Universidade Federal de Ouro Preto
Campus João Monlevade

CSI 488 – Algoritmos e Estruturas de Dados I

TAD – Árvores AVL

Prof. Mateus Ferreira Satler

Índice

1. Introdução

- A eficiência da busca em uma árvore binária depende do seu balanceamento.
  - **O(log N)**, se a árvore está balanceada.
  - **O(N)**, se a árvore não está balanceada.
  - **N** corresponde ao número de nós na árvore.

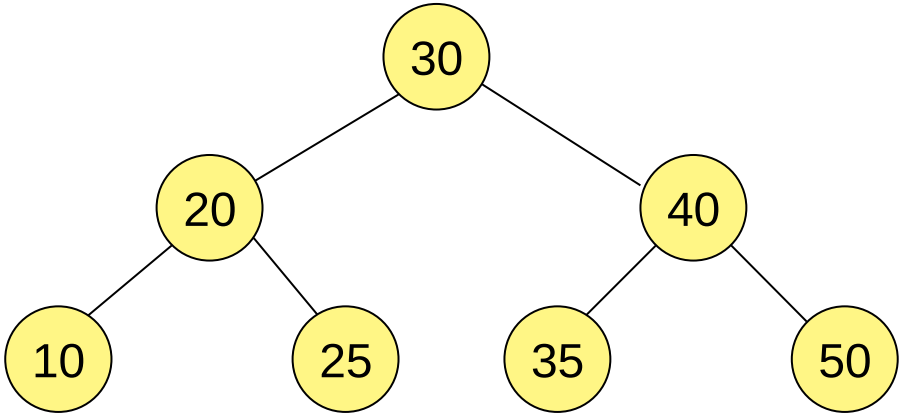
1. Introdução

- Infelizmente, os algoritmos de inserção e remoção em árvores binárias não garantem que a árvore gerada a cada passo esteja balanceada.

- Dependendo da ordem em que os dados são inseridos na árvore, podemos criar uma árvore na forma de uma escada.

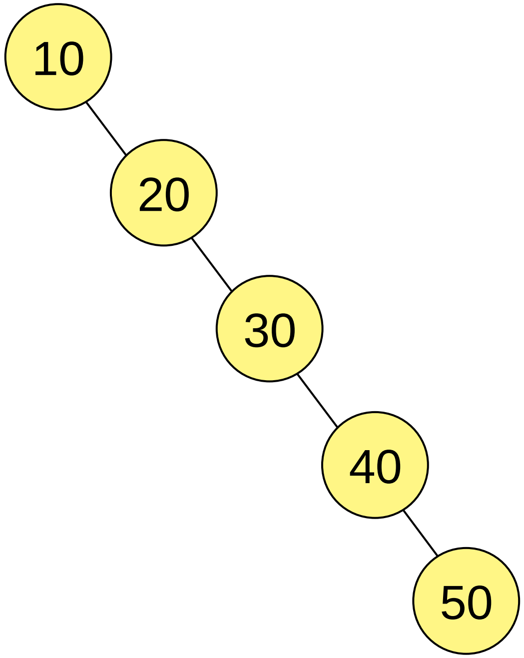
1. Introdução

- Inserindo os nós 30, 20, 40, 10, 25, 35 e 50 nesta ordem, teremos:



1. Introdução

- Inserindo os nós 10, 20, 30, 40 e 50 nesta ordem, teremos:



1. Introdução

- Como observado, existem ordens de inserção de nós que conservam o balanceamento de uma árvore binária.

- Na prática é impossível prever essa ordem ou até alterá-la.

- Solução para o problema de balanceamento:
  - Modificar as operações de inserção e remoção de modo a balancear a árvore a cada nova inserção ou remoção.
  - Garantir que a diferença de alturas das sub-árvores esquerda e direita de cada nó seja de no máximo uma unidade.

1. Introdução

- A vantagem de uma árvore balanceada com relação a uma degenerada está em sua eficiência.

  - Por exemplo: numa árvore binária degenerada de 1.000.000 nós são necessárias, em média, 500.000 comparações (semelhança com arrays ordenados e listas encadeadas).

  - Numa árvore balanceada com o mesmo número de nós essa média reduz-se a 20 comparações.

2. Árvores AVL

- **Árvores de altura balanceada** ou de altura equilibrada foram introduzidas em 1962 por Adelson-Velskii e Landis, também conhecidas como árvores AVL.

- Devido ao balanceamento da árvore, as operações de busca, inserção e remoção em uma árvore com **n** elementos podem ser efetuadas em **$O(\log_2 n)$**, mesmo no pior caso.

2. Árvores AVL

- Uma árvore AVL é definida como:
  - Uma árvore vazia é uma árvore AVL.
  - Sendo **T** uma árvore binária de busca cujas sub-árvores esquerda e direita são **L** e **R**, respectivamente, **T** será uma árvore AVL contanto que:
    - **L** e **R** sejam árvores AVL.
    - $|h_L - h_R| \leq 1$, onde h_L e h_R são as alturas das sub-árvores **L** e **R**, respectivamente.

- A definição de uma árvore binária de altura equilibrada (AVL) requer que cada sub-árvore seja também de altura equilibrada.

2.1. AVL – Implementação

```
typedef struct {
  long chave;
  /* outros componentes */
}TRegistro;

typedef struct TNo_Est {
  TRegistro reg;
  struct TNo_Est * pEsq, pDir;
}TNo;

typedef TNo* TipoDicionario;
```

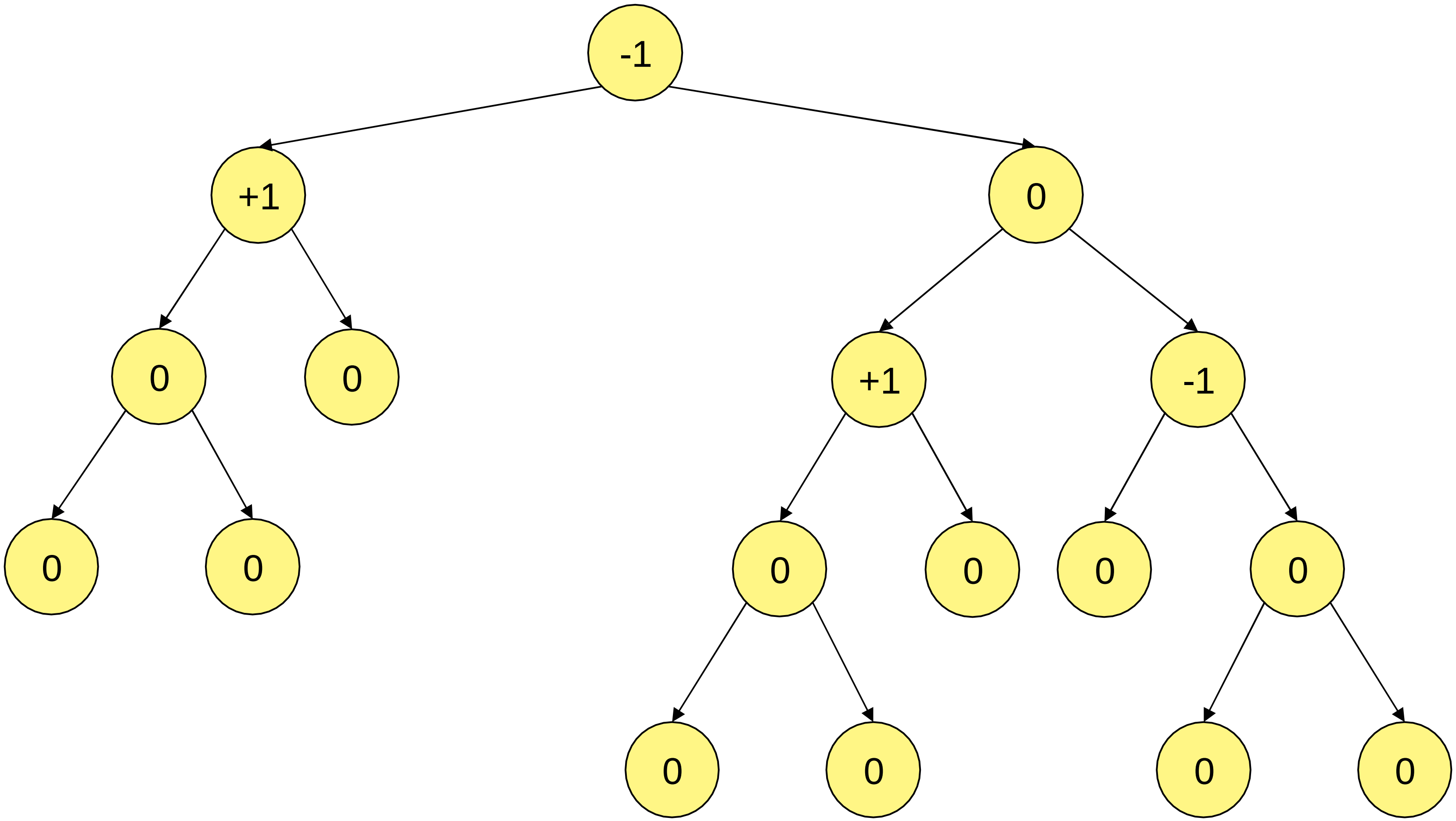
2.2. Fator de Balanceamento

- O **fator de balanceamento (fb)** ou fator de equilíbrio de um nó **T** em uma árvore binária é definido como sendo $\mathbf{h_L - h_R}$ onde $\mathbf{h_L}$ e $\mathbf{h_R}$ são as alturas das sub-árvores esquerda e direita de **T**, respectivamente.

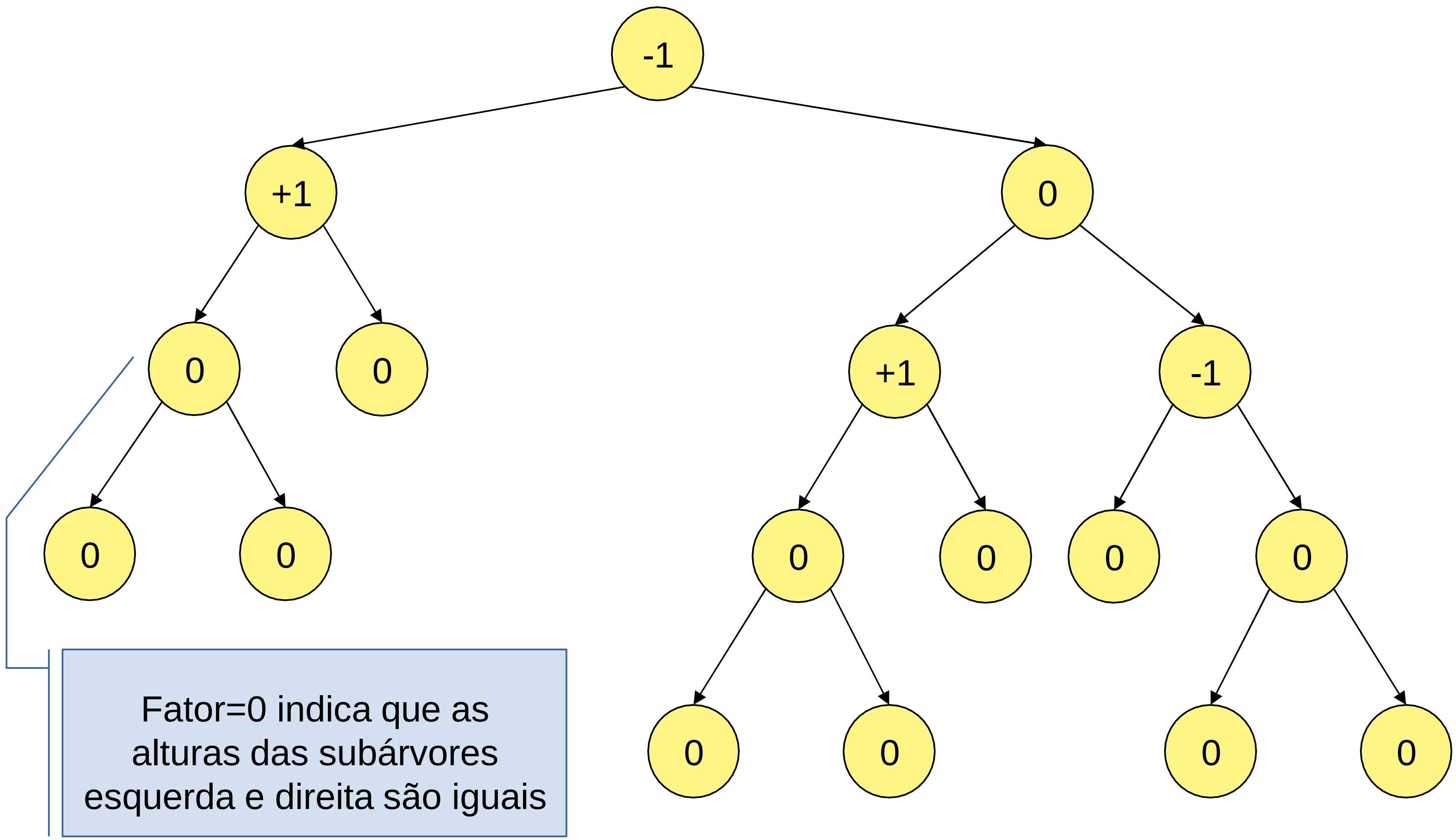
- Para qualquer nó **T** numa árvore AVL, o fator de balanceamento assume o valor **-1**, **0** ou **+1**.

  - O fator de balanceamento de uma folha é zero.

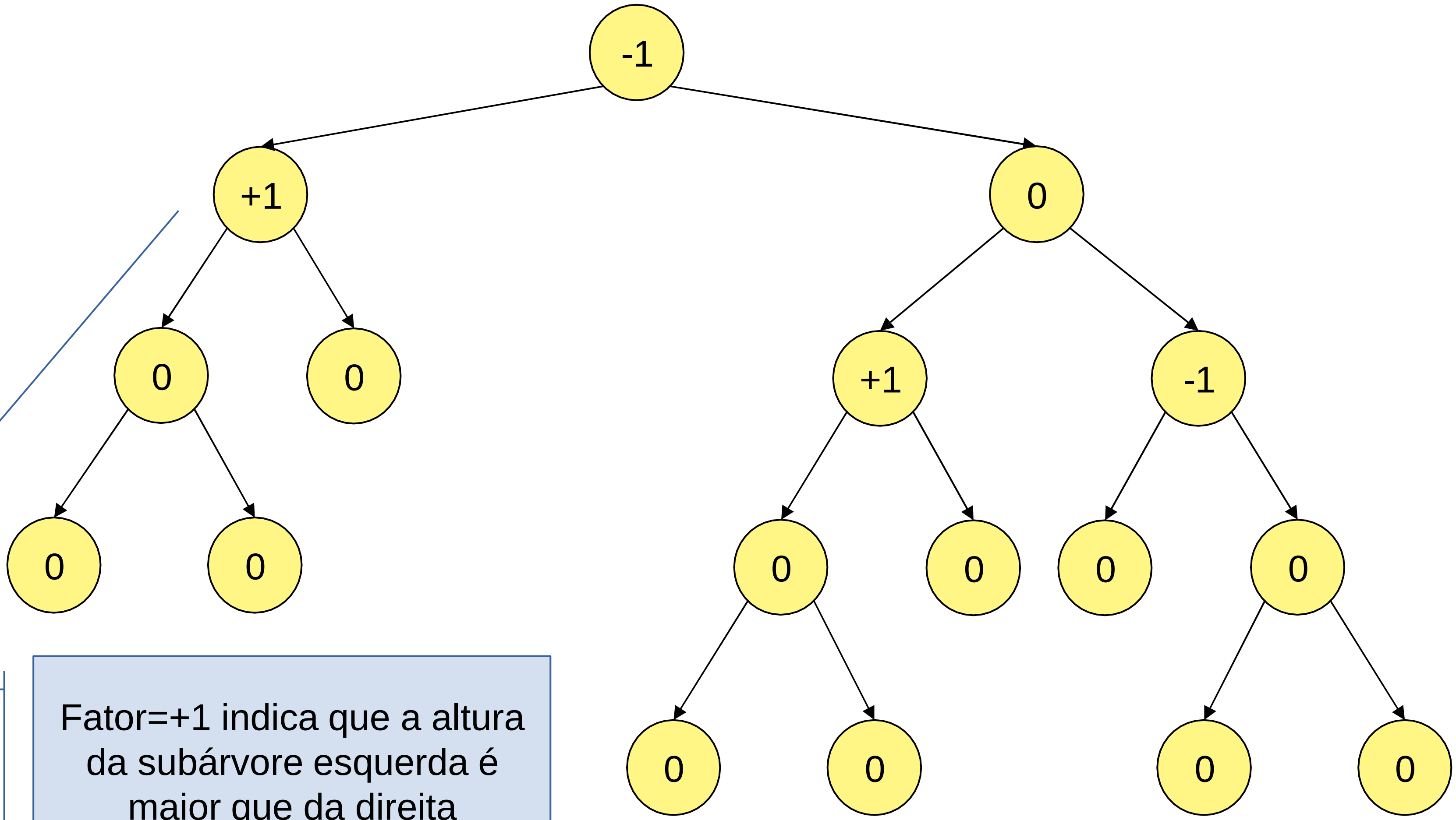
2.2. Fator de Balanceamento



2.2. Fator de Balanceamento

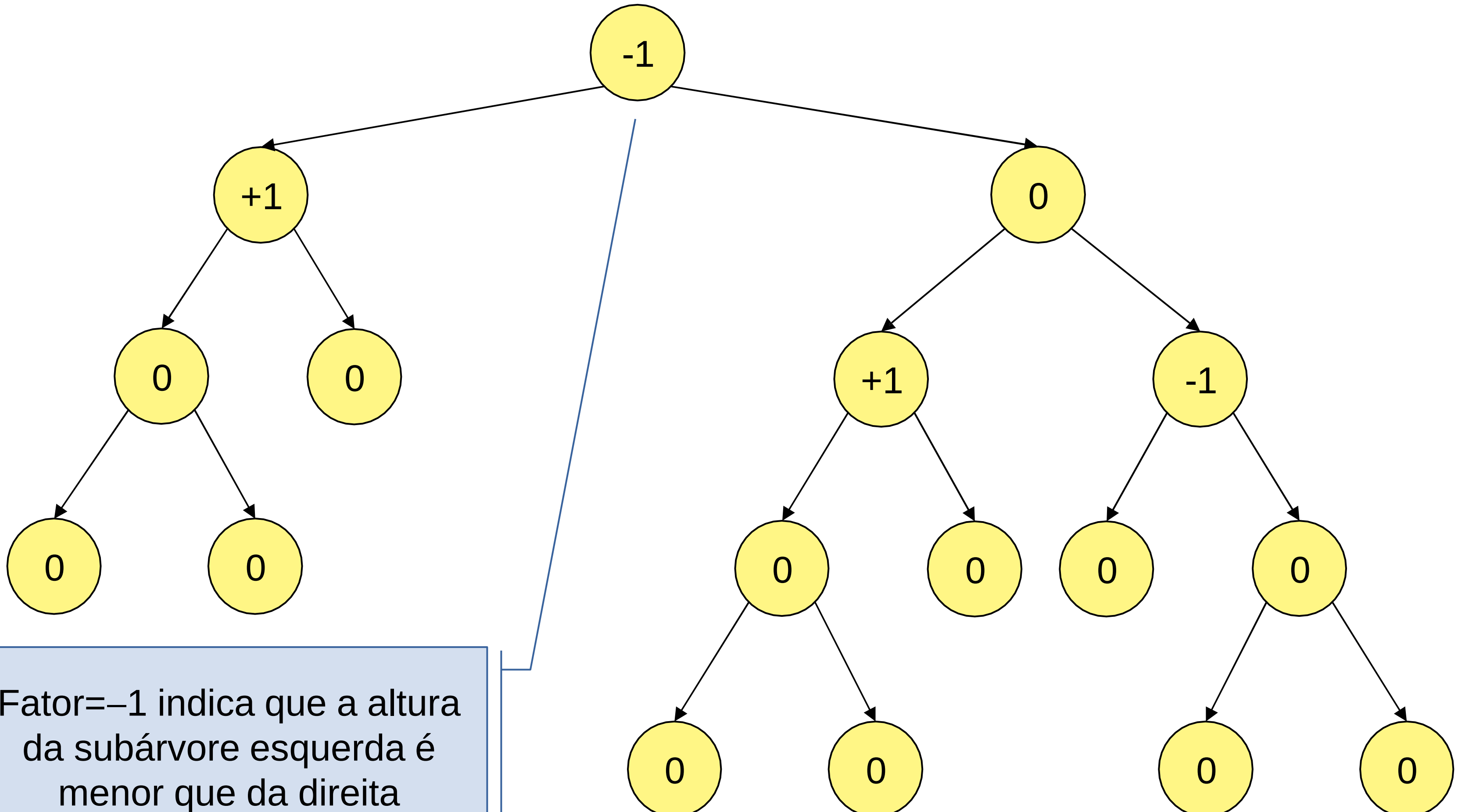


2.2. Fator de Balanceamento



Fator=+1 indica que a altura da subárvore esquerda é maior que da direita

2.2. Fator de Balanceamento



Fator=–1 indica que a altura da subárvore esquerda é menor que da direita

2.2.1. FB – Implementação

```c
int altura (TNo* pRaiz) {
  int hEsq, hDir;
  if (pRaiz == NULL)
    return 0;

  hEsq = altura (pRaiz->pEsq);
  hDir = altura (pRaiz->pDir);

  if (hEsq > hDir)
    return hEsq+1;

  else
    return hDir+1;
}
```

```c
int fb (TNo* pRaiz) {

  if (pRaiz == NULL)
    return 0;

  return altura (pRaiz->pEsq)
       - altura(pRaiz->pDir);
}
```

3. Rotação e Balanceamento

- Inicialmente a inserção e remoção de nós na árvore é feita normalmente.
  - Tais ações podem degenerar (desbalancear) a árvore.

- A restauração do balanceamento é feita através de **rotações** na árvore no nó "**pivô**".

  - Nó "pivô" é aquele que após a inserção ou remoção possui fator de balanceamento fora do intervalo.

3. Rotação e Balanceamento

- Usam-se rotações **simples** ou **duplas** na etapa de rebalanceamento.
  - Executadas a cada inserção ou remoção.
  - As rotações buscam manter a árvore binária como uma árvore quase completa.
  - Custo máximo de qualquer algoritmo é **O(log N)**.

3. Rotação e Balanceamento

- As rotações diferem entre si pelo sentido da inclinação entre o nó pai e filho.

  - **Rotação simples**
    - O nó desbalanceado (pai), seu filho e o seu neto estão todos no mesmo sentido de inclinação.

  - **Rotação dupla**
    - O nó desbalanceado (pai) e seu filho estão inclinados no sentido inverso ao neto.
    - Equivale a **duas rotações simples**.

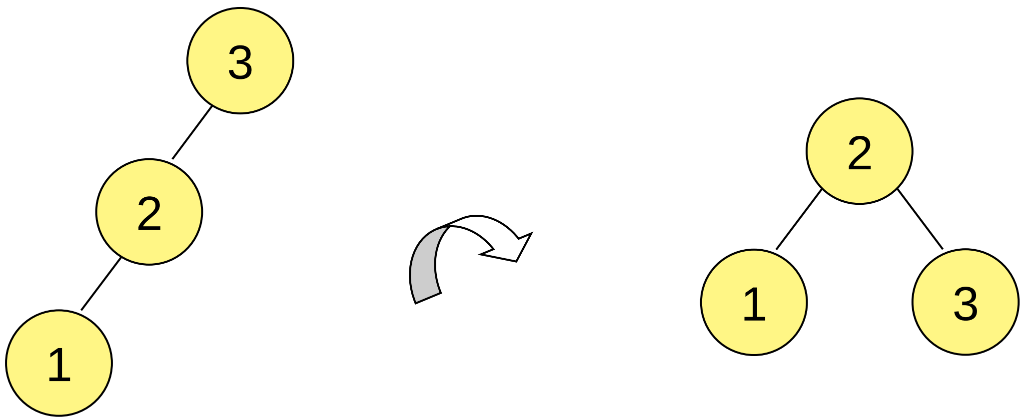
3. Rotação e Balanceamento

- Ao todo, existem duas rotações simples e duas duplas:

1. Rotação simples a direita ou **Rotação LL**
2. Rotação simples a esquerda ou **Rotação RR**
3. Rotação dupla a direita ou **Rotação LR**
4. Rotação dupla a esquerda ou **Rotação RL**

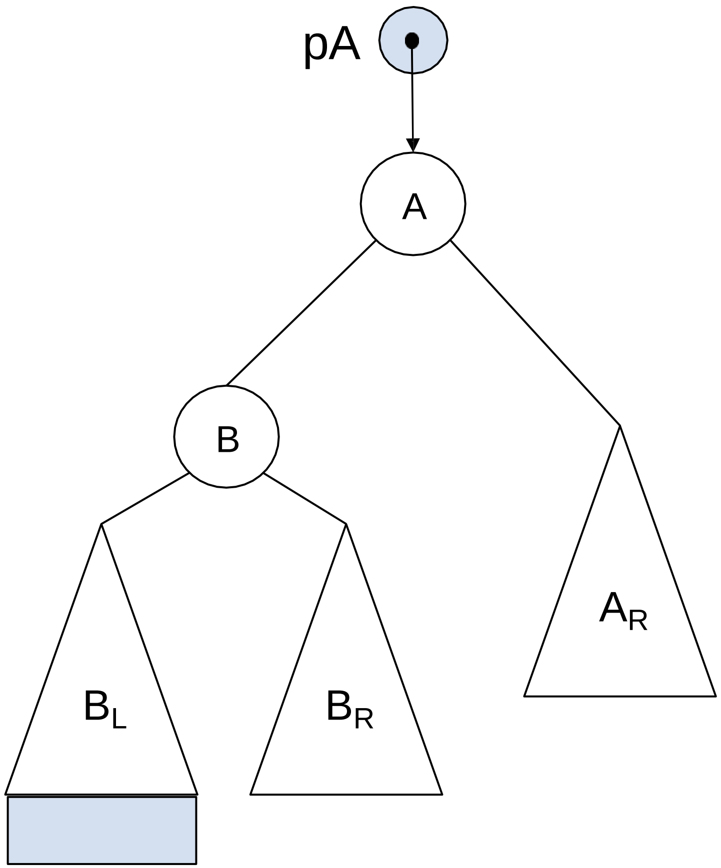
3.1. Rotação LL

- FB > 1
  - Sub-árvore esquerda maior que sub-árvore direita.
  - E a sub-árvore esquerda desta sub-árvore esquerda é maior que a sub-árvore direita dela.
  - Então realizar uma rotação simples para a direita.



3.1. Rotação LL

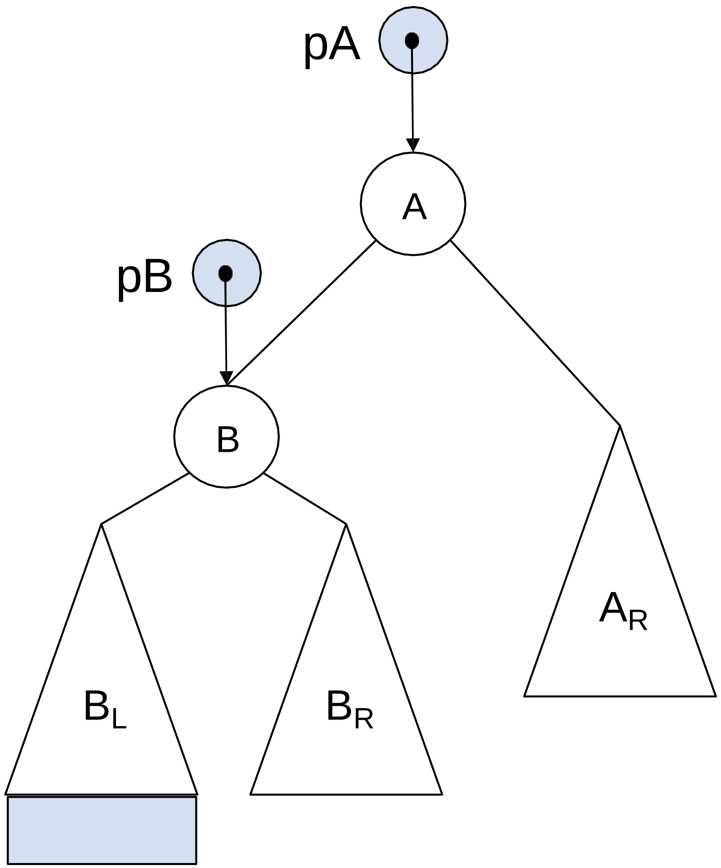
- Implementação: passo a passo




```
void LL (TNo** pA) {
   TNo *pB;
   pB = (*pA)->pEsq;
   (*pA)->pEsq = pB->pDir;
   pB->pDir = (*pA);
   (*pA) = pB;
}
```

3.1. Rotação LL

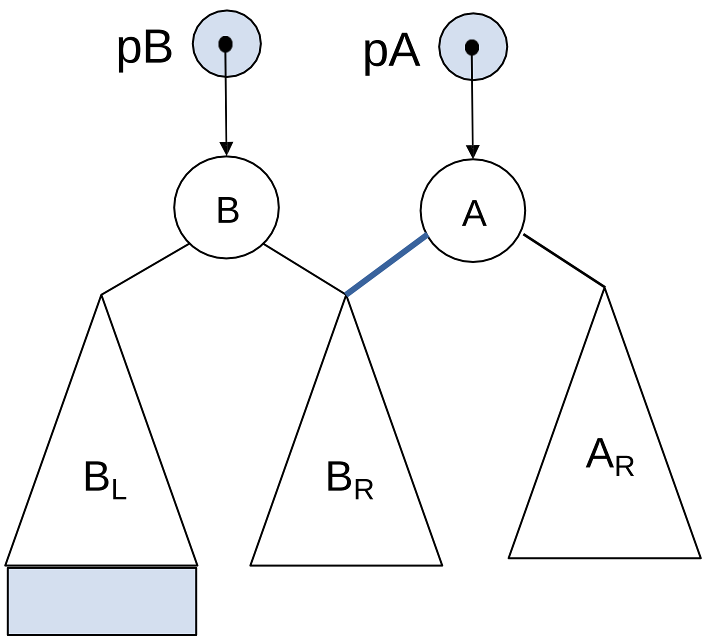
- Implementação: passo a passo




```c
void LL (TNo** pA) {
  TNo *pB;
  pB = (*pA)->pEsq;
  (*pA)->pEsq = pB->pDir;
  pB->pDir = (*pA);
  (*pA) = pB;
}
```

3.1. Rotação LL

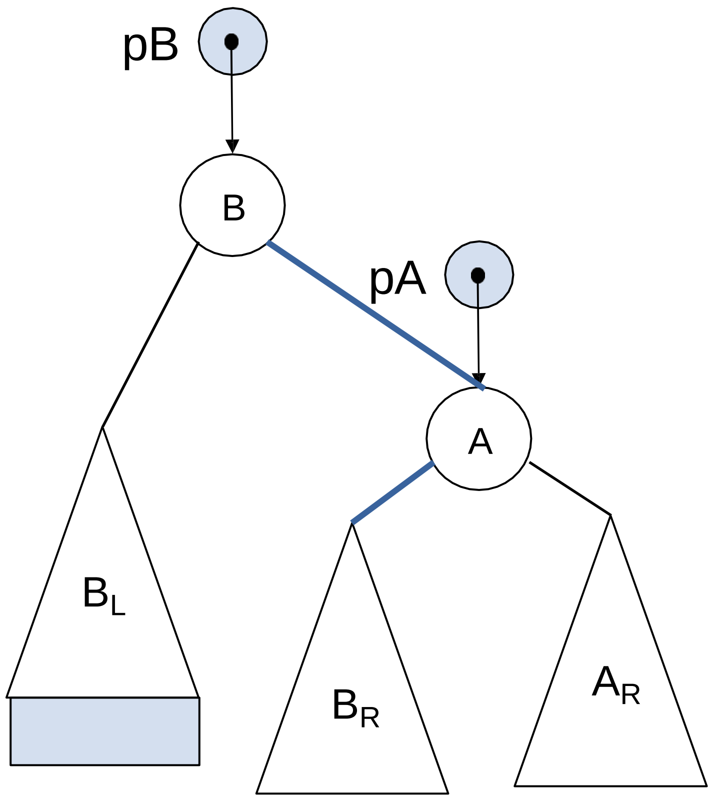
- Implementação: passo a passo




```
void LL (TNo** pA) {
   TNo *pB;
   pB = (*pA)->pEsq;
   (*pA)->pEsq = pB->pDir;
   pB->pDir = (*pA);
   (*pA) = pB;
}
```

3.1. Rotação LL

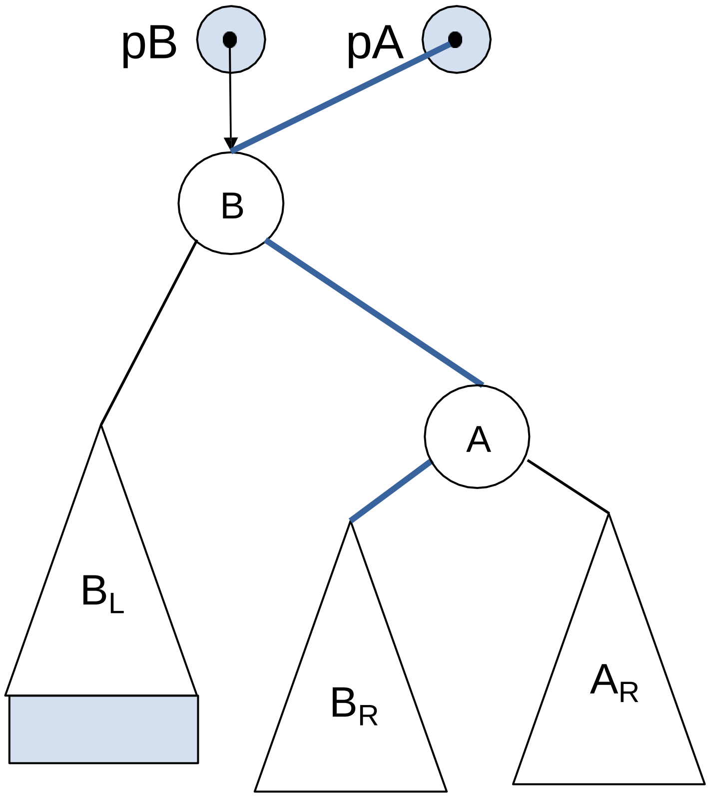
- Implementação: passo a passo




```
void LL (TNo** pA) {
   TNo *pB;
   pB = (*pA)->pEsq;
   (*pA)->pEsq = pB->pDir;
   pB->pDir = (*pA);
   (*pA) = pB;
}
```

3.1. Rotação LL

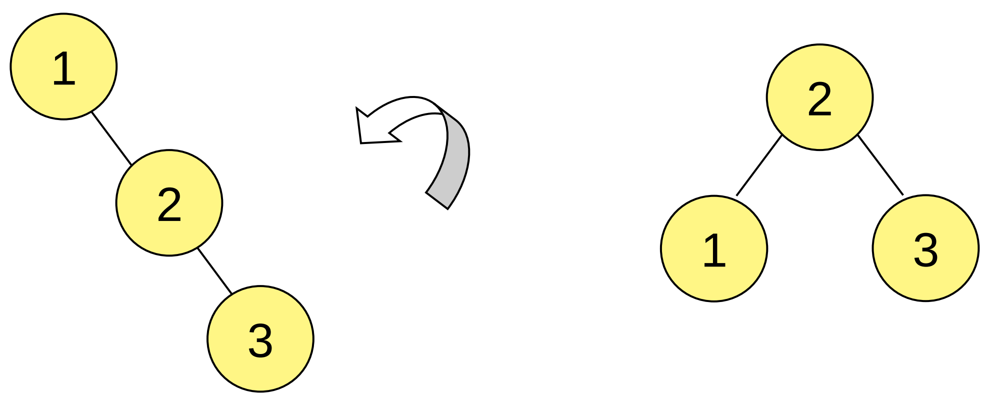
- Implementação: passo a passo




```c
void LL (TNo** pA) {
   TNo *pB;
   pB = (*pA)->pEsq;
   (*pA)->pEsq = pB->pDir;
   pB->pDir = (*pA);
   (*pA) = pB;
}
```

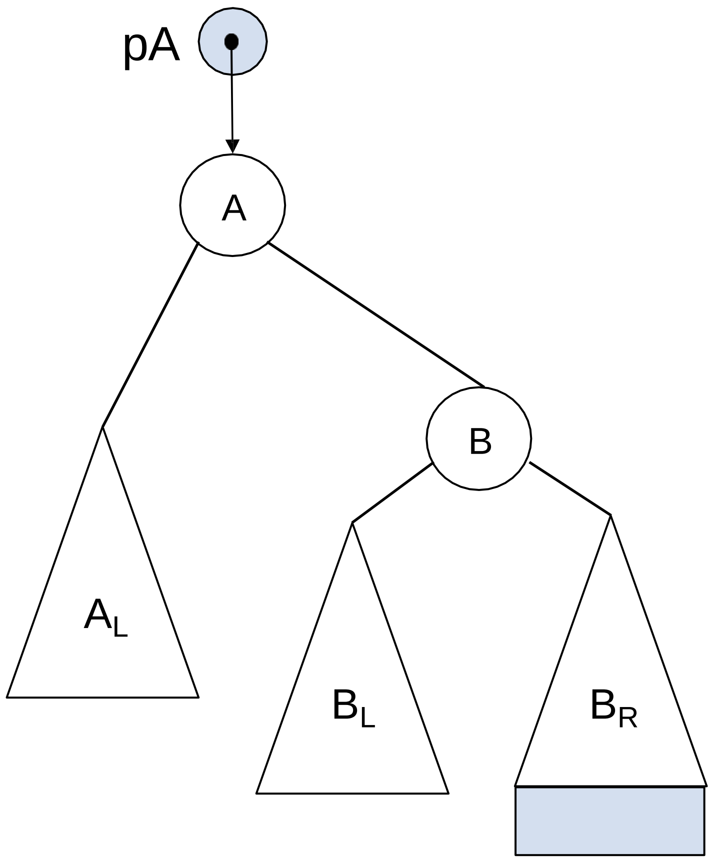
3.2. Rotação RR

- FB < -1
  - Sub-árvore esquerda menor que sub-árvore direita.
  - E a sub-árvore direita desta sub-árvore direita é maior que a sub-árvore esquerda dela.
  - Então realizar uma rotação simples para a esquerda.



3.2. Rotação RR

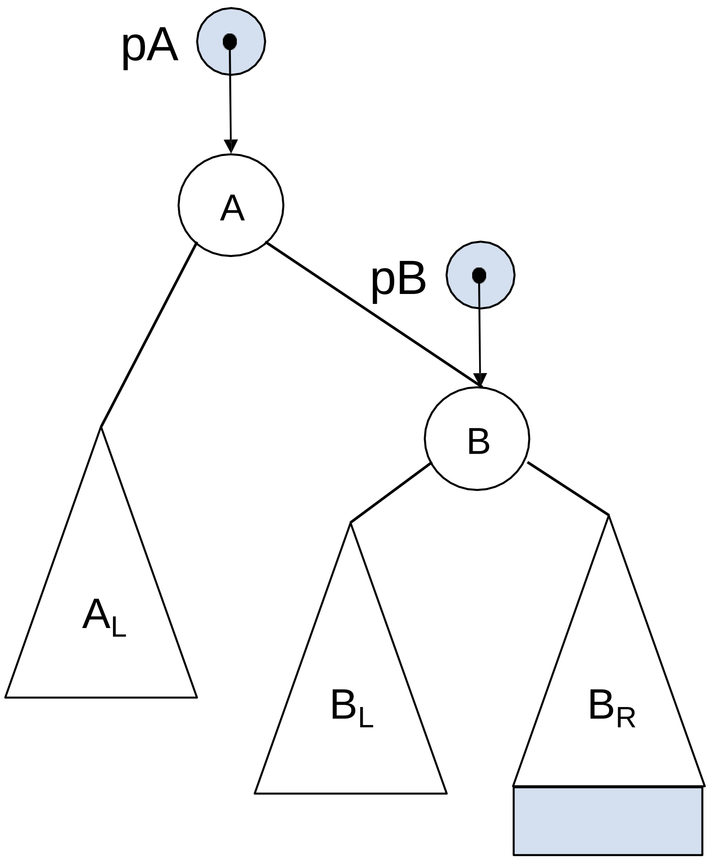
- Implementação: passo a passo




```
void RR (TNo** pA) {
  TNo *pB;
  pB = (*pA)->pDir;
  (*pA)->pDir = pB->pEsq;
  pB->pEsq = (*pA);
  (*pA) = pB;
}
```

3.2. Rotação RR

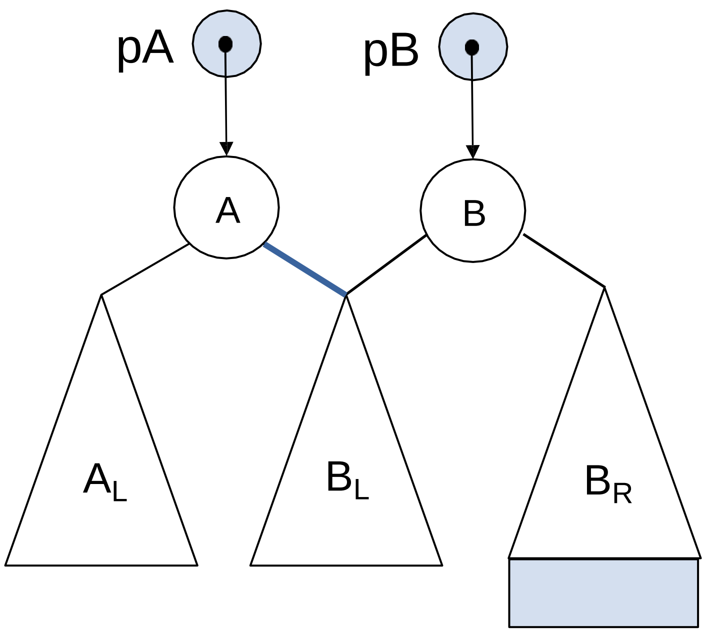
- Implementação: passo a passo




```
void RR (TNo** pA) {
   TNo *pB;
   pB = (*pA)->pDir;
   (*pA)->pDir = pB->pEsq;
   pB->pEsq = (*pA);
   (*pA) = pB;
}
```

3.2. Rotação RR

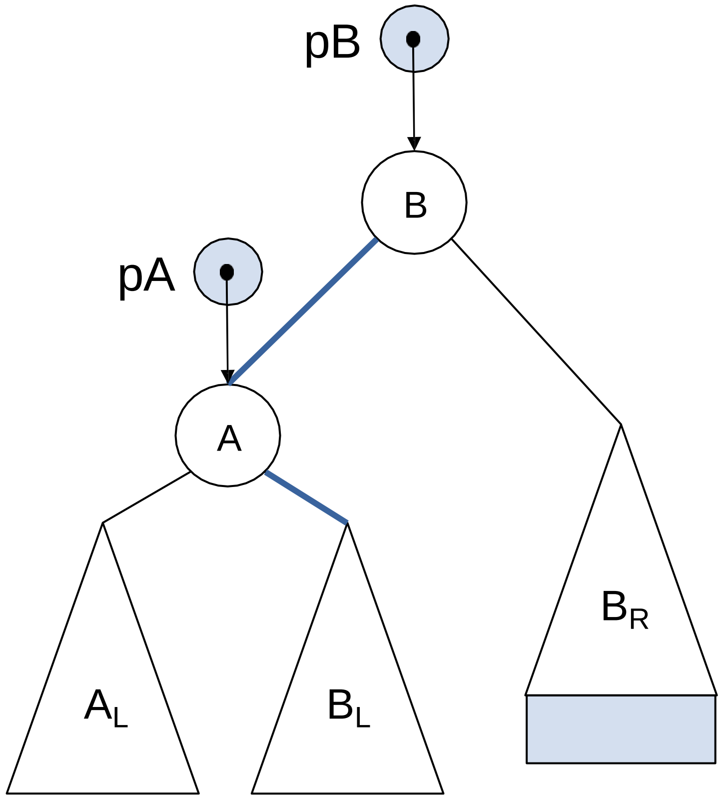
- Implementação: passo a passo




```c
void RR (TNo** pA) {
   TNo *pB;
   pB = (*pA)->pDir;
   (*pA)->pDir = pB->pEsq;
   pB->pEsq = (*pA);
   (*pA) = pB;
}
```

3.2. Rotação RR

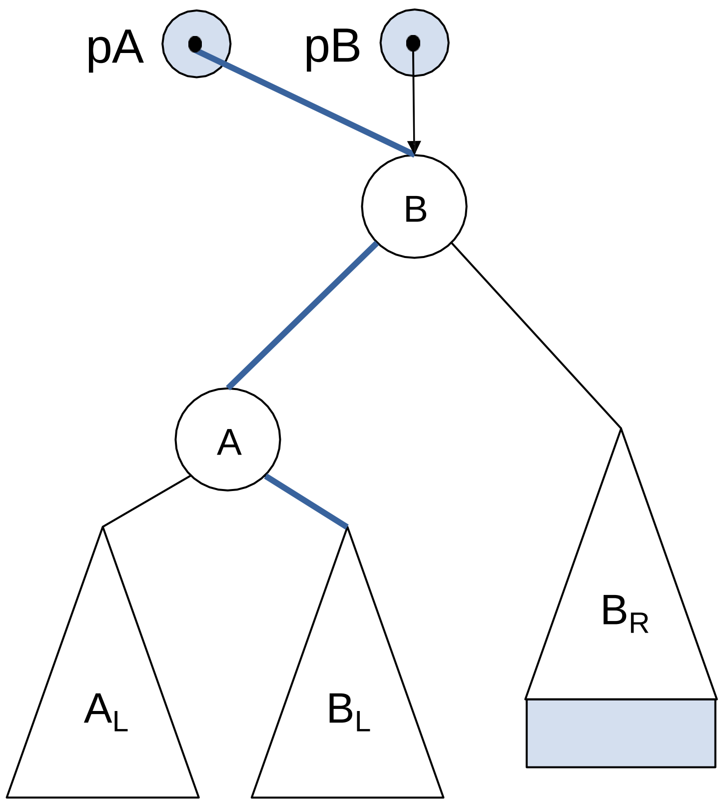
- Implementação: passo a passo




```
void RR (TNo** pA) {
   TNo *pB;
   pB = (*pA)->pDir;
   (*pA)->pDir = pB->pEsq;
   pB->pEsq = (*pA);
   (*pA) = pB;
}
```

3.2. Rotação RR

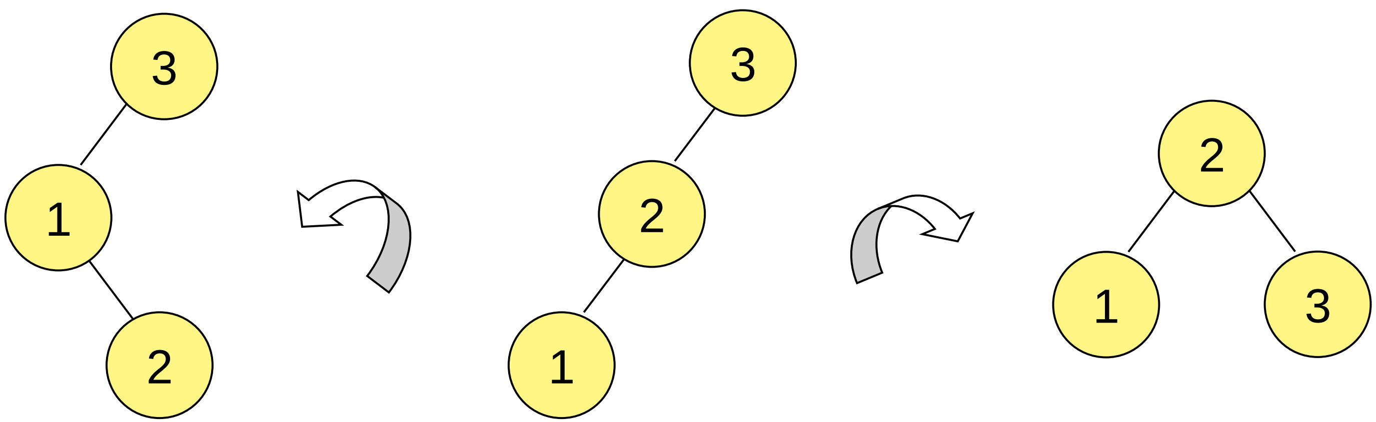
- Implementação: passo a passo




```
void RR (TNo** pA) {
  TNo *pB;
  pB = (*pA)->pDir;
  (*pA)->pDir = pB->pEsq;
  pB->pEsq = (*pA);
  (*pA) = pB;
}
```

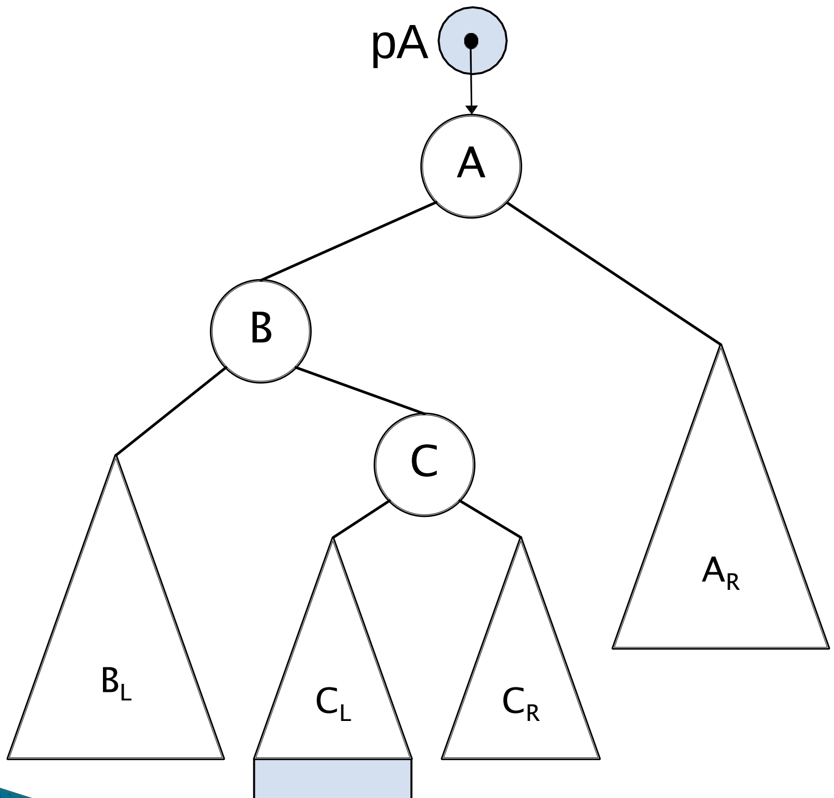
3.3. Rotação LR

- FB > 1
  - Sub-árvore esquerda maior que sub-árvore direita.
  - E a sub-árvore esquerda desta sub-árvore esquerda é menor ou igual que a sub-árvore direita dela.
  - Então realizar uma rotação dupla para a direita.



3.3. Rotação LR

- Implementação: passo a passo

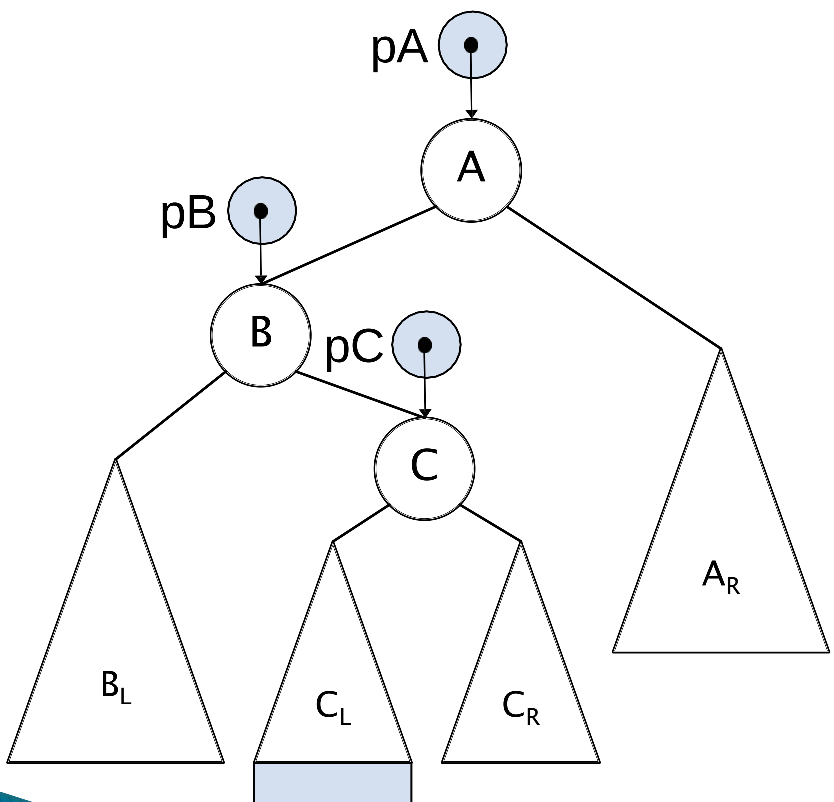



```
void LR (TNo** pA) {

  TNo *pB, *pC;
  pB = (*pA)->pEsq;
  pC = pB->pDir;
  pB->pDir = pC->pEsq;
  pC->pEsq = pB;
  (*pA)->pEsq = pC->pDir;
  pC->pDir = (*pA);
  (*pA) = pC;
}
```

3.3. Rotação LR

- Implementação: passo a passo

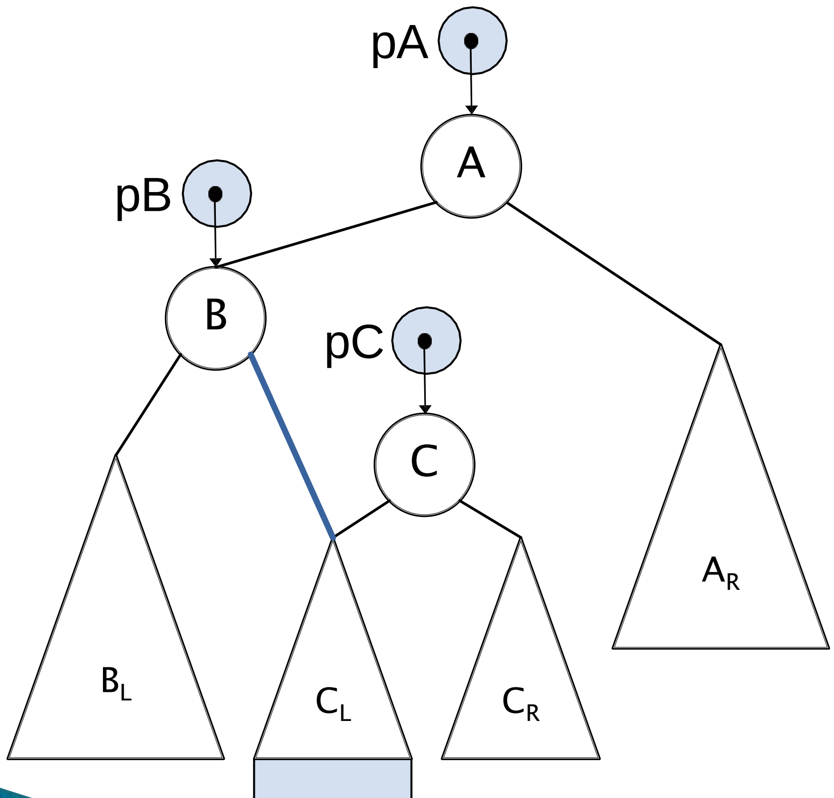



```
void LR (TNo** pA) {

   TNo *pB, *pC;
   pB = (*pA)->pEsq;
   pC = pB->pDir;
   pB->pDir = pC->pEsq;
   pC->pEsq = pB;
   (*pA)->pEsq = pC->pDir;
   pC->pDir = (*pA);
   (*pA) = pC;
}
```

3.3. Rotação LR

- Implementação: passo a passo

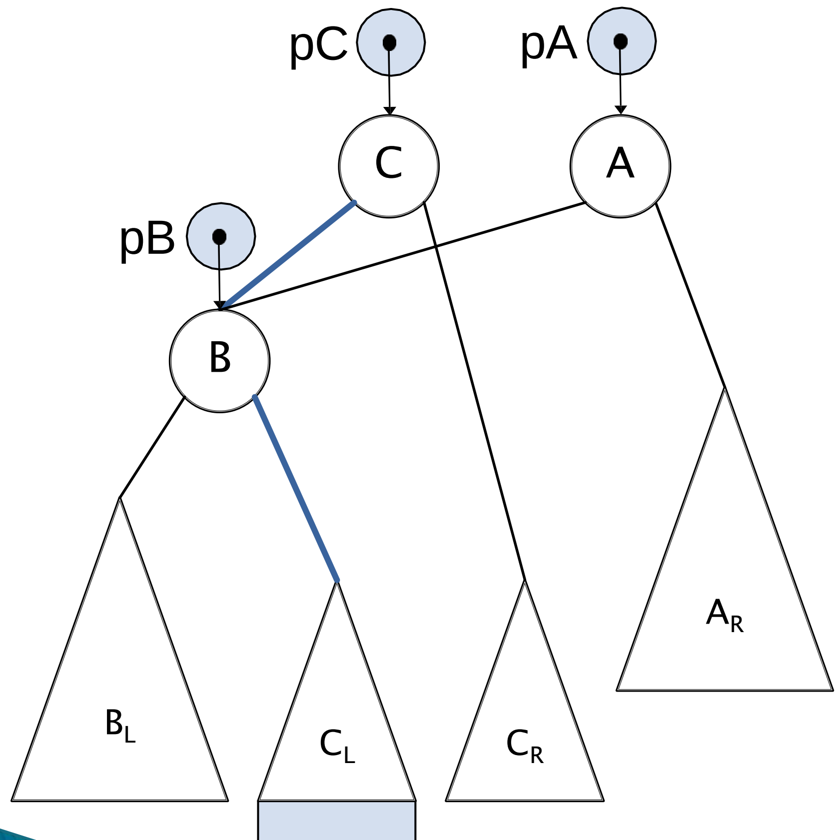



```
void LR (TNo** pA) {

   TNo *pB, *pC;
   pB = (*pA)->pEsq;
   pC = pB->pDir;
   pB->pDir = pC->pEsq;
   pC->pEsq = pB;
   (*pA)->pEsq = pC->pDir;
   pC->pDir = (*pA);
   (*pA) = pC;
}
```

3.3. Rotação LR

- Implementação: passo a passo

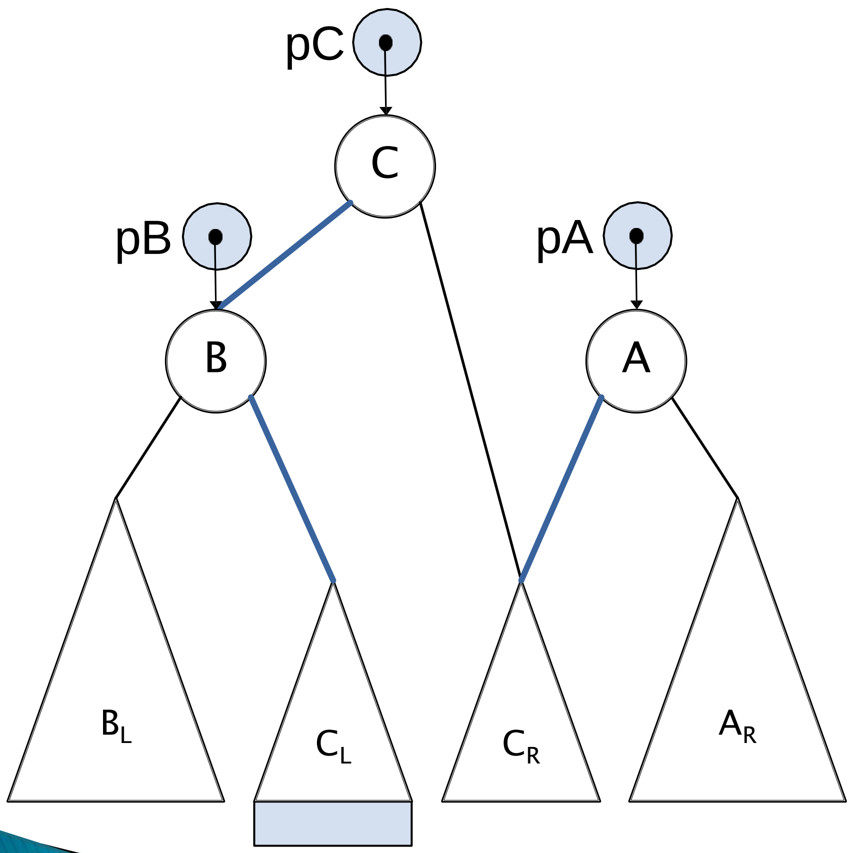



```
void LR (TNo** pA) {

   TNo *pB, *pC;
   pB = (*pA)->pEsq;
   pC = pB->pDir;
   pB->pDir = pC->pEsq;
   pC->pEsq = pB;
   (*pA)->pEsq = pC->pDir;
   pC->pDir = (*pA);
   (*pA) = pC;
}
```

3.3. Rotação LR

- Implementação: passo a passo

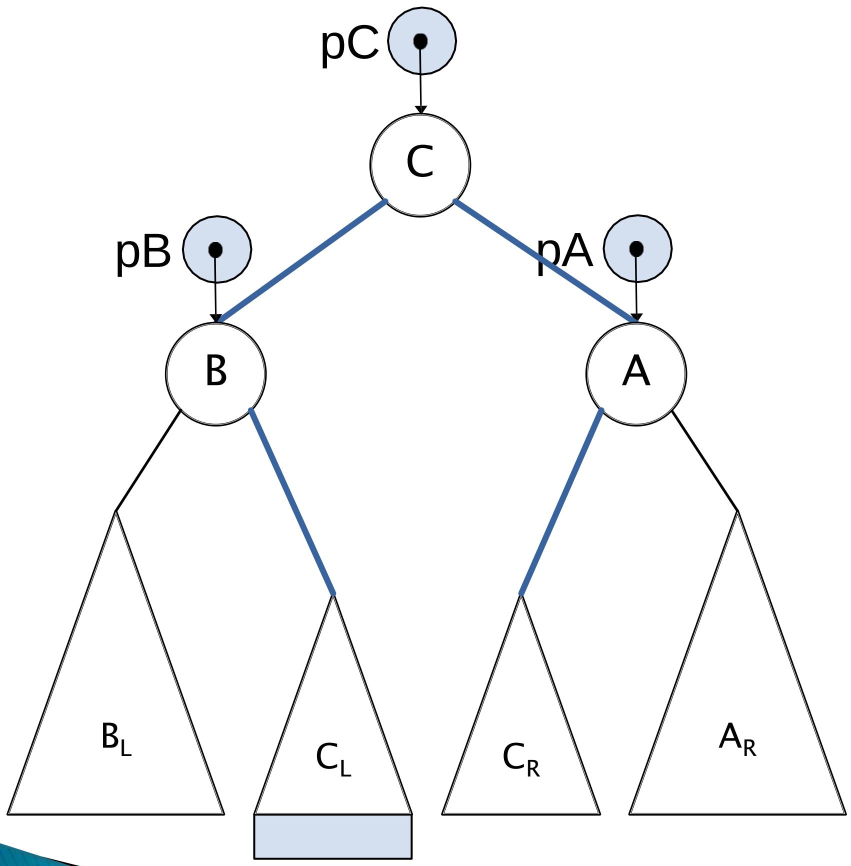



```
void LR (TNo** pA) {

   TNo *pB, *pC;
   pB = (*pA)->pEsq;
   pC = pB->pDir;
   pB->pDir = pC->pEsq;
   pC->pEsq = pB;
   (*pA)->pEsq = pC->pDir;
   pC->pDir = (*pA);
   (*pA) = pC;
}
```

3.3. Rotação LR

- Implementação: passo a passo

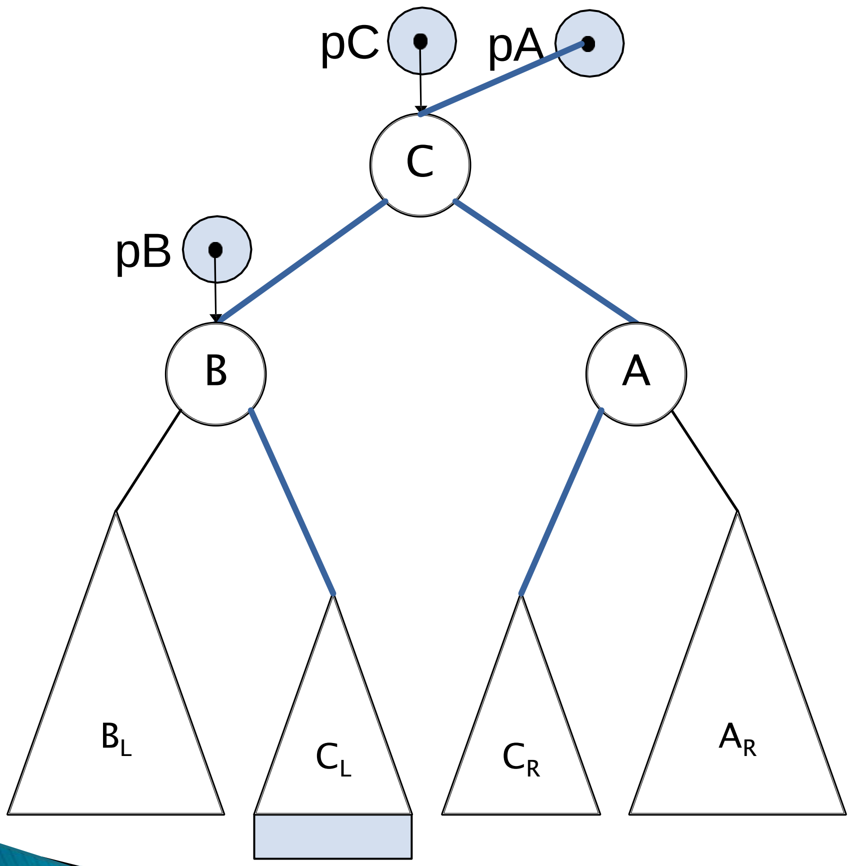



```
void LR (TNo** pA) {

   TNo *pB, *pC;
   pB = (*pA)->pEsq;
   pC = pB->pDir;
   pB->pDir = pC->pEsq;
   pC->pEsq = pB;
   (*pA)->pEsq = pC->pDir;
   pC->pDir = (*pA);
   (*pA) = pC;
}
```

3.3. Rotação LR

- Implementação: passo a passo

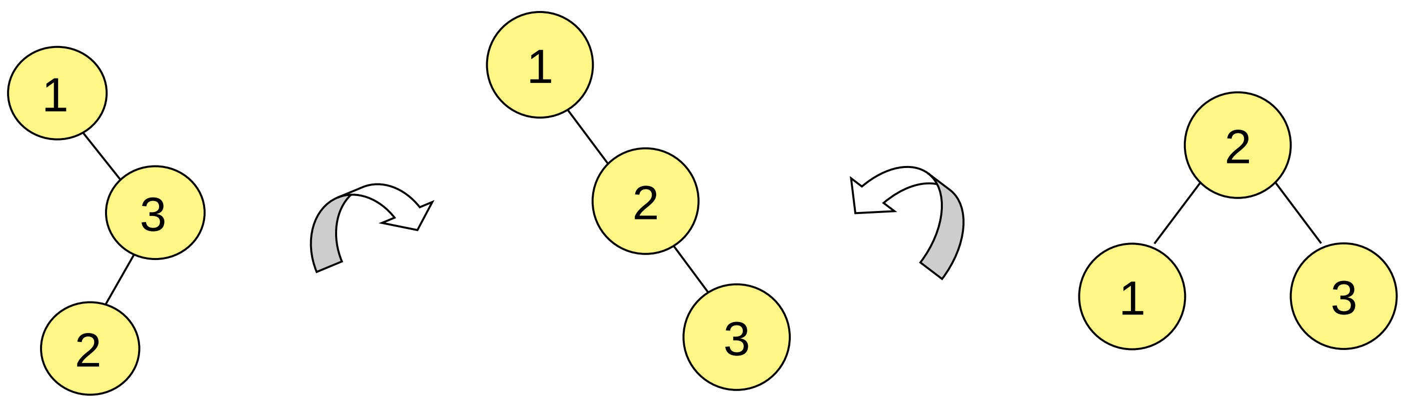



```
void LR (TNo** pA) {

   TNo *pB, *pC;
   pB = (*pA)->pEsq;
   pC = pB->pDir;
   pB->pDir = pC->pEsq;
   pC->pEsq = pB;
   (*pA)->pEsq = pC->pDir;
   pC->pDir = (*pA);
   (*pA) = pC;
}
```

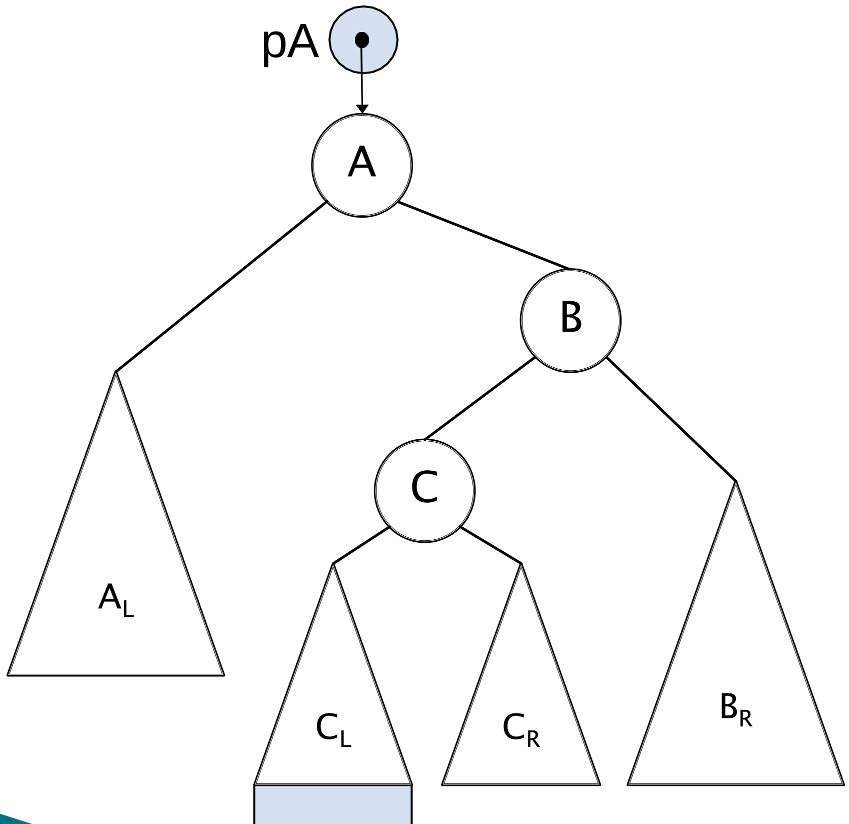
3.4. Rotação RL

- FB < -1
  - Sub-árvore esquerda menor que sub-árvore direita.
  - E a sub-árvore direita desta sub-árvore direita é menor que a sub-árvore esquerda dela.
  - Então realizar uma rotação dupla para a esquerda.



3.4. Rotação RL

- Implementação: passo a passo

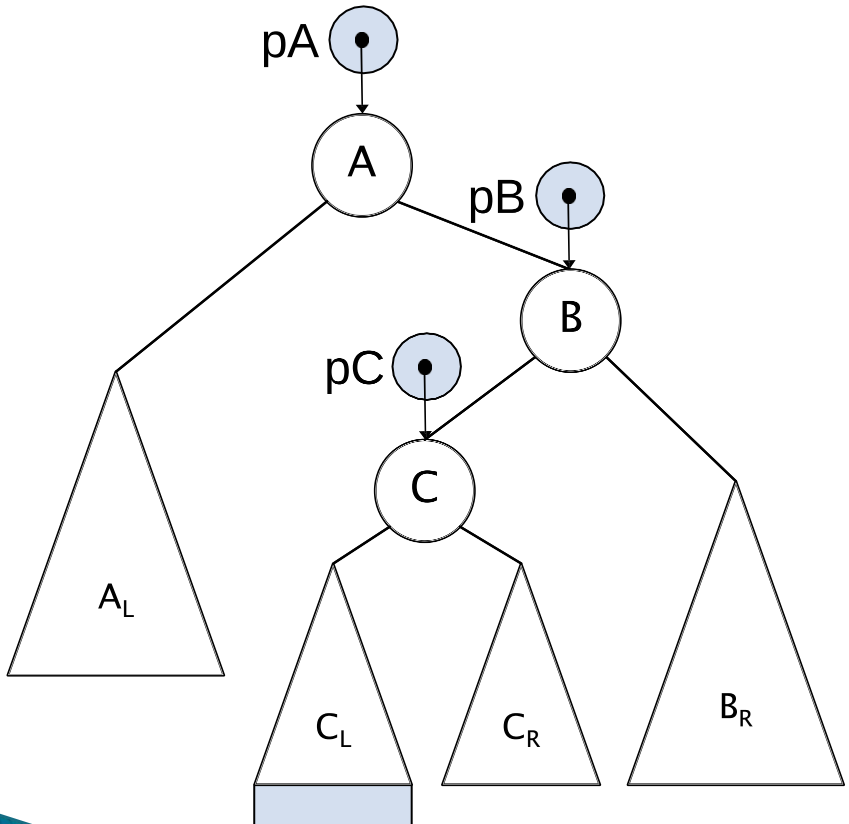



```c
void RL (TNo** pA) {

   TNo *pB, *pC;
   pB = (*pA)->pDir;
   pC = pB->pEsq;
   pB->pEsq = pC->pDir;
   pC->pDir = pB;
   (*pA)->pDir = pC->pEsq;
   pC->pEsq = (*pA);
   (*pA) = pC;
}
```

3.4. Rotação RL

- Implementação: passo a passo

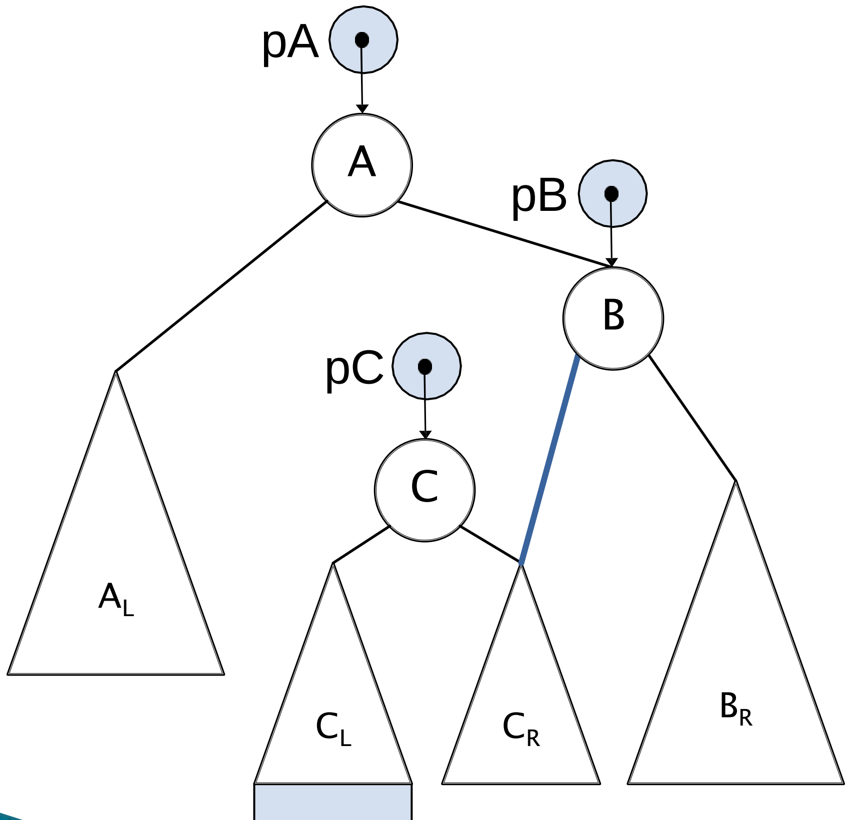



```
void RL (TNo** pA) {

   TNo *pB, *pC;
   pB = (*pA)->pDir;
   pC = pB->pEsq;
   pB->pEsq = pC->pDir;
   pC->pDir = pB;
   (*pA)->pDir = pC->pEsq;
   pC->pEsq = (*pA);
   (*pA) = pC;
}
```

3.4. Rotação RL

- Implementação: passo a passo

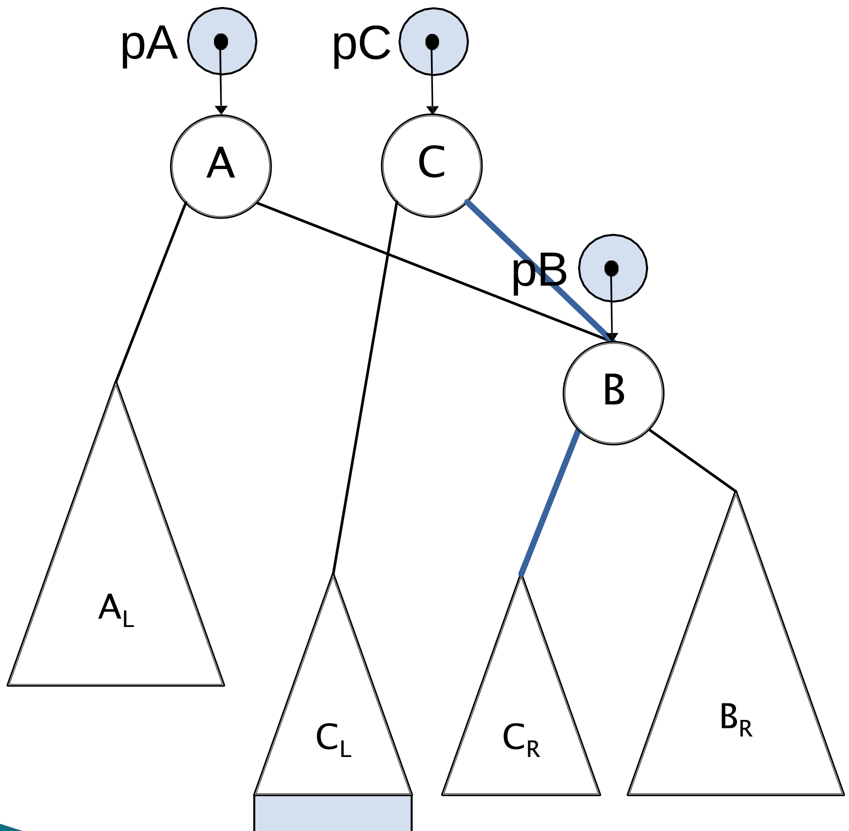



```
void RL (TNo** pA) {

   TNo *pB, *pC;
   pB = (*pA)->pDir;
   pC = pB->pEsq;
   pB->pEsq = pC->pDir;
   pC->pDir = pB;
   (*pA)->pDir = pC->pEsq;
   pC->pEsq = (*pA);
   (*pA) = pC;
}
```

3.4. Rotação RL

- Implementação: passo a passo

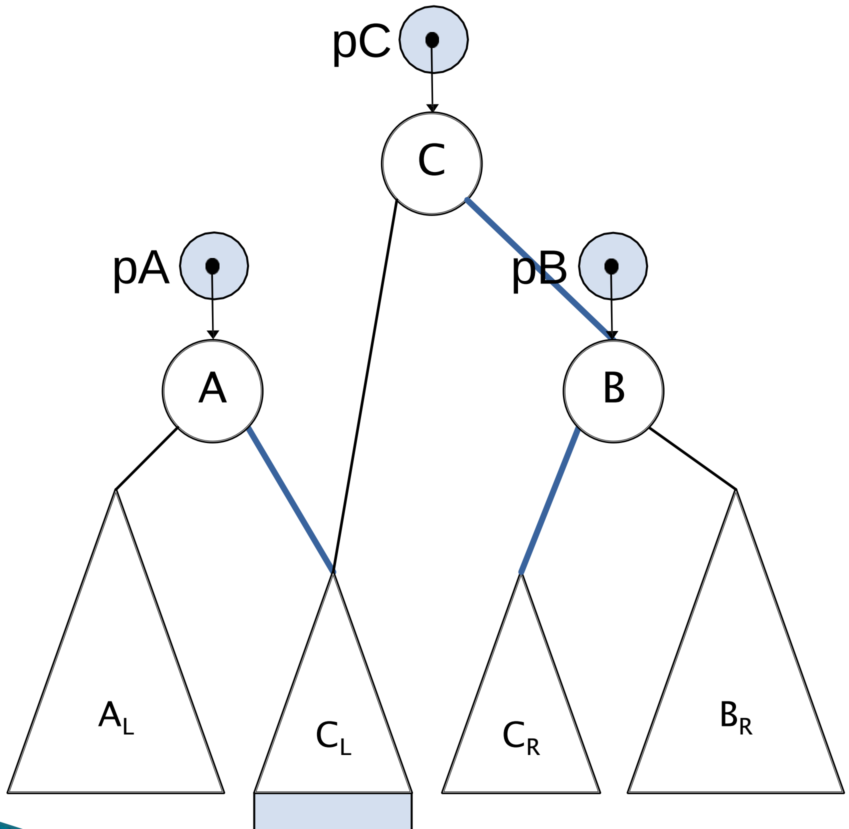



```
void RL (TNo** pA) {

   TNo *pB, *pC;
   pB = (*pA)->pDir;
   pC = pB->pEsq;
   pB->pEsq = pC->pDir;
   pC->pDir = pB;
   (*pA)->pDir = pC->pEsq;
   pC->pEsq = (*pA);
   (*pA) = pC;
}
```

3.4. Rotação RL

- Implementação: passo a passo

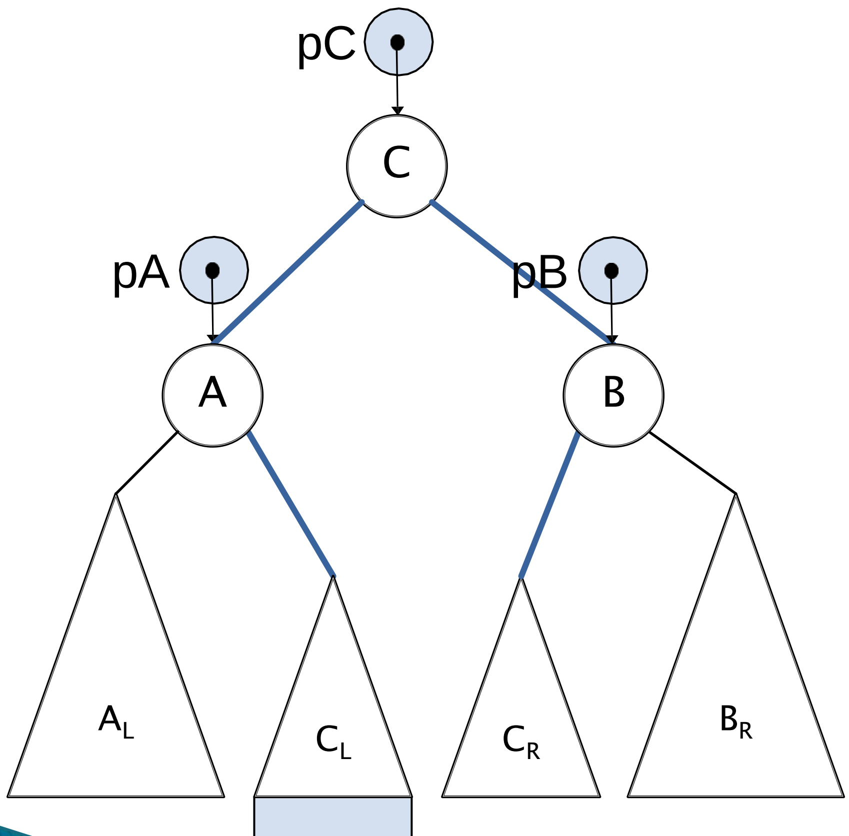



```
void RL (TNo** pA) {

   TNo *pB, *pC;
   pB = (*pA)->pDir;
   pC = pB->pEsq;
   pB->pEsq = pC->pDir;
   pC->pDir = pB;
   (*pA)->pDir = pC->pEsq;
   pC->pEsq = (*pA);
   (*pA) = pC;
}
```

3.4. Rotação RL

- Implementação: passo a passo

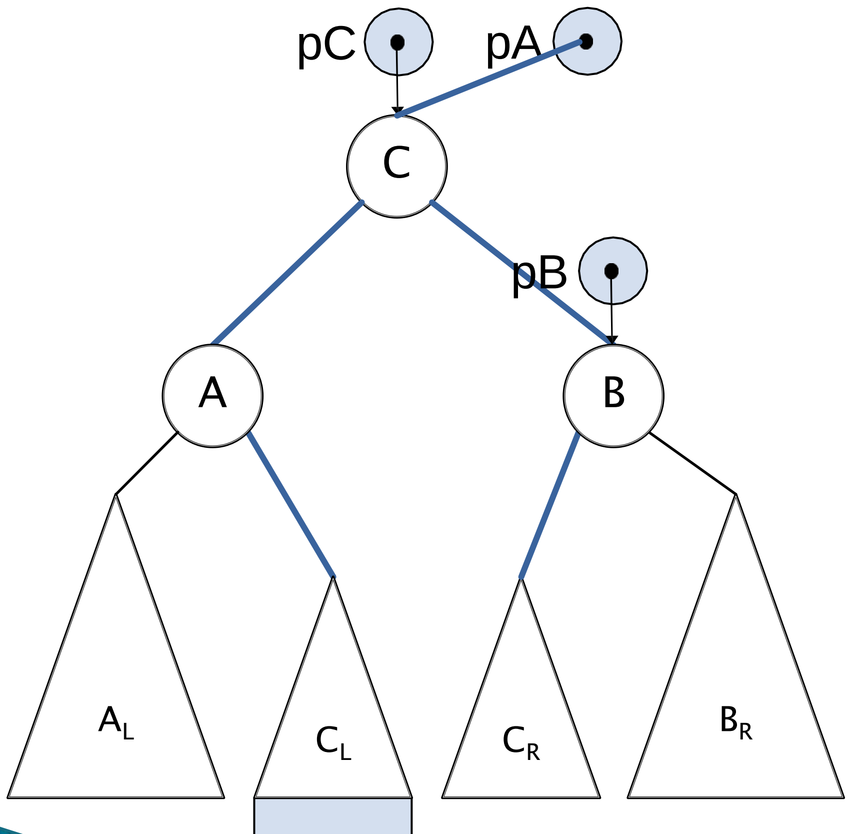



```
void RL (TNo** pA) {

   TNo *pB, *pC;
   pB = (*pA)->pDir;
   pC = pB->pEsq;
   pB->pEsq = pC->pDir;
   pC->pDir = pB;
   (*pA)->pDir = pC->pEsq;
   pC->pEsq = (*pA);
   (*pA) = pC;
}
```

3.4. Rotação RL

- Implementação: passo a passo




```
void RL (TNo** pA) {

   TNo *pB, *pC;
   pB = (*pA)->pDir;
   pC = pB->pEsq;
   pB->pEsq = pC->pDir;
   pC->pDir = pB;
   (*pA)->pDir = pC->pEsq;
   pC->pEsq = (*pA);
   (*pA) = pC;
}
```

3.5. Quando usar cada rotação?

Fator de Balenceamento de A	Fator de Balanceamento de B	Posições dos nós B e C em relação ao nó A	Rotação
+2	+1	B é filho à esquerda de A C é filho à esquerda de B	LL
−2	−1	B é filho à direita de A C é filho à direita de B	RR
+2	−1	B é filho à esquerda de A C é filho à direita de B	LR
−2	+1	B é filho à direita de A C é filho à esquerda de B	RL

3.6. Balanceamento

```
int balanceamento (TNo** ppRaiz) {

  int fatb = fb (*ppRaiz);

  if (fatb > 1) { /* Sub-arvore esquerda maior */
    int fbe = fb ((*ppRaiz)->pEsq);

    if (fbe > 0) { /* Rotação LL */
      LL (ppRaiz);
      return 1; }

    else if (fbe < 0) { /* Rotaçao LR */
      LR (ppRaiz);
      return 1; }

    return 0;
  }
```

3.6. Balanceamento

```
  else if (fatb < -1) { /* Sub-arvore direita maior */
    int fbd = fb ((*ppRaiz)->pDir);

    if (fbd < 0) { /* Rotação RR */
      RR (ppRaiz);
      return 1; }

    else if (fbd > 0) { /* Rotação RL */
      RL (ppRaiz);
      return 1; }

    return 0;
  }

  else /* Não necessita balancear */
    return 0;
}
```

4. Inserção

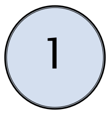
- Para inserir um valor **V** na árvore:
  - Se a raiz é igual a **`NULL`**, insira o nó.
  - Se **V** é <u>menor</u> do que a raiz: vá para a sub-árvore esquerda.
  - Se **V** é <u>maior</u> do que a raiz: vá para a sub-árvore direita.
  - Aplique o método **recursivamente**.
- Dessa forma, percorremos um conjunto de nós da árvore até chegar ao nó **folha** que irá se tornar o **pai** do novo nó.

4. Inserção

- Uma vez inserido o novo nó:
  - É necessário voltar pelo caminho percorrido e calcular o fator de balanceamento de cada um dos nós visitados.
  - Aplicar a rotação necessária para restabelecer o **balanceamento** da árvore se o fator de balanceamento for +2 ou -2.

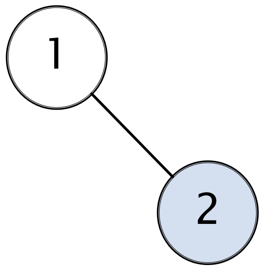
4. Inserção

- Passo a passo – Inserir: **1**



4. Inserção

- Passo a passo – Inserir: **2**

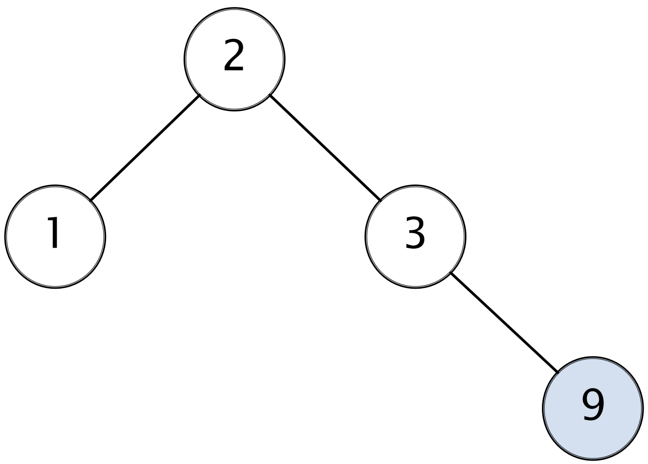


4. Inserção

- Passo a passo – Inserir: **3**

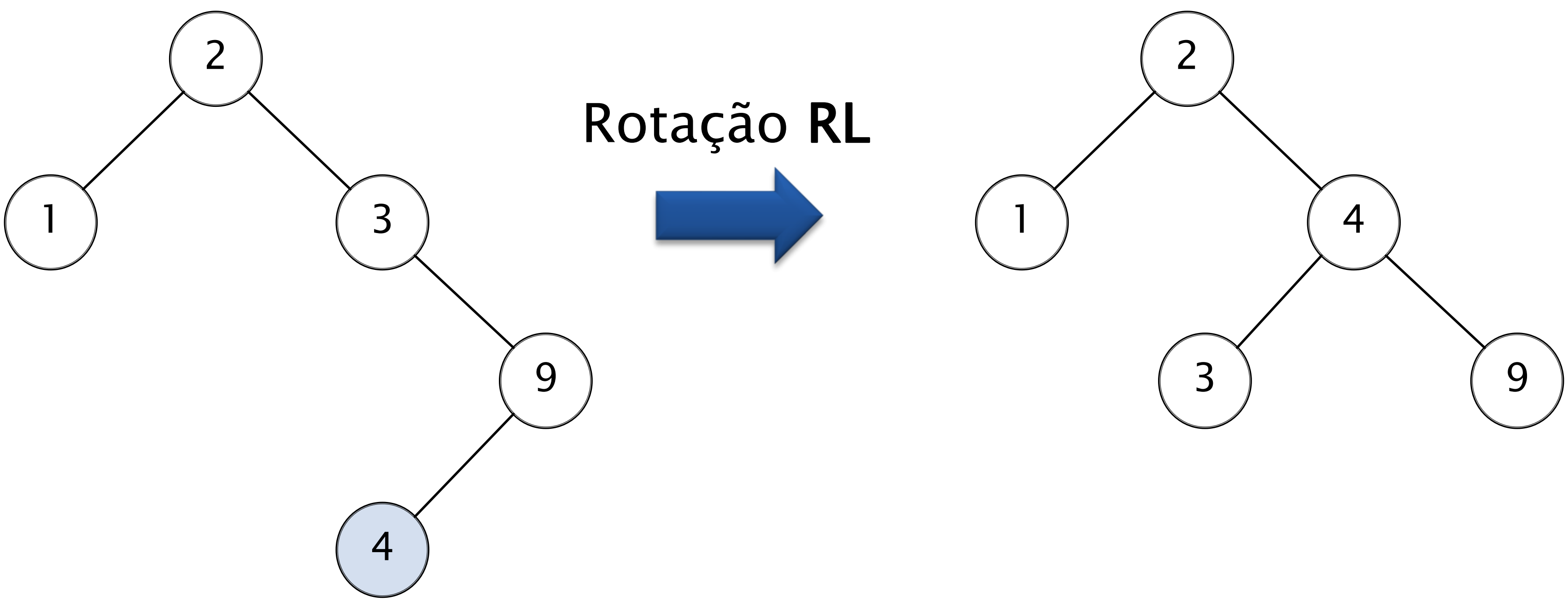
4. Inserção

- Passo a passo – Inserir: 9



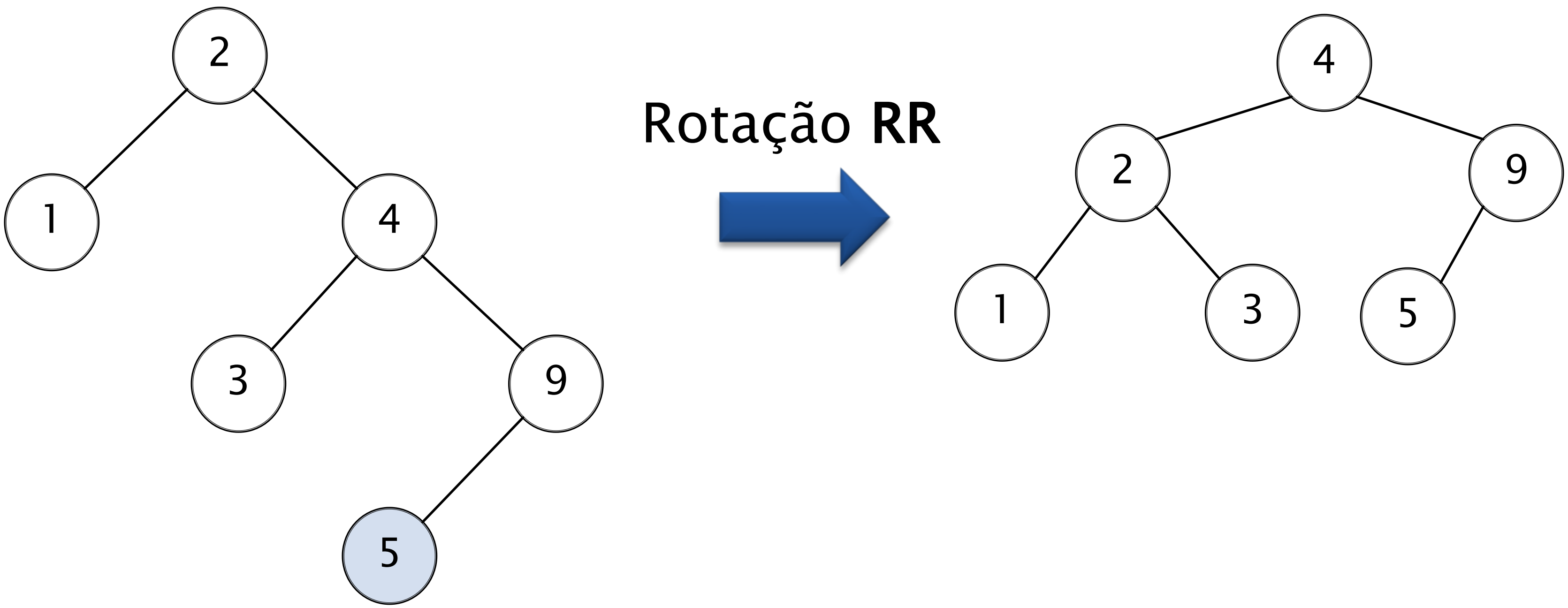
4. Inserção

- Passo a passo – Inserir: **4**



4. Inserção

- Passo a passo – Inserir: **5**



4.1. Implementação

```c
int insere (TNo** ppRaiz, TRegistro* x) {

  if (*ppRaiz == NULL) {
    *ppRaiz = (TNo*) malloc (sizeof(TNo));
    (*ppRaiz)->reg = *x;
    (*ppRaiz)->pEsq = NULL;
    (*ppRaiz)->pDir = NULL;
    return 1; }

  else if ((*ppRaiz)->reg.chave > x->chave) {
    if (insere (&(*ppRaiz)->pEsq, x)) {
      if (balanceamento (ppRaiz))
        return 0;
      else
        return 1;
    }
  }
```

4.1. Implementação

```
  else if ((*ppRaiz)->reg.chave < x->chave) {
    if (insere (&(*ppRaiz)->pDir, x)) {
      if (balanceamento (ppRaiz))
        return 0;
      else
        return 1; }
    else
      return 0; }
  else
    return 0; /* valor já presente */
}
```

5. Remoção

- Como na inserção, é preciso percorrer um conjunto de nós da árvore até chegar ao nó que será removido.

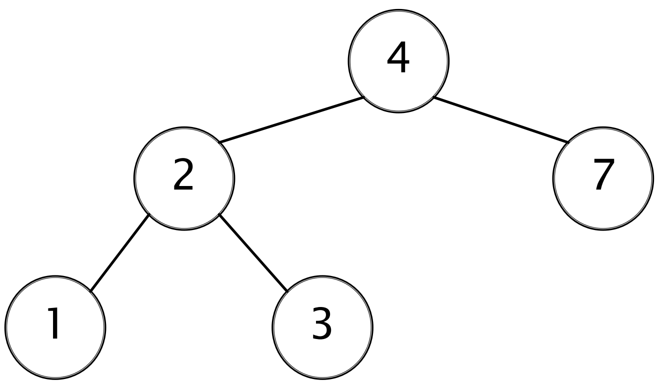
- Existem 3 tipos de remoção:
  - Nó folha (sem filhos)
  - Nó com 1 filho
  - Nó com 2 filhos

5. Remoção

- Uma vez removido o nó:
  - É necessário voltar pelo caminho percorrido e calcular o <u>fator de balanceamento</u> de cada um dos nós visitados.
  - Aplicar a <u>rotação</u> necessária para restabelecer o **balanceamento** da árvore se o fator de balanceamento for +2 ou -2.
  - Remover um nó da sub-árvore direita equivale a inserir um nó na sub-árvore esquerda.

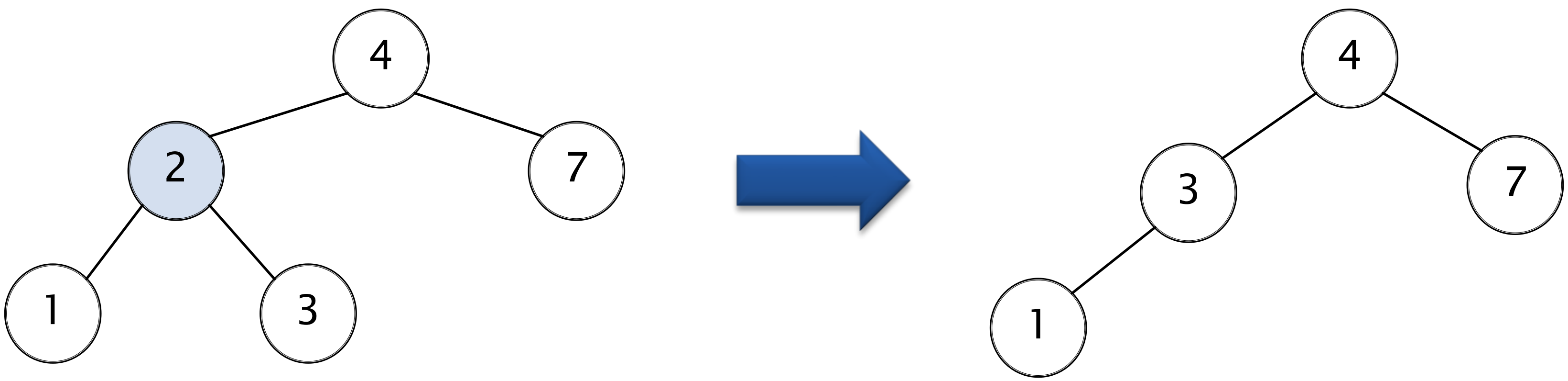
5. Remoção

- Passo a passo



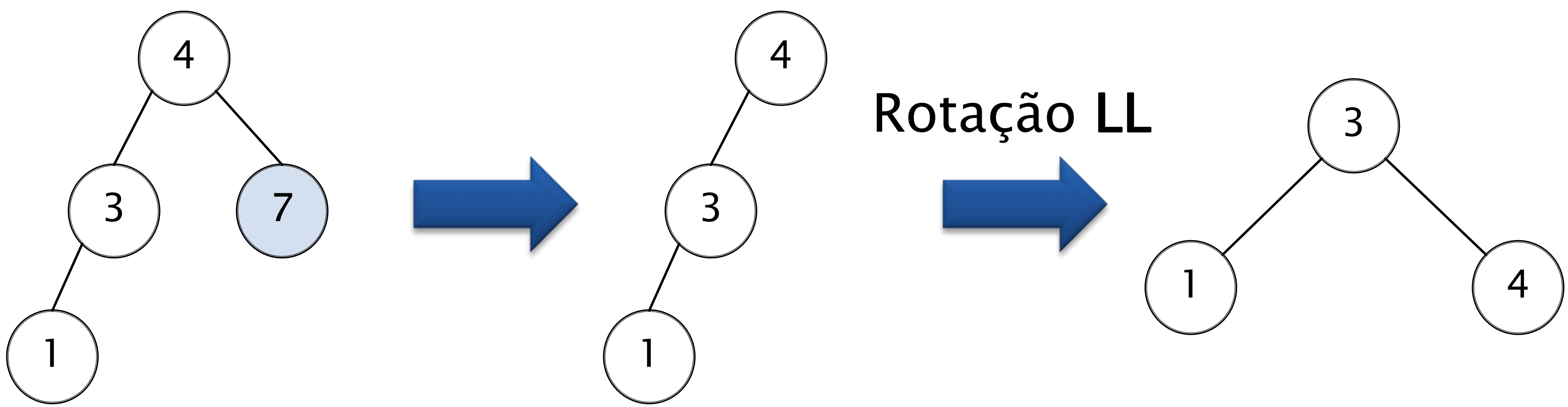
5. Remoção

- Passo a passo – Remover **2**



5. Remoção

- Passo a passo – Remover **7**



5.1. Implementação

```
int retira (TNo** ppRaiz, TRegistro* pX) {
  if (*ppRaiz == NULL)
    return 0;

  else if ((*ppRaiz)->reg.chave > pX->chave) {
    if (retira (&((*ppRaiz)->pEsq), pX)) {
      balanceamento (ppRaiz);
      return 1; }
    else
      return 0;
  }

  else if ((*ppRaiz)->reg.chave < pX->chave) {
    if (retira (&((*ppRaiz)->pDir), pX)) {
      balanceamento (ppRaiz);
      return 1; }
    else
      return 0;
  }
```

5.1. Implementação

```
else if ((*ppRaiz)->reg.chave == pX->chave) {

  TNo* pAux;
  *pX = (*ppRaiz)->reg;

  if ((*ppRaiz)->pDir == NULL) {
    pAux = *ppRaiz;
    *ppRaiz = (*ppRaiz)->pEsq;
    free (pAux);
    balanceamento(ppRaiz);
    return 1; }
```

5.1. Implementação

```
    if ((*ppRaiz)->pEsq == NULL) {
      pAux = *ppRaiz;
      *ppRaiz = (*ppRaiz)->pDir;
      free (pAux);
      balanceamento(ppRaiz);
      return 1; }

    /* Dois filhos */
    sucessor (*ppRaiz, &(*ppRaiz)->pDir);
    balanceamento(ppRaiz);
    return 1;
  }
}
```

5.1. Implementação

```c
void sucessor (TNo* q, TNo** r) {
  TNo* pAux;

  if ((*r)->pEsq != NULL) {
    sucessor (q, &(*r)->pEsq);
    return; }

  q->reg = (*r)->reg;
  pAux = *r;
  *r = (*r)->pDir;
  free (pAux);
}
```

6. Análise

- Tempos de execução para árvores AVL:
  - Uma única reestruturação é O(1).
    - Usando uma árvore binária implementada com estrutura ligada (ponteiros).
  - Pesquisa é **$O(\log_2 n)$**
    - Altura de árvore é **$O(\log_2 n)$**, não necessita reestruturação.

6. Análise

- Tempos de execução para árvores AVL:
  - Inserir é **$O(\log_2 n)$**
    - Busca inicial é **$O(\log_2 n)$.**
    - Reestruturação para manter balanceamento é **$O(\log_2 n)$.**
  - Remover é **$O(\log_2 n)$**
    - Busca inicial é **$O(\log_2 n)$.**
    - Reestruturação para manter balanceamento é **$O(\log_2 n)$.**

6. Análise

- Há um custo adicional para manter uma árvore balanceada, mesmo assim garantindo **$O(\log_2 n)$**, mesmo no pior caso, para todas as operações.

- Em testes empíricos:
  - Uma rotação é necessária a cada **duas** inserções.
  - Uma rotação é necessária a cada **cinco** remoções.

- A remoção em árvore balanceada é tão simples (ou tão complexa) quanto a inserção.

6. Análise

- Aplicações:
  - Redes de Comunicação de Dados.
    - Envio de pacotes ordenados e/ou redundantes.
  - Codificação de *Huffman*
    - Compressão e descompressão de arquivos.

6.1. Verificação

- Algoritmo para verificar se uma árvore binária é uma AVL.
  - Retorno **VERDADEIRO** (**1**) se for AVL.
  - Retorna **FALSO** (**0**) se não for AVL.

6.1. Verificação

```c
int eh_arvore_avl (TNo* pRaiz) {
  int fatb;

  if (pRaiz == NULL)
    return 1;

  if (!eh_arvore_avl (pRaiz->pEsq))
    return 0;
  if (!eh_arvore_avl (pRaiz->pDir))
    return 0;

  fatb = fb (pRaiz);
  if ( ( fatb > 1 ) || ( fatb < -1) )
    return 0;
  else
    return 1;
}
```

7. Referências

- Material de aula dos Profs. Luiz Chaimowicz e Raquel O. Prates, da UFMG: https://homepages.dcc.ufmg.br/~glpappa/aeds2/AEDS2.1%20Conceitos%20Basicos%20TAD.pdf

- Horowitz, E. & Sahni, S.; Fundamentos de Estruturas de Dados, Editora Campus, 1984.

- Wirth, N.; Algoritmos e Estruturas de Dados, Prentice/Hall do Brasil, 1989.

- Material de aula do Prof. José Augusto Baranauskas, da USP: https://dcm.ffclrp.usp.br/~augusto/teaching.htm

- Material de aula do Prof. Rafael C. S. Schouery, da Unicamp: https://www.ic.unicamp.br/~rafael/cursos/2s2019/mc202/index.html