Tópico Avançado:
Lista Encadeada de Filmes Lucrativos

Objetivos da aula

- Criar e manipular uma lista encadeada de filmes lucrativos.
- Nesta aula vamos construir uma lista encadeada personalizada para armazenar filmes com lucro acima da média. Ao final da aula você será capaz de:
  - Entender o conceito de nó e ponteiro
  - Implementar classes No e ListaEncadeada
  - Inserir elementos de forma ordenada
  - Iterar sobre nós
  - Comparar lista encadeada com listas nativas do Python

Conceito de Lista Encadeada

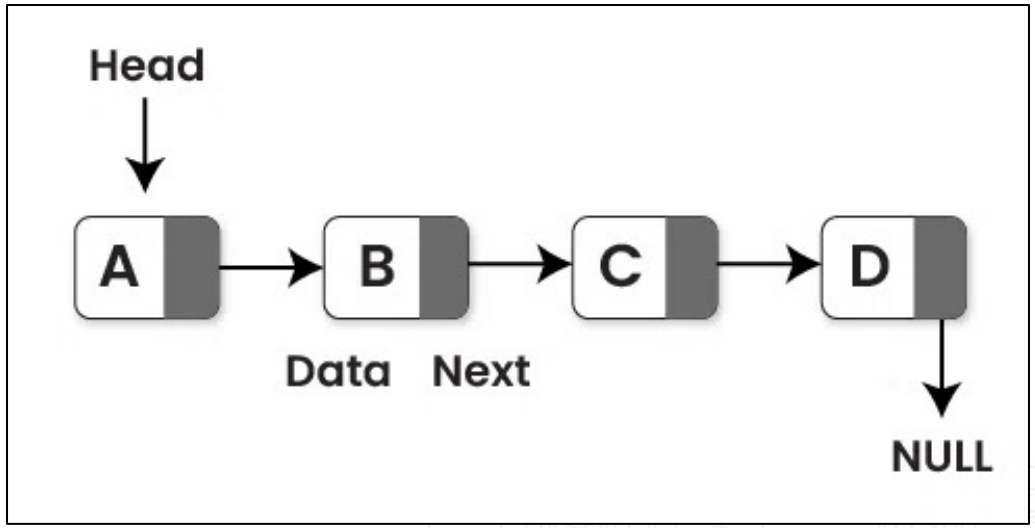
- Uma lista encadeada é uma coleção de elementos (nós), em que cada nó aponta para o próximo.
- Diferente de listas nativas, ela permite inserções e remoções eficientes em qualquer posição sem precisar deslocar elementos.

Conceito de Lista Encadeada

- Além disso, a inserção no início da lista encadeada tem ótima performance, pois basta ligar o novo nó ao primeiro elemento já existente.
- A inserção no final ou no meio da lista também é eficiente, já que não é necessário mover outros elementos na memória, apenas atualizar as ligações (ponteiros) entre os nós.

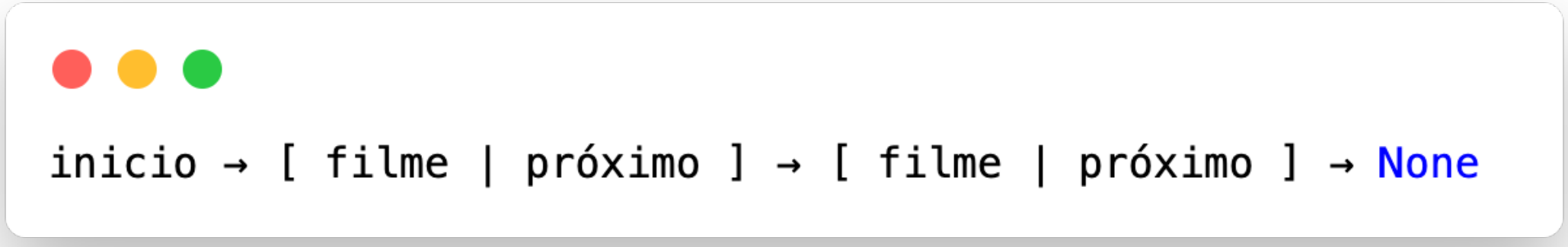
Representação visual

- Uma lista encadeada é formada por nós, onde o **head** aponta para o primeiro nó, cada nó contém um **data** (dado) e um ponteiro **next** para o próximo, e o último nó aponta para **null** (None no caso do Python), indicando o fim da lista.



Representação visual

- Visualmente, a nossa lista ficará assim:



Definindo a classe Nó

- Cada nó contém um objeto Filme e um ponteiro proximo para o próximo nó.

```
class No:
    def __init__(self, filme):
        self.filme = filme
        self.proximo = None
```

Criando a classe ListaEncadeada

- inicio aponta para o primeiro nó da lista. Inicialmente é **None**.

```
class ListaEncadeada:
    def __init__(self):
        self.inicio = None
```

Inserção no início

- Insere um novo filme no início da lista encadeada.

```
from no import No
def inserir_inicio(self, filme):
    no = No(filme)
    no.proximo = self.inicio
    self.inicio = no
```

Teste de inserção

- Testa se a inserção no início cria corretamente o primeiro nó.

```python
from lista_encadeada import ListaEncadeada
from filme import Filme

def test_inserir_inicio():
    lista = ListaEncadeada()
    lista.inserir_inicio(Filme("Matrix", 63000000, 465000000, "Ficção"))
    assert lista.inicio.filme.titulo == "Matrix"
```

Inserção ordenada por lucro

- Insere nós mantendo a lista ordenada pelo lucro, do maior para o menor.

```
from no import No
def inserir_ordenado(self, filme):
    no = No(filme)
    if not self.inicio or filme.lucro() > self.inicio.filme.lucro():
        no.proximo = self.inicio
        self.inicio = no
        return
    atual = self.inicio
    while atual.proximo and atual.proximo.filme.lucro() > filme.lucro():
        atual = atual.proximo
    no.proximo = atual.proximo
    atual.proximo = no
```

Teste de inserção ordenada

- Confirma que o nó com maior lucro fica no início.

```python
from lista_encadeada import ListaEncadeada
from filme import Filme

def test_inserir_ordenado():
    lista = ListaEncadeada()
    f1 = Filme("A", 100, 300, "Drama")
    f2 = Filme("B", 200, 500, "Ficção")
    lista.inserir_ordenado(f1)
    lista.inserir_ordenado(f2)
    assert lista.inicio.filme.titulo == "B"
```

Iterando sobre a Lista Encadeada

- Percorre todos os nós mostrando títulos e lucros.

```python
from lista_encadeada import ListaEncadeada
from filme import Filme

# Criando a lista encadeada
lista = ListaEncadeada()

# Inserindo filmes lucrativos de forma ordenada
lista.inserir_ordenado(Filme("Matrix", 63000000, 465000000, "Ficção"))
lista.inserir_ordenado(Filme("Titanic", 200000000, 2200000000, "Drama"))
lista.inserir_ordenado(Filme("Avatar", 237000000, 2847000000, "Ficção"))
lista.inserir_ordenado(Filme("Joker", 55000000, 1074000000, "Drama"))

# Iterando sobre a lista encadeada
atual = lista.inicio
while atual:
    print(atual.filme.titulo, atual.filme.lucro())
    atual = atual.proximo
```

Comparação com lista nativa

- Diferente de uma lista Python, a lista encadeada permite:
  - Inserções e remoções no meio da lista sem mover elementos
  - Acesso sequencial obrigatório
  - Maior controle sobre memória e ponteiros

Criando lista de filmes lucrativos

- Armazena apenas filmes com lucro acima da média, mantendo ordem decrescente.

```
filmes =
    Filme("Matrix", 63000000, 465000000, "Ficção"),
    Filme("Titanic", 200000000, 2200000000, "Drama"),
    Filme("Avatar", 237000000, 2847000000, "Ficção"),
    Filme("Joker", 55000000, 1074000000, "Drama")
]

media_lucro = sum(f.lucro() for f in filmes) / len(filmes)
lista_lucrativos = ListaEncadeada()
for f in filmes:
    if f.lucro() > media_lucro:
        lista_lucrativos.inserir_ordenado(f)
```

Remoção de um nó

- Remove um filme da lista encadeada sem precisar deslocar elementos.

```
def remover(self, titulo):
    if not self.inicio:
        return
    if self.inicio.filme.titulo == titulo:
        self.inicio = self.inicio.proximo
        return
    atual = self.inicio
    while atual.proximo and atual.proximo.filme.titulo != titulo:
        atual = atual.proximo
    if atual.proximo:
        atual.proximo = atual.proximo.proximo
```

Contando elementos

- Percorre todos os nós para contar quantos filmes estão na lista.

```
def contar(self):
    atual = self.inicio
    cont = 0
    while atual:
        cont += 1
        atual = atual.proximo
    return cont
```

Iteração com Generator (yield)

- Esta função cria um generator, que permite percorrer a lista encadeada nó por nó.
- A palavra **yield** devolve o nó atual e pausa a função, guardando o estado para continuar na próxima iteração.

```
def iter_lista(lista):
    atual = lista.inicio
    while atual:
        yield atual
        atual = atual.proximo
```

Iteração com Generator (yield)

- Exemplo de uso:

```
for no in iter_lista(lista_lucrativos):
    print(no.filme.titulo, no.filme.lucro())
```

- Usando **yield**, conseguimos iterar com for sem acessar diretamente lista.inicio, deixando o código mais organizado e alinhado ao estilo do Python.

Inserção eficiente

- Inserção ordenada evita necessidade de reordenar a lista inteira, diferente de listas Python que precisariam de sort().

- Como cada novo filme já é inserido na posição correta, a estrutura se mantém organizada desde o início, economizando processamento e facilitando buscas posteriores.

Comparação com listas nativas

- Iteração vs acesso por índice: Lista encadeada requer iteração sequencial, não permite acesso direto por índice como lista nativa.

- Enquanto listas Python são mais simples e rápidas para uso geral, a lista encadeada ajuda a entender como os dados são realmente organizados na memória e quando vale criar estruturas personalizadas.

Inserção no final

- Insere no final da lista encadeada.
- Uma vantagem de inserir no fim da lista encadeada é que os elementos mantêm a ordem natural de chegada, o que é útil quando queremos preservar a sequência original dos dados (por exemplo, filmes adicionados na ordem em que foram lidos do arquivo).

```
from no import No
def inserir_fim(self, filme):
    no = No(filme)
    if not self.inicio:
        self.inicio = no
        return
    atual = self.inicio
    while atual.proximo:
        atual = atual.proximo
    atual.proximo = no
```

Teste de inserção no final

- Garante que o filme foi inserido corretamente no final.

```python
from lista_encadeada import ListaEncadeada
from filme import Filme

def test_inserir_fim():
    lista = ListaEncadeada()

    lista.inserir_fim(Filme("Primeiro", 100, 500, "Drama"))
    lista.inserir_fim(Filme("Segundo", 200, 800, "Ação"))

    atual = lista.inicio
    while atual.proximo:
        atual = atual.proximo

    assert atual.filme.titulo == "Segundo"
```

Acesso ao último nó

- Mostra como acessar o último elemento da lista encadeada.

```
atual = lista.inicio
while atual.proximo:
    atual = atual.proximo
print(atual.filme.titulo)
```

Inserção em lista vazia

- Quando a lista está vazia, o primeiro nó é inserido e a referência de inicio da lista aponta para esse nó.
- Se a lista já contém elementos, o novo nó é inserido no início da lista, apontando para o antigo início.
- Se a lista estiver vazia, o novo nó será o único e apontará para None.

Inserção em lista vazia

- Se a lista estiver vazia, self.inicio será None. Então, o novo nó será o primeiro da lista e o proximo do nó será None.
- Se a lista já contiver elementos, o proximo do novo nó será o primeiro elemento existente, e o inicio da lista será atualizado para apontar para o novo nó.

```
from no import No
def inserir_inicio(self, filme):
    no = No(filme)  # Cria o nó com o filme
    no.proximo = self.inicio  # Aponta o nó para o início da lista (ou None se vazia)
    self.inicio = no  # Atualiza o início da lista para o novo nó
```

Remoção do último nó

- Remove o último nó mantendo integridade da lista.

```
def remover_ultimo(self):
    if not self.inicio:
        return
    if not self.inicio.proximo:
        self.inicio = None
        return
    atual = self.inicio
    while atual.proximo.proximo:
        atual = atual.proximo
    atual.proximo = None
```

Benefícios e limitações

- A lista encadeada é uma estrutura de dados estudada principalmente para compreender como os elementos podem ser organizados e ligados dinamicamente na memória.

- A lista encadeada é excelente para fins didáticos e para entender estruturas de dados, mas nem sempre é a opção mais prática no desenvolvimento cotidiano em Python.

Benefícios e limitações

- **Benefícios**:
  - Permite controle direto sobre nós e referências, ajudando a entender como estruturas dinâmicas funcionam internamente.
  - Inserções e remoções são feitas apenas ajustando os ponteiros, sem a necessidade de mover vários elementos.

Benefícios e limitações

- **Limitações:**
  - O acesso aos dados é sempre sequencial, sendo necessário percorrer a lista desde o início.
  - O código é mais extenso e mais complexo que o uso de listas nativas do Python.

Encerramento

- Você aprendeu:
  - Implementar classes No e ListaEncadeada
  - Inserir nós de forma ordenada Iterar e buscar filmes
  - Comparar lista encadeada com listas Python
  - Aplicar TDD em estrutura dinâmica
  - Preparar dados para futuras análises e simulações

PUC Minas