Relatório Estrutura de Dados – Prova

Objetivo

Este relatório tem o objetivo de fazer uma análise crítica sobre a aplicação de Arvore binaria não balanceada e Arvore binaria AVL, juntamente com as seguintes perguntas: **Qual funciona melhor? Em qual situação? Por quê**

Ambiente Utilizado

- Processador: Ryzen 5 5500
- Placa Gráfica: Rx 5500xt
- Memória: 16 Gb
- SSD: 500 Gb
- Fonte: 500W
- IDE: Intellij
- Software para Medição: Profiler (nativo do Intellij)

Qual funciona melhor e em qual situação?

Depende, não que tange a Inserção de “poucos” elementos, uma Arvore binaria não balanceada tende a ser mais eficiente e menos custosa, visto que a mesma não necessita seguir um algoritmo de balanceamento. Entretanto, para Buscas, remoções e inserções em grande quantidade Arvores AVL são uma escolha melhor, uma vez que, seu obrigatório balanceamento faz com que sua altura sempre seja a mais baixa possível, facilitando na hora de uma busca.

Concluindo, a escolha de qual arvore deve ser utilizada depende do número, de quais operações serão feitas e em qual contexto será utilizado. Se o objetivo é fazer inserções com uma quantidade mínima de memória e CPU disponível, uma arvore não balanceada pode ser a melhor opção. Por outro lado, se é necessário realizar muitas operações em grandes conjuntos de dados e pode arcar com a sobrecarga, a arvore AVL é a melhor escolha.

Métricas e Testes

Para as medições foi utilizado o Profiler (já nativo do Intellij), as capturas de tela abaixo foram tiradas no início da execução do programa e no final, para a arvore AVL também, a fim de termos um comparativo do uso de memória, CPU e também, a TImeline de eventos, a qual registra blue e Red events.

No início do programa, é criado uma lista de valores aleatórios, essa lista é usada para os testes em ambas as arvore. Durante os testes, são executados os mesmos comandos:

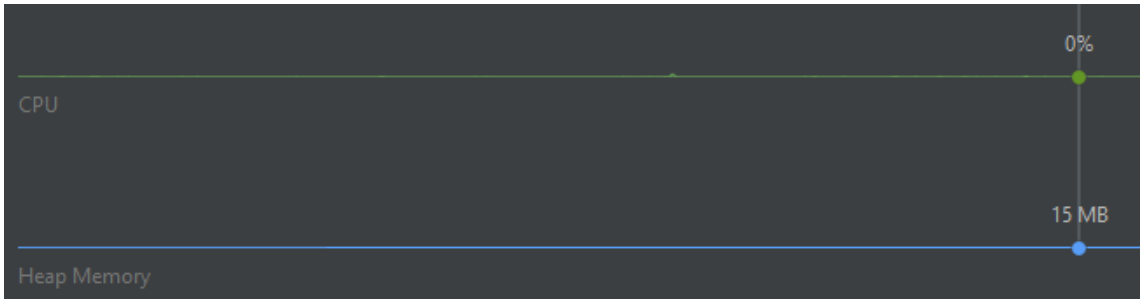
1. Adicionar: 100
2. Remover: 99
3. Buscar: 98

Esses valores foram escolhidos para garantir que os testes fossem os mais fiéis possíveis e abrangessem uma boa quantidade de cenários.

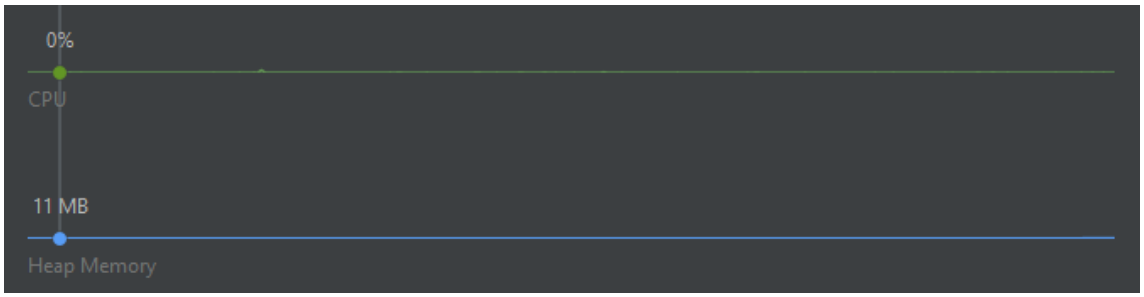
Segue abaixo capturas de tela referentes aos cenários propostos no enunciado.

Início, arvore não balanceada com 100 valores aleatórios:

Fim do programa:

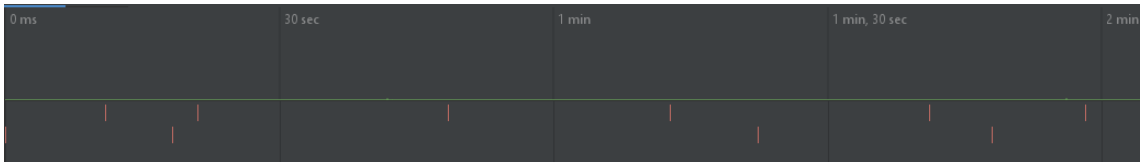


Início, arvore AVL com 100 valores aleatórios:

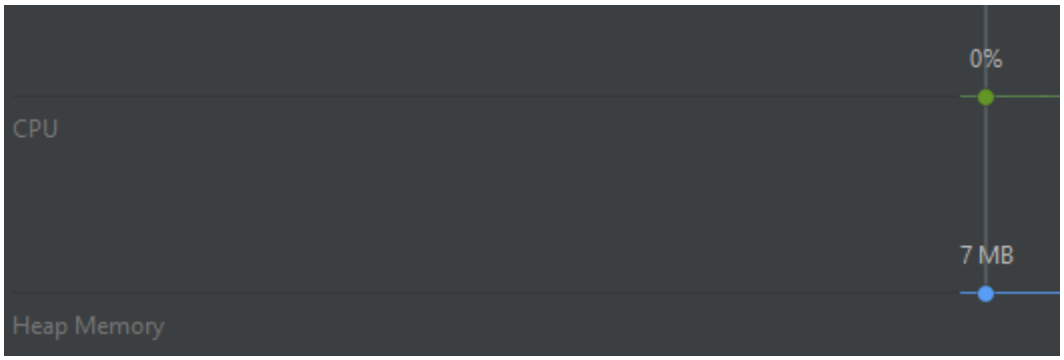


Fim do programa:

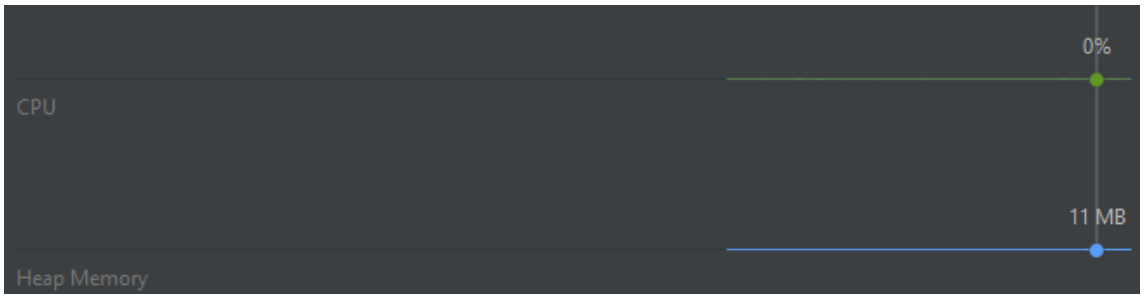
TimeLine:



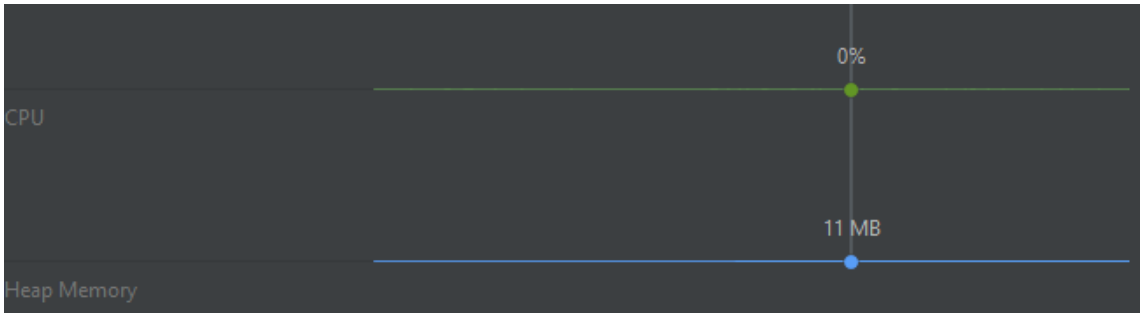
Início, arvore não balanceada com 500 valores aleatórios:



Fim do programa:

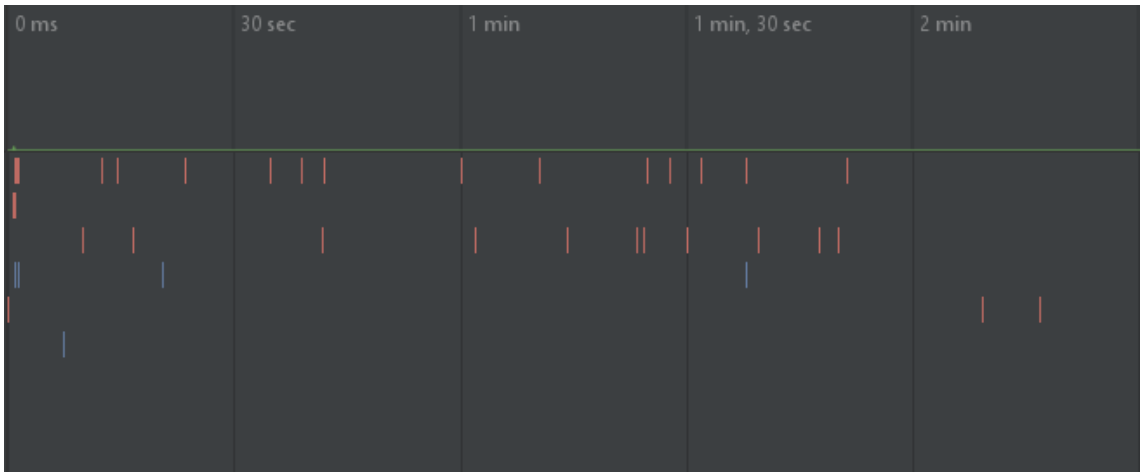


Início, arvore AVL com 500 valores aleatórios:

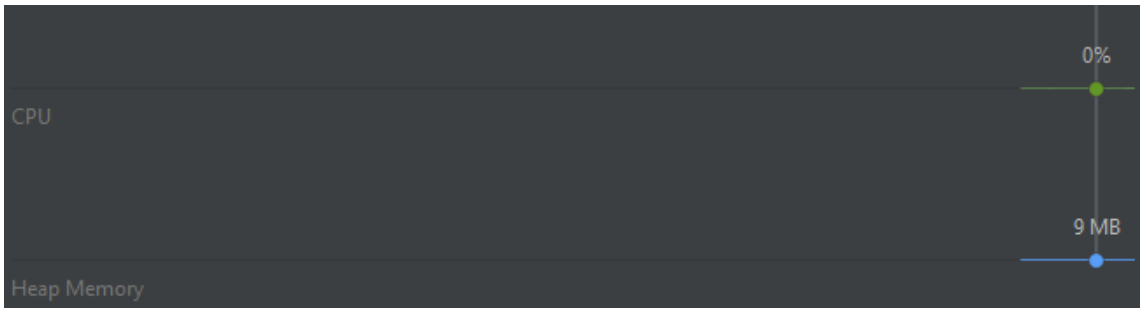


Fim do programa:

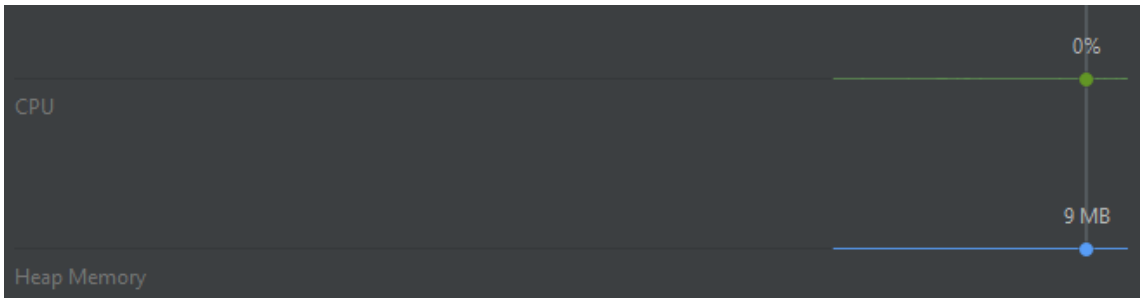
TimeLine:



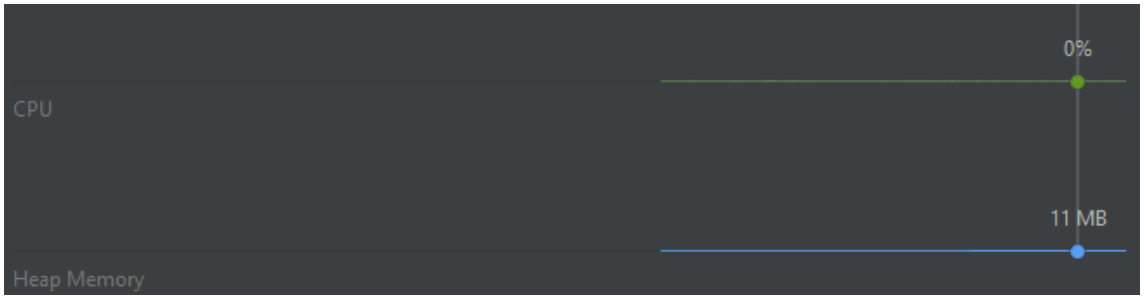
Início, arvore não balanceada com 1000 valores aleatórios:



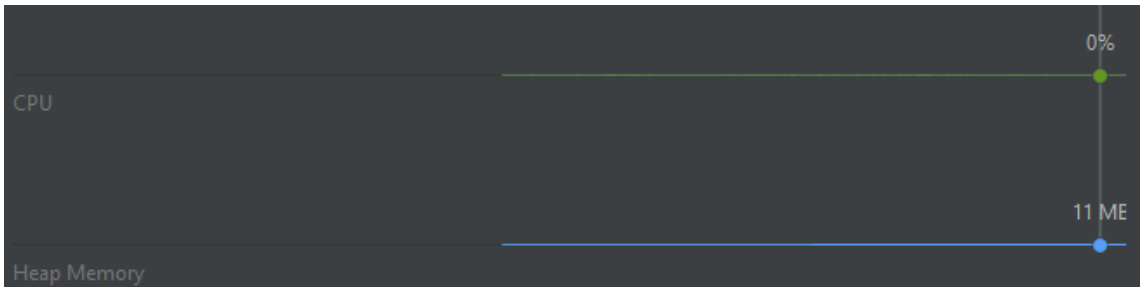
Fim do programa:



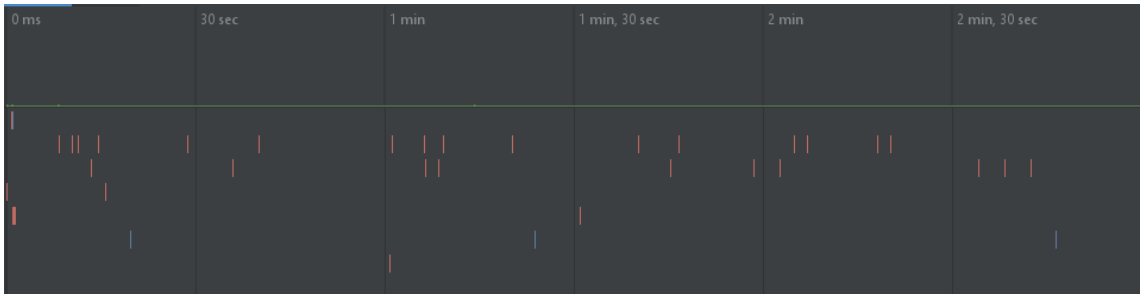
Início, arvore AVL com 1000 valores aleatórios:



Fim do programa:

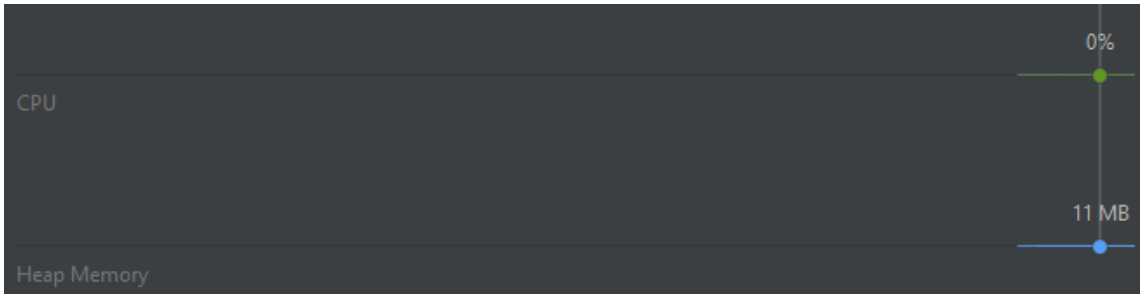


TimeLine:

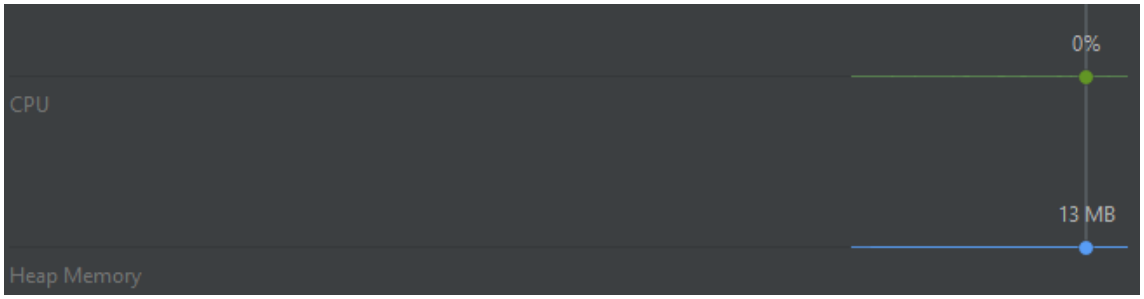


Pico de 1% de CPU

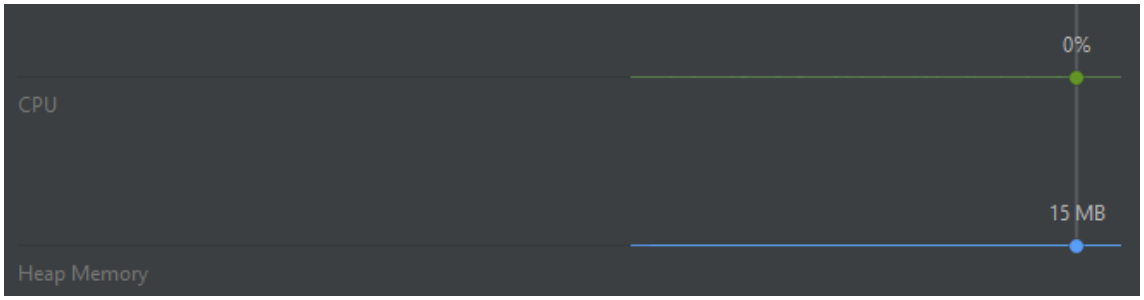
Início, arvore não balanceada com 10000 valores aleatórios:



Fim do programa:

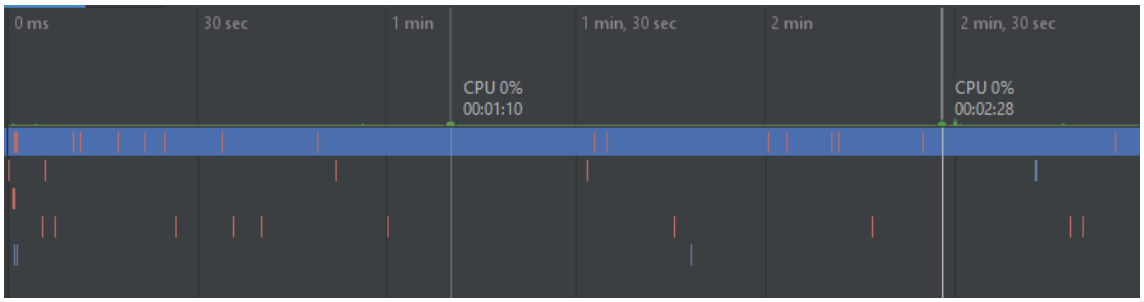


Início, arvore AVL com 10000 valores aleatórios:



Fim do programa:

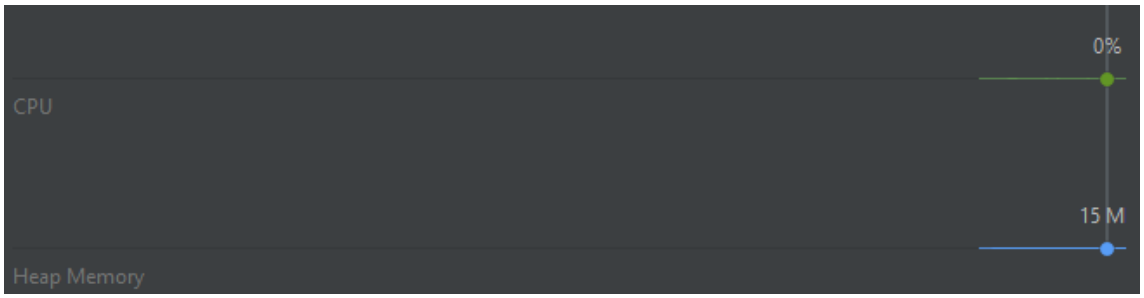
TimeLine:



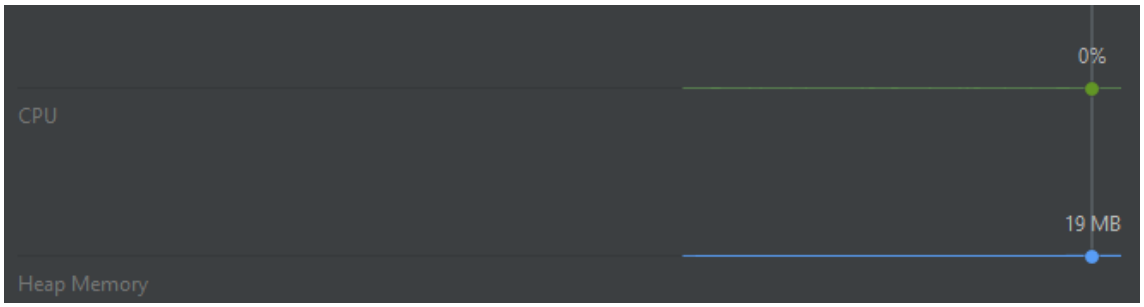
(Houve um pico de 6% de CPU)

Alocações de memória vem se tornando mais comum conforme vai aumentando a quantidade de valores

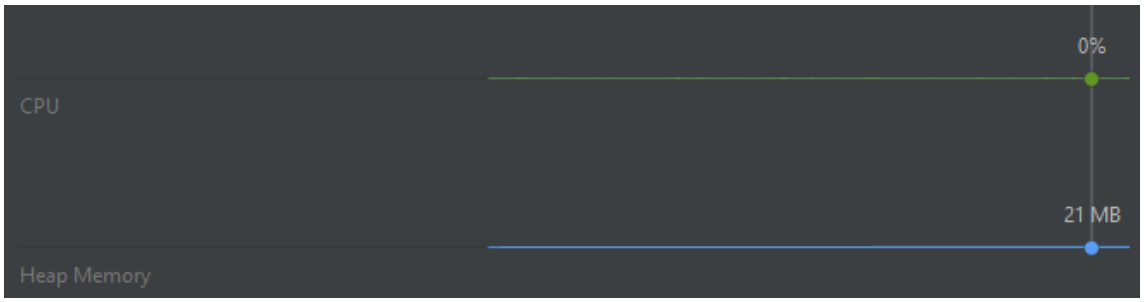
Início, arvore não balanceada com 20000 valores aleatórios:



Fim do programa:

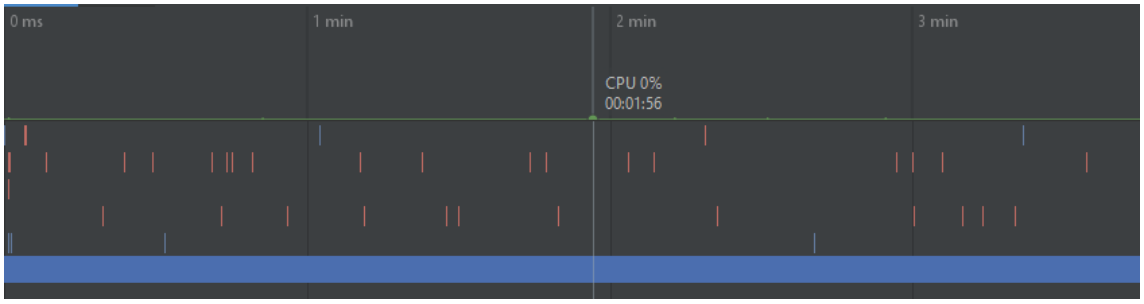


Início, arvore AVL com 20000 valores aleatórios:



Fim do programa:

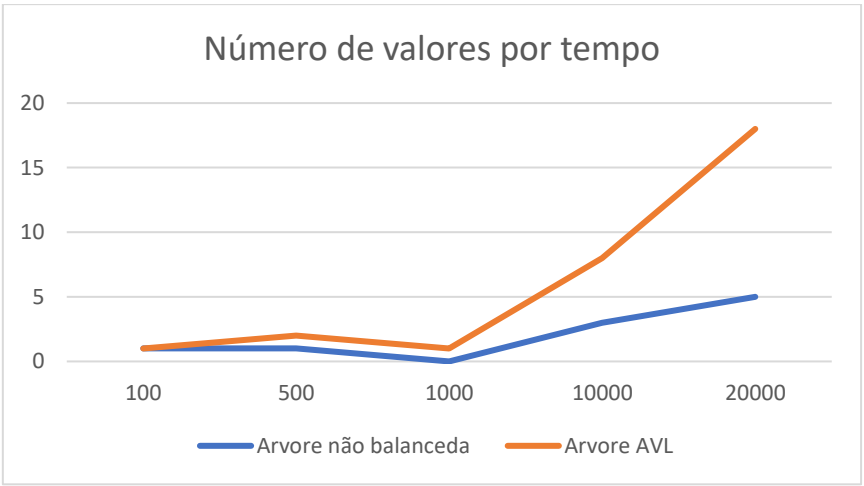
TimeLine:



Desempenho:

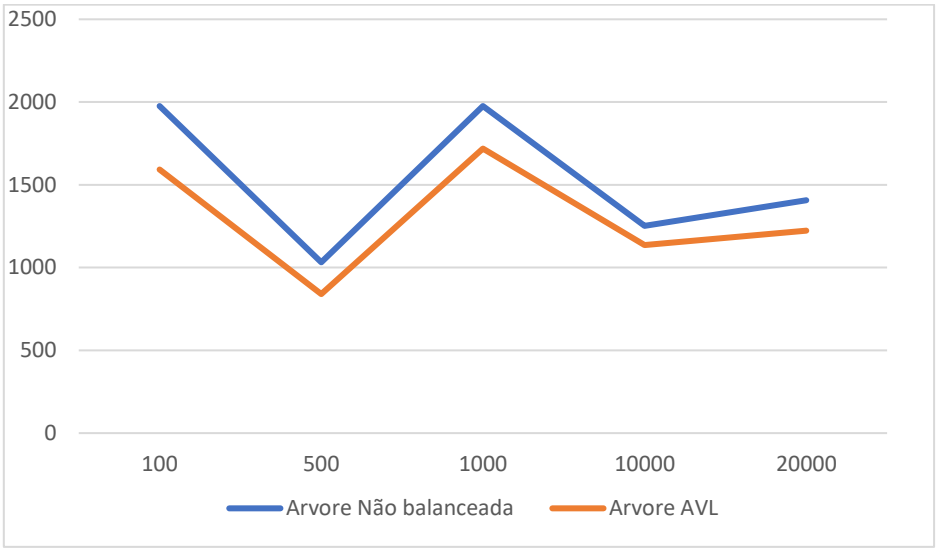
Para corroborar o que foi apresentado anteriormente, foi medido o tempo (em milissegundos) das duas arvores para o comparativo. Importante lembrar que para os testes foram utilizados os mesmos valores para todos os casos

Gráfico para a adição de valores na arvore:



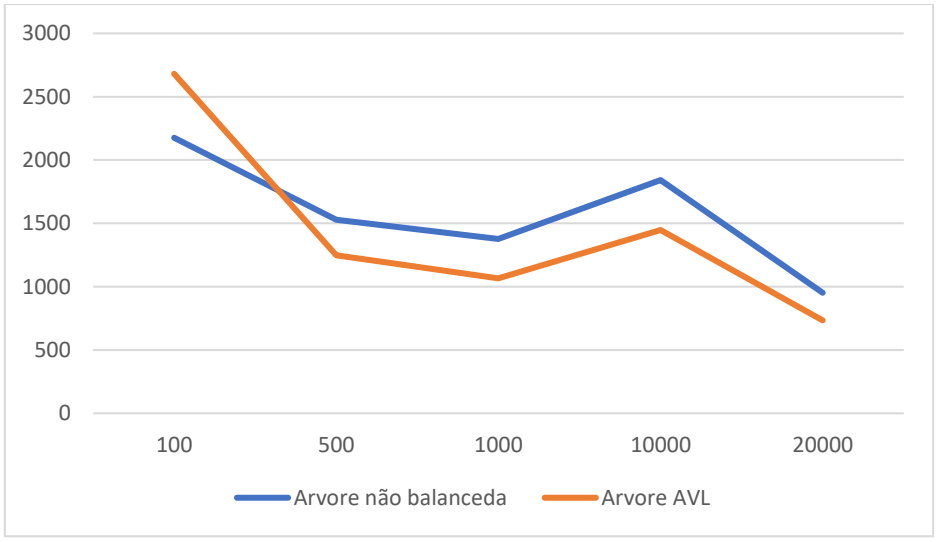
Como é possível notar, a partir de 1000 valores, acabou demandando muito mais tempo, assim como uso de memória e CPU.

Para remover valores:



Apesar de ser contraintuitivo, a remoção de valores foi mais eficiente no algoritmo AVL.

Para busca:



Apesar de ter começado com um tempo maior, o algoritmo AVL se mostrou mais eficiente na maioria dos casos para busca, provavelmente por conta de sua altura menor.

Considerações Finais:

A fim de garantir um resultado mais solido, foram feitos vários testes, ignorando os poucos arbitrários e destoantes. Entretanto, esses casos demonstraram que há momentos que os valores de estudo influenciam no resultado final (dentro do escopo deste estudo), mesmo o algoritmo sendo consistente e eficiente na sua função. Além disso, foi medido a alocação de memória, gasto de CPU e analisado a Timeline de eventos, esses valores foram importantes para confirmar a tese de que o algoritmo de balanceamento tem seu devido custo computacional, confirmado pelo crescente uso de memória, CPU e surgimento de “Red

events”. Importante lembrar que, apesar de seu custo, algoritmos de balanceamento tendem a ser a melhor escolha.

Explicação do Código fonte

Classe main

Bibliotecas utilizadas para a criação de lista de valores aleatórios e leitura de console, apenas para as funções permitidas no enunciado.

```
import java.util.ArrayList;
import java.util.List;
import java.util.Random;
import java.util.Scanner;
```

Loop FOR para criação da lista de valores aleatórios, a mesma lista é utilizada para popular as duas arvores.

```
int numeroElementos = 10000;

for (int i = 0; i < numeroElementos; i++) {
    // Gere um número aleatório e adicione à lista
    int numeroAleatorio = random.nextInt(numeroElementos);
    listaNumerosAleatorios.add(numeroAleatorio);
}
```

A estrutura de menu foi criada para facilitar os testes dos algoritmos

```
while (true) {
    System.out.println("Menu Arvore binaria genérica");
    System.out.println("0 - Iniciar");
    System.out.println("1 - Adicionar item");
    System.out.println("2 - remover item");
    System.out.println("3 - imprimir arvore em preOrdem");
    System.out.println("4 - busca");
    System.out.println("5 - Sair");
    int opc1 = scanner.nextInt();
```

Dentro da estrutura de menu, foi utilizado if/else para a chamada dos métodos desejados, ex:

```java
if (opc1 == 0) {
    long startTime = System.currentTimeMillis();
    for (int numero : listaNumerosAleatorios) {
        ar.Adicionar(numero);
    }
    System.out.println("Valores adicionados");
    long endTime = System.currentTimeMillis();
    long elapsedTime = endTime - startTime;
    System.out.println("Tempo de execução: " + elapsedTime + " milissegundos");

} else if (opc1 == 1) {
    System.out.println("Qual valor deseja adicionar: ");
    long startTime = System.currentTimeMillis();
    int valor = scanner.nextInt();
    ar.Adicionar(valor);
    System.out.println("Valor adicionado");
    long endTime = System.currentTimeMillis();
    long elapsedTime = endTime - startTime;
    System.out.println("Tempo de execução: " + elapsedTime + " milissegundos");
```

O mesmo se estende até o final das opções e se repete para o algoritmo de balanceamento.

Classe Arvore

Essa classe é feita para os métodos da arvore binaria não balanceada.

```java
Node raiz;

1 usage
public arvore() {
    raiz = null;
}
```

O método de adição funciona da seguinte forma:

```java
public void Adicionar(int valor) {
    Node novo = new Node(valor);
    novo.info = valor;
    novo.direita = null;
    novo.esquerda = null;

    if (raiz == null) {
        raiz = novo;
          System.out.println("Raiz adicionada de valor: " + novo.getInfo());
    } else {
        Node atual = raiz;
        Node anterior;
        while (true) {
            anterior = atual;
            if (valor <= atual.info) {
                atual = atual.esquerda;
                if (atual == null) {
                      System.out.println("O pai desta folha eh: " + anterior.getInfo());
                    anterior.esquerda = novo;
                      System.out.println("O valor adicionado na esquerda foi: " + anterior.esquerda.getInfo());
                    return;
                }
            } else {
                atual = atual.direita;
                if (atual == null) {
                      System.out.println("O pai desta folha eh: " + anterior.getInfo());
                    anterior.direita = novo;
```

1. Node novo = new Node(valor);: Cria um novo nó com o valor especificado e o armazena na variável novo.
2. novo.info = valor;: Define o valor do novo nó como o valor passado como argumento.
3. novo.direita = null; e novo.esquerda = null;: Inicializa os ponteiros para os nós à direita e à esquerda do novo nó como nulos, indicando que ele não tem filhos neste momento.
4. if (raiz == null) {: Verifica se a árvore está vazia, ou seja, se não há raiz.
   Se a árvore estiver vazia, raiz = novo; define a raiz da árvore como o novo nó criado.
5. Caso contrário (se a árvore não estiver vazia), o código entra no bloco else.
6. Node atual = raiz; e Node anterior;: Cria duas variáveis, atual e anterior, para ajudar na inserção do novo nó. atual é inicializado como a raiz da árvore, e anterior será usada para rastrear o nó anterior à posição onde o novo nó será inserido.
7. Entra em um loop while (true) que continuará até que o novo nó seja inserido na árvore.
8. anterior = atual;: Atualiza a variável anterior para apontar para o nó atual antes de mover para o próximo nível na árvore.
9. if (valor <= atual.info) {: Compara o valor a ser inserido com o valor no nó atual.
   Se o valor for menor ou igual ao valor no nó atual, mova-se para a esquerda na árvore.
10. Caso contrário (se o valor for maior que o valor no nó atual), mova-se para a direita na árvore.
    - atual = atual.direita; Move-se para o nó à direita.
    - if (atual == null) {: Verifica se chegou a uma posição onde não há nó à direita (ou seja, uma folha).
11. O loop continua até que o novo nó seja inserido na árvore.

Método de impressão em preOrdem

```java
public void preOrdem(Node atual) {
    if (atual != null) {
        System.out.println(atual.info + " ");
        preOrdem(atual.esquerda);
        preOrdem(atual.direita);
    }
}
```

Método de remoção

```java
public Node remove(Node raiz, int valor) {
    if (raiz == null) {
        return raiz;
    }

    if (valor < raiz.info) {
        raiz.esquerda = remove(raiz.esquerda, valor);
    } else if (valor > raiz.info) {
        raiz.direita = remove(raiz.direita, valor);
    } else {
        if (raiz.esquerda == null) {
            return raiz.direita;
        } else if (raiz.direita == null) {
            return raiz.esquerda;
        }

        raiz.info = valorMinimo(raiz.direita);
        raiz.direita = remove(raiz.direita, raiz.info);
    }

    return raiz;
}
```

```java
private int valorMinimo(Node raiz) {
    int valorMinimo = raiz.info;
    while (raiz.esquerda != null) {
        valorMinimo = raiz.esquerda.info;
        raiz = raiz.esquerda;
    }
    return valorMinimo;
}
```

1. **public Node remove(Node raiz, int valor) {**: Esta função recebe a raiz da árvore e o valor a ser removido como parâmetros.

2. **if (raiz == null) {**: Verifica se a raiz da árvore é nula, o que significa que a árvore está vazia ou que o valor não foi encontrado na árvore.
    a. Se a raiz for nula, retorna a própria raiz (sem alterações).
3. **if (valor < raiz.info) {**: Verifica se o valor a ser removido é menor do que o valor no nó raiz atual.
    a. Se for menor, a função **remove** é chamada recursivamente na subárvore esquerda.
    b. O resultado da remoção é atribuído à subárvore esquerda da raiz atual.
4. **else if (valor > raiz.info) {**: Verifica se o valor a ser removido é maior do que o valor no nó raiz atual.
    a. Se for maior, a função **remove** é chamada recursivamente na subárvore direita.
    b. O resultado da remoção é atribuído à subárvore direita da raiz atual.
5. **else {**: Se o valor for igual ao valor no nó raiz atual, isso significa que encontramos o nó a ser removido.
    a. Se o nó não tiver um filho esquerdo, o seu filho direito (ou nulo) será retornado, substituindo o nó que está sendo removido.
    b. Se o nó não tiver um filho direito, o seu filho esquerdo (ou nulo) será retornado.
    c. Se o nó tiver ambos os filhos, a função **valorMinimo** é chamada na subárvore direita para encontrar o valor mínimo nessa subárvore.
    d. O valor mínimo é atribuído ao nó raiz atual, substituindo o valor a ser removido.
    e. A função **remove** é chamada recursivamente para remover o nó com o valor mínimo da subárvore direita.
6. **private int valorMinimo(Node raiz) {**: Esta função auxiliar é usada para encontrar o valor mínimo na árvore com a raiz especificada.
7. **int valorMinimo = raiz.info;**: Inicializa **valorMinimo** com o valor no nó raiz.
8. Entra em um loop enquanto houver nós à esquerda na subárvore.
    a. Atualiza **valorMinimo** para o valor no nó à esquerda.
    b. Move-se para o nó à esquerda na árvore.
9. Retorna o **valorMinimo** após encontrar o valor mínimo na subárvore.
10. Finalmente, a função **remove** retorna a raiz da árvore atualizada após a remoção do valor especificado.

Método de busca:

```java
public boolean Busca(Node no, int info) {
    if (no == null) {
        System.out.println("O valor não foi encontrado");
        return false;
    }

    if (info == no.info) {
        System.out.println("O valor foi achado " + info);
        return true;
    }

    if (info < no.info) {
        return Busca(no.esquerda, info);
    } else {
        return Busca(no.direita, info);
    }
}
```

1. **public boolean Busca(Node no, int info) {**: Esta função recebe um nó da árvore e o valor a ser procurado como parâmetros.
2. **if (no == null) {**: Verifica se o nó atual é nulo, o que indica que a árvore está vazia ou que o valor não foi encontrado na subárvore.
    a. Se o nó for nulo, imprime uma mensagem indicando que o valor não foi encontrado e retorna **false**.
3. **if (info == no.info) {**: Verifica se o valor a ser encontrado é igual ao valor no nó atual.
    a. Se for igual, imprime uma mensagem indicando que o valor foi encontrado e retorna **true**.
4. Se o valor não foi encontrado no nó atual, o código faz uma escolha com base na comparação do valor de busca **info** com o valor no nó atual **no.info**.
    a. Se **info** for menor que **no.info**, a busca continua na subárvore esquerda, chamando a função **Busca(no.esquerda, info)**.
    b. Se **info** for maior ou igual a **no.info**, a busca continua na subárvore direita, chamando a função **Busca(no.direita, info)**.
5. O processo de busca continua recursivamente até que o valor seja encontrado ou até que um nó nulo seja alcançado, indicando que o valor não está na árvore.
6. Quando o valor é encontrado, a função retorna **true**. Caso contrário, quando um nó nulo é alcançado, a função retorna **false**.

Arvore AVL

Método para rotação para direita:

```
Node rotacaoDireita(Node noAtual) {
    if (noAtual == null || noAtual.esquerda == null) {
        return noAtual;
    }

    Node noEsquerda = noAtual.esquerda;
    Node subArvoreDireita = noEsquerda.direita;
    noEsquerda.direita = noAtual;
    noAtual.esquerda = subArvoreDireita;

    // Calcula as alturas
    int alturaNoAtualEsquerda = altura(noAtual.esquerda);
    int alturaNoAtualDireita = altura(noAtual.direita);
    int alturaNoEsquerdaEsquerda = altura(noEsquerda.esquerda);
    int alturaNoEsquerdaDireita = altura(noEsquerda.direita);

    // Atualiza alturas
    noAtual.altura = 1 + alturaMaior(alturaNoAtualEsquerda, alturaNoAtualDireita);
    noEsquerda.altura = 1 + alturaMaior(alturaNoEsquerdaEsquerda, alturaNoEsquerdaDireita);


    return noEsquerda;
}
```

1. **int altura(Node No) {**:
    - Esta função calcula a altura de um nó na árvore AVL.
    - **if (No == null)** verifica se o nó é nulo, indicando uma subárvore vazia.
    - Se o nó for nulo, a altura é 0 (base do caso).
    - Caso contrário, a função retorna a altura armazenada no próprio nó (o campo **No.altura**), que deve ser mantido atualizado durante as operações de inserção e remoção para garantir que a árvore esteja balanceada.
2. **int maior(int a, int b) {**:
    - Esta função retorna o maior valor entre dois números inteiros **a** e **b**.
    - Ela é usada para calcular a altura máxima entre dois subnós da árvore.
3. **Node rotacaoDireita(Node noAtual) {**:
    - Esta função realiza uma rotação simples à direita em torno de um nó na árvore AVL para balancear a árvore.
    - **if (noAtual == null || noAtual.esquerda == null)** verifica se o nó atual ou seu filho esquerdo é nulo, o que indica que uma rotação à direita não é possível.
    - Se a rotação à direita for possível, a função realiza os seguintes passos: a. Cria uma referência **noEsquerda** para o filho esquerdo do nó atual. b. Armazena a subárvore direita do nó à esquerda em **subArvoreDireita**. c. Atribui o nó atual como filho direito do nó à esquerda (**noEsquerda.direita = noAtual**). d. Atualiza o filho esquerdo do nó atual para ser a subárvore direita (**noAtual.esquerda = subArvoreDireita**).

- Em seguida, a função calcula e atualiza as alturas dos nós afetados. As alturas são calculadas usando a função **altura** e a função **maior** para determinar a altura correta para cada nó.
- Finalmente, a função retorna o nó à esquerda (que agora é o novo nó raiz da subárvore rotacionada).

A rotação para a esquerda funciona da mesma forma, apenas mudando os valores:

```
Node rotacaoEsquerda(Node noAtual) {
    if (noAtual == null || noAtual.direita == null) {
        return noAtual;
    }

    Node noDireita = noAtual.direita;
    Node subArvoreEsquerda = noDireita.esquerda;
    noDireita.esquerda = noAtual;
    noAtual.direita = subArvoreEsquerda;

    // Calcula as alturas
    int alturaNoAtualEsquerda = altura(noAtual.esquerda);
    int alturaNoAtualDireita = altura(noAtual.direita);
    int alturaNoDireitaEsquerda = altura(noDireita.esquerda);
    int alturaNoDireitaDireita = altura(noDireita.direita);

    // Atualiza alturas
    noAtual.altura = 1 + alturaMaior(alturaNoAtualEsquerda, alturaNoAtualDireita);
    noDireita.altura = 1 + alturaMaior(alturaNoDireitaEsquerda, alturaNoDireitaDireita);

    return noDireita;
}
```

```
int alturaMaior(int altura1, int altura2) {
    if (altura1 > altura2) {
        return altura1;
    } else {
        return altura2;
    }
}
4 usages
int fatorDeBalanceamento(Node N) {
    if (N == null)
        return 0;
    return altura(N.esquerda) - altura(N.direita);
}
```

1. **int alturaMaior(int altura1, int altura2) {**:
    - Esta função recebe duas alturas (**altura1** e **altura2**) e retorna a maior delas.
    - É usada para determinar a altura máxima entre dois subnós da árvore AVL.
2. **int fatorDeBalanceamento(Node N) {**:
    - Esta função calcula o fator de balanceamento de um nó na árvore AVL.
    - O fator de balanceamento é a diferença entre a altura da subárvore esquerda e a altura da subárvore direita do nó.

- **if (N == null)** verifica se o nó é nulo, indicando uma subárvore vazia.
- Se o nó for nulo, o fator de balanceamento é definido como 0 (uma subárvore vazia tem um fator de balanceamento de 0).
- Caso contrário, a função calcula o fator de balanceamento subtraindo a altura da subárvore direita da altura da subárvore esquerda (**altura(N.esquerda) - altura(N.direita)**).
- O fator de balanceamento é uma medida importante para determinar se um nó ou uma subárvore está desequilibrado na árvore AVL. Se o fator de balanceamento de um nó for maior que 1 ou menor que -1, a árvore estará desequilibrada, e operações de reequilíbrio, como rotações, podem ser aplicadas para corrigir o equilíbrio da árvore.

```
Node inserir(Node node, int item) {
    if (node == null)
        return (new Node(item));
    if (item < node.info)
        node.esquerda = inserir(node.esquerda, item);
    else if (item > node.info)
        node.direita = inserir(node.direita, item);
    else
        node.direita = inserir(node.direita, item);
    node.altura = 1 + maior(altura(node.esquerda), altura(node.direita));
    int balanceFactor = fatorDeBalanceamento(node);
    if (balanceFactor > 1) {
        if (item < node.esquerda.info) {
            return rotacaoDireita(node);
        } else if (item > node.esquerda.info) {
            node.esquerda = rotacaoEsquerda(node.esquerda);
            return rotacaoDireita(node);
        }
    }
```

```
if (balanceFactor < -1) {
    if (item > node.direita.info) {
        return rotacaoEsquerda(node);
    } else if (item < node.direita.info) {
        node.direita = rotacaoDireita(node.direita);
        return rotacaoEsquerda(node);
    }
}
return node;
```

1. **Node inserir(Node node, int item) {**:
    - Esta função recebe um nó da árvore (**node**) e um valor a ser inserido (**item**).

2. **if (node == null)**: Verifica se o nó atual é nulo. Se for nulo, cria um novo nó com o valor **item** e o retorna como a raiz da subárvore inserida.

3. **if (item < node.info)**: Verifica se o valor **item** é menor do que o valor no nó atual **node.info**.

    a. Se for menor, a função **inserir** é chamada recursivamente na subárvore esquerda do nó **node**.
    b. O resultado da inserção é atribuído ao nó esquerdo do **node**.

7. **else if (item > node.info)**: Verifica se o valor **item** é maior do que o valor no nó atual **node.info**.

    a. Se for maior, a função **inserir** é chamada recursivamente na subárvore direita do nó **node**.
    b. O resultado da inserção é atribuído ao nó direito do **node**.

8. **else**: Se o valor **item** for igual ao valor no nó atual, a função é chamada recursivamente na subárvore direita. Isso permite que a árvore contenha valores duplicados.

9. Após a inserção, a altura do nó atual é recalculada com base nas alturas das subárvores esquerda e direita, e a altura é armazenada no campo **node.altura**.

10. Em seguida, é calculado o fator de balanceamento do nó chamando a função **fatorDeBalanceamento(node)**.

11. O código verifica o fator de balanceamento para determinar se a árvore está desequilibrada. Se o fator de balanceamento for maior que 1, isso indica que a subárvore esquerda é mais alta, e podem ser necessárias rotações para reequilibrar a árvore.

    a. Se o **item** for menor que o valor na subárvore esquerda, é realizada uma rotação simples à direita em **node**.
    b. Se o **item** for maior que o valor na subárvore esquerda, é realizada uma rotação à esquerda em **node.esquerda**, seguida por uma rotação simples à direita em **node**.

12. Se o fator de balanceamento for menor que -1, isso indica que a subárvore direita é mais alta, e podem ser necessárias rotações para reequilibrar a árvore.

    a. Se o **item** for maior que o valor na subárvore direita, é realizada uma rotação simples à esquerda em **node**.
    b. Se o **item** for menor que o valor na subárvore direita, é realizada uma rotação à direita em **node.direita**, seguida por uma rotação simples à esquerda em **node**.

13. Por fim, a função retorna o nó atual, que pode ter sido alterado durante o processo de inserção e reequilíbrio. Esse nó atual será o novo nó raiz da subárvore onde a inserção ocorreu ou o mesmo nó se nenhuma rotação foi necessária.