

ESTRUTURAS DE DADOS

Árvores AVL

Roteiro

- **Aplicação da Estrutura**
- **Conceitos Básicos**
- **Estrutura do Nó**
- **Tipo Abstrato de Dados**
- **Detalhes de Implementação**

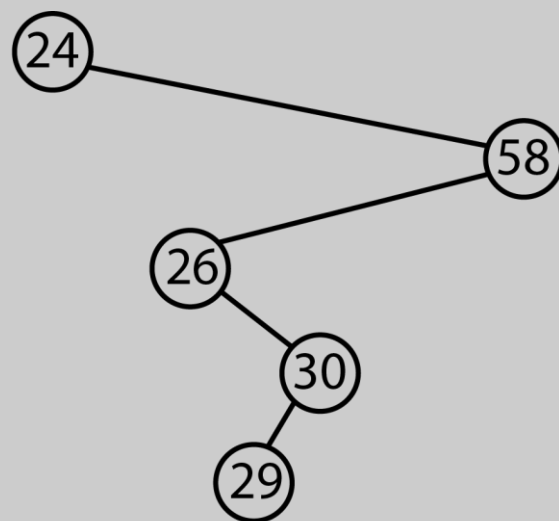
Aplicações da Estrutura

Árvores AVLs propõem uma modificação nos algoritmos de **inserção** e **remoção**, vistos nas aulas anteriores para garantir o balanceamento da árvore.

Em geral, podem ser usadas em qualquer situação em que queremos usar árvores binárias de busca, com a garantia de que as operações serão sempre **eficientes**.

Nos algoritmos de **busca**, **inserção** e **remoção**, no pior caso, o número de comparações é proporcional à **altura da árvore**.

- As buscas são eficientes em árvores balanceadas.
- Em árvores degeneradas, as operações deixam de ser eficientes.



Número de Elementos: 5

Número Máximo de Comparações: 5

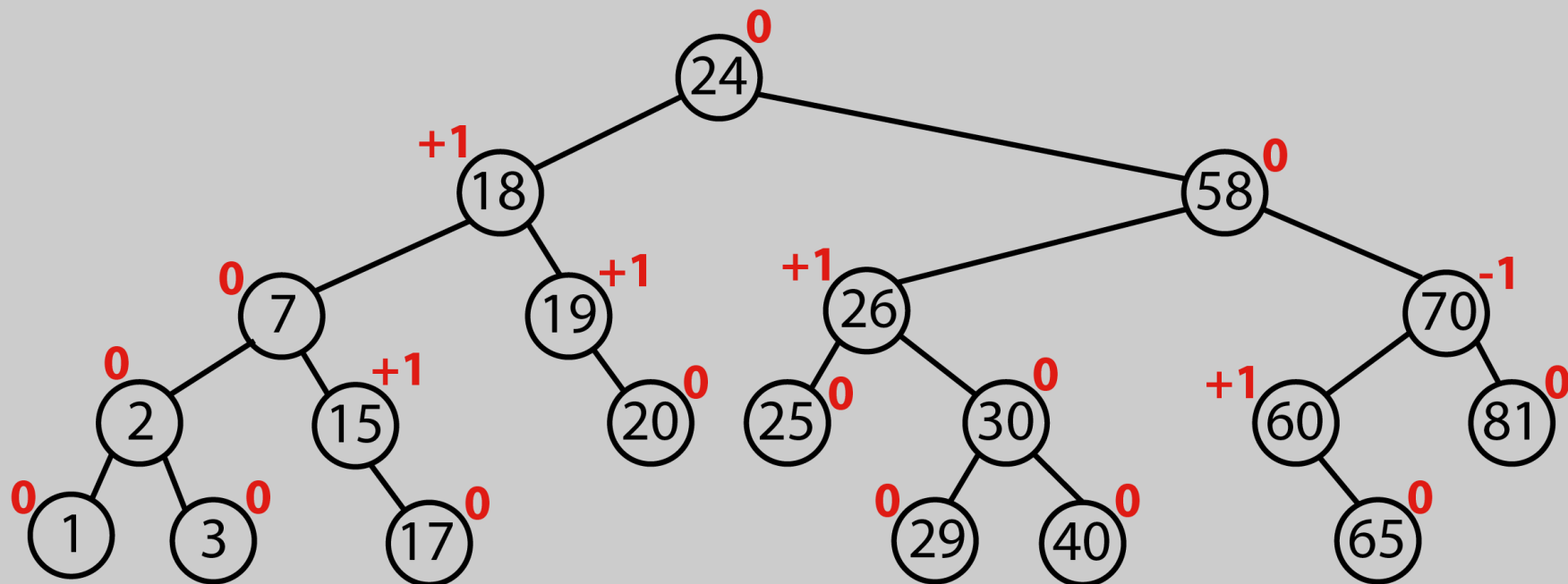
Conceitos Básicos

Fator de balanceamento: é a diferença de altura entre as subárvores da direita e da esquerda.



- Em árvores AVL, deve ficar no intervalo de **-1 a 1**.

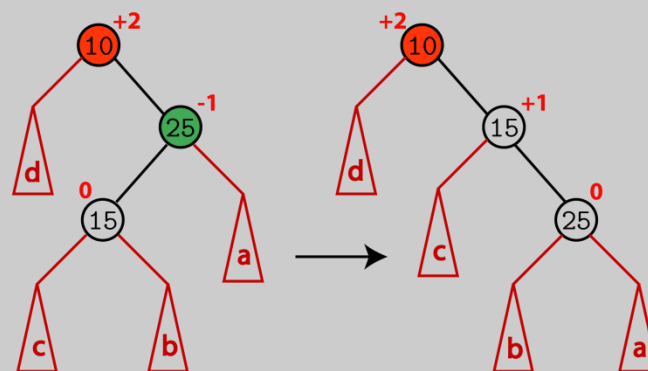
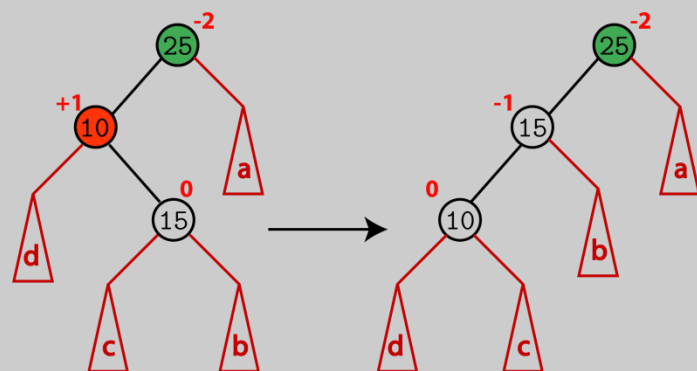
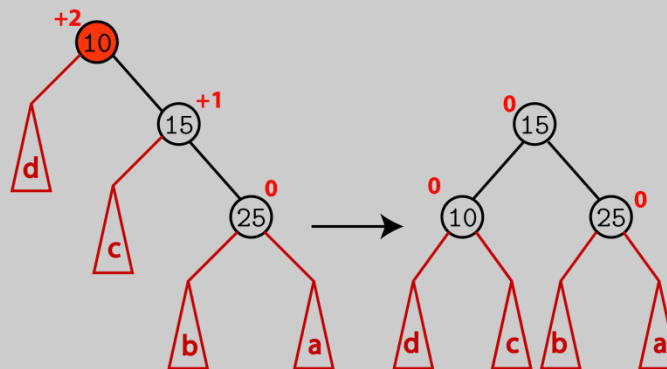
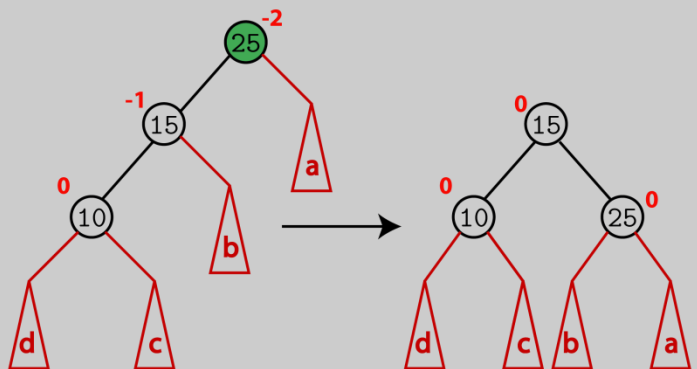
As inserções e remoções seguem os algoritmos vistos anteriormente. Entretanto, se algum nó violar a propriedade do fator de balanceamento após uma inserção ou remoção, uma rotação deve ser feita.

Fator de Balanceamento



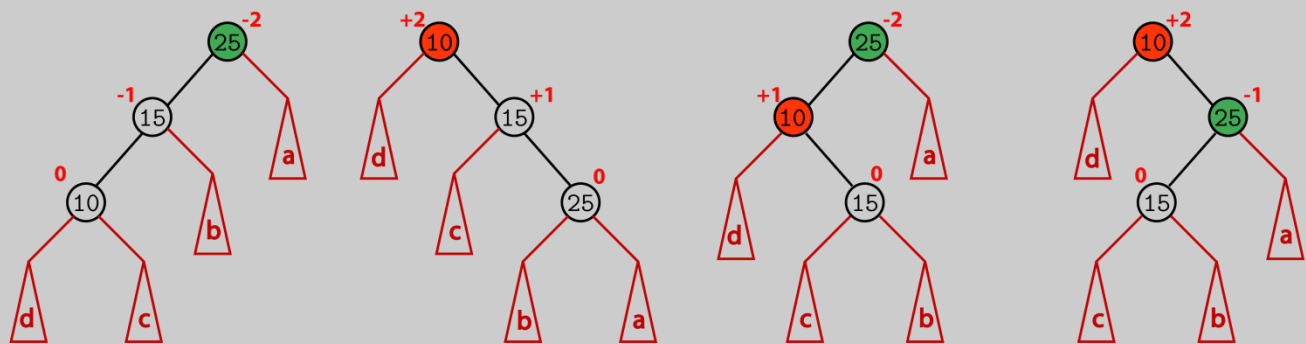
Rotações

-  Rotação para a direita
-  Rotação para a esquerda



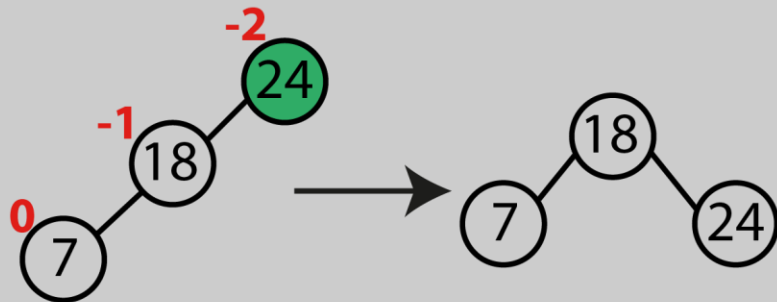
Regras para uso das rotações

Nó Desbalanceado	Filho do Nó Desbalanceado	Tipo de Rotação
+2	+1	Simple à Esquerda
+2	0	Simple à Esquerda
+2	-1	Dupla com Filho para a Direita e pai para Esquerda
-2	+1	Dupla com Filho para a Esquerda e pai para Direita
-2	0	Simple à Direita
-2	-1	Simple à Direita

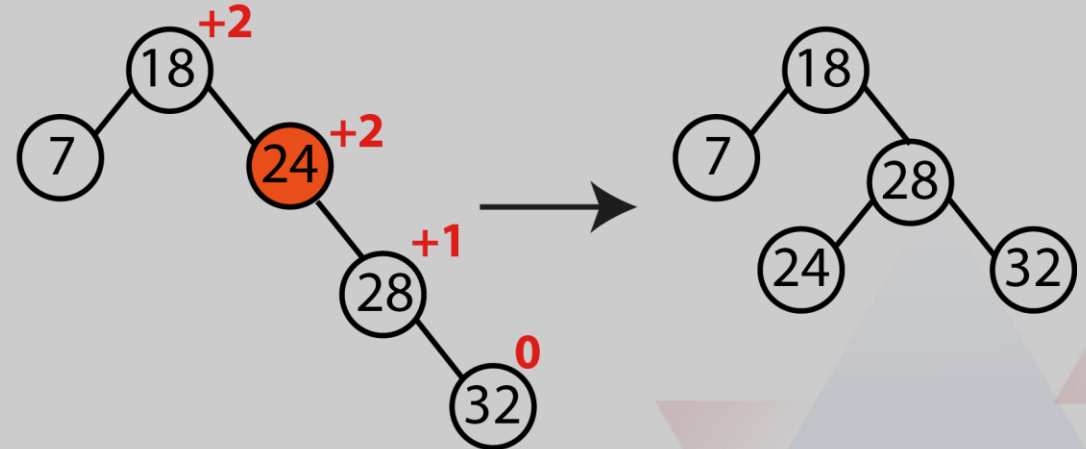


Exemplos

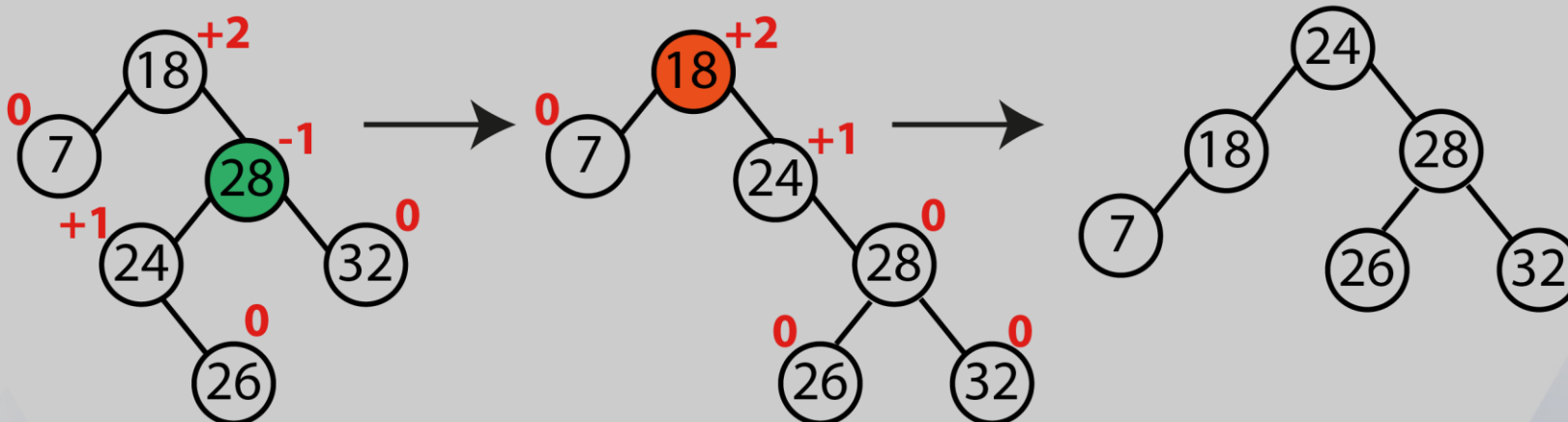
Inserir: 24, 18 e 07



Inserir: 28, 32



Inserir: 26



Estrutura do Nó

Precisaremos armazenar o fator de balanceamento, além das outras informações que já guardávamos.

- Note que não precisamos modificar a classe Aluno.

```
struct NodeType
{
    Aluno aluno;
    int      fatorB;
    NodeType* esquerda;
    NodeType* direita;
};
```

Tipo Abstrato de Dados

```
class AVLSearchTree {
public:
    AVLSearchTree() { root = NULL; }
    ~AVLSearchTree(){ destroyTree(root); }
    bool isEmpty() const;
    bool isFull() const;
    void retrieveAluno(Aluno& item, bool& found) const{
        retrieveAluno(root, item, found);
    }
    void insertAluno(Aluno item){
        bool isTaller;
        insertAluno(root, item, isTaller);
    }
    void deleteAluno(int item){
        bool isShorter;
        deleteAluno(root, item, isShorter);
    }
}
```

**Interface pública é a
mesma da classe
SearchTree de
aulas anteriores**

**Agora, insertAluno e
deleteAluno gerenciam o
crescimento da árvore.**

Apenas os métodos **insertAluno**, **deleteAluno** e **deleteNode** sofrem modificações.

- **isTaller** e **isShorter** servem para saber se a árvore cresceu na inserção ou decresceu na remoção (para gerenciar o **fator de balanceamento**).

```
private:
void destroyTree(NodeType*& tree);
void retrieveAluno(NodeType* tree,
                  Aluno& item,
                  bool& found) const;

void insertAluno(NodeType*& tree, Aluno item, bool& isTaller);
void deleteAluno(NodeType*& tree, int item, bool& isShorter);
void deleteNode(NodeType*& tree, bool& isShorter);
void getSuccessor(NodeType* tree, Aluno& data);
void printTree(NodeType *tree) const;
void printPreOrder(NodeType* tree) const;
void printInOrder(NodeType* tree) const;
void printPostOrder(NodeType* tree) const;
```

Novos métodos privados servem para realizar as rotações.

```
void rotateToLeft(NodeType*& tree) const;  
void rotateToRight(NodeType*& tree) const;  
void rotateToLeftAndRight(NodeType*& tree) const;  
void rotateToRightAndLeft(NodeType*& tree) const;  
  
void performRotations(NodeType*& tree) const;  
  
// Nó raiz da árvore.  
NodeType* root;
```

Detalhes de Implementação

Estudaremos a implementação de alguns métodos. Apenas a lógica implementada nos interessa.

Métodos que não mudaram, como **retrieveAluno**, não serão mencionados.

O método **deleteAluno** modifica, mas as ideias não mudam muito em relação ao que foi feito no método **insertAluno**.

Inserções e remoções gerenciam o fator de balanceamento com **isTaller** e **isShorter**.

- Inserções usam **isTaller**.

```
void AVLSearchTree::insertAluno(NodeType*& tree,
                                Aluno aluno,
                                bool& isTaller) {

    if (tree == NULL) {
        tree = new NodeType;
        tree->direita = NULL;
        tree->esquerda = NULL;
        tree->aluno = aluno;
        tree->fatorB = 0; // Acabamos de inserir uma folha
        isTaller = true;
        return;
    }
}
```

- Ao inserir uma folha, o fator de balanceamento é **zero**. Além disso, avisamos o nó pai sobre o aumento de tamanho.

Um nó interno que recebe a notificação de crescimento de uma das subárvores deve ajustar o fator de balanceamento.

- **Note que a atualização do fator de balanceamento depende de qual filha cresceu.**

```
if (aluno.getRa() < tree->aluno.getRa()) {  
    insertAluno(tree->esquerda, aluno, isTaller);  
    if (isTaller)  
        tree->fatorB = tree->fatorB - 1;  
} else {  
    insertAluno(tree->direita, aluno, isTaller);  
    if (isTaller)  
        tree->fatorB = tree->fatorB + 1;  
}
```


Ao final da inserção, invocamos o **performRotation** (será visto mais adiante).

- Esse método verificará se uma rotação precisa ser feita e realiza as operações.

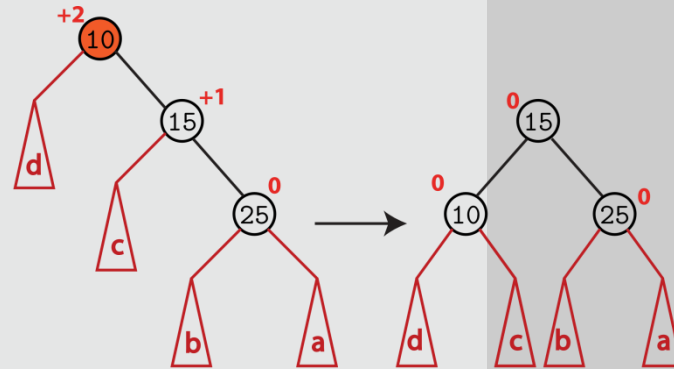
```
// O performRotations vai ajustar o fatorB
performRotations(tree);

// Após a rotação, a árvore não muda de tamanho
if (isTaller && tree->fatorB == 0) {
    isTaller = false;
}
```

As rotações são uma implementação direta dos conceitos:

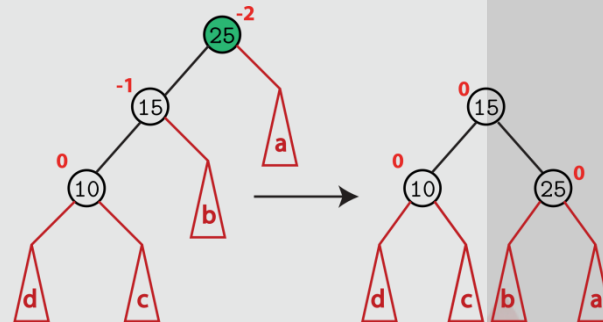
```
void AVLSearchTree::rotateToLeft(  
    NodeType*& tree  
) const{
```

```
    NodeType* p = tree->direita;  
    tree->direita = p->esquerda;  
    p->esquerda = tree;  
    tree = p;  
}
```



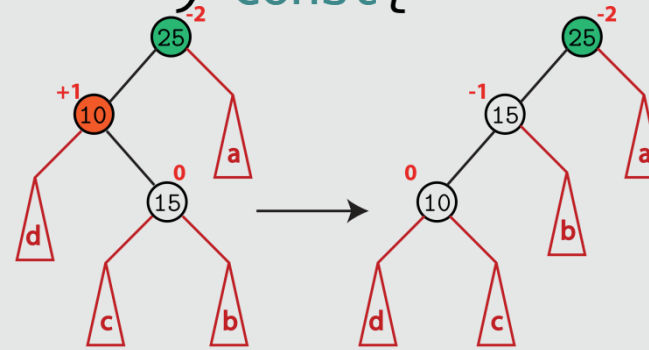
```
void AVLSearchTree::rotateToRight(  
    NodeType*& tree  
) const{
```

```
    NodeType* p = tree->esquerda;  
    tree->esquerda = p->direita;  
    p->direita = tree;  
    tree = p;  
}
```

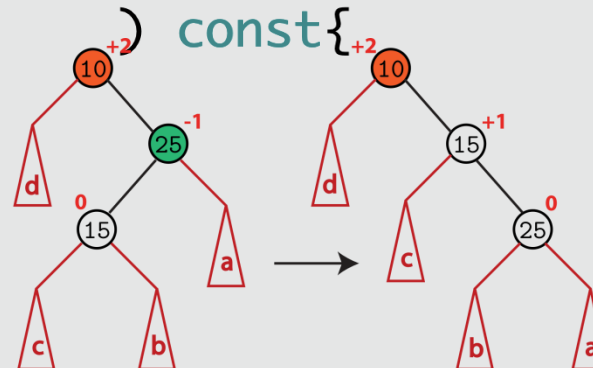


As rotações são uma implementação direta dos conceitos:

```
void AVLSearchTree::rotateToLeftAndRight(  
    NodeType*& tree  
    ) const{  
  
    NodeType* child = tree->esquerda;  
    rotateToLeft(child);  
    tree->esquerda = child;  
    rotateToRight(tree);  
}
```



```
void AVLSearchTree::rotateToRightAndLeft(  
    NodeType*& tree  
    ) const{  
  
    NodeType* child = tree->direita;  
    rotateToRight(child);  
    tree->direita = child;  
    rotateToLeft (tree);  
}
```



performRotation implementa a tabela de casos.

```
void AVLSearchTree::performRotations(NodeType*& tree) const {  
    NodeType* child;  
    NodeType* grandChild; // Usado em rotacao dupla
```

```
// Rotacionar para a direita
```

```
if (tree->fatorB == -2) {  
    child = tree->esquerda;
```

```
    switch (child->fatorB) {
```

```
    case -1 : // Simples para a direita: Caso 1
```

```
        tree->fatorB = 0;  
        child ->fatorB = 0;  
        rotateToRight(tree);  
        break;
```

```
    case 0 : // Simples para a direita: Caso 2 -> Remoções
```

```
        tree->fatorB = -1;  
        child ->fatorB = +1;  
        rotateToRight(tree);  
        break;
```

```
    case 1 : // Rotacao dupla
```

```
        grandChild = child->direita;
```

Nó Desbalanceado	Filho do Nó Desbalanceado	Tipo de Rotação
+2	+1	Simples à Esquerda
+2	0	Simples à Esquerda
+2	-1	Dupla com Filho para a Direita e pai para Esquerda
-2	+1	Dupla com Filho para a Esquerda e pai para Direita
-2	0	Simples à Direita
-2	-1	Simples à Direita

O restante do código segue a mesma linha de raciocínio.

ESTRUTURAS DE DADOS

Árvores AVL