2a Prova — PMR3201 **1o. semestre 2019**
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. Não referencie elementos de uma classe iniciados pelo caractere "_" a menos que o seu código faça parte da implementação desta classe. A menos que expressamente instruído ao contrário, as funções que você criar não devem modificar os parâmetros passados. Você não pode empregar nenhum método do *runtime* python que tenha complexidade pior do que constante.

Questões

1. (2,5 pontos) O algoritmo de Dijkstra atribui a cada nó de um grafo a sua distância mínima até um nó de origem. A cada iteração i o algoritmo atribui a cada nó k um valor $D_i[k]$ que corresponde a um *limite superior* para a distância do nó até o nó de origem. Também a cada iteração é estabelecido um conjunto de nós S_i para os quais o valor $D_i[k]$ corresponde ao menor limite superior possível (ou seja, o tamanho do menor caminho ao nó de origem).

   A cada nova iteração o algoritmo escolhe o nó n_{i+1} que não está em S_i para o qual o valor de $D[n]$ é mínimo. Este nó é adicionado ao conjunto S_{i+1}. As distâncias $D_{i+1}[k]$ são atualizadas para todos os nós k fora de S_{i+1} conectados ao nó $n+1$ com o *menor* valor entre $D_i[k]$ e $D_i[n_{i+1}] + V_{n_{i+1},k}$, onde $V_{n_{i+1},k}$ é o valor da aresta que liga n_{i+1} e k.

   Sendo o o nó de origem, na primeira iteração, $n_1 = o$, $S_1 = \{o\}$, $D_1[o] = 0$, $D_1[k] = V_{o,k}$ para todo nó conectado diretamente ao nó de origem e $D_1[k] = \infty$ para todos os nós restantes. O algoritmo procede até que todos os nós sejam adicionados ao conjunto S_i.

   Considere o grafo da Figura 1.

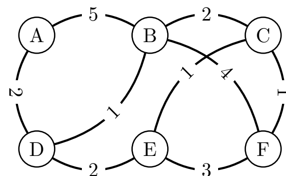


Figura 1: Grafo para a questão 1

   (a) (2,0 pontos) Preencha uma tabela com os resultados do algoritmo de Dijkstra aplicado ao grafo com nó de origem A. Sua tabela deve conter os valores de i, n_i, S_i e os valores de $D_i[A], \ldots, D_i[F]$ para cada um dos nós.

Resposta:

i	n_i	S_i	$D_i[A]$	$D_i[B]$	$D_i[C]$	$D_i[D]$	$D_i[E]$	$D_i[F]$
1	A	$\{A\}$	0	5	∞	2	∞	∞
2	D	$\{A, D\}$	0	3	∞	2	3	∞
3	B	$\{A, D, B\}$	0	3	5	2	4	7
4	E	$\{A, D, B, E\}$	0	3	5	2	4	7
5	C	$\{A, D, B, E, C\}$	0	3	5	2	4	6
6	F	$\{A, D, B, E, C, F\}$	0	3	5	2	4	6

   (b) (0,5 pontos) Qual o caminho mais curto do nó A ao nó F?

Resposta:
O caminho $\{A, D, E, C, F\}$.

2. (2,5 pontos) A classe `HashLinear` implementa uma tabela de espalhamento (*Hash*) com encadeamento na própria tabela. Esta classe usa uma estratégia de "pseudo-remoção" na qual os elementos removidos são marcados como "ausentes", embora a chave correspondente persista na tabela. O campo `_o` contém o número de entradas ocupadas na tabela, enquanto que o campo `_n` contém o número de elementos efetivamente armazenados. O campo `_e` contém a tabela propriamente dita. Cada elemento da sequência é uma trinca de valores. O primeiro valor é a chave armazenada, o segundo é um valor booleano que contém `True` se a entrada realmente contém um valor armazenado ou `False` se foi previamente removida e o terceiro é o valor armazenado (válido somente se o segundo for `True`). O método `_getindex(self, key)` retorna a posição ideal de armazenamento da chave `key`. O método `_findpos(self, key)` retorna a posição na tabela que contém a chave `key` ou a próxima posição vazia. O método `__contains__(self, key)` retorna verdadeiro se há algum valor armazenado sob a chave `key` e é acessado com a sintaxe `key in t` onde `t` é a tabela. O método `__getitem__(self, key)` retorna o valor armazenado sob a chave `key` e é acessado com a sintaxe `t[key]` onde `t` é a tabela. O método `__setitem__(self, key, value)` armazena o valor `value` sob a chave `key` (ou modifica armazenamento preexistente) e é acessado com a sintaxe `t[key]=value` onde `t` é a tabela. O método `__delitem__(self, key)` remove o valor armazenado sob a chave `key` e é acessado com a sintaxe `del t[key]` onde `t` é a tabela.

   Finalmente, o método `_checkcount(self)` verifica se o número de entradas ocupadas na tabela excede 50% do tamanho da mesma. Caso exceda, a tabela é regerada, mantendo-se somente os elementos realmente armazenados e removendo-se definitivamente os marcados como removidos. Se mesmo a limpeza de elementos previamente removidos é insuficiente para fazer o numero de entradas ocupadas cair para abaixo de 50% do tamanho da tabela, a nova tabela terá o dobro do tamanho da anterior. Forneça uma implementação do método `_checkcount(self)`. Você não deve alterar nenhum outro método da classe.

   Sugestão: Para regerar a tabela, crie novos valores consistentes para os campos `_o`, `_n` e `_e` e utilize os métodos existentes da classe.

Resposta:

```
def _checkcount(self):
    """ Limita o número de entradas a 50% da tabela"""
    if self._o*2 > len(self._e):
        old_e = self._e
        # Só dobra a tabela se o número de entradas reais
        #  também exceder 50% do tamanho
        if self._n*2 > len(self._e):
            self._e = [None]*(2*len(old_e))
        else:
            self._e = [None]*(len(old_e))
        self._n = 0
        self._o =

        for e in old_e:
            # Readiciona só os elementos não-removidos
            if e and e[1]:
                self[e[0]] = e[2]
```

3. (2,5 pontos) A classe `NoListaLigada` implementa um nó de lista simplesmente ligada. O campo `x` contém o valor armazenado no nó e o campo `n` contém uma referência para o próximo nó na lista ou `None` caso o nó seja o último.

Implemente em Python uma função que remove todos os elementos repetidos de uma lista ligada mantendo somente o *último*. Use a seguinte assinatura:

```
def remove_duplicados(raiz):
```

Onde `raiz` é o primeiro nó da sequência. A função deve retornar o novo primeiro nó. Sua função deve ter complexidade $\mathcal{O}(N)$ onde N é a quantidade de elementos na sequência. Complexidades piores valem 1,5 pontos.

Resposta: Uma tabela de hash é empregada para armazenar o pai do nó que armazena cada valor. A solução a seguir usa um "pseudo-pai" para simplificar o caso da remoção do primeiro nó. Note o cuidado especial quando um elemento é removido: É necessário verificar se este elemento não é pai de outro elemento já na tabela.

```
def remove_duplicados(raiz):
  ant = NoListaLigada(None, raiz)
  pai = ant
  t = HashLinear()
  while raiz:
      if raiz.x in t:
          pai_remover = t[raiz.x]
          pai_remover.n = pai_remover.n.n
          if pai_remover.n and pai_remover.n.x in t:
              t[pai_remover.n.x] = pai_remover
      t[raiz.x] = pai
      pai = raiz
      raiz = raiz.n
  return ant.n
```

4. (4 pontos) A classe `NoArvoreBinaria` implementa uma árvore binária. O campo `x` contém o elemento armazenado em um nó, o campo `e` é uma referência ao nó raiz da sub-árvore esquerda e o campo `d` é uma referência ao nó raiz da sub-árvore direita. O código abaixo é uma função que enumera os elementos de uma árvore binária em ordem *anterior*, ou seja, o conteúdo do nó raiz é avaliado, em seguida o conteúdo da sub-árvore esquerda e finalmente o conteúdo da sub-árvore direita.

   O código abaixo implementa uma enumeração em ordem anterior.

```
1 def anterior(raiz, v):
2     if raiz != None:
3         v.append(raiz.x)
4         anterior(raiz.e, v)
5         anterior(raiz.d, v)
```

   onde `raiz` é o nó raiz da árvore e `v` é um vetor vazio que recebe a enumeração.

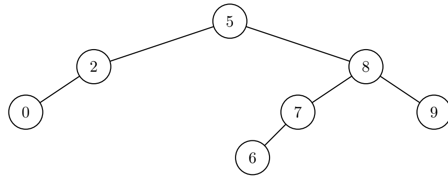
   Uma árvore binária de busca é uma árvore binária na qual cada nó é estritamente maior do que os nós da sua sub-árvore esquerda e estritamente menor do que os nós da sua sub-árvore direita.

   (a) (1,0 pontos) Verifique se os vetores abaixo são possíveis enumerações anteriores de uma árvore binária de busca. Para todas as possíveis enumerações, desenhe a árvore equivalente.

   1. `[5, 2, 0, 8, 7, 6, 9]`
   2. `[5, 1, 2, 4, 0, 7, 6]`
   3. `[5, 3, 4, 2, 6, 8, 7]`
   4. `[7, 4, 2, 3, 5, 9, 8]`

   Resposta: Uma enumeração anterior mostra os nós da sub-árvore direita *depois* dos nós da sub-árvore esquerda.

1. `[5, 2, 0, 8, 7, 6, 9]`

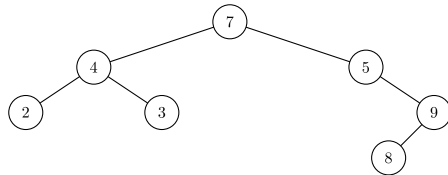
2. `[5, 1, 2, 4, 0, 7, 6]`

   O valor 4 sucede o 2. Isso significa que *nenhum* elemento subsequente pode ser menor do que 2. Deste modo, a sequência *não* pode ser enumeração anterior de árvore binária de busca.

3. `[5, 3, 4, 2, 6, 8, 7]`

   O valor 4 sucede o 3. Isso significa que *nenhum* elemento subsequente pode ser menor do que 3. Deste modo, a sequência *não* pode ser enumeração anterior de árvore binária de busca.

4. `[7, 4, 2, 3, 5, 9, 8]`

(b) (3,0 pontos) Escreva uma função em Python que verifica se um dado vetor é possível enumeração em ordem anterior de uma árvore binária de busca. Use a seguinte assinatura:

`def verifica_enumeracao_ant(v):`

onde `v` é o vetor com a possível enumeração. Seu código deve ter complexidade $\mathcal{O}(N)$ onde N é o tamanho do vetor. Complexidades piores valem 1,5 pontos.

Indique a complexidade do seu algoritmo!

Resposta: O problema é similar ao de se encontrar o "elemento mais próximo menor que". Esta implementação usa as sequências de Python como pilha.

```
def verifica_enumeracao_ant(v):
    r = -math.inf
    p = []
    for x in v:
        if x < r:
            return False
        while len(p) and p[-1] < x:
            r = p.pop()
        p.append(x)
    return True
```

Códigos-fonte de apoio

Classe `HashLinear`

```
class HashLinear:
    """Implementa uma tabela de Hash com encadeamento Linear"""
    _p = 2147483647

    def __init__(self):
        self._e = [None]*4
        self._a = 1+random.randrange(HashLinear._p-1)
        self._b = random.randrange(HashLinear._p)
        self._n = 0
        self._o = 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 que contém a chave key,
        #   ou uma entrada 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 __contains__(self, key):
        i = self._findpos(key)
        return bool(self._e[i]) and self._e[i][1]

    def __setitem__(self, key, value):
        i = self._findpos(key)
        if self._e[i]:
            if not self._e[i][1]: self._n += 1
            self._e[i] = key, True, value
        else: # Novo item
           self._e[i] = key, True, value
           self._n += 1
           self._o += 1
           self._checkcount()

    def __getitem__(self, key):
        i = self._findpos(key)
        if self._e[i] == None or not self._e[i][1]:
            raise KeyError
        else:
           return self._e[i][2]

    def __delitem__(self, key):
        i = self._findpos(key)
        if self._e[i] == None or not self._e[i][1]:
            raise KeyError
        # Pseudo-remoção, marca como vazio
        self._e[i] = key, False, None
        self._n -= 1

    def __len__(self):
        return self._n

    def _checkcount(self):
        """ Limita o número de entradas a 50% da tabela"""
```

Classe `NoListaLigada`

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

Classe `NoArvoreBinaria`

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