



ESTRUTURA DE DADOS

AULA 3



Prof. Vinicius Pozzobon Borin

Está na hora de aprender uma nova estrutura de dados! Nesta etapa, você irá se deparar com as características de implementação de uma estrutura de dados denominada *lista encadeada*. As listas encadeadas são alocadas dinamicamente na memória e diferem das implementações de arrays encontrados em linguagens de programação, trazendo vantagens e desvantagens. Mas, atenção, uma lista em Python não é uma lista encadeada! Tenha isso em mente e você compreenderá melhor este motivo.

TEMA 1 – ARRAYS E LISTAS ENCADEADAS

Arrays, ou *vetores*, são estruturas de dados que trabalham com alocação de dados sequenciais na memória. Podemos classificar os arrays em estáticos e dinâmicos. Para melhor compreensão, veremos diferentes cenários.

1.1 O problema dos assentos reservados no cinema

Você e mais quatro amigos (Tinky Winky, Dipsy, Laa-Laa, Poo) estão indo ao cinema, o qual só funciona com assentos reservados previamente. Vocês cinco gostariam de sentar juntos para aproveitar melhor o filme e, portanto, reservaram os assentos sequenciais do E5 até E9 (em verde). Então, todos se sentam juntos. Veja, a seguir, uma matriz que representa os assentos de cinema.

Figura 1 – Assentos no cinema. Em laranja, os ocupados; em verde, os livres.

	0	1	2	3	4	5	6	7	8	9
A										
B										
C										
D										
E										
F										
G										
H										



A memória de um programa é dividida de maneira semelhante aos assentos de um cinema, pois cada bloquinho de memória irá representar um dado a ser armazenado. Quando precisamos armazenar um conjunto de dados, podemos organizá-lo a partir de um *array*, o que implica que todos os dados serão armazenados sequencialmente na memória do programa. Em nosso exemplo do cinema, você e seus amigos estão organizados como se fosse um *array* sequencial, pois estão todos juntos.

Porém, de última hora, mais um amigo seu, o Solzinho, decide aparecer. Agora, para sentarem juntos seria necessário que todos se movam para uma sequência de assentos que comporte todo mundo. A fila de trás (F4 até F9) seria a única possibilidade. Infelizmente, o cinema só trabalha com lugares reservados e vocês não podem simplesmente pular para a fileira de trás. Jonas, então, para não ficar sozinho, vai embora entristecido e sem ver o filme.

Essa situação em que temos mais um dado para ser colocado no *array*, mas sua inserção é negada, corresponde a um *array* do tipo estático, que nada mais é do que um conjunto de espaços sequenciais na memória, previamente reservado, mas que em hipótese alguma pode ter o seu tamanho alterado.

Voltando ao exemplo dos assentos, seria muito simples que você e seus amigos sentassem na fileira de trás, onde caberia todo mundo, mas isso foi negado a vocês, pois o *array* estático funciona com reserva prévia de espaços de memória. O inverso também é válido. Veja, você e seus amigos compraram cinco ingressos para ocupar assentos reservados. Caso alguém falte, o assento continuará pago e reservado, sem necessidade.

1.2 O problema dos assentos sem reserva no cinema

Vejamos agora outro cenário. Você e seus quatro amigos estão indo em outro cinema, mas neste não há exigência de reserva de lugares, ou seja, quem chegar por primeiro ocupa os lugares. Suponha que a configuração de espaços vazios é igual ao da Figura 1 e vocês cinco sentam-se nas cadeiras E5-E9. Novamente, de última hora, seu amigo Solzinho, atrasado, chega para assistir ao filme com vocês. Porém, agora não existe marcação de assentos. Neste caso, vocês simplesmente pulam para a fileira de trás, que comporta seis pessoas:



Figura 2 – Assentos no cinema sem reserva. Em laranja, os ocupados; em verde, os livres

	0	1	2	3	4	5	6	7	8	9
A										
B										
C										
D										
E										
F										
G										
H										

A analogia do cinema com assentos sem reserva, neste caso, corresponde a um array dinâmico, o qual também armazena dados sequencialmente. Agora, porém, à medida que surgem novos dados, é possível realocar o conjunto em outro espaço de memória em que caibam todos os dados.

Complementando o cenário, imagine que mais um amigo seu, além do Solzinho, chegou para assistir ao filme. Considerando que todos precisam sentar-se juntos, ele não poderia assistir ao filme, porque no exemplo da Figura 2 não existe nenhuma sequência de sete assentos livres.

1.3 O problema dos amigos distantes no cinema

Vamos ao último cenário no cinema. Você e seus amigos quatro amigos agora vão ao cinema sem poltronas marcadas, mas os lugares próximos estão bastante escassos. Não existe nenhum lugar na sala de cinema com mais de dois lugares livres juntos. O filme é legal demais para ser perdido e vocês optam por assisti-lo mesmo separados fisicamente.



Figura 3 – Assentos no cinema separados. Em laranja, os ocupados; em verde, os livres

	0	1	2	3	4	5	6	7	8	9
A										
B										
C										
D										
E										
F										
G										
H										

Solzinho, sempre atrasado, chega em cima do horário. Neste cenário, ele poderia facilmente sentar-se em qualquer outro local livre e assistir ao filme com seus amigos, na mesma sala de cinema. Esse tipo de configuração, de arranjo de dados sem necessidade de estarem sequencialmente alocados é chamado de *listas encadeadas*.

1.4 Arrays e listas encadeadas: comparativo

Arrays são conjuntos de dados alocados sequencialmente na memória do programa. Quando declaramos uma estrutura de vetor, na memória do programa ele é inicializado (alocado) a partir da primeira posição (endereço da primeira célula). Cada outra célula, a partir da segunda, possui um endereço de referência relativo à primeira célula endereçada. Este endereço é calculado considerando a posição da primeira célula, acrescido do tamanho em *bytes* de cada uma – tamanho este que depende do tipo de dado armazenado no vetor. Chamamos isto de *alocação sequencial*.

Observe, na Figura 4, um vetor de valores inteiros de dimensão, ou comprimento, igual a 5, ou seja, contendo 5 células. Sua inicialização na memória é dada pela primeira posição desse vetor, neste caso, no endereço *0x0001h*. Assumimos que o vetor homogêneo é do tipo inteiro de tamanho 4 *bytes* por célula. A segunda posição (célula) está alocada, portanto, na posição da memória $0x0001h + 4 \text{ bytes}$. A terceira posição (célula) está alocada, na



posição da memória $0x0001h + 2*4 \text{ Bytes}$. E assim por diante, até a última posição do vetor, o qual contém um tamanho fixo e conhecido previamente. A seguir, a Figura 4 representa graficamente a explicação.

Figura 4 – Endereçamento de uma alocação sequencial

ÍNDICE	0	1	2	3	4
VETOR	4 Bytes	4 Bytes	4 Bytes	4 Bytes	4 Bytes
ENDEREÇO	0x0001h +0*4B	0x0001h +1*4B	0x0001h +2*4B	0x0001h +3*4B	0x0001h +4*4B

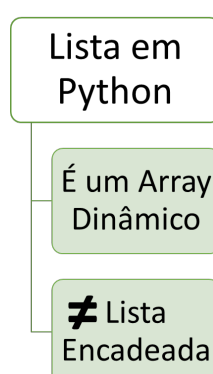
Podemos generalizar a forma de cálculo de cada posição na memória de um vetor pela equação 1:

$$Endereço_n = Endereço_0 + (Índice * Tamanho_{Bytes}) \quad (1)$$

em que $Endereço_0$ é o endereço conhecido da primeira posição do vetor, índice é a posição de cada célula e $Tamanho_{Bytes}$ é o tamanho de cada célula em *bytes*, neste exemplo, 4 *bytes*.

Diferentes linguagens de programação trabalham com maneira distintas de criar arrays estáticos e dinâmicos. A linguagem C/C++, por exemplo, só consegue criar arrays dinâmicos pela manipulação de ponteiros. Quando você declarar um vetor como *Tipo NomeDoVetor[quantidade_de_itens]* ele será estático. E atenção, não confunda! Na linguagem Python, a estrutura de lista (aquela que você cria assim: *Nome = []*) é, na verdade, um array dinâmico e não tem nenhum aspecto construtivo de uma lista encadeada.

Figura 5 – Lista em Python





O **array estático** recebe este nome quando um bloco na memória é destinado a ele previamente, isto é, mesmo que o bloco não seja usado inteiramente, ainda assim estará reservado e ocupando espaço no programa.

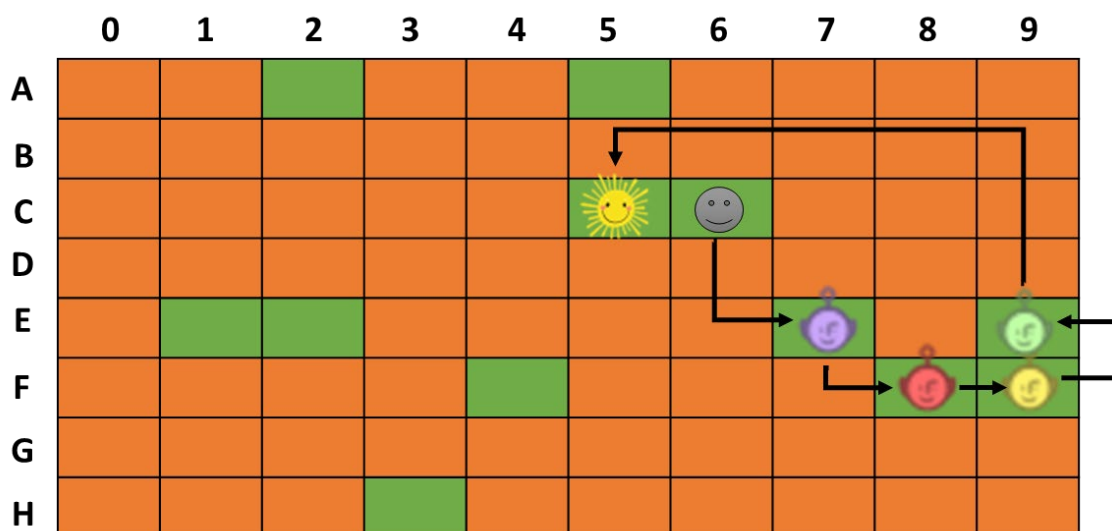
O **array dinâmico** é mais interessante, pois aloca somente o espaço sendo usado. Apesar disso, caso não exista na memória do programa nenhum bloco sequencial do tamanho desejado, haverá um problema, pois a alocação não acontecerá e o programa resultará em um erro de memória ou alocará o dado em local impróprio, podendo perder dados ou piorar o desempenho do programa.

A **lista encadeada** trabalha com alocação não sequencial. Isso significa que os dados estão esparsos ao longo da memória do programa. Essa é uma característica muito interessante, pois, além de não usar memória em excesso, a lista encadeada é capaz utilizar quaisquer pequenos espaços livres na memória.

Mas se os dados estão esparsos na memória, como a estrutura de lista encadeada localiza seus elementos? Cada elemento da lista encadeada armazena na memória não só seus dados, mas também o endereço de onde está localizado o próximo elemento na memória.

Na Figura 6, temos a representação do cinema usando uma lista encadeada. Veja que foram colocadas flechas (ponteiros) de um elemento para outro. Comparando com o exemplo do cinema, é como se cada pessoa do grupo de amigos soubesse somente onde um de seus amigos está sentado, e o outro soubesse do próximo, e ninguém soubesse onde está todo mundo.

Figura 6 – Identificação do próximo elemento da lista encadeada





Então, se a lista encadeada é tão boa assim para alocação de memória, por que as linguagens ainda insistem em trabalhar com arrays? Veja bem: para toda vantagem, existe uma desvantagem. Em uma lista encadeada, cada elemento conhece somente o próximo elemento, pois armazena somente o endereço do próximo elemento – e isso pode gerar um certo problema.

Imagine que você recebeu um livro de cem páginas, e imagine também que cada página é o equivalente a um dado de um array. Você precisa chegar à página 50 para ler as informações nela contidas. Como cada página está numerada, basta abrir na página 50 e instantaneamente ler o que tem nela. O tempo para abrir na página 50, na 10 ou na 99, em tese, é o mesmo. Basta abrir o livro, pois as páginas estão numeradas. Um array funciona desta mesma maneira. Como alocamos um bloco sequencial de dados, o endereço de cada dado é conhecido pelo seu índice (veja a Figura 4 e a Equação 1).

Agora, imagine que nosso livro de 100 páginas armazena cada página em uma lista encadeada, na qual cada dado só sabe onde o próximo está. Ou seja, se queremos abrir na página 50, não sabemos onde ela está. Isso é o equivalente a tentarmos abrir um livro não numerado em uma página específica. E como chegamos em uma página que não tem número? Precisaremos contar uma a uma, a partir da primeira, até chegar à página desejada. Assim, uma lista encadeada apresenta um tempo de acesso em média inferior ao de um array.

1.5 Arrays e listas encadeadas: complexidade

Falamos sobre características, vantagens e desvantagens de arrays e listas encadeadas. Igual a complexidade de cada um? Em ambas as estruturas de dados podemos fazer distintas manipulações. Podemos ler um dado, fazer inserções em diferentes posições do conjunto de dados ou ainda deletar algum dado. A tabela a seguir apresenta o Big-O de algumas dessas manipulações.

Tabela 1 – Big-O de algumas manipulações

Função	Array	Lista Encadeada
Leitura	$O(1)$	$O(n)$
Inserção no início	$O(n)$	$O(1)$
Inserção no fim	$O(n)$	$O(n)$
Inserção no meio	$O(n)$	$O(n)$

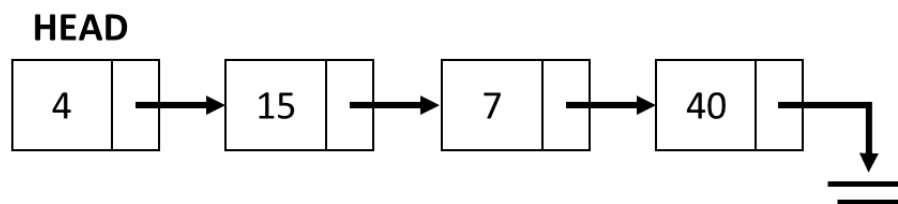
Um array, para ler um dado, independe do conjunto de dados, sendo $O(1)$; a lista precisará varrer elemento por elemento até chegar no desejado, $O(n)$.

TEMA 2 – CONSTRUINDO LISTAS ENCADEADAS

As listas encadeadas, também conhecidas como *listas ligadas*, ou ainda o termo em inglês *linked lists*, são compostas de um conjunto de dados em que cada dado é armazenado em um elemento que chamamos de *nó*, ou *nodo*. Cada elemento da lista é composto, portanto, minimamente de duas informações: o(s) dado(s) a serem armazenados e um endereço (ponteiro) na memória do próximo elemento da lista encadeada.

Chamamos o primeiro elemento da lista encadeada de *cabeça*, ou do seu termo em inglês, que é mais utilizado, *head*. A maneira de iniciar uma lista e localizar todos os seus valores é por meio do seu head. Podemos representar uma lista encadeada pelo desenho a seguir.

Figura 7 – Exemplo de lista encadeada com dados numéricos



Note que o *head* contém o número 4 e aponta, ou seja, conhece o endereço do próximo elemento, o 15. Isso ocorre assim por diante, até chegarmos ao final desta lista, representada por um ponteiro nulo. Em suma as características das listas encadeadas são:

- sucessivos elementos conectados por ponteiros;
- o último elemento aponta para um endereço nulo, que é o final da lista;
- funciona dinamicamente, aumentando e diminuindo de tamanho conforme necessita;
- pode utilizar toda a memória destinada ao programa, uma vez que cada dado é capaz de ficar isolado na memória;
- não desperdiça memória, alocando somente o que precisa; e
- apresenta tempos de leitura Big-O de dados inferior aos arrays.
-



2.1 Listas encadeadas simples

Existem algumas classificações das listas encadeadas, mas iremos focar nossos estudos, neste momento, na lista encadeada do tipo simples, em que cada elemento aponta para o próximo e o último, aponta para nulo, conforme a Figura 7. Implementaremos essa lista ligada em linguagem Python. Para isso, criaremos uma classe que representará cada elemento da lista encadeada. Veja:

```
#Cria cada elemento da lista
class ElementoDaListaSimples:
# construtor de inicialização da classe
    def __init__(self, dado):
        self.dado = dado
        self.proximo = None

#__repr__ é um método especial do Python
#use-o para criar a maneira como objeto
#é mostrado fora da função print
    def __repr__(self):
        return self.dado
```

Precisamos também criar uma segunda classe, que será, de fato, a lista encadeada. No constructo de inicialização (`__init__`) definimos que o *head* inicia-se vazio (*None*). Caso a lista não esteja vazia, temos:

```
#Cria a lista encadeada simples
class ListaEncadeadaSimples:
    def __init__(self, nodos=None):
        self.head = None
        if nodos is not None:
            nodo = ElementoDaListaSimples(dado=nodos.pop(0))
            self.head = nodo
            for elem in nodos:
                nodo.proximo = ElementoDaListaSimples(dado=elem)
                nodo = nodo.proximo

    def __repr__(self):
        nodo = self.head
        nodos = []
        while nodo is not None:
            nodos.append(nodo.dado)
            nodo = nodo.proximo
        nodos.append("None")
        return " -> ".join(nodos)
```

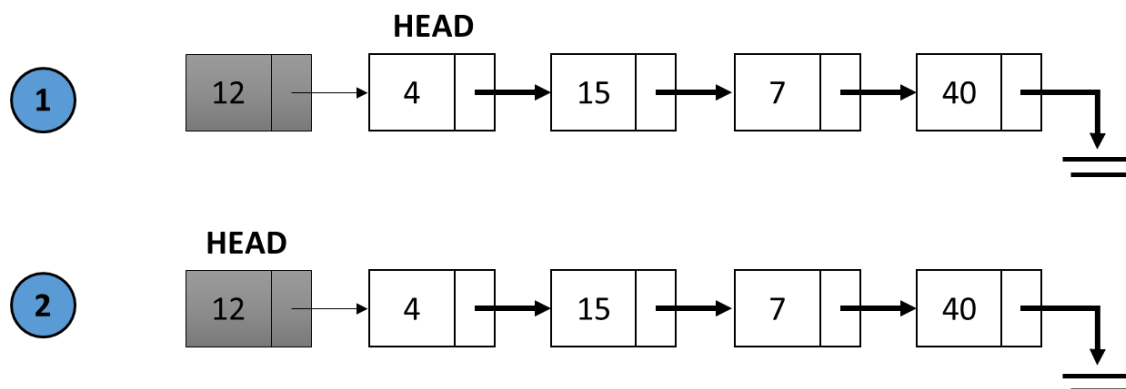


2.2 Inserindo no início da lista encadeada

Suponha agora que temos uma lista encadeada já construída com alguns dados, mas queremos inserir mais um. A inserção pode ser feita no início da lista, no meio, ou no final; inserir no início da lista significa inserir antes do *head* atual. Para tal, precisamos respeitar duas etapas na inserção.

Antes da inserção, é válido observar que primeiro criamos o novo elemento na memória, instanciando-o pela classe *ElementoDaListaSimples*. Em seguida, a primeira etapa de inserção corresponde a fazer este novo elemento apontar para o atual *head*. Na segunda etapa, transformamos o novo elemento, no novo *head*, conforme a Figura 8.

Figura 8 – Etapas de inserção no início da lista encadeada



O código de inserção no início da lista é apresentado a seguir.

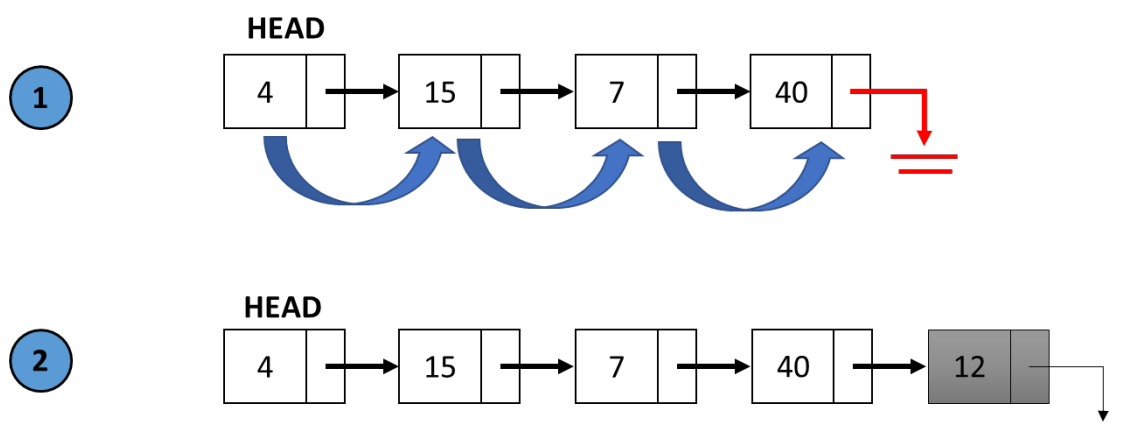
```
def inserirNoInicio(self, nodo):  
    nodo.proximo = self.head  
    self.head = nodo
```

2.3 Inserindo no final da lista encadeada

Podemos também inserir um dado no final da lista encadeada, isto é, após o elemento que contém o ponteiro nulo. Antes da inserção, é válido observar que primeiro criamos o novo elemento na memória, instanciando-o pela classe *ElementoDaListaSimples*. Em seguida, a primeira etapa de inserção corresponde a uma varredura na lista até localizarmos um ponteiro nulo na lista. Lembre-se de que na lista só conhecemos o *head* e que o *head* conhece o próximo elemento, e assim por diante, até chegarmos ao final. Uma vez localizado o ponteiro nulo, fazemos ele apontar para o novo elemento de nossa lista encadeada, conforme a Figura 9.



Figura 9 – Etapas de inserção no final da lista encadeada



O código a seguir realiza essa inserção ao final da lista ligada. Note o laço de repetição que só se encerra quando localizamos um endereço nulo (*None* no Python). É válido observar também que logo no início do código temos um teste condicional para verificar se nossa *linked list* é, ou não, completamente vazia. Caso ela esteja vazia, significa que o *head* está vazio e, portanto, ao invés de varrermos podemos simplesmente inserir o dado no *head*.

```
def inserirNoFinal(self, nodo):  
    if self.head is None:  
        self.head = nodo  
        return  
  
    nodo_atual = self.head  
    while nodo_atual.proximo != None:  
        nodo_atual = nodo_atual.proximo  
  
    nodo_atual.proximo = nodo  
    return
```

2.4 Classificação de listas encadeadas

Existem diferentes tipos de listas encadeadas. Vamos conhecer um pouco mais sobre cada uma delas na teoria, embora estejamos focando nossos estudos somente na lista simples.

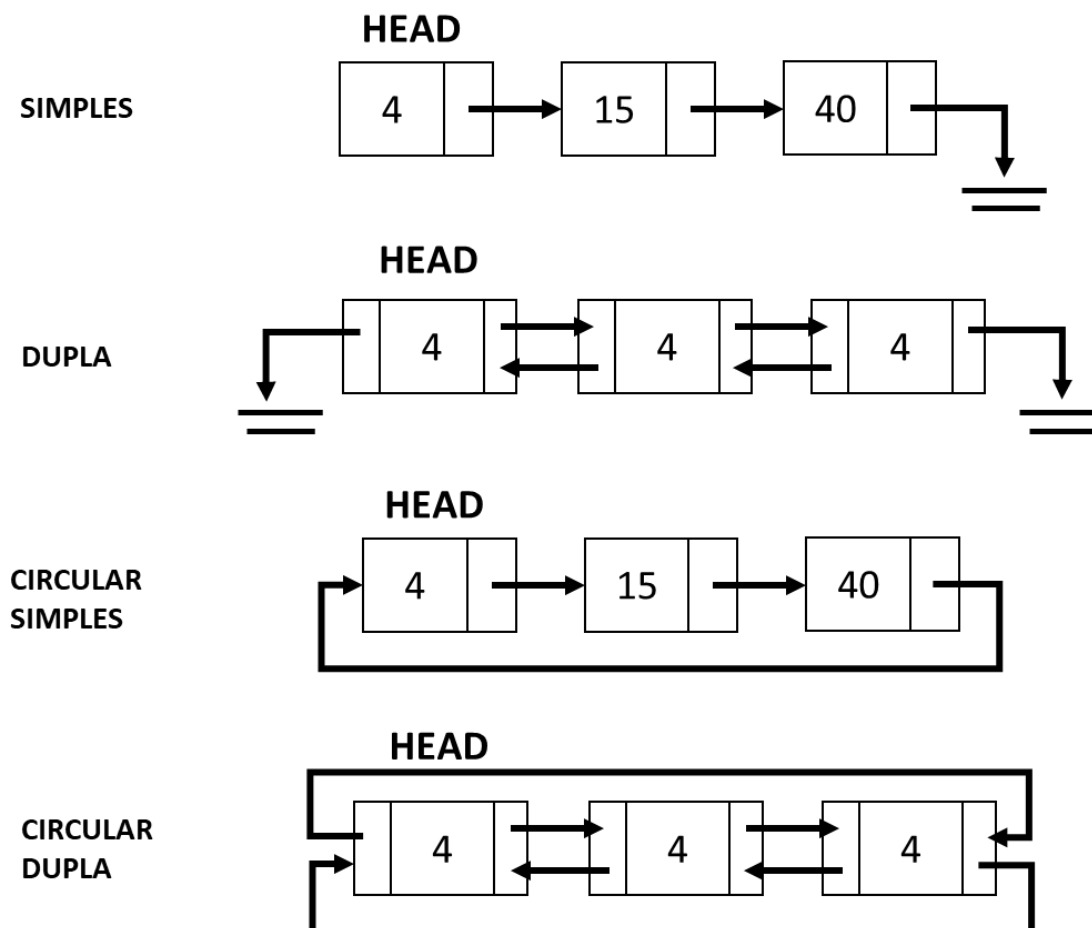
Tabela 2 – Tipos de lista encadeada

Tipo	Endereços armazenados	Último elemento
Simple	Próximo	Aponta para nulo
Dupla	Próximo e anterior	Aponta para nulo
Simple e circular	Próximo	Aponta de volta para o <i>head</i>
Dupla e circular	Próximo e anterior	Aponta de volta para o <i>head</i>



Note que a diferença entre uma lista simples e uma dupla é a quantidade de endereço conhecidos; na dupla é possível ir e voltar na lista. Já o que difere ambas de listas circulares é que em uma circular o último elemento da lista, que aponta de volta para o início da mesma, não ficando nulo.

Figura 10 – Tipos de listas encadeadas representadas graficamente



TEMA 3 – PILHAS (STACKS)

Uma estrutura de dados do tipo pilha, ou *stack*, não é exatamente uma nova estrutura de dados, mas, sim, uma maneira de operar dados dentro de uma estrutura em programação. Vamos compreender o conceito de pilha com um exemplo de pilhas de objetos.

Imagine que você está indo à academia e está organizando anilhas de pesos de 10 Kg e 20 Kg; você precisa empilhar todos esses pesos. Para isso, você pega uma anilha de 20 Kg e coloca no chão. Em seguida, pega outra anilha de 20 Kg e coloca em cima da anterior, e assim por diante. Então, você vai



pegando anilha por anilha e empilhando, formando uma pilha de anilhadas bastante pesadas. Veja a imagem a seguir.

Figura 10 – Pilha com anilhas



Crédito: Ilja Generalov / Shutterstock.

Mais tarde, outra pessoa veio à academia e decidiu utilizar suas anilhas que estavam empilhadas. Porém, a pessoa estava precisando das anilhas de 20 Kg, justamente as que estavam posicionadas mais abaixo da pilha. Para obter seu acesso, a pessoa precisou então desempilhar anilha por anilha até chegar à desejada. Aliás, imagine tentar puxar as anilhas mais abaixo sem remover as de cima? Haja força para isso!

Em resumo, uma estrutura de dados funcionando como pilha, funciona como uma pilha de anilhas. Uma estrutura de dados é uma pilha quando só conseguimos manipular o que está em seu topo. Ou seja, quando empilhamos as anilhas, estamos empilhando em cima do topo, e quando removemos as anilhas, desempilhamos somente o que está no topo. Nunca, em hipótese alguma, podemos inserir ou remover anilhas que estão no meio.

Uma estrutura de pilha em programação opera com o princípio chamado *o primeiro que entra é o último que sai*. A expressão correspondente em inglês



é *first in last out (filo)*¹. As estruturas de pilhas apresentam inúmeras aplicações em programação. Primeiro, qualquer algoritmo recursivo trabalha com uma estrutura de pilha, porque a recursividade opera empilhando funções, e somente a função mais ao topo da pilha é que está em execução. Sistemas operacionais trabalham muito com estruturas de pilhas internamente para funcionarem corretamente também.

Mas quando uso uma pilha em meu código? Vejamos alguns exemplos. Primeiro, podemos citar um exemplo clássico de implementação com pilhas que é o cálculo de expressões matemáticas. Imagine que seu algoritmo precisa validar e resolver uma equação contendo somas, subtrações, multiplicações e divisões – tudo isso ainda dentro de parênteses. Podemos criar um algoritmo que analisa a equação e a ordem dos operadores e vai empilhando todas as operações na ordem em que elas precisam ser executadas, bastando na sequência desempilhar e operar.

Outros exemplos seriam algoritmos de cálculos de conversão de base (decimal para binário, por exemplo), ou ainda podemos construir um jogo que contenha recursos de pilhas, como o jogo de cartas *FreeCell*, em que precisamos criar diferentes pilhas de cartas para manipular. Ferrari (2014) ensina a implementar o *FreeCell* com pilhas (e diversos outros jogos).

3.1 Construindo e manipulando uma pilha

A construção de pilhas pode ser realizada utilizando algumas estruturas de dados já conhecidas. Por exemplo, podemos construir pilhas empregando arrays dinâmicos (lista em Python) ou com listas encadeadas. Isso é possível justamente porque uma pilha nada mais é do que a maneira como inserimos e removemos dados na estrutura, já uma array ou uma lista encadeada define como os dados são organizados na memória; ambos se complementam.

Na Figura 11, temos graficamente duas representações de pilhas: uma envolvendo listas encadeadas e outra, os arrays. Note que em ambas o primeiro elemento é sempre chamado de *topo*, ou *top*, que é a única posição que

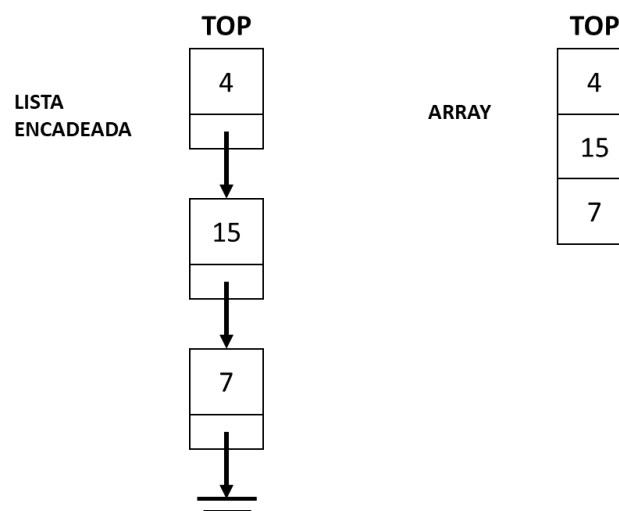
¹ Dependendo da literatura, é possível encontrar esta expressão ao contrário: o último que sai é o primeiro que entra, ou *last out first in (lofi)*.



conseguimos manipular. Mas, em um array ou uma lista encadeada, é possível manipular qualquer posição? Como isso será uma pilha? Veja bem, certamente é possível manipular qualquer posição. Todavia, para que essa manipulação seja caracterizada como uma pilha, só podemos, obrigatoriamente, manipular seu topo. Qualquer outro tipo de manipulação descaracteriza a estrutura de pilha.

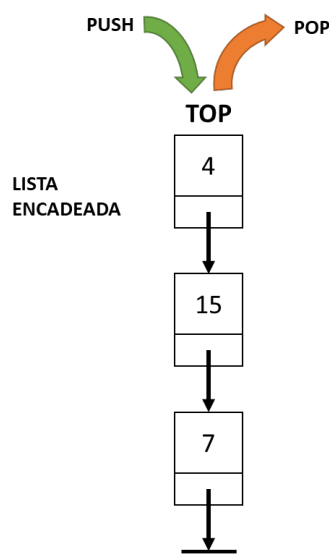
Se implementar a pilha como uma lista encadeada, o *top* seria o equivalente ao *head*, em que só poderíamos inserir ou remover dados diretamente do *head*. Ou seja, implementar uma pilha com listas encadeadas significa fazer uma inserção ou remoção somente no início dela.

Figura 11 – Pilhas representadas com listas encadeadas e com arrays



Acostume-se com os termos! A inserção em uma pilha é chamada de *empilhar*, ou em inglês, *push*. A remoção em uma pilha é *desempilhar* ou *pop*.

Figura 12 – *Push* e *pop* em pilhas com listas encadeadas

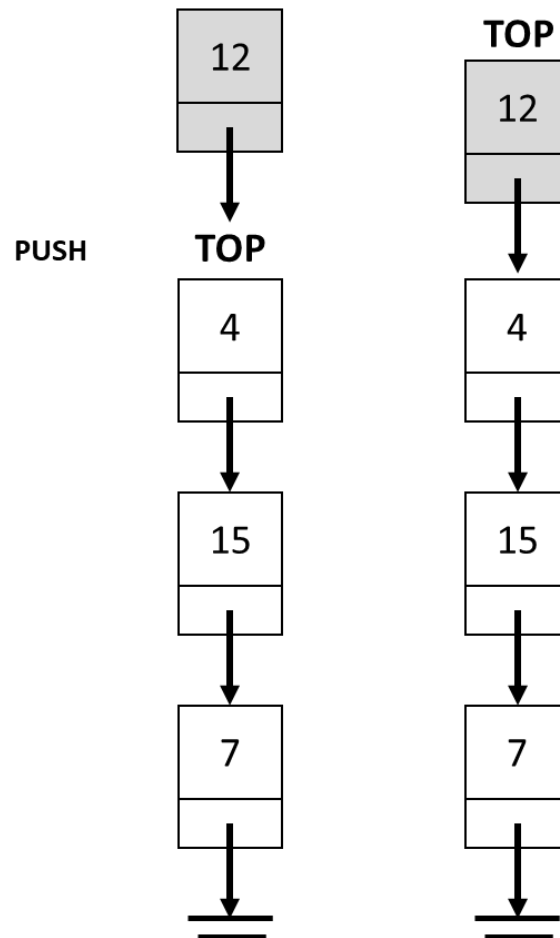




Ao fazer o *push*, se estivermos manipulando uma lista encadeada, primeiro iremos alocar o novo elemento dentro da memória do programa. Em seguida, este novo elemento irá apontar para o *top* da lista (*head*), e então transformamos o novo elemento no novo *top* (*head*) da lista ligada.

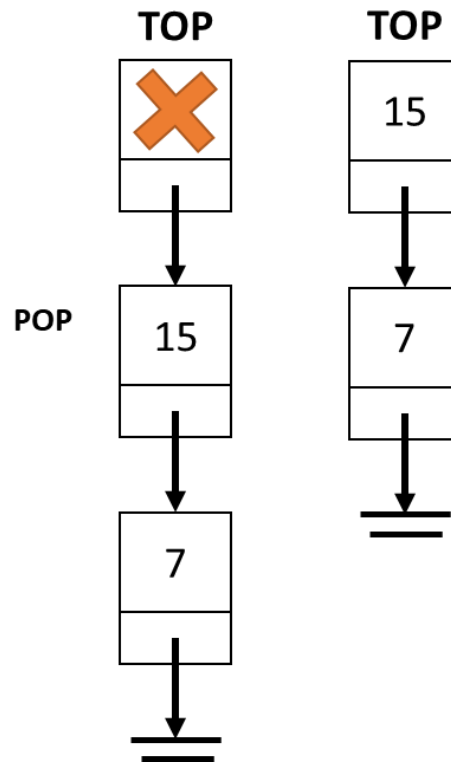
Percebeu que a sequência de passos para inserir na pilha é a mesmo para inserir no início de uma lista encadeada? De fato é o mesmo, só alteramos a terminologia de *head* para *top*. Na Figura 13 temos as duas etapas de inserção.

Figura 13 – Passo a passo do *push* com listas encadeadas



Ao fazer o *pop*, se estivermos manipulando uma lista encadeada, iremos sempre remover o elemento colocado no *top*. Para isso, fazemos o próximo elemento virar o *top*, e em seguida apagamos o elemento antigo do *top*.

Figura 14 – Passo a passo do *pop* com listas encadeadas



TEMA 4 – FILAS (QUEUES)

Uma estrutura de dados do tipo fila, ou *queue*, é uma maneira bastante particular de operar dados na programação. Vamos compreender o conceito de fila com um exemplo de fila de pessoas: imagine que o supermercado do seu bairro está fazendo uma mega promoção de queima de estoque. É claro que você não quer perder nenhum desconto, certo? Portanto, você decide chegar ao supermercado antes que ele abra, para ser o primeiro da fila. Você chega uma hora antes da abertura e fica na primeira posição da fila. Com o passar do tempo, outras pessoas começam a chegar. Naturalmente, todos que chegam depois de vocês entram ao final da fila.

Figura 15 – Fila de supermercado



Crédito: MikeDotta / Shutterstock.

Ao abrir as portas do mercado, quem será o primeiro a entrar? Você! Note que, em uma fila, sempre quem chega primeiro é atendido primeiro, e quem chega por último, entra ao final da fila e será atendido por último. Uma estrutura de dados operando como uma fila, opera com o princípio de *o primeiro que entra é o primeiro que sai*, ou em inglês, *first in first out (fifo)*².

Em resumo, um conjunto de dados (pessoas!) funciona como uma fila que sempre só inserimos no final do conjunto de dados (pessoa entram no final da fila) e sempre que só removermos do início do conjunto de dados (chegou primeiro na fila, é atendido primeiro). Nunca, em hipótese alguma, podemos inserir ou remover dados de maneiras distintas das citadas.

Como aplicações de algoritmos de filas, podemos citar a fila de impressão. Sabe quando você está enviando algum documento para ser impresso em uma impressora e outra pessoa também está enviando, simultaneamente? O algoritmo da fila de impressão irá atender à demanda e imprimir primeiro o arquivo que chegar antes. Chegou primeiro, imprimiu primeiro, e os outros vão sendo colocados ao final da fila. A área de redes de computadores manipula o

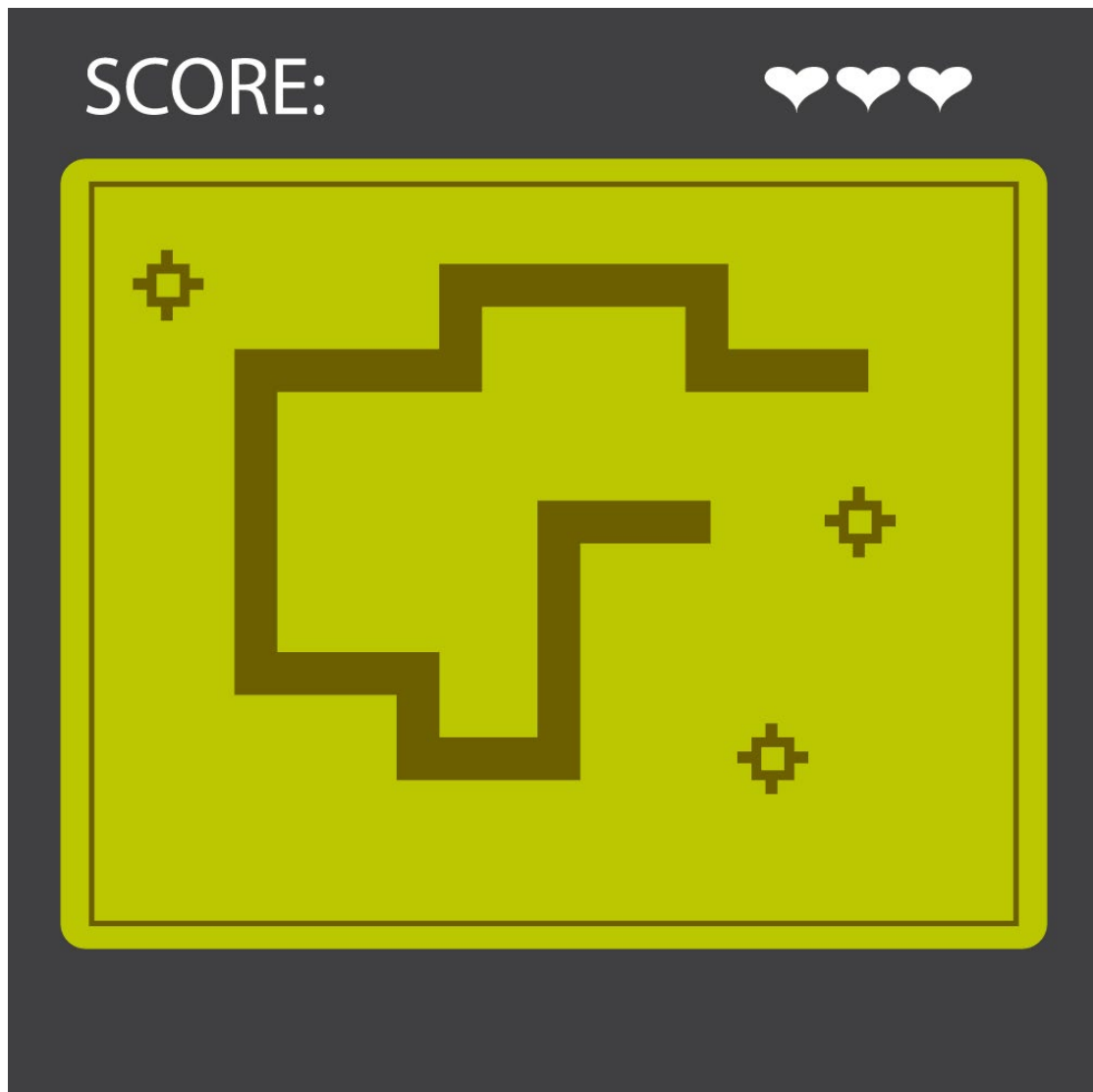
² Dependendo da literatura, é possível encontrar esta expressão ao contrário: *o primeiro que sai é o primeiro que entra*, ou *first out first in (fofi)*.



tráfego da rede por meio de teoria de filas. Um *player* de música pode ter sua fila de reprodução de músicas implementada dessa maneira.

E sabe o famoso jogo da cobrinha (*snake*), que fez muito sucesso em celulares 15 anos atrás? Pois bem, ele pode ser implementado com uma estrutura de fila. A cobra será a fila e cada vez que comer um novo bloquinho, este bloco é anexado ao final da cobra, aumentando sua cauda.

Figura 16 – Jogo *snake*



Crédito: 2DAssets / Shutterstock.

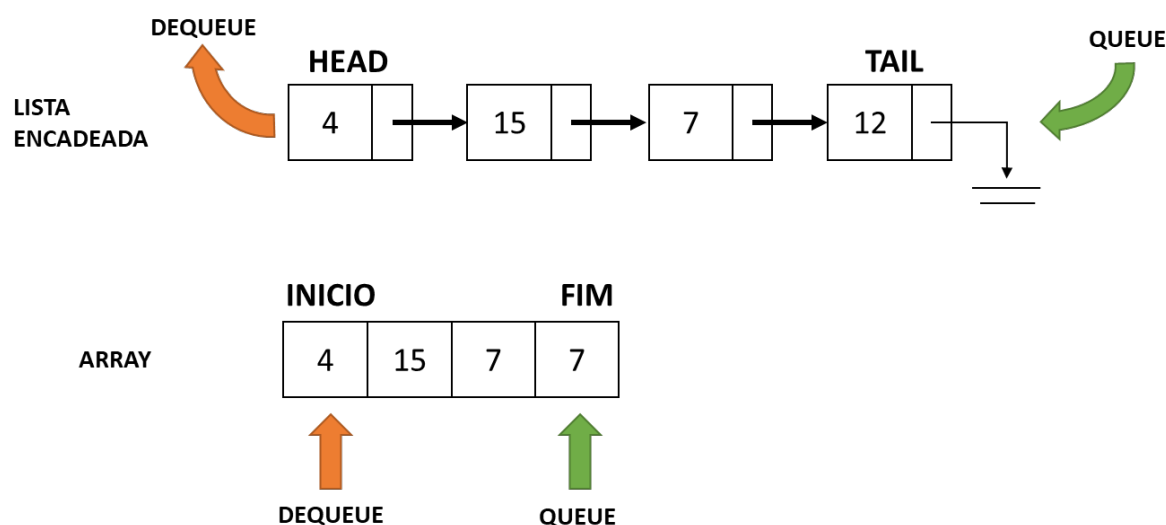
4.1 Construindo e manipulando uma fila

A construção de filas pode ser realizada utilizando algumas das estruturas de dados já conhecidas. Por exemplo, podemos construir pilhas empregando arrays dinâmicos (lista em Python) ou com listas encadeadas.



Na Figura 17, temos graficamente duas representações de filas. Uma envolvendo listas encadeadas, e outra os arrays. Se implementarmos a fila como uma lista encadeada, iremos manipular seu *head*, e só poderíamos remover dados diretamente do *head*. Já a inserção na fila ocorre sempre ao seu final, por isso temos o costume de já armazenar em outra variável esse último elemento, para facilitar seu acesso. Na lista, chamamos o último elemento de *cauda*, ou *tail*. Ou seja, implementar uma fila significa fazer uma inserção (*queue*) no final dela, e fazer a remoção (*dequeue*) no início dela.

Figura 17 – *Queue* e *dequeue* em filas com listas encadeadas e com arrays




4.2 Implementando pilhas e filas

Precisamos agora verificar o código e aprender a implementação de ambos os códigos, de pilhas e de filas, em linguagem Python. As implementações apresentadas neste momento correspondem às arrays dinâmicas (listas em Python). Iremos verificar a implementação simplificada que usufrui das características da linguagem Python, facilitando nosso aprendizado.

4.3 Código da pilha

O código de uma pilha implementada em Python pode ser verificado a seguir. O programa foi construído com listas em Python.



```
#VERSÃO SIMPLIFICADO DO PYTHON
pilha = []
tam = 5

while True:
    print('1 - Inserir na pilha')
    print('2 - Remover da pilha')
    print('3 - Listar a pilha')
    print('4 - Sair')

    op = int(input("Escolha uma opção:"))
    if op == 1:
        dado = int(input('Qual número deseja inserir?'))
        if len(pilha) < 5:
            pilha.append(dado)
        else:
            print('Pilha cheia! Impossível inserir. ')
    elif op == 2:
        if len(pilha) > 0:
            pilha.pop()
        else:
            print('Pilha vazia! Impossível remover. ')
    elif op == 3:
        pilha.reverse()
        for item in pilha:
            print(item)
    elif op == 4:
        print('Encerrando...')
        break
    else:
        print("Selecione outra opção!\n")
```

Note que criamos um código que contém um menu. Nele, o usuário é capaz de inserir um dado numérico na pilha, remover, mostrar a pilha na tela ou encerrar o programa. Como estamos trabalhando com uma lista em Python, podemos utilizar um método existente na linguagem chamado de *append*, que sempre adiciona (faz um apêndice) de um dado após todos os outros. Assim, entendemos que o topo de nossa pilha é sempre o último elemento da lista.

Para fazer a remoção de um dado na pilha, existe outro método, chamado de *pop*. Ele remove sempre o dado de um determinado índice passado como parâmetro. De acordo com a sintaxe da função, caso não passemos nenhum índice como parâmetro, o padrão é o índice -1, ou seja, sempre o último elemento, que é exatamente nosso topo da pilha. Portanto, inserimos no início com *append* e removemos do início com *pop()*, ou *pop(-1)*.



4.4 Código da fila

O código de uma fila implementada em Python pode ser verificado a seguir. Foi criado um código que contém um menu, no qual o usuário pode inserir um dado numérico na fila, remover, mostrar a fila na tela ou encerrar o programa. Como estamos trabalhando com uma lista em Python, podemos utilizar um método existente na linguagem, o *append*, que sempre adiciona (faz apêndice) de um dado após todos os outros. Diferente da pilha, podemos entender que o final de nossa fila é sempre o último elemento da lista.

Para fazer a remoção de um dado na pilha, existe outro método, o *pop*, que remove sempre o dado de um determinado índice passado como parâmetro. Na pilha, havíamos usado o valor padrão de índice, que era o -1. Agora iremos propositalmente passar como parâmetro o valor zero no *pop*. Assim, removeremos sempre o primeiro dado da lista. Portanto, inserimos no final com *append* e removemos do início com *pop(0)*.

```
#VERSÃO SIMPLIFICADO DO PYTHON
fila = []
tam = 5

while True:
    print('1 - Inserir na fila')
    print('2 - Remover da fila')
    print('3 - Listar a fila')
    print('4 - Sair')

    op = int(input("Escolha uma opção:"))
    if op == 1:
        dado = int(input('Qual número deseja inserir?'))
        if len(fila) < 5:
            fila.append(dado)
        else:
            print('Fila cheia! Impossível inserir. ')
    elif op == 2:
        if len(fila) > 0:
            fila.pop(0)
        else:
            print('Fila vazia! Impossível remover. ')
    elif op == 3:
        for item in fila:
            print(item, end=' ')
            print('\n')
    elif op == 4:
        print('Encerrando...')
        break
    else:
        print("Selecione outra opção!\n")
```



FINALIZANDO

Ao longo desta etapa estudamos novas estruturas de dados. Aprendemos as listas encadeadas, que são estruturas de dados com alocação dinâmica e não sequencial, onde cada elemento da lista contém seus respectivos dados e também o endereço para o próximo elemento desta lista.

A lista encadeada contém como principal vantagem a capacidade de ser alocada dinamicamente na memória, e como desvantagem o seu desempenho no ato da leitura dos dados, pois depende do tamanho do conjunto de dados, diferentemente de um array alocado sequencialmente.

Investigamos também as pilhas (*stacks*) e as filas (*queues*). Uma pilha opera com o princípio de FILO, enquanto as filas operam com FIFO. Ambas apresentam a remoção dos dados sempre em seu início (ou topo), enquanto a inserção na pilha ocorre no início (topo) e na fila ao final.



REFERÊNCIAS

ASCENCIO, A. F. G.; ARAÚJO, G. S. **Estruturas de Dados**: algoritmos, análise da complexidade e implementações em JAVA e C/C++. São Paulo: Pearson Prentice Hall 3, 2010.

BHARGAVA, A. Y.; **Entendendo Algoritmos**. Novatec, 2017.

DROZDEK, A. **Estrutura de Dados e Algoritmos em C++**. 4 ed. edição norte-americana (tradução). Cengage Learning Brasil, 2018.

FERRARI, R. *et al.* **Estruturas de Dados com Jogos**. Elsevier, 2014.

KOFFMAN, E. B.; WOLFGANG, P. A. T. **Objetos, Abstração, Estrutura de Dados e Projeto Usando C++**. Grupo GEN, 2008.