2º Trabalho de Estrutura de Dados

FORTALEZA
2025

UNIVERSIDADE FEDERAL DO CEARÁ

DEPARTAMENTO DE ESTATÍSTICA E MATEMÁTICA APLICADA - DEMA

Autores

1 - DAVI DA SILVA ARAÚJO, 574298;

2 - NIKELLY SANTIAGO DA SILVA, 567082;

Sumário

1 Introdução 4
2 Metodologia 4
2.1 Ferramentas de análise de dados utilizadas 5
3 Análise de resultados 5
3.1 Questão 2 - Árvore AVL 5
3.1.1 Estrutura da Classe 5
3.1.2 Principais Métodos 5
3.1.3 Complexidade das Operações 5
3.1.4 Lógica da Implementação 6
3.1.5 Código-Fonte 6
3.1.6 Análise de Desempenho 10
3.2 Questão 4 - Gerenciamento de Rankings com Árvore AVL 10
3.2.1 Estrutura da Classe 10
3.2.2 Principais Métodos 10
3.2.3 Complexidade das Operações 10
3.2.4 Código-Fonte 11
3.2.5 Análise de Desempenho 15
3.3 Questão 6 - Banco de Dados com Árvore B 15
3.3.1 Estrutura da Classe 15
3.3.2 Principais Métodos 15
3.3.3 Complexidade das Operações 16
3.3.4 Código-Fonte 16
3.3.5 Análise de Desempenho 19
3.4 Questão 8 - Estrutura de Dados de Hash 20
3.4.1 Estrutura da Classe 20
3.4.2 Principais Métodos 20
3.4.3 Complexidade das Operações 20
3.4.4 Código-Fonte 20
3.4.5 Análise de Desempenho 24
3.5 Questão 10 - Estrutura de Dados Heap e Heapsort 24
3.5.1 Estrutura da Classe 24
3.5.2 Principais Métodos 24
3.5.3 Complexidade das Operações 25
3.5.4 Código-Fonte 25
3.5.5 Análise de Desempenho 27
4 Considerações Finais 27

Lista de Códigos

1 Código questão 2 6
2 Código questão 4 11
3 Código questão 6 16
4 Código questão 8 20
5 Código questão 10 25

Lista de Tabelas

1 Tempos médios de inserção e remoção na Árvore AVL 10
2 Tempos médios de inserção e remoção na Árvore AVL para rankings 15
3 Tempos médios de inserção e remoção na Árvore B 19
4 Tempos médios de inserção e remoção na Tabela Hash 24
5 Tempos médios das operações na estrutura de Heap 27

1 Introdução

Este trabalho foi desenvolvido para a disciplina de Estrutura de Dados e tem como objetivo a implementação e análise de diferentes estruturas de dados avançadas. O foco está na resolução das questões pares (2, 4, 6, 8 e 10), conforme a divisão estabelecida pelo professor.

As questões abordadas envolvem a implementação das seguintes estruturas de dados:

- **Árvore AVL** (Questão 2): Implementação de uma árvore balanceada para operações eficientes de inserção, remoção e busca.
- **Sistema de Gerenciamento de Rankings com Árvore AVL** (Questão 4): Aplicação da árvore AVL para a organização e atualização dinâmica de rankings em uma competição.
- **Banco de Dados com Árvore B** (Questão 6): Uso da árvore B para armazenar registros de clientes, otimizando operações de busca e inserção.
- **Tabela Hash** (Questão 8): Implementação de uma estrutura de hash personalizada, incluindo tratamento de colisões e rehashing.
- **Heap e Heapsort** (Questão 10): Desenvolvimento da estrutura de Heap Mínimo e implementação do algoritmo de ordenação Heapsort.

Além da implementação, foi realizada uma análise do desempenho dessas estruturas em diferentes cenários, considerando tempo de execução e eficiência das operações. O trabalho busca demonstrar a importância dessas estruturas na resolução de problemas computacionais reais, evidenciando suas vantagens e limitações.

2 Metodologia

Para a realização deste trabalho, utilizamos a linguagem de programação **Python** para implementar as diferentes estruturas de dados solicitadas. A abordagem seguiu os seguintes passos:

1. **Implementação das Estruturas de Dados**
   - Cada uma das estruturas (Árvore AVL, Tabela Hash, Heap, etc.) foi desenvolvida seguindo os requisitos da disciplina.
   - Os códigos foram escritos e testados individualmente para garantir seu correto funcionamento.
2. **Geração de Dados Aleatórios e Testes de Desempenho**
   - Para avaliar o desempenho das operações de **inserção** e **remoção**, implementamos um gerador de valores aleatórios em diferentes tamanhos: **100, 1.000, 10.000 e 1.000.000** elementos.
   - Foram medidos os tempos de execução para cada estrutura, permitindo uma análise comparativa da eficiência das operações.
3. **Divisão das Tarefas**
   - A equipe dividiu o trabalho de forma igualitária:
     - Um membro ficou responsável pela implementação de 3 questões
     - O outro membro foi responsável por implementar duas das questões e pela elaboração do relatório, garantindo a organização e a coerência dos resultados.
4. **Análise e Documentação**
   - Após a implementação e os testes, os resultados foram analisados e documentados no relatório.
   - Foram destacados os tempos de execução observados, as dificuldades encontradas e as considerações sobre o desempenho das estruturas de dados.

Essa metodologia permitiu uma distribuição eficiente das tarefas e uma avaliação objetiva da eficiência das estruturas implementadas.

2.1 Ferramentas de análise de dados utilizadas

Para termos um desenvolvimento mais orgânico do código, utilizamos as seguintes ferramentas:

1. **Visual Studio Code**
    - Para desenvolver o código apresentado no trabalho
2. **Git e GitHub**
    - Para compartilhar os códigos desenvolvidos
3. **Python 3.13**
    - Linguagem de programação escolhida pela equipe

3 Análise de resultados

3.1 Questão 2 - Árvore AVL

A árvore AVL é uma estrutura de dados balanceada utilizada para garantir eficiência nas operações de inserção, remoção e busca. A principal característica dessa estrutura é que, após cada inserção ou remoção, a árvore se reequilibra automaticamente para manter sua altura mínima.

3.1.1 Estrutura da Classe

A classe `AVLTree` é composta pelos seguintes elementos:

- Nós que armazenam valores, ponteiros para filhos esquerdo e direito e altura do nó.
- Métodos para inserção, remoção, balanceamento e cálculo de altura.

3.1.2 Principais Métodos

Os principais métodos implementados na árvore AVL incluem:

- **insert**: Insere um novo valor na árvore, garantindo o balanceamento.
- **remove**: Remove um valor e ajusta a estrutura para manter o balanceamento.
- **leftRotate** e **rightRotate**: Realizam rotações para manter a árvore balanceada.
- **getHeight** e **getBalance**: Calculam altura e fator de balanceamento dos nós.

3.1.3 Complexidade das Operações

As complexidades das operações na árvore AVL são:

- **Inserção**: $O(\log n)$
- **Remoção**: $O(\log n)$
- **Busca**: $O(\log n)$

O balanceamento automático da árvore garante que as operações se mantenham eficientes mesmo com grandes conjuntos de dados.

3.1.4 Lógica da Implementação

A implementação da árvore AVL segue os seguintes princípios:

- Cada nó contém um valor, ponteiros para os filhos esquerdo e direito, e um campo de altura.
- A inserção de um novo elemento pode levar a um desbalanceamento da árvore, sendo corrigido por rotações simples ou duplas.
- A remoção de um elemento também pode levar a desbalanceamentos, sendo resolvidos com rotações similares.

3.1.5 Código-Fonte

Listing 1: Código questão 2

```
import random
import time

class Node:
    def __init__(self, key):
        # Inicializa um nó com uma chave, ponteiros para os filhos esquerdo
            e direito, e a altura do nó
        self.key = key
        self.left = None
        self.right = None
        self.height = 1

class AVLTree:
    def insert(self, root, key):
        # Insere um valor na árvore AVL, balanceando-a conforme necessário
        if not root:
            return Node(key)
        elif key < root.key:
            root.left = self.insert(root.left, key)
        else:
            root.right = self.insert(root.right, key)

        # Atualiza a altura do nó ancestral
        root.height = 1 + max(self.getHeight(root.left), self.getHeight(
           root.right))

        # Calcula o fator de balanceamento do nó ancestral
        balance = self.getBalance(root)

        # Realiza rotações para balancear a árvore se necessário
        # Caso 1 - Rotação à direita
        if balance > 1 and key < root.left.key:
            return self.rightRotate(root)

        # Caso 2 - Rotação à esquerda
        if balance < -1 and key > root.right.key:
            return self.leftRotate(root)

        # Caso 3 - Rotação à esquerda-direita
        if balance > 1 and key > root.left.key:
            root.left = self.leftRotate(root.left)
            return self.rightRotate(root)
```

```
41
42          # Caso 4 - Rotação à direita-esquerda
43          if balance < -1 and key < root.right.key:
44              root.right = self.rightRotate(root.right)
45              return self.leftRotate(root)
46
47          return root
48
49      def leftRotate(self, z):
50          # Realiza uma rotação à esquerda
51          y = z.right
52          T2 = y.left
53
54          # Realiza a rotação
55          y.left = z
56          z.right = T2
57
58          # Atualiza as alturas dos nós envolvidos na rotação
59          z.height = 1 + max(self.getHeight(z.left), self.getHeight(z.right))
60          y.height = 1 + max(self.getHeight(y.left), self.getHeight(y.right))
61
62          # Retorna a nova raiz após a rotação
63          return y
64
65      def rightRotate(self, z):
66          # Realiza uma rotação à direita
67          y = z.left
68          T3 = y.right
69
70          # Realiza a rotação
71          y.right = z
72          z.left = T3
73
74          # Atualiza as alturas dos nós envolvidos na rotação
75          z.height = 1 + max(self.getHeight(z.left), self.getHeight(z.right))
76          y.height = 1 + max(self.getHeight(y.left), self.getHeight(y.right))
77
78          # Retorna a nova raiz após a rotação
79          return y
80
81      def getHeight(self, root):
82          # Retorna a altura de um nó
83          if not root:
84              return 0
85          return root.height
86
87      def getBalance(self, root):
88          # Calcula e retorna o fator de balanceamento de um nó
89          if not root:
90              return 0
91          return self.getHeight(root.left) - self.getHeight(root.right)
92
93      def remove(self, root, key):
94          # Remove um valor da árvore AVL, balanceando-a conforme necessário
95          if not root:
96              return root
```

```
97          elif key < root.key:
98              root.left = self.remove(root.left, key)
99          elif key > root.key:
100             root.right = self.remove(root.right, key)
101         else:
102             if root.left is None:
103                 temp = root.right
104                 root = None
105                 return temp
106             elif root.right is None:
107                 temp = root.left
108                 root = None
109                 return temp
110             temp = self.getMinValueNode(root.right)
111             root.key = temp.key
112             root.right = self.remove(root.right, temp.key)
113
114         if root is None:
115             return root
116
117         # Atualização  da altura do nó atual
118         root.height = 1 + max(self.getHeight(root.left), self.getHeight(
                root.right))
119
120         # Calcula o fator de balanceamento do nó
121         balance = self.getBalance(root)
122
123         # Realiza rotações para balancear a árvore se necessário
124         # Caso 1 - Rotação à direita
125         if balance > 1 and self.getBalance(root.left) >= 0:
126             return self.rightRotate(root)
127
128         # Caso 2 - Rotação à esquerda-direita
129         if balance > 1 and self.getBalance(root.left) < 0:
130             root.left = self.leftRotate(root.left)
131             return self.rightRotate(root)
132
133         # Caso 3 - Rotação à esquerda
134         if balance < -1 and self.getBalance(root.right) <= 0:
135             return self.leftRotate(root)
136
137         # Caso 4 - Rotação à direita-esquerda
138         if balance < -1 and self.getBalance(root.right) > 0:
139             root.right = self.rightRotate(root.right)
140             return self.leftRotate(root)
141
142         return root
143
144     def getMinValueNode(self, root):
145         # Encontra o nó com o menor valor na árvore
146         if root is None or root.left is None:
147             return root
148         return self.getMinValueNode(root.left)
149
150     def height(self):
151         # Retorna a altura da árvore
```

```
152         return self.getHeight(self.root)
153
154     def search(self, root, key):
155         # Verifica se um valor está na árvore
156         if root is None or root.key == key:
157             return root
158         if key < root.key:
159             return self.search(root.left, key)
160         return self.search(root.right, key)
161
162     def max(self, root):
163         # Retorna o maior valor da árvore
164         current = root
165         while current.right is not None:
166             current = current.right
167         return current.key
168
169     def min(self, root):
170         # Retorna o menor valor da árvore
171         current = root
172         while current.left is not None:
173             current = current.left
174         return current.key
175
176 # Função para medir os tempos de execução
177 def measure_operations(tree, values):
178     # Mede o tempo de inserção e remoção de valores na árvore
179     start_time = time.time()
180     for value in values:
181         tree.root = tree.insert(tree.root, value)
182     insert_time = time.time() - start_time
183
184     start_time = time.time()
185     for value in values:
186         tree.root = tree.remove(tree.root, value)
187     remove_time = time.time() - start_time
188
189     return insert_time, remove_time
190
191 def main():
192     # Gera dados aleatórios e mede os tempos de execução para diferentes
        tamanhos de entrada
193     sizes = [100, 1000, 10000, 1000000]
194     for size in sizes:
195         values = random.sample(range(1, size * 10), size)
196         tree = AVLTree()
197         tree.root = None
198         insert_time, remove_time = measure_operations(tree, values)
199         print(f"Tamanho: {size}")
200         print(f"Tempo de inserção: {insert_time:.6f} segundos")
201         print(f"Tempo de remoção: {remove_time:.6f} segundos")
202         print("=" * 40)
203
204 if __name__ == "__main__":
205     main()
```

3.1.6 Análise de Desempenho

Para avaliar o desempenho da árvore AVL, testamos as operações de inserção e remoção em conjuntos de dados de diferentes tamanhos (100, 1.000, 10.000 e 1.000.000 elementos). Os tempos médios obtidos foram:

Tamanho do Conjunto	Tempo de Inserção (s)	Tempo de Remoção (s)
100	0.000767	0.000536
1.000	0.021666	0.017089
10.000	0.299384	0.306004
1.000.000	29.583716	22.132272

Tabela 1: Tempos médios de inserção e remoção na Árvore AVL

Os resultados mostram que a árvore AVL mantém tempos eficientes de inserção e remoção, devido ao balanceamento automático. No entanto, conforme o número de elementos cresce, os tempos aumentam devido à necessidade de múltiplas rotações.

3.2 Questão 4 - Gerenciamento de Rankings com Árvore AVL

A árvore AVL é uma estrutura eficiente para armazenar e manter rankings de participantes, garantindo operações rápidas de inserção, remoção e busca, mantendo a árvore balanceada automaticamente.

3.2.1 Estrutura da Classe

A implementação da árvore AVL para gerenciamento de rankings inclui os seguintes elementos:

- Nós que armazenam o ID do participante e sua pontuação.
- Métodos para inserção, remoção, rotação e balanceamento automático.
- Funções auxiliares para obtenção dos top 10 participantes e aqueles com pontuação acima de um limite.

3.2.2 Principais Métodos

Os principais métodos implementados incluem:

- **insert**: Insere um novo participante ou atualiza a pontuação de um existente.
- **remove**: Remove um participante do ranking.
- **search**: Busca um participante pelo ID.
- **update_score**: Atualiza a pontuação de um participante específico.
- **top_10**: Retorna os 10 participantes com as maiores pontuações.
- **min_score**: Retorna o participante com a menor pontuação.
- **scores_above**: Retorna todos os participantes com pontuação acima de um limite.

3.2.3 Complexidade das Operações

As complexidades das operações na árvore AVL são:

- **Inserção**: $O(\log n)$
- **Remoção**: $O(\log n)$
- **Busca**: $O(\log n)$
- **Consulta de Top 10**: $O(n \log n)$ (devido à ordenação dos elementos)

A estrutura balanceada garante que as operações sejam eficientes mesmo com grandes volumes de participantes.

3.2.4 Código-Fonte

Listing 2: Código questão 4

```
import random
import time

class Node:
    def __init__(self, id_participante, pontuacao):
        # Inicializa um novo nó com ID do participante e pontuação
        self.id_participante = id_participante
        self.pontuacao = pontuacao
        self.left = None  # Filho esquerdo
        self.right = None  # Filho direito
        self.height = 1  # Altura do nó

class AVLTree:
    def __init__(self):
        # Inicializa a árvore AVL com a raiz como None
        self.root = None

    def getHeight(self, root):
        # Retorna a altura do nó
        if not root:
            return 0
        return root.height

    def getBalance(self, root):
        # Retorna o fator de balanceamento do nó
        if not root:
            return 0
        return self.getHeight(root.left) - self.getHeight(root.right)

    def rightRotate(self, z):
        # Realiza uma rotação para a direita
        y = z.left
        T3 = y.right
        y.right = z
        z.left = T3
        z.height = 1 + max(self.getHeight(z.left), self.getHeight(z.right))
        y.height = 1 + max(self.getHeight(y.left), self.getHeight(y.right))
        return y

    def leftRotate(self, z):
        # Realiza uma rotação para a esquerda
        y = z.right
        T2 = y.left
        y.left = z
        z.right = T2
        z.height = 1 + max(self.getHeight(z.left), self.getHeight(z.right))
        y.height = 1 + max(self.getHeight(y.left), self.getHeight(y.right))
        return y

    def insert(self, root, id_participante, pontuacao):
        # Insere um novo participante ou atualiza a pontuação de um
            existente
```

```
        if not root:
            return Node(id_participante, pontuacao)
        elif id_participante < root.id_participante:
            root.left = self.insert(root.left, id_participante, pontuacao)
        elif id_participante > root.id_participante:
            root.right = self.insert(root.right, id_participante, pontuacao
                )
        else:
            root.pontuacao = pontuacao

        # Atualiza a altura do nó
        root.height = 1 + max(self.getHeight(root.left), self.getHeight(
            root.right))
        balance = self.getBalance(root)

        # Realiza rotações conforme necessário para manter balanceado
        if balance > 1 and id_participante < root.left.id_participante:
            return self.rightRotate(root)
        if balance < -1 and id_participante > root.right.id_participante:
            return self.leftRotate(root)
        if balance > 1 and id_participante > root.left.id_participante:
            root.left = self.leftRotate(root.left)
            return self.rightRotate(root)
        if balance < -1 and id_participante < root.right.id_participante:
            root.right = self.rightRotate(root.right)
            return self.leftRotate(root)

        return root

    def remove(self, root, id_participante):
        # Remove um participante da árvore AVL
        if not root:
            return root
        elif id_participante < root.id_participante:
            root.left = self.remove(root.left, id_participante)
        elif id_participante > root.id_participante:
            root.right = self.remove(root.right, id_participante)
        else:
            # Caso com um ou nenhum filho
            if root.left is None:
                temp = root.right
                root = None
                return temp
            elif root.right is None:
                temp = root.left
                root = None
                return temp
            # Caso com dois filhos
            temp = self.getMinValueNode(root.right)
            root.id_participante = temp.id_participante
            root.pontuacao = temp.pontuacao
            root.right = self.remove(root.right, temp.id_participante)

        if root is None:
            return root

```

```
106         # Atualiza a altura do nó
107         root.height = 1 + max(self.getHeight(root.left), self.getHeight(
                root.right))
108         balance = self.getBalance(root)
109 
110         # Realiza rotações conforme necessário para manter balanceado
111         if balance > 1 and self.getBalance(root.left) >= 0:
112             return self.rightRotate(root)
113         if balance > 1 and self.getBalance(root.left) < 0:
114             root.left = self.leftRotate(root.left)
115             return self.rightRotate(root)
116         if balance < -1 and self.getBalance(root.right) <= 0:
117             return self.leftRotate(root)
118         if balance < -1 and self.getBalance(root.right) > 0:
119             root.right = self.rightRotate(root.right)
120             return self.leftRotate(root)
121 
122         return root
123 
124     def search(self, root, id_participante):
125         # Busca um participante pelo ID
126         if root is None or root.id_participante == id_participante:
127             return root
128         if id_participante < root.id_participante:
129             return self.search(root.left, id_participante)
130         return self.search(root.right, id_participante)
131 
132     def update_score(self, id_participante, nova_pontuacao):
133         node = self.search(self.root, id_participante)
134         if node:
135             node.pontuacao = nova_pontuacao
136 
137     def inorder(self, root, result):
138         # Percorre a árvore em ordem e armazena os participantes
139         if root:
140             self.inorder(root.left, result)
141             result.append((root.id_participante, root.pontuacao))
142             self.inorder(root.right, result)
143 
144     def top_10(self, root):
145         # Retorna os 10 participantes com maior pontuação
146         result = []
147         self.inorder(root, result)
148         return sorted(result, key=lambda x: x[1], reverse=True)[:10]
149 
150     def getMinValueNode(self, root):
151         # Retorna o nó com a menor pontuação
152         if root is None or root.left is None:
153             return root
154         return self.getMinValueNode(root.left)
155 
156     def scores_above(self, root, threshold, result):
157         # Retorna participantes com pontuação acima de um limite
158         if root:
159             if root.pontuacao >= threshold:
160                 self.scores_above(root.right, threshold, result)
```

```
                result.append((root.id_participante, root.pontuacao))
                self.scores_above(root.left, threshold, result)

    def min_score(self, root):
        # Retorna o participante com a menor pontuação
        if root is None:
            return None
        min_node = self.getMinValueNode(root)
        return (min_node.id_participante, min_node.pontuacao)

# Função para medir o tempo de execução de uma operação
def measure_time(operation, *args):
    start_time = time.time()
    operation(*args)
    end_time = time.time()
    return end_time - start_time

# Função para gerar dados aleatórios
def generate_random_data(size):
    return [(f"Participante_{i}", random.randint(1, 100)) for i in range(
        size)]

# Função para medir o tempo de inclusão de dados na árvore AVL
def measure_insertion_time(tree, data):
    root = None
    for id_participante, pontuacao in data:
        root = tree.insert(root, id_participante, pontuacao)
    return root

# Função para medir o tempo de remoção de dados da árvore AVL
def measure_removal_time(tree, root, data):
    for id_participante, _ in data:
        root = tree.remove(root, id_participante)
    return root

# Tamanhos dos dados
sizes = [100, 1000, 10000, 1000000]

# Medir tempos de inclusão e remoção para cada tamanho
for size in sizes:
    data = generate_random_data(size)
    tree = AVLTree()

    # Medir tempo de inclusão
    insertion_time = measure_time(measure_insertion_time, tree, data)
    print(f"Tamanho: {size}, Tempo de inclusão: {insertion_time:.4f} 
        segundos")

    # Medir tempo de remoção
    root = measure_insertion_time(tree, data)
    removal_time = measure_time(measure_removal_time, tree, root, data)
    print(f"Tamanho: {size}, Tempo de remoção: {removal_time:.4f} segundos"
        )

# Exemplo de uso
if __name__ == "__main__":
```

```
    tree = AVLTree()
    root = None
    root = tree.insert(root, "Cristiano", 50)
    root = tree.insert(root, "Messi", 70)
    root = tree.insert(root, "Vini", 40)

    print("Top 10:", tree.top_10(root))
    print("Menor pontuação:", tree.min_score(root))
    result = []
    tree.scores_above(root, 45, result)
    print("Pontuações acima de 45:", result)
```

3.2.5 Análise de Desempenho

Para avaliar o desempenho da árvore AVL, testamos as operações de inserção e remoção em conjuntos de dados de diferentes tamanhos (100, 1.000, 10.000 e 1.000.000 elementos). Os tempos médios obtidos foram:

Tamanho do Conjunto	Tempo de Inserção (s)	Tempo de Remoção (s)
100	0.0004	0.0003
1.000	0.0070	0.0043
10.000	0.0881	0.0658
1.000.000	15.5517	14.2090

Tabela 2: Tempos médios de inserção e remoção na Árvore AVL para rankings

Os resultados mostram que a árvore AVL mantém tempos eficientes para inserção e remoção, garantindo que os rankings sejam mantidos de forma dinâmica e rápida, mesmo em grandes competições.

3.3 Questão 6 - Banco de Dados com Árvore B

A árvore B é uma estrutura de dados balanceada utilizada para armazenar grandes volumes de dados de forma eficiente, garantindo operações rápidas de busca, inserção e remoção. Essa estrutura é amplamente utilizada em bancos de dados e sistemas de arquivos.

3.3.1 Estrutura da Classe

A implementação da árvore B é composta pelos seguintes elementos:

- Nós que armazenam chaves e filhos, com folhas contendo registros.
- Métodos para inserção, remoção, busca e divisão de nós quando cheios.

3.3.2 Principais Métodos

Os principais métodos implementados na árvore B incluem:

- **insert**: Insere um novo ID de cliente na árvore, garantindo o balanceamento.
- **remove**: Remove um ID de cliente da árvore.
- **search**: Busca um ID de cliente armazenado.
- **split_child**: Divide um nó cheio para manter a estrutura balanceada.
- **minimo** e **maximo**: Retornam os menores e maiores IDs armazenados.
- **listar_intervalo**: Retorna os registros dentro de um intervalo de valores.
- **contar_registros**: Retorna o número total de registros na árvore sem percorrê-la completamente.

3.3.3 Complexidade das Operações

As complexidades das operações na árvore B são:

- **Inserção**: $O(\log n)$
- **Remoção**: $O(\log n)$
- **Busca**: $O(\log n)$
- **Divisão de nós**: $O(\log n)$

A árvore B mantém operações eficientes mesmo com grandes volumes de dados devido à estrutura balanceada.

3.3.4 Código-Fonte

Listing 3: Código questão 6

```
import random
import time
class BTreeNode:
    def __init__(self, leaf=False):
        self.leaf = leaf  # Define se o nó é folha
        self.keys = []  # Lista de chaves (IDs dos clientes)
        self.children = []  # Lista de filhos (nós)
        self.records = [] if leaf else None  # Apenas folhas armazenam
            registros
        self.count = 0  # Número de registros no nó e seus filhos

class BTree:
    def __init__(self, t):
        self.root = BTreeNode(True)  # Inicializa a árvore com um nó folha
        self.t = t  # Grau mínimo da árvore B

    def insert(self, id_cliente):
        """Insere um novo ID de cliente na árvore B"""
        root = self.root
        if len(root.keys) == (2 * self.t - 1):  # Se a raiz estiver cheia
            new_root = BTreeNode(False)  # Cria um novo nó raiz
            new_root.children.append(self.root)
            self.split_child(new_root, 0)  # Divide a raiz
            self.root = new_root
        self._insert_non_full(self.root, id_cliente)

    def _insert_non_full(self, node, id_cliente):
        """Insere um ID em um nó que não está cheio"""
        node.count += 1  # Atualiza a contagem
        if node.leaf:
            node.keys.append(id_cliente)
            node.keys.sort()
            node.records.append(id_cliente)
        else:
            i = len(node.keys) - 1
            while i >= 0 and id_cliente < node.keys[i]:
                i -= 1
            i += 1
            if len(node.children[i].keys) == (2 * self.t - 1):  # Se o
                filho estiver cheio
```

```
                self.split_child(node, i)
                if id_cliente > node.keys[i]:
                    i += 1
            self._insert_non_full(node.children[i], id_cliente)

    def split_child(self, parent, i):
        """Divide um filho cheio do nó pai"""
        t = self.t
        node = parent.children[i]
        new_node = BTreeNode(node.leaf)
        parent.keys.insert(i, node.keys[t - 1])  # Move a chave do meio para o pai
        parent.children.insert(i + 1, new_node)  # Adiciona um novo filho
        new_node.keys = node.keys[t:]  # Metade superior das chaves vai para o novo nó
        node.keys = node.keys[:t - 1]  # Metade inferior permanece
        if node.leaf:
            new_node.records = node.records[t:]
            node.records = node.records[:t]
        else:
            new_node.children = node.children[t:]
            node.children = node.children[:t]
        new_node.count = len(new_node.keys) if new_node.leaf else sum(child.count for child in new_node.children)
        node.count = len(node.keys) if node.leaf else sum(child.count for child in node.children)
        parent.count = sum(child.count for child in parent.children)

    def search(self, node, id_cliente):
        """Busca um ID de cliente na árvore"""
        if node is None:
            return None
        i = 0
        while i < len(node.keys) and id_cliente > node.keys[i]:
            i += 1
        if i < len(node.keys) and id_cliente == node.keys[i]:
            return node.records[i] if node.leaf else self.search(node.children[i + 1], id_cliente)
        return self.search(node.children[i], id_cliente) if not node.leaf else None

    def remove(self, id_cliente):
        """Remove um ID de cliente da árvore"""
        self._remove(self.root, id_cliente)

    def _remove(self, node, id_cliente):
        """Método auxiliar para remoção"""
        if node.leaf:
            if id_cliente in node.keys:
                idx = node.keys.index(id_cliente)
                node.keys.pop(idx)
                node.records.pop(idx)
                node.count -= 1
        else:
            i = 0
            while i < len(node.keys) and id_cliente > node.keys[i]:
```

```
                i += 1
            if i < len(node.keys) and node.keys[i] == id_cliente:
                node.keys.pop(i)
                node.children[i].count -= 1
            else:
                self._remove(node.children[i], id_cliente)
            node.count -= 1

    def update(self, id_cliente, novo_id):
        """Atualiza um ID de cliente na árvore"""
        self.remove(id_cliente)
        self.insert(novo_id)

    def minimo(self):
        """Retorna o menor ID armazenado"""
        node = self.root
        while not node.leaf:
            node = node.children[0]
        return node.keys[0]

    def maximo(self):
        """Retorna o maior ID armazenado"""
        node = self.root
        while not node.leaf:
            node = node.children[-1]
        return node.keys[-1]

    def listar_intervalo(self, inicio, fim):
        """Retorna os IDs dentro do intervalo especificado"""
        result = []
        self._listar_intervalo(self.root, inicio, fim, result)
        return result

    def _listar_intervalo(self, node, inicio, fim, result):
        if node is None:
            return
        i = 0
        while i < len(node.keys) and node.keys[i] < inicio:
            i += 1
        while i < len(node.keys) and node.keys[i] <= fim:
            if node.leaf:
                result.append(node.records[i])
            else:
                self._listar_intervalo(node.children[i], inicio, fim,
                    result)
                result.append(node.keys[i])
            i += 1
        if not node.leaf:
            self._listar_intervalo(node.children[i], inicio, fim, result)

    def contar_registros(self):
        """Retorna o número total de registros na árvore sem percorrê-la
            completamente"""
        return self.root.count

# Exemplo de uso
```

```
b_tree = BTree(3)
b_tree.insert(10)
b_tree.insert(20)
b_tree.insert(5)
b_tree.insert(6)
b_tree.insert(12)
b_tree.insert(30)
b_tree.insert(7)
b_tree.insert(17)

print("Mínimo:", b_tree.minimo())
print("Máximo:", b_tree.maximo())
print("Buscar 12:", b_tree.search(b_tree.root, 12))
print("Listar intervalo 6-17:", b_tree.listar_intervalo(6, 17))
print("Total de registros:", b_tree.contar_registros())

# Gerador aleatório e análise de tempo
tamanhos = [100, 1000, 10000, 1000000]
for tamanho in tamanhos:
    ids = [random.randint(1, 1000000) for _ in range(tamanho)]

    # Medir tempo de inserção
    b_tree = BTree(3)
    inicio = time.time()
    for id_cliente in ids:
        b_tree.insert(id_cliente)
    fim = time.time()
    print(f"Tempo para inserir {tamanho} elementos: {fim - inicio:.6f} segundos")

    # Medir tempo de remoção
    inicio = time.time()
    for id_cliente in ids:
        b_tree.remove(id_cliente)
    fim = time.time()
    print(f"Tempo para remover {tamanho} elementos: {fim - inicio:.6f} segundos")
```

3.3.5 Análise de Desempenho

Para avaliar o desempenho da árvore B, testamos as operações de inserção e remoção em conjuntos de dados de diferentes tamanhos (100, 1.000, 10.000 e 1.000.000 elementos). Os tempos médios obtidos foram:

Tamanho do Conjunto	Tempo de Inserção (s)	Tempo de Remoção (s)
100	0.000156	0.000102
1.000	0.001908	0.000852
10.000	0.027081	0.012141
1.000.000	10.627495	1.856319

Tabela 3: Tempos médios de inserção e remoção na Árvore B

Os resultados mostram que a árvore B mantém tempos eficientes para inserção e remoção, sendo uma excelente opção para armazenamento e recuperação de grandes quantidades de dados.

3.4 Questão 8 - Estrutura de Dados de Hash

A estrutura de dados *Tabela Hash* permite armazenar pares chave-valor de forma eficiente, proporcionando rápidas operações de inserção, remoção e busca. Essa implementação utiliza *encadeamento separado* para tratar colisões e rehashing para otimizar o desempenho.

3.4.1 Estrutura da Classe

A implementação da classe `MyHash` é composta pelos seguintes elementos:

- Um array que armazena as listas encadeadas para resolver colisões.
- Métodos para inserção, remoção, busca, rehashing e contagem de colisões.
- Implementação de um fator de carga para determinar quando o rehashing deve ser aplicado.

3.4.2 Principais Métodos

Os principais métodos implementados na tabela hash incluem:

- **inserir**: Insere um par chave-valor na tabela, realizando rehashing quando necessário.
- **remover**: Remove um par chave-valor da tabela.
- **buscar**: Retorna o valor associado a uma chave específica.
- **verificar_colisao**: Verifica se uma chave sofreu colisão.
- **contar_colisoes**: Retorna o número total de colisões ocorridas.
- **__rehash**: Realiza a duplicação da capacidade da tabela e reinsere os elementos.

3.4.3 Complexidade das Operações

As complexidades das operações na tabela hash são:

- **Inserção**: $O(1)$ em média, $O(n)$ no pior caso (quando ocorre rehashing).
- **Remoção**: $O(1)$ em média, $O(n)$ no pior caso.
- **Busca**: $O(1)$ em média, $O(n)$ no pior caso.
- **Rehashing**: $O(n)$ devido à necessidade de reinserir todos os elementos.

3.4.4 Código-Fonte

Listing 4: Código questão 8

```
import random
import time

class Node:
    def __init__(self, key, value):
        self.key = key
        self.value = value
        self.next = None


class MyHash:
    def __init__(self, capacity):
        self.capacity = capacity
        self.size = 0
```

```
        self.table = [None] * capacity
        self.collisions = 0  # Contador de colisões

    def _hash_function(self, key):
        return hash(key) % self.capacity

    def inserir(self, key, value):
        index = self._hash_function(key)

        if self.table[index] is None:
            self.table[index] = Node(key, value)
            self.size += 1
        else:
            current = self.table[index]
            while current:
                if current.key == key:
                    current.value = value  # Atualiza o valor se a chave já
                        existe
                    return
                current = current.next
            new_node = Node(key, value)
            new_node.next = self.table[index]  # Insere no início da lista
               encadeada
            self.table[index] = new_node
            self.size += 1
            self.collisions += 1  # Incrementa o contador de colisões

            # Verifica se precisa fazer rehashing
            if self.size > self.capacity * 0.75:  # Limiar de 75%
                self._rehash()

    def remover(self, key):
        index = self._hash_function(key)

        previous = None
        current = self.table[index]

        while current:
            if current.key == key:
                if previous:
                    previous.next = current.next
                else:
                    self.table[index] = current.next
                self.size -= 1
                return
            previous = current
            current = current.next

        raise KeyError(key)

    def buscar(self, key):
        index = self._hash_function(key)

        current = self.table[index]
        while current:
            if current.key == key:
```

```
                return current.value
            current = current.next

        raise KeyError(key)

    def tamanho(self):
        return self.size

    def verificar_colisao(self, key):
        index = self._hash_function(key)
        return self.table[index] is not None and self.table[index].next is
            not None  # Verifica se há mais de um nó no índice

    def _rehash(self):
        new_capacity = self.capacity * 2
        new_table = [None] * new_capacity
        old_table = self.table
        self.capacity = new_capacity
        self.table = new_table
        self.size = 0
        self.collisions = 0  # Reseta o contador de colisoes

        for i in range(len(old_table)):
            current = old_table[i]
            while current:
                self.inserir(current.key, current.value)  # Reinsere os
                    elementos na nova tabela
                current = current.next

    def contar_colisoes(self):
        return self.collisions

# Função para medir o tempo de execução de uma operação
def measure_time(operation, *args):
    start_time = time.time()
    operation(*args)
    end_time = time.time()
    return end_time - start_time

# Função para gerar dados aleatórios
def generate_random_data(size):
    return [(f"Item_{i}", random.randint(1, 100)) for i in range(size)]

# Função para medir o tempo de inclusão de dados na tabela hash
def measure_insertion_time(hash_table, data):
    for key, value in data:
        hash_table.inserir(key, value)

# Função para medir o tempo de remoção de dados da tabela hash
def measure_removal_time(hash_table, data):
    for key, _ in data:
        try:
            hash_table.remover(key)
        except KeyError:
            pass
```

```
# Tamanhos dos dados
sizes = [100, 1000, 10000, 1000000]

# Medir tempos de inclusão e remoção para cada tamanho
for size in sizes:
    data = generate_random_data(size)
    hash_table = MyHash(size * 2)  # Inicializa a tabela hash com o dobro da capacidade para reduzir colisões

    # Medir tempo de inclusão
    insertion_time = measure_time(measure_insertion_time, hash_table, data)
    print(f"Tamanho: {size}, Tempo de inclusão: {insertion_time:.4f} segundos")

    # Medir tempo de remoção
    removal_time = measure_time(measure_removal_time, hash_table, data)
    print(f"Tamanho: {size}, Tempo de remoção: {removal_time:.4f} segundos")

# Driver code
if __name__ == '__main__':
    ht = MyHash(5)

    ht.inserir("lapis", 3)
    ht.inserir("caderno", 2)
    ht.inserir("caneta", 5)
    ht.inserir("borracha", 7)
    ht.inserir("regua", 9)
    ht.inserir("apontador", 11)
    ht.inserir("mochila", 13)

    print("\n*** Verificando se alguns itens estão na tabela: ***\n")
    # método buscar para verificar a existência da chave
    try:
        ht.buscar("lapis")
        print("lapis está na tabela")
    except KeyError:
        print("lapis não está na tabela")

    try:
        ht.buscar("estojo")
        print("estojo está na tabela")
    except KeyError:
        print("estojo não está na tabela")

    print(f"O caderno está no slot {ht.buscar("caderno")}")  # 2

    ht.inserir("caderno", 4)
    print(f"Atualizando o slot de caderno para o slot {ht.buscar("caderno")}")  # 4

    ht.remover("lapis")
    print(f"Tamanho da hash depois de remover 'lapis': {ht.tamanho()}")  # 6

```

```
173     print("\n *** Verificando colisões, se True houve colisão, senão False ***\n")
174     print(f"Caderno: {ht.verificar_colisao("caderno")}")  # True, pois "caderno" colidiu com outros elementos
175     print(f"Borracha: {ht.verificar_colisao("borracha")}")  # True, pois "borracha" colidiu com outros elementos
176     print(f"Apontador: {ht.verificar_colisao("apontador")}")  # True, pois "apontador" colidiu com outros elementos
177     print(f"Mochila: {ht.verificar_colisao("mochila")}")  # True, pois "mochila" colidiu com outros elementos
178     print(f"Estojo: {ht.verificar_colisao("estojo")}")  # False, pois "estojo" não está na tabela
179
180     print(f"\nNúmero de colisões: {ht.contar_colisoes()}")  # Imprime o número de colisões
181
182     print(f"Capacidade da tabela: {ht.capacity}")  # Imprime a capacidade da tabela após o rehashing
```

3.4.5 Análise de Desempenho

Para avaliar o desempenho da tabela hash, testamos as operações de inserção e remoção em conjuntos de dados de diferentes tamanhos (100, 1.000, 10.000 e 1.000.000 elementos). Os tempos médios obtidos foram:

Tamanho do Conjunto	Tempo de Inserção (s)	Tempo de Remoção (s)
100	0.0001	0.0001
1.000	0.0006	0.0003
10.000	0.0062	0.0031
1.000.000	2.0306	0.5344

Tabela 4: Tempos médios de inserção e remoção na Tabela Hash

Os resultados mostram que a tabela hash apresenta tempos eficientes para inserção e remoção na maioria dos casos, sendo impactada apenas quando ocorre rehashing devido ao fator de carga excedido.

3.5 Questão 10 - Estrutura de Dados Heap e Heapsort

A estrutura de dados *Heap* é uma árvore binária que segue a propriedade de heap, onde o valor do nó pai é sempre menor (min-heap) ou maior (max-heap) que seus filhos. Essa estrutura é muito utilizada para implementar filas de prioridade e algoritmos de ordenação eficientes, como o *Heapsort*.

3.5.1 Estrutura da Classe

A implementação da `MinHeap` é composta pelos seguintes elementos:

- Um array que armazena os valores do heap.
- Métodos para inserção, remoção, construção e reorganização do heap.
- Utilização da biblioteca `heapq` para manipulação eficiente da estrutura.

3.5.2 Principais Métodos

Os principais métodos implementados na `MinHeap` incluem:

- **inserir**: Insere um novo elemento no heap mantendo a propriedade da estrutura.
- **remover**: Remove e retorna o menor elemento do heap.

- **construir_heap**: Constrói o heap a partir de uma lista de elementos.
- **heapify**: Reorganiza o heap para garantir sua propriedade.
- **heap_maximo**: Retorna o maior valor presente no heap.

Além disso, foram implementadas funções auxiliares:

- **heapsort**: Algoritmo de ordenação utilizando a estrutura de heap.
- **encontrar_5_maiores**: Retorna os cinco maiores elementos de uma lista utilizando heap.
- **construir_heap_a_partir_de_lista**: Constrói um heap de maneira otimizada.

3.5.3 Complexidade das Operações

As complexidades das operações do heap são:

- **Inserção**: $O(\log n)$
- **Remoção**: $O(\log n)$
- **Construção do heap**: $O(n)$
- **Heapsort**: $O(n \log n)$
- **Encontrar os 5 maiores**: $O(k \log n)$, onde $k = 5$

3.5.4 Código-Fonte

Listing 5: Código questão 10

```
import heapq

class MinHeap:
    def __init__(self):
        self.heap = []

    def inserir(self, valor): # Insere um valor no heap.
        heapq.heappush(self.heap, valor)

    def remover(self): # Remove o valor mínimo (raiz) do heap.
        return heapq.heappop(self.heap)

    def construir_heap(self, lista): # Constrói o heap a partir de uma
        lista de elementos.
        self.heap = lista[:]
        heapq.heapify(self.heap)

    def heapify(self):  # Reorganiza a árvore para garantir a propriedade
        do heap.
        heapq.heapify(self.heap)

    def heap_maximo(self): # Retorna o maior valor presente no heap.
        return max(self.heap)

def heapsort(lista): # Ordena uma lista usando o algoritmo Heapsort.
    heap = MinHeap()
    heap.construir_heap(lista)
    sorted_list = []
```

```
27      while heap.heap:
28          sorted_list.append(heap.remover())
29      return sorted_list
30
31  def encontrar_5_maiores(lista): # Encontra os 5 maiores valores em uma
        lista utilizando heap.
32      heap = MinHeap()
33      heap.construir_heap(lista)
34      return heapq.nlargest(5, heap.heap)
35
36  def construir_heap_a_partir_de_lista(lista): # Constrói um heap a partir de
        uma lista de números sem precisar inserir os elementos um a um.
37      heap = MinHeap()
38      heap.construir_heap(lista)
39      return heap
40
41  # Exemplo de uso
42  if __name__ == "__main__":
43      import random
44      import time
45
46      tamanhos = [100, 1000, 10000, 1000000]
47      for tamanho in tamanhos:
48          lista = [random.randint(1, 1000000) for _ in range(tamanho)]
49
50          # Medir tempo de construção do heap
51          inicio = time.time()
52          heap = construir_heap_a_partir_de_lista(lista)
53          fim = time.time()
54          print(f"Tempo␣para␣construir␣heap␣com␣{tamanho}␣elementos:␣{fim␣-␣
                inicio:.6f}␣segundos")
55
56          # Medir tempo de heapsort
57          inicio = time.time()
58          sorted_list = heapsort(lista)
59          fim = time.time()
60          print(f"Tempo␣para␣heapsort␣com␣{tamanho}␣elementos:␣{fim␣-␣inicio
                :.6f}␣segundos")
61
62          # Medir tempo para encontrar os 5 maiores
63          inicio = time.time()
64          maiores = encontrar_5_maiores(lista)
65          fim = time.time()
66          print(f"Tempo␣para␣encontrar␣os␣5␣maiores␣com␣{tamanho}␣elementos:␣
                {fim␣-␣inicio:.6f}␣segundos")
67
68          # Medir tempo de inserção
69          heap = MinHeap()
70          inicio = time.time()
71          for valor in lista:
72              heap.inserir(valor)
73          fim = time.time()
74          print(f"Tempo␣para␣inserir␣{tamanho}␣elementos:␣{fim␣-␣inicio:.6f}␣
                segundos")
75
76          # Medir tempo de remoção
```

```
77          inicio = time.time()
78          while heap.heap:
79              heap.remover()
80          fim = time.time()
81          print(f"Tempo para remover {tamanho} elementos: {fim - inicio:.6f}
                segundos")
```

3.5.5 Análise de Desempenho

Para avaliar o desempenho da estrutura de heap e do algoritmo Heapsort, testamos as operações em conjuntos de dados de diferentes tamanhos (100, 1.000, 10.000 e 1.000.000 elementos). Os tempos médios obtidos foram:

Tamanho do Conjunto	Construção Heap (s)	Heapsort (s)	Inserção (s)	Remoção (s)
100	0.000005	0.000038	0.000029	0.000035
1.000	0.000036	0.000259	0.000167	0.000223
10.000	0.000262	0.003260	0.001504	0.003058
1.000.000	0.028833	1.291376	0.160977	1.328928

Tabela 5: Tempos médios das operações na estrutura de Heap

Os resultados mostram que a estrutura de heap apresenta um tempo eficiente de inserção e remoção devido à sua natureza de árvore balanceada. O algoritmo Heapsort, apesar de ser um algoritmo de ordenação eficiente, é superado em desempenho pelo *Quicksort* na maioria dos casos práticos. No entanto, sua estabilidade e complexidade garantida fazem dele uma opção viável em diversos cenários.

4 Considerações Finais

Este trabalho permitiu uma análise aprofundada de diferentes estruturas de dados avançadas, explorando suas características, implementações e desempenho em diversos cenários. A implementação e os testes realizados evidenciaram a importância de escolher a estrutura correta para cada tipo de problema, considerando aspectos como tempo de execução, complexidade computacional e eficiência no armazenamento de dados.

As árvores AVL demonstraram-se eficientes para operações de busca, inserção e remoção, mantendo um balanceamento automático que garante um tempo de execução otimizado. No contexto do gerenciamento de rankings, essa estrutura provou ser uma escolha robusta para organizar dados dinamicamente. A árvore B, por sua vez, mostrou-se altamente eficaz para grandes volumes de dados, sendo amplamente utilizada em bancos de dados devido à sua capacidade de minimizar o número de acessos ao disco.

A tabela hash destacou-se pelo desempenho extremamente rápido em buscas, inserções e remoções, desde que a função de dispersão seja bem projetada e o tratamento de colisões seja eficiente. Já a estrutura de heap e o algoritmo Heapsort foram analisados em relação à sua aplicabilidade em filas de prioridade e ordenação, evidenciando sua estabilidade e previsibilidade de desempenho.

Além do aprendizado técnico, este trabalho proporcionou uma experiência valiosa na divisão de tarefas, colaboração em equipe e utilização de ferramentas para versionamento e compartilhamento de código. A metodologia adotada possibilitou um desenvolvimento estruturado e eficiente, garantindo uma análise comparativa detalhada do desempenho das diferentes estruturas estudadas.

Por fim, os resultados obtidos reforçam a relevância das estruturas de dados na computação moderna, evidenciando como uma escolha bem fundamentada pode impactar significativamente a eficiência e escalabilidade de aplicações computacionais.

Referências

Material de estudo disponibilizado no Sagaa UFC pelo professor

O Que é Árvore AVL, Propriedades da Árvore AVL e Por que AVL é Balanceada | Estrutura de Dados 19 - URL: https://youtu.be/l8IBdCb2BWA?si=HEXxzRjQMRAkXeCd

Árvores: O Começo de TUDO | Estruturas de Dados e Algoritmos - URL: https://youtu.be/9GdesxWtOgs?si=cqVTD-TYXWIGmoWE

Tabelas Hash - Estrutura de Dados - Unicamp - URL: https://youtu.be/Non0I$_{O}St9o?si = bTNUXbGkI0I3_{a}GL$

Binary Heap e HeapSort - O que são e COMO FUNCIONAM (passo-a-passo) - URL: https://youtu.be/-nq88TldUX0?si=-YSbfHA$_{Q}ts7VT79$

Python is used successfully in thousands of real-world business applications around the world, including many large and mission critical systems. Here are some quotes from happy Python users - URL: https://www.python.org