PCC104 - Projeto e Análise de Algoritmos

Marco Antonio M. Carvalho

(baseado nas notas de aula do prof. Túlio A. M. Toffolo)
Departamento de Computação
Instituto de Ciências Exatas e Biológicas
Universidade Federal de Ouro Preto

Conteúdo

Fonte

Este material é baseado nos livros

- T. H. Cormen, C. E. Leiserson, R. L. Rivest, and C. Stein. *Introduction to Algorithms*. The MIT Press, 3rd edition, 2009.
- S. Halim. *Competitive Programming*. 3rd Edition, 2013.
- Ian Parberry and William Gasarch. *Problems on Algorithms*. Second Edition, 2002.
- Ian Parberry *Lecture Notes on Algorithm Analysis and Complexity Theory*. Fourth Edition, 2001.

Licença

Conjuntos

Descrição

Em um **conjunto**, o valor de um elemento também o identifica, i.e., o valor é em si a chave e cada valor deve ser único.

O valor dos elementos em um conjunto é constante, i.e., não pode ser modificado uma vez na estrutura, mas podem ser inseridos, removidos e pesquisados.

Internamente, os elementos de um conjunto estão sempre ordenados seguindo um critério de ordenação estrito e específico.

Os conjuntos geralmente são implementados como árvores binárias de pesquisa.

Descrição

As árvores de pesquisa são um tipo de estrutura de dados muito eficiente para armazenar informação.

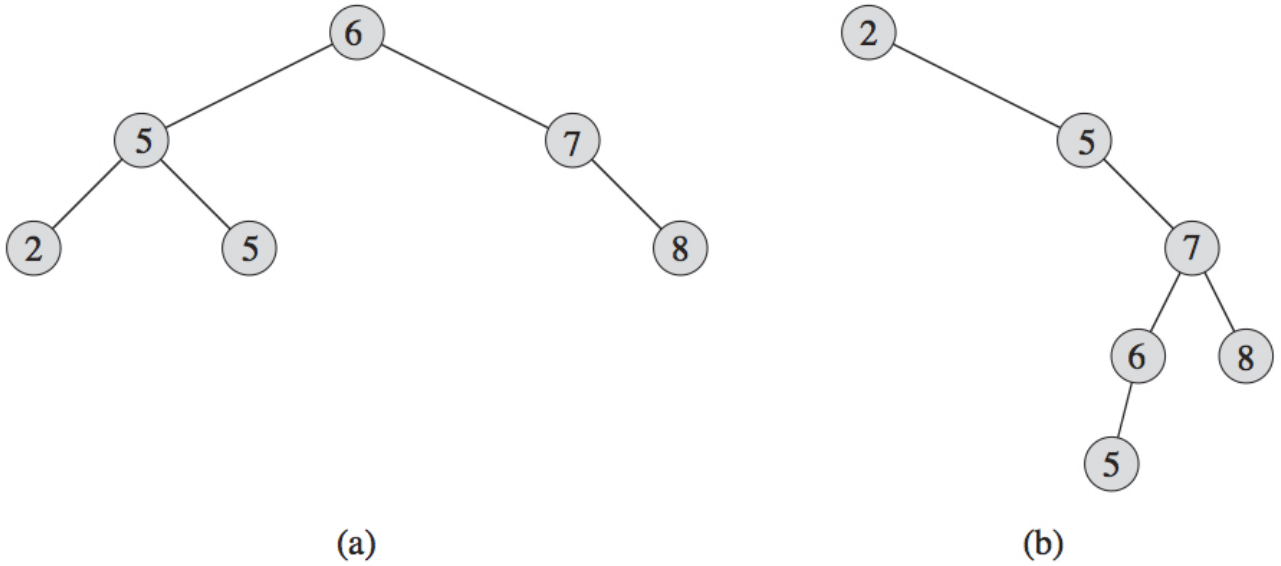
Particularmente adequada quando existe necessidade de considerar todos ou alguma combinação de:

- Acesso direto e sequencial eficientes.
- Facilidade de inserção e remoção de elementos.
- Boa taxa de utilização de memória.
- Utilização de memória primária e secundária.

Propriedades

Para qualquer nó que contenha um elemento, temos a relação invariante:

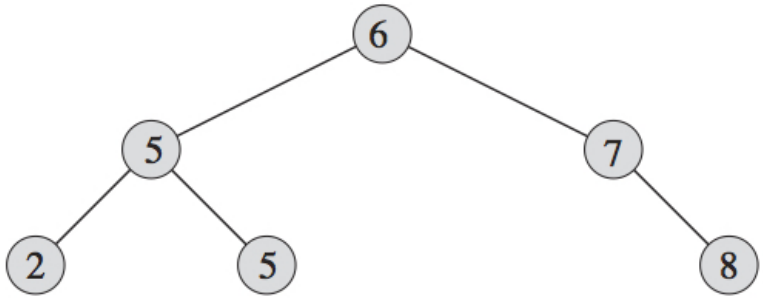
- Os elementos com chaves menores estão na subárvore à esquerda.
- Os elementos com chaves maiores estão na subárvore à direita.

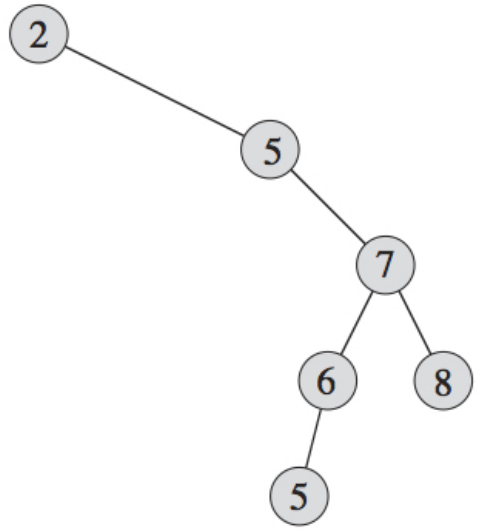


(a) Uma árvore binária balanceada. (b) Outra árvore menos eficiente, com os mesmos elementos.

Propriedades

- O nível do nó raiz é 0.
- Se um nó está no nível i então a raiz de suas subárvores estão no nível $i + 1$.
- A altura de um nó é o comprimento do caminho mais longo deste nó até um nó folha.
- A altura de uma árvore é a altura do nó raiz.





Tipo Abstrato de Dados

```c
typedef long TChave;

typedef struct {
    // outros componentes
    TChave Chave;
} TItem;

typedef struct No {
  TItem item;
  struct No *pEsq, *pDir;
} TNo;

typedef TNo *TArvore;
```

Pesquisa

Para pesquisar um elemento com uma chave x:

- Compara-se com a chave que está na raiz.
- Se x é menor, analisa-se para a subárvore esquerda.
- Se x é maior, analisa-se para a subárvore direita.
- Repete-se o processo recursivamente, até que a chave procurada seja encontrada ou um nó folha seja atingido.

Se a pesquisa tiver sucesso então o conteúdo do elemento retorna no próprio elemento x.

Pesquisa Recursiva

```
int TArvore_Pesquisa(TArvore pRaiz, TChave c, TItem *pX){
    if (pRaiz == NULL)
        return 0;

    if (c < pRaiz->item.chave)
        return TArvore_Pesquisa(pRaiz->pEsq, c, pX);
    if (c > pRaiz->item.chave)
        return TArvore_Pesquisa(pRaiz->pDir, c, pX);

    *pX = pRaiz->item;
    return 1;
}
```

Pesquisa Não Recursiva

```
int TArvore_Pesquisa(TArvore pRaiz, TChave c, TItem *pX){
    TNo *pAux;

    pAux = pRaiz;
    while (pAux != NULL) {
        if (c == pAux->item.chave) {
            *pX = pAux->item;
            return 1;
        }
        else if (c > pAux->item.chave)
            pAux = pAux->pDir;
        else
            pAux = pAux->pEsq;
    }
    return 0; // nao encontrado!
}
```

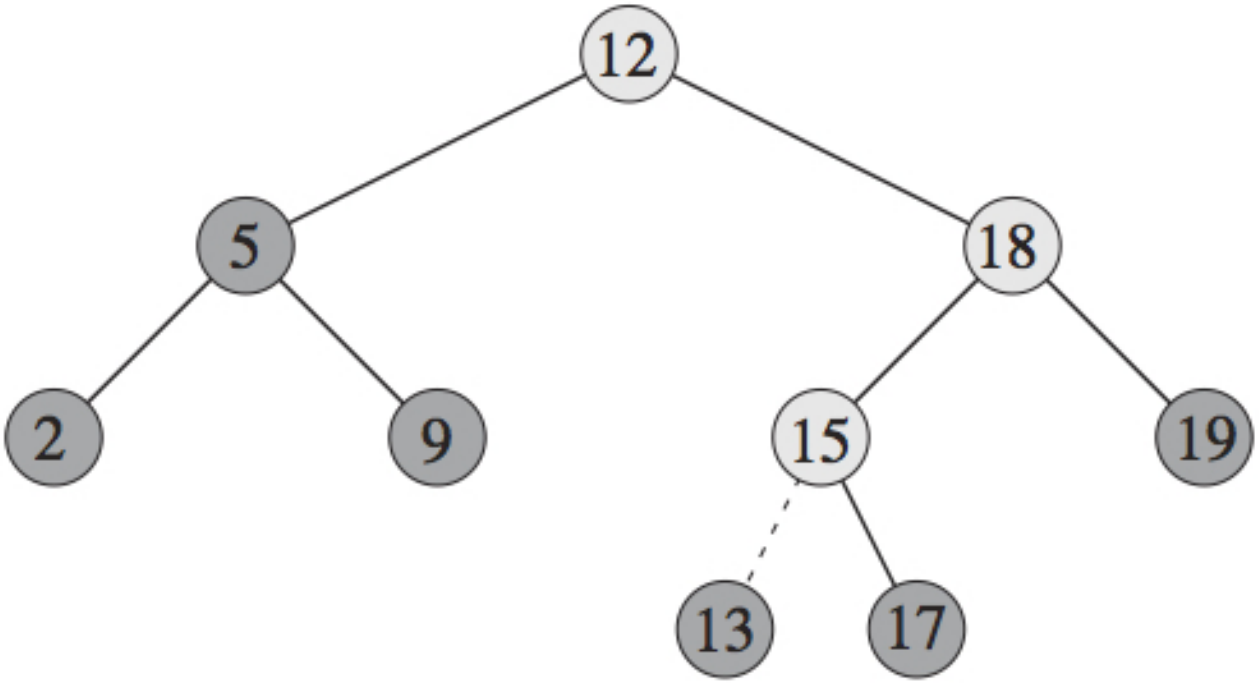
Inserção

Uma vez que os elementos são mantidos ordenados, há apenas uma posição correta para inserção.

Como inserir?

- Cria-se uma célula contendo elemento.
- Procura-se o lugar adequado na árvore.
- Se elemento não estiver na árvore, ele então é inserido

Árvores



Inserção (Árvore Não Vazia)

```
int TArvore_Insere(TNo *pRaiz, TItem x){
    if (pRaiz == NULL) return -1; // arvore vazia

    if (x.chave < pRaiz->item.chave) {
        if (pRaiz->pEsq == NULL) {
            pRaiz->pEsq = TNo_Cria(x);
            return 1;
        }
        return TArvore_Insere(pRaiz->pEsq, x);
    }
    if (x.chave > pRaiz->item.chave) {
        if (pRaiz->pDir == NULL) {
            pRaiz->pDir = TNo_Cria(x);
            return 1;
        }
        return TArvore_Insere(pRaiz->pDir, x);
    }
    return 0; // elemento ja existe
}
```

Inserção (Árvore Vazia)

```
void TArvore_Insere_Raiz(TNo **ppRaiz, TItem x){
    if (*ppRaiz == NULL) {
        *ppRaiz = TNo_Cria(x);
        return;
    }

    TArvore_Insere(*ppRaiz, x);
}
```

Inserção (Não Recursiva)

```
int TArvore_Insere(TNo **ppRaiz, TItem x){
    TNo **ppAux;
    ppAux = ppRaiz;

    while (*ppAux != NULL) {
        if (x.chave < (*ppAux)->item.chave)
            ppAux = &((*ppAux)->pEsq);
        else if (x.chave > (*ppAux)->item.chave)
            ppAux = &((*ppAux)->pDir);
        else
            return 0;
    }
    *ppAux = TNo_Cria(x);
    return 1;
}
```

Criação de um Nó

```
TNo *TNo_Cria(TItem x){
    TNo *pAux = (TNo*)malloc(sizeof(TNo));
    pAux->item = x;
    pAux->pEsq = NULL;
    pAux->pDir = NULL;
    return pAux;
}
```

Inicialização

```
void TArvore_Inicia(TNo **pRaiz) {
    *pRaiz = NULL;
}
```

Remoção

O processo de remoção de nós de uma árvore depende do tipo de nó.

Caso o nó seja uma folha, o procedimento é simples. Caso contrário, o nó pode ser a própria raiz ou um nó interno possuindo um ou dois filhos.

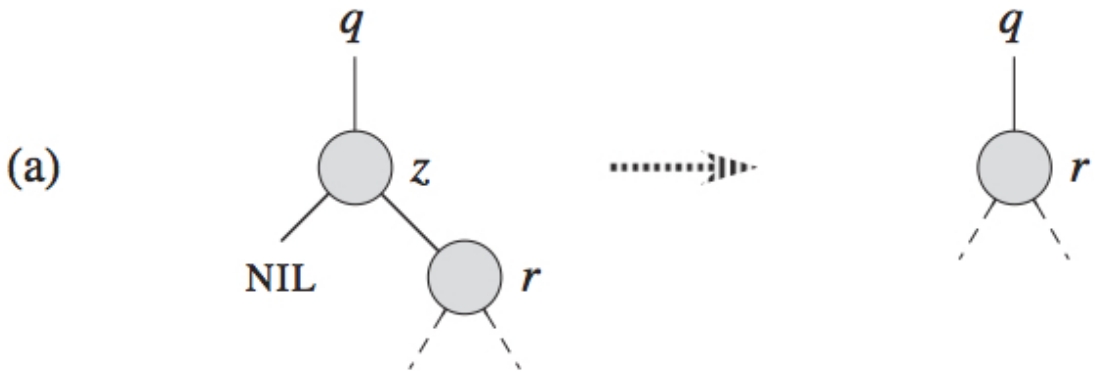
Se o nó a ser removido possuir no máximo um descendente, a operação é simples.

No caso do nó possuir dois descendentes o elemento a ser removido deve ser primeiro substituído pelo elemento mais à esquerda na subárvore direita ou pelo elemento mais à direita na subárvore esquerda.

As figuras a seguir ilustram a remoção de um nó hipotético z em cada uma destas situações.

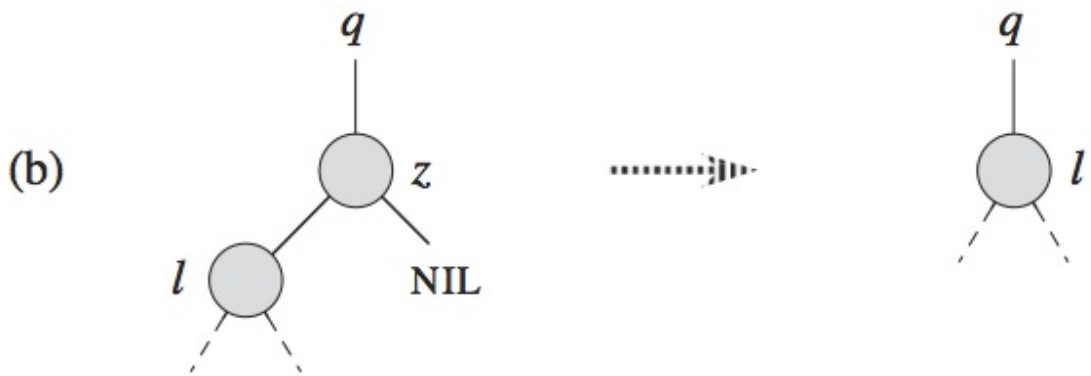
Nó com um descendente à direita

Substituímos o nó z por seu filho à direita, r, que pode inclusive ser nulo.



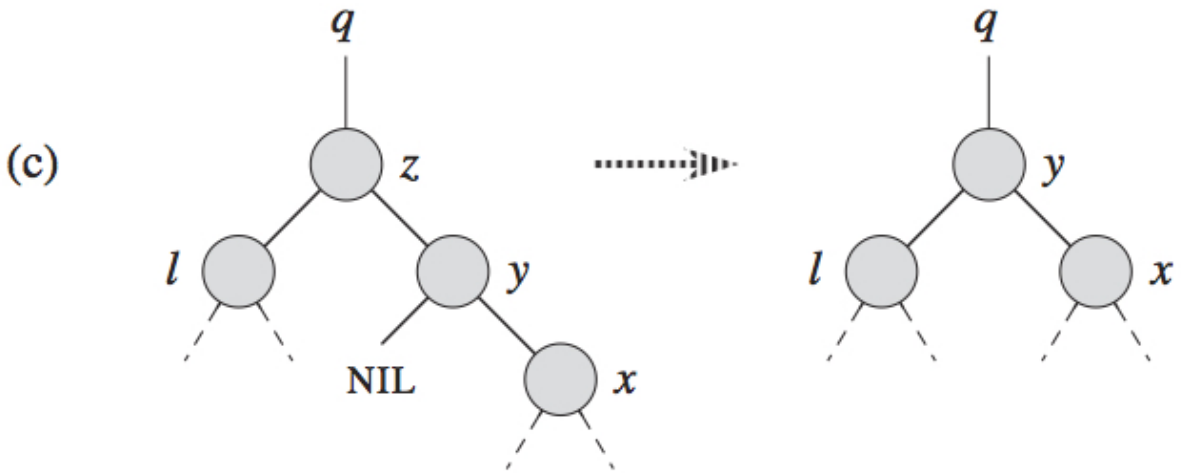
Nó com um descendente à esquerda

Substituímos o nó z por seu filho à esquerda, l, que pode inclusive ser nulo.



Nó com dois descendentes 1

Substituímos o nó z por seu sucessor y, que por sua vez é substituído pelo seu descendente à direita x.

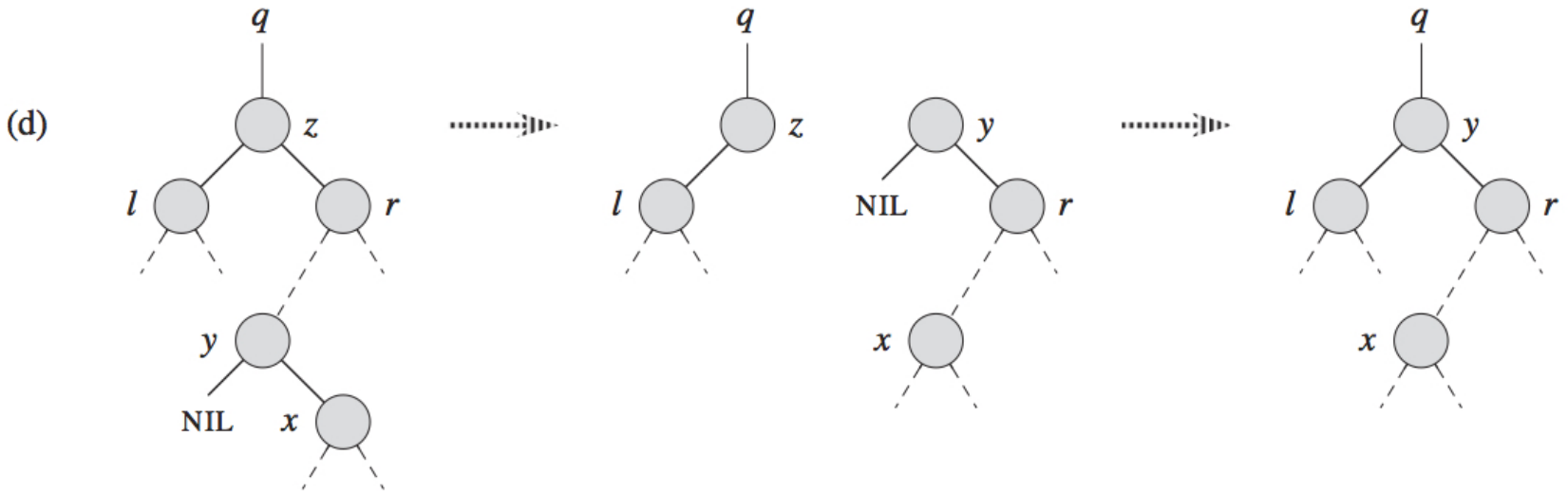


Nó com dois descendentes 2

Substituímos o nó z por seu sucessor y, o elemento mais à esquerda da subárvore direita.

Por sua vez, y é substituído pelo seu descendente à direita x.

Adicionalmente, definimos y como pai de r.



Remoção

```
int TArvore_Remove(TNo **p, TItem x){
    TNo *pAux;
    if (*p == NULL)
        return 0;

    if (x.chave < (*p)->item.chave)
        return TArvore_Remove(&((*p)->pEsq), x);
    if (x.chave > (*p)->item.chave)
        return TArvore_Remove(&((*p)->pDir), x);

    if ((*p)->pEsq == NULL && (*p)->pDir == NULL) { // no eh
        folha
        free(*p);
        *p = NULL;
        return 1;
    }

    ...
```

Remoção

```
    ...

    if ((*p)->pEsq != NULL && (*p)->pDir == NULL) { // esq
        pAux = *p;
        *p = (*p)->pEsq;
        free(pAux);
        return 1;
    }
    if ((*p)->pDir != NULL && (*p)->pEsq == NULL) { // dir
        pAux = *p;
        *p = (*p)->pDir;
        free(pAux);
        return 1;
    }
    // no possui dois filhos
    TArvore_Sucessor(*p, &((*p)->pDir));
    // equivalente a TArvore_Antecessor(*p, &((*p)->pEsq));
    return 1;
}
```

Sucessor

```
void TArvore_Sucessor (TNo *q, TNo **r){
     TNo *pAux;
     if ((*r)->pEsq != NULL) {
          TArvore_Sucessor(q, &(*r)->pEsq);
          return;
     }
     q->item = (*r)->item;
     pAux = *r;
     *r = (*r)->pDir;
     free(pAux);
}
```

Complexidade

Ambas a operações de inserção e remoção envolvem operações de pesquisa, tendo a complexidade limitada por ela.

- Melhor caso: $O(1)$.
- Pior caso: $O(n)$.
- Caso médio: $O(\log\, n)$.

O tempo de execução dos algoritmos para árvores binárias de pesquisa dependem muito do formato da árvore, ou seja, se ela está balanceada ou não.

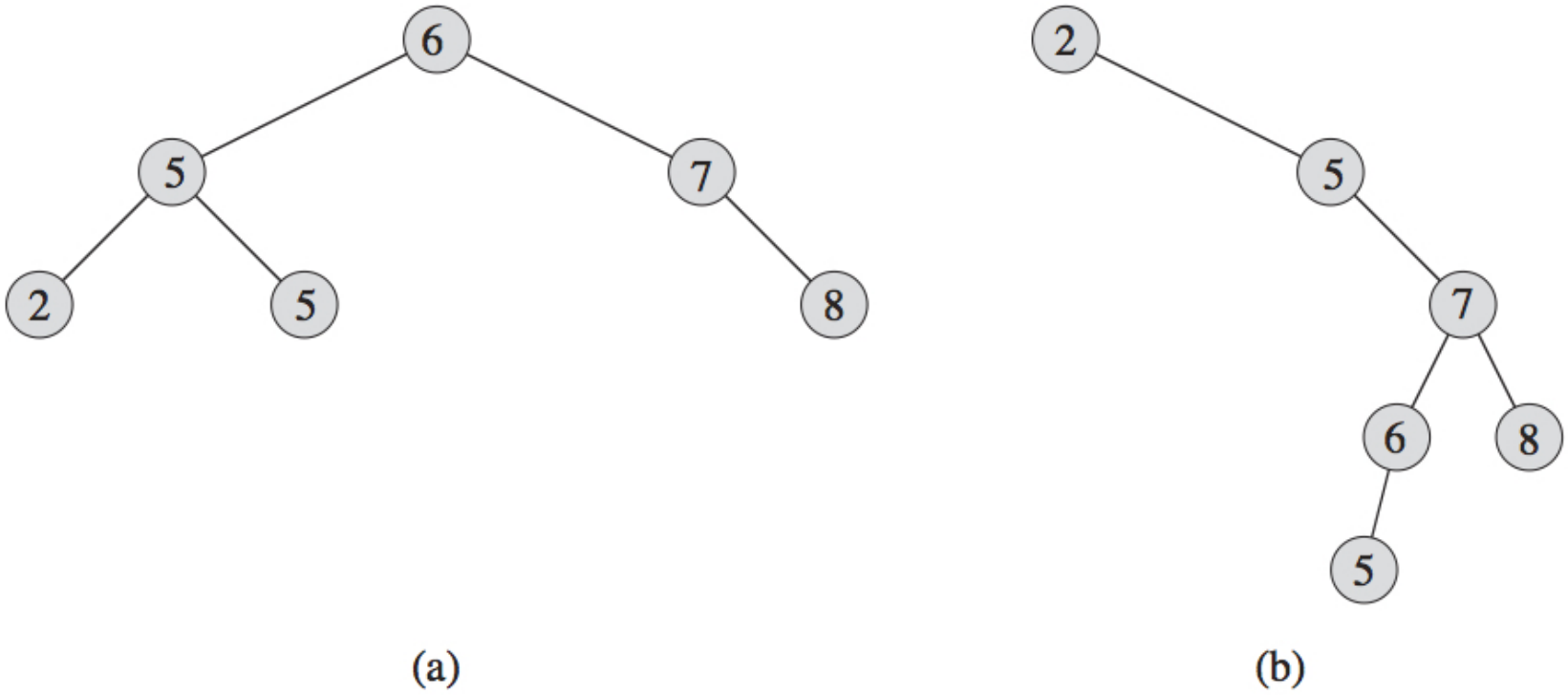
Complexidade

Para obter o pior caso basta que as chaves sejam inseridas em ordem crescente ou decrescente.

Neste caso a árvore resultante é uma lista linear, cujo número médio de comparações é $n/2$.

Para uma árvore de pesquisa aleatória o número esperado de comparações para recuperar um elemento qualquer é cerca de 1,39 log n, "apenas" 39% pior que a árvore completamente balanceada.

Na prática é impossível prever a ordem de inserção dos nós ou até alterá-la, portanto, utilizamos algoritmos para balanceamento de árvores.



(a) Uma árvore binária balanceada. (b) Outra árvore menos eficiente, com os mesmos elementos.

Características

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

Por exemplo, em uma árvore binária degenerada de 10.000 nós são necessárias, em média, 5.000 comparações (semelhante a arranjos ordenados e listas encadeadas).

Em uma árvore balanceada com o mesmo número de nós essa média reduz-se a 14 comparações.

Características

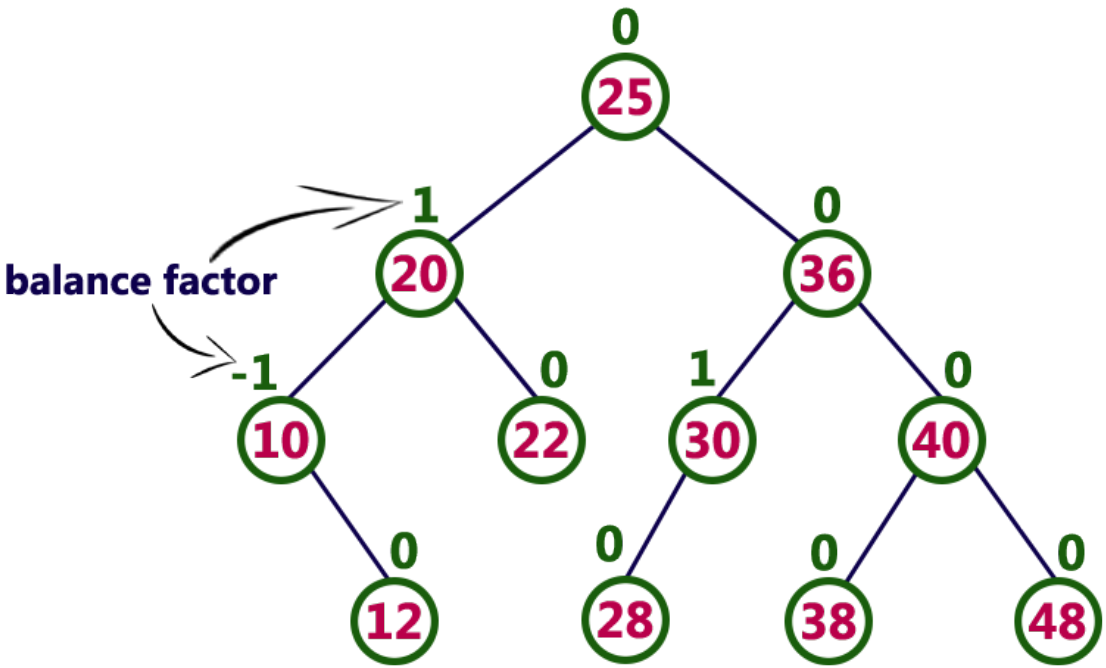
Uma árvore binária balanceada é aquela na qual, para cada nó, as alturas de suas subárvores esquerda e direita diferem em, no máximo, 1.

O Fator de Balanceamento (FB) de um nó é a diferença entre a altura da sua subárvore esquerda em relação à sua subárvore direita:

$$\text{FB}(p) = \text{altura(subárvore esquerda de } p) - \text{altura(subárvore direita de } p)$$

Em uma árvore binária balanceada os FB de todos os nós estão no intervalo $-1 \leq \text{FB} \leq 1$.

Árvores AVL



Descrição

Trata-se de um algoritmo de 1962 para balanceamento de árvores binárias, cujo nome também denota um tipo de árvores balanceadas.

A origem da denominação vem dos seus dois criadores: Adel'son-Vel'skii e Landis.

Consiste em organizar uma árvore binária de busca tal que, para qualquer nó interno v, a diferença das alturas dos filhos de v é no máximo 1 e no mínimo -1.

Para manter o balanceamento da árvore, a cada operação de inserção ou remoção é necessário atualizar o fator de balanceamento de partir do nó pai do nó inserido/removido até a raiz da árvore.

Ao determinar um desbalanceamento, são aplicadas operações de rotação de nós, de acordo com quatro casos específicos.

Estrutura

```
typedef long TipoChave;

typedef struct Registro {
  TipoChave Chave;
  // outros componentes
} Registro;

typedef Struct No {
  Registro Reg;
  TNo *pEsq, *pDir;
} No;

typedef TNo *TipoAVL;
```

Determinação da Altura

```
int Altura(TNo*pRaiz)
{
  int iEsq,iDir;

  if (pRaiz == NULL)
    return 0;

  iEsq = Altura(pRaiz->pEsq);
  iDir = Altura(pRaiz->pDir);

  if (iEsq > iDir)
    return iEsq + 1;
  else
    return iDir + 1;
}
```

Fator de Balanceamento

```
int FB (TNo*pRaiz)
{
  if (pRaiz == NULL)
    return 0;

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

Verificação de Propriedade AVL

```
int EhArvoreAVLl(TNo*pRaiz){
  int fb;

  if (pRaiz == NULL)
    return 1;

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

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

Rotações

As operações de inserção e remoção são realizadas como em árvores binárias, inicialmente.

Após as operações, atualizam-se as alturas e os fatores de balanceamento dos nós afetados.

Eventualmente, uma destas operações pode degenerar a árvore, desbalanceando-a.

A restauração do balanceamento e manutenção da propriedade da árvore binária de busca é feita através de rotações na árvore em relação a um nó pivô.

O nó pivô é a raiz da subárvore que após uma operação causa fator de balanceamento fora do intervalo [-1, 1] em seu pai.

Primeiro caso: Rotação simples para a direita

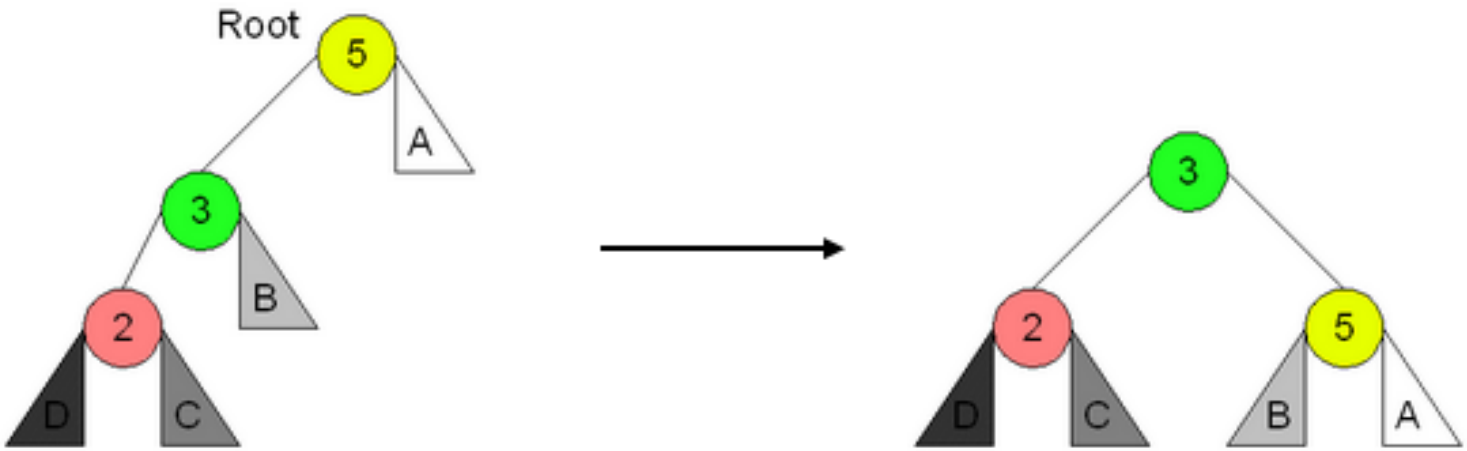
FB > 1

Caracterização: A diferença das alturas dos filhos de um nó pai é igual a 2 e a diferença das alturas dos filhos do filho esquerdo é igual a 1.

Em outras palavras, um nó está desbalanceado e seu filho está no mesmo sentido da inclinação, formando uma linha reta à esquerda (*Left Left*, LL).

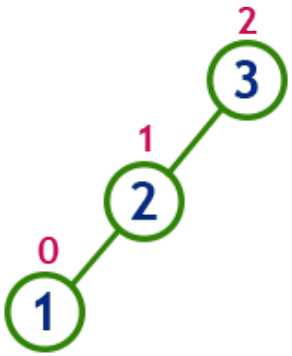
Solução: Uma rotação RR. O filho esquerdo deve se tornar o novo pai e o nó pai deve se tornar o seu filho da direita do filho esquerdo.

Árvores AVL



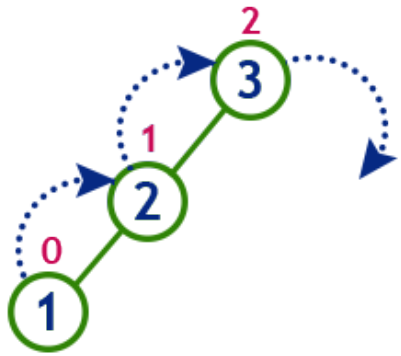
Árvores AVL

insert 3, 2 and 1

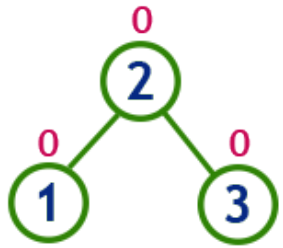


Tree is imbalanced

because node 3 has balance factor 2



To make balanced we use RR Rotation which moves nodes one position to right



After RR Rotation Tree is Balanced

Segundo caso: Rotação simples para a esquerda

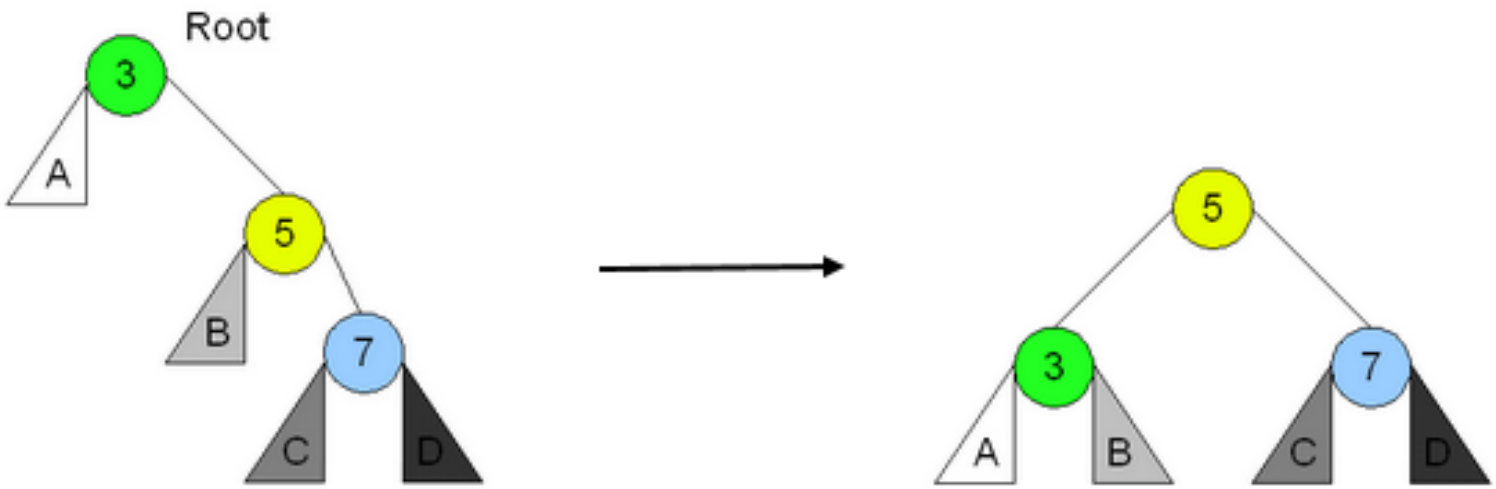
FB $<$ -1

Caracterização: A diferença das alturas dos filhos de um nó pai é igual a -2 e a diferença das alturas dos filhos do filho direito é igual a -1.

Em outras palavras, um nó está desbalanceado e seu filho está no mesmo sentido da inclinação, formando uma linha reta à direita (*Right Right*, RR).

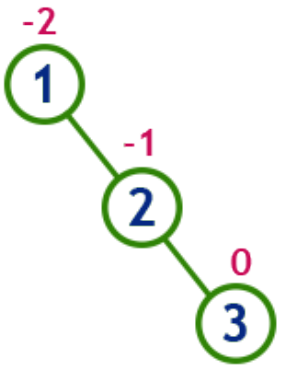
Solução: Uma rotação LL. O filho direito deve se tornar o novo pai e o nó pai deve se tornar o seu filho da esquerda do filho direito.

Árvores AVL

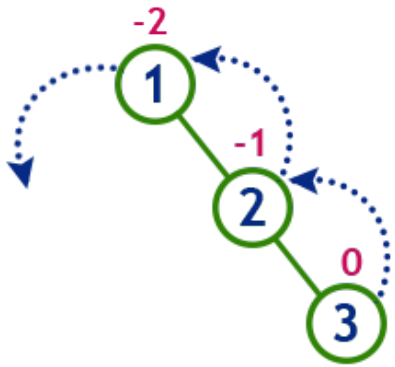


Árvores AVL

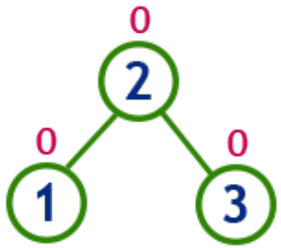
insert 1, 2 and 3



Tree is imbalanced



To make balanced we use LL Rotation which moves nodes one position to left



After LL Rotation Tree is Balanced

Terceiro caso: Rotação dupla para a direita

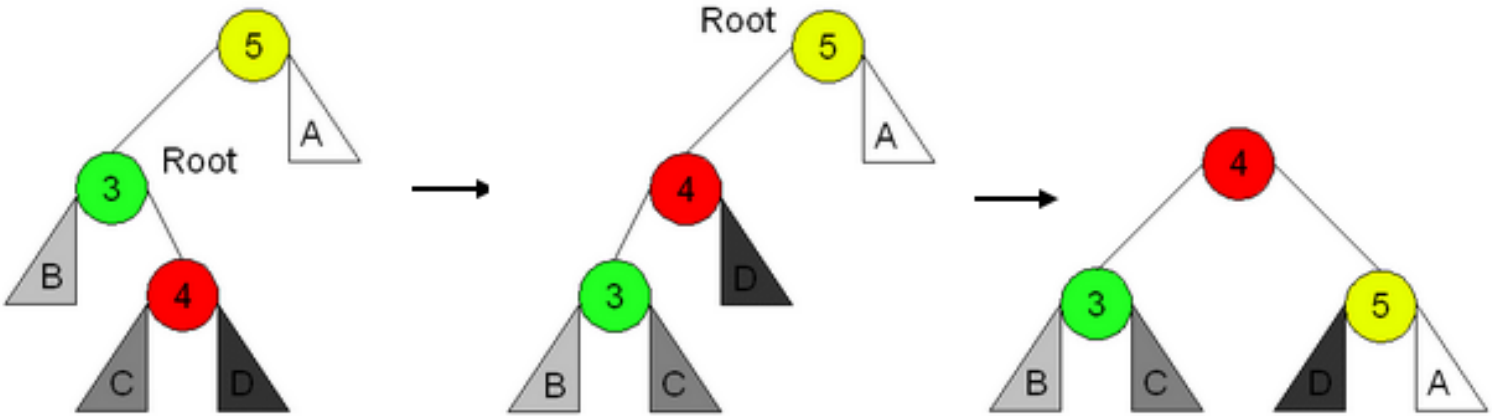
FB > 1

Caracterização: A diferença das alturas dos filhos de um nó pai é igual a 2 e a diferença das alturas dos filhos do filho esquerdo é igual a -1.

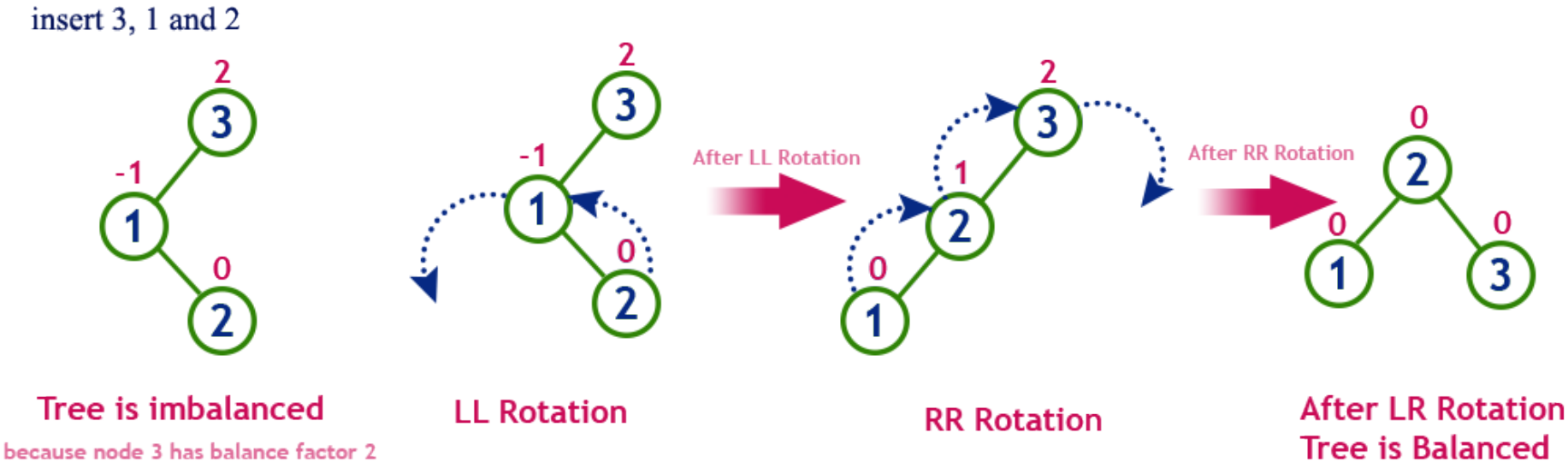
Em outras palavras, um nó está desbalanceado e seu filho está inclinado no sentido inverso ao pai, formando uma curva (*Left Right*, LR).

Solução: Uma rotação LR. Aplicar uma rotação à esquerda no filho esquerdo e depois uma rotação à direita no nó pai.

Árvores AVL



Árvores AVL


insert 3, 1 and 2
2
3
-1
1
0
2
Tree is imbalanced
because node 3 has balance factor 2
2
3
-1
1
0
2
LL Rotation
After LL Rotation
2
3
1
2
0
1
RR Rotation
After RR Rotation
0
2
0
1
0
3
After LR Rotation
Tree is Balanced

Quarto caso: Rotação dupla para a esquerda

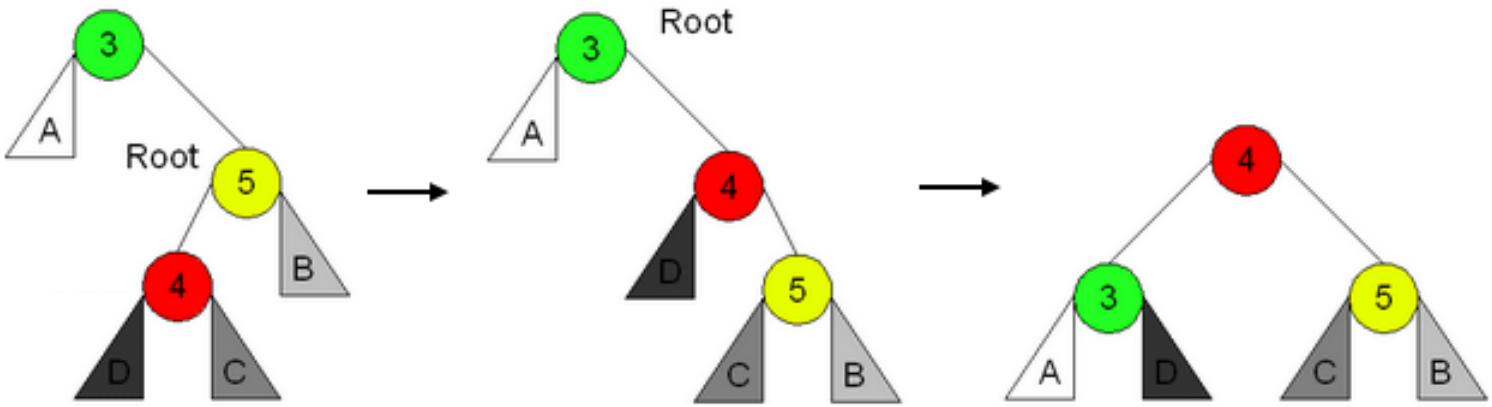
FB $<$ -1

Caracterização: A diferença das alturas dos filhos de um nó pai é igual a -2 e a diferença das alturas dos filhos do filho direito é igual a 1.

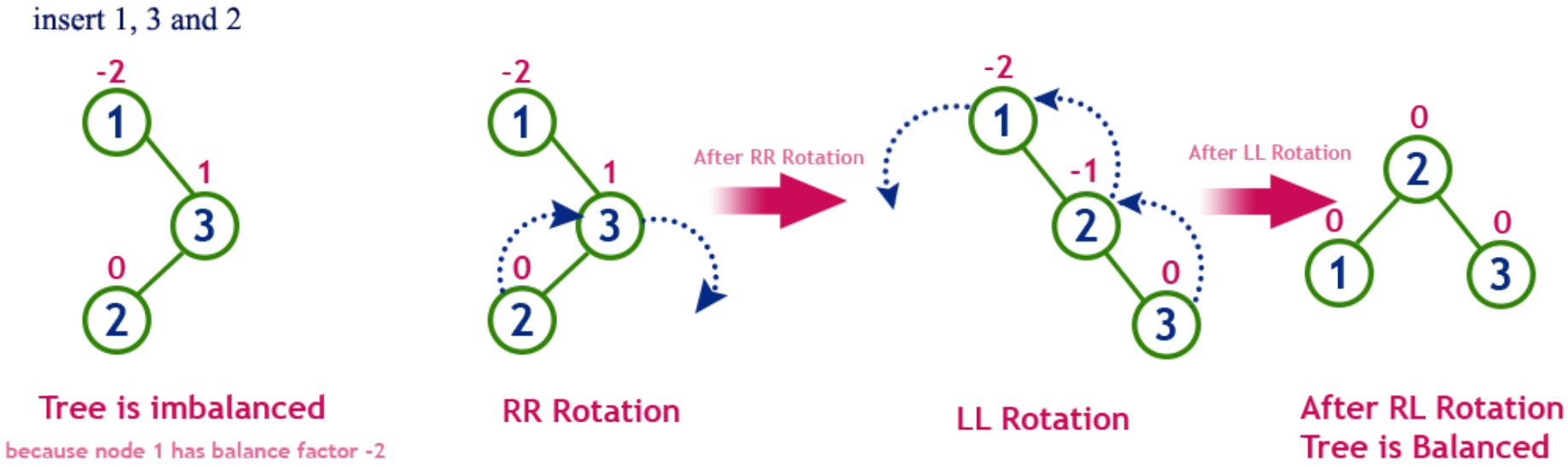
Em outras palavras, um nó está desbalanceado e seu filho está inclinado no sentido inverso ao pai, formando uma curva (*Right Left*, RL).

Solução: Uma rotação RL. Aplicar uma rotação à direta no filho direito e depois uma rotação à esquerda no nó pai.

Árvores AVL



Árvores AVL


insert 1, 3 and 2
-2
1
1
3
0
2
Tree is imbalanced
because node 1 has balance factor -2
-2
1
1
3
0
2
RR Rotation
After RR Rotation
-2
1
-1
2
0
3
LL Rotation
After LL Rotation
0
2
0
1
0
3
After RL Rotation
Tree is Balanced

Rotações Simples

```
  void RR(TNo**ppRaiz){
    TNo *pAux;

    pAux = (*ppRaiz)->pEsq;
    (*ppRaiz)->pEsq = pAux->pDir;
    pAux->pDir = (*ppRaiz);
    (*ppRaiz) = pAux;
  }

void LL(TNo**ppRaiz){
  TNo *pAux;

  pAux = (*ppRaiz)->pDir;
  (*ppRaiz)->pDir = pAux->pEsq;
  pAux->pEsq = (*ppRaiz);
  (*ppRaiz) = pAux;
}
```

Balanceamento

```
int Balanceamento(TNo**ppRaiz)
{
    int fb = FB(*ppRaiz);
    if (fb > 1)
        return BalanceiaEsquerda(ppRaiz);
    else if (fb < -1)
        return BalanceiaDireita(ppRaiz);
    else
        return 0;
}
```

Balanceamento

```
int BalanceiaEsquerda(TNo**ppRaiz)
{
  int fbe = FB ((*ppRaiz)->pEsq);
  if (fbe > 0)
  {
    RR(ppRaiz);
    return 1;
  }
  else if (fbe < 0)
  { // Rotacao Dupla Direita
    LL(&((*ppRaiz)->pEsq));
    RR(ppRaiz); // &(*ppRaiz)
    return 1;
  }
  return 0;
}
```

Balanceamento

```
int BalanceiaDireita(TNo**ppRaiz)
{
  int fbd = FB((*ppRaiz)->pDir);
  if (fbd < 0)
  {
    LL (ppRaiz);
    return 1;
  }
  else if (fbd > 0)
  { // Rotacao Dupla Esquerda
    RR(&((*ppRaiz)->pDir));
    LL(ppRaiz); // &(*ppRaiz)
    return 1;
  }
  return 0;
}
```

Inserção

A inserção em árvores AVL é realizada como em uma árvore binária de pesquisa, sempre feita expandindo um nó folha.

Entretanto, esta inserção pode alterar a altura dos nós da mesma subárvore e pode desbalancear toda a árvore.

Vejamos um exemplo de inserção dos elementos 1, 2, 3, 4, 5, 6, 7 e 8 em uma árvore AVL.

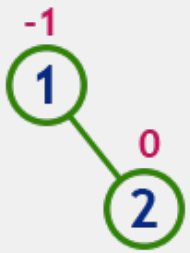
Exemplo de Inserção

insert 1

Tree is balanced

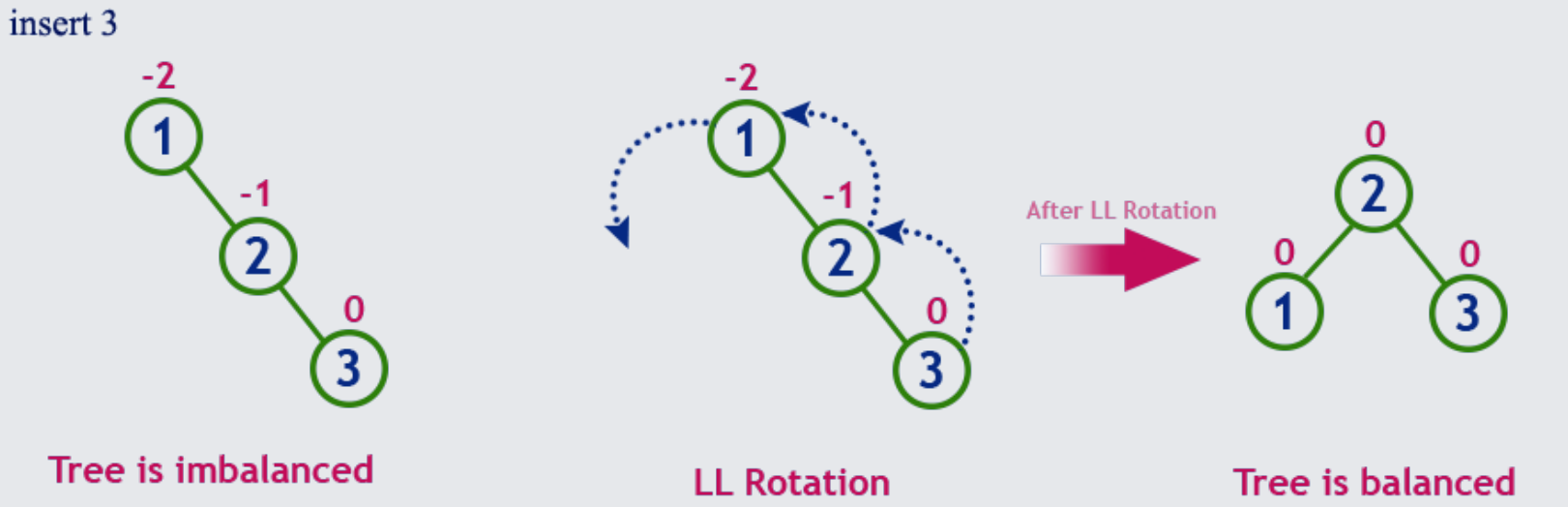
Exemplo de Inserção

insert 2

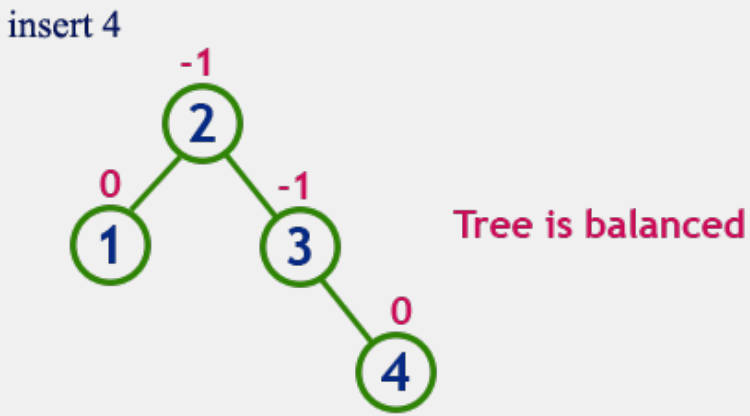


Tree is balanced

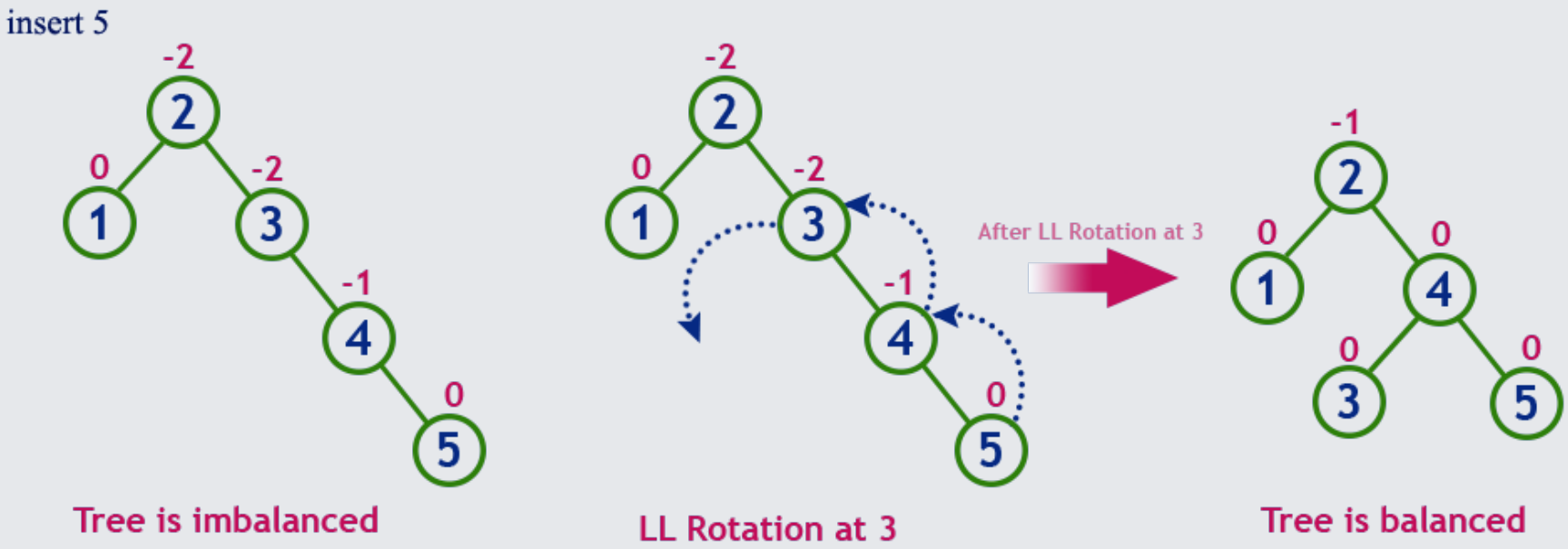
Exemplo de Inserção



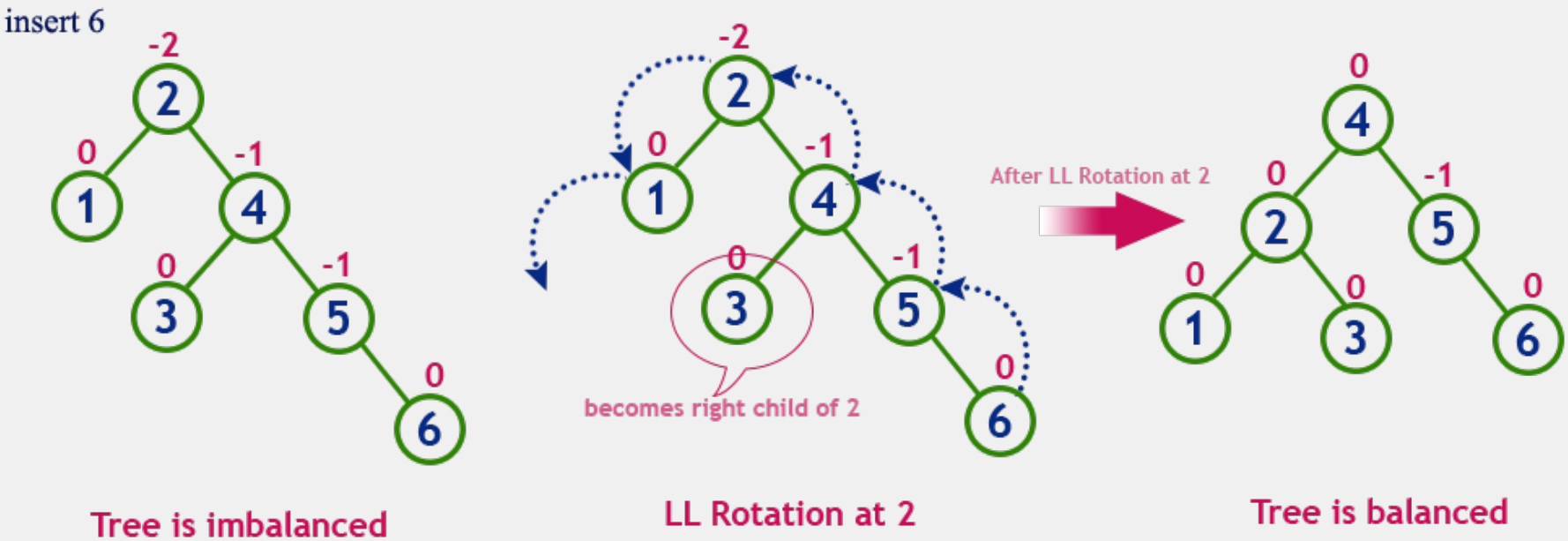
Exemplo de Inserção



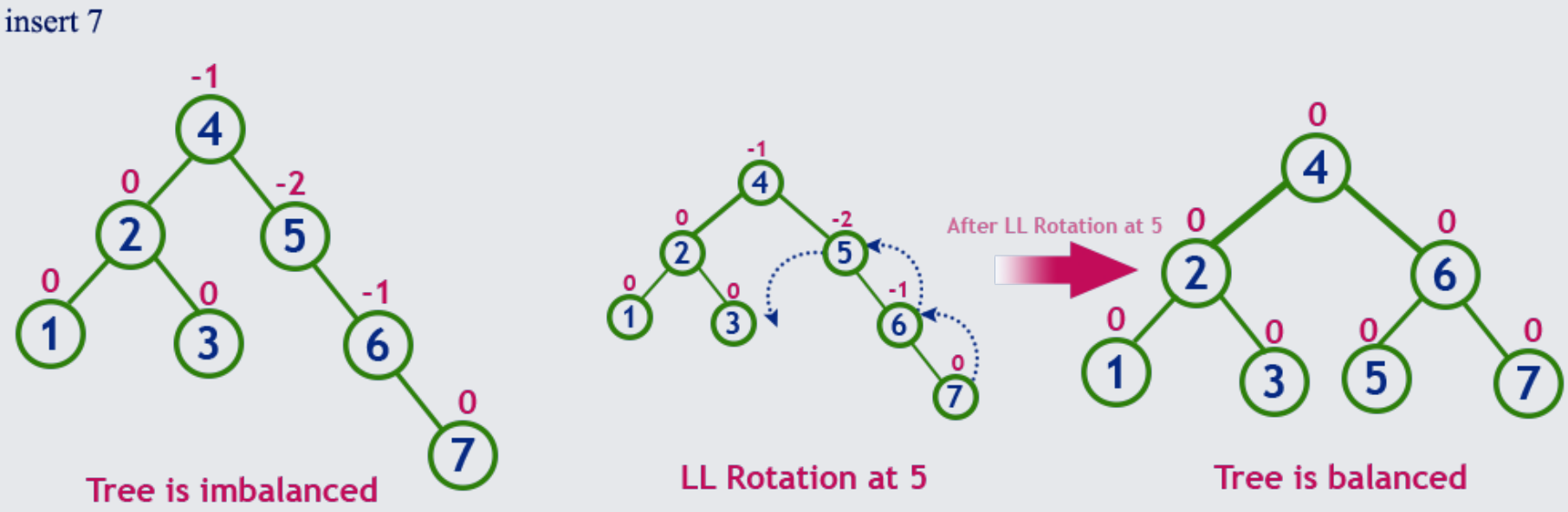
Exemplo de Inserção



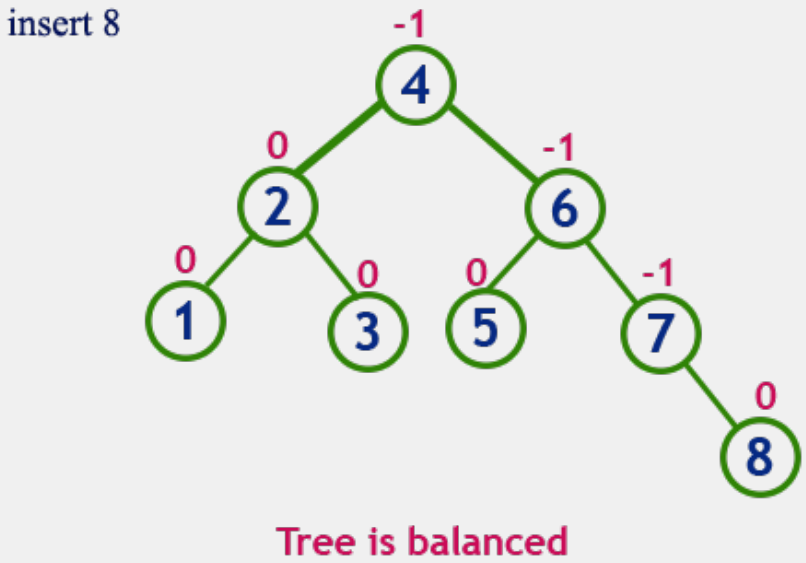
Exemplo de Inserção



Exemplo de Inserção



Exemplo de Inserção



Inserção

```
int Insere(TNo**ppRaiz,Registro*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;
    }
  }
```

Inserção

```
 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 ja presente
}
```

Remoção

A remoção em árvores AVL é realizada como em uma árvore binária de pesquisa, depende do tipo de nó e pode envolver substituições.

Entretanto, esta remoção pode alterar a altura dos nós da mesma subárvore e pode desbalancear toda a árvore.

Remoção

```
int Remove (TNo**ppRaiz,Registro*pX){
  if (*ppRaiz == NULL)
    return 0;
  else if ((*ppRaiz)->Reg.chave == pX->chave){
    *pX = (*ppRaiz)->Reg;
    Antecessor(ppRaiz,&((*ppRaiz)->pEsq));
    Balanceamento(ppRaiz);
    return 1;
  }
  else if ((*ppRaiz)->Reg.chave > pX->chave){
    if (Remove((*ppRaiz)->pEsq,pX)){
       Balanceamento(ppRaiz);
       return 1;
  }
    else
       return 0;
  }
  else // codigo para sub-arvore direita
}
```

Complexidade

- Uma única reestruturação baseada em rotações custa $O(1)$, usando uma árvore binária implementada com ponteiros.
- Pesquisa custa $O(\log n)$, padrão.
- Inserção custa $O(\log n)$, (pesquisa + reestruturação).
- Remoção custa $O(\log n)$, (pesquisa + reestruturação).

Exercício

Mostre (desenhe) uma árvore binária de pesquisa após a inserção dos seguintes elementos, em ordem: 10, 20, 5, 8, 12, 22, 23, 24, 11, 13, 18.

Mostre como ficará a árvore acima após a remoção dos seguintes elementos, na ordem 22, 11, 10 .

Exercício

Apresente uma árvore AVL após a inserção dos elementos 10, 20, 5, 8, 12, 22, 23, 24, 11, 13 e 18.

Mostre como ficará a mesma árvore após a remoção dos elementos 22, 11, 5 e 10.

Dúvidas?