# Árvores AVL

Prof. Jefferson T. Oliva

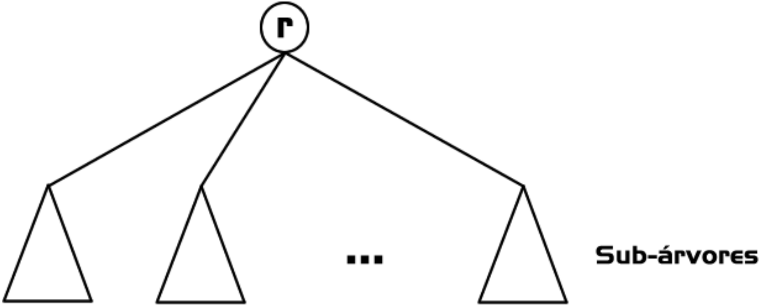
Algoritmos e Estrutura de Dados II (AE23CP)
Engenharia de Computação
Departamento Acadêmico de Informática (Dainf)
Universidade Tecnológica Federal do Paraná (UTFPR)
Campus Pato Branco

# Sumário

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

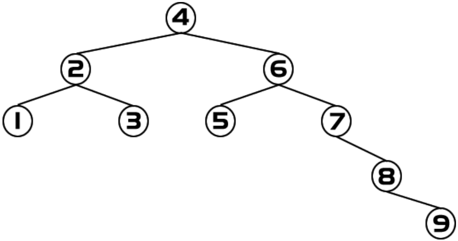
# Considerações Iniciais

- 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 $O(n)$



# Sumário

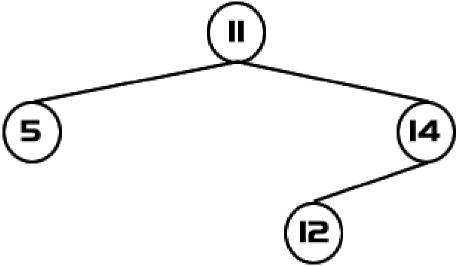
**Balanceamento**

# 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

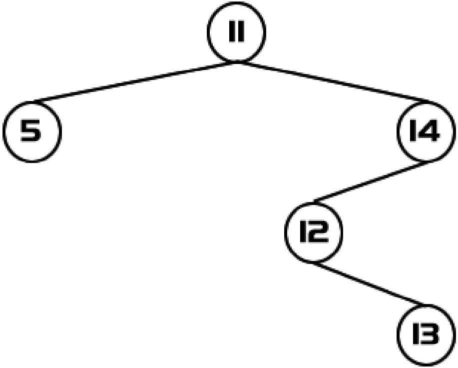
# Balanceamento

- Árvore balanceada



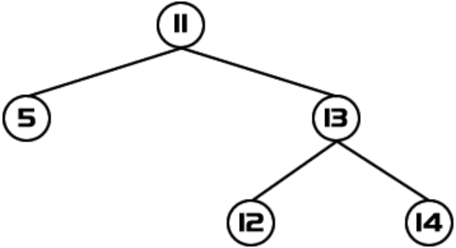
# Balanceamento

- Inserção do item 13 torna a árvore desbalanceada



# Balanceamento

- Árvore rebalanceada



# Balanceamento

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

# Sumário

## Árvores AVL

# Árvores AVL

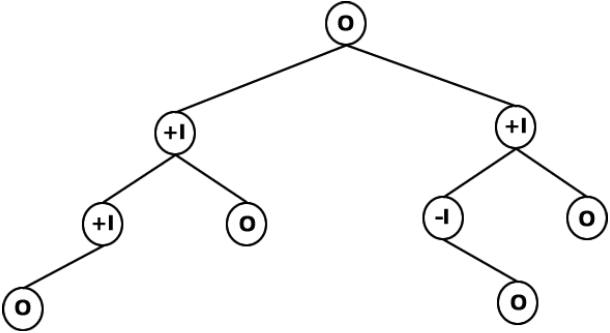
- Adelson, Velsky e Landis (1962)
- Árvore de altura balanceada
- As operações de busca, inserção e remoção podem ser realizadas a um custo de tempo $O(\log n)$
- Uma árvore vazia é uma árvore AVL

- 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)

- 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

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



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

# Árvores AVL
## Rotações

- Exemplo
  - Inserção do item maio

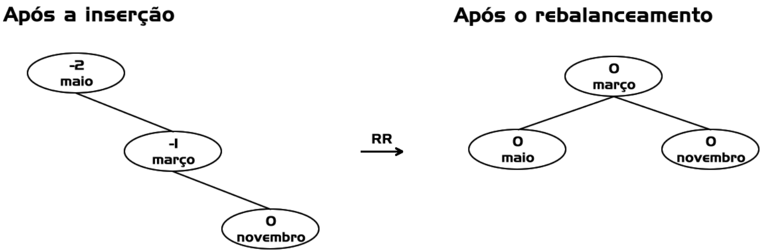
**Após a inserção**


**Após o rebalanceamento**

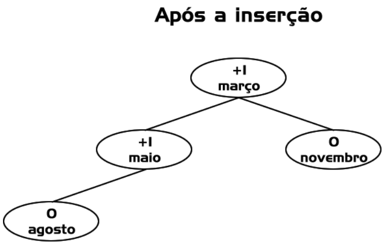
**Sem necessidade de rebalanceamento**

- Exemplo
  - Inserção do item março

- Exemplo
  - Inserção do item novembro



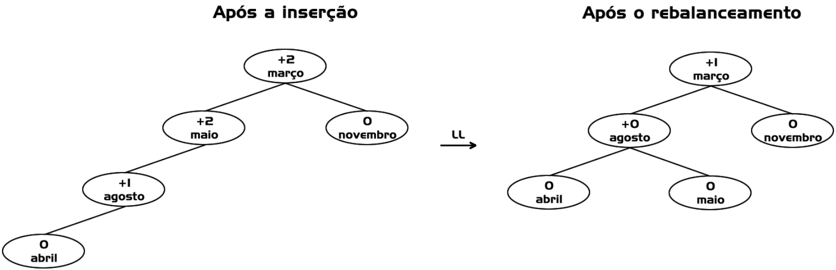
- Exemplo
  - Inserção do item agosto



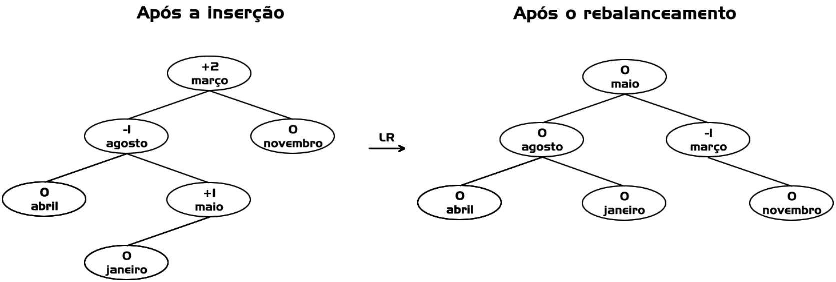

**Após o rebalanceamento**

Sem necessidade de rebalanceamento

- Exemplo
    - Inserção do item abril

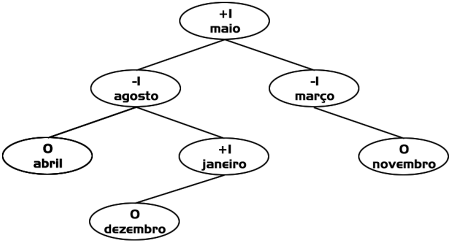


- Exemplo
  - Inserção do item janeiro



- Exemplo
  - Inserção do item dezembro

**Após a inserção**




**Após o rebalanceamento**

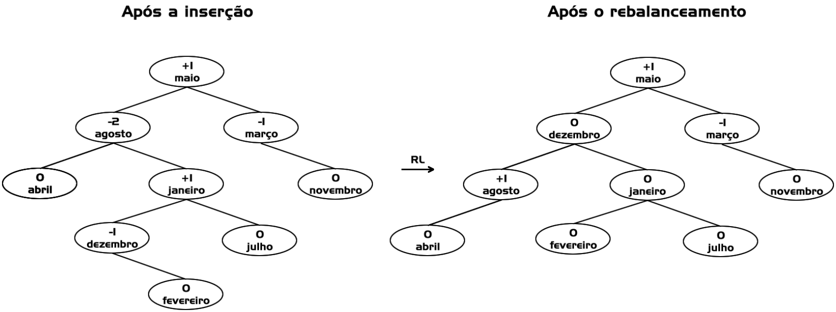
Sem necessidade de rebalanceamento

- Exemplo
  - Inserção do item julho


Após o rebalanceamento

Sem necessidade de rebalanceamento

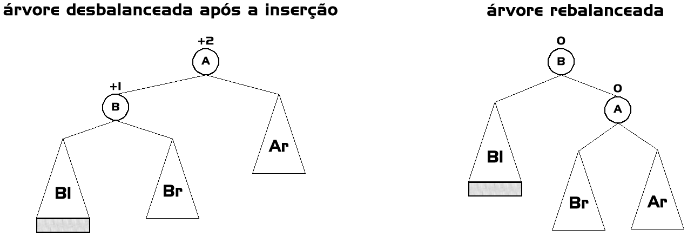
- Exemplo
    - Inserção do item fevereiro



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

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

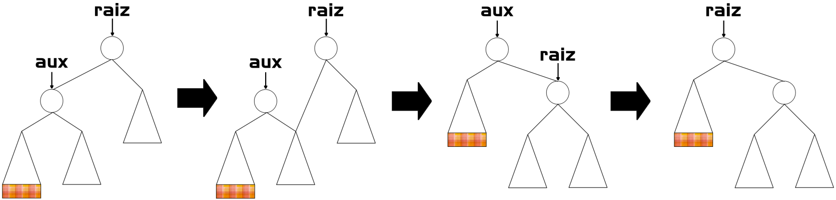
- Rotação LL



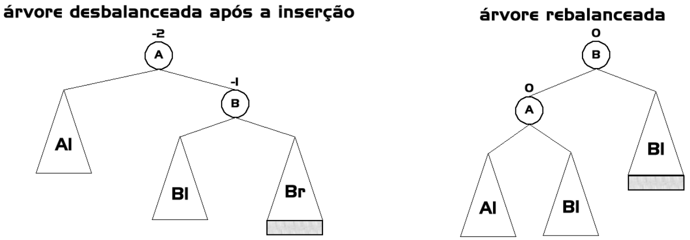

- Legenda:
  - Círculo: representa um nó
  - Triângulo: representa uma sub-árvore equilibrada
    - Nos exemplos ilustrados para as rotações **LL** e **RR**, cada sub-árvore possui o mesmo tamanho
  - Retângulo: representa o aumento da altura de uma sub-árvore (inclusão de um novo nó)

- Rotação LL

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

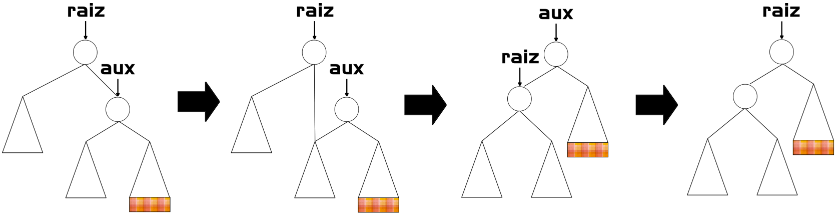


- Rotação RR

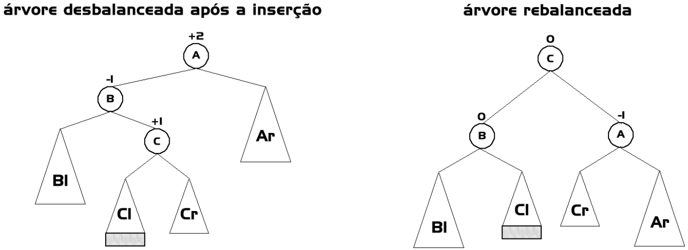

árvore desbalanceada após a inserção
-2
A
-1
B
Al
Bl
Br
árvore rebalanceada
0
B
0
A
Bl
Al
Bl

- Rotação RR

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

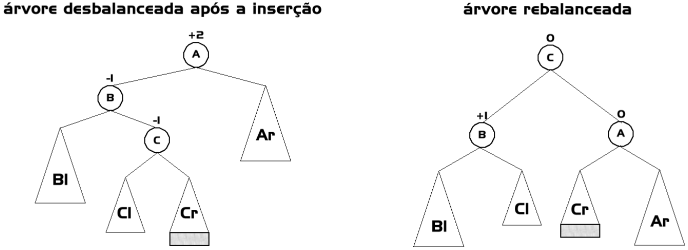


- Rotação LR: caso 1



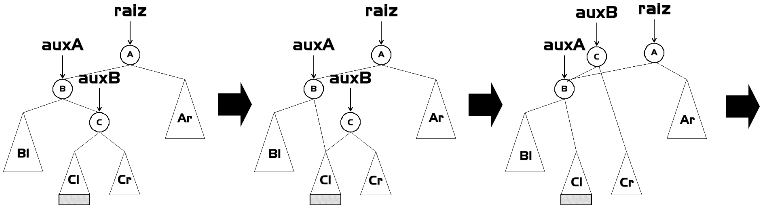

- Legenda:
  - Círculo e retângulo possuem o mesmo significado em relação aos exemplos de rotações LL e RR
  - Triângulos: representa uma sub-árvore equilibrada
    - A e B: pode-se dizer que possuem o mesmo significado em relação aos exemplos de rotações LL e RR
    - C: sub-árvore equilibrada, mas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

- Rotação LR: caso 2



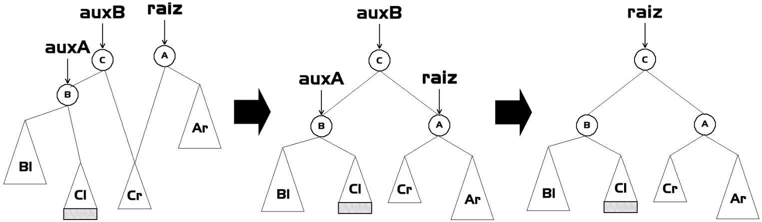
- Rotação LR

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

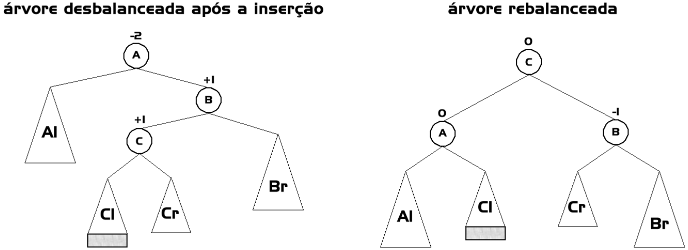


- Rotação LR

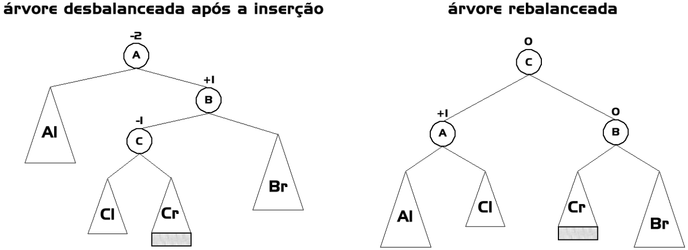
```
Node *auxA = raiz->left;
Node *auxB = auxA->right;
auxA->right = auxB->left;
auxB->left = auxA;
raiz->left = auxB->right;
auxB->right = raiz;
raiz = auxB;
```



- Rotação RL: caso 1

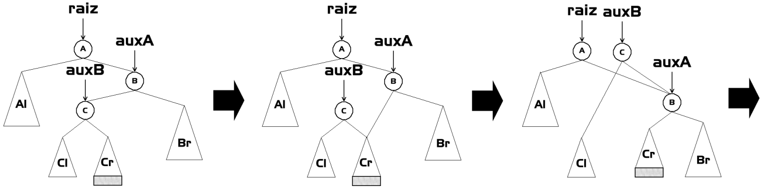


- Rotação RL: caso 2



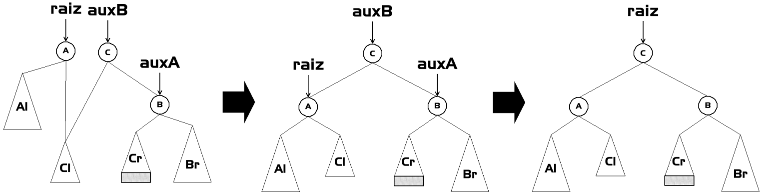
- Rotação RL

```
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 tem a complexidade de tempo $O(\log n)$

```
void insert(Node* raiz, int value, bool *grown){
  Node *auxA, *auxB;

  if(raiz == NULL){ // Caso base
    create(raiz);

    raiz->item = value;
    raiz->fb = 0;

    *grown = true; // Assim que um nó é adicionado, a árvore cresce
  }else if (value < raiz->item){
    insert(raiz->left, value, grown);

    if (*grown){
      switch (raiz->fb){
        case -1:  raiz->fb = 0; *grown = false; break; // árvore balanceada
        case 0:  raiz->fb = +1; break; // árvore balanceada
        case +1:  rotationL(raiz); raiz->fb = 0; *grown = false; // fb seria +2
      }
    }
  }else if (value < raiz->item){
    insert(raiz->right, value, grown);

    if (*grown){
      switch (raiz->fb){
        case +1:  raiz->fb = 0; *grown = false; break; // árvore balanceada
        case 0:  raiz->fb = -1; break; // árvore balanceada
        case -1:  rotationR(raiz); raiz->fb = 0; *grown = false; // fb seria -2
      }
    }
  }
}
```

```
void 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;
  }
}
```

```
void 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;
   }
}
```

# Referências I


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.

# Referências II


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

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