ESTRUTURA DE DADOS

AULA 3

Prof. Vinicius Pozzobon Borin

CONVERSA INICIAL

Listas encadeadas, pilhas e filas

O objetivo desta aula é apresentar os conceitos que envolvem a estrutura de dados do tipo lista e ao longo da qual serão examinados diferentes tipos de listas. Também será mostrado como realizar manipulações dos dados em uma estrutura de lista, como a inserção e a remoção dos dados. Os tipos de lista e suas características serão:

- Lista simplesmente encadeada (circular e não circular);
- Lista duplamente encadeada (circular e não circular).

Também serão apresentadas outras duas estruturas de dados que podem ser construídas com base em listas, mas que mantêm características únicas de operação:

- Pilhas;
- Filas.

Todos os conceitos trabalhados anteriormente – como a análise assintótica e a recursividade, bem como conceitos básicos de programação estruturada e manipulação de ponteiros, aparecerão de forma recorrente ao longo de toda esta aula.

Todos os códigos analisados ao longo desta aula estarão na forma de pseudocódigos. Para a manipulação de ponteiros e endereços em pseudocódigo, será adotada a seguinte nomenclatura, ao longo de todo este material:

- Para indicar o endereço da variável, será adotado o símbolo $(->]$ **antes do nome da variável**. Por exemplo: $px = (->]x$. Isto significa que a variável *px* recebe o endereço da variável *x*.
- Para indicar um ponteiro, será adotado o símbolo $[->)$ **após o nome da variável.** Por exemplo: $x(->]$: inteiro. Isto significa que a variável *x* é uma variável do tipo ponteiro de inteiros.

TEMA 1 – CONCEITO DE LISTAS ENCADEADAS

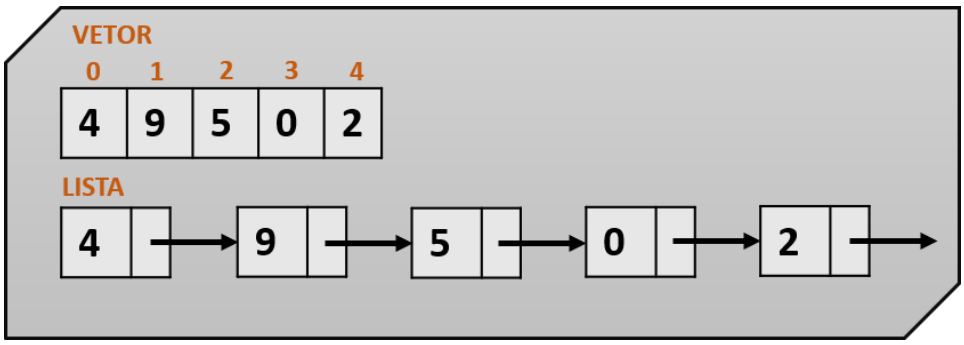
Conforme foi visto anteriormente, existem estruturas de dados com alocação sequencial. Vetores e matrizes, por exemplo, são estruturas homogêneas e registros são estruturas heterogêneas, todos com alocação sequencial. Contudo, nem todas as estruturas de dados são sequenciais. Existem estruturas de dados com alocação não sequencial.

Em estruturas de dados com alocação não sequencial, cada dado (ou conjunto de dados) pode estar posicionado em qualquer parte da memória destinada a um programa em execução. Se isso é verdade, surge uma dúvida: como o programa consegue localizar cada dado da estrutura, uma vez que eles estão posicionados aleatoriamente na memória? A solução para esse problema foi encontrada com o uso de **ponteiros**.

Em uma lista, cada elemento conterá um dado (ou um conjunto de dados). Esse(s) dado(s) poderá(ão) ser de qualquer tipo: numérico(s), caractere(s), lógico(s) etc. Mesmo com alocação em não sequência, pode ocorrer a alocação de conjuntos de dados de tipos diferentes. Nesse caso, teremos uma estrutura de dados heterogênica não sequencial. Além dos dados úteis, cada elemento da lista ainda precisa manter pelo menos um endereço armazenado. Esse endereço corresponderá ao endereço do próximo elemento da lista. Dessa forma, cada elemento da lista sabe onde o próximo está e todos estão conectados (virtualmente) por meio desses endereços, ou ponteiros, armazenados.

Na Figura 1 temos uma ilustração da estrutura de um vetor de inteiros, comparado com uma estrutura de lista também com valores inteiros. Em um vetor, um dado é acessível pelo seu índice (colocado em laranja na Figura 1). Para acessar o terceiro dado desse vetor, basta fazermos uma chamada no código com $leia(Vetor[2])$, e o valor 5 será encontrado.

Figura 1 – Comparativo das estruturas do vetor e da lista encadeada



Para uma lista, o conceito de índice é inexistente. Perceba que, na Figura 1, não foi apresentada a numeração dos índices na lista. A inexistência de um índice fomenta uma nova questão: como acessamos cada dado de uma determinada lista?

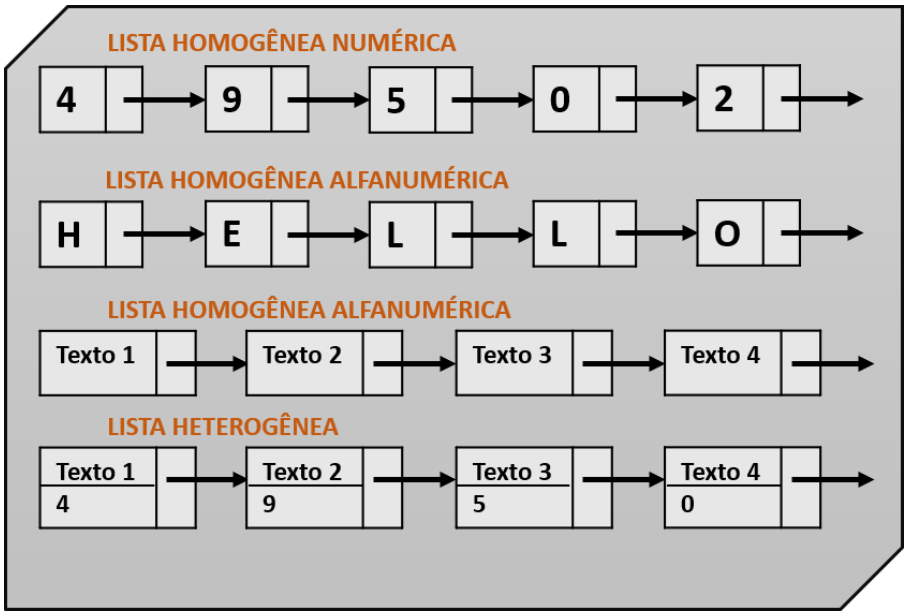
Em uma lista, cada elemento contém o endereço do próximo elemento (representado pela flecha/ponteiro na Figura 1). Portanto, somente o elemento atual conhece o subsequente. Assim, a única forma de acessarmos, por exemplo, o terceiro elemento dessa lista seria iniciarmos no primeiro elemento e, por meio do acesso aos endereços para os próximos elementos, passarmos de um elemento para o seguinte até atingirmos o elemento desejado. Esse tipo de lista é chamado de *encadeado* por esse motivo, já que os elementos estão encadeados por meio do endereço do elemento seguinte.

Uma das grandes vantagens do uso de listas encadeadas está na possibilidade de alocação dinâmica de memória e no dinamismo de manipulação dos dados dessa lista, o que torna fáceis a inserção e a remoção de novos elementos.

Dentre as desvantagens das listas está o desempenho para acesso aos dados. Um vetor, por exemplo, apresenta o mesmo tempo de resposta para acessar qualquer dado da sua estrutura ($O(1)$ – conforme visto antes). Uma lista, conforme será mostrado no Tema 2, precisa sempre varrer os elementos da estrutura até encontrar o que precisa; assim, o tempo de acesso ao dado não é constante.

As listas podem ser homogêneas ou heterogêneas e armazenar qualquer tipo de dado. Na Figura 2 temos alguns exemplos de listas e dados armazenados.

Figura 2 – Exemplos de listas encadeadas homogêneas e heterogêneas



As listas têm aplicação em várias áreas da computação e são utilizadas para resolver alguns problemas que encontramos no dia a dia, como:

- Agenda de contato. Utilizamos estruturas de lista para armazenar contatos de agenda envolvendo dados pessoais, endereços, telefones etc.;
- Programa de solução de equações matemáticas. Podemos manipular uma equação matemática (de qualquer grau) em uma lista e realizar operações matemáticas com ela;
- Tocador de música. Reproduzimos músicas, com uma listagem de músicas, seus dados, e podemos avançar ou retroceder nas músicas executadas. Isso pode ser manipulado em forma de uma estrutura do tipo lista.

TEMA 2 – 2 LISTA SIMPLESMENTE ENCADEADA (*LINKED LIST*)

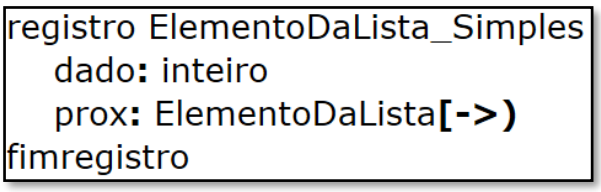
A lista que vimos até agora, lista encadeada, também é conhecida como *lista simplesmente encadeada*, pois cada elemento dela aponta e conhece somente o próximo elemento da lista. Para tal, cada nó necessita ter um único ponteiro, com o endereço do próximo elemento.

Em programação, podemos representar cada elemento da lista encadeada como sendo um registro. Esse registro conterá todos os dados que se deseja armazenar, sejam quais forem seus tipos, e também um ponteiro. Esse

ponteiro conterá o endereço para o próximo elemento da lista. O tipo desse ponteiro deverá ser igual ao do registro criado.

Na Figura 3 temos um exemplo de registro para uma lista simplesmente encadeada. Observe que temos um dado numérico e o ponteiro para o próximo elemento. Esse ponteiro é do tipo *ElementoDaLista*.

Figura 3 – Pseudocódigo de criação de um elemento (nó) da lista simplesmente encadeada



A lista encadeada simples pode ser classificada em dois tipos: as **não circulares** e as **circulares**.

Antes de entrarmos diretamente no assunto dos algoritmos de listas, vamos conceituar e diferenciar as listas encadeadas simples. Cada elemento (ou nó) de uma lista conterá um dado (ou um conjunto de dados) e um ponteiro com o endereço para o próximo elemento da lista. A lista encadeada simples funciona como se fosse uma via de mão única. Sendo assim, cada elemento conhece, só e somente só, o elemento subsequente. Uma vez que o programa tenha começado a percorrer a lista, não é possível retornar para o elemento imediatamente anterior.

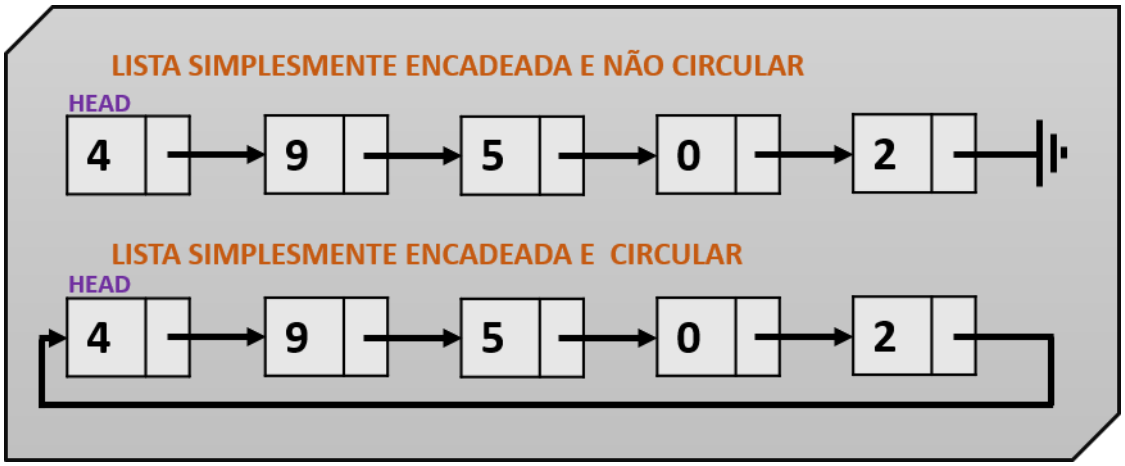
O primeiro elemento da lista é chamado de *início* ou de *head* (*cabeça*) da lista. Esse primeiro elemento é o único que sempre será conhecido pelo programa que está manipulando uma determinada lista. Todos os outros dados da lista são descobertos à medida que os elementos da lista vão sendo acessados.

Em uma **lista simples não circular**, o último elemento não apontará para nada (ponteiro nulo). Na Figura 5 temos a representação das listas encadeadas simples e circular. Observe que o último elemento contém um símbolo que corresponde ao nulo e que esse elemento não se conecta.

De uma forma diferente, a **lista simples circular** difere da não circular somente pelo ponteiro do último nó. Esse ponteiro, ao invés de apontar para o nulo, irá apontar para o primeiro elemento da lista, fechando um círculo. Novamente, observe a Figura 5.

O uso de uma lista circular torna mais fáceis certas aplicações. Por exemplo, em um reprodutor de música, quando você está reproduzindo sua *playlist* e chega ao seu final, se você a desenvolver com uma lista circular você poderá facilmente atingir a última música e retornar para a primeira.

Figura 5 – Estrutura das listas simplesmente encadeadas não circular e circular



Neste Tema 2 iremos entender como inserir dados em uma lista simples encadeada. Para tal, serão apresentadas funções que realizam essas tarefas.

Iniciaremos nossa análise com um algoritmo principal. Esse algoritmo conterá um menu de interação com o usuário. Com base nesse programa principal, podemos realizar as chamadas de inserção de dados na estrutura de dados proposta, a lista encadeada, considerando isso e também que modular um código é sempre uma ótima prática de programação, pois deixa o código mais legível ao usuário, melhorando o entendimento e tornando seu manuseio melhor. Portanto, todos os códigos propostos serão exibidos no formato de funções ou procedimentos chamados pelo algoritmo principal.

Na Figura 6, temos o pseudocódigo principal. Nesse algoritmo, há a criação do registro que define como serão os elementos de nossa lista, bem como a declaração do *head*, único elemento sempre conhecido globalmente pelo programa.

No código é mostrado um seletor do tipo **escolha**, que realiza a chamada de três funções distintas: uma para inserir um novo elemento no início da lista, outra para inseri-lo no final da lista e mais uma para inseri-lo em qualquer posição, no meio da nossa lista simples. Essas funções serão mais bem apresentadas nas próximas subseções.

Figura 6 – Pseudocódigo principal que declara a lista e chama as funções de inserção

```
1   algoritmo "ListaMenu"
2   var
3      //Cria uma lista simples encadeada
4      registro ElementoDaLista_Simples
5         dado: inteiro
6         prox: ElementoDaLista[->)
7      fimregistro
8      //Declara o Head como sendo do tipo da lista
9      Head: registro ElementoDaLista_Simples
10     op, numero, posicao: inteiro
11  inicio
12        //Lê um valor para inserir e a operação desejada
13        leia(numero)
14        leia(op)
15
16        escolha (op) //Escolhe o que deseja fazer
17           caso 0:
18              InserirInicio(numero)
19           caso 1:
20              InserirFim(numero)
21           caso 2:
22              leia(posicao)
23              InserirMeio(numero, posicao)
24        fimescolha
25  fimalgoritmo
```

Outras manipulações da nossa lista, como a remoção e a listagem de elementos, não serão exploradas nesta aula, mas é de suma importância que sejam estudadas também. Por favor, realize a leitura sobre esse tópico no livro de estrutura de dados de Ascencio e Araújo (2011).

2.1 Inserindo um novo elemento no início da lista encadeada simples não circular

Para que consigamos colocar um novo elemento na lista, seguimos algumas etapas predefinidas. A inserção de um novo elemento em uma lista encadeada é um processo que acontece seguindo uma sequência-padrão de passos. Portanto, nesta subseção será explorado como inserimos um dado no início de uma lista encadeada. O processo de inserção seguirá sempre o que será descrito a seguir.

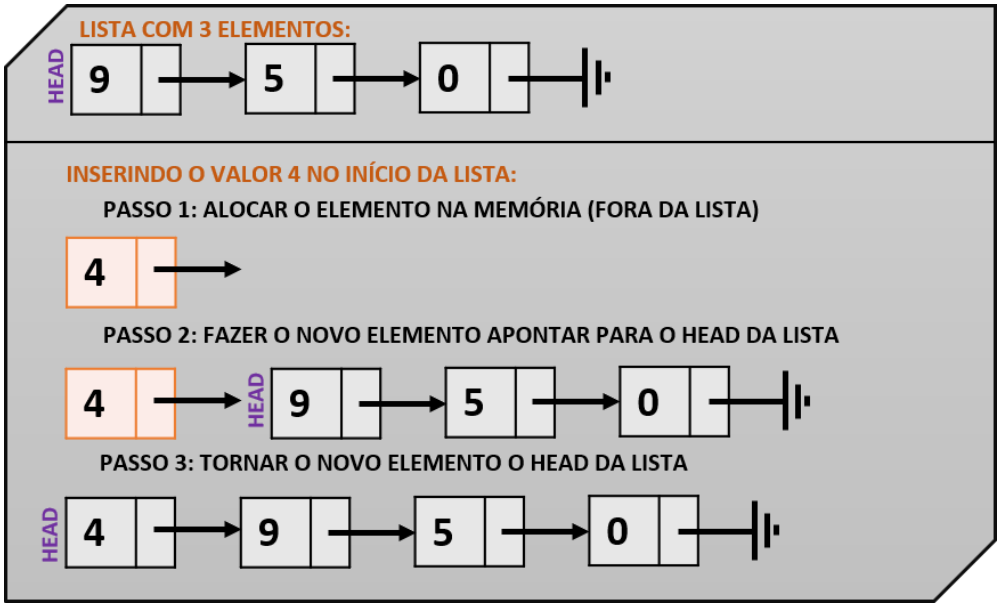
A Figura 7 apresenta a inserção de um elemento no início da nossa lista encadeada simples. Observe, na imagem, que temos uma lista já construída,

com três elementos numéricos, lembrando que o último elemento aponta para o nulo.

Agora, queremos inserir um novo elemento (valor 4), no início dessa lista. Inserir no início significa inserir antes do *head*. Em nosso exemplo, o *head* é o valor 9. Portanto, o 4 deve vir antes do 9. Os passos para a realização dessa tarefa são:

- **Passo 1**: alocar o novo elemento na memória. Perceba que alocá-lo na memória ainda não significa inseri-lo na lista. Nesse momento, ele ainda está fora da lista.
- **Passo 2**: fazer o ponteiro do novo elemento apontar para o *head*.
- **Passo 3**: transformar o novo elemento no novo *head* da lista.

Figura 7 – Funcionamento da inserção de um elemento no início de uma lista encadeada simples



Na Figura 8, temos o pseudocódigo de inserção no início da lista encadeada simples. Os passos descritos anteriormente estão listados no código e comentados. Observe que, no algoritmo, temos uma etapa a mais, que é a verificação, utilizando uma condicional composta, do *head*. Se o *head* estiver vazio, significa que nossa lista está vazia. Então, nosso elemento será inserido no *head* e ele próprio será o início e o fim da lista, ao mesmo tempo. A inserção antes do *head* só acontece caso ele não esteja vazio.

Figura 8 – Pseudocódigo de inserção de elemento no início de uma lista encadeada simples

```
1   //Cria um Procedimento que recebe como parâmetro o dado a ser inserido no inicio
2   função InserirInicio (numero: inteiro): sem retorno
3   var
4      //Declara um novo nó do tipo REGISTRO
5      NovoElemento[->): registro ElementoDaLista_Simples
6   inicio
7   //Insere no novo nó o dado recebido como parâmetro
8      NovoElemento->dado = numero
9
10     //Verifica se o HEAD está vazio
11     se (Head == NULO)  então
12        //Se HEAD está vazio, a lista está vazia!
13        //Portanto, novo elemento será o HEAD!
14        Head = NovoElemento
15        Head->prox = NULO
16     senão
17        //Se HEAD não está vazio, existem dados na lista
18        //Portanto, novo elemento aponta para o HEAD
19        NovElemento->prox = Head
20        //Novo elemento vira o HEAD da lista
21        Head = NovoElemento
22     fimse
23  fimfunção
```

2.2 Inserindo um novo elemento no fim da lista encadeada simples não circular

A inserção de um novo elemento em uma lista encadeada é um processo que acontece seguindo uma sequência-padrão de passos. Portanto, nesta subseção será explorado como inserimos um dado no final dessa lista encadeada. O processo de inserção de um novo elemento no final da lista seguirá sempre o que será descrito a seguir.

De modo geral, como conhecemos somente o início da nossa fila, precisaremos varrê-la até atingir o final dela. Portanto, iniciamos no *head* e vamos adiante até encontrarmos o elemento com um ponteiro nulo. Esse é o final da fila.

A Figura 9 apresenta a inserção de um elemento no final da lista encadeada simples. Observe, na imagem, que temos uma lista já construída, com três elementos numéricos, lembrando que o último elemento aponta para o nulo.

Agora, queremos inserir um novo elemento (valor 4), no fim dessa lista. Inseri-lo no final significa inseri-lo após o elemento da lista que aponta para o

nulo. Em nosso exemplo, quem aponta para o nulo é o elemento 0. Portanto, o 4 deve vir após o 0. Os passos para a realização dessa tarefa são:

- **Passo 1**: alocar o novo elemento na memória. Perceba que alocá-lo na memória ainda não significa inseri-lo na lista. Nesse momento ele ainda está fora da lista.
- **Passo 2**: realizar uma varredura, usando um laço de repetição e começando no *head*, até localizar um elemento que aponte para o nulo. Esse elemento será o último e a inserção do novo elemento ocorrerá após ele.
- **Passo 3**: fazer esse último elemento apontar para o nosso novo valor 4. O novo valor, como será então o último, irá apontar para o nulo.

Figura 9 – Funcionamento da inserção de um elemento no final de uma lista encadeada simples

Na Figura 10 temos o pseudocódigo de inserção no fim da lista encadeada simples. Os passos descritos anteriormente estão listados no código e comentados. Observe que, no algoritmo, temos uma etapa a mais que é a verificação, utilizando uma condicional composta, do *head*. Se o *head* estiver vazio, significa que nossa lista está vazia. Então nosso elemento será inserido

no *head* e ele próprio será o início e o fim da lista, ao mesmo tempo. A inserção depois do *head* só acontece caso ele não esteja vazio.

Observe também a declaração de duas variáveis do tipo registro. Uma delas será de fato o novo elemento que será alocado na lista. A outra, chamada *ElementoVarredura*¸ servirá como variável temporária de varredura da lista. Ou seja, ela irá passar no laço de repetição por todos os elementos, até chegar ao ponteiro. Somente aí fará o ponteiro nulo apontar para o novo elemento, nesse caso o com valor 4.

Figura 10 – Pseudocódigo de inserção de um novo elemento no final de uma lista encadeada simples

```
1   //Cria um Procedimento que recebe como parâmetro o dado a ser inserido no final
2   função InserirFim (numero: inteiro): sem retorno
3   var
4       //Declara um novo nó do tipo REGISTRO
5       NovoElemento[->): registro ElementoDaLista_Simples
6       //Declara um nó para fazer a verredura da lista até localizar o final
7       ElementoVarredura[->): registro ElementoDaLista_Simples
8   inicio
9       //Insere no novo nó o dado recebido como parâmetro
10      NovoElemento->dado = numero
11
12      //Verifica se o HEAD está vazio
13      se (Head == NULO)  então
14          //Se HEAD está vazio, a lista está vazia!
15          //Portanto, novo elemento será o HEAD!
16          Head = NovoElemento
17          Head->prox = NULO
18      senão
19          //Se HEAD não está vazio, existem dados na lista
20          //Portanto, precisamos achar o final da lista (ponteiro nulo)
21          ElementoVarredura = Head
22          //Varre um elemento por vez procurando o ponteiro nulo
23          enquanto (ElementoVarredura->prox <> NULO)
24              ElementoVarredura = ElementoVarredura->prox
25          fimenquanto
26          //Após encontrar o ponteiro nulo,
27          //faz o último elemento da lista apontar para o novo nó
28          ElementoVarredura->prox = NovoElemento
29          //Novo nó agora terá o ponteiro nulo (final da lista)
30          NovoElemento->prox = NULO
31      fimse
32  fimfunção
```

2.3 Inserindo um novo elemento no meio da lista encadeada simples não circular

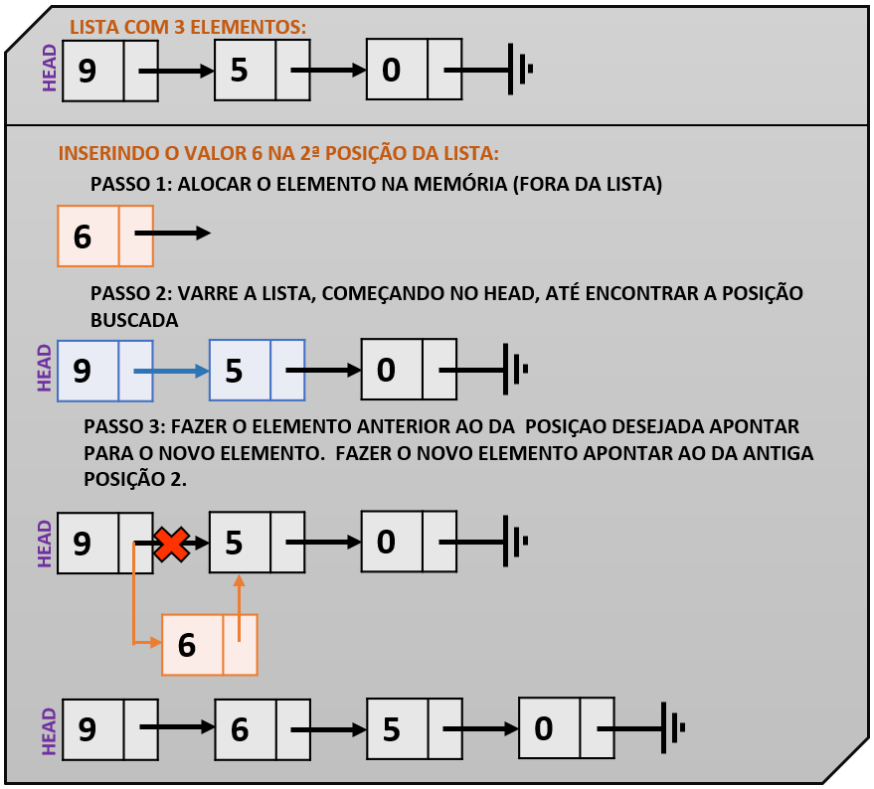
A Figura 11 apresenta a inserção de um novo elemento no meio de uma lista encadeada simples. Observe, na imagem, que temos uma lista já

construída, com três elementos numéricos, lembrando que o último elemento aponta para o nulo.

Agora, queremos inserir um novo elemento (valor 6), no meio dessa lista e na segunda posição, ou seja, entre o 9 e o 5. Inseri-lo no meio significa localizar, por intermédio de uma varredura, a posição desejada e colocar o novo elemento naquela posição, deslocando o elemento atual para a próxima posição. Em nosso exemplo, quem está na posição 2 é o valor 5. Portanto, o 6 deve vir após o 9 e antes do 5. Os passos para a realização dessa tarefa são:

- **Passo 1**: alocar o novo elemento na memória. Perceba que alocá-lo na memória ainda não significa inseri-lo na lista. Nesse momento, ele ainda está fora da lista.
- **Passo 2**: realizar uma varredura, usando um laço de repetição e começando no *head* até localizar a posição 2.
- **Passo 3**: fazer o elemento da posição anterior, posição 1, apontar para o novo elemento e fazer o novo elemento apontar o antigo da posição 2.

Figura 11 – Funcionamento da inserção de um elemento no meio de uma lista encadeada simples



Na Figura 12 temos o pseudocódigo de inserção no meio da lista encadeada simples. Os passos descritos anteriormente estão listados no código

e comentados. Observe que, no algoritmo, temos uma etapa de verificação, utilizando uma condicional composta, para saber se a posição é 0; se for, iremos inserir o elemento no *head*. Se o *head* estiver vazio, significa que nossa lista está vazia. Então nosso elemento será inserido no *head* e ele próprio será o início e o fim da lista, ao mesmo tempo. A inserção do elemento depois do *head* só acontece caso ele não esteja vazio.

Observe também a declaração de três variáveis do tipo registro. Uma delas será de fato o novo elemento que será alocado na lista. A outra, chamada *ElementoVarredura*, serve como variável temporária de varredura da lista. Ou seja, ela irá passar no laço de repetição até atingir a posição desejada. Temos um terceiro elemento, que servirá de variável auxiliar na inserção do novo elemento no meio da lista encadeada.

Figura 12 – Pseudocódigo de inserção do novo elemento no meio da lista encadeada simples

```
1  //Cria um Procedimento que recebe como parâmetro o dado a ser inserido
2  //e a posição da lista que ele será inserido
3  função InserirMeio (numero: inteiro, posicao: inteiro): sem retorno
4  var
5     i: inteiro
6     //Declara um novo nó do tipo REGISTRO
7     NovoElemento[->): registro ElementoDaLista_Simples
8     //Declara um nó para fazer a verredura da lista até localizar o final
9     ElementoVarredura[->): registro ElementoDaLista_Simples
10    //Declara um nó auxiliar para ajudar no armazenamento temporário
11    ElementoAuxiliar[->): registro ElementoDaLista_Simples
12 inicio
13    //Insere no novo nó o dado recebido como parâmetro
14    NovoElemento->dado = numero
15
16    //Verifica se o HEAD está vazio
17    se (posicao == 0)  então
18       //HEAD é a posição 0, então insere nele
19       Head = NovoElemento
20       Head->prox = NULO
21    senão
22       //Qualquer outra posição, insere após o HEAD
23       ElementoVarredura = Head
24       //Varre um elemento por vez até chegar no elemento da posição desejada
25       para i de 0 até posicao faça
26          ElementoVarredura = ElementoVarredura->prox
27       fimpara
28       //Após encontrar a posição desejada
29       //faz uma troca usando um nó auxiliar
30       ElementoAuxiliar = ElementoVarredura->prox
31       ElementoVarredura->prox = NovoElemento
32       NovoElemento->prox = ElementoAuxiliar
33       //Novo nó agora terá o ponteiro nulo (final da lista)
34       NovoElemento->prox = NULO
35    fimse
36 fimfunção
```

TEMA 3 – LISTA DUPLAMENTE ENCADEADA (*DOUBLY LINKED LIST*)

Uma lista duplamente encadeada é assim chamada pois cada elemento dela aponta e conhece o próximo elemento da lista, bem como o elemento imediatamente anterior a ele na lista. Para tal, cada nó necessita ter dois ponteiros (anterior e próximo).

Em programação, podemos representar cada elemento da lista encadeada como sendo um registro. Esse registro conterá todos os dados que se deseja armazenar, sejam quais forem seus tipos, e também os dois ponteiros. Um ponteiro conterá o endereço para o próximo elemento da lista e o outro, o endereço para retornar ao elemento anterior. O tipo desse ponteiro deverá ser igual ao do registro criado. Na Figura 13 temos um exemplo de registro para uma lista duplamente encadeada. Observe que temos um dado numérico, um ponteiro para o próximo elemento e outro ponteiro que irá apontar para o elemento imediatamente anterior a ele.

Figura 13 – Pseudocódigo de criação de um elemento (nó) da lista duplamente encadeada

```
registro ElementoDaLista_Dupla
   dado: inteiro
   prox: ElementoDaLista[->)
   ant: ElementoDaLista[->)
fimregistro
```

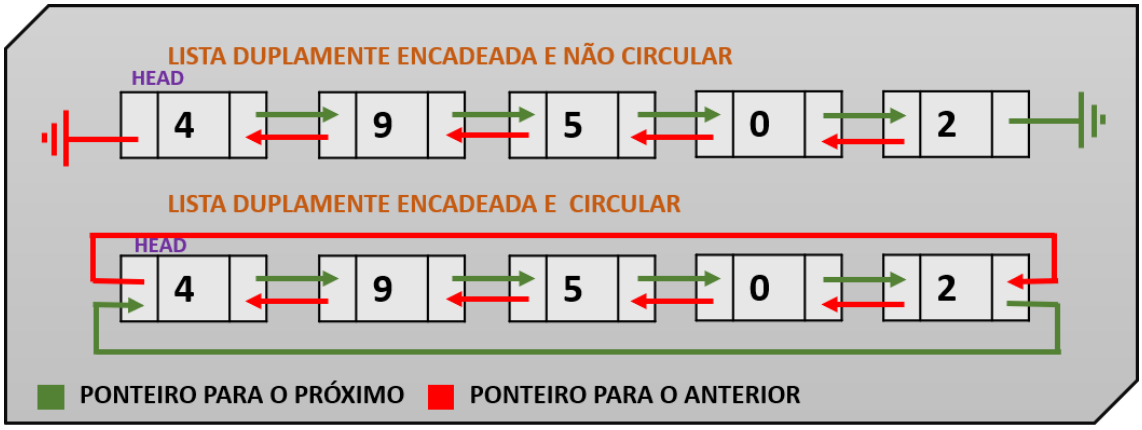
A lista encadeada dupla funciona como se fosse uma via de mão dupla. Sendo assim, cada elemento conhece, somente, o seu subsequente e o seu antecessor, sendo possível ir e voltar na lista, quando necessário.

A lista encadeada dupla pode ser classificada em dois tipos: as **não circulares** e as **circulares**. Neste Tema 3, vamos investigar essas listas, entendendo como funciona o processo de inserção de dados na lista dupla. Os pseudocódigos desse tipo de lista podem ser visualizados no livro-base de Ascencio e Araújo (2011).

Em uma **lista dupla não circular**, o último elemento apontará para o nulo. Na Figura 14 temos a representação das listas encadeadas duplas. Observe que o último elemento contém um símbolo que corresponde ao nulo e não se conecta com nada. O primeiro elemento contém o ponteiro anterior, que também aponta para o nulo.

De uma forma diferente, a **lista dupla circular** difere da não circular somente pelo ponteiro próximo do último nó. Esse ponteiro, ao invés de apontar para o nulo, irá apontar de volta para o primeiro elemento da lista, fechando um círculo novamente (Figura 14). Não obstante, o ponteiro anterior do *head* irá apontar diretamente para o último elemento da lista, fechando mais um círculo.

Figura 14 – Estrutura das listas duplamente encadeadas não circular e circular



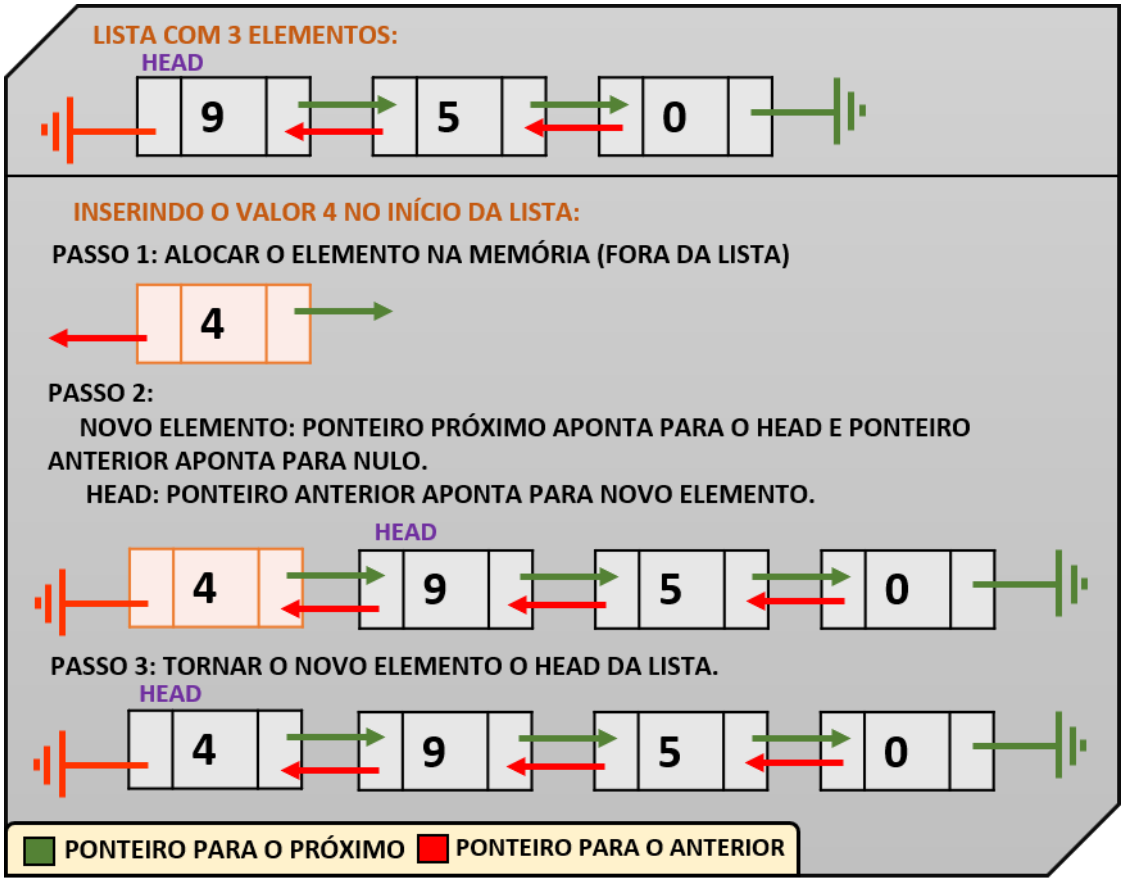
3.1 Inserindo um novo elemento no início da lista encadeada dupla

A Figura 15 apresenta a inserção de um novo elemento no início de uma lista encadeada dupla. Observe, na imagem, que temos uma lista já construída, com três elementos numéricos, lembrando que o ponteiro próximo do último elemento aponta para o nulo, assim como o ponteiro anterior ao *head*.

Agora, queremos inserir um novo elemento (valor 4) no início dessa lista. Inseri-lo no início significa inseri-lo antes do *head*. Em nosso exemplo, o *head* é o valor 9. Portanto, o 4 deve vir antes do 9. Os passos para a realização dessa tarefa são:

- **Passo 1**: alocar o novo elemento na memória. Perceba que alocá-lo na memória ainda não significa inseri-lo na lista. Nesse momento, ele ainda está fora da lista.
- **Passo 2**: fazer o ponteiro próximo do novo elemento apontar para o *head*. Fazer o ponteiro anterior ao novo elemento apontar para o nulo. Fazer o ponteiro anterior ao *head* apontar para o novo elemento.
- **Passo 3**: transformar o novo elemento no novo *head* da lista.

Figura 15 – Funcionamento da inserção de um novo elemento no início de uma lista encadeada dupla



3.2 Inserindo um novo elemento no fim da lista encadeada dupla

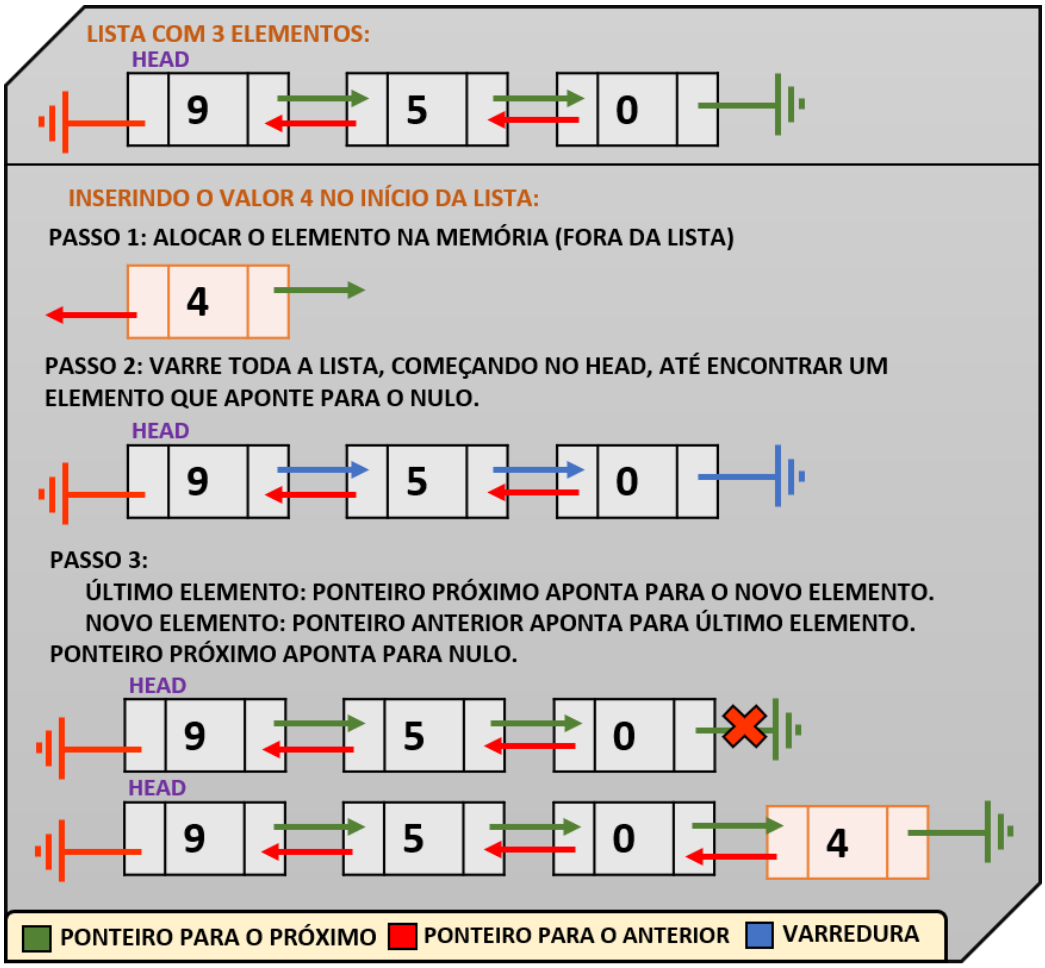
A Figura 16 apresenta a inserção de um elemento no fim de uma lista encadeada dupla. Observe, na imagem, que temos uma lista já construída, com três elementos numéricos, lembrando que o ponteiro próximo do último elemento aponta para o nulo, assim como o ponteiro anterior ao *head*.

Agora, queremos inserir um novo elemento (valor 4) no final dessa lista. Inseri-lo no final significa inseri-lo após o elemento com ponteiro próximo nulo. Em nosso exemplo, o último elemento é o 0. Portanto, o 4 deve vir depois do 0. Os passos para a realização dessa tarefa são:

- **Passo 1**: alocar o novo elemento na memória. Perceba que alocá-lo na memória ainda não significa inseri-lo na lista. Nesse momento, ele ainda está fora da lista.
- **Passo 2**: realizar uma varredura na lista existente, iniciando no *head*, até localizar o elemento com ponteiro próximo nulo (último elemento).

- **Passo 3**: fazer o elemento encontrado, com ponteiro próximo nulo, apontar para o novo elemento. Fazer o ponteiro anterior ao novo elemento apontar para o último elemento. Fazer o ponteiro próximo do novo elemento apontar para o nulo.

Figura 16 – Funcionamento da inserção de um elemento no fim de uma lista encadeada dupla



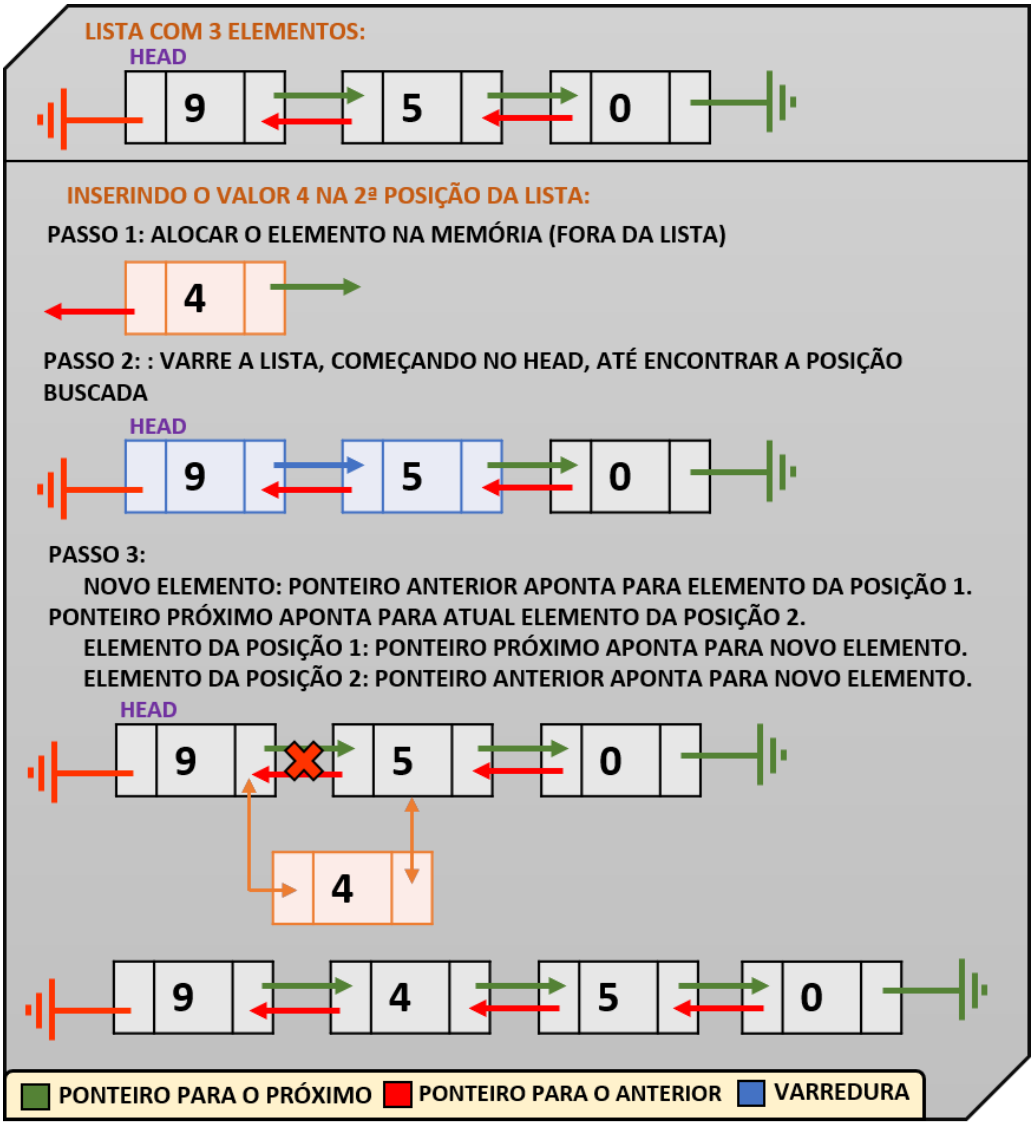
3.3 Inserindo um novo elemento no meio da lista encadeada dupla

A Figura 17 apresenta a inserção de um elemento no meio de uma lista encadeada dupla. Observe, na imagem, que temos uma lista já construída, com três elementos numéricos, lembrando que o ponteiro próximo do último elemento aponta para o nulo, assim como o ponteiro anterior ao *head*.

Agora, queremos inserir um novo elemento (valor 4) na posição 2 dessa lista, em que o valor 5 está localizado, atualmente. Portanto, o 4 deve vir entre o 9 e o 5. Os passos para a realização dessa tarefa são:

- **Passo 1**: alocar o novo elemento na memória. Perceba que alocá-lo na memória ainda não significa inseri-lo na lista. Nesse momento, ele ainda está fora da lista.
- **Passo 2**: executar uma varredura na lista existente, iniciando no *head*, até localizar a posição desejada (posição 2).
- **Passo 3**: utilizando uma variável de lista auxiliar, fazer a inserção do elemento no meio. O elemento da posição 1 irá apontar para o novo elemento, com seu ponteiro próximo. O antigo elemento da posição 2 apontará, com seu anterior, também para o novo elemento. O novo elemento irá apontar, com seu ponteiro anterior, para o 9 e, com seu ponteiro próximo, para o 5.

Figura 17 – Funcionamento da inserção de um elemento no meio de uma lista encadeada dupla



TEMA 4 – PILHA (*STACK*)

Em uma lista encadeada, conforme visto no Tema 2 e no Tema 3, a inserção dos dados pode ser feita em qualquer posição da estrutura de dados. Pilhas são estruturas de dados com um comportamento bastante específico. Em uma fila, a inserção e a remoção dos dados só podem ser realizadas em posições específicas da estrutura.

Uma pilha se comporta seguindo a regra chamada: *o primeiro que entra é o último que sai*. Em inglês, essa regra é chamada de *first in last out (Filo)*.

Para entender o que significa o Filo, vamos supor um exemplo prático. Imagine que você esteja realizando o seu treino na academia e resolva fazer supino. Para o exercício, você precisa colocar anilhas de carga na barra que irá levantar. Você percebe que as anilhas de pesos para o supino estão todas empilhadas em um local. A anilha que você precisa está mais embaixo da pilha de pesos. Portanto, para você conseguir pegar essa anilha, deverá remover todas as outras que estão em cima dela. Desse modo, **você só consegue remover o que está no topo da pilha**.

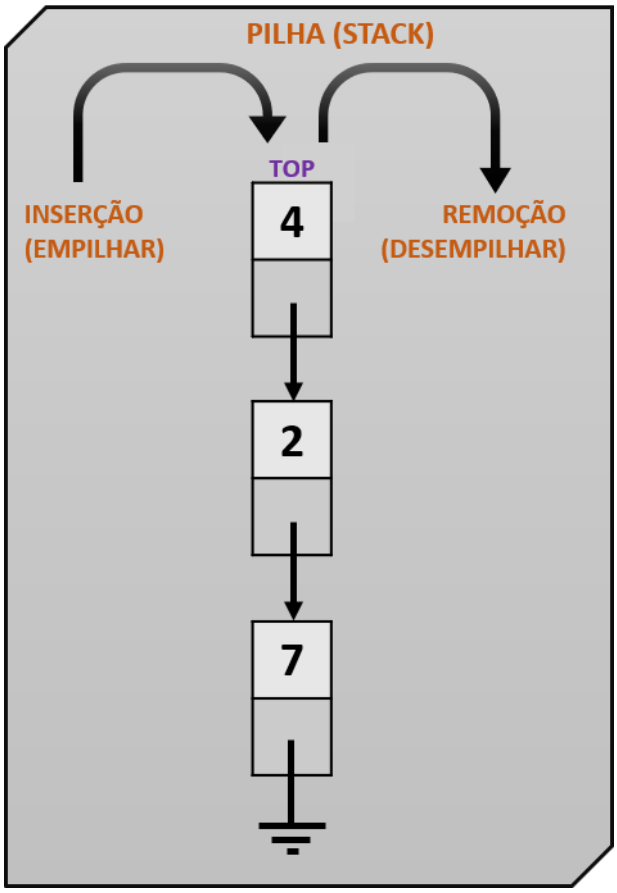
Após terminar seu treino de supino, você decide colocar as anilhas que utilizou de volta na pilha de anilhas para outra pessoa utilizá-las mais tarde. Ao empilhar os pesos, você só consegue **inserir no topo da pilha**, colocando um peso em cima do outro.

Sendo assim, as anilhas que você colocar primeiro ficarão mais para baixo da pilha e serão as últimas a que você terá acesso caso precise utilizá-las novamente, caracterizando a regra: a primeira anilha que entra é a última que sai (o primeiro que entra é o último que sai).

Em programação, podemos criar uma estrutura de dados que também funciona com esse princípio que acaba de ser explicado. Essa estrutura pode ser construída tanto com vetores quanto com listas encadeadas. Ambas funcionarão da mesma maneira detalhada anteriormente. As diferenças estarão nas características da estrutura de dados, características essas intrínsecas a elas. Vetores funcionarão com alocação sequencial da memória e tempo de acesso aos dados constante. Listas funcionarão com alocação dinâmica e tempo de acesso às informações $O(n)$.

A Figura 18 ilustra uma pilha. Toda a manipulação acontece no topo da pilha (comparável ao *head* de uma lista encadeada). No topo é onde você pode empilhar e desempilhar, ou seja, inserir e remover elementos.

Figura 18 – Pilha construída com uma estrutura de dados do tipo lista encadeada



Dentre algumas aplicações comuns de pilhas está a própria técnica de programação chamada *recursividade*, já bastante desenvolvida nas aulas anteriores. Cada instância de uma função recursiva aberta é tratada como um novo elemento inserido na pilha. Esse novo elemento precisa ser resolvido para então desempilharmos os demais elementos e assim termos acesso ao elemento (outra função) que está abaixo, na pilha.

Na Figura 19 temos o pseudocódigo principal. Nesse algoritmo, há a criação do registro que define como serão os elementos de nossa pilha. Temos também a declaração do nosso *top*, único elemento sempre conhecido globalmente pelo programa. No código, é apresentado um seletor do tipo **escolha**, que realiza a chamada de duas funções distintas: uma para inserir um novo elemento na pilha e outra para remover um elemento da pilha. Essas funções serão mais bem apresentadas nas próximas subseções.

Figura 19 – Pseudocódigo principal que declara a pilha e contém funções de inserção e remoção

```
1  algoritmo "PilhaMenu"
2  var
3     //Cria uma pilha usando lista
4     registro Pilha
5        dado: inteiro
6        prox: Pilha[->)
7     fimregistro
8     //Declara o Top como sendo do tipo da lista
9     Top: Pilha
10    op, numero: inteiro
11 inicio
12       //Lê um valor para inserir e a operação desejada
13       leia(numero)
14       leia(op)
15
16       escolha (op) //Escolhe o que deseja fazer
17          caso 0:
18             InserirNaPilha(numero)
19          caso 1:
20             RemoverDaPilha()
21       fimescolha
22 fimalgoritmo
```

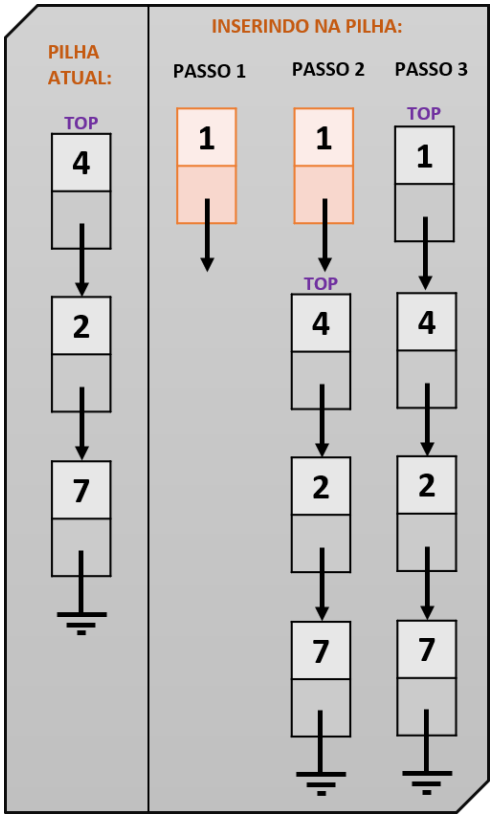
4.1 Inserindo um novo elemento na pilha (empilhando/*push*)

Inserir um novo elemento na pilha significa sempre inseri-lo no topo dela. Portanto, o processo será muito semelhante à inserção no início de uma lista encadeada, porém estaremos trocando o termo *head* por *topo*/*top*.

Na Figura 20 temos um exemplo de inserção de um elemento em uma pilha. Os passos para a realização dessa tarefa são:

- **Passo 1**: alocar o novo elemento na memória. Perceba que alocá-lo na memória ainda não significa inseri-lo na pilha. Nesse momento, ele ainda está fora da pilha.
- **Passo 2**: fazer o ponteiro do novo elemento apontar para o *top*.
- **Passo 3**: transformar o novo elemento no novo *top* da lista.

Figura 20 – Inserindo na pilha (*push*) com uma estrutura de dados do tipo lista encadeada



Na Figura 21, temos o pseudocódigo de inserção de um novo elemento na pilha. O valor a ser inserido é passado como parâmetro. Testamos se a pilha está vazia. Se ela estiver, o novo elemento é colocado no seu topo. Caso contrário, o novo elemento também é colocado no topo, mas agora deve apontar para o antigo topo.

Figura 21 – Pseudocódigo de inserção de um novo elemento na pilha

```
24  função push (numero: inteiro)
25  var
26     ElementoNovo: Pilha[->)
27  inicio
28     //Insere o valor no novo elemento que será inserido na pilha
29     NovoElemento->dado = numero
30     //Verifica que existe algo na pilha
31     se (Top == NULO) então
32        //Se a pilha está vazia, o novo elemento será a pilha
33        //e apontará para nulo
34        NovoElemento->prox = NULO
35     senão
36        //Se a pilha já tem algo, novo elemento aponta para o topo
37        NovoElemento->prox = Top
38     fimse
39     //Novo elemento vira o topo, pois a inserção é sempre no topo
40     Top = NovoElemento
41  fimfunção
```

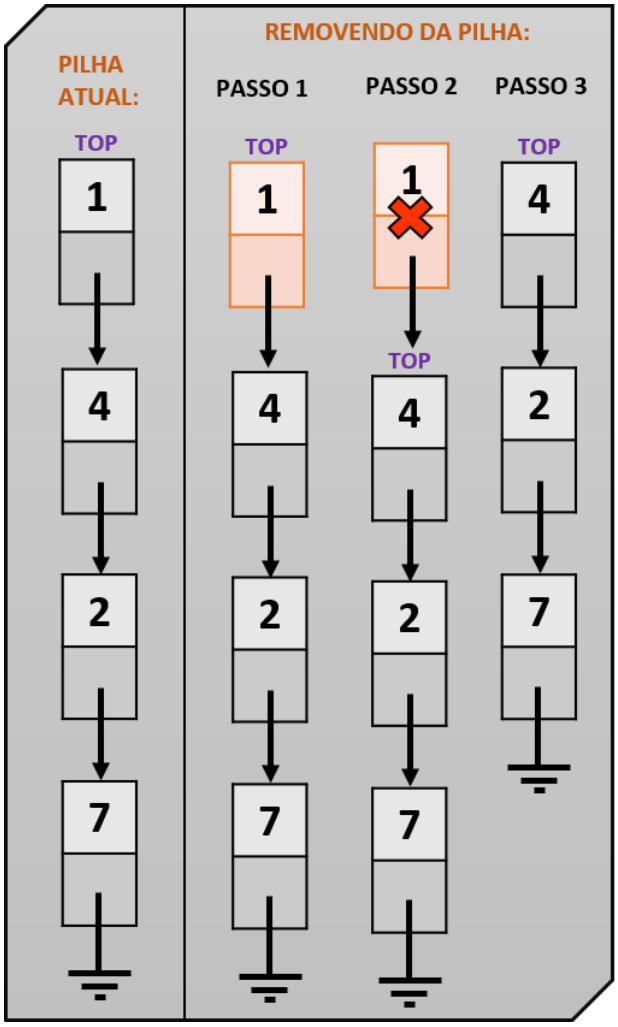
4.2 Removendo um elemento da pilha (desempilhando/*pop*)

Remover um elemento da pilha significa sempre removê-lo do topo dela. Sendo assim, estaremos sempre removendo o que estiver no *top*. E o próximo elemento depois do topo virará o *top*.

Na Figura 21, temos um exemplo de remoção de elemento da pilha. Os passos para a realização dessa tarefa são:

- **Passo 1**: localizar o topo da pilha.
- **Passo 2**: transformar o elemento subsequente ao topo no novo topo.
- **Passo 3**: liberar da memória o topo antigo, para que ele não ocupe espaço desnecessário na memória do programa.

Figura 21 – Removendo um elemento da pilha (*pop*) com uma estrutura de dados do tipo lista encadeada



Na Figura 22, temos o pseudocódigo de remoção do elemento da pilha. Perceba que o valor a ser removido nem precisa ser passado como parâmetro,

porque sempre iremos remover o que estiver no topo, não importando o seu valor. Testamos se a pilha está vazia. Se ela estiver, não temos o que remover. Caso contrário, realizamos o procedimento de avançar o topo para o próximo elemento e liberar da memória o topo atual.

Figura 22 – Pseudocódigo de remoção de elemento, de uma pilha

```
43    //Não passa nenhum valor como parâmetro,
44    //pois a remoção é sempre do valor do topo
45    função pop ()
46    var
47       ElementoParaRemover: Pilha[->)
48    inicio
49       //Verifica que existe algo na pilha
50       se (Top <> NULO) então
51          //Existe algo para remover então
52          //Salva temporariamente o topo atual
53          ElementoParaRemover = Top
54          //Incrementa o top (passa para o próximo nó)
55          Top = Top->prox
56          //Limpa da memória o topo antigo
57          libera(ElementoParaRemover)
58       fimse
59    fimfunção
```

TEMA 5 – FILA (*QUEUE*)

Uma fila se comporta seguindo a regra chamada: *o primeiro que entra é o primeiro que sai*. Em inglês, essa regra é chamada de *first in first out (Fifo)*.

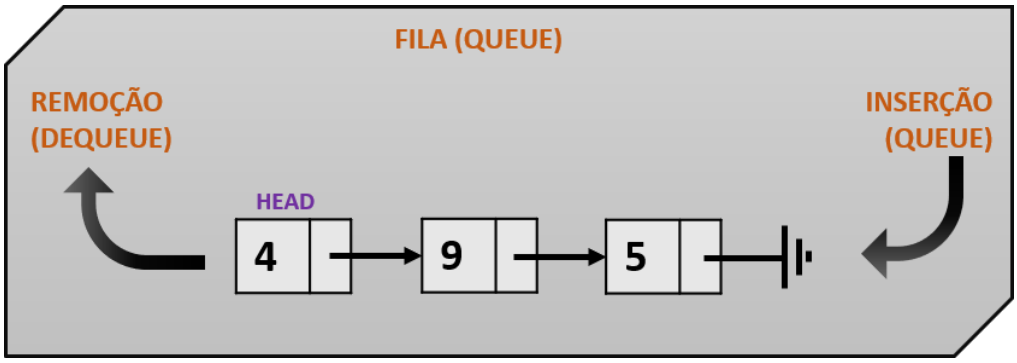
Para entender o que significa o Fifo, vamos supor um exemplo prático. Imagine que você esteja sentado na praça de sua cidade e um vendedor de pipoca se aproxime para iniciar suas vendas. O vendedor então abre sua venda e você corre para ser o primeiro a comprar um saco de pipoca. Uma fila de pessoas começa a se formar atrás de você, para saborear a deliciosa pipoca.

A primeira pessoa que entrou na fila (você) será a primeira pessoa a ser atendida e a receber sua pipoca. Todas as outras pessoas que forem chegando **entrarão na fila atrás da pessoa que chegou antes**. Desse modo, **você, que foi o primeiro a chegar à fila, será o primeiro a ser atendido**.

Sendo assim, a primeira pessoa que chega à fila é a primeira que sai dela. E cada nova pessoa que chega entra no final da fila (o primeiro que entra é o primeiro que sai).

Em programação, podemos criar uma estrutura de dados que também funciona com esse princípio que acaba de ser explicado. Essa estrutura pode ser construída tanto com vetores quanto com listas encadeadas. A Figura 23 ilustra uma fila com lista encadeada. Toda a manipulação acontecerá no início ou no final dessa fila. No início da fila (*head*), teremos a remoção e, no final dela, teremos a inserção, e sempre será assim.

Figura 23 – Fila construída com uma estrutura de dados do tipo lista encadeada



Dentre algumas aplicações comuns de filas na área de tecnologia, podemos citar o processo de fila de impressão, em uma rede. Cada arquivo que é colocado para imprimir, em uma mesma impressora, é inserido no final da fila. E o que está na frente da fila é removido (impresso) primeiro.

Na Figura 24, temos o pseudocódigo principal de uma fila. Nesse algoritmo, temos a criação do registro que define como serão os elementos de nossa fila. Temos também a declaração do *head*, único elemento sempre conhecido globalmente pelo programa. No código, é apresentado um seletor do tipo **escolha**, que realiza a chamada de duas funções distintas: uma para inserir um novo elemento na fila e outra para remover o elemento da fila. Essas funções serão mais bem explicadas nas próximas subseções.

Figura 24 – Pseudocódigo principal que declara a fila e contém funções de inserção e remoção de elementos

```
1   algoritmo "FilaMenu"
2   var
3      //Cria uma fila usando lista
4      registro Fila
5         dado: inteiro
6         prox: Fila[->)
7      fimregistro
8      //Declara o Top como sendo do tipo da lista
9      Head: Fila
10     op, numero: inteiro
11  inicio
12        //Lê um valor para inserir e a operação desejada
13        leia(numero)
14        leia(op)
15
16        escolha (op) //Escolhe o que deseja fazer
17           caso 0:
18              InserirNaFila(numero)
19           caso 1:
20              RemoverDaFila()
21        fimescolha
22  fimalgoritmo
```

5.1 Inserindo um elemento na fila (*queuing*)

Inserir um novo elemento em uma fila significa sempre inseri-lo no final dela. Portanto, o processo será muito semelhante à inserção no final de uma lista encadeada.

Na Figura 25, temos um exemplo de inserção de elemento em uma fila. Os passos para a realização dessa tarefa são:

- **Passo 1**: alocar o novo elemento na memória. Perceba que alocá-lo na memória ainda não significa inseri-lo na pilha. Nesse momento, ele ainda está fora da fila.
- **Passo 2**: varrer a fila até encontrar o ponteiro nulo (último elemento).
- **Passo 3**: fazer o ponteiro nulo do último elemento apontar para o novo elemento.

Figura 25 – Inserindo elemento na fila (*queueing*) com uma estrutura de dados do tipo lista encadeada

Na Figura 26, temos o pseudocódigo de inserção do elemento na fila. O valor a ser inserido é passado como parâmetro. Testamos se a pilha está vazia. Se não estiver, realizamos o processo de varredura e inserção do elemento no final.

Figura 26 – Pseudocódigo de inserção de elemento na fila

```
24  função queue (numero: inteiro)
25  var
26     ElementoNovo: Fila[->)
27     ElementoVarredura: Fila[->)
28  inicio
29     //Insere o valor no novo elemento que será inserido na fila
30     NovoElemento->dado = numero
31     //Verifica que existe algo na fila
32     se (Head == NULO) então
33        //Se a fila está vazia, o novo elemento será a fila
34        Head = NovoElemento
35     senão
36        //Se a pilha já tem algo, novo elemento entra no final
37        ElementoVarredura = Head
38        //Varre um elemento por vez procurando o ponteiro nulo
39        enquanto (ElementoVarredura->prox <> NULO)
40           ElementoVarredura = ElementoVarredura->prox
41        fimenquanto
42        //Após encontrar o ponteiro nulo,
43        //faz o último elemento da fila apontar para o novo nó
44        ElementoVarredura->prox = NovoElemento
45        //Novo nó agora terá o ponteiro nulo (final da lista)
46        NovoElemento->prox = NULO
47     fimse
48  fimfunção
```

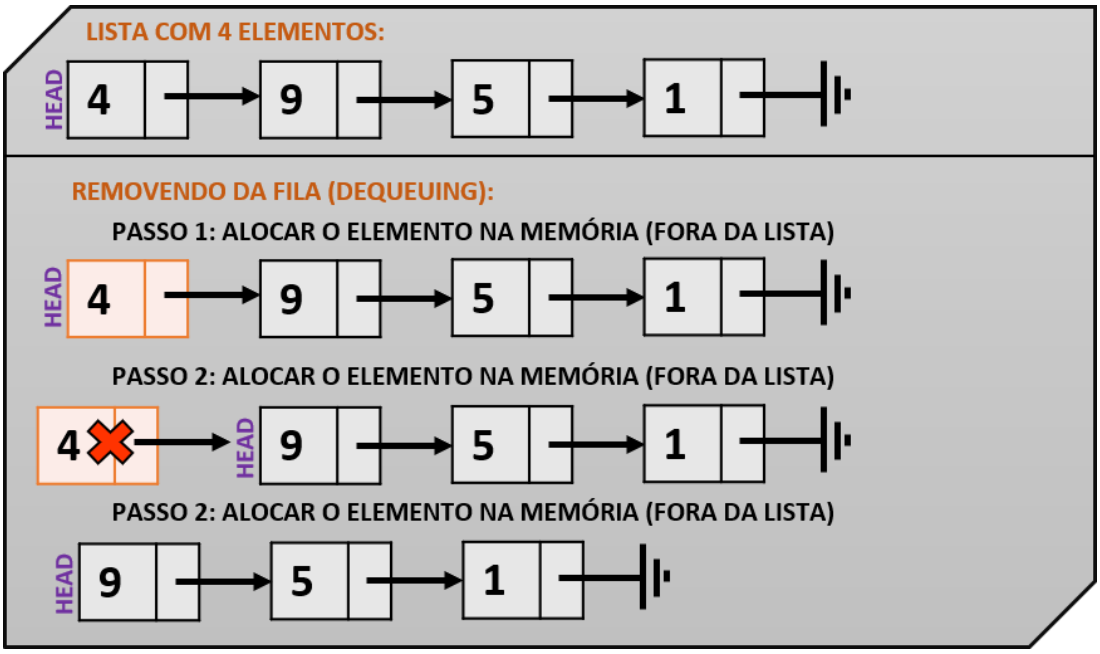
5.2 Removendo da fila (*dequeuing*)

Remover um elemento de uma fila significa sempre removê-lo do início dela. Sendo assim, estaremos sempre removendo o que estiver no *head*. E o próximo elemento depois do *head* assumirá seu lugar.

Na Figura 27, temos um exemplo de remoção da fila. Os passos para a realização dessa tarefa são:

- **Passo 1**: localizar o *head* da fila.
- **Passo 2**: transformar o elemento subsequente ao *head* no novo *head*.
- **Passo 3**: liberar da memória o *head* antigo, para que ele não ocupe espaço desnecessário na memória do programa.

Figura 27 – Removendo elemento da fila (*dequeueing*) com uma estrutura de dados do tipo lista encadeada



Na Figura 28, temos o pseudocódigo de remoção do elemento da fila. Perceba que o valor a ser removido nem precisa ser passado como parâmetro, porque sempre iremos remover o que estiver no *head*, não importando o seu valor. Testamos se a fila está vazia. Se ela estiver, não temos o que remover. Caso contrário, realizamos o procedimento de avançar o *head* para o próximo elemento e liberar da memória o *head* atual.

Figura 28 – Pseudocódigo de remoção de elemento da fila

```
50  //Não passa nenhum valor como parâmetro,
51  //pois a remoção é sempre do valor do head
52  função dequeue ()
53  var
54      ElementoParaRemover: Fila[->)
55  inicio
56      //Verifica que existe algo na pilha
57      se (Head <> NULO) então
58          //Existe algo para remover então
59          //Salva temporariamente o head atual
60          ElementoParaRemover = Head
61          //Incrementa o head (passa para o próximo nó)
62          Head = Head->prox
63          //Limpa da memória o head antigo
64          libera(ElementoParaRemover)
65      fimse
66  fimfunção
```

FINALIZANDO

Nesta aula, aprendemos sobre estrutura de dados do tipo lista encadeada. Essas estruturas apresentam como vantagem a alocação dinâmica na memória e não trabalharem com alocação sequencial.

Foram vistas, na aula, listas encadeadas simples e duplas. O que diferencia ambas é a quantidade de ponteiros em cada elemento da lista, um ponteiro e dois ponteiros, respectivamente.

É também possível classificar as listas em não circulares ou circulares. As não circulares contêm o último elemento da lista com um ponteiro nulo. Enquanto que a circular fecha o círculo e seu último elemento aponta de volta para o início da lista e é chamado de *head*. Se a lista se forma dupla e circular, o primeiro elemento ainda aponta diretamente para o último da lista.

Vimos também duas estruturas bastante peculiares, que podem ser construídas tanto com vetores quanto com listas encadeadas, mas que nesta aula foram trabalhadas somente no formato de listas. Estamos falando das pilhas, que operam seguindo o *fisrt in first out* (Filo); e das filas, com o *first in first out* (Fifo).

REFERÊNCIAS

ASCENCIO, A. F. G.; ARAÚJO, G. S. de. **Estrutura de dados**: algoritmos, análise da complexidade e implementações em Java e C/C++. São Paulo: Pearson, 2011.

ASCENCIO, A. F. G.; CAMPOS, E. A. V. de. **Fundamentos da programação de computadores**: algoritmos, Pascal, C/C++ (padrão Ansi) Java. 3. ed. São Paulo: Pearson, 2012.

CORMEN, T. H.; LEISERSON, C.; RIVEST, R. **Algoritmos**: teoria e prática. 3. ed. São Paulo: Elsevier, 2012.

PUGA, S.; RISSETI, G. **Lógica de programação e estrutura de dados**. 3. ed. São Paulo: Pearson, 2016.

LAUREANO, M. **Estrutura de dados com algoritmos e C**. Rio de Janeiro: Brasport, 2008.

MIZRAHI, V. V. **Treinamento em linguagem C**. 2. ed. São Paulo: Pearson, 2008.