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

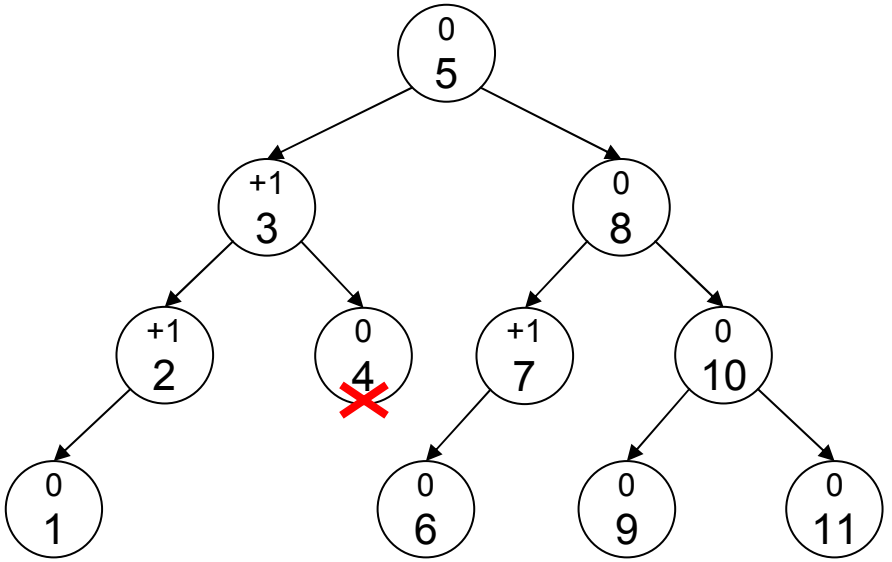
```
else
  if(x > pA->Entry)
  { SearchInsert(x, pA->RightNode, h);
    if(h)                                  // subárvore direita cresceu
    {  switch (pA->bal)
       { case +1: pA->bal = 0; h = false; break;
         case  0: pA->bal = -1;           break;
         case -1: pB = pA->RightNode;
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

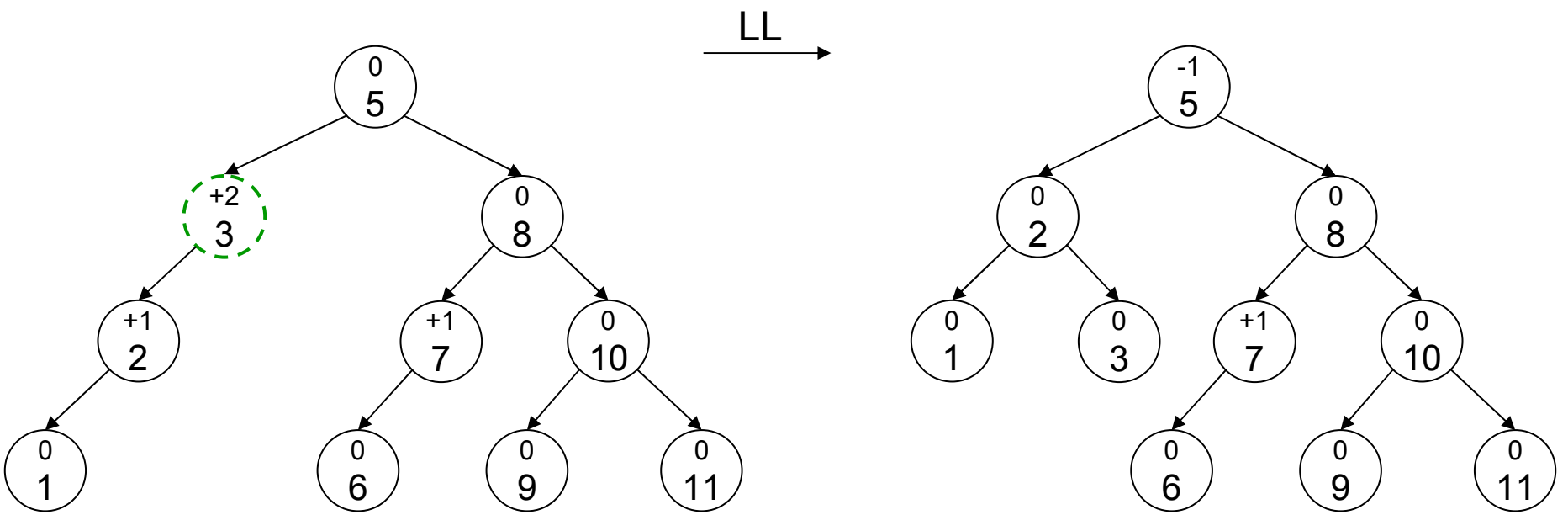


Depois do rebalanceamento

Remoção

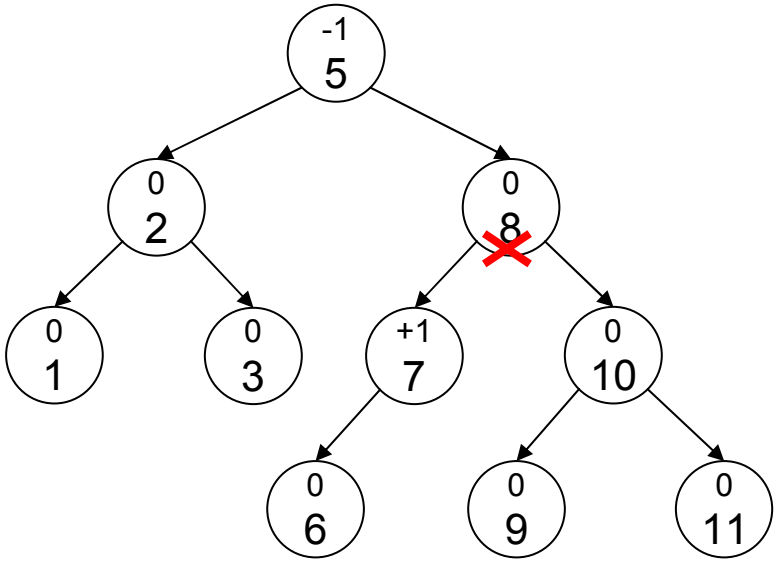
Depois da remoção

Depois do rebalanceamento



Remoção

Antes da remoção

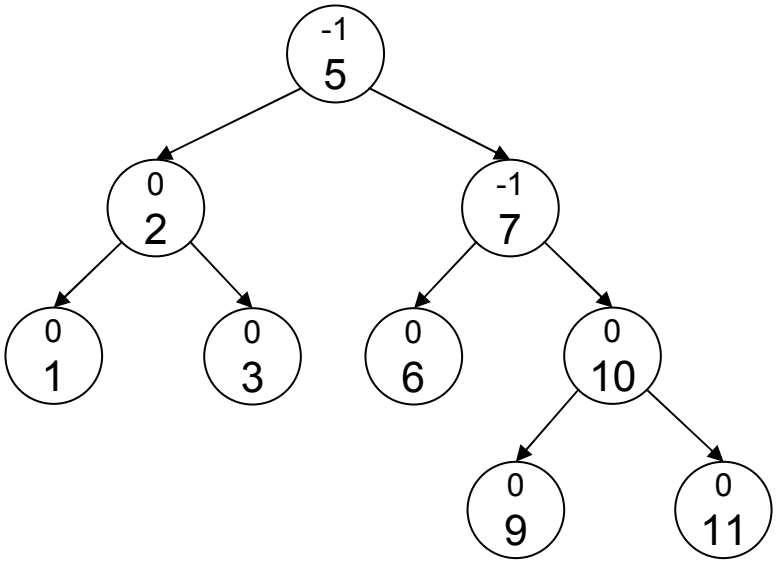


Depois do rebalanceamento

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

Remoção

Depois da remoção



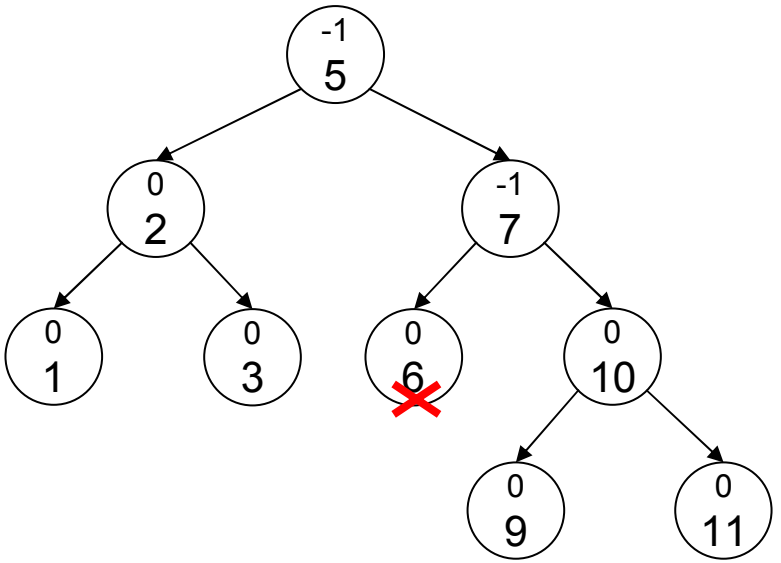
Depois do rebalanceamento

Sem necessidade de rebalanceamento

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

Remoção

Antes da remoção

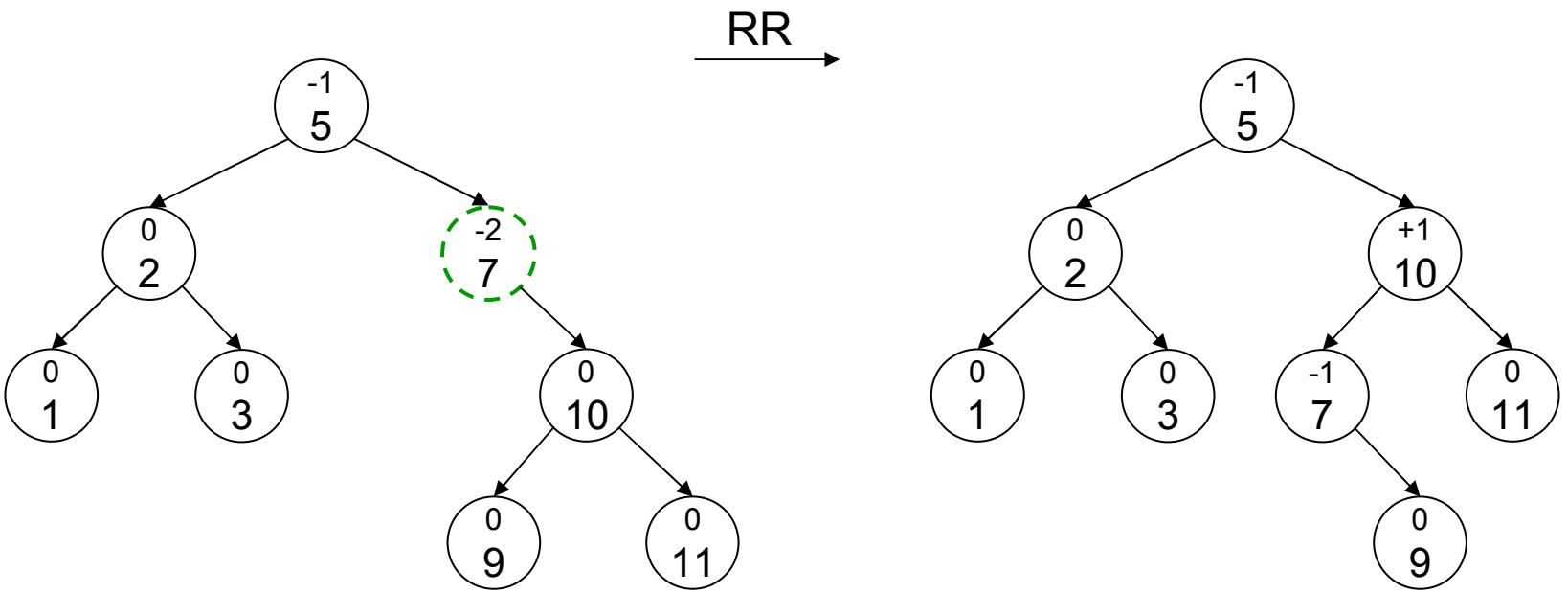


Depois do rebalanceamento

Remoção

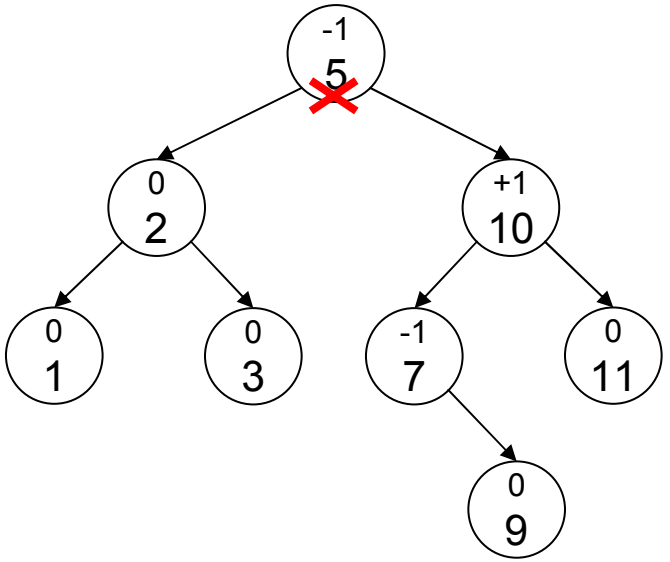
Depois da remoção

Depois do rebalanceamento



Remoção

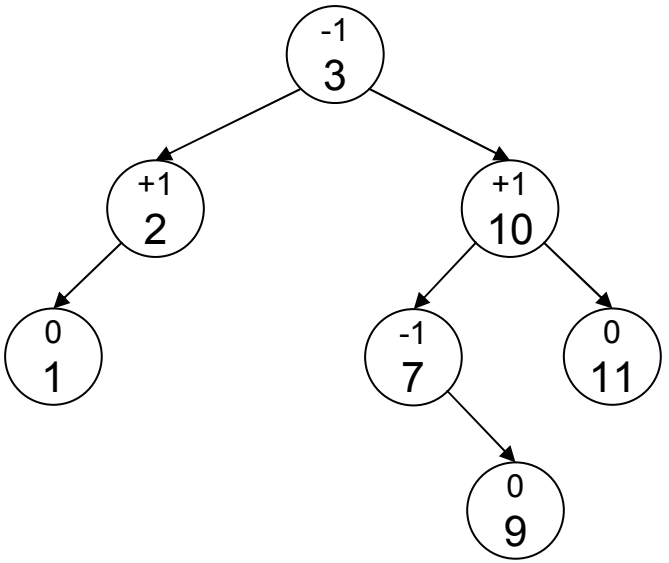
Antes da remoção



Depois do rebalanceamento

Remoção

Depois da remoção

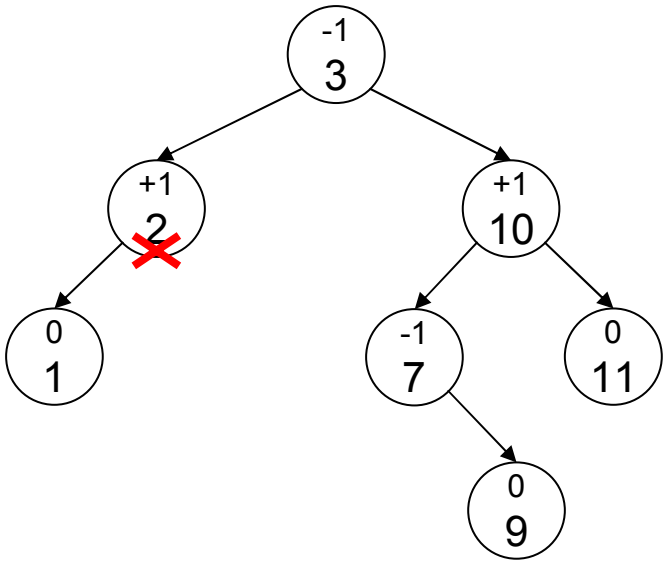


Depois do rebalanceamento

Sem necessidade de rebalanceamento

Remoção

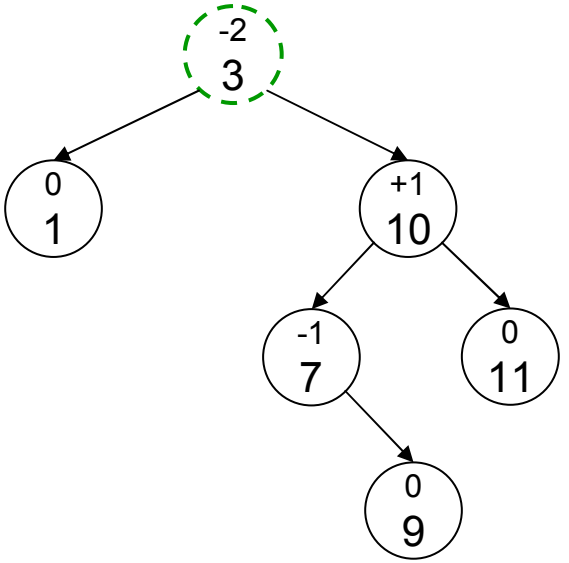
Antes da remoção



Depois do rebalanceamento

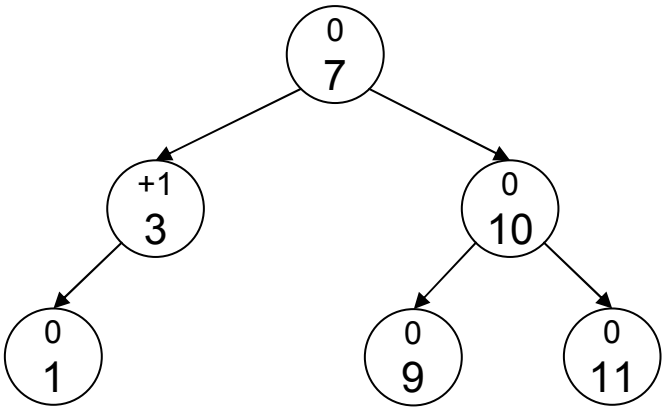
Remoção

Depois da remoção



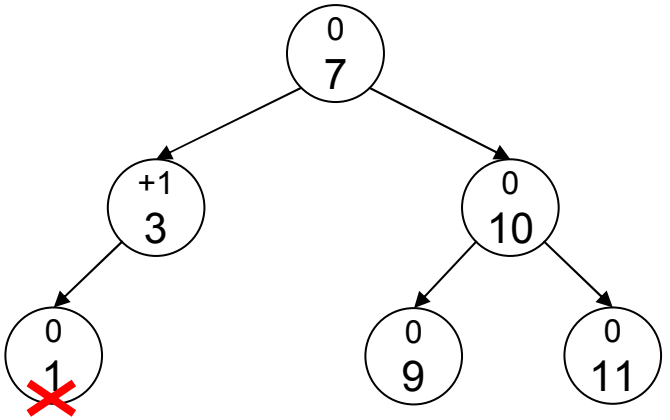
RL

Depois do rebalanceamento

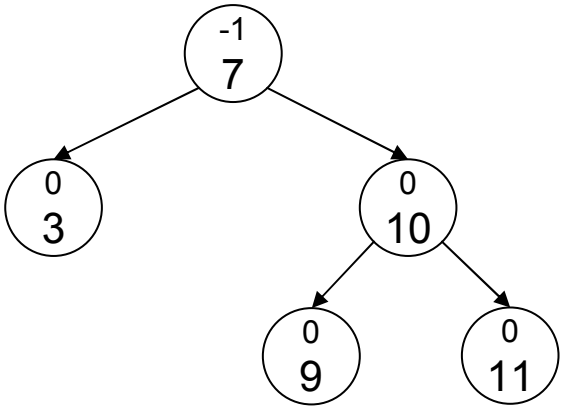


Remoção

Antes da remoção

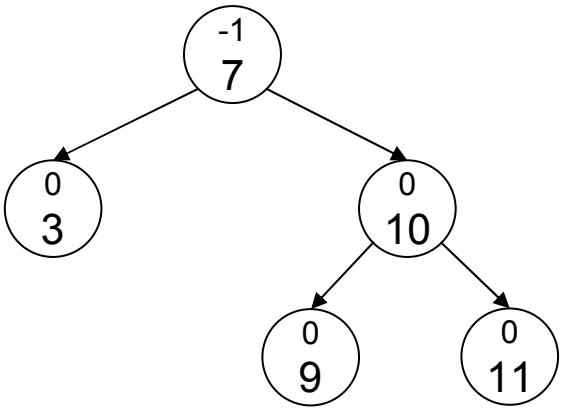


Depois do rebalanceamento



Remoção

Depois da remoção

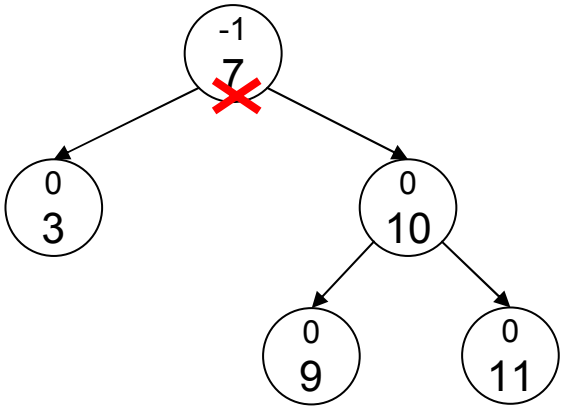


Depois do rebalanceamento

Sem necessidade de rebalanceamento

Remoção

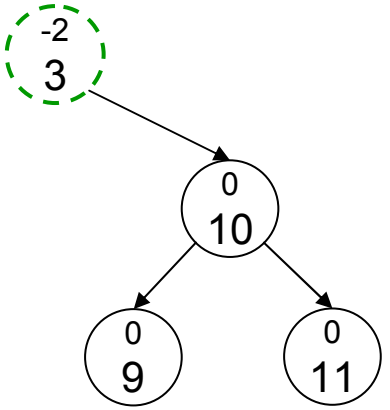
Antes da remoção



Depois do rebalanceamento

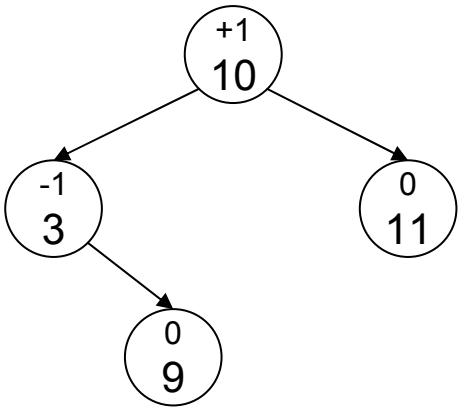
Remoção

Depois da remoção



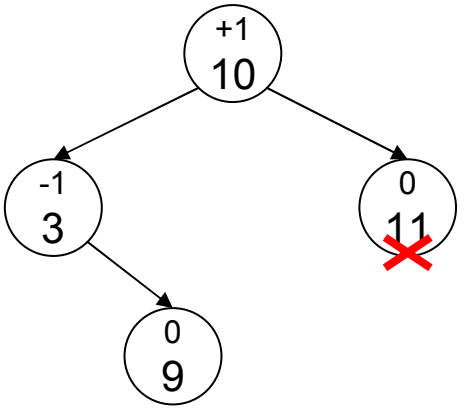
RR

Depois do rebalanceamento



Remoção

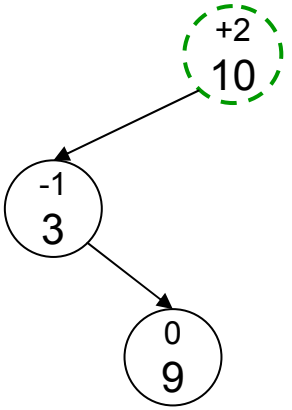
Antes da remoção



Depois do rebalanceamento

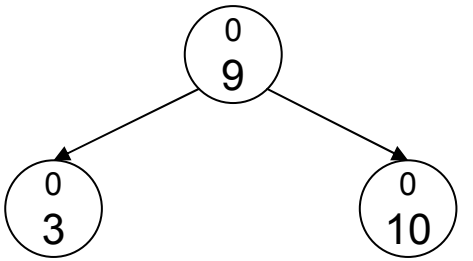
Remoção

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