AVL - Adelson Velsky e Landis

Gabriel Simon[1], Matheus Telles Batista[2]

,2[1]Departamento de Informática Universidade Federal do Paraná – UFPR Curitiba, Brasil

Abstract
Relatório contendo a sintetização da implementação do árvore AVL.

Keywords
balanceamento, rotação, inserção, remoção

1. Introdução

A AVL é uma árvore binária balanceada, que por definição, sempre tem sua altura das sub-árvores com uma diferença de maius ou menos 1. Favorece a busca, a inserção e remoção da AVL, e por consequência a sua complexidade O(log n).

A estrutura da árvore AVL utilizou-se atributos para identificação do nó, como o valor atribuido, a altura naqual ele está na BST(Binary Search Tree) em relação com a raiz da árvore, e ponteiros para os filhos da direita e da esquerda, como podemos ver no codigo a seguir:

```
struct node {
    int value;
    int height;
    struct node* lef;
    struct node* rig;
}; typedef struct node node_t;
```

2. Implementação

2.1. Inserção

As funções de inserção recebem uma raiz e um valor. Sendo possível ser **criado** a árvore/nodo, ou ser **inserido** o nodo.

```
node_t* createNode(int value);
```

Apenas cria o nodo, declarando os filhos como NULL. O uso dessa função ocorre apenas quando não há árvore e é necessário **criar** o nodo.

```
node_t* insertNode(node_t* root, int value);
```

Insere o valor dentro da árvore. Caso a raíz da árvore seja nula, é chamado a função **createNode()**, caso contrário incia-se um processo recursivo no qual é **inserido** o nodo, com este sendo uma **folha** da árvore.

```
int heightNode(node_t *node);
```

Calcula a altura da do nodo atual com base na altura de seus filhos + 1.

```
int larger(int a, int b);
```

CI 057 - Algoritmos e Estrutura de Dados III.
gsbr21@inf.ufpr.br (G. Simon); mtb21@inf.ufpr.br (M. T. Batista)
https://github.com/gab-simon (G. Simon); https://github.com/TilTelles (M. T. Batista)

Compara qual dos valores são maiores entre a e b: usado nesse trabalho para verificar a altura entre dois nodos filhos.

```
node_t* balance(node_t *root);
```

Realiza o **balanceamento** com base na altura de seus filhos. Dependendo da situação realiza diferentes funções de rotação.

2.2. Rotações

As rotações são parte fundamentais do balanceamento, e feito as rotações **zig, zag, zig-zig, zig-zag e zag-zig** pelas seguintes funções:

```
node_t* leftRotation(node_t* node);
```

Realiza-se a rotação a **esquerda** do nó passado como parâmetro.

```
node_t* rightRotation(node_t* node);
```

Realiza-se a rotação a **direita** do nó passado como parâmetro.

```
node_t* leftRightRotation(node_t *node);
```

Realiza-se a rotação a **esquerda** e depois a **direita** do nó passado como parâmetro.

```
node_t* rightLeftRotation(node_t* node);
```

Realiza-se a rotação a **direita** e depois a **esquerda** do nó passado como parâmetro.

2.3. Remoção

Ao chamar a remoção de um valor, passando 'r' seguido do valor na entrada do programa, procura a chava desejada, quando encontra remove de acordo com o caso que a chave é, por exemplo, se ela é a raiz da árvore, se ela é folha ou se está no 'meio'.

```
node_t* removeNode(node_t* root, int key);
```

Remove o nodo que possui o valor inserido no segundo parâmetro.

```
node_t *maxNo(node_t *no);
```

É retornado o maior nodo dentro da sub-árvore.

2.4. stdin e Impressão

A main do programa recebe entrada padrão (stdin), cada linha é quebrada em duas partes, onde a primeira indica se é remoção ou inserção e a segunda é o valor, por fim, imprime a árvore em ordem.

```
while(fgets(recive, 100, stdin)){
    char *token = strtok(recive, " ");
    if(strcmp(token, "i") == 0){
        token = strtok(NULL, " ");
        value = atoi(token);
        root = insertNode(root, value);
    }
    // ...else if para 'r' e um else para quando eh nenhum caso.
}
```