# Árvore Binária de Busca

Professor: Rodolfo Miranda Pereira
Mestrando: Allainclair Flausino dos Santos

# Roteiro

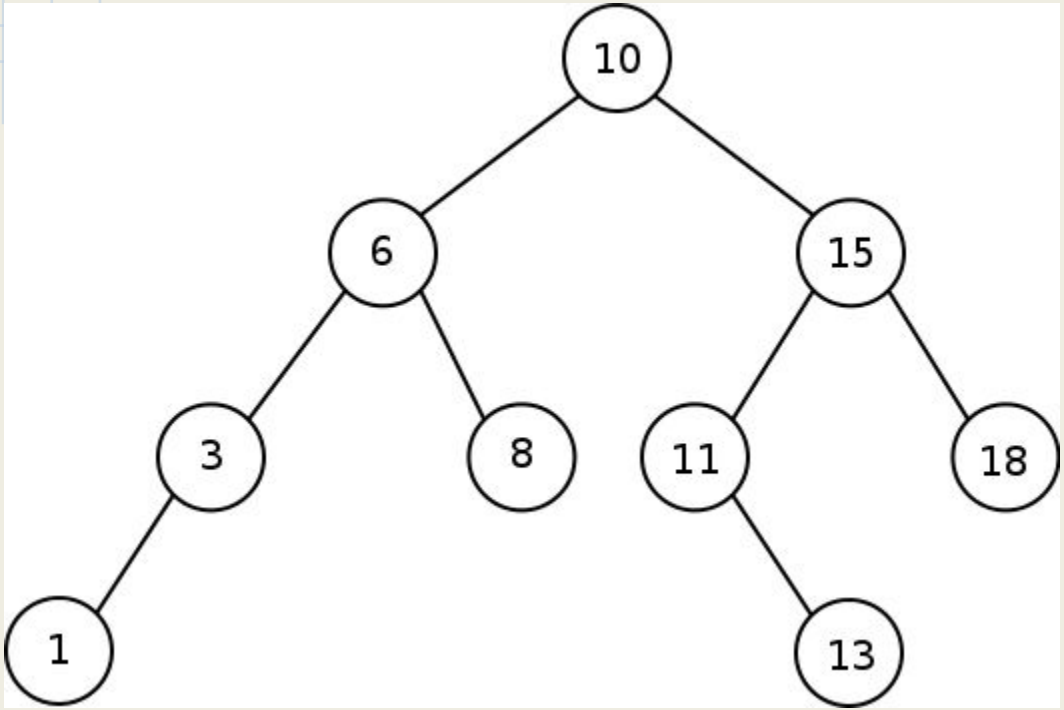
- Revisão de Árvore Binária (de Busca)
- Altura de Árvores
- Operações Básicas
- Estrutura de Dados da Árvore Binária
- Operações:
  - Inserir
  - Ponteiros
  - Número de Nós
  - Altura
  - Número de Folhas
  - Busca
  - Teste ABB (test driven development)
- Árvores AVL

# Revisão

**Entrada:**

**1** 10
**2** 6
**3** 15
**4** 3
**5** 8
**6** 11
**7** 18
**8** 13
**9** 1

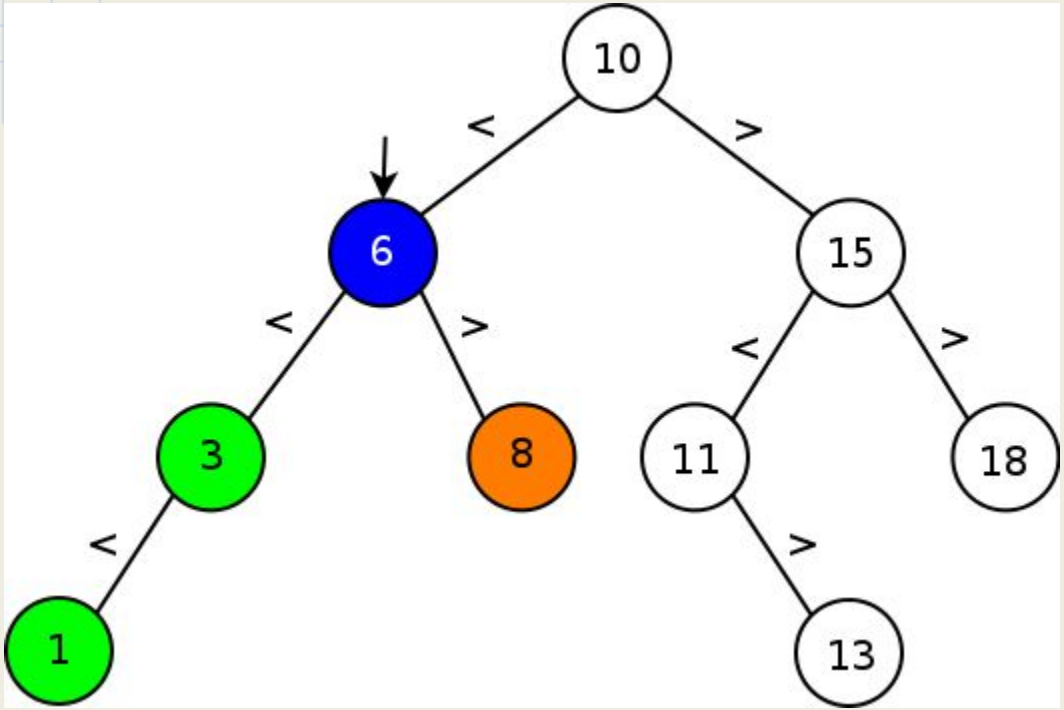
# Revisão



# Revisão

- **Características:**
  - Nós com no máximo 2 filhos (0, 1 ou 2);
  - Nós folhas não tem filhos;
  - **Primeira entrada** (entrada 1) **é o nó raiz** (valor 10);
  - **Sub-árvores** esquerda de um nó **X** contêm elementos menores que **X**;
  - **Sub-árvores** direita de um nó **X** contêm elementos maiores que **X**.

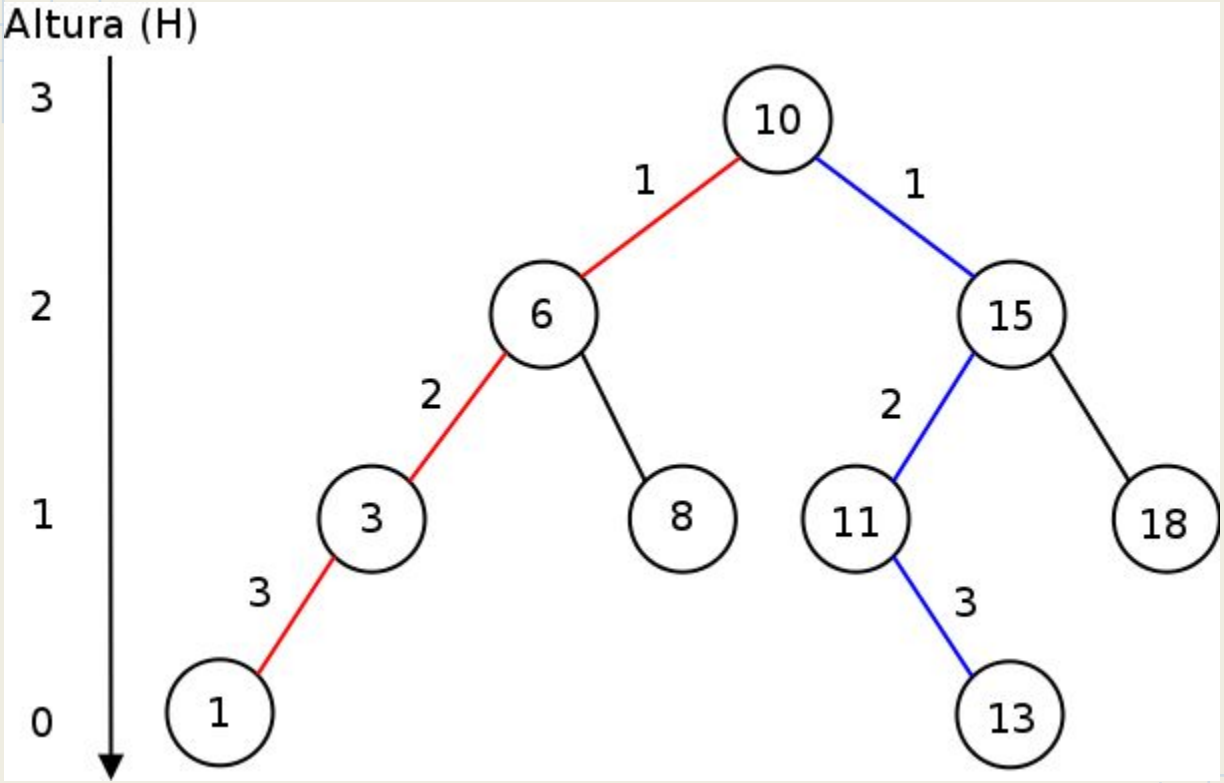
# Revisão



# Altura (h)

- Dentre todos os **caminhos** do nó **Raiz (r)** até as **folhas**, o **comprimento** do **maior caminho** é a **Altura (h)** da Árvore.

# Altura (h)



# Operações Básicas

- **Criar:** inicializarAbb(&abb);
- **Inserir:** inserirNo(&abb, chave);
- **Buscar:** no = buscar(chave, abb);
- **Percurso:** preOrdem(abb); emOrdem(abb); posOrdem(abb);
- ...

# Tipo Definido

```
typedef int TIPOCHAVE;

typedef struct estrutura {
    TIPOCHAVE chave;
    struct estrutura
        *esq,
        *dir;
} NO;
```

# Declaração Variável ABB

```c
int main() {
    NO *abb;// = NULL;

    inicializarArvore(&abb);

    inserirAbb(&abb, 10);
    inserirAbb(&abb, 6);
    inserirAbb(&abb, 15);
    inserirAbb(&abb, 3);
    inserirAbb(&abb, 8);
    inserirAbb(&abb, 11);
    inserirAbb(&abb, 18);
    inserirAbb(&abb, 13);
    inserirAbb(&abb, 1);
```

# Inserir

```c
void inserirAbb(NO **no, TIPOCHAVE ch) {
    if (*no)
        if (ch < (*no)->chave) // vai pra esquerda
            if ((*no)->esq) // esquerda existe
                inserirAbb(&(*no)->esq, ch); // entra na esquerda
            else
                (*no)->esq = novoNo(ch); // esquerda nula, entao atribu
        else // vai pra direita
            if ((*no)->dir) // direita existe
                inserirAbb(&(*no)->dir, ch); // entra na direita
            else
                (*no)->dir = novoNo(ch); // direita nula, entao atribui
    else // raiz
        *no = novoNo(ch);
}
```

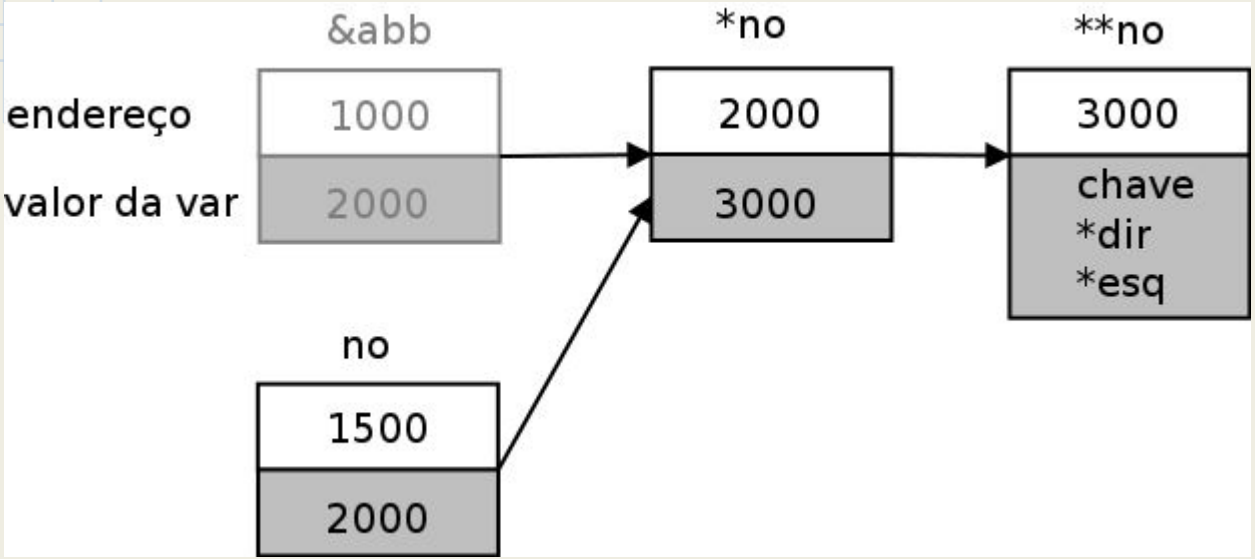
# Novo Nó

```
NO *novoNo(TIPOCHAVE ch) {
    NO *novo = (NO *) malloc(sizeof(NO));
    novo->chave = ch;
    novo->esq = NULL;
    novo->dir = NULL;
    return novo;
}
```

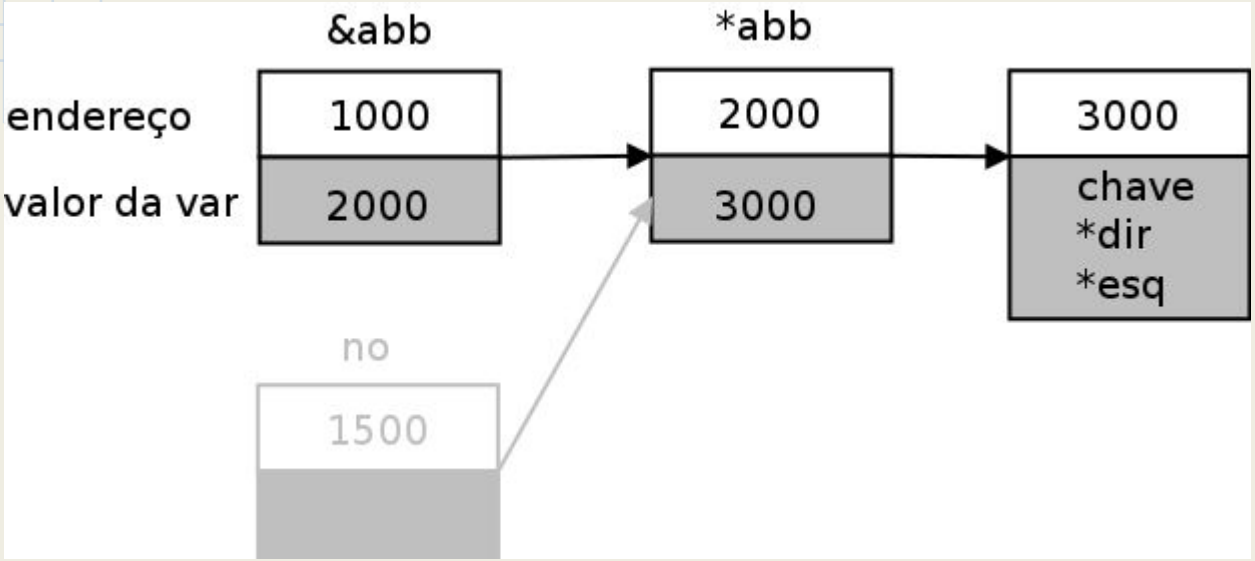
# Ponteiros *_*

```
int main() {
    No *abb = NULL; No *abb;
```

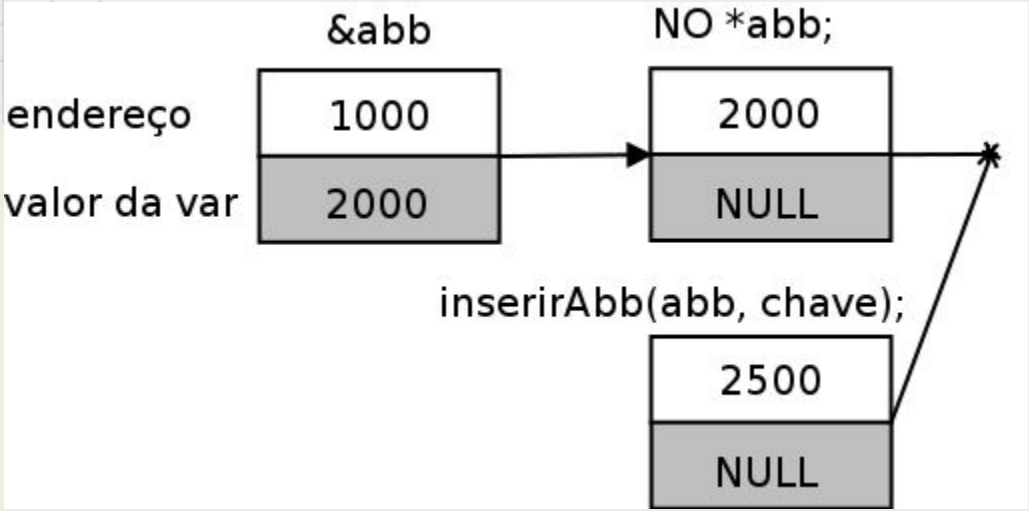
# Ponteiros *_*


&abb
*no
**no
endereço
1000
2000
3000
valor da var
2000
3000
chave
*dir
*esq
no
1500
2000

# Ponteiros *_*


&abb
*abb
endereço
1000
2000
3000
valor da var
2000
3000
chave
*dir
*esq
no
1500

# Ponteiros Erro



# Número Nós

int numNos(NO* no);

**Dica:** Recursivo; Percurso em Árvore.

# Número Nós

```
int numNos(NO *no) {
    int u, v;
    if (no == NULL)
        return 0;
    u = numNos(no->esq);
    v = numNos(no->dir);
    return u + v + 1;
}
```

# Número de Nós

- **u:** representa a quantidade de nós já contabilizados da sub-árvore **esquerda** do nó **u**.

- **v:** representa a quantidade de nós já contabilizados da sub-árvore **direita** do nó **v**.

# Num Nós Iterativo

int numNosIt(NO *no);

Dica: estrutura de dados -> pilha.

# Num Nós Iterativo

```c
int numNosIt(NO *no) {
    NO_PI *pi = NULL;
    int numNos = 0;

    if (no)
        empilha(&pi, no);
    while (pi) {
        no = desempilha(&pi);
        numNos += 1;
        if (no->esq)
            empilha(&pi, no->esq);
        if (no->dir)
            empilha(&pi, no->dir);
    }
    //freePilha(pi);
    return numNos;
}
```

# Alg. Recursivo x Iterativo

- **Recursivo:**
  - +recursos (memória);
  - -complexo;
- **Iterativo:**
  - -recurso;
  - +complexo;

**Linguagens funcionais** tem uma implementação de recursão eficiente.

# Altura

int altura(NO* no);

**Dica:** Recursivo; Percurso em Árvore; pensar em na contabilização da altura de **u** e **v** também.

# Altura

```
int altura(NO *no) {
    int u, v;
    if (no == NULL)
        return -1;
    u = altura(no->esq);
    v = altura(no->dir);
    if (u > v)
        return u+1;
    else
        return v+1;
}
```

# Altura

- **u:** representa a altura já contabilizada da sub-árvore **esquerda** do nó **u**.

- **v:** representa a altura já contabilizada da sub-árvore **direita** do nó **v**.

# Número de Folhas

int numFolhas(NO *no);

**Dica:** Percurso em Árvores (pre, em, pos); da pra resolver pensando em **u** em **v** também, porém existem outras formas.

# Número de Folhas

```c
int ehFolha(NO *p) {
    if (!p->esq && !p->dir)
        return 1;
    else
        return 0;
}

int numFolhasPre(NO *p) {
    if (p)
        return ehFolha(p) + numFolhasPre(p->esq) + numFolhasPre(p->dir);
    else
        return 0;
}
```

# Busca

NO *busca(NO *no, TIPOCHAVE chave);

# Busca

```
NO *busca(NO *no, TIPOCHAVE chave) {
    if (no) {
        if (no->chave == chave)
            return no; // retorna o no da chave
        else if (chave < no->chave)
            return busca(no->esq, chave);
        else if (chave > no->chave)
            return busca(no->dir, chave);
    } else // nao achou retorna NULL
        return NULL;
}
```

# teste ABB

int test_abb(NO *no);

Retorna 1 se for ABB; 0 caso contrário.

# teste ABB Recursivo

```c
int test_abb(NO *no) {
    int retEsq,
        retDir;

    if (no) {
        if (no->esq)
            if (no->chave > no->esq->chave)
                retEsq = test_abb(no->esq);
            else
                return 0;
        else
            retEsq = 1;
        if (no->dir)
            if (no->chave < no->dir->chave)
                retDir = test_abb(no->dir);
            else
                return 0;
        else
            retDir = 1;
        return retEsq && retDir;
    } else
        return 1;
}
```

# teste ABB Iterativo

```c
int test_abbIt(NO *no) {
    NO_PI *pi = NULL;

    if (no)
        empilha(&pi, no);
    while (pi) {
        no = desempilha(&pi);
        if (no->esq)
            if (no->chave > no->esq->chave)
                empilha(&pi, no->esq);
            else
                return 0;
        if (no->dir)
            if (no->chave < no->dir->chave)
                empilha(&pi, no->dir);
            else
                return 0;
    }
    //freePilha(pi);
    return 1;
}
```

# Test Driven Development

- Escreve o Teste antes da funcionalidade;
- O teste não vai passar (pois a funcionalidade não existe);
- Escreve a funcionalidade até passar pelo teste;
- Continua esse processo...

# Remover

```
void removerAbb(NO **no, TIPOCHAVE chave);
```

# Regras para Remover

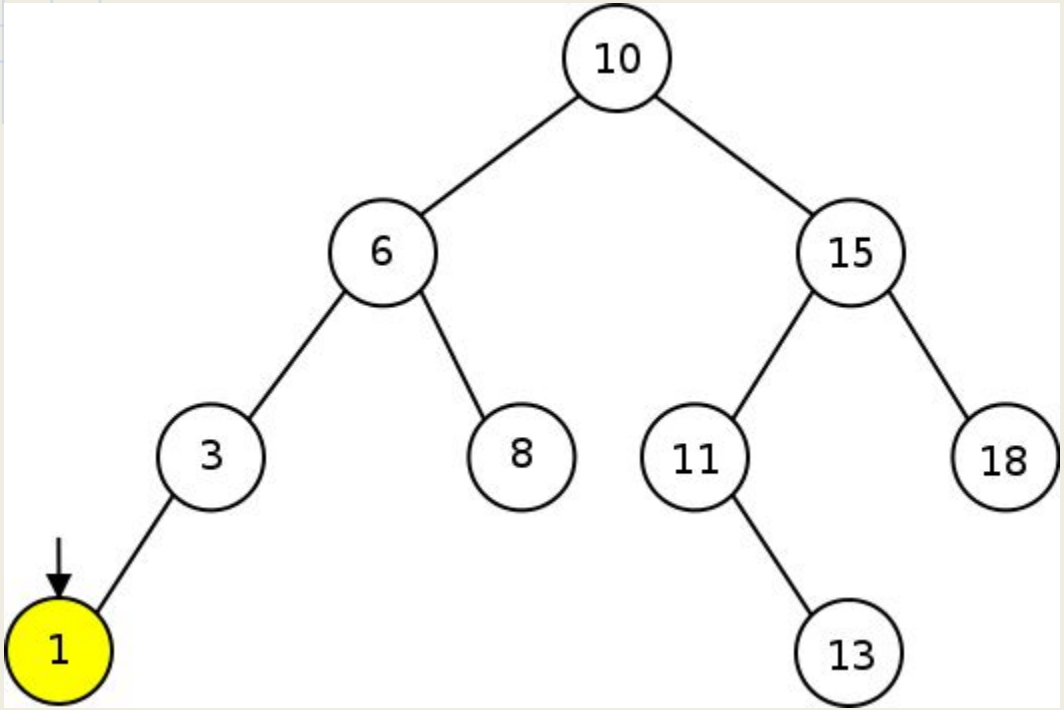
1. Se o nó a ser removido não tiver filhos, basta atualizar o ponteiro correto do seu pai para NULL;
2. Se o nó a ser removido tiver um filho, basta atualizar o ponteiro do correto do seu pai (esq ou dir) para o filho correto (esq ou dir) do nó a ser removido;
3. Se o nó a ser removido tiver 2 filhos, procura-se o maior descendente a esquerda do nó a ser removido, ou o menor descendente a direita do nó a ser removido;
    a. Se o nó encontrado não tiver filhos: aplica-se a regra **1.** para esse nó;
    b. Caso ele tenha um filho: aplica-se a regra **2.** para esse nó.
    c.

**É necessário garantir que o nó promovido mantenha a propriedade da ABB: Maior dos menores, ou menor dos maiores garante isso pós remoção**

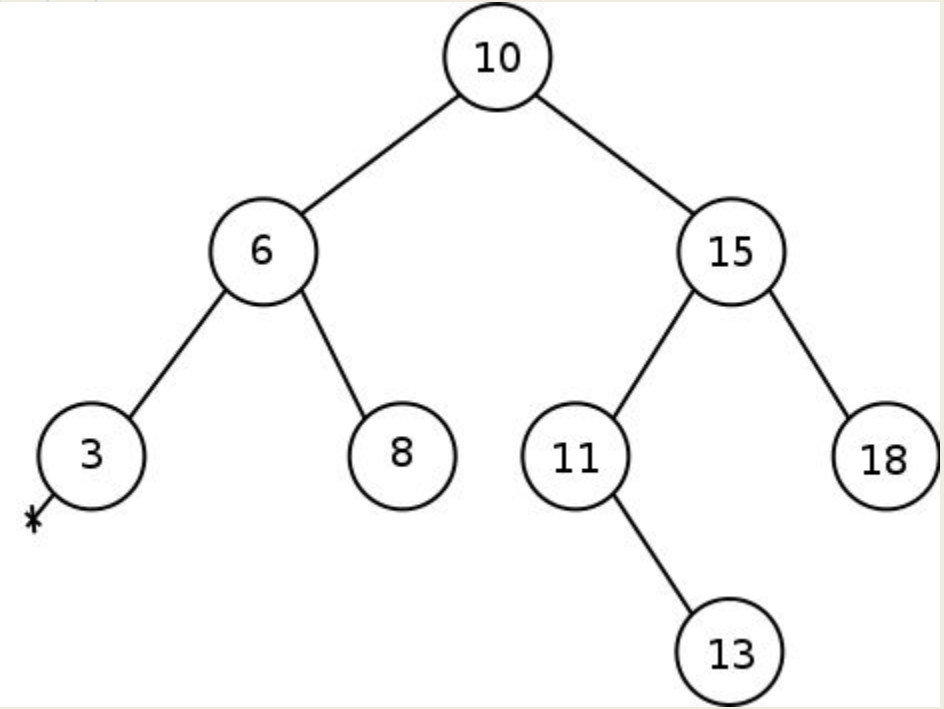
# Detalhes de Implementação

1. Cuidados do caso Caso 3:
   a. Se o Pai do Nó promovido for o próprio nó a ser removido, a ligação do Pai promovido deve ser com seu ponteiro filho da esquerda;
   b. Se o Pai do Nó promovido for qualquer outro nó, a ligação do Pai promovido deve ser com o seu ponteiro filho da direita;
   c. Se o Nó promovido tiver filho (apenas 1), será o da esquerda, e a ligação com o Pai do nó promovido deve seguir as regras **a.** e **b.**
   d. Caso o Nó promovido não tiver filho, o nó Pai do nó promovido recebe NULL de acordo com as regras **a.** e **b.**

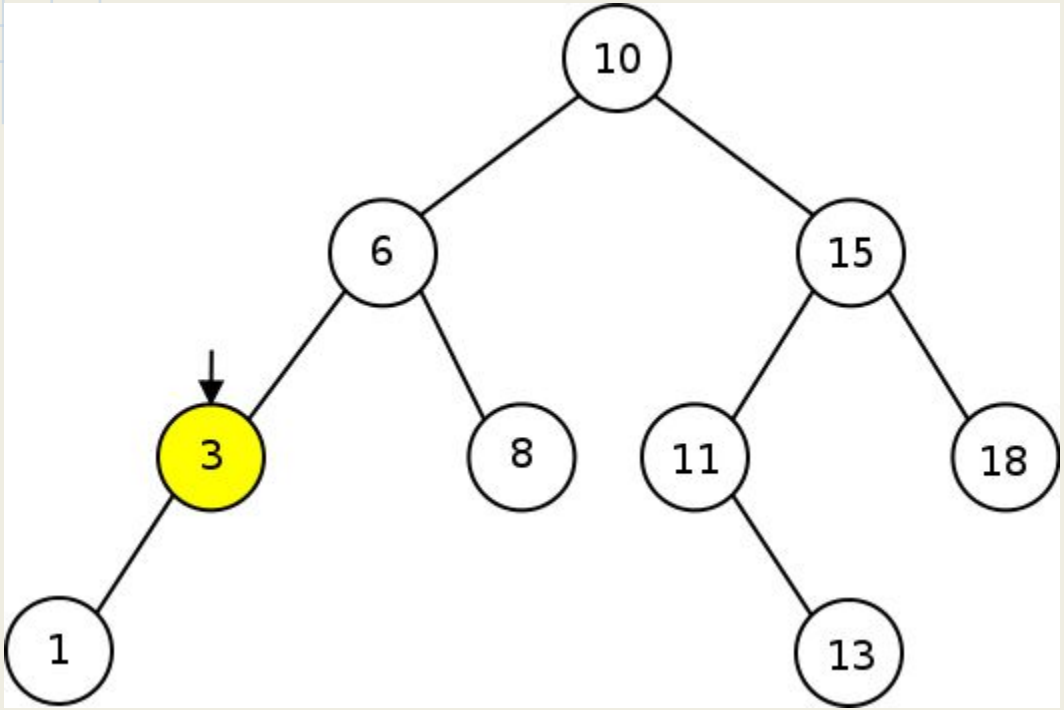
# Remover Folha



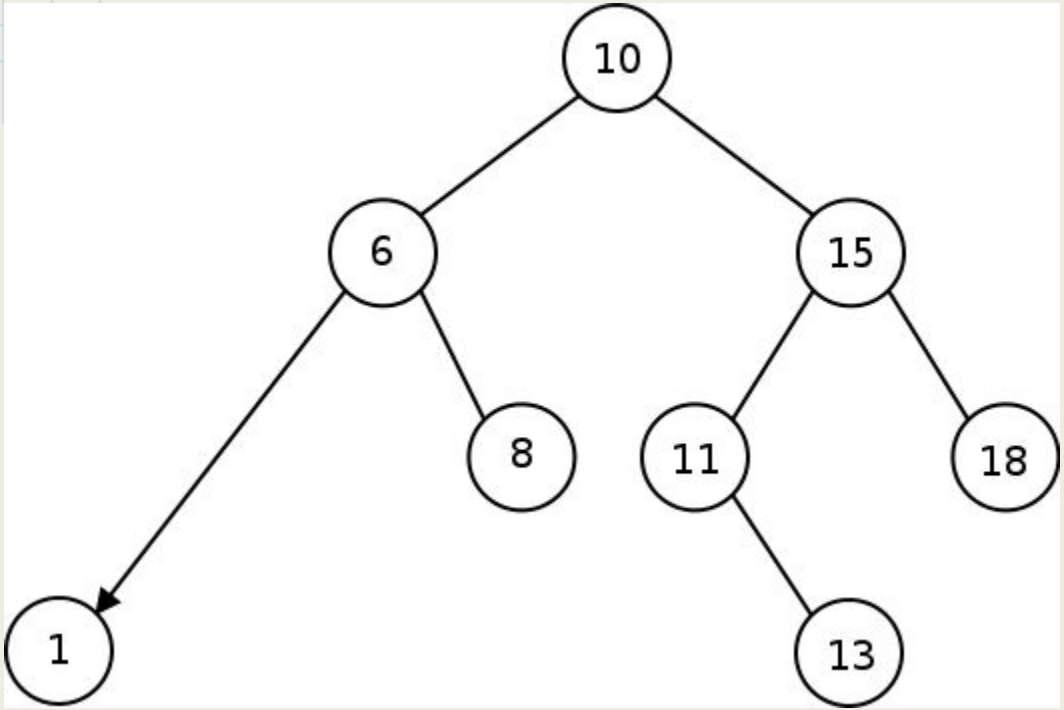
# Remover Folha



# Remover com 1 Filho

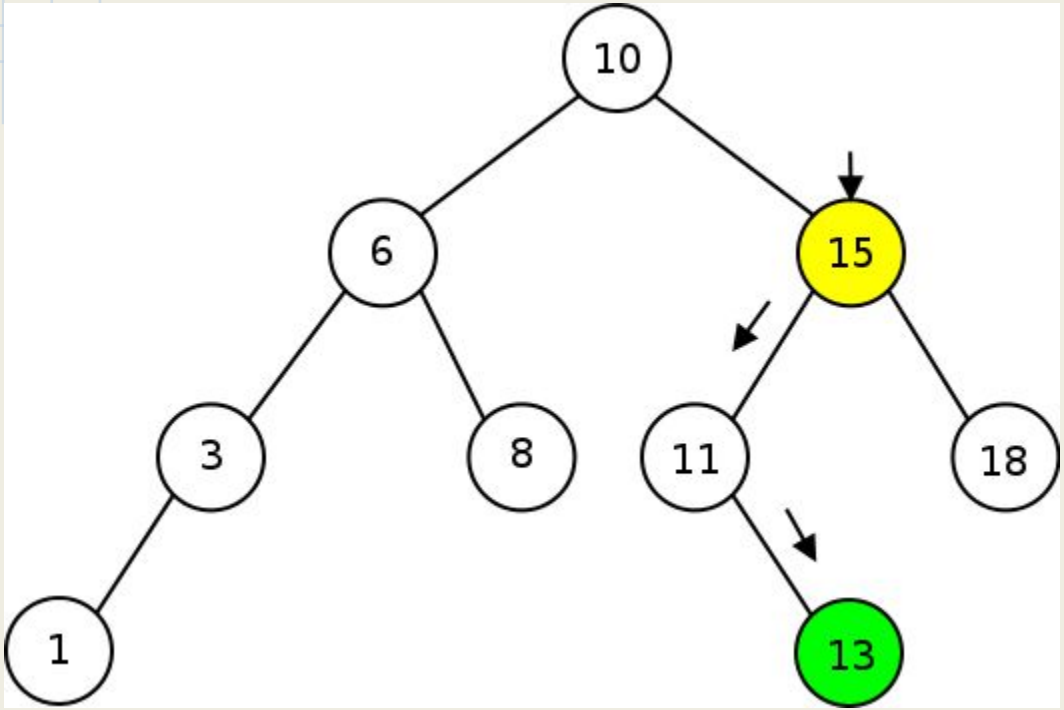


# Remover com 1 Filho

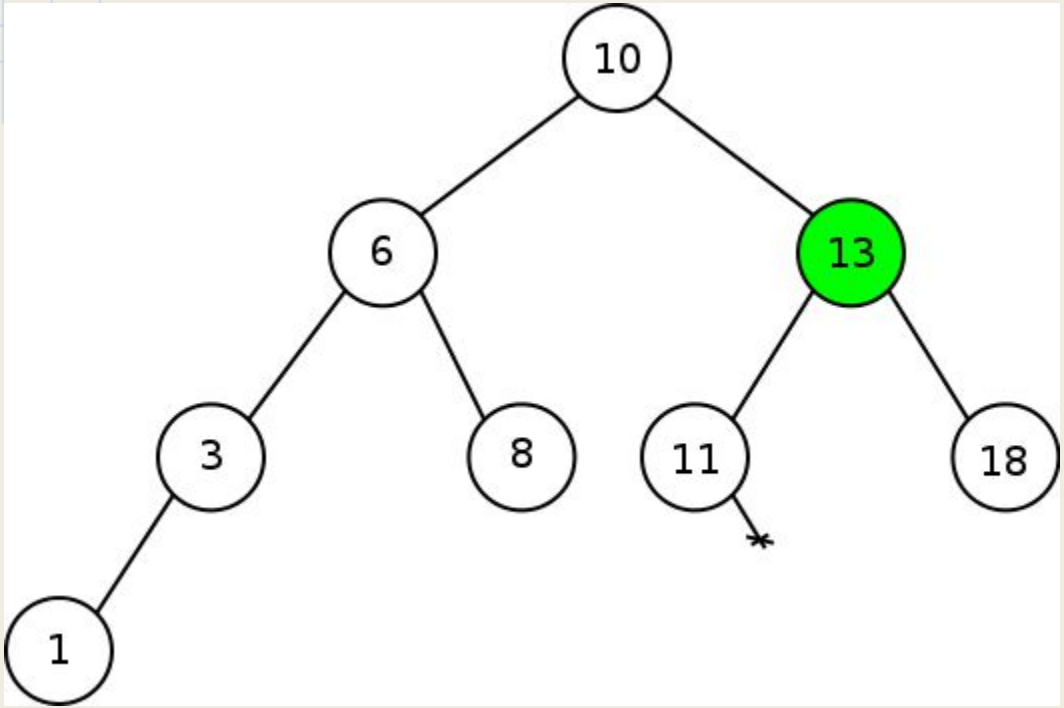


# Remover com 2 Filhos

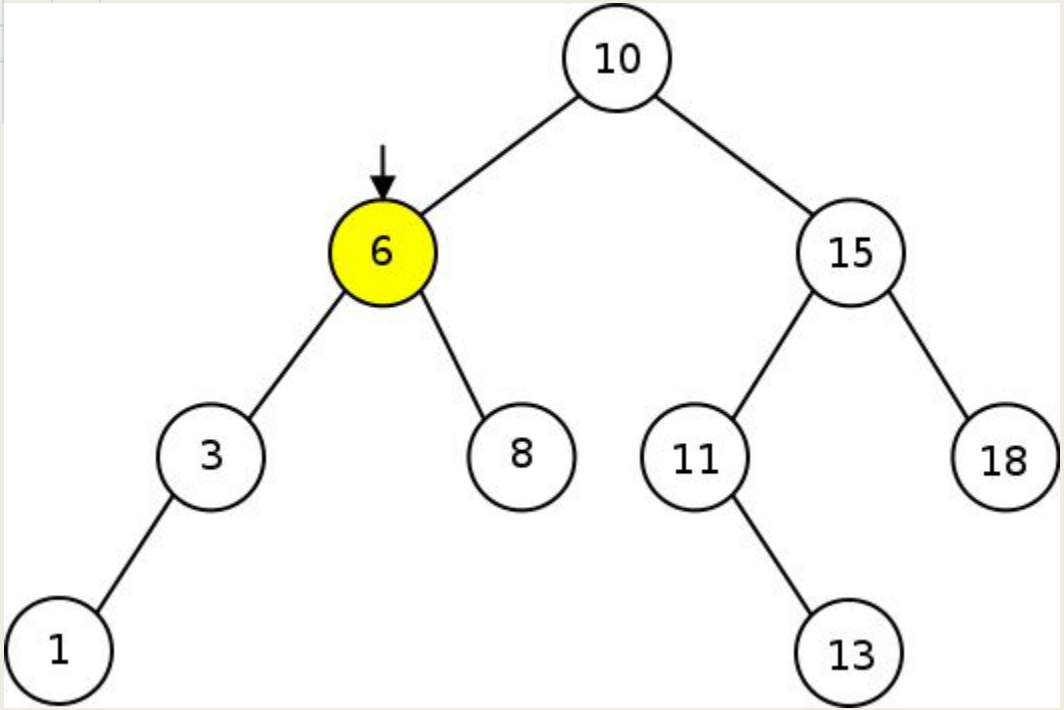
# Remover com 2 Filhos



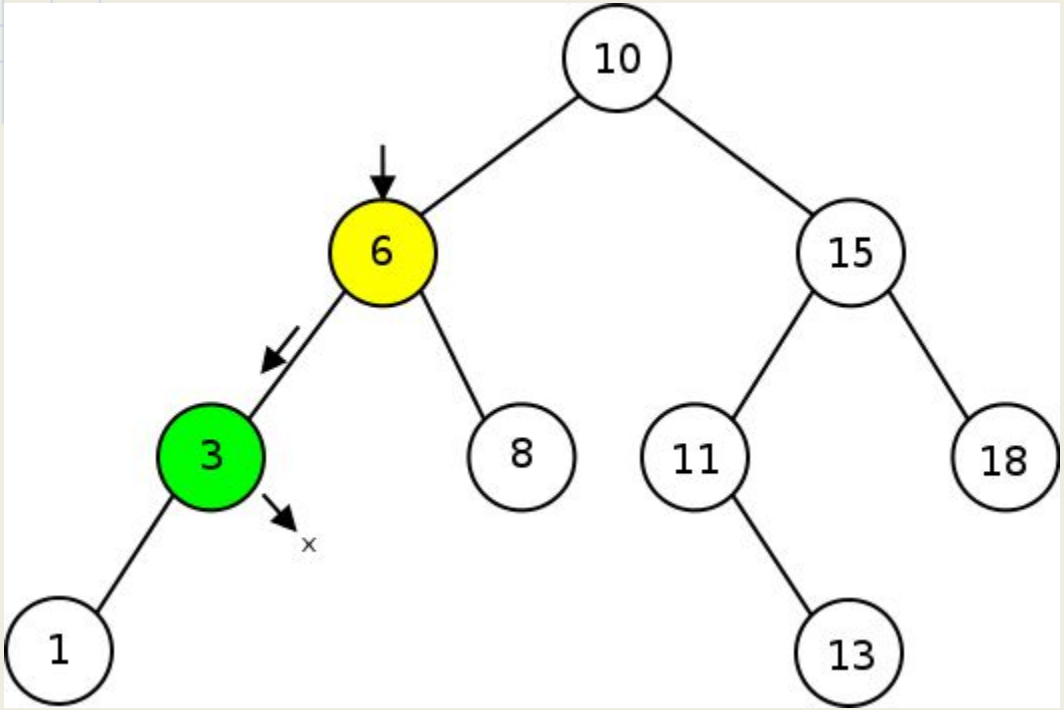
# Remover com 2 Filhos



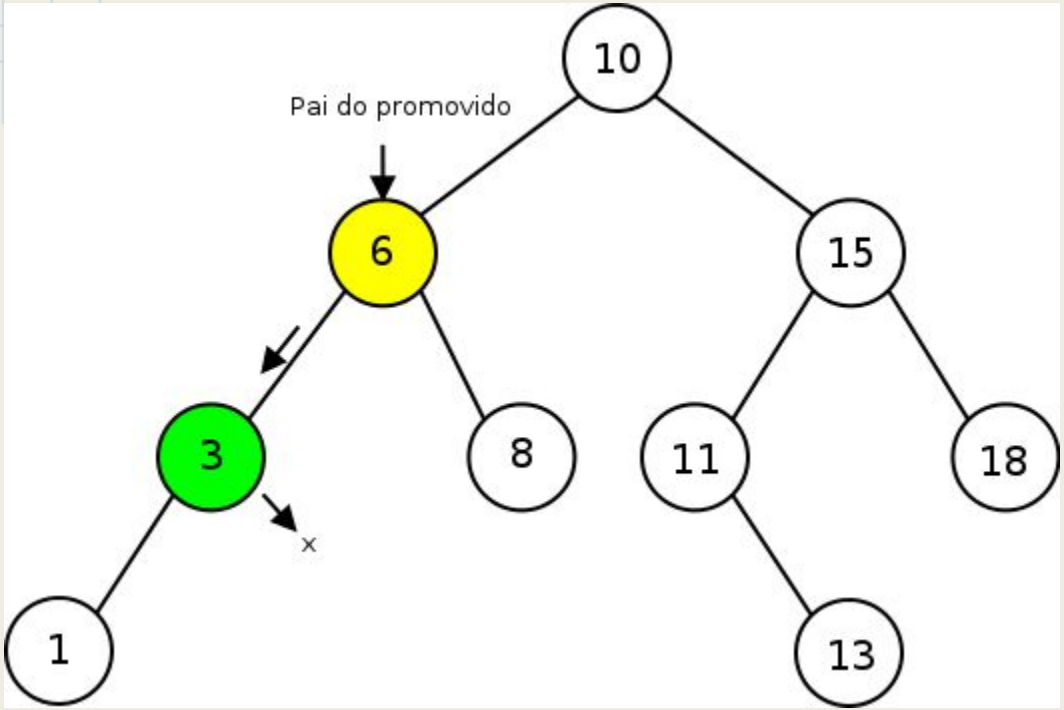
# Remover com 2 Filhos



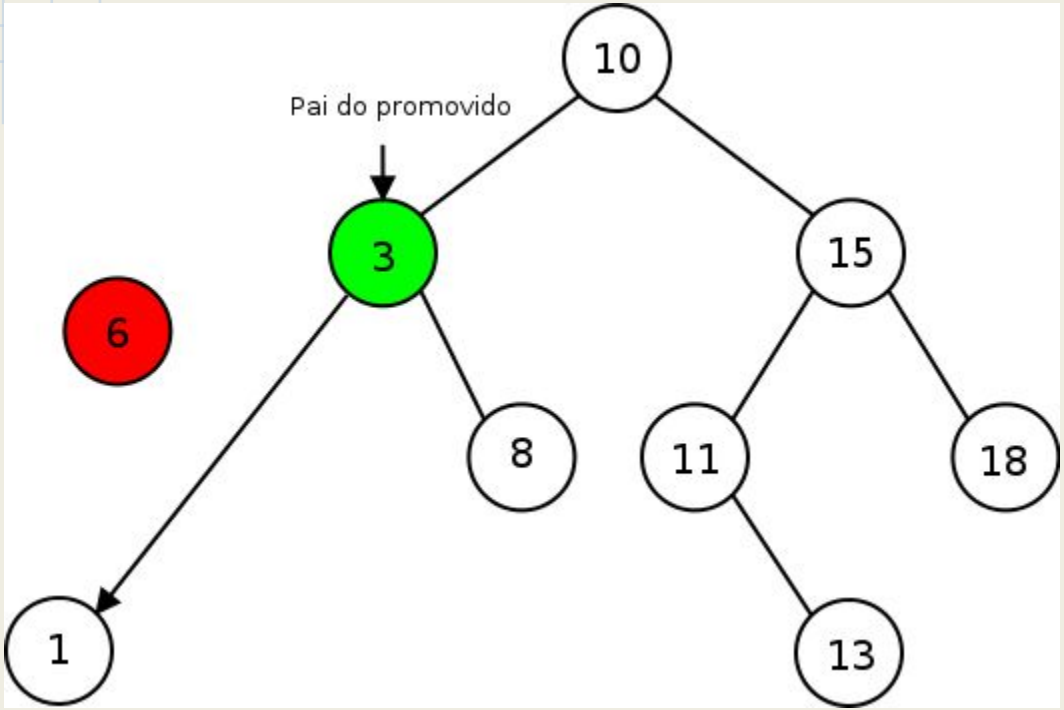
# Remover com 2 Filhos



# Remover com 2 Filhos



# Remover com 2 Filhos



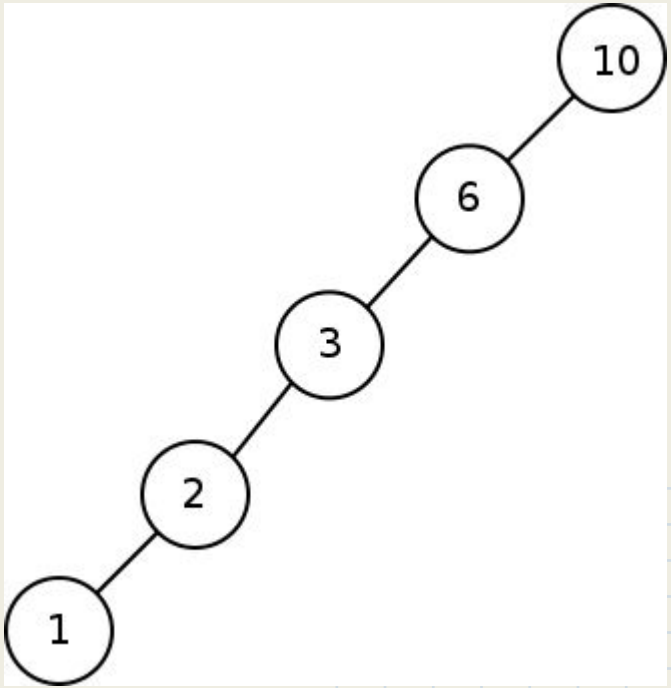
# Árvore Binária de Busca

- Busca pode ser ineficiente devido ao balanceamento da árvore;
- Não garante por si só balanceamento;

# Árvore Binária de Busca

Sub-árvores de cada nó com diferenças de nível muito grande;

É necessário um mecanismo de balanceamento dessas ABBs.



# Árvore AVL

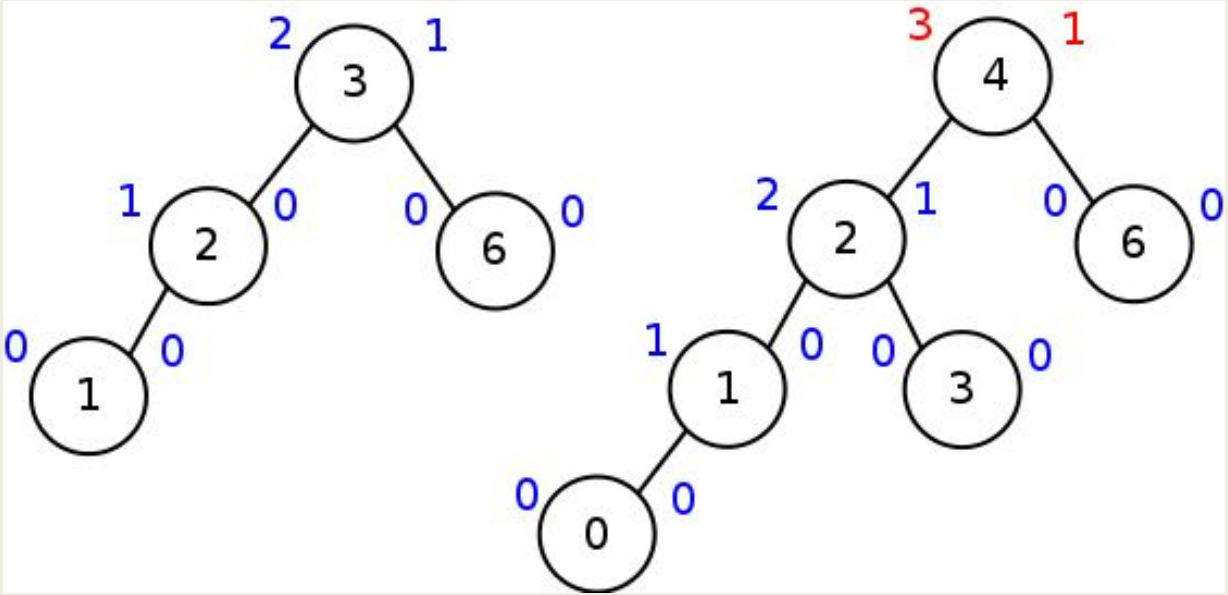
- Mantém a árvore balanceada
  - Gera buscas eficientes;
  - Porém Inserções e remoções fazem operações extras para manter a árvore balanceada;
  - Essas operações são de rotacionamento de nós.
  - **Quando a diferença de nível das sub-árvores (esquerda e direita) de um dado nó for maior do que 1, é necessário rotacionar.**

# Árvore AVL

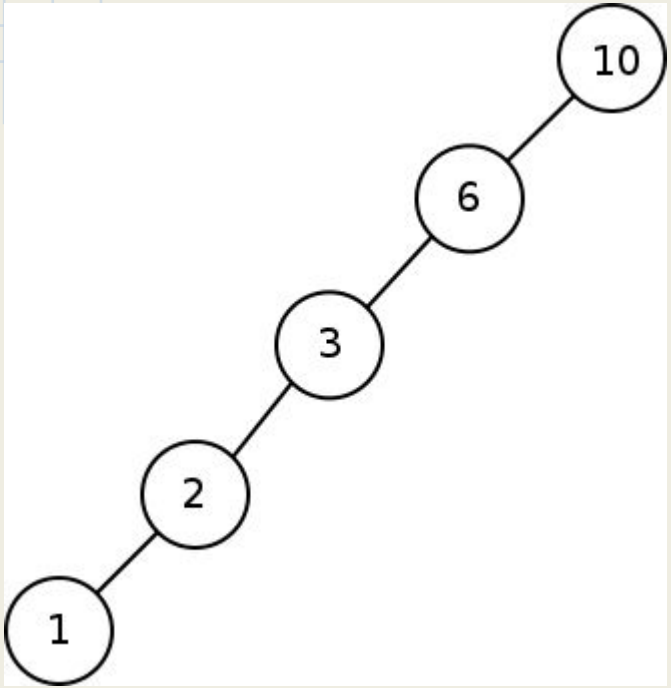
**Propriedade da Árvore AVL:**
Para cada nó da Árvore AVL, suas sub-árvores (esquerda e direita) tem uma diferença de níveis de no máximo 1;
**A** = conjunto de nós da árvore
**h(x)** = altura de um nó **x**;
**fesq(x)** = filho à esquerda de **x**;
**fdir(x)** = filho à direita de **x**;

$$\forall x \in A, |h(fesq(x)) - h(fdir(x))| \leq 1$$

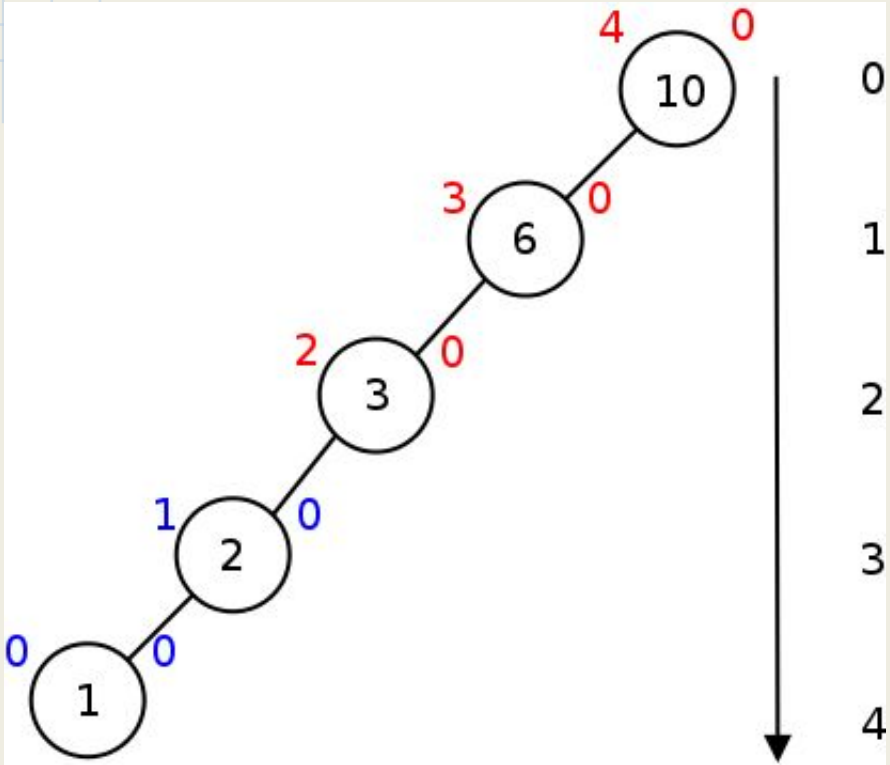
# Árvores AVL



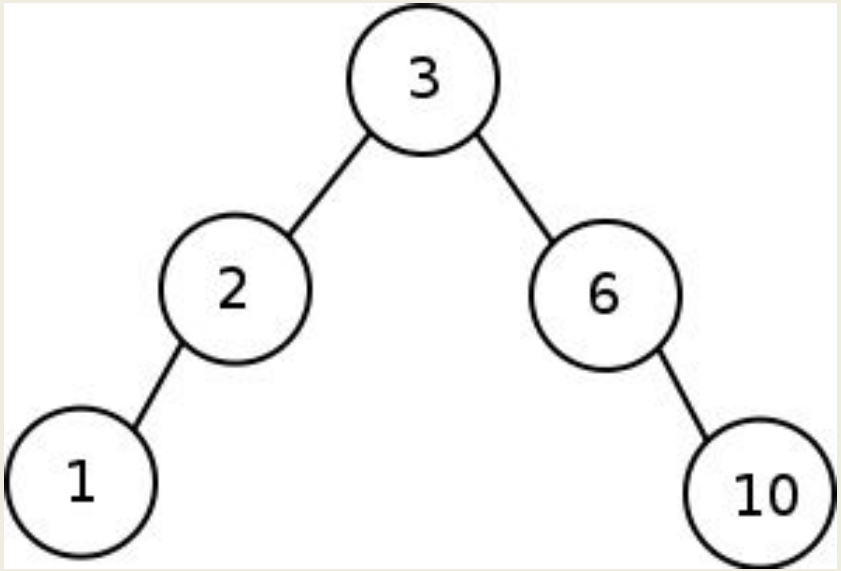
# Ex:



# Ex para Balancear

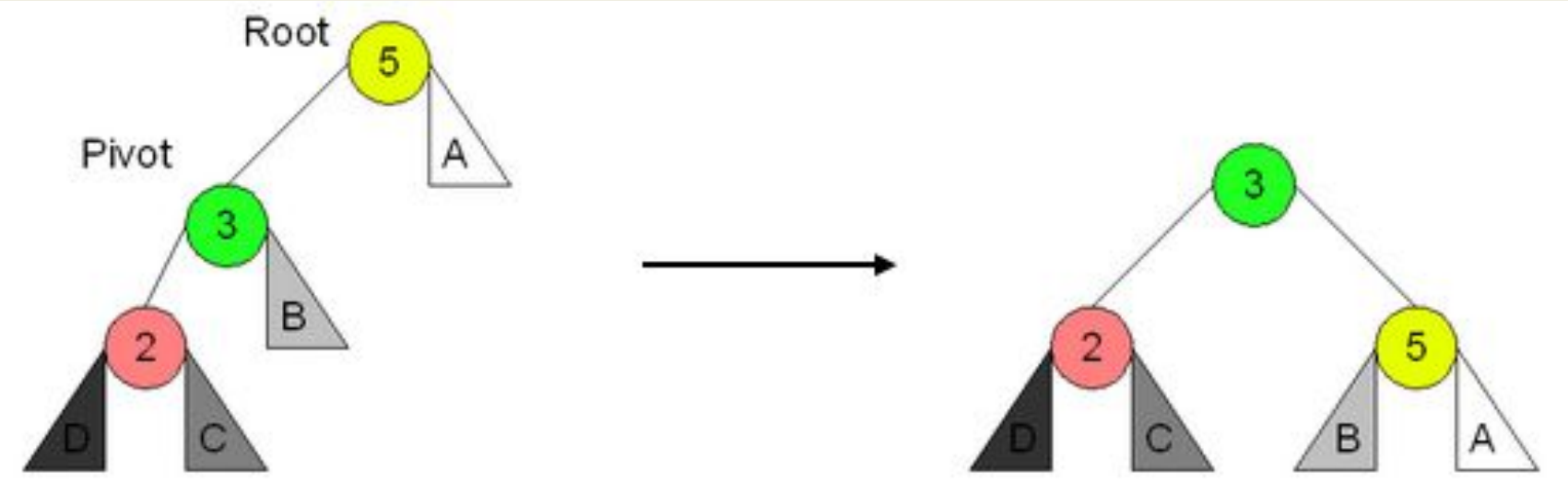


# Ex:



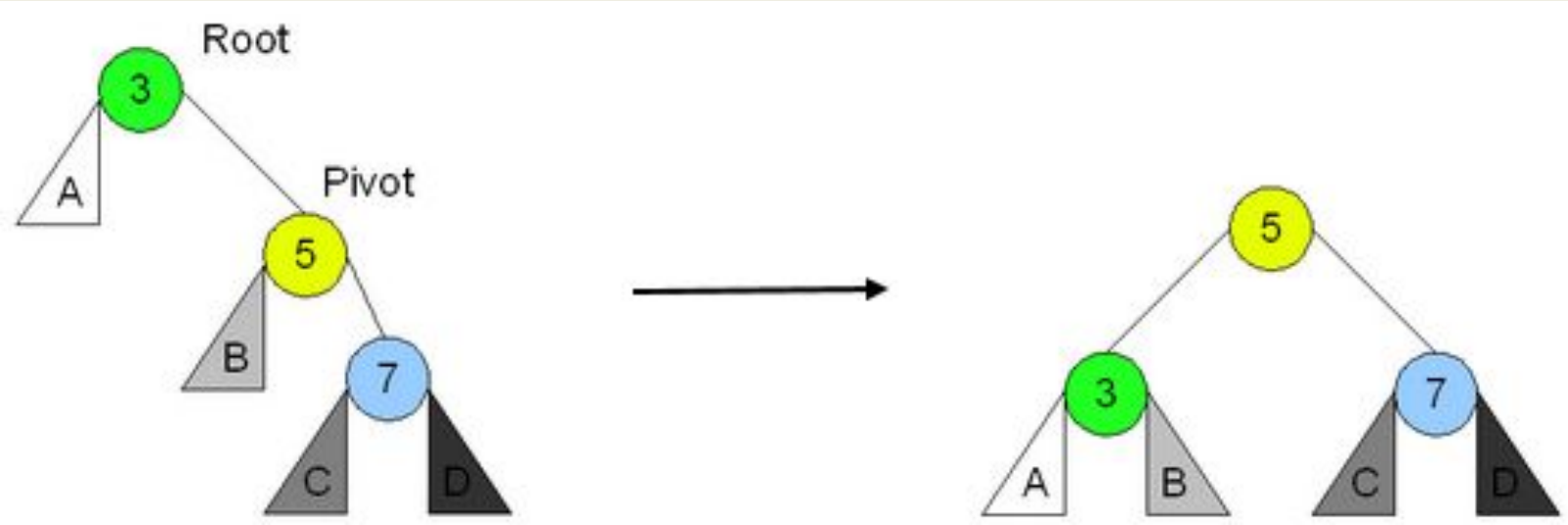
# Árvore AVL

Rotacionamento à direita:

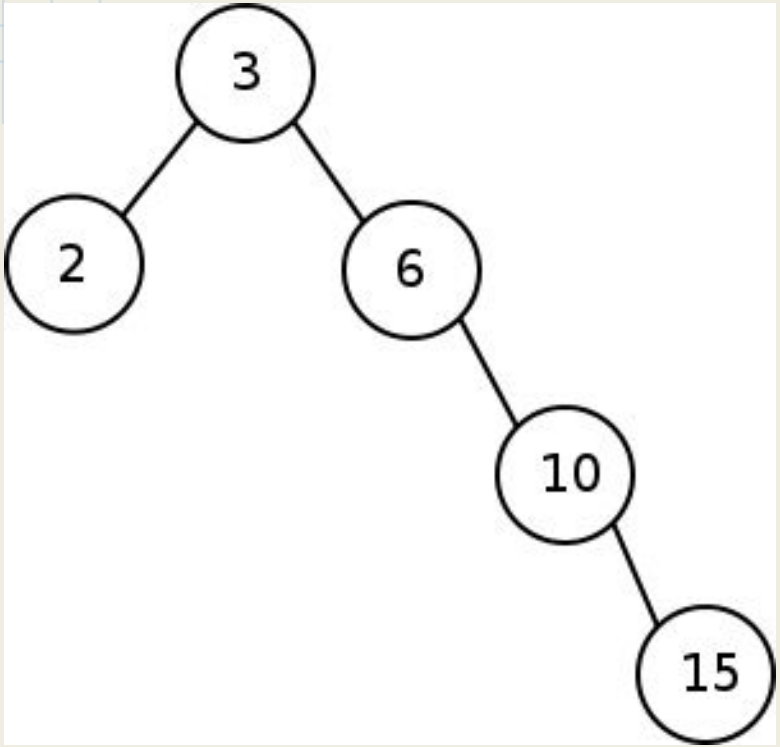


# Árvore AVL

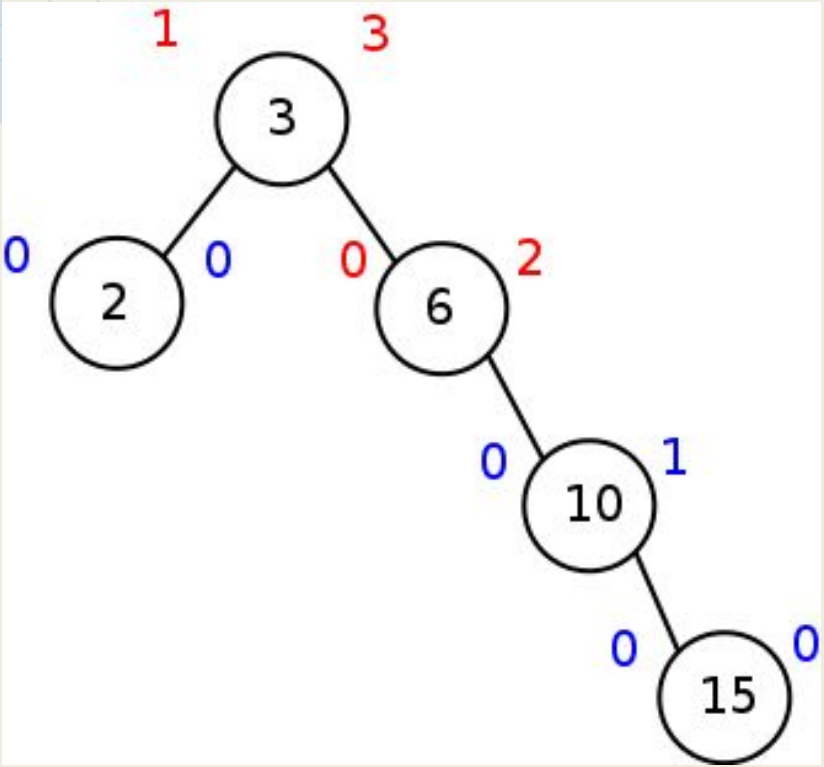
Rotacionamento à esquerda:



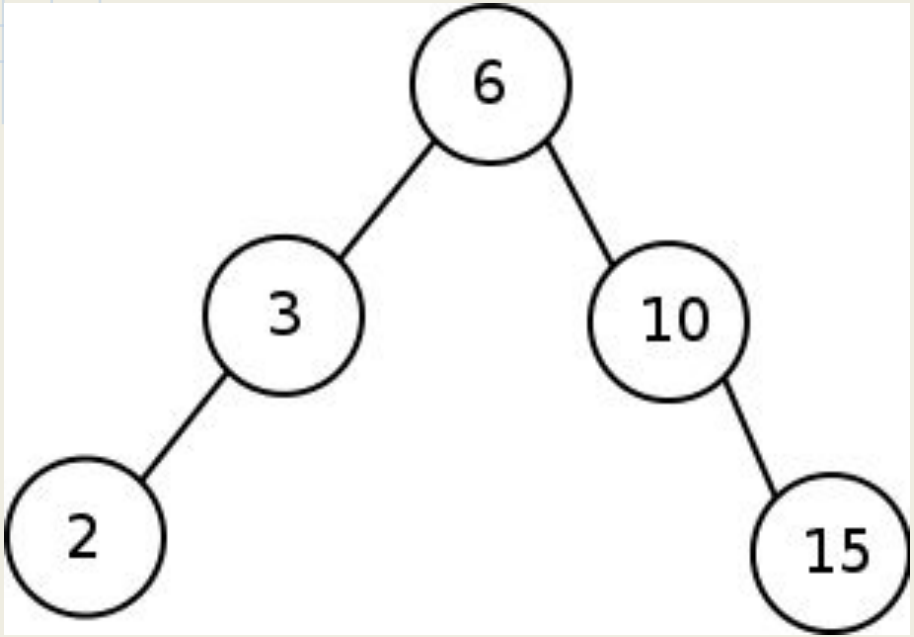
# Ex:



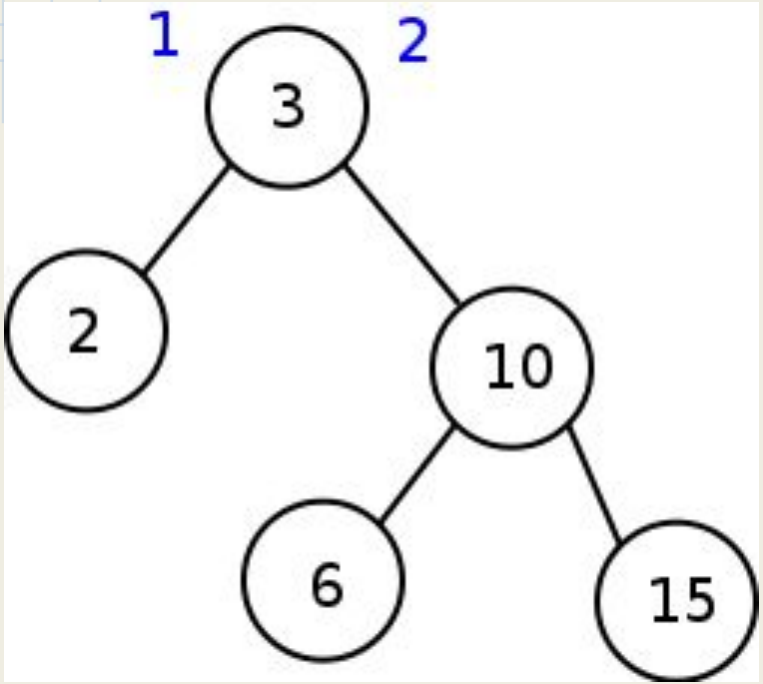
# Ex:



# Ex:

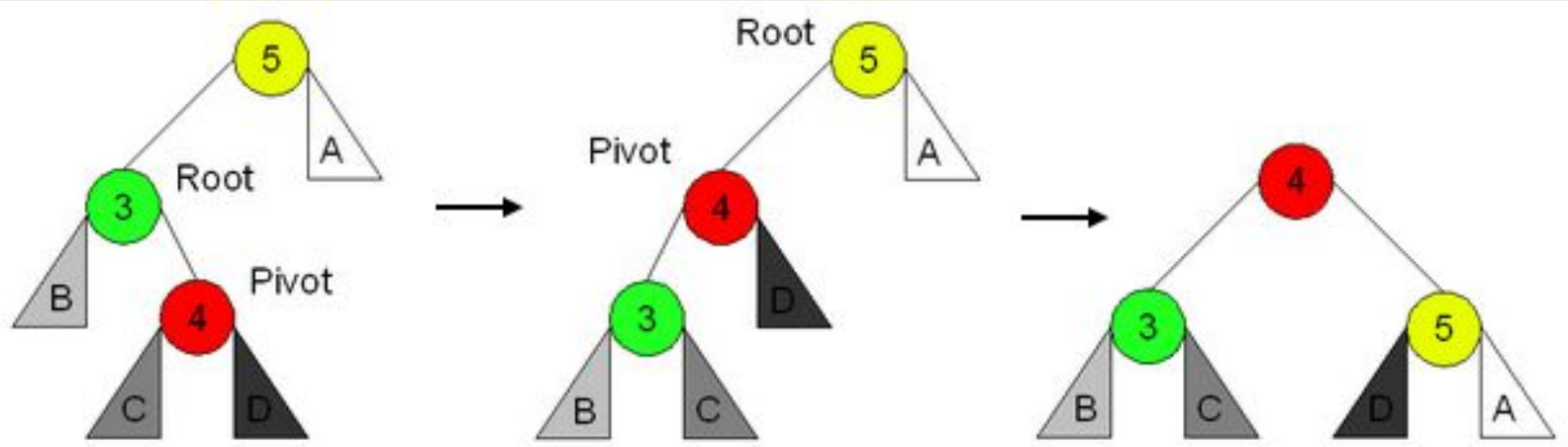


# Ex



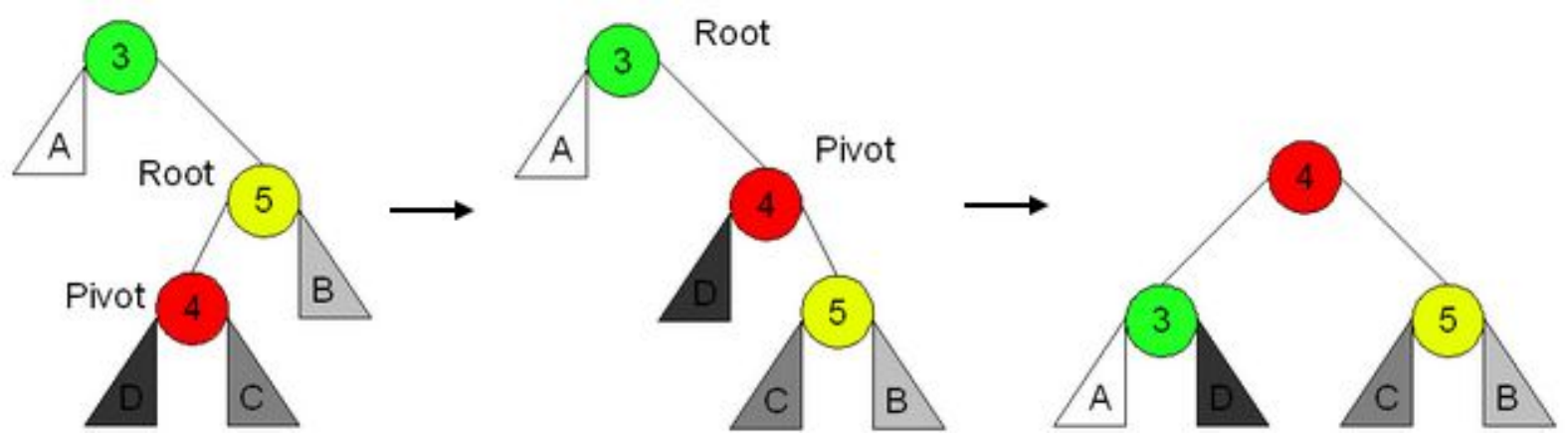
# Árvore AVL

## Rotação dupla à direita:



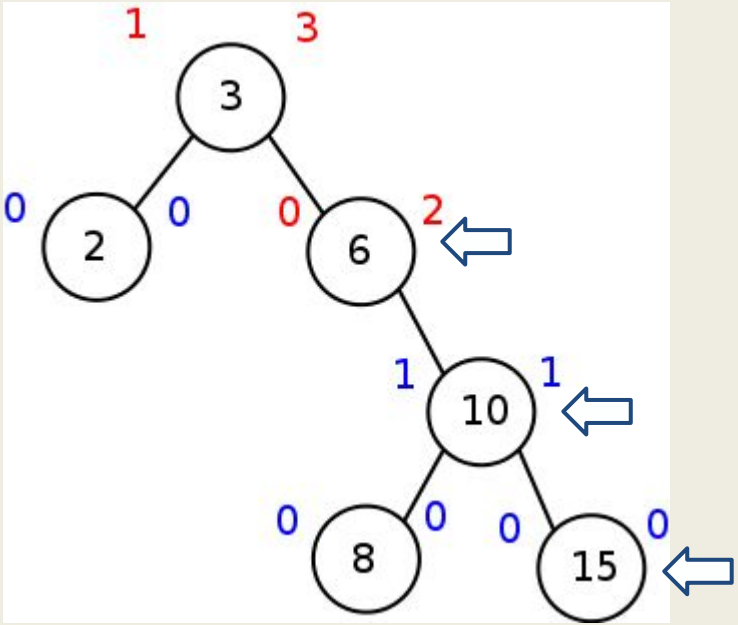
# Árvore AVL

## Rotação dupla à esquerda:

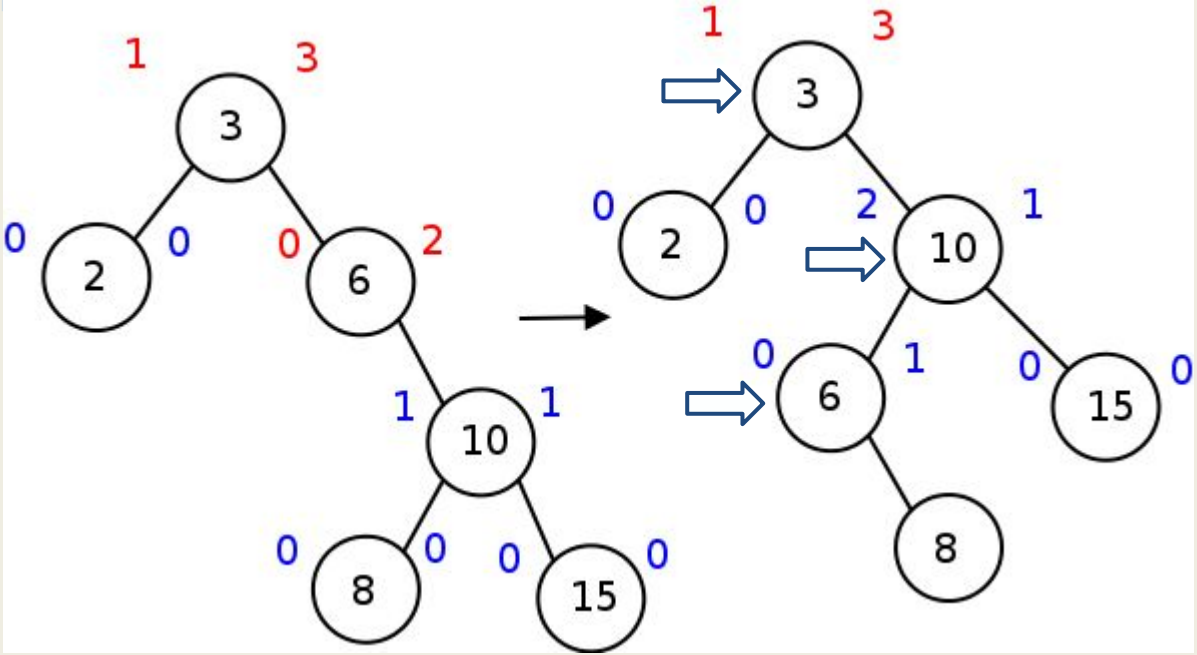


# Ex

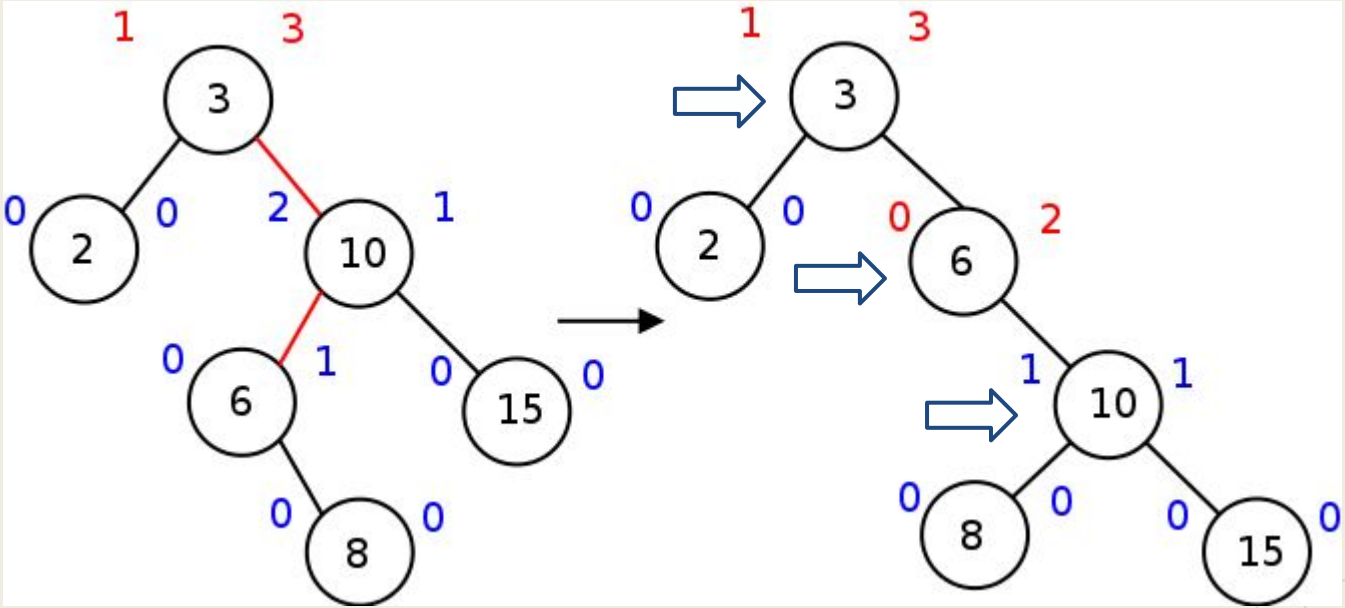
## Rotacionar 6 10 15



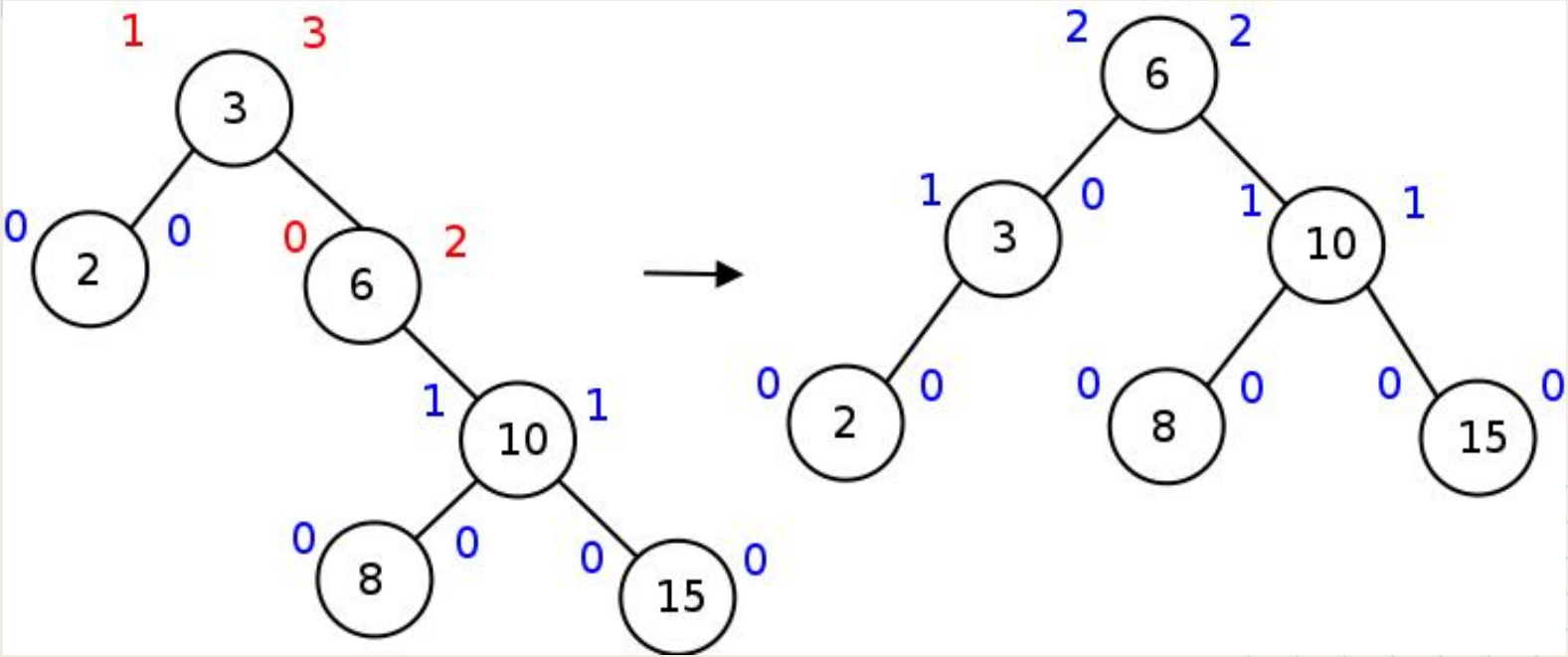
# Ex



# Ex

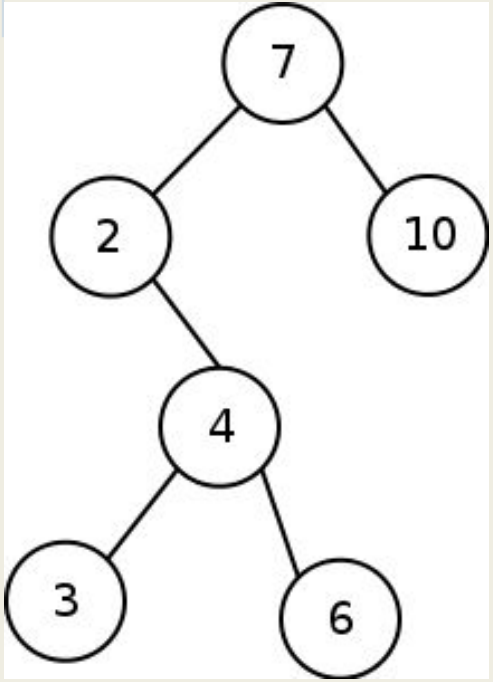


# Ex



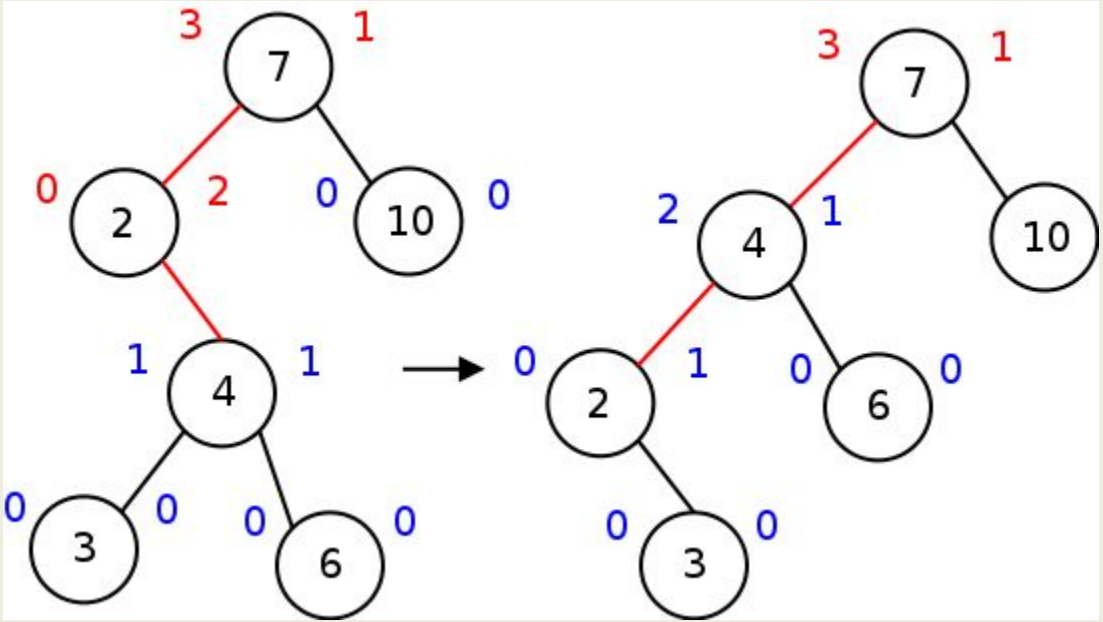
# Ex

Rotacionar 7 2 4

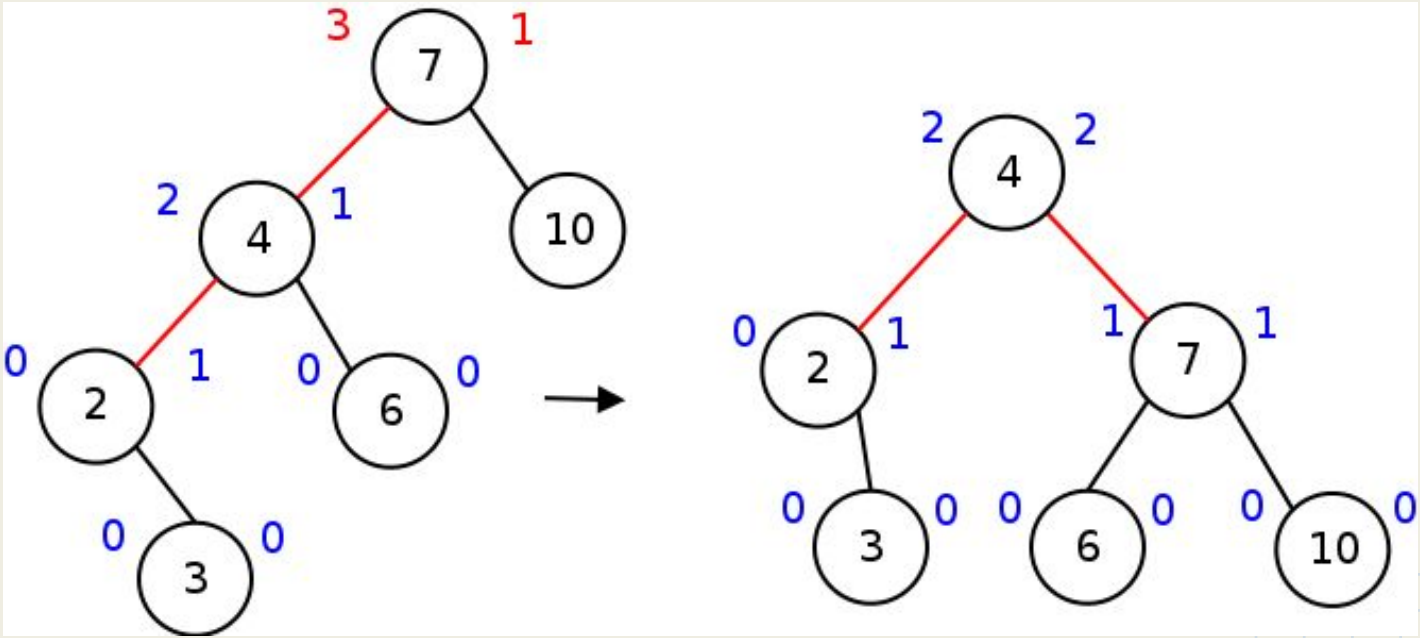


# Ex

Ex



# Ex



# Árvore AVL

- A cada inserção, verifica-se se a propriedade da Árvore AVL é mantida, caso não seja, aplica-se o rotacionamento (simples ou duplo, caso necessário);
- A cada remoção o mesmo deve ser feito;
- Dessa forma a Árvore é sempre mantida balanceada.

http://www.ime.usp.br/~pf/mac0122-2002/aulas/trees.html