PROVA 1 - SEMESTRE 2021.2

Pedro Santi Binotto [20200634]

December 19, 2021

Questão 1

ITEM A)

Utilizando uma lista encadeada simples, é possível manter um registro de cada palavra (e quantas vezes a mesma foi utilizada) na forma de um "nodo" da lista, realizando o encadeamento direto de cada palavra diferente na ordem em que esta aparece no texto.

Para ilustrar esta solução, foram escritas (em Python ~~eca~~) as classes referentes à lista encadeada, os nodos, e uma classe "Reader", com o propósito de encapsular o a lógica utilizada ao ler o texto que servirá de exemplo na demonstração.

(Todo o código fonte apresentado neste documento (e mais!) está disponível no diretório "fonte" do projeto)

Nodo

A classe "Node" apresenta campos para o texto do elemento, isto é, a palavra em sí - a quantidade de vezes que foi utilizada, e um ponteiro para o próximo elemento na lista:

```
# Classe "Nodo"
class Node:
    def __init__(self, text: str, next=None):
        self.__quant = 1
        self.__text = text
        self.__next = next

    # getters e setters ... (disponiveis no fonte completo)
```

Lista

A classe "Lista" permite adicionar elementos e procurar por palavras com base no conteúdo do texto. A busca é realizada com base na iteração sobre cada elemento até que se econtre a palavra desejada ou que o fim da lista seja alcançado. Na adição, realiza-se uma busca sobre os elementos pela palavra; caso esta já esteja presente na lista, incrementa-se o valor do campo

de contagem do elemento - caso contrário, adiciona-se um novo elemento ao fim da lista.

```
# Classe "Lista"
class LinkedList:
    def __init__(self):
        self.__head: Node = None
        self.__tail: Node = None

    def __addElem(self, elem: Node):
        if self.__head == None:
            self.__head = elem
            self.__tail = elem
            return
        self.__tail.next = elem
        self.__tail = elem

    def storeWord(self, word: str):
        word = word.lower().translate(str.maketrans('', '',
            string.punctuation))
        tmp = self.__head
        if tmp == None:
            self.__addElem(Node(word))
            return
        while tmp.next is not None:
            if tmp.text == word:
                tmp.quant += 1
                return
            tmp = tmp.next
        self.__addElem(Node(word))

    # implementacao de demais metodos disponiveis no fonte completo
```

Reader

A classe "Reader" serve o propósito de abstrair o funcionamento necessário para ler o texto base, irrelevante para os propósitos desta demonstração.

```
# Classe "Reader"
class Reader:
```

```
    def __init__(self, textFile):
        text = textFile.open()
        self.__words = self.__parseFile(text)
        self.__cursorPos = 0
        text.close()

    # implementacao e metodos privados disponiveis no fonte completo

    def readNextWord(self):
        # [...]
```

Fazendo uso destas classes, é possível adicionar palavras com base em um texto de exemplo (neste caso, o texto está disponível em ./fonte/questao1/textoExemplo.txt) e, após isso, visualizar todas as palavras usadas e quantas vezes cada uma foi repetida.

```
def itemA():
    scriptLocation = Path(__file__).absolute().parent
    fileLocation = scriptLocation / 'textoExemplo.txt'

    lista = LinkedList()
    r = Reader(fileLocation)
    while True:
        el = r.readNextWord()
        if el == None:
            break
        lista.storeWord(el)

    tmp = lista.head
    while True:
        print( f"""{{ PALAVRA: '{tmp.text}', QUANTIDADE:
            {tmp.quant} }},""")
        if tmp.next == None:
            return
        tmp = tmp.next
```

Saída:

```
{ PALAVRA: 'spam', QUANTIDADE: 15 },
```

```
{ PALAVRA: 'lovely', QUANTIDADE: 2 },
{ PALAVRA: 'wonderful', QUANTIDADE: 2 },
{ PALAVRA: 'yes', QUANTIDADE: 1 },
{ PALAVRA: 'monty', QUANTIDADE: 1 },
{ PALAVRA: 'python', QUANTIDADE: 1 },
{ PALAVRA: 'unwittingly', QUANTIDADE: 1 },
{ PALAVRA: 'inspired', QUANTIDADE: 1 },
{ PALAVRA: 'the', QUANTIDADE: 3 },
{ PALAVRA: 'current', QUANTIDADE: 1 },
[ ... ]
```

ITEM B)

Utilizar uma tabela hash para realizar a mesma tarefa do item anterior produz uma solução mais eficiente quando se analisa o tempo de busca de um elemento qualquer em meio aos demais. Ao indexar os elementos de acordo com o seu conteúdo (atravéz de uma função hash), a probabilidade de colisões entre os elementos torna-se menor conforme a tabela cresce em capacidade (contanto que utilize-se uma função que gere resultados diversos o suficiente), assim aproximando o tempo de acesso dos elementos à O(1) (em casos ideais).

Para demonstrar a diferença em eficiência, foi desenvolvida uma classe "HashTable", também em Python, que armazena dados do mesmo tipo de "Nodo" que a classe "LinkedList" previamente observada:

```python
class HashTable:
    def __init__(self, tableSize: int):
        self.__size = tableSize
        self.__table = [LinkedList()] * self.__size

    @staticmethod
    def __hash(val: str, max: int) -> int:
        return sum([ord(char) for char in val]) \% max

    def storeWord(self, word: str):
        self.__table[self.__hash(word, self.__size)].storeWord(word)
```

```
    def lookUpWord(self, word: str):
        return self.__table[self.__hash(word,
            self.__size)].lookUpWord(word)
```

Após isso, a performance de cada estrutura foi testada e temporizada contra o mesmo texto-base (neste caso, o roteiro do filme "O Grande Lebowski"); armazenando todas as palavras do texto e realizando operações de busca pelas mesmas palavras em cada caso.

```
# item_a.py
def itemABenchmark():
    scriptLocation = Path(__file__).absolute().parent
    fileLocation = scriptLocation / 'biglebowski.txt'

    lista = LinkedList()
    r = Reader(fileLocation)
    while True:
        el = r.readNextWord()
        if el == None:
            break
        lista.storeWord(el)

    lista.lookUpWord('dude')
    lista.lookUpWord('opinion')
    lista.lookUpWord('dollars')
    lista.lookUpWord('vietnam')

# item_b.py
def itemB():
    scriptLocation = Path(__file__).absolute().parent
    fileLocation = scriptLocation / 'biglebowski.txt'

    tabela = HashTable(10)
    r = Reader(fileLocation)
    while True:
        el = r.readNextWord()
        if el == None:
            break
        tabela.storeWord(el)
```

```
    tabela.lookUpWord('dude')
    tabela.lookUpWord('opinion')
    tabela.lookUpWord('dollars')
    lista.lookUpWord('vietnam')
```

Resultados:

```
# utilizando uma tabela de tamanho 10
$ time python3 item_a.py
python3 item_a.py 5,88s user 0,04s system 99\% cpu 5,956 total

$ time python3 item_b.py
python3 item_b.py 5,28s user 0,01s system 99\% cpu 5,315 total

$ time python3 item_a.py
python3 item_a.py 6,14s user 0,01s system 99\% cpu 6,175 total

$ time python3 item_b.py
python3 item_b.py 5,73s user 0,01s system 99\% cpu 5,760 total

# lookUpWord('dude')  => 786 repeticoes
# lookUpWord('opinion') => 1 repeticoes
# lookUpWord('dollars') => 11 repeticoes
# lookUpWord('vietnam') => 4 repeticoes
```

ITEM C)

Para encontrar a palavra mais utilizada, ou ordenar as palavras de qualquer maneira, o hash não apresenta nenhuma vantagem significativa sobre a lista encadeada - o hashing, afinal, é feito sobre o conteúdo (texto) de cada palavra, portanto só torna-se mais eficiente em relação à lista quando o acesso é feito com base neste mesmo critério. Ao buscar pelo elemento de maior número (de repetições), seria necessário iterar sobre os elementos "cegamente", como no caso da lista encadeada.

ITEM D)

PROVA PROVA PROVA PROVA PROVA PROVA PROVA

Questão 2

ITEM A)

ITEM B)

Questão 3

PROVA PROVA PROVA PROVA PROVA PROVA PROVA

Questão 4

PROVA PROVA PROVA PROVA PROVA PROVA PROVA

Questão 5

PROVA PROVA PROVA PROVA PROVA PROVA PROVA