Entrega-04-ED - Teoria

1. BUBBLE SORT

Def.: algoritmo de ordenação. Ele compara pares de elementos adjacentes e os troca se estiverem fora de ordem, ou seja, é um MÉTODO DE ORDENAÇÃO SEQUENCIAL.

Ele examina cada conjunto de elementos adjacentes na string, da esquerda para a direita, trocando suas posições se estiverem fora de ordem, dessa forma ele repete o processo até que consiga percorrer toda a string e não encontrar dois elementos que precisem ser trocados

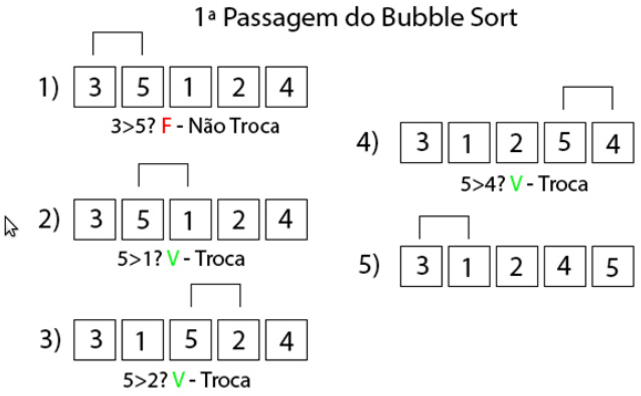


Figura 2: Esquema de funcionamento do Buble Sort

Uma das estruturas do bubble sort em python:

```
def bubble_sort(arr):
    n = len(arr)
    for i in range(n):
        # Últimos i elementos já estão ordenados
        for j in range(n-i-1):
            if arr[j] > arr[j+1]:
                # Troca os elementos de posição
                arr[j], arr[j+1] = arr[j+1], arr[j]
    return arr
```

obs.: O código a cima define a função de nome bubble_sort e recebe uma lista/array como entrada.

Exemplo de aplicação do bubble sort:

```
# Criamos uma lista de numeros desordenada
numeros = [64, 34, 25, 12, 22, 11, 90]

# inserimos dentro do método
bubble_sort(numeros)
```

```
---
o que acontece?
n = len(numeros) -> calculamos o tamanho da lista e guardamos o valor na variável n

for i in range(n): -> inicio do loop externo que irá percorrer todos os elementos/posições da lista ...
                     vai de (0 até n-1)

for j in range(n-i-1): -> inicio do loop interno que percorre os elementos da lista que ainda
                          NÃO FORAM ORDENADOS
OBS:porque n-i ? os últimos elementos já estarão em suas posições ordenadas, logo não precisamos considerá-los

if arr[j] > arr[j+1]: -> condição que verifica se o elemento atual é maior que o próximo
                        , se for True, então trocamos os "elementos"

arr[j], arr[j+1] = arr[j+1], arr[j]   Linha que troca os "elementos" se o if for verdadeiro
                                      -> a sintaxe do python fácilita, dessa forma não precisamos usar
                                      uma varíavel auxiliar

 return numeros                      -> retorna a lista ordenada para nós
```

2. Insertion Sort

O algoritmo de ordenação por inserção (Insertion Sort) é um método eficiente para ordenar pequenos conjuntos de elementos. Ele funciona construindo uma sequência ordenada de elementos um de cada vez, pegando um elemento da lista e inserindo-o na posição correta.

#Para explicar mais detalhadamente o funcionamento dessa ferramenta irei utilizar de um exemplo e de comentários ao código a seguir:

```
class Medicamento: #1
    def __init__(self, nome, preco): #2 #Esta linha define o método __init__ da classe Medicamento.
        self.nome = nome #3
        self.preco = preco #4

def insertion_sort_farmacia(medicamentos): #5
    for i in range(1, len(medicamentos)): #6
        chave = medicamentos[i] #7
        j = i - 1 #8

        while j >= 0 and chave.preco < medicamentos[j].preco: #9
            medicamentos[j + 1] = medicamentos[j] #10
            j -= 1 #11

        medicamentos[j + 1] = chave #12
```

- #1 A classe Medicamento é usada para representar os medicamentos com seus atributos nome e preço.
- #2 O método __init__ é chamado automaticamente quando um objeto da classe é criado. O parâmetro self é uma referência ao próprio objeto sendo criado, e nome e preco são parâmetros que o usuário passa ao criar um novo objeto da classe.
- #3 self.nome é um atributo da instância (objeto) da classe Medicamento, e nome é o parâmetro que o usuário fornece ao criar um novo objeto. Isso significa que, quando você cria uma instância da classe Medicamento e fornece um valor para nome, esse valor é atribuído ao atributo nome do objeto.

- #4 self.preco é um atributo da instância da classe Medicamento, e preco é o parâmetro que o usuário fornece ao criar um novo objeto. O valor fornecido para preco é atribuído ao atributo preco do objeto.

- #5 A função insertion_sort_farmacia recebe uma lista de medicamentos e a ordena com base nos preços.
- #6 o for usado é um loop que itera sobre a lista de medicamentos a partir do segundo elemento
- #7 chave é uma variável que armazena temporariamente o medicamento atual que estamos considerando para inserção na posição correta.
- #8 A variável j é usada para comparar o preço do medicamento atual com os preços dos medicamentos anteriores na lista.
- #9 o loop while continua enquanto j é maior ou igual a zero e o preço do medicamento atual é menor que o preço do medicamento na posição j na lista.
- #10 Se o preço do medicamento atual for menor do que o preço do medicamento na posição j, o medicamento na posição j é deslocado para a direita para abrir espaço para o medicamento atual.
- #11 O índice j é então decrementado para comparar o medicamento atual com o próximo medicamento anterior na lista.
- #12 Após o término do loop while, o medicamento atual é inserido na posição correta na sequência ordenada, que pode ser a posição original de j ou a posição à direita da última comparação bem-sucedida.

Agora iremos popular a nossa classe Medicamentos e inseri-la em uma lista

```
#Cadastro e passagem de parâmetros dos produtos
medicamento1 = Medicamento("Paracetamol", 15.50)
medicamento2 = Medicamento("Aspirina", 10.20)
medicamento3 = Medicamento("Amoxicilina", 25.00)
medicamento4 = Medicamento("Ibuprofeno", 12.75)

#cria-se uma lista com os produtos cadastrados
lista_medicamentos = [medicamento1, medicamento2, medicamento3, medicamento4]
```

A função insertion_sort_farmacia é chamada para ordenar a lista por preço, e por fim, a lista ordenada é impressa

```
insertion_sort_farmacia(lista_medicamentos)

print("+======================================+")
print("|            Insertion Sort           |")
print("+======================================+")
print("\nLista de medicamentos ordenada por preço:\n")
for medicamento in lista_medicamentos:
    print(f"{medicamento.nome}: R${medicamento.preco:.2f}")
```

💡 Em resumo o algoritmo percorre a lista de medicamentos da esquerda para a direita, considerando cada elemento um por vez. Ele compara o preço do medicamento atual com os preços dos medicamentos anteriores na lista, movendo os elementos maiores para a direita até encontrar a posição correta para inserir o medicamento atual.

3. Selection Sort

O Selection Sort é um algoritmo de ordenação simples, mas não muito eficiente para conjuntos de dados grandes. Ele funciona selecionando repetidamente o menor (ou maior, dependendo da ordem desejada) elemento da lista e trocando-o com o primeiro elemento não ordenado.

#Para explicar mais detalhadamente o funcionamento dessa ferramenta irei utilizar de um exemplo e de comentários ao código a seguir:

```
class Medicamento: #1
    def __init__(self, nome, preco): #2
        self.nome = nome #3
        self.preco = preco #4
```

Descrição das linhas

1. A classe Medicamento é definida com dois atributos, nome e preco, representando o nome e o preço do medicamento, respectivamente.
2. é um método construtor, chamado automaticamente quando um novo objeto da classe é criado.
3. Dentro do método __init__, essa linha atribue o valore do parâmetro nome aos atributos da classe Medicamento.
4. Dentro do método __init__, essa linha atribue o valore do parâmetro preco aos atributos da classe Medicamento.

```
def selection_sort(medicamentos): #5
    n = len(medicamentos) #6

    for i in range(n - 1): #7
        indice_menor = i #8

        for j in range(i + 1, n): #9
            if medicamentos[j].preco < medicamentos[indice_menor].preco: #10
                indice_menor = j

        medicamentos[i], medicamentos[indice_menor] = medicamentos[indice_menor], medicamentos[i] # 11
```

5. A função selection_sort implementa o algoritmo de Selection Sort para ordenar a lista de medicamentos com base no preço.
6. Determina o comprimento da lista de medicamentos, ou seja, o número de elementos na lista. Isso será usado para controlar os loops de iteração.
7. Este é o loop externo, que percorre a lista de medicamentos até o penúltimo elemento.

8. Inicializa a variável indice_menor com o valor do índice atual do loop externo.
9. Este é o loop interno que começa do próximo elemento após o elemento atual do loop externo até o final da lista.
10. Compara os preços dos medicamentos nos índices j e indice_menor. Se o preço do medicamento no índice j for menor, atualiza indice_menor para j.
11. Após encontrar o índice do menor preço no loop interno, realiza a troca dos medicamentos nos índices i e indice_menor

```
def exibir_medicamentos(medicamentos): #12
    for medicamento in medicamentos: # 13
        print(f"{medicamento.nome}: R${medicamento.preco:.2f}")
```

12. função exibir_medicamentos é responsável por imprimir na tela os medicamentos, mostrando o nome e o preço formatado.
13. Este é um loop que percorre cada objeto da lista medicamentos e associa cada objeto à variável medicamento durante cada iteração.

Agora populamos a classe medicamentos e inserimos dentro de uma lista

```
#Cadastro e passagem de parâmetros dos produtos
medicamento1 = Medicamento("Paracetamol", 15.50)
medicamento2 = Medicamento("Aspirina", 10.20)
medicamento3 = Medicamento("Amoxicilina", 25.00)
medicamento4 = Medicamento("Ibuprofeno", 12.75)

#cria-se uma lista com os produtos cadastrados
lista_medicamentos = [medicamento1, medicamento2, medicamento3, medicamento4]

selection_sort(lista_medicamentos) # chamada para ordenar a lista de medicamentos com base nos preços.

print("+====================================+")
print("|            Selection Sort          |")
print("+====================================+")
print("\nLista de medicamentos ordenada por preços:")

#chamada para mostrar a lista de medicamentos ordenada na tela.
exibir_medicamentos(lista_medicamentos)
```

4. Lista Encadeada

Def.: Uma lista encadeada é uma estrutura de dados dinâmica que consiste em um conjunto de elementos, cada um contendo um valor e um ponteiro para o próximo elemento da lista. Em outras palavras, cada elemento é armazenado separadamente e possui um ponteiro para o próximo elemento. O primeiro elemento da lista é chamado de cabeça e o último elemento não aponta para nenhum outro elemento, o que o torna o nó final da lista.

- As listas encadeadas são usadas para armazenar dados de forma dinâmica, permitindo a inserção e remoção de elementos em tempo constante.

- Elas são úteis quando precisamos de uma estrutura de dados que possua um tamanho dinâmico, sem valor máximo pré-definido. Por exemplo, arranjos não são a estrutura de dados ideal se desejamos inserir elementos no meio da estrutura de dados. Se usarmos um arranjo e precisarmos inserir um elemento no meio da sequência, precisamos mover todos os elementos do arranjo 1.

> 💡 Listas encadeadas possuem duas desvantagens principais. Primeiro, precisamos de espaço adicional para armazenar os ponteiros. Segundo, se quisermos acessar um elemento em uma dada posição da lista, precisamos percorrer todos os elementos anteriores a ele na lista 1.

Abaixo está um exemplo de implementação de uma lista encadeada em Python:

```
class NodoLista:
    """Esta classe representa um nodo de uma lista encadeada."""
    def __init__(self, dado=0, proximo_nodo=None):
        self.dado = dado
        self.proximo = proximo_nodo


    def __repr__(self):
        return '%s -> %s' % (self.dado, self.proximo)


class ListaEncadeada:
    """Esta classe representa uma lista encadeada."""
    def __init__(self):
        self.cabeca = None


    def __repr__(self):
        return " [" + str (self.cabeca) + "]"
```

Observe que nas definições acima criamos duas classes. A classe NodoLista define apenas um nodo (ou nó) da lista. Cada nodo da nossa lista possui um campo dado e um campo próximo. O campo próximo é um apontador para o próximo nodo da lista. Essa cadeia de nodo referenciando outro nodo mostra o caráter encadeado da lista. A classe ListaEncadeada contém apenas um apontador para o primeiro elemento da lista. O primeiro e o último elementos de uma lista são muito importantes, por isso recebem nomes especiais: cabeça e cauda da lista, respectivamente 1.

---

5. Listas circulares

Def.: Listas circulares são estruturas de dados onde o último elemento da lista está ligado ao primeiro, formando um ciclo. Ou seja, as listas circulares, têm a peculiaridade de que, ao chegarmos ao seu final e chamarmos o método próximo, podemos voltar a encontrar o mesmo elemento inicial. Em Python, isso pode ser implementado de diversas maneiras.

Em Python, se criarmos uma classe para funcionar como nó de lista circular que tenha o método **iter** para iterar a lista toda, e uma forma de inserir novos elementos (por exemplo um .append que insira algo depois do nó atual), já temos uma lista circular

- com alguns outros métodos (**repr**, extend, **len**) esse objeto pode ser funcional o suficiente para ser usado em código de produção. Segue exemplo:

  No exemplo a seguir iremos criar a classe "nó" composta por 3 atributos: data, next e prev, onde data é o elemento que o nó guarda, next é o nó seguinte e prev é o nó anterior.

  Também irei implementar o método "append", que será o método de adição cujo primeiro passo é criar um novo nó e adicionar o nó principal a uma variável temporária. Assim, se o nó principal for nulo, então o novo nó é atribuído como principal e também para next e prev. Agora, se não for nulo, a lista é percorrida de maneira reversa, convertendo o nó temporário no anterior de temp e atribuindo ao seguinte o novo nó, ao anterior o nó temporário e ao seguinte a cabeça.

```
class Node():
    def __init__(self,data : str):
        self.data : str = data
        self.next : Node = None
        self.prev : Node = None

class double_linked():

    def __init__(self):
        self.head : Node = None
        self.end  : Node = None

    def append(self,data: str):

        new_node = Node(data)

        if self.end != None:
            new_node.prev = self.end
            self.end.next = new_node
            self.end = new_node
            return

        self.head = new_node
        self.end  = new_node

    def __str__(self):
        result = ""
        temp_node = self.head

        while temp_node != None:
            result += temp_node.data + ","
            temp_node = temp_node.next
        return result

    def find(self,data : str) -> Node:

        temp_node = self.head

        while temp_node != None:
            if temp_node.data == data:
                return temp_node

            temp_node = temp_node.next

        return None

    def print_reverse(self) -> str:

        result = ""
        temp_node = self.end
```

```
        while temp_node != None:
            result += temp_node.data + ","
            temp_node = temp_node.prev

        return result


if __name__ == "__main__":
    list_double = double_linked()
    list_double.append("Python")
    list_double.append("JavaScript")
    list_double.append("Java")

    print("List :  " , str(list_double))
    print("List reverse: ", list_double.print_reverse())
    print("Find ", list_double.find("Java").data)
```

6-7. Pilhas e filas

As pilhas e filas são estruturas de dados semelhantes, ambas baseadas em uma lista encadeada. As pilhas tem a política LIFO - *Last In First Out*, isto é, “o último que entra é o primeiro que sai”; enquantos as filas baseiam-se em FIFO - *First In First Out*, isto é, “o primeiro que entra é o primeiro que sai”. Ambas sem atributos e métodos semelhantes, porém tem as diferenças, ambas citadas a seguir. Começaremos com algo que é igual nos dois, a classe Nó, depois trataremos das especificidades.

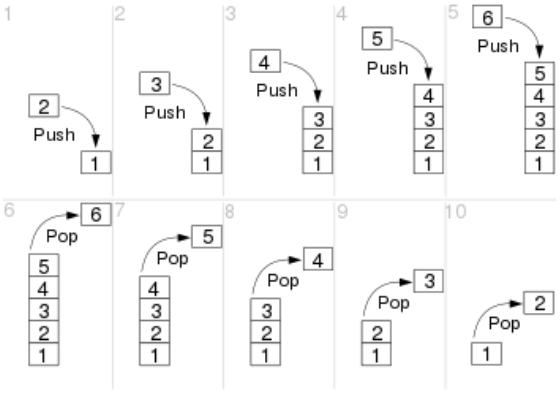
A classe nó

Como são listas encadeadas, deve-se, para se usar as pilhas e filas em Python, usar a classe Node(nó) que foi exposta na parte sobre listas encadeadas:

```
class Node:
    def __init__(self, data):
        self.data = data
        self.next = None
```

6. Pilhas

As pilhas são listas encadeadas de que se pode tirar só um nó por vez, e isso é feito utilizando-se uma política chamada LIFO - *Last In First Out*, isto é, “o último que entra é o primeiro que sai”. Nesse sentido, o elemento é retirado ou posto sempre num lugar chamado *topo*. Tais operações de inserção e remoção são chamadas respectivamente de *push* e *pop.* Vide imagem.



fonte: https://en.wikipedia.org/wiki/Stack_(abstract_data_type)

Estrutura da classe Pilha

Abaixo a estrutura da classe Pilha(*Stack*, em Inglês)

```
from node import Node

#IMPLEMENTAÇÕES
# 1. inserir na pilha
# 2. remover da pilha
# 3. observar o topo da pilha
class Stack:

    def __init__(self):
        self.top = None
        self._size = 0

    def __len__(self):
        """Retorna o tamanho da lista"""
        return self._size

    def push(self, elem):
            #insere um elemento na pilha
            node = Node(elem)
            node.next = self.top *4 #self.top estaria valendo 1, por exemplo
            self.top = node *4
            self._size = self._size + 1

    def pop(self):
            # remove o elemento do topo da pilha
            if self._size > 0:
                node = self.top
                self.top = self.top.next
                self._size = self._size - 1
                return node.data
            raise IndexError("The stack is empty")

    def peek(self):
        # retorna o topo sem remover
        if self._size > 0:
            return self.top.data
        raise IndexError("The stack is empty")

    def __repr__(self):
            r = ""
            pointer = self.top
            while(pointer):
                r = r + str(pointer.data) + "\n"
                pointer = pointer.next
            return r

    def __str__(self):
        return self.__repr__()
```

Atributos

A pilha armazena somente o *topo*, pois é nele que se realizarão as operações. Além disso, vamos armazenar o tamanho da pilha. Assim, poremos os dois atributos `self.first` e `self.__size` para armazenar e representar o topo e o tamanho, respectivamente, e predefinindo-os como `None` e `0` pois a pilha começa vazia.

```
def __init__(self):
  self.top = None
  self._size = 0
```

Métodos

Para acrescentar nós na pilha, o chamado `push`, faz-se:

```
#insere um elemento na pilha
def push(self, elem):
    node = Node(elem) #É o nó com que vamos estar trabalhando agora
    node.next = self.top
    #Vamos atribuir ao atributo next a posição do próximo
    #nó(o nó abaixo)
    self.top = node
    #no topo atual, vamos atribuir esse nó que acabamos
    #de construir
    self._size = self._size + 1
```

Para retirar nós na pilha, o chamado `pop`, faz-se:

```
 # remove o elemento do topo da pilha
def pop(self):
    if self._size > 0:
    #só removeremos se houver pelo menos um nó na pilha
        node = self.top
        #É uma variável temporária para podermos imprimir o nó que retiramos
        #da pilha
        self.top = self.top.next
        #O atributo self.top vai passar a apontar
        #não mais a certo nó na posição n, mas a um na posição n - 1(a posição
        #à qual apontava o atributo next do nó n)
        self._size = self._size - 1
        #Refaz-se a contagem de nós da pilha para conter um a menos.
        return node.data
        #Retorna a posição do topo da pilha
    raise IndexError("The stack is empty")
      #se não houver nenhum nó, vamos retornar que está vazio(não é possível
      #retirar)
```

Para retornar o topo da pilha, o chamado `peek`, faz-se:

```
# retorna o topo sem remover
def peek(self):
    if self._size > 0:
    #só retornaremos o topo se houver pelo menos um nó na pilha
        return self.top.data
        #retorna o valor do nó que está em self.top
    raise IndexError("The stack is empty")
      #se não houver nenhum nó, vamos retornar que está vazio(não há topo)
```

Para a representação dos dados:

Utiliza-se da mesma estrutura abaixo para representar os dados. O pointer recebe o primeiro. Se o primeiro for `None` , não vai entrar no `while` ; se não for, vai entrar no while. Lá, a representação `r`, vai ser incrementada a cada iteração, colocando-se o `"\n"` após cada valor posto. Coloca-se os valores começando no topo e indo até a base da pilha. Quando o `pointer` for `None`, a condição do while vai ser falsa. Assim, vai sair do bloco e retornar `r`.

```
def __repr__(self):
      r = ""
      pointer = self.top
      while(pointer):
          r = r + str(pointer.data) + "\n"
          pointer = pointer.next
      return r
```

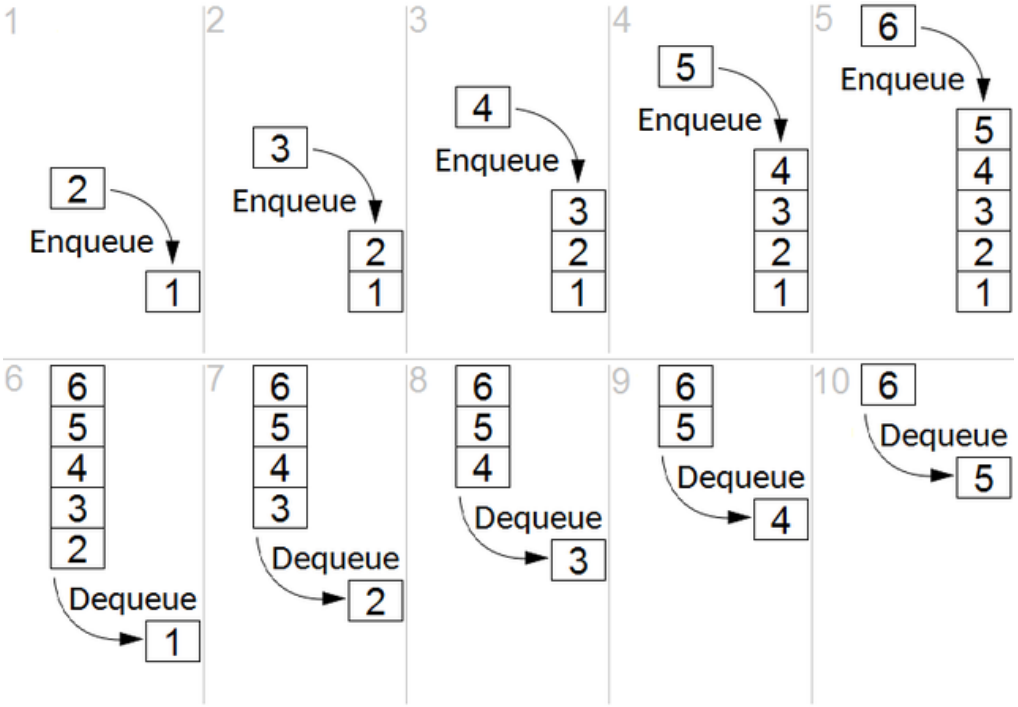
Para a impressão dos dados:

Basicamente, vamos imprimir a representação dos dados.

```
def __str__(self):
  return self.__repr__()
```

7. Filas

As filas também são listas encadeadas de que se pode tirar só um nó por vez, porém, ao contrário das pilhas, nesta utiliza-se uma política chamada FIFO - *First In First Out*, isto é, “o primeiro que entra é o primeiro que sai” Nesse sentido, há tanto o *primeiro da fila* (ou *topo*, como nas pilhas) de onde se retirará; e um elemento *último da fila*, que será sempre substituído por um novo elemento quando este chega. Como nas pilhas, as operações de inserção e remoção são chamadas respectivamente de *push*(ora chamado de *enqueue*) e *pop*(ora chamado de *dequeue*). Vide imagem.



fonte: https://pt.wikipedia.org/wiki/FIFO

Esta também vai precisar da classe Node, citada anteriormente.

Estrutura da classe Fila

```
from node import Node

class Queue:
    def __init__(self):
        self.first = None
        self.last = None
        self._size = 0

    # inserir na fila
    def push(self, elem):
        """Insere um elemento na fila"""
        node = Node(elem)
        if self.last is None:
            self.last = node
        else:
            self.last.next = node
            self.last = node

        if self.first is None:
            self.first = node

        self._size = self._size + 1

    # remover da fila
    def pop(self):
        """Remove o elemento do topo da pilha"""
        if self._size > 0:
            elem = self.first.data
            self.first = self.first.next
            # faltou tratar este caso no vídeo #
            if self.first is None:
                self.last = None
            ####################################
            self._size = self._size - 1
            return elem
        raise IndexError("The queue is empty")

    # observar o primeiro da fila
    def peek(self):
        """Retorna o topo sem remover"""
        if self._size > 0:
            elem = self.first.data
            return elem
        raise IndexError("The queue is empty")


    def __len__(self):
        """Retorna o tamanho da lista"""
        return self._size


    def __repr__(self):
        if self._size > 0:
            r = ""
            pointer = self.first
            while(pointer):
                r = r + str(pointer.data) + " "
                pointer = pointer.next
            return r
        return "Empty Queue"
```

```
    def __str__(self):
        return self.__repr__()
```

Atributos

Como citado, a fila armazena tanto o primeiro como o último elemento. Assim, poremos os dois atributos `self.first` e `self.last` para armazenar e representar o primeiro e o último elementos, respectivamente, e predefinindo-os como `None`, pois a fila começa vazia. Além disso, armazenaremos o tamanho da fila, que tem tamanho inicial de valor `0`.

```
 def __init__(self):
    self.first = None
    self.last = None
    self._size = 0
```

Métodos

Para acrescentar nós na fila, o chamado `push`, faz-se:

```
def push(self, elem):
        """Insere um elemento na fila"""
        node = Node(elem)

        #Faremos esse primeiro tratamento de caso porque se self.last == None,
        #então, se fizéssemos self.last.next = node, estaríamos fazendo algo
        #como None.next = node, o que seria um erro.
        if self.last is None:
            #Vamos atribuir à última/primeira posição o elemento recebido
            self.last = node
        else:
            #Se não há mais o valor None, então é possível fazer self.last.next = node
            self.last.next = node
            self.last = node

        #se o self.first é None, então devemos atribuir esse elemento que chegou
        # como o primeiro, é o primeiro "de todos".
        if self.first is None:
            self.first = node

        self._size = self._size + 1
```

Para retirar nós na fila, o chamado `pop`, faz-se:

```
def pop(self):
        """Insere um elemento na fila"""
      #tirar o primeiro
        #armazenando para imprimir o nó que saiu
        node_data = self.first.data
        #A operação só deverá ser realizada se houver um elemento para ser retirado
        if self._size > 0
          #O primeiro valor agora recebe o segundo
          self.first = self.first.next
          #diminui-se o tamanho da fila
          self._size = self._size - 1
          #retorna para mostrar o elemento que saiu da fila
```

```
            return node_data
        #Se não houver elemento para ser tirado, vamos retornar um erro
        raise IndexError("The queue is empty")
```

Para retornar o primeiro elemento da fila(topo), o chamado `peek`, faz-se:

```
def peek(self):
        #A operação só deverá ser realizada se houver um elemento para ser retirado
        if self._size > 0:
            elem = self.first.data
            return elem
        #Se não houver elemento para ser retornado, vamos retornar um erro
        raise IndexError("The queue is empty")
```

Para a representação dos dados:

Utiliza-se da mesma estrutura abaixo para representar os dados. O pointer recebe o primeiro. Se o primeiro for `None` , não vai entrar no `while`; se não for, vai entrar no while. Lá, a representação `r`, vai ser incrementada a cada iteração. Coloca-se os valores começando no topo e indo até a base da pilha. Quando o `pointer` for `None`, a condição do while vai ser falsa. Assim, vai sair do bloco e retornar `r`. A diferença para a pilha é que põe um `" "` ao final, em vez de um `"\n"`.

```
def __repr__(self):
      r = ""
      pointer = self.top
      while(pointer):
          r = r + str(pointer.data) + "\n"
          pointer = pointer.next
      return r
```

Para a impressão dos dados:

Vamos imprimir a representação dos dados.

```
def __str__(self):
  return self.__repr__()
```

8. Estrutura de árvores

Def.: É uma estrutura de dados hierárquica formada por um conjunto de nodos(vértices) que são conectados de forma específica pelo conjunto de arestas. O nodo no nível 0, chamado de raiz da árvore, está no topo hierárquico. A raiz é conectada a outros nodos, no nível 1, que estão conectados a outros nodos, no nível 2, e assim por diante.

Árvores binárias

As árvores binárias podem ter, em cada nodo, no máximo 2 filhos e no mínimo nenhum. Cada nodo de uma árvore binária possui 1 valor chave e 2 apontadores(representam as ligações/arestas).

Funções para a Implementação de uma árvore binária(inicialização e representação)

```
class nodoArvore:
    def __init__(self, chave= None, esqueda=None, direita=None):
        # Atributos da classe são acessados atráves do self(representa a instância da classe)
        # É criado um nó com uma chave, um ponteiro esquerdo e outro direito
        # para os respectivos filhos
        self.chave = chave
        self.esquerda = esqueda
        self.direita = direita

    def __repr__(self):  # defini um padrão na representação da árvore
        #Apresenta chave dos filhos a esquerda, chave do nó atual, chave dos filhos a direita
        return '%s <- %s -> %s' % (self.esquerda and self.esquerda.chave, self.chave,
                                   self.direita and self.direita.chave)
```

Podemos mudar a forma que será representado no terminal, demonstrando de forma mais organizada:

```
class NodoArvore:
    def __init__(self, chave=None, esquerda=None, direita=None):
        self.chave = chave
        self.esquerda = esquerda
        self.direita = direita

    def __repr__(self, nivel=0, prefixo='Raiz: '):
        ret = ' ' * nivel * 4 + prefixo + repr(self.chave) + '\n'
        if self.esquerda:
            ret += self.esquerda.__repr__(nivel + 1, 'Esquerda: ')
        if self.direita:
            ret += self.direita.__repr__(nivel + 1, 'Direita: ')
        return ret

# Criação da árvore
raiz = NodoArvore(8)
raiz.esquerda = NodoArvore(4)
raiz.direita = NodoArvore(3)

# Adicionando filhos ao nodo de valor 4
raiz.esquerda.esquerda = NodoArvore(2)  # Adiciona um filho à esquerda com valor 2
raiz.esquerda.direita = NodoArvore(0)   # Adiciona um filho à direita com valor 0

# Adicionando filhos ao nodo de valor 3
raiz.direita.esquerda = NodoArvore(5)   # Adiciona um filho à esquerda com valor 5
raiz.direita.direita = NodoArvore(7)    # Adiciona um filho à direita com valor 7

print("Árvore: \n", raiz)
```

Saída do terminal:

```
Árvore:
 Raiz: 8
    Esquerda: 4
        Esquerda: 2
        Direita: 0
    Direita: 3
        Esquerda: 5
        Direita: 7
```

Árvores binárias de pesquisa (BSTs)

I. Definição:

Se X for um nodo e Y um nodo na “sub-árvore” esquerda de X , então : Y.chave ≤ X.chave

Se X for um nodo e Y um nodo na “sub-árvore” direita de X, então: Y.chave ≥ X.chave

Ou seja, as árvores binárias de pesquisa são árvores que seguem as seguintes propriedades:

1. Dado um nodo qualquer da árvore, todos os nodos à **esquerda** dele são **menores** ou iguais a ele.
2. Dado um nodo qualquer da árvore, todos os nodos à **direita** dele são **maiores** ou iguais a ele.

Em uma árvore binária de pesquisa temos diferentes formas de visitar todos os nodos de uma árvore, chamado de caminhamentos em árvores. Dentre eles temos pré-ordem, em ordem, e pós-ordem, no qual a única diferença é a ordem em que os nodos serão impressos.

1. caminhamento em ordem - visitamos recursivamente o nodo da esquerda, o nodo corrente, e recursivamente o nodo da direita. Dessa forma estaremos de fato visitando os nodos em **ordem crescente de chaves**.

Função de caminhamento **em ordem**:

```
class NodoArvore:
    def __init__(self, chave=None, esquerda=None, direita=None):
        self.chave = chave
        self.esquerda = esquerda
        self.direita = direita

def em_ordem(raiz):
    if not raiz:
        return

    # Visita filho da esquerda.
    em_ordem(raiz.esquerda)

    # Visita nodo corrente.
    print(raiz.chave),

    # Visita filho da direita.
    em_ordem(raiz.direita)
```

Nos caminhamento **em pré-ordem e pós-ordem muda apenas a posição:**

```
#pré-ordem
def pre_ordem(raiz):
    if not raiz:
        return
    # Visita nodo corrente.
    print(raiz.chave)

    # Visita filho da esquerda.
    em_ordem(raiz.esquerda)

    # Visita filho da direita.
    em_ordem(raiz.direita)
```

```
#pós-ordem
def pos_ordem(raiz):
    if not raiz:
        return
    # Visita filho da esquerda.
    em_ordem(raiz.esquerda)

    # Visita filho da direita.
    em_ordem(raiz.direita)

    # Visita nodo corrente.
    print(raiz.chave)
```

II. Inserção em Árvores Binárias de Pesquisa

Agora iremos implementar uma árvore binária de **forma implícita à posição dos nodos**

Vamos criar as regras de BSTs dentro do código para cada nodo inserido:

```
# Insere um nodo na árvore
def insere(raiz, nodo):
    # Nodo deve ser inserido na raiz.
    # árvore vazia -> nodo inserido vira raiz
    if raiz is None:
        raiz = nodo

    # Nodo deve ser inserido na subárvore direita.
    elif raiz.chave < nodo.chave:
        if raiz.direita is None:
            raiz.direita = nodo
        else:
            insere(raiz.direita, nodo)

    # Nodo deve ser inserido na subárvore esquerda.
    else:
        if raiz.esquerda is None:
            raiz.esquerda = nodo
        else:
            insere(raiz.esquerda, nodo)
```

Podemos inserir essas regras dentro da árvore:

Saída do terminal:

```
10 20 30 40 50 60 70
```

III. Busca em Árvores Binárias de Pesquisa

```
class NodoArvore:
    def __init__(self, chave=None, esquerda=None, direita=None):
        self.chave = chave
        self.esquerda = esquerda
        self.direita = direita

def insere(raiz, nodo):
    if raiz is None:
        raiz = nodo
```

```
    elif raiz.chave < nodo.chave:
        if raiz.direita is None:
            raiz.direita = nodo
        else:
            insere(raiz.direita, nodo)

    else:
        if raiz.esquerda is None:
            raiz.esquerda = nodo
        else:
            insere(raiz.esquerda, nodo)

def em_ordem(raiz):
    if not raiz:
        return

    em_ordem(raiz.esquerda)
    print(raiz.chave,end=' ')
    em_ordem(raiz.direita)

raiz = NodoArvore(40)
chaves = [20, 60, 50, 70, 10, 30]
for chave in chaves:
  nodo = NodoArvore(chave)
  insere(raiz, nodo)

em_ordem(raiz)
```

A ideia da busca em uma árvore binária de busca é semelhante à busca binária em uma lista ordenada. Você começa no nó raiz e compara o elemento alvo com o elemento no nó atual. Com base na comparação, você decide se deve continuar a busca na sub-árvore esquerda, na sub-árvore direita ou se encontrou o elemento desejado.

Possui 3 casos possíveis:

1. A chave procurada está na raiz da árvore. Nesse caso, simplesmente retornamos a raiz da árvore como resultado da busca.
2. A chave procurada é menor que a chave do nodo raiz. Nesse caso, precisamos procurar pela chave somente na sub-árvore esquerda.
3. A chave procurada é maior que a chave do nodo raiz. Nesse caso, precisamos procurar pela chave somente na sub-árvore direita.

Implementando a ideia em código:

```
#Procura por uma chave em uma árvore binária de pesquisa.
def busca(raiz, chave):
    # Trata o caso em que a chave procurada não está presente.
    if raiz is None:
        return None

    # A chave procurada está na raiz da árvore.
    if raiz.chave == chave:
        return raiz

    # A chave procurada é maior que a da raiz.
    if raiz.chave < chave:
        return busca(raiz.direita, chave)
```

```
    # A chave procurada é menor que a da raiz.
    return busca(raiz.esquerda, chave)
```

Aplicando na estrutura da árvore binária de pesquisa (BST):

```
class NodoArvore:
    def __init__(self, chave=None, esquerda=None, direita=None):
        self.chave = chave
        self.esquerda = esquerda
        self.direita = direita

def insere(raiz, nodo):
    if raiz is None:
        raiz = nodo

    elif raiz.chave < nodo.chave:
        if raiz.direita is None:
            raiz.direita = nodo
        else:
            insere(raiz.direita, nodo)

    else:
        if raiz.esquerda is None:
            raiz.esquerda = nodo
        else:
            insere(raiz.esquerda, nodo)

def busca(raiz, chave):
    if raiz is None:
        return None

    if raiz.chave == chave:
        return raiz

    if raiz.chave < chave:
        return busca(raiz.direita, chave)

    return busca(raiz.esquerda, chave)

# Cria uma árvore binária de pesquisa.
raiz = NodoArvore(40)
for chave in [20, 60, 50, 70, 10, 30]:
    nodo = NodoArvore(chave)
    insere(raiz, nodo)

# Procura por valores na árvore.
for chave in [-50, 10, 30, 70, 100]:
    resultado = busca(raiz, chave)
    if resultado:
        print("Busca pela chave {}: Sucesso!".format(chave))
    else:
        print("Busca pela chave {}: Falha!".format(chave))
```

Saída no terminal:

```
Busca pela chave -50: Falha!
Busca pela chave 10: Sucesso!
Busca pela chave 30: Sucesso!
Busca pela chave 70: Sucesso!
Busca pela chave 100: Falha!
```

Árvore balanceada

Definição:

Uma árvore binária é considerada balanceada quando a diferença de profundidade entre duas folhas quaisquer é no máximo 1, sendo a profundidade o número de níveis até o nodo mais distante.

Após criarmos a estrutura da árvore, basta implementarmos estas funções nela:

```
def altura(nodo):
    if nodo is None:
        return 0
    return 1 + max(altura(nodo.esquerda), altura(nodo.direita))

def is_balanced(nodo):
    if nodo is None:
        return True

    esquerda_altura = altura(nodo.esquerda)
    direita_altura = altura(nodo.direita)

    # Verifica se a diferença nas alturas é -1, 0 ou 1
    if abs(esquerda_altura - direita_altura) <= 1 and is_balanced(nodo.esquerda) and is_balanced(nodo.direita):
        return True

    return False
```

---

9. BUSCA SEQUÊNCIAL

A busca sequencial é um método simples de encontrar um elemento em uma lista, onde cada elemento é verificado sequencialmente(um a um) até que o elemento desejado seja encontrado ou todo o conjunto de dados seja percorrido.

Basta implementarmos a seguinte função:

```
def busca_sequencial(lista, alvo):
    for i, elemento in enumerate(lista):
        if elemento == alvo:
            return i  # Retorna o índice do elemento alvo se encontrado

    return -1  # Retorna -1 se o elemento alvo não for encontrado na lista
```

---

10. BUSCA BINÁRIA

A busca binária é um algoritmo eficiente para encontrar um elemento em uma lista ordenada. A ideia básica é dividir repetidamente a lista pela metade e comparar o elemento alvo com o elemento no meio da lista. Se o elemento alvo for igual ao elemento do meio, a busca é bem-sucedida. Se o elemento alvo for menor, a busca é então realizada na metade inferior da lista; se for maior, a busca é realizada na metade superior.

Basta implementarmos a seguinte função:

```
def busca_binaria(lista, alvo):
    # Inicializa os índices para a busca
    inicio, fim = 0, len(lista) - 1

    # Executa a busca enquanto o início não ultrapassa o fim
    while inicio <= fim:
        meio = (inicio + fim) // 2  # Calcula o índice do elemento do meio

        # Verifica se o elemento alvo está no meio da lista
        if lista[meio] == alvo:
            return meio  # Retorna o índice do elemento alvo

        # Se o elemento alvo for menor, busca na metade inferior da lista
        elif lista[meio] > alvo:
            fim = meio - 1

        # Se o elemento alvo for maior, busca na metade superior da lista
        else:
            inicio = meio + 1

    return -1  # Retorna -1 se o elemento alvo não for encontrado na listas
```