AULA 08: Exercício teórico Métodos de Pesquisa e Árvores AVL

Aluno: Gian Franco Joel Condori Luna

October 23, 2024

Exercices

1 (0,4) Insira as seguintes chaves: 924, 220, 911, 244, 898, 258, 362, 363, 360, 350.

a) em uma árvore binária comum.

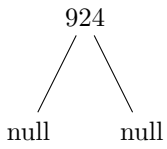
b) depois em uma árvore AVL mostrando as rotações passo-a-passo.

c) remova o nó 362 em ambas as árvores.

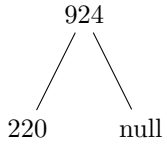
Solução:

a) em uma árvore binária comum.

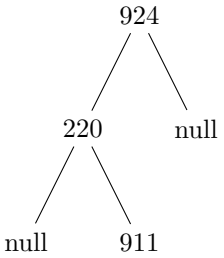
1. Inserindo 924:



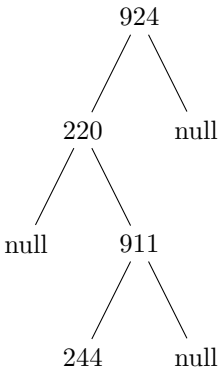
2. Inserindo 220:



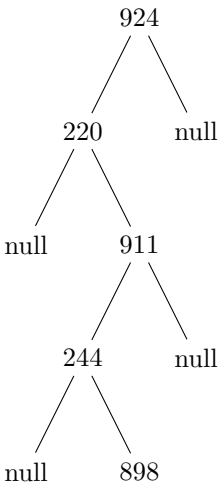
3. Inserindo 911:



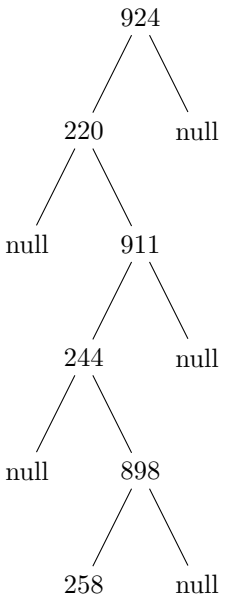
4. Inserindo 244:



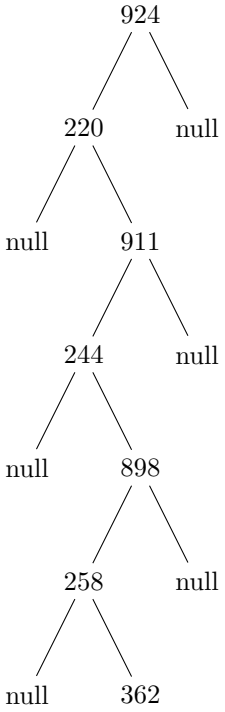
5. Inserindo 898:



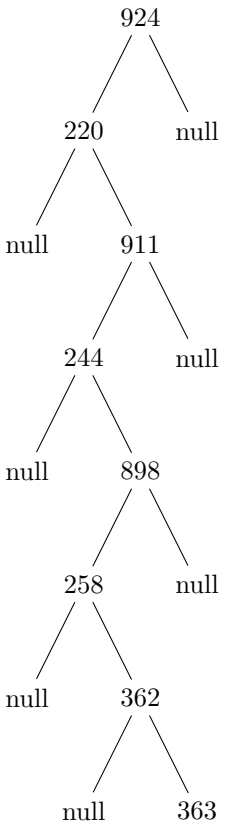
6. Inserindo 258:



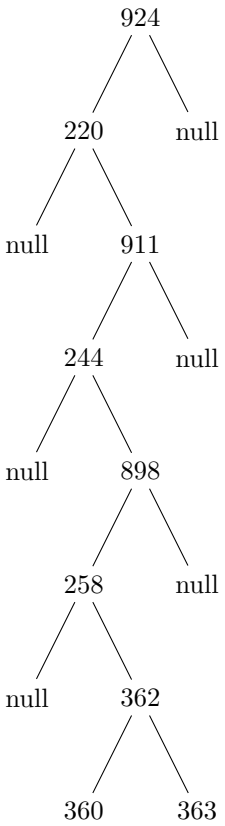
7. Inserindo 362:



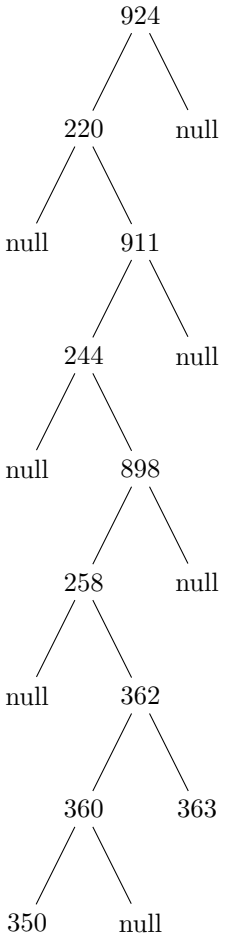
8. Inserindo 363:



9. Inserindo 360:

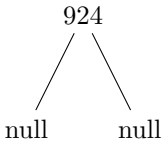


10. Inserindo 350:

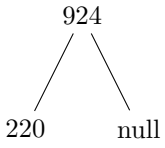


b) depois em uma árvore AVL mostrando as rotações passo-a-passo.

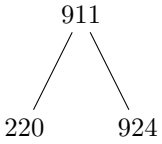
1. Inserindo 924:



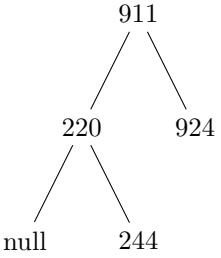
2. Inserindo 220:



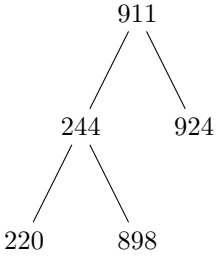
3. Inserindo 911:



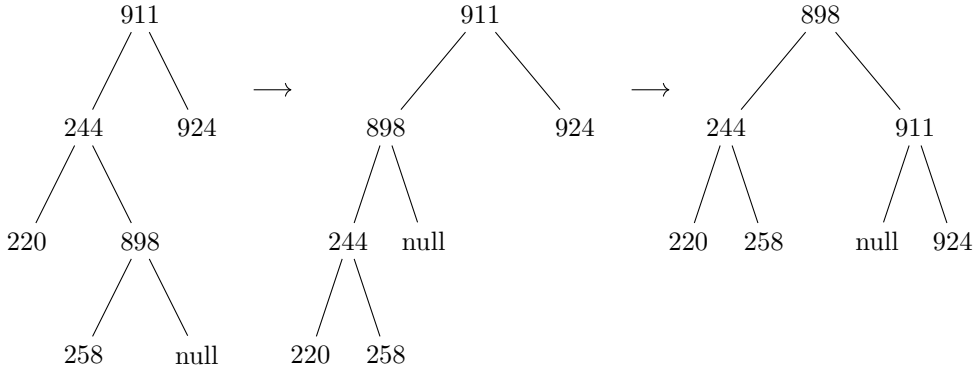
4. Inserindo 244:



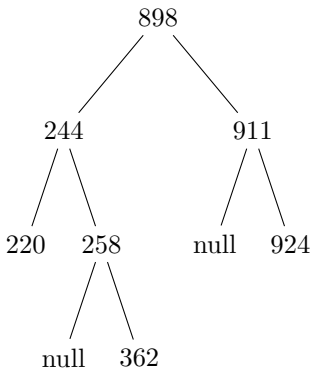
5. Inserindo 898:



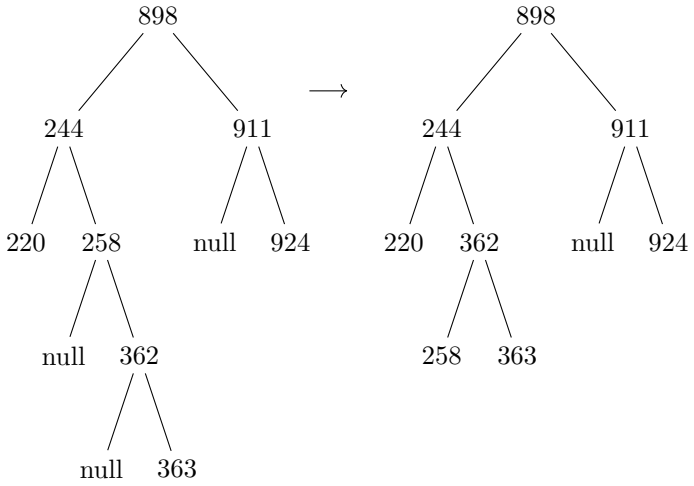
6. Inserindo 258:



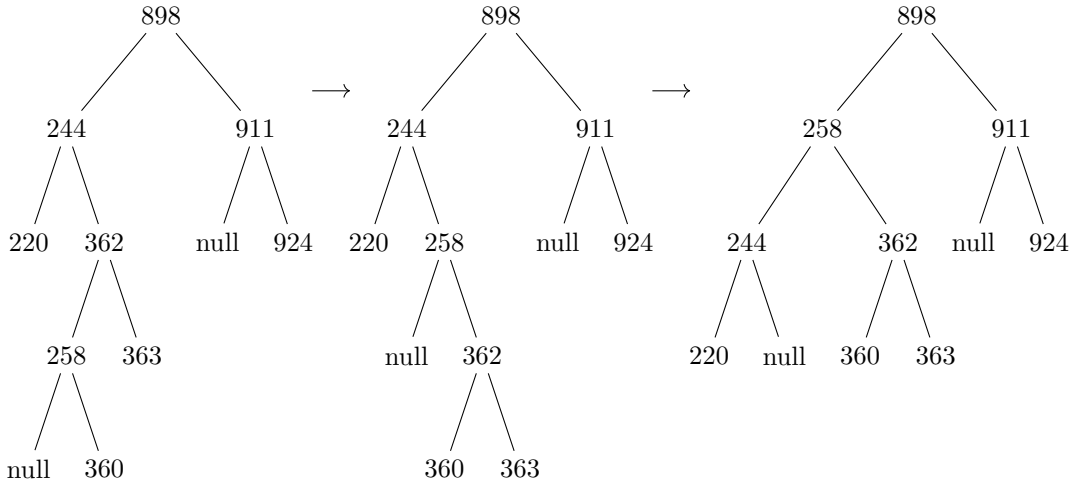
7. Inserindo 362:



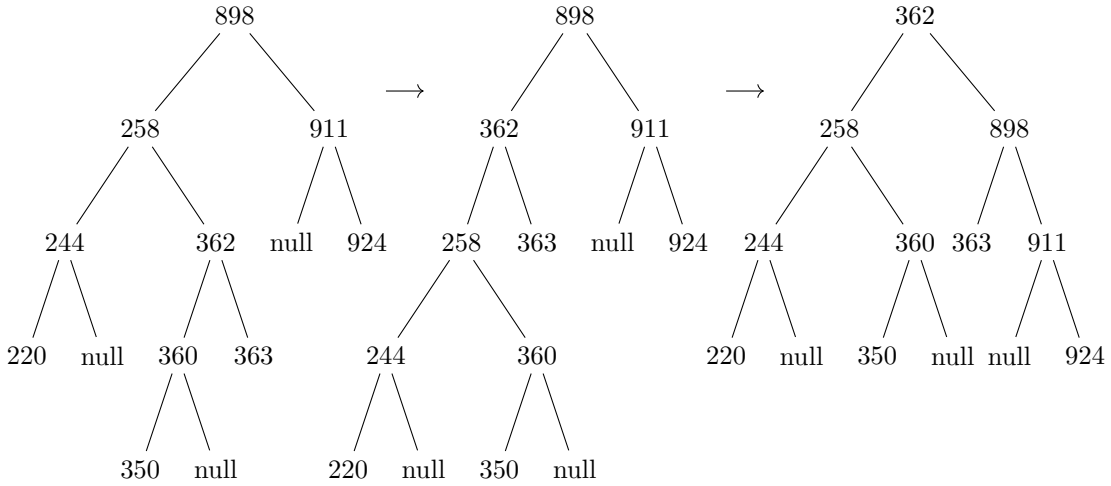
8. Inserindo 363:



9. Inserindo 360:

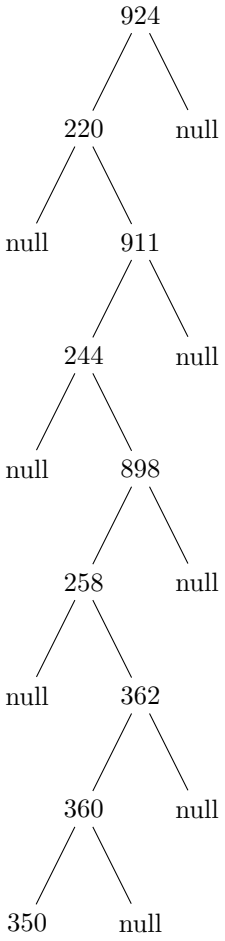


10. Inserindo 350:

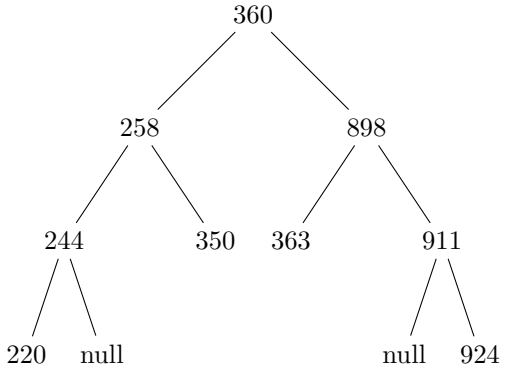


c) remova o nó 362 em ambas as árvores.

1. Removendo o nó 362 da árvore binária:



2. Removendo o nó 362 da árvore AVL:



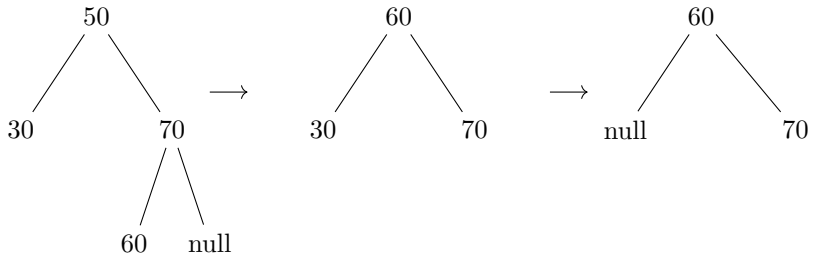
2 (0,2) A operação de eliminação é comutativa, ie, a eliminação de x e depois y resulta na mesma árvore que a eliminação de y e depois x? Mostre um contra- exemplo.

Solução:

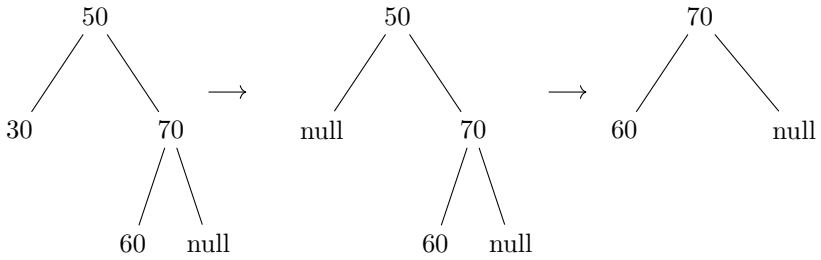
Fonte: https://www.cs.cornell.edu/courses/cs409/2000SP/Homework/hw03Solution.htm

Delete não é comutativo. O exemplo abaixo usa a regra de que a exclusão de um nó com dois filhos é feita excluindo seu sucessor.

1. Removendo o nó 50 depois a nó 30:



2. Removendo o nó 30 depois a nó 50:



3 (0,4) Escreva o código de remoção em árvore AVL. Explique quais são as principais diferenças comparadas ao algoritmo de remoção em árvore binária.

Solução:

Fonte: ChatGPT

A remoção em uma árvore AVL segue o mesmo princípio de remoção de uma árvore binária de busca (BST - Binary Search Tree), mas com o adicional de reequilibrar a árvore após a remoção, para manter a propriedade de balanceamento da árvore AVL, onde a diferença de altura entre as subárvores esquerda e direita de qualquer nó não pode ser maior que 1.

```python
# FONTE: ChatGPT
class Node:
  def __init__(self, key):
    self.key = key
    self.left = None
    self.right = None
    self.height = 1

class AVLTree:

  # Fun\c{c}ão para calcular a altura de um nó
  def get_height(self, node):
    if not node:
      return 0
    return node.height

  # Fun\c{c}ão para calcular o fator de balanceamento
  def get_balance(self, node):
    if not node:
      return 0
    return self.get_height(node.left) - self.get_height(node.right)

  # Rota\c{c}ão à direita
  def rotate_right(self, y):
    x = y.left
    T2 = x.right

    # Realiza a rota\c{c}ão
    x.right = y
    y.left = T2

    # Atualiza as alturas
    y.height = 1 + max(self.get_height(y.left), self.get_height(y.
        right))
    x.height = 1 + max(self.get_height(x.left), self.get_height(x.
        right))

    # Retorna a nova raiz
    return x

  # Rota\c{c}ão à esquerda
  def rotate_left(self, x):
    y = x.right
    T2 = y.left

    # Realiza a rota\c{c}ão
    y.left = x
    x.right = T2

    # Atualiza as alturas
    x.height = 1 + max(self.get_height(x.left), self.get_height(x.
        right))
    y.height = 1 + max(self.get_height(y.left), self.get_height(y.
        right))

    # Retorna a nova raiz
    return y
```

```
54
55    # Fun\c{c}ão para inserir um nó (usada para criar a árvore antes
          da remo\c{c}ão)
56    def insert(self, root, key):
57      if not root:
58        return Node(key)
59      elif key < root.key:
60        root.left = self.insert(root.left, key)
61      else:
62        root.right = self.insert(root.right, key)
63
64      # Atualiza a altura do nó pai
65      root.height = 1 + max(self.get_height(root.left), self.
            get_height(root.right))
66
67      # Verifica o balanceamento
68      balance = self.get_balance(root)
69
70      # Caso LL
71      if balance > 1 and key < root.left.key:
72        return self.rotate_right(root)
73
74      # Caso RR
75      if balance < -1 and key > root.right.key:
76        return self.rotate_left(root)
77
78      # Caso LR
79      if balance > 1 and key > root.left.key:
80        root.left = self.rotate_left(root.left)
81        return self.rotate_right(root)
82
83      # Caso RL
84      if balance < -1 and key < root.right.key:
85        root.right = self.rotate_right(root.right)
86        return self.rotate_left(root)
87
88      return root
89
90    # Fun\c{c}ão para encontrar o nó com valor mínimo (usada durante
          a remo\c{c}ão)
91    def find_min_value_node(self, node):
92      current = node
93      while current.left is not None:
94        current = current.left
95      return current
96
97    # Fun\c{c}ão para remover um nó em uma árvore AVL
98    def remove(self, root, key):
99      # Passo 1: Realiza a remo\c{c}ão padrão da árvore binária de
            busca
100     if not root:
101       return root
102     elif key < root.key:
103       root.left = self.remove(root.left, key)
104     elif key > root.key:
105       root.right = self.remove(root.right, key)
106     else:
```

```
107         # Nó encontrado, realiza a remo\c{c}ão
108         if root.left is None:
109             temp = root.right
110             root = None
111             return temp
112         elif root.right is None:
113             temp = root.left
114             root = None
115             return temp
116 
117         # Nó com dois filhos: pegar o sucessor in-order
118         temp = self.find_min_value_node(root.right)
119         root.key = temp.key
120         root.right = self.remove(root.right, temp.key)
121 
122     # Se a árvore tiver apenas um nó, retorna
123     if root is None:
124         return root
125 
126     # Passo 2: Atualiza a altura do nó atual
127     root.height = 1 + max(self.get_height(root.left), self.
            get_height(root.right))
128 
129     # Passo 3: Calcula o fator de balanceamento
130     balance = self.get_balance(root)
131 
132     # Passo 4: Verifica se o nó está desbalanceado e realiza rota\c
            {c}ões apropriadas
133 
134     # Caso LL
135     if balance > 1 and self.get_balance(root.left) >= 0:
136         return self.rotate_right(root)
137 
138     # Caso LR
139     if balance > 1 and self.get_balance(root.left) < 0:
140         root.left = self.rotate_left(root.left)
141         return self.rotate_right(root)
142 
143     # Caso RR
144     if balance < -1 and self.get_balance(root.right) <= 0:
145         return self.rotate_left(root)
146 
147     # Caso RL
148     if balance < -1 and self.get_balance(root.right) > 0:
149         root.right = self.rotate_right(root.right)
150         return self.rotate_left(root)
151 
152     return root
```

Principais Diferenças:

a) Balanceamento: A árvore AVL mantém um fator de balanceamento (diferença de alturas entre as subárvores esquerda e direita) de nó máximo 1, realizando rotações após a remoção para garantir essa propriedade. A árvore binária comum não possui essa garantia.

b) Rotações: Na árvore AVL, podem ser necessárias rotações para corrigir o

balanceamento após a remoção, o que não ocorre em árvores binárias de busca simples.

c) Eficiência: Devido ao balanceamento, a árvore AVL mantém sua altura controlada em O(log n), enquanto a árvore binária de busca pode crescer desbalanceada, levando a uma altura O(n) no pior caso.