



Universidade Federal de Mato Grosso do Sul
Campus de Três Lagoas
Bacharelado em Sistemas de Informação
Estruturas de Dados e Programação I

Dicionário em estrutura AVL

Árvores AVL

Julio Cezar Lossavaro

2018.0743.029-4

Três Lagoas

2021



Sumário

1.	Estrutura das Classes	3
1.1.	Classe Node.....	3
1.1.1.	Variáveis (Atributos).....	3
1.1.2.	Métodos da Classe.....	4
1.2.	Classe Tree	4
1.1.3.	Variáveis (Atributos).....	4
1.1.4.	Métodos da Classe.....	4
1.3.	Classe BinaryTreeWords.....	7
1.3.1.	Variáveis.....	7
1.3.2.	Funções da Classe	7
2.	Funcionalidades.....	8
2.1.	Desenvolvimento.....	8
2.1.1.	Reestruturação	8
2.1.2.	Novos Métodos AVL	9

1. Estrutura das Classes

Nesta sessão será explicado a estrutura de cada classe definindo variáveis e métodos.

1.1. Classe Node

Classe que será modelo de todos os nós da árvore, tendo como principais atributos referências aos seus filhos e seu valor, que será utilizado como parâmetro para percorrer a estrutura(arvore).

1.1.1. Variáveis (Atributos)

```
char language;  
int size;  
String value;  
ArrayList<String> synonyms;  
Node left;  
Node right;  
  
public Node(String value, char language) {  
    this.language = language;  
    this.value = value;  
    this.synonyms = new ArrayList<String>();  
}
```

- **Node left, right:** referências aos filhos do nó em si.
- **int size:** atributo utilizado para definir o fator de balanceamento do nó.
- **ArrayList synonyms:** atributo que armazena os sinônimos da palavra.
- **Char language:** atributo responsável por armazenar a linguagem da palavra, podendo assumir “e” (en) ou “p” (pt).
OBS: Esse atributo é usado para diferenciar as palavras, já que podemos ter palavras cognatas, como “banana” (inglês) e “banana” (português).
- **String value:** atributo responsável por armazenar a palavra que será associada ao nó.



1.1.2. Métodos da Classe

- **getLeftSize():**
Retorna -1 caso o nó à esquerda não exista e caso exista, retorna seu tamanho.
- **getRightSize():**
Retorna -1 caso o nó à direita não exista e caso exista, retorna seu tamanho.
- **updateSize():**
Este método atualiza o fator de balanceamento do nó, chamando os métodos `getLeftSize()` e `getRightSize()` e em seguida verificando o maior valor entre ambos e somando um.

1.2. Classe Tree

Esta classe é o modelo da árvore, ela possui um nó que armazena a raiz e métodos responsáveis por retornar, inserir, remover, balancear e atualizar valores.

1.1.3. Variáveis (Atributos)

Node root: Atributo que recebe um nó referenciado a raiz da árvore;

1.1.4. Métodos da Classe

- **recursiveAdd(Node, String, char, String):**
Método que percorre recursivamente a estrutura com base no valor lexicográfico da palavra, além de verificar se a palavra já existe na estrutura, ao encontrar um valor nulo cria-se um novo nó e adiciona um sinônimo a sua lista, caso necessário, e por fim chama a função `updateBalance(node)`.
Caso 1 (Nó nulo): Adiciona um novo nó e encadeia como sinônimo a próxima palavra.

Caso 3 (Nó encontrado): verifica se a próxima palavra está na lista de sinônimos do nó encontrado e em seguida a encadeia na lista.

Caso 4 (Palavra cognata): Em caso de palavra cognata, ou seja, que sejam iguais, porém com linguagem diferente, a recursão é chamada a direita.

- **getLowestNode(Node):**

Método que percorre toda a esquerda de um dado nó, com a finalidade de retornar o nó com menor valor lexicográfico.

- **getHighestNode(Node):**

Método que percorre toda a direita de um dado nó, com a finalidade de retornar o nó com maior valor lexicográfico.

- **updateBalance(Node):**

Atualiza a altura de um dado nó e em seguida calcula seu fator de balanceamento com base na altura de suas sub árvores, realizando o balanceamento do nó, caso necessário.

Caso 1 (Fator Negativo): Quando o fator é negativo significa que a sub árvore a direita é maior, já que o calculo é feito a partir da subtração da sub árvore à direita(negativa) pela esquerda.

Caso 1.1: Já que o balanceamento será feito a direita, tomamos como base a altura das sub árvores do nó a direita, onde caso a árvore a direita seja maior, realizamos uma rotação simples, e dupla, caso esquerda.

Caso 2 (Fator Positivo): Realiza o inverso do Caso 1, ou seja, todas as rotações a direita são feitas a esquerda.

- **rotateRight(Node):**

Este método realiza uma rotação a direita em um nó. Executa as devidas trocas de posicionamento entre os nós referenciados, atualiza suas alturas e por fim devolve o nó balanceado.

- **rotateLeft(Node):**

Este método realiza uma rotação a esquerda em um nó. Executa as devidas trocas de posicionamento entre os nós referenciados, atualiza suas alturas e por fim devolve o nó balanceado.

- **findWord(String):**

Realiza uma busca recursiva na estrutura utilizando uma palavra como parâmetro e por fim retorna o seu nó equivalente.

- **Find(String):**

Chama a função findWord, se a palavra for encontrada, lista seus sinônimos, caso não, retorna "hein?".

- **inOrder(Node, char):**

Função que percorre a árvore em ordem de maneira recursiva, percorrendo primeiro a esquerda e em seguida a direita.

Caso 1: Caso o char (linguagem) informado seja igual ao do nó retorna o valor do nó junto aos seus sinônimos.

- **inOrderBetwen(Node, char, String[]):**

Método que percorre a árvore em ordem de maneira recursiva, percorrendo primeiro a esquerda e em seguida a direita.

Caso 1: Caso o char (linguagem) informado seja igual ao do nó e esteja entre os valores de char[0] e char[1], retorna o valor do nó junto aos seus sinônimos.

- **Remove(Node, String):**

Método que percorre a estrutura em busca de um nó, chamando removeRoot(), caso encontrado, e ao final da recursão atualizando o balanceamento.

- **RemoveRoot(Node):**

Método responsável por remover um dado nó na estrutura, realizando a troca do mesmo por um sucessor, caso necessário.

Caso 1 (Altura 0): Neste caso o nó é uma folha, então retorna-se um valor nulo.

Caso 2 (Sub árvore à esquerda maior): Neste caso chamamos o método `getHighestNode()`, passando como parâmetro seu filho a esquerda, para retornar o nó com menor valor lexicográfico a direita, usamos esse nó como substituto, e por ultimo chamamos o método `remove()` recursivamente, dessa vez passando a esquerda do nó a ser removido como raiz junto ao valor do seu sucessor.

Caso 2 (Sub árvore à direita maior): é realizado o processo inverso do caso 3, porém ao invés de chamarmos `getHighestNode()`, chamamos `getLowestNode()`, passando seu filho a direita como parâmetro, já que o valor substituto será o menor valor a direita.

- **RemoveSyns(String, String):**
Este método recebe duas palavras e as remove dos sinônimos de ambas as palavras correspondentes, caso não haja mais sinônimos em sua lista, chama-se o método `remove()` para o seu nó correspondente.

1.3. Classe BinaryTreeWords

Essa classe contém o método principal do programa, ele é responsável por receber as informações passadas pelo sistema e armazenar em um array de strings, e em seguida os passando por parâmetro para os métodos da árvore.

1.3.1. Variáveis

1.3.2. Funções da Classe

main(): método principal que cria um objeto Scanner para ler o fluxo de entrada passado pelo usuário e um objeto Tree, que será a árvore propriamente dita. Um laço é criado com a finalidade de ler todos os parâmetros passados e invocar os métodos da classe Tree ou finalizar o programa.



2. Funcionalidades

2.1. Desenvolvimento

Durante o desenvolvimento não houve grandes dificuldades, uma vez que ao entender o conceito de árvores AVL e de como elas funcionam, realizar as operações de balanceamento se tornou pura lógica de programação.

2.1.1. Reestruturação

O código foi praticamente refeito, na implementação anterior verificava-se o estado de um dado nó, olhando seus filhos e parentes, já esta implementação realiza essas verificações com base no fator de balanceamento atribuído a cada nó. Portanto muitos métodos se tornaram desnecessários, já que agora conseguimos verificar a altura de cada nó, além calcular o fator de balanceamento.

- **recursiveAdd():** Neste método, antes de qualquer inserção era realizada uma busca para verificar se o nó já se encontrava na estrutura, porém uma solução mais simples foi implementar dentro do próprio método uma comparação de elementos, já que o mesmo possui uma busca implícita.
- **remove() e removeRoot():** O método remove(), agora é responsável apenas por percorrer a estrutura, chamando removeRoot() ao encontrar a palavra a ser removida e por fim invocando o método updateBalance(). Essa alteração facilitou e muito o processo de remoção e balanceamento, além do atributo altura que pode ser utilizado para verificar se o nó é uma folha ou qual o maior lado da árvore com base nas sub árvores de um nó.
- **inOrder() e inOrderBetwen:** Este método seria o equivalente ao “lista”, anteriormente ao imprimir os sinônimos de uma palavra, criava uma String para armazenar todos os sinônimos e por fim os imprimia, agora imprime os separadamente.

- **findWord():** Anteriormente este método chamava uma função de busca chamada search(), a qual foi retirada do código, tal função buscava uma palavra com base no seu idioma, portanto para encontrar uma palavra em específico chamávamos ela duas vezes, primeiro passando 'e'(en) e 'p'(pt) como parâmetro, o que obviamente não era uma boa solução para o problema, além de não ser necessário, portanto agora ela simplesmente busca uma palavra e retorna um nó, caso encontrada.

2.1.2. Novos Métodos AVL

Os métodos para balanceamento foram relativamente simples de se implementar, uma vez que ao entender que ao especializar o método de balanceamento e chamarmos o mesmo no final das recursões que alteram a árvore, podemos facilmente verificar e aplicar as devidas rotações em um nó com base na altura de suas sub árvores a medida que retornamos da recursão.

- **rotateLeft() e rotateRight():** Ambos os métodos foram implementados com a finalidade de serem chamados sozinhos ou em conjunto no caso de uma rotação dupla.
- **updateBalance():** Método responsável por verificar a altura das sub árvores de um dado nó e retornar o mesmo balanceado, caso necessário.