Árvores AVL

- Nesta aula será apresentado o ADT árvore AVL que são árvores binárias de altura equilibrada

Algoritmos e Estruturas de Dados I

José Augusto Baranauskas
Departamento de Física e Matemática – FFCLRP-USP

augusto@usp.br
http://dfm.ffclrp.usp.br/~augusto

Introdução

- 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 $O(\log_2 \mathbf{n})$, mesmo no pior caso
- Um teorema provado por Adelson-Velskii e Landis garante que a árvore balanceada nunca será 45% mais alta que a correspondente árvore perfeitamente balanceada, independentemente do número de nós existentes

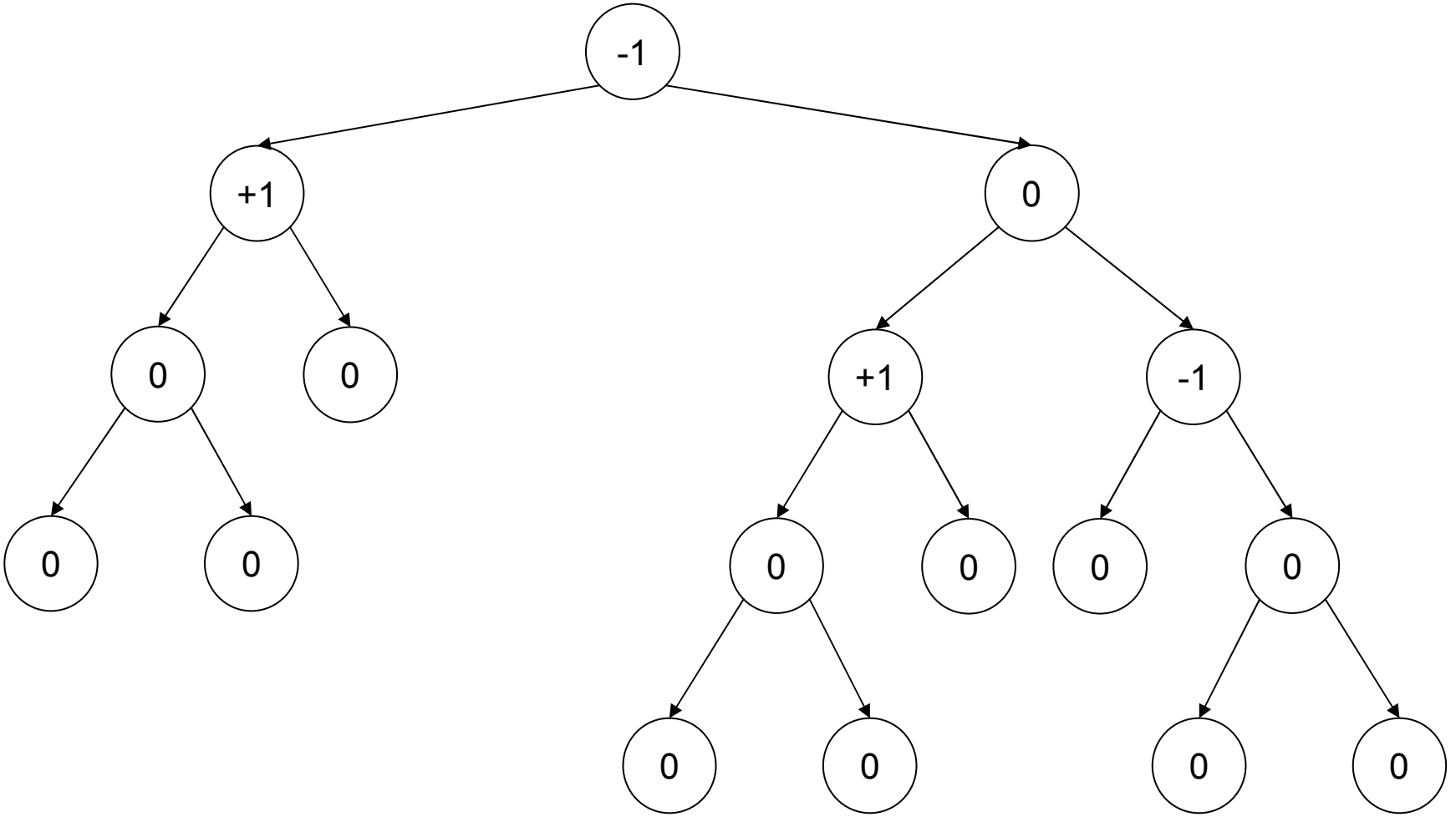
Árvore AVL

- Uma árvore AVL é definida como:
  - Uma árvore vazia é uma árvore AVL
  - Sendo **T** uma árvore binária de busca cujas subárvores esquerda e direita são **L** e **R**, respectivamente, **T** será uma árvore AVL contanto que:
    - **L** e **R** são árvores AVL
    - $|h_L - h_R| \leq 1$, onde h_L e h_R são as alturas das subárvores **L** e **R**, respectivamente
- A definição de uma árvore binária de altura equilibrada (AVL) requer que cada subárvore seja também de altura equilibrada

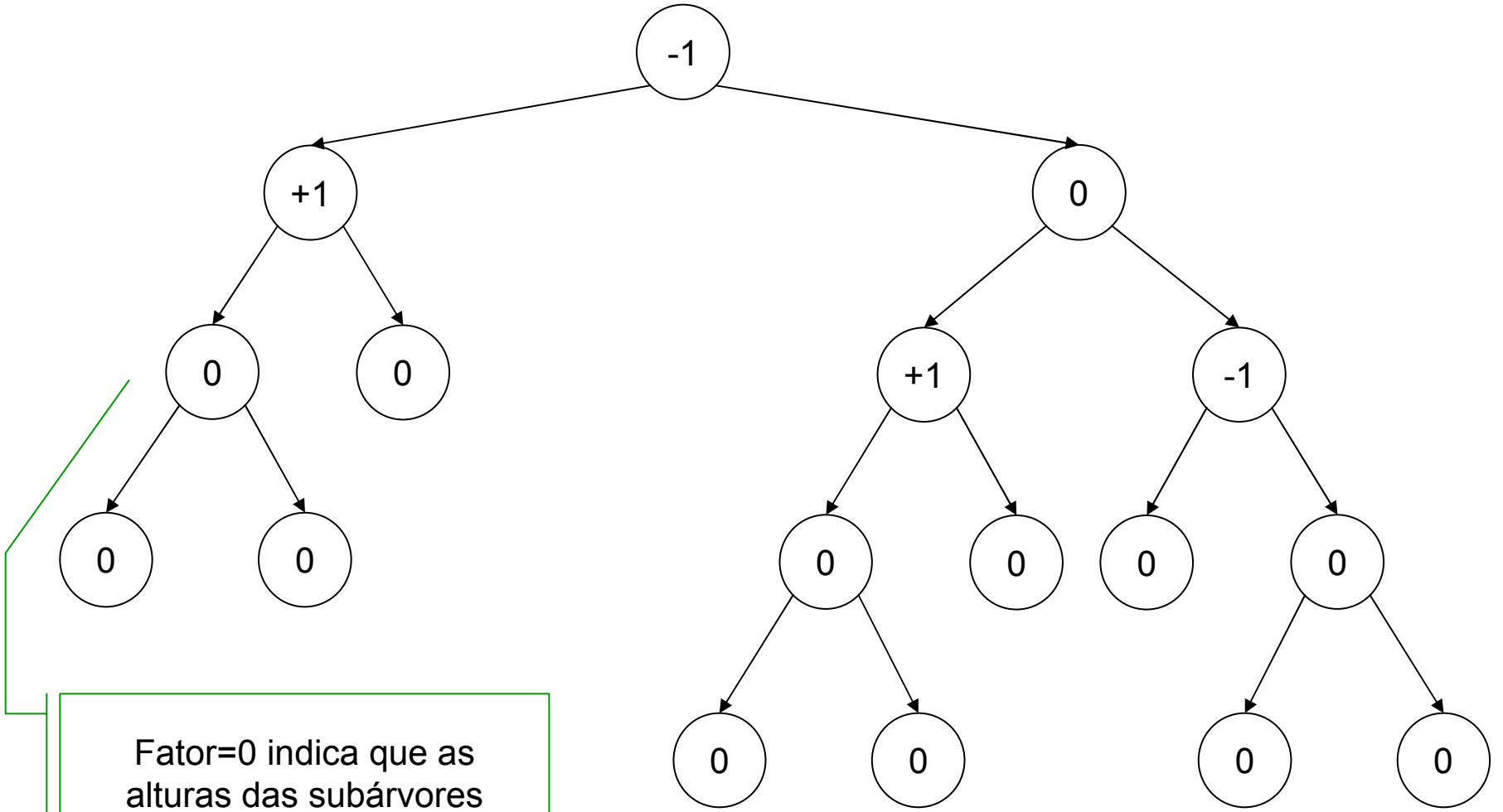
Fator de Balanceamento

- O **fator de balanceamento** ou **fator de equilíbrio** de um nó **T** em uma árvore binária é definido como sendo $\mathbf{h_L - h_R}$ onde h_L e h_R são as alturas das subárvores 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

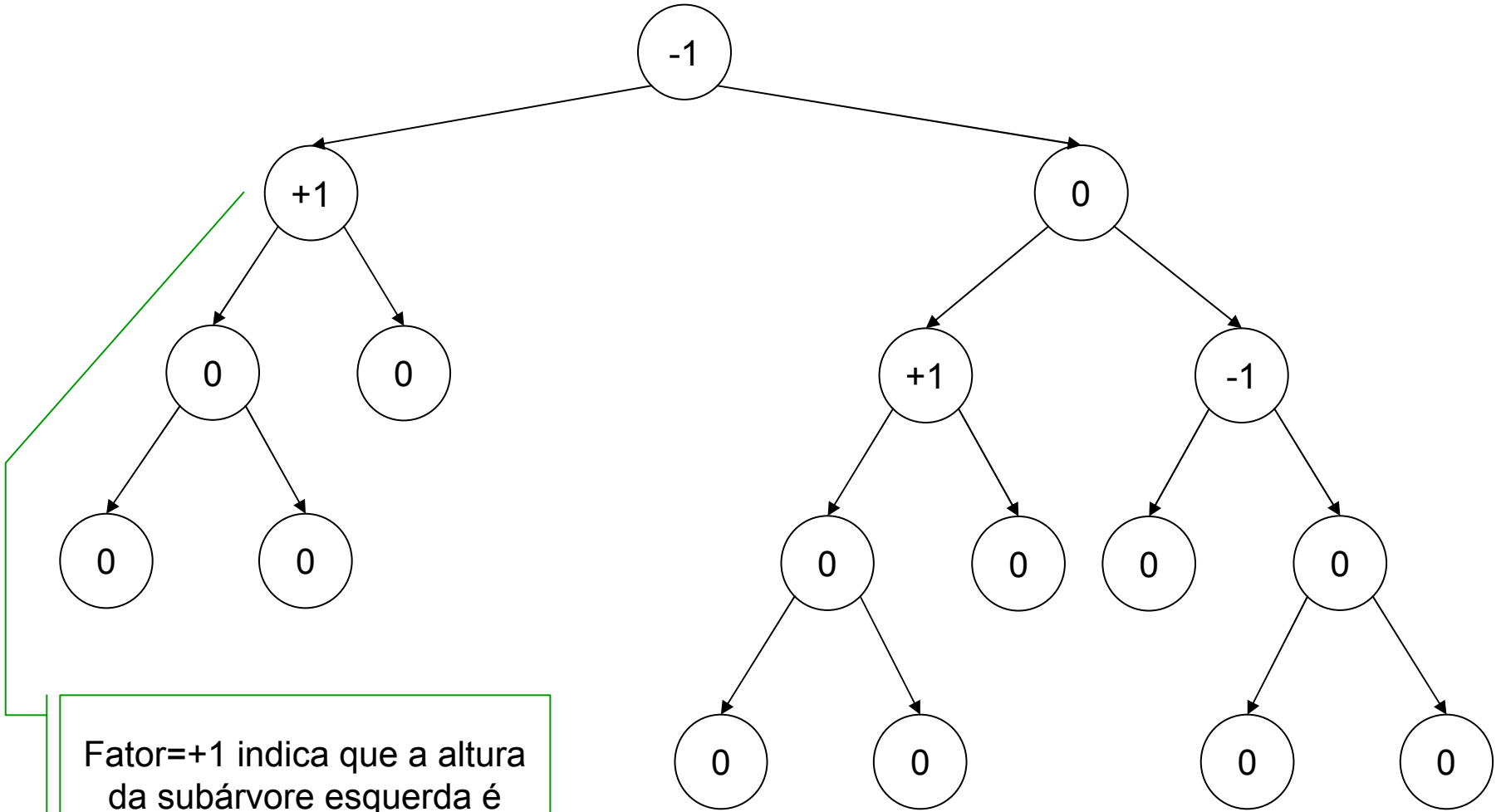
Fator de Balanceamento



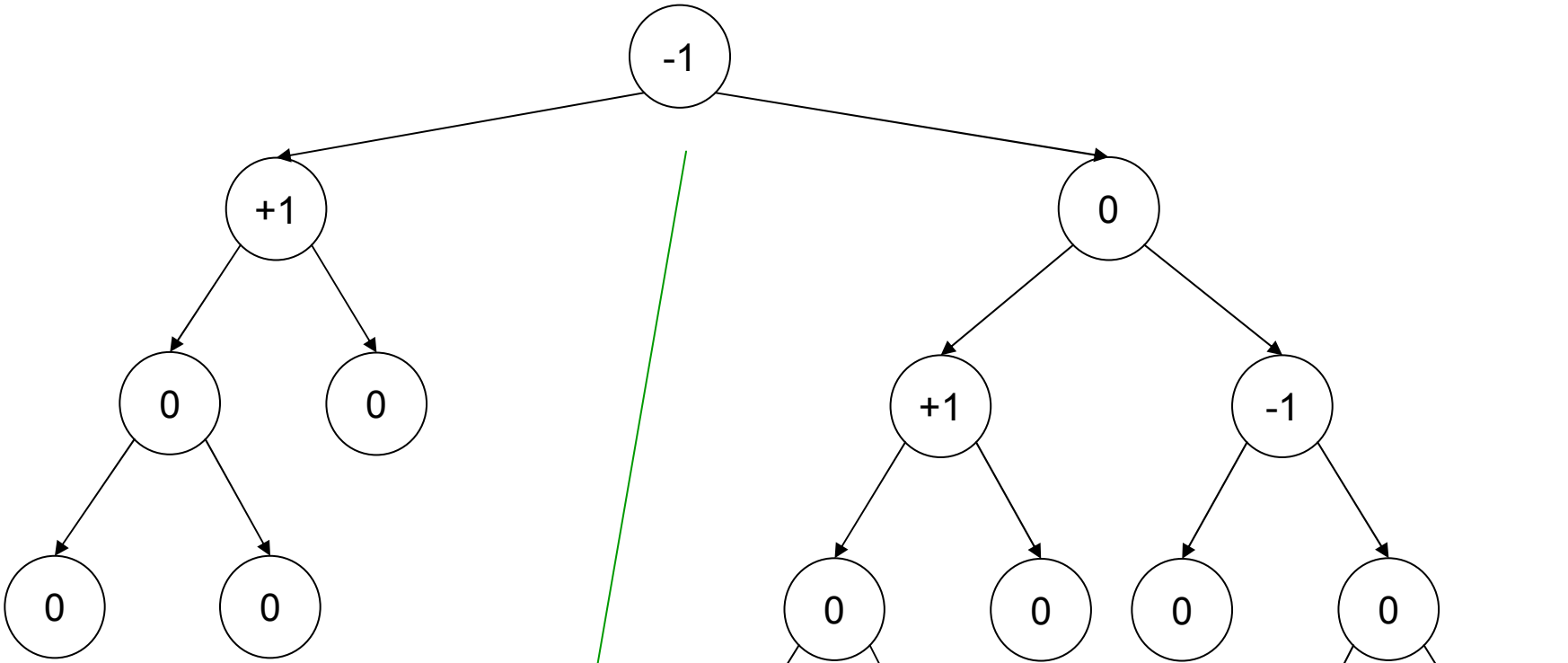
Fator de Balanceamento



Fator de Balanceamento



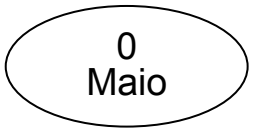
Fator de Balanceamento



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

Inserção Maio

Depois da inserção

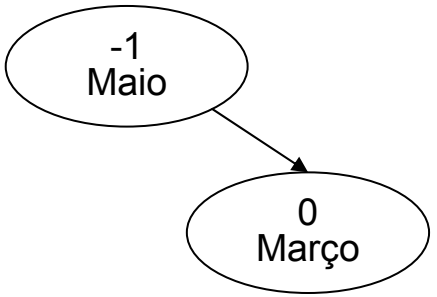


Depois do rebalanceamento

Sem necessidade de rebalanceamento

Inserção Março

Depois da inserção

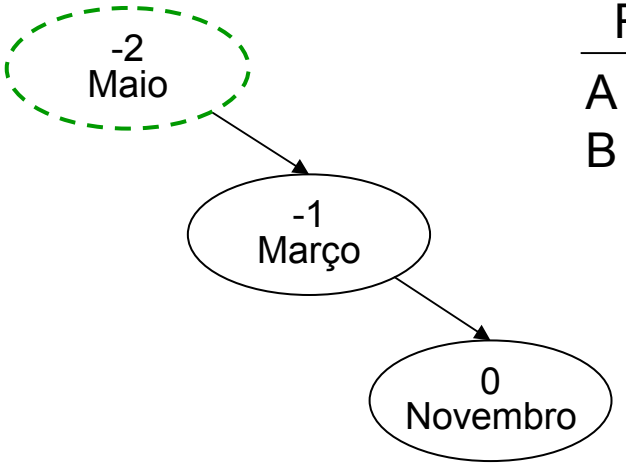


Depois do rebalanceamento

Sem necessidade de rebalanceamento

Inserção Novembro

Depois da inserção

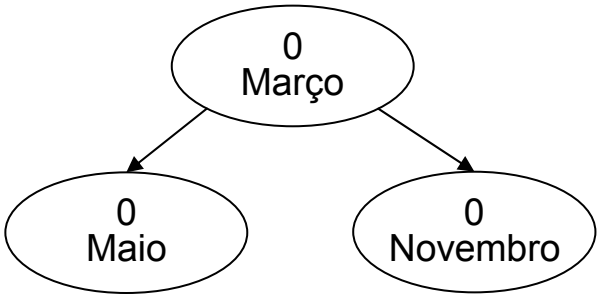


RR

A = -2

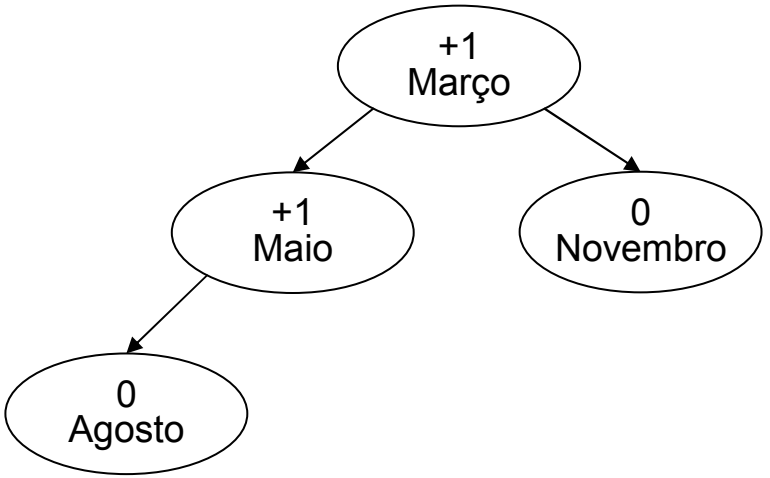
B = -1

Depois do rebalanceamento



Inserção Agosto

Depois da inserção



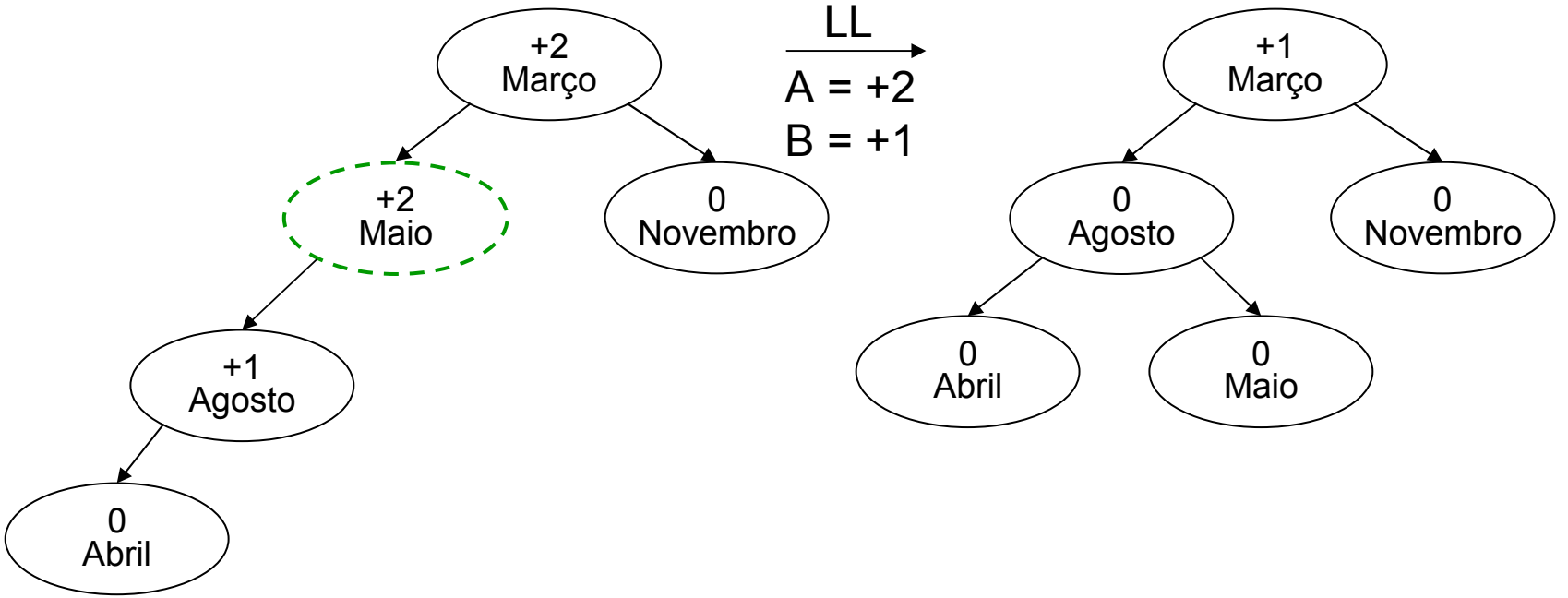
Depois do rebalanceamento

Sem necessidade de rebalanceamento

Inserção Abril

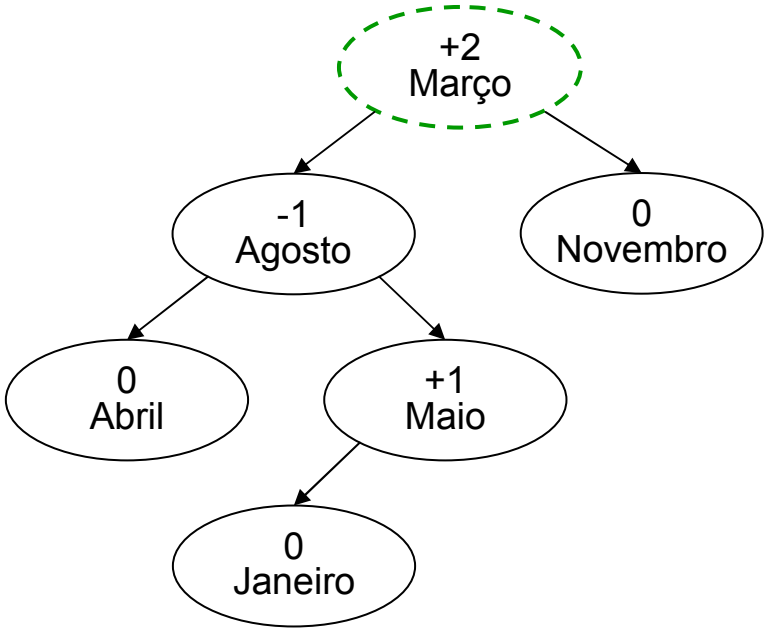
Depois da inserção

Depois do rebalanceamento



Inserção Janeiro

Depois da inserção

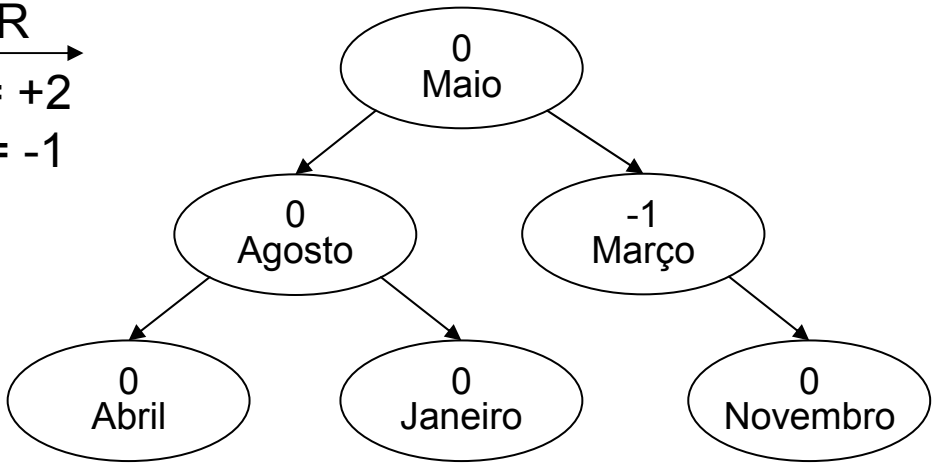


LR

A = +2

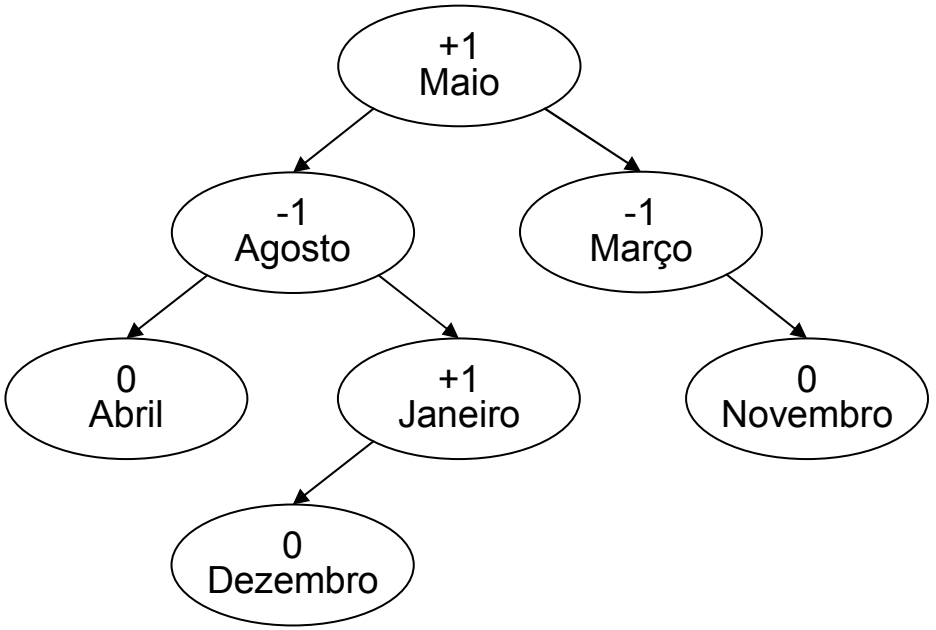
B = -1

Depois do rebalanceamento



Inserção Dezembro

Depois da inserção

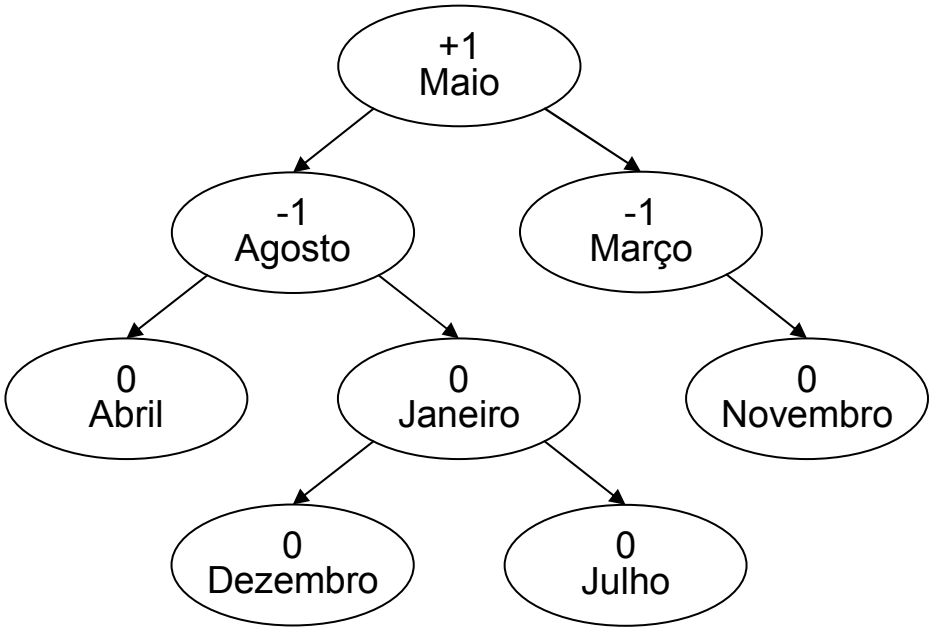


Depois do rebalanceamento

Sem necessidade de rebalanceamento

Inserção Julho

Depois da inserção

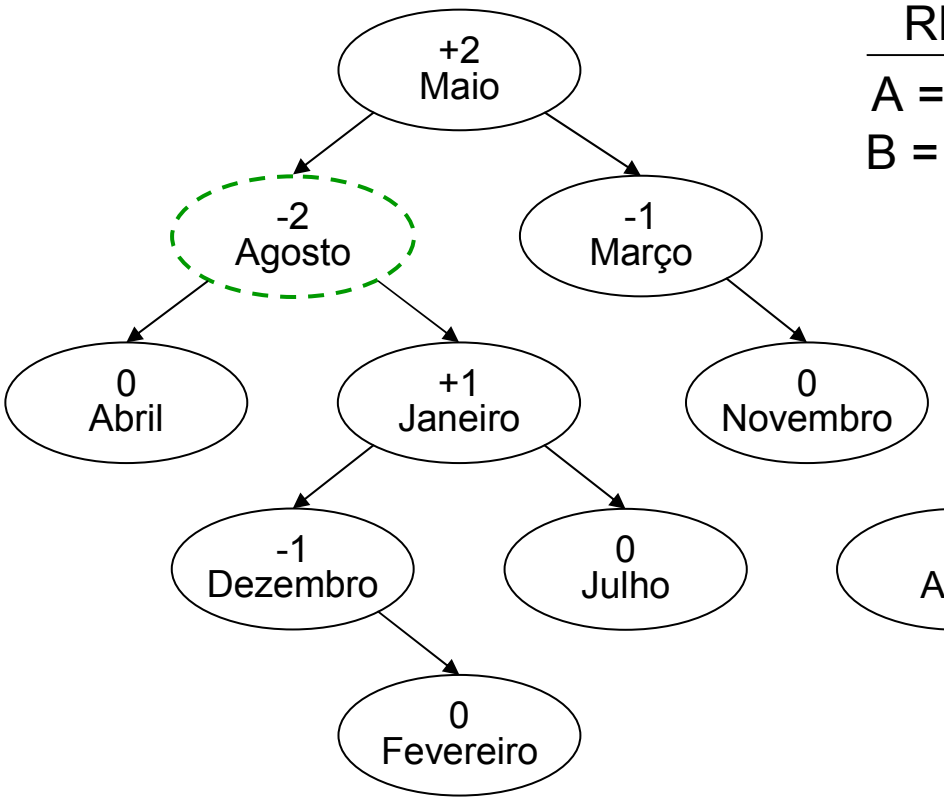


Depois do rebalanceamento

Sem necessidade de rebalanceamento

Inserção Fevereiro

Depois da inserção

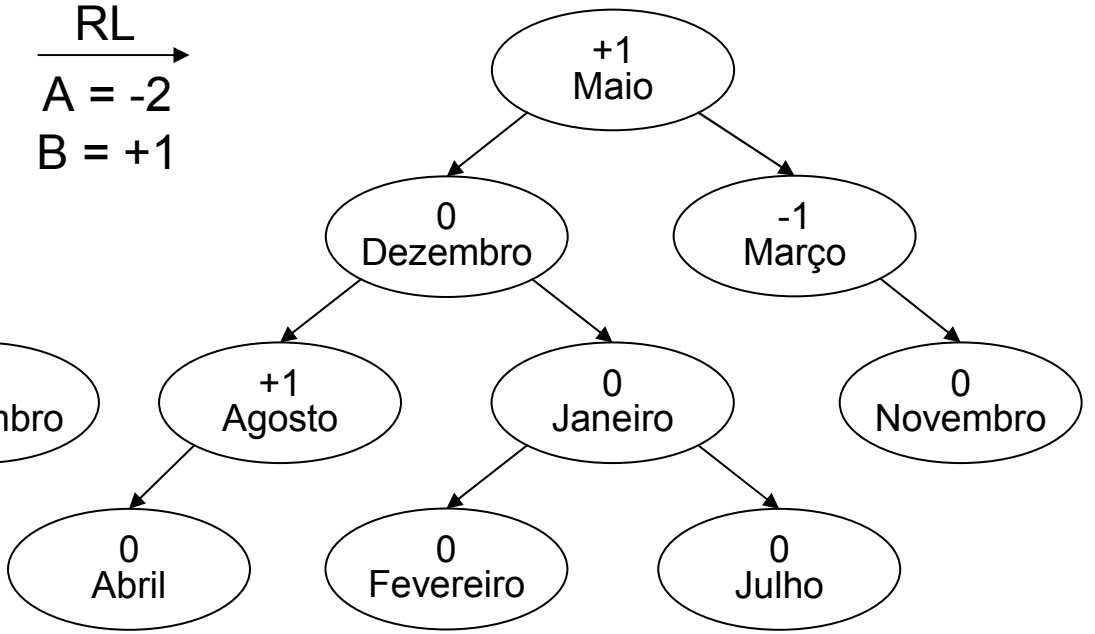


RL

A = -2

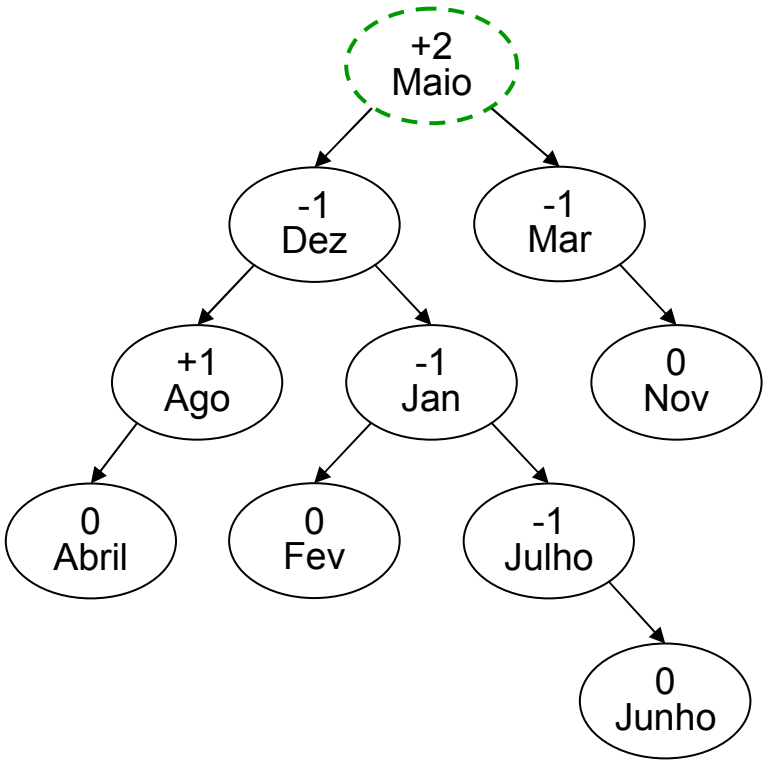
B = +1

Depois do rebalanceamento



Inserção Junho

Depois da inserção

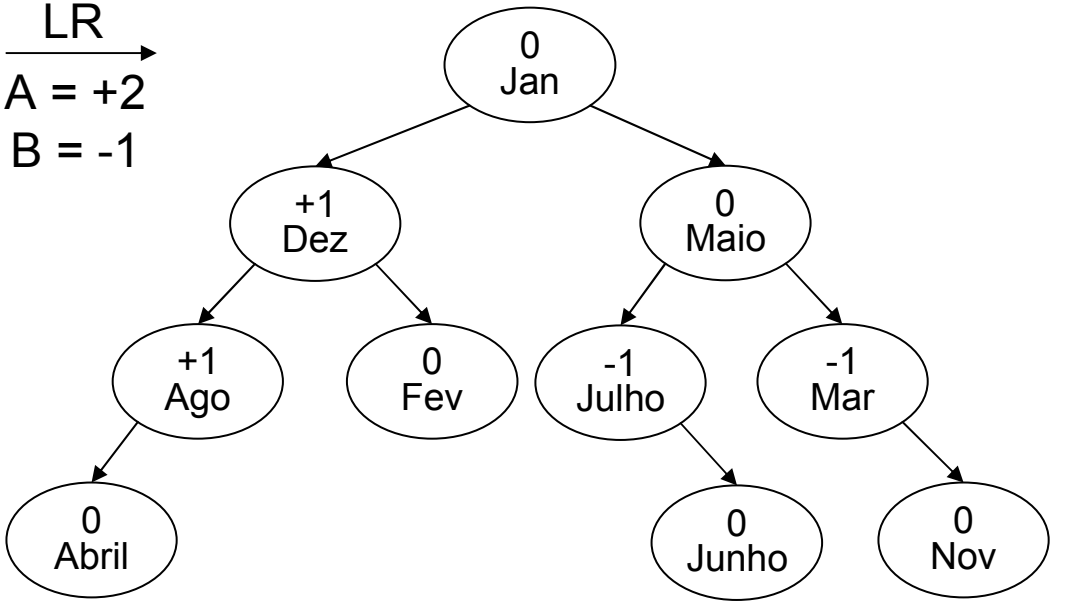


LR

A = +2

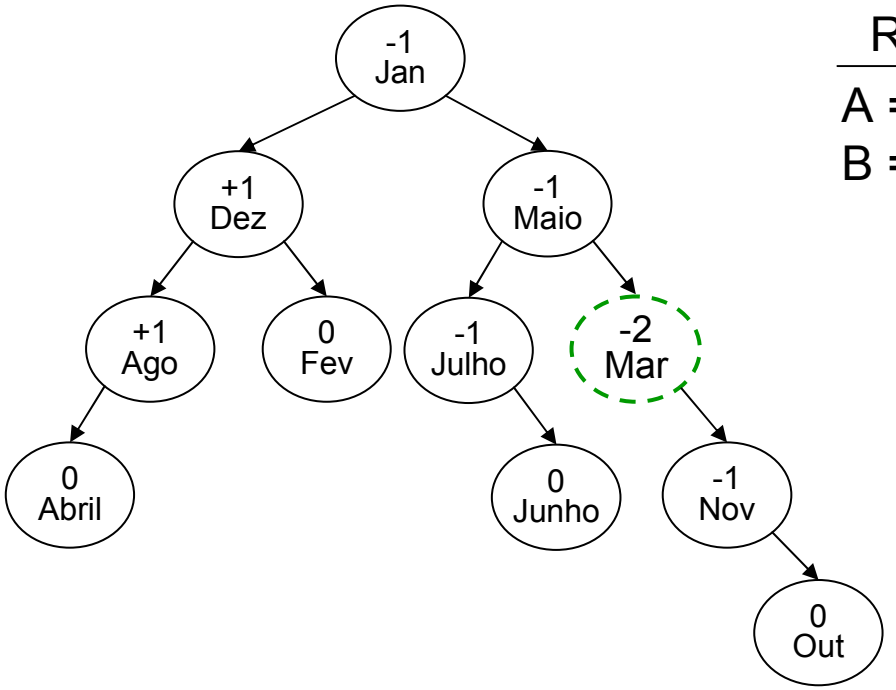
B = -1

Depois do rebalanceamento



Inserção Outubro

Depois da inserção

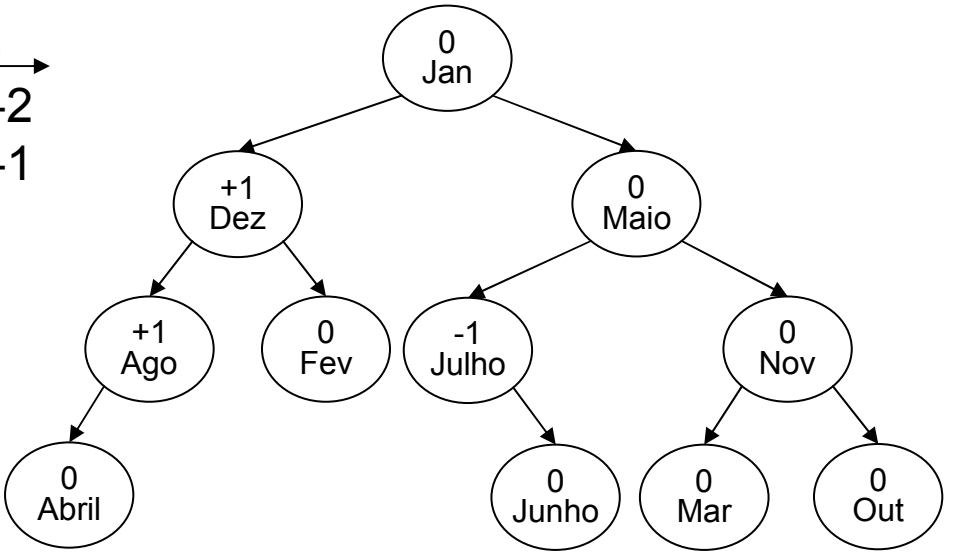


RR

A = -2

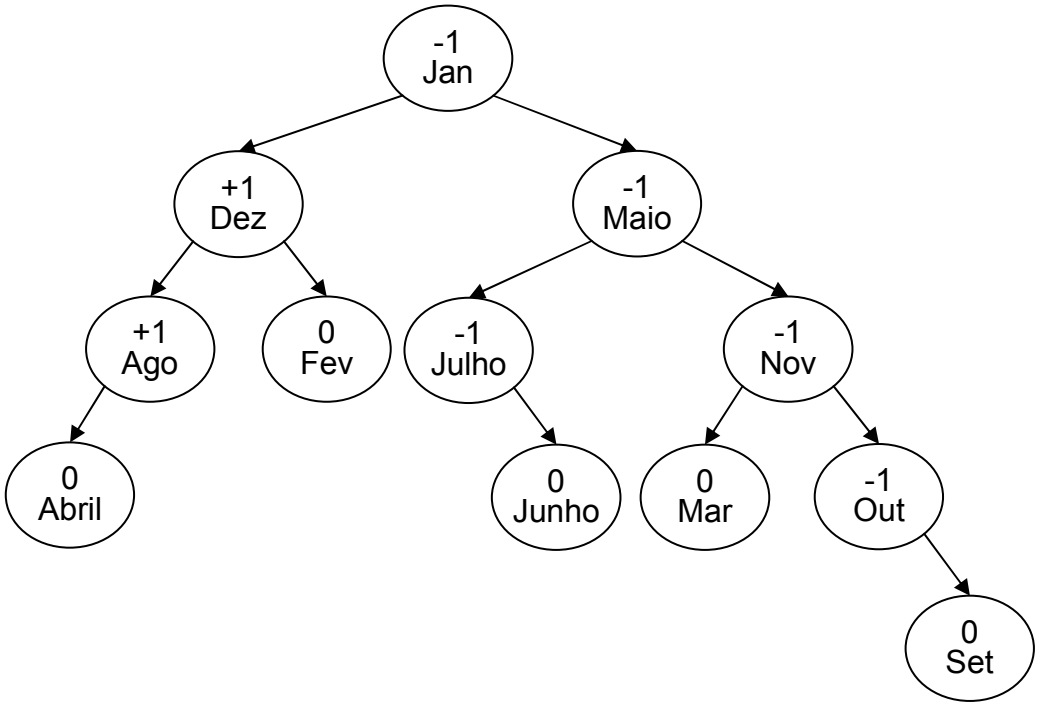
B = -1

Depois do rebalanceamento



Inserção Setembro

Depois da inserção



Depois do rebalanceamento

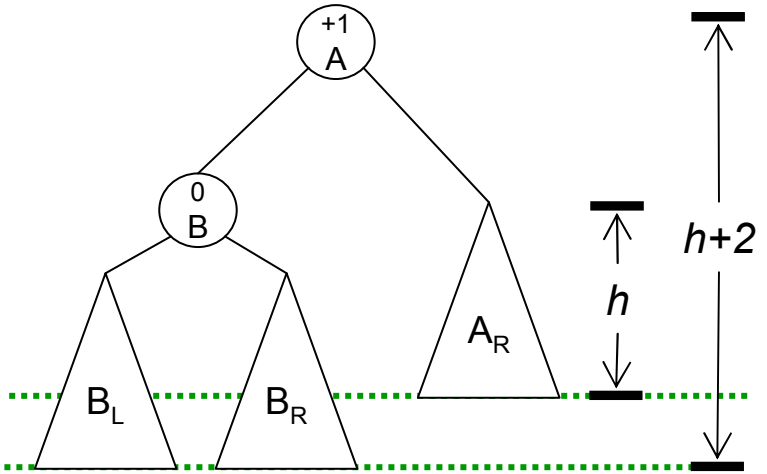
Sem necessidade de rebalanceamento

Rotações

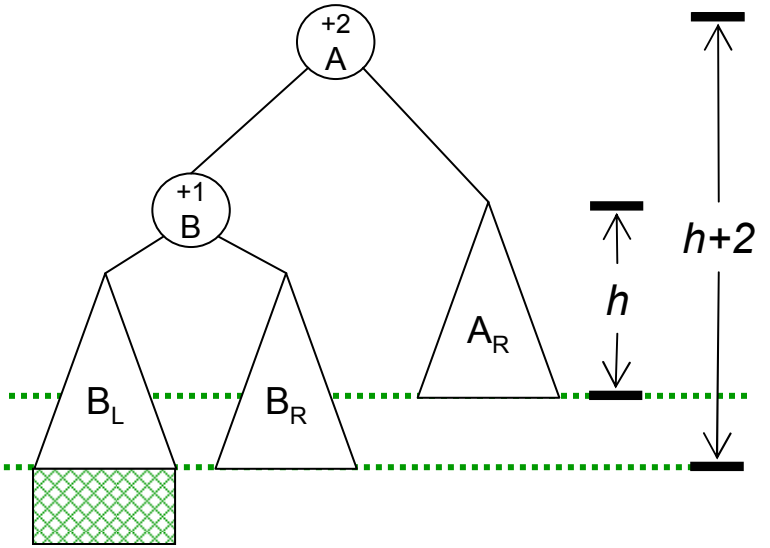
- O processo de rebalanceamento é conduzido utilizando 4 tipos de rotações: LL, RR, LR, RL
  - LL e RR são simétricas entre si assim como LR e RL
- As rotações são caracterizadas pelo ancestral mais próximo **A** do novo nó inserido **Y** cujo fator de balanceamento passa a ser +2 ou -2
  - LL: **Y** inserido na subárvore esquerda da subárvore esquerda de **A**
  - LR: **Y** inserido na subárvore direita da subárvore esquerda de **A**
  - RR: **Y** inserido na subárvore direita da subárvore direita de **A**
  - RL: **Y** inserido na subárvore esquerda da subárvore direita de **A**
- Seja **B** o filho de **A** no qual ocorreu a inserção de **Y**
  - LL (A = +2; B = +1) RR (A = -2; B = -1)
  - LR (A = +2; B = -1) RL (A = -2; B = +1)
- **C** é o filho de **B** no qual ocorreu a inserção de **Y**

Rotação LL

Subárvore balanceada

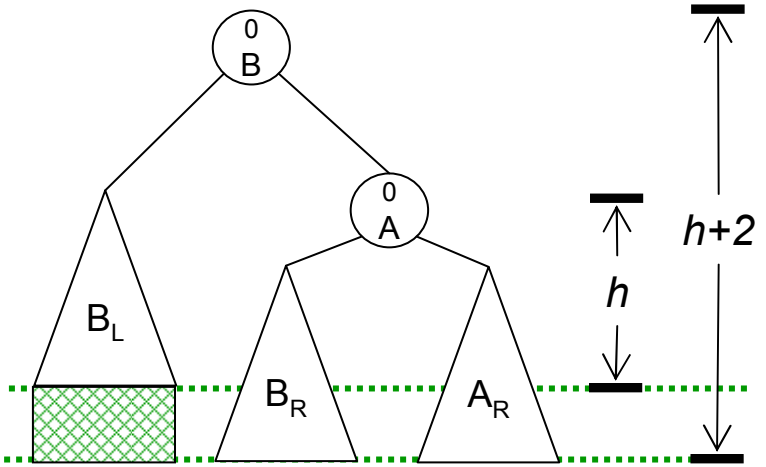


Subárvore desbalanceada após inserção

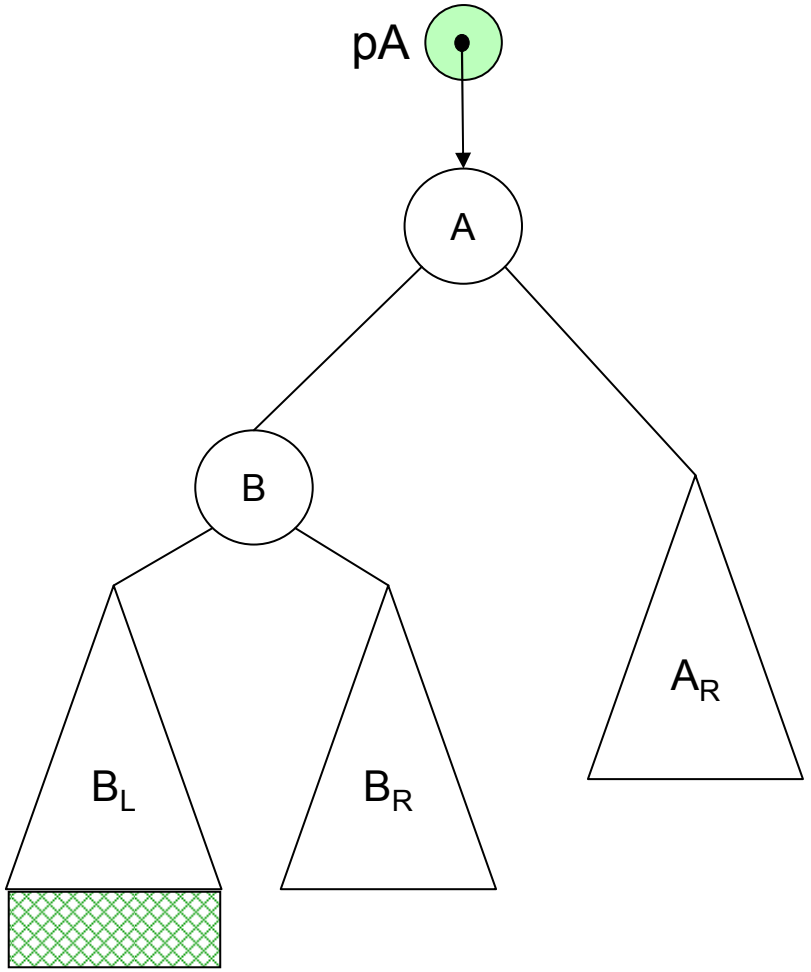


Altura de B_L aumenta para h+1

Subárvore rebalanceada

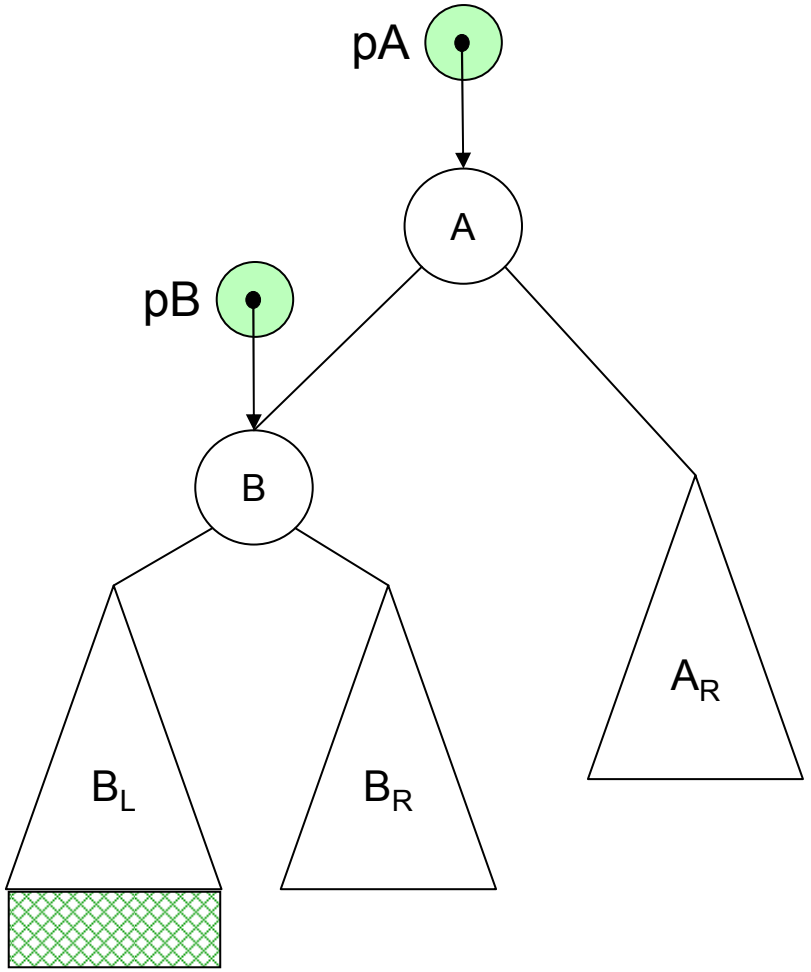


Rotação LL



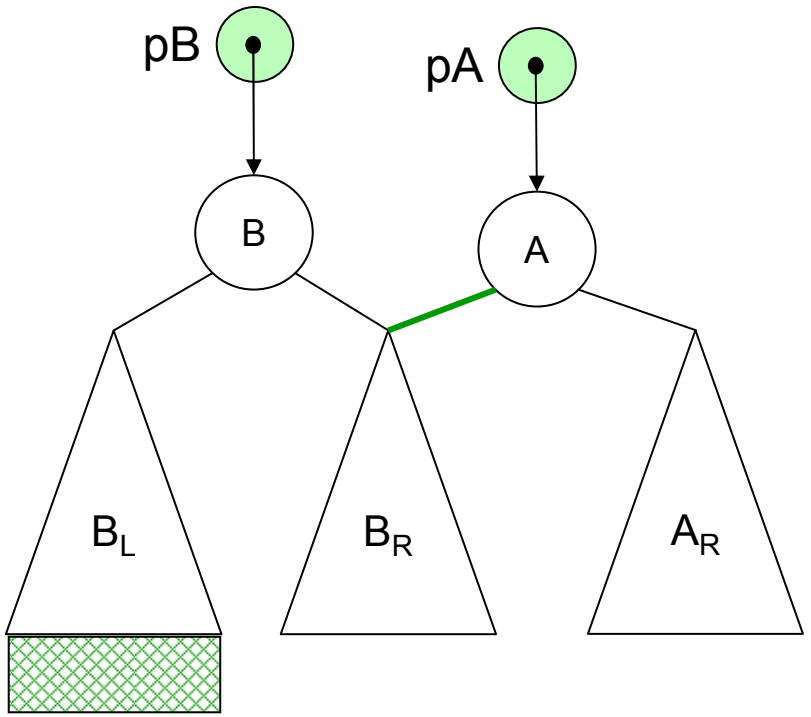
- Assumindo pA e pB ponteiros para as subárvores com raízes A e B:
  - pB = pA->LeftNode;
  - pA->LeftNode = pB->RightNode;
  - pB->RightNode = pA;
  - pA = pB;

Rotação LL



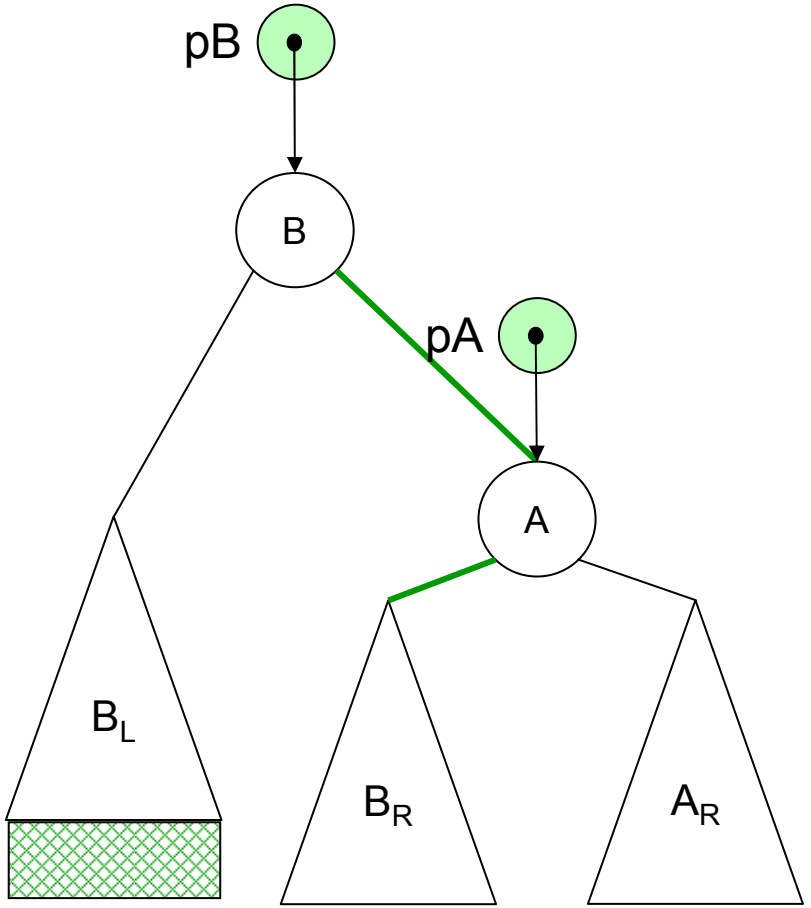
- Assumindo pA e pB ponteiros para as subárvores com raízes A e B:
  - **pB = pA->LeftNode;**
  - pA->LeftNode = pB->RightNode;
  - pB->RightNode = pA;
  - pA = pB;

Rotação LL



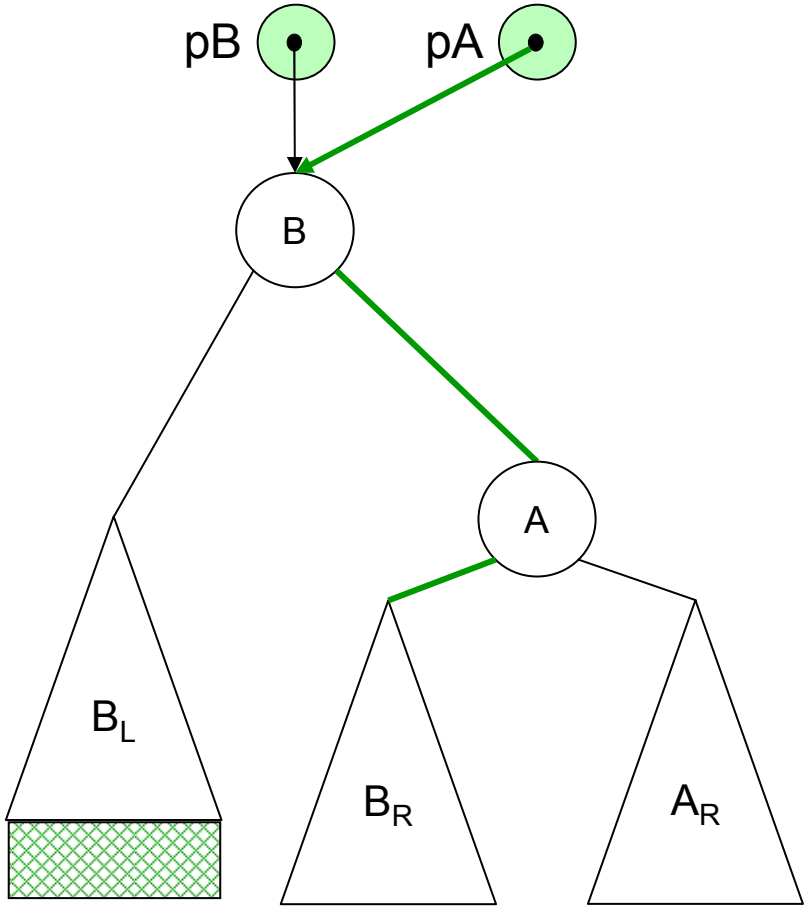
- Assumindo pA e pB ponteiros para as subárvores com raízes A e B:
  - pB = pA->LeftNode;
  - **pA->LeftNode = pB->RightNode;**
  - pB->RightNode = pA;
  - pA = pB;

Rotação LL



- Assumindo pA e pB ponteiros para as subárvores com raízes A e B:
  - pB = pA->LeftNode;
  - pA->LeftNode = pB->RightNode;
  - **pB->RightNode = pA;**
  - pA = pB;

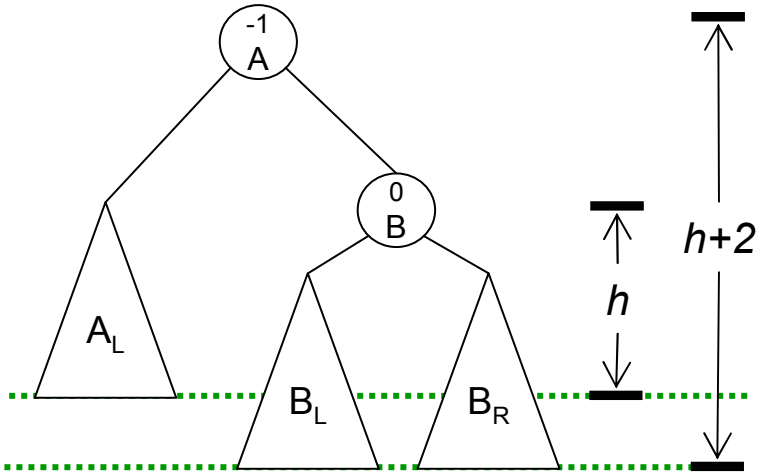
Rotação LL



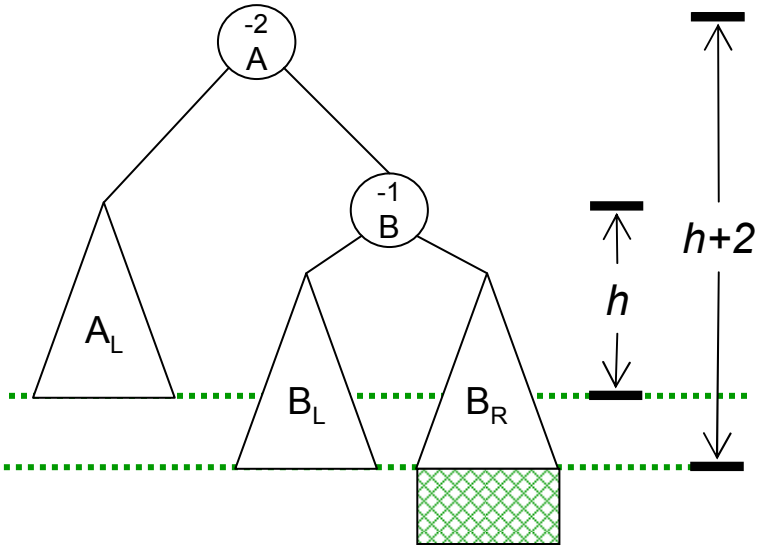
- Assumindo pA e pB ponteiros para as subárvores com raízes A e B:
  - pB = pA->LeftNode;
  - pA->LeftNode = pB->RightNode;
  - pB->RightNode = pA;
  - **pA = pB;**

Rotação RR

Subárvore balanceada

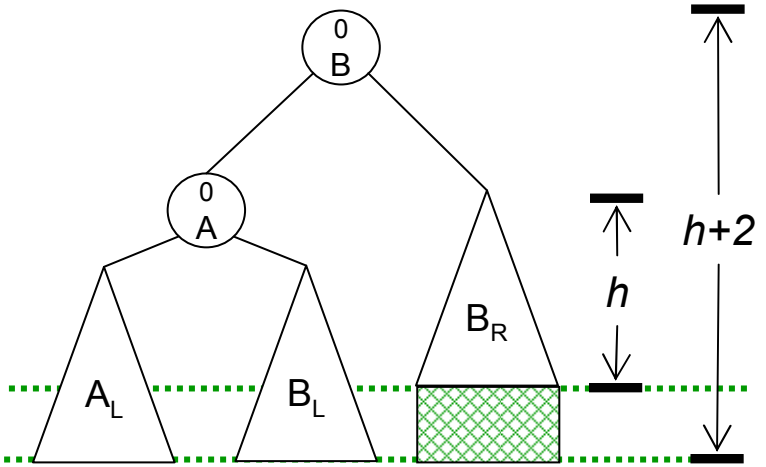


Subárvore desbalanceada após inserção

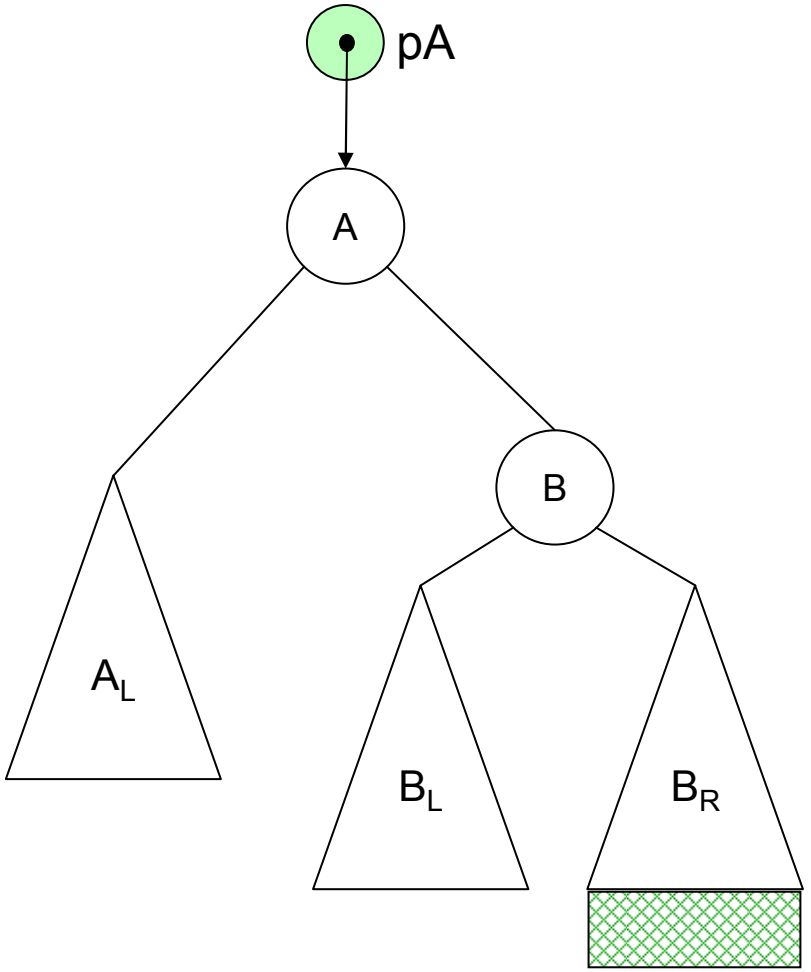


Altura de B_R aumenta para *h*+1

Subárvore rebalanceada

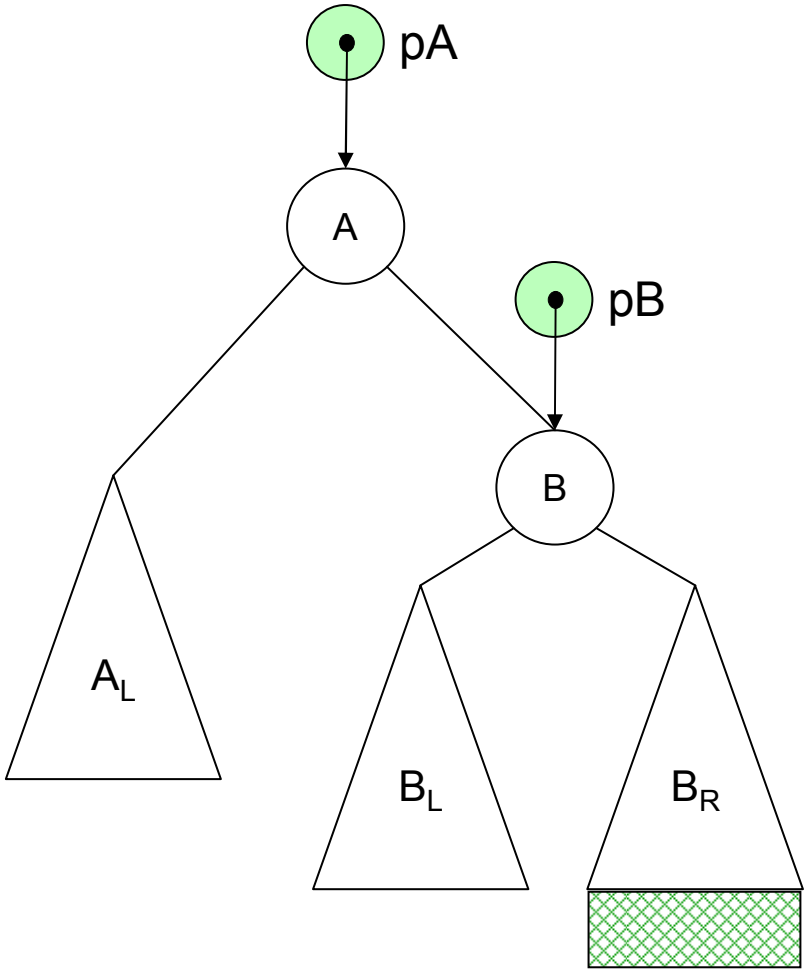


Rotação RR



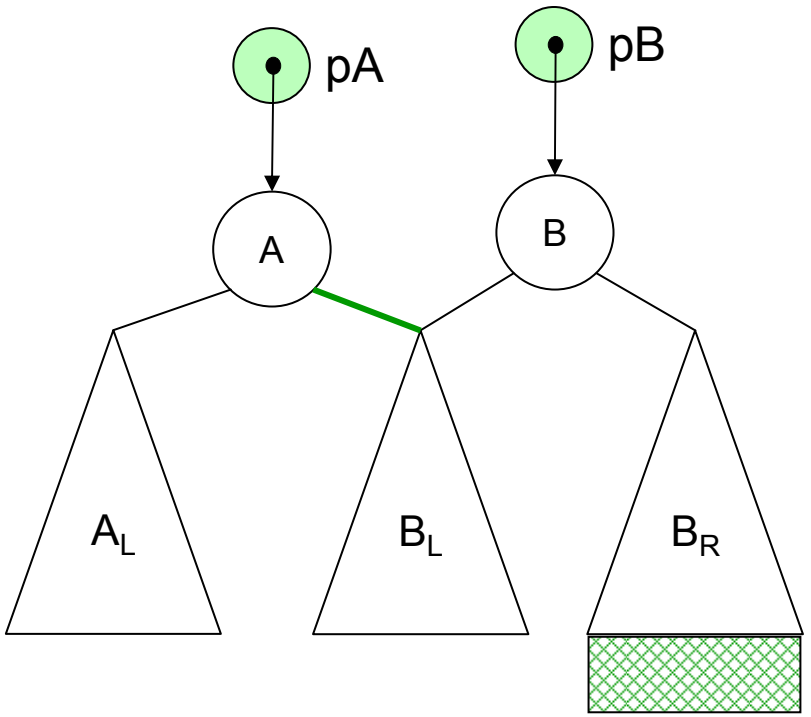
- Assumindo pA e pB ponteiros para as subárvores com raízes A e B:
  - pB = pA->RightNode;
  - pA->RightNode = pB->LeftNode;
  - pB->LeftNode = pA;
  - pA = pB;

Rotação RR



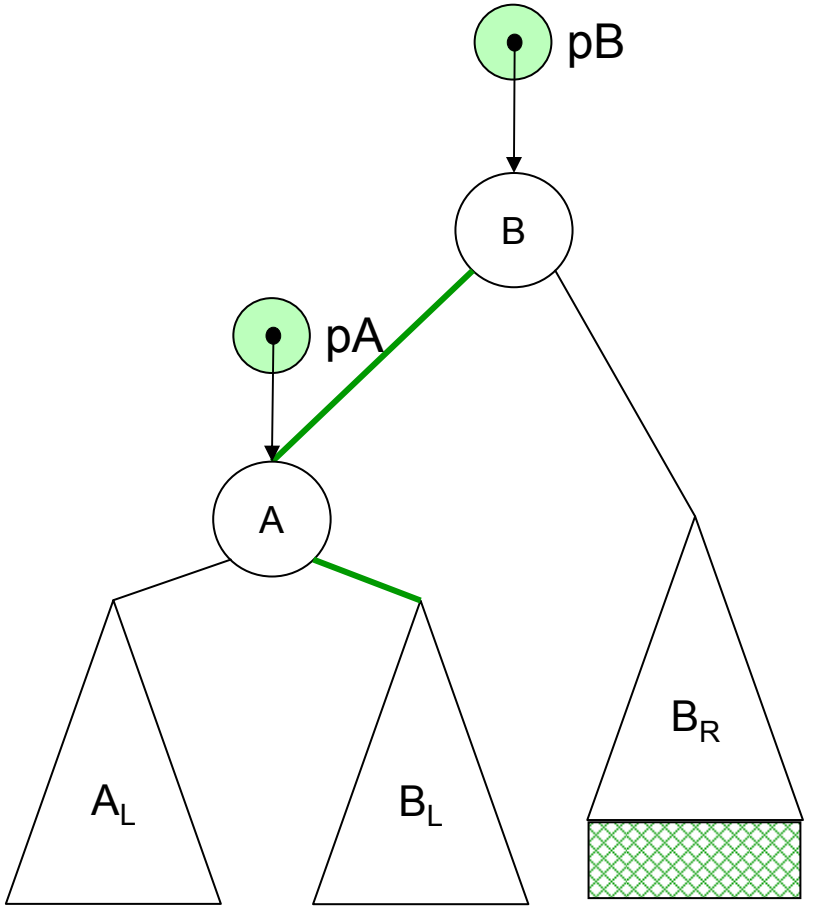
- Assumindo pA e pB ponteiros para as subárvores com raízes A e B:
  - **pB = pA->RightNode;**
  - pA->RightNode = pB->LeftNode;
  - pB->LeftNode = pA;
  - pA = pB;

Rotação RR



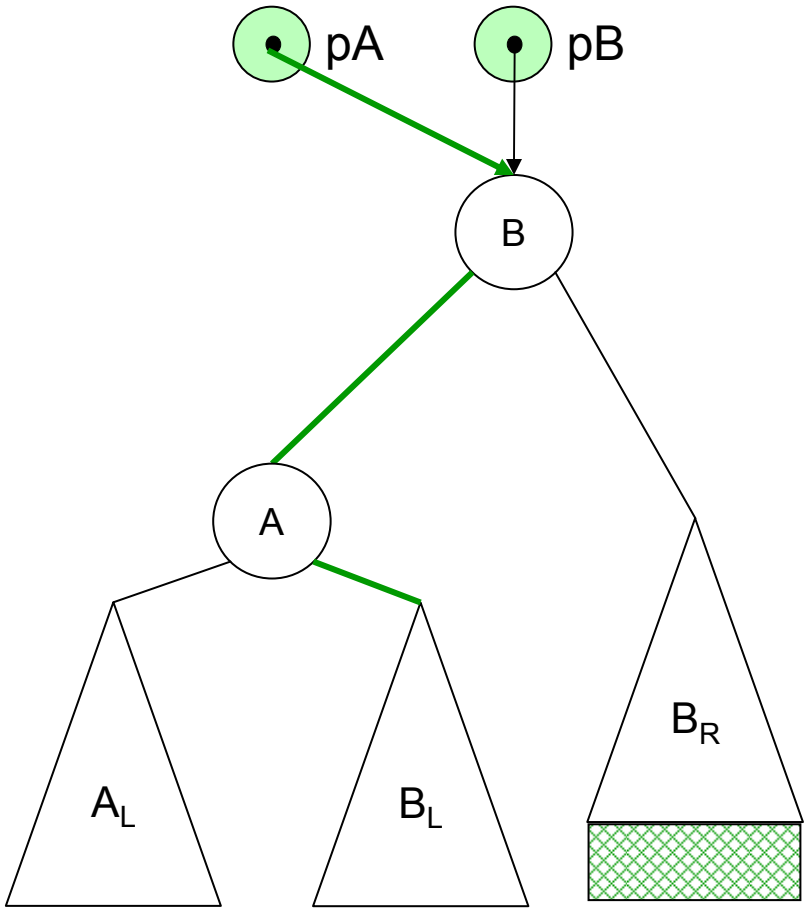
- Assumindo pA e pB ponteiros para as subárvores com raízes A e B:
  - pB = pA->RightNode;
  - **pA->RightNode = pB->LeftNode;**
  - pB->LeftNode = pA;
  - pA = pB;

Rotação RR



- Assumindo pA e pB ponteiros para as subárvores com raízes A e B:
  - pB = pA->RightNode;
  - pA->RightNode = pB->LeftNode;
  - **pB->LeftNode = pA;**
  - pA = pB;

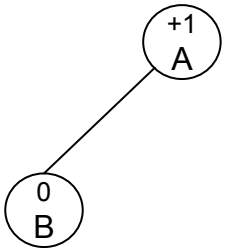
Rotação RR



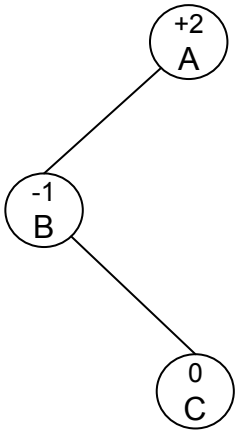
- Assumindo pA e pB ponteiros para as subárvores com raízes A e B:
  - pB = pA->RightNode;
  - pA->RightNode = pB->LeftNode;
  - pB->LeftNode = pA;
  - **pA = pB;**

Rotação LR(a)

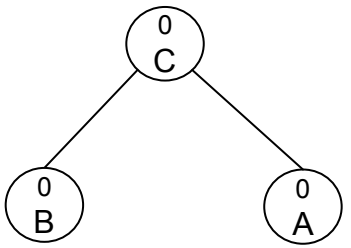
Subárvore balanceada



Subárvore desbalanceada após inserção

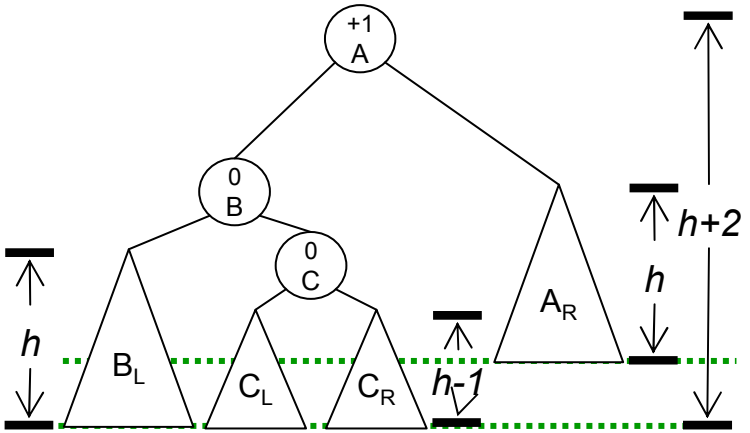


Subárvore rebalanceada

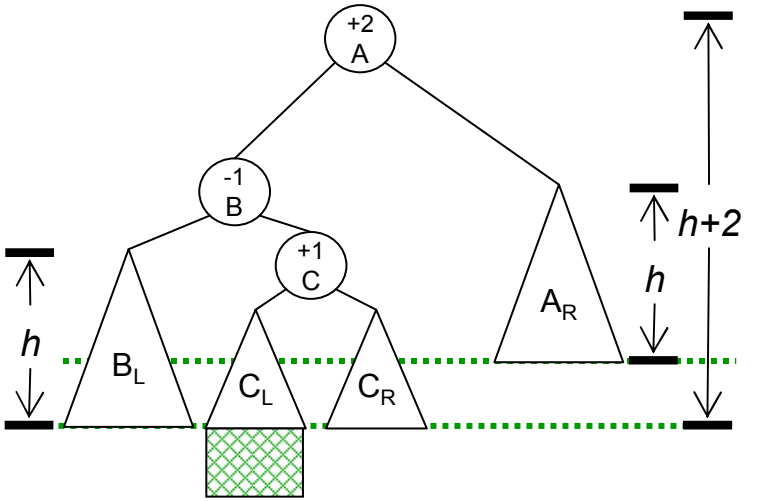


Rotação LR(b)

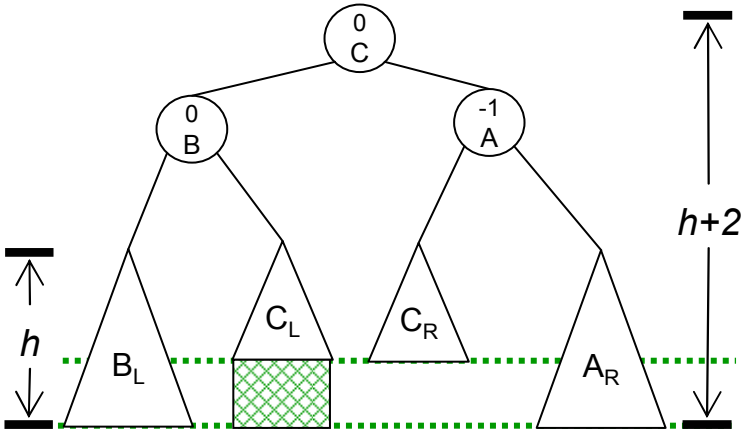
Subárvore balanceada



Subárvore desbalanceada após inserção

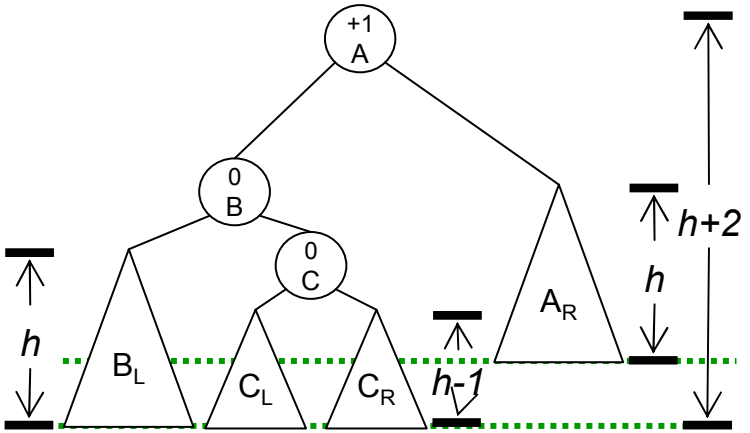


Subárvore rebalanceada

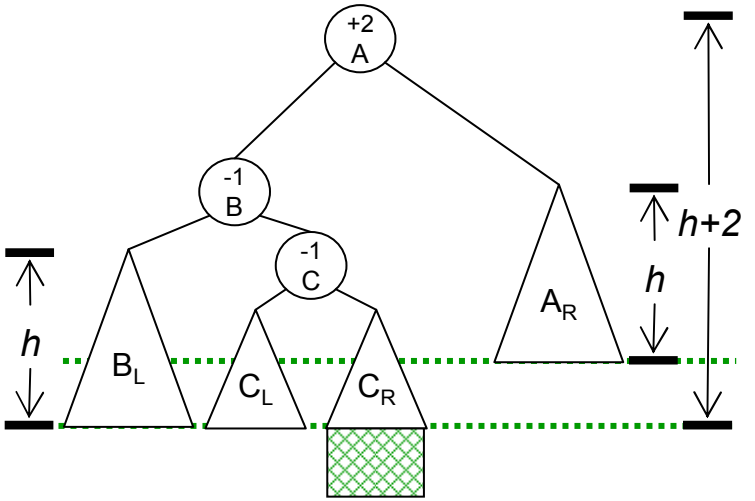


Rotação LR(c)

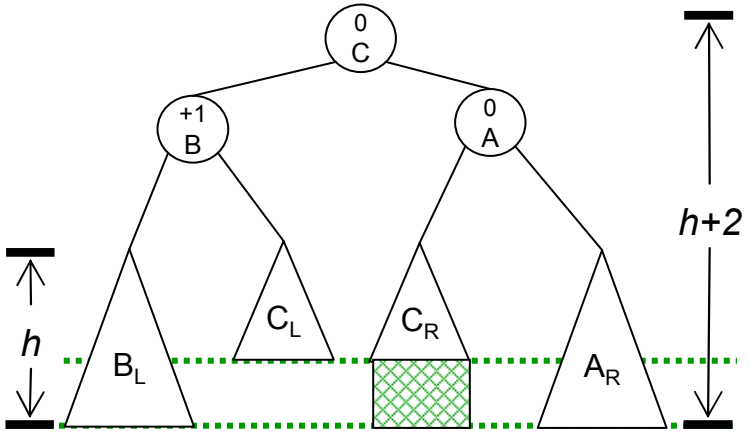
Subárvore balanceada



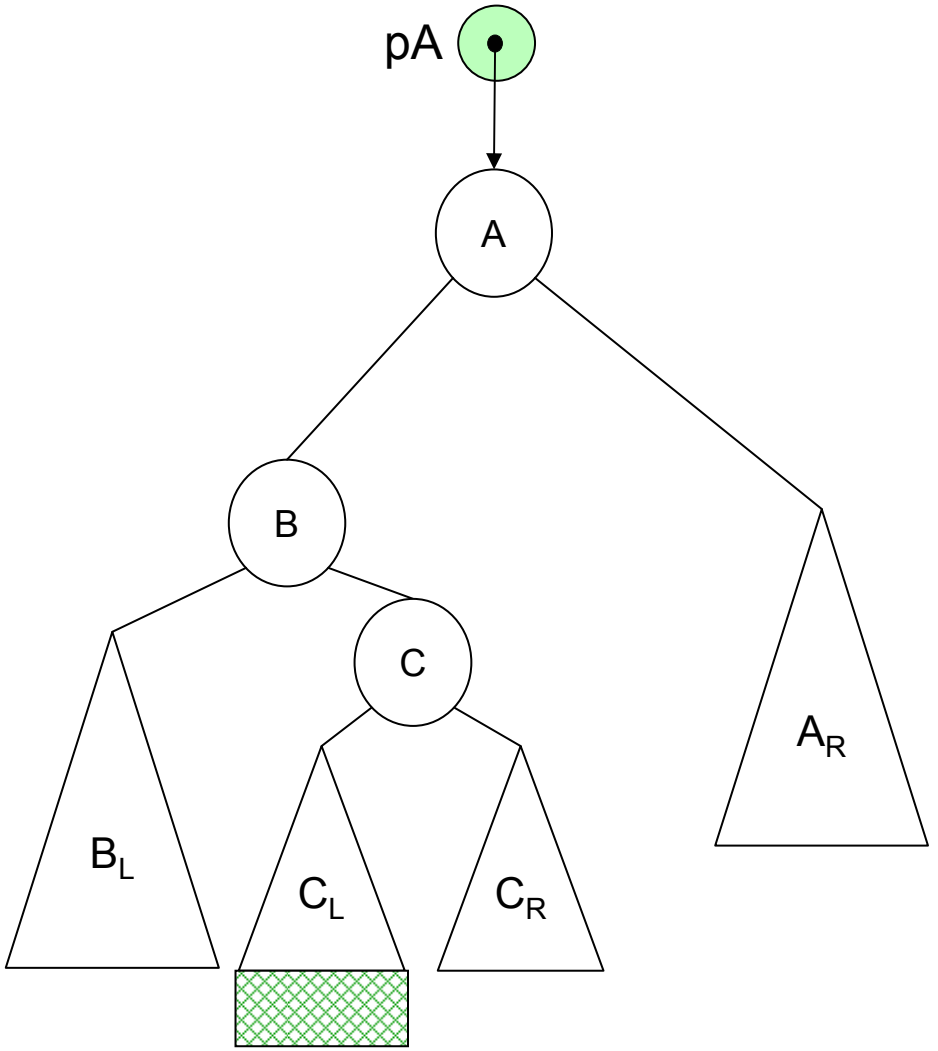
Subárvore desbalanceada após inserção



Subárvore rebalanceada

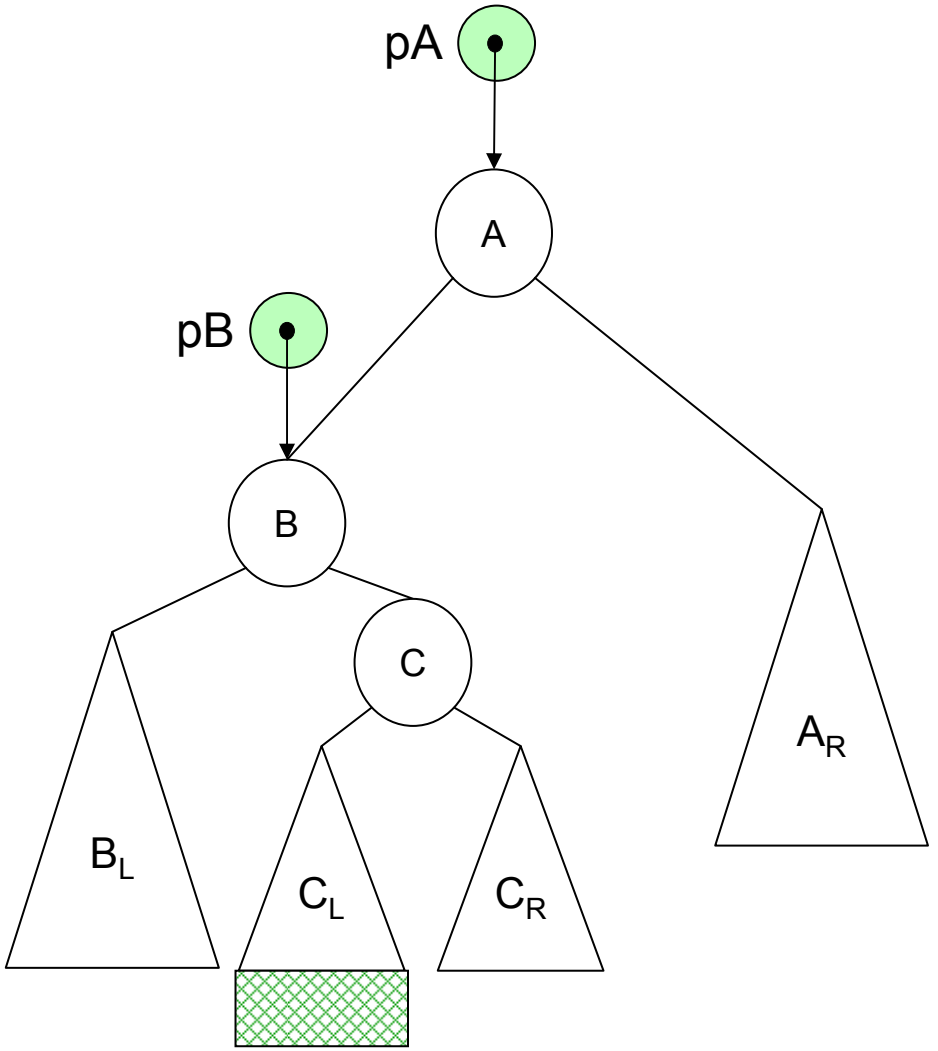


Rotação LR



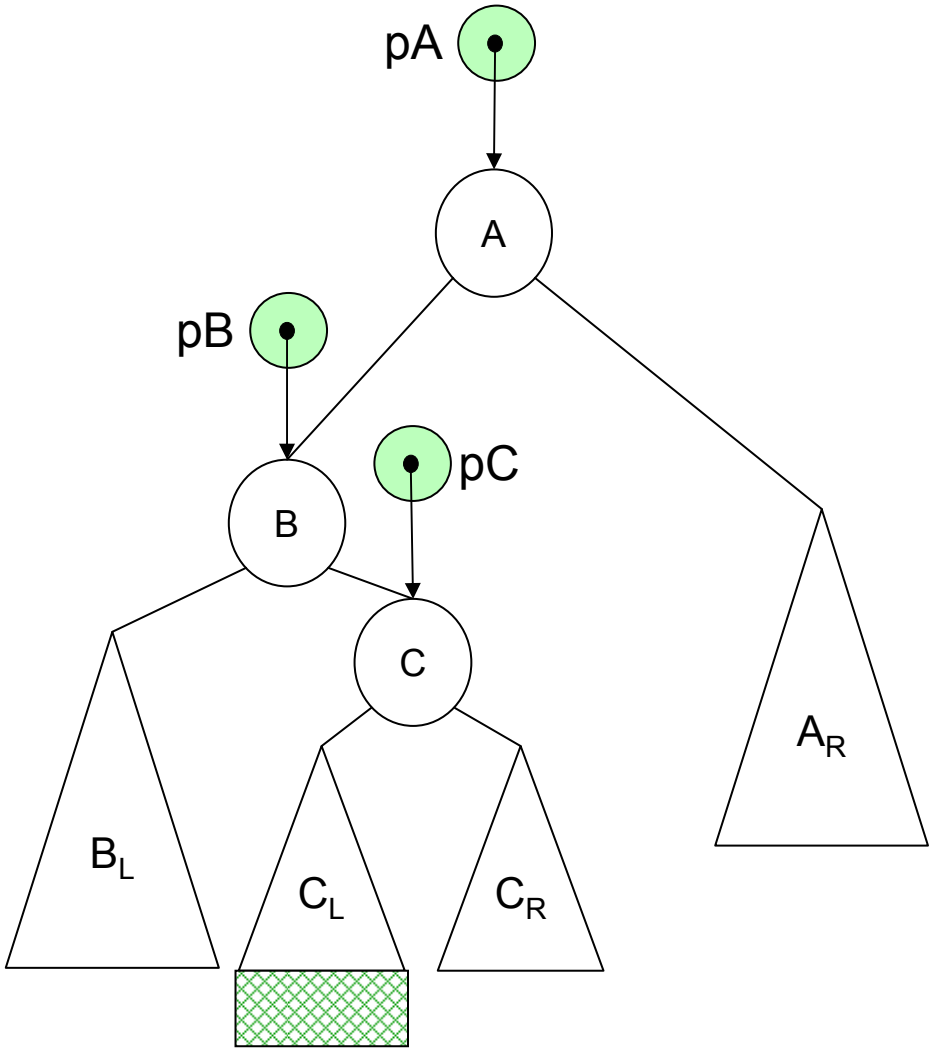
- Assumindo pA, pB e pC ponteiros para as subárvores com raízes A, B e C:
  - pB = pA->LeftNode;
  - pC = pB->RightNode;
  - pB->RightNode = pC->LeftNode;
  - pC->LeftNode = pB;
  - pA->LeftNode = pC->RightNode;
  - pC->RightNode = pA;
  - pA = pC;

Rotação LR



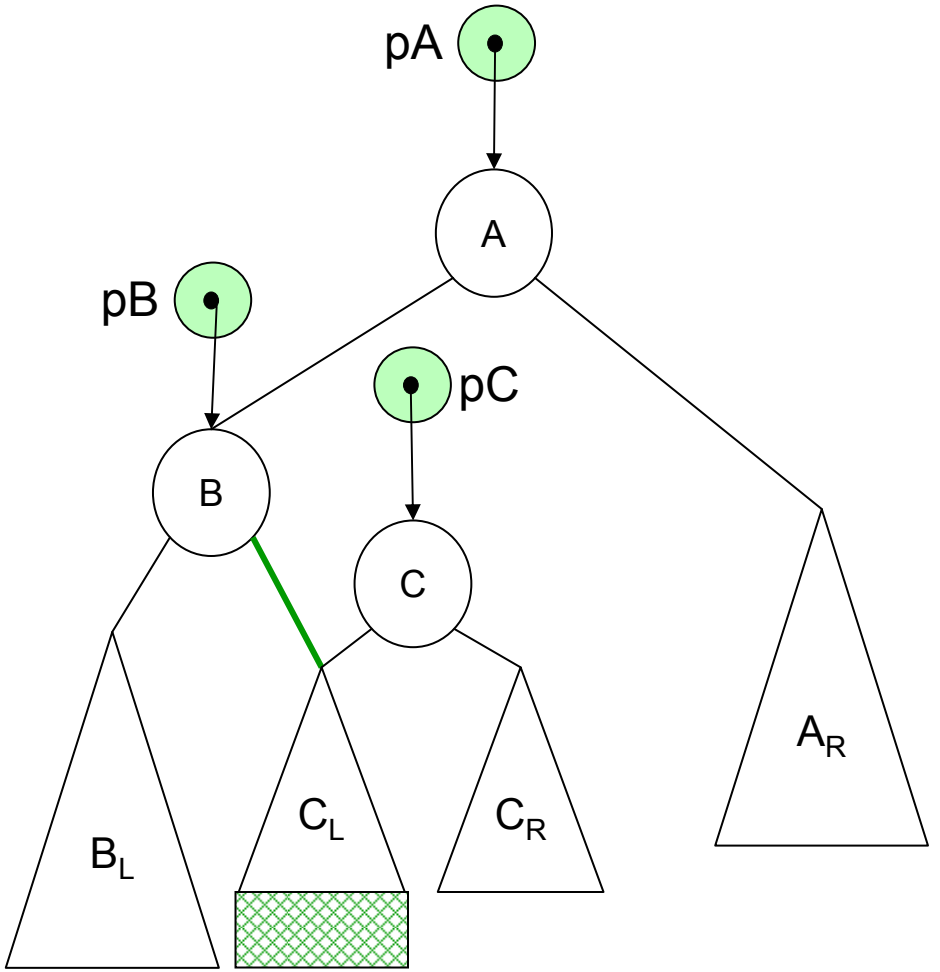
- Assumindo pA, pB e pC ponteiros para as subárvores com raízes A, B e C:
  - **pB = pA->LeftNode;**
  - pC = pB->RightNode;
  - pB->RightNode = pC->LeftNode;
  - pC->LeftNode = pB;
  - pA->LeftNode = pC->RightNode;
  - pC->RightNode = pA;
  - pA = pC;

Rotação LR



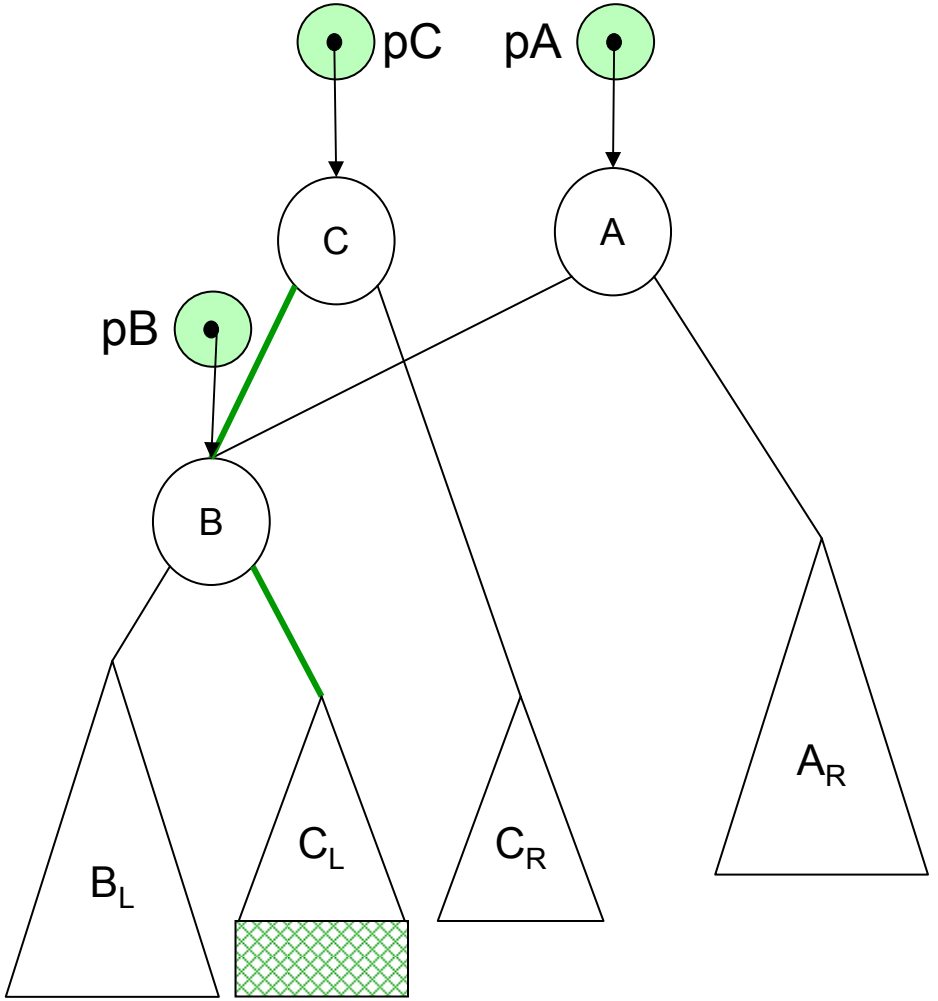
- Assumindo pA, pB e pC ponteiros para as subárvores com raízes A, B e C:
  - pB = pA->LeftNode;
  - **pC = pB->RightNode;**
  - pB->RightNode = pC->LeftNode;
  - pC->LeftNode = pB;
  - pA->LeftNode = pC->RightNode;
  - pC->RightNode = pA;
  - pA = pC;

Rotação LR



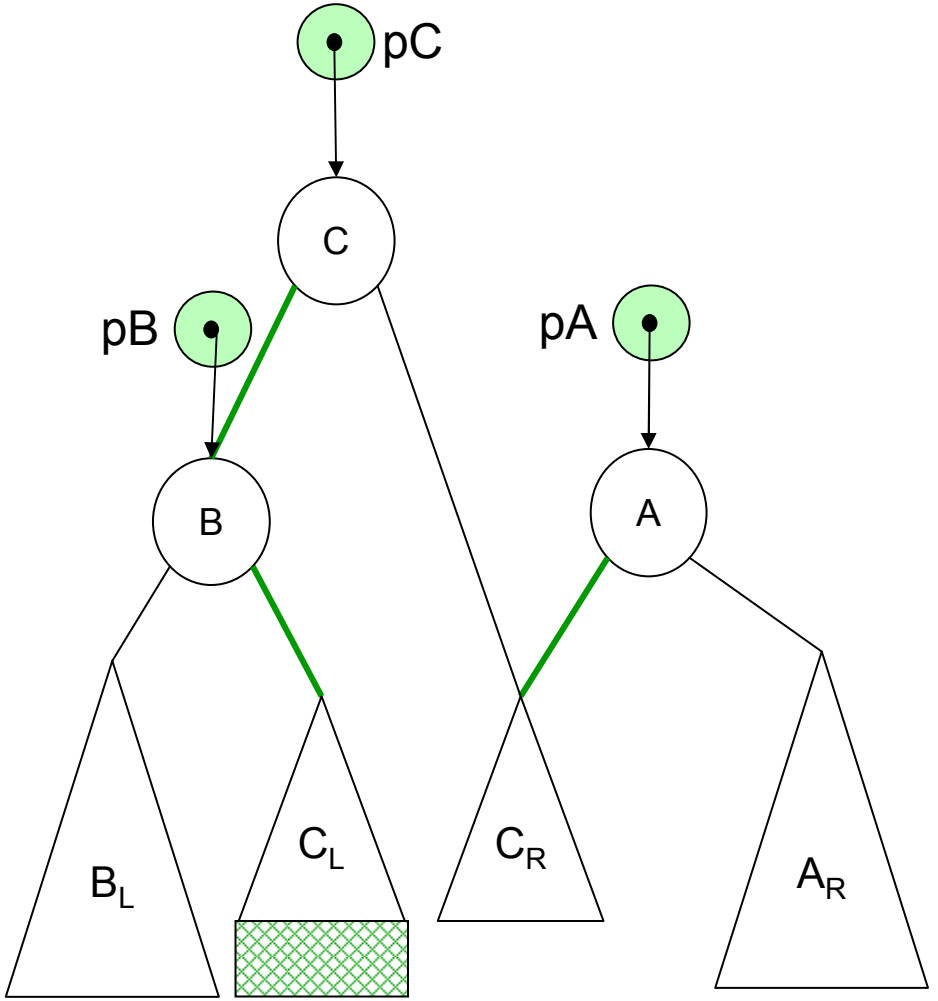
- Assumindo pA, pB e pC ponteiros para as subárvores com raízes A, B e C:
  - pB = pA->LeftNode;
  - pC = pB->RightNode;
  - **pB->RightNode = pC->LeftNode;**
  - pC->LeftNode = pB;
  - pA->LeftNode = pC->RightNode;
  - pC->RightNode = pA;
  - pA = pC;

Rotação LR



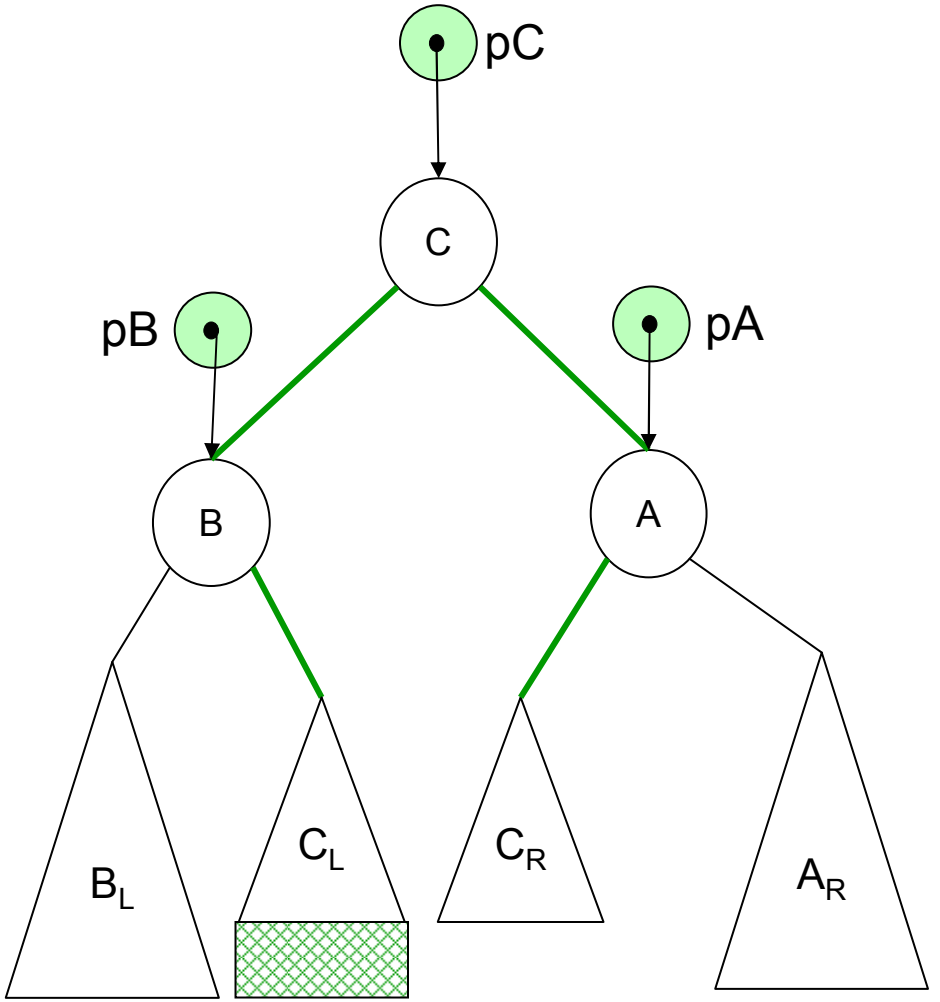
- Assumindo pA, pB e pC ponteiros para as subárvores com raízes A, B e C:
  - pB = pA->LeftNode;
  - pC = pB->RightNode;
  - pB->RightNode = pC->LeftNode;
  - **pC->LeftNode = pB;**
  - pA->LeftNode = pC->RightNode;
  - pC->RightNode = pA;
  - pA = pC;

Rotação LR



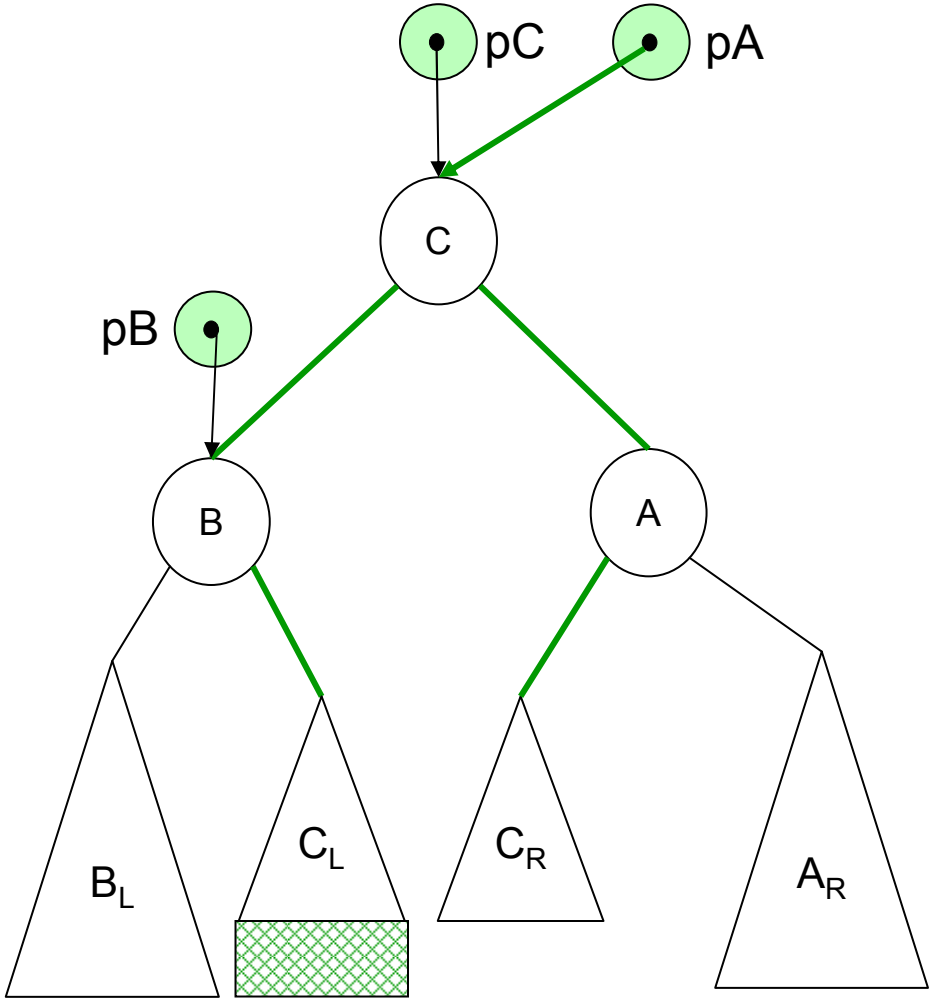
- Assumindo pA, pB e pC ponteiros para as subárvores com raízes A, B e C:
  - pB = pA->LeftNode;
  - pC = pB->RightNode;
  - pB->RightNode = pC->LeftNode;
  - pC->LeftNode = pB;
  - **pA->LeftNode = pC->RightNode;**
  - pC->RightNode = pA;
  - pA = pC;

Rotação LR



- Assumindo pA, pB e pC ponteiros para as subárvores com raízes A, B e C:
  - pB = pA->LeftNode;
  - pC = pB->RightNode;
  - pB->RightNode = pC->LeftNode;
  - pC->LeftNode = pB;
  - pA->LeftNode = pC->RightNode;
  - **pC->RightNode = pA;**
  - pA = pC;

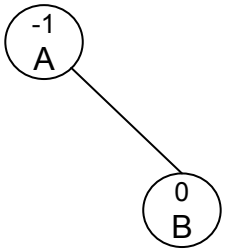
Rotação LR



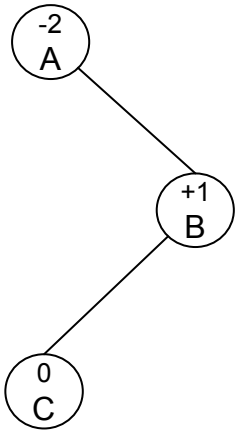
- Assumindo pA, pB e pC ponteiros para as subárvores com raízes A, B e C:
  - pB = pA->LeftNode;
  - pC = pB->RightNode;
  - pB->RightNode = pC->LeftNode;
  - pC->LeftNode = pB;
  - pA->LeftNode = pC->RightNode;
  - pC->RightNode = pA;
  - **pA = pC;**

Rotação RL(a)

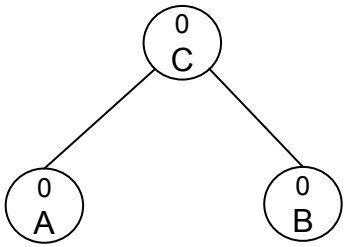
Subárvore balanceada



Subárvore desbalanceada após inserção

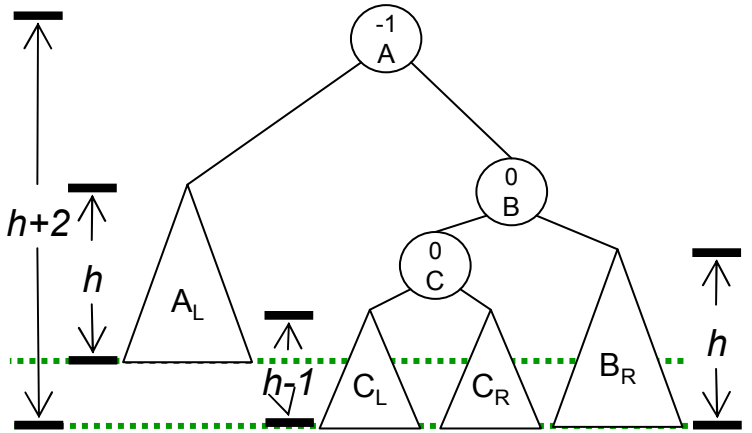


Subárvore rebalanceada

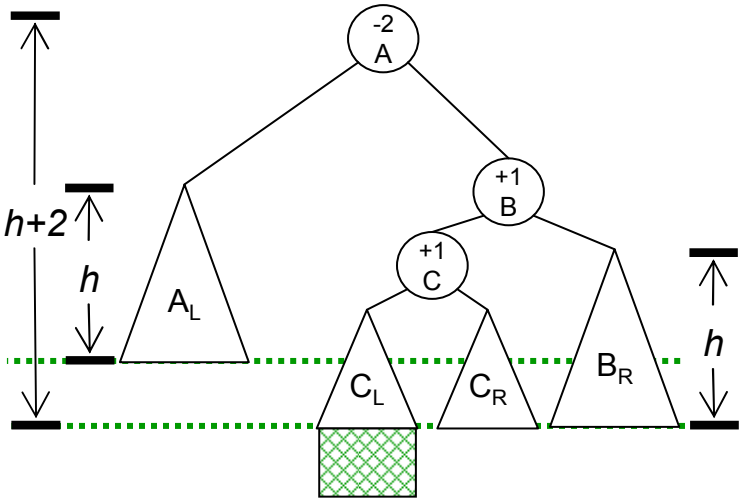


Rotação RL(b)

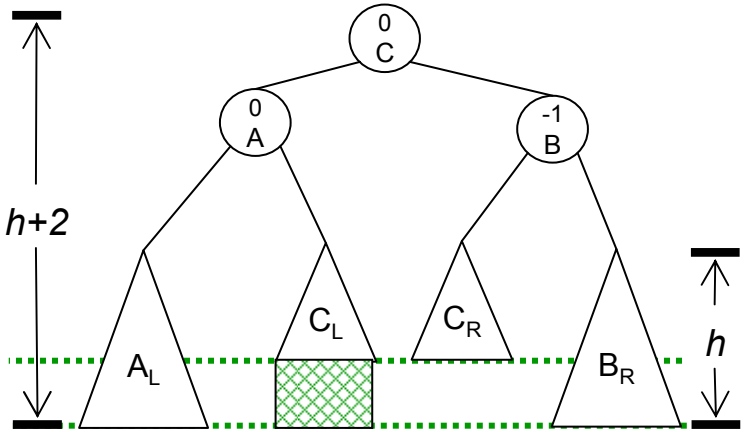
Subárvore balanceada



Subárvore desbalanceada após inserção

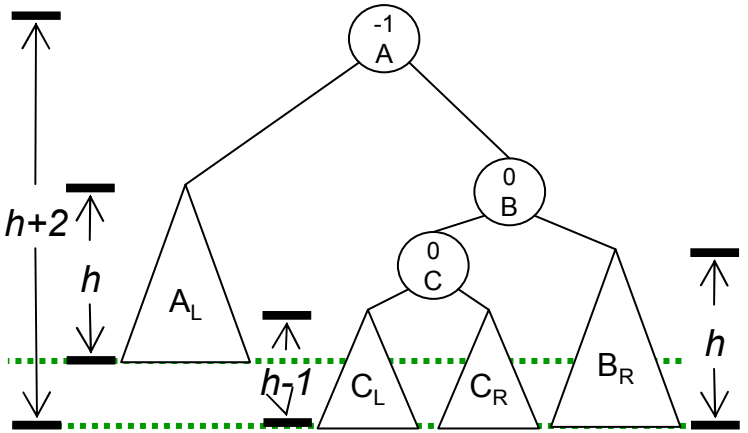


Subárvore rebalanceada

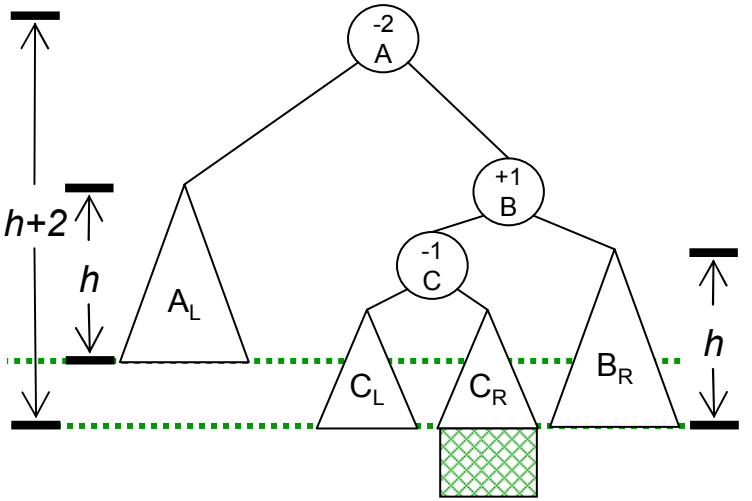


Rotação RL(c)

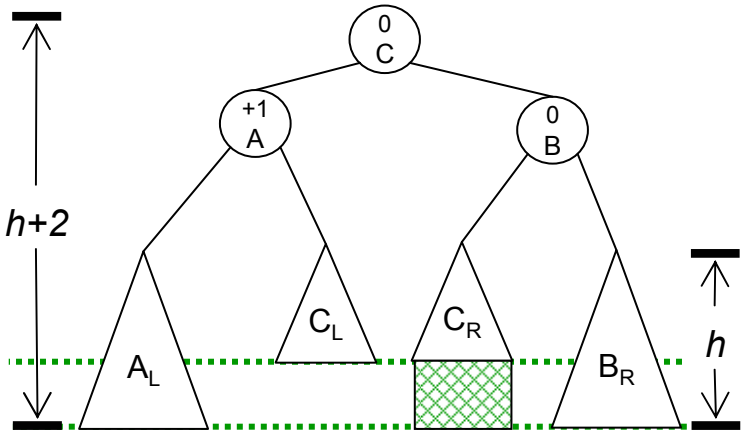
Subárvore balanceada



Subárvore desbalanceada após inserção

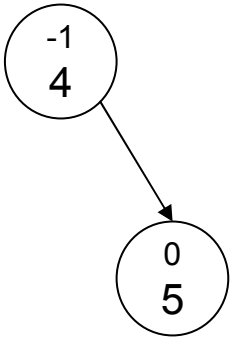


Subárvore rebalanceada



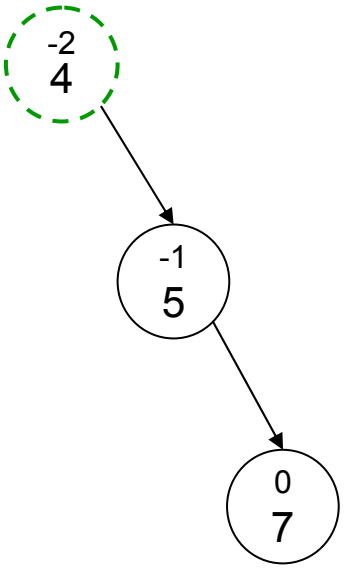
Inserções

- Inserir x=7



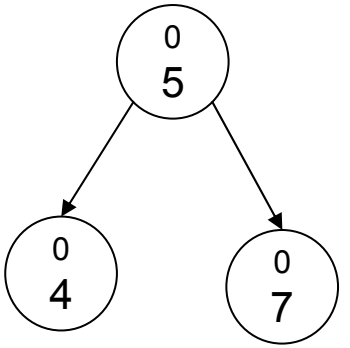
Inserções

- Inserido x=7
- A inserção produz uma árvore desbalanceada...



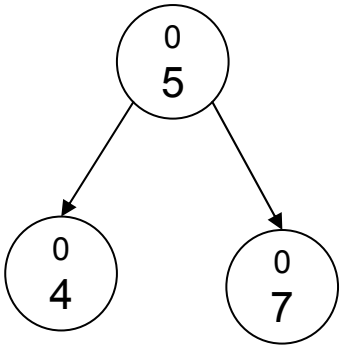
Inserções

- Inserido x=7
- A inserção produz uma árvore desbalanceada, cujo balanceamento envolve uma rotação RR



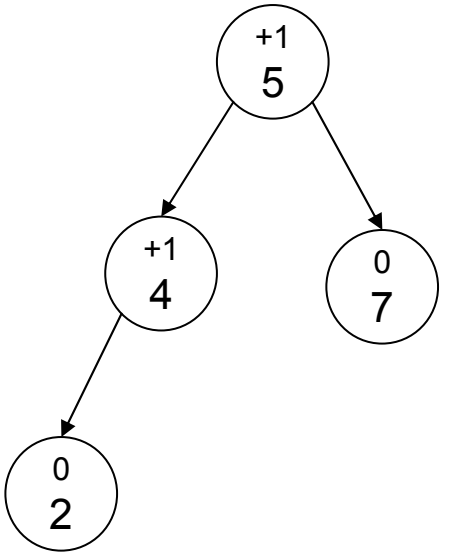
Inserções

- Inserir x=2



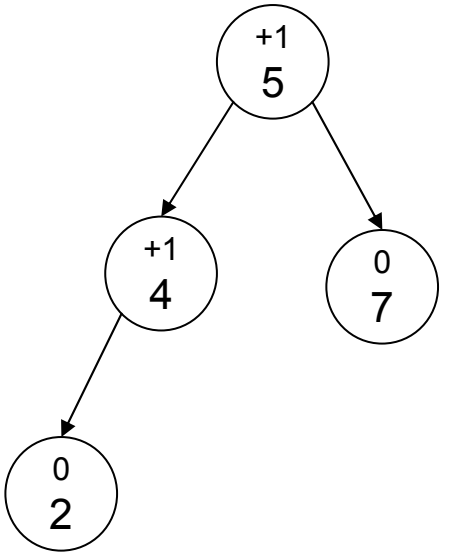
Inserções

- Inserido x=2



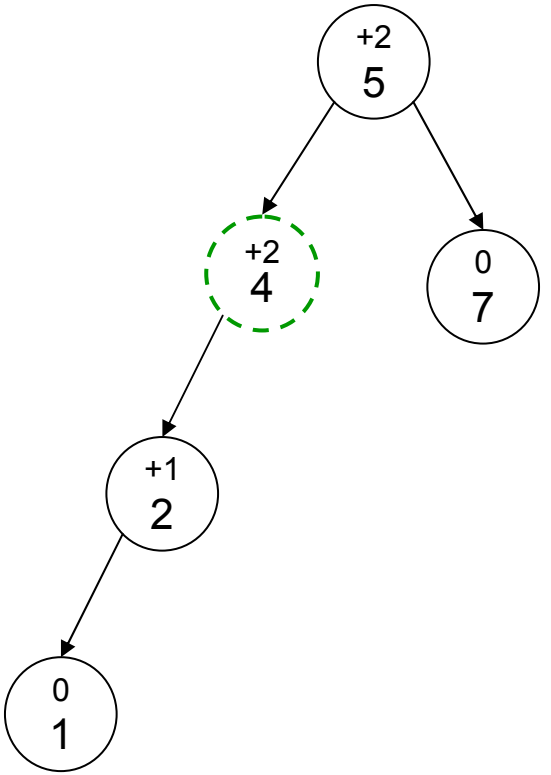
Inserções

- Inserido x=2
- Inserir x=1



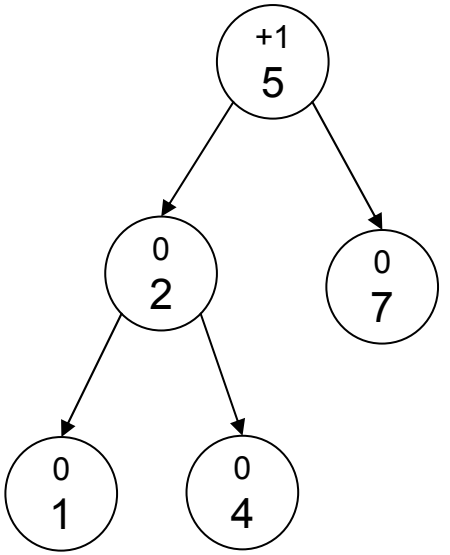
Inserções

- Inserido x=2
- Inserido x=1
- Ocorre desbalanceamento da subárvore de raiz 4...



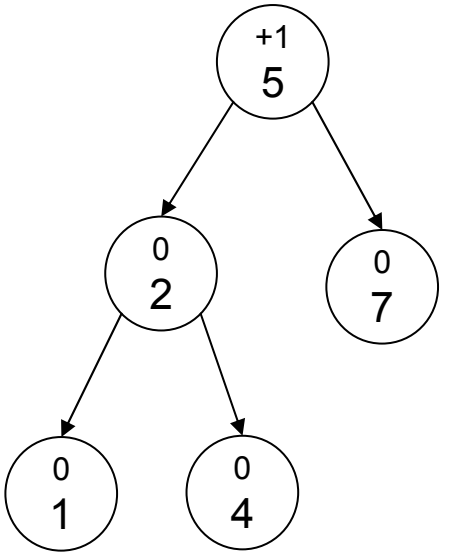
Inserções

- Inserido x=2
- Inserido x=1
- Ocorre desbalanceamento da subárvore de raiz 4, que é corrigido por uma rotação LL



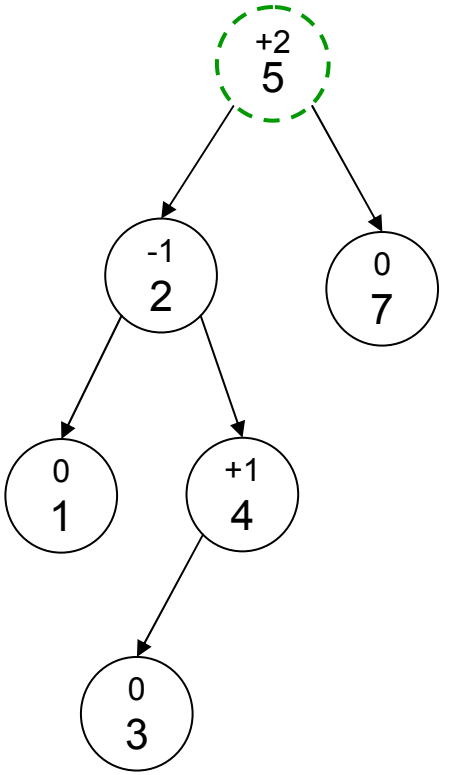
Inserções

- Inserir x=3



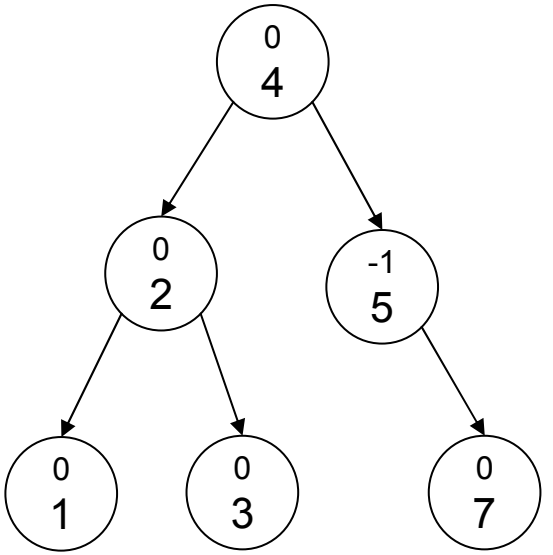
Inserções

- Inserido x=3
- Ocorre desbalanceamento da subárvore de raiz 5...



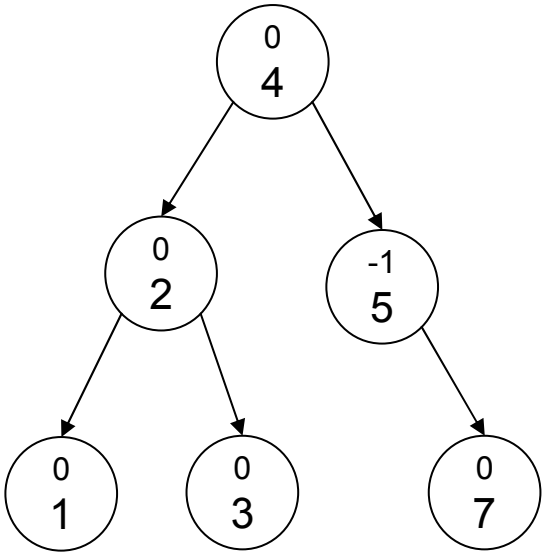
Inserções

- Inserido x=3
- Ocorre desbalanceamento da subárvore de raiz 5, que é corrigido por uma rotação LR



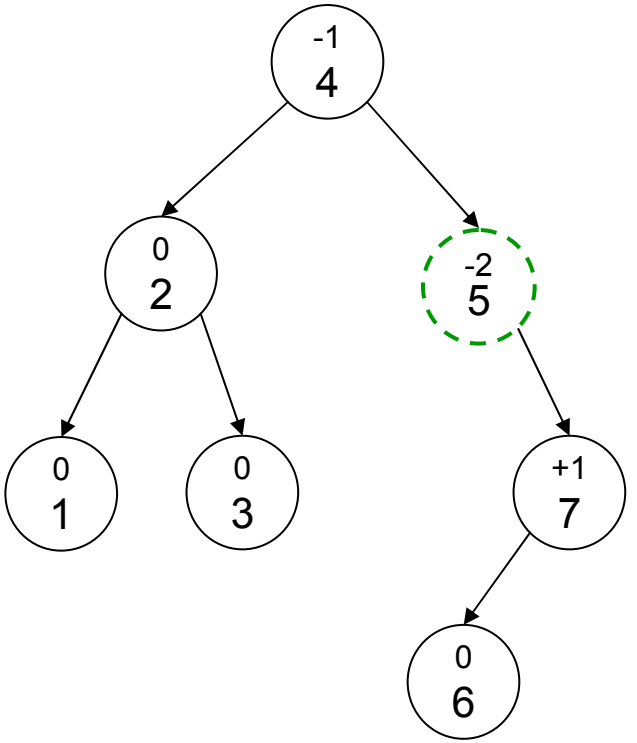
Inserções

- Inserir x=6



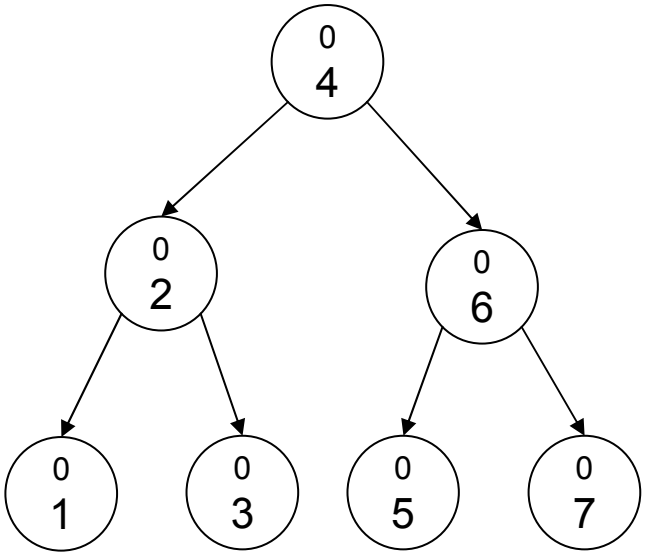
Inserções

- Inserido x=6
- Ocorre desbalanceamento da subárvore de raiz 5...



Inserções

- Inserido x=6
- Ocorre desbalanceamento da subárvore de raiz 5, que é corrigido por uma rotação RL



Representação

- Vamos representar um nó AVL de forma similar a um nó em uma árvore binária de busca, contendo um campo adicional **bal** para manter o fator de balanceamento do nó:

```
struct TreeNode
{ int Entry;          // chave
  int count;          // contador
  int bal;            // -1, 0, +1
  TreeNode *LeftNode, *RightNode; //subárvores
};
typedef TreeNode *TreePointer;
```

Busca com Inserção

- Para realizar as operações necessárias em cada nó isolado é descrito a seguir um algoritmo recursivo
- Por ser recursivo, é simples a incorporação de uma operação adicional a ser executada no caminho de volta ao longo do trajeto de busca
- A cada passo é necessário descobrir e informar se a altura da subárvore em que foi feita a inserção cresceu ou não
- Para tanto, foi incluído um parâmetro booleano **h**, passado por referência (com valor inicial **false**), que indica que a *subárvore cresceu em altura*
- O algoritmo de inserção deve ser chamado com o ponteiro **pA** como sendo a raiz da árvore (**root**)

Busca com Inserção

- Suponha que o algoritmo retorne, partindo da subárvore esquerda a um nó **p** com a indicação que houve um incremento em sua altura
- Existem três condições relacionadas com as alturas das subárvores **antes** da inserção
  - $h_L < h_R$, p->bal == -1
    - Após a inserção, o desbalanceamento em **p** foi equilibrado
  - $h_L == h_R$, p->bal == 0
    - Após a inserção, a altura da árvore é maior à esquerda
  - $h_L > h_R$, p->bal == +1
    - Após a inserção, é necessário fazer o rebalanceamento

Busca com Inserção

- Após uma rotação LL ou RR, os nós **A** e **B** passam a ter fator de balanceamento iguais a zero
- No caso de rotações LR ou RL
  - Os fatores de balanceamento dos nós **A** e **B** podem ser recalculados com base no fator de balanceamento do nó **C**
  - O novo fator de balanceamento do nó **C** passa a ser zero

Busca com Inserção

```
void AVLTree::SearchInsert(int x, TreePointer &pA, bool &h)
{ TreePointer pB, pC;

  if(pA == NULL) // inserir
  { pA = new TreeNode;
    h = true;
    pA->Entry = x;
    pA->count = 1;
    pA->LeftNode = pA->RightNode = NULL;
    pA->bal = 0;
  }
```

Busca com Inserção

```
else
  if(x < pA->Entry)
  { SearchInsert(x, pA->LeftNode, h);
    if(h)                              // subárvore esquerda cresceu
    { switch (pA->bal)
      { case -1: pA->bal = 0; h = false; break;
        case  0: pA->bal = +1;           break;
        case +1: pB = pA->LeftNode;
                 if(pB->bal == +1)  // rotação LL
                 { pA->LeftNode = pB->RightNode;  pB->RightNode = pA;
                   pA->bal = 0;                   pA = pB;
                 }
                 else               // rotação LR
                 { pC = pB->RightNode;   pB->RightNode = pC->LeftNode;
                   pC->LeftNode  = pB;   pA->LeftNode  = pC->RightNode;
                   pC->RightNode = pA;
                   if(pC->bal == +1) pA->bal = -1; else pA->bal = 0;
                   if(pC->bal == -1) pB->bal = +1; else pB->bal = 0;
                   pA = pC;
                 }
                 pA->bal = 0; h = false;
      } // switch
    }
  }
```

Busca com Inserção

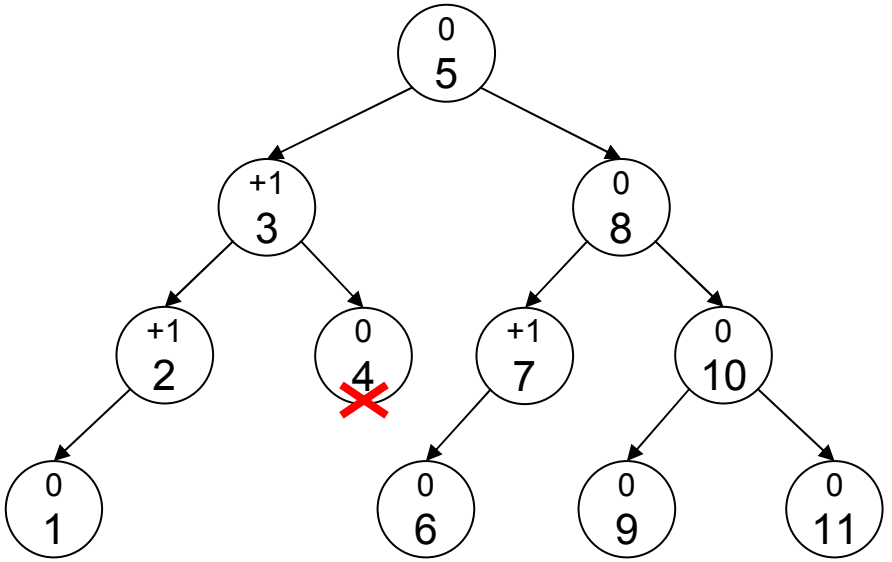
```
else
  if(x > pA->Entry)
  { SearchInsert(x, pA->RightNode, h);
    if(h)                                  // subárvore direita cresceu
    {  switch (pA->bal)
       { case +1: pA->bal = 0; h = false; break;
         case  0: pA->bal = -1;           break;
         case -1: pB = pA->RightNode;
                  if(pB->bal == -1)  // rotação RR
                  { pA->RightNode = pB->LeftNode;  pB->LeftNode = pA;
                    pA->bal = 0;                   pA = pB;
                  }
                  else               // rotação RL
                  { pC = pB->LeftNode;    pB->LeftNode  = pC->RightNode;
                    pC->RightNode = pB;   pA->RightNode = pC->LeftNode;
                    pC->LeftNode  = pA;
                    if(pC->bal == -1) pA->bal = +1; else pA->bal = 0;
                    if(pC->bal == +1) pB->bal = -1; else pB->bal = 0;
                    pA = pC;
                  }
                  pA->bal = 0; h = false;
       } // switch
    }
  }
```

Busca com Inserção

```
        else // elemento encontrado
          pA->count++;
}
```

Remoção

Antes da remoção

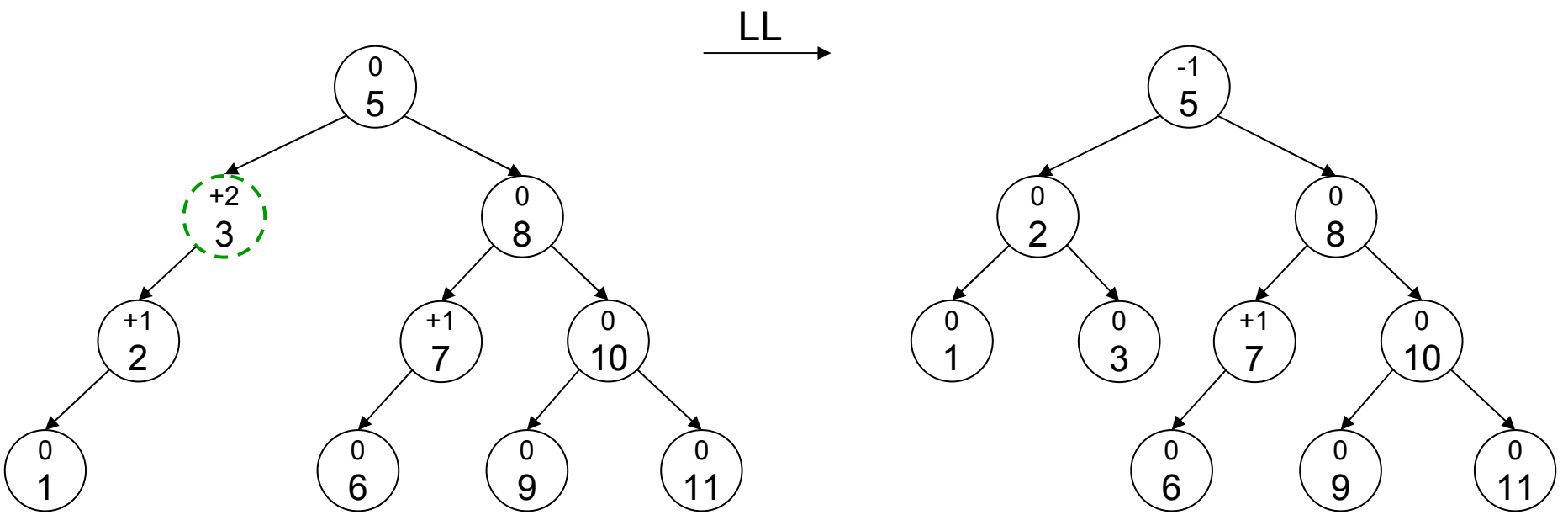


Depois do rebalanceamento

Remoção

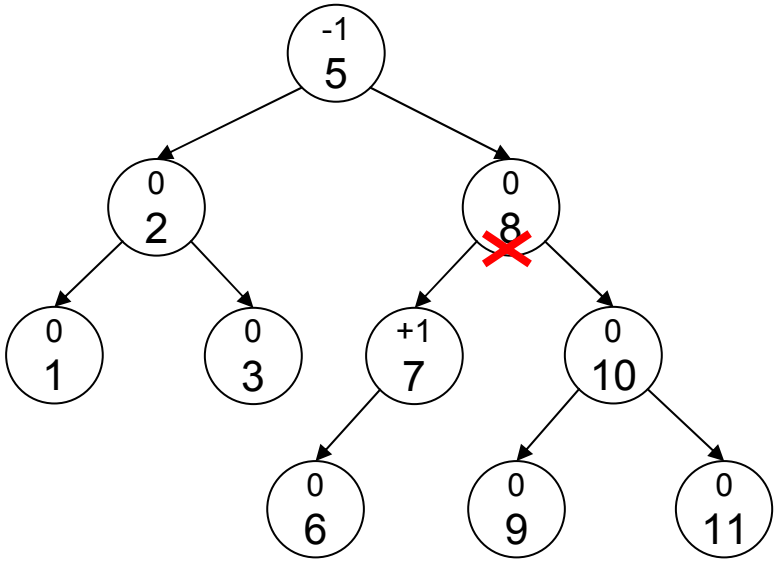
Depois da remoção

Depois do rebalanceamento



Remoção

Antes da remoção

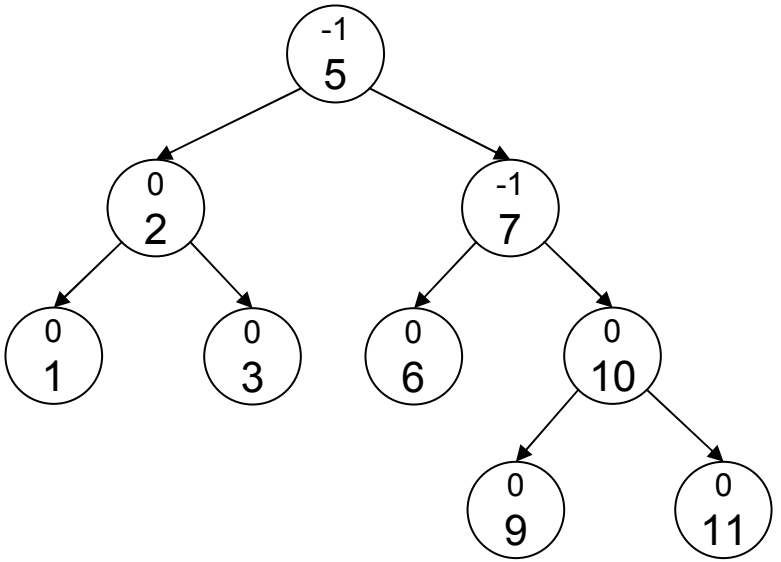


Depois do rebalanceamento

Obs: foi utilizado o maior elemento da subárvore esquerda do nó sendo removido

Remoção

Depois da remoção



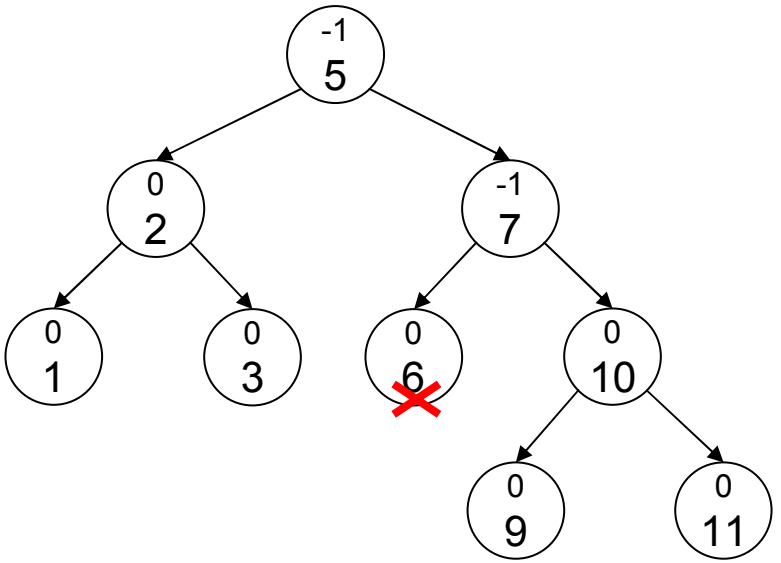
Depois do rebalanceamento

Sem necessidade de rebalanceamento

Obs: foi utilizado o maior elemento da subárvore esquerda do nó sendo removido

Remoção

Antes da remoção

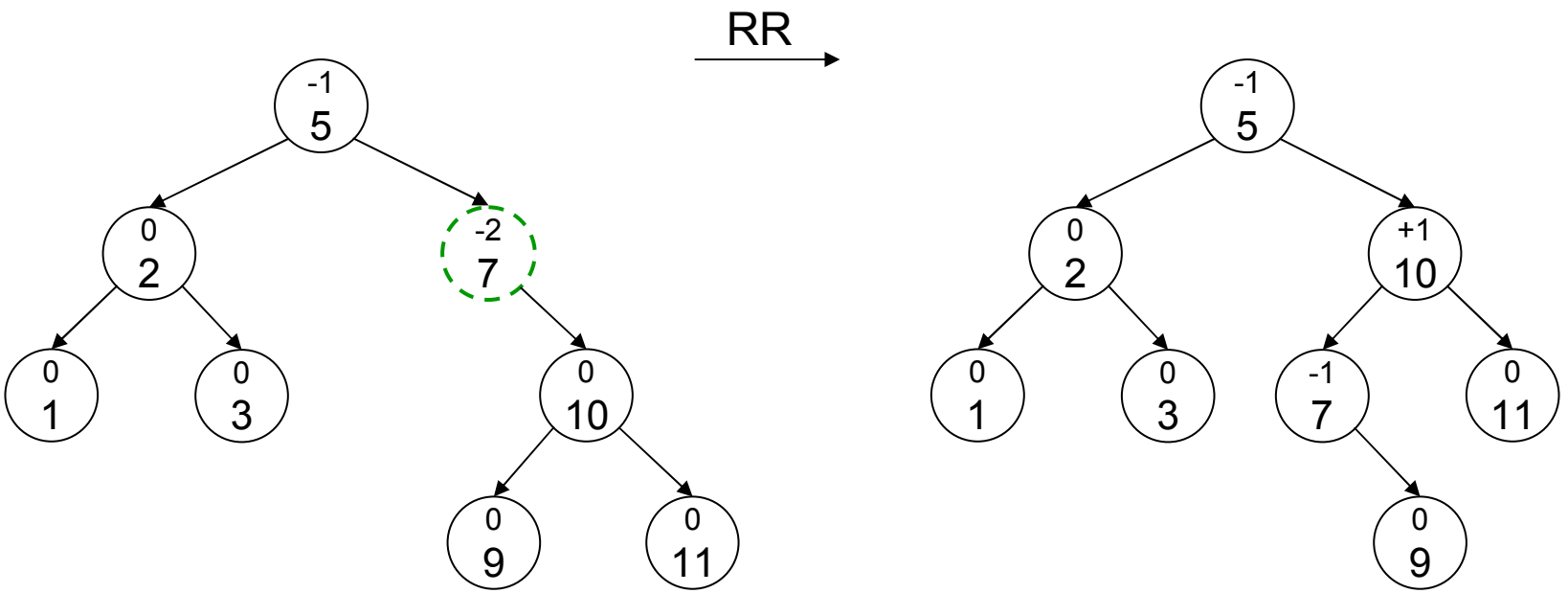


Depois do rebalanceamento

Remoção

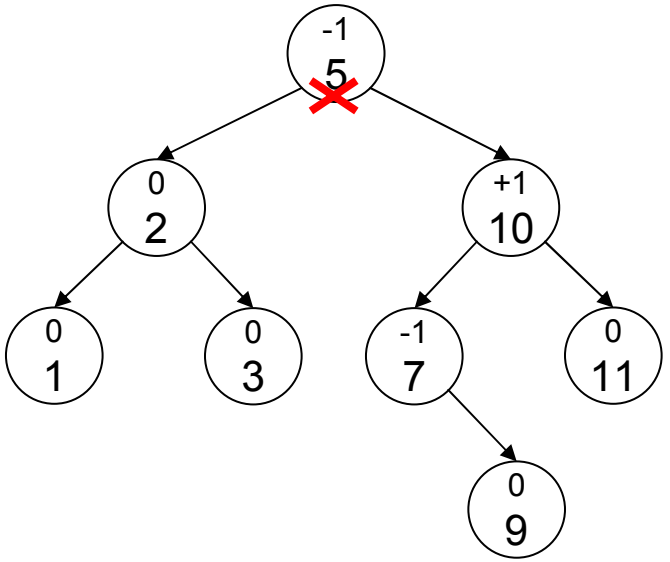
Depois da remoção

Depois do rebalanceamento



Remoção

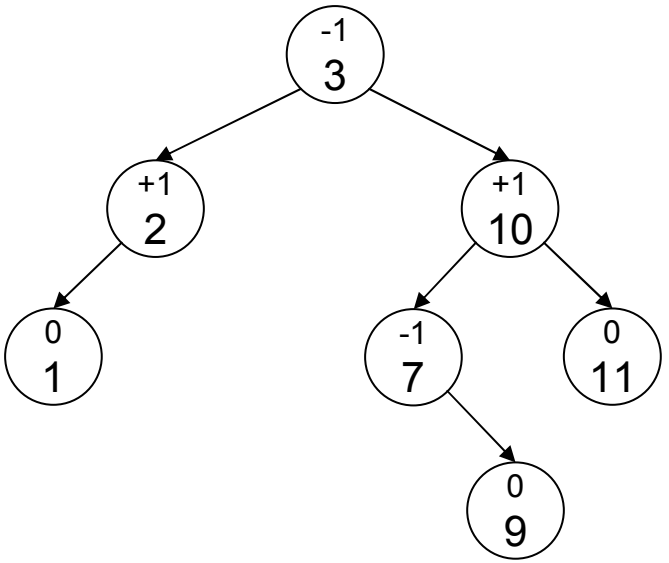
Antes da remoção



Depois do rebalanceamento

Remoção

Depois da remoção

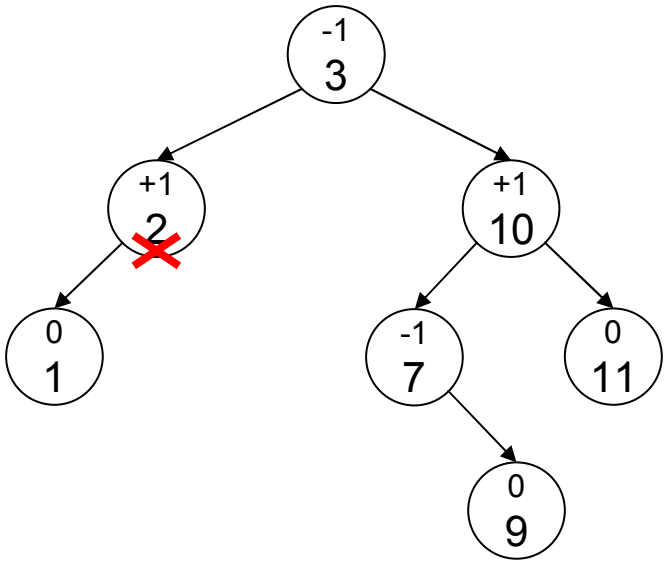


Depois do rebalanceamento

Sem necessidade de rebalanceamento

Remoção

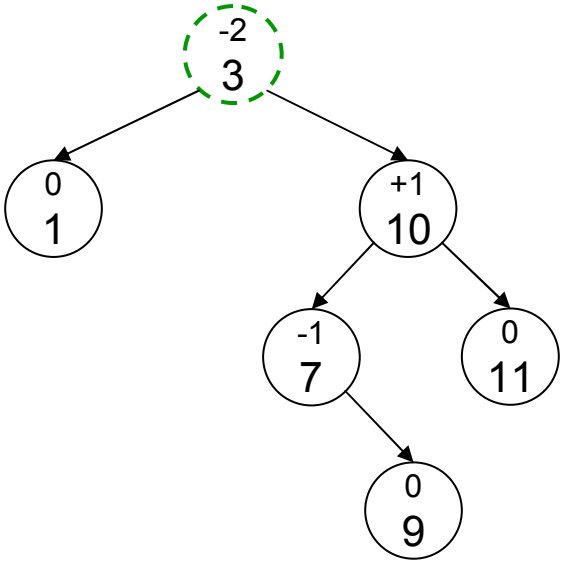
Antes da remoção



Depois do rebalanceamento

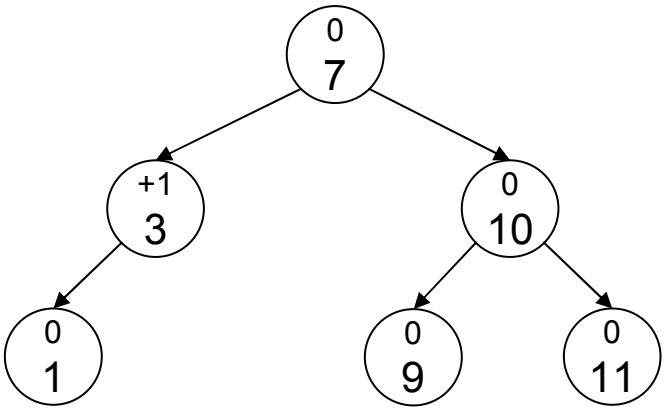
Remoção

Depois da remoção



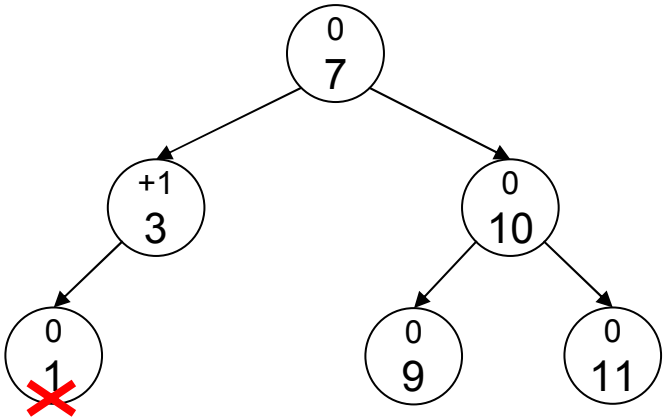
RL

Depois do rebalanceamento

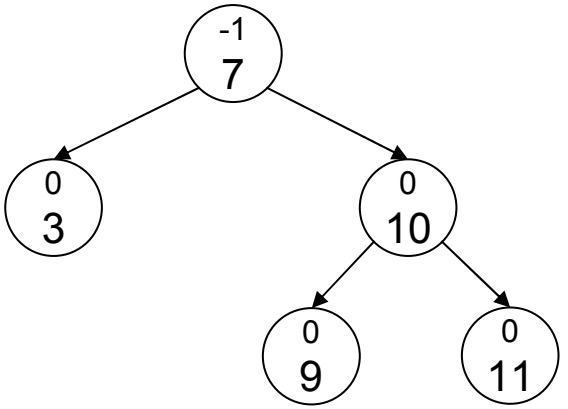


Remoção

Antes da remoção

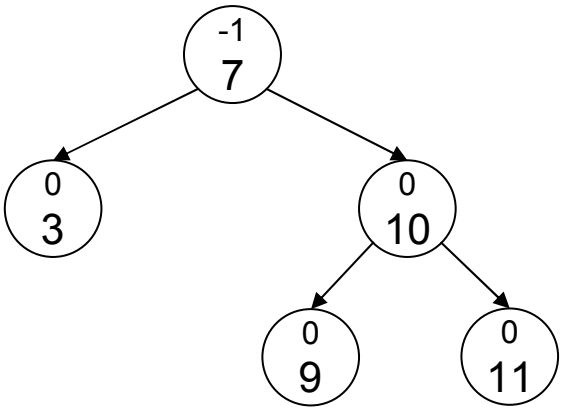


Depois do rebalanceamento



Remoção

Depois da remoção

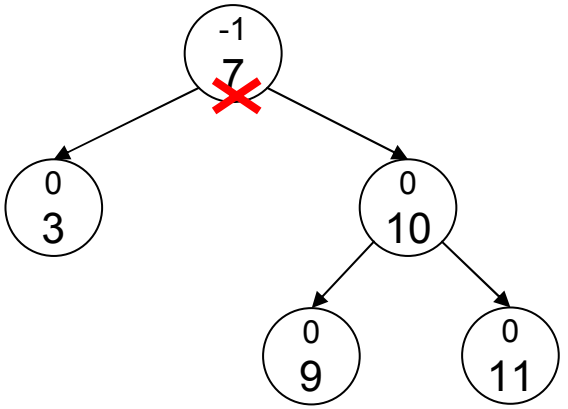


Depois do rebalanceamento

Sem necessidade de rebalanceamento

Remoção

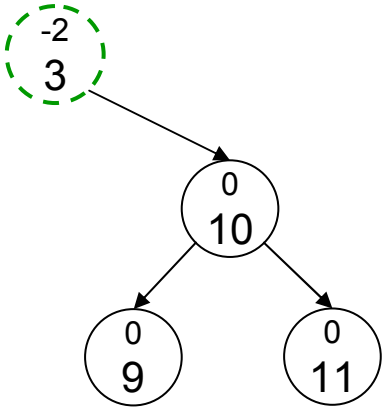
Antes da remoção



Depois do rebalanceamento

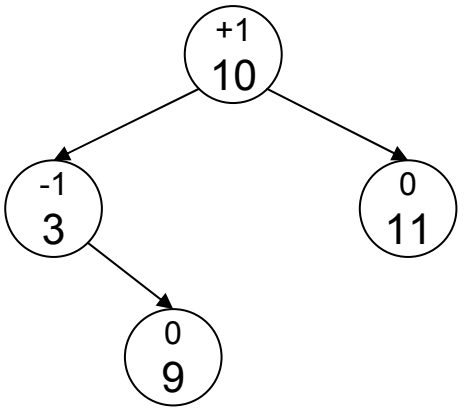
Remoção

Depois da remoção



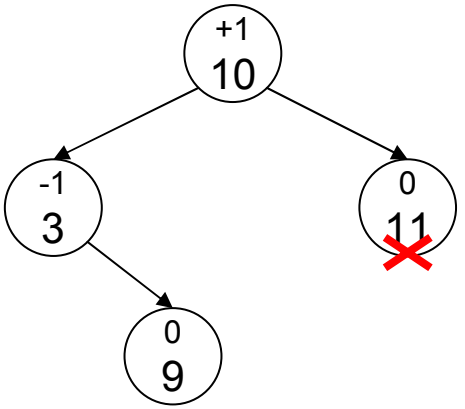
RR

Depois do rebalanceamento



Remoção

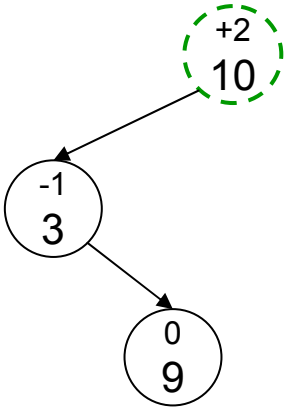
Antes da remoção



Depois do rebalanceamento

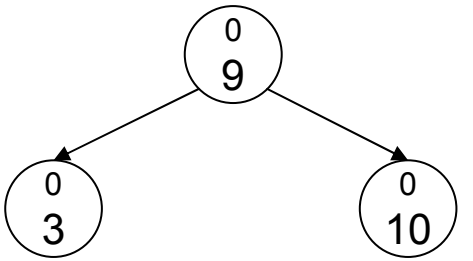
Remoção

Antes da remoção



LR

Depois do rebalanceamento



Remoção

- A remoção em árvores AVL é similar à remoção em uma árvore binária de busca
- Todavia, é preciso verificar o balanceamento e, se necessário, aplicar algumas das rotações
- Nos algoritmos, foi acrescentado um parâmetro **h**, passado por referência (cujo valor inicial deve ser **false**) indicando que a *altura da subárvore foi reduzida*
- O método DelMin procura, na subárvore direita, pelo menor valor e só é chamando quando o nó com chave **x** possui as duas subárvores
- O rebalanceamento somente é efetuado se **h** é **true**
- O algoritmo de remoção deve ser chamado com o ponteiro **p** como sendo a raiz da árvore (**root**)

Remoção

```
void AVLTree::Delete(int x,
   TreePointer &p, bool &h)
{ TreePointer q;

   if(p == NULL)
   { cout << "Elemento inexistente";
     abort();
   }
   if(x < p->Entry)
   {  Delete(x,p->LeftNode,h);
      if(h)
        balanceL(p,h);
   }
   else
     if(x > p->Entry)
     {  Delete(x,p->RightNode,h);
        if(h)
          balanceR(p,h);
     }
     else // remover p->
       { q = p;
         if(q->RightNode == NULL)
         { p = q->LeftNode;
           h = true;
         }
         else
           if(q->LeftNode == NULL)
           { p = q->RightNode;
             h = true;
           }
           else
           { DelMin(q,q->RightNode,h);
             if(h)
               balanceR(p,h);
           }
         delete q;
       }
}
```

Remoção

```
void AVLTree::DelMin(TreePointer &q, TreePointer &r,
  bool &h)
{
   if(r->LeftNode != NULL)
   { DelMin(q, r->LeftNode, h);
      if(h)
        balanceL(r,h);
   }
   else
   { q->Entry = r->Entry;
     q->count = r->count;
     q = r;
     r = r->RightNode;
     h = true;
   }
}
```

Remoção

```
void AVLTree::balanceL(TreePointer &pA, bool &h)
{ TreePointer pB, pC;
  int balB, balC;
  // subarvore esquerda encolheu
  switch(pA->bal)
  { case +1: pA->bal = 0;              break;
    case  0: pA->bal = -1; h = false; break;
    case -1:
      pB = pA->RightNode; balB = pB->bal;
      if(balB <= 0) // rotacao RR
      { pA->RightNode = pB->LeftNode;
        pB->LeftNode = pA;
        if(balB == 0)
        { pA->bal = -1; pB->bal = +1; h = false; }
        else
        { pA->bal = 0;  pB->bal = 0; }
        pA = pB;
      }
      else           // rotacao RL
      { pC = pB->LeftNode; balC = pC->bal;
        pB->LeftNode = pC->RightNode;
        pC->RightNode = pB;
        pA->RightNode = pC->LeftNode;
        pC->LeftNode = pA;
        if(balC==-1) pA->bal=+1; else pA->bal=0;
        if(balC==+1) pB->bal=-1; else pB->bal=0;
        pA = pC;  pC->bal = 0;
      }
  }
}
```

```
void AVLTree::balanceR(TreePointer &pA, bool &h)
{ TreePointer pB, pC;
  int balB, balC;
  // subarvore direita encolheu
  switch(pA->bal)
  { case -1: pA->bal = 0;              break;
    case  0: pA->bal = +1; h = false; break;
    case +1:
      pB = pA->LeftNode; balB = pB->bal;
      if(balB >= 0) // rotacao LL
      { pA->LeftNode = pB->RightNode;
        pB->RightNode = pA;
        if(balB == 0)
        { pA->bal = +1; pB->bal = -1; h = false; }
        else
        { pA->bal = 0;  pB->bal = 0; }
        pA = pB;
      }
      else           // rotacao LR
      { pC = pB->RightNode; balC = pC->bal;
        pB->RightNode = pC->LeftNode;
        pC->LeftNode = pB;
        pA->LeftNode = pC->RightNode;
        pC->RightNode = pA;
        if(balC==+1) pA->bal=-1; else pA->bal=0;
        if(balC==-1) pB->bal=+1; else pB->bal=0;
        pA = pC;  pC->bal = 0;
      }
  }
}
```

Resumo

- Há um custo adicional para manter uma árvore balanceada, mesmo assim garantindo $O(\log_2 \mathbf{n})$, 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

Slides baseados em:

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 elaborado por
José Augusto Baranauskas
Elaboração inicial 2006; Revisão atual 2007