2a Prova — 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. (2,5 pontos) Uma lista duplamente ligada é uma cadeia de elementos nos quais há uma referência para o *próximo* e o *anterior*. A classe `NoListaLigada` implementa tal nó. A propriedade `n` é uma referência ao *próximo* nó da cadeia, ou `None` se o nó for o último. A propriedade `p` é uma referência ao nó *anterior* da cadeia, ou `None` se o nó for o primeiro.

   Implemente em Python uma função que retira de uma lista duplamente ligada uma subsequência de elementos. Use a seguinte assinatura:

   ```
   def remove(head, a, b):
   ```

   Onde `head` é uma referência ao *primeiro* elemento da lista duplamente ligada, `a` é o nó inicial da subsequência e `b` é o nó final. A função deve retornar o primeiro elemento da lista ligada *após* a remoção.

   Resposta: A remoção de uma subsequência é similar à remoção de um único nó. É necessário vincular o nó antecessor de `a` ao nó sucessor de `b`. Há um caso especial em que `head` aponta para o mesmo nó que `a`.

   ```
   def remove(head, a, b):
       if head is a:
           head = b.n
           if b.n:
             b.n.p = None
       else:
           a.p.n = b.n
           if b.n:
             b.n.p = b.p
       return head
   ```

2. (2,5 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 7, 11 e 31 nesta implementação) e n é o tamanho da sequência (fixa em 8 nesta implementação). O campo `_e` é a sequência que armazena as entradas na tabela (em pares com a chave na posição 0 e o valor na posição 1). Entradas vazias contém o valor `None`. O método `__setitem__(self, key, value)` armazena o valor em `value` sob a chave `key`. O método `__getitem__(self, key)` retorna o valor armazenado sob a chave `key` ou lança

a exceção `KeyError`. Por simplicidade, estão omitidos métodos tradicionais como o de remoção e a sequência possui comprimento fixo.

(a) (1,5 pontos) As tabelas abaixo apresentam sequências de valores de chaves. Quais delas são possíveis estados da sequência `_e`?

A	
i	k
0	None
1	2
2	None
3	10
4	None
5	18
6	None
7	5

B	
i	k
0	14
1	2
2	4
3	5
4	None
5	18
6	None
7	15

C	
i	k
0	5
1	2
2	None
3	14
4	None
5	18
6	None
7	15

D	
i	k
0	4
1	2
2	1
3	0
4	9
5	7
6	6
7	5

Resposta: Se a tabela é válida todas as chaves são recuperáveis. Valores de Hash para os valores de chave indicados:

k	0	1	2	4	5	6	7	9	10	14	15	18
$H(k)$	3	2	1	0	7	6	5	4	3	0	7	5

Como se pode ver, para as tabelas A e B, todos os elementos são recuperáveis. Na tabela C, o elemento com chave 14 *não* pode ser recuperado. A tabela D possui todos os valores recuperáveis. O seu fator de carga, no entanto, é igual a 1 (todas as posições ocupadas), o que não é compatível com uma tabela de hash desta natureza. De fato, qualquer busca por um elemento que não está na tabela jamais retornaria.

(b) (1,0 pontos) Para cada sequência válida do item (a), indique uma possível ordem de inserção de chaves na tabela que a produziria.

Resposta:

Tabela A: Qualquer sequência é possivel, por exemplo, 2, 10, 18, 5

Tabela B: O elemento 4 está deslocado da sua posição ideal pelo 14 e 2. Já o elemento 5 está deslocado de sua posição ideal pelo 15, 14, 2, e 4. Uma sequência que gera este deslocamento é 2, 14, 4, 15, 5, 18.

3. (2,5 pontos) O *nível* de uma árvore é um conjunto de nós aos quais se chega pela mesma quantidade de passos a partir do nó raiz. Uma árvore binária é uma árvore na qual cada nó tem no máximo dois descendentes. Uma árvore binária diz-se *perfeitamente balanceada* quando todos os seus níveis exceto o último estão completamente preenchidos. A classe `NoArvoreBinaria` implementa um nó de árvore binária. O campo `e` é uma referência para um dos seus descendentes (a sub-árvore "à esquerda") e o `d` para o outro (a sub-árvore "à direita"). Caso algum descendente não exista, a referência correspondente assume o valor `None`.

   Escreva em Python uma função que retorna `True` se a árvore está perfeitamente balanceada e `False` se não está. Use a seguinte assinatura:

```
def perf_balanceada(n):
```

   onde `n` é uma referência para o objeto da classe `NoArvoreBinaria` raiz da árvore em questão.

Resposta: A solução é similar ao cálculo de altura de uma árvore. O truque é obter duas alturas, a máxima e a mínima. Quando a altura máxima difere da mínima por no máximo uma posição, a árvore está perfeitamente balanceada.

```
def perf_balanceada(n):
    def max_alt(n):
        if n:
            return 1 + max(max_alt(n.e), max_alt(n.d))
        else:
            return 0
    def min_alt(n):
        if n:
            return 1 + min(max_alt(n.e), max_alt(n.d))
        else:
            return 0

    return max_alt(n)-min_alt(n) <= 1
```

4. (2,5 pontos) O problema do *valor mais recente menor que* é o problema de, dada uma sequência de n números $A = \{a_1, \ldots, a_n\}$, produzir outra sequência de comprimento igual $B = \{b_1, \ldots, b_n\}$ tal que b_i é o mais próximo antecessor de a_i tal que $b_i < a_i$ ou um símbolo nulo $\varnothing$ se a_i é o menor elemento até o índice i.

   Por exemplo, dada a sequência $\{0, 1, 2, 3, 4, 2, 1, -1, 5, 4, 3, 4, 5, 9, 5\}$, a sequência correspondente de valores mais recentes menores que é $\{\varnothing, 0, 1, 2, 3, 1, 0, \varnothing, -1, -1, -1, 3, 4, 5, 4\}$ (verifique!).

   Escreva uma função em Python que resolve o problema do valor mais recente menor que em tempo *linear*, ou seja, $\mathcal{O}(n)$. Use a seguinte assinatura:

   ```
   def recente_menor(a, b)::
   ```

   Onde `a` é a sequência de números e `b` é a sequência de saída, com o mesmo comprimento de `a`. Sua função deve sobreescrever a sequência em `b` com a sequência de valores mais recentes menores que dos valores de `b`. Use `None` como símbolo nulo.

   Sugestão: Pode parecer desafiador escrever este código em tempo linear, mas considere os valores anteriormente calculados. Eles podem ser útil para algo?

Resposta: A solução é surpreendentemente simples. Basta manter uma pilha com os valores de `a`. Assim, procura-se na pilha o primeiro elemento menor do que o atual.

```
def recente_menor(a, b):
    p = Pilha()
    for i, x in enumerate(a):
        while p.tamanho()>0 and p.top() >= x:
            p.pop()
        if p.tamanho() == 0:
            b[i] = None
        else:
            b[i] = p.top()
        p.push(x)
```

Embora os dois laços sugiram uma complexidade $\mathcal{O}(n^2)$, o máximo de iterações totais que o laço interno faz é igual à quantidade de elementos empilhados, e esta é igual a n.

Códigos-fonte de apoio

Classe `HashLinear`

```
class HashLinear:
    """Implementa uma tabela de Hash com encadeamento Linear"""
    _p = 31

    def __init__(self):
        self._e = [None]*8
        self._a = 7
        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 entrada na sequência com chave key ou None"""
        i = self._getindex(key)
        while self._e[i] and self._e[i][0]!=key:
            i = (i+1)%len(self._e)
        return i

    def __setitem__(self, key, value):
        i = self._findpos(key)
        if self._e[i]:
            self._e[i] = key, value
        else: # Novo item
            if self_n >= len(self._e):
                raise Exception("Tabela␣Cheia")
            self._e[i] = key, value
            self._n += 1
            self._checkcount()

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

Classe `NoArvoreBinaria`

```
class NoArvoreBinaria:
    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
```

Classe `Pilha`

```
class Pilha:
    """Implementa uma pilha usando listas ligadas"""
    def __init__(self):
        self._e = None
        self._n = 0

    def tamanho(self):
        return self._n

    def push(self, x):
        """Adiciona um elemento à Pilha"""
        novo = NoListaLigada(x)
        novo.set_after(self._e)
        self._e = novo
        self._n += 1

    def pop(self):
        """Remove um elemento da Pilha"""
        if self._e is None:
            raise Exception("Pilha␣Vazia")
        x = self._e._x
        self._e = self._e.n
        if self._e:
            self._e.p = None
        self._n -= 1
        return x

    def top(self):
        """Retorna o elemento do topo sem alterar a Pilha"""
        if self._e is None:
            raise Exception("Pilha␣Vazia")
        return self._e._x
```

Classe `Fila`

```
class Fila:
    """Implementa uma fila usando listas ligadas"""
    def __init__(self):
        self._e = None
        self._s = None
        self._n = 0

    def tamanho(self):
        return self._n

    def enqueue(self, x):
        """Adiciona um elemento à fila"""
        if self._e is None: # Fila Vazia
            self._s = self._e = NoListaLigada(x)
        else:
            novo = NoListaLigada(x)
            novo.set_after(self._e)
            self._e = novo
        self._n += 1

    def dequeue(self):
        if self._s is None: # Fila Vazia
            raise Exception("Fila␣Vazia")
        x = self._s._x
        self._s = self._s.p
        if self._s:
            self._s.n = None
        else: # Fila esvaziada!
            self._e = None
        self._n -= 1
        return x
```