AVL

Características:

- É uma árvore altamente balanceada, isto é, nas inserções e exclusões, procura-se executar uma rotina de balanceamento tal que as alturas das sub-árvores esquerda e sub-árvores direita tenham alturas bem próximas
- Idealmente a árvore deve ser razoavelmente equilibrada e a sua altura será dada (no caso de estar completa) por - h=log2(n+1)
- A árvore AVL tem **complexidade O(log n)** para todas operações e ocupa espaço n, onde n é o número de nós da árvore.

Complexidade da árvore AVL em notação O

	Média	Pior caso
Espaço	$O(n)$	$O(n)$
Busca	$O(\log n)$	$O(\log n)$
Inserção	$O(\log n)$	$O(\log n)$
Deleção	$O(\log n)$	$O(\log n)$

- **Definição:** Uma árvore AVL é uma árvore na qual as alturas das subárvores esquerda e direita de cada nó diferem no máximo por uma unidade.

- ## Como ocorre a Busca

A busca é a mesma utilizada em árvore binária de busca.
A busca pela chave de valor K inicia sempre pelo nó raiz da árvore.
Seja pt_u um ponteiro para o nó u sendo verificado. Caso o pt_u seja nulo então a busca não foi bem sucedida (K não está na árvore ou árvore vazia). Verificar se a chave K igual pt_u->chave (valor chave armazenado no nó u), então a busca foi bem sucedida. Caso contrário, se K < pt_u->chave então a busca segue pela subárvore esquerda; caso contrário, a busca segue pela subárvore direita.

Exemplo de algoritmo de busca em Java.

```java
// O método de procura numa AVL é semelhante ao busca binária de uma árvore binária de busca comum.
public BSTNode<T> search(T element) {
   return search(element, this.root);
}

// Método auxiliar à recursão.
private BSTNode<T> search(T element, BSTNode<T> node) {
   if (element == null || node.isEmpty()) {
      return new BSTNode<T>();
   }

   if (node.isEmpty() || node.getData().equals(element)) {
      return node;
   } else if (node.getData().compareTo(element) > 0) {
      return search(element, node.getLeft());
   } else {
      return search(element, node.getRight());
   }
}
```

Inserção

Para inserir um novo nó de valor K em uma árvore AVL é necessária uma busca por K nesta mesma árvore. Após a busca, o local correto para a inserção do nó K será em uma subárvore vazia de uma folha da árvore. Depois de inserido o nó, a altura do nó pai e de todos os nós acima deve ser atualizada. Em seguida o algoritmo de rotação simples ou dupla deve ser acionado para o primeiro nó pai desregulado.

Os parâmetros p e mudou_h são passados por referência. O ponteiro p aponta para o nó atual. O parâmetro mudou_h é do tipo lógico e informa ao chamador se a subárvore apontada por p mudou sua **altura.**

Como identificar mudança de altura?

Considerar que o nó p é raiz da subárvore T_p e houve inserção em uma de suas subárvores.

Caso a subárvore T_p tenha mudado de altura, decrementar fb (inserção na subárvore esquerda) ou incrementar fb (inserção na subárvore direita).

Caso 1: Ao inserir um nó folha, a subárvore T_p passa de altura 0 para altura 1, então T_p mudou de altura.

Caso 2: fb=0 antes da inserção foi alterado para 1 ou -1, então a subárvore T_p mudou de altura.

Caso 3: fb=1 ou -1 antes da inserção, passou a ter valor 0, então a subárvore T_p não mudou de altura.

Caso 4: O fb passou a ter valor -2 ou 2 após a inserção, então há necessidade de aplicação de alguma operação de rotação. Após a rotação, a subárvore T_p terá a mesma altura anterior à inserção.

Exemplo de algoritmo de inserção em Java

```java
/* Por definição, a árvore AVL é uma árvore binária de busca (BST).
*  Por este motivo utiliza-se aqui a mesma definição (classe) de Nós que uma BST
simples.
*/
public void insert(T element) {
   insertAux(element);
   BSTNode<T> node = search(element); // Pode-se utilizar o mesmo search
exemplificado acima.
   rebalanceUp(node);
}
```

```java
private void insertAux(T element) {
   if (element == null) return;
   insert(element, this.root);
}

private void insert(T element, BSTNode<T> node) {
   if (node.isEmpty()) {
      node.setData(element);
      node.setLeft(new BSTNode<T>());
      node.setRight(new BSTNode<T>());
      node.getLeft().setParent(node);
      node.getRight().setParent(node);
   } else {
      if (node.getData().compareTo(element) < 0) {
         insert(element, node.getRight());
      } else if (node.getData().compareTo(element) > 0) {
         insert(element, node.getLeft());
      }
   }
}
```

Algoritmos de complemento à inserção e/ou algoritmos para identificar desbalanceamento em Java

```java
protected void rebalanceUp(BSTNode<T> node) {
   if (node == null || node.isEmpty()) return;
   rebalance(node);
   if (node.getParent() != null) {
      rebalanceUp(node.getParent());
   }
}

protected int calculateBalance(BSTNode<T> node) {
   if (node == null || node.isEmpty()) return 0;
   return height(node.getRight()) - height(node.getLeft());
}

protected void rebalance(BSTNode<T> node) {
   int balanceOfNode = calculateBalance(node);

   if (balanceOfNode < -1) {
      if (calculateBalance(node.getLeft()) > 0) {
         leftRotation(node.getLeft());
      }
      rightRotation(node);
   } else if (balanceOfNode > 1) {
```

```java
            if (calculateBalance(node.getRight()) < 0) {
                rightRotation(node.getRight());
            }
            leftRotation(node);
        }
    }
```

Rotação para Direita e para Esquerda em Java

```java
protected void leftRotation(BSTNode<T> no) {
        BTNode<T> noDireito = no.getRight();

        no.setRight(noDireito.getLeft());

        noDireito.getLeft().setParent(no);
        noDireito.setLeft(no);
        noDireito.setParent(no.getParent());
        no.setParent(noDireito);

        if (no != this.getRoot()) {
                if (noDireito.getParent().getLeft() == no) {
                        noDireito.getParent().setLeft(noDireito);
                } else {
                        noDireito.getParent().setRight(noDireito);
                }
        } else {
                        this.root = (BSTNode<T>) noDireito;
        }
}

protected void rightRotation(BSTNode<T> no) {
        BTNode<T> noEsquerdo = no.getLeft();

        no.setLeft(noEsquerdo.getRight());

        noEsquerdo.getRight().setParent(no);
        noEsquerdo.setRight(no);
        noEsquerdo.setParent(no.getParent());
        no.setParent(noEsquerdo);

        if (no != this.getRoot()) {
                if (noEsquerdo.getParent().getLeft() == no) {
                        noEsquerdo.getParent().setLeft(noEsquerdo);
                } else {
                        noEsquerdo.getParent().setRight(noEsquerdo);
                }
        } else {
                this.root = (BSTNode<T>) noEsquerdo;
```

```
        }
    }
```

Remoção

O primeiro passo para remover uma chave K consiste em realizar uma busca binária a partir do nó raiz. Caso a busca encerre em uma subárvore vazia, então a chave não está na árvore e a remoção não pode ser realizada. Caso a busca encerre em um nó u o nó que contenha a chave então a remoção poderá ser realizada da seguinte forma:

Caso 1: O nó u é uma folha da árvore, apenas exclui-lo.

Caso 2: O nó u tem apenas uma subárvore, necessariamente composta de um nó folha, basta apontar o nó pai de u para a única subárvore e excluir o nó u.

Caso 3: O nó u tem duas subárvores: localizar o nó v predecessor ou sucessor de K, que sempre será um nó folha ou possuirá apenas uma subárvore; copiar a chave de v para o nó u; excluir o nó v a partir da respectiva subárvore de u.

O último passo consiste em verificar a desregulagem de todos nós a partir do pai do nó excluído até o nó raiz da árvore. Aplicar rotação simples ou dupla em cada nó desregulado.

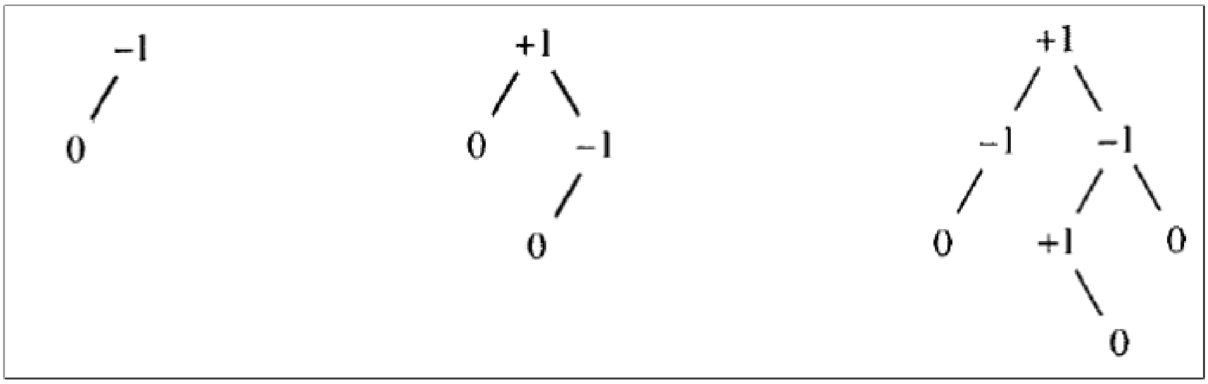
Exemplo de algoritmo de remoção em Java

```java
    public void remover(int valor) {
            removerAVL(this.raiz, valor);
        }
```

```java
private void removerAVL(No atual, int valor) {
	if (atual != null) {

		if (atual.getChave() > valor) {
			removerAVL(atual.getEsquerda(), valor);

		} else if (atual.getChave() < valor) {
			removerAVL(atual.getDireita(), valor);

		} else if (atual.getChave() == valor) {
			removerNoEncontrado(atual);
		}
	}
}

private void removerNoEncontrado(No noARemover) {
	No no;

	if (noARemover.getEsquerda() == null || noARemover.getDireita() ==
null) {

		if (noARemover.getPai() == null) {
			this.raiz = null;
			noARemover = null;
			return;
		}
		no = noARemover;

	} else {
		no = sucessor(noARemover);
		noARemover.setChave(no.getChave());
	}

	No no2;
	if (no.getEsquerda() != null) {
		no2 = no.getEsquerda();
	} else {
		no2 = no.getDireita();
	}

	if (no2 != null) {
		no2.setPai(no.getPai());
	}

	if (no.getPai() == null) {
		this.raiz = no2;
	} else {
		if (no == no.getPai().getEsquerda()) {
			no.getPai().setEsquerda(no2);
```

```
                } else {
                        no.getPai().setDireita(no2);
                }
                verificarBalanceamento(no.getPai());
        }
        no = null;

}
```

- **Balanceamento: hd – he є {0, 1, -1}**



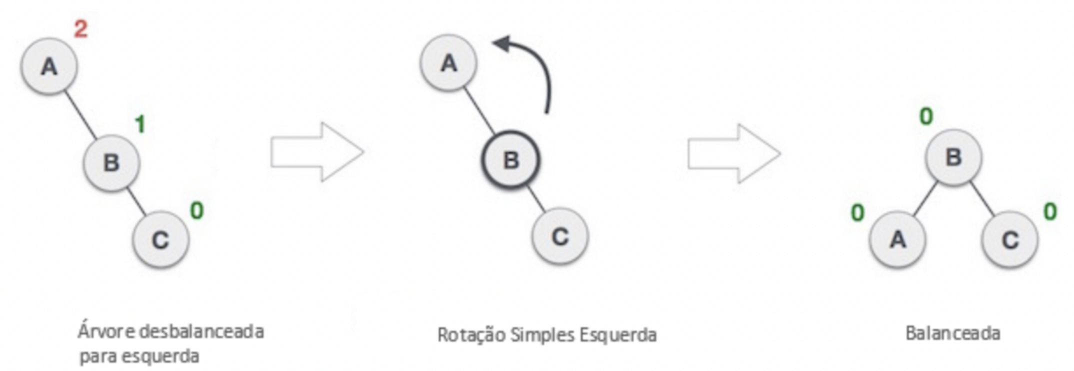
Se o **fator de balanceamento** de qualquer nó ficar menor do que -1 ou maior do que 1 então a árvore tem que ser balanceada.

Rotações em AVL

Nas árvores AVL, após cada operação, como inserção e exclusão, o fator de balanceamento de cada nó precisa ser verificado. Se cada nó satisfizer a condição do fator de balanceamento, a operação pode ser concluída. Caso contrário, a árvore precisa ser rebalanceada utilizando as operações de rotação. Existem quatro rotações e elas são classificadas em dois tipos.

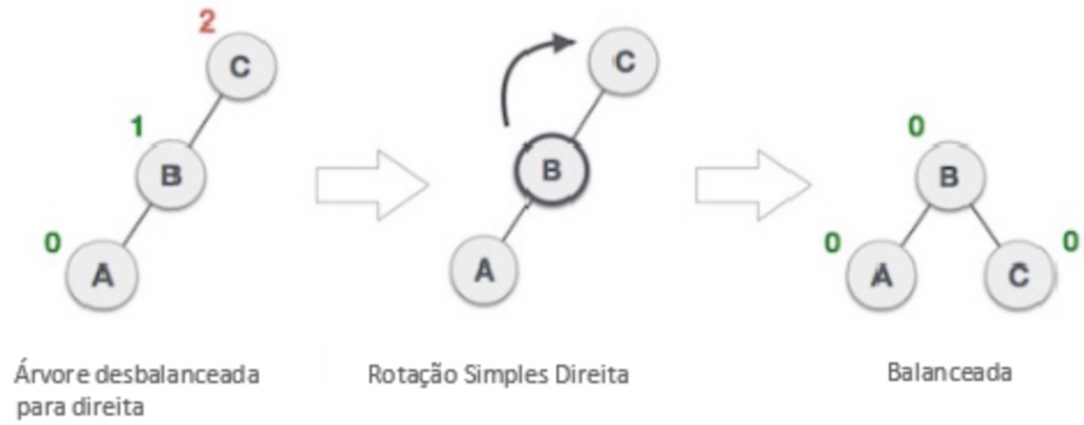
Rotação simples à esquerda (rotação SE - RR)

Na rotação simples à esquerda, cada nó se move uma posição para a direita da posição atual.



Rotação simples à direita (rotação SD - LL)

Na rotação simples à direita, cada nó se move uma posição para a direita da posição atual.



Rotação dupla à direita (rotação DD)

As rotações duplas à direita são uma combinação de uma única rotação para a esquerda seguida de uma rotação para a direita.

Primeiro, cada nó se move uma posição para a esquerda. Depois, se move uma posição para a direita da posição atual.

Rotação dupla à esquerda (rotação DE)

As rotações duplas à esquerda são uma combinação de uma única rotação para a direita seguida de uma rotação para a esquerda. Primeiro, cada nó se move uma posição para a direita. Depois, se move uma posição para a esquerda da posição atual.

OBS: As operações de rotação possuem tempo constante, já que a única coisa que acontece nessas ações é a mudança de alguns ponteiros. Assim como as rotações, recuperar o fator de balanceamento e atualizar a altura também possuem tempo constante. Dessa forma, a complexidade de inserção em uma AVL continua sendo O(h), em que h é a altura da árvore, tal qual em uma árvore binária de busca. Mas, como a AVL é balanceada, a altura da árvore é O(Log n), sendo também a

complexidade de inserção de nó nesse tipo de árvore.