

**INSTITUTO POLITÉCNICO DE BEJA**

**kd-trees**

**Trabalho de Época Normal  
Estruturas de Dados e Algoritmos**



**Pedro Miguel Clemente Dias Moreira**

**n.º 10015**

**17 de março 2012**



# Conteúdo

|  |           |
|--|-----------|
| Índice Geral . . . . .                 | I         |
| <b>1 Introdução</b>                    | <b>3</b>  |
| <b>2 Teoria</b>                        | <b>5</b>  |
| <b>3 Parte Experimental</b>            | <b>7</b>  |
| 3.1 Realização Experimental . . . . .  | 7         |
| 3.2 Sistema Experimental . . . . .     | 7         |
| 3.3 Resultados Experimentais . . . . . | 10        |
| <b>4 Conclusão</b>                     | <b>15</b> |
| <b>5 Códigos</b>                       | <b>17</b> |
| <b>6 Bibliografia</b>                  | <b>31</b> |



# 1 Introdução

## Contexto

Este trabalho é uma componente lectiva da disciplina Estruturas de Dados e Algoritmos que abrange o estudo da organização e armazenamento de informação de modo a que esta seja utilizada eficientemente.

Nele é abordado um tipo de representação hierárquica de dados, nomeadamente uma árvore k-d que tem como principal característica a organização de dados num espaço k-dimensional.

## Objectivos

Foi proposta a criação de uma biblioteca para operação com árvores k-d balanceadas que permita, pelo menos:

- a construção da árvore a partir de uma lista de dados
- a junção de elementos
- a remoção de elementos
- a pesquisa do elemento mais próximo

A árvore deve ser balanceada e ser construída sobre uma representação da memória por listas ligadas

## Estrutura do relatório

Este relatório apresenta uma introdução sobre o tema abordado no trabalho, o código efectuado com as partes mais relevantes explicadas, os resultados experimentais, qual o protocolo utilizado para os obter e, finalmente, as conclusões retiradas dos mesmos.



## 2 Teoria

As árvores k-d são estruturas de dados de partição do espaço destinadas a organizar pontos num espaço k-dimensional. As mesmas são utilizadas em várias aplicações, nomeadamente pesquisas envolvendo uma chave multidimensional.

A biblioteca envolvida neste trabalho deve seguir o modelo de representação de memória através de listas ligadas, em que cada nó tem a sua representação na memória através de um ponteiro para uma lista de tamanho pré-definido.

A árvore deve estar balanceada o que, na prática, significa que não deve existir uma grande diferença na profundidade dos nós limítrofes da mesma (folhas).

A ordenação dos dados é feita pela comparação da chave dos nós, alterando o eixo de comparação a cada mudança de nível da árvore.

Quando o eixo a comparar é igual a mesma é feita no eixo seguinte. No caso de a chave ser igual a alguma existente na árvore apenas é actualizado o valor desse nó pelo novo.





## 3 Parte Experimental

### 3.1 Realização Experimental

A linguagem de programação escolhida para a elaboração do trabalho foi Python. O código foi desenvolvido com auxílio da ferramenta Sublime instalado num sistema operativo Windows 7 (64-bit). O computador utilizado para o desenvolvimento e aplicação dos testes foi:

- Marca: Insys
- Processador: AMD Athlon TF-20 1.60GHz
- Ram: 3 GB

### 3.2 Sistema Experimental

#### Class No

Classe que representa um Nó que será inserido na árvore

#### Atributos

- key: Chave nó
- valor: Valor que o nó terá
- LC: contador dos nós sucessores localizados à esquerda do mesmo
- RC: contador dos nós sucessores localizados à direita do mesmo
- dim: última dimensão pela qual o nó foi inserido (parte da chave utilizada na ordenação)
- parent: nó antecessor
- left: nó localizado à direita
- right: nó localizado à esquerda

### Métodos

- str: Override do método str, para representação personalizada do nó

### Class Quicksort

Classe utilizada para a ordenação dos dados, neste caso foi optada a utilização do randomized quicksort que tem sempre como tempo esperado  $O(n \log n)$ .

### Atributos

- A: Lista a ser ordenada
- dim: dimensão da chave que será utilizada como base para a ordenação

### Métodos

- init: Construtor que recebe uma lista, e a chave de ordenação que por defeito fica definida como o
- sort: método inicial para ordenar a lista, recebe como argumentos uma lista (A), o ponto inicial para se efectuar a ordenação (p), e o ponto final para a ordenação (q), este método dá início à ordenação chamando o método randomizedPartition
- randomizedPartition: para garantir que a lista não esta ordenada de forma decrescente e garantir o tempo de execução  $O(n \log n)$  este método escolhe uma chave aleatória e troca-a com a final para que seja feita a ordenação, através do chamamento do método partition
- partition: método que ordena uma subpartição
- randomizedQuicksort: método recursivo que vai dividindo a lista em subpartições e ordenando-o com auxílio dos outros métodos até que fique totalmente ordenada.

### Class KDTree

Classe principal onde é criada, armazenada e manipulada a árvore.

### Atributos

- root: valor que vai representar a raiz da árvore
- nil: Nó vazio (sentinela), para onde vão apontar os nós que não tenha sucessores ou antecessores
- dimension: número de dimensões da árvore (K)

- stack: Pilha auxiliar utilizada para o rebalanceamento da árvore
- lista: lista auxiliar utilizada para o rebalanceamento da árvore
- free: lista de slots disponíveis para o armazenamento de nós
- pointers: lista de slots ocupadas por nós

## Métodos

- malloc: Método utilizado para a alocação de um nó na árvore, começa por verificar a existência de slots livres e, caso exista, ocupa uma slot com o respectivo nó
- freeNo: Libertar uma slot removendo o apontador para o nó e marcando a slot como disponível para um novo objecto. Após esta operação dá início ao processo da sua remoção da árvore posterior balanceamento.
- item insert: Método para a inserção de um nó na árvore. Apesar o nó para onde será iniciada a inserção do mesmo, o que pouparia algum tempo de inserção *logn* o mesmo não é utilizado neste trabalho. Após a conclusão da inserção o método dá início ao processo de confirmação do balanceamento da árvore. Esta opção pode ser desactivada chamando o método com o atributo *balance* marcado a False. Cada nó por onde o que vai ser inserido passar será actualizado o seu contador LC ou RC de acordo com a direcção por este tomada. Assim mantém-se actualizada a contagem dos nós sucessores de cada um.
- clear: Conta os nós sucessores somando os atributos LC e RC e vai retirando o valor aos nós antecessores do mesmo. No final coloca todos os nós sucessores na pilha auxiliar para posterior reinserção na árvore.
- inorderWalk: Percorre toda a árvore e coloca os nós de forma ordenada na lista fornecida
- reInsert: Método utilizado para a reinserção de nós colocados na pilha auxiliar, seja por motivos de balanceamento ou por alguma operação de remoção. A forma de operar é retirar a primeira lista da pilha, encontrar o elemento do meio, dividir a lista nesse local e voltar a inserir na pilha as duas partes resultantes de tal divisão, então insere o nó extraído do meio de novo na árvore, sem conferir o balanceamento e volta a chamar-se enquanto houver elementos na pilha. Só então dá início à verificação do balanceamento.
- nearestNeighbour: Percorre todos os nós da árvore e calcula a distância euclidiana

$$\sqrt{\sum_{i=0}^k (a.key[k] - b.key[k])^2} \quad (3.1)$$

(raiz quadrada do somatório dos quadrados das diferenças entre os pontos das várias dimensões das chaves dos nós) para o nó fornecido, guardando sempre a distância mais curta, no final retorna o nó que teve o resultado mais baixo.

- isBalanced: Verificação se o número de nós sucessores para cada lado, está de acordo com os limites para o balanceamento. Assumindo que para estar balanceada as folhas não poderão ter mais do que um nível de profundidade entre si e sabendo que cada nível difere o número de elementos por potências de dois dignifica que o lado com menor número de elementos não poderá ter menos que o valor da potência do nível inferior ao do lado com maior número de elementos:

$$\min(a,b) < 2 * \text{math.pow}(\text{int}(\text{math.floor}(\log(\max(a,b)))) - 1 \quad (3.2)$$

- checkBalance: Percorre todos os elementos da árvore em largura e verifica se os nós estão com valores válidos para uma árvore balanceada, com o auxílio do método isBalanced, caso não esteja balanceado, esse nó é retirado da árvore e os seus sucessores, a lista resultante é ordenada pela dimensão do nó com o problema e a lista é colocada na pilha auxiliar para serem reinseridos na árvore.
- delete: eliminar um nó colocando os seus sucessores na pilha auxiliar, chamando então o método para a reinserção dos nós e, por fim, confirmação do balanceamento

## 3.3 Resultados Experimentais

### Protocolo Experimental

A forma utilizada para testar os dados consiste na criação de listas de nós em quantidades de 1 até 1000 com incrementos de 100. Para cada lista criada o método é testado 40 vezes e são somados os tempos das operações. No final o resultado é dividido pelo número de testes efectuados (40) e guardado o valor para posterior desenho do gráfico.

### Apresentação e Discussão dos Resultados

#### Inserção de Elementos

Como é mostrado abaixo a inserção de elementos é aproximadamente  $O(n * \log(n^2))$  Este resultado acontece devido ao balanceamento da árvore em que pode ser retirada uma parte da árvore para ser reinserida, o que pode tornar a operação muti lenta, ao contrário de inserção sem balanceamento que levaria a  $O(\log(n))$ .

#### Pesquisa do elemento mais próximo

A pesquisa do elemento mais próximo tem uma complexidade  $O(n)$  pois são sempre percorridos todos os elementos da lista para se encontrar o elemento mais próximo

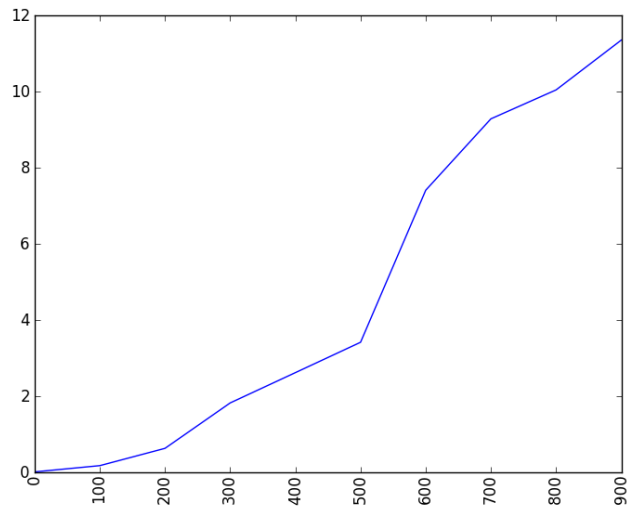


Figura 3.1: Inserção de dados.

| N   | tempo de execução (seg) |
|-----|-------------------------|
| 0   | 0.0                     |
| 100 | 0.163800001144          |
| 200 | 0.61779999733           |
| 300 | 1.81072499752           |
| 400 | 2.60599999428           |
| 500 | 3.40314999819           |
| 600 | 7.39674999714           |
| 700 | 9.27110001445           |
| 800 | 10.030825007            |
| 900 | 11.3387999892           |

Tabela 3.1: inserção de dados.

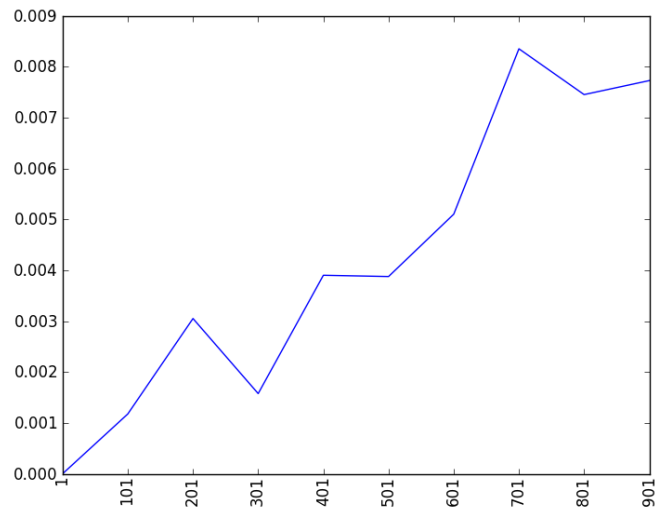


Figura 3.2: Pesquisa do elemento mais próximo.

| N   | tempo de execução (seg) |
|-----|-------------------------|
| 1   | 0.0                     |
| 101 | 0.00117500424385        |
| 201 | 0.00304999947548        |
| 301 | 0.00157500505447        |
| 401 | 0.0038999915123         |
| 501 | 0.00387500524521        |
| 601 | 0.00510001182556        |
| 701 | 0.00835000872612        |
| 801 | 0.00745000243187        |
| 901 | 0.00772499442101        |

Tabela 3.2: Pesquisa do Elemento mais próximo.

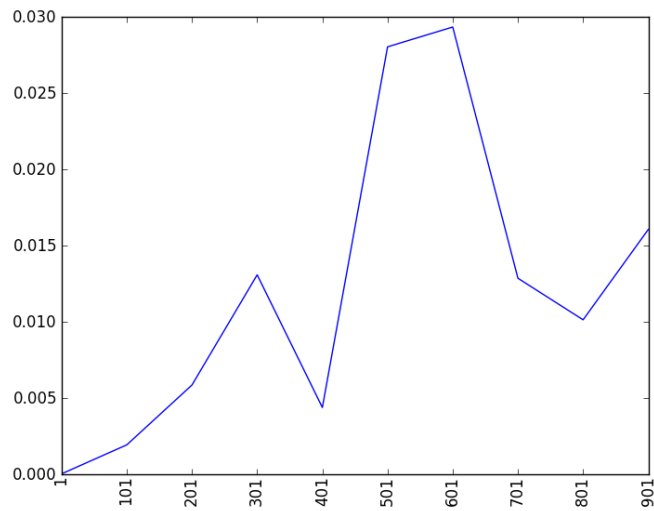


Figura 3.3: Remocao de um elemento.

| N   | tempo de execução (seg) |
|-----|-------------------------|
| 1   | 0.0                     |
| 101 | 0.001900000534058       |
| 201 | 0.00582498311996        |
| 301 | 0.0130500078201         |
| 401 | 0.00434999465942        |
| 501 | 0.027999997139          |
| 601 | 0.0292999744415         |
| 701 | 0.0128249943256         |
| 801 | 0.0101000010967         |
| 901 | 0.0160250008106         |

Tabela 3.3: Remoção de um elemento.

### Remoção de um elemento

A complexidade para a remoção de um elemento, como se pode verificar no gráfico (ignorando os valores mais distantes) é de  $O(n * \log(n))$ .





## 4 Conclusão

O trabalho deveria ter sido elaborado com base no tipo Adaptive Kd-Tree. Pois julgo que seria mais simples efectuar as operações pretendidas. Fica essa importante alteração para trabalho futuro.

A parte da pesquisa do elemento mais próximo também pode ser bastante melhorada baseando-se na pesquisa em profundidade e em largura, tirando partido da alteração para uma Adaptive Kd-Tree.

Nesta parte assumi que a pesquisa é feita pela distância do ponto no espaço a partir das suas coordenadas e não a distância dentro da árvore.

Os gráficos também precisam ser melhorados, efectuando testes com valores mais elevados e obtendo a linha do gráfico uniforme, utilizando por exemplo o método dos mínimos quadrados.

Outra parte a melhorar neste trabalho é o relatório, que apesar de ter sido feito em latex, não aproveita grande parte das suas potencialidades, como por exemplo a bibliografia.



## 5 Códigos

### Classe No

```
1  #!/usr/bin/env python
2  # -*- coding: utf-8 -*-
3  '''
4  autor: Pedro Moreira, 10015
5  data: 17 de Junho de 2012
6  '''
7
8
9  class No(object):
10     '''
11     Classe dos nos de cada elemento
12     '''
13     def __init__(self, key, valor):
14         """
15         Criar um no para utilizacao em RB Trees
16         @param key, chave do no
17         @param valor, valor que o no tem guardado
18         """
19         self.key = key
20         self.valor = valor
21         self.LC = 0
22         self.RC = 0
23         self.dim = 0
24
25         # Ao ser criado o no fica com os apontadores para si proprio (NIL)
26         self.parent = self.left = self.right = None
27         pass
28
29     def __str__(self):
30         """
31         Override do metodo str
32         Impressao personalizada do no
33         """
34         s = str(self.key) + " : " + str(self.valor)
35         s += '('
36         s += str(self.parent.key) + ', '
37         s += str(self.left.key) + ', '
38         s += str(self.right.key)
39         s += ', ' + str(self.size) + ')'
40         s += ', LC = ' + str(self.LC)
```

## 5 Códigos

---

```
41         s += ', RC = ' + str(self.RC)
42         s += ', dim = ' + str(self.dim)
43     return s
```

No.py

---

## Class QuickSort

```
1 #!/usr/bin/env python
2  #-*- coding: utf-8 -*-
3  ,,,
4
5  autor: Pedro Moreira, 10015
6  data: 31 de maio de 2012
7
8  algoritmo quicksort
9  ,,,
10
11 import random
12 class QuickSort():
13     ,,,
14     o que a classe faz...
15     ,,,
16
17     def __init__(self, A = [], dim = 0):
18         ,,,
19         construtor, pode inicializar
20         uma lista de dados e ordena-la
21         ,,,
22         self.A = A
23         self.dim = dim
24         if len(A) > 1:
25             self.sort(self.A, 0, len(self.A) - 1)
26
27     #####
28     # ORDENAR A LISTA
29
30     def sort(self, A, p, q):
31         ,,,
32         ordenar lista
33         ,,,
34         #utilizando quicksort
35         self.__randomizedQuickSort__(A, p, q)
36         pass
37
38     #####
39     # livro pag 177
40     # RANDOMIZED PARTITION
41
42     def __randomizedPartition__(self, A, p, r):
43         i = random.randint(p, r)
44         A[r], A[i] = A[i], A[r]
45         return self.__partition__(A, p, r)
46     # FIM DO RANDOMIZED PARTITION
47     #####
48
49     #####
50     # livro pag 169
51     # PARTITION
```

```

51 def __partition__(self, A, p, r):
52     x = A[r].key[self.dim:]
53     i = p - 1
54     for j in range(p, r):
55         if A[j].key[self.dim:] <= x:
56             i += 1
57             A[i], A[j] = A[j], A[i]
58         pass
59     pass
60     A[i + 1], A[r] = A[r], A[i + 1]
61     return i + 1
62 # FIM DO PARTITION
63 #####
64
65
66 #####
67 # livro pag 177
68 # RANDOMIZED QUICKSORT
69
70 def __randomizedQuicksort__(self, A, p, r):
71     if p < r:
72         q = self.__randomizedPartition__(A, p, r)
73         #q = self.__partition__(A, p, r)
74         self.__randomizedQuicksort__(A, p, q - 1)
75         self.__randomizedQuicksort__(A, q + 1, r)
76     pass
77     pass
78 # FIM DO RANDOMIZED QUICKSORT
79 #####

```

Quicksort.py

---

## Class KDTree

```
#!/usr/bin/env python
# -*- coding: utf-8 -*-
'''
4 autor: Pedro Moreira, 10015
data: 16 de Junho de 2012
6
kd-tree
8 operacoes:
- criar arvore a partir de uma lista
10 - inserir
- apagar
12 - pesquisa do no mais proximo
14
16
18 USAGE:
criar uma arvore para 1000 nos e 3 dimensoes na chave:
20 arvore = KDTree(1000, 3)
22
inserir um no na arvore
no = No((0,0,0), "valor")
24 arvore.malloc(no)
26
procurar o no mais proximo
maisProximo = arvore.maisProximo(no)
28
eliminar um no
30 arvore.freeNo(no)
32
'''
34 from Quicksort import *
from No import *
36 import math
import sys
38
class KDTree(object):
    '''
40 Estrutura de dados kdtree
    '''
42
    def __init__(self, size, dimention):
        '''
44 Construtor
46 @param dimention dimensao da chave para ordenacao dos dados
@ param size tamanho da arvore
48 '''
        #sys.setrecursionlimit(10000)
50 self.root = self.nil = No(None, None)
```

```
self.dimension = dimension
52 self.stack = []
self.lista = []
54 self.free = [i for i in range(size)]
self.pointers = [None for i in range(size)]
56
def malloc(self, no):
58     """
    Tentar a alocação de um No na árvore
    @param no Objecto a ser inserido na árvore
    @return -1 em caso de erro ou o no já inserido na árvore
    """
    62     if len(self.free) > 0:
    64         x = self.free.pop()
        self.pointers[x] = no.key
    66         no.pointer = x
        self.insert(self.root, no)
    68         return no
    else:
    70         print "out of space"
        return -1
    72
def freeNo(self, x):
    74     """
    Eliminar no da árvore
    @param x no a Eliminar
    @return -1 caso o no não exista ou o se a operação correr com sucesso
    """
    78     if self.pointers[x.pointer] != None:
    80         self.pointers[x.pointer] == None
        self.free.append(x.pointer)
    82         self.delete(x)
        return 0
    84     else:
        return -1
    86
def insert(self, a, z, balance = True):
    88     """
    Inserir no na árvore
    @param a local a inserir o no
    @param z no a inserir
    @param balance informação para balancear ou não a árvore (true por
    90         defeito)
    """
    94     z.parent = self.nil
    z.left = self.nil
    96     z.right = self.nil
    z.LC = z.RC = 0
    98
    y = self.nil
    100     x = a
    dim = -1
```



---

```

102 while x != self.nil:
104     dim = (dim + 1) % self.dimension
106     if x.key == z.key:
108         x.valor = z.valor
110         break
112
114     z.dim = dim
116     y = x
118     if z.key[dim:] < x.key[dim:]:
120         x.LC += 1
122         x = x.left
124     else:
126         x.RC += 1
128         x = x.right
130     z.parent = y
132     if y == self.nil:
134         self.root = z
136     elif z.key[dim:] < y.key[dim:]:
138         y.left = z
140     else:
142         y.right = z
144
146     if balance:
148         self.checkBalance()
150
152 def __clear(self, x):
154     """
156     Contar os nos sucessores ao no pretendido
158     e percorrer todos os seus antecessores retirando a respectiva contagem
160     @param x no a partir do qual sera para limpar as contagens
162     """
164     if x == self.root:
166         self.root = self.nil
168     else:
170         z = x
172         i = x.RC + x.LC + 1
174         while (z != self.nil):
176             if z.parent.left == z:
178                 z.parent.LC -= i
180
182             else:
184                 z.parent.RC -= i
186                 z = z.parent
188
190         if x.parent.left == x:
192             x.parent.left = self.nil
194         else:
196             x.parent.right = self.nil
198
199 def inorderWalk(self, x, lista):
200     """

```

```
154     Percorrer a arvore devolvendo uma lista ordenada com os nos
155     @param x no a partir do qual se constroi a lista
156     @param lista para guardar os dados
157     """
158     if x != self.nil:
159         self.inorderWalk(x.left, lista)
160         lista.append(x)
161         self.inorderWalk(x.right, lista)
162
163     def __reInsert(self):
164         """
165         Metodo para voltar a inserir na arvore nos
166         que tenham sido retirados por motivos de balanceamento
167         """
168         if len(self.stack) > 0:
169             a = self.stack.pop()
170             if len(a) == 1:
171                 self.insert(self.root, a.pop(), False)
172             elif len(a) == 0:
173                 pass
174             else:
175                 k = int(math.floor(len(a)/2))
176                 x = a.pop(k)
177                 x.parent = x.left = x.right = None
178                 self.insert(self.root, x, False)
179                 b = a[:k]
180                 if len(b) > 0:
181                     self.stack.append(b)
182                 b = a[k:]
183                 if len(b) > 0:
184                     self.stack.append(b)
185             if len(self.stack) > 0:
186                 self.__reInsert()
187         else:
188             self.checkBalance()
189
190     def nearestNeighbour(self, no):
191         """
192         Percorre todos os elementos da arvore e calcula a distancia eucladiana
193         devolve o no com menor distancia
194         @param no No que pretendemos fazer a pesquisa
195         @return No mais proximo do fornecido
196         """
197         lista = []
198         self.inorderWalk(self.root, lista)
199         distancia = sys.maxint
200         x = self.nil
201         for i in lista:
202             if no != i:
203                 temp = 0
204                 for k in range(self.dimension):
```

---

```

206         temp += (i.key[k] - no.key[k])**2
207         temp = math.sqrt(temp)
208         if distancia > temp:
209             distancia = temp
210             x = i
211         return x
212
213 def isBalanced(self, x):
214     """
215     verificar se o numero de sucessores para cada lado
216     do no em causa corresponde a dois ramos com a mesma altura
217     @param x no a analisar
218     """
219     a = x.LC
220     b = x.RC
221     if (a+b) < 2 : return True
222     if (a+b) <= 4 and min(a,b) == 1: return True
223     return min(a, b) > 2**(int(math.floor(math.log(max(a,b),2)))-1)
224
225 def checkBalance(self):
226     """
227     percorrer todos os nos para
228     verificar se a arvore esta balanceada
229     caso existam problemas, os nos sao retirados
230     e ordenados para serem reinseridos na arvore
231     """
232     if self.root == self.nil:
233         return
234     stack = []
235     stack.append(self.root)
236     while len(stack) > 0:
237         x = stack.pop(0)
238         if x.left != self.nil : stack.append(x.left)
239         if x.right != self.nil : stack.append(x.right)
240
241         if self.isBalanced(x): continue
242         parent = x.parent
243         dim = x.dim
244         self.__clear(x)
245         lista = []
246         self.inorderWalk(x, lista)
247         Quicksort(lista, (dim + 1) % self.dimension)
248         self.stack.append(lista)
249         self.__reInsert()
250     pass
251
252 def delete(self, z):
253     """
254     Eliminar um no e colocar todos os seus sucessores
255     na lista para serem reinseridos na arvore
256     @param z no a ser eliminado

```

```
258     '''
259     if z.parent != self.nil:
260         if z.parent.left == z.key:
261             z.parent.left = self.nil
262             z.parent.LC = o
263         else:
264             z.parent.right = self.nil
265             z.parent.RC = o
266     lista = []
267     self.inorderWalk(z.left, lista)
268     self.inorderWalk(z.right, lista)
269     self.stack.append(lista)
270
271     z.parent = z.left = z.right = self.nil
272     z.LC = z.RC = o
273     self.__reInsert()
274     self.checkBalance()
275     pass
```

KDTree.py

---

## Testes ao desempenho computacional

```
1  #!/usr/bin/env python
2  # -*- coding: utf-8 -*-
3  from KDTree import *
4  from random import randint as rand
5  import time
6  from pylab import *
7
8
9  def drawGraph(ys, labels, title):
10     a = lambda: range(1, len(ys) + 1)
11     plot(a(), ys)
12     locs, labels = xticks(a(), labels)
13     setp(labels, 'rotation', 'vertical')
14     subtitle(title, fontsize=20)
15
16     #x1,x2,y1,y2 = axis()
17     #axis((x1,x2,0,1))
18     show()
19     pass
20
21
22 def inserirDados(lista, x):
23     for i in range(x):
24         lista.append((randint(0,1000),randint(0,1000)), randint(0, 1000000))
25
26 def construirArvore(arvore, lista):
27     #Quicksort(lista)
28     for i in lista:
29         arvore.malloc(i)
30
31 def inserir():
32
33     maxN = 1000
34     startN = 0
35     increment = 100
36     N = 40
37     dados = []
38     labels = []
39     f1 = open('labels.dat', 'w')
40     f2 = open('dados.dat', 'w')
41     for k in range(startN, maxN, increment):
42         b = 0.0
43         for j in range(N):
44             lista = []
45             inserirDados(lista, k)
46             arv = KDTree(k, 2)
47             t1 = time.time()
48             construirArvore(arv, lista)
49             t2 = time.time()
50             b += t2-t1
```

```
51     f1.write(str(k))
52     f2.write(str(b/N))
53     print k, (b/N)
54     dados.append(b/N)
55     labels.append(k)
56
57     f1.close()
58     f2.close()
59
60     drawGraph(dados, labels, "titulo")
61
62 def apagar():
63     maxN = 1000
64     startN = 1
65     increment = 100
66     N = 40
67     dados = []
68     labels = []
69     f1 = open('labels.dat', 'w')
70     f2 = open('dados.dat', 'w')
71     for k in range(startN, maxN, increment):
72         b = 0.0
73         for j in range(N):
74             lista = []
75             inserirDados(lista, k)
76             arv = KDTree(k, 2)
77             construirArvore(arv, lista)
78             t1 = time.time()
79             arv.freeNo(lista[randint(0, len(lista))])
80             t2 = time.time()
81             b += t2 - t1
82         f1.write(str(k))
83         f2.write(str(b/N))
84         print k, (b/N)
85         dados.append(b/N)
86         labels.append(k)
87
88     f1.close()
89     f2.close()
90
91     drawGraph(dados, labels, "titulo")
92
93 def maisProximo():
94     maxN = 1000
95     startN = 1
96     increment = 100
97     N = 40
98     dados = []
```

---

```

103 labels = []
104 f1 = open('labels.dat', 'w')
105 f2 = open('dados.dat', 'w')
106 for k in range(startN, maxN, increment):
107     b = 0.0
108     for j in range(N):
109         lista = []
110         inserirDados(lista, k)
111         arv = KDTree(k, 2)
112         construirArvore(arv, lista)
113         t1 = time.time()
114         arv.nearestNeighbour(lista[randint(0, len(lista))])
115         t2 = time.time()
116         b += t2 - t1
117     f1.write(str(k))
118     f2.write(str(b/N))
119     print k, (b/N)
120     dados.append(b/N)
121     labels.append(k)
122
123 f1.close()
124 f2.close()
125
126
127 drawGraph(dados, labels, "titulo")
128
129 #TESTES
130 print "INSERIR"
131 inserir()
132 print
133 print "APAGAR"
134 apagar()
135 print
136 print "MAIS PROXIMO"
137 maisProximo()

```

teste.py





## 6 Bibliografia

- <sup>1</sup> Cormen Thomas H., et all, Introduction to Algorithms, The MIT Press, 3 edition, 2009
- <sup>2</sup> k-d tree, wikipedia, available at [http://en.wikipedia.org/wiki/K-d\\_tree](http://en.wikipedia.org/wiki/K-d_tree), visited 17/06/2012
- <sup>3</sup> Leonardo Rodriguez Heredia, Cirano Iochpe e João Comba, Explorando a Multidimensionalidade da Kd-Tree para Suporte a Temporalidade em Dados Espaciais Vetoriais do Tipo Ponto, available at <http://www.inf.ufrgs.br/~comba/papers/2003/geo-info.pdf>, visited 17/06/2012