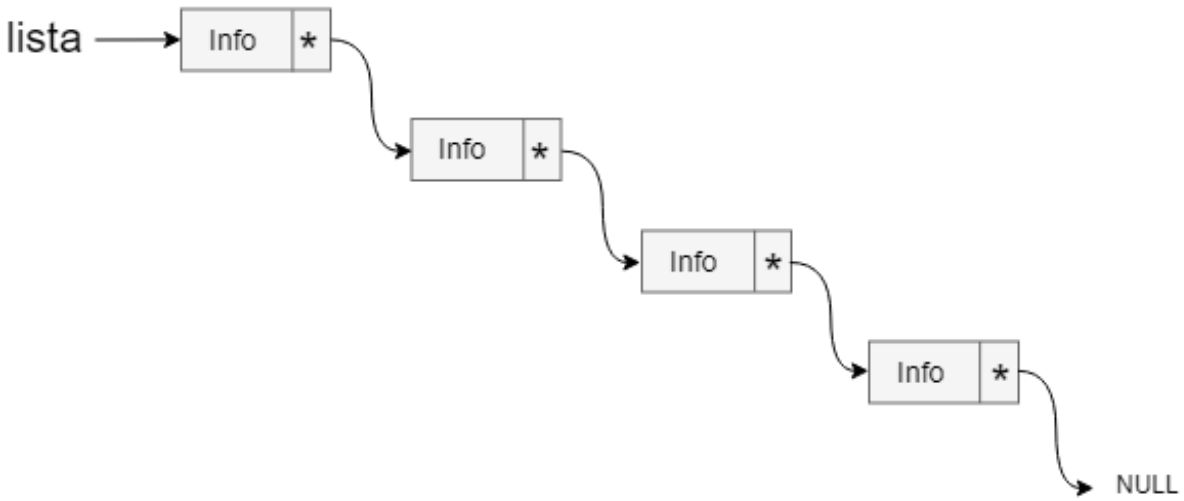
LISTA ENCADEADA



3º SEMESTRE - BANCO DE DADOS

DANIELLY GARCIA JARDIM

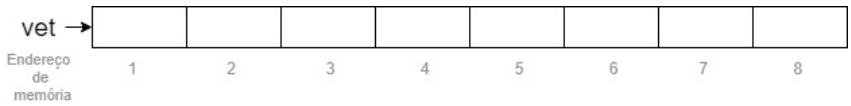
FEVEREIRO DE 2019.

Sumário

1 Introdução

1.1 Lista encadeada x vetor

Quando criamos um vetor, estamos reservando um espaço sequencial na memória, de tamanho limitado (como mostra a imagem abaixo), e o tamanho desse vetor não poderá ser alterado posteriormente.



Como podemos ver, *vet* é um ponteiro para o primeiro elemento do vetor. A partir dele, podemos acessar os próximos elementos através do operador de indexação (*vet[índice]*). Então podemos dizer que o vetor é uma estrutura de acesso randômico aos seus elementos, pois podemos acessar qualquer elemento sem precisar antes passar pelos que o antecedem.

Porém, um vetor tem um tamanho fixo, e isso pode ser um problema dependendo do que você precisar fazer. A solução para este problema é utilizar um outro tipo estrutura, uma que possa crescer ou diminuir, de acordo com a necessidade. Tal estrutura é chamada de lista encadeada (ou *linked list*, em inglês).

Na lista encadeada, a cada novo elemento inserido, é necessário que antes seja alocado um espaço na memória para armazena-lo. Desta maneira, não ocuparemos mais espaço do que o necessário. Porém, na lista encadeada, não poderemos acessar randomicamente cada elemento, ou seja, para que possamos acessar um elemento qualquer, devemos antes passar pelos que o antecedem. Para que consigamos fazer isso, devemos saber onde o próximo estará armazenado, ou seja, deve haver uma referência a ele. Cada elemento deverá conter um ponteiro que irá armazenar o endereço para o próximo, formando assim, o encadeamento. Como os elementos são alocados individualmente, eles não ficam armazenados em sequência na memória, diferente do que acontece com o vetor.

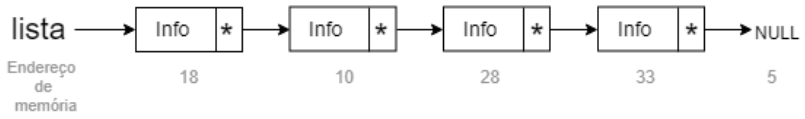
2 Implementação da struct

Cada espaço dessa lista é chamado de nó. Cada nó é representado por uma *struct*, que deve conter um ponteiro para o próximo nó e pode armazenar N informações. Segue um exemplo:

```c
#include <stdio.h>

struct pessoa {
    char nome[50];
    int idade;
    float altura, peso;

    struct pessoa *prox;
};

typedef struct pessoa Pessoa;
```

O ponteiro deve ser do mesmo tipo da estrutura, pois ela irá apontar para outra estrutura idêntica (exceto pelo conteúdo dos valores, mas a tipagem e a quantidade serão sempre as mesmas).

A lista será representada por um ponteiro para o seu primeiro elemento, do mesmo tipo da *struct* que no caso, é pessoa. Dessa forma, saberemos onde a lista se inicia. Mas também precisamos saber onde ela termina. Por isso, o último elemento deve apontar para *NULL*.

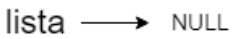


Para simplificar a explicação, utilizarei a seguinte estrutura:

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

struct lista {
    int valor;
    struct Lista *prox;
};

typedef struct lista Lista;
```

3 Inicializando a lista

A função que inicializa a lista, irá cria-la sem elemento algum. Então, o primeiro elemento também é o último. A lista deverá se iniciar com *NULL*. Também é correto atribuirmos *NULL* no momento em que criamos o ponteiro, isso irá variar de acordo com seu gosto.



Implementação da função *criarLista()*:

```
Lista *criarLista() {
    return NULL;
}
```

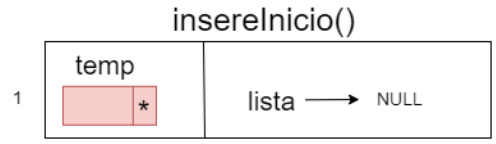
4 Inserção de valores

Uma vez criada e inicializada a lista, podemos começar a inserir valores. Sempre que inserirmos um novo valor na lista, devemos alocar dinamicamente a memória para que possamos armazena-lo. Existem duas formas de inserirmos de valores em uma lista: No início e no final. Exibirei um exemplo de implementação de cada uma, e uma explicação sobre elas. A que você utilizará, irá depender de cada caso e de sua preferência.

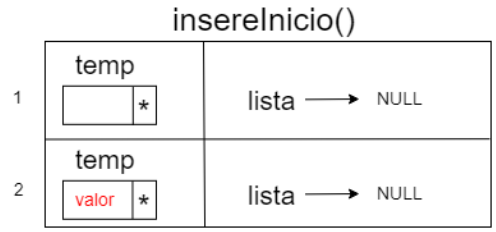
4.1 Inserção no início

Quando precisamos inserir um valor no início da lista, não podemos esquecer que o ponteiro para o primeiro elemento deverá ser atualizado. Se não fizermos isso, o valor adicionado será perdido. A função deverá ser do tipo Lista, e retornará um ponteiro (para o novo primeiro elemento). Ela deverá receber dois valores como parâmetro: a *lista* e o *valor* a ser adicionado. No início da função, devemos criar um ponteiro para o novo nó que iremos criar. Chamaremos esse ponteiro de *temp*.

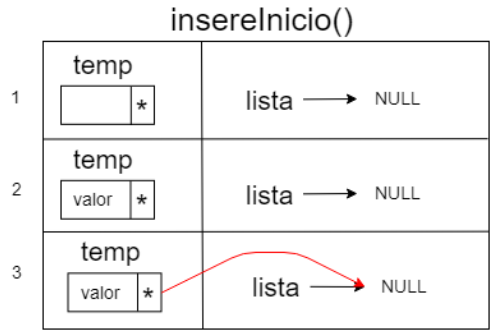
O novo nó que criamos por enquanto está vazio, e a lista ainda está como *NULL*, pois não inserimos valor algum nela ainda..



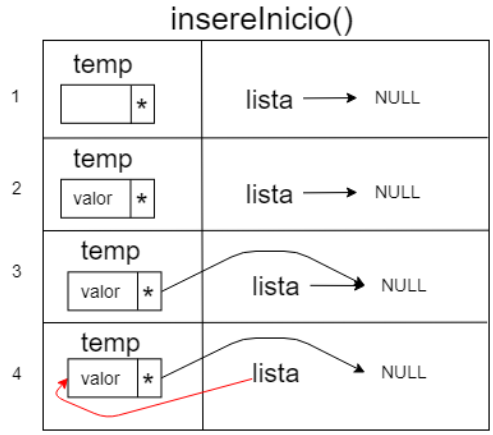
Então, adicionamos um valor ao nó que acabamos de criar:



E fazemos com que o ponteiro do nó criado receba o conteúdo de *lista*, ou seja, que receba o endereço do primeiro elemento da lista, que passará a ser o segundo.



E por fim, devemos retornar *temp*, para que a lista seja atualizada. A variável que criamos na função *main* (que nesse caso é *lista*) deve receber o retorno dessa função para apontar para o novo primeiro elemento, que é *temp*.



Esta função aloca dinamicamente o espaço para armazenar o novo nó da lista, guarda o valor no nele, e o faz apontar para o elemento que era o primeiro da lista. Então a função retorna o novo valor, atualizando a lista.

Possível implementação da função:

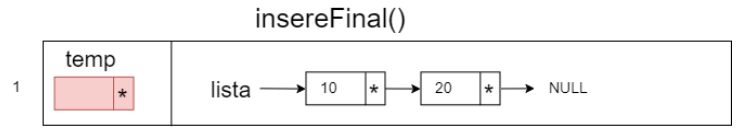
```
Lista *insereInicio(Lista *lista, int valor) {
    Lista *temp = malloc (sizeof(Lista));
    temp->valor = valor;
    temp->prox = lista;

    return temp;
}
```

4.2 Inserção no final

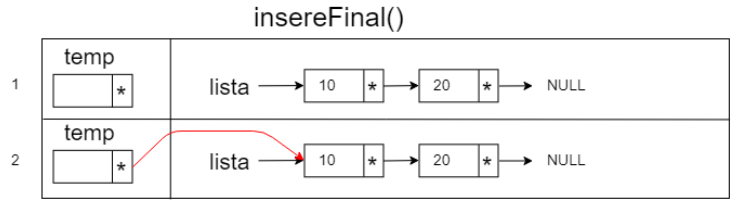
Para fazer a inserção no final, precisamos percorrer toda a lista, até encontrarmos o valor *NULL*. Ao inserirmos o novo valor, não podemos esquecer de fazer com que o último elemento aponte para *NULL*.

A função retornará um ponteiro, para que a lista seja atualizada. Ela deverá receber dois valores como parâmetro: a lista e o valor a ser adicionado. Assim como a função *insereInicio()*, utilizaremos a um ponteiro *temp* do tipo lista, que deverá apontar para o novo valor a ser inserido.

Aqui temos uma lista com alguns elementos já inseridos e com *temp* já criado.

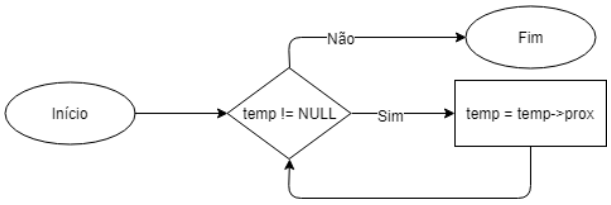


Porém, ao invés de *temp* receber o novo valor, *temp→prox* receberá a lista, ou seja, ele deverá apontar para o primeiro elemento dela.

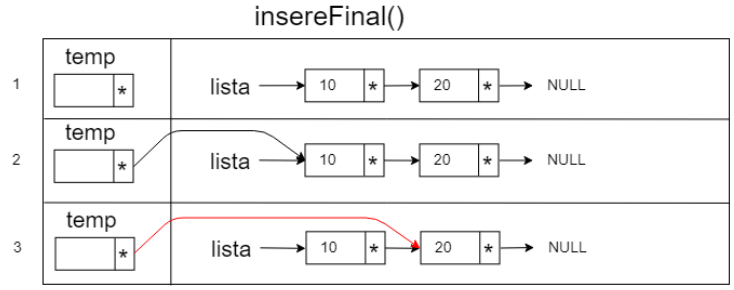


Agora devemos percorrer toda a lista, utilizando uma estrutura de repetição a sua escolha. O *loop* deve rodar até que *temp* seja igual a *NULL*.

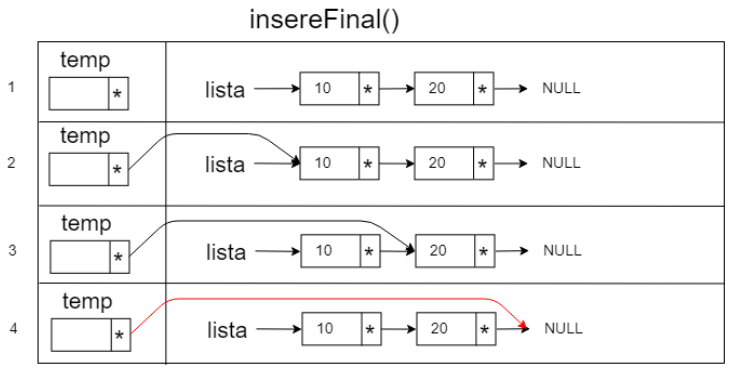
Então se *temp→prox* for diferente de *NULL*, devemos avançar o ponteiro *temp* para o próximo elemento da lista, para que ele possa percorrer toda ela. Observe o fluxograma a seguir.



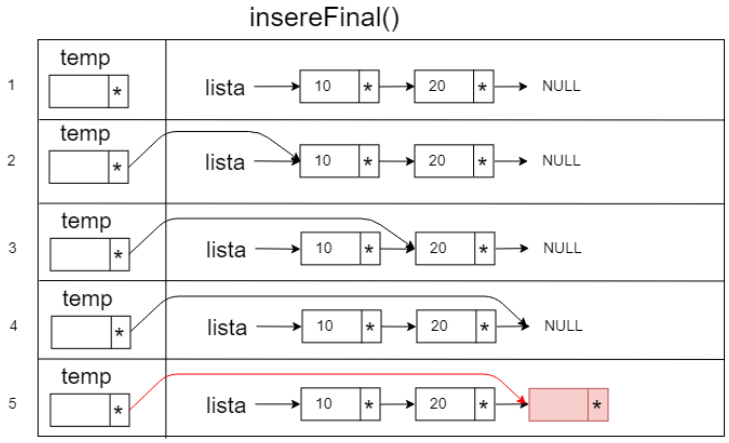
Enquanto a condição for satisfeita (*temp→prox* for diferente de *NULL*), *temp* irá avançar pela lista.



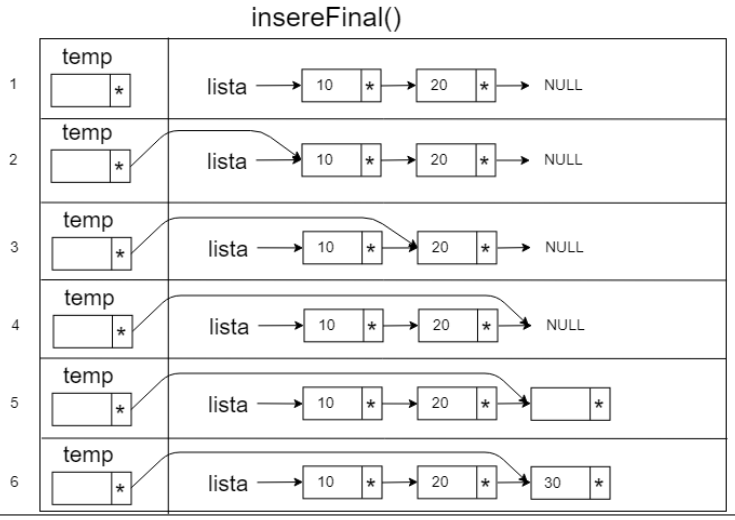
temp→prox continua sendo diferente de *NULL*, então o ponteiro avança novamente.



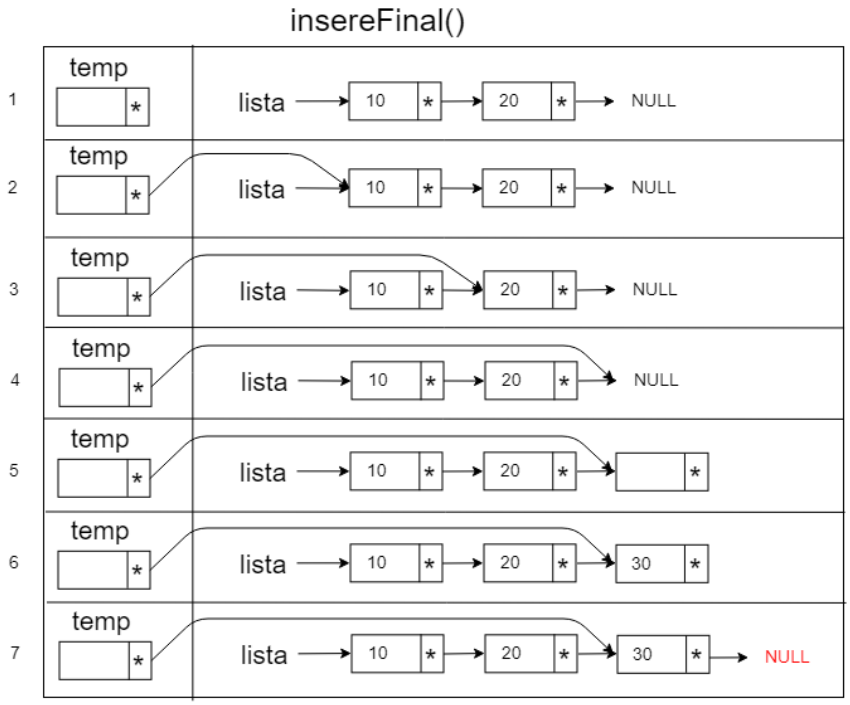
Quando a condição deixar de ser verdadeira, o *loop* se encerrará e *temp* estará agora apontando para o endereço que o último elemento da lista está apontando, ou seja, terá o endereço de *NULL*. Sabendo disso, podemos alocar espaço para o novo elemento.



Após alocar espaço, adicionamos o novo valor:



Vale lembrar que nem sempre que alocarmos memória, esse espaço estará vazio. Provavelmente conterá lixo de memória, que será sobrescrito com o novo valor que você adicionar. Após isso, é necessário fazer com que o novo elemento aponte para *NULL*.



Agora precisamos apenas retornar *temp* para que a lista seja atualizada. Segue uma possível implementação da função:

```
Lista *insereFinal(Lista *lista, int valor) {
    Lista *temp = lista;

    while (temp != NULL)
        temp = temp->prox;

    temp = malloc(sizeof(lista));
    temp->valor = valor;
    temp->prox = NULL;

    return temp;
}
```

Exemplo de utilização:

```
int main() {
    Lista *lista;

    lista = criarLista;

    lista = insereFinal(lista, 16);

    lista = insereInicio(lista, 13);
}
```

5 Imprimindo a lista

Caso seja necessário imprimir a lista, basta percorrê-la da mesma forma que fizemos anteriormente, imprimindo os valores nela contidos. No exemplo da seção 4.2, eu utilizei o laço *while* para percorrer a lista, agora mostrarei um exemplo utilizando o *for*.

```
void imprimir(Lista *lista) {
    Lista *temp;

    for (temp = lista; temp != NULL; temp = temp->prox)
        printf("%d\n", temp->valor);
}
```

6 Verificando se a lista está vazia

Algumas vezes torna-se necessário verificar se uma lista está vazia ou não. A função deve receber a lista e retornar 1 se estiver vazia, e, caso contrário, deverá retornar 0. Sabemos se uma lista está vazia quando seu valor é *NULL*.

```
int vazia(Lista *lista) {
    if (lista == NULL)
        return 1;
    return 0;
}
```

7 Buscando um valor

Com uma função de busca, será possível verificar se um determinado elemento está presente ou não na lista. A função deve receber a lista e o valor que deseja buscar. Caso o valor seja encontrado, é retornado um ponteiro para o nó em que o valor se encontra, caso contrário, deverá retornar *NULL*.

```
Lista *buscar(Lista *lista, int valor) {
    Lista *temp;

    for (temp = lista; temp != NULL; temp = temp->prox)
        if (temp->valor == valor)
            return temp;

    return NULL;
}
```

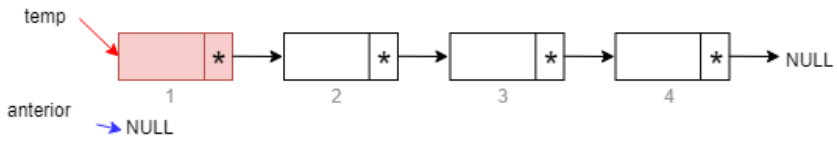
8 Removendo um valor

Para completar, devemos criar uma função que nos permita remover um elemento da lista. A função deverá receber como parâmetro a lista e o valor do elemento que deseja retirar. O retorno deve ser a lista, para que a mesma seja atualizada.

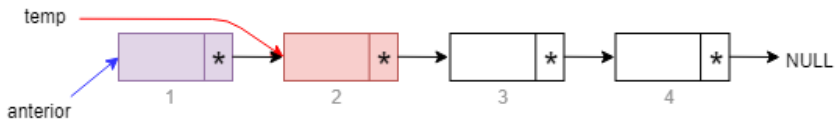
Essa função é um pouco mais complexa. Caso o elemento a ser retirado seja o primeiro, devemos fazer com que o novo valor da lista passe a ser o ponteiro para o segundo elemento, para só então liberarmos a memória. Se o elemento a ser removido estiver no meio da lista, faremos com que o elemento anterior passe a apontar para o próximo, para só então liberarmos a memória. Observe que, no segundo caso, devemos criar um ponteiro para o elemento anterior para podermos acertar o encadeamento da lista. Então utilizaremos dois ponteiros auxiliares: um para percorrer a lista (*temp*) e outro para armazenarmos o valor anterior (*anterior*). *Temp* começará recebendo a lista, e *anterior* receberá *NULL*.

Suponha que queremos remover o elemento que está no terceiro nó da lista. Para isso, devemos percorre-la até chegarmos onde queremos. Mas, não podemos nos esquecer de guardar o endereço do nó anterior. Isso porque devemos fazer com que o ponteiro do nó anterior ao que desejamos remover aponte para o posterior.

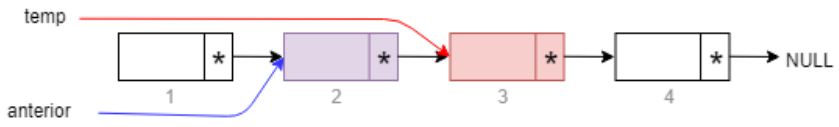
Então *temp* inicia apontando para o primeiro elemento da lista, e *anterior* inicia apontando para *NULL*.



Como o valor de *temp* ainda não é o que desejamos encontrar, *NULL* deverá receber o conteúdo de *temp* (que é o endereço do primeiro elemento da lista) e *temp* irá avançar a lista.

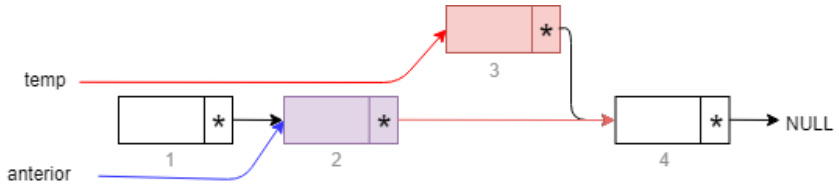


O valor de *temp* ainda não é o que estamos buscando, então *anterior* recebe o valor de *temp* que avança a lista novamente.

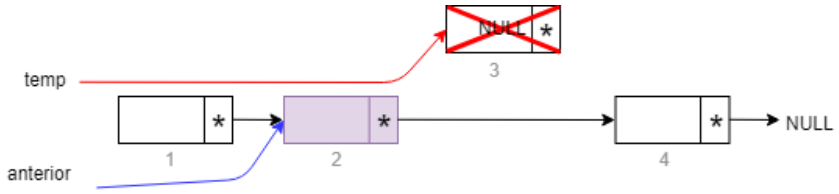


Finalmente chegamos na terceira posição da lista, que é a que contém o nó que desejamos remover.

Para remover o terceiro valor, devemos fazer com que o anterior (que é o segundo) aponte para o nó posterior a ele (que é o quarto).



Agora sim podemos apagar o terceiro nó sem que percamos a referência ao restante da lista.

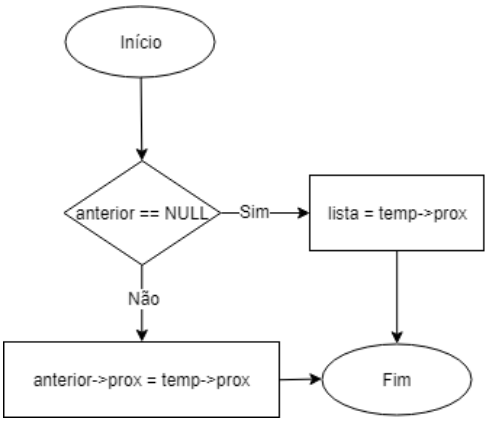


Devemos também colocar uma condição de saída, caso a lista esteja vazia ou valor não seja encontrado.

Sabamos que *NULL* indica o final de uma lista. Se a lista estiver vazia, ela já será nula logo no primeiro nó. Nesses dois casos, devemos encerrar a função e retornar a lista.

Caso não entre nessa condição, significa que o valor foi encontrado. Então, iniciaremos a remoção. Para isso, devemos verificar se o elemento está no início ou no final da lista. Então, basta verificar se o anterior é NULL. Isso porque nós inicializamos o ponteiro anterior com NULL, e se o valor estiver no primeiro nó não irá satisfazer a condição do while, então o loop não iniciará e o valor de temp não será alterado.

Dentro deste condicional, a lista receberá o próximo valor, que será NULL. Caso não seja, significa que a remoção ocorrerá no meio da lista. Nesse caso, o nó que anterior está apontando, deverá apontar para o nó seguinte ao que queremos retirar da lista, e assim, removendo-o. Observe o fluxograma:



Agora, como já acertamos o encadeamento da nossa lista, podemos liberar a memória que o elemento que removemos estava ocupando (*temp* está apontando para ele).

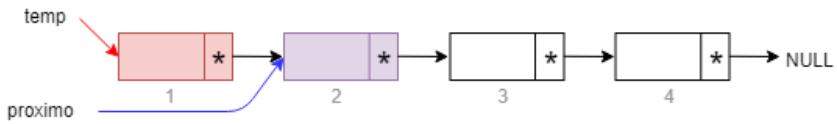
Possível implementação da função:

```
Lista *remover(Lista *lista, int valor) {
    Lista *anterior = NULL;
    Lista *temp = lista;

    while (temp != NULL && temp->valor != valor)
        anterior = temp;
        temp = temp->prox;
    }

    if (temp == NULL) return lista;

    if (anterior == NULL)
        lista = temp->prox;
    else
        anterior->prox = temp->prox;

    free(temp);

    return lista;
}
```

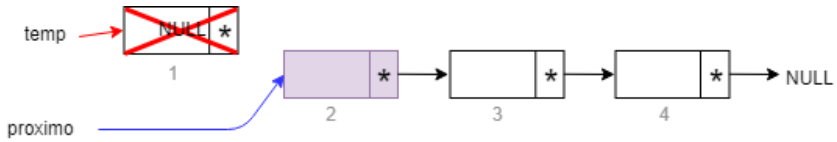
9 Excluindo uma lista

Essa função deverá passar por cada elemento da lista, liberando-os. É importante observar que devemos guardar o endereço para o próximo elemento, pois se liberarmos a memória antes disso, não teríamos como acessá-la pois não possuímos mais o endereço da mesma. Pois estaríamos tentando acessar o elemento anterior a ela, que não existe mais. Então, utilizando uma lógica semelhante a de remover um elemento(seção 8), nós iremos apagar a lista inteira.

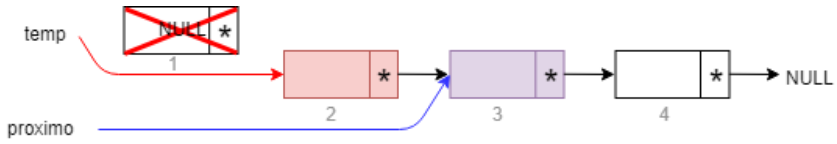
Então *temp* inicializará recebendo o endereço do primeiro nó., e *proximo* receberá o do posterior.



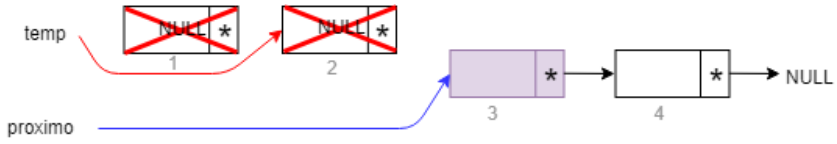
Após isso, o endereço que *temp* está apontando, será liberado.



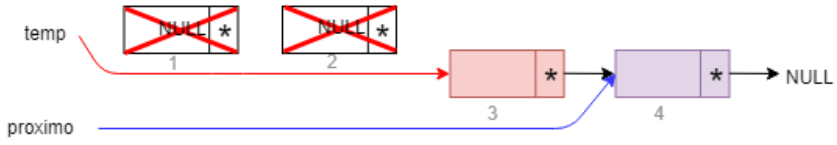
Então, *temp* recebe o conteúdo de *proximo*(que é o endereço do próximo nó) e *proximo* recebe temp→prox.



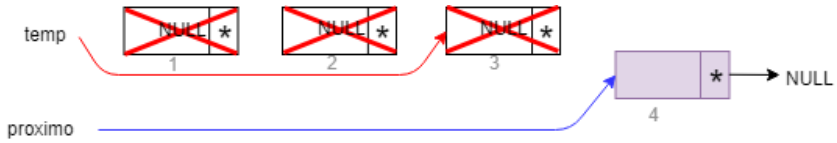
E o endereço que *temp* está apontando é liberado.



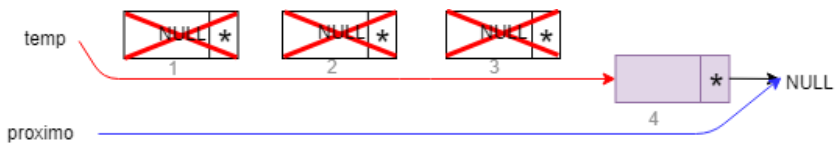
E novamente *temp* recebe o conteúdo de *proximo* e *proximo* recebe temp→prox.



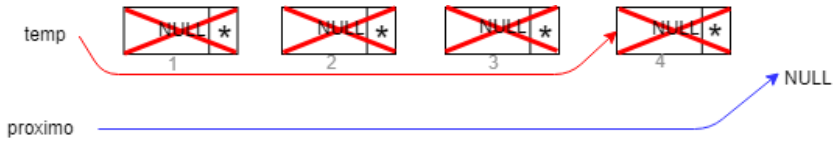
Então liberamos a memória que *temp* está apontando.



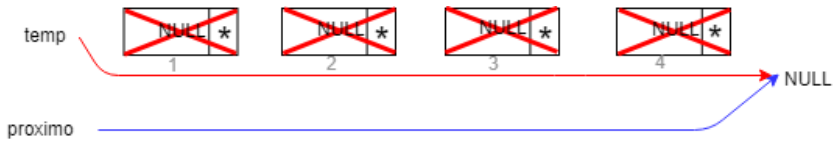
temp recebe o conteúdo de *proximo* e *proximo* recebe temp→prox.



A memória que *temp* está apontando é liberada.



temp recebe o conteúdo de *proximo*



Assim que *temp* recebe o conteúdo de *prox*, a condição para que o *loop* continue rodando passa a ser falsa, pois *temp* agora é *NULL*, e o *loop* irá encerrar.

Possível implementação da função:

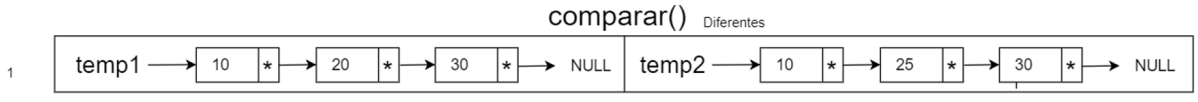
```
void liberar(Lista *lista) {
    Lista *temp = lista;

    while (temp != NULL) {
        Lista *proximo = temp->prox;
        free(temp);

        temp = proximo;
    }
}
```

10 Comparando duas listas

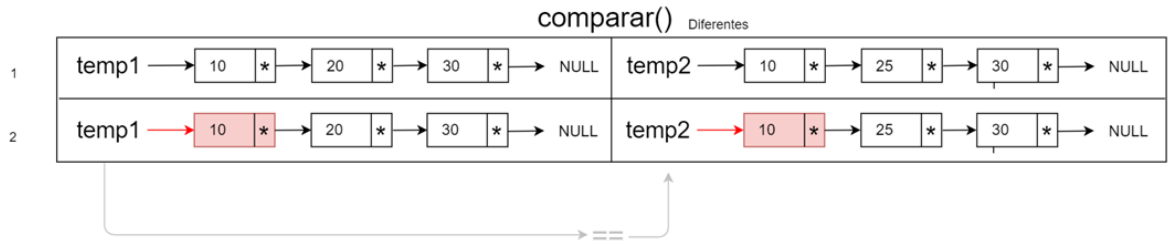
Essa função é semelhante a de encontrar algum valor especifico, mas, aqui estaremos verificando se os valores das duas listas são idênticas ou não.

A função receberá como argumento as duas listas que desejamos comparar, e utilizará dois ponteiros auxiliares: *temp1* e *temp2*. Utilizaremos um laço para a verificação, e caso um valor seja diferente, já poderemos encerrar a função, pois basta uma diferença para que deixem de ser iguais. Então se forem iguais, a função deverá retorna 1, caso contrário, o retorno será 0.

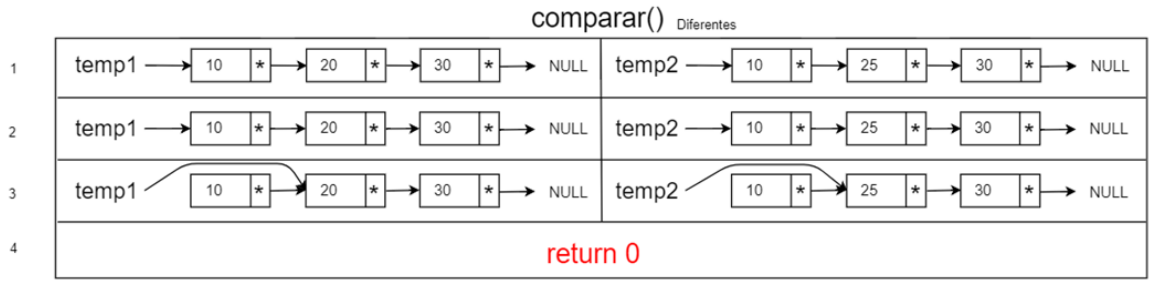
Ambas variáveis auxiliares servirão para que possamos percorrer toda a lista sem que se perca a referência ao primeiro elemento, então elas se inicializam recebendo a *lista1*, e a *lista2*.



Após isso, a comparação é feita. Como podemos ver na imagem, os valores são iguais. Então o *loop* continua.

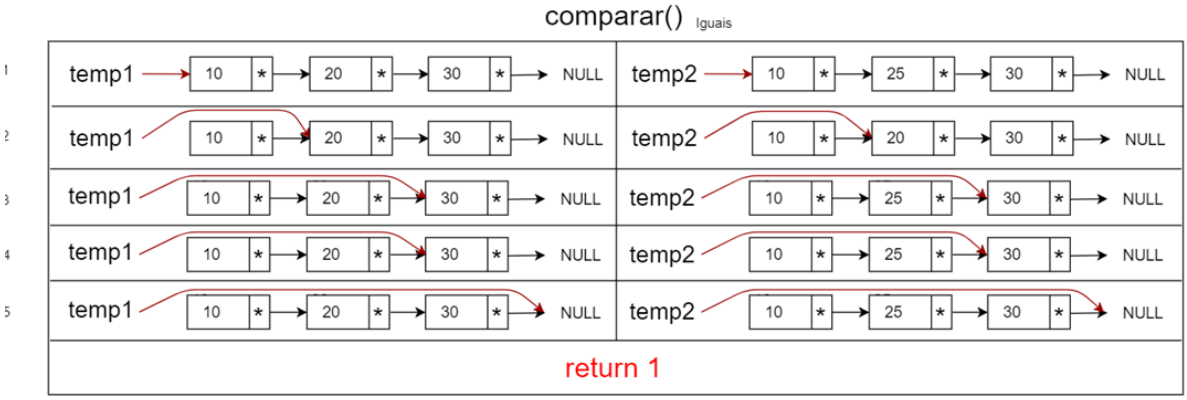


Ambos os ponteiros avançam a lista e a comparação é feita novamente. Mas agora, como podemos notar, os valores são diferentes. Então a função retornará 0 e será encerrada.

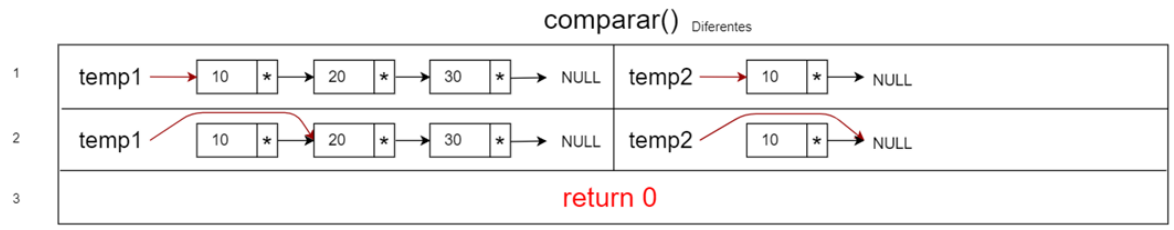


Segue outros dois exemplos:

Quando ambos chegam em *NULL*, significa que as duas listas são iguais, então a função se encerra e retorna 1.



Se apenas um deles chegar em *NULL*, significa que são diferentes.



Possível implementação da função:

```
int comparar(Lista *lista1, Lista *lista2) {
    Lista *temp1 = lista1, *temp2 = lista2;

    while (temp1 != NULL && temp2 != NULL) {
        if (temp1->valor != temp2->valor)
            return 0;

        temp1 = temp1->prox;
        temp2 = temp2->prox;
    }

    return 1;
}
```