Trabalho Computacional 2
Algoritmos e Estruturas de Dados II

Davi Juliano Ferreira Alves
Instituto de Ciência e Tecnologia
Bacharelado em Ciência e Tecnologia
Universidade Federal de São Paulo
Av. Cesare Monsueto Giulio Lattes, 1201, 12247-014
Email: davi.juliano@unifesp.br

***Abstract*—A manipulação de dados vem-se ficando mais e mais requisitada por profissionais da área das tecnologias. Uma das áreas mais requisitadas na manipulação de dados é justamente a área de manipulação de imagens. Há dois tipos classificação de dados: os ordenados e os não ordenados, ou seja, parte dos dados são refinados e ordenados, enquanto outra parte é desordenada e não refinada. Além da ordenação, existem diversos tipos de estruturas de dados que geram uma otimalidade no tempo para cada tipo de aplicação necessária. Nesse trabalho, utilizaremos a estrutura de dados da Árvore B e suas funções para a manipulação de um banco de imagens dado pelo autor.**

I. Introdução

Um dos fatos mais gritantes e atenuantes é a realidade da grande massa de informações que precisam ser organizadas e manipuladas. Após a organização e a manipulação dos dados de acordo com uma técnica de refinamento, obtemos um tipo de dado chamado dado estruturado. De acordo com [1], os dados estruturados são aqueles organizados e representados com uma estrutura rígida, a qual foi previamente planejada para armazená-los, por exemplo um banco de dados, que é a representação mais típica e comum de dados estruturados. Em um banco de dados, os dados são estruturados conforme a definição de um esquema, que define como as tabelas e suas respectivas linhas e colunas serão armazenadas. Podemos conceituar o esquema de um banco de dados como sendo uma descrição sobre uma organização, ou sobre o minimundo que se deseja representar, definindo quais dados que serão armazenados.

Fig. 1. Logo da Google©, a maior representante de sites de busca da Internet

Uma das aplicações mais ilustrativas do sistema de bancos de dados é o sistema de banco de imagens, como é feito pela própria Google©. As estruturas de dados utilizadas para melhor performance e para melhor orientar cada usuário do banco de imagens é um dos fatores de otimalidade e de bom rendimento do serviço de dados oferecido. Para que possa ser dado essa certa otimalidade no serviço de banco de imagens, há a utilização de diversos algoritmos de otimização de tempo de busca e etc. No trabalho sobre grandes bases de dados nos bancos de imagens, foi-se utilizado uma estrutura de dados propícia para que haja uma otimalidade na busca e inserção de imagens. O banco de imagens, que foi adquirido em sua maioria por [2] e pelo já citado Google Images©, é dado pelo link https://drive.google.com/drive/folders/1WhRd9uuokJkXgr09uybV2DOUkE6a6e0m?usp=sharing. Nele também está o link para o Colab que será o berço da manipulação dos dados pela Árvore B.

A. Árvore B

De acordo com [3], a motivação da Árvore B é de que:

- Ela é usada quando parte das chaves devem ser armazenadas num dispositivo de armazenamento secundário de acesso em blocos.
- Cada nó da árvore deve ocupar exatamente um bloco (setor) do dispositivo secundário.
- Nós podem conter um número variável de chaves.

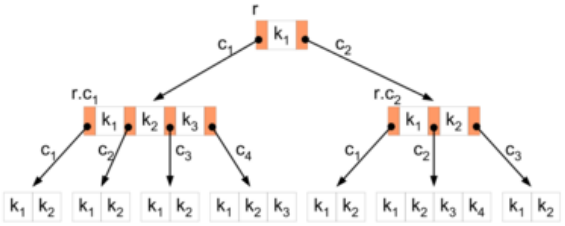

Fig. 2. Exemplo da estrutura da Árvore B.

Definição 1. *Seja $d \in Z$, tal que $d \geq 2$. Uma árvore é dita árvore B de ordem d se possui as seguintes propriedades:*

1) *Todo nó tem no mínimo d e no máximo 2d chaves. (Com exceção da raiz que tem de 0 a 2d chaves).*
2) *Nós não folha, com x chaves, tem exatamente x+1 filhos. (Em cada nó, ponteiros e chaves se intercalam, numa sequência que começa e termina com um ponteiro. Cada chave, de nó não folha, possui um filho direito e um filho esquerdo).*
3) *Dada uma chave c temos que: chaves à esquerda de c (tanto dentro do nó quanto no filho esquerdo) devem*

ser menores que c e chaves à direita de c devem ser maiores que c. (Propriedade análoga àquela que define uma ABB).

4) *Todas as folhas devem estar num mesmo nível.*

A complexidade da estrutura de dados é dada por O(log(n)), que é uma estrutura mais veloz do que a estrutura linear O(n). A complexidade da estrutura de dados ser dada em uma complexidade menor do que a linear se dá à definição da mesma, pois a altura da árvore é a quantidade de iterações que a árvore será percorrida, e a altura da árvore B é sempre a menor possível, visto que ela é auto-balanceada e também possui chaves que aceitam diversos dados distintos.

II. Definição dos Algoritmos Utilizados

O algoritmo computacional utilizado foi dado por [4] que está com o link de acesso nas referências. O código da árvore B é dado por:

```
"""
author = Mateor
referência(github) = https://gist.github.com/mateor/885eb950df7231f178a5
Sujeito a algumas alterações por Davi Juliano para adaptação ao projeto
PYTHON 3.3.5
"""

from __future__ import (nested_scopes, generators, division, absolute_import, with_statement,
                        print_function, unicode_literals)


class BTree(object):
  """A implementação da Árvore B com funções de busca e inserção. Capaz para qualquer ordem t"""

  class Node(object):
    """Um simples nó da Árvore B."""

    def __init__(self, t):
      self.keys = []
      self.children = []
      self.leaf = True
      # t é a ordem de parentesco da Árvore B. Nós precisam ter esse valor para definir tamanho máximo e splitting.
      self._t = t

    def split(self, parent, payload):
      """Splita um nó e reatribui chaves/filhos."""
      new_node = self.__class__(self._t)

      mid_point = self.size//2
      split_value = self.keys[mid_point]
      parent.add_key(split_value)

      # Adiciona chaves e filhos para o nó apropriado
      new_node.children = self.children[mid_point + 1:]
      self.children = self.children[:mid_point + 1]
      new_node.keys = self.keys[mid_point+1:]
      self.keys = self.keys[:mid_point]

      # Se o novo nó é filho, ele faz "seta" como nó interno
      if len(new_node.children) > 0:
        new_node.leaf = False

      parent.children = parent.add_child(new_node)
      if payload < split_value:
        return self
      else:
        return new_node

    @property
    def _is_full(self):
      return self.size == 2 * self._t - 1

    @property
    def size(self):
      return len(self.keys)

    def add_key(self, value):
      """Adiciona uma chave ao nó. Por definição, o nó terá espaço para a chave."""
      self.keys.append(value)
      self.keys.sort()

    def add_child(self, new_node):
      """
      Adiciona um filho a um nó. Isso classificará os filhos do nó, permitindo que os filhos
      sejam ordenados mesmo após a divisão dos nós do meio.
      Retorna uma lista de ordem de nós filhos
      """
      i = len(self.children) - 1
      while i >= 0 and self.children[i].keys[0] > new_node.keys[0]:
        i -= 1
      return self.children[:i + 1]+ [new_node] + self.children[i + 1:]


  def __init__(self, t):
    """
    Cria a Árvore B. t é a ordem da árvore. A árvore não tem chaves quando criada.
    Esta implementação permite valores-chave duplicados, embora isso não tenha sido verificado
    vigorosamente.
    """
    self._t = t
    if self._t <= 1:
      raise ValueError("A Árvore B precisa ter ordem 2 ou mais.")
    self.root = self.Node(t)

  def insert(self, payload):
    """Insere uma nova chave de valor na Árvore B.
    """
    node = self.root
    # A raiz é tratada explicitamente, pois requer a criação de 2 novos nós em vez do usual.
    if node._is_full:
      new_root = self.Node(self._t)
      new_root.children.append(self.root)
      new_root.leaf = False
      #o nó está sendo definido como o nó que contém os intervalos que desejamos para a inserção (Devido à ordem dos elementos)
      node = node.split(new_root, payload)
      self.root = new_root
    while not node.leaf:
      i = node.size - 1
      while i > 0 and payload < node.keys[i] :
        i -= 1
      if payload > node.keys[i]:
        i += 1
```

```
      next = node.children[i]
      if next._is_full:
        node = next.split(node, payload)
      else:
        node = next
    #Desde que nós splitamos todos os nós cheios em
    novos nós, nós podemos simplesmente inserir os
    valores na folha.
    node.add_key(payload)

  def search(self, value, node=None):
    """Retorna 1 se a Árvore B contiver uma chave
    que corresponda ao valor e 0 no contrário."""
    if node is None:
      node = self.root
    if value in node.keys:
      return 1
    elif node.leaf:
      #Se estivermos em uma folha, não há mais o que
     verificar.
      return 0
    else:
      i = 0
      while i < node.size and value > node.keys[i]:
        i += 1
      return self.search(value, node.children[i])

  def print_order(self):
    """Imprima uma representação de ordem de nível.
    """
    this_level = [self.root]
    while this_level:
      next_level = []
      output = ""
      for node in this_level:
        if node.children:
          next_level.extend(node.children)
        output += str(node.keys) + " "
      print(output)
      this_level = next_level
```

Já definida a estrutura de dados, iremos definir as funções interessantes para que a estrutura de dados seja preenchida de acordo com um certo catálogo. Note que a inserção é dada por meio de catálogos, ou seja, serão adicionadas 5 fotos de cada catálogo na árvore. As funções são dadas por:

```
def insere_catalogo(arvore, string):
  for i in range(1,6):
    arvore.insert(string+str(i))

def printa_imagem(string):
  display(Image(string+'.jpg'))

def procura_nome_na_Arv_B(arvore, string):
  a = arvore.search(string)
  if a == 1:
    print("O nó", string , "foi encontrado\n")
    printa_imagem(string)
  else:
    print("O nó", string, "não foi encontrado\n")

def procura_na_Arv_B(arvore, string):
  for i in range(1,6):
    auxstr = string+str(i)
    a = arvore.search(auxstr)
    if a == 1:
      print("O nó", auxstr , "foi encontrado\n")
      printa_imagem(auxstr)
    else:
      print("O nó", auxstr, "não foi encontrado\n")
```

A primeira função é a de inserção dos catálogos utilizando concatenação de strings. O print da imagem é feito de maneira semelhante e tudo isso culmina na função de busca na árvore, que retornará que os nós do catálogo foram encontrados ou não.

III. Resultados

Para que haja o plot das fotos, primeiro foi-se feito um tratamento nas imagens, visto que nem sempre sabe-se o tipo da imagem(JPG, JPEG, PNG e etc), então todas as imagens foram transformadas em arquivos JPG para melhor facilitar os plots das imagens. Para que também haja o plot, foi upado o banco de imagens no próprio Colab.Primeiramente, iremos criar uma arvore para inserir uma imagem no repositório e logo após, buscaremos ela na mesma árvore e printaremos ela na tela:

```
####Criação da Árvore B no problema dado, ou seja, Á
    rvore B com t=3####
arv = BTree(3)

####Inserção de uma imagem à árvore####
ini=time.time()
nome='caderno1'
arv.insert(nome)
fim=time.time()

print(f'Tempo de execução da inserção da imagem: {
    fim-ini} segundos')

####Print da árvore com 1 elemento####
arv.print_order()

####Print da imagem e da árvore B####
ini=time.time()
procura_nome_na_Arv_B(arv, nome)
fim=time.time()

print(f'Tempo de execução da busca e do plot da
    imagem: {fim-ini} segundos')
```

O tempo de inserção da imagem foi de $5.10 \cdot 10^{-5}$ segundos e o tempo de busca e do plot da imagem foi de 0.003835 segundos. O código de inserção de todos os catálogos é dado por:

```
####Criação da Árvore B no problema dado, ou seja, Á
    rvore B com t=3####

arvore = BTree(3)

####Inserir todos as imagens na árvore B usando do
    repositório e pasta do Colab-Google####

ini=time.time()

insere_catalogo(arvore, 'gato')
insere_catalogo(arvore, 'cafe')
insere_catalogo(arvore, 'camera')
insere_catalogo(arvore, 'computador')
insere_catalogo(arvore, 'diversao')
insere_catalogo(arvore, 'doce')
insere_catalogo(arvore, 'flor_amarela')
insere_catalogo(arvore, 'flor_branca')
insere_catalogo(arvore, 'flor_rosa')
insere_catalogo(arvore, 'flor_roxa')
insere_catalogo(arvore, 'frutas')
insere_catalogo(arvore, 'mesa')
insere_catalogo(arvore, 'neve')
```

```
insere_catalogo(arvore, 'notebook')
insere_catalogo(arvore, 'objeto')
insere_catalogo(arvore, 'pessoas_flor')
insere_catalogo(arvore, 'pincel')
insere_catalogo(arvore, 'pipoca')
insere_catalogo(arvore, 'tecnologia')
insere_catalogo(arvore, 'trabalho')

fim= time.time()

print(f'Tempo de execução da inserção dos catálogos: {fim-ini} segundos')
####Após inserir, iremos verificar como está a Árvore B, e fica aparente a altura da mesma também####

arvore.print_order()
```

O output deixa claro a altura da Árvore B na situação de manipulação, ou seja, com todos os objetos inseridos, a Árvore B fica com altura 4. O tempo de execução para a inserção de todos os catálogos dentro da árvore, temos um resultado de 0.001411 segundos. Para o plot e busca na árvore B, temos:

```
string = input('Coloque o catálogo que deseja buscar:')

print('\n')

ini=time.time()
procura_na_Arv_B(arvore, string)
fim=time.time()

print(f'Tempo de execução da busca por um catálogo: {fim-ini} segundos')
```

Diante dessa, podemos procurar o catálogo inserido pelo usuário na função "input". O tempo de execução para um catálogo inexistente é de 0.0002391 segundos, enquanto o tempo de execução para o catálogo de "cafe" é de 0.02789 segundos.

IV. Conclusão

No comparativo de uma única imagem e de um catálogo, com um diferencial, ou seja, o caso trivial e o caso aplicado, temos uma diferença na grandeza de milisegundos de diferença na busca de dados. Podemos observar na tabela abaixo:

	Inserção	Busca e Plot
Uma imagem	$5.10 \cdot 10^{-5}$ s	$3.835 \cdot 10^{-3}$ s
Um catálogo	$1.1411 \cdot 10^{-4}$ s	$2.789 \cdot 10^{-2}$ s

Diante do comparativo, temos uma semelhante ordem de grandeza, sendo que de uma imagem na árvore trivial é dado pela ordem de grandeza de 10^{-5} segundos enquanto a do catálogo em uma árvore cheia é de 10^{-4}. Ou seja, com relação à inserção, a ordem de grandeza é dada de maneira bem rápida mesmo que seja numa árvore cheia de informações. Para a busca e plot, uma ordem de grandeza variando entre 10^{-3} e 10^{-2} é muito boa, visto que o programa demora relativamente para plotar a imagem. Ou seja, a árvore B tem seus fatores de complexidade, entretanto ela é uma estrutura de dados super eficiente no quesito de otimização de tempo e de armanezamento de dados.

References

[1] Sérgio Sierro. Dados estruturados vs dados não-estruturados. Disponível em: https://blog.grancursosonline.com.br/dados-estruturados-vs-dados-nao-estruturados/, Jun 2020. Accessed: 2020-02-01.

[2] Pixabay. Free images. Disponível em: https://pixabay.com/pt/. Accessed: 2020-01-28.

[3] Hemerson Pistori. Árvores b (b-trees). Disponível em: http://www.gpec.ucdb.br/pistori/disciplinas/ed/aulas_II/bt. Accessed: 2020-02-01.

[4] Mateor. A simple b-tree in python that supports insert, search and print. Disponível em: https://gist.github.com/mateor/885eb950df7231f178a5. Accessed: 2020-02-01.