

# Um Equilíbrio Delicado

## Algoritmos e Estrutura de Dados II

Daniela Pereira Rigoli

<sup>1</sup>Escola Politécnica – Pontifícia Universidade Católica do Rio Grande do Sul (PUCRS)  
Caixa Postal 1429 – 90619-900 – Porto Alegre – RS – Brazil

**Resumo.** *Este artigo descreve uma solução para o primeiro problema proposto na disciplina de Algoritmos e Estruturas de Dados II, que se trata de uma rede de computadores que distribuem tarefas entre si. É apresentado mais detalhes sobre a problemática, assim como, resultados para casos de testes propostos na disciplina, ideias, estruturas usadas, pseudo-código dos algoritmos, dificuldades encontradas e testes para validar os resultados.*

### 1. Introdução

Este trabalho foi desenvolvido para um projeto que planeja construir uma rede de centros de computadores espalhados pelo mundo, que trocarão tarefas entre si e terão poder computacional nunca visto. O funcionamento do conjunto de centros consiste em:

- Um computador central recebe uma tarefa  $T$  e avalia quanto esforço  $E$  é necessário para resolvê-la. Por exemplo, ele pode receber uma tarefa e avaliar que  $E = 10430$ .
- Se o computador avaliar que o esforço é pequeno, ele mesmo resolve a tarefa e devolve o resultado.
- Se o esforço for considerado grande demais, o computador divide a tarefa em duas e envia os pedaços para outros dois computadores, que repetem o processo. Mais tarde os dois pedaços da resposta são devolvidos ao primeiro computador, que reconstrói o resultado final.

A parte que “planeja” a divisão já está pronta, este trabalho irá focar em como saber se ela é uma divisão equilibrada, ou seja: se um computador  $A$  divide a tarefa entre os computadores  $B$  e  $C$  e ela tem exatamente o mesmo tamanho, então o trabalho de  $A$  é equilibrado. (Como não é sempre que dá para dividir exatamente ao meio, existem máquinas com trabalho desequilibrado.)

Os testes de casos consistem em uma longa lista com o formato apresentado na figura 1, que sempre descreve o nome de um computador e como ele divide o trabalho para outros dois computadores. Por exemplo, o computador  $X2$  dividirá sua tarefa entre  $X5$  e  $X6$ . Já  $X3$  divide sua tarefa entre outros dois computadores (sem nome), mas que resolvem fazer as tarefas de tamanho 6 e 6 sem dividi-las com outros computadores. Neste exemplo, os computadores  $X4$ ,  $X5$  e  $X0$  estão em equilíbrio.

Para melhor compreensão dos computadores em equilíbrio se montou o esquema da figura 2, onde é possível ver quanto de trabalho cada um dos computadores recebeu mesmo quando divide a tarefa.

Com interesse em aprender mais sobre algoritmos e estrutura de dados, além de trabalhar o raciocínio de otimização, este estudo pretende iniciar com um programa para a resolução da problemática apresentada e buscar maneiras de melhorá-lo.

X0	X1	X2
X1	X3	X4
X2	X5	X6
X3	6	7
X4	8	8
X5	7	7
X6	7	8

Figura 1. Formato de caso de teste.

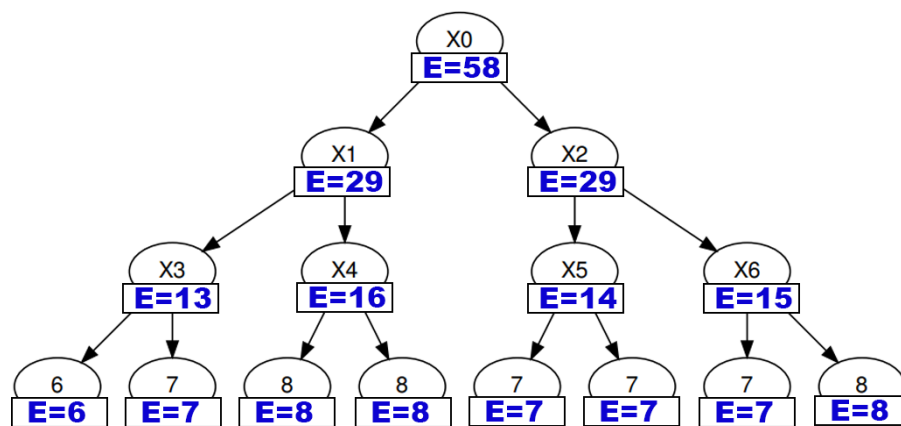


Figura 2. Árvore binária de máquinas.

## 2. Desenvolvimento

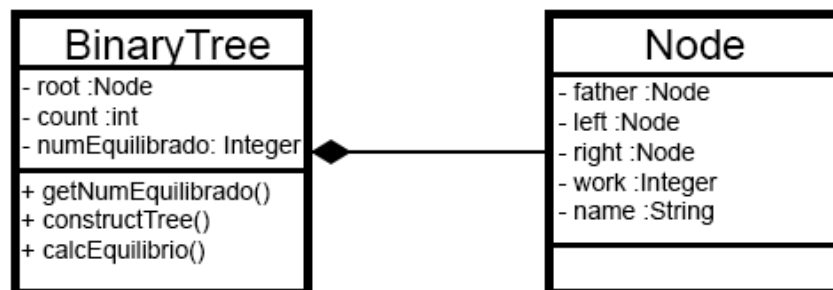
Para resolver essa problemática, inicialmente decidiu-se fazer a construção das classes conforme mostra o diagrama da figura 3. Adaptou-se o modelo padrão de árvore binária adicionando métodos e atributos para calcular o equilíbrio, sendo assim, adicionado à árvore uma variável para armazenar o resultado, um método para calcular esse equilíbrio e outro para retornar o valor. Enquanto na classe Node resolveu-se adicionar um atributo “work” que armazena o trabalho realizado pela máquina e “name” que armazena o nome da máquina como por exemplo “X0”.

O método “constructTree” constrói a árvore recebendo uma String que contém o caminho para o arquivo que contém as informações de como o trabalho foi dividido. Após a leitura do arquivo o método segue como o programa abaixo:

```

vetor temp = divide a linha por " "
addRaiz(temp[0])
addFilhoEsquerda(temp[1], null, raiz.nome)

```



**Figura 3. Diagrama de classe para solução com árvore binária.**

```
addFilhoDireita(temp[2], null, raiz.nome)
```

```
enquanto true
```

```
    le a proxima linha
```

```
    se a linha estiver vazia quebra o loop
```

```
temp = divide a linha por " "
```

```
se temp[1] começar com X
```

```
    addFilhoEsq(temp[1], null, temp[0])
```

```
    addFilhoDireita(temp[2], null, temp[0])
```

```
se nao
```

```
    addFilhoEsq(null, temp[1], temp[0])
```

```
    addFilhoDireita(null, temp[2], temp[0])
```

Adicionar a raiz ficou fora do “looping” porque adicionar o primeiro elemento da raiz é diferente em Java, linguagem escolhida para a implementação. E os métodos utilizados para adicionar o filho recebem em ordem: o nome da ser adicionada, o elemento e o nome da máquina pai, por isso que se adiciona a primeira posição de temporário para o filho da esquerda e na segunda posição para o filho da direita. Dentro do “looping” verifica-se primeiro se contém algo na linha e depois um dos filhos começa com X, porque isso significa que ainda não sabemos o quanto essa máquina trabalha, ou seja, temos apenas o nome dela.

Para verificar se a árvore estava sendo criada corretamente foi adicionado um método que percorre toda a árvore adicionando as informações de cada nodo em uma lista, para tornar possível a visualização da árvore.

Para saber quantos nodos são equilibrados decidiu-se fazer um método recursivo que passa por toda a árvore e calcula o trabalho do nodo. Como é mostrado no programa abaixo:

```
ArrayList<String> calculaEquilibrio()
```

```
    inicializa ArrayList<String> resultado
```

```
    calculaEquilibrioAux(raiz, resultado, raiz.trab)
```

```

numDeEquilibrados = resultado.tamanho()
return resultado

Integer calculaEquoAux(Node n,int res , Integer work)
    se n for diferente de null
        se o filho da esquerda for null
            return n.trab

        se o n.trabalho for null
            n.filhoEsq.trab = calcEquAux(n.filhoEsq , res , n.trab)
            n.filhoDir.trab = calcEquAux(n.filhoDir , res , n.trab)

        n.trab = n.filhoEsq.trabalho + n.filhoDir.trab
        se o trabalho dos filhos for igual
            res.add(n.nome + ';' + n.trab)
        return n.trab

```

O armazenamento em lista foi escolhido apenas para poder verificar quais máquinas estavam em equilíbrio e poder validar se os resultados estavam corretos. Além de alguns outros testes feitos, como comparar o número de nodos criados com a quantidade de linhas no arquivo.

Este programa faz algumas verificações para ver se deve avançar e se avançar irá repetir o método para os filhos. Fazendo com que possamos saber o trabalho de todos os nodos da árvore assim como se estão equilibrados.

Após concluído, pensando na ideia de que a quantidade de linhas do arquivo era igual ao número de nodos em que a máquina tem nome “X\*”, percebeu-se uma forma mais eficiente de se calcular o equilíbrio das árvores sem utilizar a árvore binária, dando início a solução final.

## 2.1. Solução final

Na solução utilizou-se a construção das classes conforme mostra o diagrama da figura 4. Utilizando um Array para armazenar as informações de cada máquina, foi criada a classe Node para armazenar as informações de cada máquina.

O programa vê a quantidade de linhas do arquivo e cria o array com esse tamanho, vai lendo o arquivo e inserindo as máquinas na lista considerando como posição o valor que vem depois do “X”. Na hora de se inserir a máquina já se verifica em quais posições vão ficar os filhos e se armazena em uma variável para ser possível encontrá-los depois e caso não tenha filhos que começam em “X”, ou seja, que passam o trabalho adiante já é calculado o trabalho da máquina e verificado se é uma máquina equilibrada e para validar isso colocamos os nomes dos filhos como “-1”.

Com isso é possível afirmar que o cálculo de quantas máquinas são equilibradas começa na construção da lista, verificando os nodos das pontas. Então após ler todas as linhas é chamado o método privado que calcula o equilíbrio, dando como máquina inicial a que se encontra na posição 0 da lista. O programa a seguir exemplifica o método usado para calcular o equilíbrio:

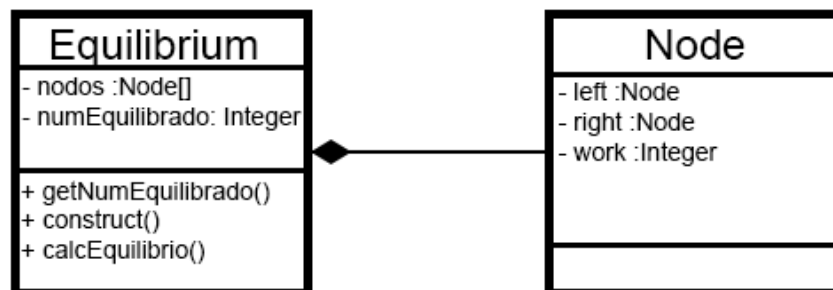


Figura 4. Diagrama de classe para solução com array.

```

int calcEquilibrio(int pos)
    se o nodos[pos].esq for -1
        retorna nodos[pos].trab

    se o nodo[pos].trab for null
        int trabEsq = calcEquilibrio(nodos[pos].esq)
        int trabDir = calcEquilibrio(nodos[pos].dir)
        nodos[pos].trab = trabEsq + trabDir

        se trabEsq for igual ao trabDir
            this.numEquilibrado++

    retorna nodos[pos].trab
  
```

O método começa na posição 0 da lista pois a máquina “X0” tem se apresentado sempre como a máquina na raiz da problemática. Se uma das máquinas que recebe a divisão de trabalho (filhos) tiver como posição -1 significa que não está na lista, portanto seu trabalho já foi calculado na construção da lista. Se o nodo tiver trabalho null então ainda é preciso fazer o cálculo, fazendo com que o campo de trabalho seja preenchido usando recursão.

### 3. Resultados e Conclusão

A tabela abaixo mostra os resultados encontrados para os casos de teste.

Pode-se perceber uma maior otimização na versão final devido ao fato de que, na primeira versão para inserir um nodo na árvore era preciso procurar o nodo pai fazendo com que o método de criação usasse  $O(n)$  para cada nodo adicionado, totalizando para a construção total  $O(n^2)$ . Enquanto a versão final durante todo o método usava apenas  $\Theta(n)$ . E o método que calcula o equilíbrio não teve grande diferença de otimização, ambos são  $O(n)$ . E até mesmo na questão de memória a versão final utiliza uma quantidade menor de variáveis.

Com esse trabalho foi possível perceber que é necessário analisar as possibilidades

<b>Caso de Teste</b>	<b>Máquinas em Equilíbrio</b>
5	4
6	10
7	21
8	38
9	51
10	80
11	95
12	129
13	187

de uma implementação diferente da intuitiva porque isso auxilia no processo de encontrar uma opção mais otimizada.