

Relatório Avaliação 02 – Resolução de Problemas Estruturados em Computação

Gabriel Coradin

Gustavo Coradin

Vinicius Guidotti Macedo

Alexandre de Moraes Bueno

1- Introdução

Uma árvore padrão, também conhecida como árvore binária de busca (BST), tem uma maneira especial de organizar seus elementos. Ela garante que todos os elementos à esquerda de um nó sejam menores do que esse nó, e todos os elementos à direita sejam maiores. Isso cria uma estrutura onde você pode encontrar um elemento com eficiência, geralmente em tempo médio de logaritmo base 2 do número de elementos na árvore, o que é bastante rápido. No entanto, se essa árvore não estiver bem equilibrada e se parecer mais com uma lista simples, a busca pode se tornar muito mais lenta, exigindo tempo proporcional ao número total de elementos, o que não é tão eficiente.

Por outro lado, temos as árvores AVL, que são uma variação esperta das árvores de busca binária. Elas são como as BSTs, mas com uma diferença importante: elas automaticamente mantêm um equilíbrio. Isso significa que, a qualquer momento, a diferença na altura entre a subárvore esquerda e a subárvore direita de qualquer nó é, no máximo, 1. Essa característica fundamental garante que, não importa o quão você insira ou remova elementos, a busca em uma árvore AVL permanecerá eficiente, com um tempo médio de busca sempre logarítmico, independente de quão equilibrada a árvore esteja.

2- Descrição dos Codigos

```
3- public class Arvore2 {
4-     private Node lista;
5-     public Arvore2(){
6-         lista = null;
7-     }
8-     public void inOrdem(Node raiz){
9-         if(raiz!=null){
10-             inOrdem(raiz.getEsquerda());
11-             System.out.println(raiz.getInformacao());
12-             inOrdem(raiz.getDireita());
13-         }
14-     }
15-     public void preOrdem(Node raiz){
16-         if(raiz!=null){
17-             System.out.println(raiz.getInformacao());
18-             preOrdem(raiz.getEsquerda());
19-             preOrdem(raiz.getDireita());
20-         }
21-     }
22-     public void posOrdem(Node raiz){
23-         if(raiz!=null){
24-             posOrdem(raiz.getEsquerda());
25-             posOrdem(raiz.getDireita());
26-             System.out.println(raiz.getInformacao());
27-         }
28-     }
29-
30-     public Node inserir(Node atual,int numero,Node anterior) {
31-         if (atual == null) {
32-             Node no = new Node();
33-             no.setInformacao(numero);
34-             no.setAnterior(anterior);
35-             return no;
36-         }
37-         if(numero >=atual.getInformacao()){
38-             atual.setDireita(inserir(atual.getDireita(),numero,atual));
39-         }else{
40-             atual.setEsquerda(inserir(atual.getEsquerda(),numero,atual));
41-         }
42-         return atual;
43-     }
```

```

44- public Node BuscarInOrdem(Node raiz){
45-     if(raiz!=null){
46-         BuscarInOrdem(raiz.getEsquerda());
47-         BuscarInOrdem(raiz.getDireita());
48-         return raiz;
49-     }
50-     return raiz;
51- }
52- public void removerMaior(Node raiz,Node maior){
53-     Node maiorNode = BuscarInOrdem(raiz);
54-     if(raiz==null){
55-         Node anterior = maior.getAnterior();
56-         anterior.setDireita(maior.getDireita());
57-         posOrdem(raiz);
58-     }
59-     else{
60-         if(raiz.getInformacao()>maior.getInformacao()){
61-             maiorNode = raiz;
62-         }
63-         removerMaior(raiz.getDireita(),maiorNode);
64-     }
65- }
66- }
67- public void removerMenor(Node raiz,Node menor){
68-     Node menorNode = BuscarInOrdem(raiz);
69-     if(raiz==null){
70-         Node anterior = menor.getAnterior();
71-         anterior.setEsquerda(menor.getEsquerda());
72-         posOrdem(raiz);
73-     }
74-     else{
75-         if(raiz.getInformacao()>menor.getInformacao()){
76-             menorNode = raiz;
77-         }
78-         removerMenor(raiz.getEsquerda(),menorNode);
79-     }
80- }
81- public Node removerElemento(Node raiz, int elemento) {
82-     if (raiz == null) {
83-         return raiz;
84-     }
85-     if (elemento < raiz.getInformacao()) {
86-         raiz.setEsquerda(removerElemento(raiz.getEsquerda(),
87-             elemento));
88-     } else if (elemento > raiz.getInformacao()) {

```

```

88-         raiz.setDireita(removerElemento(raiz.getDireita(), elemento));
89-     } else {
90-         if (raiz.getEsquerda() == null) {
91-             return raiz.getDireita();
92-         } else if (raiz.getDireita() == null) {
93-             return raiz.getEsquerda();
94-         }
95-         Node menorSubArvoreDireita = BuscarInOrdem(raiz.getDireita());
96-         raiz.setInformacao(menorSubArvoreDireita.getInformacao());
97-         raiz.setDireita(removerElemento(raiz.getDireita(),
menorSubArvoreDireita.getInformacao()));
98-     }
99-     return raiz;
100- }
101-
102- }

```

Construtor Arvore2:

Este é o construtor da classe Arvore2.

Ele inicializa o nó raiz da árvore (lista) como nulo, indicando que a árvore está vazia no início.

Função inOrdem:

Esta função realiza uma travessia em ordem (in-order) na árvore.

Ela recebe um nó raiz como parâmetro e imprime os valores dos nós em ordem crescente.

Função preOrdem:

Esta função realiza uma travessia em pré-ordem (pre-order) na árvore.

Ela recebe um nó raiz como parâmetro e imprime os valores dos nós em pré-ordem.

Função posOrdem:

Esta função realiza uma travessia em pós-ordem (post-order) na árvore.

Ela recebe um nó raiz como parâmetro e imprime os valores dos nós em pós-ordem.

Função inserir:

Esta função insere um novo elemento (representado pelo valor numero) na árvore.

Ela recebe o nó atual (atual), o valor a ser inserido (numero), e o nó anterior (anterior) como parâmetros.

A função percorre a árvore de forma recursiva para encontrar a posição correta para inserir o novo elemento.

Função BuscarInOrdem:

Esta função parece ser uma tentativa de percorrer a árvore em ordem, mas não está retornando valores. Ela pode não estar funcionando conforme o esperado.

Função removerMaior:

Esta função remove o maior elemento da árvore.

Ela recebe dois nós como parâmetros: raiz e maior.

A função encontra o maior nó na árvore e o remove, ajustando as conexões necessárias.

Função removerMenor:

Esta função remove o menor elemento da árvore.

Ela recebe dois nós como parâmetros: raiz e menor.

A função encontra o menor nó na árvore e o remove, ajustando as conexões necessárias.

Função removerElemento:

Esta função remove um elemento específico da árvore.

Ela recebe dois parâmetros: raiz (o nó raiz da árvore) e elemento (o valor a ser removido).

A função percorre a árvore de forma recursiva para encontrar o elemento a ser removido e realiza a remoção, mantendo a propriedade de árvore binária de busca.

```
public class Arvore {
    public Node raiz;

    public Arvore() {
        raiz = null;
    }

    public void inOrdem(Node raiz) {
        if (raiz != null) {
            inOrdem(raiz.getEsquerda());
            System.out.println(raiz.getInformacao());
            inOrdem(raiz.getDireita());
        }
    }

    public void preOrdem(Node raiz) {
        if (raiz != null) {
            System.out.println(raiz.getInformacao());
            preOrdem(raiz.getEsquerda());
            preOrdem(raiz.getDireita());
        }
    }

    public void posOrdem(Node raiz) {
        if (raiz != null) {
            posOrdem(raiz.getEsquerda());
            posOrdem(raiz.getDireita());
            System.out.println(raiz.getInformacao());
        }
    }

    public int altura(Node node) {
        if (node == null) {
            return 0;
        }
        return Math.max(altura(node.getEsquerda()),
            altura(node.getDireita())) + 1;
    }

    private int balanceamento(Node node) {
        if (node == null) {
            return 0;
        }
        return altura(node.getEsquerda()) - altura(node.getDireita());
    }
}
```

```

private Node rotacaoEsquerda(Node raiz) {
    Node novaRaiz = raiz.getDireita();
    Node temp = novaRaiz.getEsquerda();
    novaRaiz.setEsquerda(raiz);
    raiz.setDireita(temp);
    return novaRaiz;
}

private Node rotacaoDireita(Node raiz) {
    Node novaRaiz = raiz.getEsquerda();
    Node temp = novaRaiz.getDireita();
    novaRaiz.setDireita(raiz);
    raiz.setEsquerda(temp);
    return novaRaiz;
}

public Node inserir(Node atual, int numero) {
    if (atual == null) {
        Node no = new Node();
        no.setInformacao(numero);
        return no;
    }
    if (numero >= atual.getInformacao()) {
        atual.setDireita(inserir(atual.getDireita(), numero));
    } else {
        atual.setEsquerda(inserir(atual.getEsquerda(), numero));
    }

    int balanceamento = balanceamento(atual);

    if (balanceamento > 1) {
        if (numero < atual.getEsquerda().getInformacao()) {
            // Rotação direita simples
            return rotacaoDireita(atual);
        } else {
            // Rotação esquerda-direita (dupla)
            atual.setEsquerda(rotacaoEsquerda(atual.getEsquerda()));
            return rotacaoDireita(atual);
        }
    }

    if (balanceamento < -1) {
        if (numero >= atual.getDireita().getInformacao()) {
            // Rotação esquerda simples

```

```

        return rotacaoEsquerda(atual);
    } else {
        // Rotação direita-esquerda (dupla)
        atual.setDireita(rotacaoDireita(atual.getDireita()));
        return rotacaoEsquerda(atual);
    }
}

return atual;
}

public Node buscar(Node raiz, int valor) {
    if (raiz == null || raiz.getInformacao() == valor) {
        return raiz;
    }

    if (valor < raiz.getInformacao()) {
        return buscar(raiz.getEsquerda(), valor);
    }

    return buscar(raiz.getDireita(), valor);
}

public Node remover(Node raiz, int valor) {
    if (raiz == null) {
        return raiz;
    }

    if (valor < raiz.getInformacao()) {
        raiz.setEsquerda(remover(raiz.getEsquerda(), valor));
    } else if (valor > raiz.getInformacao()) {
        raiz.setDireita(remover(raiz.getDireita(), valor));
    } else {
        if (raiz.getEsquerda() == null || raiz.getDireita() == null) {
            Node temp = null;
            if (temp == raiz.getEsquerda()) {
                temp = raiz.getDireita();
            } else {
                temp = raiz.getEsquerda();
            }

            if (temp == null) {
                temp = raiz;
                raiz = null;
            } else {

```



```

        raiz = temp;
    }
    } else {
        Node temp = minValueNode(raiz.getDireita());
        raiz.setInformacao(temp.getInformacao());
        raiz.setDireita(remover(raiz.getDireita(),
temp.getInformacao()));
    }
}

if (raiz == null) {
    return raiz;
}

int balanceamento = balanceamento(raiz);

if (balanceamento > 1) {
    if (balanceamento(raiz.getEsquerda()) >= 0) {
        return rotacaoDireita(raiz);
    } else {
        raiz.setEsquerda(rotacaoEsquerda(raiz.getEsquerda()));
        return rotacaoDireita(raiz);
    }
}

if (balanceamento < -1) {
    if (balanceamento(raiz.getDireita()) <= 0) {
        return rotacaoEsquerda(raiz);
    } else {
        raiz.setDireita(rotacaoDireita(raiz.getDireita()));
        return rotacaoEsquerda(raiz);
    }
}

return raiz;
}

private Node minValueNode(Node node) {
    Node current = node;
    while (current.getEsquerda() != null) {
        current = current.getEsquerda();
    }
    return current;
}
}

```

Construtor Arvore:

Este é o construtor da classe Arvore.

Ele inicializa o nó raiz da árvore (raiz) como nulo, indicando que a árvore está vazia no início.

Função inOrdem:

Realiza uma travessia em ordem (in-order) na árvore.

Recebe um nó raiz como parâmetro e imprime os valores dos nós em ordem crescente.

Função preOrdem:

Realiza uma travessia em pré-ordem (pre-order) na árvore.

Recebe um nó raiz como parâmetro e imprime os valores dos nós em pré-ordem.

Função posOrdem:

Realiza uma travessia em pós-ordem (post-order) na árvore.

Recebe um nó raiz como parâmetro e imprime os valores dos nós em pós-ordem.

Função altura:

Calcula a altura da árvore a partir de um nó específico.

Recebe um nó como parâmetro e retorna a altura da subárvore enraizada nesse nó.

Função balanceamento:

Calcula o fator de balanceamento de um nó, que é a diferença entre as alturas das subárvores esquerda e direita.

Recebe um nó como parâmetro e retorna o valor do fator de balanceamento.

Funções rotacaoEsquerda e rotacaoDireita:

Implementam rotações simples (à esquerda e à direita) e rotações duplas (esquerda-direita e direita-esquerda) para equilibrar a árvore.

São usadas durante a inserção para manter a propriedade de árvore AVL (Árvore de Busca Binária Balanceada).

Função inserir:

Insere um novo elemento (representado pelo valor número) na árvore. Recebe o nó atual (atual) e o valor a ser inserido (número) como parâmetros.

A função percorre a árvore de forma recursiva e realiza as rotações necessárias para manter o equilíbrio da árvore.

Função buscar:

Realiza uma busca na árvore para encontrar um valor específico. Recebe a raiz da árvore (raiz) e o valor a ser buscado como parâmetros. Retorna o nó que contém o valor ou null se o valor não for encontrado.

Função remover:

Remove um elemento específico da árvore. Recebe a raiz da árvore (raiz) e o valor a ser removido como parâmetros. Realiza a remoção de forma recursiva, mantendo o equilíbrio da árvore por meio de rotações quando necessário.

Função minValueNode:

Encontra o nó com o menor valor na árvore. Recebe um nó como parâmetro e retorna o nó com o menor valor na subárvore enraizada nesse nó.

3- Diferença entre os códigos

Árvore (Código Arvore):

Balanceamento: O código arvore não possui nenhum mecanismo de balanceamento automático para manter a árvore em uma estrutura AVL (Árvore de Busca Binária Balanceada). Isso significa que a árvore pode se tornar

desequilibrada, levando a piores casos de desempenho em operações de busca, inserção e remoção.

Operações de Balanceamento: O código `Arvore` não inclui funções específicas para realizar rotações e balancear a árvore durante operações de inserção e remoção. Como resultado, não garante o equilíbrio da árvore, e o tempo de execução pode variar significativamente, dependendo da distribuição dos valores inseridos.

Método altura: Esse código implementa um método `altura` para calcular a altura de um nó em uma árvore, mas ele não é usado para equilibrar a árvore automaticamente.

Árvore2 (Código Arvore2):

Balanceamento Automático: O código `Arvore2` inclui um mecanismo de balanceamento automático para manter a árvore em uma estrutura AVL. Isso garante que a diferença de altura entre as subárvores esquerda e direita de qualquer nó seja no máximo 1, resultando em um desempenho consistente em todas as operações.

Funções de Balanceamento: Este código implementa funções de rotação (`rotacaoEsquerda` e `rotacaoDireita`) para equilibrar a árvore durante operações de inserção e remoção. Ele também realiza rotações duplas quando necessário para manter o equilíbrio.

Verificação de Balanceamento: O código `Arvore2` verifica o fator de balanceamento de um nó e aplica rotações sempre que o fator de balanceamento excede o limite de 1 ou -1. Isso garante que a árvore permaneça balanceada após cada operação.

4- Análise Crítica do Código

Adicionando Elementos na Árvore BST:

Adicionar elementos na Árvore BST é, na maioria das vezes, um processo tranquilo e rápido, especialmente se ela já estiver bem balanceada. Contudo, se a árvore estiver desbalanceada, a inclusão de novos elementos pode se tornar um tanto quanto ineficiente e lenta.

Buscando Elementos:

Na Árvore BST, buscar elementos é bastante ágil quando ela está balanceada. No entanto, se a árvore estiver desorganizada e desbalanceada, encontrar um elemento específico pode levar um pouco mais de tempo.

Removendo Elementos:

Se você precisa retirar um elemento da árvore, o tempo gasto para isso será semelhante ao de inserção ou busca, variando conforme o equilíbrio da árvore.

Cuidados com o Código:

O código da Árvore BST é relativamente simples, mas não realiza o balanceamento da árvore de modo automático. Isso significa que pode não ser o mais eficiente em todos os cenários e pode necessitar de melhorias futuras, sobretudo aquelas voltadas para o balanceamento.

Como Melhorar:

Uma sugestão de melhoria é implementar funções que realizem o balanceamento da árvore de forma automática, garantindo bom desempenho em diferentes situações.

Já na Árvore AVL:

Adicionando Elementos:

Na Árvore AVL, a adição de elementos é geralmente eficiente, pois ela possui mecanismos automáticos que mantêm o balanceamento durante a inserção, garantindo bom desempenho.

Buscando Elementos:

A busca de elementos na Árvore AVL costuma ser rápida e eficiente, graças ao balanceamento automático que ela realiza.

Removendo Elementos:

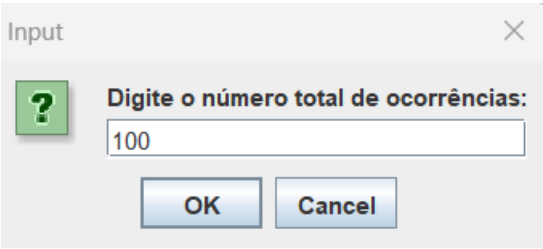
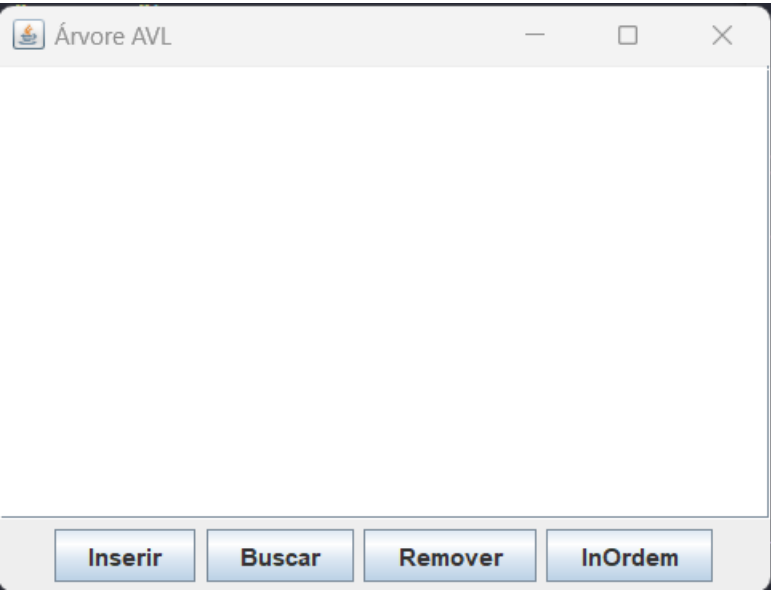
Na Árvore AVL, remover elementos também é um processo bastante ágil, e a árvore continua balanceada mesmo depois da remoção de um ou mais elementos.

Cuidados e Manutenção:

O código da Árvore AVL é robusto e confiável, oferecendo desempenho consistente em diferentes situações devido ao seu balanceamento automático. Além disso, ele é fácil de ser mantido e aprimorado, facilitando trabalhos futuros de manutenção e melhoria.


5- Resultado

Código AVL




Árvore AVL:
6957
L: 3421
L: 2028
L: 849
L: 261
L: 156
L: 64
R: 576
L: 426
R: 652
R: 1234
L: 945
L: 935
R: 1065
R: 1571
L: 1343
R: 1627
L: 1613
R: 2023
R: 2723
L: 2276
L: 2111
R: 2263
R: 2719
L: 2334
R: 3072
L: 3031
L: 2829
L: 2745
R: 2947
R: 3035
R: 3140
L: 3093
R: 3287
L: 3175
R: 5370
L: 3802
L: 3502
L: 3496
L: 3481
R: 3498
R: 3665
L: 3602
R: 4814
L: 4336
L: 4078

Input

 Digite o número a ser buscado:


OK Cancel

Message

 Número não encontrado na árvore.

OK

Input

 Digite o número a ser removido:

OK Cancel

Código Arvore BST

```
Digite 1 para inserir elementos na árvore
2 - Execução PosOrdem
3 - Execução InOrdem
4 - Execução PreOrdem
5 - Remover o menor
6 - Remover o maior
7 - Remover um número de sua escolha
8 - Buscar um número na árvore
0 - Sair
```

```
1
Digite a quantidade de números aleatórios que deseja inserir na árvore:
100
100 números aleatórios foram inseridos na árvore.
```


Se eu clicar 2/3/4 o programa printa de acordo com a regra.

Se eu clicar 7, o programa pedirá ao usuário para selecionar um número e removerá da árvore.

Se eu clicar 8, o programa pedirá ao usuário para selecionar um número e buscará na árvore, se achar ("Número encontrado na árvore") caso contrário ("Número não encontrado na árvore")

Se eu clicar 0, o programa será finalizado.

6- Conclusão

Após uma análise cuidadosa das estruturas de árvores em questão, podemos concluir que a Árvore AVL demonstra desempenho notavelmente superior e mais estável quando comparada à Árvore BST. Tal superioridade advém dos mecanismos intrínsecos de balanceamento da AVL, que asseguram rapidez e eficácia nas operações de inserção, busca e remoção de elementos, independentemente da configuração inicial da árvore.

Em contrapartida, a Árvore BST, quando desbalanceada, pode experimentar sérias dificuldades de desempenho, o que compromete a velocidade e eficiência em operações habitualmente rápidas, como é o caso da inserção e busca de elementos. Esse desbalanceamento demanda intervenções manuais para reestabelecer o equilíbrio, acrescentando tempo e complexidade tanto no desenvolvimento quanto na manutenção do código.

Desse modo, para projetos cuja prioridade recai sobre desempenho e eficiência consistentes nas operações relacionadas a estruturas de árvores, a Árvore AVL emerge como a opção mais acertada. Seu mecanismo auto-balanceável garante agilidade operacional, consolidando-a como uma alternativa robusta e de confiança para a manipulação eficiente de dados.