CENTRO FEDERAL DE EDUCAÇÃO TECNOLÓGICA DE MINAS GERAIS
CAMPUS TIMÓTEO

João Pedro Pereira de Assis Castro Santos
Juliana Silva Cruz Sartori

BST
(ÁRVORE DE BUSCA BINÁRIA)

Timóteo

2022

1 Introdução

Em 1960, PF Windley, AD Booth, AJT Colin e TN Hibbard inventaram a Árvore de Busca Binária(binary search trees = BSTs). O objetivo da Árvore de Pesquisa Binária é garantir que o usuário tenha bons resultados na pesquisa, inserção e exclusão de elementos da árvore. A vantagem da árvore de pesquisa binária em comparação com uma lista vinculada é que a árvore de pesquisa binária é mais rápida na pesquisa quando balanceada(Sutori, 2021). É interessante notar que, quando se faz um percurso em ordem em uma árvore binária de busca, os valores dos nós aparecem em ordem crescente, generalizando a ideia de listas encadeadas crescentes0 . (Paulo Feofiloff, 2018)

Uma árvore binária é uma estrutura de dados recursiva onde cada nó pode ter no máximo 2 filhos. Um tipo comum de árvore binária é uma árvore de busca binária , na qual cada nó tem um valor maior ou igual aos valores dos nós na subárvore esquerda e menor ou igual aos valores dos nós na subárvore direita.(Marcos Lopez Gonzalez, 2022)

Em 1962, Georgy Adelson-Velsky e EM Landis inventaram a Árvore AVL, no qual seu nome vem como homenagem Esta árvore foi a primeira árvore de busca binária auto-balanceada a ser inventada . Na Árvore AVL as alturas das duas subárvores filhas de qualquer nó diferem em no máximo 1 para manter um tempo de busca de O(log n). A vantagem da Árvore AVL é que há uma garantia de complexidade de tempo de pior caso de O(log n) para cada uma das operações do dicionário. A desvantagem da árvore é que ela precisa de armazenamento de memória extra para a altura de cada nó para garantir que ela seja equilibrada.(Sutori, 2021)

Especificamente, o tempo de execução da pesquisa para uma árvore de pesquisa binária é O(log n) enquanto a lista vinculada é O(n). A desvantagem da árvore é que ela pode se degenerar e o tempo de busca pode ser longo. A estrutura Binary Search Tree terá os mesmos tempos de execução de uma lista vinculada, mas terá uma estrutura mais complicada, o que não é um resultado desejável.(Paulo Feofiloff, 2018)

2 Desenvolvimento

A BST possui alguns métodos importantes para ser originada, no qual foi implementado no código elaborado. O método de busca em uma árvore binária por um valor específico pode ser um processo recursivo ou iterativo. A busca começa examinando o nó raiz, se a árvore está vazia, o valor procurado não pode existir na árvore. Caso contrário, se o valor é igual a raiz, a busca foi bem sucedida. Se o valor é menor do que a raiz, a busca segue pela subárvore esquerda. Similarmente, se o valor é maior do que a raiz, a busca segue pela subárvore direita. Esse processo é repetido até o valor ser encontrado ou a subárvore ser nula (vazia). Se o valor não for encontrado até a busca chegar na subárvore nula, então o valor não deve estar presente na árvore.(Wikipedia, 2021)

Analogamente, o método de inserção começa com uma busca, procurando pelo valor, mas se não for encontrado, procuram-se as subárvores da esquerda ou direita, como na busca. Eventualmente, alcança-se a folha, inserindo-se então o valor nesta posição. Ou seja, a raiz é examinada e introduz-se um nó novo na subárvore da esquerda se o valor novo for menor do que a raiz, ou na subárvore da direita se o valor novo for maior do que a raiz.(Wikipedia, 2021)

Em síntese, a retirada de um registro não é tão simples quanto a inserção. Se o nó que contém o registro a ser retirado possui no máximo um descendente, a operação é simples. Já no caso do nó conter dois descendentes o registro a ser retirado deve ser primeiro: – substituído pelo registro mais à direita na subárvore esquerda; – ou pelo registro mais à esquerda na subárvore direita(PROJETO. . . , 2006). Destarte, o método de exclusão de um nó é um processo mais complexo. Para excluir um nó de uma árvore binária de busca, quando se vai fazer a remoção de nó que possui dois filhos, pode-se operar de duas maneiras diferentes. É viável substituir o valor do nó a ser retirado pelo valor sucessor (o nó mais à esquerda da subárvore direita) ou pelo valor antecessor (o nó mais à direita da subárvore esquerda), removendo-se aí o nó sucessor (ou antecessor). No entanto, a exclusão na folha é a mais simples, basta removê-lo da árvore onde, excluindo-o, o filho sobe para a posição do pai.(Wikipedia, 2021)

Ademais, a Árvores Binárias de Pesquisa possui níveis, onde o nível do nó raiz é 0, porém se um nó está no nível i, então a raiz de suas subárvores estão no nível i + 1, e assim sucessivamente. Contudo a altura de uma árvore é analisada de um nó é o comprimento do caminho mais longo deste nó até um nó folha, sendo assim a altura de uma árvore é a altura do nó raiz.(PROJETO. . . , 2006)

Não obstante, é preciso compreender a complexidade das operações sobre ABB depende diretamente da altura da árvore. Uma árvore binária de busca com chaves aleatórias uniformemente distribuídas tem altura O(log n). No pior caso, uma ABB poderá ter altura O(n). Neste caso a árvore é chamada de árvore zig-zag e corresponde a uma degeneração da árvore em lista encadeada. Em função disso, a árvore binária de busca é de pouca utilidade para ser aplicada em problemas de busca em geral. Conseguinte, é por esse motivo que há um grande interesse em árvores balanceadas, cuja altura seja O(log n) no pior caso.(Wikipedia, 2021).

Na execução do álgoritmo requisitado foram preenchidas árvores de 3 maneiras, um método com entradas ordenadas, um método com entradas aleatórias, e um terceiro método com inserção balanceada. Outro objetivo era a ideia de utilizar esses métodos para 5 tamanhos de árvore, contendo respectivamente 10^1,10^3,10^5,10^7 e 10^9 elementos, o que não foi possível por limitações de poder computacional, então apenas os 2 primeiros casos foram testados. Ao preencher os vetores com valores aleatórios, por limitações do xorshift usado no trabalho anterior, não foi possível produzir valores específicos dentro do período, o que levou a tempo de execução demorado, sem garantia de que uma nova semente produziria valores inteiros diferentes dos já existentes, então em nome da praticidade, no preenchimento foi usado java.util.Random. Por fim, foram feitas 30 buscas de valores aleatórios diferentes gerados pelo nosso algoritmo xorshift, calculamos a média e implementamos uma função standardDeviation para calcular o desvio padrão dos tempos de busca. A seguir o código implementado:

A classe Main, que executa:

```
package binarytree1;
import java.util.Arrays;
import java.util.Random;

public class BinaryTree1 {

    public static void main(String[] args) {
        doTreeSizes(10);
        doTreeSizes(1000);
    }

    static void doTreeSizes(int k) {
        PRNG ourrng = new PRNG();
        Tree treeste = new Tree();
        Tree treeste2 = new Tree();
        Tree treeste3 = new Tree();
        Random rng = new Random();
        int vetknow[] = new int[144];
        int v[] = new int[k];
        int v2[] = new int[v.length];
        int v3[] = new int[v.length];

        System.out.println("I-rvores binrias com valores ordenados(" + k + ")");
        long timestart = System.nanoTime();
        for (int i = 0; i < v.length; i++) {
            treeste.add(i);
        }
        System.out.println("Tempo Insercao:" + (System.nanoTime() - timestart));

        for (int i = 0; i < 30; i++) {
            int b1;//quantidade maxima de bits
            do {
                b1 = ourrng.nextInt(144);
```

```
    } while (vetknow[b1] == b1);
    vetknow[b1] = b1;
}
int n;
boolean v2bool[] = new boolean[v.length];
for (int i = 0; i < v2.length; i++) {
    n = rng.nextInt(v2.length);
    while (v2bool[n]) {
        n = rng.nextInt(v2.length);
    }
    v2[i] = n;
    v2bool[n] = true;
}

System.out.println("B-rvores binrias com valores ordenados(" + k + ")");
double media = 0;
long timesearch = System.nanoTime();
long vetTimeArm[] = new long[v2.length];
for (int i = 0; i < v2.length; i++) {
    timestart = System.nanoTime();
    treeste.search(v2[i]);
    vetTimeArm[i] = System.nanoTime() - timestart;
    media += vetTimeArm[i]; //media = media + vetTimeArm[i];
}
System.out.println("Tempo Busca " + (System.nanoTime() - timesearch));
System.out.println("Media: " + (media / vetTimeArm.length));
System.out.println("Desvio padrao: " + standardDeviation(vetTimeArm));

System.out.println("I-rvores binrias com valores aleatrios(" + k + ")");
timestart = System.nanoTime();
for (int i = 0; i < v2.length; i++) {
    treeste2.add(v2[i]);
}
System.out.println("Tempo Insercao:" + (System.nanoTime() - timestart));
vetknow = new int[144];
for (int i = 0; i < 30; i++) {
    int b2;
    do {
        b2 = ourrng.nextInt(144);
    } while (vetknow[b2] == b2);
    vetknow[b2] = b2;
}
for (int i = 0; i < v3.length; i++) {
    v3[i] = i;
}
Arrays.sort(v3);
ArrayToBST(v3);
```

```
        System.out.println("B-rvores binrias com valores aleatrios(" + k + ")");
        media = 0;
        timesearch = System.nanoTime();
        vetTimeArm = new long[v2.length];
        for (int i = 0; i < v2.length; i++) {
            timestart = System.nanoTime();
            treeste2.search(v2[i]);
            vetTimeArm[i] = System.nanoTime() - timestart;
            media += vetTimeArm[i]; //media = media + vetTimeArm[i];
        }
        System.out.println("Tempo Busca " + (System.nanoTime() - timesearch));
        System.out.println("Media: " + (media / vetTimeArm.length));
        System.out.println("Desvio padrao: " + standardDeviation(vetTimeArm));

        System.out.println("I-rvores binrias balanceadas(" + k + ")");
        timestart = System.nanoTime();
        for (int i = 0; i < v3.length; i++) {
            treeste3.add(v3[i]);
        }
        System.out.println("Tempo Insercao:" + (System.nanoTime() - timestart));
        vetknow = new int[144];
        for (int i = 0; i < 30; i++) {
            int b3;
            do {
               b3 = ourrng.nextInt(144);
            } while (vetknow[b3] == b3);
            vetknow[b3] = b3;
        }
        System.out.println("B-rvores binrias balanceadas(" + k + ")");
        media = 0;
        timesearch = System.nanoTime();
        vetTimeArm = new long[v2.length];
        for (int i = 0; i < v2.length; i++) {
            timestart = System.nanoTime();
            treeste2.search(v2[i]);
            vetTimeArm[i] = System.nanoTime() - timestart;
            media += vetTimeArm[i]; //media = media + vetTimeArm[i];
        }
        System.out.println("Tempo Busca " + (System.nanoTime() - timesearch));
        System.out.println("Media: " + (media / vetTimeArm.length));
        System.out.println("Desvio padrao: " + standardDeviation(vetTimeArm));
    }

    static void ArrayToBST(int v[]) {
        sortedArrayToBST(v, 0, v.length, 0);
    }
```

```
    static int sortedArrayToBST(int v[], int first, int last, int count) {
        boolean v2[] = new boolean[v.length];
        int middle = (first + last) / 2;
        if (count < v.length && v2[middle] == false) {
            v[count] = middle;
            v2[middle] = true;
            if (middle != first) {
                count = sortedArrayToBST(v, first, middle - 1, count + 1);
            }
            if (middle != last) {
                count = sortedArrayToBST(v, middle + 1, last, count + 1);
            }
            return count;
        }
        return 0;
    }

    static double standardDeviation(long v[]) {
        double standard_deviation = calculateSD(v);
        return standard_deviation;
    }

    public static double calculateSD(long v[]) {
        double sum = 0.0, standard_deviation = 0.0;
        int v_size = v.length;
        for (double temp :v) {
            sum += temp;
        }
        double mean = sum / v_size;
        for (double temp : v) {
            standard_deviation += Math.pow(temp - mean, 2);
        }
        return Math.sqrt(standard_deviation / v_size);
    }
}
```

A classe de Nós, na qual cria os nós da árvore:

```
package binarytree1;

class Node {
    int item;
    Node leftbranch;
    Node rightbranch;

    Node(int item) {
        this.item = item;
        leftbranch = null;
```

```
        rightbranch = null;
    }

    @Override
    public String toString() {
        return String.valueOf(this.item);
    }

}
```

A classe PRNG, cria os 30 números aleatórios:

```
public class PRNG {

    private long last;
    private long inc;
    private int count = 0;

    public PRNG() {
        long seed = System.nanoTime();
        this.last = seed | 1;
        inc = seed;
    }

    public int nextInt(int max) {
        if (count == 144) {
            long seed = System.nanoTime();
            this.last = seed | 1;
            inc = seed;
        }
        last ^= (last << 400);
        last ^= (last >>> 700);
        last ^= (last << 100);
        inc += 1234567891234567 89L; //num magico(incremento)
        int out = (int) ((last + inc) % max);
        count++;
        return (out < 0) ? -out : out;
    }
}
```

A classe Tree, na qual cria a estrutura da árvore:

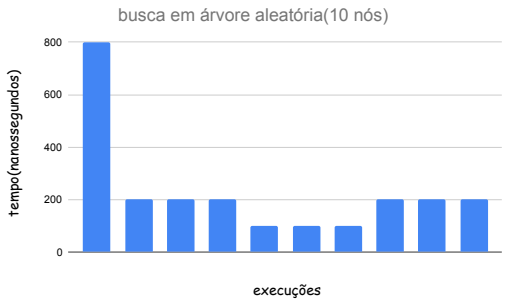
```
public class Tree {

    Node root;

    public void add(int newitem) {
        addRecursive(root, newitem);
```

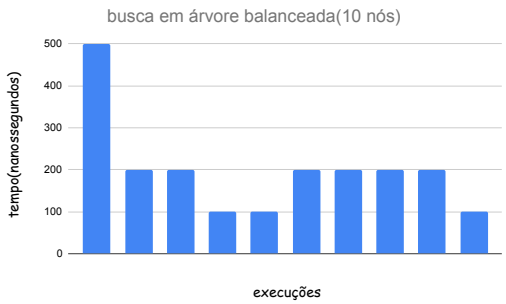
```
}

public void addRecursive(Node currentnode, int newitem) {//no atual, item a ser
    inserido
    if (root == null) {//retorna um no criado com o item a ser inserido
        root = new Node(newitem);
    } else if (newitem < currentnode.item) {//se o valor for menor que o do no
        atual vamos pra esquerda
        if (currentnode.leftbranch != null) {
            addRecursive(currentnode.leftbranch, newitem);
        } else {
            currentnode.leftbranch = new Node(newitem);
        }

    } else if (newitem > currentnode.item) {//se o valor for maior que o do no
        atual vamos pra direita
        if (currentnode.rightbranch != null) {
            addRecursive(currentnode.rightbranch, newitem);
        } else {
            currentnode.rightbranch = new Node(newitem);
        }
    }          //se o valor for igual ao do no atual retornamos o no atual

}

public boolean search(int searchitem) {
    return searchRecursive(root, searchitem);
}

public boolean searchRecursive(Node currentnode, int searchitem) {
    if (currentnode == null) {
        return false;
    }
    if (searchitem < currentnode.item) {
        return searchRecursive(currentnode.leftbranch, searchitem);
    } else if (searchitem > currentnode.item) {
        return searchRecursive(currentnode.rightbranch, searchitem);
    } else if (searchitem == currentnode.item) {
        return true;
    }
    return false;
}

public int findFilhotao(Node root) {
    if (root.rightbranch == null) {
        return root.item;
    } else {
        return findFilhotao(root.rightbranch);
```

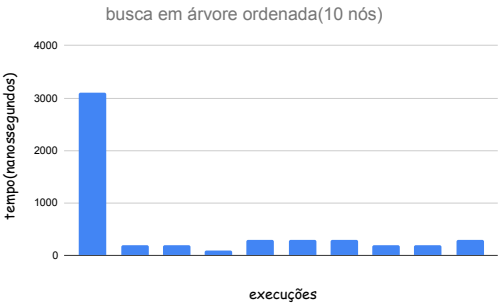
```
    }
}

public void delete(int deleteitem) {
    root = deleteRecursive(root, deleteitem);
}

public Node deleteRecursive(Node currentnode, int deleteitem) {
    if (currentnode == null) {
        return null;
    }
    if (deleteitem < currentnode.item) {
        deleteRecursive(currentnode.leftbranch, deleteitem);
    } else if (deleteitem > currentnode.item) {
        deleteRecursive(currentnode.rightbranch, deleteitem);
    }
    if (deleteitem == currentnode.item) {
        if ((currentnode.leftbranch == null) && (currentnode.rightbranch == null))
            {
            return null;
        }
        if (currentnode.leftbranch == null) {
            return currentnode.rightbranch;
        }
        if (currentnode.rightbranch == null) {
            return currentnode.leftbranch;
        }
        int highest = findFilhotao(currentnode.leftbranch);
        currentnode.item = highest;
        currentnode.leftbranch = deleteRecursive(currentnode.leftbranch, highest);
        return currentnode;
    }
    return currentnode;
}

public void printree() {
    printPretty(root, 0, 0);
}

public void traverseInOrder(Node node, int lvl) {
    if (node != null) {
        traverseInOrder(node.leftbranch, lvl - 1);
        System.out.print(lvl + ": " + node.item + ". ");
        traverseInOrder(node.rightbranch, lvl + 1);
    }
}

public void printPretty(Node node, int left, int right) {
```

```
        if (node != null) {
            System.out.println("("+"L: "+node.leftbranch+" R: "+node.rightbranch+"
                item: "+node.item+")");
            printPretty(node.leftbranch, left+1,right);
            printPretty(node.rightbranch, left, right+1);
        }

    }
}
```

GRÁFICOS DE TEMPO DE EXECUÇÃO DA BUSCA:







3 Conclusão

Concluiu-se que os tempos de inserção e busca se comportaram como esperado, sendo os piores quando a árvore recebe valores ordenados, formando uma lista, pois para cada inserção e busca, todos os elementos devem ser percorridos até encontrar o local de inserção ou o nó onde está inserido o valor em questão. O caso onde os elementos são randomizados é um pouco variante, mas tende a ser o caso "médio", pois às vezes consegue um elemento favorável para uma inserção rápida, mas da mesma forma, pode ter um elemento com pior tempo de inserção ou busca possível. Já o caso onde os elementos são inseridos de forma que suas posições são mapeadas para balanceamento tende a ser o caso mais rápido, pois devido a sua organização, o elemento tem seu lugar "reservado"e um caminho bem definido que faz com que a busca ou inserção percorra o mínimo possível de nós da árvore para alcançar seu alvo.

Referências

Marcos Lopez Gonzalez. *Implementing a Binary Tree in Java*. 2022. Disponível em: <https://www.baeldung.com/java-binary-tree>. Acesso em: 13 de setembro 2022. Citado na página 1.

Paulo Feofiloff. *Árvores binárias de busca*. 2018. Disponível em: <https://www.ime.usp.br/~pf/algoritmos/aulas/binst.html>. Acesso em: 13 de setembro 2022. Citado na página 1.

PROJETO de Algoritmos. *Nivio Ziviani*, 2006. Citado na página 2.

Sutori. *O que veio antes da árvore do bode expiatório?* 2021. Disponível em: <https://www.sutori.com/en/story/what-came-before-the-scapegoat-tree--2hEjTfKVQuKJ8SbJBnLES4oD>. Acesso em: 14 de setembro 2022. Citado na página 1.

Wikipedia. *Árvore binária de busca*. 2021. Disponível em: <https://pt.wikipedia.org/wiki/>. Acesso em: 16 de setembro 2022. Citado na página 2.