Aula 09: 30/AGO/2018

Aula passada

Problema: dada uma lista de palavras, contar quantas vezes cada palavra ocorre na lista.

Um ingrediente chave para resolver esse problema foi a implementação de um dicionário através de uma classe `Dicio`. Os atributos de estado dessa classe são duas listas: uma lista `chaves` e uma lista `valores`. Os métodos da intercae (=`API`) dessa classe foram `get(chave)` e `put(chave, valor)`

Na classe `Dicio` a lista `chaves` não é ordenada e as busca feita pelo método `get()` é sequencial. O consumo de tempo para construir o dicionário dessa classe é, no pior caso, proporciona a $\mathtt{n} \times \mathtt{m}$ (=$\mathtt{O}(\mathtt{nm})$) onde `n` é o número de palavras na lista e `m` é o número de chaves no dicionário. Experimentalmente essa solução começa a ser impraticável para listas grandes de palavras.

Este é o ponto de início de análise de algoritmos experimental e analítica (= *método científico*).

Hoje

Rudimentos de análise de algoritmos experimental e analítica. O pano de fundo continua sendo o problema de contar número de ocorrência de cada palavra em uma lista de palavras (dada através de um arquivo):

No momento temos uma classe `Dicio` com os métodos *privado* `indice()` e os métodos *públicos* `get()` e `put()` e um `main()` que contrói o dicionário.

Hoje faremos uma classe `Dicio` que utiliza o algoritmo de *ordenação por inserção* junto com *busca binária*. Veremos a análise experimental e análitica dessas implementações.

No método `put()` o uso do método `list.insert()` faz bastante diferença no tempo. A lição aqui é que métodos e funções nativas fazem o serviço mais rapidamente.

Começamos com o dicionário mais simples (o da aula passada) e vamos analisando e melhorando as implementações:

- incluir *busca binária* no método `indice()`;
- fazermos os *deslocamentos* em `put()`;
- usarmos `insert()` para os deslocamento.

O consumo de tempo dos programas dependem da entrada. Por isso consideramos as análises de **melhor caso**, **caso médio** e **pior caso**.

Em MAC0122 frequentemente deixamos de lado o *caso médio* devido a sua dificuldade. Entre outras coisas precisaríamos de uma distribuição de probabilidade dos dados.

Arquivos

No diretório `py` há três implementações de dicionário (`Dicio`):

- `dicio.py`: busca sequencial
- `dicio_bin.py`: busca binário com implementação do deslocamento
- `dicio_de_luxe.py`: busca binária com delocameto sendo feito com `list.insert()`

O programa *cliente* é o `conta_Dicio.py`. Para usar algum desses dicionários devemos alterar a importação no início do arquivo.

```
from dicio import Dicio
```

`main()`: conta palavras

Arquivo: `conta_Dicio.py`

Dado um arquivo com uma lista de palavras, conta o número de ocorrências de cada palavra na lista.

O trecho relevante é

```
    dicio = Dicio()
    for pal in lst_pals:
        valor = dicio.get(pal)
        if valor == None:
            dicio.put(pal, 1)
        else:
            dicio.put(pal, valor+1)
```

Aqui vai toda um versão simplificada da `main()`

```
PROMPT  = 'consulta >>> '

#-------------------------------------------------------------
def main():
    # leia o nome do arquivo
    nome = input("Digite o nome do arquivo com palavras: ")

    # abra o arquivo
    arq_entrada = open(nome, 'r', encoding='utf-8')

    print("lendo as palavras")
    # leia o texto no aqruivo
    texto = arq_entrada.read()

    # feche o arquivo
    arq_entrada.close()
    print("leitura encerrada")

    # crie lista de palavras
    print("criando dicionario de palavras")
    lista_pals = texto.split()

    # crie dicionário
    dicio = Dicio()
    for palavra in lista_pals:
        valor = dicio.get(palavra)
        if valor == None:
            dicio.put(palavra, 1)
        else:
            dicio.put(palavra, valor+1)

    # consulte o dicionário interativamente
    palavra = input(PROMPT).strip()
    while palavra != "":
```

```
valor = get(palavras, ocorrs, palavra)
print('%s: %s' %(palavra,str(valor)))
palavra = input(PROMPT).strip()
```

Classe `Dicio` com busca sequencial

Arquivo: `dicio.py`

Essa é a implementação da aula passada e o começo da aula. Os resultados experimentais estão mais adiante.

Implementação usando uma lista `chaves` e uma lista `valores`.

```
class Dicio:
    #--------------------------------------------
    def __init__(self):
        ''' (Dicio) -> None'''
        self.chaves  = []
        self.valores = []

    #--------------------------------------------
    def indice(self, chave):
        '''(Dicio, str) -> int ou None
        '''
        # apelidos
        n = len(self)
        chaves  = self.chaves
        for i in range(n):
            if chaves[i] == chave:
                return i
        return None

    #--------------------------------------------
    def get(self, chave):
        '''(Dicio, str) -> int ou None
        '''
        i = self.indice(chave)
        if i != None: return self.valores[i]
        return None


    #--------------------------------------------
    def put(self, chave, valor):
        '''(Dicio, str) -> None
        '''
        i = self.indice(chave)
        if i == None:
            self.chaves.append(chave)
            self.valores.append(valor)
        else:
            self.valores[i] = valor
```

Análise experimental

```
% python experimentos.py pal/hugo.pal -d0 -m300000
lendo as palavras no arquivo
leitura encerrada
criando listas de palavras
lista de palavras criada

  Análise experimental para criar dicionários
  -------------------------------------------

      m    tempo(s)  len(dicio)     tempo/m
  -----------------------------------------
    256      0.00         162        0.0000143
    512      0.01         279        0.0000239
   1024      0.04         476        0.0000401
   2048      0.10         833        0.0000506
   4096      0.35        1422        0.0000843
   8192      1.05        2312        0.0001285
  16384      2.90        3584        0.0001769
  32768      7.41        5444        0.0002261
  65536     19.46        8242        0.0002969
 131072     44.32       10947        0.0003381
 262144    111.10       16133        0.0004238
```

Análise

Suponha que cada linha consome uma quantidade de tempo constante. Esse hipótese é razoável pois blá-blá-blá.

```
    def indice(self, chave):          Melhor caso          Pior caso
        # apelidos                  no de execuções      no de execuções
        #                             da linha             da linha
        n = len(self)                 = 1                  = 1
        chaves  = self.chaves         = 1                  = 1
        i = 0                         = 1                  = 1
        while i < n:                  = 1                  = n + 1
            if chaves[i] == chave:    = 1                  = n
                return i              = 1                  = 0
            i += 1                    = 0                  = n
        return None                   = 0                  = 1
                                    -----------          ---------
                              TOTAL   = 6 (constante)      = 3n + 5 (linear)
```

O consumo de tempo de `get()` e `put()` são essencialmente iguais ao consumo de tempo de `indice()`

Conclusões

Melhor caso: Cada busca consome tempo constante. Uma sequência de `m` buscas pode consumir tempo proporcional a `m`.

Pior caso: Cada busca pode examinar todas as `n` chaves no dicionário. Uma sequência de `m` buscas pode consumir tempo proporcional a $m \times n$.

Classe `Dicio` com busca binária

Ideia: alterar `indice()` trocando busca sequencial por binária. Busca binária já foi vista no final de MAC0110/0115/2166.

O preço a pagar é manter a lista ordenada. Precisamos alterar `put()` para fazer os deslocamentos. Isso vai custar algo, mas já melhora, como mostram os experimentos.

Alteramos os métodos `indice()` e `put()` a alteração de `get()` é pequena.

O método `indice()` agora retorna `True` e `False` para indicar se a chave foi ou não encontrada. A implementação é sutil pois precisamos retornar a posição onde uma possível nova chave deve ser inserida ...

A relação invariante básica do método `indice()` está ilustrada logo a seguir.

```
<=   indica valor menor ou igual a chave (aqui = é supérfluo)
>    indica valor maior que a chave

+----+----+----+----+----+----+----+----+----+----+----+----+----+----+----+
| <= | <= | <= | <= | <= | ?  | ?  | ?  | ?  | ?  | >  | >  | >  | >  | >  |
+----+----+----+----+----+----+----+----+----+----+----+----+----+----+----+
                          ini                   fim

- para toda chave x em chaves[:ini] vale que x <= chave
- para toda chave x em chaves[fim:] vale que x > chave

quando a função pára sem encontrar a chave temos que ini == fim e a posição
onde a chave deve ser inserida é chaves[ini]
```

```
#-------------------------------------------          melhor caso     pior caso
def indice(self, chave):                             -----------     ---------
    n = len(self)                                      = 1             = 1
    chaves  = self.chaves                              = 1             = 1
    ini = 0                                            = 1             = 1
    fim = n                                            = 1             = 1
    while ini < fim:                                   = 1             ~ lg(n)
        meio = (ini+fim)//2                            = 1             ~ lg(n)
        if    chaves[meio] == chave: return True, meio = 1             = 0
        elif  chaves[meio] <= chave: ini = meio + 1    = 0             a a+b ~ lg(n)
        else: fim = meio                               = 0             b
    return False, ini                                  = 0             = 1
                                                       ------          --------
                                                TOTAL    = 7             = 3*lg(n)+5
```

```
    #-------------------------------------------
    def get(self, chave):
        '''(Dicio, str) -> int ou None

        Recebe um dicionário referenciado por `self` e uma chave (string).
        Retorna o valor correspondente à chave.

        Se a chave não está no dicionário, o método retorna None.
        '''
        achou, i = self.indice(chave)
        if achou: return self.valores[i]
        return None


    #-------------------------------------------
    def put(self, chave, valor):
        '''(Dicio, str) -> None

        Recebe um dicionário referenciado por `self`, uma chave (string) e o
        valor a ser associado à chave.
        Se a chave não está no dicionário ela é inserida com o valor.
        Se a chave está no dicionário, o seu valor é atualizado.
        '''
        achou, i = self.indice(chave)
        if achou:
            self.valores[i] = valor
        else:                                          # Melhor caso: insere no final e
            # apelidos                                  # consumo de tempo é constante
            chaves  = self.chaves                      # Pior caso: insere no começo inicio e
            valores = self.valores                     # o consumo de tempo é proporcional a n
            # abre espaco para a nova chave e valor
            n = len(self)
            chaves.append(0)
            valores.append(0)
            for j in range(n, i, -1):
                chaves[j], valores[j] = chaves[j-1], valores[j-1]
            # insere a chave e o valor
            chaves[i], valores[i] = chave, valor
```

Simulação de busca binária

```
chaves    n=11
+----+----+----+----+----+----+----+----+----+----+----+
| 10 | 20 | 25 | 35 | 38 | 40 | 44 | 50 | 55 | 65 | 99 |
+----+----+----+----+----+----+----+----+----+----+----+
   0    1    2    3    4    5    6    7    8    9   10   11
 ini                       m                        fim   chave = 53
                                ini          m        fim
                                ini    m   fim
```

```
                                        ini,m fim
                                             ini=fim=8
```

número de iterações

```
no de comparacoes   no de elementos
      0                   n
      1                   n/2
      2                   n/4 = n/2^2
     ...                  ...
      k                   n/2^k
```

k é tal que

```
     n/2^k > 1 => n > 2^k  => log_2 n > k
```

Logo, o número de comparacoes é $< \log_2 n$.

```
     n       log n
   ---------------
        16      4
       256      8
     65536     16
4294967296     32
```

Análise experimental

```
% python experimentos.py pal/hugo.pal -d1 -m300000
lendo as palavras no arquivo
leitura encerrada
criando listas de palavras
lista de palavras criada

  Análise experimental para criar dicionários
  -------------------------------------------

      m    tempo(s)  len(dicio)     tempo/m
  -----------------------------------------
     256     0.00        162        0.0000102
     512     0.01        279        0.0000132
    1024     0.02        476        0.0000175
    2048     0.05        833        0.0000236
    4096     0.13       1422        0.0000312
    8192     0.33       2312        0.0000397
   16384     0.75       3584        0.0000456
   32768     1.71       5444        0.0000523
   65536     3.86       8242        0.0000590
  131072     7.20      10947        0.0000549
  262144    15.61      16133        0.0000595
```

Conclusões

No melhor caso: `indice()` pode consumir tempo constante e uma sequencia de m buscas pode consumir tempo proporcional a m.

No pior caso: `indice()` consume tempo proporciona $\lg(n)$ e `put()` consome tempo proporcional a n e uma sequencia de m buscas pode consumir tempo proporcional a $m \times (n + \lg(n))$.

O consumo de tempo de `get()` é determinado por `indice()`.

Os experimentos mostram (e a intuição dizia) que valeu a pena o trabalho.

Classe `Dicio` com busca binária e insert()

Apenas alteramos `put()` para usar o método `insert()`. Na prática o consumo de tempo melhora.

```
    def put(self, chave, valor):
        achou, i = self.indice(chave)
        if achou:
            self.valores[i] = valor
        else:
            self.chaves.insert(i, chave)
            self.valores.insert(i, valor)
```

Análise experimental

```
% python experimentos.py pal/hugo.pal -d2 -m300000
lendo as palavras no arquivo
leitura encerrada
criando listas de palavras
lista de palavras criada

  Análise experimental para criar dicionários
  -------------------------------------------

      m    tempo(s)  len(dicio)     tempo/m
  -----------------------------------------
    256     0.00        162        0.0000050
    512     0.00        279        0.0000056
   1024     0.01        476        0.0000060
   2048     0.01        833        0.0000066
   4096     0.03       1422        0.0000072
   8192     0.06       2312        0.0000075
  16384     0.13       3584        0.0000079
  32768     0.27       5444        0.0000082
  65536     0.58       8242        0.0000089
 131072     1.21      10947        0.0000092
 262144     2.44      16133        0.0000093
```

Conclusão

Vale a pena usar métodos nativos. O consumo de tempo na prática melhora apesar de analiticamente continua o mesmo.

Teaser

Análise experimental com dicionário nativos.

```
% python experimentos.py pal/hugo.pal -d3 -m300000
lendo as palavras no arquivo
leitura encerrada
criando listas de palavras
lista de palavras criada

  Análise experimental para criar dicionários
  -------------------------------------------

      m    tempo(s)  len(dicio)     tempo/m
  -----------------------------------------
    256     0.00        162        0.0000004
    512     0.00        279        0.0000003
   1024     0.00        476        0.0000003
   2048     0.00        833        0.0000003
   4096     0.00       1422        0.0000004
   8192     0.00       2312        0.0000004
  16384     0.01       3584        0.0000004
  32768     0.01       5444        0.0000003
  65536     0.02       8242        0.0000004
 131072     0.05      10947        0.0000004
 262144     0.10      16133        0.0000004
```

Parece que o tempo por operação independe do tamanho do dicionário!

Resumo

A tabela a seguir mostra o consumo de tempo de `get()` e `put()` e do método auxiliar `indice()`. A linha com criação corresponde ao consumo de tempo para resolver o problema do número de ocorrência de palavras. Na tabela `m` representa o número de chaves no dicionário e `n` corresponde ao número de palavras na lista.

O símbolo `~` é uma abreviatura de *proporcional a*. O fator de proporcionalidade está sendo omitido.

O pior caso corresponde a *busca com sucesso* (= encontramos a chave). O pior caso corresponde a *busca com fracasso* (= não encontramos a chave).

busca sequencial	melhor caso	pior caso
`get()`	`~1`	`~m`
`put()`	`~1`	`~m`
`indice()`	`~1`	`~m`
`criação`	`~n`	`~nm`

busca binária	melhor caso	pior caso
`get()`	`~1`	$\lg \mathtt{m}$
`put()`	`~1`	$\mathtt{m} + \lg \mathtt{m}$
`indice()`	`~1`	$\lg \mathtt{m}$
`criação`	`~n`	$\sim \mathtt{nm} + \mathtt{n} \lg \mathtt{m}$

Quando possível devemos usar métodos e funções nativas do Python.

Apêndice

Métodos da classe nativa `list`

```
Método   Parâmetros    Resultado   Descrição
-----------------------------------------
append   item          mutador     Acrescenta um novo item no final da lista
insert   posição,item  mutador     Insere um novo item na posição dada
pop      nenhum        híbrido     Remove e retorna o último item
pop      posição       híbrido     Remove e retorna o item da posição.
sort     nenhum        mutador     Ordena a lista
reverse  nenhum        mutador     Ordena a lista em ordem reversa
index    item          retorna idx Retorna a posição da primeira ocorrência do item
count    item          retorna ct  Retorna o número de ocorrências do item
remove   item          mutador     Remove a primeira ocorrência do item
```