# UFPI-CCN-DC- Estrutura de Dados 2024.2

## Relatório da atividade sobre Listas Encadeadas

Davi Soares de Macedo

João Gabriel Miranda Queiroz

Tiago Gabriel Costa Sampaio

## Objetivos

Este relatório irá descrever a construção e execução de um código em Python que representa uma lista encadeada tanto simples quanto dupla, com operações de inserção, exclusão, busca e verificação de duplicidade de elementos. Este tipo de lista permite armazenar e gerenciar uma variedade de elementos. Ainda, foi adicionado uma comparação case-insensitive para assegurar a comparação de strings independentemente de letras maiúsculas e minúsculas. Também, foi implementado um menu independente a fim de modularizar e organizar o código.

## Estrutura do código **simplesmenteEncadeada.py**

O código foi dividido em duas classes principais:

1. Classe Node: Representa um nó na lista encadeada, que armazena um dado e uma referência para o próximo nó.
2. Classe ListaEncadeada: Controla a lista, gerencia a inserção, exclusão e busca dos nós.

## Classe **Node**

Essa classe é responsável por armazenar os elementos de cada nó da lista encadeada e a referência que aponta para o próximo nó (node).

**Estrutura**

**\_\_init\_\_(self, entrada)**: Este é o método construtor, é responsável por inicializar o dado(entrada) em determinado nó. A estrutura **entrada.lower()** assegura que o conteúdo recebido vai ser convertido para caracteres minúsculos, evitando problemas de incompatibilidade devido à capitalização dos caracteres. Também,o atributo **next** é inicializado como **None**, já que, nesse momento, o nó ainda não aponta para outro nó.

class Node:

def \_\_init\_\_(self, entrada):

self.entrada = entrada.lower() *# Armazena o dado em minúsculo*

self.next = None  *# Inicialmente, o nó não aponta para outro nó*

## Classe **ListaEncadeada**

Aqui ocorre a implementação da lista encadeada em si, com a implementação de métodos para gerenciar a lista, como inserir no início, final, buscar elementos e a impressão da lista completa.

**Estrutura**

Iniciamos com o atributo **head**, que é um ponteiro que indica o início da lista. Este atributo será **None** caso a lista esteja vazia.

Métodos da classe ListaEncadeada

**\_\_init\_\_(self)**: Inicializa a lista encadeada vazia, onde o head (cabeça) é None.

class ListaEncadeada:

def \_\_init\_\_(self):

self.head = None *# Inicializa a lista vazia*

**1- ja\_existe(self, entrada)**: Este método tem o fito de verificar se já existe o elemento buscado na lista encadeada. A lista é percorrida do início (**head**) até o final, comparando o elemento “entrada” com cada elemento (**nó**) da lista encadeada. Sempre fazendo a comparação case-insensitive.

def ja\_existe(self, entrada):

node\_atual = self.head

while node\_atual:

if node\_atual.entrada == entrada.lower(): *# Comparação case-insensitive*

return True *# Elemento já existe*

node\_atual = node\_atual.next

return False

**2- inserir\_no\_inicio(self, entrada)**: Este método insere um novo nó no início da lista. Antes da inserção, o método anterior (**ja\_existe**) é chamado para garantir que o elemento a ser inserido ainda não exista na lista. Esse método vai invocar a classe Node para criar um novo nó que vai apontar para o antigo primeiro nó, depois a cabeça da lista vai ser atualizada para o novo elemento inserido.

def inserir\_no\_inicio(self, entrada):

if self.ja\_existe(entrada): # Evita duplicidade

print("Já cadastrado.")

return

new\_node = Node(entrada) # Cria um novo nó

new\_node.next = self.head # O novo nó aponta para o antigo primeiro nó

self.head = new\_node # Atualiza a cabeça da lista

**3- inserir\_no\_final(self, entrada):** Este método insere um novo nó no final da lista.

Novamente, o método **ja\_existe** é chamado para garantir a unicidade dos elementos. Este método vai percorrer a lista até o último nó e em seguida vai apontá-lo para o novo elemento (nó) inserido.

def inserir\_no\_final(self, entrada):

if self.ja\_existe(entrada):

print("Já cadastrado.")

return

new\_node = Node(entrada)

if self.head is None: # Se a lista está vazia

self.head = new\_node

return

ultimo\_node = self.head

while ultimo\_node.next: # Percorre até o último nó

ultimo\_node = ultimo\_node.next

ultimo\_node.next = new\_node # Aponta o último nó para o novo nó

**4- busca(self, entrada)**: Este método simples percorre a lista para buscar um elemento específico. Caso o elemento seja encontrado, retorna True, caso contrário, retorna False.

def busca(self, entrada):

node\_atual = self.head

while node\_atual:

if node\_atual.entrada == entrada.lower():

return True # Elemento encontrado

node\_atual = node\_atual.next

return False # Elemento não encontrado

**5- deletar(self, entrada)**: Este método vai remover um determinado nó da lista, caso o elemento escolhido seja o primeiro da lista, o **head** é atualizado. Caso contrário, vai percorrer a lista a fim de encontrar e remover o nó selecionado. Possui uma cláusula **if** para, caso o elemento escolhido não esteja presente na lista, exibir uma mensagem de aviso.

def deletar(self, entrada):

node\_atual = self.head

if node\_atual is None:

print("A lista está vazia")

returna

if node\_atual.entrada == entrada.lower():

self.head = node\_atual.next *# Atualiza o head*

node\_atual = None *# Remove o nó*

return

node\_anterior = None

while node\_atual and node\_atual.entrada != entrada.lower():

node\_anterior = node\_atual

node\_atual = node\_atual.next

if node\_atual is None:

print(f"O elemento {entrada} não está na lista.")

return

node\_anterior.next = node\_atual.next # Remove o nó

node\_atual = None

**6- printar\_lista(self)**: Percorre a lista e imprime os dados de cada nó em sequência.

A estrutura **end=" "** e **end=" -> "** foram utilizadas para mostrar a lista na mesma linha e com uma seta entre elas, a fim de simular a estrutura de dados que estamos representando.

def printar\_lista(self):

node\_atual = self.head

contador = 0

if node\_atual is None:

print('A lista está vazia.')

return

print("Elementos da lista:", end=" ")

while node\_atual:

print(node\_atual.entrada, end=" -> ")

contador += 1

node\_atual = node\_atual.next

print("None") # Indica o final da lista

print(f"Quantidade de elementos: {contador}")

## Estrutura do código **duplamenteEncadeada.py**

Diferentemente de uma lista encadeada simples, a lista duplamente encadeada possui ponteiros para o **nó anterior** e para o **nó sucessor**, permitindo a navegação bidirecional pela estrutura.

## Classe **Node**

Esta classe define a estrutura de um nó, cada nó irá armazenar um dado, e dois ponteiros, que apontam tanto para o nó anterior quanto posterior.

Este método inicializa um nó, atribuindo o valor **entrada** para o campo **data**. Os ponteiros **next** e **prev** são inicializados como None a fim de indicar que o nó criado não possui sucessor nem antecessor no momento de sua criação.

def \_\_init\_\_(self, entrada):

self.data = entrada

self.next = None

self.prev = None

## Classe **ListaDuplamenteEncadeada**

Esta classe define e gerencia a lista em si, manipulando os nós para realizar operações de inserção, busca, excluir elementos e printar a lista.

**Estrutura**

1-Método Construtor **\_\_init\_\_(self)**:

Este método inicializa a lista com a cabeça (head) da lista apontando para None, indicando que a lista está vazia.

2-**inserir\_ordenado(self, entrada):** Este método vai inserir um novo elemento na lista de maneira que ela vai permanecer ordenada, este processo poderá ocorrer no início, meio ou fim da lista.

Inicialmente, é criado um novo nó invocando a classe Node.

Para inserir o novo nó no início da lista, verificaremos se a lista está vazia, para o nó recém criado se tornar a cabeça da lista.

if self.head is None:

self.head = new\_node

return

Para inserir um nó antes da cabeça da lista, quando o valor inserido for menor que o valor da cabeça, vamos utilizar a seguinte estrutura:

if entrada < self.head.data:

new\_node.next = self.head

self.head.prev = new\_node

self.head = new\_node

return

O novo nó será inserido e os ponteiros serão ajustados, ligando os ponteiros **prev** e **next** da cabeça e do novo nó.

Para inserir um novo nó no meio ou final foi utilizada uma estrutura com um loop que percorre a lista para encontrar a posição adequada para o novo nó, se o novo nó for adicionado no meio da lista, seus ponteiros são ajustados com os do elemento anterior e posterior, a fim de preservar a integridade da lista. Ainda, caso o nó seja adicionado no final da lista, atualizamos o ponteiro **next** para apontar para None.

while atual.next and atual.next.data < entrada:

atual = atual.next

if atual.next is None:

atual.next = new\_node

new\_node.prev = atual

**3-busca(self, entrada):** Este método percorre a lista para verificar se determinado elemento já está presente na lista. Vai percorrer os nós até encontrar o elemento desejado, retorna True caso encontre e False caso contrário.

while node\_atual:

if node\_atual.data == entrada:

return True

node\_atual = node\_atual.next

return False

**4-deletar(self, data):** Este método remove o nó com algum valor determinado pelo usuário, pode remover tanto a cabeça quanto qualquer outro nó da lista.

Caso o nó removido seja a cabeça, o ponteiro **head** é atualizado para apontar para o próximo nó, e o ponteiro **prev** da nova cabeça é ajustado para apontar para None.

Se o nó a ser removido estiver no meio ou fim da lista, os ponteiros dos nós adjacentes são ajustados para “ignorar” o nó que o usuário deseja remover.

if node\_atual.data == data:

self.head = node\_atual.next

if self.head:

self.head.prev = None

node\_atual = None

return

if node\_atual.next:

node\_atual.next.prev = node\_atual.prev

if node\_atual.prev:

node\_atual.prev.next = node\_atual.next

node\_atual = None

**5-printar\_lista(self):** Este método exibe todos os elementos da lista, na mesma linha e com uma seta bifurcada, representando a lista duplamente encadeada, além de utilizar um contador para exibir a quantidade de nós presentes na lista.

print("Elementos da lista:", end=" ")

while node\_atual:

print(node\_atual.data, end=" <-> ")

contador += 1

node\_atual = node\_atual.next

print("None")

## Programa principal **main.py**

O arquivo principal importa os outros códigos e suas classes, gera um menu com opções para o usuário construir e editar listas encadeadas da maneira que for mais conveniente.

# 