Relatório avaliação RA2

Arthur Przygocki,
Carlos Eduardo Rodrigues Mello,
Henrique Tetilha Golias.

Github: https://github.com/Henrick1/Av-RA02

Compare as árvores analisando seu desempenho com a inserção de 100,500,1000 e 10.000 e 20.000 elementos gerados aleatoriamente.

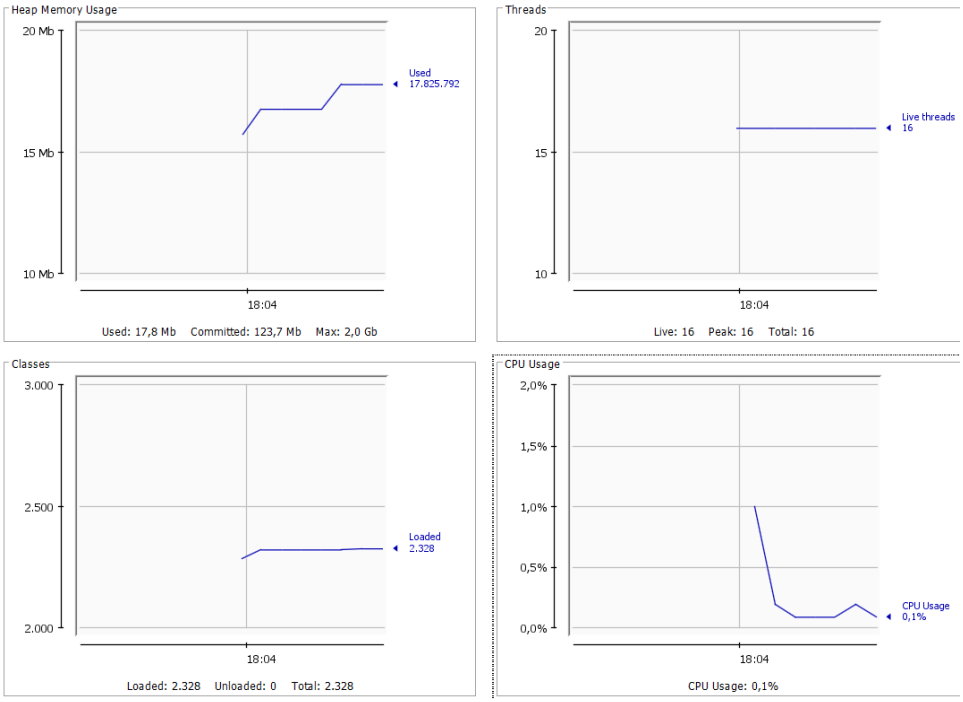
Os testes foram feitos em uma máquina com:
Processador: AMD Ryzen 7 4800H with Radeon Graphics 2.90 GHz

RAM: 8,00 GB (utilizável: 7,37 GB)

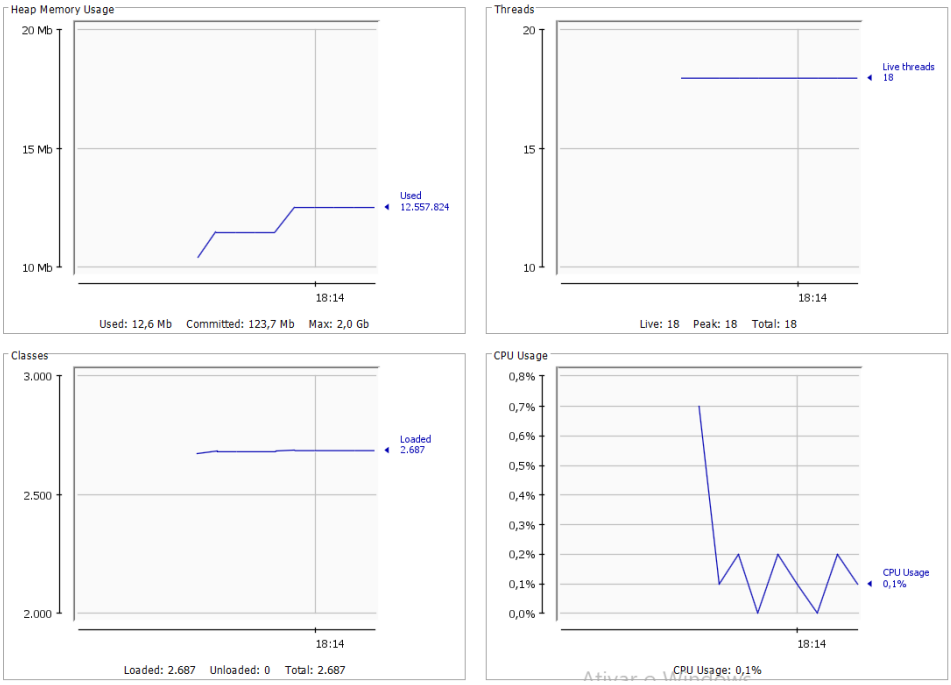
SO: Windows 10 64 bit

Árvore binária com 100 elementos:

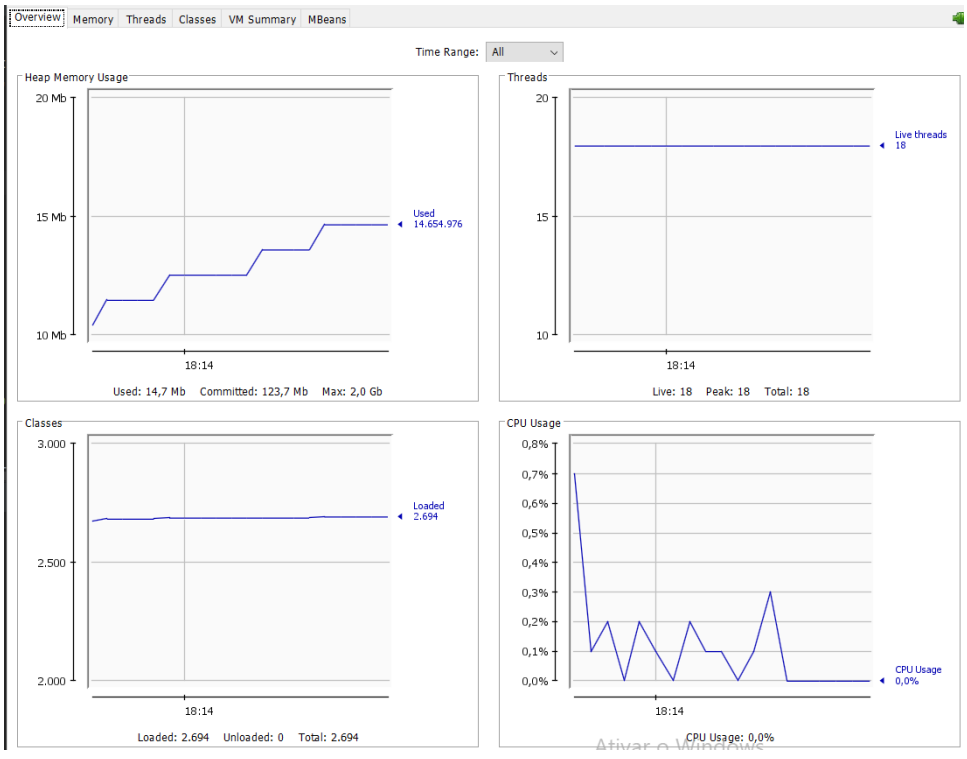
Inserção dos elementos aleatórios: 0.001 segundos



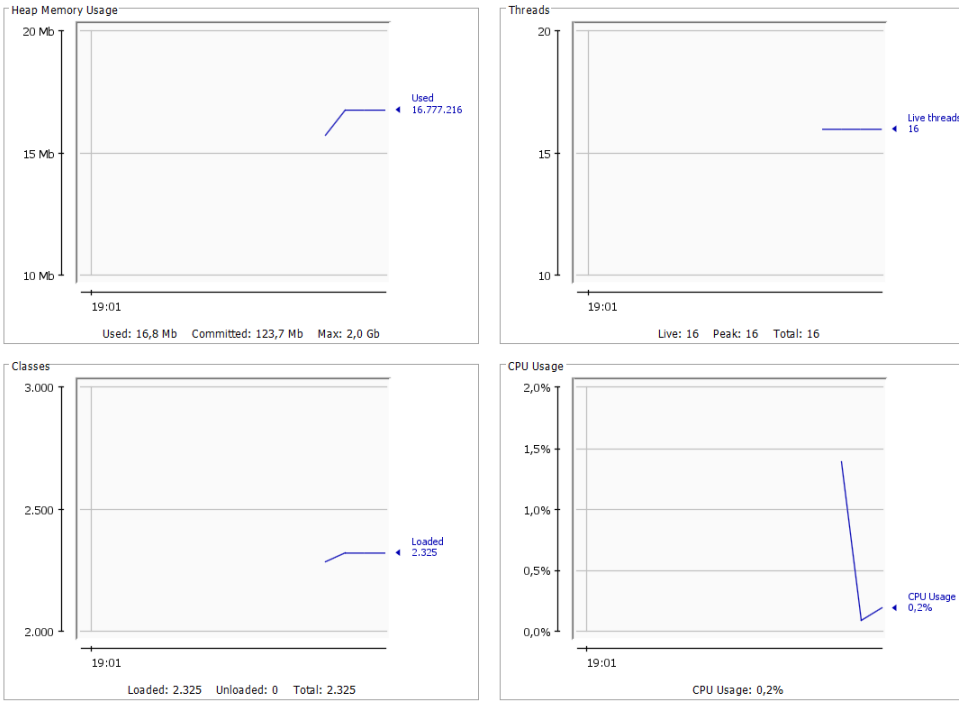
Impressão Pré-Ordem:



Remoção:

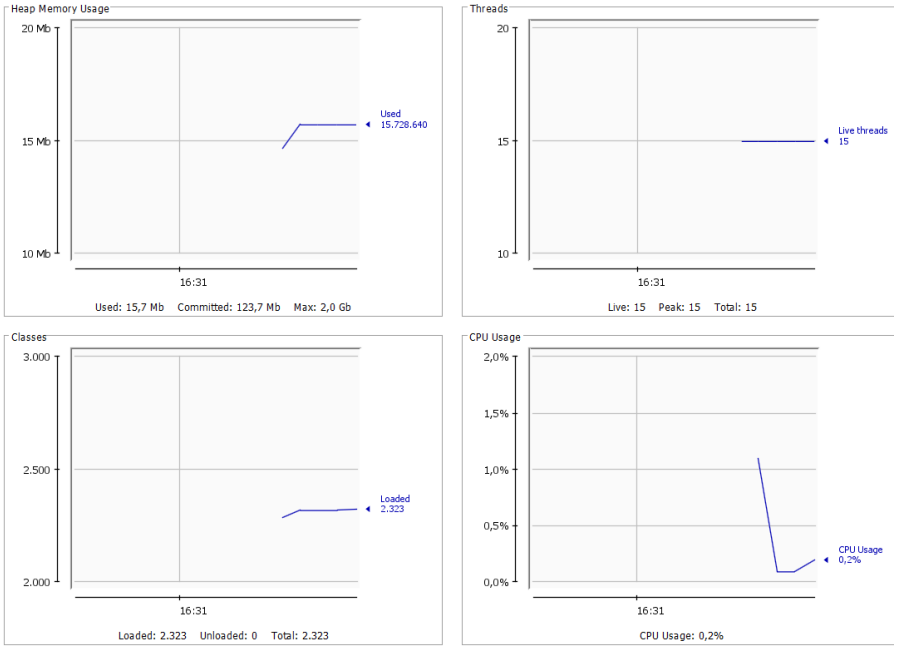


Busca:

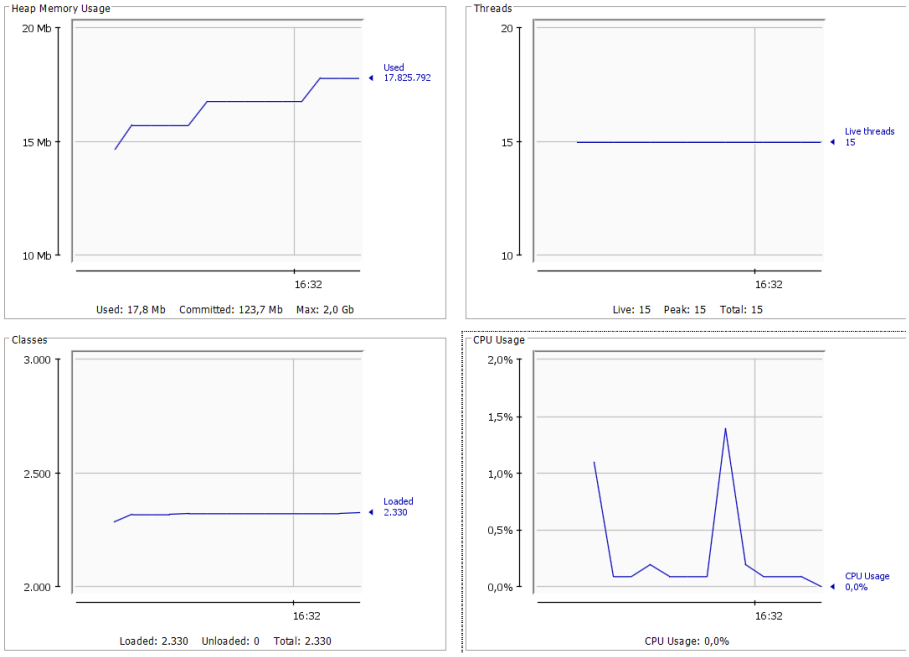


Árvore AVL com 100 elementos:

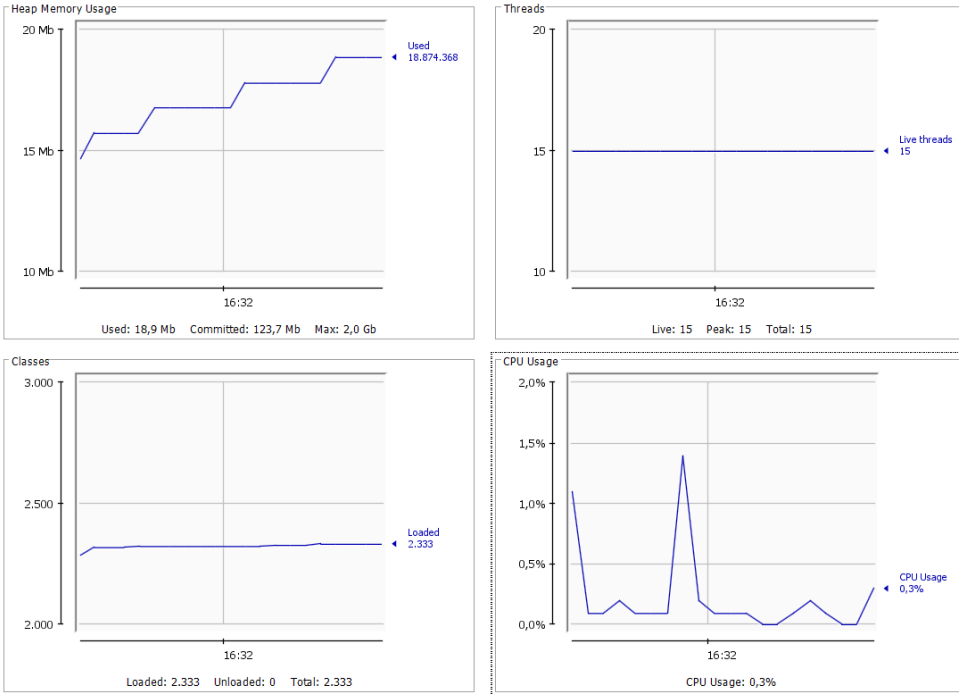
Inserção: 2ms



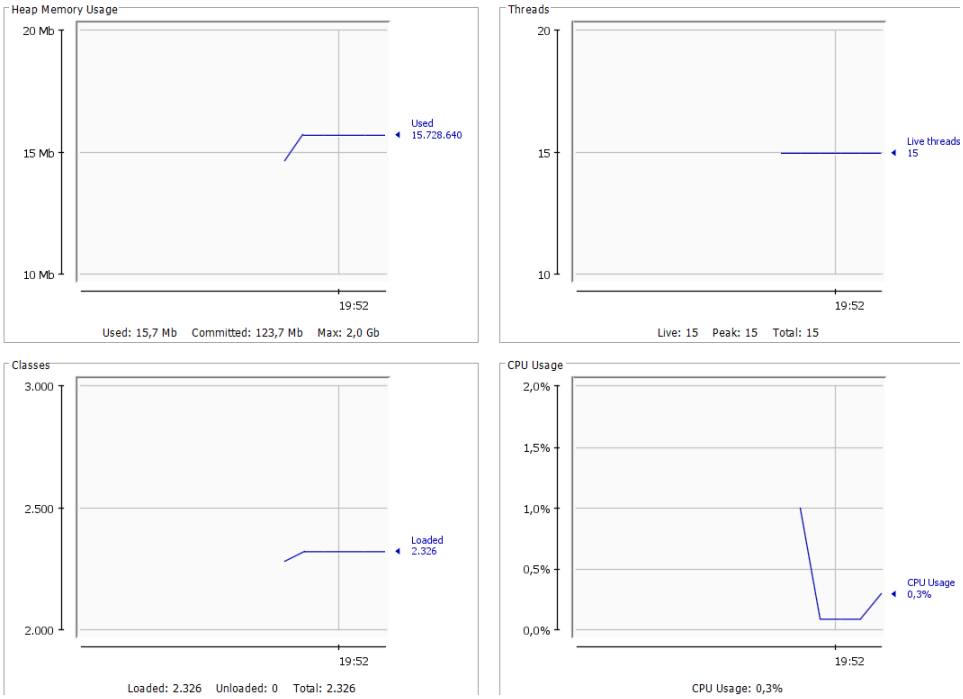
Impresão Pré-Ordem:



Remoção:

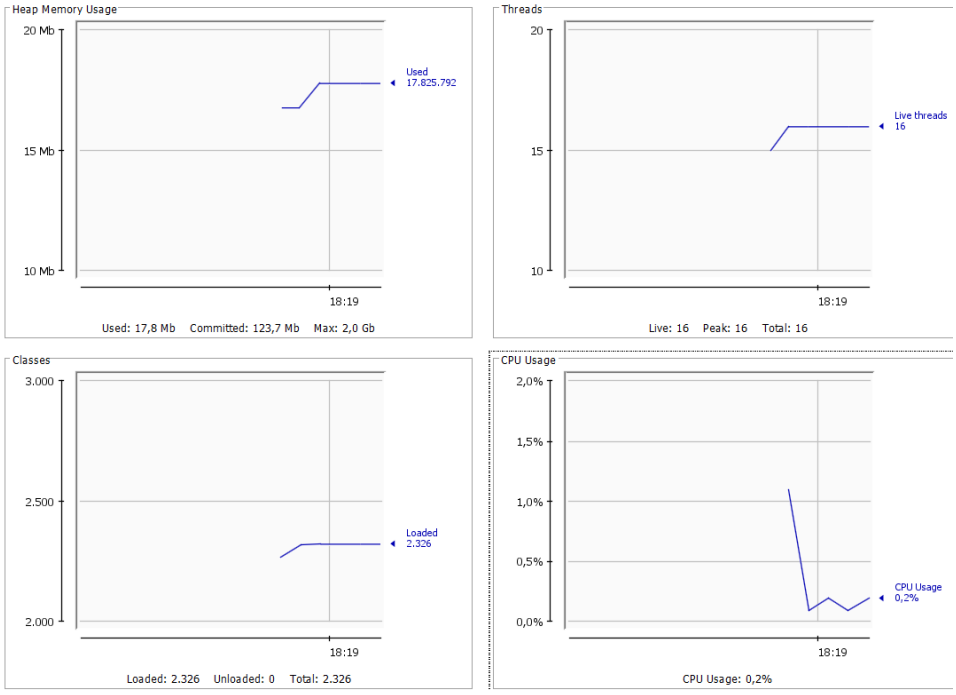


Busca:

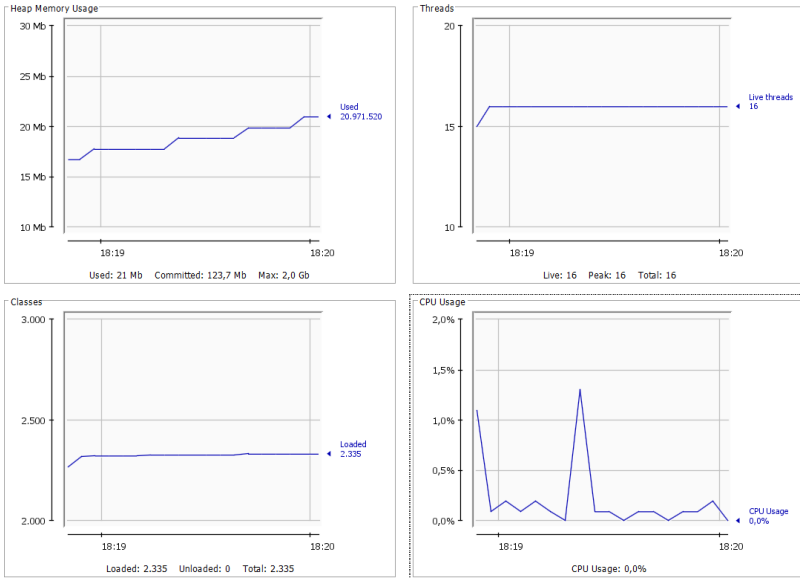


Árvore binária com 500 elementos:

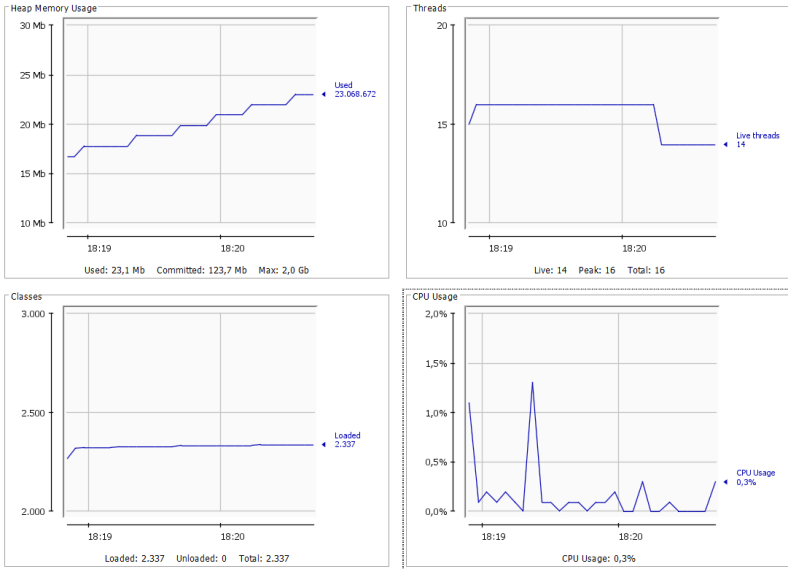
Inserção: 0.002 segundos



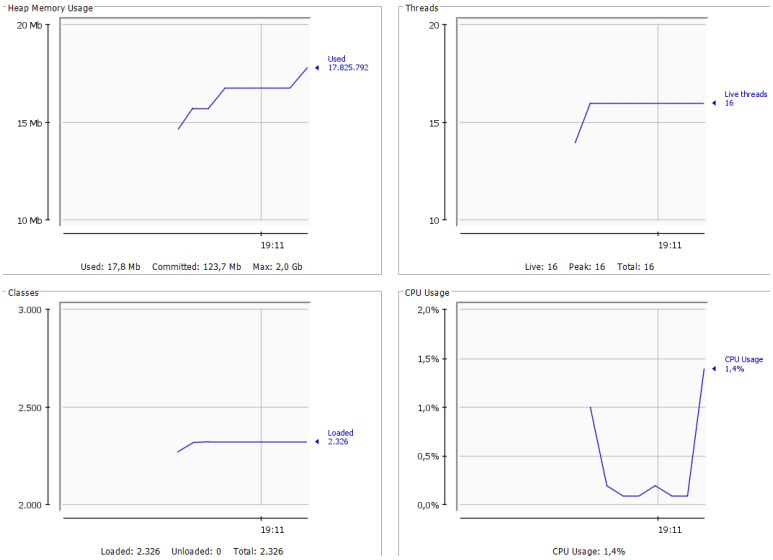
Impressão Pré-Ordem:



Remoção: 1 milissegundo

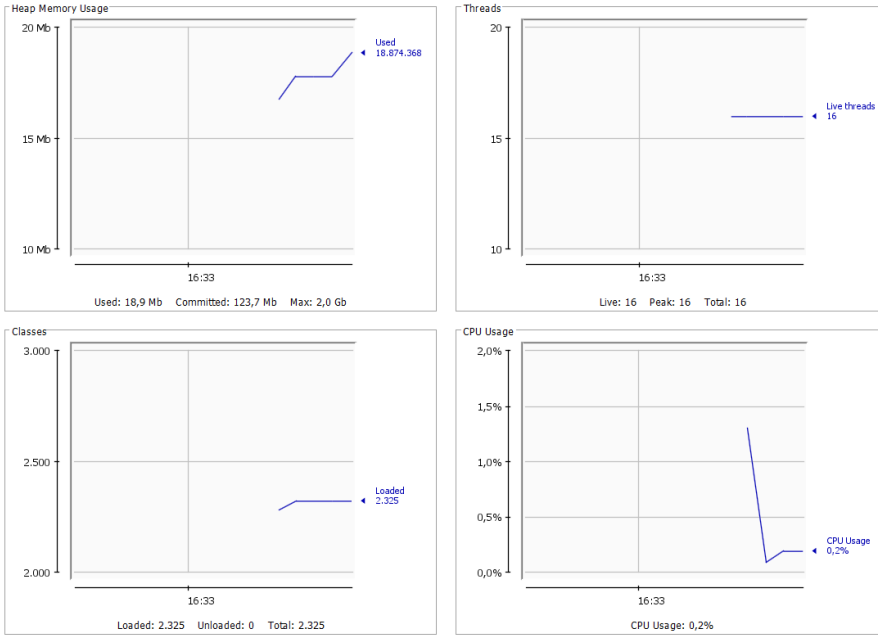


Busca:

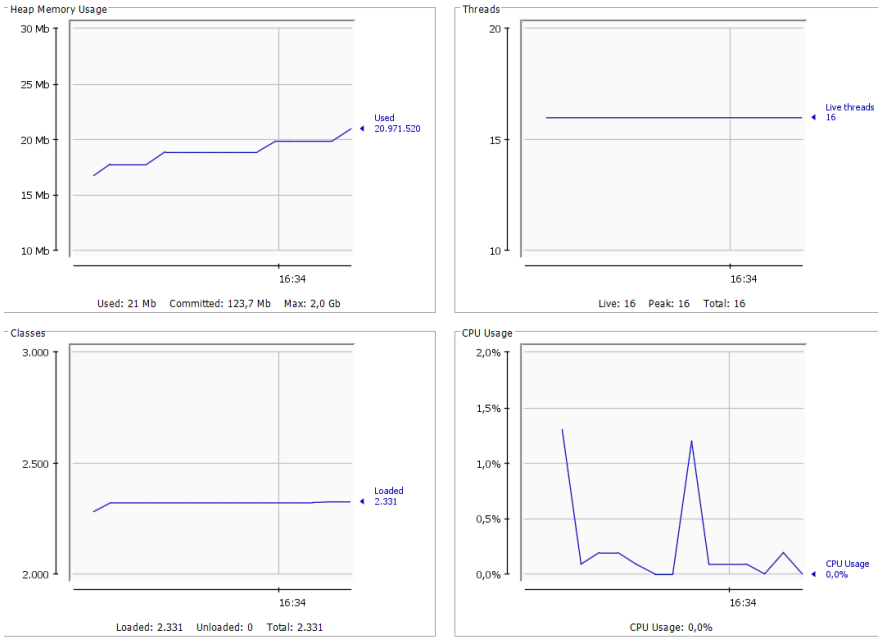


Árvore AVL com 500 elementos:

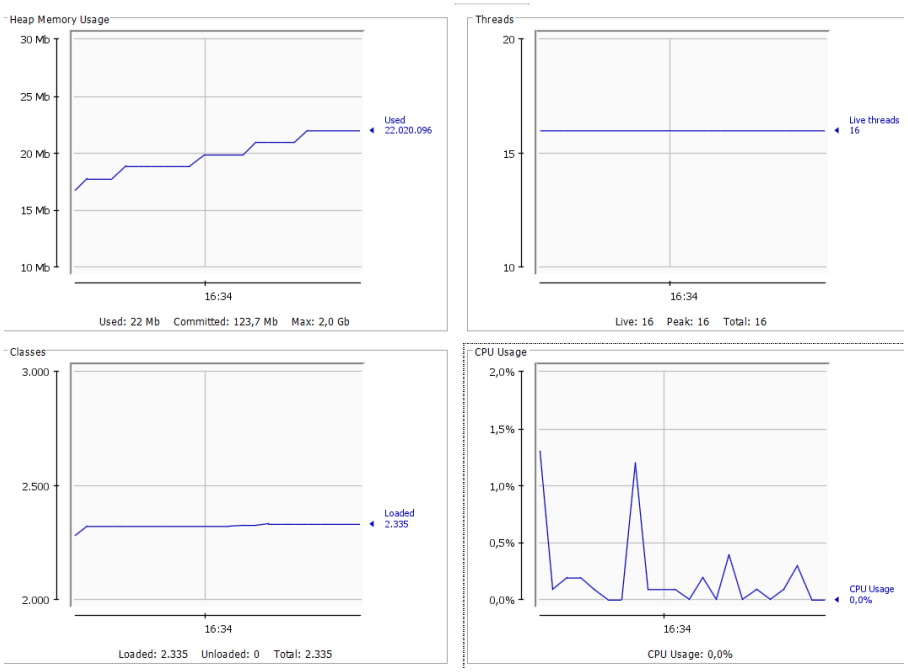
Inserção: 8 ms



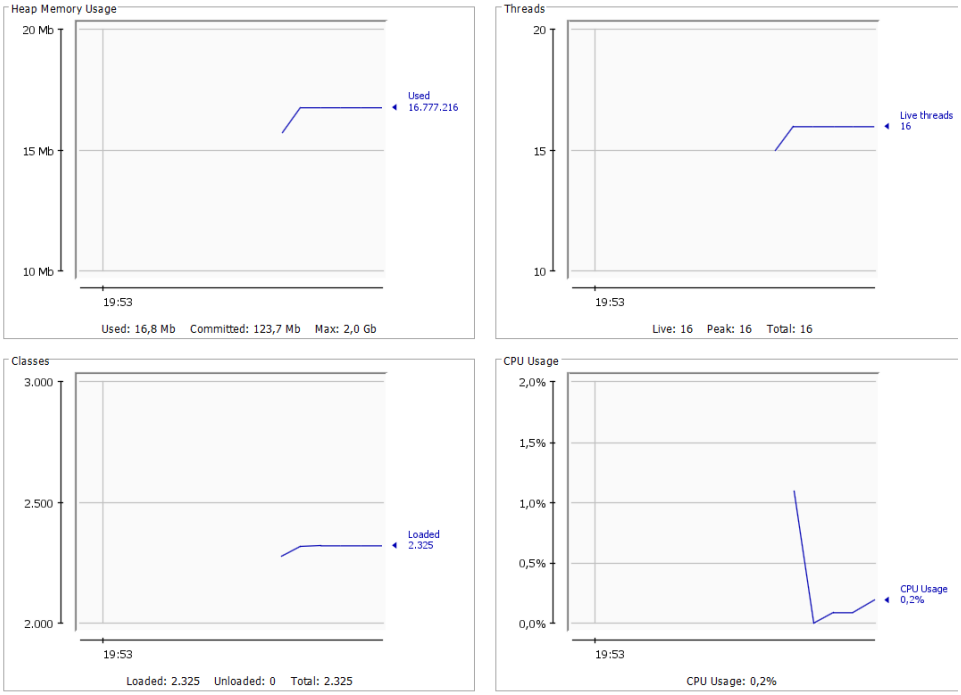
Impresão Pré-Ordem:



Remoção:

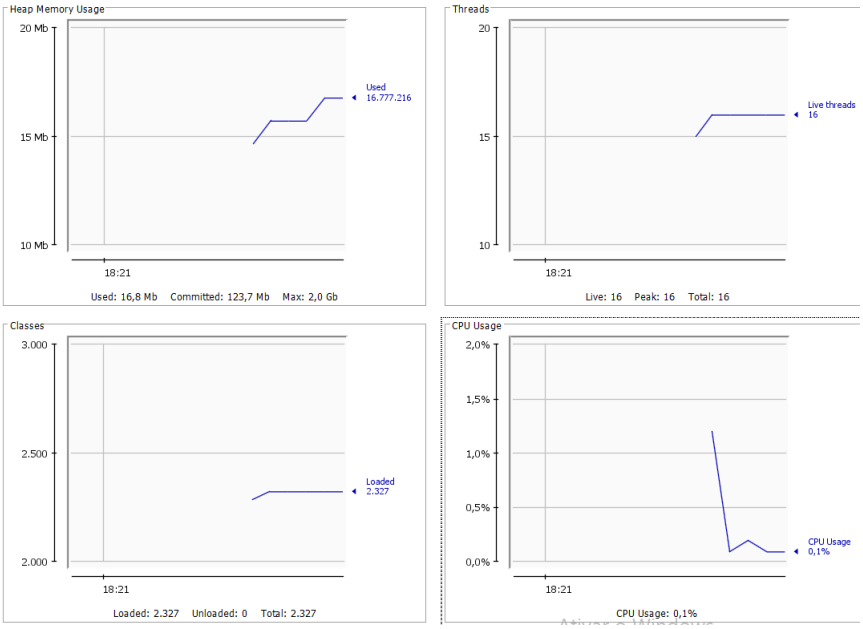


Busca:

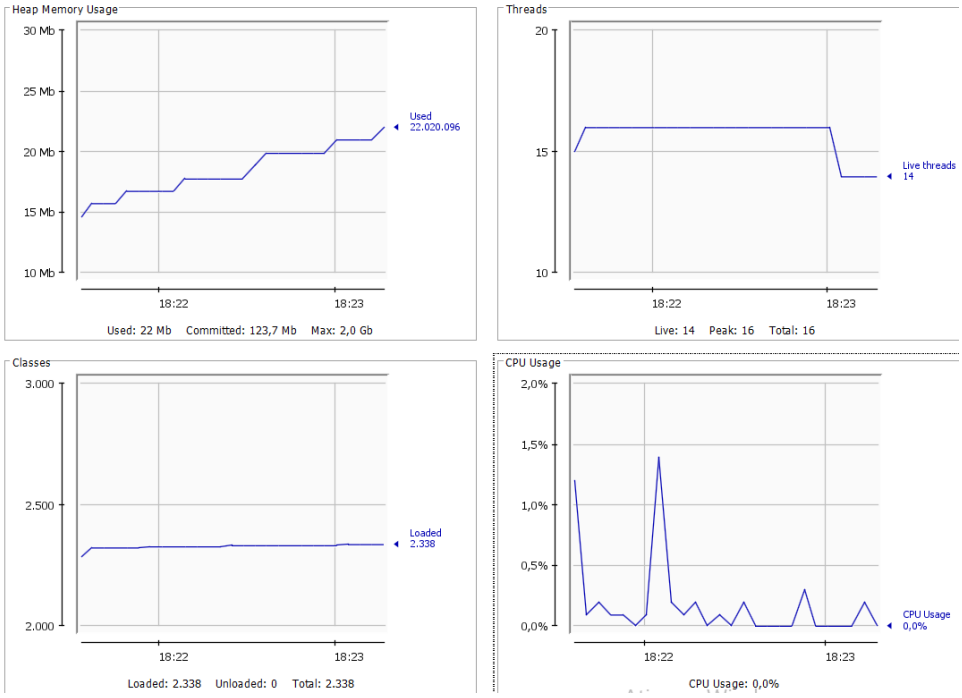


Árvore binária com 1000 elementos:

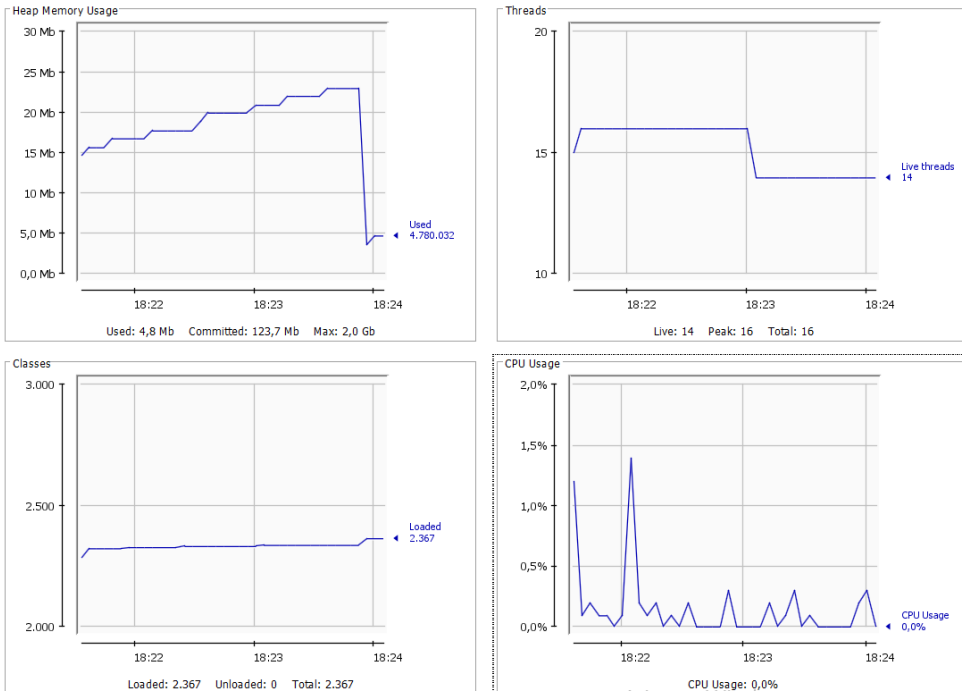
Inserção: 0.002 segundos



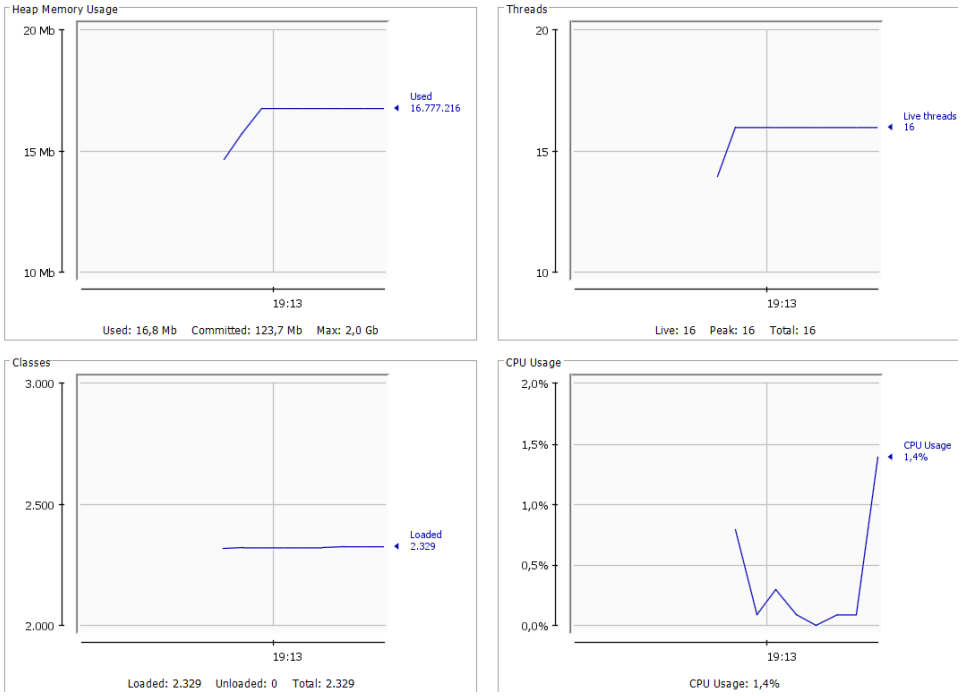
Impressão Pré-Ordem:



Remoção:

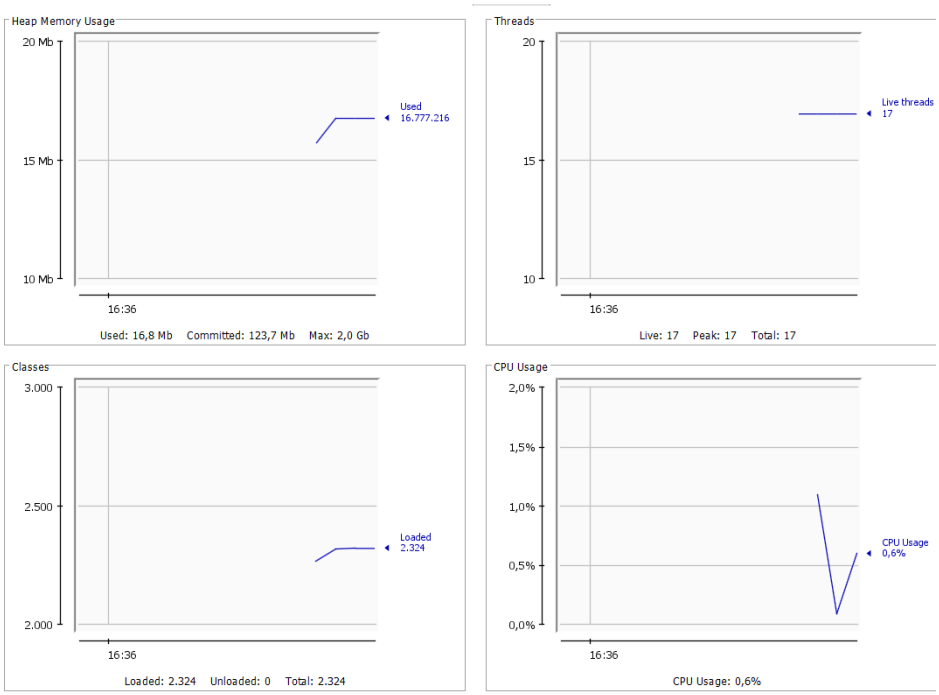


Busca: 1 ms

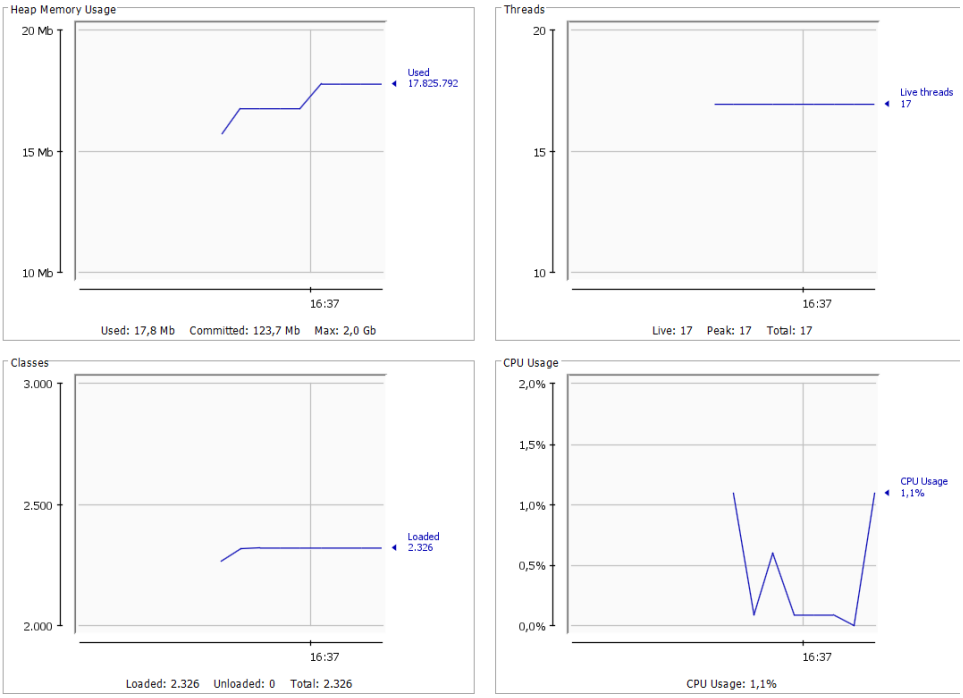


Árvore AVL com 1000 elementos:

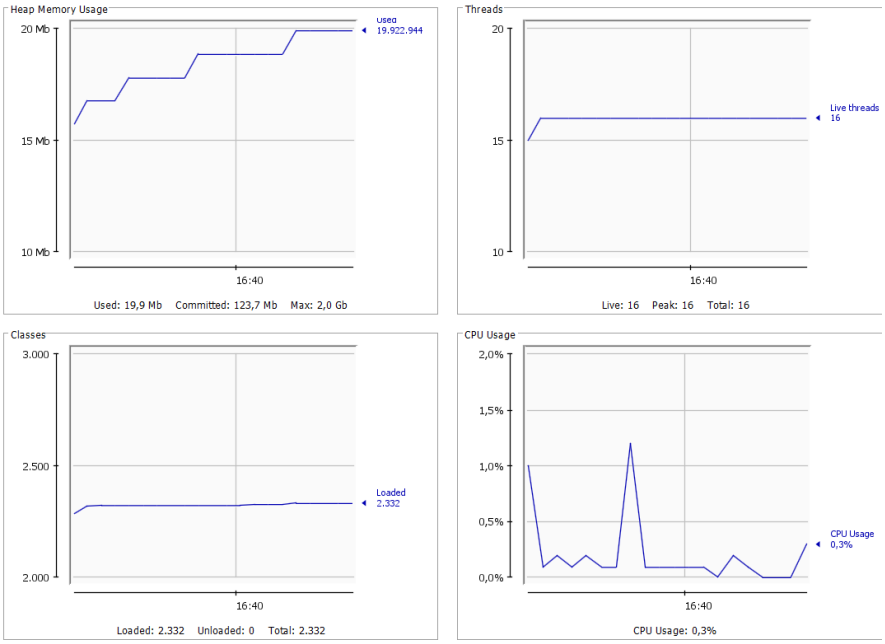
Inserção: 15 ms



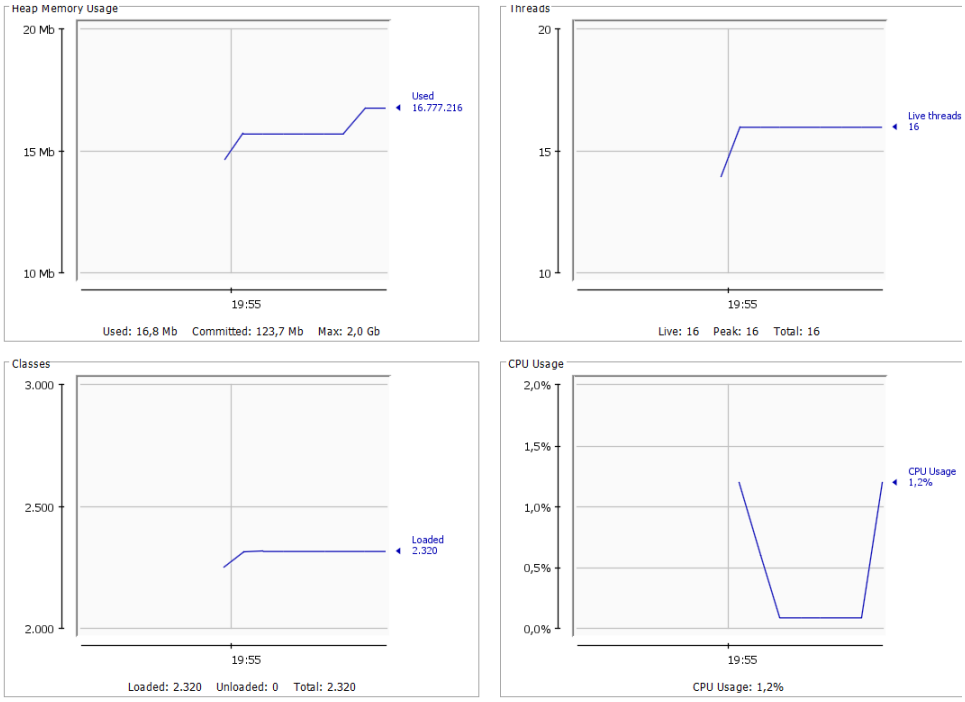
Impresão Pré-Ordem:



Remoção:

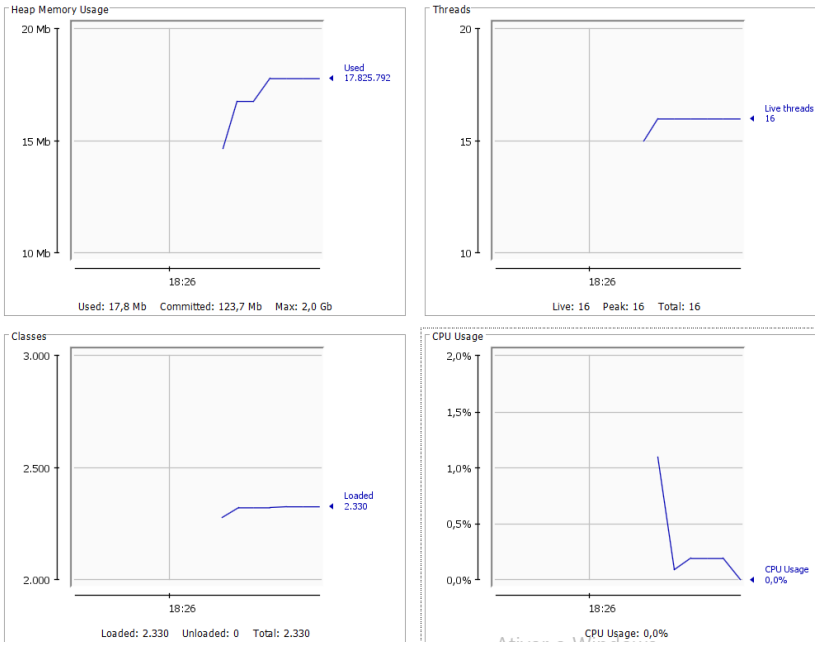


Busca: 1ms

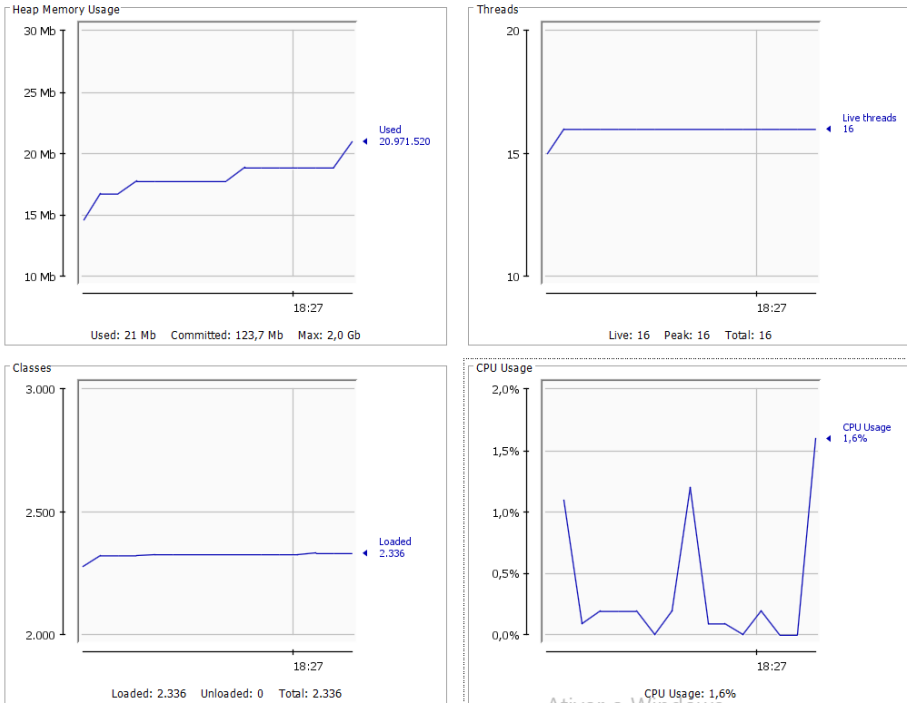


Árvore binária com 10000 elementos:

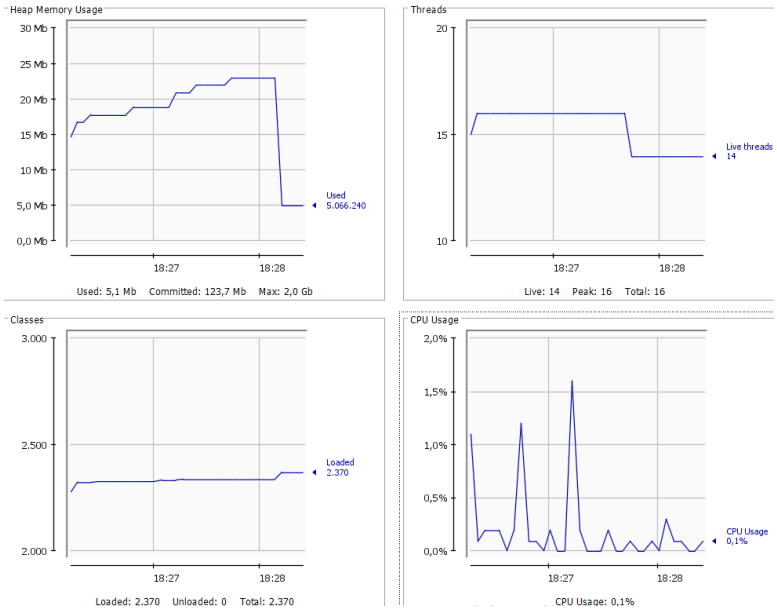
Inserção: 0.013 segundos



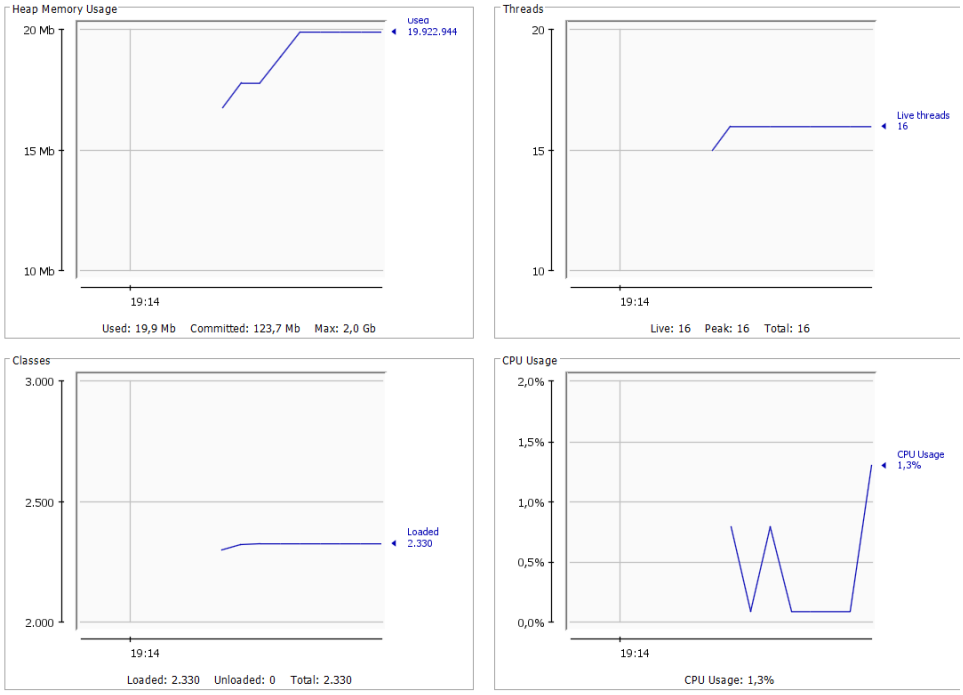
Impressão Pré-Ordem:



Remoção:

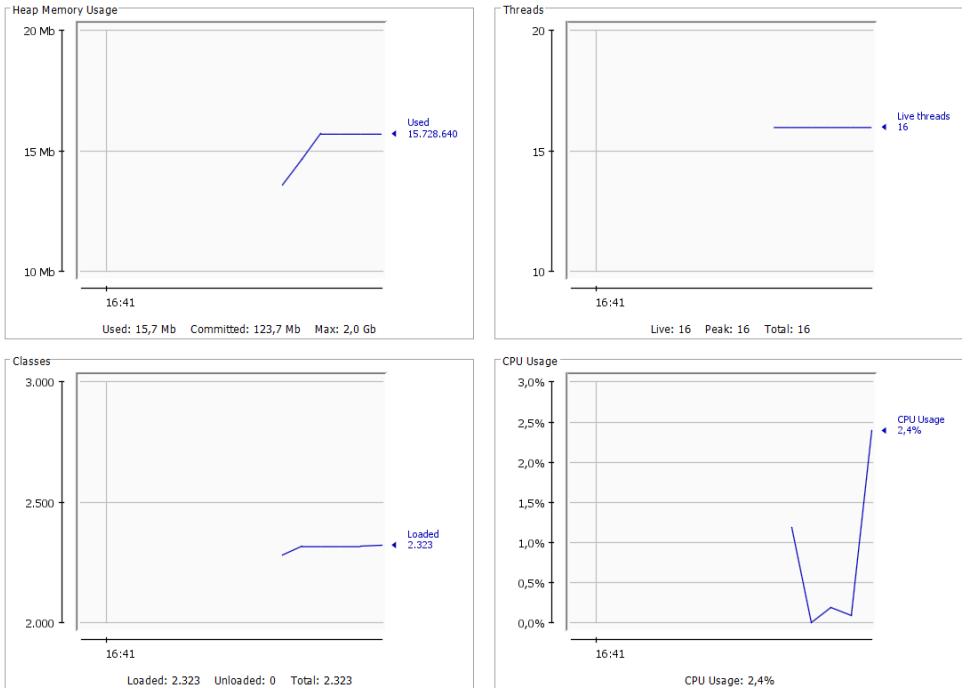


Busca: 1ms

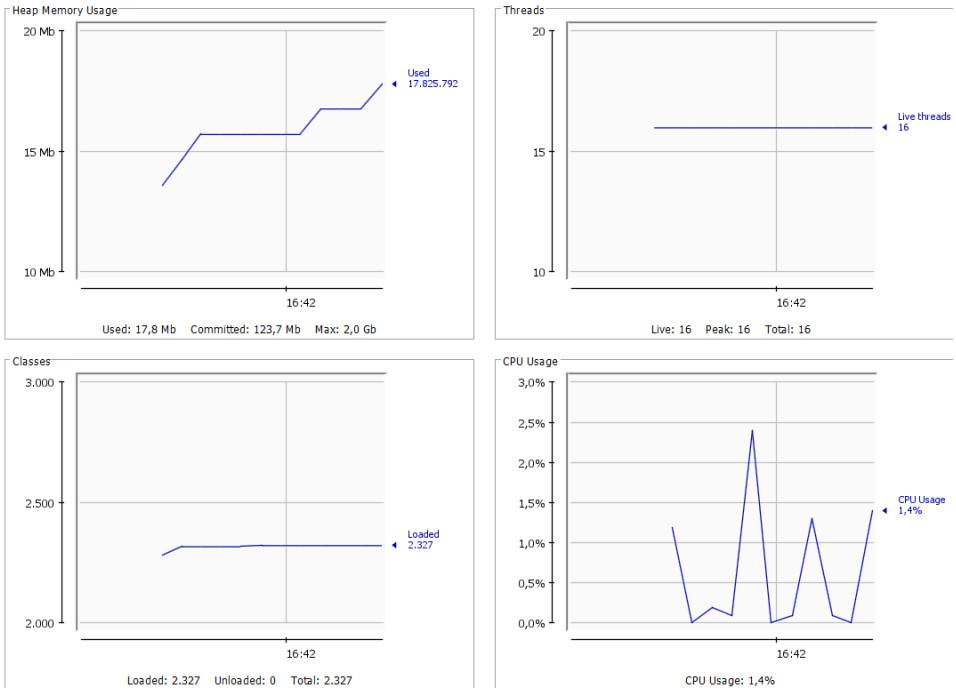


Árvore AVL com 10000 elementos:

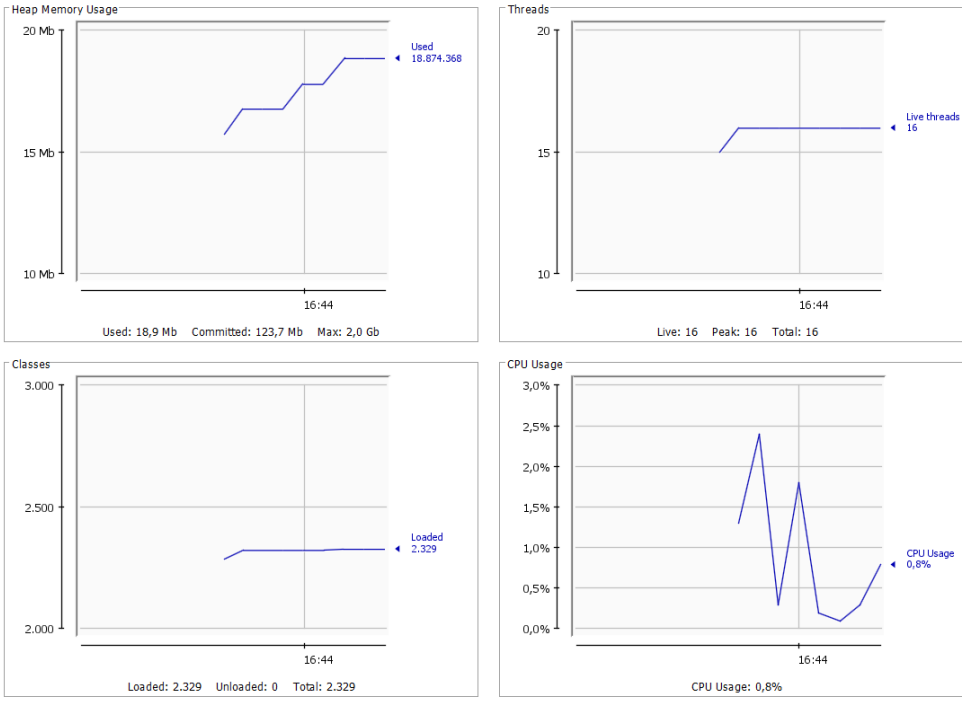
Inserção: 1298 ms



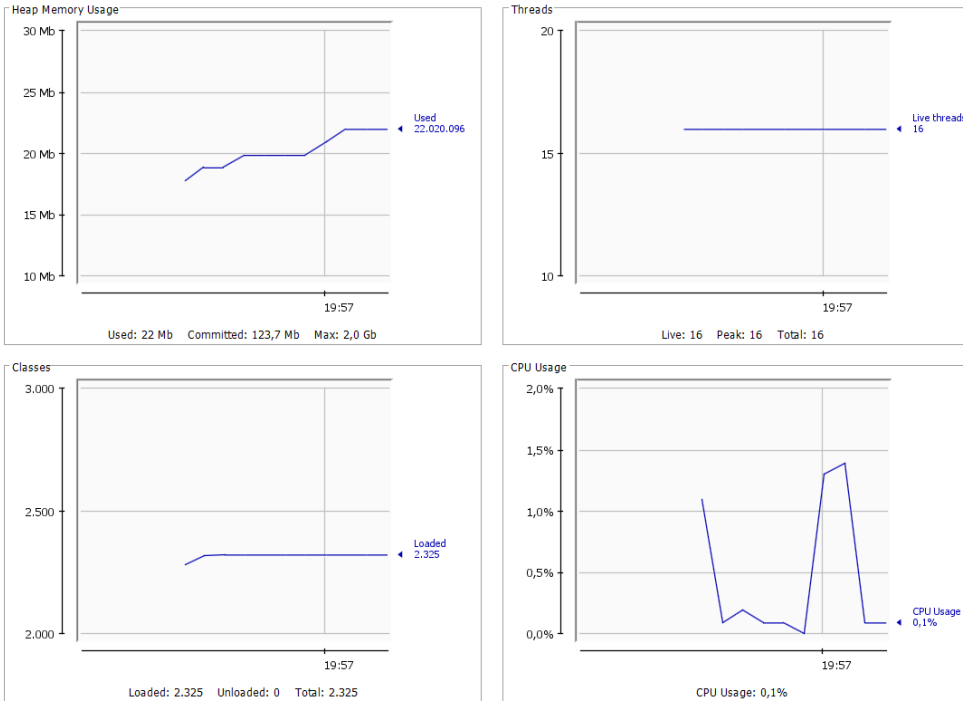
Impresão Pré-Ordem:



Remoção:

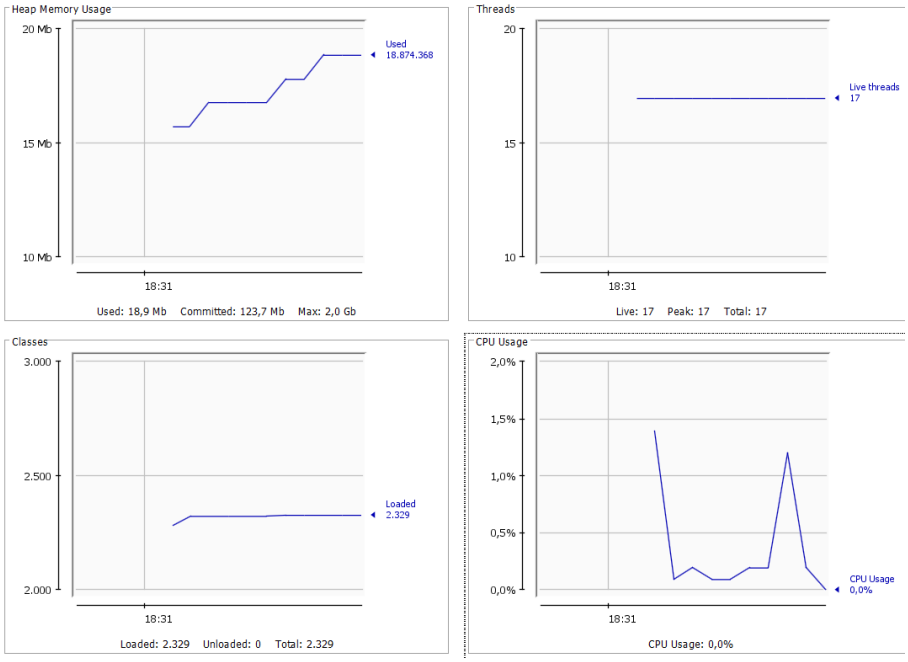


Busca: 2ms

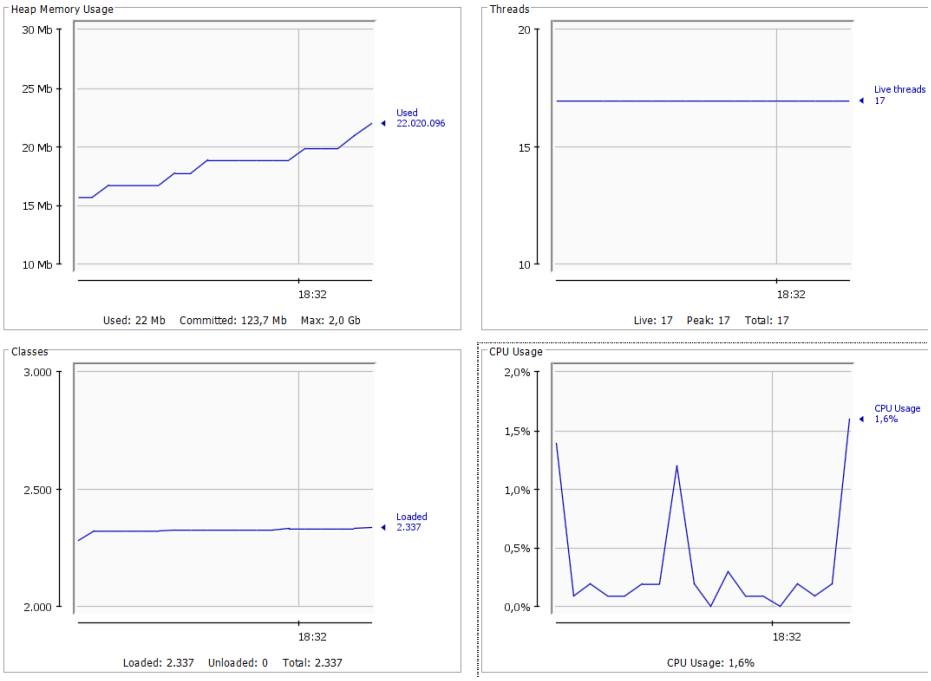


Árvore binária com 20000 elementos:

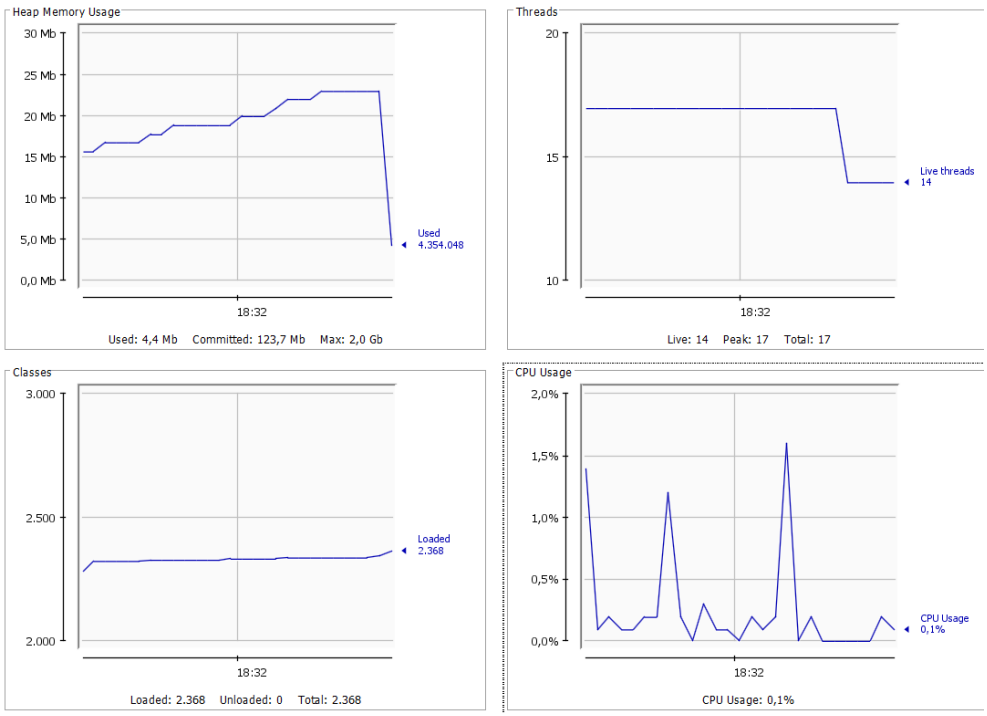
Inserção: 0.024



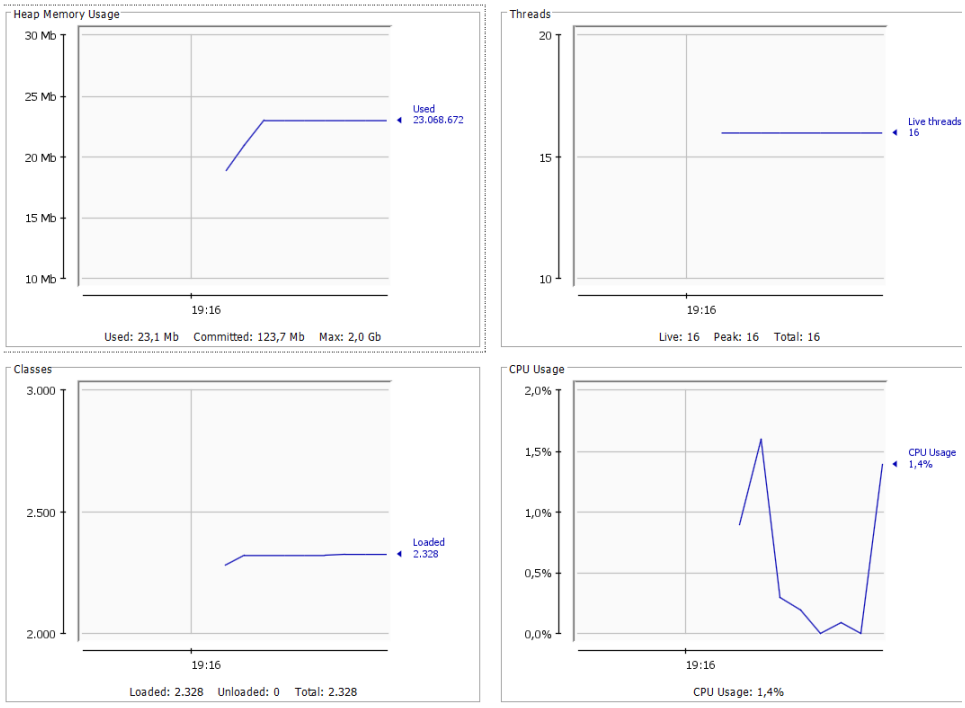
Impressão Pré-Ordem:



Remoção:

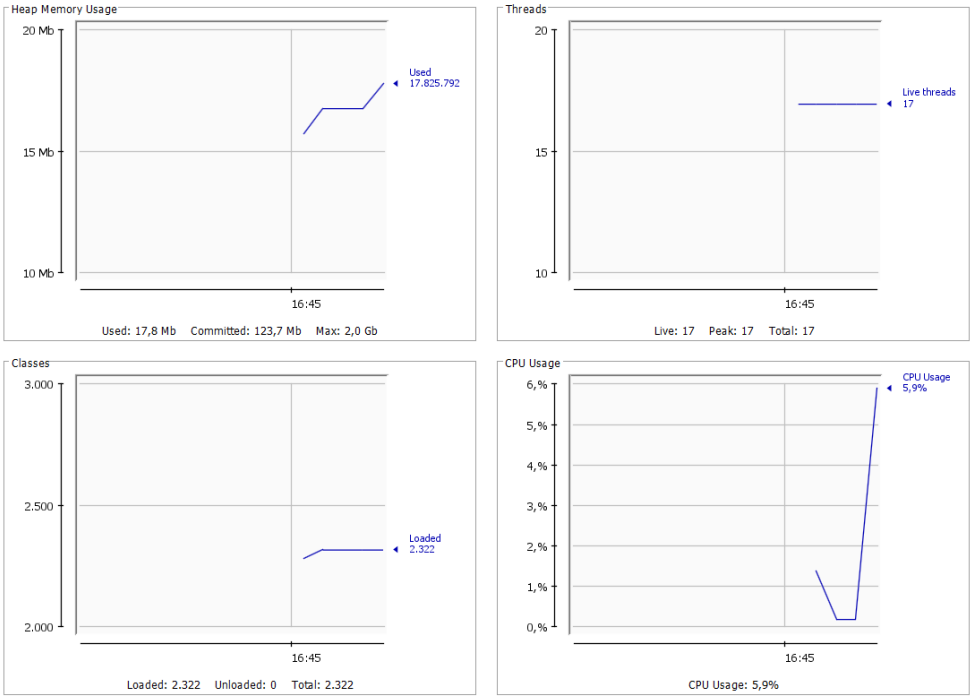


Busca: 2 ms

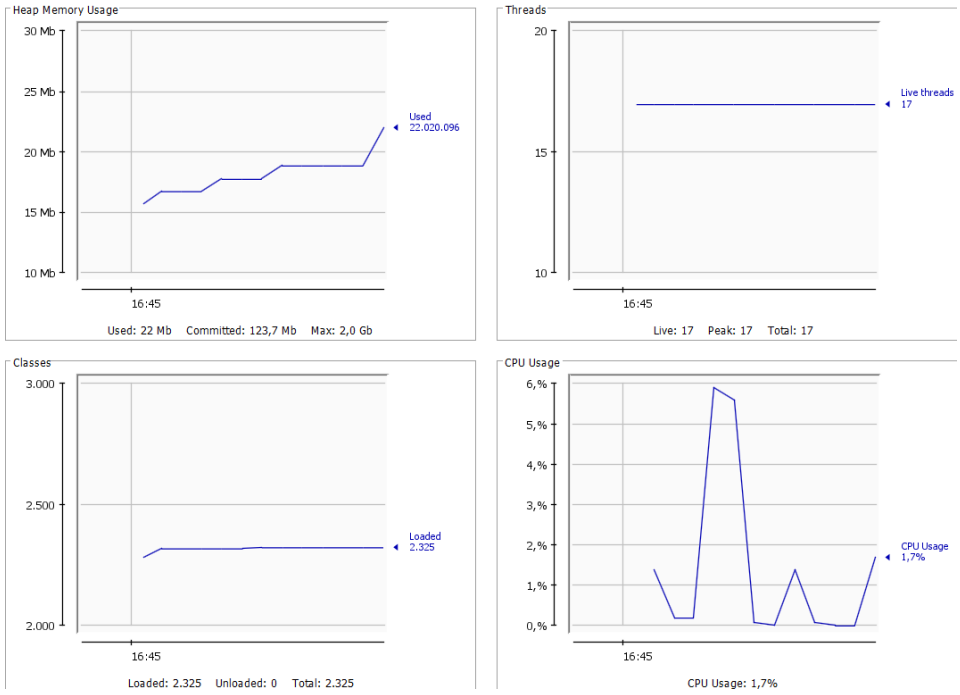


Árvore AVL com 20000 elementos:

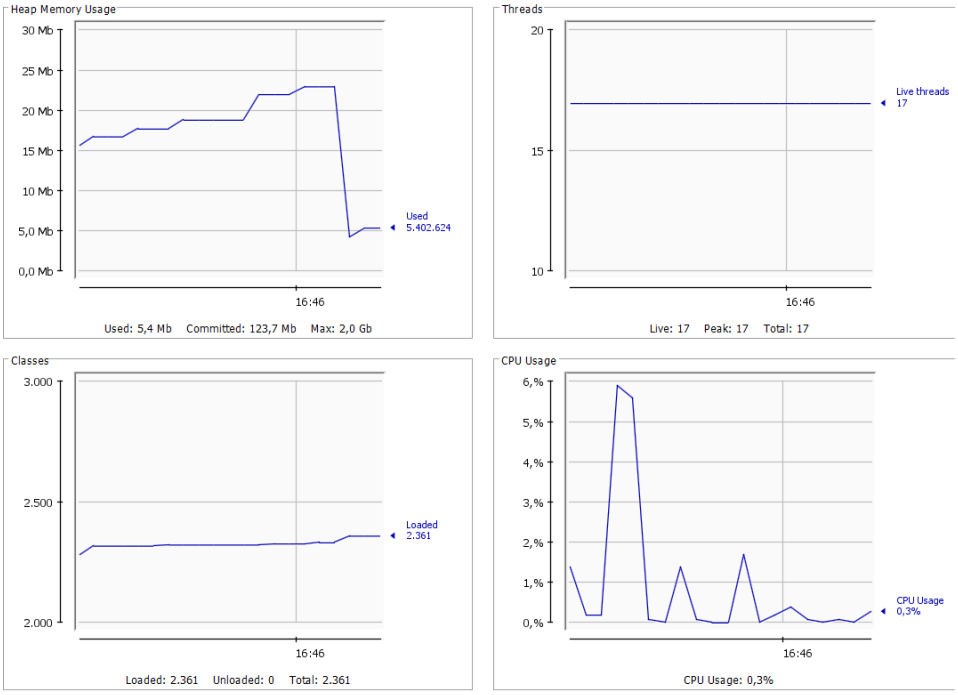
Inserção: 6735 ms



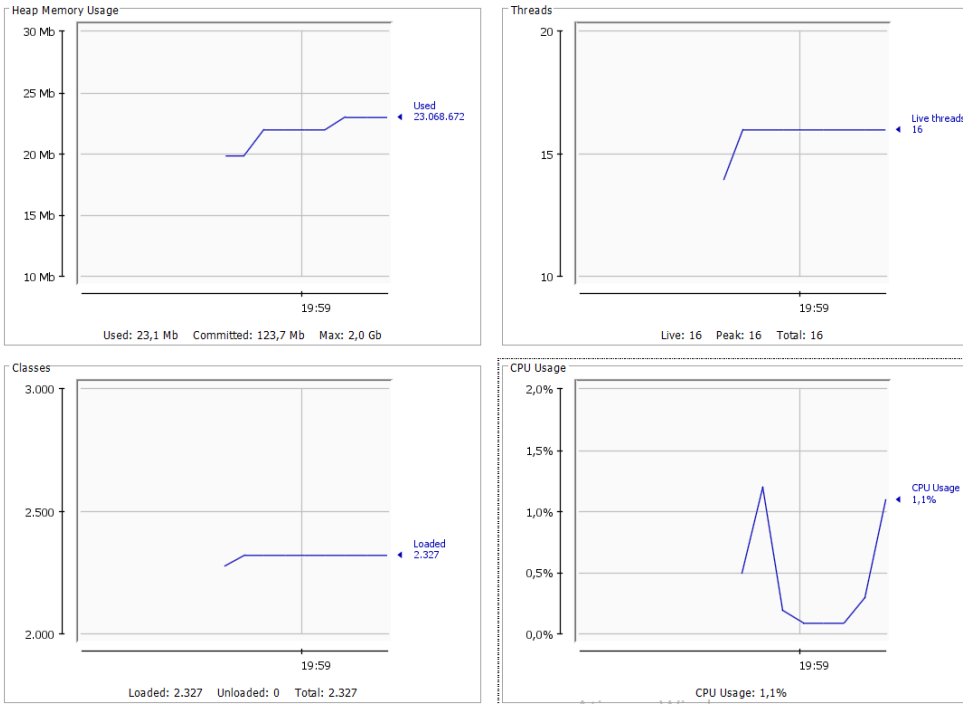
Impresão Pré-Ordem:



Remoção:



Busca: 2 ms



Compare também a remoção e a busca.

- A diferença entre a remoção e a busca encontra o node e verifica os filhos e faz a retação da árvore se necessário, a busca procura o elemento como a remoção e retorna um valor se achar.

Realize uma análise crítica quanto a implementação das arvores e seus métodos.

- Para realizar uma analise critica desse trabalho talvez seja melhor ver os pontos positivos e negativos de cada método:

  Binária:

  - Métodologia de remoção, ciração e inserção mais simples

  - Ocupam um pouco menos espaço e podem ser mais rápidas pois não precisam fazer operações adicionais

  - Dependendo da árvore ela pode ser desbalanceada, o que torna percorrer ela menos eficiente

  AVL:

  - O principal apelo da AVL e o balanceamento automatico de uma árvore, desse jeito percorrer a árvore fica mais eficiente

  - A inserção e remoção de elementos pode ser mais lento, sendo que teremos que fazer as rotações para manter o balanceamento

  - Como criamos mais variaveis(As temporarias para a rotação como exemplo) o programa pode ser um pouco mais pesado e pode utilizar mais memoria para rodar.

Qual funciona melhor? Em qual situação? Porque?

- Dado um vetor ou uma quantidade de números aleatórios, se a sequencia que vai ser inserida na árvore estiver mais ou menos balanceada o método mais rápido e eficiente vai ser a árvore binaria, já que ela não precisa fazer as rotações para deixar a árvore balanceada.

Código

Árvore Binária:

Função de inserir na árvore binária:

```
public static Nodes inserir(Nodes raiz, int numero) {
        if (raiz == null) {
                return new Nodes(numero);
        }
        if (numero >= raiz.getInfo()) {
                 raiz.setDireita(inserir(raiz.getDireita(), numero));
        } else {

                raiz.setEsquerda(inserir(raiz.getEsquerda(), numero));
         }
         return raiz;

}
```

Verifica se é menor ou maior que a raiz e adiciona nos ramos baseado nesta ordem.

Funções de impressão na árvore binária.:

```
public static void imprimirIn(Nodes raiz){

      // Forma inordem

      if (raiz != null) {

          imprimirIn(raiz.getEsquerda());

          System.out.print(raiz.getInfo() + " ");

          imprimirIn(raiz.getDireita());

      }

  }
```

O método imprimirIn funciona da seguinte forma: vai partir da raiz e vai até o máximo da esquerda usando chamada recursiva, raiz.getEsquerda for null o codigo passa para a próxima linha, será printado o numero correspondente, e depois vai passar para a direita e repete o processo até acabar os elementos.

```
public static void imprimirPre(Nodes raiz){
    // Forma préordem
    if (raiz != null) {
        System.out.print(raiz.getInfo() + " ");
        imprimirPre(raiz.getEsquerda());
        imprimirPre(raiz.getDireita());
    }
}
```

ImprimirPre funciona da seguinte maneira: irá printar a raiz e ir para a esquerda com chamada recursiva, printando todo elemento a esquerda e depois vai para a direita printando todo elemento à direita.

```
public static void imprimirPos(Nodes raiz){
    // Forma posordem
    if (raiz != null) {
        imprimirPos(raiz.getEsquerda());
        imprimirPos(raiz.getDireita());
        System.out.print(raiz.getInfo() + " ");
    }
}
```

O método imprimirPos funciona da seguinte maneira: com a chamada recursiva, é percorrido até o último nó, quando esquerda e direita forem nulos, será printado o elemento e por conta da recursividade será printado os elementos acima.

In ordem, pré ordem e pós ordem respectivamente.

Funções para encontrar elementos maiores ou menores:

```
public static Nodes encontrarMaior(Nodes raiz) {
    if (raiz == null) {
```

```
            return null;
        }
        while (raiz.getDireita() != null) {
            raiz = raiz.getDireita();
        }
        return raiz;
    }

public static Nodes encontrarMenor(Nodes raiz) {
        if (raiz == null) {
            return null;
        }
        while (raiz.getEsquerda() != null) {
            raiz = raiz.getEsquerda();
        }
        return raiz;
    }
```

Encontrar os elementos maiores ou menores para substituição.

Função de encontrar posição na árvore:

```
public static int encontrarPosicao(Nodes raiz) {
        if (raiz == null) {
            return 0;
        } else {
            // Para saber onde estamos em uma árvore podemos calcular a altura do lado esquerdo e direito
            // Se a esquerda for maior estamos no lado esquerdo e vice versa.
            int alturaEsquerda = encontrarPosicao(raiz.getEsquerda());
```

```
        int alturaDireita = encontrarPosicao(raiz.getDireita());
        if (alturaEsquerda > alturaDireita) {
            return 1 + alturaEsquerda;
        } else {
            return 1 + alturaDireita;
        }
    }
}
```

Para saber em qual lado da árvore estamos, dado um node. É adicionado 1 pois estamos contando o nó atual como parte do caminho.

Função para remover um elemento da árvore:

```
public static Nodes removerElemento(Nodes raiz, int elemento){
    if(raiz != null){
        if(raiz.getInfo() == elemento){
            System.out.println("Encontramos o número!");
            // Verificamos se o node não possui filhos
            if(raiz.getEsquerda() == null && raiz.getDireita() == null){
                return null
        }
             // Aqui verificamos se ele tem um filho em alguma de suas raizes
            if (raiz.getEsquerda() == null) {
                return raiz.getDireita();
            } else if (raiz.getDireita() == null) {
                return raiz.getEsquerda();
            }
             // Calcule as alturas das subárvores para ver qual lado da árvore estamos
```

```
            int alturaEsquerda = encontrarPosicao(raiz.getEsquerda());
            int alturaDireita = encontrarPosicao(raiz.getDireita());

            // Escolha o elemento substituto com base na altura das subárvores
            if (alturaEsquerda > alturaDireita) {
                // Substituir pelo maior elemento da subárvore esquerda
                Nodes maiorEsquerda = encontrarMaior(raiz.getEsquerda());
                raiz.setInfo(maiorEsquerda.getInfo());
                raiz.setEsquerda(removerElemento(raiz.getEsquerda(),
maiorEsquerda.getInfo()));
            } else {
                // Substituir pelo menor elemento da subárvore direita
                Nodes menorDireita = encontrarMenor(raiz.getDireita());
                raiz.setInfo(menorDireita.getInfo());
                raiz.setDireita(removerElemento(raiz.getDireita(), menorDireita.getInfo()));
            }
        }
        else if (elemento < raiz.getInfo()) {
            // O elemento a ser removido está na subárvore esquerda já que e menor
            raiz.setEsquerda(removerElemento(raiz.getEsquerda(), elemento));
        } else {
            // O elemento a ser removido está na subárvore direita já que e maior
            raiz.setDireita(removerElemento(raiz.getDireita(), elemento));
        }
    }
    return raiz;
}
```

O código de remoção funciona da seguinte maneira, será verificado se o elemento escolhido tem algum filho, se esquerda for null da getDireita, se direita for null da getEsquerda, após isso, será calculado a altura das subarvores esquerda e direita para depois ser escolhido o elemento substituto se a altura da esquerda for maior que da direita vai ser usada a regra “substituir pelo maior elemento da subárvore esquerda”, como foi passado em sala de aula. Caso direita seja maior, usará a regra “substituir pelo menor elemento da subárvore direita”. Depois do elemento ser substituido, o elemento que foi escolhido para substituir deve ser removido e com a recursão isso é resolvido.

No main nós inicializamos o scanner e o random, pedimos o input de elementos para o usuário. Então logo em seguida iniciamos o timer e inserimos os elementos aleatórios na árvore e pega quanto tempo levou este processo e imprimimos o tempo e a árvore criada usando a pré ordem. Perguntamos um número para inserção e outro para remoção para testar a árvore, juntamente mostrando quanto tempo levou cada ação.

Função de busca na árvore:

```
public static boolean busca(Nodes raiz, int elemento){
    if(raiz != null){
        if(raiz.getInfo() == elemento){
            // Encontramos o elemento que estavamos buscando
            System.out.println("Elemento encontrado!");
            return true;
        }

        busca(raiz.getEsquerda(), elemento);
        busca(raiz.getDireita(), elemento);
    }

    return false;
}
```

A busca percorre a árvore procurando o elemento, verificando o nó atual, se direcionando para a esquerda ou direita.

Árvore AVL

Inserção:

```
public static Nodes inserir(Nodes raiz, int numero) {
    // Se a raiz for nula, criamos um novo nó com o número e o retornamos como a nova raiz
    if (raiz == null) {
        return new Nodes(numero);
    }
    // Se o número for maior ou igual ao valor na raiz atual, inserimos à direita
    if (numero >= raiz.getInfo()) {
        raiz.setDireita(inserir(raiz.getDireita(), numero));
    } else {
        // Caso contrário, inserimos à esquerda
        raiz.setEsquerda(inserir(raiz.getEsquerda(), numero));
    }
    // Verificamos o fator de equilíbrio da raiz para determinar se a árvore está desbalanceada
    int equilibrio = calcularAltura(raiz);
     // Rotações necessárias com base no fator de equilíbrio
    if (equilibrio > 1) {
        // Se o fator de equilíbrio for maior que 1, a árvore está desbalanceada para a esquerda
        if (numero < raiz.getEsquerda().getInfo()) {
            // Se o número for menor que o valor no filho esquerdo da raiz, realizamos uma rotação à direita simples
            return rotacaoDireita(raiz);
```

```
            } else {
                // Caso contrário, realizamos uma rotação à esquerda seguida por uma rotação à direita
                raiz.setEsquerda(rotacaoEsquerda(raiz.getEsquerda()));
                return rotacaoDireita(raiz);
            }
        }
        if (equilibrio < -1) {
            // Se o fator de equilíbrio for menor que -1, a árvore está desbalanceada para a direita
            if (numero >= raiz.getDireita().getInfo()) {
                // Se o número for maior ou igual ao valor no filho direito da raiz, realizamos uma rotação à esquerda simples
                return rotacaoEsquerda(raiz);
            } else {
                // Caso contrário, realizamos uma rotação à direita seguida por uma rotação à esquerda
                raiz.setDireita(rotacaoDireita(raiz.getDireita()));
                return rotacaoEsquerda(raiz);
            }
        }
        // Se a árvore está balanceada ou as rotações foram realizadas, retornamos a raiz atualizada
        return raiz;
    }
```

O processo começa verificando se a raiz da árvore é nula. Se for, um novo nó é criado com o número fornecido e é definido como a nova raiz da árvore. Em seguida, com base no valor do número, o novo nó é inserido na subárvore à direita ou à esquerda, seguindo a propriedade de busca binária. Após a inserção, o código calcula o fator de equilíbrio da raiz para determinar se a árvore está desbalanceada. Se o fator de equilíbrio indicar um desbalanceamento à esquerda (maior que 1), o código realiza rotações à direita para restaurar o equilíbrio. Por outro lado, se o fator de equilíbrio

indicar um desbalanceamento à direita (menor que -1), o código realiza rotações à esquerda para restaurar o equilíbrio.

Calcular Altura:

```
public static int calcularAltura(Nodes raiz){
        if(raiz == null){
            return 0;
        }
        int alturaEsquerda = encontrarPosicao(raiz.getEsquerda());
        int alturaDireita = encontrarPosicao(raiz.getDireita());
        return alturaEsquerda - alturaDireita;
    }
```

Vai chamar encontrarPosicao, que vai retornar o valor das alturas esquerda e direita, e depois fazer esquerda – direita e retornar o resultado.

Rotações:

```
public static Nodes rotacaoDireita(Nodes raiz){
        if(raiz != null) {
            Nodes novaRaiz = raiz.getEsquerda();
            Nodes temp = novaRaiz.getDireita();
            novaRaiz.setDireita(raiz);
            raiz.setEsquerda(temp);
            return novaRaiz;
        }
        return raiz;
    }
```

Primeiro, selecionamos a nova raiz, que é o filho esquerdo da raiz original. Em seguida, armazenamos temporariamente a subárvore direita da nova raiz. A nova raiz se torna a raiz original, e a subárvore direita temporária se torna o filho esquerdo da raiz original.

```
public static Nodes rotacaoEsquerda(Nodes raiz){
        If(raiz != null) {
            Nodes novaRaiz = raiz.getDireita();
            Nodes temp = novaRaiz.getEsquerda();
            novaRaiz.setEsquerda(raiz);
            raiz.setDireita(temp);
            return novaRaiz;
}
        return raiz;}
```

Remoção:

```
public static Nodes removerElemento(Nodes raiz, int elemento) {
        if (raiz == null) {
            return raiz;
        }

        if (elemento < raiz.getInfo()) {
            raiz.setEsquerda(removerElemento(raiz.getEsquerda(), elemento));
        } else if (elemento > raiz.getInfo()) {
            raiz.setDireita(removerElemento(raiz.getDireita(), elemento));
        } else {
            // Encontrou o elemento a ser removido

            // Verificar se o nó possui um ou nenhum filho
            if (raiz.getEsquerda() == null || raiz.getDireita() == null) {
                Nodes temp = null;
                if (temp == raiz.getEsquerda()) {
                    temp = raiz.getDireita();
```

```
        } else {
            temp = raiz.getEsquerda();
        }

        // Se não tem filhos, apenas retorna o filho (pode ser nulo)
        if (temp == null) {
            raiz = null;
        } else {
            // Caso com um filho, copia o conteúdo do filho
            raiz = temp;
        }
    } else {
        // Caso com dois filhos, encontrar o sucessor em ordem
        Nodes temp = encontrarMenor(raiz.getDireita());

        // Copiar o conteúdo do sucessor em ordem para este nó
        raiz.setInfo(temp.getInfo());

        // Remover o sucessor em ordem
        raiz.setDireita(removerElemento(raiz.getDireita(), temp.getInfo()));
    }
}

// Se a árvore tinha apenas um nó, retorne
if (raiz == null) {
    return raiz;
}
```

```
    // Verificar o fator de equilíbrio da raiz
    int estado = calcularAltura(raiz);

    // Realizar as rotações necessárias com base no fator de equilíbrio
    if (estado > 1) {
        if (raiz.getInfo() > raiz.getEsquerda().getInfo()) {
            // Rotação à direita simples
            return rotacaoDireita(raiz);
        } else {
            // Rotação à esquerda seguida por rotação à direita
            raiz.setEsquerda(rotacaoEsquerda(raiz.getEsquerda()));
            return rotacaoDireita(raiz);
        }
    }

    if (estado < -1) {
        if (raiz.getInfo() < raiz.getDireita().getInfo()) {
            // Rotação à esquerda simples
            return rotacaoEsquerda(raiz);
        } else {
            // Rotação à direita seguida por rotação à esquerda
            raiz.setDireita(rotacaoDireita(raiz.getDireita()));
            return rotacaoEsquerda(raiz);
        }
    }

    return raiz;
}
```

O método de remoção é semelhante ao da árvore binária comum, vai achar o elemento, verificar se tem filhos e aplicar a regra. A diferença está em que depois da remoção será feito o balanceamento aplicando a rotação com base no valor da variável estado que chama o método calcularAltura que então e usada para comparar com os valores dos Nodes da esquerda e Direita para ver se alguma rotação é possível.

Função de busca na árvore:

```
public static boolean busca(Nodes raiz, int elemento){
    if(raiz != null){
        if(raiz.getInfo() == elemento){
            // Encontramos o elemento que estavamos buscando
            System.out.println("Elemento encontrado!");
            return true;
        }

        busca(raiz.getEsquerda(), elemento);
        busca(raiz.getDireita(), elemento);
    }

    return false;
}
```

A busca percorre a árvore procurando o elemento, verificando o nó atual, se direcionando para a esquerda ou direita.

Conclusão:

A escolha entre a árvore Binária e AVL depende de como ela vai ser utilizada, levando em conta quantos elementos ela vai ter, se os elementos já estão mais balanceados e o quanto de poder de processamento e espaço deve utilizar. Se você necessita de uma árvore simples e que não ocupe muito espaço a árvore binaria seria recomendada, ou se precisar de uma árvore que consiga cuidar de muitos elementos

com desbalanceamento, seja mais complexa mas pode ter um melhor desempenho a árvore AVL seria a recomendada.

Foi observado que a árvore binária tem suas inserções mais rápidas, porém ela fica desordenada, e é na ordenação que a árvore AVL tem vantagem, como por exemplo na busca que para a árvore binária, foi utilizado mais recurso do que na árvore AVL. A remoção de um nó de uma árvore binária resulta em um desbalanceamento, já na árvore AVL, mesmo que demande um pouco mais de processamento, a árvore sempre estará balanceada. Em geral, a árvore AVL compensa o seu custo adicional com um desempenho consistente.