Implementação de Estrutura de Dados: Vetores e Contiguidade

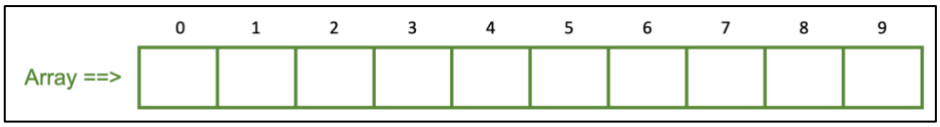
Na aula passada, exploramos o conceito de vetores, que são sequências de dados localizadas em posições consecutivas de memória. Sua principal vantagem é permitir o trabalho com diversas variáveis utilizando apenas um único nome. Em Python, o pacote Numpy é uma das ferramentas mais eficientes para manipular vetores.

Para um entendimento mais detalhado, que não abordamos anteriormente, vamos nos aprofundar nos aspectos técnicos dos vetores. Os vetores (ou arrays) são sequências contíguas de memória, onde cada elemento é uniforme em tipo e tamanho. Devido a essa característica, acessar qualquer elemento do vetor é feito em tempo constante. No entanto, algumas operações, como inserir ou remover um elemento no meio do vetor, podem ser demoradas, uma vez que podem necessitar da realocação de vários elementos.

Agora, utilizando esse conhecimento sobre vetores, imaginemos um cenário em que precisamos desenvolver um script para controlar a entrada de alunos em uma sala de aula. Dada a necessidade de uma estrutura eficiente para armazenar os dados, optaremos pelo uso do numpy.

 # Implementação da estrutura de dados por contiguidade – array

- 1º Crie um vetor vazio, que receberá 10 elementos (“nomes dos alunos”).




```
import numpy as np

vetor_alunos = np.empty (10, dtype=bytearray)

print(f'\nvetor_alunos: {vetor_alunos}')
print(f'\nTipo de estrutura do vetor alunos: {type(vetor_alunos)}')
```

- 2º Usando estrutura de repetição for() insira os nomes no array;

```
for n in range (3):
    vetor_alunos [n] = input (f'\nNome do {str(n+1)}º aluno(a): ')

print(f'\nOs alunos(as) que entraram em sala são: {vetor_alunos} \n')
```

- 3º Apresente o resultado na tela

```
Claudinei@Marcias-MBP Edados_2023 % Python 002_aula_09_08.py
Nome do 1º aluno(a): Maria
Nome do 2º aluno(a): Magareth
Nome do 3º aluno(a): Menergilda
Os alunos que entraram em sala são: ['Maria' 'Magareth' 'Menergilda' None None None None None None None]
```

Faça analise descritiva do resultado:

- -

- Desenvolva um script para verificar se um aluno(a) específico(a) está no vetor. Se estiver, apresente na tela a referência que ocupa e seu nome. Caso contrário, exiba a mensagem “Aluno(a) não encontrado(a)...”.”

Pesquisa Linear de dados

```
aluno = input (f'\nInforme o nome do aluno(a) a pesquisar: ')

for n in range(10):
    if aluno == vetor_alunos[n]:
        print(f'\naluno(a) encontrado(a): índice: {str (n)} - nome: {vetor_alunos[n]} \n')

if vetor_alunos[n] != aluno:
    print(f'\nAluno(a) não encontrado(a)!!! \n')
```

```
Claudinei@Marcias-MBP Edados_2023 % Python 002_aula_09_08.py
vetor_alunos: [None None None None None None None None None None]
Tipo de estrutura do vetor alunos: <class 'numpy.ndarray'>
Nome do 1º aluno(a): José
Nome do 2º aluno(a): João
Nome do 3º aluno(a): Juca
Os alunos(as) que entraram em sala são: ['José' 'João' 'Juca' None None None None None None None]

Informe o nome do aluno(a) a pesquisar: Juca
aluno(a) encontrado(a): índice: 2 - nome: Juca
Aluno(a) não encontrado!!!
```

Implementação de pesquisa linear e **ALTERAÇÃO** de dados

- Desenvolva um script para verificar se um aluno(a) específico(a) está no vetor. Se o nome for encontrado, mostre sua referência e nome na tela. Em seguida, solicite ao usuário que digite um novo nome. Se o nome digitado for o mesmo que o inicialmente pesquisado, exiba a mensagem ‘Você digitou o mesmo nome que pesquisou...’. Caso contrário, atualize o nome no vetor, na mesma posição do nome pesquisado e apresente todo o vetor na tela. Se o nome não estiver no vetor, mostre a mensagem “Aluno(a) não encontrado(a)...”

Pesquisa linear e alteração de dados

```
import numpy as np

aluno = input('\nInforme o nome do aluno(a) a pesquisar para alterar seu nome: ')

for n in range(3):
    if aluno == vetor_alunos[n]:

        print(f'\naluno(a) encontrado(a): índice: {str(n)} - nome: {vetor_alunos[n]} \n')
        aluno_alterar = input('Digite as alterações: ')

        if aluno == aluno_alterar:
            print('\nVocê digitou o mesmo nome que pesquisou...')
        else:
            vetor_alunos[n] = aluno_alterar
        break
else:
    print('Aluno(a) não encontrado(a)!!!')

print(f'\nNome dos alunos(as): {vetor_alunos} \n')
```

```
Claudinei@Marcias-MBP Edados_2023 % Python 002_aula_09_08.py
vetor_alunos: [None None None None None None None None None None]
Tipo de estrutura do vetor alunos: <class 'numpy.ndarray'>
Nome do 1º aluno(a): Jose
Nome do 2º aluno(a): Joao
Nome do 3º aluno(a): Juca
Os alunos(as) que entraram em sala são: ['Jose' 'Joao' 'Juca' None None None None None None None]

Informe o nome do aluno(a) a pesquisar para alterar seu nome: Juca
aluno(a) encontrado(a): índice: 2 - nome: Juca
Digite as alterações: Juca Batista
Nome dos alunos(as): ['Jose' 'Joao' 'Juca Batista' None None None None None None None]
```

Implementação de pesquisa linear e **EXCLUSÃO** de dados

- Desenvolva um script para pesquisar um aluno(a) específico(a) no vetor. Se o nome for localizado, exiba sua referência e o nome na tela. Em seguida, pergunte ao usuário se deseja excluir esse aluno(a) do vetor, oferecendo as opções 'sim' ou 'não', transforme qualquer uma das respostas em maiúsculas.Se a resposta for 'sim', remova o aluno(a) da respectiva posição, caso contrário encerre o loop. Se o nome não for encontrado no vetor, apresente a mensagem “Aluno(a) não encontrado(a)...”. Ao concluir, mostre todos os alunos presentes no vetor."

Pesquisa linear e **EXCLUSÃO** de dados

```
aluno = input('\nInforme o nome do aluno(a) a ser pesquisado: ')

for n in range(len(vetor_alunos)):
    if aluno == vetor_alunos[n]:

        print(f'\naluno(a) encontrado(a): índice: {str(n)} - nome: {vetor_alunos[n]} \n')
        deseja_excluir = input('Deseja excluir esse aluno(a) do vetor? (sim/não): ').upper()

        if deseja_excluir == 'SIM':
            vetor_alunos = np.delete(vetor_alunos, n)
            print('\nAluno(a) excluído(a) com sucesso!')
        else:
            print('\nOperação cancelada pelo usuário.')
        break
else:
    print(f'\nAluno(a) não encontrado(a)!!!')

print(f'\nNome dos alunos(as): {vetor_alunos} \n')
```

```
Claudinei@Marcias-MBP Edados_2023 % Python 002_aula_09_08.py

vetor_alunos: [None None None None None None None None None None]

Tipo de estrutura do vetor alunos: <class 'numpy.ndarray'>

Nome do 1º aluno(a): Joao

Nome do 2º aluno(a): Jose

Nome do 3º aluno(a): Juca

Os alunos(as) que entraram em sala são: ['Joao' 'Jose' 'Juca' None None None None None None None]


Informe o nome do aluno(a) a ser pesquisado: Joao

aluno(a) encontrado(a): índice: 0 - nome: Joao

Deseja excluir esse aluno(a) do vetor? (sim/não): sim

Aluno(a) excluído(a) com sucesso!

Nome dos alunos(as): ['Jose' 'Juca' None None None None None None None]
```

Faça analise descritiva do código ou blocos do código:

- -

Implementação de **ORDENAÇÃO** de arrays

- Desenvolva um script que crie um array com 10 posições. Receba o nome de 4 alunos(as) e armazene-os no array. Em seguida, organize os nomes em ordem ascendente e apresente-os na tela.

ORDENAÇÃO de arrays

```python
def custom_sort(item): # com número menor de itens
    return item if item is not None else "zzzz"

# Ordenando o vetor usando a função custom_sort e reatribuindo ao próprio vetor
vetor_alunos[:] = sorted(vetor_alunos, key=custom_sort)

print(f'\nNome dos alunos(as) após ordenação: {vetor_alunos} \n')

# Supondo que vetor_alunos já possui informações e está completamente preenchido:
# Orden ascendente
vetor_alunos = np.array(['Zen','Marga','Al','Bru','Car','Dia','Ev','Fáb','Gab','Hel'], dtype=object)

print(f'\nNome dos alunos(as) após ordenação: {sorted(vetor_alunos)} \n')

# Orden descendente
print(f'\nNome dos alunos(as) após ordenação descendente: {sorted(vetor_alunos, reverse=True)} \n')
```

```
Claudinei@Marcias-MBP Edados_2023 % Python 002_aula_09_08.py
vetor_alunos: [None None None None None None None None None None]
Tipo de estrutura do vetor alunos: <class 'numpy.ndarray'>
Nome do 1º aluno(a): Zenaide
Nome do 2º aluno(a): Marta
Nome do 3º aluno(a): Almerindo
Os alunos(as) que entraram em sala são: ['Zenaide' 'Marta' 'Almerindo' None None None None None None None]

Nome dos alunos(as) após ordenação: ['Almerindo' 'Marta' 'Zenaide' None None None None None None None]

Nome dos alunos(as) após ordenação: ['Al', 'Bru', 'Car', 'Dia', 'Ev', 'Fáb', 'Gab', 'Hel', 'Marga', 'Zen']

Nome dos alunos(as) após ordenação descendente: ['Zen', 'Marga', 'Hel', 'Gab', 'Fáb', 'Ev', 'Dia', 'Car', 'Bru', 'Al']
```

Atenção: A função **custom_sort** garante que os valores None sejam considerados "maiores" durante a ordenação, fazendo com que eles fiquem no final do vetor.

Implementação de arrays: sem **valores duplicados**

- Desenvolva um vetor com 3 posições que não aceite nomes repetidos ou "duplicatas". Popule o vetor organizando-o em ordem alfabética reversa. No final, apresente um nome por linha na tela.

```
import numpy as np

vetor_nomes = np.empty(3, dtype=object)
nomes_inseridos = 0  # Controlar o número de nomes inseridos

while nomes_inseridos < 3:
    nome = input(f'\nDigite o {nomes_inseridos + 1}º nome: ')

    # Verifica se o nome já existe no vetor
    if nome not in vetor_nomes:
        vetor_nomes[nomes_inseridos] = nome
        nomes_inseridos += 1
    else:
        print("Este nome já foi inserido. Por favor, digite um nome diferente.")

# Ordena o vetor em ordem alfabética reversa
nomes_ordenados = sorted([nome for nome in vetor_nomes if nome is not None], reverse=True)

# Imprime cada nome em uma linha separada
print()
for nome in nomes_ordenados:
    print(f'Nomes em ordem reversa: {nome}')

print()
```

```
Claudinei@Marcias-MBP Edados_2023 % Python 002_aula_09_08.py

Digite o 1º nome: Ana

Digite o 2º nome: Beatriz

Digite o 3º nome: Zico

Os nomes em ordem reversa: Zico
Os nomes em ordem reversa: Beatriz
Os nomes em ordem reversa: Ana

Claudinei@Marcias-MBP Edados_2023 %
```

Faça analise descritiva do código ou blocos do código:

- -

Arrays x Listas

Em Python, tanto listas quanto arrays são estruturas de dados utilizadas para armazenar coleções de itens. Contudo, existem diferenças fundamentais entre eles:

Arrays:

- Armazenam elementos de um único tipo.
- Precisam ser importados para serem usados (from array import array) ou, caso contrário, é necessário instalar a biblioteca numpy e importá-la (import numpy as np).
- Têm funcionalidades limitadas se comparadas às listas, mas oferecem operações básicas como append e pop. A principal vantagem é a eficiência no armazenamento e manipulação de elementos homogêneos.
- O pacote NumPy, amplamente utilizado em computação científica e análise de dados, oferece opções poderosas e uma vasta gama de operações otimizadas, especialmente projetadas para operações matemáticas e lógicas em grandes conjuntos de dados.

Listas:

- Armazenam elementos de diferentes tipos de dados, como inteiros, strings, outros objetos etc. Por exemplo, [1, "a", 3.5, [1,2,3]].
- Não requerem nenhuma biblioteca adicional para serem usadas.
- São mais flexíveis e oferecem uma gama mais ampla de operações embutidas, como inserção, remoção, anexação etc.
- São mais lentas em comparação com arrays devido à flexibilidade e gama de operações que oferecem.

Referências: MCKINNEY (2017), VANDERPLAS (2018), JOHANSSON (2018).

Listas x Listas Simplesmente encadeadas:

Listas:

- São implementadas internamente de maneira dinâmica. Isso significa que, à medida que itens são adicionados à lista, o Python pode realocar memória para acomodar o crescimento da lista.
- Quando adicionamos ou removemos elementos em uma lista, todos os elementos subsequentes podem precisar ser deslocados na memória para manter a contiguidade. Isso é gerenciado automaticamente pela implementação da lista em Python.
- Quando acessamos um elemento em uma lista em Python, é uma operação de tempo constante O(1) (lê-se "big O de 1"), graças à contiguidade dos dados. Isso significa que o tempo para acessar um elemento não depende do tamanho da lista. No entanto, a inserção e remoção de elementos (exceto no final da lista) são operações de tempo linear (O(n)), pois podem exigir o deslocamento de todos os outros elementos.
- Tende a ser mais eficiente em termos de memória para armazenar os próprios elementos, pois não há necessidade de armazenar referências adicionais para cada elemento.
- Ideal para usos gerais onde você precisa de acesso rápido por índice e as inserções/remoções não são frequentemente feitas no meio da lista.

Listas Simplesmente encadeadas:

- Consistem em uma série de nós. Cada nó contém um valor e uma referência para o próximo nó. Diferentemente de arrays, não há alocação contínua de memória; cada nó é alocado individualmente.
- O acesso a um elemento requer percorrer a lista desde o primeiro nó até o desejado, resultando em um tempo de acesso O(n). Contudo, com uma referência direta ao nó anterior ao ponto de inserção ou remoção, estas operações são O(1). Sem essa referência, encontrar uma posição específica leva O(n).
- Embora cada nó requeira memória adicional para armazenar a referência ao próximo, oferece flexibilidade nas operações de inserção e remoção. No entanto, isso pode levar a um aumento no uso de memória comparado a listas nativas.

• São especialmente úteis para estruturas como listas ligadas, dicionários, árvores e grafos, onde a contiguidade física não é essencial, mas as conexões, através dos ponteiros, são vitais.

• Em comparação com arrays, listas encadeadas geralmente são mais rápidas para criação e adição de dados devido ao seu mecanismo de vinculação. No entanto, o acesso aos elementos tende a ser mais lento.

Referências: MCKINNEY (2017), VANDERPLAS (2018), JOHANSSON (2018).

Lista simplesmente encadeada: Sua estrutura é uma coleção de itens onde cada item detém uma posição relativa aos demais.

Operações possíveis sobre lista simplesmente encadeadas.

⇒ List(): cria uma lista que está vazia;

⇒ add(item): insere um item novo à lista;

⇒ remove(item): remove um item da lista;

⇒ search(item): procura o item na lista;

⇒ isEmpty(): verifica se a lista está vazia;

⇒ size(): retorna o número de itens na lista.

⇒ append(item): adiciona um item ao final da lista;

⇒ index(item): retorna a posição do item na lista;

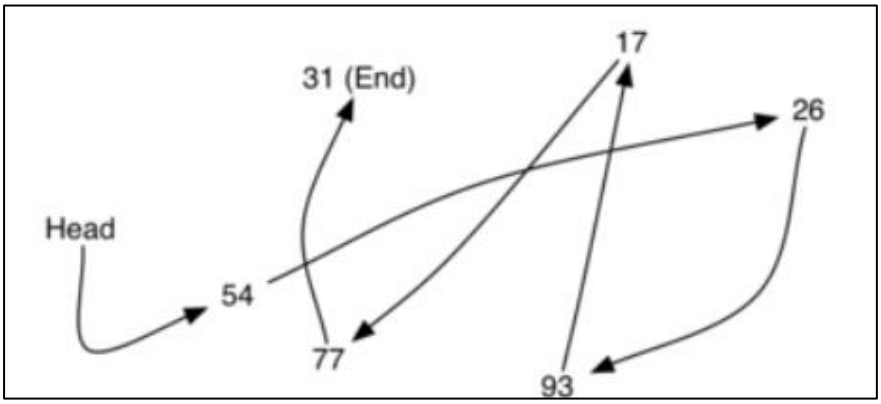
⇒ insert(pos, item): adiciona um novo item à lista na posição pos;

⇒ pop() remove e retorna o último item da lista;

⇒ pop(pos) remove e retorna o item na posição pos.

Lista: Simplesmente encadeada ou ligadas

Para implementá-la precisamos ter certeza de que podemos manter o posicionamento relativo dos itens. Portanto devemos manter informações da posição relativa de cada item, seguindo por exemplo um link.



Posição Relativa Mantida por links.

Observe que a localização do primeiro item da lista está bem specificada e recebe o nome de cabeça (*head)*. Esse primeiro item nos dirá onde está o segundo, o terceiro e assim por diante. o último item precisa saber que não há próximo item.

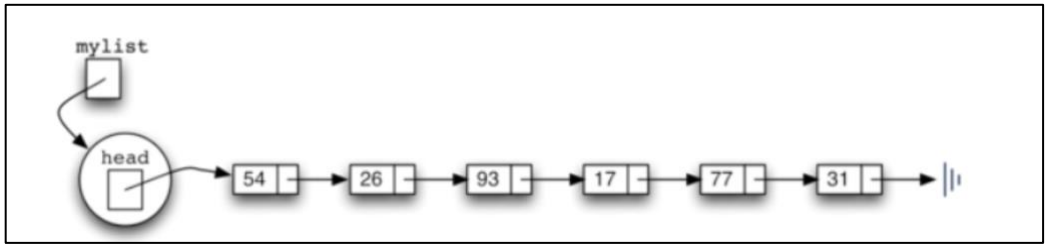
Para construir uma lista simplesmente encadeada ou ligada ao **nó** devemos criar o nó base ou **Node**.

⇒ Cada nó deve conter no mínimo duas informações.

- Primeiro, o nó deve conter um item da lista - chamado de **campo de dado;**
- Cada nó deve conter uma referência para o próximo nó.

⇒ Para construir um nó, você precisa:

- fornecer os dados valores dos dados iniciais para o nó.
- Executar a declaração de atribuição que produzirá o nó



Representação de uma lista simplesmente encadeada. Fonte: https://panda.ime.usp.br/pythonds/static/pythonds_pt/03-EDBasicos/21-ListaImplementacao.html

Implementando lista: Simplesmente encadeada ou ligadas

```python
# Definindo a função que cria um nó.
def create_node(initdata):
    return {'data': initdata, 'next': None}

# Função obtem valor armazenado em um nó
def get_data(node):
    return node['data']

# Função obtem a referência ao próximo nó de um nó
def get_next(node):
    return node['next']

# Função definir (ou alterar) o valor armazenado em um nó
def set_data(node, newdata):
    node['data'] = newdata

# Função para definir (ou alterar) a referência ao próximo
#  nó de um nó
def set_next(node, newnext):
    node['next'] = newnext

# Cada nó é um dicionário com duas chaves: "data" (os dados armazenados no nó) e
# "next" (a referência para o próximo nó)
```

Note que cada nó é um dicionário com duas chaves: "data" (os dados armazenados no nó) e # "next" (a referência para o próximo nó)

Utilizando a Lista simplesmente encadeada implementada

```python
# Criando um nó com o valor "primeiro"
# Neste momento, o nó não tem referência para um próximo nó
node1 = create_node('Abacate')

# Criando outro nó com o valor "segundo". Da mesma forma, esse
#  nó também não tem uma referência para um próximo nó
node2 = create_node('Abóbrinha')

# Conectando o primeiro nó ao segundo. Estamos definindo que o
# próximo nó após "node1" é "node2"
set_next(node1, node2)

# Imprimindo os dados armazenados no primeiro nó.
print(f'\nO primeiro nó é: {get_data(node1)}')

# Imprimindo os dados armazenados no nó que vem após
# "node1", que é "node2".
print(f'\nO segundo nó é: {get_data(get_next(node1))} \n')
```

```
Claudinei@Marcias-MBP Edados_2023 % Python 002_aula_09_08.py

O primeiro nó é: Abacate

O segundo nó é: Abóbrinha
```

Referências:

ASCENCIO, Ana Fernanda Gomes; ARAÚJO, Graziela dos Santos de. **Estruturas de dados : algoritmos, análise da complexidade e implementações.** São Paulo : Pearson Prentice Hall, 2010. ISBN 978-85-7605-881-6

RAMALHO, Luciano. **Fluent Python: Clear, Concise, and Effective Programming".** Editora O'Reilly Media. Agosto de 2021

MCKINNEY, W. **Python para Análise de Dados: Tratamento de Dados com Pandas, NumPy e IPython. 2. ed**. São Francisco: O'Reilly Media, 2017.

VANDERPLAS, **Python Data Science Handbook: Essential Tools for Working with Data. São Francisco:** O'Reilly Media, 2016.

JOHANSSON, **Numerical Python: Scientific Computing and Data Science Applications with Numpy, SciPy and Matplotlib (2ª ed.).** Nova Iorque: Apress, 2018.