Documentação do Código

Guilherme Altmeyer Soares
Igor Correa Domingues de Almeida

- Árvore.h

Primeiramente definimos a classe node (nó), como elemento privado dessa classe declaramos dois ponteiros de nó apontado para a direita e esquerda, pois se trata de uma árvore binária (AVL) onde temos no máximo referência a 2 filhos. Cada nó tem um inteiro (chave) associado a ele e também uma altura (height). As funções públicas que merecem destaque pois auxiliam no processo de balanceamento da árvore AVL são:

```
int fatorBalanceamento(node *no);
node* rotDir(node *no);
node* rotEsq(node *no);
node* rotDuplaEsq(node *no);
node* rotDuplaDir(node *no);
```

Também temos uma classe Árvore que seta sua raiz == NULL quando é criada, temos a função `node* balancear(node *no);` como diferencial de uma árvore binária normal.

- Funções

```
node* arvore::inserir(node *no, int chave) {
    if (no == NULL) {
        return new node(chave);
    } else {
        if (chave < no->getChave()) {
            no->setEsq(inserir(no->getEsq(), chave));
        } else if (chave > no->getChave()) {
            no->setDir(inserir(no->getDir(), chave));
        } else
            cout << "Elemento já presente na arvore \n";
    }
    no->setHeight(no, max(no->getHeight(no->getEsq()), no->getHeight(no->getDir())) + 1);
    no = balancear(no); //balancear a arvore a partir de cada no
    return no;
}
```

Toda vez que é inserido um novo nó, a altura do nó atual é recalculada. Para isso pegamos a altura do nó do filho a esquerda e do da direita, colocamos no max para pegar o maior valor entre as duas comparações e somamos 1 para incluir o nó atual na contagem da altura. após calcular a altura chamamos a função de balanceamento para ver se é necessário fazer alguma modificação. O processo de remoção é similar ao de uma árvore binária, também apenas acrescentando ao final a atualização da altura dos nós e depois checando o balanceamento da árvore

```
node* arvore::balancear(node *no) {
    short fb = no->fatorBalanceamento(no);
    if (fb < -1 && no->fatorBalanceamento(no->getDir()) <= 0) {
        no = no->rotEsq(no);
    } else if (fb > 1 && no->fatorBalanceamento(no->getEsq()) >= 0) {
        no = no->rotDir(no);
    } else if (fb > 1 && no->fatorBalanceamento(no->getEsq()) < 0) {
        no = no->rotDuplaDir(no);
    } else if (fb < -1 && no->fatorBalanceamento(no->getDir()) > 0) {
        no = no->rotDuplaEsq(no);
    }
    return no;
}
```

A função de balanceamento verifica os 4 casos possíveis quando se vai balancear uma árvore AVL. Primeiro verificamos se o fator de balanceamento está pendendo para esquerda ou direita. Se for < 1 comparamos com o fator de balanceamento da árvore à direita para definir se a rotação será simples ou dupla para esquerda. Analogamente quando o fb > 1 verificamos o fator de balanceamento do nó à esquerda para definir entre uma rotação simples ou uma dupla à esquerda. O fator de balanceamento é calculado na diferença da altura do nó à esquerda com o nó à direita.

```
// Rotação simples à direita
node* node::rotDir(node *no) {
    node *aux = no->esq;
    no->esq = aux->dir;
    aux->dir = no;
    no->height = max(getHeight(no->esq), getHeight(no->dir)) + 1;
    aux->height = max(getHeight(aux->esq), getHeight(aux->dir)) + 1;
    no = aux;
    return aux;
}
```

No início da rotação criamos um nó auxiliar que recebe o nó à esquerda (se fosse rotação simples à esquerda estaríamos recebendo o nó à direita) . Após isso, o ponteiro à esquerda do nó pai aponta para o ponteiro à direita do auxiliar, e o ponteiro à direita do auxiliar aponta para o pai. Atualizamos as alturas de ambos os nós e finalizamos sentando o pai como o auxiliar. a rotação simples à esquerda ocorre analogamente invertendo apenas esquerda com direita.

```
// Percurso em ordem na árvore
void arvore::emOrdem(node* no, int level) {
    if (no != NULL) {
        emOrdem(no->getEsq(), level+1);
        cout << no->getChave() << "," << level << "\n";
        emOrdem(no->getDir(), level+1);
    }
}
```

A saída do algoritmo ou seja a árvore será printada usando o percurso em ordem o qual segue a sequência **filho esquerdo → nó atual → filho direito**, garantindo que os elementos sejam processados em ordem crescente em árvores binárias de busca.