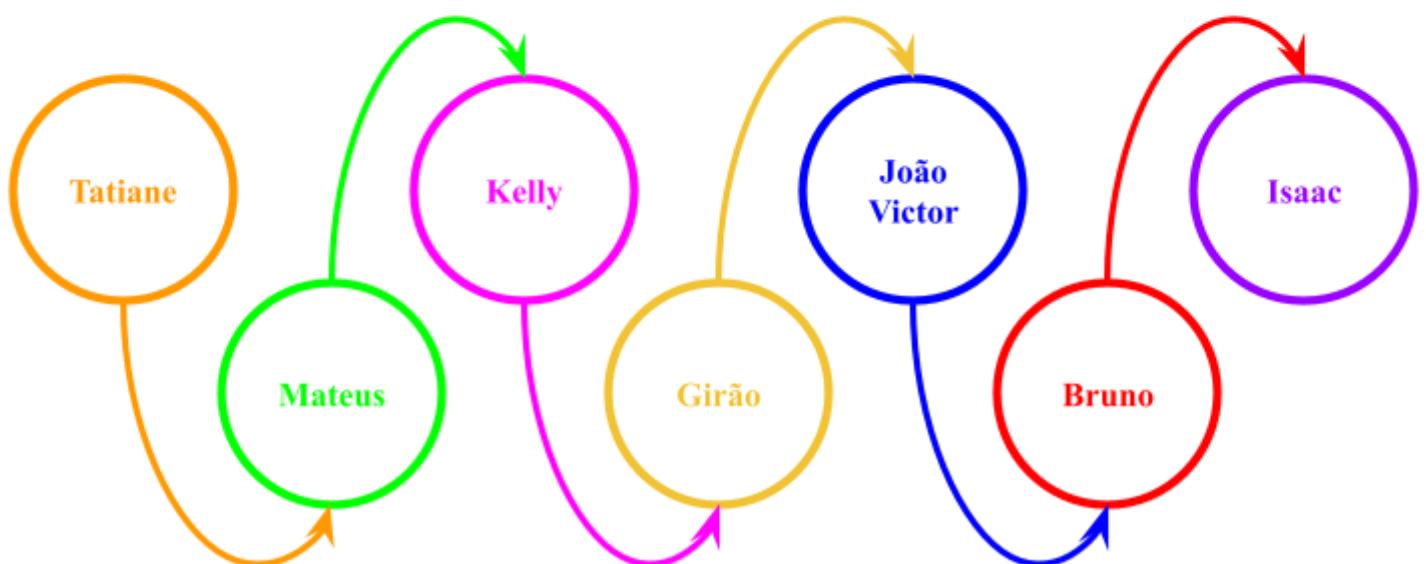


ESTRUTURA DE DADOS

UMA ABORDAGEM GRÁFICA

DO LABORATÓRIO DE ESTRUTURA DE DADOS



UNIVERSIDADE
FEDERAL DO CEARÁ

SUMÁRIO:

Sumário:	1
Estrutura de dados: o que é? Para que servem?	3
Capítulo 1: Introdução, registros, ponteiros e o conceito de No:	4
1.1 Exercícios	14
Capítulo 2: Lista Encadeada (LE):	16
2.1 Conceito Básico de Lista Encadeada (LE)	16
2.2 Estrutura de uma Lista Encadeada	17
2.2.1 Representação abstrata de uma Lista Encadeada	19
2.3 Implementação de uma lista encadeada	21
2.3.1 Adicionar em uma lista encadeada	22
2.3.1.1 A lista está Vazia	24
2.3.1.2 Adicionar no Início com a lista não vazia	25
2.3.1.3 Adicionar no “meio - fim”	27
2.3.2 Remover em uma lista encadeada	31
2.3.2.1 Remoção do início da lista	33
2.3.2.2 Remoção no “meio - fim”	34
2.4 Exercícios	37
Capítulo 3: Melhorando nossa Lista Encadeada:	40
3.1 Conceito Básicos sobre Complexidade	40
3.1.1 Análise de Algoritmos	40
3.1.2 Lista encadeada com ponteiro no fim	45
3.1.3 Exercícios	52
Capítulo 4: Lista duplamente encadeada	54
4.1 Conceito de lista duplamente encadeada	54
4.2 Estrutura de uma LDE (lista duplamente encadeada)	55
4.3 Implementação de uma LDE	57
4.3.1 Adicionar em uma LDE	57
4.3.1.1 Adicionar com LDE vazia	58
4.3.1.2 Adicionar no início com elementos na LDE	59

4.3.1.3 Adicionar na última posição da LDE	61
4.3.1.4 Adicionar no “meio” da LDE	62
4.3.2 Remover em uma LDE	65
4.3.2.1 Remover no início da LDE	65
4.3.2.2 Remover no fim da LDE	65
4.3.2.3 Remover no “meio” da LDE	65

ESTRUTURA DE DADOS: O QUE É? PARA QUE SERVEM?

Neste livro, iremos aprender de uma forma visual e lógica algumas estruturas de dados básicas. Utilizaremos a linguagem de programação C para todos os exemplos de codificação. Antes de adentrarmos nesse conteúdo tão importante, vamos entender melhor o que é uma estrutura de dados.

Uma **Estrutura de Dados** ou **Tipo Abstrato de Dados (TAD's)** em computação, é um conjunto de dados armazenados em um computador ou dispositivo de armazenamento, organizados de forma que seu processamento seja o mais eficiente possível. Hoje em dia, dificilmente imagina-se algum sistema computacional que não haja a necessidade da utilização de estrutura de dados, assim podemos concluir que é de grande necessidade ter o entendimento sobre diversas estruturas de dados e o seu funcionamento.

Existem vários tipos de estruturas de dados para uma infinidade de problemas específicos. Cada TAD deve ser utilizado buscando solucionar o problema em estudo com a máxima eficácia possível. Ao decorrer dos nossos estudos, iremos entender melhor algumas delas.

Para começar vamos entender a definição formal de TAD's?

- O **Tipo Abstrato de Dado (TAD)** é uma especificação de um conjunto de dados e operações que podem ser executadas sobre esse conjunto de dados.
- As **TAD's** buscam trazer ao programador uma maior abstração no uso de um conjunto de dados. Ou seja, TADs buscam fazer uma representação simplificada de um conjunto de dados de tal forma que o programador consiga operar uma TAD sem necessidade de entender questões complexas sobre como ela foi criada.

Neste livro estudaremos as TAD's: lista, pilha, fila e árvore binária de busca. Para tal, iremos desenvolver estudos sobre as operações de adição, remoção, alteração e busca que serão utilizadas para gerenciar cada uma das TAD's apresentadas. Iremos entender um pouco seus pontos fortes e fracos, assim como suas aplicações em determinados problemas.

CAPÍTULO 1: INTRODUÇÃO, REGISTROS, PONTEIROS E O CONCEITO DE *Nos*

Um **Tipo Abstrato de Dados (TAD)** muito importante para aprender todo o conteúdo deste livro é conhecido como **registro**, em linguagem C é denominado ***struct***. Um ***struct*** é uma TAD que agrupa variáveis em um formato de conjunto, podendo estas variáveis serem homogêneas ou heterogêneas.

O uso de um ***struct*** permite a possibilidade de agrupar dados de diferentes tipos em um único conjunto, denominando-o como uma única variável. Um ***struct*** é definido com o seguinte padrão:

```
1 struct nome_da_sua_struct {
2     //lista das variáveis que pertencem ao conjunto
3 };
4
```

Note que a palavra ***struct*** é uma palavra reservada em linguagem C. Por exemplo, se for preciso armazenar o conjunto de dados: altura, peso e idade de uma pessoa, pode-se criar um ***struct*** chamado Pessoa e agrupar os dados em um único tipo, conforme o exemplo a seguir.

```
1 struct Pessoa{
2     float peso;
3     int idade;
4     float altura;
5 };
```

Após a definição da ***struct*** Pessoa é possível criar variáveis do tipo Pessoa que quando existirem terão peso, idade e altura. Como por exemplo :

```
1 struct Pessoa P1; //cria uma unica pessoa que tem um peso, uma idade e
uma altura
2 struct Pessoa povo[10]; //cria um vetor com 10 pessoas
```

O entendimento e utilização de ***structs*** são imprescindíveis para criação das TAD's que serão apresentadas neste livro, pois a partir do conceito de ***struct*** criaremos ***Nos***.

Podemos considerar um *No* como sendo a principal ferramenta que utilizaremos para construção de listas, pilhas, filas e árvores binárias de busca. Um *No* irá conter sempre os seguintes dados:

1. Os **valores**, que serão armazenados na estrutura da dados estudada. Estes valores dependem sempre da aplicação na qual a estrutura de dados estará sendo utilizada;

Exemplo: uma lista utilizada em um sistema de gerenciamento de uma biblioteca poderia ter valores relacionado a livros, ou seja, uma **lista de livros**, onde **cada elemento da lista** teria dados de um livro: nome, autor, ISBN, etc

2. Um **mecanismo para unir os Nos** em um só conjunto, gerando assim uma única TAD.

Para conseguirmos criar nosso **mecanismo para unir os Nos** precisaremos de dois conceitos de programação muito importantes, os **endereços** e **ponteiros**. Primeiro, vamos entender o que é um **endereço**:

A memória RAM (*Random Access Memory*) de qualquer computador é uma sequência de *bytes*. A posição (0, 1, 2, 3, etc.) que um *byte* ocupa na sequência é o **endereço** do *byte* (apenas lembrando que **1 byte = 8 bits**). Podemos visualizar a memória como um vetor com várias unidades, onde cada unidade seja o espaço de **1 byte**. A Figura 1 apresenta uma memória simplificada bem pequena de **1 megabyte**, ou seja, **1048575 bytes**.

Figura 1: Exemplo de memória simplificada

Endereço	0	1	...	524288	524289	524290	524291	524292	524293	...	1048575
Valor	NULL	SO e IDE para linguagem C		Lixo de memória		Lixo de memória					

Em nosso exemplo, a posição de memória **0** está guardando um valor muito importante para as TADs futuras: o valor **NULL**. Da posição **1** até a posição **524288** estão os dados necessários para o funcionamento do Sistema Operacional (SO), assim como o software que será utilizado para codificar programas em linguagem C, ou seja, uma **IDE - Integrated Development Environment**. Assim, a Figura 1 representa a memória RAM de um computador, nele está sendo executado um SO e uma IDE.

Suponha agora que você irá executar um código em linguagem C. Cada variável de um programa de computador, ocupa um certo número de *bytes* consecutivos na memória deste computador. Uma variável do tipo **char** ocupa **1 byte**. Uma variável do tipo **int** ocupa **4 bytes** e uma variável do tipo **float** ocupa **8 bytes** em muitos computadores. O número exato de bytes de uma variável é dado pelo operador **sizeof**. A expressão **sizeof (char)**, por exemplo, vale **1** em todos os computadores e a expressão **sizeof (int)** vale **4** em muitos computadores.

Cada variável que está na memória tem um **endereço**. Na maioria dos computadores, o **endereço** de uma variável é o **endereço do seu primeiro byte**. Por exemplo, depois das declarações:

```

1 char c = 'f';
2 int i;
3 struct exemplo{
4     int x;
5 };
6 struct exemplo a;

```

As variáveis poderiam ter os seguintes **endereços** em nosso exemplo fictício:

Figura 2: Variáveis na memória

	Variável c				Variável i				Variável a				
Endereço	0	1	...	524288	524289	524290	...	524293	524294	...	524297	...	1048575
Valor	NULL	SO e IDE para linguagem C		'F'	Lixo de memória			Lixo de memória		...	Lixo de memória		

O **endereço** de uma variável é dado pelo operador **&**. Assim, se **i** é uma variável então **&i** é o seu **endereço**. Considerando o exemplo acima, veja o código abaixo:

```

1 printf("%d\n", &i);
2 printf("%d\n", &c);
3 printf("%d\n", &a);

```

O valor informado pelo **printf** da linha 1 resultaria em **5242890** e o valor informado pelo **printf** da linha 2 retornaria **524289**, enquanto o **printf** da linha 3 seria **5242894**. Um exemplo interessante de utilização de endereços é o segundo argumento da função de biblioteca **scanf**, o **endereço** da variável que deve receber um valor lido do teclado, por isso o **scanf** utiliza **&** na passagem de parâmetro:

```

1 int i;
2 scanf("%d", &i);

```

Um ponteiro é um tipo especial de variável que armazena um **endereço**.

Há vários tipos de **ponteiros**: ponteiros para *int*, ponteiros para *char*, ponteiros para *double*, dentre outros. O ponteiro mais importante para nós neste momento é o **ponteiro** para um *struct*. Criamos um ponteiro da seguinte forma:

```
tipo * nome_da_variável;
```

Vamos ver alguns exemplos de variáveis do tipo ponteiro:

Declarando uma variável do tipo **ponteiro** *p* que receberá o endereço de um *int* na memória:

```
int *p;
```

Declarando uma variável do tipo **ponteiro** *p* que receberá o endereço de um *struct* pessoa na memória:

```
struct Pessoa *p;
```

Observação: o nome da variável do tipo **ponteiro** pode ser qualquer um que siga as mesmas regras de criação de variáveis em geral, que já sabemos como criar: primeiro caracter tem que começar com uma letra e os demais caracteres podem ser letras, números e o caractere especial *underline* "_".

Um ponteiro pode ter o valor de endereço **NULL**. A macro **NULL** está definida na biblioteca **stdlib.h** e seu endereço de memória é o valor **0** (zero) na maioria dos computadores (veja a Figura 1) ou em algum outro endereço de memória existente, podendo variar de acordo com o seu sistema operacional.

Após a criação da variável do tipo **ponteiro**, podemos recuperar diversas informações dela. Se uma variável do tipo **ponteiro** tem valor diferente de **NULL**, ela está armazenando um endereço de memória. Neste caso, utilizamos o termo **p** para obter este endereço. Podemos também usar a operação ***p** para obter o valor que está armazenado no endereço de memória que o ponteiro **p** referencia. Veja o exemplo do seguinte bloco:

```
1 int *p;
2 int i = 10;
3 p = &i;
4 printf("%d", p);
5 printf("%d", *p);
```

Nossa memória simplificada agora estaria desta forma:

Figura 3.1: Explorando valores de ponteiros

	Variável P						Variável i						
Endereço	0	1	...	524288	524289	...	524292	524293	...	524296	524297	...	1048575
Valor	NULL	SO e IDE para linguagem C			524293			10			Lixo de memória		

Figura 3: Explorando valores de ponteiros

	Variável P						Variável i						
Endereço	0	1	...	524288	524289	524290	...	524293	524294	...	524297	...	1048575
Valor	NULL	SO e IDE para linguagem C			522490	10			Lixo de memória				

Ou seja, na **linha 1** foi criada a variável do tipo **ponteiro p**, enquanto na **linha 2** foi criada a variável do tipo **valor i** que recebeu o valor **10**. Na **linha 3**, a variável **p** recebe o **endereço** de memória da variável **i**. Assim, o **printf** da **linha 4** retornará o valor **524293** enquanto o **printf** da **linha 5** retornará o valor **10**.

Vamos focar nossa **ATENÇÃO** neste momento em criar **variáveis do tipo ponteiro**, toda vez que dissermos que estamos criando um **ponteiro**: entenda que é uma **variável do tipo ponteiro**. Considere agora o seguinte código:

```

1 #include <stdio.h>
2 #include <stdlib.h>
3 struct Pessoa{
4     float altura;
5     int idade;
6     float peso;
7 };
8 int main(){
9     //como criar um ponteiro que receba o endereço de memória que tenha um struct pessoa?
10    return 0;
11 }

```

Para usarmos um **ponteiro**, precisamos alocar espaço na memória do computador e então guardar na variável **ponteiro** o **endereço** de memória deste espaço alocado. Como faremos isso?

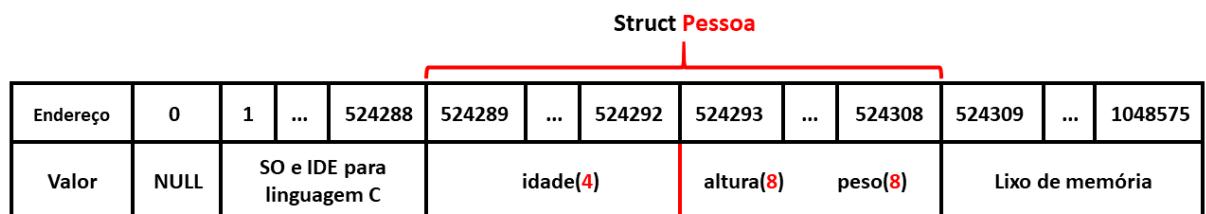
Usando uma função da linguagem C denominada *malloc*. A função *malloc()* **aloca espaço na memória** do tamanho que você desejar passar por parâmetro. A expressão *malloc(10)*, por exemplo, aloca **10 bytes** de memória do computador e retorna o endereço do primeiro byte de memória alocado, enquanto a expressão *malloc(sizeof(char))*, por exemplo, aloca o espaço de memória de **1 byte** e retorna o endereço de memória que deste **1 byte** foi alocado.

A expressão:

`malloc(sizeof(struct Pessoa));`

Irá alocar o espaço de memória de **20 bytes**, necessários para alocar **2 variáveis** do tipo **float** (8 bytes cada) e **1 variável** do tipo **int** (4 bytes), retornando o primeiro endereço de memória dos **20 bytes** alocados.

Figura 3.2: Alocação com o *malloc* e *sizeof*



Desta forma, podemos criar a seguinte codificação na função **main** do exemplo anterior:

```

1 int main(){
2     //Como criar um ponteiro que receba o endereço de memória que tenha um struct pessoa?
3     struct Pessoa *p=malloc(sizeof(struct Pessoa));
4     return 0;
5 }

```

Note que **p** recebeu o endereço de memória alocado pelo *malloc*. Apenas alocamos espaço de memória, mas ainda não colocamos os valores em cada uma das variáveis do **struct Pessoa *p**. Utilizamos o operador → ("seta") para acessar cada uma das variáveis do **struct Pessoa *p** criado:

```

1 int main(){
2     //Como criar um ponteiro que receba o endereço de uma struct pessoa?
3     struct Pessoa *p=malloc(sizeof(struct Pessoa));
4     p->peso=56.5;
5     p->idade=18;
6     p->altura=1.67;
7     return 0;
8 }

```

Pronto! Agora podemos fazer alguns testes. Como será a resposta da execução da função **main** se adicionarmos o seguinte código?

```

1 int main(){
2     //Como criar um ponteiro que receba o endereço de uma struct pessoa?
3     struct Pessoa *p=malloc(sizeof(struct Pessoa));
4     p->peso=56.5;
5     p->idade=18;
6     p->altura=1.67;
7     printf("%d",p);
8     printf("%d",p->idade);
9     return 0;
10 }

```

Se você tiver pensando, que o primeiro **printf** retornará o endereço de memória que está armazenado a **struct Pessoa *p** e o segundo **printf** irá retornar o valor **18**. Você entendeu muito bem todos os conceitos de **registro**, **endereço** e **ponteiro** e pode continuar a leitura do livro. Caso você ainda esteja confuso, recomendamos que refaça mais uma vez a leitura dos conceitos iniciais desta seção até chegar aqui novamente.

A partir de agora iremos juntar nossos conhecimentos para criarmos um **No**. Lembrando que um **No** consiste na junção de um conjunto de valores acrescentado de um campo que permite ligá-los. Vamos rever a definição de **No**:

Um **No** terá sempre os seguintes dados:

1. Os **valores** que serão armazenados na estrutura de dados estudada.
2. Um **mecanismo para unir os Nos** em um só conjunto, gerando assim uma única TAD.

Neste momento, como estamos muito interessados em entender TAD's, ou seja, seu formato e sua importância, iremos simplificar os **valores** que estarão em cada um dos nossos **Nos** para apenas **um** número **inteiro**. Futuramente, iremos mostrar que não importa qual o conjunto de valores iremos armazenar, toda a lógica que estudaremos irá funcionar da mesma forma!

Nosso **mecanismo para unir Nos** será um **ponteiro** que irá apontar para um outro **No**.

Como assim? Considere que iremos criar um *struct No* da forma a seguir:

```

1 #include <stdio.h>
2 #include <stdlib.h>
3
4 struct No{
5     int valor;
6     //aqui ficar o nosso mecanismo para unir Nos
7 };
8 int main(){
9
10    return 0;
11 }
```

Note que nenhum *struct No* existe ainda na memória! Estamos definindo apenas o formato que ele irá ter quando existir. Para existir, precisamos incluir na função **main** o chamado da função *malloc* para alocar espaço na memória para um *struct No*. Vamos chamar esse *struct No* de **NOVO**.

```

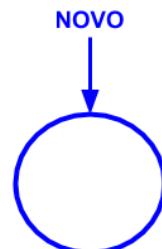
1 int main(){
2     struct no *NOVO=malloc(sizeof(struct no));
3     NOVO->valor=18;
4     return 0;
5 }
```

Podemos imaginar a execução deste código de forma gráfica. Quando realizamos a operação da **linha 2**:

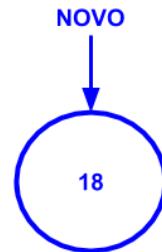
struct No *NOVO = malloc (sizeof (struct No));

A Figura 4 apresenta a forma gráfica do *struct No*. Note que, há um **ponteiro** chamado **NOVO** apontando para o espaço de memória que foi alocado o novo *No*:

Figura 4: Representação gráfica de um novo No



Quando efetuamos a operação: **NOVO→valor = 18;** Podemos visualizar a inclusão do valor 18 no *No* **NOVO** na Figura 5:

Figura 5: Atualização do valor de um No

Agora, vamos adicionar nosso **mecanismo para unir Nos**. Como queremos unir *Nos* seria importante que um *No* apontasse para um outro *No*.

Faremos isso colocando dentro do *struct No*, um **ponteiro** que receberá o **endereço** de um outro *struct No* alocado na memória ou então o valor **NULL** indicando que o ponteiro está apontando para o endereço "vazio". Vamos chamar este ponteiro de **Prox** (apelido carinhoso para **próximo**):

```
struct No *Prox;
```

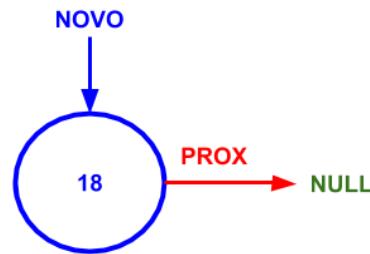
Ou seja, o ponteiro **Prox** armazena o endereço de memória que contém um *struct No*. Como dito, este ponteiro fica dentro de cada *No* quando o mesmo for criado. Portanto, ele deverá ser incluído dentro do *struct No*. Assim, nosso código ficará dessa forma:

```
1 struct No{
2     int valor;
3     struct No *Prox;
4 };
```

Agora podemos adicionar na função **main** um **endereço** para o **ponteiro Prox**. Vamos apontar inicialmente o ponteiro **Prox** para o endereço especial **NULL** até que tenhamos um outro *No* para unir a ele.

```
1 int main(){
2     No* NOVO=malloc(sizeof(No));
3     NOVO->valor=18;
4     NOVO->prox=NULL;
5 }
```

Podemos visualizar o *No* **NOVO** completo após a operação: **NOVO→Prox = NULL;** na Figura 6.

Figura 6: Inicialização de ponteiro NULL

DICA: Para facilitar a criação de um *struct* em nosso código, podemos utilizar uma função da linguagem C denominada *typedef* que nomeia o tipo definido a partir de uma *struct*, simplificando a declaração da *struct* criada. Segue o exemplo do uso de *typedef* para criação da nossa *struct* *No*. Note que estamos "definindo um tipo novo" (*type* = tipo e *def* = acrônimo de *definition* (definição), ou seja, *typedef*). Em nosso exemplo, estamos definindo que o tipo *struct* *No* se chamará *No*.

```

1 typedef struct no{
2     int valor;
3     struct no *Prox;
4 }No;
  
```

Dessa forma, podemos usar a palavra *No* para criar variáveis do tipo *struct* *No* diminuindo a quantidade de palavras que você terá que digitar. Nossa exemplo de criação de *No*, utilizando *typedef*, ficaria desta forma:

```

1 typedef struct no{
2     int valor;
3     struct no *Prox;
4 }No;
5
6 int main(){
7     No* NOVO= malloc(sizeof(No));
8     NOVO->valor=18;
9     NOVO->prox=NULL;
10 }
  
```

Com isso, chegamos ao final deste capítulo! Para garantir que você tenha entendido tudo iremos pedir que faça a lista de exercícios que será repassada a seguir.

1.1 Exercícios

- I. Sabemos que, em uma TAD qualquer, possuímos elementos que se ligam de uma forma lógica com outros elementos. Para que tenhamos uma TAD sem erros, implemente um registro com suas funcionalidades necessárias. O que mais se poderia colocar caso o código esteja relacionado, por exemplo, com um sistema de cartão de crédito?

- II. Inspecione o código abaixo com cuidado. No fim da execução, o valor de x permanece o mesmo ou foi alterado? Caso sim, ele recebeu o número de um endereço de memória? A qual variável pertence o valor de *p, x, a ou k?

```

1 #include<stdio.h>
2 #include<stdlib.h>
3
4 int main(){
5     int x = 1;
6     int *p = &x;
7     *p = x;
8     int a = (*p);
9     int *k = &a;
10    x = a;
11    p = k;
12    a = x;
13    return 0;
14 }
15

```

- III. Observe o código em C abaixo. Depois, descreva o que acontece no código e, por fim, anote a frase que sairá no *printf* do código.

```

1 #include<stdio.h>
2 #include<stdlib.h>
3 typedef struct Aviso_Covid{
4     int coronaASolta;
5     struct Aviso_Covid *fiqueEmCasa;
6 }Corona
7
8 int main(){
9     Corona *laveAsMaos, *useAlcool;
10    laveAsMaos = malloc(sizeof(Corona));
11    useAlcool= malloc(sizeof(Corona));
12    laveAsMaos->coronaASolta = 19;
13    laveAsMaos->fiqueEmCasa = useAlcool;
14    useAlcool->coronaASolta = 100;
15    useAlcool->fiqueEmCasa = NULL;
16    printf("Cuidado com o COVID-%d! Fique %d porcento dentro de casa lendo o livro de ED!", 
17          laveAsMaos->coronaASolta, useAlcool->coronaASolta);
18    return 0;
19 }

```

- IV. Usando ponteiros, faça uma função, em C, que troque os valores de dois inteiros entre si, que serão chamados nos parâmetros ao fazer a chamada da função. A função não retorna nada e tem como parâmetros dois ponteiros para inteiro.
- V. Em uma certa cidade, a prefeitura utiliza um programa para guardar dados de certas ruas. Certo dia, alguns dados desse programa foram **danificados** e **bagunçados** devido a um vírus no computador. Suponha que você é um técnico que deve completar o código abaixo com sua lógica, entendimento de programação e as seguintes informações: os 3 elementos desse programa são os imóveis de uma certa rua. No programa, esses imóveis são

```

1 #include <stdio.h>
2 #include <stdlib.h>
3 ... struct ImoveisDaRua{
4     char *complemento;
5     char imovelComercial;
6     ImoveisDaRua *prox;
7     ...
8 } ... ;
9
10 int main(){
11     IDR * = malloc(sizeof( ... ));
12     IDR apartamento17 = malloc( ... );
13     ... ->imovelComercial = "N";
14     ...
15     mercantill-> = 1;
16     mercantill complemento = "Casa Comercial, 521 m2";
17     apartamento17-> = "N";
18     casa342->numero = ... ;
19     ... ->numero = 17;
20     apartamento17->complemento = "Predio, 4 Andares, 182 m2.";
21     mercantill->imovelComercial = "...";
22     casa342->complemento = "Casa, 284 m2.";
23     ...
24     casa342->prox = ... ;
25     ... ->prox = mercantill;
26     mercantill = NULL;
27     return 0;
28 }
```

ligados usando o ponteiro de dentro do registro "ImoveisDaRua", na seguinte ordem, Casa->Apartamento->Mercantil->Fim da rua.

CAPÍTULO 2: LISTA ENCADEADA (LE)

2.1 Conceito Básico de Lista Encadeada (LE)

Em estrutura de dados, podemos representar uma lista por meio de *Nos* encadeados, a qual chamamos de Lista Encadeada. Neste capítulo, aprenderemos melhor o que é esta lista e como utilizar *Nos* encadeados para representá-la.

Nossas listas encadeadas possuirão um **início** e um **fim**. Seguindo as propriedades de uma lista encadeada, temos que nossa lista nunca possuirá espaços vazios entre os elementos. Portanto, uma lista encadeada funciona exatamente como uma lista do mundo “real”, ou seja, devemos sempre adicionar um elemento logo após o outro. Para que um programador possa operar uma lista encadeada é importante que ele tenha pelo menos as opções de adicionar, remover, alterar e buscar elementos.

Agora que temos o entendimento básico sobre listas encadeadas, vamos compreender melhor como elas funcionam.

As Figuras 7 e 8 apresentam exemplos de listas do mundo real.

Figura 7: Exemplo de uma lista dos monitores do LED

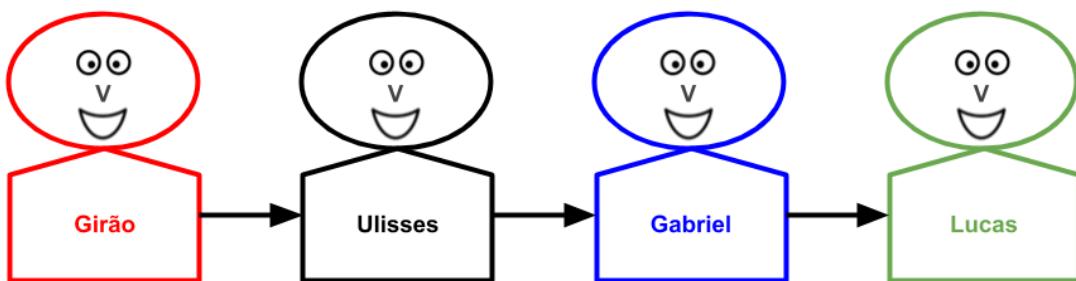
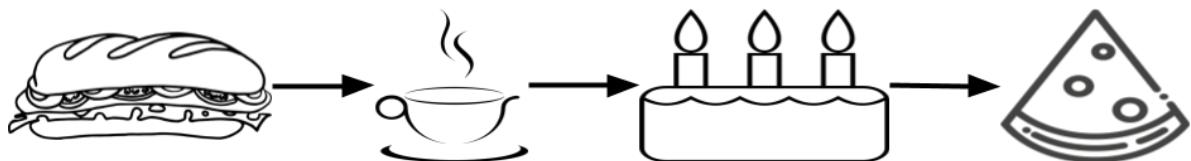


Figura 8: Exemplo de uma lista de alimentos



Uma **Lista Encadeada** é uma forma de representar uma lista do mundo “real” em um programa de computador, para tal, utilizamos um conjunto de *Nos* organizados sequencialmente, ou seja:

Um No logo após o outro, de forma a não permitir espaços vazios entre dois Nos.

A Figura 9 e 10 apresentam como representamos a Figura 7 e 8 no formato de Listas encadeadas.

Figura 9: Representação de uma lista encadeada no computador

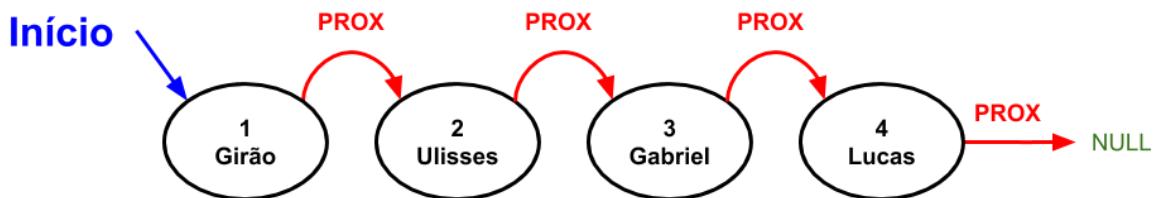
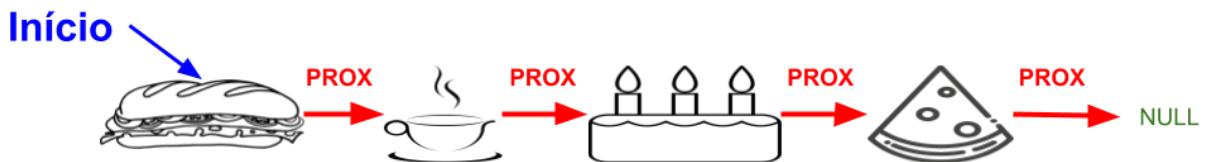


Figura 10: Exemplo de uma lista encadeada de alimentos



Como aprendemos no Capítulo 1, cada **No** guarda seus **valores** e um **mecanismo para unir Nos**, o nosso famoso **ponteiro**, que na Lista Encadeada será utilizado para **unir o próximo No da lista**. Já sabemos agora, que este mecanismo é um **ponteiro** denominado **Prox** que aponta para o endereço de memória do **próximo No** da lista. Note também que é **muito importante saber onde está o Início da lista**, pois é a partir dele que você consegue acessar todos os **Nos** de uma **Lista Encadeada**. Na próxima seção iremos aprender como construir uma **Lista Encadeada**.

2.2 Estrutura de uma Lista Encadeada

Nesta seção, iremos entender melhor como é a estrutura de uma lista encadeada, aprenderemos como construí-la e quais operações são importantes codificar em nossa lista. Vamos começar relembrando um importante conceito: **ponteiro**.

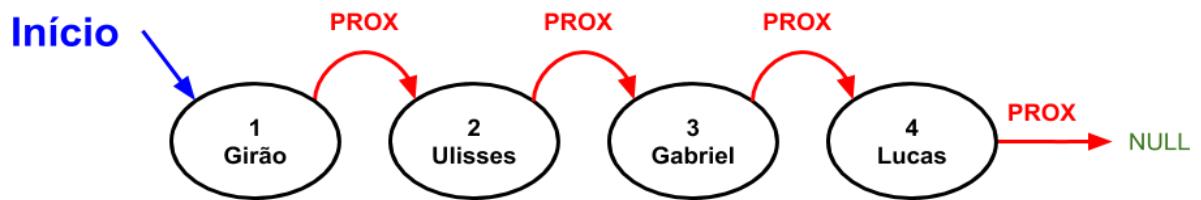
Um ponteiro é um tipo especial de variável que armazena um **endereço**.

A declaração de uma variável do tipo ponteiro, em linguagem C, é feita da seguinte forma:

```
tipo *nome_da_variável;
```

Vamos visualizar mais uma vez nosso exemplo de lista encadeada de monitores do LED na Figura 11.

Figura 11: Exemplo de uma lista de monitores

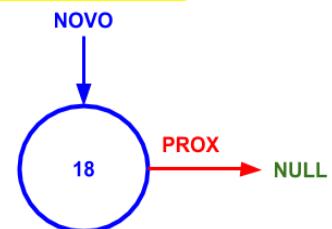


Note que o Início é um ponteiro que recebe o **endereço de memória onde está o primeiro nó da lista** (1 - Girão). Vamos relembrar como criar *Nos* simples que possuam apenas um valor inteiro como dado.

Exemplo da definição de um No e sua alocação de memória:

```
1 typedef struct No{
2     int valor;
3     struct No *Prox;
4 }No;
5
6 int main(){
7     No* NOVO= malloc(sizeof(No));
8     NOVO->valor=18;
9     NOVO->prox=NULL;
10 }
```

Figura 12: Exemplo visual do ponteiro NOVO apontando para um No quando ele foi alocado na memória:



Como descrito no Capítulo 1, podemos criar um ponteiro do tipo **No**, que poderá receber o endereço de um **No** alocado na memória ou o valor especial **NULL**. Vamos criar um ponteiro **No** denominado **início** e atribuir o endereço de memória **NULL**, desta forma podemos dizer que o ponteiro **início** está "vazio".

```
No *início = NULL;
```

A Figura 13 apresenta como nosso ponteiro **início** pode ser representado visualmente, desta forma podemos visualizar que a nossa **Lista Encadeada** está "vazia".

Figura 13: Representação visual de uma lista encadeada vazia

Início → NULL

É muito importante notar que o ponteiro `início` é da **Lista Encadeada**, ou seja, só existe um por **Lista Encadeada**. Diferente do que acontece com o ponteiro `prox`, pois existe um ponteiro em cada `No`. Por isso, dizemos que o ponteiro `prox` é de um `No` e por este motivo ele está declarado dentro do `struct No`. Note também que, em uma Lista Encadeada o ponteiro `início` deverá sempre existir e guardar o endereço do primeiro `No` da lista, quando este `No` existir ou `NULL`, se a lista for vazia. Assim, podemos criar nossa lista encadeada com dados inteiros e com uma variável ponteiro `início` global que por padrão estará inicialmente vazia como no código abaixo.

```

1 #include <stdio.h>
2 #include <stdlib.h>
3
4 typedef struct No{
5     int valor;
6     struct *No prox;
7 }No;
8
9 NO*inicio=NULL;
10
11 int main(){
12     return 0;
13 }
```

2.2.1 Representação abstrata de uma lista encadeada

A partir de agora é necessário ter entendido muito bem o conceito de `No` (caso ainda não tenha compreendido totalmente o conceito indicamos a releitura novamente do Capítulo 1), podemos representar um `No` de uma forma mais **Abstrata**, ou seja, considerando que cada `No` pode ter qualquer conjunto de dados.

Abstração é a habilidade de concentrar nos aspectos essenciais de um contexto qualquer, ignorando características menos importantes.

Desta forma, podemos dizer que todo `No` pode possuir um **conjunto de variáveis** que representam os dados de um `No`, podendo este conjunto representar uma infinidade de dados existentes no mundo real. Como exemplo, podemos citar as listas apresentadas neste livro: lista de pessoas, lista de monitores do LED, lista de alimentos, etc.

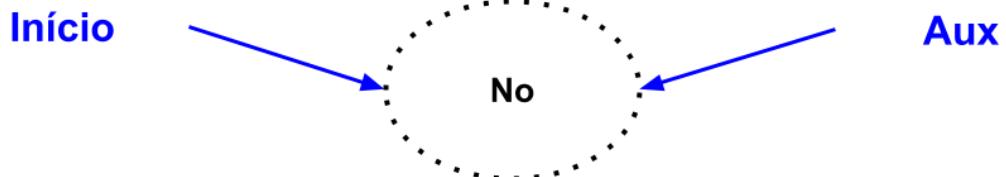
Iremos apresentar abaixo o conjunto de variáveis como elemento, sempre junto ao campo “elemento” teremos o ponteiro `prox` que armazena o endereço do próximo `No` da lista, sendo assim o responsável pelo encadeamento da lista.

Figura 14: Representação de um `No` qualquer de uma lista encadeada



Ainda a respeito do conceito de *No*, para completar nossa lista, teremos dois ponteiros para *Nos* especiais: o **ponteiro início** que será responsável por indicar qual o primeiro *No* da lista e um segundo ponteiro que será **eventualmente** utilizado para percorrer a lista, denominado **Aux** (apelido carinhoso para auxiliar). Vejamos a representação gráfica desses *Nos* auxiliares:

Figura 15: Representação de ponteiros auxiliares do tipo No



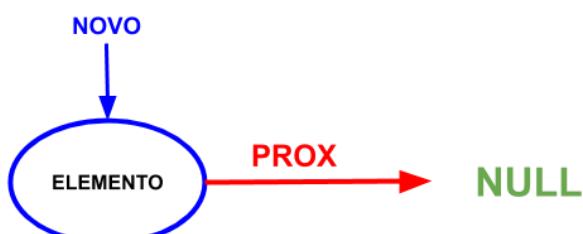
Para finalizar a parte estrutural de uma lista encadeada, veremos como representaremos um "novo" *No* quando este for inserido na lista.

Sempre que for necessário a inserção de um conjunto de dados novo na lista encadeada será preciso a criação de um "novo" *No*. Para tal, iremos seguir os passos abaixo:

1. Criar um ponteiro denominado **NOVO** e alocar espaço de memória com a função `malloc` para este "novo" *No*;
2. O **elemento** receberá os novos dados a serem inseridos, ou seja, cada variável receberá seu respectivo valor;
3. Por fim, o ponteiro **prox** será inicialmente atribuído para vazio, em linguagem de programação C, utilizaremos a palavra chave **NULL**.

Para finalizar esta seção, vamos relembrar nosso exemplo de "novo" *No*, onde o elemento do *No* está bem simplificado e é apenas um valor inteiro (Seção 2.2).

Figura 16: Representação de um No NOVO



2.3 Implementação de uma Lista Encadeada

Para criação da lista encadeada, nosso foco é base para a criação de *Nos*, que no código é o `struct No`. Vamos continuar pensando de forma simplificada, e imaginar

```

1 typedef struct no{
2     int valor;
3     struct no*Prox;
4 }No;

```

que ele será composto inicialmente por um valor inteiro e o nosso ponteiro **prox**. Seguindo essa ideia, considere o seguinte bloco de código:

Como já mencionamos, em uma lista sempre existe um **ponteiro início** que começará sempre indicando que a lista está "vazia". Também é interessante para nós sabermos quantos elementos existem na lista, portanto vamos adicionar uma variável denominada **tam** que irá contar a quantidade de *Nos* que há na lista. Como nossa lista sempre inicia vazia, o contador **tam** começa com o valor **0** (zero).

A Figura 17 apresenta de maneira visual a nossa lista quando ela está vazia.

Figura 17: Lista vazia

```

1 No*inicio=NULL;
2 int tam=0;

```



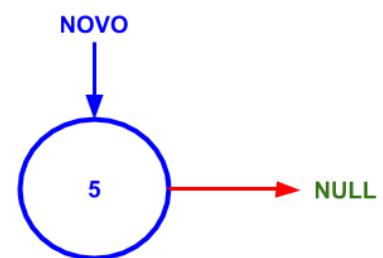
Vamos tentar adicionar um *No* em nossa lista? Já sabemos criar um *No* com auxílio da função **malloc** e guardar seu endereço de memória. O código abaixo e a Figura 18 apresentam como criamos um ponteiro **NOVO** que receberá o endereço definido pela função **malloc** para a alocação de memória de um *No*.

Figura 18: Representação gráfica do no Novo

```

1 int main(){
2     No*NOVO=(No*)malloc(sizeof(No));
3     NOVO->valor=5;
4     NOVO->Prox=NULL;
5     return 0;
6 }

```



Porém, como seria se precisarmos de vários *Nos*? Por exemplo, se precisarmos inserir 1000 *Nos* em uma lista? Poderíamos declarar explicitamente 1000 ponteiros (**NOVO1**, **NOVO2**, **NOVO3**... **NOVO1000**). Não é uma ideia muito interessante. Não é? Note também que ao criarmos cada *No*, eles ficaram separados, não há nenhuma codificação que une eles. Então, como podemos juntar os *Nos* todos em uma só lista? Ou seja, como vamos apontar os ponteiros **prox** de cada *No* para uní-los? Na próxima seção apresentamos uma ideia muito esperta para adicionar *Nos*.

2.3.1 Adicionar em uma lista encadeada

Para adicionarmos dados em uma lista encadeada, podemos adicionar por posição, valor ou qualquer outra forma que você consiga implementar, **desde que não se crie espaços entre dois Nos.**

Estudaremos agora como adicionar dados considerando qual **posição** os dados serão inseridos, mas futuramente estudaremos também como adicionar *Nos* em uma lista, considerando o valor dos seus elementos. Para os exemplos a seguir, iremos considerar novamente nossa lista encadeada onde cada *No* contém apenas um valor inteiro como seu elemento.

Nosso primeiro passo será criar uma função denominada **adicionar**, que receberá dois parâmetros:

1. Uma variável do tipo **int**, denominada **valor**, que será atribuída a variável "valor" do *No* a ser criado;
2. Uma variável do tipo **int**, denominada **pos**, que representará a **posição** que *No* ficará na lista.

Em resumo, essa função irá criar um "novo" *No*, adicionar o dado do seu elemento através da variável **valor** recebida por parâmetro e **fazer a ligação deste "novo" No com a nossa lista.**

Como estamos adicionando **Nos** por posição passada por parâmetro, temos que tomar o cuidado de verificar sempre em qual local da lista esse elemento deverá ser adicionado. Para tal, utilizaremos condições para essa verificação importante:

Como garantir que a posição a ser adicionada é válida? Precisamos verificar se o valor da variável **pos** que o programador passar por parâmetro tem o valor maior que **0** (zero), pois não existe posição negativa na contagem de *Nos* em uma lista. Também é importante verificar se o valor da variável **pos** garante nossa regra de sequência obrigatória de *Nos*.

Por exemplo, se nossa lista tem 3 *Nos*, então nossa variável **tam** está com o valor 3. Portanto, o programador não pode pedir para adicionar um elemento na posição 6, pois neste caso estaríamos criando 2 espaços sem *Nos* em nossa lista (posição 4 e 5) e assim estaríamos ferindo a única regra para que nossa TAD seja uma **Lista Encadeada**:

Um *No* logo após o outro, de forma a não permitir espaços vazios entre dois *Nos*.

É importante mencionar que nossa contagem por posição sempre começa da posição **0** (zero). Vamos codificar essa verificação da seguinte forma, verificamos se a posição é maior ou igual que zero e se essa posição é menor ou igual o tamanho da lista:

```

1 void adicionar(int valor, int pos){
2     if(pos>=0 && pos<=tam){
3         //aqui ficará o nosso código para adicionar um No
4     }
5 }
6
7 }
```

Agora que temos a garantia que a posição estará correta, podemos criar um novo *No* e adicioná-lo em nossa lista. Vamos considerar os seguintes casos para que esta adição aconteça:

1. A lista está vazia;
2. A lista não está vazia e queremos adicionar na posição 0, ou seja, no início;
3. A lista não está vazia e não queremos adicionar no início, ou seja, a variável *pos* está com um valor entre 1 e *tam*. Vamos chamar esse intervalo de valores de "meio e fim" da lista.

Nosso código respeitando essas condições pode ser escrito da seguinte forma:

```

1 void adicionar(int valor, int pos){
2     if(pos >= 0 && pos <= tam){
3         /* verificamos se a posição é válida, sendo maior ou igual que zero e menor
4             ou igual ao tamanho atual da lista */
5         ...
6         if(inicio == NULL){ //verificamos se a nossa lista ainda não foi iniciada
7             ...
8         }else if(pos == 0){ //verificamos se o novo no vai ser o inicio da nossa lista
9             ...
10        }else{ //em ultimo caso ficará no fim ou no meio da lista
11            ...
12        }
13        tam++; //acrescentamos um ao tamanho da lista
14    }
15 }
```

Agora que todas as condições foram criadas, precisamos criar o ponteiro **NOVO** que receberá a alocação de memória de um *No* (linhas de 3 a 5) do código abaixo.

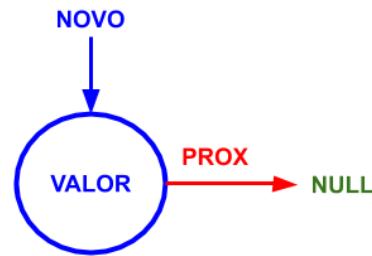
```

1 void adicionar(int valor, int pos){
2     if(pos>=0 && pos<=tam){
3         No*NOVO=malloc(sizeof(No)); //alocamos na memória o espaço para o No NOVO
4         NOVO->valor=valor; //atribuimos o valor passado para função a variável valor do NOVO
5         NOVO->Prox=NULL; //atruimos um valor nulo para o ponteiro Prox desse No
6         if(inicio==NULL){
7             ...
8         }else if(pos==0){
9             ...
10        }else{
11            ...
12        }
13    }
14    tam++;
15 }
16

```

A Figura 19 apresenta como podemos visualizar a criação do *No*.

Figura 19: Representação gráfica pós alocação do No NOVO (linhas 3 a 5)



Após alocamos o *No NOVO* na memória e atribuirmos os valores corretos as suas variáveis, **podemos analisar em qual posição o No NOVO irá ser adicionado**. Para todos os casos abaixo, considere que o programador irá inserir um valor e uma posição que satisfaça a condição de viabilidade da lista. Portanto, o *No NOVO* sempre será criado corretamente.

2.3.1.1 A lista está vazia

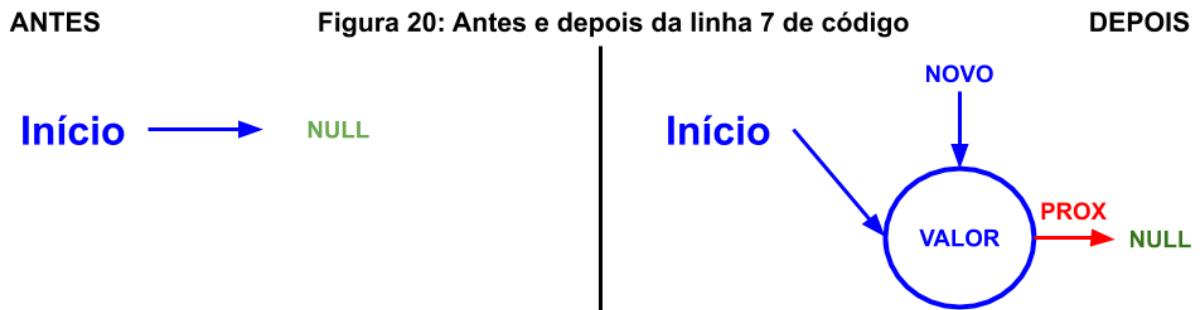
Caso o ponteiro *início* esteja com o valor *NULL*, ou seja, a lista está vazia, o *No NOVO* construído será o primeiro *No* a ser adicionado na lista. Logo, para encaixá-lo devemos fazer o ponteiro *início* receber o endereço do ponteiro *NOVO* (linha 7).

```

1 void adicionar(int valor, int pos){
2     if(pos>=0 && pos<=tam){
3         No*NOVO=malloc(sizeof(No)); //alocamos na memória o espaço para o No NOVO
4         NOVO->valor=valor; //atribuimos o valor passado para função a variável valor do NOVO
5         NOVO->Prox=NULL; //atruimos um valor nulo para o ponteiro Prox desse No
6         if(inicio==NULL){
7             inicio=NOVO;
8         }else if(pos==0){
9             ...
10        }else{
11            ...
12        }
13    }
14    tam++;
15 }
16

```

A Figura 20 apresenta como podemos visualizar a operação da linha 7.

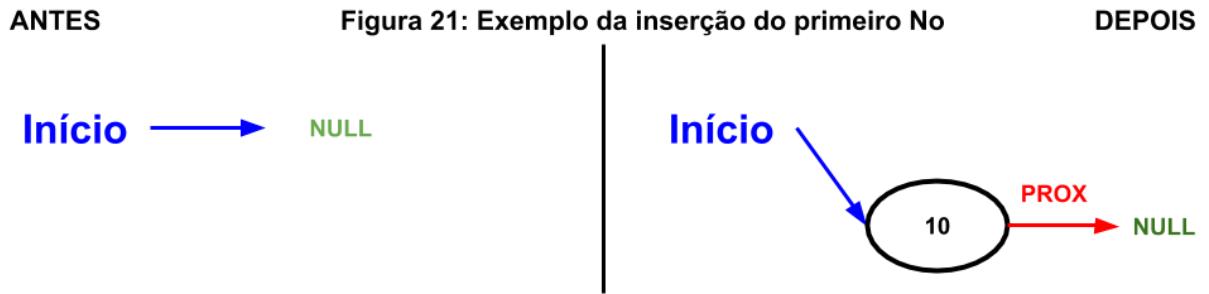


Suponha que o programador tenha passado por parâmetro o valor igual a **10** e a posição igual a **0** (linha 26). Nossa código completo ficará da seguinte forma:

```

1 #include <stdio.h>
2 #include <stdlib.h>
3
4 typedef struct no{
5     int valor;
6     struct no*Prox;
7 }No;
8
9 int tam=0;
10 No*inicio=NULL;
11
12 void adicionar(int valor, int pos){
13     if(pos>=0 && pos<=tam){
14         No*NOVO=malloc(sizeof(No));
15         NOVO->valor=valor;
16         NOVO->Prox=NULL;
17         if(inicio==NULL){
18             inicio=NOVO;
19         }else if(pos==0){
20             /...
21         }else{
22             //..
23         }
24         tam++;
25     }
26 }
27 int main(){
28     adicionar(10,0);
29     return 0;
30 }
```

É importante notar que, o **NOVO** é uma variável do tipo ponteiro criada LOCALMENTE, ou seja, ele só existirá dentro da função adicionar quando esta função for chamada. Porém, isso não é um problema, uma vez que o ponteiro **Início** é GLOBAL. Portanto, continuará apontando para o endereço de memória alocado que o ponteiro **NOVO** estava apontando ao final das operações da função adicionar. Assim, após a execução deste código podemos imaginar nossa lista de forma visual como apresentada na Figura 21.



2.3.1.2 Adicionar no início com a lista não vazia

Considere a lista não vazia da Figura 21. Suponha agora que iremos adicionar mais um valor inteiro novamente na **posição 0** (zero), ou seja, no início da lista, sendo que esta **lista não está vazia**. Novamente, iremos criar o **No NOVO** e agora podemos imaginar que o **No NOVO** está a frente da lista como na Figura 22, pois, ele deverá ocupar a primeira posição da lista.

Figura 22: Inserção no início com lista não vazia

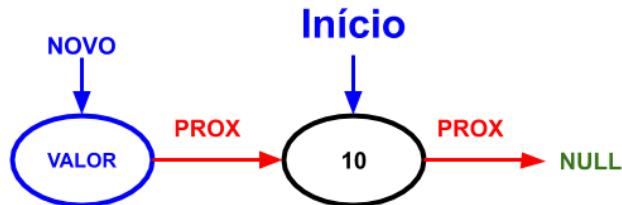


Nosso objetivo é fazer o ponteiro **prox** do **No NOVO** aponte para o **No** que é referenciado pelo ponteiro **início**. Podemos fazer isso com a seguinte operação:

$$\text{NOVO} \rightarrow \text{Prox} = \text{início};$$

A Figura 23 apresenta a lista encadeada após a operação acima.

Figura 23: Atualização do ponteiro prox do NOVO



Em seguida, é necessário a atualização do **início**, visto que, o **No NOVO** agora deve ocupar a posição **0**. Ou seja, o ponteiro **início** deve receber o endereço de memória do **No NOVO**:

$$\text{início} = \text{NOVO};$$

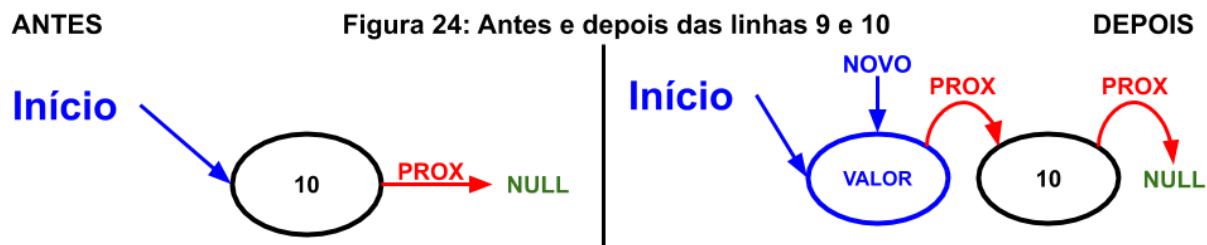
O código completo ficará da seguinte forma (linhas 9 e 10 atualizadas):

```

1 void adicionar(int valor, int pos){
2     if(pos>=0 && pos<=tam){
3         No*NOVO=malloc(sizeof(No));
4         NOVO->valor=valor;
5         NOVO->Prox=NULL;
6         if(inicio==NULL){
7             inicio=NOVO;
8         }else if(pos==0){
9             NOVO->Prox=inicio;
10            inicio=NOVO;
11        }else{
12            //...
13        }
14    }
15    tam++;
16 }
17

```

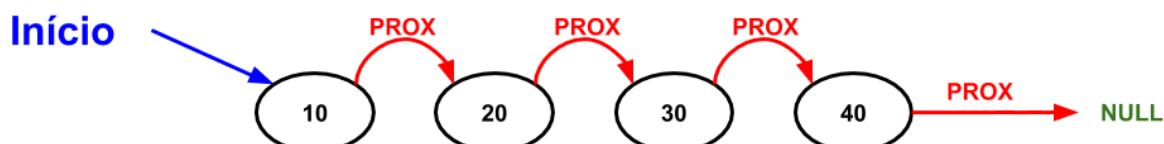
A Figura 24 apresenta como podemos visualizar as operações da linha 9 e 10.



2.3.1.3 Adicionar no “meio - fim”

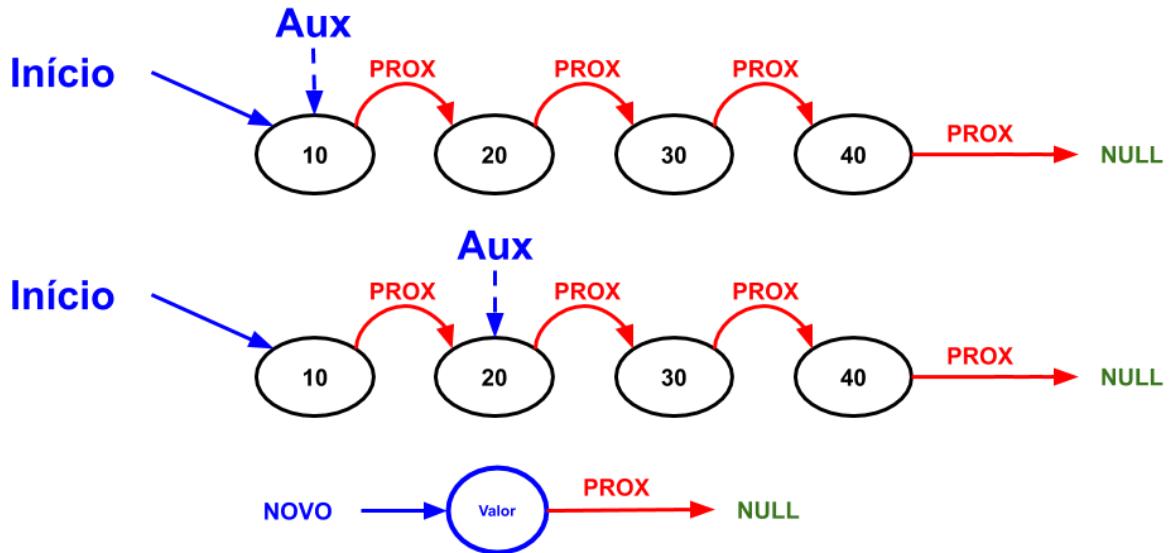
Suponha que o programador tenha adicionado os valores 40, 30, 20 e 10 sempre escolhendo a posição com o valor igual a 0 (zero). Como ficaria a lista encadeada? Caso você tenha imaginado a lista representada na Figura 25, você está acompanhando bem o conteúdo e podemos pensar em como adicionar em outra posição que não seja a de valor 0 (índice).

Figura 25: Exemplo de lista encadeada após algumas inserções no início



Caso o programador queira inserir novos dados em uma posição da lista diferente da posição 0 (zero) precisaremos percorrer os Nos até que possamos chegar no No da **posição anterior** a posição que desejamos adicionar. Por exemplo, considere a lista da Figura 25, se desejamos inserir um No com novos dados na posição 2 (`pos = 2`), necessariamente teremos que percorrer da posição 0 (índice) até a posição 1 (`pos - 1`). Para percorrer do índice da lista até (`pos - 1`) utilizaremos pela primeira vez nosso querido amigo `aux` que assim como índice é um ponteiro que faz referência a uma região de memória do tamanho de um No. O ponteiro `aux` caminhará sempre partindo do índice da lista até a posição que desejarmos. A Figura 26 apresenta a nossa ideia de caminhar na lista utilizando o ponteiro `aux` de forma visual.

Figura 26: Uso do ponteiro Aux para percorrer a lista



No exemplo da Figura 26, consideramos que o programador deseja inserir um *Node* com novos dados na posição **2**. Assim, o *aux* irá percorrer do *Node* da posição **0** (que contém o valor 10) até o *Node* da posição **1** (que contém o valor 20), pois o *Node* de posição **2** da lista será o *Node* **NOVO** que quando adicionado fará com que o *Node* que estava na posição **2** (que contém o valor 30) ocupe a posição **3**.

Agora que temos o conhecimento de um motivo para utilizar nosso ponteiro *aux*, iremos codificar nossa ideia para as inserções no meio-fim da lista (linhas 11 à 15):

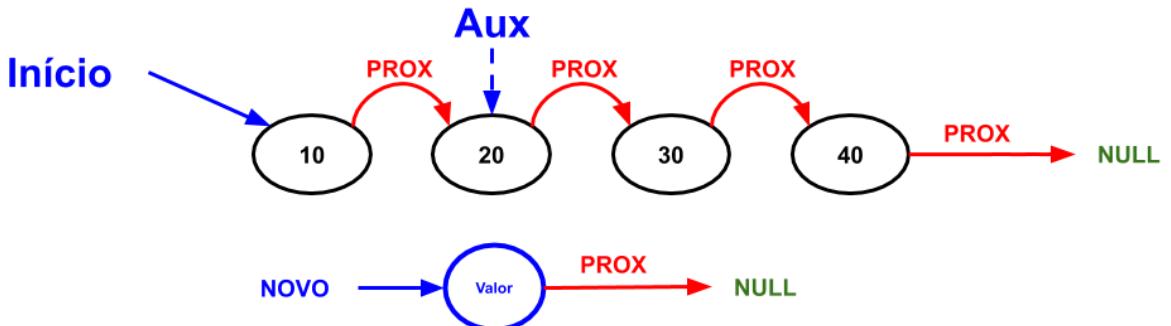
```

1 void adicionar(int valor, int pos){
2     if(pos>=0 && pos<=tam){
3         No*NOVO=malloc(sizeof(No));
4         NOVO->valor=valor;
5         NOVO->Prox=NULL;
6         if(inicio==NULL){
7             inicio=NOVO;
8         }else if(pos==0){
9             NOVO->Prox=inicio;
10            inicio=NOVO;
11        }else{
12            No*aux=inicio;
13            int i;
14            for(i=0;i<pos-1;i++){
15                aux=aux->Prox;
16            } //ao final do laço de repetição o aux estará
17            //apontando para o No que está na pos -1
18            tam++;
19        }
20    }

```

Na linha 12, estamos fazendo com que o ponteiro *aux* receba o endereço de memória do primeiro *Node* da lista (*inicio*). Nas linhas 13 a 15, utilizamos o laço de repetição **for**, para que em cada iteração possa-se **atualizar** o valor de ponteiro *aux*, com destino ao endereço do próximo *Node* que o *aux* está apontando (*aux->prox*) até chegarmos no *Node* desejado que será o *Node* que ocupa a posição (*pos - 1*).

Por fim, podemos finalmente analisar como inserir o *No NOVO* na lista. Podemos imaginar que o *No Aux* está bem próximo do *No* que está na posição (*pos*) que gostaríamos de adicionar nosso *No NOVO*. Veja a Figura 26 novamente.



Logo, para encaixar o *No NOVO* na lista iremos fazer o ponteiro *prox* do *No NOVO* receber o endereço de memória do *No* que está na posição **2**. No exemplo, este *No* é o que contém o valor **30**. Mas independente do valor e do exemplo, sabemos que este *No* sempre será o *aux->prox*, concordam? Assim, podemos fazer a seguinte operação para encaixar o *No* como apresentado na Figura 26:

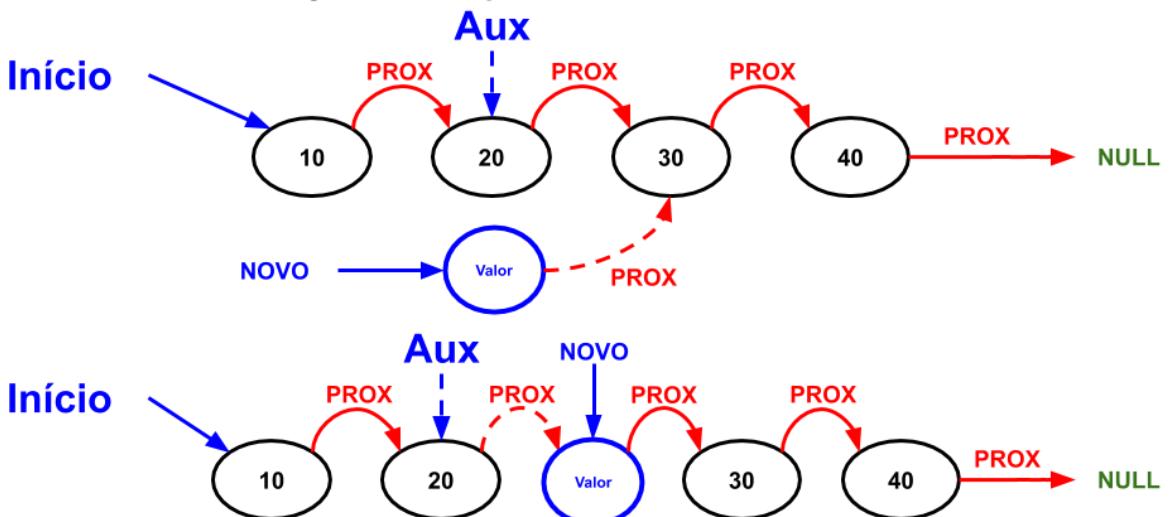
$$\text{NOVO} \rightarrow \text{prox} = \text{aux} \rightarrow \text{prox};$$

Agora resta fazer com que o ponteiro *prox* do *No aux* receba o endereço do *No NOVO*. Ou seja:

$$\text{aux} \rightarrow \text{prox} = \text{NOVO};$$

A Figura 27 apresenta como ficará a lista encadeada após as operações.

Figura 27: Inserção de um *No* no meio da lista



Então para completar nossa codificação adicionamos nas linhas 28 e 29 as duas operações que efetuamos para inserir o *No NOVO*:

```

1 #include <stdio.h>
2 #include <stdlib.h>
3
4 typedef struct no{
5     int valor;
6     struct no*Prox;
7 }No;
8
9 int tam=0;
10 No*inicio=NULL;
11
12 void adicionar(int valor, int pos){
13     if(pos>=0 && pos<=tam){
14         No*NOVO=malloc(sizeof(No));
15         NOVO->valor=valor;
16         NOVO->Prox=NULL;
17         if(inicio==NULL){
18             inicio=NOVO;
19         }else if(pos==0){
20             NOVO->Prox=inicio;
21             inicio=NOVO;
22         }else{
23             No*aux=inicio;
24             int i;
25             for(i=0;i<pos-1;i++)
26                 aux=aux->Prox;
27             NOVO->Prox=aux->Prox;
28             aux->Prox=NOVO;
29         }
30         tam++;
31     }
32 }
33 int main(){
34     adicionar(10,0);
35     adicionar(20,1);
36     adicionar(30,1);
37     adicionar(40,3);
38     return 0;
39 }
```

Agora que todas as possibilidades para adicionarmos um *No* na lista estão implementadas, independente da possibilidade escolhida, o tamanho dessa lista precisa ser acrescido em uma unidade a cada inserção de um novo *No*. Desta forma, a linha 31 incrementa a variável *tam* responsável por contabilizar o número de *Nos* da lista. Como ficará a lista visualmente após a execução deste código?

2.3.2 Remover em uma lista encadeada

Se podemos **adicionar** elementos em uma lista encadeada, é importante também darmos a opção para o programador **remover** elementos desta lista. De forma semelhante ao adicionar, iremos analisar 3 casos para uma possível remoção, considerando novamente a posição que o *No* se encontra na lista encadeada:

1. A lista está vazia;
2. A lista não está vazia e queremos remover o elemento da posição **0**, ou seja, do início;
3. A lista não está vazia e não queremos remover do início, ou seja, a variável **pos** está com um valor entre **1** e **tam-1**. Vamos chamar esse intervalo de valores de "**meio e fim**" da lista.

É importante relembrar que, independente da função que for implementada na lista, temos sempre que garantir sua propriedade: **não podemos ter espaços vazios**. Também temos que garantir que a posição a ser removida seja válida, novamente, iremos utilizar uma condição para garantir essas questões (veja a linha 3).

```

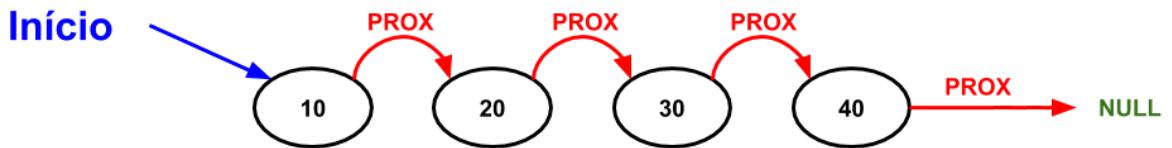
1 int remover(int pos){
2     if(pos>=0 && pos<tam){
3         //aqui ficará o nosso código para remover um No
4     }
5     tam--;
6 }
```

Primeiramente, vamos entender melhor o que a verificação da linha 3 nos permite fazer. Em uma lista encadeada, não podemos realizar operações em posições negativas, por isso na verificação da linha 3 o **pos** deve ser maior ou igual a **0**. Analisando ainda a condição criada, note que a posição (**pos**) não pode ser igual ao tamanho da lista (**tam**) e portanto temos exclusivamente que **pos < tam**.

Observação: Note que esta condição é diferente da função adicionar que tem um **<=** (menor e IGUAL). No remover temos apenas o **<** (menor), pois como estamos contando a partir da posição **0**, ou seja, o último elemento que existe na lista é o de posição igual a **tam**. Diferente da função adicionar, onde podemos adicionar na próxima posição depois da última, ou seja, neste caso o **No NOVO** será o "novo" último **No**.

Para um melhor entendimento, vamos analisar a Figura 28, abaixo.

Figura 28: Exemplo de lista encadeada



A lista encadeada da Figura 28 possui 4 **Nós**, portanto a variável **tam** está com o valor igual a **4**. De acordo com o que estudamos anteriormente, a primeira posição da lista é a posição **0**. Logo, se **tam == 4**, então teremos elementos nas posições **0, 1, 2 e 3**, por esse motivo só podemos remover entre as posições **0 e (tam - 1)**.

Para analisar os casos de remoção, considere que o programador irá inserir uma posição que satisfaça a condição de viabilidade (linha 3). Note que esta condição garante que não será possível remover caso a lista seja vazia (**caso 1**), então, podemos nos preocupar em analisar apenas os **casos 2 e 3** que são de fato possíveis para realizar uma remoção:

2. Remoção no **início** da lista (**pos = 0**);
3. Remoção no “**meio - fim**” da lista (**$0 < pos < tam - 1$**)

Independentemente se o elemento a ser removido estiver no **início**, “**meio ou fim**”, é sempre interessante que o elemento que for removido seja retornado para o programador.

Esta questão é importante, porque muitas vezes, o programador deseja remover um conjunto de dados de uma TAD para processá-la e não somente excluí-la. Como exemplo, podemos pensar em uma conta de e-mail ou de redes sociais. Quando você pede para excluir sua conta, antes de efetuar de fato a exclusão, o **site** sempre apresenta seus dados antes de pedir que você confirme a exclusão. Isso ocorre, para que você possa verificar se realmente quer de fato, excluir os dados. Alguns **sites** até apresentam seus dados de forma dramática no intuito de fazer você desistir da exclusão! :D

A pergunta é: como mostrar estes dados seria possível, se o programador tivesse pedido para remover uma conta e não tivesse como retornar os dados removidos para mostrar no **site**? Os dados apenas seriam excluídos e você não poderia ver pela última vez seus dados e filosofar se gostaria mesmo de apagar sua conta, não é mesmo? Imagine então em sistemas mais críticos, como por exemplo, a remoção de um valor em sua conta bancária, não existe sempre uma confirmação antes?

Logo, iremos aprender como guardar os dados que estão na posição que o programador quer remover, para retornar a ele quando este **No** for removido da lista. Para resolver este problema, vamos criar uma ideia muito semelhante a de um ponteiro já conhecido por nós: ponteiro **aux**. Vamos criar um ponteiro do tipo **No** denominado **lixo**. Já sabemos como criar ponteiros:

No *lixo;

Sempre iremos apontar o ponteiro **lixo** para o **No** que queremos remover. Então antes de efetuar a remoção, iremos fazer uma cópia dos dados que estão no

elemento *No* que o ponteiro lixo está "apontando", a fim de retornar estes dados para o programador quando terminarmos a remoção. Veremos abaixo como iremos fazer isso para cada um dos casos.

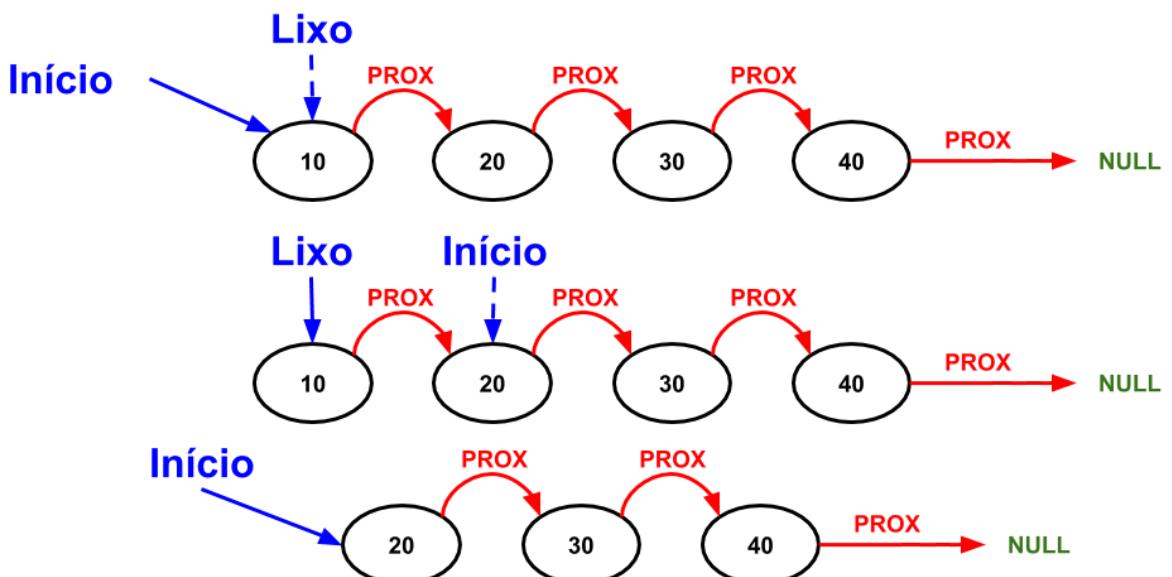
2.3.2.1 Remoção do início da lista

Caso o usuário solicite a remoção do primeiro *No*, ou seja, a variável *pos* contém o valor igual a **0**. Inicialmente criamos o ponteiro lixo e fazemos ele receber o endereço de memória do primeiro *No* da lista (início), com a seguinte operação:

lixo = inicio;

Afinal, queremos remover o início, portanto, ele é nosso lixo! Podemos imaginar esta operação como na primeira lista encadeada da Figura 29, abaixo.

Figura 29: Remoção com pos = 0



Logo, após "apontar" o ponteiro lixo para o início da lista, precisamos atualizar o ponteiro *início*, pois note que como o *No* com valor **10** será removido, o "novo" início será o *No* com valor **20** (veja a segunda lista da Figura 29). Faremos esta atualização do início com a operação a seguir.

inicio = inicio → prox;

Note que quando efetuarmos esta operação, o "antigo" *No* início ainda existe, porém só conseguimos acessá-lo usando o ponteiro lixo, não é mais possível acessá-lo com o ponteiro *início* (veja a segunda lista da Figura 29).

Outro detalhe importante, é lembrar que alocamos espaço na memória para cada um dos *Nos* da lista com a função *malloc*, portanto devemos desalocar este espaço para que de fato, possamos apagar este *No*. Antes de fazermos isso, vamos criar uma variável simples para guardar o valor inteiro que estava na primeira posição, e assim no futuro retorná-lo para o programador. Vamos denominar essa variável de *ret* (apelido carinhoso para retorno):

```
int ret = lixo->valor;
```

Note que, quando descrevemos o ponteiro *No lixo* em nosso código, podemos sempre recuperar duas informações: o ponteiro *prox*, que contém o endereço do próximo *No* da lista após o *No lixo* e o valor contido no elemento *No lixo*. No exemplo da Figura 29, este valor é **10**. Lembrando que, para escolhermos qual das duas informações desejamos utilizamos o operador *->* (seta), assim podemos afirmar que a variável *ret* após a operação acima, está guardando o valor **10**.

Depois de realizar as operações exibidas acima, podemos de fato, remover o *No* que o ponteiro *lixo* está "apontando", ou seja, iremos desalocar o espaço de memória que a variável do tipo ponteiro *lixo* está armazenando. Para tal, faremos o uso da função *free* que recebe como parâmetro um endereço de memória, desalocando-o este espaço de memória, ou seja, o espaço é liberado para outros usos, portanto podemos considerar que este espaço após desalocado, está nulo (**NULL**). Para desalocar o *No lixo* criamos a seguinte linha de código:

```
free(lixo);
```

A função *free* fará com que a memória alocada ao *No* que está sendo referenciado pelo ponteiro *lixo*, seja desalocada. Visualmente, a lista ficará como apresentada na terceira lista da Figura 29. Podemos codificar nossa ideia completa como no código a seguir:

```
1 int remover(int pos){
2     if(pos>=0 && pos<tam){
3         No* lixo;
4         int ret;
5         if(pos==0){
6             lixo=inicio;
7             inicio=inicio->prox;
8         }else{
9             //...
10        }
11    }
12    ret=lixo->valor;
13    free(lixo);
14    tam--;
15    return ret;
16 }
```

Note que, não há problemas se a lista possuir somente 1, 10, 1000, ou 1000000 *Nos*, nossa lógica sempre funcionará! Caso você ainda não esteja "enxergando" bem o "por que" sempre funciona, indicamos que você faça um desenho de exemplo com uma lista que tenha apenas um elemento e uma lista que tem 10 elementos. Siga a lógica e remova o primeiro elemento, o que você conseguiu notar?

2.3.2.2 Remoção no “meio - fim”

Caso o programador faça o uso da função *remover*, passando uma posição (*pos*) tal que $0 < pos < tam - 1$, ou seja, diferente de *início* (*pos == 0*), precisaremos utilizar novamente o nosso querido amigo *aux* que é responsável por nos ajudar a percorrer a lista desde o *início*, *No* a *No*, até uma determinada posição desejada. A utilização do *aux* na função *remover*, será similar ao seu uso na função *adicionar*. Como talvez

você possa ter uma memória um pouco ruim -- igual alguns integrantes LED! :D -- iremos fazer uma breve revisão abaixo.

Revisão: para caminhar com o ponteiro `aux`, inicialmente, criamos o ponteiro `aux` e fazemos ele receber o mesmo endereço que o ponteiro `início`:

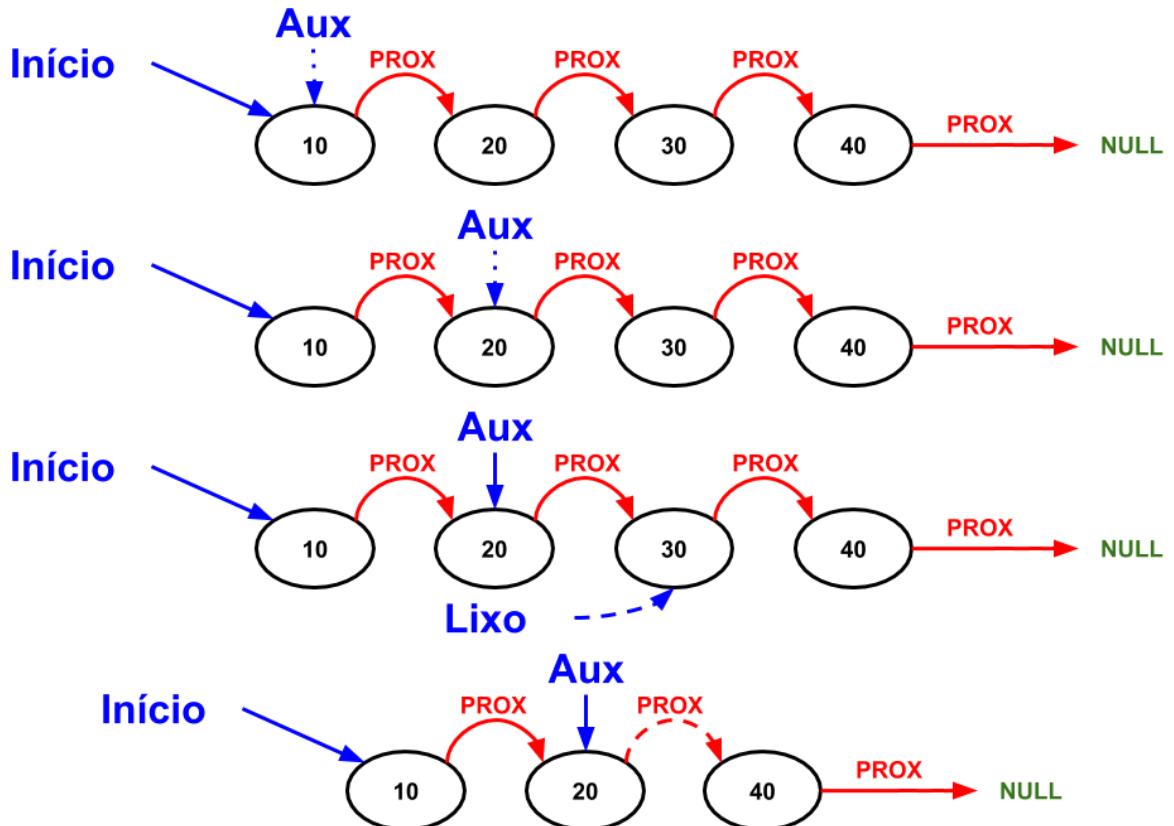
`No *aux = início;`

Utilizamos um laço de repetição, para que o ponteiro `aux` seja atualizado com o próximo `No` da lista (`aux->prox`) até o `No` de posição (`pos - 1`):

```
1 int i=0;
2 for (i=0; i<pos-1; i++){
3     aux = aux->prox;
4 }
```

Como visto na revisão, para remover um `No` de uma determinada posição da lista diferente da posição **0**, iremos percorrer nossa lista até um `No antes` da posição que desejamos remover, ou seja, a posição `pos - 1`. A Figura 30, apresenta um exemplo de remoção de um elemento que está na posição **2** da lista, neste caso o elemento do `No` desta posição contém o valor igual a 30. Após a execução do laço de repetição, o `aux` estará com o desenho exemplificado na segunda lista da Figura 30.

Figura 30: Exemplo de remoção com pos = 2 (Remover o No 30)



Depois de percorrer a lista até a posição (`pos - 1`) com o ponteiro aux, iremos utilizar nosso ponteiro lixo para que ele aponte para posição (`pos`), ou seja, exatamente para o *No* que gostaríamos de remover (veja a terceira lista da Figura 30). Para realizar essa operação iremos utilizar nosso ponteiro aux, pois note que o *No* que gostaríamos de remover é exatamente o `aux->prox`. Assim devemos fazer:

$$\text{lixo} = \text{aux} \rightarrow \text{prox};$$

Novamente, vamos criar a variável `ret` para guardar o valor do elemento que será removido.

$$\text{int ret} = \text{lixo} \rightarrow \text{valor};$$

Importante: visualize bem a terceira lista da Figura 30, se possível desenhe a Figura 30 em seu caderno e acompanhe lado a lado com a leitura. Note que, se deslocarmos o espaço de memória do *No lixo* (No de valor 30), o *No aux* (No de valor 20) terá agora seu ponteiro **prox** apontando para **NULL**, ou seja, perderíamos os demais *Nos* da lista (no exemplo, perderíamos o *No* de valor 40), pois nunca mais poderíamos acessá-lo, uma vez que só é possível acessar a lista com nosso ponteiro **Início** que é **global**.

Desta forma, antes de remover o *No* que está sendo apontado pelo ponteiro lixo, é necessário atualizar o ponteiro `aux->prox` para manter o encadeamento da lista, pois da forma como está o `aux->prox` está apontando para o *No* que será removido.

Veja novamente a terceira lista da Figura 30. Você consegue concordar que o **No** que **aux→prox** deverá apontar após a remoção do **No lixo** é o **lixo→prox**? Ou seja, se eu desejo remover o **No** com valor **30** é necessário que o próximo **No** ao **No** com valor **20** seja o **No** com valor **40**. Assim, o que desejamos é:

aux→prox = aux→prox→prox;

Após essa operação o **No lixo** existe, porém, não será mais alcançado na lista. Isso acontece pois, quando estivermos caminhando na lista e alcançarmos o **No** com valor **20**, o **próximo No** será o **No** com valor **40**. Portanto, logicamente o **No** com valor **30** já não pertence mais a lista e assim, podemos desalocar o espaço de memória do **No lixo** (**No** com valor **30**) sem perder nenhum **No** da lista. Para tal, basta efetuar a seguinte operação:

free(lixo);

Juntando todas nossas ideias podemos codificar este caso como apresentado nas linhas 8 a 15.

```

1 int remover(int pos){
2     if(pos>=0 && pos<tam){
3         No* lixo;
4         int ret;
5         if(pos==0){
6             lixo=inicio;
7             inicio=inicio->prox;
8         }else{
9             No*aux=inicio;
10            int i;
11            for(i=0;i<pos-1;i++)
12                aux=aux->prox;
13            }
14            lixo=aux->prox;
15            aux->prox=aux->prox->prox;
16        }
17    }
18    ret=lixo->valor;
19    free(lixo);
20    tam--;
21    return ret;
22 }
```

Agora que você já tem as funções adicionar e remover completas, você pode garantir que compreendeu todo esse conteúdo? **Dica:** para ter certeza se seu aprendizado está indo em um bom caminho, tente implementar o código da lista completo: com o adicionar e o remover.

Como você já sabe caminhar na lista, te **desafiamos** a criar uma função que percorre a lista imprimindo seus elementos inteiros, caso você consiga realizar nosso desafio, você poderá testar sua lista fazendo várias chamadas a funções de adicionar e remover na função **main** e imprimir sua lista ao final das chamadas!! Desta forma, você poderá fazer verificações em seu código! Segue também, uma lista de exercícios para que você possa treinar mais ainda seus conhecimentos.

2.4 Exercícios

- I. Verifique o código abaixo e indique, em ordem, por quais linhas ele passou em cada inserção. Como deve ficar, visualmente, a lista no fim do código?

```

1 #include <stdio.h>
2 #include <stdlib.h>
3 typedef struct no{
4     int valor;
5     struct no *prox;
6 }No;
7
8 int tam = 0;
9 No* inicio = NULL;
10
11 void adicionar(int valor, int pos){
12     if(pos >= 0 && pos <= tam){
13         No *novo = malloc(sizeof(No));
14         novo->valor = valor;
15         novo->prox = NULL;
16         if(inicio == NULL){
17             inicio = novo;
18         }else if(pos == 0){
19             novo->prox = inicio;
20             inicio = novo;
21         }else{
22             No *aux = inicio;
23             int i;
24             for(i = 0; i < pos - 1; i++){
25                 aux = aux->prox;
26             }
27             novo->prox = aux->prox;
28             aux->prox = novo;
29         }
30         tam++;
31     }
32 }
33 int main (){
34     adicionar(3, 0);
35     adicionar(14, 1);
36     adicionar(15, 1);
37     adicionar(9, 2);
38     adicionar(26, 0);
39     return 0;
40 }
```

- II. Complete a função abaixo, que deve: 1. Somar todos os pares positivos da lista e 2. Retornar o resultado da soma ao quadrado. Esse código deve funcionar para qualquer lista encadeada de inteiros.

Dica: Você pode usar o ponteiro "aux" aprendido anteriormente para percorrer a lista do começo e ir executando os cálculos.

```
1 float questao2 (No *aux){  
2     ...  
3 }
```

- III. Faça uma função que imprima ao contrário os valores de uma lista encadeada (Ex.: 1->2->3, imprimirá 3 2 1). Como essa função imprimiria a lista da questão número 1?
- IV. Desenhe e explique o passo a passo da remoção de um elemento no meio de uma lista encadeada.
- V. No próximo capítulo, aprenderemos sobre a complexidade de um programa e por quê um programa com alta complexidade não é viável. Tendo em mente que laços de repetição aumentam a complexidade, tente modificar o código de adicionar em uma lista encadeada, com o objetivo de criar um caso para adicionar no fim da lista. No código mostrado anteriormente, para adicionarmos no fim, teríamos que percorrer toda a lista por meio de um laço de repetição. No modificado, a adição no fim não poderá ser feita por meio de um laço, porém você poderá modificar qualquer parte do código, do registro ao "int main".

CAPÍTULO 3: MELHORANDO NOSSA LISTA ENCADEADA

3.1 Conceito Básicos sobre Complexidade

Neste momento do livro, você já pode afirmar que sabe como implementar uma **Lista Encadeada!** Logo, como bons programadores após um novo aprendizado, é importante refletir e fazer alguns questionamentos como, por exemplo:

As funções implementadas (adicionar, remover, etc.) foram criadas da melhor forma possível?

Este questionamento é muito importante na criação de estruturas de dados, pois além de implementar funções que permitam estruturar dados de formas úteis para programadores, as funções de uma TAD sempre devem ser **eficientes**. Mas como medir a eficiência de um bloco de código, digamos uma função, em linguagem C? Para entendermos melhor como analisar a eficiência de uma função, vamos introduzir de forma simplificada alguns conceitos sobre **Análise de Algoritmos**.

Observação: Apenas para relembrar, um algoritmo pode ser visto como a descrição passo a passo da sequência lógica de um programa de computador. Logo, podemos visualizar as nossas funções, em linguagem C, descritas neste livro como algoritmos.

3.1.1 Análise de algoritmos

A análise de algoritmos estuda a **correção** e o **desempenho de algoritmos**. Neste momento vamos focar apenas no estudo do desempenho dos nossos códigos, ok? Neste caso, podemos dizer que utilizaremos a análise de algoritmos para responder perguntas do seguinte tipo:

"Quanto tempo o algoritmo consome para processar a chamada da função adicionar para um elemento qualquer quando a lista tem 5 elementos?"

"Quanto tempo o algoritmo consome para processar a chamada da função adicionar, para um elemento qualquer, quando a lista tem n elementos?"

Considere n um número arbitrário, ou seja, n pode ser qualquer valor, por exemplo, $n = \{5, 10, 100, 1000, \dots\}$.

A resposta à segunda pergunta é bem mais complexa que a primeira, não é? Para casos como este, iremos responder o questionamento de uma forma um tanto quanto grosseira, algo como "o consumo de tempo é proporcional a $n^2 \log n$ no pior caso".

Como chegaremos a essa conclusão? Contando o número de operações realizadas pelo algoritmo! Vejamos um exemplo:

Vamos relembrar nosso código completo para adicionar em uma lista encadeada.

```

1 #include <stdio.h>
2 #include <stdlib.h>
3
4 typedef struct no{
5     int valor;
6     struct no*Prox;
7 }No;
8
9 int tam=0;
10 No*inicio=NULL;
11
12 void adicionar(int valor, int pos){
13     if(pos>=0 && pos<=tam){
14         NOVO=(No*)malloc(sizeof(No));
15         NOVO->valor=valor;
16         NOVO->Prox=NULL;
17         if(inicio==NULL){
18             inicio=NOVO;
19         }else if(pos==0){
20             NOVO->Prox=inicio;
21             inicio=NOVO;
22         }else{
23             NO*aux=inicio;
24             int i;
25             for(i=0;i<pos-1;i++)
26                 aux=aux->Prox;
27             NOVO->Prox=aux->Prox;
28             aux->Prox=novo;
29         }
30         tam++;
31     }
32 }
33 int main(){
34     adicionar(10,0);
35     adicionar(20,0);
36     adicionar(30,1);
37     adicionar(40,3);
38     return 0;
39 }
```

Considere a chamada da função realizada na linha 34, ou seja, adicionar o elemento de valor **10** na primeira posição da lista. Como é a primeira chamada de função na *main*, sabemos que nossa lista está vazia. Analisando a função adicionar podemos verificar que, neste caso, além da *linha 30*, apenas as *linhas 13 a 18* serão processadas, pois as demais linhas tratam de outros casos que não acontecem quando a lista está vazia. Como sabemos exatamente quais linhas serão processadas, podemos contabilizar o número de operações que serão feitas para o caso "lista vazia". De forma simplificada, vamos contabilizar uma operação para cada operador utilizado em cada linha.

Figura 31: Linhas executadas quando a lista está vazia

```

13 if(pos>=0 && pos<=tam){      3 operações (Duas comparações e uma operação lógica)
14     NOVO=malloc(sizeof(No));   1 operação (Alocação de memória do No)
15     NOVO->valor=valor;       1 operação (Alteração do valor do No NOVO)
16     NOVO->Prox=NULL;        1 operação (Definição da referência de *prox)
17     if(inicio==NULL){        1 operação (Comparação lógica)
18         inicio=NOVO;          1 operação (Atribuição de um valor ao início)
19     ...
20     tam++;                  1 operação (Incremento da variável 'TAM')

```

Assim, podemos afirmar que, para o caso "lista vazia", o algoritmo efetua **9 operações**. No momento pode parecer confuso, mas analisar mais uma linha do código irá melhorar o entendimento da ideia. Vamos analisar a próxima linha da *main* (linha 35), neste caso vamos adicionar o elemento de valor **20** na posição **0**, ou seja, no início, porém agora a lista já tem um elemento, o elemento de valor **10**. Veja a Figura 32.

Figura 32: Caso onde o novo elemento é adicionado no início de uma lista encadeada que já contém elementos.



Analizando a função *adicionar*, podemos verificar que, neste caso, além da *linha 30*, apenas as **linhas 13 a 16 e 19 a 21** serão processadas, pois as demais linhas tratam de outros casos que não acontecem quando a posição passada por parâmetro é **0**, ou seja, no início da lista. Vamos contabilizar o número de operações para este caso.

Figura 33: Linhas executadas ao adicionar na posição 0 quando existem elementos

```

13 if(pos>=0 && pos<=tam){      3 operações (Duas comparações e uma operação lógica)
14     NOVO=malloc(sizeof(No));   1 operação (Alocação de memória do No)
15     NOVO->valor=valor;       1 operação (Alteração do valor do No NOVO)
16     NOVO->Prox=NULL;        1 operação (Definição da referência de *prox)
17     ...
18 }else if(pos==0){
19     NOVO->Prox=inicio;
20     inicio=NOVO;
21     ...
22     tam++;                  1 operação (Incremento da variável 'TAM')

```

Podemos afirmar que, para o caso "adicionar no início", o algoritmo efetua **10 operações**. Note que para os dois casos estudados: "lista vazia" e "adicionar no início", podemos refletir e fazer o questionamento da pergunta inicial do capítulo:

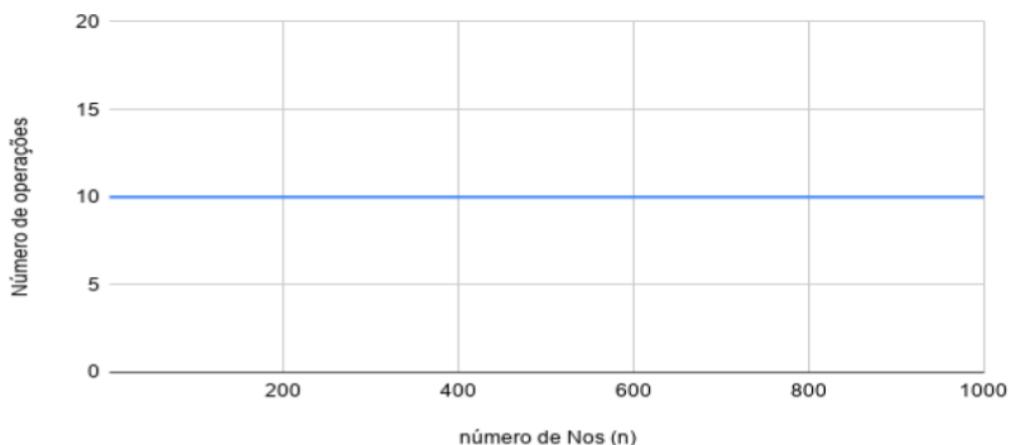
"Quanto tempo o algoritmo consome para processar a chamada da função adicionar, nos casos citados acima, para um elemento qualquer, quando a lista tem n elementos?"

O primeiro caso só existirá quando $n = 0$. Portanto o número de operações nunca mudará, ou seja, serão sempre **9** operações. Já para o segundo caso, note que não importa o valor de n , ou seja, para $n = \{1, 2, \dots, 5, \dots, 100, \dots, 1000, \dots\}$ sempre iremos alterar apenas o ponteiro **início** e ponteiro **prox** do novo **No**, então para este caso nunca precisaremos percorrer a lista. Logo, podemos afirmar também que o número de operações nunca mudará, sempre serão **10** operações.

Para casos como este dizemos que o bloco de código realiza as operações em tempo **constante**, uma vez que independente do tamanho da lista o número de operações é sempre o mesmo, ou seja, constante. Podemos ver essa análise de forma gráfica, como na Figura 34.

Figura 34: Função que representa o número de operações realizadas quando adicionamos um elemento no início de uma lista com n elementos.

Adicionar no início em uma lista encadeada com n elementos

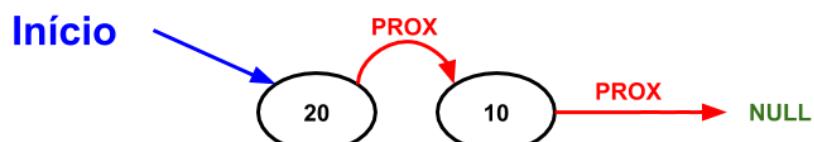


Desta forma, podemos afirmar que a função $f(n) = 10$ representa o número de operações realizadas em nosso código para adicionar um elemento no início da lista encadeada com n Nós, como esta função é **constante**, utilizaremos esta nomenclatura simplificada, dizendo que este bloco de código possui tempo **constante**.

Mas o que acontece quando queremos adicionar em nossa lista encadeada em uma posição diferente do **início**? Quantas operações são realizadas?

Vamos analisar a **linha 36** da função da **main**, neste caso vamos adicionar o elemento de valor **30** na posição **1**, ou seja, no "meio", pois neste momento a lista já tem dois elementos, o elemento de valor **10** e **20**. Veja a Figura 35.

Figura 35: Lista depois das linhas 34 e 35.



Analizando a função adicionar, podemos verificar que, neste caso (inserção do valor **30** na posição **1**), além da **linha 30**, apenas as **linhas 13 a 16** e **22 a 28** serão

processadas, pois a posição definida é maior que **0** e menor que **tam**, ou seja, no "meio ou fim" da lista. Vamos contabilizar o número de operações para este caso.

Figura 36: Linhas executadas ao adicionar no "meio ou fim".

```

13   if(pos>=0 && pos<=tam){      3 operações (Duas comparações e uma operação lógica)
14     NOVO=malloc(sizeof(No));    1 operação (Alocação de memória do No)
15     NOVO->valor=valor;
16     NOVO->Prox=NULL;          1 operação (Alteração do valor do No NOVO)
                                1 operação (Definição da referência de *prox)
17
18   ...
19
20
21
22 }else{                      1 operação (Atribuição de valor)
23   NO*aux=inicio;            int i;
24   for(i=0;i<pos-1;i++)      aux=aux->Prox;
25     aux=aux->Prox;          NOVO->Prox=aux->Prox;
26   aux->Prox=NOVO;          aux->Prox=NOVO;
27
28 }
29 tam++;                      1 operação (Incremento)
30

```

Note que, em nossa contagem, temos um pequeno problema: existe um laço de repetição (linha **25**) que tem como condição de parada uma posição (**pos**) que é passada por referência na função. Como não sabemos qual será o valor de **pos**, como iremos contar quantas operações **aux = aux→Prox;** serão realizadas?

Em casos como este, iremos analisar sempre o **pior caso**. O pior caso em análise de algoritmos é muito importante, pois ele nos dará um **limite máximo de operações** que um algoritmo realizará, sendo possível, informar o tempo máximo que devemos esperar para obter uma resposta final deste algoritmo. No código que está sendo analisado, o **pior caso** nos informará o número máximo de operações que serão realizadas para remover de um posição do "meio ou fim".

Para contabilizar essas operações, precisamos responder a seguinte pergunta: *qual seria o valor de pos no pior caso?* Bom, quanto maior o valor de **pos**, maior o número de operações **aux = aux→Prox;** (operação para caminhar na lista) serão feitas. Logo, o pior caso é quando **pos = tam**, pois estariamos adicionando o novo *No* no final da lista e para tal devemos percorrer todos os seus elementos.

Então agora podemos voltar a nossa pergunta: *quantas operações são realizadas na linha 26 no pior caso?* Bom, se você estiver entendendo bem os conceitos até o momento, sua resposta será: depende do **tamanho da lista!**

Observação: Caso você ainda não esteja entendendo bem os conceitos, sugerimos que você faça uma releitura deste capítulo até este momento mais uma vez. Caso você tenha respondido corretamente a pergunta, ou seja, o tamanho da lista, vamos seguir em frente!

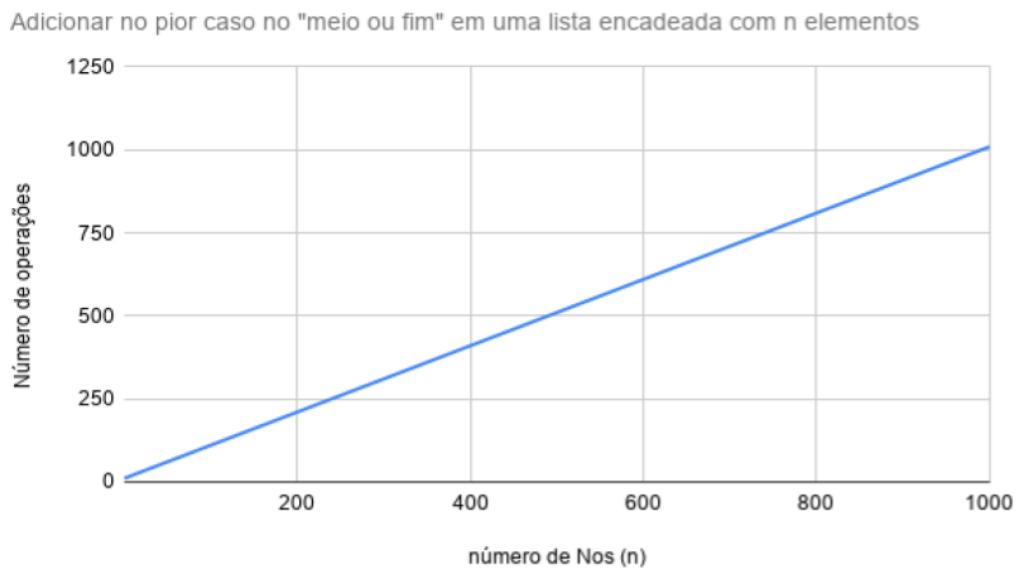
Sim, realmente depende do tamanho da lista (**tam**). Para conseguir responder essa pergunta em termos matemáticos, considere que o tamanho da lista seja **n**. Logo podemos voltar a nossa ideia de pensar que o **n** pode ser qualquer valor, ou seja, **n = {1, 2, ..., 5,... 100,... 1000,...}**. Desta forma, podemos dizer que serão realizadas **n – 1** operações **aux = aux→Prox;** Para entender como chegamos a conclusão de **n – 1** operações visualize a condição de parada do laço de repetição na **linha 25** do

nosso código, note que como `pos = n` e a condição de parada é `i < pos-1`, temos $n - 1$ operações `aux = aux->Prox;` sendo realizadas.

Agora que já conseguimos contar o número de operações da **linha 26** podemos contabilizar o número de operações total, no **pior caso**, do bloco completo. São (**11 operações** das outras linhas) + ($n - 1$ **operações** da linha 26), resultando em um total $n + 10$ **operações**, onde podemos dizer que n pode assumir qualquer valor, ou seja, $n = \{1, 2, \dots, 5, \dots, 100, \dots, 1000, \dots\}$.

Para casos como este dizemos que o bloco de código realiza as operações em tempo **linear**, uma vez que o número de operações **depende do tamanho da lista**. Esta dependência é representada pela função linear $f(n) = n + 10$, podemos ver essa análise de forma gráfica, como na Figura 33.

Figura 37: Pior caso de inserções no “meio ou fim” de uma lista encadeada.



Como podemos analisar através dos dois gráficos apresentados, podemos afirmar que é muito melhor quando o tempo de resolução do algoritmo (número de operações), em nossos códigos, é **constante** do que **linear**, principalmente à medida que nossa lista aumenta de tamanho. Desta forma, até o momento, podemos afirmar que adicionar no ínicio é mais rápido que adicionar em uma posição no "meio ou fim".

Estas análises nos levam a novas perguntas! Uma vez que não estamos apenas interessados em construir uma lista encadeada, mas construir uma lista encadeada eficiente! Considerando nosso interesse, seria possível alterar nossos blocos de códigos dos casos lineares para um que de alguma forma o número de operações fosse constante?

3.1.2 Lista encadeada com ponteiro no fim

Atualmente, de acordo com o que estudamos em relação à listas encadeadas e análise de algoritmos, nos deparamos com a necessidade de, além de possuir uma

estrutura de dados que supra a necessidade do programador, a TAD construída também precisa ser eficiente, principalmente se considerarmos a grande quantidade de dados que são manipulados nos sistemas atuais. Dificilmente iremos trabalhar com sistemas de informação que tenham listas de dados com poucos elementos. Como exemplo, podemos citar listas de alunos da UFC, clientes de um banco, usuários de uma rede social, com certeza essas listas ultrapassam centenas, milhares e milhões de dados. Desta forma, é extremamente necessário garantir a eficiência de nossa estrutura de dados e suas operações. A Tabela 1 define as complexidades, de forma simplificada (**Linear** ou **Constante**), de cada função implementada para nossa TAD lista encadeada do Capítulo 2. Lembrando que funções **lineares** consomem mais operações em comparação com funções **constantes**, quando o tamanho da lista é suficientemente grande.

Tabela 1: Complexidade simplificada dos métodos estudados.

Local Operação	Inserir	Remover	Buscar
1º Posição (Início)	Constante	Constante	Constante
Nº Posição (Fim)	Linear	Linear	Linear
Meio da lista	Linear	Linear	Linear

Analisando a Tabela 1, podemos ver que existem muitos métodos que possuem comportamento **linear**, ou seja, quanto mais elementos existem na lista, mais operações eles irão utilizar em suas execuções. Para facilitar a visualização de como melhorar a lista encadeada faremos uma análise baseada nos 3 locais listados em vermelho na tabela (início, meio e fim). Como já estudamos, as operações realizadas no início são **constantes**:

1. Quando precisamos adicionar no início basta atualizar o ponteiro **prox** do **No NOVO** e atualizar o **início** para o **NOVO**.
2. Quando queremos remover basta atualizar o **início** para o próximo **No** da lista (**inicio**→**prox**).
3. Quando precisamos buscar o primeiro elemento basta retornarmos o valor do **início** (**início**→**elemento**).

Para nossas modificações futuras, é importante que você note que comportamento **linear** nas funções da lista encadeada se baseia na necessidade de percorrer a lista para chegar em algum dos locais onde se efetuará a operação: **final** ou **meio da lista**.

A respeito de operações realizadas no meio da lista, podemos afirmar que necessariamente precisaremos percorrer a partir do **início** da lista **No** a **No**, até encontrar a posição desejada, uma vez que esta posição pode ser qualquer uma definida pelo programador.

Vocês conseguem enxergar qual motivo dessa necessidade? Vejamos.

Se possuímos uma lista encadeada com **1000 Nos** e queremos fazer uma operação no meio dela podemos necessitar acessar qualquer posição que não seja a posição **0** ou a **999**. Assim seria bem complicado implementar este caso em uma função com número de operações constante, pois em um momento, poderíamos ter que acessar o **No** na posição **23** e em outro momento outra posição como, por exemplo, **100**,

120, 4, 998 e assim adiante. Dessa forma não seria possível, criar um "atralho fixo" para os *Nos* do meio, assim como temos para o *No início*, que neste momento já sabemos que é o um ponteiro denominado *início*.

No entanto, quando refletimos sobre a ideia de criar um "atralho fixo" para um *No*, com a intenção de evitar caminhar em uma lista, a criação de um ponteiro para fixar o fim da lista se torna uma solução interessante, pois desta forma poderíamos evitar o uso de uma função **linear** para representar o número de operações utilizadas para este caso.

Até o momento, quando desejamos adicionar um novo elemento na última posição da nossa lista, devemos percorrer todos os *Nos* da lista, verificando o valor do ponteiro **prox** de cada *No*, até chegar ao último *No* atual, cujo seu ponteiro **prox** aponta para **NULL** (veja as linhas **23** a **26** da função adicionar). Da forma como está implementado, iremos sempre realizaremos n operações, onde n será a quantidade de elementos da nossa lista.

Para mudar esta questão, tal como nosso ponteiro **início**, onde referenciamos o *No* inicial da lista, iremos criar um ponteiro **fim** que irá sempre apontar para o *No* final da lista, facilitando o acesso ao último elemento da lista. Desta forma, sempre que decidimos acessar o *No* final, **poderemos fazê-lo de forma direta, através da referência** **fim**.

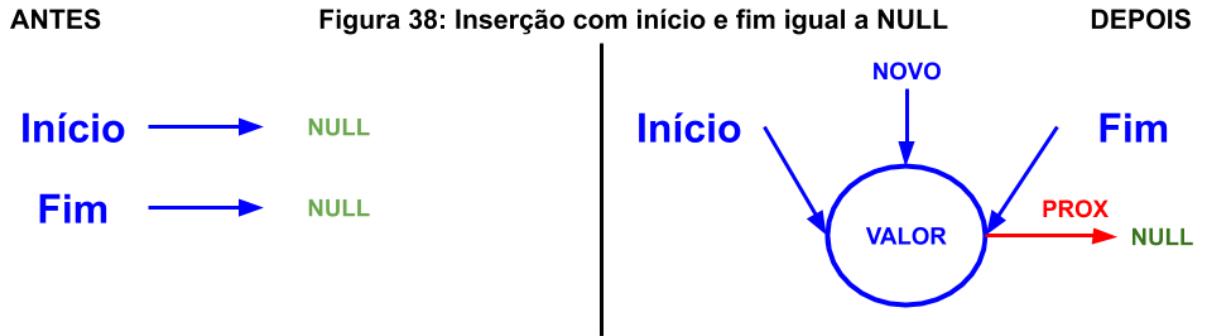
Assim, devemos atualizar nossa codificação para adicionar o ponteiro **fim** de forma semelhante ao ponteiro **início** na lista, ou seja, quando inicializamos a lista, tanto o ponteiro **início** quanto o ponteiro **fim** começam apontando para **NULL**, uma vez que nossa lista inicializa sempre vazia.

```

1 typedef struct no{
2     int valor;
3     struct no*prox;
4 }NO;
5
6 int tam=0;
7 NO*inicio=NULL;
8 NO*fim=NULL;
```

É importante efetuarmos as alterações em todo o código para que o ponteiro **fim** funcione da forma correta. Assim, iremos repassar por todas as funções e em cada caso, quando necessário, iremos adicionar novas linhas de código, para que possamos garantir que o ponteiro **fim** de fato irá sempre apontar para o *No* correto: o *No* final da lista.

Vamos começar pela função **adicionar**. O primeiro caso representa quando a lista está vazia, logo podemos apenas adicionar na posição **0**. Neste caso, tanto o **início** quanto o **fim** da lista devem referenciar o único *No* existente que será o *No NOVO*. A Figura 38 representa de forma visual a atualização necessária para que o ponteiro **fim** funcione corretamente.



Em nossa codificação, a alteração é simples, basta adicionar a linha 8 que faz o ponteiro fim também receber o endereço de memória do novo No.

```

1 void adicionar(int valor, int pos){
2     if(pos>=0 && pos<=tam){
3         No*NOVO=(No*)malloc(sizeof(No));
4         NOVO->valor=valor;
5         NOVO->Prox=NULL;
6         if(inicio==NULL){
7             inicio=NOVO;
8             fim=NOVO;
9         }else if(pos==0){
10            //...
11        }else if(pos==tam){
12            //...
13        }else{
14            //...
15        }
16        tam++;
17    }
18 }
```

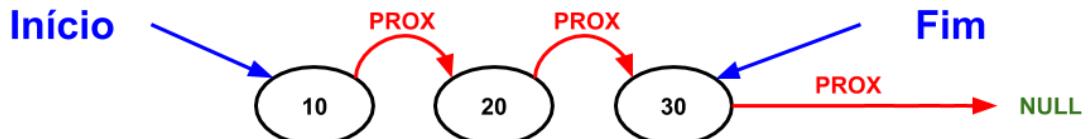
Note que para o caso onde desejamos adicionar um novo No no início da lista que já possui elementos, linhas 9 a 11 do código abaixo, como não iremos efetuar nenhuma alteração em outras partes da lista, incluindo o fim da lista, não há necessidade de acrescentar nenhuma alteração do código, ou seja, o código permanece como estava.

```

1 void adicionar(int valor, int pos){
2     if(pos>=0 && pos<=tam){
3         NOVO=(No*)malloc(sizeof(No));
4         NOVO->valor=valor;
5         NOVO->Prox=NULL;
6         if(inicio==NULL){
7             inicio=NOVO;
8             fim=NOVO;
9         }else if(pos==0){
10            NOVO->Prox=inicio;
11            inicio=NOVO;
12        }else if(pos==tam){
13            //...
14        }else{
15            //...
16        }
17    }
18 }
19 }
```

O caso que consideramos adicionar no "meio ou fim" da lista, agora poderá ser quebrado em dois casos. Vamos inicialmente tratar o caso onde desejamos adicionar no fim da lista. Para isso consideraremos lista encadeada da Figura 39 como exemplo.

Figura 39: Exemplo de lista encadeada



Agora que possuímos um ponteiro indicando o último *No* da lista a forma de inserir será alterada. Note que, com este novo ponteiro, podemos acessar diretamente o último *No*, evitando ter que caminhar desde o início da lista para acessá-lo. Desta forma, quando desejamos adicionar um **NOVO** *No* no final da lista, precisaremos somente atualizar o ponteiro **prox** do último *No* atual (*fim*→**prox**) para o **NOVO** *No* (que será o novo fim da lista):

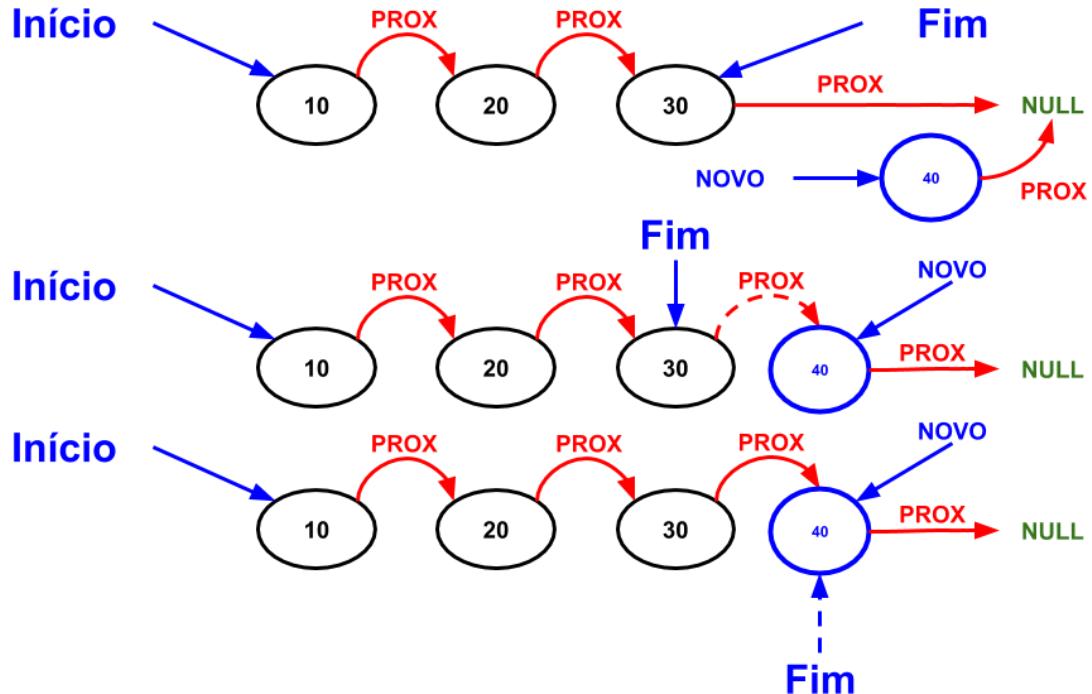
$$\text{fim} \rightarrow \text{prox} = \text{NOVO};$$

Em seguida, precisamos atualizar quem é o **NOVO** *No* final da lista, para tal, basta fazer o ponteiro *fim* apontar para o **NOVO**:

$$\text{fim} = \text{NOVO};$$

O exemplo da Figura 39, apresenta as operações completas de forma visual, quando inserimos o valor 40 na posição 2 (*tam - 1*), ou seja, no fim da lista, característica fundamental para identificar uma inserção na última posição.

Figura 39: Inserção do 40 na posição 2



A adição do ponteiro fim irá tornar o número de operações, para adicionar um novo elemento no fim da lista, **constante**, pois independente da quantidade de elementos da nossa lista, podemos fazer um acesso imediato ao No final. Em nosso código iremos criar uma nova condição para o caso adicionar no fim (linhas 12 e 14) a seguir, que serão executadas sempre que uma inserção for feita na posição `tam - 1`.

```

1 void adicionar(int valor, int pos){
2     if(pos>=0 && pos<=tam){
3         No*NOVO=(No*)malloc(sizeof(No));
4         NOVO->valor=valor;
5         NOVO->Prox=NULL;
6         if(inicio==NULL){
7             inicio=NOVO;
8             fim=NOVO;
9         }else if(pos==0){
10            NOVO->Prox=inicio;
11            inicio=NOVO;
12        }else if(pos==tam){
13            fim->Prox=NOVO;
14            fim=NOVO;
15        }else{
16            No*aux=inicio;
17            int i;
18            for(i=0;i<pos-1;i++)
19                aux=aux->Prox;
20            NOVO->Prox=aux->Prox;
21            aux->Prox=NOVO;
22        }
23        tam++;
24    }
}

```

Por fim, como vimos anteriormente, o caso que desejamos adicionar no "meio" da lista ainda necessitará do laço de repetição para percorrer cada *No* da lista até o *No* anterior a posição desejada pelo programador. Assim as linhas **15 a 22** continuarão sem alterações e portanto continuam utilizando um número de operações representado por uma função **linear**.

Por fim, podemos notar o efeito das alterações em nossa análise de complexidade simplificada na Tabela 2, considerando que, por enquanto, não vamos buscar melhorar os métodos que alteram uma posição no "meio" da lista. Veja a seguir.

Tabela 2: Atualização após utilização do ponteiro fim na inserção

Local Operação	Inserir	Remover	Buscar
1º Posição (Início)	Constante	Constante	Constante
Nº Posição (Fim)	Constante	Linear	Constante

Com essa simples alteração (acrescentar o ponteiro *fim* na lista encadeada) tornamos dois casos que antes tinham um número de operações **linear** para um número de operações **constante**. A inserção no final da lista agora utiliza **3 operações** e a busca utiliza 2 operações, independente do tamanho da lista.

Porém, é importante entender que toda nossa análise até o momento teve como foco a função adicionar, ainda possuímos um caso linear na Tabela 2, referente a remoção do último elemento em nossa lista encadeada. Logo cabe a nós um novo questionamento: seria possível remover o último elemento de uma lista encadeada de forma constante?

Agora vamos analisar este caso da remoção do último elemento para ver os efeitos do ponteiro *fim* na remoção.

Para remover um elemento da última posição de uma lista encadeada podemos pensar da seguinte forma:

"Bom já que eu sei diretamente quem é o último *No* da lista basta fazer com que o ponteiro *fim* aponte para o penúltimo *No* do encadeamento e o antigo *fim* seja desalocado"

Se você imaginou isso te digo uma coisa. **PERFEITO!!!** Mas isso tornaria o remover na última posição constante? Como fazer o *fim* apontar para a para o penúltimo *No*?

Em nossa estrutura de dados (**lista encadeada**) mesmo tendo acesso direto à última posição não conseguimos chegar até a penúltima de forma constante para realizar a operação acima. Então para essa estrutura teremos que nos contentar com um remover na última posição linear pois mesmo com o ponteiro *fim* será

necessário percorrer até a penúltima posição para manter as propriedades das listas encadeadas.

Mas vejamos... Pense comigo... Se a ação de adicionar um ponteiro fim fez com que dois métodos que eram lineares se tornassem constantes, será que de alguma forma parecida nós seríamos capazes de fazer com que a partir de qualquer *No* nós possamos acessar seu *No anterior*?

É lógico que não vou responder porque ainda teremos que elaborar perguntas para o capítulo :D .Mas se querem uma dica basta observarem a cor da palavra **anterior** e aguardar as cenas dos próximos capítulos literalmente.

3.1.3 Exercícios

- I. Analise a função em C a seguir e diga se cada caso é linear ou constante. Após isso, mostre uma função $f(n)$ que representa a complexidade da função, sendo n um inteiro maior que 1 que representa a potência no código.

Obs.: Para simplificar, considere apenas as operações por operador ($=$, $>$, $<$, \neq , etc.), uma por cada operador, como abordado no livro

```

1 int potenciacao (int indice, int potencia){
2     if (potencia >= 0){
3         if (potencia == 0){
4             return 1;
5         }
6         int i;
7         for (i = 1; i < potencia; i++){
8             indice *= indice;
9         }
10        return indice;
11    }
12 }
```

- II. Explique com suas palavras o que você entendeu da diferença entre códigos com complexidades lineares ou constantes. É certo que um código constante é bom, mas você concorda que existem situações em que códigos lineares são melhores? Dê um exemplo e/ou explique sua resposta para a pergunta anterior.
- III. Faça um código de uma lista encadeada de inteiros com as funções de adicionar, imprimir e uma busca diferenciada que será explicada a seguir, porém com as seguintes diferenças:
 - A. O último nó da lista, em que deverá ter um ponteiro fim apontando para ele, será o nó excluído. A função que imprimirá a lista não deve imprimir esse nó.

- B. Toda vez que um nó for adicionado, o nó do fim da lista será alterado. Ele deve possuir o valor da parte inteira da média de todos os elementos anteriores a ele (Por exemplo, na lista: 3→5→1→1→NULL, a média será 2.5, então a lista ficará: 3→5→1→1→2→NULL). Caso queira, pode fazer uma função para que isso aconteça.
- C. A função busca, nesta lista, não buscará um valor pedido pelo usuário, mas sim irá procurar na lista, se há algum nó com o valor que o nó final possui, sem contar com o próprio nó final. Caso haja, a busca retornará o valor do nó final, ou seja, a lista possui um nó com o valor igual a parte inteira da média da lista. Caso contrário, a função retornará -1, ou seja, a lista não possui tal valor.
- IV. Questão Desafio! Faça uma função que imprima a lista encadeada de inteiros com ponteiro no fim ao contrário, com os parâmetros mostrados abaixo. Serão necessárias alterações no código da lista. Caso não consiga fazer a questão, não se preocupe, tente refazer depois de ler o próximo capítulo.

Dica: Se, para imprimir normalmente uma lista, "caminhamos" a partir do início, logo, para imprimir ao contrário...

```
1 void imprimeInverso (NO *aux){  
2     //...  
3 }
```

CAPÍTULO 4: LISTA DUPLAMENTE ENCADEADA (LDE)

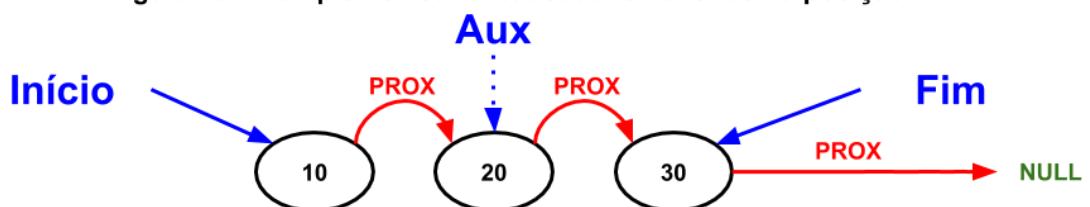
Continuando nossos estudos em relação a estrutura de dados, com a intenção de melhorar ainda mais a TAD estudada até o momento (lista encadeada) daremos início aos estudos de uma melhoria que transforma a lista encadeada em uma lista **duplamente encadeada**. Assim, para a continuação dos estudos, é necessário que todos os conceitos vistos anteriormente estejam fixados em suas mentes, pois agora estamos prestes a dar um grande passo adiante.

4.1 Conceito de Lista Duplamente Encadeada

Uma **lista duplamente encadeada** é uma extensão de uma lista encadeada. A grande pergunta que gerou a criação da lista duplamente encadeada por pesquisadores de estruturas de dados foi:

A remoção de um No que está localizado no fim de uma lista encadeada tem complexidade linear, mas seria possível solucionar este caso de forma que o número de operações fosse constante?

Figura 40: Exemplo de lista encadeada removendo na posição 2



Revendo os conhecimentos passados, quando consideramos apenas uma lista encadeada, precisamos "caminhar" desde o **início** da lista até um **No** anterior ao último **No** da lista, pois não há como voltar a partir do ponteiro **fim** ao **No anterior** para que possamos atualizar o seu ponteiro **prox** para **NULL** e por fim removê-lo. Desta forma, o número de operações realizadas é representada por uma função linear relacionada ao número de elementos da lista.

Porém, note que, se houvesse um **novo ponteiro** semelhante ao ponteiro **prox** (lembre-se que cada **No** possui um ponteiro **prox** que aponta para o próximo **No** da lista) que apontasse para o **No anterior**, poderíamos denominá-lo de **ant** (apelido carinhoso para anterior :D)

Cada *No* contém dois campos, chamados de links ou enlaces, que são referências para o *No* anterior e para o *No* posterior na sequência de *Nos*. A lista duplamente encadeada é muito similar à estrutura de dados estudada no capítulo anterior, porém com uma alteração na estrutura (struct) de cada *No* que será apresentada posteriormente. Nesta nova fase de estudos analisaremos as diferenças nas estruturas das duas listas, nas operações básicas das listas e, por fim, uma nova análise dos algoritmos apresentados. A principal diferença entre as **LE** (listas encadeadas) e as **LDE** (listas duplamente encadeadas) é o formato dos *Nos* que compõem as duas estruturas.

Para justificar a criação e os estudos sobre uma nova estrutura de dados podemos relembrar uma pergunta feita no final da explicação do capítulo anterior:

“Será que de alguma forma parecida nós seríamos capazes de fazer com que a partir de qualquer *No* nos possamos acessar seu *No* anterior?”

Bom, essa pergunta foi feita no capítulo anterior após percebermos que, com as **LE**, iríamos encontrar alguns problemas para realizar a operação de remover no final em tempo constante. Agora com os estudos que iremos realizar veremos ao final do capítulo que a resposta das dúvidas que ficaram em aberto anteriormente serão respondidas. Mas, para isso, estudaremos mais sobre as **LDE** (lista duplamente encadeadas).

4.2 Estrutura de uma LDE (lista duplamente encadeada)

Para que possamos iniciar os estudos da **LDE** teríamos que definir e relembrar muitos conceitos vistos no capítulo anterior, porém, para uma evolução mais ágil, iremos listar todos os conceitos que se aplicam tanto as **LE** quanto as **LDE**:

1. Conceito de *No*:

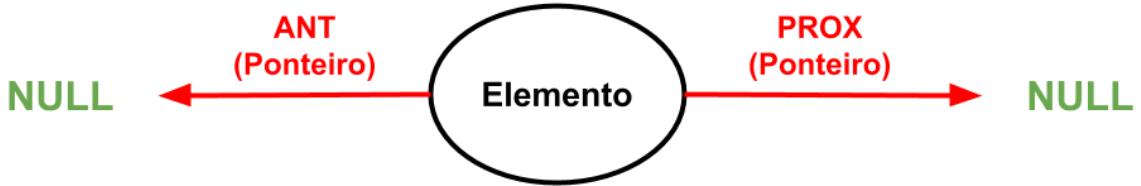
A definição de um *No* continua a mesma, porém com um acréscimo na hora de implementar.

2. Conceitos de início, fim, aux e **NOVO**:

Esses conceitos dos nossos ponteiros do tipo *No* seguem iguais aos que foram apresentados no capítulo anterior.

Basicamente todos os conceitos apresentados anteriormente serão os mesmos, mas existe algo que irá mudar e nos ajudar muito que é o acréscimo, na estrutura do nosso registro, de um ponteiro similar ao nosso **prox** (referência ao próximo *No*). Ele será o ponteiro **ant**. O **ant** fará referência ao *No* anterior de cada *No* da **LDE**. Vejamos a seguir a nova forma de representar os *Nos* na **LDE**.

Figura 40: Representação de um No de uma lista duplamente encadeada



Como dito anteriormente o ponteiro **ant** é responsável por indicar o *No* antecessor de um *No* qualquer. Para realizar a introdução desse novo campo, devemos fazer uma alteração da definição da **struct** da seguinte forma:

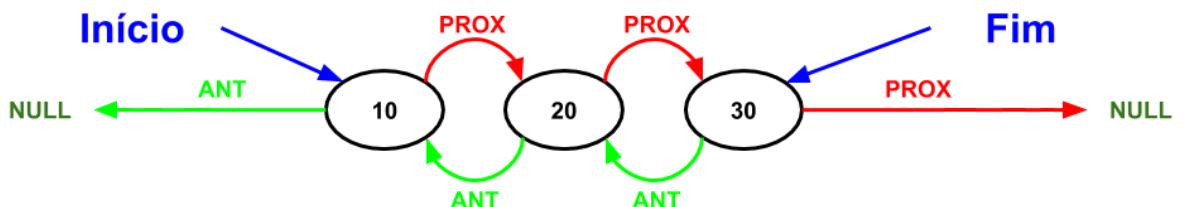
```

1 typedef struct no{
2     int valor;
3     struct no* Prox;
4     struct no* Ant;
5 }NO;
```

O ponteiro **ant** possui funcionalidade e características muito similares ao ponteiro **prox**, porém, podemos dizer que eles funcionam do sentido contrário. Considerando que as **LDE** serão apresentadas com a utilização do ponteiro **fim**, agora poderemos, assim como percorremos do início ao fim uma **LE**, percorrer a lista do último *No* ao primeiro *No* dela, utilizando o ponteiro **fim** como referência e nossos novos ponteiros **ant**.

Vale lembrar que, nas listas encadeadas, o ponteiro **prox** do último *No* recebe a referência de **NULL**, indicando que não existem mais *Nos* na sequência. Agora, com a utilização do ponteiro **ant**, todos os *Nos*, com exceção do primeiro, irão referenciar algum endereço de memória de outro *No*. Similar ao conceito que diz que o **prox** do último *No* é igual a **NULL**, podemos observar que o **ant** do primeiro *No* também irá fazer referência a região de memória vazia, pois, se ele é o primeiro da lista, não existirá ninguém antes dele. A seguir poderemos ver todos os conceitos apresentados até agora em um exemplo de uma lista encadeada com os valores 10, 20 e 30:

Figura 41: Exemplo de lista duplamente encadeada



Como foi dito anteriormente podemos percorrê-la em dois sentidos, que são eles **início**→**fim** e **fim**→**início**. Se optarmos por imprimir os dados na forma **início**→**fim** iremos obter o resultado 10,20,30. Porém da forma **fim**→**início** iríamos obter 30,20,10, pois nessa situação estariamos utilizando os ponteiros **ant** para percorrer a **LDE**.

4.3 Implementação de uma LDE

4.3.1 Adicionar em uma LDE

A função usada para adicionar valores em uma LDE é tão simples quanto as funções adicionar vistas até então, contudo, temos a criação do ponteiro `ant`, remetente ao elemento anterior a um dado elemento, então iremos refatorar nosso código para que agora ele trabalhe com esse novo ponteiro sempre que adicionarmos um novo elemento. A seguir iremos trabalhar com esse novo ponteiro em cada caso e demonstrar ele na nossa implementação.

Precisamos alterar algumas coisas na definição da nossa `struct`. Continuaremos criando uma `struct No` com um valor do tipo inteiro e um ponteiro `Prox` do tipo `struct No`, que vai apontar para o próximo item da lista, para essa estrutura representar uma LDE vamos precisar criar um novo ponteiro, agora chamado de `ant` do tipo `struct No` e que aponta para o seu antecessor.

```

1 typedef struct no{
2     int valor;
3     struct no*Prox;
4     struct no*Ant;
5 }NO;
6
7 int tam=0;
8 NO*inicio=NULL;
9 NO*fim=NULL;
```

No Capítulo 3, apresentamos e começamos a utilizar o ponteiro `fim` na nossa lista encadeada. Iremos construir o nosso código baseado na necessidade de existência desse ponteiro para que possamos adicionar, remover e imprimir elementos a partir dele, mantendo essas operações com complexidades constantes.

Para adicionarmos em uma LDE temos as mesmas condições de uma LE. Vamos adicionar ao início? Ao meio? Ao fim? Essa lista possui itens ou é uma lista vazia? Com essas perguntas em mãos, podemos trabalhar em todos os casos para que a nossa função de adicionar seja eficiente. Sempre lembrando que para adicionarmos dados em uma lista encadeada, podemos adicionar por posição, por valor ou qualquer outra forma que você desejar implementar, **desde que não se crie espaços entre dois Nos**, ou seja, respeite o mesmo princípio das listas em geral. A seguir podemos identificar cada caso existente na LDE.

```
1 void adicionar(int valor, int pos){  
2     if(pos>=0 && pos<=tam){  
3         //...  
4         if(inicio==NULL){  
5             //...  
6             }else if(pos==0){  
7                 //...  
8                 }else if(pos==tam){  
9                     //...  
10                }else{  
11                    //...;  
12                }  
13                tam++;  
14            }  
15        }  
16    }
```

A seguir iremos estudar cada possível caso quando se trata de inserir um novo **No**. Cada operação que for representada por meio de figuras poderá ser identificada pelas linhas pontilhadas que irão indicar qual ponteiro foi alterado e para onde ele estará apontando.

4.3.1.1 Adicionar com LDE vazia

Como vimos anteriormente, a **LDE** é muito similar a uma **LE**, então iremos analisar os casos ao adicionar um elemento. Levando em consideração que a primeira condição de validação das posições foi satisfeita (linha 2 do código acima), podemos seguir e analisar cada caso. O primeiro caso que analisaremos é a verificação se o nosso **início** é igual **NULL**, já sabemos que ao cairmos nesse primeiro caso, nossa **lista estará vazia**, então iremos tratar o caso em que a nossa **LDE** é vazia neste tópico.

Tal como anteriormente, iremos seguir os mesmos passos, precisaremos criar um **No NOVO** que receberá o valor do elemento passado. Logo após precisaremos inicializar os ponteiros **prox** e **ant** do nosso ponteiro **NOVO** fazendo com que eles inicialmente apontem para **NULL**, assim como podemos ver entre as linhas 3 e 6.

```

1 void adicionar(int valor, int pos){
2     if(pos>=0 && pos<=tam){
3         NO*NOVO=(NO*)malloc(sizeof(NO));
4         NOVO->valor=valor;
5         NOVO->Prox=NULL;
6         NOVO->Ant=NULL;
7         if(inicio==NULL){
8             inicio=NOVO;
9             fim=NOVO;
10        }else if(pos==0){
11            //..
12        }else if(pos==tam){
13            //..
14        }else{
15            //...
16        }
17        tam++;
18    }
19 }
```

Já sabemos que nossa LDE está vazia e que de acordo com as definições da LDE, nosso elemento **NOVO** inserido na LDE vazia será tanto nosso início como nosso fim. Podemos imaginar a seguinte situação, se você for lanchar na cantina da Universidade Federal do Ceará e não existir nenhum pedido na lista de pedidos, ao solicitar sua refeição ela será tanto a primeira (**início**) quanto a última (**fim**). Portanto, em nosso primeiro caso iremos realizar as seguintes operações.

início = NOVO;

fim = NOVO;

Feito isso já temos nosso elemento **NOVO** criado e com seu valor setado. Temos nosso **início** atualizado, nosso **fim** atualizado e logo, nosso primeiro elemento da LDE inserido na estrutura. Vejamos a seguir a inserção do valor 20, na posição 0 com a LDE vazia.

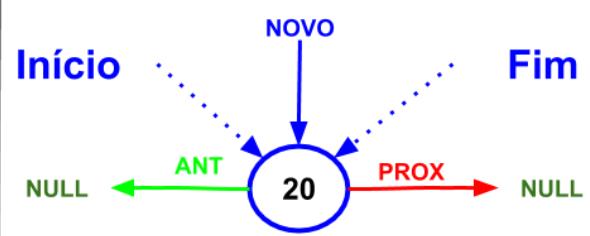
Antes

Início
Fim

Figura 42: Inserir na LDE sem elementos

→ NULL
→ NULL

Depois



Podemos observar o tratamento desse caso no código abaixo entre as linhas 7 e 9.

```

1 void adicionar(int valor, int pos){
2     if(pos>=0 && pos<=tam){
3         NO*NOVO=(NO*)malloc(sizeof(NO));
4         NOVO->valor=valor;
5         NOVO->Prox=NULL;
6         NOVO->Ant=NULL;
7         if(inicio==NULL){
8             inicio=NOVO;
9             fim=NOVO;
10        }else if(pos==0){
11            //..
12        }else if(pos==tam){
13            //..
14        }else{
15            //...
16        }
17        tam++;
18    }
19}
20
21 int main(){
22     adicionar(20,0);
23     return 0;
24 }
```

4.3.1.2 Adicionar no início com elementos na LDE

Agora iremos fazer uma análise do segundo caso existente na LDE, que é o de inserção no início com existência de elementos da lista. A principal característica desse caso é a solicitação de inserção com (`pos == 0`), ou seja, na primeira posição da lista. A diferença do caso anterior é que agora precisaremos manter a lista ligada de forma correta manipulando os ponteiros `ant` e `prox` do *No NOVO* e do *No* que o início faz referência.

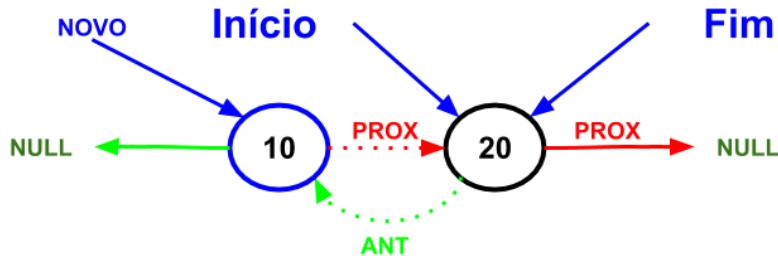
Agora que sabemos que nosso ponteiro `NOVO` deve ser o primeiro *No* da LDE podemos concluir que o “antigo” início será o *No* posterior ao `NOVO`. Vejamos, se você está na lista/fila de pessoas para entrar em uma festa e você é o primeiro da fila e deixar que um amigo entre na sua frente, logo você será o “próximo” do seu amigo e seu amigo o “anterior” a você. Dessa forma, para começar a inserir realizaremos as seguintes operações:

`NOVO->prox = início;`

`início->ant = NOVO;`

Assim a LDE ficará da seguinte forma, considerando a inserção do valor 10 na posição 0 e já existindo o valor 20 inserido anteriormente:

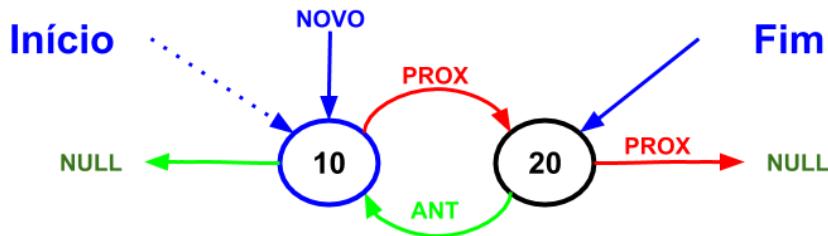
Figura 43: LDE depois das operações acima:



Logo após essas operações podemos identificar que cada *No* já está na sua devida posição, ou seja, o **NOVO** já é o **ant** do “antigo” **início** e o **início** é o **prox** do **NOVO**. Dessa forma agora só precisamos atualizar o *No* que o **início** faz referência, que neste exemplo é o *No* 20, para o **NOVO**.

início = NOVO;

Figura 44: Resultado final da inserção do 10 na posição 0:



Podemos observar que nesse caso não existiu nenhuma alteração no ponteiro **fim** e que, independentemente do tamanho da **LDE**, as operações serão as mesmas. Como dito anteriormente, as operações são muito similares a das **LE** porém com a utilização do ponteiro **ant** em cada *No*. Podemos observar o tratamento desse caso no código abaixo entre as linhas 10 e 13.

```

1 void adicionar(int valor, int pos){
2     if(pos>=0 && pos<=tam){
3         NOVO=(NO*)malloc(sizeof(NO));
4         NOVO->valor=valor;
5         NOVO->Prox=NULL;
6         NOVO->Ant=NULL;
7         if(inicio==NULL){
8             inicio=NOVO;
9             fim=NOVO;
10        }else if(pos==0){
11            NOVO->Prox=inicio;
12            inicio->Ant=NOVO;
13            inicio=NOVO;
14        }else if(pos==tam){
15            //...
16        }else{
17            //...
18        }
19        tam++;
20    }
21 }
```

4.3.1.3 Adicionar na última posição da LDE

Similar ao caso de inserção no `início` (`pos == 0`), agora temos um caso particular para uma inserção na última posição (`pos == tam`), pelo fato de estarmos trabalhando com a **LDE** com ponteiro `fim`, lembrando que a inserção desse ponteiro foi explicada no capítulo anterior.

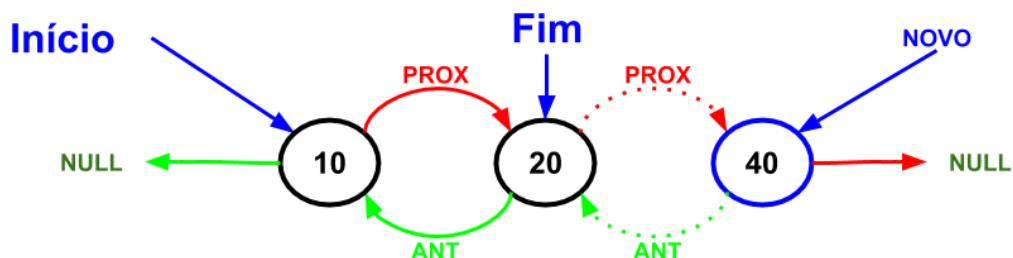
Como sabemos que nosso `No NOVO` deverá ser o último `No` da **LDE**, ele deverá ser o `No` posterior (`prox`) do `No` referenciado pelo `fim`. Consequentemente, o `No` referenciado pelo `fim` será o anterior (`ant`) `No NOVO`. Dessa forma precisaremos realizar as seguintes operações:

$$\text{fim} \rightarrow \text{prox} = \text{NOVO};$$

$$\text{NOVO} \rightarrow \text{ant} = \text{fim};$$

Vejamos como ficará a **LDE**, com a inserção do valor 40 na posição 2, que caracteriza o caso atual depois da realização das operações acima:

Figura 45: LDE depois das operações acima

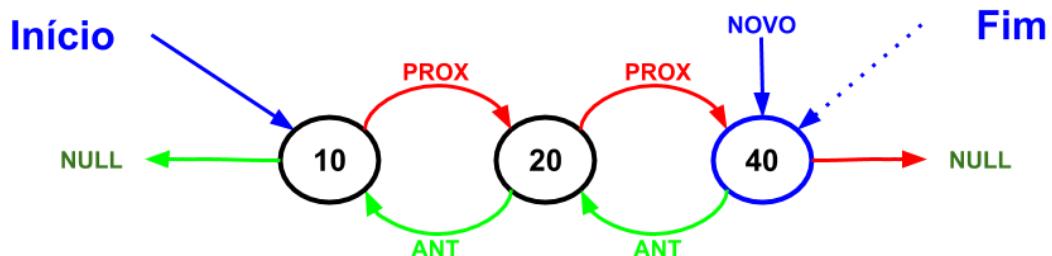


Agora podemos observar que cada `No` já está na sua devida posição, ou seja, nosso `NOVO` na última posição e o `No` referenciado pelo `fim` na penúltima posição. Com isso agora precisamos atualizar nosso ponteiro `fim` para o `NOVO`, pois agora ele é o último `No` da **LDE**. Assim para finalizar este caso precisamos realizar a seguinte operação:

$$\text{fim} = \text{NOVO};$$

Após essa operação nossa inserção será concluída e nossa **LDE** ficará da forma indicada na figura 46 (logo abaixo). Logo após a representação gráfica da **LDE** pós inserção, você pode observar o tratamento do caso entre as linhas 14 e 17 da codificação:

Figura 46: Resultado da inserção do 40 na posição 2 da LDE



```

1 void adicionar(int valor, int pos){
2     if(pos>=0 && pos<=tam){
3         NO*NOVO=(NO*)malloc(sizeof(NO));
4         NOVO->valor=valor;
5         NOVO->Prox=NULL;
6         NOVO->Ant=NULL;
7         if(inicio==NULL){
8             inicio=NOVO;
9             fim=NOVO;
10        }else if(pos==0){
11            NOVO->Prox=inicio;
12            inicio->Ant=NOVO;
13            inicio=NOVO;
14        }else if(pos==tam){
15            fim->Prox=NOVO;
16            NOVO->Ant=fim;
17            fim=NOVO;
18        }else{
19            //...
20        }
21        tam++;
22    }
23 }
24
25 int main(){
26     adicionar(20,0);
27     adicionar(10,0);
28     adicionar(40,2);
29     return 0;
30 }
```

4.3.1.4 Adicionar no “meio” da LDE

Por fim, chegamos no último caso da nossa função adicionar. Este caso ele se baseia na inserção em uma **LDE** entre duas posições, ou seja, esse caso ocorre sempre que uma a **pos** estiver no seguinte intervalo ($0 < \text{pos} < \text{tam}$). Este caso se torna um pouco mais complexo, pois existem algumas coisas que devem ser feitas.

A primeira delas consiste em percorrer até a posição exata que deseja inserir, diferentemente das listas encadeadas que devíamos percorrer até a posição anterior. Assim, precisaremos utilizar novamente nosso ponteiro aux, responsável por nos auxiliar quando for necessário percorrer qualquer estrutura de dados. Assim dentro de um *For* (de 0 até a posição desejada) iremos atualizá-lo até onde queremos inserir (**pos**). Se temos uma lista com “N” elementos e desejamos inserir na posição “N/2”, precisaremos realizar a “N/2” atualizações do ponteiro aux na estrutura *FOR*, considerando que inicialmente ele irá referenciar o *íncio*.

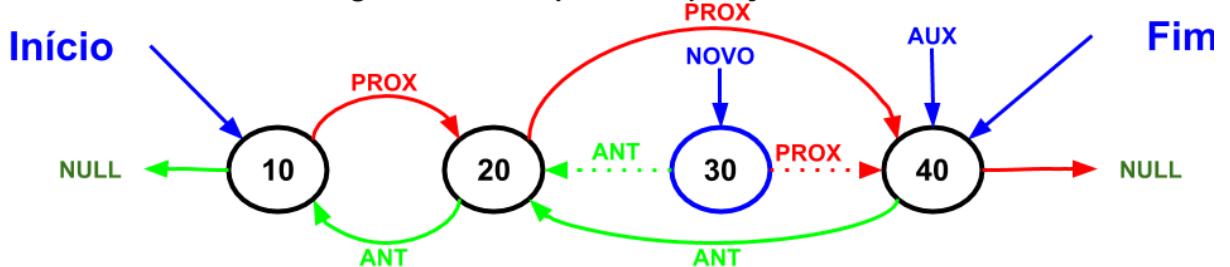
Depois disso iremos setar os ponteiros **ant** e **prox** do nosso **No NOVO**, é fundamental que todas as operações apresentadas nesse caso sejam feitas na ordem que serão apresentadas, pois uma simples alteração nessa ordem podem causar perdas de dados muito importante. Assim, vamos usar como exemplo a inserção do valor 30 na posição 2 da nossa **LDE**. Dessa forma, depois de percorrer a lista e atualizar nosso **aux** até a posição 2 iremos alterar os ponteiro do nosso **No NOVO** com as seguinte operações:

$\text{NOVO} \rightarrow \text{prox} = \text{aux};$

$\text{NOVO} \rightarrow \text{ant} = \text{aux} \rightarrow \text{ant};$

Logo após essas operações nossa LDE estará da seguinte forma:

Figura 47: LDE depois das operações acima



Para finalizar, iremos realmente colocar nosso No **NOVO** na lista, fazendo alterações no ponteiro **prox** do No 20 e no ponteiro **ant** do No 40. Vejamos que, na figura 47, ainda não conseguimos acessar o No de valor 30 ao percorrer a lista, por isso iremos realizar algumas operações para concluir a inserção.

Perceba a seguinte situação, se um programador deseja colocar um novo No na posição 2, quem estiver na posição 2 será o terceiro da lista, pois deve dar lugar ao **NOVO**, com isso nosso **NOVO** deverá ser o próximo (**prox**) de quem está na posição 1 (**aux → ant**) e o anterior (**ant**) de quem era o antigo dono da posição 2 (**aux**). Dessa forma precisamos realizar as seguinte operações:

$\text{aux} \rightarrow \text{ant} \rightarrow \text{prox} = \text{NOVO};$

$\text{aux} \rightarrow \text{ant} = \text{NOVO};$

Após essas operações obteremos o seguinte resultado final na nossa LDE:

Figura 48: Representação da operação ($\text{aux} \rightarrow \text{ant} \rightarrow \text{prox} = \text{NOVO};$)

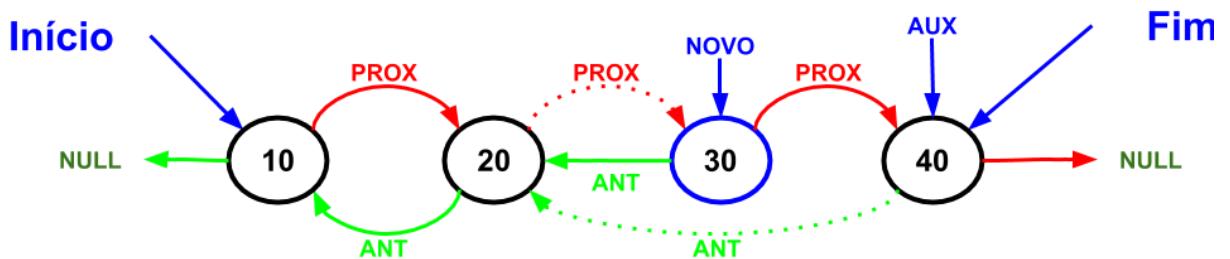
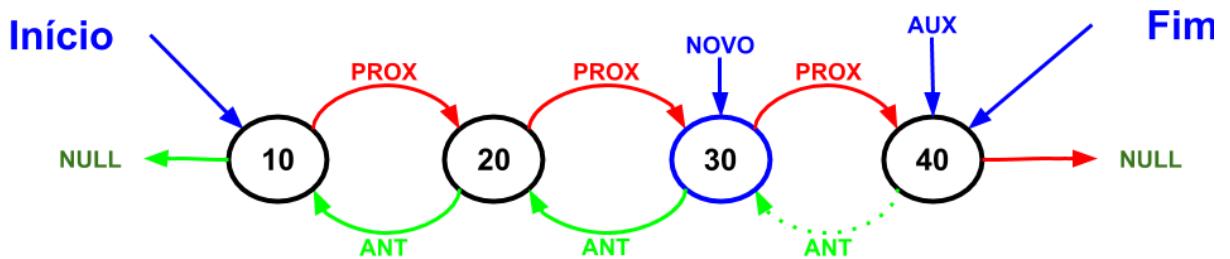


Figura 49: Representação da operação ($\text{aux} \rightarrow \text{ant} = \text{NOVO};$)



O tratamento desse caso pode ser visualizado na codificação abaixo entre as linhas 19 e 27:

```

1 void adicionar(int valor, int pos){
2     if(pos>=0 && pos<=tam){
3         NO*NOVO=(NO*)malloc(sizeof(NO));
4         NOVO->valor=valor;
5         NOVO->Prox=NULL;
6         NOVO->Ant=NULL;
7         if(inicio==NULL){
8             inicio=NOVO;
9             fim=NOVO;
10        }else if(pos==0){
11            NOVO->Prox=inicio;
12            inicio->Ant=NOVO;
13            inicio=NOVO;
14        }else if(pos==tam){
15            fim->Prox=NOVO;
16            NOVO->Ant=fim;
17            fim=NOVO;
18        }else{
19            NO*aux=inicio;
20            int i;
21            for(i=0;i<pos;i++){
22                aux=aux->Prox;
23            }
24            NOVO->Prox=aux;
25            novo->Ant=aux->Ant;
26            aux->Ant->Prox=NOVO;
27            aux->Ant=NOVO;
28        }
29        tam++;
30    }
31 }
32 }
```

4.3.2 Remover em uma LDE

Se podemos **adicionar** elementos em uma lista duplamente encadeada é importante também termos a opção **remover** elementos desta lista. De forma análoga ao adicionar, iremos analisar 4 casos para uma possível remoção considerando novamente a posição que o *No* se encontra na lista duplamente encadeada:

1. A lista está vazia;
2. A lista não está vazia e queremos remover o elemento da **posição 0**, ou seja, do **início**;
3. A lista não está vazia e queremos remover do **fim**, ou seja, o **pos** tem o valor de **tam-1**;
4. A lista não está vazia e não queremos remover do **início** e nem no **fim**, ou seja, a variável **pos** está com um valor entre **1** e **tam-2** ($0 < \text{pos} < \text{tam} - 1$).

Independente da função que for implementada na lista, temos sempre que garantir sua propriedade: não podemos ter espaços vazios. Também temos que garantir que a posição a ser removida seja válida, precisaremos de uma condição na nossa função que garanta isso (veja a linha 3):

```

1 int remover(int pos){
2     if(pos>=0 && pos<tam){
3         //aqui vai ficar o código para remover um NO
4         tam--;
5     }
6 }
7
8 }
```

Como em uma lista encadeada, uma lista duplamente encadeada não permite operações em posições negativas ou acima da quantidade de NO's da lista, para isso o valor do `pos` passado na função precisa estar na faixa de valores positivos e abaixo (ou igual) ao valor do `tam`.

Como estamos removendo um elemento da lista, podemos retornar o dado que o *No* continha dentro de si, iremos incrementar a nossa lista duplamente encadeada um retorno do valor armazenado no *No*, para isso usaremos uma variável chamada 'retorno' que irá armazenar o valor que vai ser removido e que será retornado ao fim da função.

Na linha 2 da imagem abaixo temos a nossa variável que irá salvar o valor do *No* que será removido. Na lista duplamente encadeada que estamos construindo, estamos utilizando apenas valores inteiros positivos, por isso, durante a inicialização da variável, estamos atribuindo o valor -1, pois se a posição do *No* a ser removido não for encontrado na lista, a função irá retornar o código -1.

```

1 int remover(int pos){
2     int retorno=-1;
3     if(pos>=0 && pos<tam){
4         ...
5         tam--;
6     }
7     return retorno;
8 }
```

4.3.2.1 Remover no início da LDE

Inicialmente devemos considerar que os casos que serão estudados de agora em diante assumem que nossa **LDE** não está vazia, pois nesse caso deve-se retornar o valor de -1 como foi dito anteriormente. A remoção no início da **LDE** é caracterizada por uma chamada da função `remover` com `pos == 0`, ou seja, na primeira posição. Dessa forma, as operações que devem ser feitas são similares às operações das listas encadeadas. No entanto, devemos fazer o tratamento dos nossos ponteiros `ant`. Devemos lembrar, como foi dito anteriormente, que nosso primeiro *No* da **LDE** possui ponteiro `ant` igual a `NULL`. Dessa forma, para realizar a remoção na primeira posição, inicialmente devemos guardar a referência do *No* inicio para que possamos retornar o valor do *No* removido. Então, faremos nossa variável `retorno` receber o valor de `inicio->elemento`. Em seguida devemos atualizar nosso ponteiro `início` para

o $\text{início} \rightarrow \text{prox}$ e em seguida atualizar o ponteiro ant do nosso novo início para NULL . Veja a seguir:

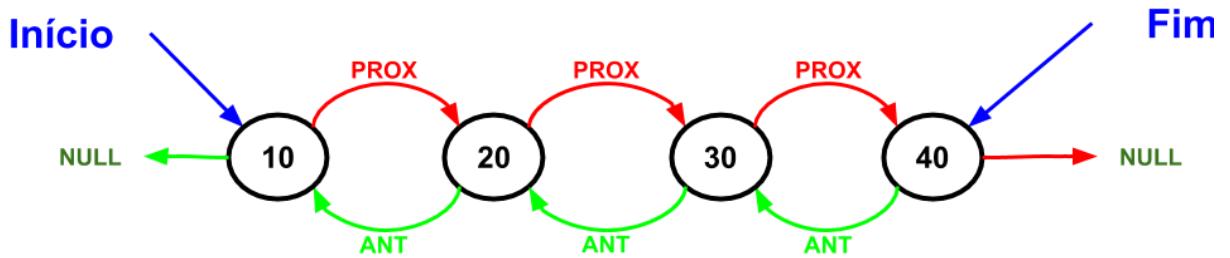
retorno = $\text{início} \rightarrow \text{elemento}$

$\text{início} = \text{início} \rightarrow \text{prox}$

$\text{início} \rightarrow \text{ant} = \text{NULL}$

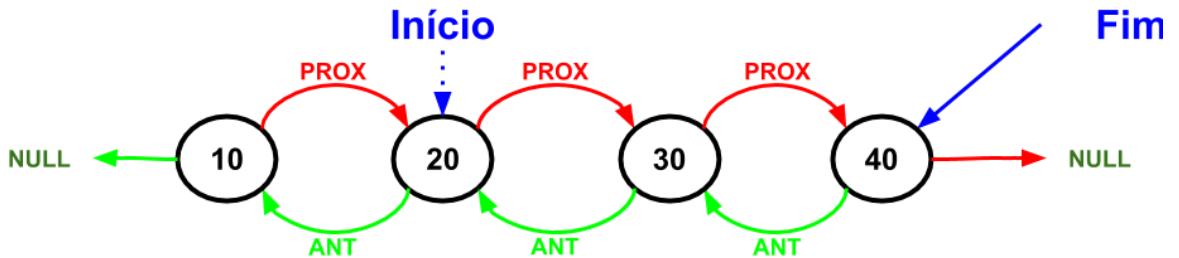
Dessa forma, iremos tomar como exemplo o resultado da nossa lista encadeada da figura 49:

Figura 50: Exemplo para remoção



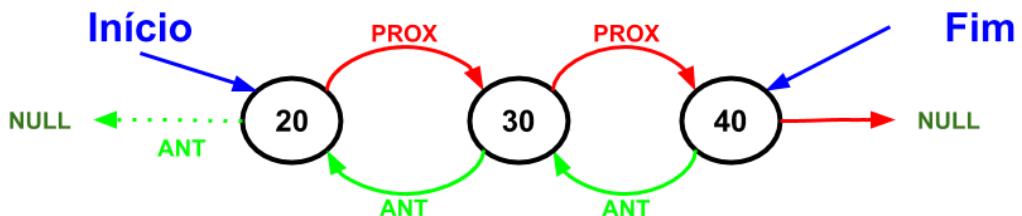
Agora vejamos a realização das operações vistas acima para remover o *No* de valor 10, o qual está ocupando a posição 0 com a seguinte chamada da função *Remover(0)*.

Figura 51: Exemplo da operação $\text{início} = \text{início} \rightarrow \text{prox}$



Assim podemos ver que, agora, o ponteiro *início* referencia o *No* de valor 20, que deve ser o primeiro *No* da lista depois que a posição 0 for removida. Dessa forma devemos prosseguir e atualizar o ponteiro *ant* do nosso *início* para *NULL*.

Figura 52: Exemplo da operação $\text{início} \rightarrow \text{ant} = \text{NULL}$



A realização de todas as operações citadas acima pode ser vista na codificação abaixo, entre as linhas 5 e 9.

```

1 int remover(int pos){
2     int retorno=-1;
3     if(pos>=0 && pos<tam){
4         if(pos==0){
5             No* lixo=inicio;
6             retorno=inicio->valor;
7             inicio=inicio->Prox;
8             inicio->Ant=NULL;
9             free(lixo);
10        }else if(pos==tam-1){
11            //...
12        }else{
13            //...
14        }
15        tam--;
16    }
17    return retorno;
18 }
```

Após essa operação podemos finalizar a função `remover` retornando o valor da variável `retorno` para que possamos indicar que ela foi concluída com sucesso.

4.3.2.2 Remover no fim da LDE

A remoção na última posição da **LDE** é caracterizada pela chamada da função `remover` com o parâmetro `pos == tam - 1` (`Remover(tam-1)`), pois, em uma lista de “n” elementos, podemos realizar operações entre as posições 0 e n-1. Assim como em todos os outros casos devemos guardar na variável `retorno` o valor do **No** que será removido, neste caso o valor do **No** que nosso **fim** referência será guardado nessa variável. Primeiramente devemos realizar a operação `retorno = fim→elemento`. Em seguida, precisamos fazer com que nosso ponteiro **fim** aponte para o penúltimo **No** da **LDE**, pois, após a última posição ser removida, a penúltima irá substituí-la. Logo depois, iremos realizar a operação `fim = fim→ant`, após isso precisaremos fazer nosso `fim→prox` receber `NULL`, pois uma das características de nossa **LDE** diz que não existe **No** após o último elemento. Assim vejamos em ordem as operações acima:

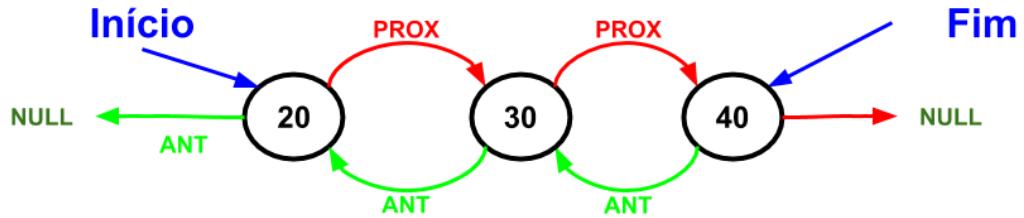
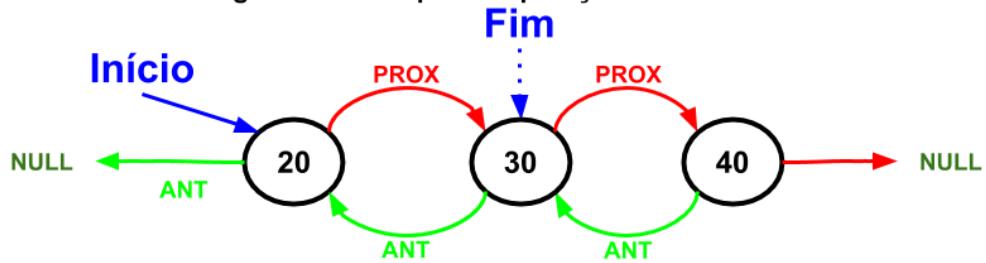
$$\text{retorno} = \text{fim} \rightarrow \text{elemento}$$

$$\text{fim} = \text{fim} \rightarrow \text{ant}$$

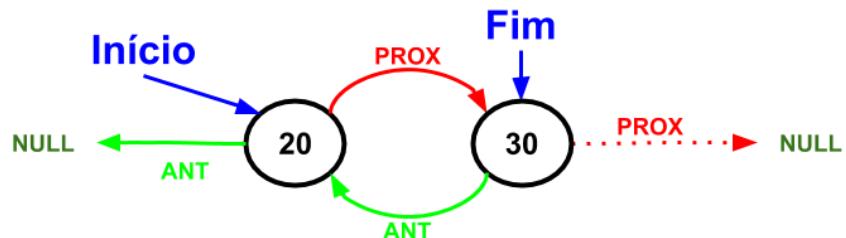
$$\text{fim} \rightarrow \text{prox} = \text{NULL}$$

Tomando como exemplo a **LDE** da figura 52, iremos remover a última posição, que trata-se do **No** de valor 40. Vejamos a seguir a representação gráfica das operações necessárias para realizar a remoção na última posição de uma **LDE**.

Figura 53: Exemplo de remoção na última posição

Figura 54: Exemplo da operação $\text{fim} = \text{fim} \rightarrow \text{ant}$ 

Após realizar a atualização do ponteiro **fim**, que agora está referenciando o penúltimo elemento da LDE já que o último será removido, devemos retirar o nosso antigo fim da LDE realizando a seguinte operação:

Figura 54: Exemplo da operação $\text{fim} \rightarrow \text{prox} = \text{NULL}$ 

Agora podemos ver como ficará a implementação das operações vistas nesse caso de remover no fim. A codificação pode ser vista na figura abaixo entre as linhas 11 e 15.

```

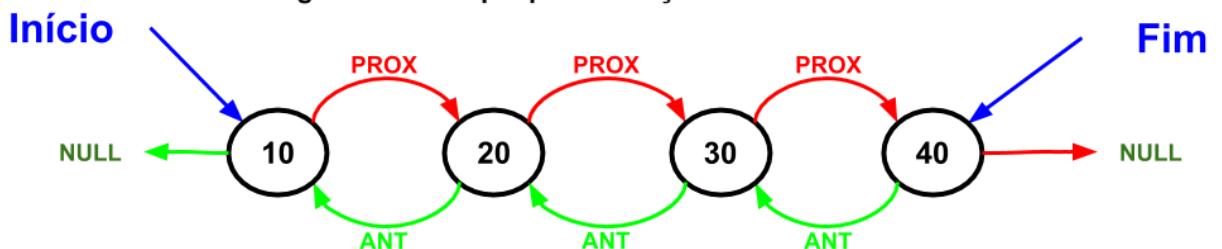
1 int remover(int pos){
2     int retorno=-1;
3     if(pos>=0 && pos<tam){
4         if(pos==0){
5             N0* lixo=inicio;
6             retorno=inicio->valor;
7             inicio=inicio->Prox;
8             inicio->Ant=NULL;
9             free(lixo);
10        }else if(pos==tam-1){
11            N0* lixo=fim;
12            retorno=lixo->valor;
13            fim->Ant->Prox=NULL;
14            fim=fim->Ant;
15            free(lixo);
16        }else{
17            //...
18        }
19        tam--;
20    }
21    return retorno;
22 }
23

```

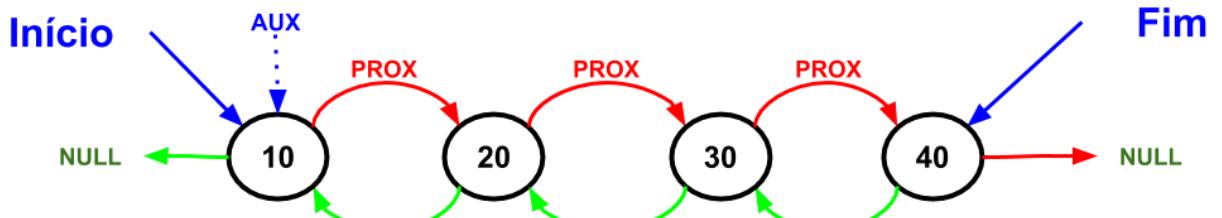
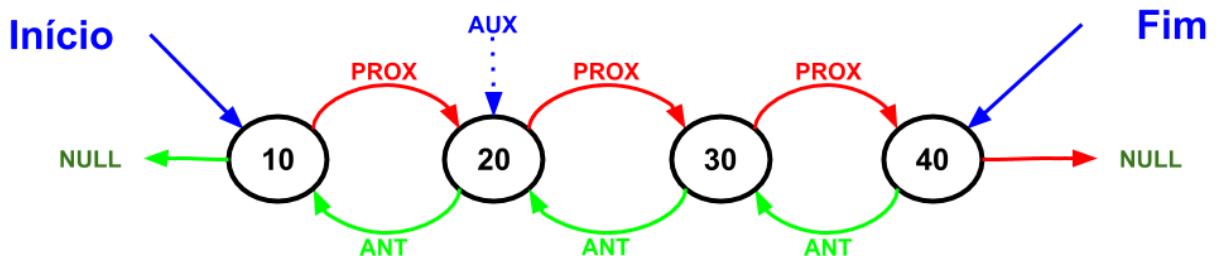
4.3.2.3 Remover no “meio” da LDE

Por fim, agora que já sabemos como remover a primeira e a última posição de uma forma simples, pelo fato de termos acesso direto ao primeiro e ao último elemento, chegamos ao caso contrário, onde a remoção pode ocorrer em qualquer elemento diferente do **início** e **fim**. Para isso, utilizaremos novamente nosso ponteiro aux para percorrer a **LDE** até a posição que desejamos remover. Para entender melhor como iremos percorrer para remover, tomaremos como exemplo a seguinte **LDE**.

Figura 56: Exemplo para remoção no “meio” da LDE

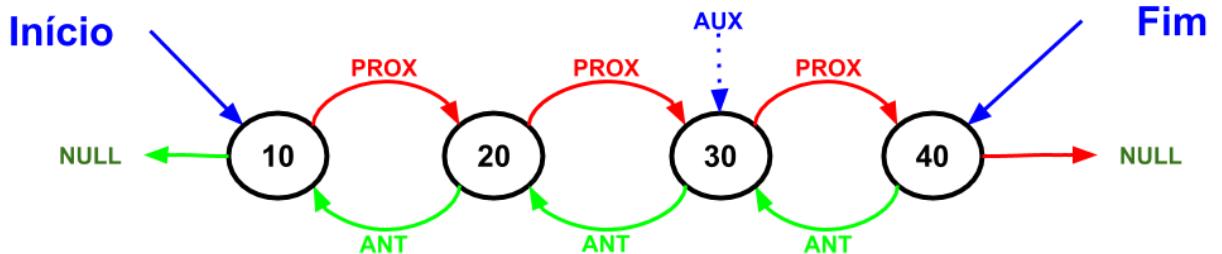


Iremos supor que o usuário deseja remover o **No** que ocupa a posição 2 de nossa **LDE**, ou seja, o **No** que possui valor 30. Dessa forma, como foi dito acima, temos que percorrer do **início** (posição 0) até a posição que será removida (posição 2). Logo faremos nosso aux receber o **início** e iremos percorrer enquanto uma variável “*i*” for menor que a posição a ser removida. O percorrido da **LDE** pode ser visto na codificação mais a frente entre as linhas 17 e 20. Por enquanto, vejamos a seguir o percorrido de forma visual, considerando que nossa variável “*i*” vai de 0 até **pos**:

Figura 58: Exemplo para $i == 1$ 

Logo após a última iteração, nosso ponteiro aux estará referenciando o No de valor 30, assim como podemos ver abaixo:

Figura 59: Exemplo após última iteração



Após isso, devemos realizar a movimentação dos ponteiros que estão ligados ao No que será removido. No entanto, vamos acessar esses ponteiros a partir do próprio aux, pois, como foi dito anteriormente, colocamos nosso ponteiro aux exatamente na posição que irá ser apagada. Para ficar mais claro vamos raciocinar na seguinte forma:

Com a remoção do No de valor 30, a lista deve ficar com $10 \rightarrow 20 \rightarrow 40$, ou seja, o prox do No de valor 20 deve ser o No de valor 40, assim como o ant do No de valor 40 dever ser o No de valor 20.

Em outras palavras, precisamos eliminar o espaço vazio no meio da lista que aparecerá com a remoção do No de valor 30, e, para isso, utilizaremos as seguintes operações:

$$\text{aux} \rightarrow \text{ant} \rightarrow \text{prox} = \text{aux} \rightarrow \text{prox}$$

$$\text{aux} \rightarrow \text{prox} \rightarrow \text{ant} = \text{aux} \rightarrow \text{ant}$$

Dessa forma estaremos fazendo exatamente o que foi dito acima:

- O prox do No de valor 20 deve ser o No de valor 40.

$$\text{aux} \rightarrow \text{ant} \rightarrow \text{prox} = \text{aux} \rightarrow \text{prox}$$

2. O **ant** do **No** de valor 40 dever ser o **No** de valor 20.

$$\text{aux} \rightarrow \text{prox} \rightarrow \text{ant} = \text{aux} \rightarrow \text{ant}$$

Para finalizar, iremos ver estas operações serem realizadas de forma gráfica. Podemos observar a codificação abaixo entre as linhas 21 e 24.

Figura 60: Representação da operação $\text{aux} \rightarrow \text{ant} \rightarrow \text{prox} = \text{aux} \rightarrow \text{prox}$

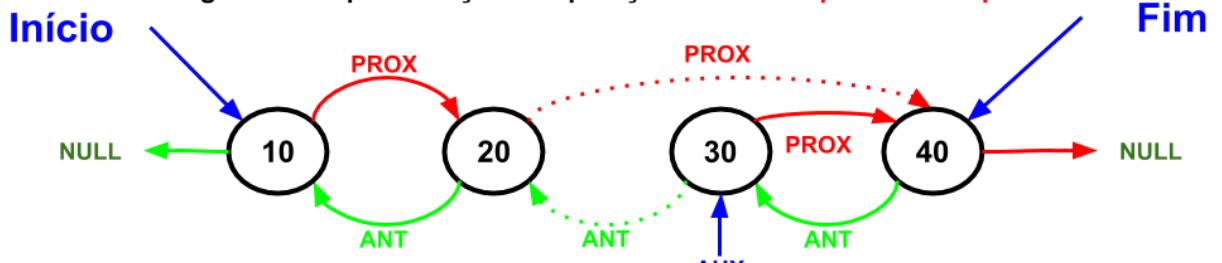
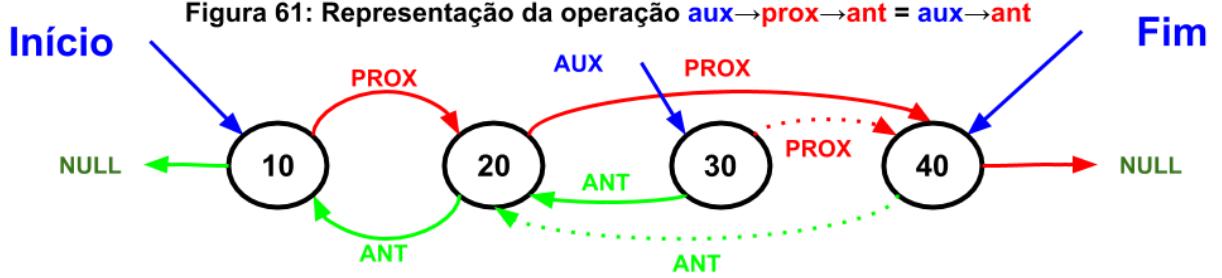
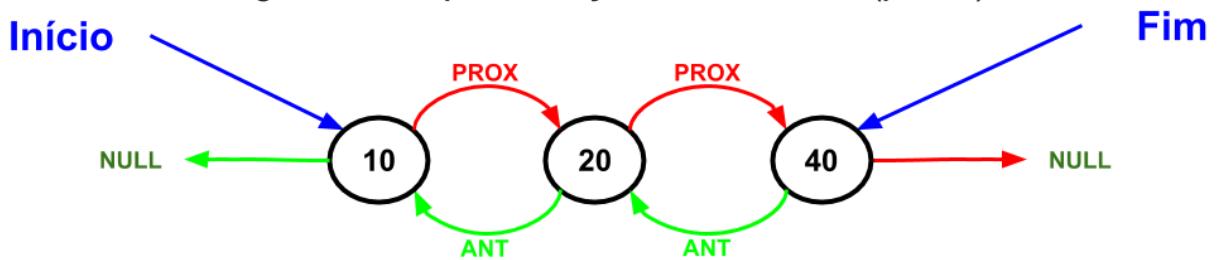


Figura 61: Representação da operação $\text{aux} \rightarrow \text{prox} \rightarrow \text{ant} = \text{aux} \rightarrow \text{ant}$



Após essas operações iremos obter a seguinte Lista Duplamente Encadeada:

Figura 62: LDE após a remoção do No de valor 30 (pos = 2)



```
1 int remover(int pos){
2     int retorno=-1;
3     if(pos>=0 && pos<tam){
4         if(pos==0){
5             N0* lixo=inicio;
6             retorno=inicio->valor;
7             inicio=inicio->Prox;
8             inicio->Ant=NULL;
9             free(lixo);
10        }else if(pos==tam-1){
11            N0* lixo=fim;
12            retorno=lixo->valor;
13            fim->Ant->Prox=NULL;
14            fim=fim->Prox;
15            free(lixo);
16        }else{
17            N0* aux=inicio;
18            for(int i=0;i<pos;i++){
19                aux=aux->Prox;
20            }
21            retorno=aux->valor;
22            aux->Ant->Prox=aux->Prox;
23            aux->Prox->Ant=aux->Ant;
24            free(aux);
25        }
26        tam--;
27    }
28    return retorno;
29 }
```

4.3.3 Exercícios

- I. ...
- II. ...
- III. ...
- IV. ...
- V. ...

CAPÍTULO 5: PILHA

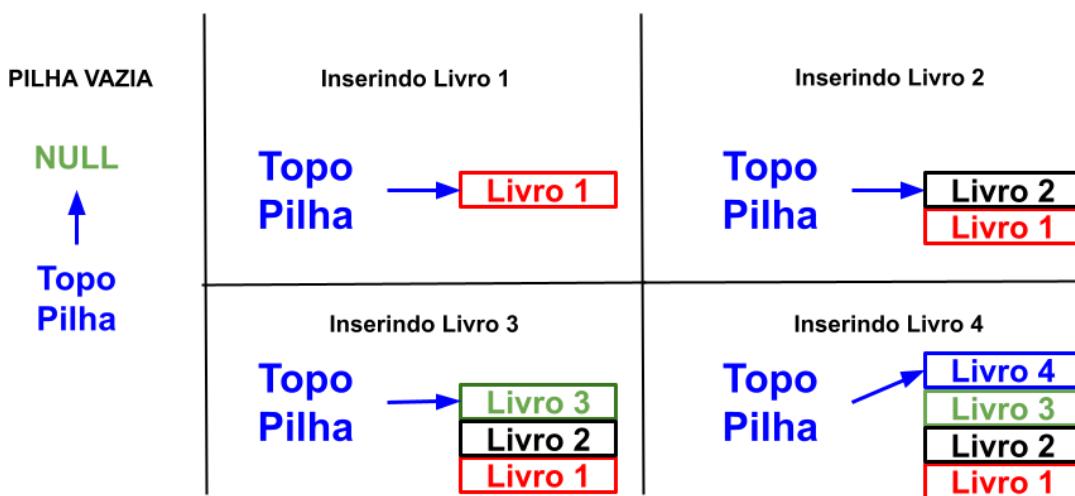
Fazer com lista encadeada

Agora que temos conhecimentos de lista encadeada, conceitos de estrutura de TAD's e estrutura de dados, podemos dar início a outras duas estruturas: Pilha e Fila. Neste capítulo iremos abordar uma Pilha sendo implementada com uma lista encadeada, seus conceitos, quando utilizar e por que utilizá-las.

5.1 Conceito de uma pilha

Uma **pilha** é uma estrutura de dados aplicada para casos específicos com finalidade de obter mais eficácia de acordo com a problemática que será implementada. O funcionamento das TAD's vistas anteriormente nos permitem representar uma grande variedade de situações cotidianas. Já as **pilhas** podem ser caracterizadas como uma TAD mais específica, que busca resolver um subconjunto de problemas que as listas encadeadas (**LE** e **LDE**) também resolvem, porém utilizando menos recursos e buscando trabalhar da forma mais eficiente possível.

As **pilhas** têm grande importância na resolução de problemas caracterizados pela existência do pensamento LIFO (last-in first-out), o primeiro a entrar deverá ser o último a sair. Esse pensamento pode ser caracterizado simplesmente pela ideia de pilhas de objetos no mundo real, uma pilha de livros, por exemplo. Em tal situação, um novo livro deverá ser empilhado no topo e quando for necessário desempilhar a remoção dos livros seja feita na ordem contrária da inserção, o que caracteriza o LIFO. Veja a imagem a seguir:



Os estudos realizados nos capítulos anteriores nos permitiram entender estruturas de dados muito úteis e versáteis (**LE** e **LDE**), que possuem um arsenal enorme de operações (**Inserir** e **Remover**: **início**, meio e **fim**) para resolverem uma gama enorme de problemas. No entanto, nem sempre precisamos utilizar todos os recursos para resolver problemas menores ou que necessitam de abordagens mais específicas para sua resolução.

5.2 Estrutura de uma pilha

```

1 typedef struct no{
2     int id;
3     struct no* prox;
4 }NO;

```

5.3 Implementação de uma pilha

```

1 typedef struct no{
2     int id;
3     struct no* prox;
4 }NO;
5
6 NO* topo=NULL;
7 int tam=0;

```

Figura 63: Inicialização da pilha

topo → NULL
tam = 0

5.3.1 Adicionar em uma pilha

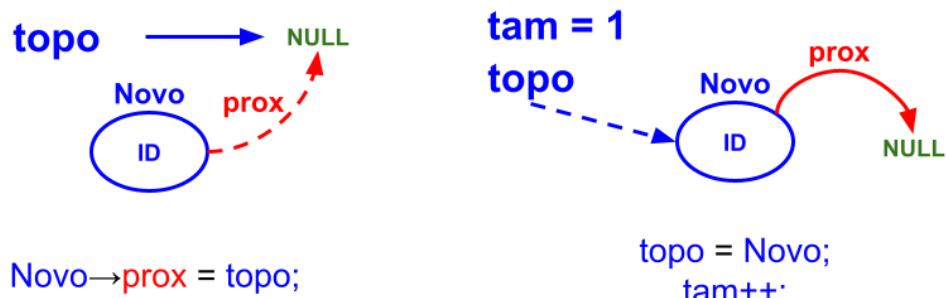
```

1 void adicionar(int id){
2     NO *NOVO=(NO*) malloc (sizeof (NO));
3     NOVO->id=id;
4     NOVO->prox=inicio;
5     inicio=NOVO;
6     tam++;
7 }

```

Aqui vamos explicar inserção do primeiro elemento

Figura 64: Primeira inserção, topo igual a NULL



Aqui vamos explicar o caso que já tem um elemento, é a mesma coisa mas a ideia é dar exemplos de outras inserções para que eu possa construir um desenho com uma sequência. Também será base para realizar remoções.

5.3.2 Remover em uma pilha

```
1 int remover( ){
2     if(tam>0){
3         N0* lixo=topo;
4         topo=topo->prox;
5         int retorno=lixo->valor;
6         tam--;
7         free(lixo)
8         return retorno;
9     }else{
10        printf("A pilha está vazia. \n");
11    }
12 }
```

5.5 Exercícios

CAPÍTULO 6: FILA

6.1 Conceito de uma fila

6.2 Estrutura de uma fila

```
1 typedef struct no{  
2     int valor;  
3     struct no *prox;  
4 }NO;
```

6.3 Implementação de uma fila

```
1 typedef struct no{  
2     int valor;  
3     struct no *prox;  
4 }NO;  
5  
6 NO *inicio = NULL;  
7 NO *fim = NULL;  
8 int tam = 0;
```

6.3.1 Adicionar em uma fila

```
1 void adicionar(int valor){  
2     NO *novo = (NO*) malloc (sizeof (NO));  
3     novo->valor = valor;  
4     novo->prox = NULL;  
5     if(inicio==NULL){  
6         //...  
7     }else if(tam==1){  
8         //...  
9     }  
10    else{  
11        //...  
12    }  
13    tam++;  
14 }
```

```
1 void adicionar(int valor){  
2     NO *novo = (NO*) malloc (sizeof (NO));  
3     novo->valor = valor;  
4     novo->prox = NULL;  
5     if(inicio==NULL){  
6         inicio=novo;  
7         fim=novo;
```

```
1 void adicionar(int valor){  
2     NO *novo = (NO*) malloc (sizeof (NO));  
3     novo->valor = valor;  
4     novo->prox = NULL;  
5     if(inicio==NULL){  
6         inicio=novo;  
7         fim=novo;  
8     }else if(tam==1){  
9         inicio->prox=novo;  
10        fim=novo;  
11    }  
12    else{  
13        fim->prox=novo;  
14        fim=novo;  
15    }  
16    tam++;  
17 }
```

6.3.2 Remover em uma fila

```
1 int remover( ){  
2     //inicio  
3     int lix=inicio;  
4     NO*lixo=inicio;  
5     inicio=inicio->prox;  
6     free(lixo);  
7     return lix;  
8 }
```

6.5 Exercícios

CAPÍTULO 7: ÁRVORE BINÁRIA

Entendido as TAD's ensinadas até agora, devemos iniciar o estudo da estrutura mais **importante** em Estrutura de Dados. A essa altura do campeonato, vocês já devem ter noção da utilidade de listas, pilhas e filas no dia a dia de um profissional da TI. Porém, se levarmos em conta a existência de softwares que necessitam de um grande banco de dados, essas simples TAD's podem não ser o suficiente para a satisfação de um cliente.

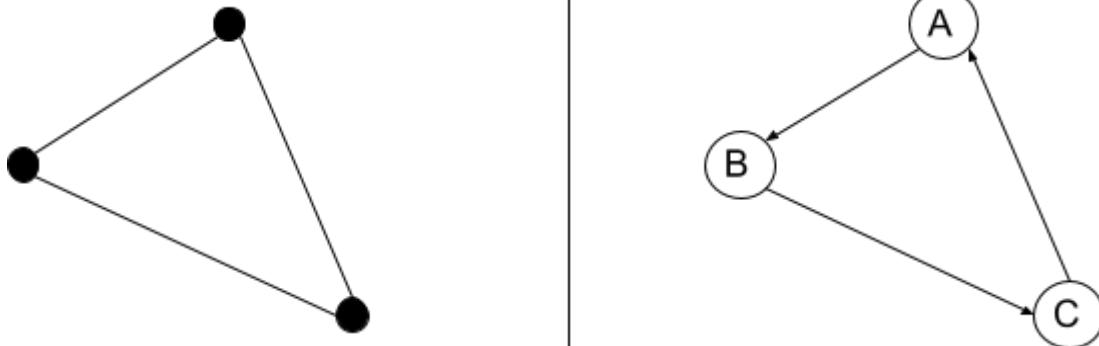
Peguemos um exemplo claro. Pense em um caso bem comum na atualidade, as redes sociais (Instagram, Facebook, TikTok, etc.). Considere que cada conta criada nessas redes são *Nos* em uma estrutura de dados. Então, se considerarmos que essas contas podem se conectar uma com a outra a partir de funcionalidades dentro de cada aplicativo (amizade, *followers*, etc.), devemos ligar cada um desses *Nos* entre si, com uma estrutura ideal que satisfaça as condições. Agora, pense que vamos utilizar uma LDE para esse caso. Você tem certeza de que, por exemplo, para remover uma dessas contas, ou seja, um desses *Nos* da lista, você iria percorrer toda a lista atrás de removê-lo?

Levando em conta a imensa quantidade de elementos dessa lista, sua resposta provavelmente é não. Então, para sanar esse problema, vamos lhe apresentar o conceito de **Árvores Binárias**. Mas, primeiro, devemos conhecer os conceitos básicos de Grafos e Árvores, para entendermos melhor como elas funcionam, visto que elas têm uma lógica diferente das estruturas anteriores.

7.1: Grafos

Antes de começar a falar sobre eles, os **Grafos** possuem disciplinas e livros voltados especialmente para os estudos deles. Então, caso tenha interesse, é importante buscar aprender essas estruturas, pois elas são muito utilizadas em toda a área da TI. Além disso, é importante lembrar que este livro busca facilitar a ti o ensinamento de Estrutura de Dados, ou seja, pode haver conceitos ou aspectos de grafos que serão mostrados com nomes ou formas diferentes de acordo com a situação, para facilitar o entendimento do que a gente mais precisa compreender agora, que são as **Árvores**. Sabendo disso, devemos começar nossos estudos sobre grafos.

Os que já sabem o que são grafos devem ter percebido que já utilizamos e muito deles neste livro. A própria capa do livro possui um grafo, e você pode não ter percebido. Sem enrolar, um **Grafo** é uma estrutura composta por Vértices e Arestas, em que, para ser considerada um grafo, ela deve possuir pelo menos um vértice, não necessitando a existência das arestas (sim, um simples ponto no papel branco é considerado um grafo). Para melhor entendimento, veja a figura abaixo, que representa dois grafos quase idênticos.

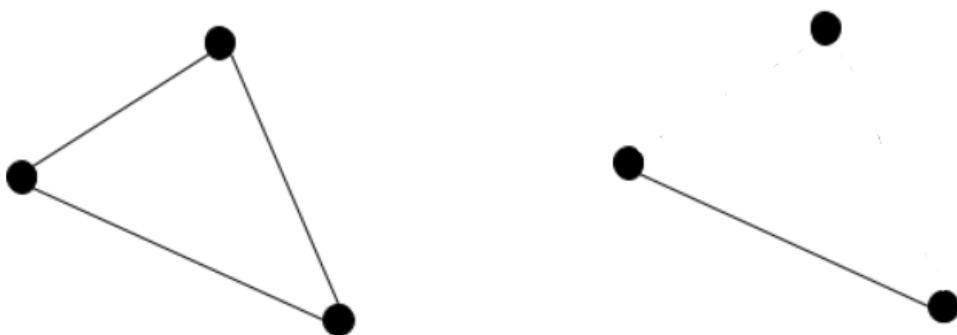
Figura 65

Olhe primeiramente para a estrutura à esquerda. Os pontos pretos representam os vértices, enquanto as linhas representam as arestas.

Agora compare com a estrutura à direita. Os dois representam o mesmo grafo, com uma única diferença que será abordada mais tarde. Neste tópico, será utilizado o padrão da imagem direita. Isso porque utilizaremos os vértices como se fossem nossos nós das árvores, e as arestas, os ponteiros da TAD.

7.1.1 Grafos conexos e desconexos

Vamos pegar novamente o primeiro exemplo:



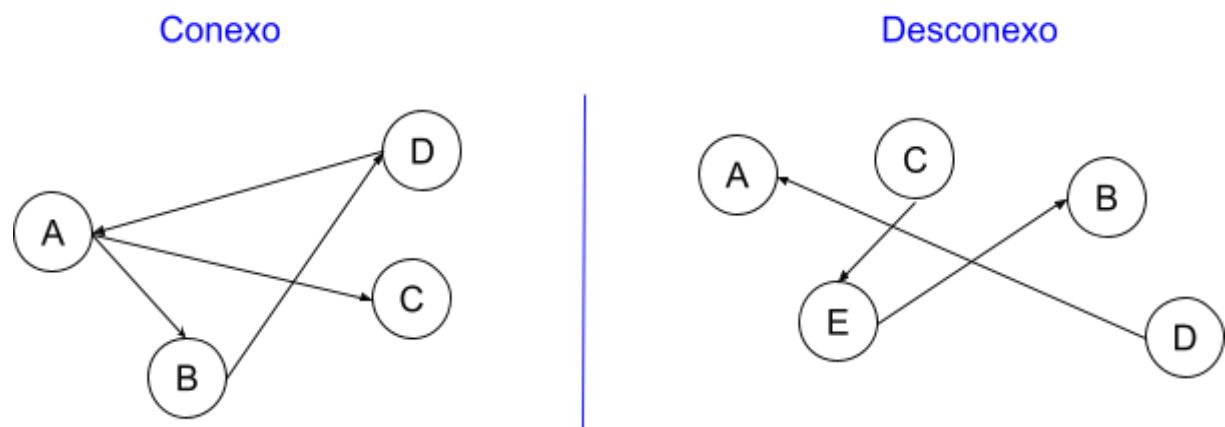
Perceba que removemos duas arestas da figura à esquerda, e criamos um novo grafo à direita, que dessa vez possui apenas uma aresta. Note que, ao tirar essas duas arestas, o grafo se dividiu em duas partes. Atenção para não se confundir! O grafo da direita não se dividiu em dois grafos, apenas em duas partes. Note também

que, sem remover qualquer aresta ou vértice, não conseguimos dividir a figura da esquerda em mais de uma parte. É isso que determina se um grafo é **conexo** ou **desconexo**.

Explicando mais claramente, um grafo é dito conexo quando, caminhando a partir de um vértice qualquer, você consegue chegar a **qualquer outro vértice do grafo** a partir das arestas. Se conseguir fazer tal caminho, independente do número de vezes que tiver que passar por uma aresta, esse grafo é conexo. Já os grafos desconexos são exatamente o oposto, ou seja, haverá pelo menos um vértice que você não alcançará, independente do quanto você caminhar pelas arestas.

Eis outro exemplo de um grafo conexo e um desconexo:

Figura 66



Atenção também às arestas que passam por cima de outras. É comum a confusão de considerar que, quando uma aresta passa por dentro de outra, a parte onde elas se encontram seja um vértice. Porém, isso é um mito, ou seja, nada se cria quando duas arestas se encontram.

7.1.2 Grafo direcionado e não direcionado

Veja novamente a Figura 65. Se olhar bem, percebe-se uma diferença realmente significativa no grafo da esquerda para o grafo da direita. O grafo da direita possui arestas **direcionadas**, enquanto o grafo da esquerda possui arestas **não direcionadas**. Isto é, enquanto as arestas da estrutura da direita possuem setas, as da estrutura esquerda não. Um grafo é dito direcionado caso seja estruturado com arestas direcionadas, e o contrário acontece com um grafo não direcionado.

Quando um grafo é não direcionado, consideramos que podemos fazer um caminho de ida e volta em cada aresta desse grafo, ou seja, diferente do grafo direcionado, em que as arestas só levam para apenas um vértice, as arestas do grafo não direcionado podem ir e voltar para até dois vértices diferentes.

Figura 67



A figura acima mostra que duas arestas direcionadas, uma indo e outra voltando entre dois vértices, são equivalentes a uma aresta não direcionada entre os mesmos vértices. Lembrar que equivalência não é o mesmo que igualdade.

7.1.3 Grafos cíclicos e acíclicos

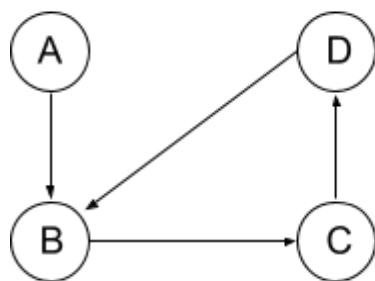
Para facilitar o entendimento de ciclos em grafos, vamos entender um pouco sobre caminhos.

Um caminho em um grafo é qualquer segmento que se possa fazer de um vértice para outro, seguindo de aresta em aresta, com a importante característica de que neles **os vértices não podem se repetir**. Por exemplo, na figura 65, podemos fazer alguns caminhos no grafo da direita. Caminho de A para B, de B para C, de C para A, de A para B e depois para C, de B para C e depois para A, enfim, esses todos são caminhos possíveis.

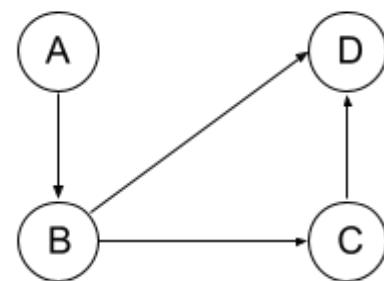
Perceba, ainda olhando para o grafo da direita da figura 65, que temos um caminho especial, que vai de A para B para C e depois de volta para A. Esse tipo de caminho, em que o vértice inicial é o mesmo que o vértice final, é chamado de **ciclo**, e grafos que possuem ciclos são chamados de **grafos cíclicos**. O inverso ocorre com os **grafos acíclicos**.

Figura 68

Cíclico



Acíclico



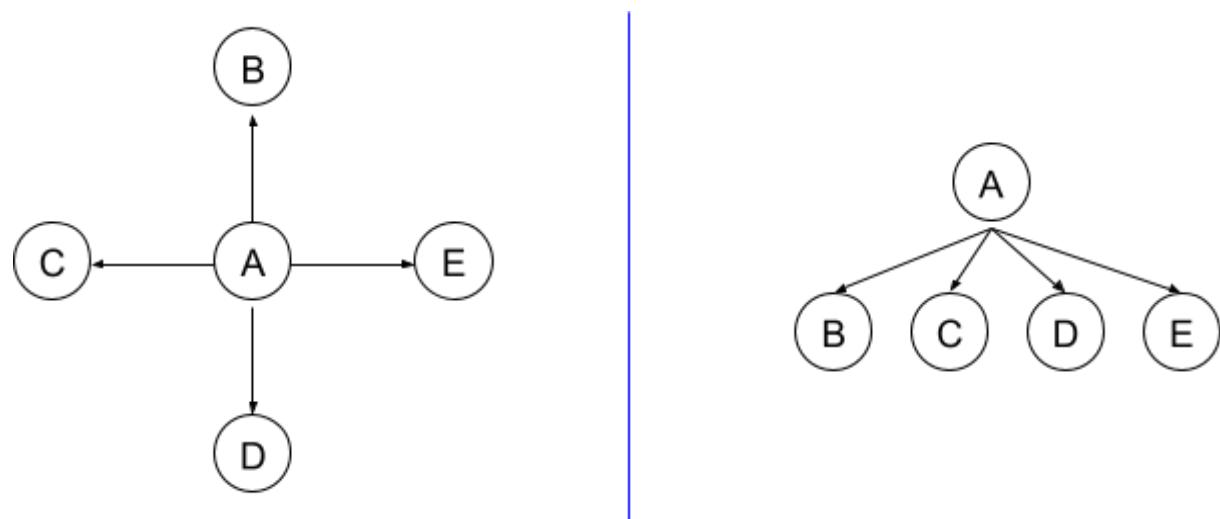
Devemos ter muito cuidado ao analisar se um grafo é cíclico ou acíclico. Perceba que uma simples inversão de orientação de uma aresta em um grafo pode mudar essa característica cíclica dele.

7.2: Conceitos de árvore e árvore binária

Compreendido alguns conceitos de grafos, vamos partir para o assunto principal.

Uma árvore em estrutura de dados é uma estrutura que herda as características da topologia em árvores. Ou seja, diferente das listas, pilhas e filas, onde os dados se encontravam em uma sequência, as árvores possuem ramificações que levam aos nós dessa TAD. Veja um exemplo abaixo para facilitar a compreensão.

Figura 69



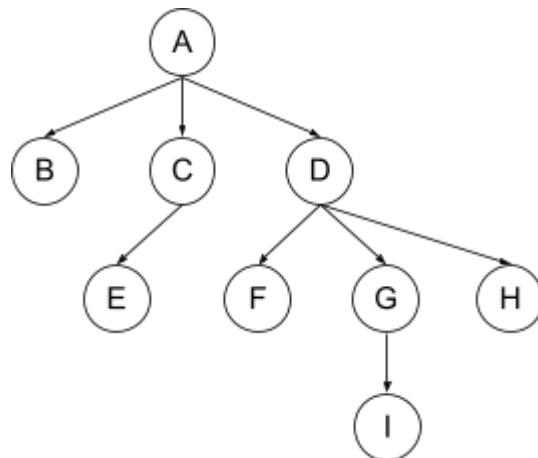
A figura acima mostra **o mesmo grafo**, porém, representado de formas diferentes. Independentemente de como ele está sendo representado, esse grafo é uma árvore. Normalmente, as árvores estão no formato do grafo à direita, pois facilita a compreensão (se você pôr essa imagem de cabeça para baixo, perceberá que ela parece um pouco com uma árvore).

Falando em relação a grafos, as árvores são grafos **acíclicos, conexos** e, pelo menos nessa disciplina, todas elas serão **direcionadas**.

As árvores, assim como as outras TAD's, possuem alguns elementos que ganham certas nomeações.

As **raízes** são o primeiro [No](#) de cada árvore. A partir da raiz é que vão surgir as ramificações para o próximo nó. Na figura anterior, a raiz da árvore é o [No A](#).

Os elementos que se ramificam para outros elementos são chamados de **pais**, enquanto os elementos que provêm de outros elementos são chamados de **filhos**. A raiz da árvore nunca será filho de ninguém, enquanto os outros elementos sempre serão filhos de alguém. Além disso, os elementos que não possuem filhos são chamados de **folhas**. No exemplo anterior, apenas a raiz da árvore possuía filhos, ou seja, todos os outros [Nos](#) eram folhas e filhos da raiz. A próxima figura irá mostrar um exemplo de árvore com vários pais e filhos diferentes.

Figura 70

Por fim, é importante saber sobre a **profundidade** de uma árvore.

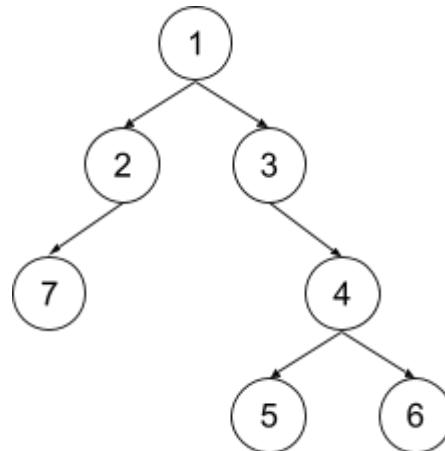
Perceba na Figura 70 que a árvore mostrada vai ficando cada vez maior, mesmo que se direcionando para baixo. Perceba também que esse tamanho muda de acordo com os filhos da árvore. Porém, ele não muda devido a quantidade de filhos, mas sim devido ao **nível** que cada filho pertence.

Pode ter ficado confuso, então vamos esclarecer. Suponha que a raiz da árvore seja o **nível 0** dela. Logo, todos os filhos da raiz serão de **nível 1**, os filhos dos filhos (pode-se associar com netos) da raiz serão de **nível 2**, os filhos dos filhos dos filhos (bisnetos) da raiz serão de **nível 3**, e por aí vai.

Essa definição de profundidade de uma árvore será importante para entender um certo conceito da **Árvore Binária**, que estudaremos a seguir.

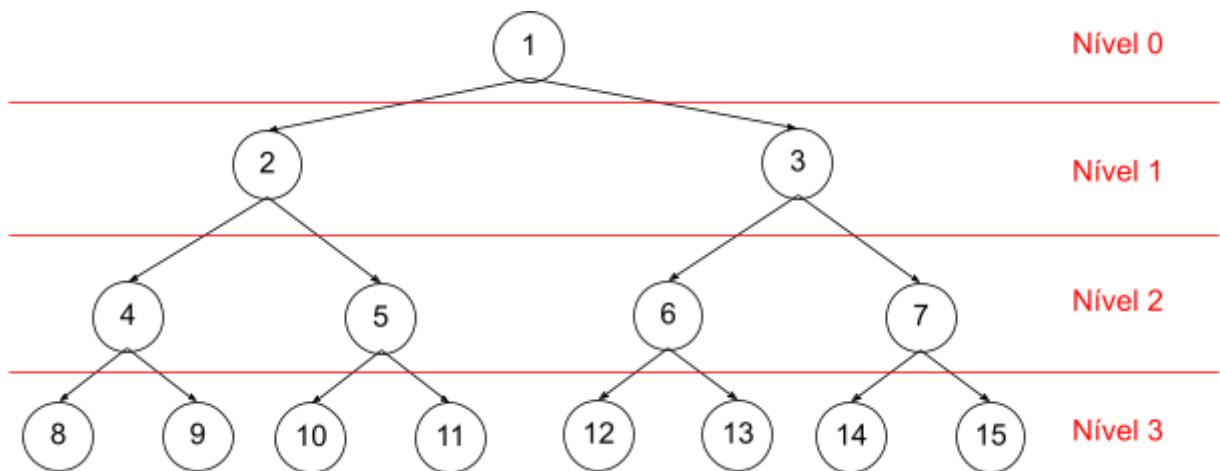
7.3: Árvores Binárias

Indo direto ao ponto, **Árvores Binárias (AB)** são todas aquelas árvores que possuem pais com, no máximo, 2 filhos. Simples assim. Porém, mesmo tendo pouca diferença em relação às outras árvores, as árvores binárias possuem diversas aplicações no dia a dia, e são um dos assuntos que **não podem e nem devem** sair da cabeça de um bom profissional da área de TI.

Figura 71: Exemplo de uma Árvore Binária

Devido à sua binariedade, esse tipo de árvore possui um conceito bem interessante em relação à profundidade que as outras árvores não têm.

Examine bem a imagem abaixo, que mostra uma Árvore Binária **Completa**.

Figura 72

Provavelmente você deve já deve ter notado que uma árvore binária completa é uma árvore binária em que **todos os pais possuem o máximo de filhos (2 filhos)**. Pode não parecer muita coisa, mas esse tipo de árvore possui algumas características únicas, tais como:

- ❖ Não haverá nenhuma folha que não esteja no último nível de profundidade da árvore. Ou seja, todas as folhas dela estarão no último nível.

- ❖ O número máximo de *Nos* em um nível n de uma árvore binária qualquer pode ser calculado por 2^n . Logo, em uma árvore binária completa, o resultado desse cálculo é o número exato de nós desse nível da árvore.

CAPÍTULO 8: ÁRVORE BINÁRIA DE BUSCA
