

TRYBE

Modulo IV – Ciência da Computação

Bloco 39: Estrutura de Dados: Pilhas, Filas e Listas

1) Estrutura de dados III - Pilhas

Pilhas usadas para:

- **executar aplicações** (interpretador do Python encontra uma chamada a uma função, salva o estado da função atual, a adiciona na pilha de execução ou *call stack*. Quando a função chamada termina a execução, o interpretador volta na pilha e lê a função que esta no topo para continuar)
- **resolver expressões matemáticas, respeitando a ordem dos operadores** (via estrutura *pós fixa*)

Pilha

Estrutura do tipo **LIFO (Last In First Out)**.

Pode ser criada a partir de *listas ou arrays*.

Tem um *limite* na quantidade de valores (para evitar *stack overflow*).

// Imagem boa para entender conceito: pilha de pratos.

Operações mais comuns

push adiciona um item no **topo** da pilha (podemos adicionar novos valores somente no topo dela);
pop para **ler** valores do topo da pilha, **removendo** esse item;
peek para **ler** valores do topo da pilha, apenas lendo esse item;
clear para **limpar** todos os elementos da pilha.

Utilizando pilhas

→ **Code**

```
class Stack():
    def __init__(self):
        self._data = list()

    def size(self):
        return len(self._data)

    def is_empty(self):
        return not bool(self.size())

    def push(self, value):
        self._data.append(value)

    def pop(self):
        if self.is_empty():
            return None

        # -1 se refere ao último objeto da pilha.
        # Ou seja, o valor do topo da pilha
        value = self._data[-1]
```

```

del self._data[-1]
return value

def peek(self):
    if self.is_empty():
        return None
    value = self._data[-1]
    return value

def clear(self):
    self._data.clear()

# método __str__ que fará a impressão de todos os elementos que estão empilhados
def __str__(self):
    str_items = ""
    for i in range(self.size()):
        value = self._data[i]
        str_items += str(value)
        if i + 1 < self.size():
            str_items += ", "

    return "Stack(" + str_items + ")"

```

→ Quando usar

- Para resolver diversos problemas em linguagens de programação: controlar o estado das chamadas de funções, resolver expressões matemáticas e lógicas...;
- quando precisarmos representar o comportamento de uma pilha.

→ Utilização de pilhas no controle da chamada de funções

O Python mantém uma pilha com quais funções devem ser executadas após a execução de uma. Função `traceback.print_stack(file=sys.stdout)` para conseguir ver os itens presentes na call stack.

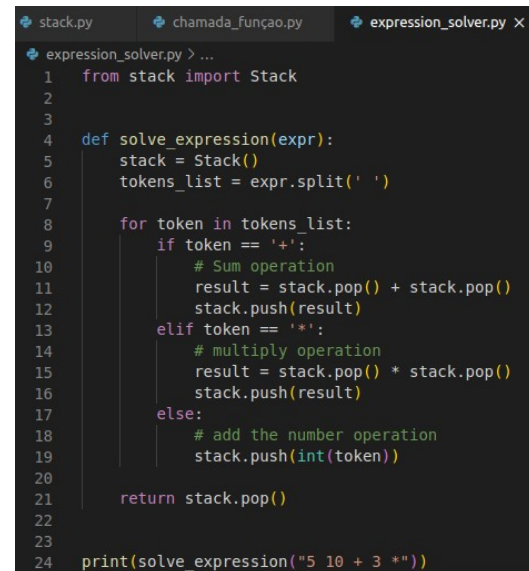
→ Utilização de pilhas na resolução de expressões

Tipos de representação de expressões:

infixa $((A + B) * C)$, naturalmente usa pilhas se acumulando ao calcular

pós fixa $(A B + C *)$

→ Implementar uma função que resolve expressões pós fixas →



```

1  from stack import Stack
2
3
4  def solve_expression(expr):
5      stack = Stack()
6      tokens_list = expr.split(' ')
7
8      for token in tokens_list:
9          if token == '+':
10             # Sum operation
11             result = stack.pop() + stack.pop()
12             stack.push(result)
13         elif token == '*':
14             # multiply operation
15             result = stack.pop() * stack.pop()
16             stack.push(result)
17         else:
18             # add the number operation
19             stack.push(int(token))
20
21     return stack.pop()
22
23
24     print(solve_expression("5 10 + 3 *"))

```

Dicas diversas

- Visualizar pilhas no [VisualGo](#) (e outros tipos de algoritmos)
 - Praticar no [leetcode](#) .
-

2) Estrutura de dados III - Deque

Exemplo uma aplicação de deque:

- filas de clientes em espera;
- armazenar o histórico de um navegador da web;
- qualquer coleta de tempo em que o tempo de inserção seja importante.

TAD deque

Um dos tipos abstratos de dados (**TAD**) **mais flexíveis** em meio as estruturas de dados.
Variação da fila: a fila de dois fins ou **Double Ended QUEUE**.

Em uma fila, os elementos seguem a **ordem FIFO (First In First Out)**.

Mas também tem possibilidade de:

- **deque de saída restrita**: remoção restrita em um dos lados, inserção segue em ambas extremidades;
- **deque de entrada restrita**: entrada restrita a apenas um lado, saída pode ser realizada em ambas extremidades.

//Exemplo: fila para pessoas prioritárias

Operações mais comuns

PushBack - adicionar um item no fim da deque;

PushFront - adicionar um item no início da deque, mantendo a ordenação preexistente;

Para remover e visualizar itens nas respectivas extremidades:

PopBack

PopFront

peekFront

peekBack

Restrições de overflow (inserir em deque cheia) e **underflow** (remover em deque vazia).

Quando usar

Deque é preferível à outros TADs como a lista, nos casos em que a maioria das operações serão de inserção e remoção nas duas extremidades da estrutura, pois a deque fornece uma **complexidade de tempo $O(1)$** , para push e pop, em comparação à lista que fornece complexidade de tempo $O(n)$.

Deque com Python

class Deque:

FIRST_ELEMENT = 0

def __init__(self):

self._data = list()

```

def __len__(self):
    return len(self._data)

def __str__(self):
    return "Deque(" + ", ".join(map(lambda x: str(x), self._data)) + ")"

def push_front(self, value):
    self._data.insert(self.FIRST_ELEMENT, value)

def push_back(self, value):
    self._data.append(value)

def pop_front(self):
    if self._data:
        return self._data.pop(self.FIRST_ELEMENT)
    return None

def pop_back(self):
    if self._data:
        return self._data.pop()
    return None

def peek_front(self):
    if self._data:
        return self._data[self.FIRST_ELEMENT]
    return None

def peek_back(self):
    if self._data:
        return self._data[len(self) - 1]
    return None

```

Dicas diversas

- Praticar na [leetcode](https://leetcode.com/);
- deque em python do [geekforgeeks](https://www.geekforgeeks.org/python-deque/);

```

1  # TAD (Interface) de Filas
2  # .push (ENQUEUE)
3  # .pop (DEQUEUE)
4  # .length (opcional)
5  # .peek (opcional)

```

```

6  0 := tamanho
7  1 := inicio
8
9  0 1 2 3 4 5 6 7 8 9
10 [2 2 0 0 0 0 0 0 0 0]
11
12 # .push (complexidade constante)
13 # 1 coloca o item no espaço zero + tamanho + inicio + 2
14 # 2 atualiza o tamanho (tamanho++)
15
16 # .pop
17 # 1 cria uma variável que aponta para (zero + inicio + 2)
18 # 2 remove o item (zero + inicio + 2)
19 # 3 atualiza o tamanho (tamanho--)
20 # 4 atualiza o inicio (inicio++)

```

3) Estrutura de Dados III - Nó & Listas ligadas

Vantagens das **listas ligadas (LinkedLists)**:

- estrutura de dados **dinâmica**;
- **redimensionável** em tempo de execução;
- **facilidade** e eficiência das operações de inserção e exclusão;
- podem **fazer outras estruturas** de dados como pilha, lista;
- sendo uma estrutura de dados **linear**, pode ajudar entender outras como árvores

Exemplos de aplicação: implementações de gráficos, alocação de memória dinâmica.

Node (nó)

Componente mais importante da estrutura.

Composto por um **valor de qualquer tipo** e um **indicador/ponteiro para o próximo Node**.

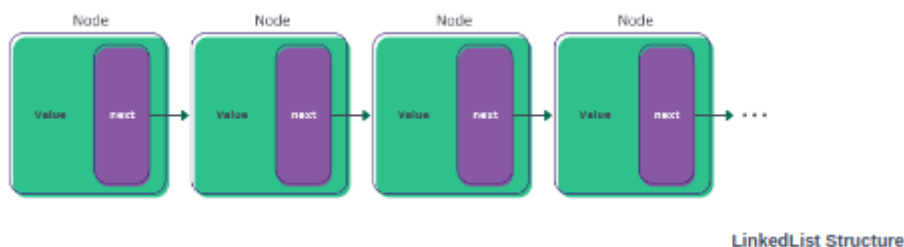
LinkedList (lista ligada)

Coleção de Nodes.

Sequência **finita** de elementos, que podem ser **repetidos** e não são armazenados em memória contígua (como array): a própria **estrutura é responsável por indicar a posição** dos elementos.

Diferença com pilhas e deque: em uma lista encadeada é possível **acessar qualquer elemento**, sem exceção.

Diferença com lista: sequencialidade.



Operações mais comuns

Complexidade assintótica por operação	
Operação	Complexidade
insert_first	$O(1)$
insert_last	$O(1)$
insert_at	$O(n)$
remove_first	$O(1)$
remove_last	$O(1)$
remove_at	$O(n)$
clear	$O(n)$
get_element_at	$O(n)$
is_empty	$O(1)$

#

insert_first
insert_last
insert_at
remove_first
remove_last
remove_at
clear
get_element_at
is_empty

Implementação de um Node

```
class Node:
    def __init__(self, value):
        self.value = value
        self.next = None
        # valor default None

    def __str__(self):
        return f"Node(value={self.value}, next={self.next})"
```

Implementação de uma LinkedList

```
from node import Node
```

```
class LinkedList:

    def __init__(self):
        self.head_value = None
        self.__length = 0

    def __str__(self):
        return f"LinkedList(len={self.__length}, value={self.head_value})"

    def __len__(self):
        return self.__length

    def insert_first(self, value):
        first_value = Node(value)
        first_value.next = self.head_value
        self.head_value = first_value
        self.__length += 1


    def insert_last(self, value):
        last_value = Node(value)
        current_value = self.head_value

        if self.is_empty():
            return self.insert_first(value)

        while current_value.next:
            current_value = current_value.next
        current_value.next = last_value
        self.__length += 1

    def insert_at(self, value, position):
        if position < 1:
            return self.insert_first(value)
        if position >= len(self):
            return self.insert_last(value)
        current_value = self.head_value
        while position > 1:
            current_value = current_value.next
            position -= 1
        next_value = Node(value)
        next_value.next = current_value.next
        current_value.next = next_value
        self.__length += 1

    def is_empty(self):
        return self.__length == 0
```



```
33 > ... def insert_first(self, value): ...
38
39 > ... def insert_last(self, value): ...
51
52 > ... def insert_at(self, value, position): ...
65
66 > ... def remove_first(self): ...
73
74 > ... def remove_last(self): ...
88
89 > ... def remove_at(self, position): ...
107
108 > ... def get_element_at(self, position): ...
118
119 > ... def is_empty(self): ...
```

```

def remove_first(self):
    value_to_be_removed = self.head_value
    if value_to_be_removed:
        self.head_value = self.head_value.next
        value_to_be_removed.next = None
        self.__length -= 1
    return value_to_be_removed

def remove_last(self):
    if len(self) <= 1:
        return self.remove_first()

    previous_to_be_removed = self.head_value
    value_to_be_removed = previous_to_be_removed.next

    while value_to_be_removed.next:
        previous_to_be_removed = previous_to_be_removed.next
        value_to_be_removed = previous_to_be_removed.next

    previous_to_be_removed.next = None
    self.__length -= 1
    return value_to_be_removed

def remove_at(self, position):
    if position < 1:
        return self.remove_first()
    if position >= len(self):
        return self.remove_last()

    previous_to_be_removed = self.head_value

    while position > 1:
        previous_to_be_removed = previous_to_be_removed.next
        position -= 1

    value_to_be_removed = previous_to_be_removed.next
    previous_to_be_removed.next = value_to_be_removed.next
    value_to_be_removed.next = None
    self.__length -= 1

    return value_to_be_removed

def get_element_at(self, position):
    value_returned = None
    value_to_be_returned = self.head_value
    if value_to_be_returned:
        while position > 0 and value_to_be_returned.next:
            value_to_be_returned = value_to_be_returned.next
            position -= 1
        if value_to_be_returned:
            value_returned = Node(value_to_be_returned.value)
    return value_returned

def is_empty(self):
    return not self.__length
# nos informa se a estrutura está vazia, já que not 0 == True .

```

Dicas diversas

→ Ver como funciona as linked lists no [visualgo](#)

→ Index para de existir, o que tem é uma origem:

$A(b) \rightarrow B(c) \rightarrow C(\text{null})$

$A(c) \rightarrow C(\text{null})$

→ Artigos [medium](#), [realpython](#) e [geeksforgeeks](#) como recursos adicionais sobre linked lists

→ TailedList a partir de LinkedList

```
tailed_list.py > TailedList > insert_first
1  from linked_list import LinkedList
2
3  class TailedList(LinkedList):
4      def __init__(self):
5          super().__init__()
6          self.tail = None
7
8      def insert_first(self, value):
9          if self.is_empty():
10             super().insert_first(value)
11             self.tail = self.head
12          else:
13             super().insert_first(value)
14
15
16
17  lista = TailedList()
18  lista.insert_first("A")
```

para melhorar condicional:

```
def insert_first(self, value):
    estava_vazia = self.is_empty()
    super().insert_first(value)
    if estava_vazia:
        self.tail = self.head
```

4) Estrutura de dados III – Listas duplamente ligadas

Exemplos de aplicação: sistemas de navegação, onde a navegação frontal e traseira é necessária – caso do botão voltar e avançar.

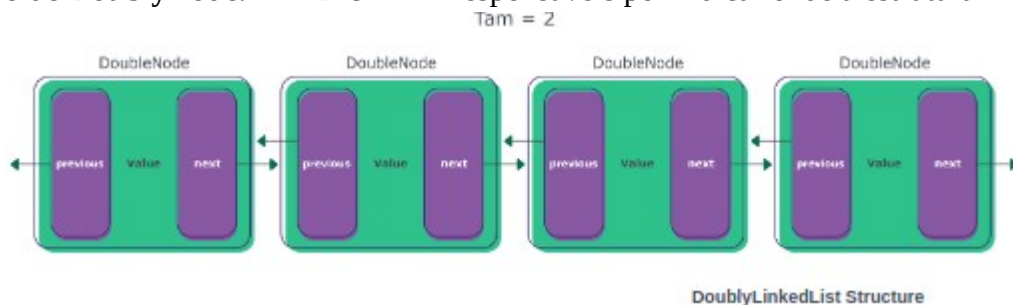
DoublyNode

DoublyNode possui também um **outro ponteiro para indicar o elemento anterior**.

Para termos acesso a um refetido elemento, devemos **acessar primeiramente seu container**.

DoublyLinkedList

Coleção de DoublyNode. *HEAD & TAIL* responsáveis por indicar onde a estrutura inicia e finaliza.



Operações mais comuns

As mesmas que a LinkedList (ver acima).

Vantagem da DoubleLinkedList: sua capacidade de **otimização** em operações nas **extremidades**. Podemos obter complexidade $O(1)$ em vez de $O(n)$ para *insert_last* e *remove_last*, já que sentinelas guardam essas posições.

Implementação de um DoublyNode

```
class DoublyNode:
    def __init__(self, value=None):
        self.value = value
        self.next = None
        self.previous = None

    def __str__(self):
        return f"Node(value={self.value} next={self.next})"
```

Implementação de uma DoublyLinkedList

```
from doubly_node import DoublyNode
```

```
class DoublyLinkedList:
```

```
    # Para criar nossa estrutura, devemos usar o DoublyNode feito anteriormente
```

```
    def __init__(self):
        self.head = DoublyNode("HEAD")
        self.tail = DoublyNode("TAIL")
        self.head.next = self.tail
        self.tail.previous = self.head
        self.__length = 0
```

```
    def __str__(self):
        return f"DoublyLinkedList(len={self.__length} value={self.head})"
```

```
    def __len__(self):
        return self.__length
```

```
    # Para inserir no início, devemos informar que o elemento que estamos inserindo seja o próximo next da cabeça, HEAD
```

```
    def insert_first(self, value):
        node_value = DoublyNode(value)
        node_value.next = self.head.next
        node_value.previous = self.head
        node_value.next.previous = node_value
        self.head.next = node_value
        self.__length += 1
```

```
    # Para inserir no final, devemos informar que o elemento que estamos inserindo seja o anterior previous da cauda TAIL
```

```
    def insert_last(self, value):
        node_value = DoublyNode(value)
        node_value.previous = self.tail.previous
        node_value.next = self.tail
        node_value.previous.next = node_value
        self.tail.previous = node_value
        self.__length += 1
```

```
    # Para inserir em qualquer posição, devemos informar que o elemento que estamos inserindo seja adicionado na posição desejada em nossa estrutura
```

```
    def insert_at(self, value, position):
        if position < 1:
            return self.insert_first(value)
        if position >= len(self):
            return self.insert_last(value)
        new_node = DoublyNode(value)
```

```
position_node = self.__get_node_at(position)
new_node.next = position_node
new_node.previous = position_node.previous
new_node.previous.next = new_node
position_node.previous = new_node
self.__length += 1
```

Para remover no início, devemos fazer com que a nossa estrutura remova o next da HEAD

```
def remove_first(self):
    value_to_be_removed = None
    if not self.is_empty():
        value_to_be_removed = self.head.next
        element_later_than_removed = value_to_be_removed.next
        self.head.next = element_later_than_removed
        element_later_than_removed.previous = self.head
        value_to_be_removed.reset()
        self.__length -= 1
    return value_to_be_removed
```

Para remover no final, devemos fazer com que a nossa estrutura remova o previous da TAIL

```
def remove_last(self):
    value_to_be_removed = None
    if not self.is_empty():
        value_to_be_removed = self.tail.previous
        element_later_than_removed = value_to_be_removed.previous
        self.tail.previous = element_later_than_removed
        element_later_than_removed.next = self.tail
        value_to_be_removed.reset()
        self.__length -= 1
    return value_to_be_removed
```

Para remover em qualquer posição, devemos informar a posição do elemento que desejamos a remoção de nossa estrutura

```
def remove_at(self, position):
    if position < 1:
        return self.remove_first()
    if position >= len(self):
        return self.remove_last()
    value_to_be_removed = None
    if not self.is_empty():
        value_to_be_removed = self.__get_node_at(position)

        previous_to_be_removed = value_to_be_removed.previous
        next_to_be_removed = value_to_be_removed.next

        previous_to_be_removed.next = next_to_be_removed
        next_to_be_removed.previous = previous_to_be_removed

        value_to_be_removed.reset()
        self.__length -= 1

    return value_to_be_removed
```

Para obter elemento em qualquer posição, devemos informar a posição do elemento que desejamos visualizar o conteúdo, esta função deve retornar uma cópia do node existente em nossa estrutura

```
def get_element_at(self, position):
    value_returned = None
    value_to_be_returned = self.__get_node_at(position)
    if value_to_be_returned:
        value_returned = DoublyNode(value_to_be_returned.value)
    return value_returned
```

```
def __get_node_at(self, position):
    value_to_be_returned = None
    if self.head.next != self.tail:
        value_to_be_returned = self.head.next
        while position > 0:
            value_to_be_returned = value_to_be_returned.next
            position -= 1
    return value_to_be_returned
```

```
# Devemos informar se a estrutura está vazia
def is_empty(self):
    return not self.__length
```

Dicas diversas

→ Fazer **lista circular**: passando do último nó, cai no primeiro nó.

```
# Head -> $ -> A -> B -> C -> D -> E -> A
# Head <-$ <- A <- B <- C <- D <- E <- Tail
```

Usar o DoublyLinkedList com a ideia de adaptar o tail do último e o head do primeiro.
Criar function de apoio para remendar.
Code inteiro na branch 39.4 dos [codes de aulas ao vivo](#).

→ *pass*

```
class FazNada:
    pass
```

5) Projeto – TING (Trybe Is Not Google)

Dicas diversas do projeto:

→ Escrever erro com [sys.srder](#) ;
[write](#) ou [print](#)

→ Dividir linhas ;
[spitlines](#) ou [rstrip](#)

→ Retornar resposta desejada com [sys.stdout](#) ;

→ Contar com [enumerate](#) .

That's all folks!! Obrigada, Trybe & T5.

