Pesquisa e Ordenação de Dados

Gustavo Bottega Falcão
gustavobottegafalcao.gb@gmail.com

UFSM

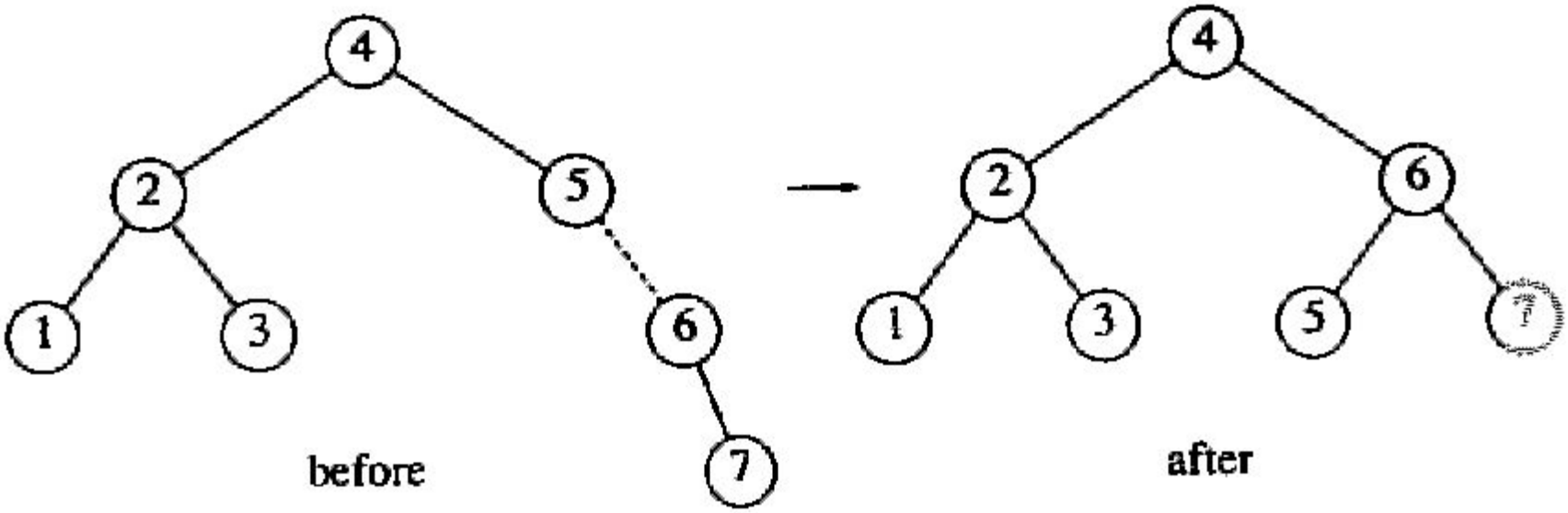
Universidade Federal de Santa Maria
Departamento de Tecnologia da Informação

<Árvore AVL>

- **Definição**: Árvore AVL.

Utilizar uma Árvore AVL para realizar pesquisas é uma técnica bastante utilizada em algoritmos e estruturas de dados. Essa Estrutura é chamada assim porque os nomes de seus criadores são Adelson-Velskii e Landis, e ela é basicamente uma Árvore binária equilibrada, a qual mantém sua altura balanceada para que seja feitos mais rápido a inserção, remoção e pesquisa nos dados.

Exemplo de balanceamento de árvore binaria:



<Árvore AVL>

- **História e Aplicação**:

  - A árvore AVL origina-se de 1962, e foi criada por dois matemáticos soviéticos Georgy Adelson-Velskii e Evgenii Landis. Sua ideia foi de criar uma árvore binária que se equilibra de maneira automática, assim evitando problemas que iriam atrapalhar a eficiência das operações.

  - Ela é amplamente utilizada quando as aplicações exigem pesquisas bastante rápidas, como banco de dados, algoritmos de busca, compiladores e sistemas de arquivos. O momento que ela mais se destaca é quando precisa-se fazer grande número de inserções e remoções sempre mantendo a árvore equilibrada o tempo todo.

  - Além dessas aplicações, a árvore avl é comumente utilizada em Sistemas de roteamento de rede, onde ela permite uma busca rápida e eficiente pelas melhores rotas em uma rede de grande escala, armazenando e pesquisando informações sobre as diversas rotas disponíveis. Também é bastante utilizada em estruturas que gerenciam a memória cache, pois com suas pesquisas rápidas, pode buscar com eficiência na memória os últimos dados acessados, assim melhorando o desempenho do sistema.

  - É importante avaliar o sistema que ela vai ser utilizada para saber se realmente vale a pena sua utilização, pois ela necessita de um bom espaço de armazenamento, além de possuir certos requisitos de desempenho, e ela pode exigir às vezes estruturas de dados especializadas para otimizar operações específicas, como pesquisa em largura ou profundidade por exemplo.

<Árvore AVL>

- **Funcionamento**:

Já que o desempenho da árvore avl depende de estar balanceada, durante seu funcionamento ela tem o objetivo de garantir que a diferença de altura entre as subárvores esquerda e direita de cada nó seja no máximo 1.

Sempre que um novo nó é inserido na árvore, primeiro deve-se encontrar o local apropriado para inseri-lo, e para isto, é utilizada as regras de uma árvore binária de pesquisa. Após isso, é verificado o balanceamento dela, se ela deixar de ser balanceada, serão aplicadas rotações para que ela seja equilibrada, e assim como explicado no parágrafo anterior, a diferença de altura das subárvores não passe de 1.

Durante a remoção de um nó na árvore, novamente é necessário utilizar as regras da árvore de pesquisa binária para encontrar o nó a ser removido. E após a remoção, como na inserção, é verificado se o balanceamento da árvore foi prejudicado, se sim, irá ocorrer as rotações corretas para que ela volte a ser balanceada.

Por fim, sobre a pesquisa na árvore avl, não há muito segredo, já que é utilizado o mesmo princípio de árvore de pesquisa binária, onde se começa pela Raiz, comparando o valor que se quer achar com o valor do nó atual, e com base dessa comparação, segue-se para direita ou para a esquerda, e fica nisso até que o nó desejado seja alcançado, ou acha-se um nó nulo, oque indica que o valor não está ali na árvore.

<Árvore AVL>

- **Prós e contras**:

Busca Eficiente: A árvore avl tem uma complexidade O(log n), onde n é o número de nós da árvore. O Que significa que ela é mais rápida que em estruturas que não são balanceadas, como listas lineares, ainda mais quando a árvore possui muitos nós.

A árvore avl possui balanceamento automático, onde ela mantém sempre sua altura balanceada, oque garante a diferença da altura seja no maximo de 1. Evitando assim em um desempenho consistente em toas as operações.

Ela possui o desempenho previsível, mesmo para árvores muito grandes. Isso é vantajoso quando precisa-se manipular um volume grande de dados, e garante um tempo de resposta rápido.

Contras.

A árvore AVL é bem complexa na hora de sua implementação, e requer atenção aos seus detalhes, além de bom conhecimento de algoritmos de rotação e de atualização. Quando a implementação é feita de maneira incorreta, pode ocasionar em erros de balanceamento e a árvore passa a se conportar de forma estranha.

Overhead adicional: Na árvore, o balanceamento é feito de forma automática por rotações e atualizações, e isso pode resultar em um tempo de execução maior do que em estruturas não balanceadas, além de requerer informações adicionais sobre seu balanceamento, como fatores de equilíbrio ou altura. Isso pode aumentar o espaço necessário para guardar cada nó da árvore.

<Árvore AVL>

- **Teste de Mesa**:

```
# Criar uma instância da Pesquisa AVL
pesquisa_avl = PesquisaAVL()

# Carregar o arquivo CSV
pesquisa_avl.carregar_dados('salary.csv')

# Realizar a pesquisa por um valor
valor_pesquisa = input("Digite o nome da coluna a ser pesquisada: ")
resultado = pesquisa_avl.pesquisar_coluna(valor_pesquisa)

if resultado:
    print(f"Coluna encontrada: {resultado}")

    # Imprimir os valores da coluna correspondente
    valores_coluna = pesquisa_avl.df[resultado].values
    print("Valores da coluna:")
    print(valores_coluna)



    if valor_pesquisa == 'age' or valor_pesquisa (variable) pesquisa_avl: PesquisaAVL quisa == 'salary':
            valor_pesquisa = int(input("Digite o
            valores_filtrados = pesquisa_avl.df[pesquisa_avl.df[resultado] == valor_pesquisa]
            print(f"Valores encontrados para {valor_pesquisa}:")
            print(valores_filtrados)
             # Salvar os resultados no arquivo
            pesquisa_avl.salvar_resultados(resultado, valores_coluna, valor_pesquisa, valores_filtrados)


    else:



            # Pesquisar um valor entre os correspondentes da coluna
            valor_pesquisa = input("Digite o valor a ser pesquisado: ")
            valores_filtrados = pesquisa_avl.df[pesquisa_avl.df[resultado] == valor_pesquisa]
            print(f"Valores encontrados para {valor_pesquisa}:")
            print(valores_filtrados)
            # Após a pesquisa e obtenção dos resultados
            pesquisa_avl.salvar_resultados(resultado, valores_coluna, valor_pesquisa, valores_filtrados)


else:
    print("Coluna não encontrada.")
```

<Árvore de Busca Binária>

- **Definição e Origem**:

Uma árvore Binária é uma estrutura de dados a qual é em forma de árvore, ela permite armazenar os elementos de forma organizada, para após isso, conseguir realizar buscas eficientes. A mesma possui uma regra predominante que é bem simples, a qual é que em cada nó, todos os elementos a sua esquerda são menores que o valor dele mesmo, e todos os elementos da direita são maiores que eles.

Esta estrutura originou-se do conceito de árvore binária, desenvolvido pelo matemático húngaro László Kalmár em 1920. Porém, foi popularizado em 1960 no livro The Art of Computer Programming, escrito por Donald Knuth. Desde então, ela é estudada e aplicada em muitas áreas da ciência da computação.

As principais operações de uma árvore e busca binária incluem a inserção, remoção e a busca por elementos, a busca é feita de forma eficiente, pois a cada comparação, o tamanho do espaço a busca é reduzido pela metade.

Além disso, as árvores de busca binária podem ser percorridas de três maneiras diferentes: em ordem, pré-ordem e pós-ordem .

Em ordem: Os elementos são acessados em ordem crescente, primeiro a subárvore esquerda, depois a raiz e, por fim, a subárvore direita.

Pré-ordem: A raiz é visitada primeiro, seguida pela subárvore esquerda e, por fim, pela subárvore direita.

Pós-ordem: As subárvores esquerda e direita são visitadas primeiro, e depois a raiz é visitada por último.

<Árvore de Busca Binária>

- **Aplicação**:

Cada método listado no slide anterior pode ser aplicado em diferentes aplicações. Como obter uma sequência ordenada dos elementos (Em ordem), criar uma cópia da árvore (Pré ordem) ou liberar a memória alocada pela árvore (Pós Ordem). Esses métodos são utilizados em algoritmos de busca, processamento de elementos e manipulação de árvores de busca binária.

Algumas das utilizações mais comuns para a árvore de busca binária são:

Algoritmos de busca e classificação. Onde são utilizadas para otimizar os algoritmos, um exemplo disso, é encontrar um elemento na árvore de busca binária ordenada, que será feito de forma eficiente e rápida.

São utilizadas em aprendizado de máquinas e inteligências artificiais, onde comumente ajudam a construir árvores de decisão, as quais são estruturas que ajudam a tomar decisões perante a um número de condições.

Além disso, são utilizadas nas estruturas de dados quando um requisito delas seria a eficiência, já que faz de maneira organizada e eficiente a inserção, remoção e a busca dos dados.

São exclusivamente importantes quando a ordem dos elementos é importante no programa. Uns exemplos disso seriam dicionários, mapas e conjuntos ordenados.

<Árvore de Busca Binária>

- **Vantagens e desvantagens de sua utilização**:

Vantagens:

As árvores de busca binária possuem uma busca bastante eficiente, já que, como foi falado nos slides anteriores, o espaço de busca é reduzido pela metade a cada comparação. Isso resulta em um tempo de resposta de busca bem rápido, ainda mais quando a árvore está equilibrada.

Além disso, suas inserções e remoções de dados são muito eficientes também, só que isso a partir do pressuposto de que a árvore esteja bem balanceada. E isso é feito com algoritmos de balanceamento, um deles que pode ser utilizado é o tópico anterior, a árvore AVL. E a partir disso, é possível manter uma boa eficiência e desempenho durante as operações.

Desvantagens:

Desbalanceamento: Com a falta de um metodo de balanceamento adequado, ao decorrer das das operações, a árvore de busca binária pode acabar se tornando desbalanceada, o que implica diretamente na eficiência do método nas buscas.

O espaço de armazenamento pode ser um fator a se tornar uma desvantagem que ocorre quando o número de elementos a armazenar é muito grande e o espaço de armazenamento é limitado.

Concluindo, as árvores de busca binária são estruturas adaptáveis que são muito utilizadas na ciência da computação, pois oferecem eficiência na realização dos processos de busca, inserção e remoção de dados. Porém, é importante ter um certo cuidado no balanceamento da árvore para que possa sempre se manter a eficiência e desempenho desejados.

<Árvore de Busca Binária>

- **Teste de Mesa**:

```python
class Node:
    def __init__(self, value):
        # Inicializa um nó com um valor e define os nós esquerdo e direito como None
        self.value = value
        self.left = None
        self.right = None

class BinaryTree:
    def __init__(self):
        # Inicializa uma árvore binária com a raiz definida como None
        self.root = None

    def insert(self, value):
        # Se a árvore estiver vazia, insere o valor como a raiz
        if self.root is None:
            self.root = Node(value)
        else:
            # Caso contrário, chama o método _insert_recursive para inserir o valor de forma recursiva
            self._insert_recursive(self.root, value)

    def _insert_recursive(self, node, value):
        if value < node.value:
            if node.left is None:
                # Se o valor for menor que o valor do nó atual e o nó esquerdo for None, insere o valor como nó esquerdo
                node.left = Node(value)
            else:
                # Caso contrário, chama o método _insert_recursive no nó esquerdo para inserir o valor de forma recursiva
                self._insert_recursive(node.left, value)
        elif value > node.value:
            if node.right is None:
                # Se o valor for maior que o valor do nó atual e o nó direito for None, insere o valor como nó direito
                node.right = Node(value)
            else:
                # Caso contrário, chama o método _insert_recursive no nó direito para inserir o valor de forma recursiva
                self._insert_recursive(node.right, value)

    def search(self, value):
        # Chama o método _search_recursive para procurar o valor na árvore
        return self._search_recursive(self.root, value)

    def _search_recursive(self, node, value):
        if node is None or node.value == value:
            # Se o nó atual for None ou tiver o valor procurado, retorna o nó atual
            return node
        if value < node.value:
            # Se o valor for menor que o valor do nó atual, chama o método _search_recursive no nó esquerdo
            return self._search_recursive(node.left, value)
        # Caso contrário, chama o método _search_recursive no nó direito
        return self._search_recursive(node.right, value)
```

<Jump Search>

Origem e Funcionamento:

Também conhecido como pesquisa por salto, o algoritmo de pesquisa Jump Search é um algoritmo de busca em dados sequenciais, como listas ordenadas ou arrays. Sua principal proposta foi melhorar a eficiência da busca linear, reduzindo o número de comparações necessárias para realizar a busca.

O algoritmo foi desenvolvido pelo cientista o qual se chamava Arne Basse Jensen no ano de 1988. O cientista propôs que o algoritmo seria uma alternativa à utilização além da busca linear, tendo como objetivo uma abordagem que conseguisse ser mais eficiente que a busca linear, economizando assim, tempo e comparações.

Seu funcionamento é relativamente simples comparado aos que vimos anteriormente. Primeiramente ele divide o array em blocos de tamanho fixo, os quais geralmente são chamados de saltos. Aps isso, o algoritmo realiza um salto para frente, no qual ele verifica o valor do elemento nesta posição, e se for maior que o elemento que se quer achar, o algoritmo realiza uma busca linear nos valores anteriores a esse. E se o valor for menor que o procurado, ele realiza outro salto para frente, e fica realizando este movimento até encontrar o elemento ou cair em um maior que o procurado.

<Jump Search>

Vantagens e desvantagens:

A principal vantagem do algoritmo de pesquisa, é que, em relação a busca linear, ele melhora significativamente a eficiência. O algoritmo de Busca linear tem uma complexidade de tempo O(n), em que o n é o tamanho do array, Já o Jump Search possui uma complexidade O(√n), isso ocasiona na redução do número de comparações necessárias. E também o torna mais recomendado para a manipulação de grandes quantidades de dados.

Porém deve-se observar alguns pontos específicos na utilização do Jump Search. Primeiramente, os conjuntos em que ele irá ser utilizado devem estar ordenados, pois o algoritmo depende disso para que possa funcionar de maneira correta. Outro ponto a se considerar, é que sua eficiência pode variar dependendo do tamanho do bloco escolhido, um bloco muito grande pode diminuir a eficiência do algoritmo, já um muito pequeno pode resultar em saltos e comparações adicionais e desnecessários.

<Jump Search>

Aplicação:

O algoritmo de pesquisa Jump Search tem diversas aplicações práticas nas quais pode ser utilizado, um exemplo são banco de dados, sistemas de arquivos e até algoritmos de outras estruturas de dados sequenciais, desde que os mesmos estejam ordenados.

Além disso, como falei no slide anterior, o Jump Search pode ser combinado com outros algoritmos de pesquisa, como a busca binária, e a partir disso, pode-se obter uma maior eficiência no processo de pesquisa.

Resumindo, este algoritmo de pesquisa é considerado uma técnica eficiente para buscar elementos em estruturas de dados desde que elas estejam previamente ordenadas. Assim, ele reduz o número de comparações necessárias dividindo o array em blocos de tamanho fixo.

E embora tenha suas vantagens, como a melhora na eficiência comparado à busca linear, é importante considerar que ele precise que os dados estejam previamente ordenados e escolher o tamanho adequado dos blocos de tamanho fixo, assim obtendo os melhores resultados possíveis.

<Jump Search>

Teste de mesa:

```
# Carregar o arquivo CSV
df = pd.read_csv('salary.csv', delimiter=';')

# Ordenar a lista pela coluna 'Gender'
df.sort_values(by='age', inplace=True)
column_values = df['age'].values

# Realizar a pesquisa por um valor usando Jump Search
def jump_search(arr, x):
    n = len(arr)
    step = int(math.sqrt(n))  # Determina o tamanho do salto
    prev = 0  # Índice anterior

    # Enquanto o elemento no índice atual for menor que x
    while arr[min(step, n) - 1] < x:
        prev = step
        step += int(math.sqrt(n))  # Aumenta o tamanho do salto
        if prev >= n:  # Se o tamanho do salto for maior ou igual ao tamanho do array
            return -1

    # Realiza uma busca linear a partir do índice prev
    while arr[prev] < x:
        prev += 1
        if prev == min(step, n):  # Se o índice prev alcançar o tamanho do salto ou o tamanho do array
            return -1

    # Se o elemento no índice prev for igual a x, retorna prev
    if arr[prev] == x:
        return prev

    # Caso contrário, o valor não foi encontrado
    return -1

# Solicitar o valor a ser pesquisado
search_value = input("Digite o valor a ser pesquisado: ")

# Realizar a pesquisa usando Jump Search
```

<Jump Search>

Teste de mesa:

```
# Realizar a pesquisa usando Jump Search
result = jump_search(column_values, search_value)

if result != -1:
    print(f"Valores encontrados para {search_value}:\n")
    indices = []

    # Encontrar todos os índices correspondentes ao valor pesquisado
    while result != -1:
        indices.append(result)

        # Realizar a próxima pesquisa a partir do próximo índice
        result = jump_search(column_values[result + 1:], search_value)

        # Se a pesquisa encontrar um novo índice correspondente, ajustar o índice encontrado
        if result != -1:
            result += indices[-1] + 1

    # Mostrar todas as linhas correspondentes ao valor pesquisado
    for index in indices:
        print(df.iloc[index])
        print()

    # Salvar os resultados da pesquisa em um arquivo de texto
    result_file_name = "result_pesquisa_jump.txt"
    with open(result_file_name, 'w') as file:
        for index in indices:
            file.write(str(df.iloc[index]) + '\n\n\n')
    print(f"Resultados da pesquisa salvos no arquivo '{result_file_name}'.")
else:
    print(f"Nenhum valor encontrado para {search_value}.")
```

<Pesquisa exponencial>

Origem e funcionamento:

O algoritmo de pesquisa exponencial, que também é popularmente chamado de pesquisa por duplicação, é um método bastante eficiente para encontrar um elemento quando o mesmo está em uma lista ordenada. Sua ideia base, é de que a comparação e elementos distantes pode ser pulada em vez de verificar sequencialmente o elemento.

Primeiramente, o algoritmo foi proposto por Jon Bentley em 1986. E foi construída com o intuito de melhorar a eficiência em relação a pesquisa binária, onde ela aproveita que a estrutura já está ordenada para dar saltos exponenciais que são utilizados para reduzir o número e comparações necessárias.

Seu funcionamento não é complexo, podemos dividi-lo em 4 passos:

1. Começa com um intervalo inicial de tamanho 1.
2. Verifica se o elemento no índice atual é menor que o elemento desejado. Se sim, dobra o tamanho do intervalo.
3. Realiza uma pesquisa binária dentro do intervalo encontrado, limitando-o pelos índices mínimo e máximo da lista.
4. Se o elemento for encontrado dentro do intervalo, a pesquisa é concluída e retorna a posição do elemento na lista. Caso contrário, repete os passos 2 e 3 até encontrar o elemento ou até ultrapassar o tamanho total da lista.

<Pesquisa exponencial>

Funcionamento:

Como falei no slide anterior, a Pesquisa Exponencial possui um funcionamento simples.

Ela Basicamente aproveita o fato de que a lista já está ordenada e quanto maior ela for, os elementos distantes terão valores maiores ainda. Então, sua ideia é pular para posições que são exponencialmente maiores até que ela encontre um intervalo que tenha o elemento desejado.

Após isso, é realizado uma pesquisa binária no referido intervalo para que se possa encontrar o elemento desejado.

<Pesquisa exponencial>

Vantagens e desvantagens:

Quando falamos de vantagens e desvantagens, sua principal vantagem e proposta é o fato de ele ser mais eficiente em listas previamente ordenadas. Pois com os saltos que o algoritmo realiza, lhe permitem diminuir o número de comparações necessárias quando comparado a busca linear.

Sua complexidade de tempo é de $O(\log_2(n))$, onde 'n' é o tamanho da lista. Isso o torna mais rápido que a pesquisa linear, que possui uma complexidade de tempo linear.

Outro fator é de que esta pesquisa não precisa de estruturas de dados complexas e é bem simples e fácil de implementar.

Já falando dos fatores negativos do algoritmo de pesquisa, destaca-se o fato de que ele só pode ser aplicado em listas ordenadas.

O desempenho da pesquisa exponencial geralmente é afetado se se o elemento estiver muito perto do início da lista. Isso porque os saltos exponenciais avançam muito rápido no início.

Por fim, a pesquisa exponencial pode exigir um número significativo de iterações antes de encontrar o elemento desejado em listas muito grandes.

<Pesquisa exponencial>

Aplicação:

A pesquisa exponencial pode ser aplicada em diferentes ocasiões quando o conjunto de dados está ordenado. alguns exemplos são:

Pesquisa em banco de dados: Nos bancos de dados que possuem os índices ordenados, esta pesquisa pode ser utilizada para encontrar dados à partir de critérios específicos. Como buscar um nome específico numa lista ordenada em ordem alfabética.

Jogos: Quando os jogos fazem a utilização de listas ordenadas, como jogos de palavras ou quebra cabeças. A pesquisa exponencial pode ser utilizada para encontrar a palavra desejada ou achar a solução do problema rapidamente. Isso pode melhorar a eficiência do carregamento dos jogos e fornecer aos jogadores uma melhor experiência.

Busca em navegadores da web: Quando se realiza uma pesquisa na web ou em um documento muito extenso, a pesquisa exponencial pode ser utilizada para encontrar palavras-chave ou termos. Assim, podendo proporcionar uma navegação mais eficiente e rápida pelo conteúdo da página.

<Pesquisa exponencial>

Teste de mesa:

```
class ExponentialSearch:
    def __init__(self):
        self.df = None
        self.columns = None

    def load_data(self, file_name):
     # Carrega o arquivo CSV utilizando o pandas e define o delimitador como ';'

        self.df = pd.read_csv(file_name, delimiter=';')

        # Obtém os nomes das colunas do dataframe e os armazena como uma lista
        self.columns = self.df.columns.tolist()

    def exponential_search(self, value):
        if self.columns is None:
            return None

        # Verifica se o valor procurado está na primeira coluna

        if self.columns[0] == value:
            return self.columns[0]

        n = len(self.columns)
        i = 1
        # Realiza a busca exponencial
        while i < n and self.columns[i] != value:
            i *= 2

        if i >= n:
            return None

        return self.columns[i]
    # O método exponential_search implementa o algoritmo de pesquisa exponencial.
    # Ele recebe um valor como argumento e verifica se a lista de colunas está vazia.
    # Se estiver vazia, retorna None. Em seguida, verifica se o valor procurado está
    # na primeira coluna. Se estiver, retorna essa coluna.
    # Caso contrário, realiza a pesquisa exponencial, dobrando o valor de i a cada iteração
    #  até que o valor seja encontrado ou ultrapasse o tamanho da lista de colunas.
    # Se o valor não for encontrado, retorna None. Caso contrário, retorna a coluna encontrada.
```

<Referências>

"Árvores AVL" - ICMC USP - Disponível em: http://wiki.icmc.usp.br/images/f/fa/Árvores_AVL.pdf

"Árvore AVL" - Wikipedia - Disponível em: https://pt.wikipedia.org/wiki/Árvore_AVL

"Árvores de Busca Balanceadas" - Cláudio Cesar de Sá - Disponível em: https://www.facom.ufu.br/~claudio/Cursos/Antigos/EDxxxx/Artigos/bcc-ed05.pdf

"Estruturas de Dados: Árvores de Busca" - UFSC - Disponível em: https://www.inf.ufsc.br/~aldo.vw/estruturas/Estruturas.ArvBusca.html

"Binary Search Trees (BST) Explained with Examples" - freeCodeCamp - Disponível em: https://www.freecodecamp.org/news/binary-search-trees-bst-explained-with-examples/

"Árvores Binárias de Busca" - Vanessa Braganholo - Disponível em: http://www2.ic.uff.br/~vanessa/material/ed/04-ArvoresBinariasBusca.pdf

"Árvore binária de busca" - Wikipedia - Disponível em: https://pt.wikipedia.org/wiki/Árvore_binária_de_busca

"Jump Search" - Acervo Lima - Disponível em: https://acervolima.com/jump-search/

"Jump search" - Wikipedia - Disponível em: https://en.wikipedia.org/wiki/Jump_search

"Jump Search Algorithm" - Study Tonight - Disponível em: https://www.studytonight.com/data-structures/jump-search-algorithm

"Busca exponencial" - Wikipedia - Disponível em: https://pt.wikipedia.org/wiki/Busca_exponencial

"Exponential Search" - Techie Delight - Disponível em: https://www.techiedelight.com/pt/exponential-search/

"Busca Exponencial" - Desenvolvendo Software - Disponível em: http://desenvolvendosoftware.com.br/algoritmos/busca/busca-exponencial.html