Estruturas de Dados

Árvores AVL

Universidade Estadual Vale do Acaraú – UVA

Paulo Regis Menezes Sousa

paulo_regis@uvanet.br

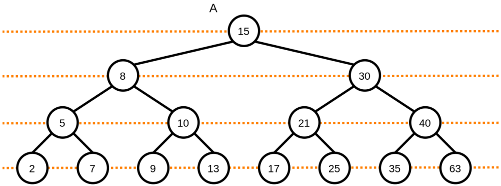
Árvores Balanceadas

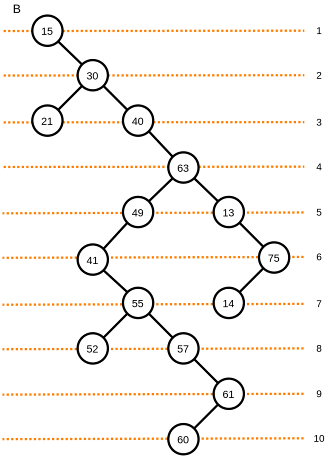
Árvores AVL

Operação de Seleção

Operação de Inserção

- Uma árvore completa possui altura proporcional a $\log n$ (A).
- Quando a árvore perde essa característica, chamamos ela de árvore *degenerada* (B).





- Aplicar um algoritmo que tornasse a árvore novamente completa, teria custo no mínimo proporcional a n, ou seja, $\Omega(n)$.
- **Objetivo**: manter o custo das operações na mesma ordem de grandeza da altura de uma árvore completa, ou seja, $O(\log n)$, onde n é o número de nós da árvore.

Árvore de busca binária balanceada

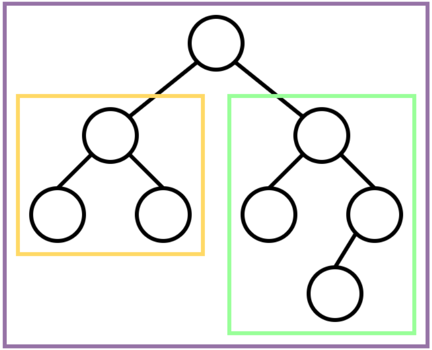
É toda árvore binária de busca cujo custo das operações de busca, inserção, remoção e reorganização da árvore mantém-se em $O(\log n)$.

Definição

Árvore AVL *é uma árvore de busca binária altamente balanceada. Em tal árvore, as alturas das duas sub-árvores a partir de cada nó diferem no máximo em uma unidade.*

*__AVL__ = __A__delson-__V__elskii, G. e __L__andis, E. M.

- Também chamada de árvore balanceada pela altura.
- Se uma dada árvore é dita AVL, então todas as suas sub-árvores também são AVL.



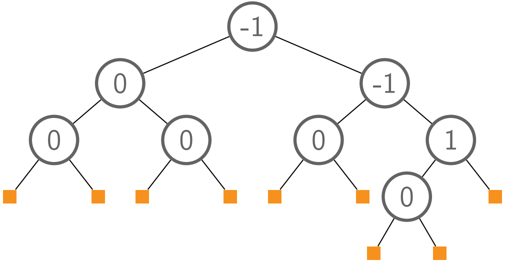
- As operações feitas em uma árvore AVL geralmente envolvem os mesmos algoritmos de uma árvore de busca binária.
- A altura de uma árvore AVL com n nós é $O(\log n)$. Assim, suas operações levam um tempo $O(\log n)$.
- Para definir o balanceamento é utilizado um fator específico:

$$FB(v) = h_e(v) - h_d(v)$$

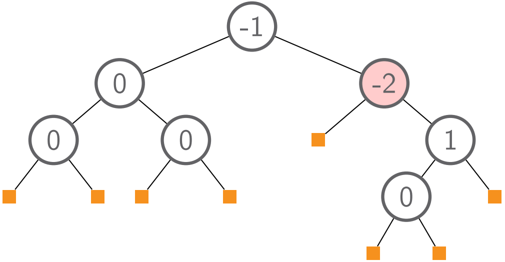
  - $FB(v)$: fator de balanceamento do nó v
  - $h_e(v)$: altura da sub-árvore esquerda
  - $h_d(v)$: altura da sub-árvore direita

- Nós balanceados (ou regulados) são aqueles onde os valores de FB são -1, 0 ou +1.
  - -1 sub-árvore direita mais alta que a esquerda
  - 0 sub-árvore esquerda igual a direita
  - +1 sub-árvore esquerda mais alta que a direita
- Qualquer nó com FB diferente desses valores é dito desregulado
  - > 1 sub-árvore esquerda está desregulando o nó
  - $<$ -1 sub-árvore direita está desregulando o nó
- Se todos os nós da árvore são regulados, então a árvore é AVL.

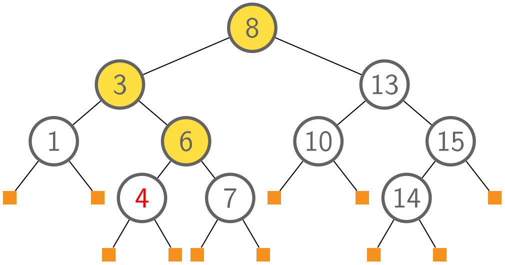
- Nos dois exemplos de árvore não AVL abaixo, percebe-se em vermelho o nó desregulado com $FB <$ -1.



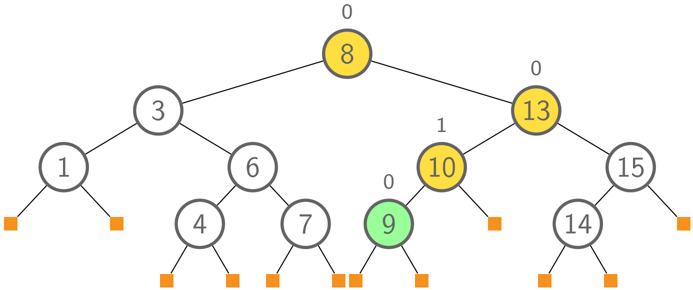
AVL



Não-AVL

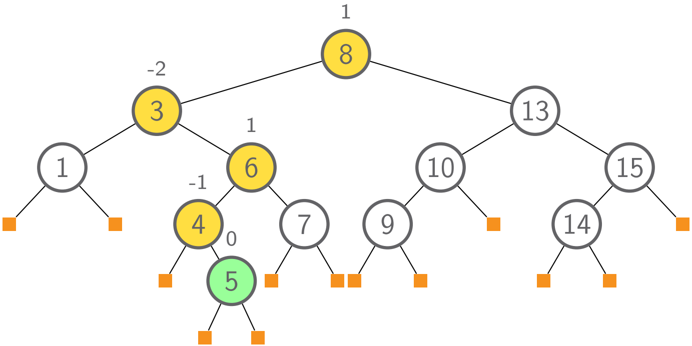


- Busca pela chave 4 na árvore binária acima: 8-3-6-4
- A chave mínima é 1, seguindo os ponteiros a esquerda a partir da raiz.
- A chave máxima é 15, seguindo os ponteiros a direita a partir da raiz.



- Inserção do nó com chave 9. Os nós coloridos indicam o caminho desde a raiz até a posição em que o item é inserido.
- A cada inserção, é preciso verificar se a árvore continua sendo AVL.

- Se T não continua AVL depois da inclusão de um vértice q. Procure o nó mais próximo da folha q que se tornou desregulado.
- Seja p o nó desregulado, ele está entre a raiz e a folha q



- A operação de remoção de um nó z em uma árvore binária de busca consiste de 3 casos:
  a) Se z não tem filhos, modificar o pai de z para que este aponte para `NULL`.
  b) Se z possui apenas um filho, fazemos o pai de z apontar para o filho de z.
  c) Se z possui dois filhos, colocamos no lugar de z o seu sucessor, que com certeza não possui filho à esquerda.
- O sucessor de um nó x é o nó y com a menor chave que seja maior do que a chave de x.

- Quando as operações de inserção e remoção alteram o balanceamento da árvore, é necessário efetuar uma **rotação** para manter as propriedades da árvore AVL, tal que:
  a) O percurso **em ordem** fique inalterado em relação a árvore desbalanceada. Em outras palavras, a árvore continua a ser uma árvore de busca binária.
  b) A árvore modificada saiu de um estado de desbalanceamento para um estado de **balanceamento**.

Operação de Rotação

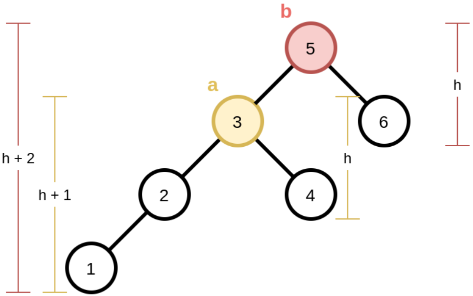
Definição

A operação de rotação altera o balanceamento de uma árvore T, garantindo a propriedade AVL e a sequência de percurso em ordem.

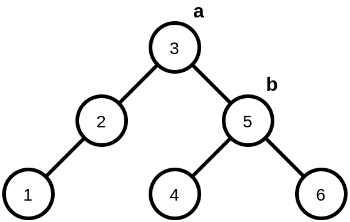
- Podemos definir 4 tipos diferentes de inserções, que tornam necessário o uso de rotação:
  1. Inserção à esquerda, depois á esquerda (LL)
  2. Inserção à direita, depois á direita (RR)
  3. Inserção à esquerda, depois á direita (LR)
  4. Inserção à direita, depois á esquerda (RL)

- Quando aplicar a rotação à direita?

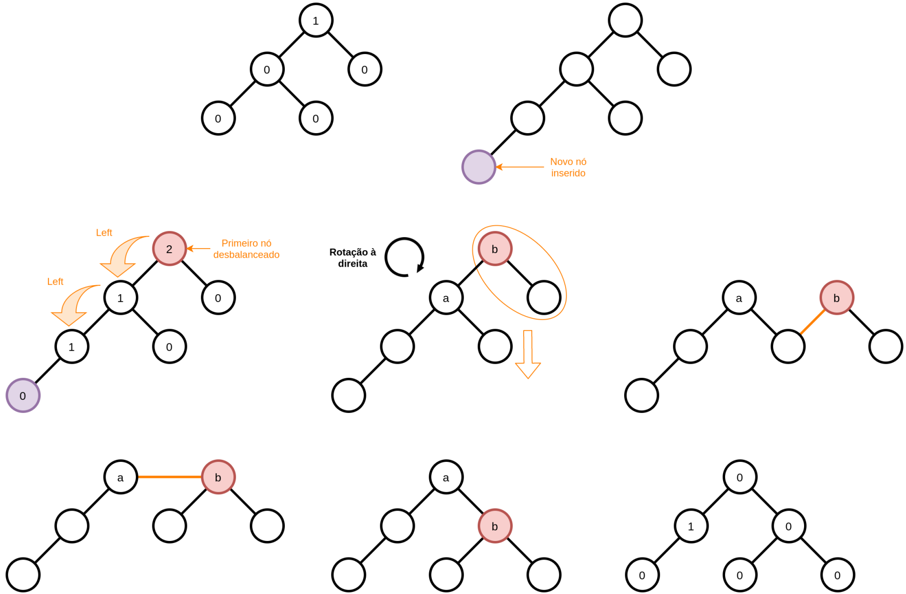
$$h_e(a) > h_d(a)$$
$$h_e(b) > h_d(b)$$



Desbalanceada
após inserção

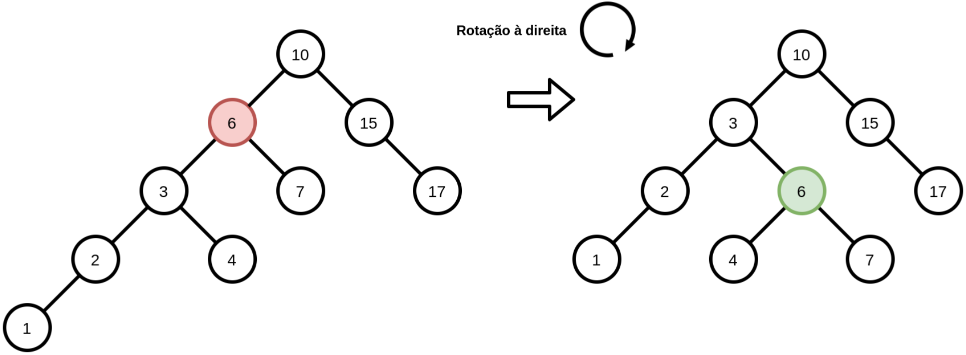


Rebalanceada



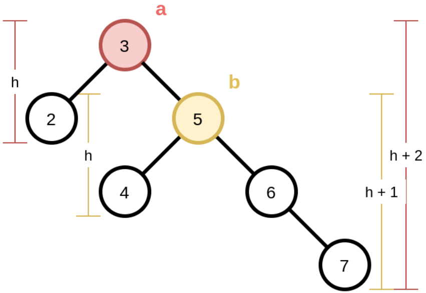
Novo nó
inserido
Left
Left
Primeiro nó
desbalanceado
Rotação à
direita

- Exemplo:

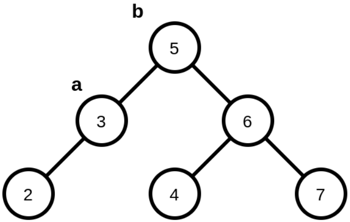


- Quando aplicar a rotação à esquerda?

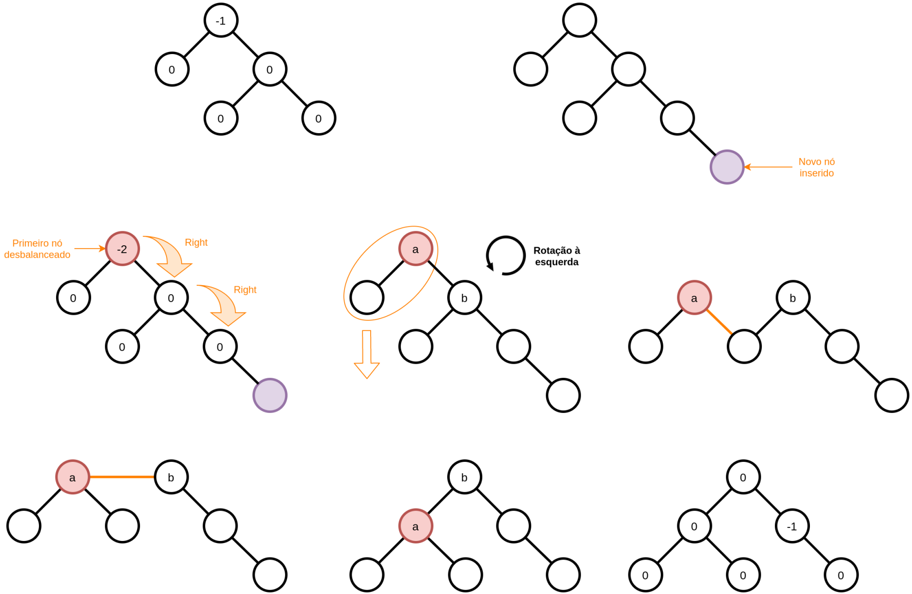
$$h_e(a) < h_d(a)$$
$$h_e(b) < h_d(b)$$



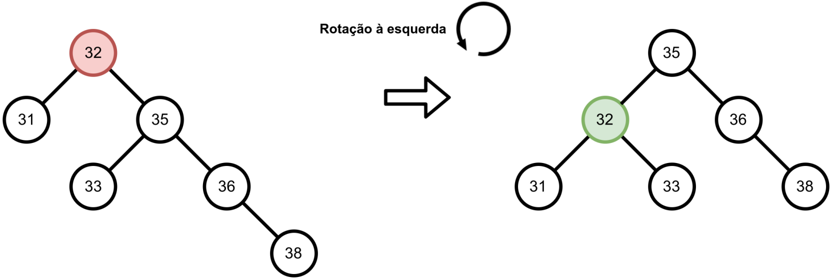
Desbalanceada
após inserção



Rebalanceada

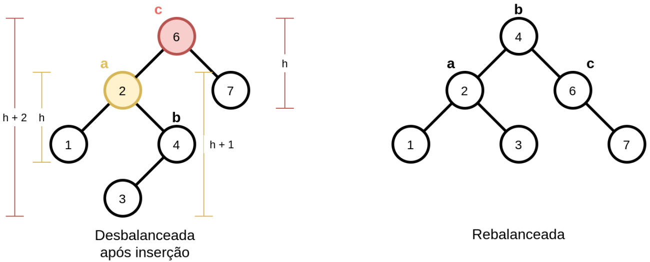

-1
Novo nó
inserido
Primeiro nó
desbalanceado
-2
Right
Right
a
b
Rotação à
esquerda

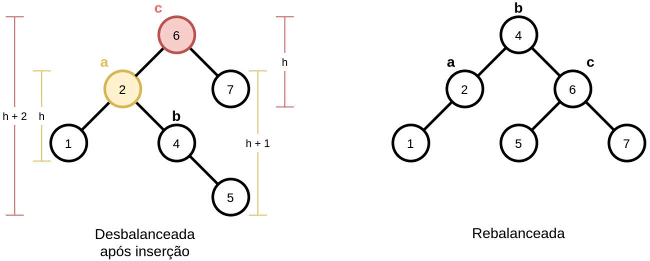
- Exemplo:

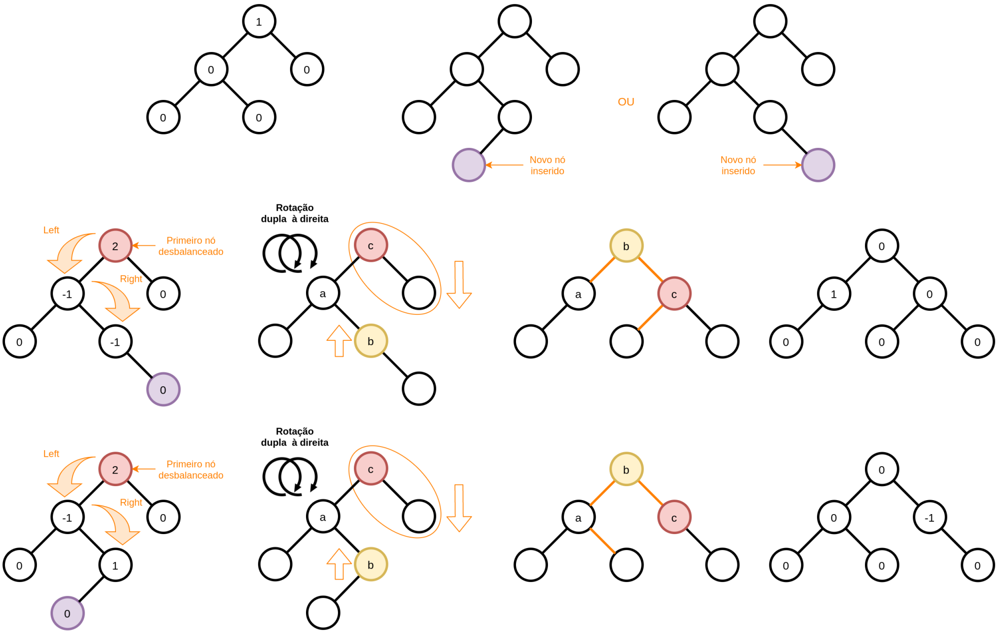


- Quando aplicar a rotação dupla à esquerda?

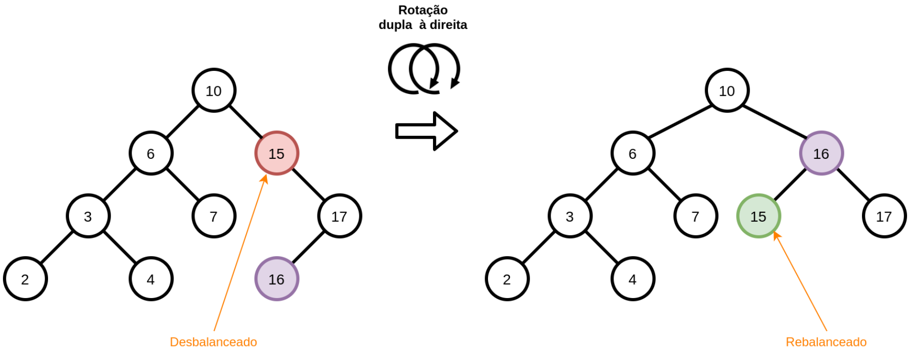
$$h_e(a) < h_d(a)$$
$$h_e(c) > h_d(c)$$






OU
Novo nó inserido
Novo nó inserido
Left
Right
Primeiro nó desbalanceado
Rotação dupla à direita
Left
Right
Primeiro nó desbalanceado
Rotação dupla à direita

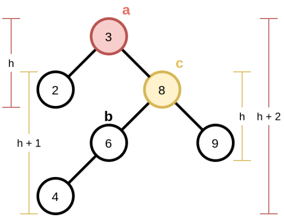
- Exemplo:



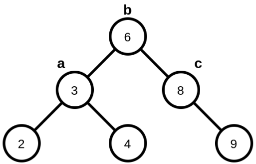
Rotação
dupla à direita
10
6
15
3
7
17
2
4
16
Desbalanceado
10
6
16
3
7
15
17
2
4
Rebalanceado

- Quando aplicar a rotação dupla à direita?

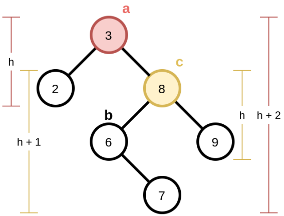
$$h_e(a) > h_d(a)$$
$$h_e(c) < h_d(c)$$



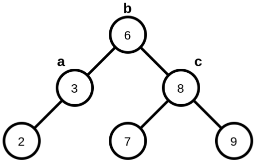
Desbalanceada após inserção



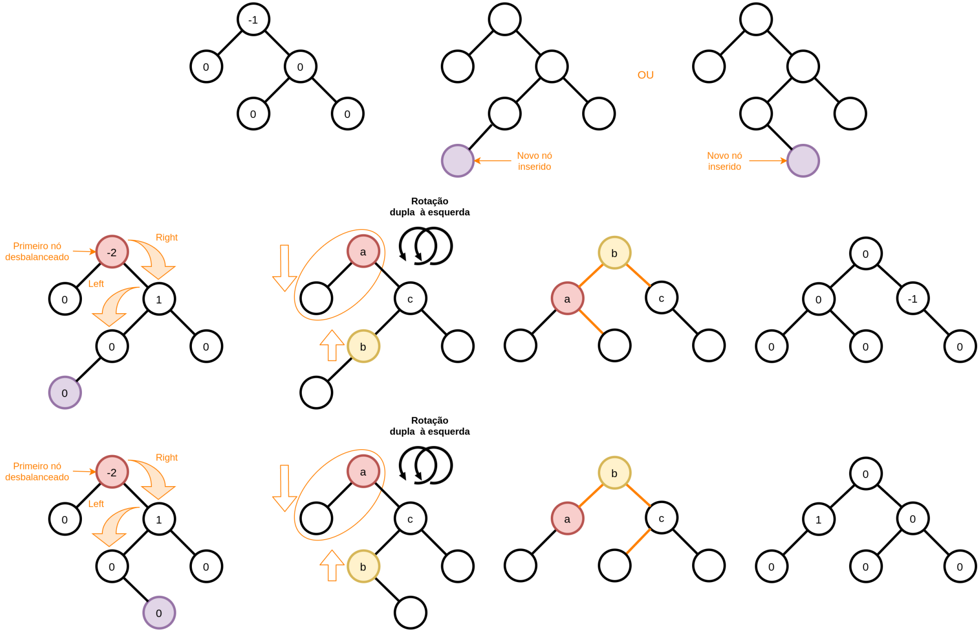
Rebalanceada



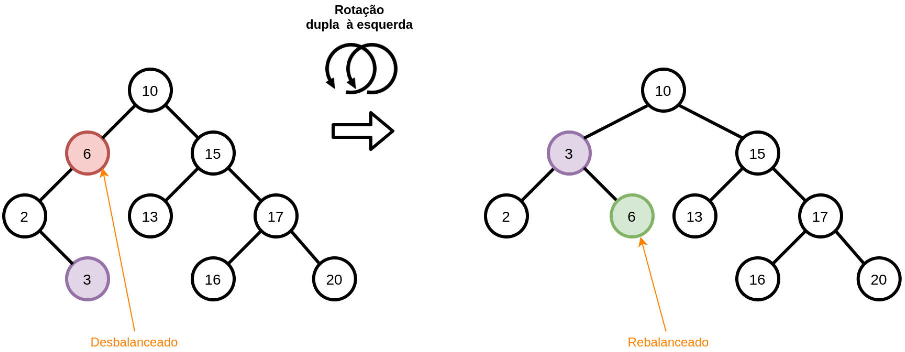
Desbalanceada após inserção



Rebalanceada


OU
Novo nó inserido
Novo nó inserido
Rotação dupla à esquerda
Primeiro nó desbalanceado
Right
Left
Rotação dupla à esquerda
Primeiro nó desbalanceado
Right
Left

- Exemplo:


Rotação
dupla à esquerda
10
6
15
2
13
17
3
16
20
Desbalanceado
10
3
15
2
6
13
17
16
20
Rebalanceado

Código 1: Exemplo de implementação

```c
typedef struct AVL_node {
    void *item;
    struct AVL_node *left, *right;
    signed char balance;
} AVL_node_t;

typedef struct AVL {
    AVL_node_t *root;
    int n;
    int (* compar)(const void *, const void *);
} AVL_t;

AVL_t *AVL_alloc(int (* compar)(const void *, const void *));
void  AVL_free(AVL_t *t);
void *AVL_insert(AVL_t *t, void *item);
void *AVL_find(AVL_t *t, void *key_item);
void *AVL_find_min(AVL_t *t);
void *AVL_delete(AVL_t *t, void *key_item);
void *AVL_delete_min(AVL_t *t);
```