Prova Substitutiva — PMR3201 1o. semestre 2017

Prof. Thiago Martins

Instruções: Escreva o nome e o número USP na folha de papel almaço. Numere cada página. Indique o total de páginas na primeira página. Os códigos fornecidos na seção "Códigos-fonte de apoio" podem ser referenciados em qualquer resposta sem necessidade de reprodução.

1. (2,0 pontos) Uma tabela de Hash, ou espalhamento, com encadeamento linear, é uma sequência que armazena elementos com uma determinada chave. A posição de armazenamento de um determinado elemento é obtida pela função de Hash. Como funções de Hash colidem, ou seja, há chaves distintas que produzem valores de Hash idênticos, é possível que a posição ideal de um elemento já esteja preenchida. Neste caso o elemento é armazenado na próxima posição disponível. Para encontrar um elemento pela chave, tal tabela inicia a busca na posição indicada pela função de Hash e prossegue sequencialmente até encontrar a chave ou uma entrada vazia.

   A classe `HashLinear` apresenta uma implementação parcial de uma tabela de Hash. A função de Hash empregada é:

   $$H(k) = ((ak + b) \mod p)) \mod n$$

   Onde k é o valor da chave, a, b e p são constantes (respectivamente 5, 11 e 2147483647 nesta implementação) e n é o tamanho da sequência. Em particular, considere o método que *remove* um elemento da tabela:

```
def remove_valor(self, key):
    i = self._findpos(key)
    if self._e[i] == None:
        raise KeyError
    self._n -= 1
    j=i
    while True:
        self._e[i] = None
        skip = True;
        while skip:
            j = (j+1)%len(self._e)
            if self._e[j] == None:
                return
            k = self._getindex(self._e[j][0])
            if i<j:
               skip = (k>i) and (k<=j)
            else:
               skip = (i<k) or (k>=j)
        self._e[i] = self._e[j]
        i = j
```

   Este método retira da tabela o elemento armazenado sob a chave `key` ou lança a exceção `KeyError` caso a chave não exista. A operação de remoção deve remanejar de forma específica a tabela de hash para mantê-la consistente.

   Considere que em um determinado momento o estado de um objeto `tabela`, da classe `HashLinear` é:

   $$\texttt{tabela.\_n} = 5 \qquad \texttt{tabela.\_e} = \left[\underset{0}{9}, \underset{1}{12}, \underset{2}{6}, \underset{3}{0}, \underset{4}{\text{None}}, \underset{5}{\text{None}}, \underset{6}{\text{None}}, \underset{7}{4}\right]$$

   As entradas em `tabela._e` estão representadas pelos valores de suas chaves.

   Mostre o estado da tabela se forem aplicadas as operações:

(a) `tabela.remove_valor(0)`

(b) `tabela.remove_valor(6)`

(c) `tabela.remove_valor(4)`

As operações *não* são sucessivas, ou seja, cada uma inicia-se com a tabela no estado descrito acima.

Resposta:

(a) `tabela.remove_valor(0)`:

$$\texttt{tabela.\_n} = 4 \qquad \texttt{tabela.\_e} = \left[\underset{0}{9}, \underset{1}{12}, \underset{2}{6}, \underset{3}{\text{None}}, \underset{4}{\text{None}}, \underset{5}{\text{None}}, \underset{6}{\text{None}}, \underset{7}{4}\right]$$

(b) `tabela.remove_valor(6)`:

$$\texttt{tabela.\_n} = 4 \qquad \texttt{tabela.\_e} = \left[\underset{0}{9}, \underset{1}{12}, \underset{2}{\text{None}}, \underset{3}{0}, \underset{4}{\text{None}}, \underset{5}{\text{None}}, \underset{6}{\text{None}}, \underset{7}{4}\right]$$

(c) `tabela.remove_valor(4)`:

$$\texttt{tabela.\_n} = 4 \qquad \texttt{tabela.\_e} = \left[\underset{0}{9}, \underset{1}{6}, \underset{2}{\text{None}}, \underset{3}{0}, \underset{4}{\text{None}}, \underset{5}{\text{None}}, \underset{6}{\text{None}}, \underset{7}{12}\right]$$

2. (2,0 pontos) Considere sequências aritméticas iniciadas em 0, de razão 1, nas quais possivelmente falta um elemento. Por exemplo, na sequência $\{0, 1, 2, 4, 5, 6\}$ falta o elemento 3. Escreva uma função em Python que retorna o elemento faltante em uma sequência ou o elemento seguinte ao último, se nenhum elemento falta. A complexidade de sua função deve ser $\mathcal{O}(\log N)$. Complexidades piores valem (1,0) pontos. Use a seguinte assinatura:

```
def elemento_faltante(a):
```

Onde `a` é a sequência em questão.

Resposta: A complexidade desejada pode ser encontrada por uma busca binária:

```
def elemento_faltante(a):
    e = 0
    d = len(a)
    while e < d:
        m = (e+d)//2
        if a[m] > m:
            d = m
        else:
            e = m+1
    return e
```

3. (2,0 pontos) A função a seguir calcula o quadrado da soma dos números em uma sequência:

```
1  def quadrado_soma(a):
2      def recursivo(inicio, fim):
3          if inicio==fim:
4              return 0, 0
5          elif inicio==fim-1:
6              return a[inicio], a[inicio]*a[inicio]
7          else:
8              mid = (inicio+fim)//2
9              x, x2 = recursivo(inicio, mid)
10             y, y2 = recursivo(mid, fim)
11             return x+y, x2 + 2*x*y + y2
12     return recursivo(0,len(a))[1]
```

(a) (1,0 pontos) Mostre que a função está correta.

Resposta: A função `recursivo` interna divide a sequência em duas metades e chama a si mesma em cada uma. O caso-base é o de uma sequência nula ou unitária.

O caso-base é *sempre* atingido, pois o tamanho das sequências é estritamente decrescente.

A função retorna um par contendo a soma dos elementos em cada subsequência e seu quadrado. Isso é trivialmente verdadeiro nos casos base (Em que a função retorna ou $(0, 0)$ ou (a_i, a_i^2)).

Nos demais, suponha que a chamada na linha 9 atribui a `x` e `x2` respectivamente o total dos elementos da subsequência esquerda e o seu quadrado. Suponha o análogo para a linha 10 com `y` e `y2` para a subsequência da direita. Assim, o par retornado com $\left(x + y, x^2 + 2xy + y^2\right)$ contém respectivamente o total da sequência completa e o seu quadrado.

(b) (1,0 pontos) Calcule a complexidade da função em notação *Big-Oh* em função do comprimento da sequência n.

Resposta: A função recursiva chama a si mesma duas vezes em uma entrada com metade do tamanho da original e faz operações em custo constante. Na notação do *Master Theorem*:

$$T(N) = 2T\left(\frac{N}{2}\right) + \mathcal{O}(1)$$

Cuja solução é $T(N) = \mathcal{O}(N)$.

4. (2,0 pontos) Uma árvore binária de busca, os elementos da sub-árvore esquerda são estritamente menores do que o pai e os elementos da sub-árvore direita são estritamente maiores. A classe `NoArvoreBinariaBusca` implementa um nó de tal árvore. O campo `x` contém o valor armazenado no nó. O campo `e` é uma referência para a sub-árvore da esquerda. O campo `d` é uma referência para a sub-árvore da direita.

   Escreva uma função em Python que encontra em uma árvore binária de busca os dois elementos mais próximos (ou seja, o par de elementos da árvore cuja diferença é a menor). Use a seguinte assinatura:

   `def mais_proximos(r):`

   Onde `r` é uma referência ao nó raiz da árvore. A função deve retornar um par com os dois elementos mais próximos.

Resposta: Uma árvore binária de busca, enumerada por ordem interior, produz uma ordenação do seu conteúdo.

```
def mais_proximos(r):
    menor_init = False
    anterior_init = False
    anterior = None
    menor_dif = None
    dupla = None
    def procura(n):
        nonlocal anterior_init, anterior, menor_init, menor_dif, dupla
        if n.e:
            procura(n.e)
        if anterior_init:
            dif = n.x - anterior
            if not(menor_init) or dif < menor_dif:
                menor_dif = dif
                dupla = anterior, n.x
                menor_init = True
        anterior = n.x
        anterior_init = True
        if(n.d): procura(n.d)
    procura(r)
    return dupla
```

5. (2,0 pontos) Em uma sequência de inteiros *não-ordenada*, determine em tempo *linear* se há dois elementos *distintos* cuja soma seja exatamente x. Sua função deve ter complexidade $\mathcal{O}(N)$, onde N é o tamanho da sequência.

   Use a seguinte assinatura:

   `def existe_soma(a, x):`

   Onde `a` é a sequência de inteiros e `x` é o valor a ser atingido pela soma de dois elementos.

Resposta: A solução é usar uma tabela de Hash, como a da classe `HashLinear`:

```
def existe_soma(a, x):
    h = HashLinear()
    for i in a:
        if h.contem_chave(x-i):
            return True
        h.define_valor(i, None)
    return False
```

Formulário

Solução de equações de recorrência pelo *Master Theorem*:

A equação:

$$T(N) = A \cdot T\left(\frac{N}{B}\right) + cN^L,$$

com $T(1) = \mathcal{O}(1)$, tem solução

$$T(N) = \begin{cases} \mathcal{O}\left(N^{\log_B A}\right) & \text{se } A > B^L \\ \mathcal{O}\left(N^L \log N\right) & \text{se } A = B^L \\ \mathcal{O}\left(N^L\right) & \text{se } A < B^L \end{cases}$$

Códigos-fonte de apoio

Classe `HashLinear`

```
class HashLinear:
    """Implementa uma tabela de Hash com encadeamento Linear"""
    _p = 2147483647
    def __init__(self):
        self._e = [None]*8
        self._a = 5
        self._b = 11
        self._n = 0

    def _getindex(self, key):
        return ((self._a*hash(key)+self._b)%HashLinear._p)%len(self._e)

    def _findpos(self, key):
        """Encontra a posição que armazena key ou a próxima com None"""
        i = self._getindex(key)
        while self._e[i] and self._e[i][0]!=key:
            i = (i+1)%len(self._e)
        return i

    def contem_chave(self, key):
        return bool(self._e[self._findpos(key)])

    def define_valor(self, key, value):
        i = self._findpos(key)
        if self._e[i]:
            self._e[i] = key, value
        else: # Novo item
           self._e[i] = key, value
           self._n += 1
           self._checkcount()

    def recupera_valor(self, key):
        i = self._findpos(key)
        if self._e[i] == None:
            raise KeyError
        else:
           return self._e[i][1]

    def remove_valor(self, key):
        i = self._findpos(key)
        if self._e[i] == None:
            raise KeyError
        self._n -= 1
        j=i
        while True:
            self._e[i] = None
            skip = True;
            while skip:
                j = (j+1)%len(self._e)
                if self._e[j] == None:
                    return
                k = self._getindex(self._e[j][0])
                if i<j:
                   skip = (k>i) and (k<=j)
                else:
                   skip = (i<k) or (k>=j)
            self._e[i] = self._e[j]
            i = j

    def _checkcount(self):
        """ Limita o número de entradas a 75% da tabela"""
        if self._n*3 > len(self._e)*2:
            old_e = self._e
            self._e = [None]*(2*len(old_e))
            self._n = 0
            for e in old_e:
                if e:
                    self[e[0]] = e[1]
```

Classe `NoArvoreBinaria`

```
class NoArvoreBinariaBusca:
    def __init__(self, x, e = None, d = None):
        self.x = x
        self.e = e
        self.d = d
```

Classe `NoListaLigada`

```
class NoListaLigada:
    def __init__(self, x):
        """Inicializa nó isolado"""
        self._x = x
        self.p = None
        self.n = None

    def set_after(self, other):
        """define other como o próximo nó de self"""
        self.n = other
        if other:
            other.p = self

    def set_before(self, other):
        """define other como o nó anterior a self"""
        self.p = other
        if other:
            other.n = self
```