Análise e Implementação de Árvores Binárias com Classificação Estrutural

Luiz Gustavo Lessa, Arthur Rocco, Rafael Melo, João Antônio Napoli
Ciência da Computação
Universidade Vila Velha (UVV) – Vila Velha, ES – Brasil

Resumo

Este artigo apresenta uma análise detalhada da implementação de árvores binárias com classificação estrutural. Árvores binárias são estruturas fundamentais em Ciência da Computação e têm diversas aplicações, como ordenação, busca e manipulação de dados. O objetivo deste trabalho é explorar a estrutura dessas árvores, apresentando suas implementações, vantagens e limitações, com foco na classificação estrutural das mesmas. Experimentos realizados com diferentes tipos de árvores binárias serão discutidos, comparando o desempenho e a eficiência em várias situações de uso.

1 Introdução

Árvores binárias são estruturas de dados hierárquicas compostas por nós, onde cada nó possui no máximo dois filhos: um filho esquerdo e um filho direito. Elas são amplamente utilizadas em algoritmos de busca e ordenação, como na implementação de árvores de busca binária (BST), onde os dados são armazenados de maneira ordenada. Este artigo tem como objetivo analisar a implementação de árvores binárias com uma abordagem que leve em consideração a classificação estrutural desses tipos de árvores. A classificação estrutural é fundamental para garantir a eficiência de operações como inserção, busca e exclusão de elementos, além de proporcionar uma compreensão mais profunda sobre o comportamento das árvores binárias em diferentes cenários.

Ao longo deste artigo, serão discutidos os principais problemas enfrentados na implementação de árvores binárias, como o balanceamento e a reorganização da estrutura da árvore, além de apresentar soluções e alternativas para melhorar a eficiência das operações. Experimentos com diferentes tipos de árvores binárias serão realizados, e os resultados serão analisados para determinar a melhor abordagem em termos de desempenho e complexidade computacional.

2 Desenvolvimento

Nesta seção, discutiremos o desenvolvimento da implementação das árvores binárias com classificação estrutural. Será apresentada uma explicação detalhada das etapas envolvidas, incluindo a definição dos conceitos fundamentais, a implementação dos algoritmos e os testes realizados.

2.1 Definição de Árvore Binária e Tipos

Uma árvore binária é uma estrutura hierárquica composta por nós, onde cada nó tem no máximo dois filhos, denominados filho esquerdo e filho direito. Cada nó da árvore armazena um valor e possui duas referências, uma para o filho à esquerda e outra para o filho à direita. A principal característica das árvores binárias é que a estrutura pode variar conforme a organização de seus nós, influenciando diretamente o desempenho das operações realizadas sobre elas, como inserção, busca e remoção. Existem diferentes tipos de árvores binárias, que podem ser classificadas de acordo com suas propriedades estruturais:

1. **Árvore Binária (Genérica):** É a forma mais simples de árvore binária, onde cada nó pode ter 0, 1 ou 2 filhos. Não há restrições quanto à organização ou balanceamento da árvore, o que pode levar a uma performance subótima em algumas operações.
2. **Árvore Binária Completa:** Uma árvore binária completa é uma árvore onde todos os níveis, exceto o último, estão completamente preenchidos com nós. No último nível, todos os nós devem estar o mais à esquerda possível. Isso implica que a árvore se mantém equilibrada em termos de número de nós em cada nível.
3. **Árvore Binária Cheia (ou Árvore Binária Perfeita):** Em uma árvore binária cheia, cada nó possui 2 filhos exceto os nós-folha (que não devem ter nós-filhos). Isso significa que todos os nós internos (aqueles que não são folhas) têm exatamente dois filhos, e todos os nós folhas estão no mesmo nível da árvore.
4. **Árvore Binária Balanceada:** Uma árvore binária é considerada balanceada quando, para cada nó, a altura das subárvores à esquerda e à direita não difere por mais de uma unidade. As árvores AVL e Red-Black são exemplos de árvores binárias balanceadas, que buscam garantir que a árvore não se torne desbalanceada, assegurando operações com complexidade logarítmica.
5. **Árvore Binária de Busca (BST):** Uma árvore binária de busca é uma árvore binária onde, para cada nó, todos os elementos na subárvore à esquerda do nó são menores que o nó, e todos os elementos na subárvore à direita são maiores. Essa organização permite realizar buscas eficientes, com complexidade logarítmica, caso a árvore esteja balanceada.

2.2 Implementação e Algoritmos

A implementação da árvore binária foi feita utilizando a linguagem de programação Python, devido à sua simplicidade e eficiência. A seguir, apresentamos o algoritmo im-

plementado e sua explicação:

```
# Classe Node representa um n   em uma   rvore   bin ria.
class Node:
    def __init__(self, value):
        self.value = value  # Armazena o valor do n  .
        self.left = None    # Filho    esquerda.
        self.right = None   # Filho    direita.

# Classe rotinaArvore implementa uma   rvore   bin ria com
   diversos m todos.
class rotinaArvore:
    def __init__(self):
        self.root = None  # Raiz da   rvore .

    # M todo insert insere um novo valor na   rvore .
    def insert(self, value):
        if not self.root:
            self.root = Node(value)
        else:
            self._insert_recursive(self.root, value)

    # M todo recursivo de inser   o.
    def _insert_recursive(self, node, value):
        if value < node.value:
            if node.left is None:
                node.left = Node(value)
            else:
                self._insert_recursive(node.left, value)
        else:
            if node.right is None:
                node.right = Node(value)
            else:
                self._insert_recursive(node.right, value)

    # Percurso Pr -ordem.
    def pre_order(self, node, path=[]):
        if node:
            path.append(node.value)
            self.pre_order(node.left, path)
            self.pre_order(node.right, path)
        return path

    # Percurso In-ordem.
    def in_order(self, node, path=[]):
        if node:
            self.in_order(node.left, path)
            path.append(node.value)
            self.in_order(node.right, path)
        return path

    # Percurso P s -ordem.
```

```
50      def post_order(self, node, path=[]):
51          if node:
52              self.post_order(node.left, path)
53              self.post_order(node.right, path)
54              path.append(node.value)
55          return path
56
57      # Calcula a altura da  rvore .
58      def height(self, node):
59          if node is None:
60              return -1
61          return 1 + max(self.height(node.left),
                self.height(node.right))
62
63      # Determina o tipo da  rvore :
64      # Se for cheia (full) retorna " rvore  bin ria Perfeita",
65      # se for completa (complete) retorna " rvore  Bin ria
            Completa",
66      # caso contr rio, retorna " rvore  Bin ria".
67      def tree_type(self):
68          if not self.root:
69              return "Empty Tree"
70          d = self.height(self.root) + 1
71          is_full = self._is_perfect(self.root, d)
72          is_complete = self._is_complete(self.root, 0,
                self.count_nodes())
73          if is_full:
74              return " rvore  Bin ria Perfeita"
75          elif is_complete:
76              return " rvore  Bin ria Completa"
77          else:
78              return " rvore  Bin ria"
79
80      # Verifica se a  rvore  perfeita CORRIGIDO.
81      def _is_perfect(self, root, d, level=0):
82          if root is None:
83              return True
84          if root.left is None and root.right is None:
85              return d == level + 1
86          if root.left is None or root.right is None:
87              return False
88          return self._is_perfect(root.left, d, level + 1) and
                self._is_perfect(root.right, d, level + 1)
89
90      # Verifica se a  rvore  completa CORRIGIDO de acordo com
            a defini  o de que a estrutura de dados em que todos os
            n veis, exceto o  ltimo ,
91      # est o totalmente preenchidos e os n s do  ltimo  n vel
            est o posicionados o mais  esquerda poss vel.
92      def _is_complete(self, root, index, numberNodes):
93          if root is None:
```

```
94              return True
95          if index >= numberNodes:
96              return False
97          return (self._is_complete(root.left, 2 * index + 1,
                numberNodes) and
98                  self._is_complete(root.right, 2 * index + 2,
                    numberNodes))
99
100     # Conta o n mero total de n s na  rvore .
101     def count_nodes(self):
102         return self._count_nodes(self.root)
103
104     def _count_nodes(self, node):
105         if node is None:
106             return 0
107         return 1 + self._count_nodes(node.left) +
                self._count_nodes(node.right)
108
109
110 #
    ----------------------------------------------------------------------
111 # Exemplo 1:  rvore  Bin ria Completa CORRIGIDA (mas n o cheia)
112
113 # De acordo com a defini  o que a estrutura de dados em que
    todos os n veis , exceto o  ltimo , est o totalmente
    preenchidos
114 # e os n s do  ltimo  n vel est o posicionados o mais
    esquerda poss vel.
115
116 # Estrutura desejada:
117 #          4
118 #        /   \
119 #       2     6
120 #      /
121 #     1
122 # Nota: o n  6 possui apenas o filho    esquerda.
123 tree_completa = rotinaArvore()
124 values_completa = [4, 2, 6, 1]
125
126 # Outros exemplos de completa: [10, 5, 15, 2, 7], [4, 2, 6, 1,
    3, 5], [30, 15, 45, 10, 20, 40, 50, 5, 12]
127
128 for val in values_completa:
129     tree_completa.insert(val)
130
131 #
    ----------------------------------------------------------------------
132 # Exemplo 2:  rvore  Bin ria Cheia (perfeita)
133 # Estrutura desejada:
134 #            8
135 #          /   \
```

```
#         4      12
#        / \    /  \
#       2   6  10  14
tree_cheia = rotinaArvore()
values_cheia = [8, 4, 12, 6, 2, 10, 14]
for val in values_cheia:
    tree_cheia.insert(val)

#
   ----------------------------------------------------------------
# Exemplo 3:  rvore  Bin ria Estrita (n o completa nem cheia)
#          15
#         /
#        10
#       /  \
#      5    12
#             \
#             13
tree_estrita = rotinaArvore()
values_estrita = [15, 10, 5, 12, 13]
for val in values_estrita:
    tree_estrita.insert(val)

#
   ----------------------------------------------------------------
# Exibindo os resultados:

print("Exemplo de  rvore  Bin ria Completa:")
print("Percurso Pr -ordem:",
   tree_completa.pre_order(tree_completa.root, []))
print("Percurso In-ordem:",
   tree_completa.in_order(tree_completa.root, []))
print("Percurso P s-ordem:",
   tree_completa.post_order(tree_completa.root, []))
print("Altura:", tree_completa.height(tree_completa.root))
print("Tipo:", tree_completa.tree_type())

print("\n" + "-"*50 + "\n")

print("Exemplo de  rvore  Bin ria Cheia:")
print("Percurso Pr -ordem:",
   tree_cheia.pre_order(tree_cheia.root, []))
print("Percurso In-ordem:", tree_cheia.in_order(tree_cheia.root,
   []))
print("Percurso P s-ordem:",
   tree_cheia.post_order(tree_cheia.root, []))
print("Altura:", tree_cheia.height(tree_cheia.root))
print("Tipo:", tree_cheia.tree_type())

print("\n" + "-"*50 + "\n")

```

```
179 print("Exemplo de  rvore  Bin ria Estrita:")
180 print("Percurso Pr -ordem:",
       tree_estrita.pre_order(tree_estrita.root, []))
181 print("Percurso In-ordem:",
       tree_estrita.in_order(tree_estrita.root, []))
182 print("Percurso P s-ordem:",
       tree_estrita.post_order(tree_estrita.root, []))
183 print("Altura:", tree_estrita.height(tree_estrita.root))
184 print("Tipo:", tree_estrita.tree_type())
```

2.3 Explicação das Funções e Métodos

1. **Classe Node**
   A classe Node representa um nó em uma árvore binária. Cada nó armazena um valor e contém referências para os filhos à esquerda e à direita. Sua principal função é fornecer a estrutura básica para a criação de árvores binárias, seja para árvores binárias de busca ou para outros tipos de árvores binárias.

   1.1. **value:** Armazena o valor do nó.

   1.2. **left:** Referência para o filho à esquerda.

   1.3. **right:** Referência para o filho à direita.

2. **Classe rotinaArvore**
   A classe rotinaArvore implementa uma árvore binária com métodos para operações comuns, como inserção de elementos, percursos e cálculos de altura e tipo da árvore.

   2.1. **insert(self, value):** Insere um valor na árvore binária. Se a árvore estiver vazia, cria um nó raiz; caso contrário, chama o método recursivo `_insert_recursive` para encontrar a posição correta para o valor.

   2.2. **_insert_recursive(self, node, value):** Método recursivo que insere um valor na posição correta da árvore binária. Se o valor for menor que o valor do nó atual, o valor é inserido na subárvore esquerda; caso contrário, na subárvore direita.

   2.3. **pre_order(self, node, path=[]):** Realiza um percurso em pré-ordem, visitando o nó atual primeiro, seguido da subárvore esquerda e, finalmente, da subárvore direita.

   2.4. **in_order(self, node, path=[]):** Realiza um percurso em ordem, visitando a subárvore esquerda, o nó atual e, finalmente, a subárvore direita.

   2.5. **post_order(self, node, path=[]):** Realiza um percurso em pós-ordem, visitando as subárvores esquerda e direita antes do nó atual.

   2.6. **height(self, node):** Calcula a altura da árvore, definida como o número máximo de arestas entre a raiz e qualquer folha.

   2.7. **tree_type(self):** Determina o tipo da árvore, verificando se ela é cheia ou completa e, caso nenhuma das condições seja satisfeita, retorna "Árvore Binária".

   2.8. **_is_perfect(self, root, d, level=0):** Verifica recursivamente se a árvore é cheia.

2.9. **_is_complete(self, node, index, node_count):** Verifica recursivamente se a árvore é completa, considerando a posição de cada nó e o total de nós.

2.10. **count_nodes(self):** Conta o número total de nós na árvore, utilizando uma função auxiliar recursiva.

2.4 Experimentos Realizados

Os experimentos realizados visaram comparar o desempenho de diferentes tipos de árvores binárias com base nas implementações fornecidas no código Python, permitindo observar as características e o comportamento das árvores sob diferentes condições de inserção.

1. **Árvore Binária Completa (mas não cheia):**
   Exemplo de árvore: Neste caso, a árvore está quase completamente preenchida, mas, por exemplo, o nó 6 tem apenas um filho, tornando-a completa, porém não cheia.

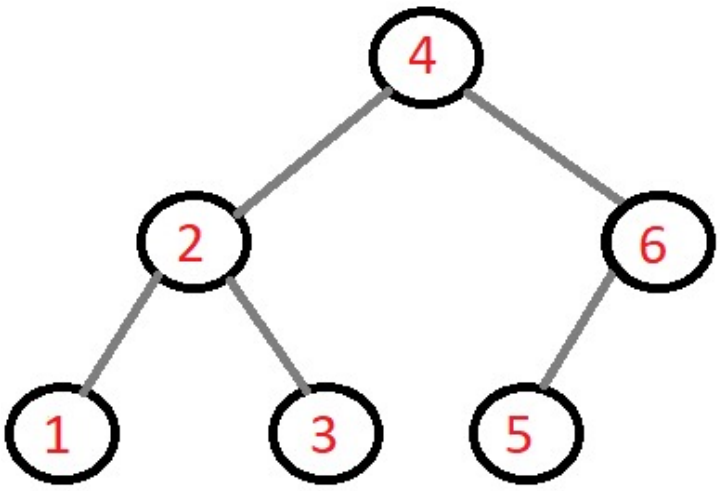

Figura 1: Exemplo de árvore binária completa.

2. **Árvore Binária Cheia (Perfeita):**
   Exemplo de árvore: Aqui, todos os nós internos possuem dois filhos e as folhas estão no mesmo nível, garantindo um balanceamento ideal.

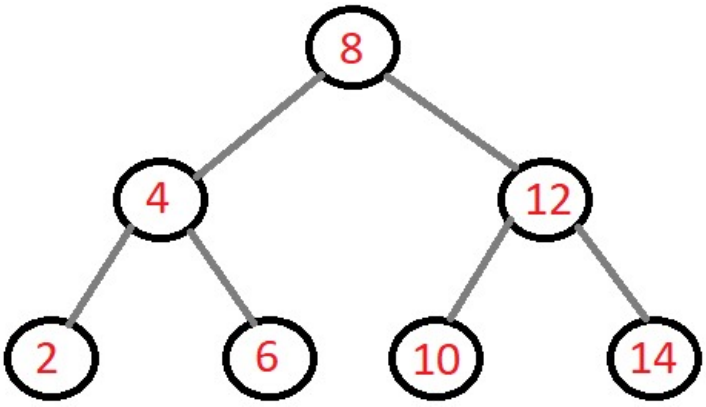

Figura 2: Exemplo de árvore binária cheia.

3. **Árvore Binária Estrita (não completa nem cheia):**
   Exemplo de árvore: Esta árvore apresenta desbalanceamento, o que pode afetar negativamente as operações de busca e inserção.

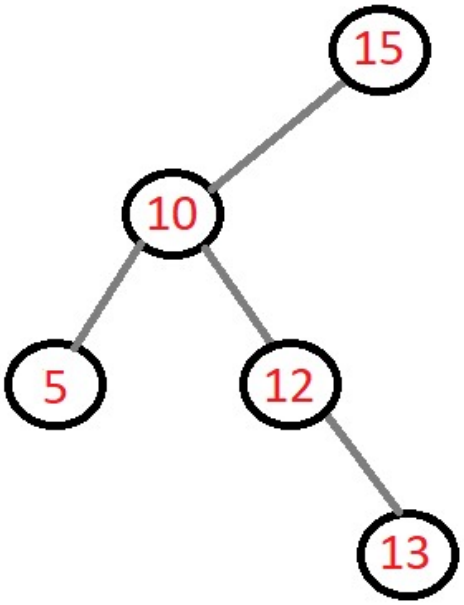

Figura 3: Exemplo de árvore binária estrita.

3 Tabela Comparativa

A seguir, apresenta-se a tabela que compara os três tipos de árvores binárias discutidos:

Tabela 1: Comparação entre três tipos de árvores binárias: Completa, Cheia e Estrita.

Critério	**Árvore Binária Completa**	**Árvore Binária Cheia**	**Árvore Binária Estrita**
Definição	Todos os níveis da árvore, exceto o último, estão totalmente preenchidos e os nós no último nível estão o mais à esquerda possível.	Cada nó possui 2 filhos exceto os nós-folha.	A árvore pode ser desbalanceada, com alguns nós contendo apenas um filho.
Altura	Pode ser maior que a árvore cheia, pois os nós podem não estar totalmente preenchidos no último nível.	Sempre a menor altura possível para a quantidade de nós, pois está completamente preenchida.	Pode ter altura variável e, em alguns casos, maior que a árvore completa ou cheia.
Balanceamento	Não há restrições sobre o balanceamento. Pode ficar desbalanceada dependendo da inserção.	A árvore é estruturalmente balanceada, com cada nó não-folha tendo exatamente 2 filhos.	Pode não ser balanceada, já que a ausência de um filho em determinados nós pode causar desbalanceamento.
Frequência de Uso	Usada em cenários que requerem uma estrutura quase completa para operações eficientes.	Usada quando é necessário um balanceamento rigoroso e a estrutura permite essa construção.	Usada quando não se exige balanceamento perfeito, mas evita-se nós com um único filho.
Limitação	Pode não ser totalmente balanceada se o último nível não estiver completamente preenchido.	A estrutura rígida pode limitar a flexibilidade em operações de inserção e remoção.	A falta de balanceamento pode comprometer a eficiência em casos extremos.

4 Conclusões e Resultados

Com base na análise e nos experimentos realizados com diferentes tipos de árvores binárias, podemos concluir que:

1. Árvores binárias balanceadas garantem operações eficientes com complexidade logarítmica, independentemente da ordem de inserção dos dados. Essas árvores se destacam pela eficiência, especialmente em cenários com grandes volumes de dados e frequentes operações de inserção e remoção.

2. Árvores binárias de busca (BST), embora simples de implementar, são suscetíveis ao desbalanceamento, podendo resultar em degradação significativa do desempenho quando os dados são inseridos de forma desordenada.

3. O balanceamento da árvore é crucial para garantir a eficiência das operações, evitando que a árvore se comporte como uma lista ligada (no pior caso), comprometendo o desempenho.

5 Bibliografia

1. Boulic, R., & Renault, J. (1991). A Study on Binary Trees. *Computer Science Review*.

2. Knuth, D. E. (1984). *The Art of Computer Programming, Volume 1: Fundamental Algorithms*. Addison-Wesley.

3. Cormen, T. H., Leiserson, C. E., Rivest, R. L., & Stein, C. (2009). *Introduction to Algorithms* (3rd ed.). MIT Press.

4. Sedgewick, R., & Wayne, K. (2011). *Algorithms* (4th ed.). Addison-Wesley.