2a Prova — PMR3201 1o. semestre 2018
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) A codificação de Huffman codifica símbolos em cadeias de *bits* organizando-os em árvores binárias a partir de sua distribuição relativa de modo a minimizar a quantidade de *bits* necessária para codificar uma mensagem com estes símbolos.

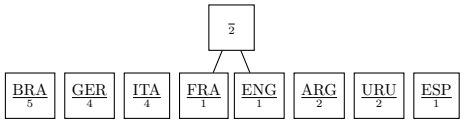
   (a) (1,5 pontos) A tabela 1 mostra a quantidade de vitórias por país na copa do mundo FIFA. Construa o código de Huffman que codifica o país vitorioso usando esta distribuição. Caso durante a construção do dicionário duas árvores tenham exatamente a mesma frequência, considere que a árvore com *maior* total de símbolos vem *antes* da ávore com *menor*. Se ainda assim as árvores empatam, a árvore que vem *antes* é a que tem o país em mais alta posição na tabela 1.

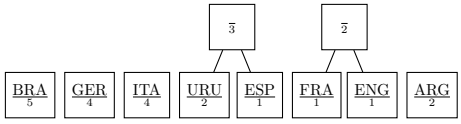
Tabela 1:

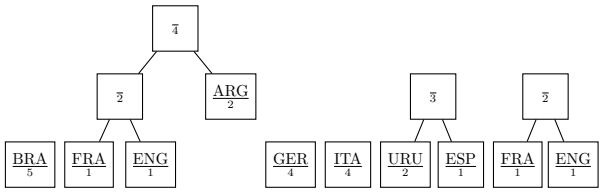
País	Vitórias
BRA	5
GER	4
ITA	4
ARG	2
URU	2
ESP	1
FRA	1
ENG	1

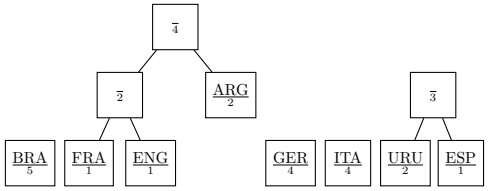
Resposta: A construção segue os seguintes passos:

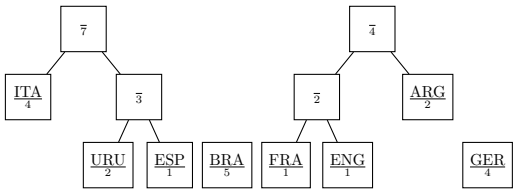
1. 

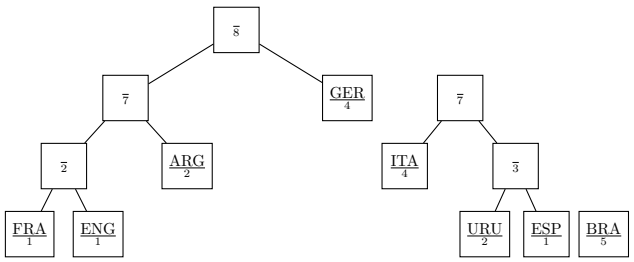
2. 

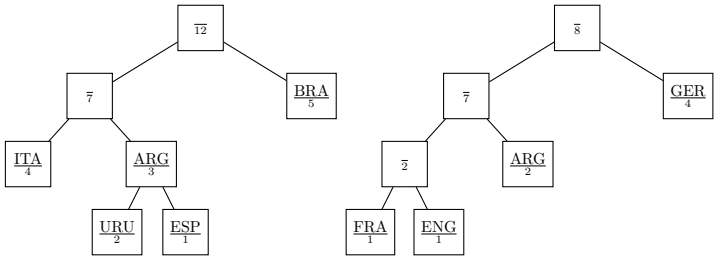
3. 

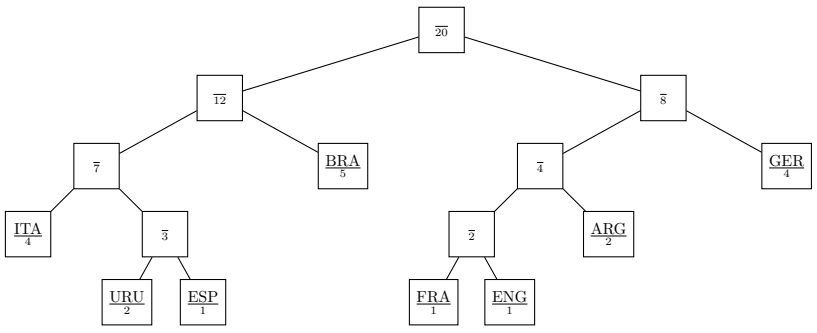
4. 

5. 

6. 

7. 

8. 

9. 

E o código equivalente é:

País	código	tam.
BRA	`01`	2
GER	`11`	2
ITA	`000`	3
ARG	`010`	3
URU	`0010`	4
ESP	`0011`	4
FRA	`1000`	4
ENG	`1001`	4

(b) (1,0 pontos) A vitória em 2018 de quais países na tabela 1 *não* modifica o dicionário da parte a)?

Resposta: Para alterar o dicionário da parte a), a vitória de um país em 2018 deve mudar a ordem de alguma sub-árvore na construção do código. No primeiro nível, modificações em ENG, FRA, URU e ITA modificam a ordem. No segundo nível, uma modificação em ARG altera a ordem. No terceiro nível, uma modificação em ESP altera a ordem. No quarto nível uma modificação em GER altera a ordem.

Resta apenas BRA que, ainda que vitorioso em 2018, não modifica o dicionário.

2. (2,5 pontos) Uma lista ligada é uma estrutura de dados formada pelo encadeamento de *nós*. Neste encadeamento, cada nó contém, além de algum dado armazenado, uma referência ao *próximo* nó na sequência. A classe `NoListaLigada` implementa esta estrutura de dados. O método `get_next(self)` retorna o próximo nó na cadeia ou o valor `None` caso este seja o último. O método `set_next(self, n)` define o nó referenciado por `n` como o próximo nó da cadeia. O método `get_val(self)` retorna o valor armazenado no nó.

   Escreva uma função em Python que *remove* de uma lista ligada o seu *menor* valor. Presuma que todos os elementos da lista são *distintos*.

   Use a seguinte assinatura:

   ```
   def remove_menor(raiz):
   ```

   Onde `raiz` é o *primeiro* nó da lista ligada. Sua função deve retornar o primeiro nó da lista ligada após a remoção.

Resposta:

É necessário manter uma referência tanto ao nó a ser removido quanto ao nó anterior a este. Note o tratamento especial à remoção ao nó raiz e o cuidado de remover a referência ao próximo nó do nó a ser removido.

```
def remove_menor(raiz):
    pai_menor = None
    menor_no = raiz
    menor_val = raiz.get_val()
    pai = raiz
    no = raiz.get_next()
    while no:
```

```
        if menor_val < no.get_val():
            pai_menor = pai
            menor_val = no.get_val()
            menor_no = no
        pai = no
        no = no.get_next()
    if pai_menor != None:
        pai_menor.set_next(menor_no.get_next())
    else
        raiz = raiz.get_next()
    menor_no.set_next(None)
    return raiz
```

3. (2,5 pontos) Uma árvore AVL é uma árvore binária auto-balanceada que atende ao seguinte critério: A diferença de altura entre duas sub-árvores de cada nó é de no máximo 1.

   A classe `NoArvoreBinaria` implementa um nó de árvore binária. O campo `e` aponta para a sub-árvore esquerda, ou `None` se esta não existe. O campo `d` aponta para a sub-árvore direita, ou `None` se esta não existe.

   Escreva em Python uma função que verifica em tempo linear se uma determinada árvore atende ao critério AVL.

   Use a seguinte assinatura:

   `def verifica_avl(raiz):`

   Onde `raiz` é nó raiz da árvore a ser verificada. Sua função deve retornar `True` caso a árvore atenda o critério AVL ou `False` caso contrário. Sua função deve executar em tempo $\mathcal{O}(N)$, onde N é o total de nós da árvore.

Resposta: Normalmente uma árvore AVL armazena em cada nó a altura total de sua sub-árvore. Isso evita repetições da (custosa) operação de cálculo de altura. Não é o caso aqui. É necessário então calcular a altura da árvore e verificar a condição AVL simultaneamente. O código abaixo verifica a condição em cada nó em ordem posterior.

```
def verifica_avl(raiz);
    def verifica_avl_calc_altura(no):
        if no = None:
            return True, 0
        esq = verifica_avl_calc_altura(no.e)
        if esq[0]:
            dir = verifica_avl_calc_altura(no.e)
            if dir[0]:
                maior = max(esq[1],dir[1])
                menor = min(esq[1],dir[1])
                if maior - menor > 1:
                    return False
                return True, maior+1

    return verifica_avl_calc_altura(raiz)
```

4. (2,5 pontos) Em uma tabela de hash com encadeamento aberto linear, elementos são armazenados em um vetor na *primeira* posição disponível a partir da sua posição ideal, correspondente ao valor da função de hash aplicada à sua chave. O *deslocamento* de um elemento é definido como a diferença entre a posição ideal de um elemento e sua posição efetiva. Diz-se que um elemento a é *deslocado* por um elemento b quando a posição do elemento b está *entre* a posição efetiva do elemento a e sua posição ideal.

   A classe `HashLinear` implementa uma tabela de hash com encadeamento aberto linear. O campo `_e` é o vetor que armazena chaves e elementos em pares, nesta ordem (de modo que a chave armazenada na `i`-gésima posição é armazenada em `_e[i][0]` e o valor em `_e[i][1]`. O método `_calcula_indice(self, key)` retorna a posição ideal de armazenamento da chave `key`. O método `_deslocamento(self, i)` retorna o deslocamento do elemento armazenado na posição `i` do vetor `_e`. O método `adiciona_item(self, key, val)` adiciona à tabela o objeto apontado por `val` sob a chave indicada por `key`. O metodo `recupera_item(self, key)` retorna o objeto armazenado sob a chave `key`.

   Modifique o método `adiciona_item(self, key, val)` de forma a evitar que exista um elemento a deslocado por um elemento b na posição i tal que o deslocamento que a teria na posição i seja *maior* do que o de b na mesma posição. Desconsidere a possibilidade de remoções de elementos na tabela.

   Por exemplo:

Hash "bom"		
pos	chave	hash
0	A	0
1	B	0
2	C	0
3	D	2

Hash "ruim"		
pos	chave	hash
0	A	0
1	B	1
2	C	0
3	D	2

Hash "consertado"		
pos	chave	hash
0	A	0
1	C	0
2	B	1
3	D	2

   Neste exemplo, no Hash "bom", o deslocamento de A é 0, de B é 1, de C é 2 e de D é 1. O elemento B é deslocado pelo elemento A na posição 0, o elemento C é deslocado pelo elemento A na posição 0 e pelo B na posição e o elemento D é deslocado pelo elemento C na posição 2. Todos estes deslocamentos satisfazem a condição proposta, pois o deslocamento de B na posição 0 seria 0. O deslocamento de C na posição 0 seria 0 e na posição 1 seria 1. O deslocamento de D na posição 2 seria 0.

   A diferença do Hash "ruim" é que a posição ideal de B agora é a posição 1. Nesta situação, o deslocamento de C por B viola a condição, pois C teria um deslocamento de 1 na posição 1, que é maior que o deslocamento de B na mesma posição.

   O Hash "consertado" mostra a mesma situação do Hash "ruim", mas com uma nova ordem de elementos que satisfaz novamente a condição.

Resposta:

Basta trocar o elemento a ser inserido pelo elemento já presente quando o deslocamento do último for inferior ao deslocamento atual do primeiro.

```
def adiciona_item(self, key, val):
  """ Adiciona um novo valor sob a chave key """
  i = self._calcula_indice(key)
  deslocamento = 0
  while self._e[i] and self._e[i][0]!=key:
      deslocamento += 1
      if deslocamento > self.deslocamento[i]:
          deslocamento = self.deslocamento[i]+1
          temp = self._e[i]
```

```
                (self._e[i]) = (key, val)
                (key, val) = temp
            i = (i+1)%len(self._e)
        if self._e[i]:
            raise ValueError("Chave␣já␣existe")
        self._e[i] = key, value
        self._n += 1
        self._checkcount()
```

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 = 1 + random.randrange(HashLinear._p - 1)
        self._b = random.randrange(HashLinear._p)
        self._n = 0

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

    def adiciona_item(self, key, val):
        """ Adiciona um novo valor sob a chave key """
        i = self._calcula_indice(key)
        while self._e[i] and self._e[i][0]!=key:
            i = (i+1)%len(self._e)
        if self._e[i]:
            raise ValueError("Chave já existe")
        self._e[i] = key, value
        self._n += 1
        self._checkcount()

    def recupera_item(self, key):
        i = self._calcula_indice(key)
        while self._e[i] and self._e[i][0]!=key:
            i = (i+1)%len(self._e)
        if self._e[i]:
            return self._e[i][1]
        else:
            raise ValueError("Chave não existe!")

    def _deslocamento(self, i)
        """ Calcula o deslocamento do elemento armazenado na posição i """
        return (i - self._calcula_indice(self._e[0][0]))%len(self._e)

    def __len__(self):
        return self._n

    def _checkcount(self):
        """ Limita o número de entradas a 2/3"""
        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.adiciona_item(e[0]], e[1])
```

Classe `NoListaLigada`

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

    def get_next(self):
        """Retorna próximo nó na cadeia"""
        return self._n

    def set_next(self, n):
        """Modifica próximo nó na cadeia"""
        self._n = n
        return self

    def get_val(self):
        """Retorna valor armazenado"""
        return self._x
```

Classe `NoArvoreBinaria`

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