retorna o endereço do nodo novo para o cabeça da lista (na função *main()*), pois o nodo novo recém incluído será o primeiro da lista (Figura X (d)).

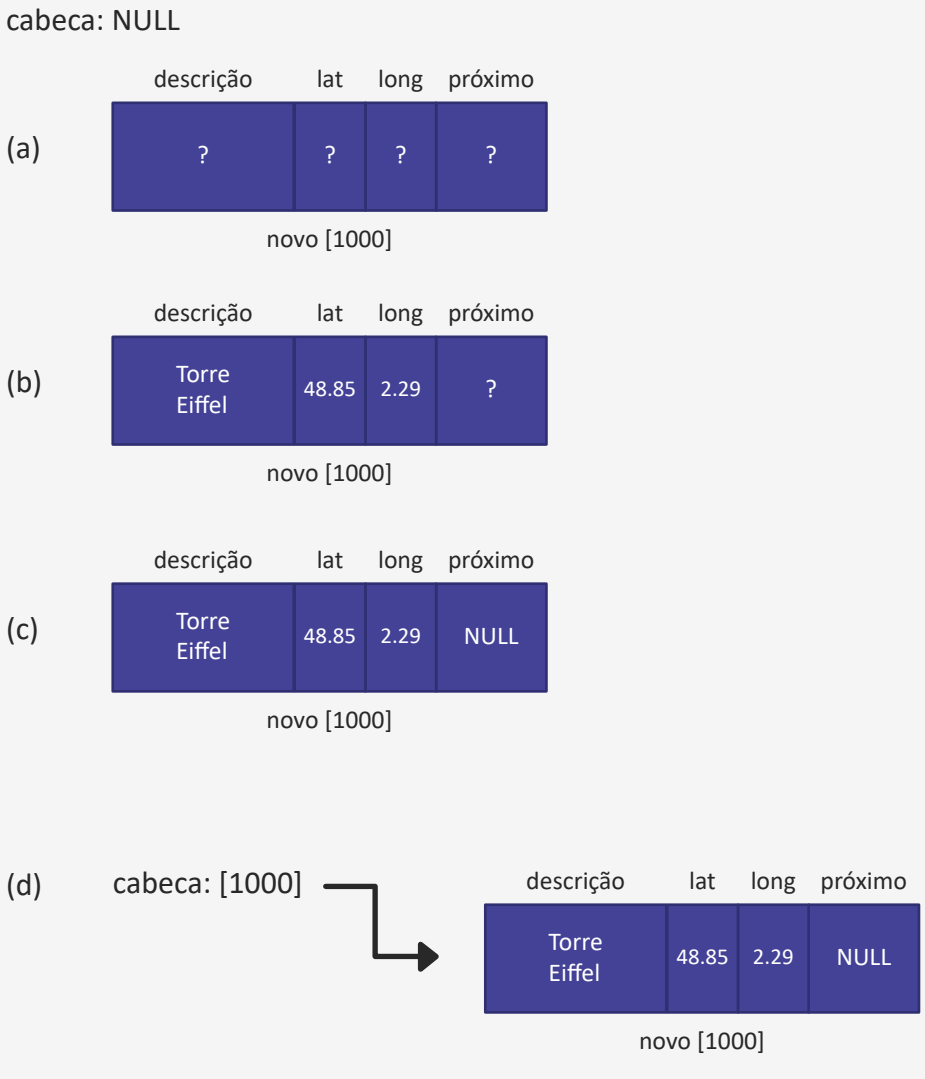

Figura X – Passo a passo da inserção do primeiro nodo na lista simplesmente encadeada.

A Figura XI ilustra a inserção de um outro nodo na lista. Lembrando que esse novo nodo a ser inserido será o primeiro da lista, pois essa função sempre insere um nodo novo no início dela.

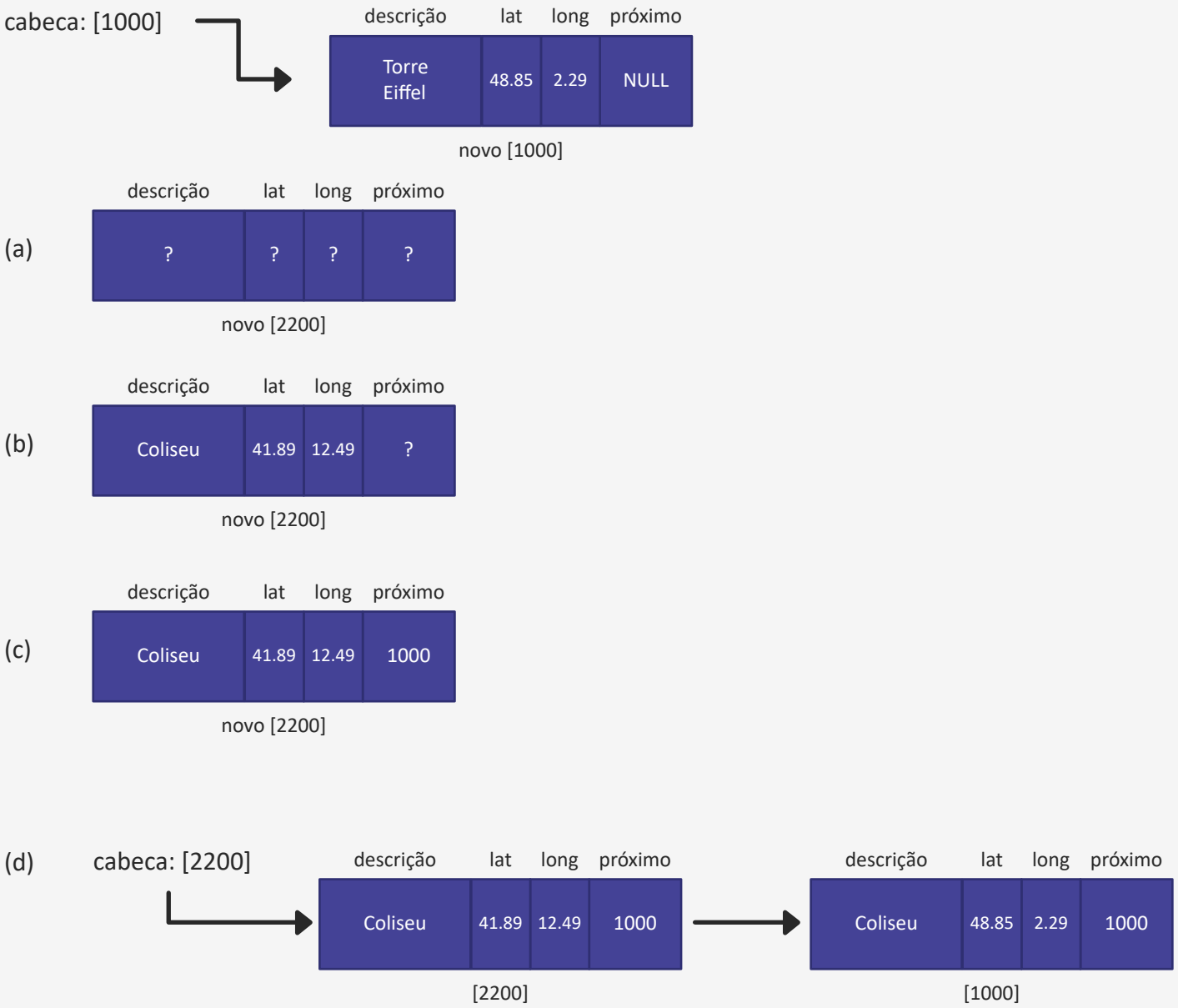

Figura XI – Passo a passo da inserção de um outro nodo na lista.

A Figura XII ilustra a função `lerPontoTuristico()`, responsável por solicitar ao usuário os dados do novo ponto turístico a ser inserido na lista.

```c
void lerPontoTuristico(struct PontoTuristico *aux) {
    printf("\nDescricao: ");
    scanf("%[^\n]", aux->descricao);
    fflush(stdin);
    printf("\nLatitude: ");
    scanf("%f", &aux->latitude);
    fflush(stdin);
    printf("\nLongitude: ");
    scanf("%f", &aux->longitude);
    fflush(stdin);
}
```

Figura XII – Preenchimento dos dados no novo nodo.

Impressão dos dados dos nodos da lista

Uma função importante no contexto da lista é a impressão de seu conteúdo. A função `imprimir()`, representada pela Figura XIII, recebe por parâmetro o cabeça da lista, ou seja, o endereço inicial da lista. De posse de um ponteiro auxiliar *(paux)* a função passa nodo a nodo, imprimindo o conteúdo de cada nodo. A função é interrompida quando o ponteiro *paux* encontra o valor NULL, ou seja, recebe o valor do ponteiro próximo do último nodo. A Figura XIV ilustra os diversos valores assumidos pelo ponteiro auxiliar *paux*, considerando que a lista esteja com três pontos turísticos.

```
void imprimir(struct PontoTuristico *cabeca) {
    struct PontoTuristico *paux;
    for (paux = cabeca; paux != NULL; paux = paux->proximo)
       printf("Descricao: %s\tLatitude: %.2f\tLongitude: %.2f\t\n",paux->descricao,
paux->latitude, paux->longitude);

}
```

Figura XIII – Impressão dos dados de todos os nodos da lista.

Logo, se precisarmos manter vários nodos em uma lista simplesmente encadeada, ela pode ser representada como ilustrado na Figura III.

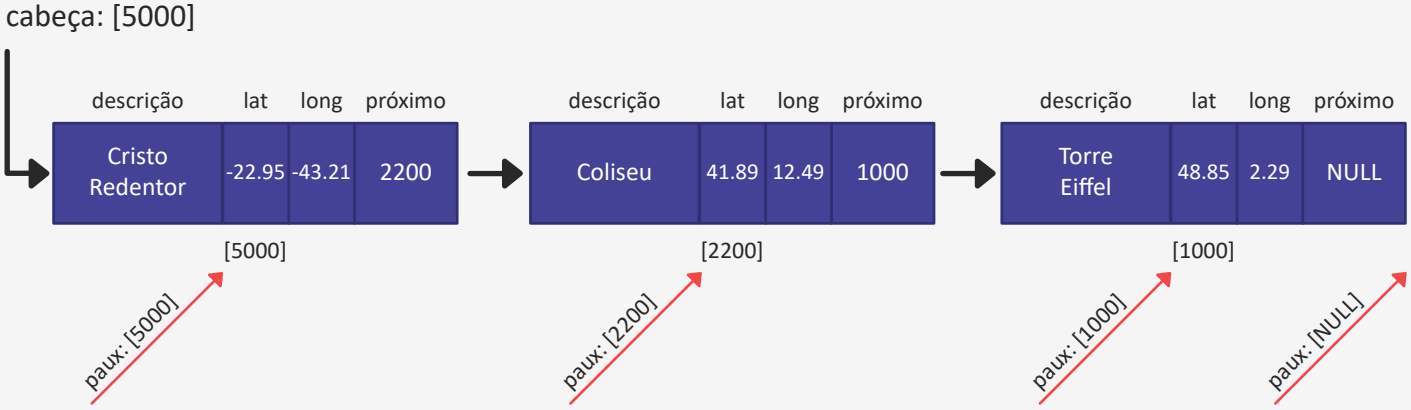

Figura XIV – Comportamento do ponteiro auxiliar *paux*, a cada iteração do laço *for*.

Busca de um nodo específico da lista

A função `buscar_lista()` recebe dois parâmetros de entrada: o cabeça da lista e uma informação que represente o nodo que se deseja encontrar. Caso o nodo seja encontrado na lista, seu endereço em memória é retornado, caso não seja encontrado, retorna-se NULL. A ideia de retornar o endereço do nodo permite ao responsável pela chamada da função `buscar_lista()`, saber onde o nodo encontra-se na memória e manipular suas informações como quiser.

A Figura XV ilustra a função `buscar_lista()`. Para o exemplo dos pontos turísticos, ela recebe uma *string* no parâmetro *desc*, por exemplo, “Coliseu”. Posteriormente, usando um ponteiro auxiliar *paux*, é realizada uma varredura sobre todos os nodos da lista, iniciando-se pelo cabeça dela. A cada iteração (a cada nodo encontrado) verifica-se se o campo descrição do nodo atual é igual a *string* “Coliseu”. Caso seja igual, o endereço do nodo atual é devolvido para quem chamou a função *buscar_lista()*. Se não houver um nodo cuja descrição seja “Coliseu”, a função retornará o valor NULL.

```
struct PontoTuristico* buscar_lista(struct PontoTuristico* cabeca, char *desc) {
    struct PontoTuristico  *paux;
    for (paux=cabeca; p!=NULL; paux=paux->proximo)
        if (strcmpi(paux->descricao, desc) == 0)
            return paux;
    return NULL;
}
```

Figura XV – Código ilustrando a busca de um nodo específico na lista.

Nota: a função *strcmp()* compara duas *strings* e retorna 0 caso elas sejam iguais. Essa função faz distinção de maiúsculas e minúsculas (*sensitive*). A função *strcmpi()* também compara duas *strings* e também retorna 0 caso elas sejam iguais. No entanto essa função não faz distinção de maiúsculas e minúsculas (*insensitive*).

Remoção de um nodo específico da lista

A função `remover_lista()` recebe dois parâmetros de entrada: o cabeça da lista e a informação sobre qual nodo excluir da lista. E ela devolve um ponteiro para um nodo. Essa função basicamente avalia três situações:

i. O nodo a ser excluído não é encontrado, logo a função retorna NULL para quem a chamou.
ii. O nodo a ser excluído é o primeiro da lista. Nesse caso, o cabeça da lista passa a ser o segundo nodo, o bloco de memória alocado para o cabeça da lista anterior é removido e é retornado a quem chamou a função o endereço do novo cabeça da lista.
iii. Um outro nodo, que não o primeiro é excluído. Nesse caso, o nodo anterior ao que se quer excluir deve apontar para o nodo posterior daquele que se deseja excluir.

A Figura XVI apresenta o código da função `remover_lista()`, com diversos comentários para ilustrar as três situações descritas.

```
struct PontoTuristico* remover_lista(struct PontoTuristico* cabeca, char *desc) {
    struct PontoTuristico *paux = cabeca,
    struct PontoTuristico *anterior_paux = NULL;
    /* Laço while responsável por verificar qual nodo deve ser excluído.
       Observe que o ponteiro paux varrerá nodo a nodo, a cada iteração.
       Atente também para o ponteiro anterior_paux, o qual estará sempre apontando
para o nodo anterior ao de paux.
    */
    while (paux != NULL && strcmpi(paux->descricao, desc) != 0) {
        anterior_paux = paux;
        paux = paux->proximo;
    }
    // A função cairá no if abaixo caso o nodo a ser excluído não exista na lista.
    if (paux == NULL)
        return cabeca;
    // A função cairá no if abaixo caso o nodo a ser excluído seja o primeiro da
lista.
    if (anterior_paux == NULL)
        cabeca = paux->proximo;
    /* O else será realizado caso o nodo a ser excluído seja qualquer um diferente
do primeiro da lista.
    */
    else
        anterior_paux->proximo = paux->proximo;
    free(paux);
    return cabeca;
}
```

Figura XVI – Função *remover_lista()* comentada.

Imagine que se deseje remover o nodo que tem a descrição com valor igual a “Coliseu”, ou seja, o parâmetro *desc* entrou na função com o valor “Coliseu”. O laço *while* portanto irá procurar pela posição específica do nodo com essa descrição, utilizando para isso os ponteiros *paux* e *anterior_paux*. Ao encontrar o nodo de interesse, o ponteiro *paux* estará apontando para o endereço [2200] e o ponteiro *anterior_paux* estará apontando para o endereço [5000]. A Figura XVII (a) ilustra esse instante.

Como o nodo a ser excluído não encontra-se na primeira posição a função executará a linha posterior à instrução *else*: `anterior_paux->proximo = paux->proximo;` . Após a execução dessa linha a lista ficará conforme a Figura XVII (b). A Figura XVII (c) detalha a liberação do espaço alocado que antes estava reservado para o nodo, instrução `free(paux)`.

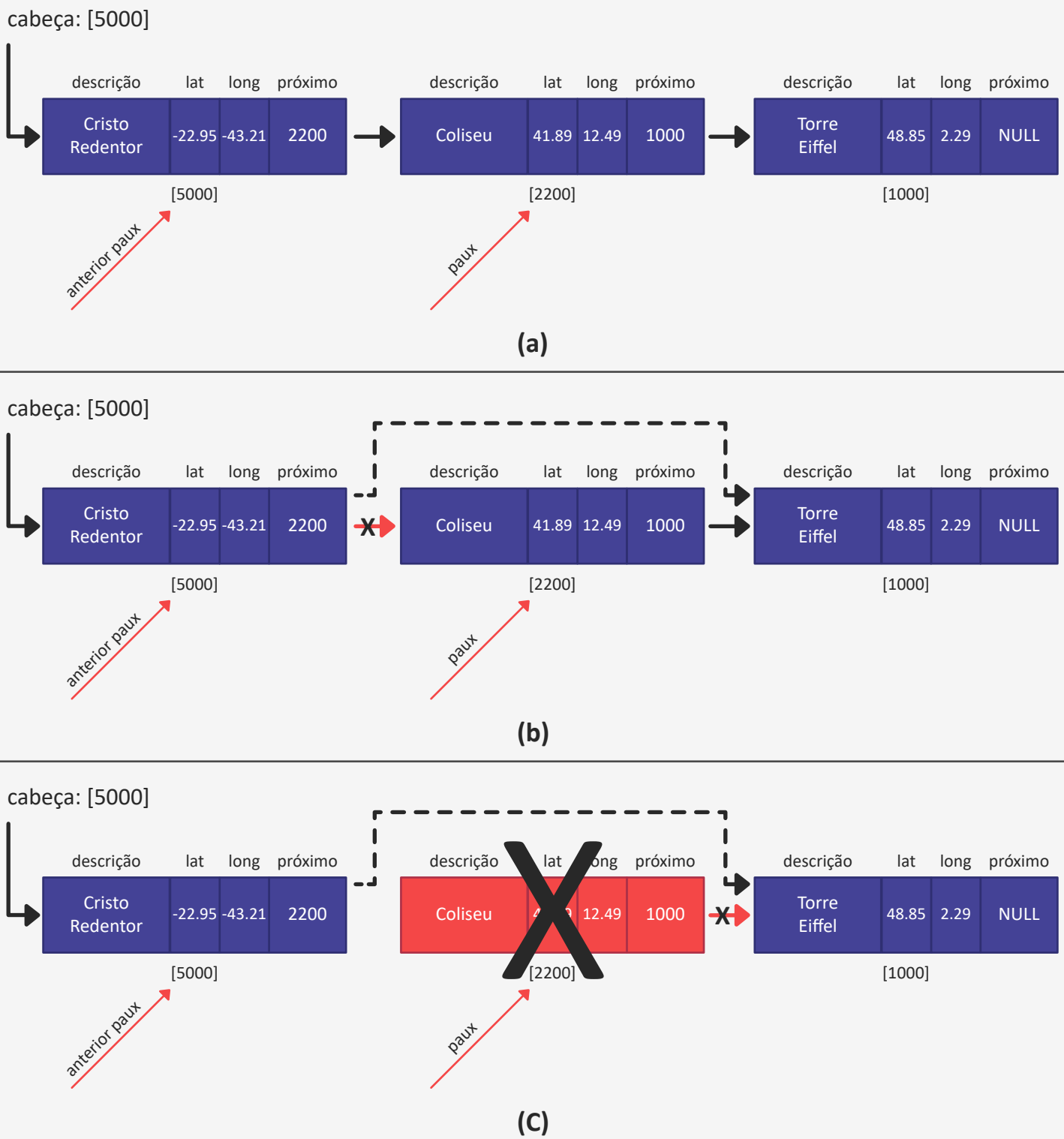

Figura XVII (a), (b), (c) – Passo a passo da remoção de nodos da lista.

Liberação dos nodos da lista

Essa função tem como objetivo desalocar todos os nodos da lista. Ela é responsável por passar por cada nodo da lista e, a cada iteração, liberar o bloco de memória utilizado pelo nodo. A Figura XVIII ilustra o código responsável por essa operação.

```
struct PontoTuristico* liberar_lista(struct PontoTuristico* cabeca) {
    struct PontoTuristico *paux = cabeca, *aux = NULL;
    while (paux != NULL) {
        aux = paux->proximo;
        free(paux);
        paux = aux;
    }
    return NULL;
}
```

Figura XVIII – Função que libera todos os nodos da lista.

Referências:

CELES, Waldemar; CERQUEIRA, Renato; RANGEL, José Lucas. **Introdução a estruturas de dados: com técnicas de programação em C. 1.** ed. São Paulo, SP Campus, 2004. 294 p. ISBN 8535212280