

# Árvores AVL

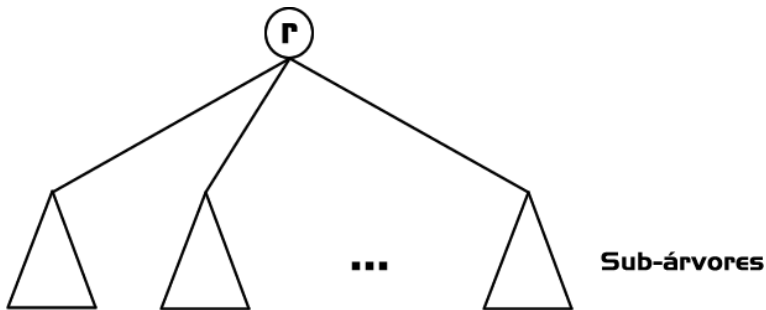
Prof. Jefferson T. Oliva

Algoritmos e Estrutura de Dados II (AE23CP)  
Engenharia de Computação  
Departamento Acadêmico de Informática (Dainf)  
Universidade Tecnológica Federal do Paraná (UTFPR)  
Campus Pato Branco

- Balanceamento
- Árvores AVL
  - Fator de balanceamento
  - Rotações
  - Operações

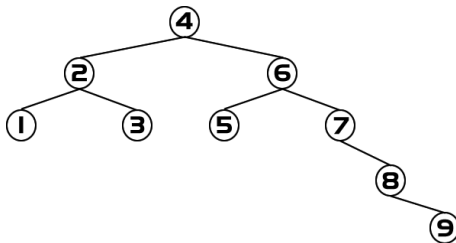
# Considerações Iniciais

- Árvore



# Considerações Iniciais

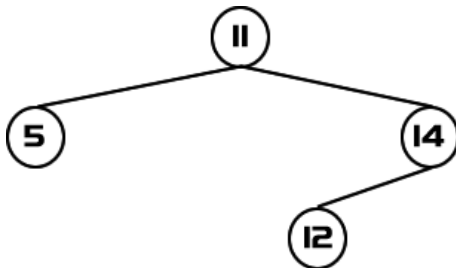
- As árvores binárias de busca (pesquisa) são projetadas para um acesso rápido à informação
  - Idealmente a árvore deve ser razoavelmente equilibrada
- Tempo de busca é de  $O(\log n)$  para uma árvore balanceada
- Sucessivas inserções de itens podem acarretar no aumento da complexidade de tempo para  $O(n)$



## Balanceamento

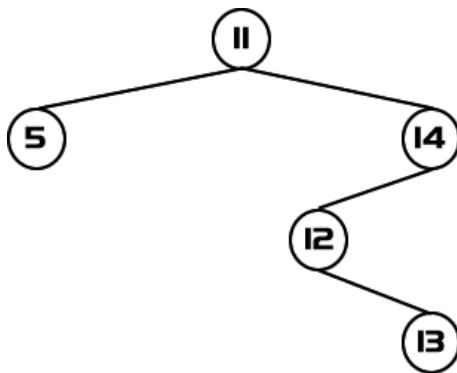
- Árvores binárias de busca balanceadas minimizam o número de comparações em comparação com o pior caso ( $O(n)$ )
  - A altura da árvore é mantida baixa (por volta de  $O(\log n)$ ) após sucessivas inserções
  - Uma árvore de altura  $h$  pode conter, no máximo,  $2^{h+1} - 1$  elementos
  - A diferença de altura das sub-árvores direita e esquerda deve ser no máximo um
- A manutenção de árvores de busca balanceadas é considerada uma tarefa complexa

- Árvore balanceada



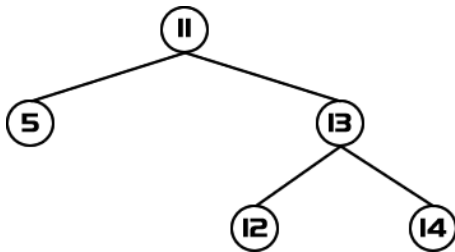
# Balanceamento

- Inserção do item 13 torna a árvore desbalanceada





- Árvore rebalanceada



- Exemplos de tipos de árvores binárias balanceadas:
  - Árvores AVL
  - Árvores vermelha-preta (rubro-negra)

## Árvores AVL

- Adelson, Velsky e Landis (1962)
- Árvore de altura balanceada
- As operações de busca, inserção e remoção podem ser realizadas a um custo de tempo  $O(\log n)$
- Uma árvore vazia é uma árvore AVL

# Árvores AVL

## Fator de balanceamento

- Dada pela diferença de altura entre as sub-árvores esquerda ( $h_e$ ) e direita ( $h_d$ )
  - $h_e - h_d$
- Em uma Árvore AVL, cada sub-árvore deve ter altura equilibrada (de acordo com o fator de balanceamento)

# Árvores AVL

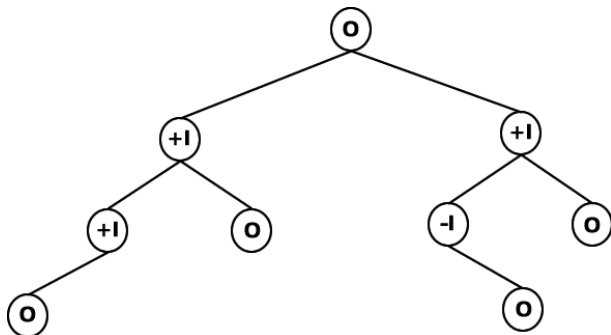
## Fator de balanceamento

- Cada nó de uma árvore AVL deve ter um valor de fator de balanceamento
  - -1: a altura da sub-árvore direita é maior que a da esquerda
  - 0: a altura das sub-árvores direita e esquerda são iguais
  - +1: a altura da sub-árvore esquerda é maior que a da direita
- Em uma operação de inserção ou remoção, caso uma sub-árvore fique com altura menor que -1 ou maior que +1, a árvore deve ser rebalanceada

# Árvores AVL

## Fator de balanceamento

- Exemplo de árvore balanceada com fator de balanceamento em cada nó



- *Left-left* (LL)
- *Right-right* (RR)
- *Left-right* (LR)
- *Right-left* (RL)



- Exemplo
  - Inserção do item maio

**Após a inserção**



**Após o rebalanceamento**

Sem necessidade de  
rebalanceamento

- Exemplo
  - Inserção do item março

**Após a inserção**



**Após o rebalanceamento**

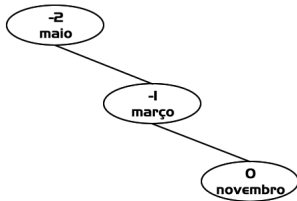
Sem necessidade de  
rebalanceamento

# Árvores AVL

## Rotações

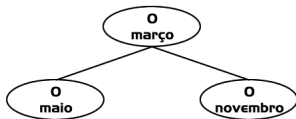
- Exemplo
  - Inserção do item novembro

**Após a inserção**



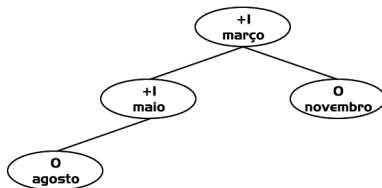
RR  
→

**Após o rebalanceamento**



- Exemplo
  - Inserção do item agosto

**Após a inserção**



**Após o rebalanceamento**

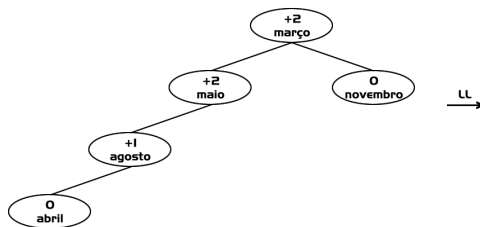
Sem necessidade de  
rebalanceamento

# Árvores AVL

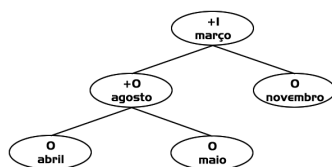
## Rotações

- Exemplo
  - Inserção do item abril

Após a inserção



Após o rebalanceamento

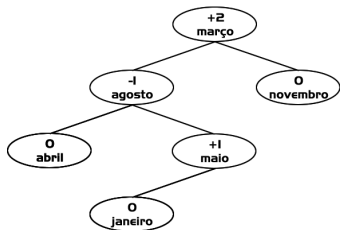


# Árvores AVL

## Rotações

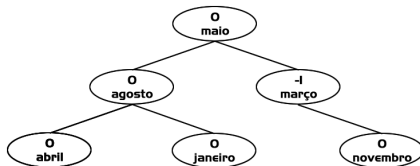
- Exemplo
  - Inserção do item janeiro

Após a inserção



LR →

Após o rebalanceamento

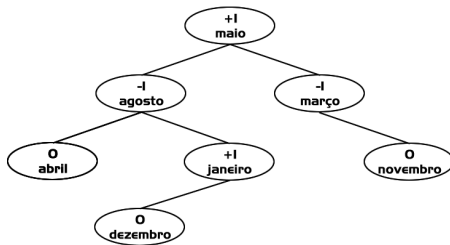


# Árvores AVL

## Rotações

- Exemplo
  - Inserção do item dezembro

**Após a inserção**



**Após o rebalanceamento**

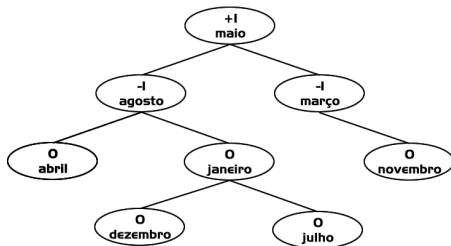
Sem necessidade de rebalanceamento

# Árvores AVL

## Rotações

- Exemplo
  - Inserção do item julho

**Após a inserção**



**Após o rebalanceamento**

Sem necessidade de  
rebalanceamento

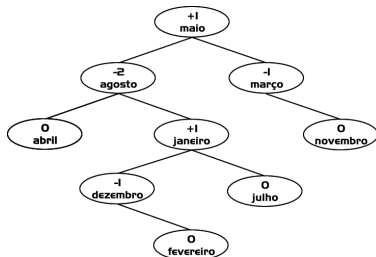


# Árvores AVL

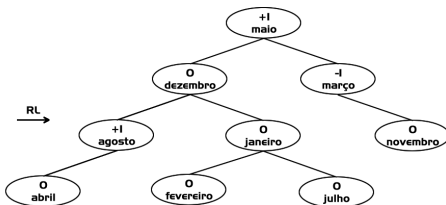
## Rotações

- Exemplo
  - Inserção do item fevereiro

Após a inserção



Após o rebalanceamento



- Estrutura de dados para a representação de uma árvore AVL:

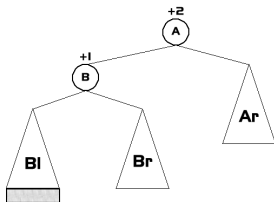
```
typedef struct Pointer{  
    int item;  
    int bf;  
    struct Pointer* right;  
    struct Pointer* left;  
}Node;
```

# Árvores AVL

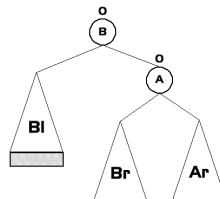
## Rotações

### • Rotação LL

árvore desbalanceada após a inserção



árvore rebalanceada

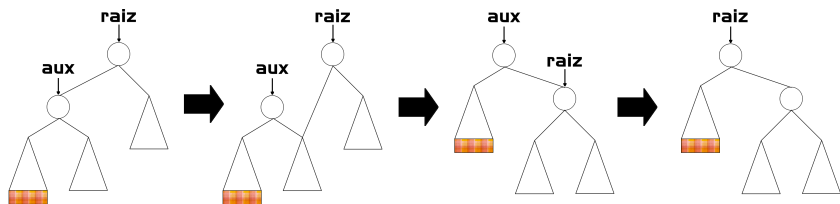


### • Legenda:

- Círculo: representa um nó
- Triângulo: representa uma sub-árvore equilibrada
  - Nos exemplos ilustrados para as rotações **LL** e **RR**, cada sub-árvore possui o mesmo tamanho
- Retângulo: representa o aumento da altura de uma sub-árvore (inclusão de um novo nó)

### • Rotação LL

```
Node *aux = raiz->left;  
raiz->left = aux->right;  
aux->right = raiz;  
raiz = aux;
```

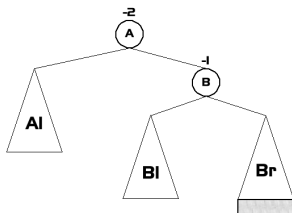


# Árvores AVL

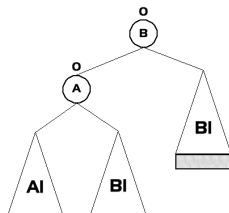
## Rotações

- Rotação RR

árvore desbalanceada após a inserção

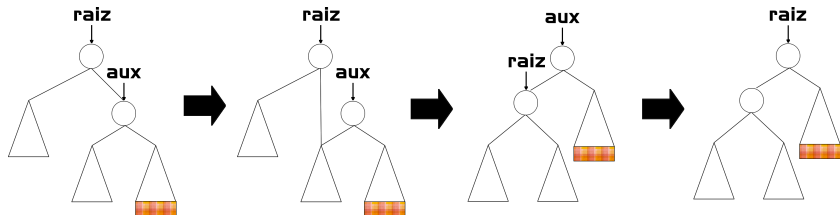


árvore rebalanceada



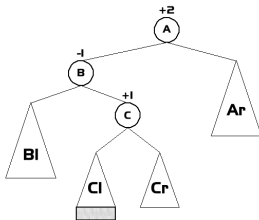
### • Rotação RR

```
Node *aux = raiz->right;  
raiz->right = aux->left;  
aux->left = raiz;  
raiz = aux;
```

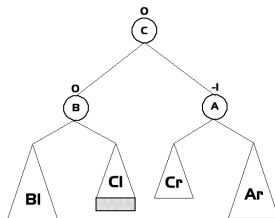


- Rotação LR: caso 1

árvore desbalanceada após a inserção



árvore rebalanceada



- Legenda:

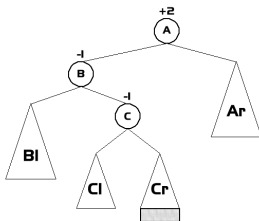
- Círculo e retângulo possuem o mesmo significado em relação aos exemplos de rotações LL e RR
- Triângulos: representa uma sub-árvore equilibrada
  - A e B: pode-se dizer que possuem o mesmo significado em relação aos exemplos de rotações LL e RR
  - C: sub-árvore equilibrada, mas com altura brevemente menor (diferença de uma unidade) em relação às sub-árvores A e B

# Árvores AVL

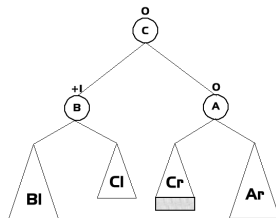
## Rotações

- Rotação LR: caso 2

árvore desbalanceada após a inserção



árvore rebalanceada



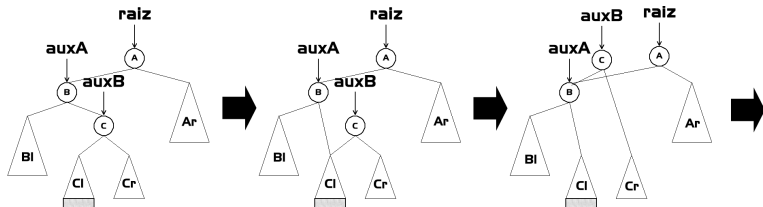


# Árvores AVL

## Rotações

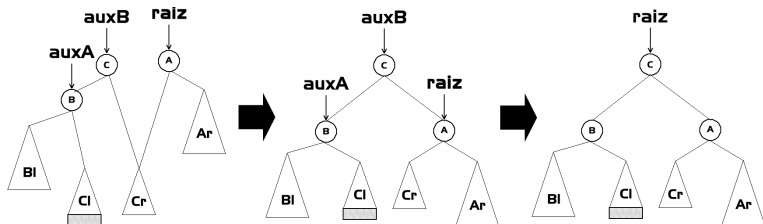
### • Rotação LR

```
Node *auxA = raiz->left;  
Node *auxB = auxA->right;  
auxA->right = auxB->left;  
auxB->left = auxA;  
raiz->left = auxB->right;  
auxB->right = raiz;  
raiz = auxB;
```



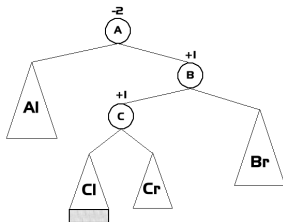
### • Rotação LR

```
Node *auxA = raiz->left;  
Node *auxB = auxA->right;  
auxA->right = auxB->left;  
auxB->left = auxA;  
raiz->left = auxB->right;  
auxB->right = raiz;  
raiz = auxB;
```

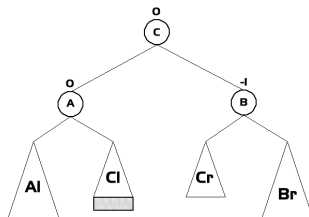


- Rotação RL: caso 1

árvore desbalanceada após a inserção

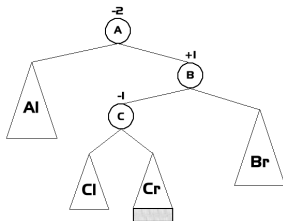


árvore rebalanceada

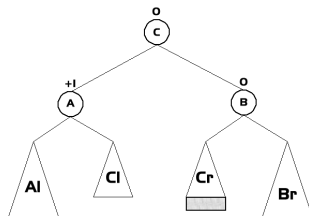


- Rotação RL: caso 2

árvore desbalanceada após a inserção

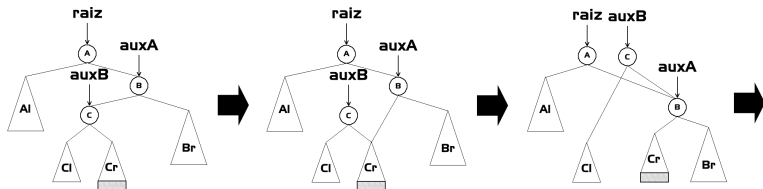


árvore rebalanceada



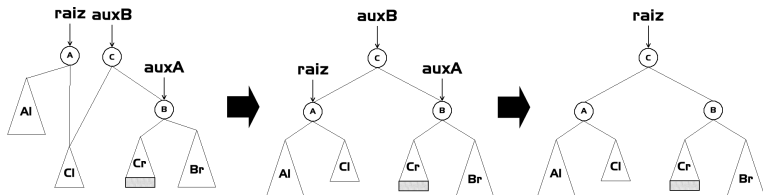
- Rotação RL

```
Node *auxA = raiz->right;  
Node *auxB = auxA->left;  
auxA->left = auxB->right;  
auxB->right = auxA;  
raiz->right = auxB->left;  
auxB->left = raiz;  
raiz = auxB;
```



### • Rotação RL

```
Node *auxA = raiz->right;  
Node *auxB = auxA->left;  
auxA->left = auxB->right;  
auxB->right = auxA;  
raiz->right = auxB->left;  
auxB->left = raiz;  
raiz = auxB;
```



- A inserção e a remoção de itens em árvores AVL são realizadas da mesma forma que em árvores binárias de busca apresentadas na aula anterior
  - A diferença é que pode ser necessário rebalanceamento da árvore após essas operações
- A operação de busca, inserção e remoção tem a complexidade de tempo  $O(\log n)$
- Os códigos de inserção e de remoção em árvores AVL estão disponíveis em: <[https://github.com/jefferson-oliva/material\\_grad/tree/main/AE23CP-Algoritmos-2/aula\\_09\\_%C3%A1rvores\\_AVL](https://github.com/jefferson-oliva/material_grad/tree/main/AE23CP-Algoritmos-2/aula_09_%C3%A1rvores_AVL)>

# Árvores AVL

## Operações: inserção

```
Node rotateL(Node *raiz){
    Node *auxA = raiz->left, *auxB;

    if (auxA->fb == +1){ // Rotação LL
        raiz->left = auxA->right;
        auxA->right = raiz;
        raiz->fb = 0;
        raiz = auxA;
    }else{ Rotação RL, pois fb será negativo
        auxB = auxA->right;
        auxA->right = auxB->left;
        auxB->left = auxA;
        raiz->left = auxB->right;
        auxB->right = raiz;

        // Se a rotação LR foi para o caso 1
        if (auxB->fb == +1) raiz->fb = -1;
        else raiz->fb = 0;

        // Se a rotação LR foi para o caso 2
        if (auxB->fb == -1) auxA->fb = +1;
        else auxA->fb = 0;

        raiz = auxB;
    }
    return tree;
}
```



# Árvores AVL

## Operações: inserção

```
Node* rotateR(Node *raiz){
    Node *auxA = raiz->right, *auxB;

    if (auxA->fb == -1){ // Rotação RR
        raiz->right = auxA->left;
        auxA->left = raiz;
        raiz = auxA;
    }else{ // Rotação RL
        auxB = auxA->left;
        auxA->left = auxB->right;
        auxB->right = auxA;
        raiz->right = auxB->left;
        auxB->left = raiz;

        // Se a rotação RL foi para o caso 1
        if (auxB->fb == -1) raiz->fb = +1;
        else raiz->fb = 0;

        // Se a rotação RL foi para o caso 1
        if (auxB->fb == +1) auxA->fb = -1;
        else auxA->fb = 0;

        raiz = auxB;
    }
    return tree;
}
```

# Árvores AVL

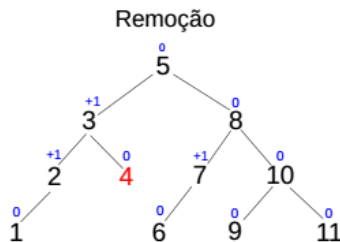
## Operações: inserção

```
void inserir(Node* raiz, int value, int *grown){
    if(tree == NULL){
        tree = criar(value);
        *grown = 1;
    }else if (value < tree->item){
        tree->left = inserir(tree->left, value, grown);
        if (*grown){ // verificar se aumentou a árvore pelo lado esquerdo
            switch (tree->fb){
                case -1: tree->fb = 0; *grown = 0; break;
                case 0:  tree->fb = +1; break
                case +1: tree = rotateL(tree); tree->fb = 0; *grown = 0;
            }
        }
    }else if (value > tree->item){
        tree->right = inserir(tree->right, value, grown);
        if (*grown){
            switch (tree->fb){ // verificar se aumentou a árvore pelo lado direito
                case +1: tree->fb = 0; *grown = 0; break;
                case 0:  tree->fb = -1; break;
                case -1: tree = rotateR(tree); tree->fb = 0; *grown = 0;
            }
        }
    }
    return tree;
}
```

# Árvores AVL

Operações: remoção

- Exemplo
  - Remoção do item 4

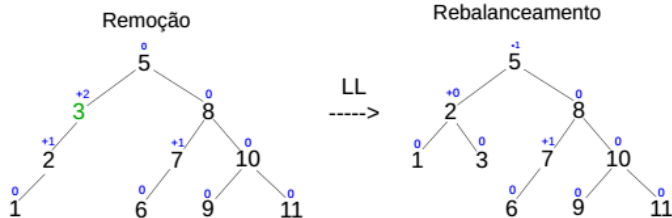


Rebalanceamento

# Árvores AVL

Operações: remoção

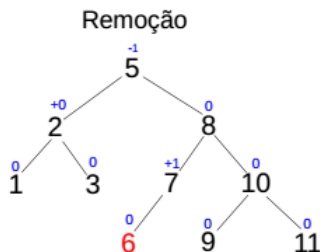
- Exemplo
  - Remoção do item 4



# Árvores AVL

Operações: remoção

- Exemplo
  - Remoção do item 6

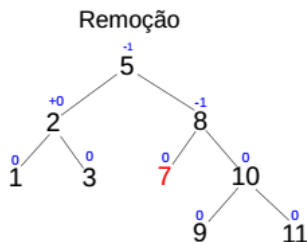


Rebalanceamento

# Árvores AVL

Operações: remoção

- Exemplo
  - Remoção do item 7

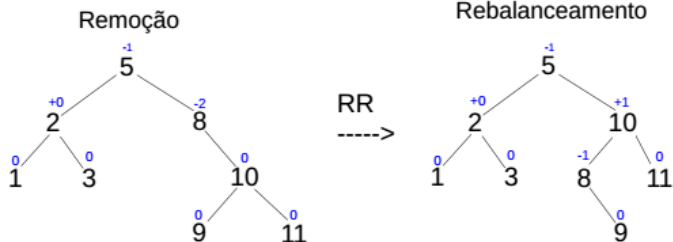


Rebalanceamento

# Árvores AVL

Operações: remoção

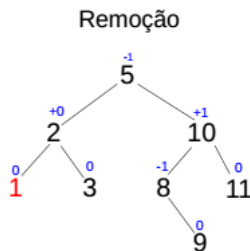
- Exemplo
  - Remoção do item 7



# Árvores AVL

Operações: remoção

- Exemplo
  - Remoção do item 1



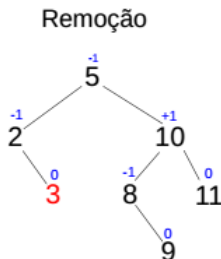
Rebalanceamento



# Árvores AVL

Operações: remoção

- Exemplo
  - Remoção do item 3

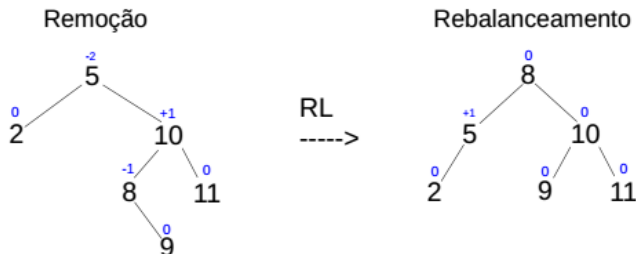


Rebalanceamento

# Árvores AVL

Operações: remoção

- Exemplo
  - Remoção do item 3



# Árvores AVL

## Operações: remoção

```
Node* rotateLRM(Node *tree, int *reduced){
    Node *auxA, *auxB;
    switch (tree->fb){
        case +1: tree->fb = 0; break; // AVL balanceada
        case 0:  tree->fb = -1; *reduced = 0; break; // AVL balanceada
        case -1:
            auxA = tree->right;
            if (auxA->fb <= 0){ // rotação RR
                tree->right = auxA->left;
                auxA->left = tree;
                if(auxA->fb == 0){
                    tree->fb = -1; auxA->fb = +1; *reduced = 0;
                }else
                    tree->fb = auxA->fb = 0;
                tree = auxA;
            }else{ // Rotação RL
                auxB = auxA->left;
                auxA->left = auxB->right;
                auxB->right = auxA;
                tree->right = auxB->left;
                auxB->left = tree;
                if (auxB->fb == -1) tree->fb = +1; else tree->fb = 0;
                if (auxB->fb == +1) auxA->fb = -1; else auxA->fb = 0;
                tree = auxB;
                auxB->fb = 0;
            }
        }
    }
    return tree;
}
```

# Árvores AVL

## Operações: remoção

```
Node* rotateRRM(Node *tree, int *reduced){
    Node *auxA, *auxB;
    switch (tree->fb){
        case -1: tree->fb = 0; *reduced = 0; break;
        case 0:  tree->fb = 1; break;
        case +1:
            auxA = tree->left;
            if (auxA->fb >= 0){ // rotação LL
                tree->left = auxA->right;
                auxA->right = tree;
                if(auxA->fb == 0){
                    tree->fb = +1; auxA->fb = -1; *reduced = 0;
                }else
                    tree->fb = auxA->fb = 0;
                tree = auxA;
            }else{ // Rotação LR
                auxB = auxA->right;
                auxA->left = auxB->left;
                auxB->left = auxA;
                tree->left = auxB->right;
                auxB->right = tree;
                if (auxB->fb == +1) tree->fb = -1; else tree->fb = 0;
                if (auxB->fb == -1) auxA->fb = +1; else auxA->fb = 0;
                tree = auxB;
                auxB->fb = 0;
            }
        }
    }
    return tree;
}
```


# Árvores AVL

## Operações: remoção

```
Node* remover(Node* tree, int valor, int *reduced){
    Node *aux, *auxP, *auxF;
    if (tree != NULL){
        if (valor < tree->item){
            tree->left = remover(tree->left, valor, reduced);
            if (*reduced) tree = rotateLRM(tree, reduced);
        }else if (valor > tree->item){
            tree->right = remover(tree->right, valor, reduced);
            if (*reduced) tree = rotateRRM(tree, reduced);
        }else{
            aux = tree;
            if (aux->left == NULL) tree = tree->right;
            else if (aux->right == NULL) tree = tree->left;
            else{
                auxF = auxP = aux->right;
                while (auxF->left != NULL){
                    auxP = auxF; auxF = auxF->left;
                }
                if (auxP != auxF){
                    auxP->left = auxF->right;
                    auxF->left = aux->left;
                }
                auxF->right = aux->right;
                tree = auxF;
            }
            *reduced = 1;
            free(aux);
        }
    }
    return tree;
}
```

 Baranauskas, J. A. *Árvores AVL – Algoritmos e Estruturas de Dados I*.

*Slides*. Ciência da Computação FFCLRP-USP, Ribeirão Preto, 2013.

 Cormen, T. H., Leiserson, C. E., Rivest, R. L., Stein, C. *Introduction to Algorithms*.

Third edition, The MIT Press, 2009.

 Marin, L. O.

*Árvores AVL. AE23CP – Algoritmos e Estrutura de Dados II*.


*Slides*. Engenharia de Computação. Dainf/UTFPR/Pato

Branco, 2017.

 Szwarcfiter, J.; Markenzon, L.

*Estruturas de Dados e Seus Algoritmos*.

LTC, 1994.

 Tenenbaum, A.; Langsam, Y.  
*Estruturas de Dados usando C.*  
Pearson, 1995.

 Ziviani, N.  
*Projeto de Algoritmos - com implementações em Java e C++.*  
Thomson, 2007.