

October 9, 2024

1 Aula 07 - Exercício prático Variações de listas

Aluno: Gian Franco Joel Condori Luna

1.1 Resolver os exercícios no Beecrowd Judge: <https://judge.beecrowd.com/>

- 1242 Ácido Ribonucleico Alienígena
- 1083 LEXSIM - Avaliador Lexico e Sintático
 - Anexar no classroom um pdf com a análise de complexidade dos algoritmos que vc desenvolveu.

1.2 Solução:

1.2.1 Ácido Ribonucleico Alienígena (1242)

Passos:

- Para resolver este problema decidi usar Listas Duplas Encadeadas, pois elas me permitiram saber quem era meu nó anterior. É aqui que é armazenada a lista de letras inseridas pelo usuário.
- Então você começa a percorrer a lista letra por letra (S, F, B, C) perguntando se o próximo nó tem sua combinação certa, por exemplo se é a letra F você procura se o próximo nó é a letra S ou se a letra for B, pergunte se o próximo nó é a letra S e assim por diante.
- Quando a combinação de letras apropriada é encontrada no próximo nó, ambos os nós são removidos da nossa lista e adicionamos 1 ao nosso contador que tem o valor da resposta final. Se a combinação apropriada não for encontrada, a lista é rolada até chegar ao último elemento.

```
[2]: # PYTHON version 3.10.12

# Classe Nó
class Nodo:
    def __init__(self, valor):
        self.valor = valor # Valor nó
        self.siguiete = None
        self.anterior = None

class ListaEnlazada:
    def __init__(self):
```

```

        self.cabeza = None # Inicialmente a lista está vazia

def agregar(self, valor):
    nodo_nuevo = Nodo(valor) # Crie um novo nó
    if not self.cabeza: # Se a lista estiver vazia
        self.cabeza = nodo_nuevo # O novo nó é a cabeça
    else:
        nodo_actual = self.cabeza
        while nodo_actual.siguiente: # Vá até o fim
            nodo_actual = nodo_actual.siguiente
        nodo_actual.siguiente = nodo_nuevo # Vincule o novo nó ao final
        nodo_nuevo.anterior = nodo_actual # Link para o nó anterior

def eliminar(self, nodo):
    # Veja se é a cabeça
    if nodo.anterior:
        nodo.anterior.siguiente = nodo.siguiente
        if nodo.siguiente:
            nodo.siguiente.anterior = nodo.anterior
    else:
        if nodo.siguiente:
            nodo.siguiente.anterior = None
            self.cabeza = nodo.siguiente
        else:
            self.cabeza = None

def imprimir(self):
    nodo_actual = self.cabeza
    while nodo_actual: # Vá até o fim da lista
        print(nodo_actual.valor) # Imprimir valor do nó
        nodo_actual = nodo_actual.siguiente # Mover para o próximo nó

```

```

[3]: # Função principal
def main(texto):
    texto = texto.upper()
    letras = list(texto)

    # Crie uma lista vinculada
    lista = ListaEnlazada()

    for letra in letras:
        # Adicionar itens
        lista.agregar(letra)

    #-----
    # RESOLVENDO O PROBLEMA
    #-----

```

```

nodo_actual = lista.cabeza
# Contador da resposta
respuesta = 0

# Vá até o fim da lista
while nodo_actual:
    # Pergunte que letra é
    letra = nodo_actual.valor
    if nodo_actual.siguiete:
        letra_siguiete = nodo_actual.siguiete.valor
    else:
        letra_siguiete = None
    # Só pode ser S, B, F, C e S->B, B->S, F->C, C->F
    if (letra == 'S' and letra_siguiete == 'B') or (letra == 'B' and
↪letra_siguiete == 'S') or (letra == 'F' and letra_siguiete == 'C') or
↪(letra == 'C' and letra_siguiete == 'F'):
        # Nó a ser excluído
        nodo_eliminar = nodo_actual
        # Alterar nó atual
        # Pergunte se não é a cabeça
        if nodo_actual.anterior:
            nodo_actual = nodo_actual.anterior
        else:
            nodo_actual = nodo_actual.siguiete.siguiete

        # Excluir segundo nó
        lista.eliminar(nodo_eliminar.siguiete)
        # Excluir primeiro nó
        lista.eliminar(nodo_eliminar)
        # Adicione um à resposta
        respuesta = respuesta+1
    else:
        nodo_actual = nodo_actual.siguiete

print(respuesta)
#lista.imprimir()

```

```

[4]: input_data = []

try:
    while True:
        line = input() # Capturar entrada do console

        if line == "": # Condição para finalizar a entrada (pressione Enter em
↪uma linha vazia)
            break

```

```

        # Verifique se o comprimento está dentro do intervalo permitido
        if 1 <= len(line) <= 300:
            input_data.append(line) # Adicione a linha à lista se ela atender
            ↳ao comprimento
        else:
            print("Erro: a entrada deve ter entre 1 e 300 caracteres.") #
            ↳Mostra uma mensagem se não estiver em conformidade
except EOFError:
    pass # Lidar com EOF se ocorrer

# Imprima as entradas capturadas que atendem à condição
for texto in input_data:
    main(texto)

```

1.2.2 Complexidade do código

Classe “Nodo”

- Complexidade de tempo para criar um nó: $O(1)$ (constante).

Métodos da classe “ListaEnlazada”

a) Método “agregar”

- Complexidade de tempo: $O(n)$ (onde n é o número de nós na lista no momento da adição).

b) Método “eliminar”

- Complexidade de tempo: $O(1)$ (constante).

c) Método “imprimir”

- Complexidade de tempo: $O(n)$ (onde n é o número de nós na lista).

Função main

a) Adicionar letras à lista encadeada

- O método agregar é chamado para cada letra. No pior caso, adicionar um elemento tem complexidade de $O(n)$ (percorrendo toda a lista). Portanto, para um texto de comprimento m , a complexidade total de adicionar as letras é: $O(m^2)$ (para adicionar todas as letras).

b) Remover pares de nós

- Como a remoção de cada nó ocorre em tempo constante ($O(1)$), a complexidade do processo de remoção é: $O(m)$ (para percorrer e eliminar nós).

Laço principal para captura de entrada

- Complexidade de tempo: $O(p * m)$, onde p é o número de linhas de entrada e m é o comprimento médio de cada linha.

Resposta:

- Complexidade total de tempo: $O(n^2)$

1.2.3 LEXSIM - Avaliador Lexico e Sintático (1083)

Fonte: chatGPT

```
[2]: class Stack:
    def __init__(self):
        self.items = []

    def is_empty(self):
        return len(self.items) == 0

    def push(self, item):
        self.items.append(item)

    def pop(self):
        return self.items.pop() if not self.is_empty() else None

    def peek(self):
        return self.items[-1] if not self.is_empty() else None

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

# Definindo as prioridades dos operadores
precedence = {
    '^': 6,
    '*': 5,
    '/': 5,
    '+': 4,
    '-': 4,
    '>': 3,
    '<': 3,
    '=': 3,
    '#': 3,
    '.': 2, # AND
    '|': 1 # OR
}

# Função para verificar se um caractere é um operando
def is_operand(c):
    return c.isalnum() # Letras e números

# Função para verificar se um caractere é um operador
def is_operator(c):
```

```

    return c in precedence

# Função para transformar a expressão infixa em pós-fixa
def infix_to_postfix(expression):
    stack = Stack()
    output = []
    last_token_was_operator = True # Indicador se o último token foi um
    ↪ operador

    for token in expression:
        if is_operand(token): # Se for um operando, adicionar à saída
            output.append(token)
            last_token_was_operator = False # O último token não é um operador
        elif token == '(': # Se for um parênteses esquerdo, empilhar
            stack.push(token)
            last_token_was_operator = True # O último token é um operador
        elif token == ')': # Se for um parênteses direito
            while not stack.is_empty() and stack.peek() != '(':
                output.append(stack.pop())
            if stack.is_empty(): # Parênteses esquerdo faltando
                return "Syntax Error!"
            stack.pop() # Remover o '('
            last_token_was_operator = False # Ao fechar um parênteses, não é
            ↪ um operador
        elif is_operator(token): # Se for um operador
            if last_token_was_operator: # Verificar se há dois operadores
            ↪ consecutivos
                return "Syntax Error!"
            while (not stack.is_empty() and
                    stack.peek() in precedence and
                    precedence[stack.peek()] >= precedence[token]):
                output.append(stack.pop())
            stack.push(token)
            last_token_was_operator = True # O último token é um operador
        else:
            return "Lexical Error!" # Caractere inválido

    while not stack.is_empty():
        top = stack.pop()
        if top == '(':
            return "Syntax Error!" # Parênteses não balanceados
        output.append(top)

    if last_token_was_operator: # Verificar se o último token é um operador
        return "Syntax Error!"

    return ''.join(output)

```

```

# Função principal para processar as expressões
def process_expressions(expressions):
    for expression in expressions:
        expression = expression.strip() # Remover espaços em branco
        if expression: # Se não estiver vazio
            # Verificar se a expressão começa ou termina com um operador
            if expression[0] in precedence or expression[-1] in precedence:
                print("Syntax Error!")
            else:
                result = infix_to_postfix(expression)
                print(result)

# Exemplo de uso
expressions = [
    "(A+",
    "(A+B)*c",
    "(A+B)*%",
    "(a+b*c)/2*e+a",
    "(a+b*c)/2*(e+a)",
    "(a+b*c)/2*(e+a",
    "(ab+*c)/2*(e+a)",
    "(a+b*cc)/2*(e+a)",
    "('a+b*cc)/2*(e+a)",
    "a+b-c",
    "a-b*c/d+e"
    "(",
    "(A+",
    "(A+B)*c",
    "(A+B)*%",
    "(a+b*c)/2*e+a",
    "(a+b*c)/2*(e+a)",
    "(a+b*c)/2*(e+a",
    "(ab+*c)/2*(e+a)",
    "(a+b*cc)/2*(e+a",
    "('a+b*cc)/2*(e+a",
    "a+b-c",
    "a-b*c/d+e",
]

process_expressions(expressions)

```

Syntax Error!
 Syntax Error!
 AB+c*
 Lexical Error!
 abc**2/e*a+
 abc**2/ea+*

Syntax Error!
Syntax Error!
Syntax Error!
Lexical Error!
ab+c-
abc*d/-e+

```
[ ]: input_data = []

try:
    while True:
        line = input() # Capturar entrada do console

        if line == "": # Condição para finalizar a entrada (pressione Enter em
            ↪ uma linha vazia)
            break

        input_data.append(line)

except EOFError:
    pass # Lidar com EOF se ocorrer

# As entradas capturadas enviadas a funcao
process_expressions(input_data)
```

1.2.4 Complexidade do código

Fonte: ChatGPT

Classe “Stack”:

- **init:** $O(1)$ — Inicializa uma lista vazia.
- **is_empty:** $O(1)$ — Verifica se a pilha está vazia.
- **push:** $O(1)$ — Adiciona um item ao final da lista.
- **pop:** $O(1)$ — Remove e retorna o último item da lista, se não estiver vazia.
- **peek:** $O(1)$ — Retorna o último item da lista, se não estiver vazia, sem removê-lo.
- **size:** $O(1)$ — Retorna o número de itens na pilha.

Verificação de Operandos e Operadores:

- **is_operand:** $O(1)$ — Verifica se o caractere é um operando (letra ou número).
- **is_operator:** $O(1)$ — Verifica se o caractere é um operador através de um dicionário.

Função “infix_to_postfix”:

- Complexidade Geral: $O(n)$, onde n é o número de tokens na expressão.

Função “process_expressions”:

- Complexidade Geral: $O(m * n)$, onde m é o número de expressões e n é o número de tokens na expressão mais longa.

Resposta

- Complexidade total de tempo: $O(m*n)$