

Algoritmos e Estruturas de Dados I

Aula 11: Árvores de Pesquisa

Prof. Márcio Porto Basgalupp

créditos: Prof. Jurandy G. Almeida Jr.

Universidade Federal de São Paulo
Departamento de Ciência e Tecnologia

- A árvore de pesquisa é uma estrutura de dados muito eficiente para armazenar e recuperar informação
- A informação é dividida em registros, em que cada registro possui uma chave para ser usada na pesquisa
- **Objetivo da pesquisa:** Encontrar uma ou mais ocorrências de registros com chaves iguais à chave de pesquisa
- **Pesquisa com sucesso X sem sucesso**

- É importante considerar árvores de pesquisa como tipos abstratos de dados (TADs), de forma que haja independência da implementação para as operações
- Operações mais comuns:
 1. Inicializar a estrutura de dados
 2. Pesquisar um ou mais registros com determinada chave
 3. Inserir um novo registro
 4. Retirar um registro específico

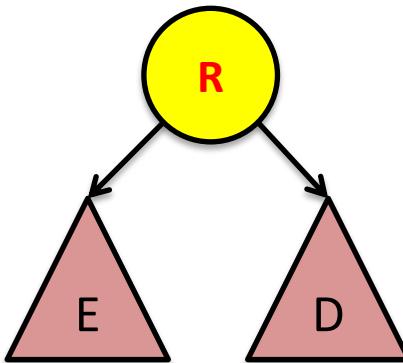
- **Dicionário** é um TAD com as operações:
 1. Inicializa
 2. Pesquisa
 3. Insere
 4. Retira
- Analogia com um dicionário da língua portuguesa:
 - chaves \Leftrightarrow palavras
 - registros \Leftrightarrow entradas associadas com cada palavra:
pronúncia, definição, sinônimos, outras informações, etc

■ Conjunto de Operações

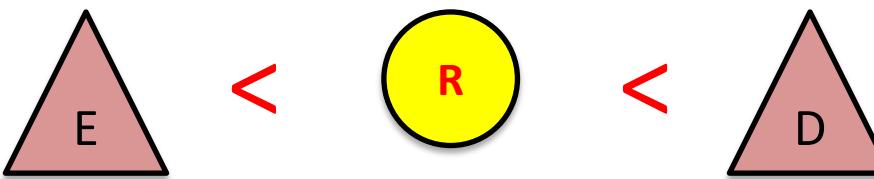
- **TABB_Inicia()**: Inicia uma árvore binária de busca vazia
- **TABB_Pesquisa(ABB, c)**: Retorna a árvore binaria de busca contendo o item com chave c ; caso não haja nenhum item com chave c na árvore, retorna uma árvore vazia
- **TABB_Insere(ABB, x)**: Insere o item x com chave c na árvore binária de busca e retorna *true*, caso ele não esteja na árvore; caso contrário, retorna *false*
- **TABB_Remove(ABB, c)**: Retira o item com chave c na árvore binária de busca e retorna *true*, caso ele esteja na árvore; caso contrário, retorna *false*

Árvores Binárias de Pesquisa

- Para qualquer nó que contenha um registro:



- Temos a relação invariante:



1. Todos os registros com chaves menores estão na subárvore à esquerda
2. Todos os registros com chaves maiores estão na subárvore à direita

Estrutura da Árvore Binária

```
typedef int TChave;

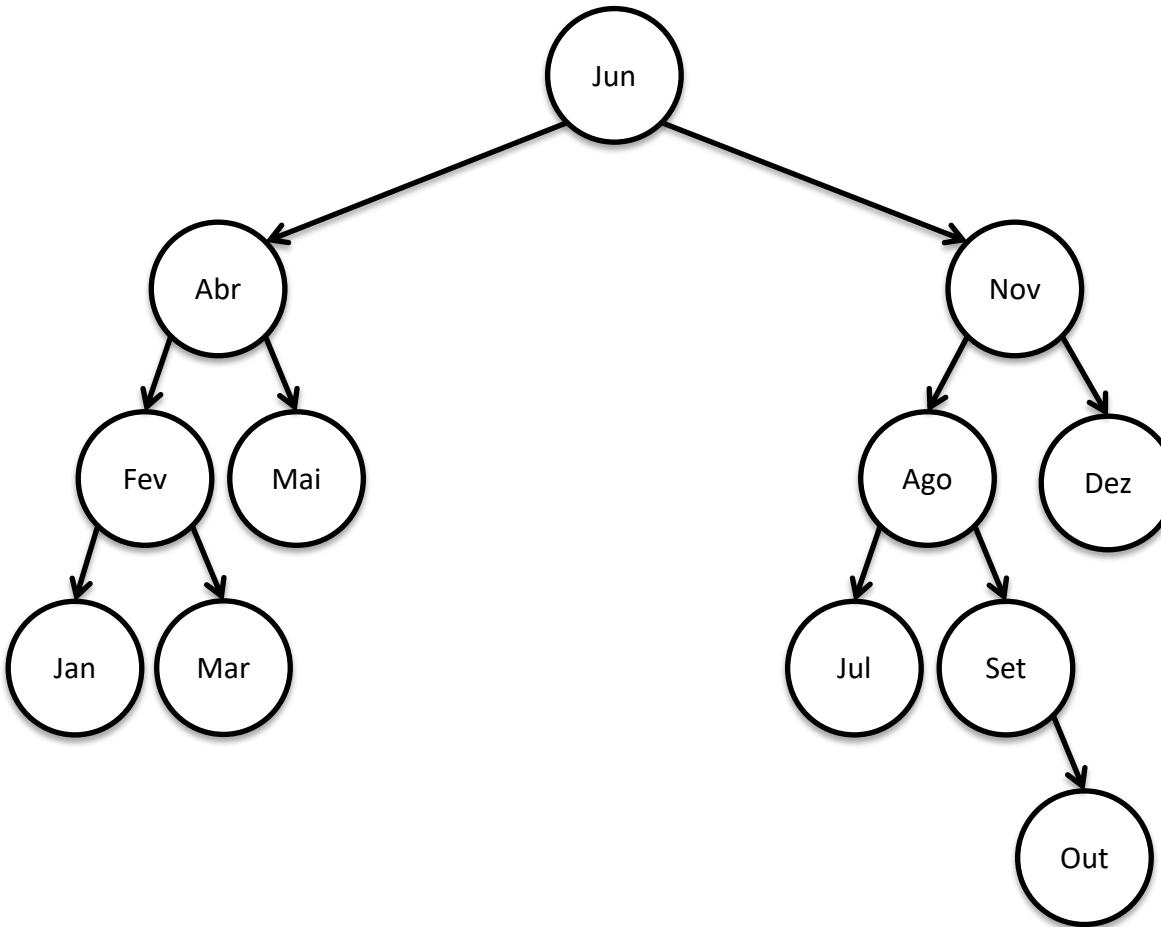
typedef struct {
    TChave Chave;
    /* outros componentes */
} TItem;

typedef struct SNo *TArvBin;

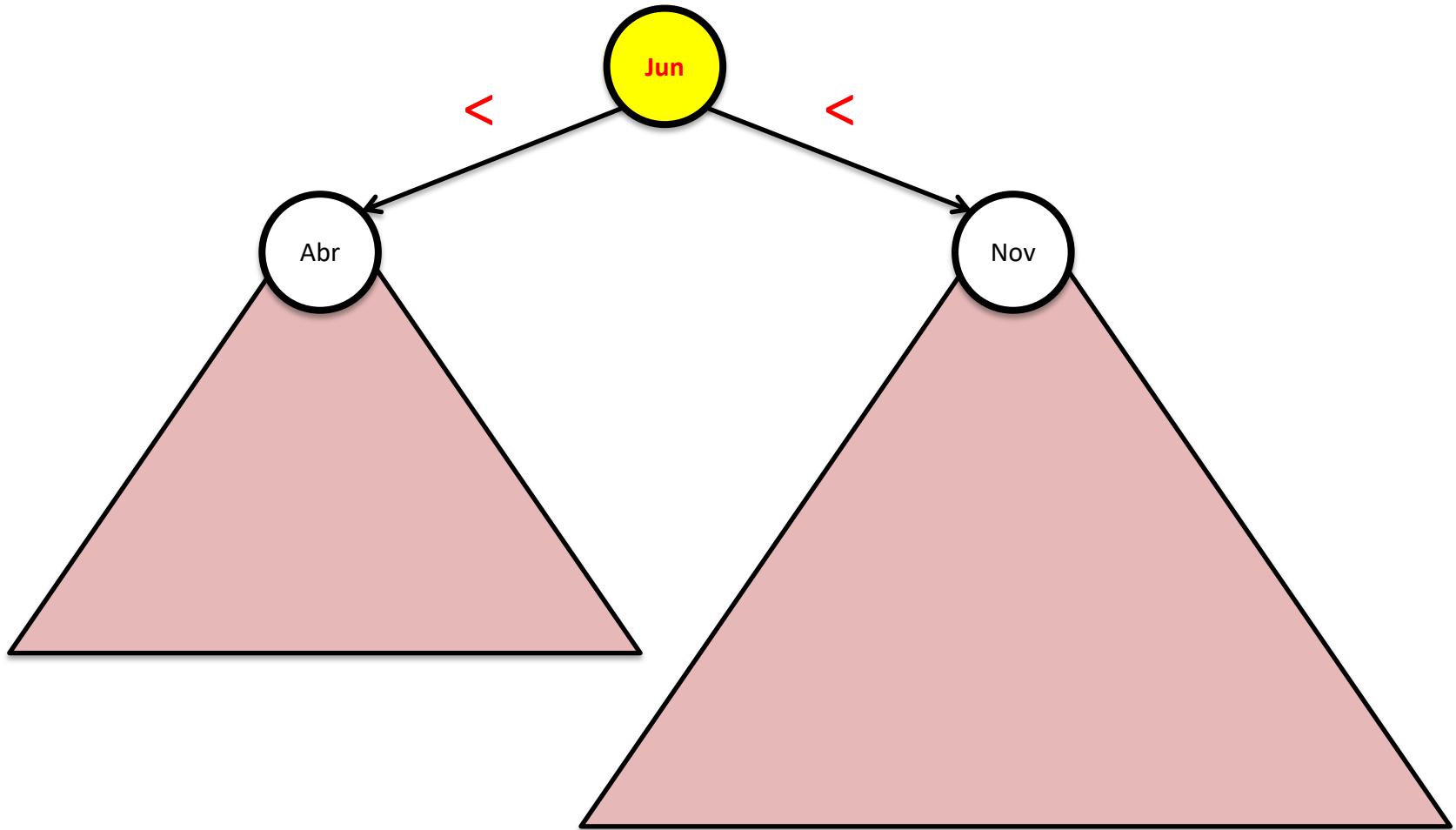
typedef struct SNo {
    TItem Item;
    TArvBin Esq, Dir;
} TNo;

typedef TArvBin TABB;
```

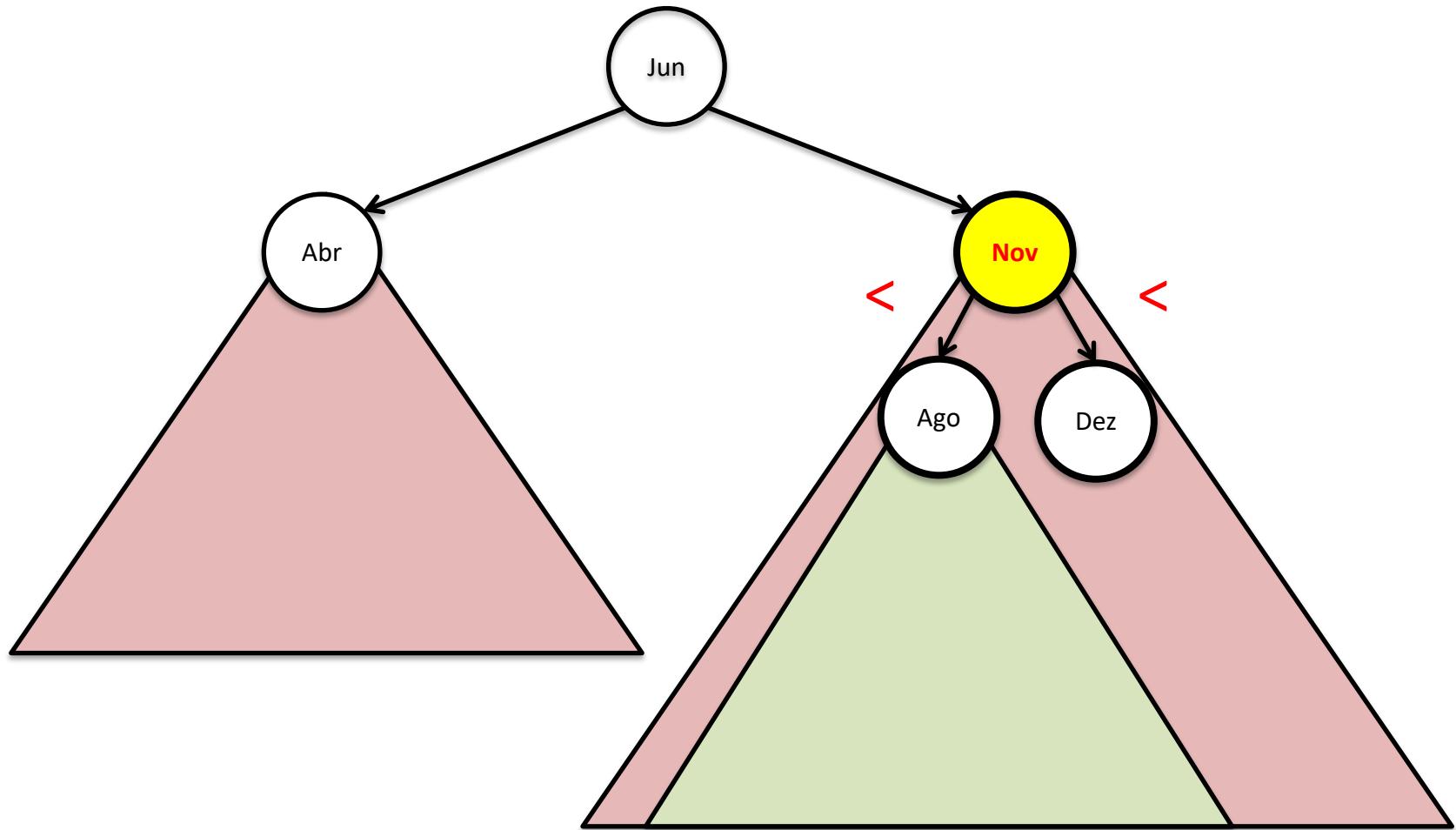
Árvores Binárias de Pesquisa



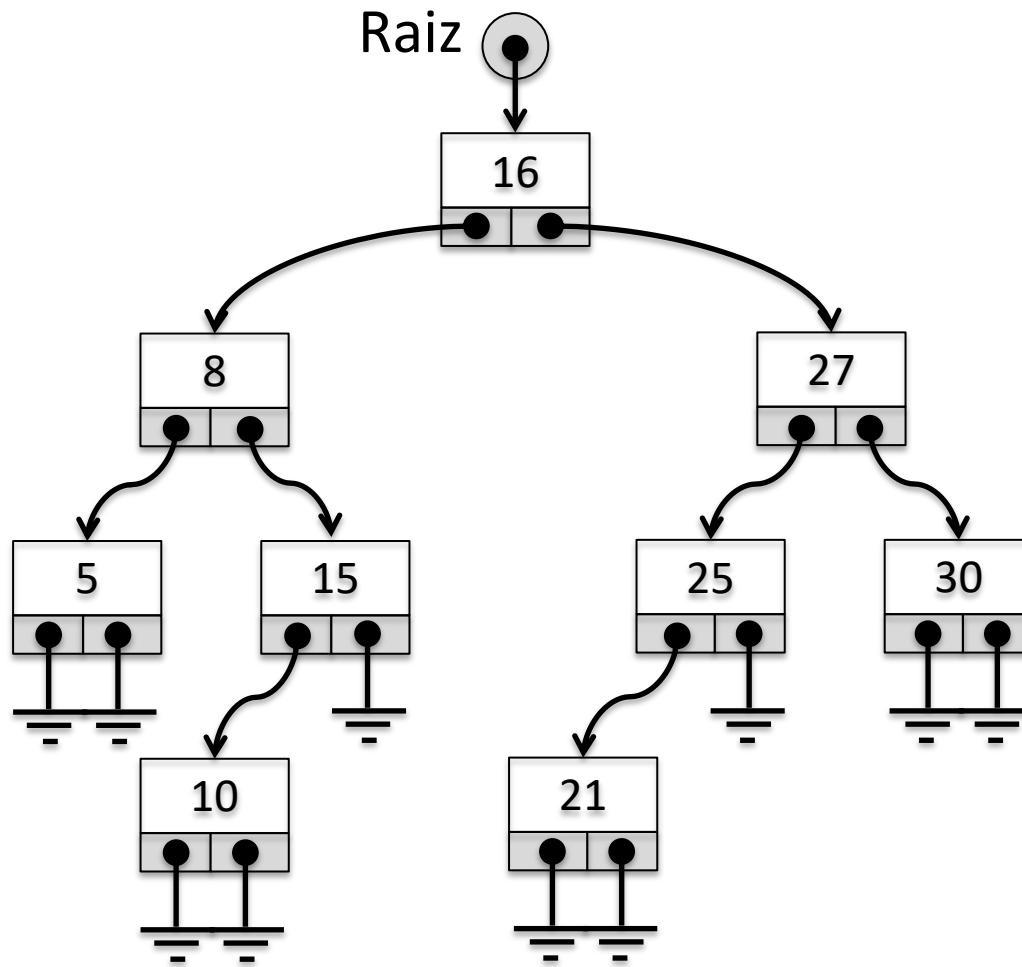
Árvores Binárias de Pesquisa



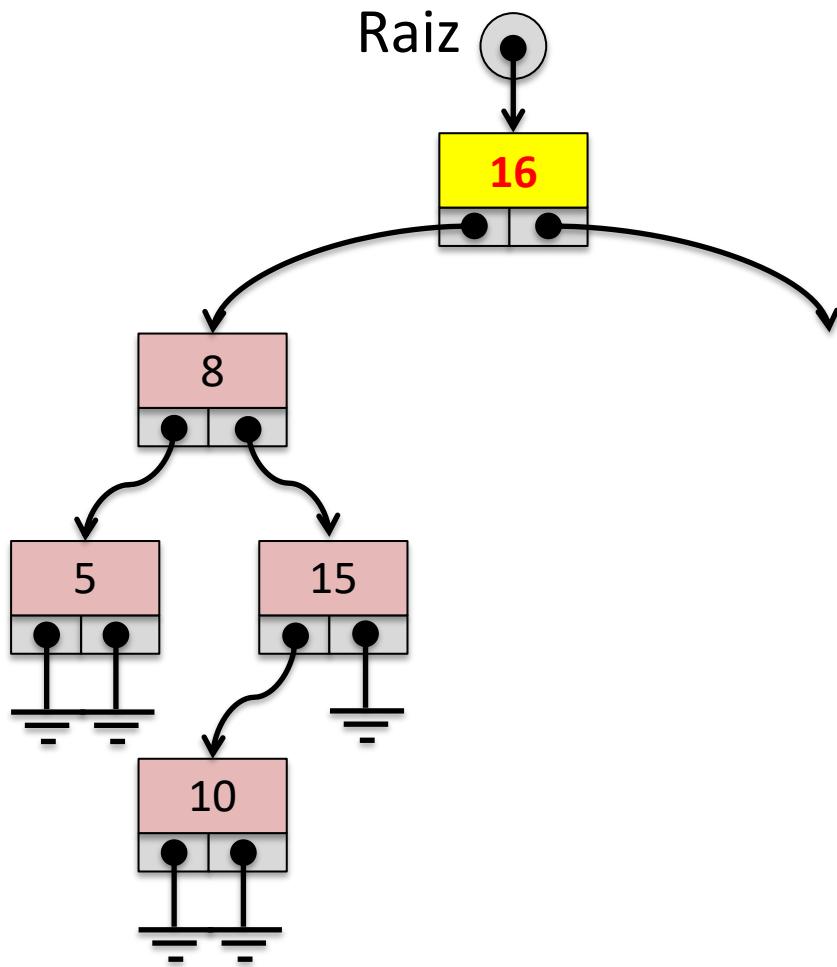
Árvores Binárias de Pesquisa



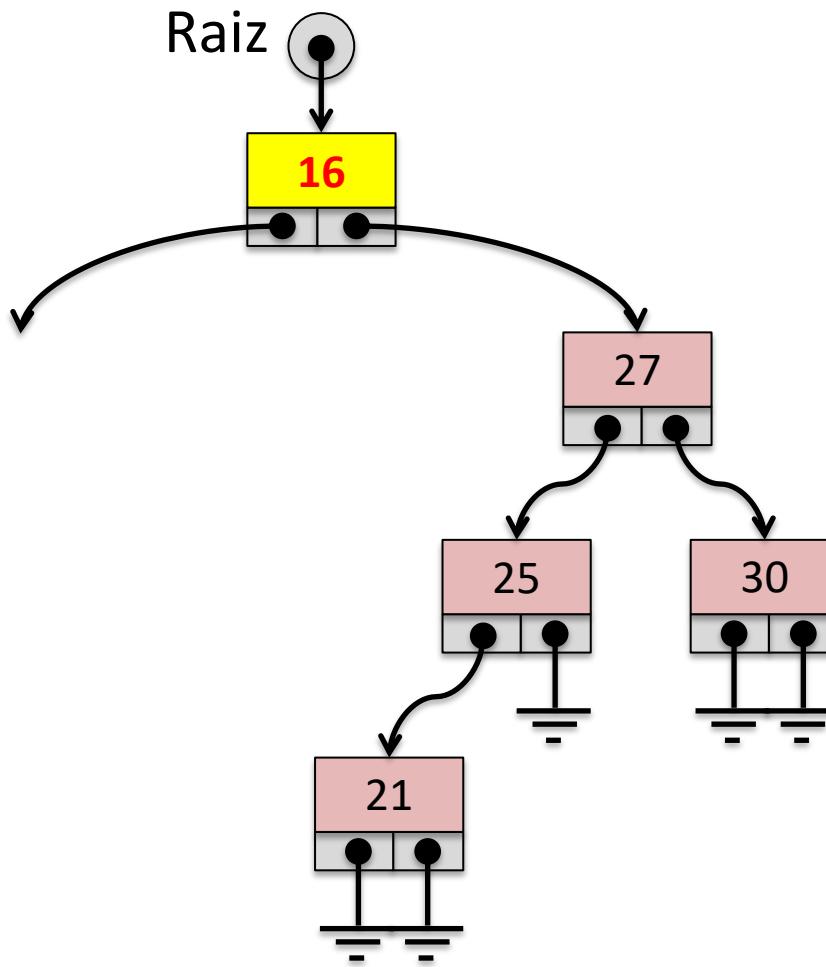
Árvores Binárias de Pesquisa



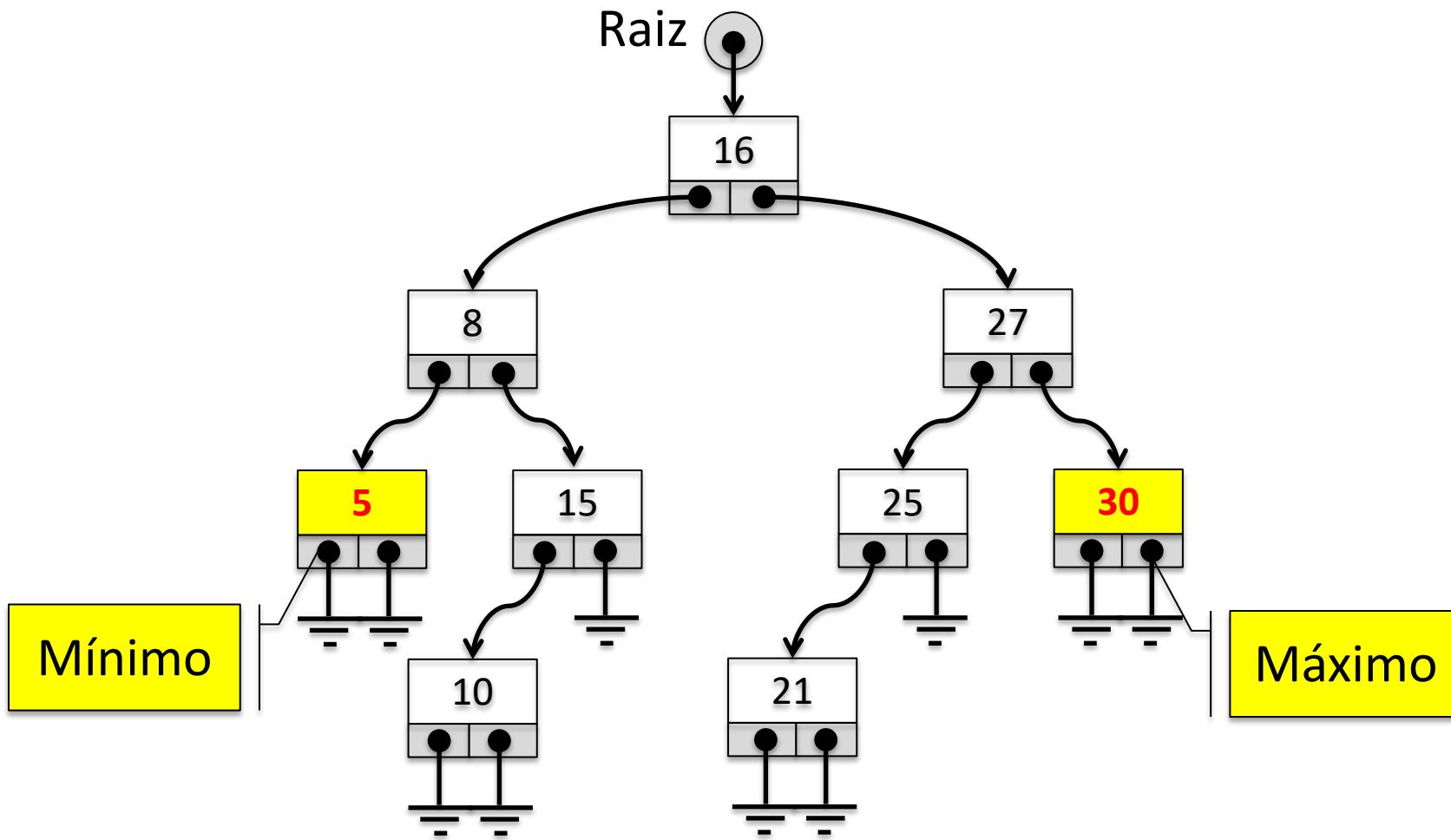
Árvores Binárias de Pesquisa



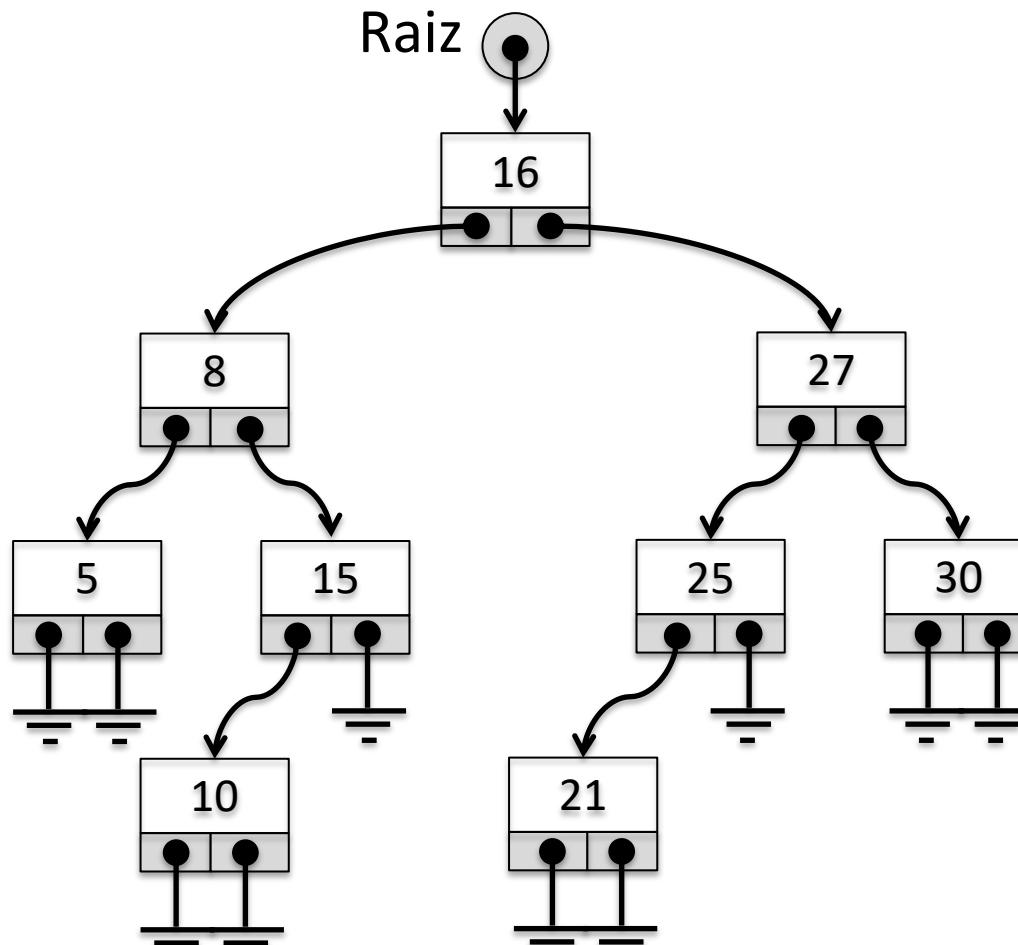
Árvores Binárias de Pesquisa



Árvores Binárias de Pesquisa

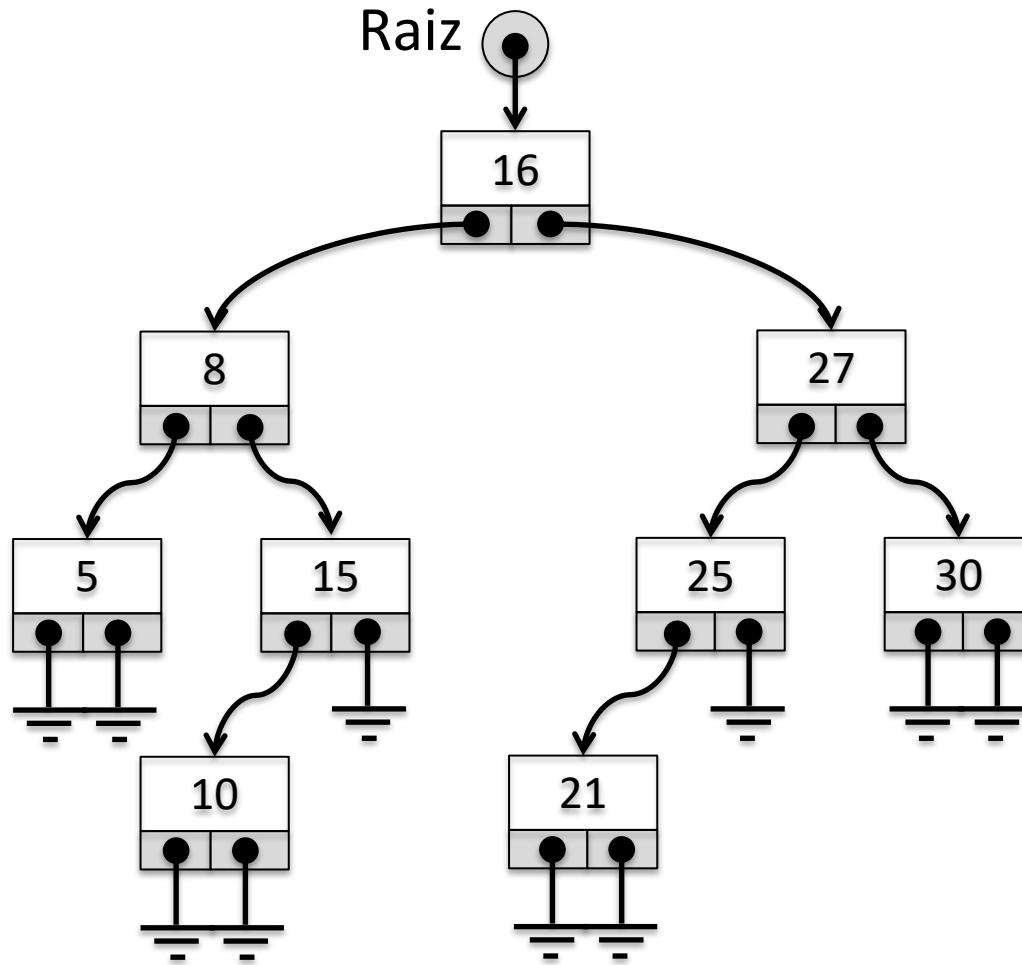


Árvores Binárias de Pesquisa



O que acontece se efetuarmos um **percurso em-ordem**?

Árvores Binárias de Pesquisa

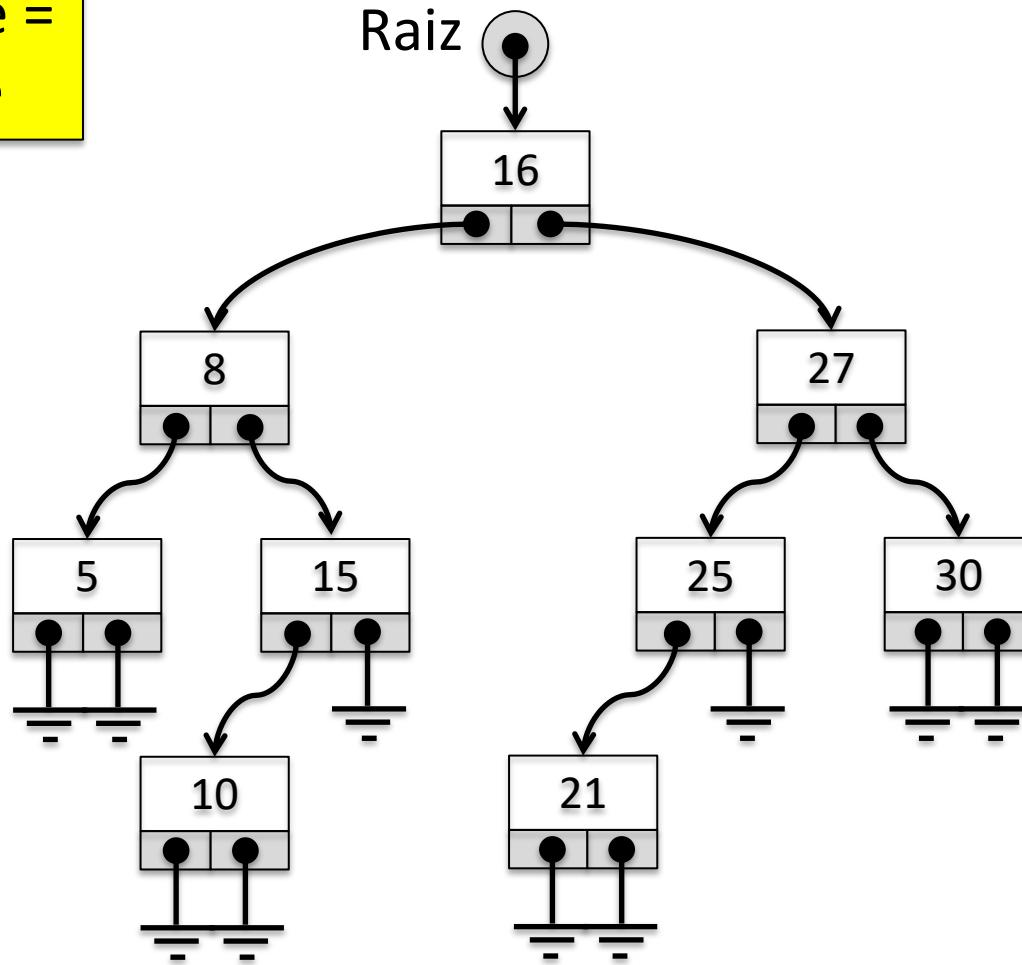


Percorso em-ordem: **5 8 10 15 16 21 25 27 30**

- Para encontrar um registro com uma chave **X**:
 - Compare-a com a chave que está na raiz
 - Se **X** é menor, vá para a subárvore esquerda
 - Se **X** é maior, vá para a subárvore direita
 - Repita o processo recursivamente, até que a chave procurada seja encontrada ou um nó folha é atingido
 - Se a pesquisa tiver sucesso, então o nó que contém a chave é retornado

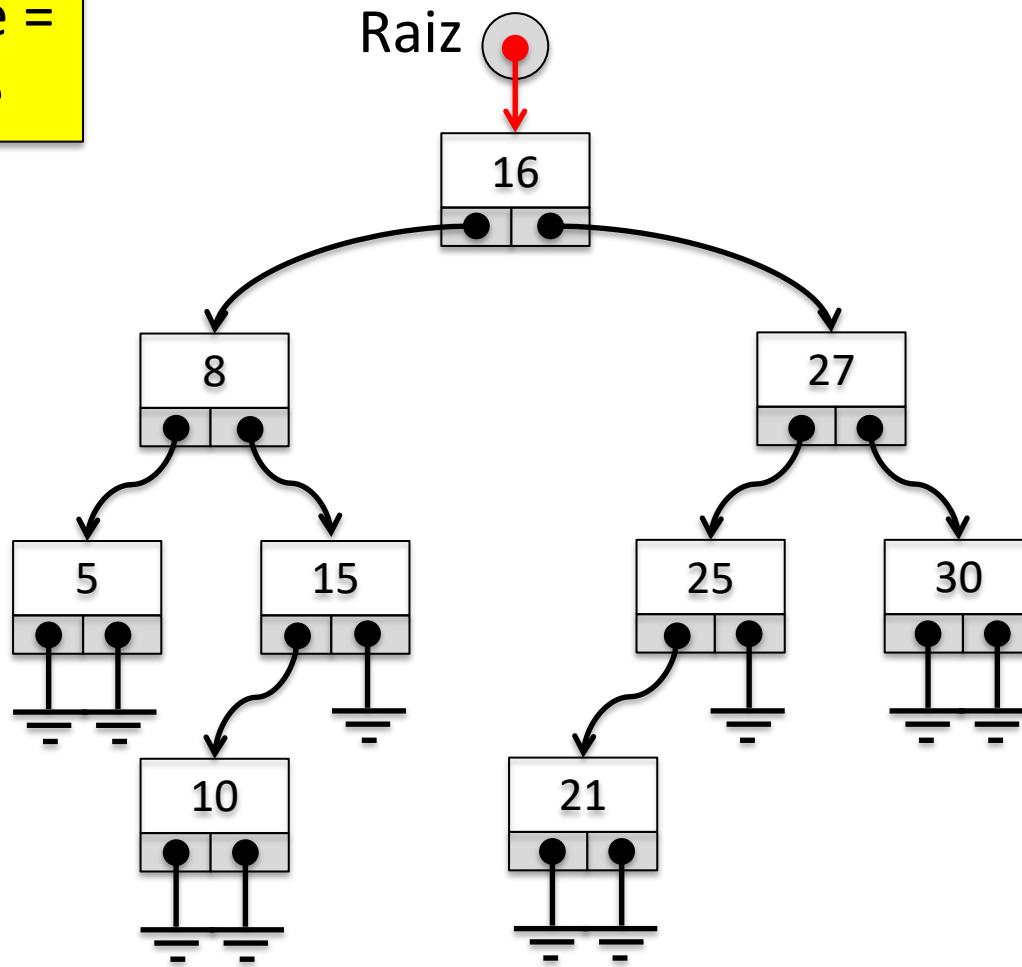
Árvores Binárias de Pesquisa

buscar a chave =
25 na árvore



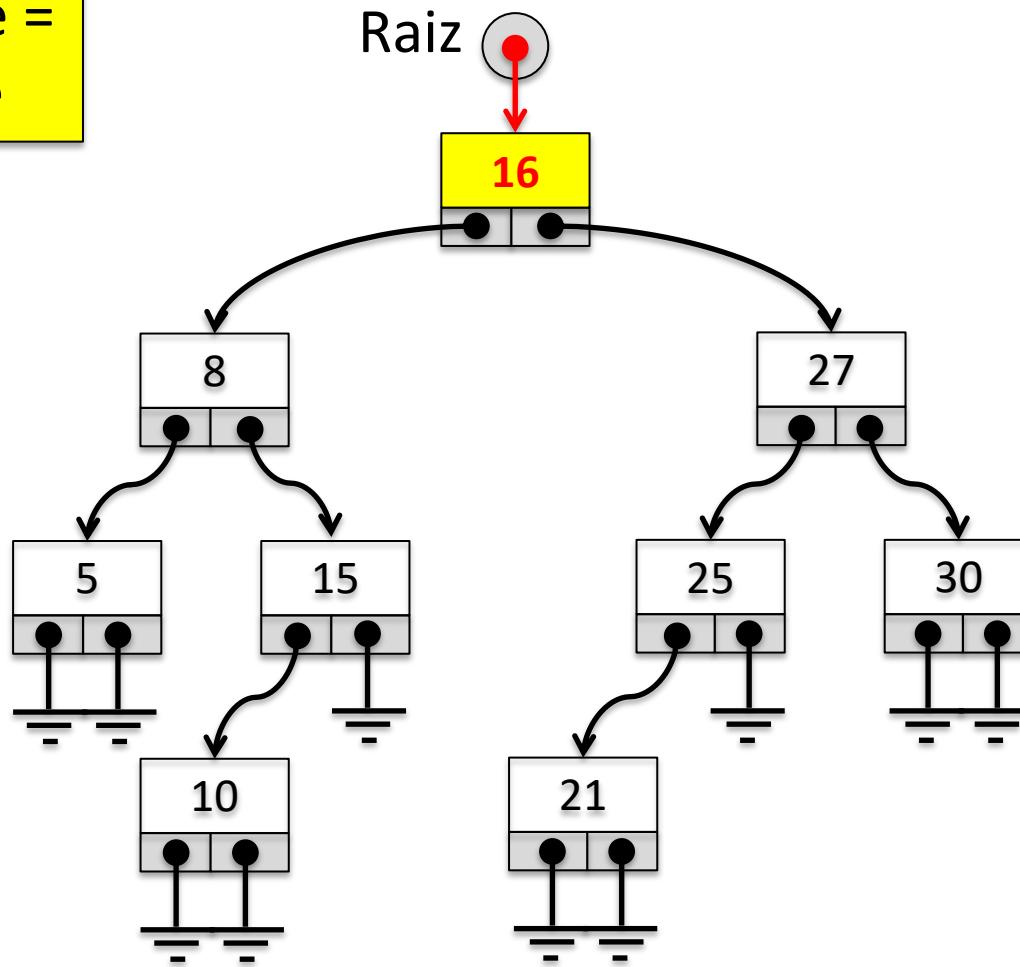
Árvores Binárias de Pesquisa

buscar a chave =
25 na árvore



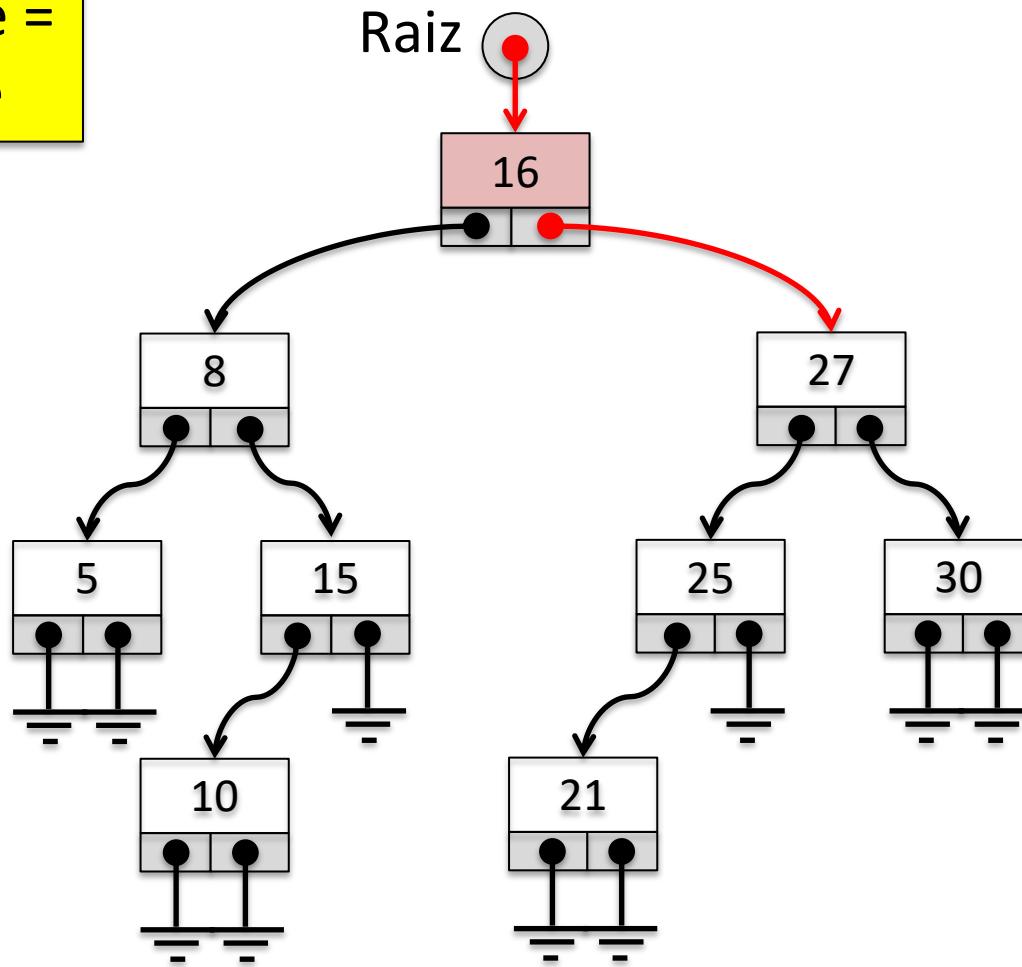
Árvores Binárias de Pesquisa

buscar a chave =
25 na árvore



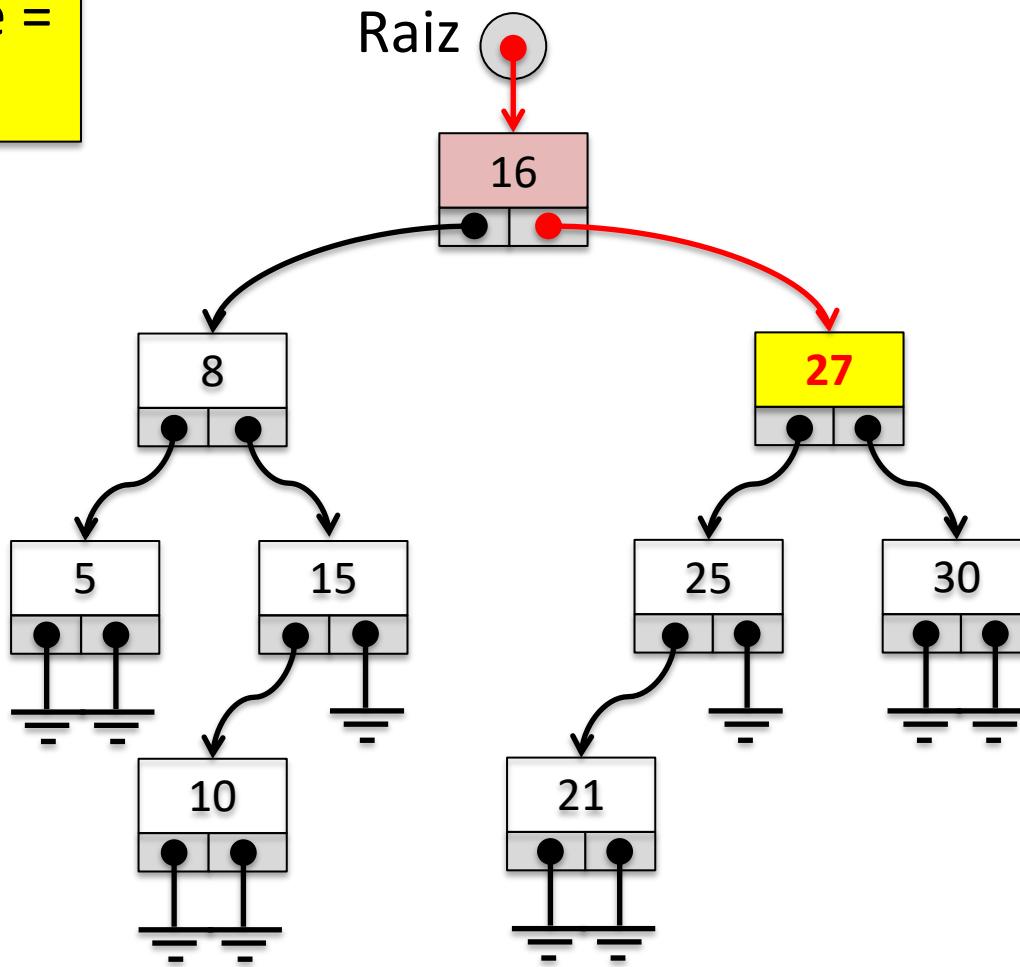
Árvores Binárias de Pesquisa

buscar a chave =
25 na árvore



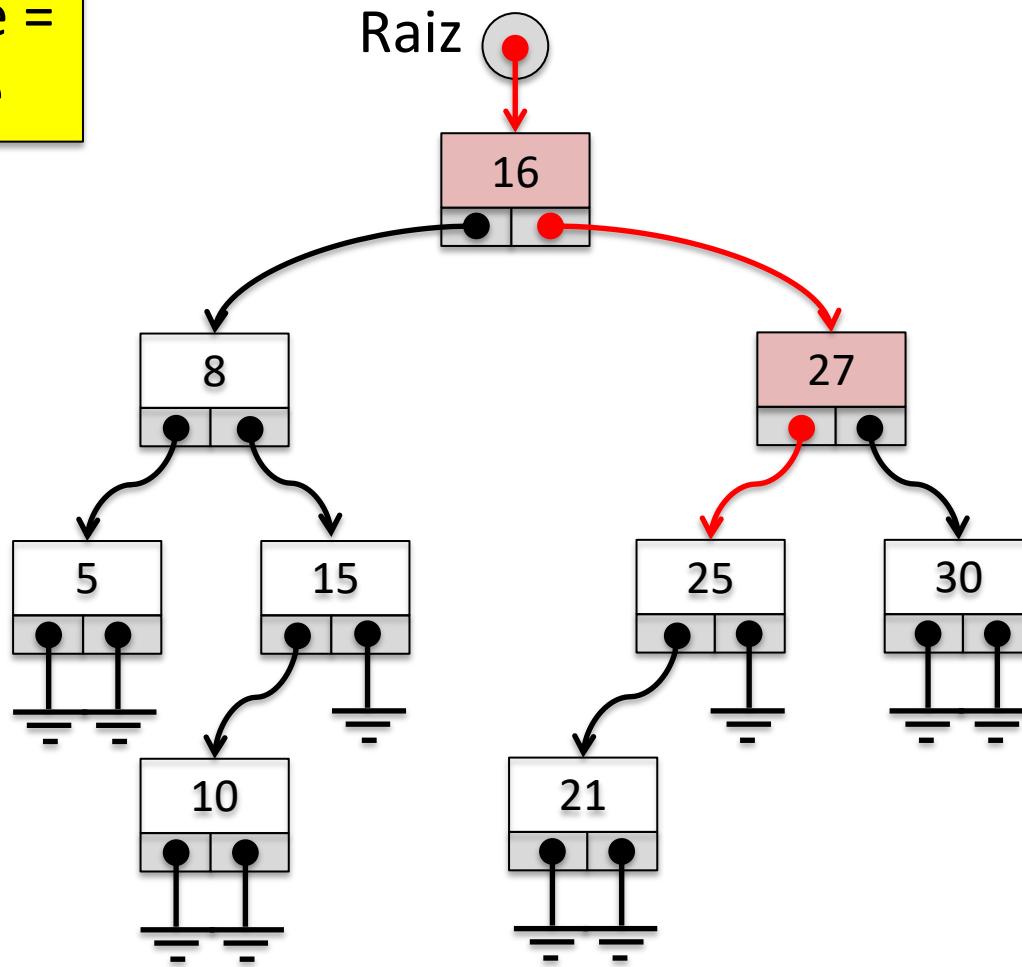
Árvores Binárias de Pesquisa

buscar a chave =
25 na árvore



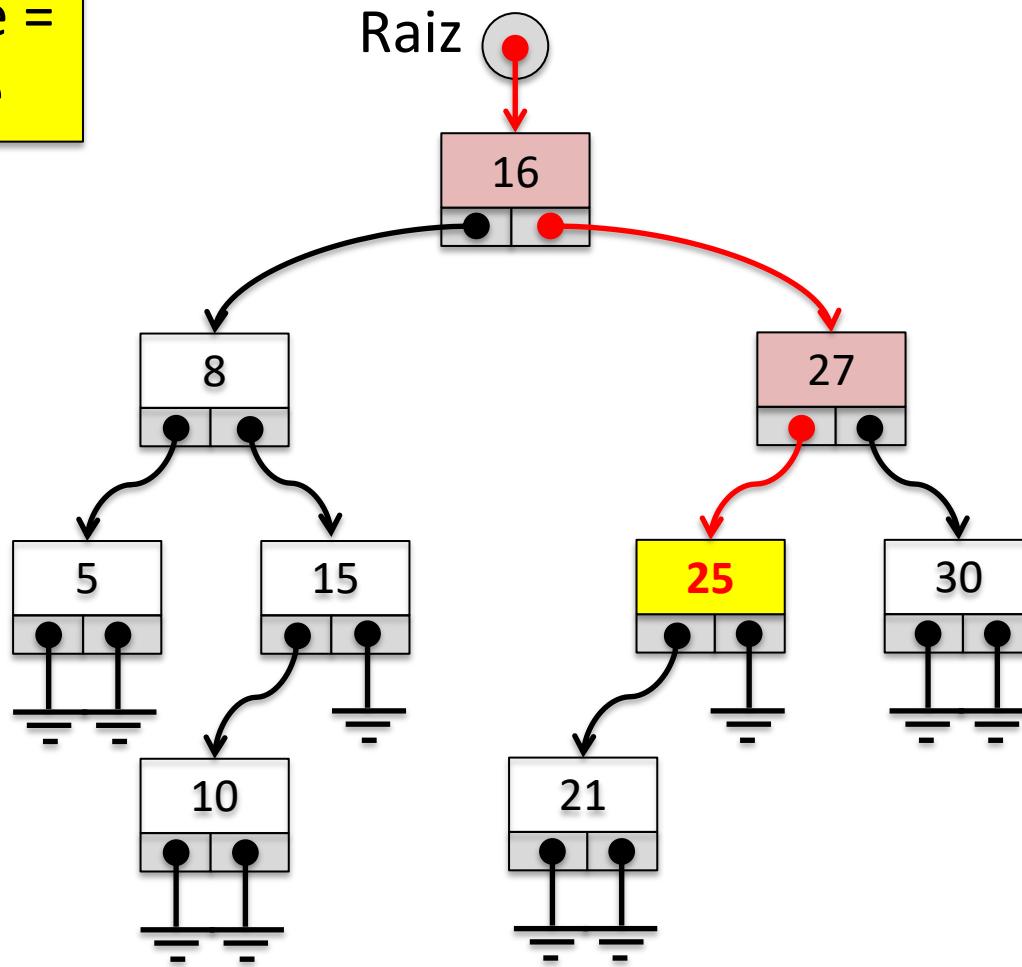
Árvores Binárias de Pesquisa

buscar a chave =
25 na árvore



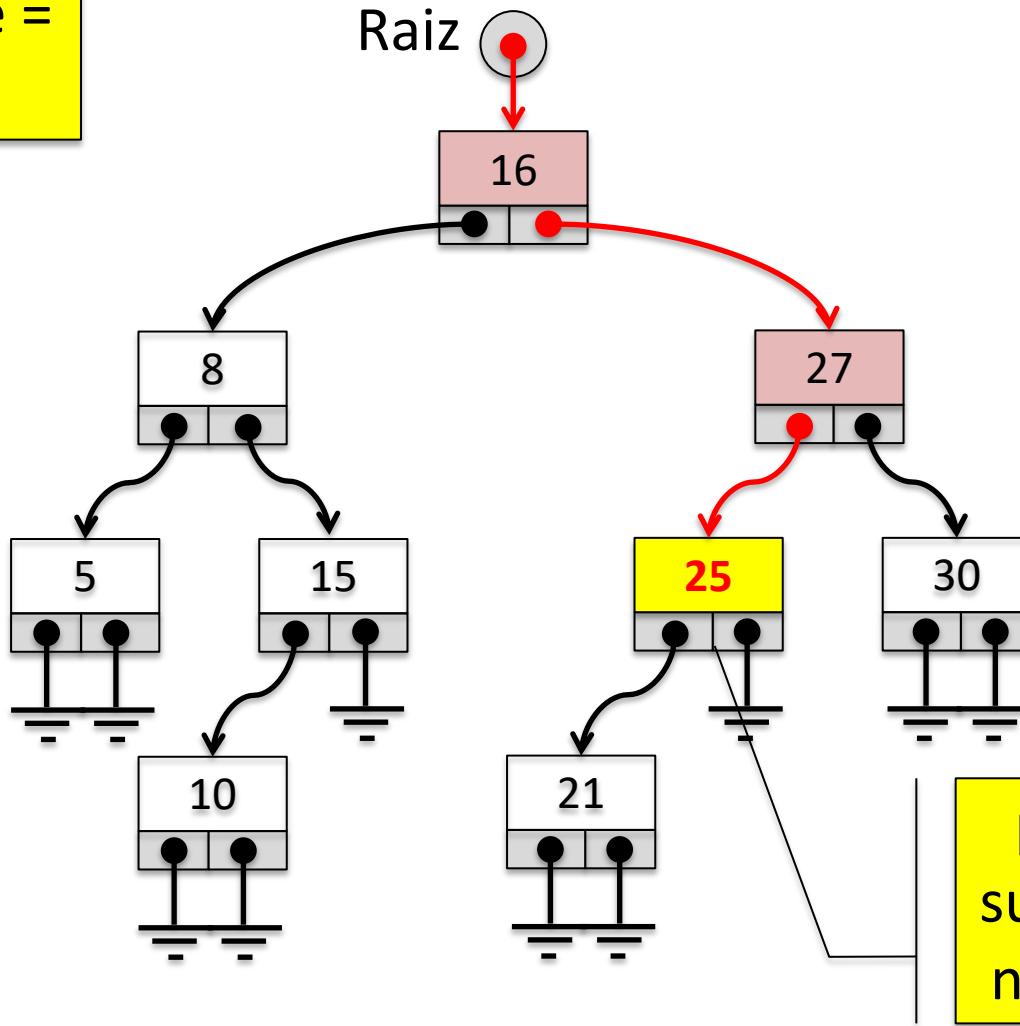
Árvores Binárias de Pesquisa

buscar a chave =
25 na árvore



Árvores Binárias de Pesquisa

buscar a chave =
25 na árvore



Árvores Binárias de Pesquisa

```
TABB TABB_Pesquisa(TABB No, TChave x)
{
    if (No == NULL)
        return NULL; // retorna NULL caso a chave nao seja encontrada
    else if (x < No->Item.Chave)
        return TABB_Pesquisa(No->Esq, x);
    else if (x > No->Item.Chave)
        return TABB_Pesquisa(No->Dir, x);
    else
        return No;
}
```

Árvores Binárias de Pesquisa

```
TABB TABB_Pesquisa(TABB Raiz, TChave x)
{
    TABB No;

    No = Raiz;
    while ((No != NULL) && (x != No->Item.Chave)) {
        if (x < No->Item.Chave)
            No = No->Esq;
        else if (x > No->Item.Chave)
            No = No->Dir;
    }

    return No;
}
```

■ Onde inserir?

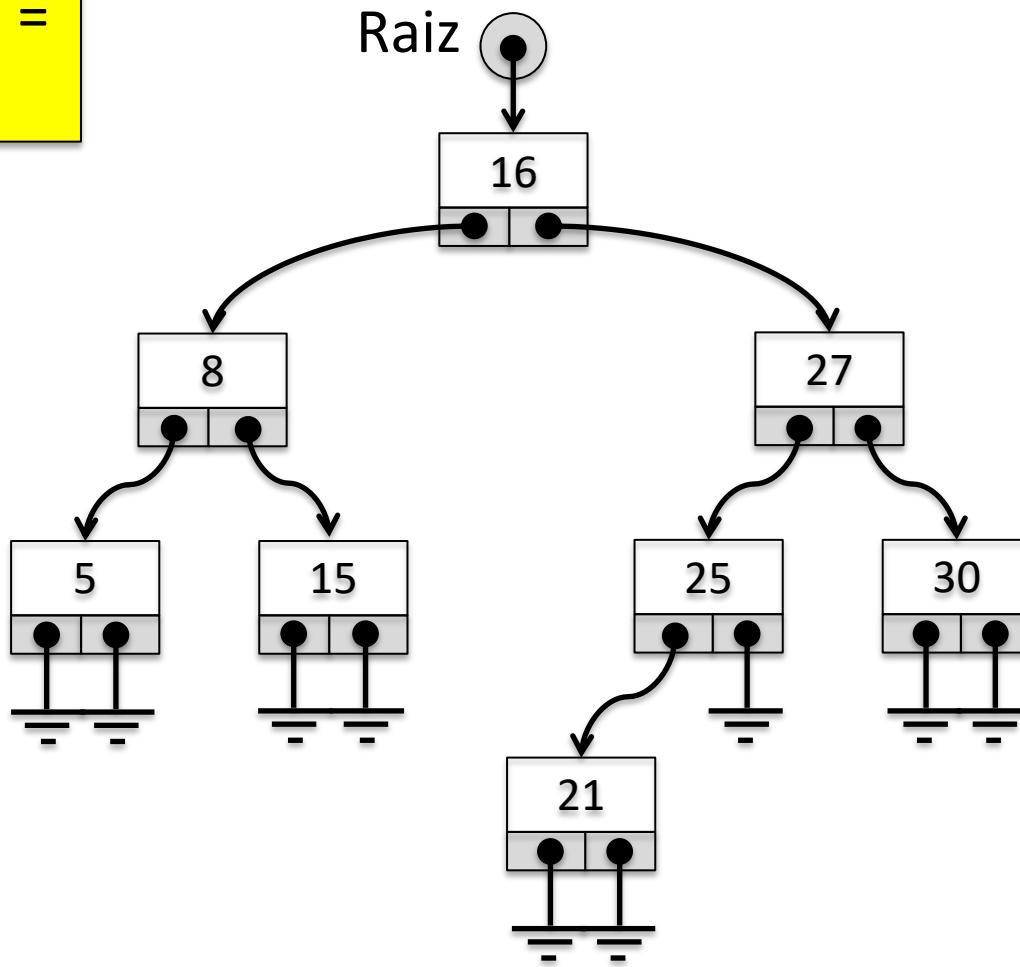
- Atingir um ponteiro nulo em um processo de pesquisa significa uma pesquisa sem sucesso
- O ponteiro nulo atingido é o ponto de inserção

■ Como inserir?

1. Cria nó contendo o registro
2. Procura o lugar de inserção na árvore
3. Se o registro não estiver na árvore, insere-o

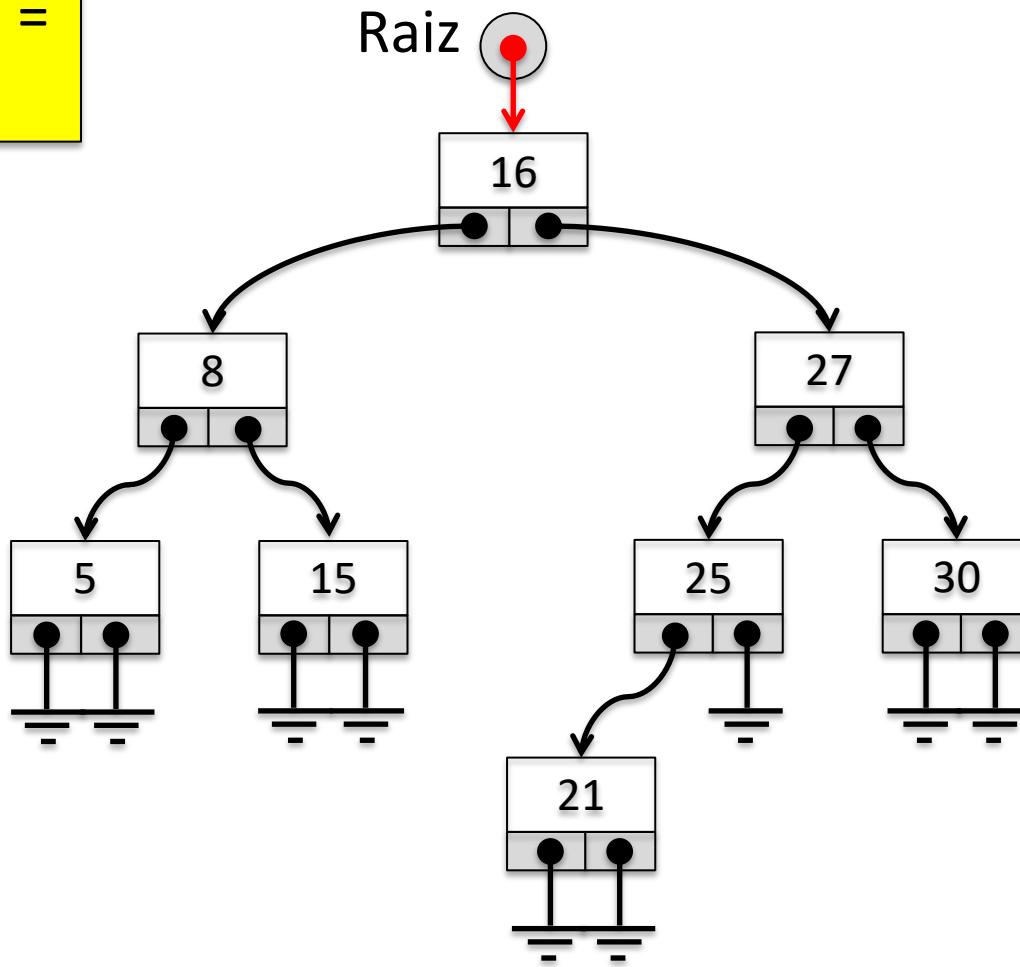
Árvores Binárias de Pesquisa

inserir a chave =
10 na árvore



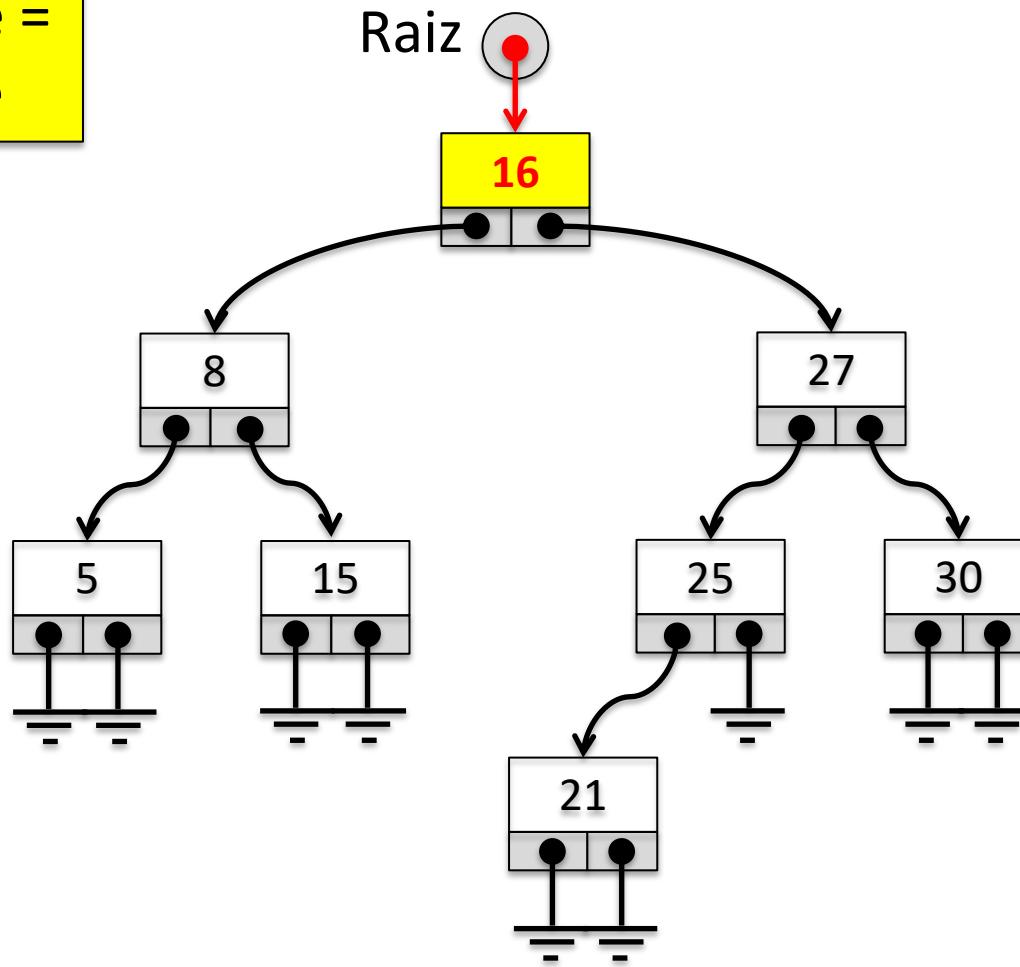
Árvores Binárias de Pesquisa

inserir a chave =
10 na árvore



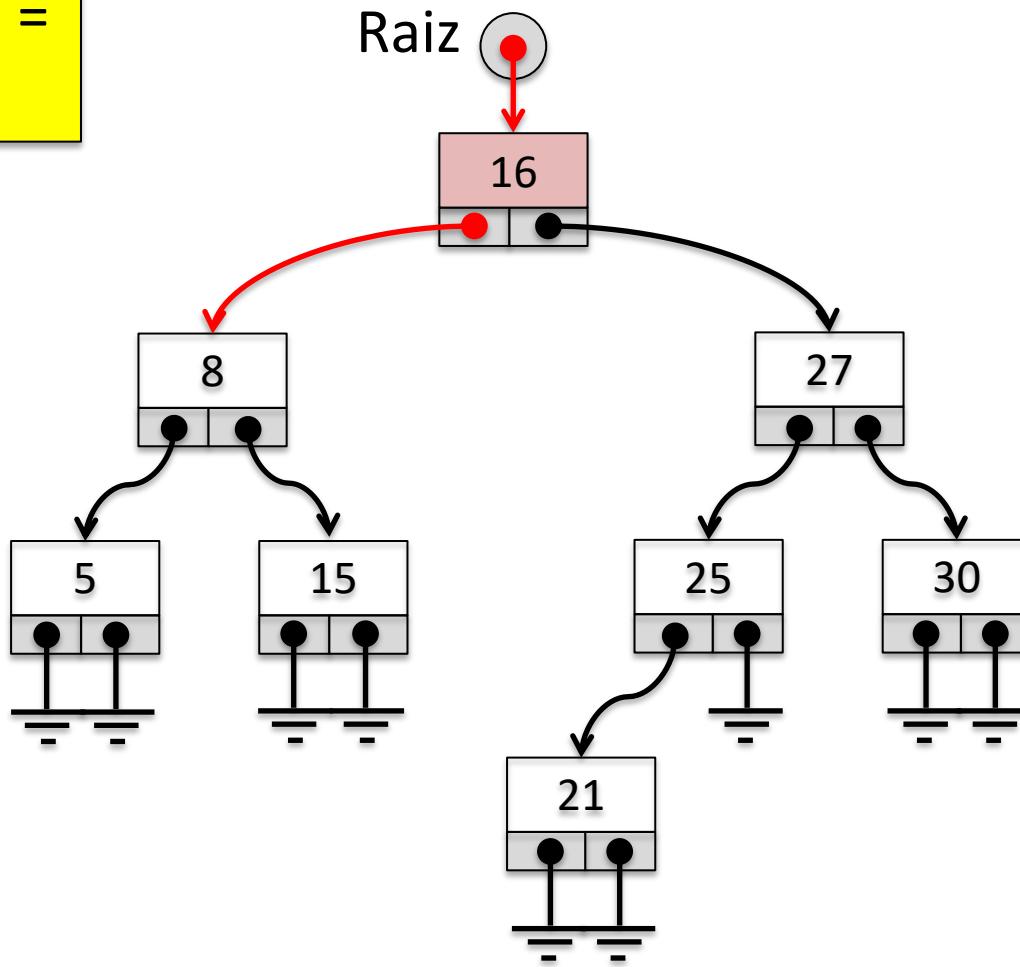
Árvores Binárias de Pesquisa

inserir a chave =
10 na árvore



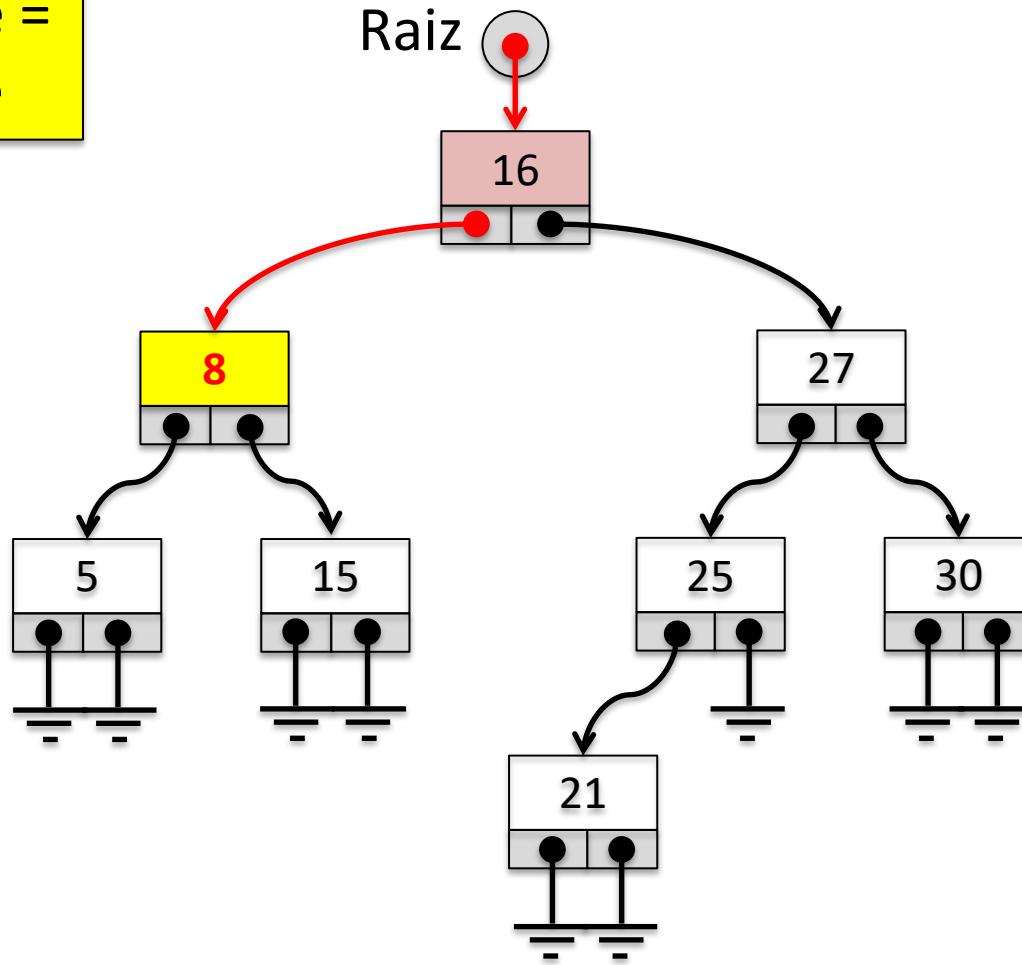
Árvores Binárias de Pesquisa

inserir a chave =
10 na árvore



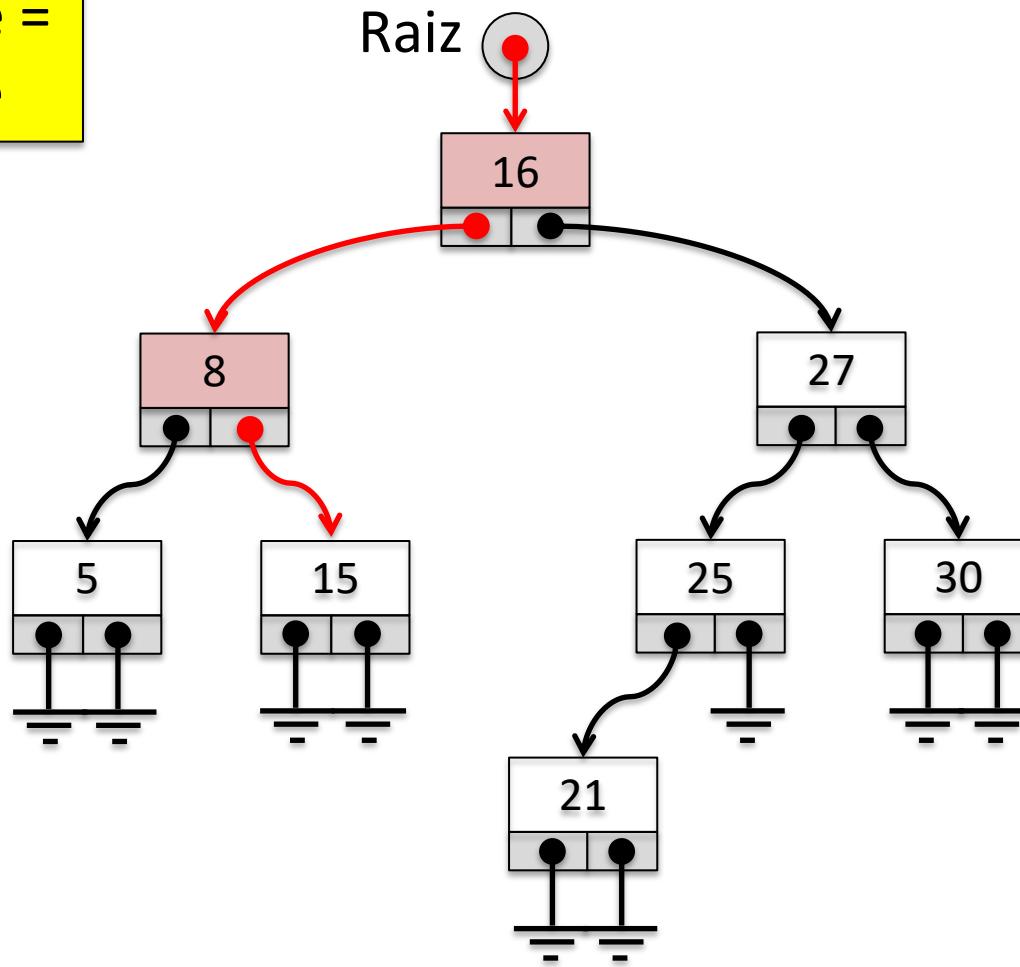
Árvores Binárias de Pesquisa

inserir a chave =
10 na árvore



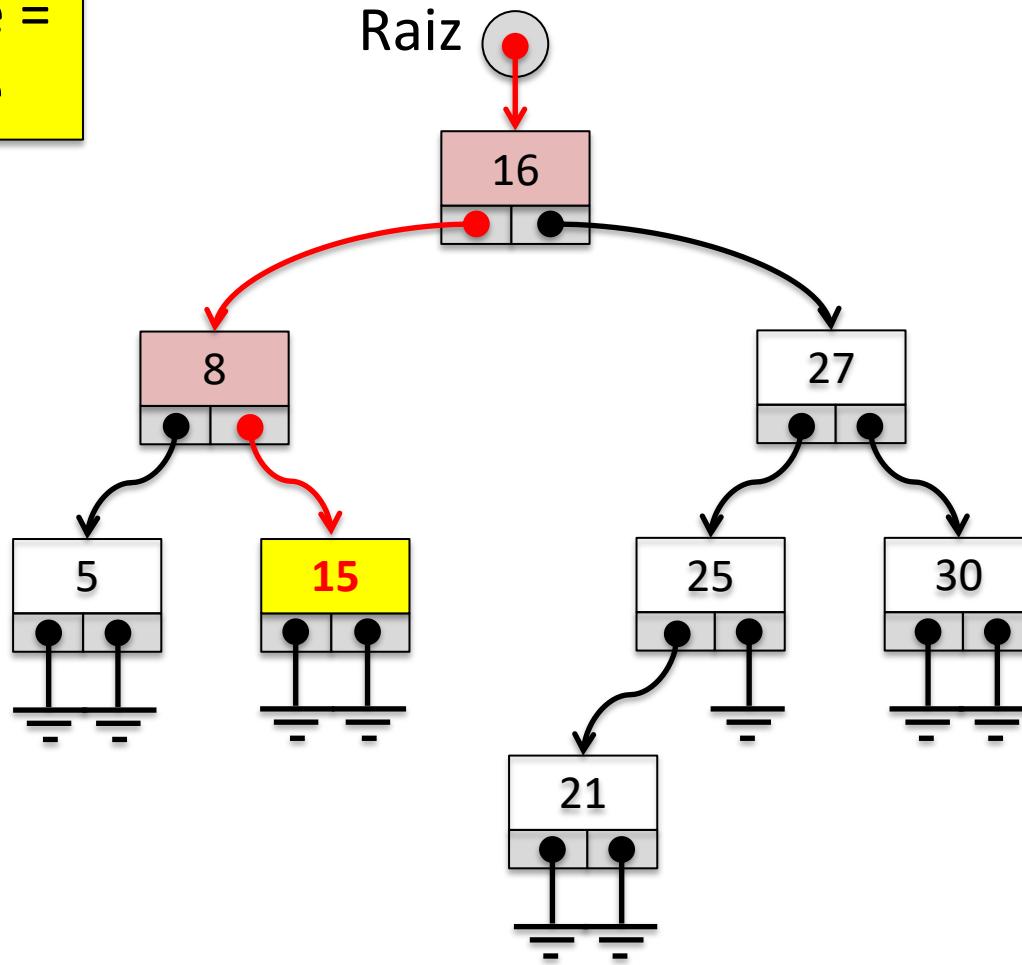
Árvores Binárias de Pesquisa

inserir a chave =
10 na árvore



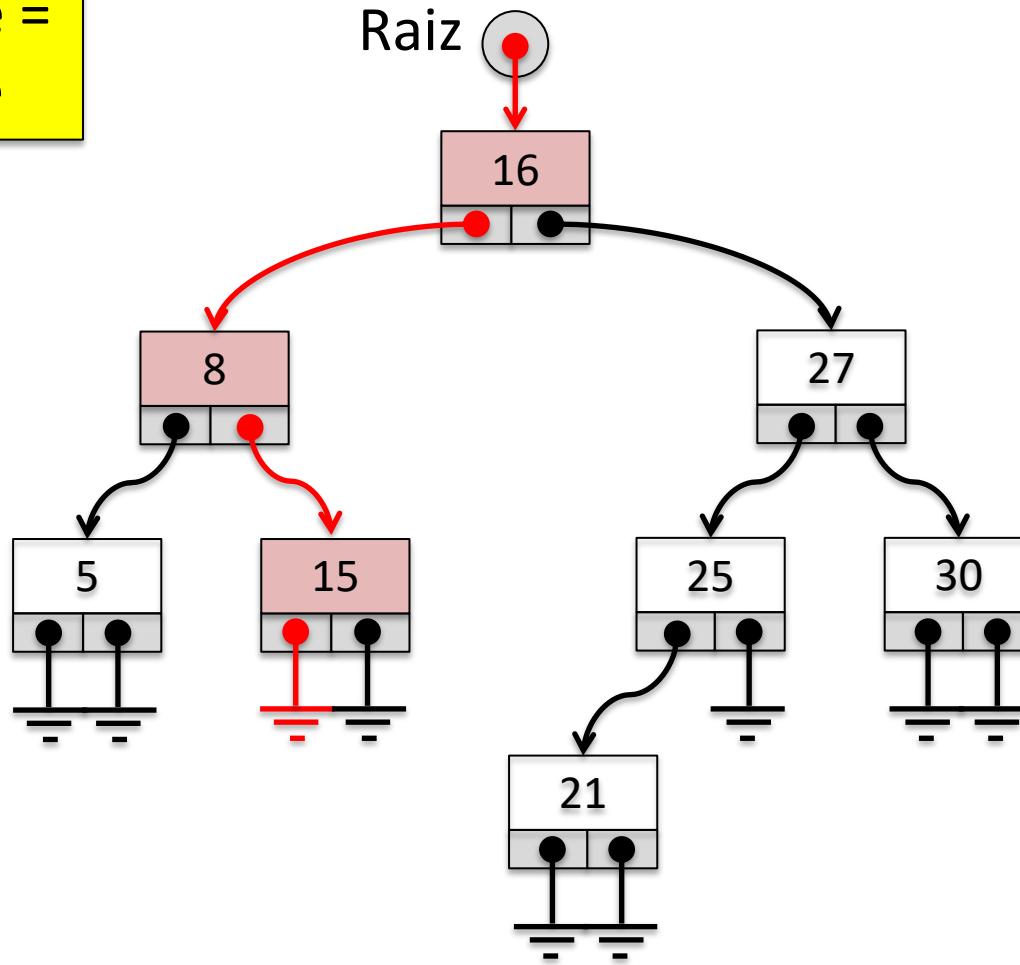
Árvores Binárias de Pesquisa

inserir a chave =
10 na árvore



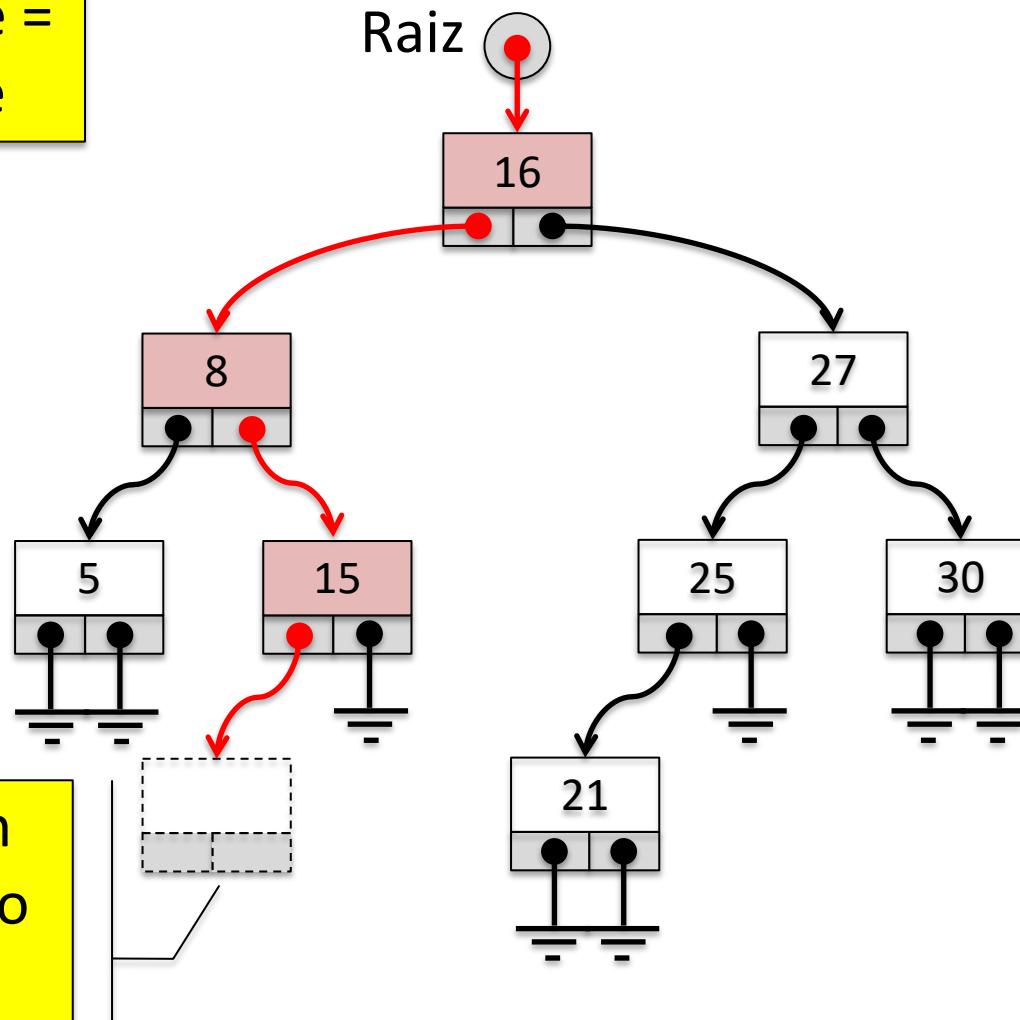
Árvores Binárias de Pesquisa

inserir a chave =
10 na árvore



Árvores Binárias de Pesquisa

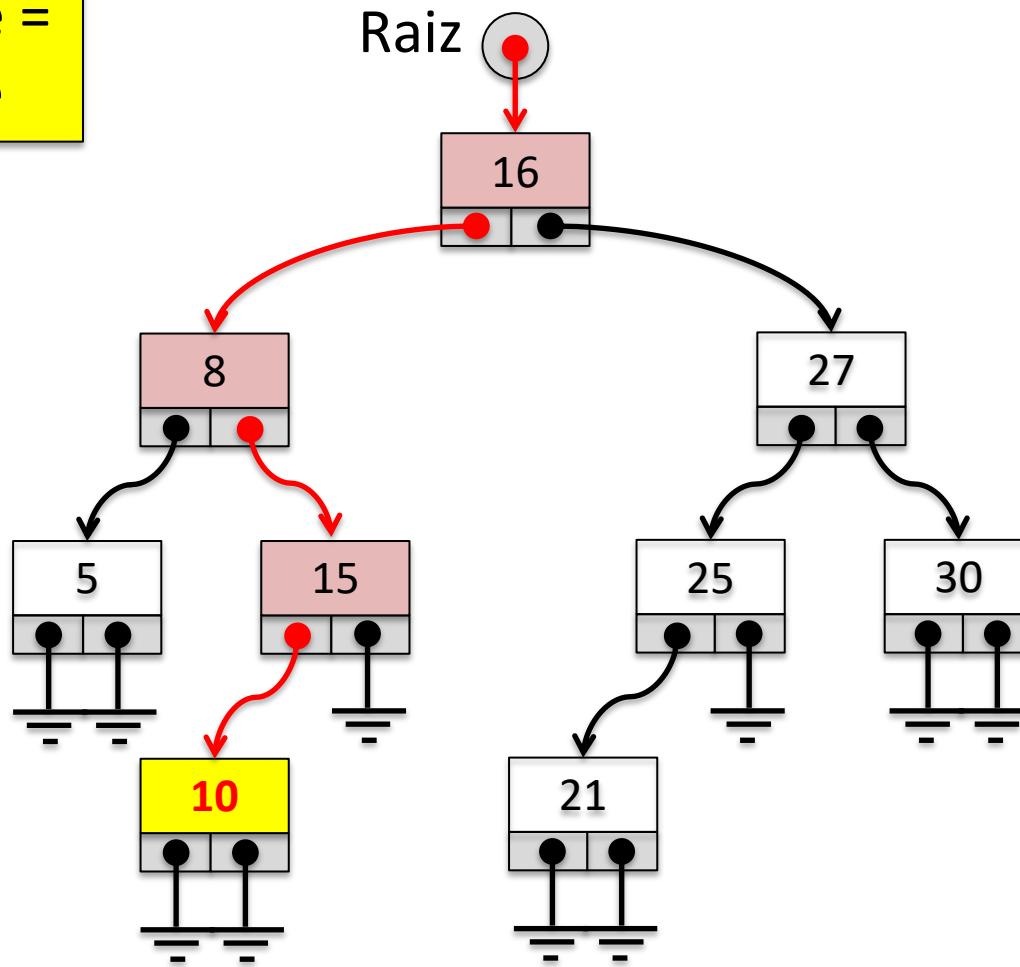
inserir a chave =
10 na árvore



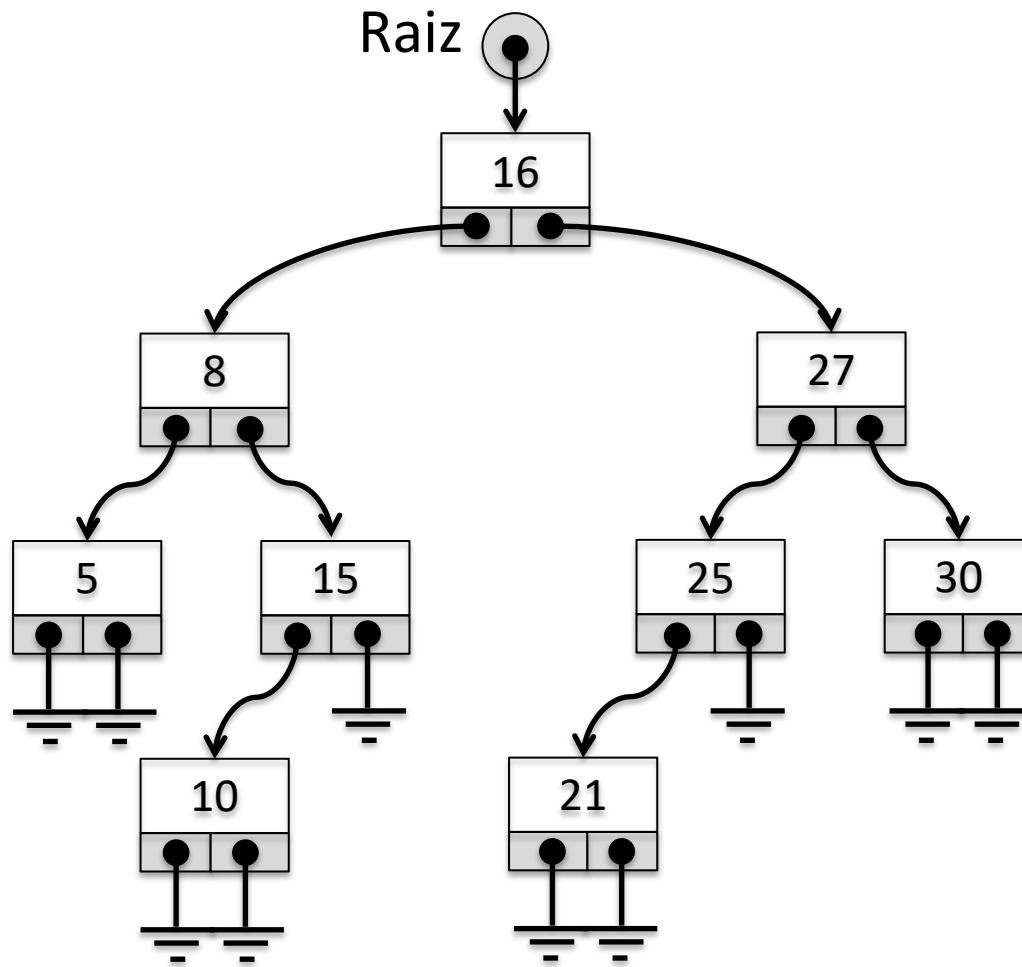
pesquisa sem
sucesso, ponto
de inserção

Árvores Binárias de Pesquisa

inserir a chave =
10 na árvore



Árvores Binárias de Pesquisa



Árvores Binárias de Pesquisa

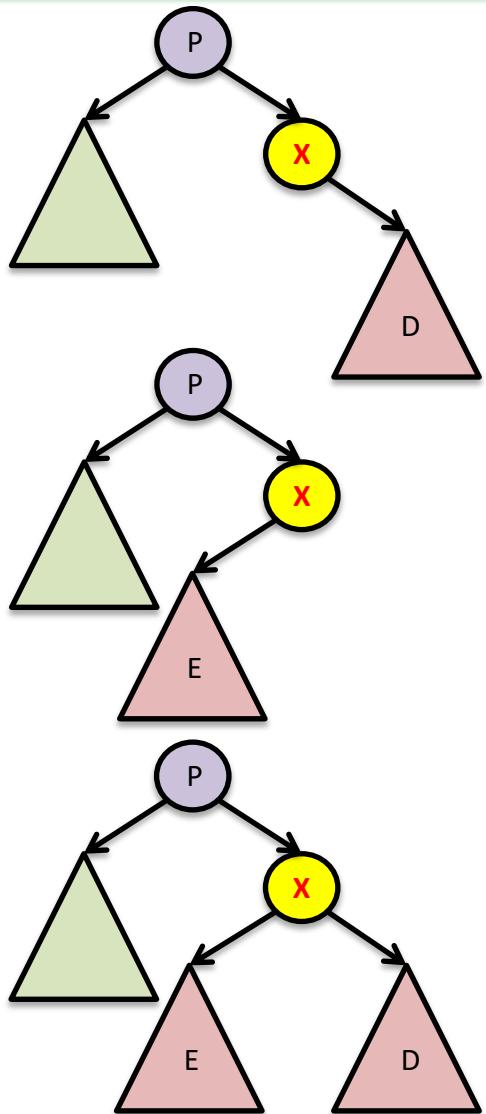
```
int TABB_Insere(TABB *pNo, TItem x)
{
    if (*pNo == NULL) {
        *pNo = TArvBin_CriaNo(x, TArvBin_Inicia(), TArvBin_Inicia());
        return 1;
    }
    else if (x.Chave < (*pNo)->Item.Chave)
        return TABB_Insere(&(*pNo)->Esq, x);
    else if (x.Chave > (*pNo)->Item.Chave)
        return TABB_Insere(&(*pNo)->Dir, x);
    else
        return 0; // retorna 0 caso o item ja estiver na arvore
}
```

Árvores Binárias de Pesquisa

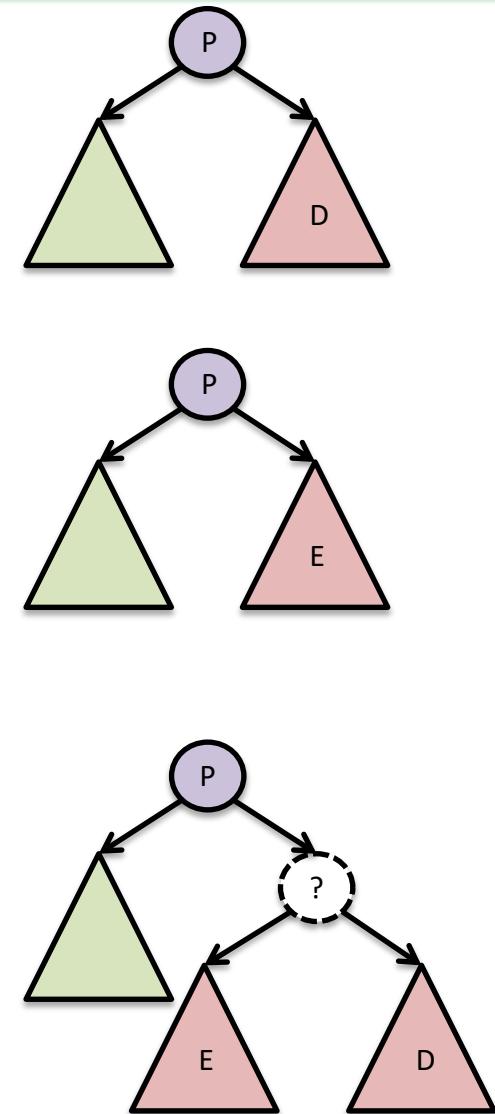
```
int TABB_Insere(TABB *pRaiz, TItem x)
{
    TABB *pNo;
    pNo = pRaiz;
    while ((*pNo != NULL) && (x.Chave != (*pNo)->Item.Chave)) {
        if (x.Chave < (*pNo)->Item.Chave)
            pNo = &(*pNo)->Esq;
        else if (x.Chave > (*pNo)->Item.Chave)
            pNo = &(*pNo)->Dir;
    }
    if (*pNo == NULL) {
        *pNo = TArvBin_CriaNo(x, TArvBin_Inicia(), TArvBin_Inicia());
        return 1;
    }
    return 0; // retorna 0 caso o item ja estiver na arvore
}
```

- Para retirar um registro com uma chave **x** é preciso considerar três situações possíveis:
 - Não existe nenhum nó com chave igual a **x**
 - O nó com chave **x** possui, no máximo, um descendente
 - O nó com chave **x** possui dois descendentes

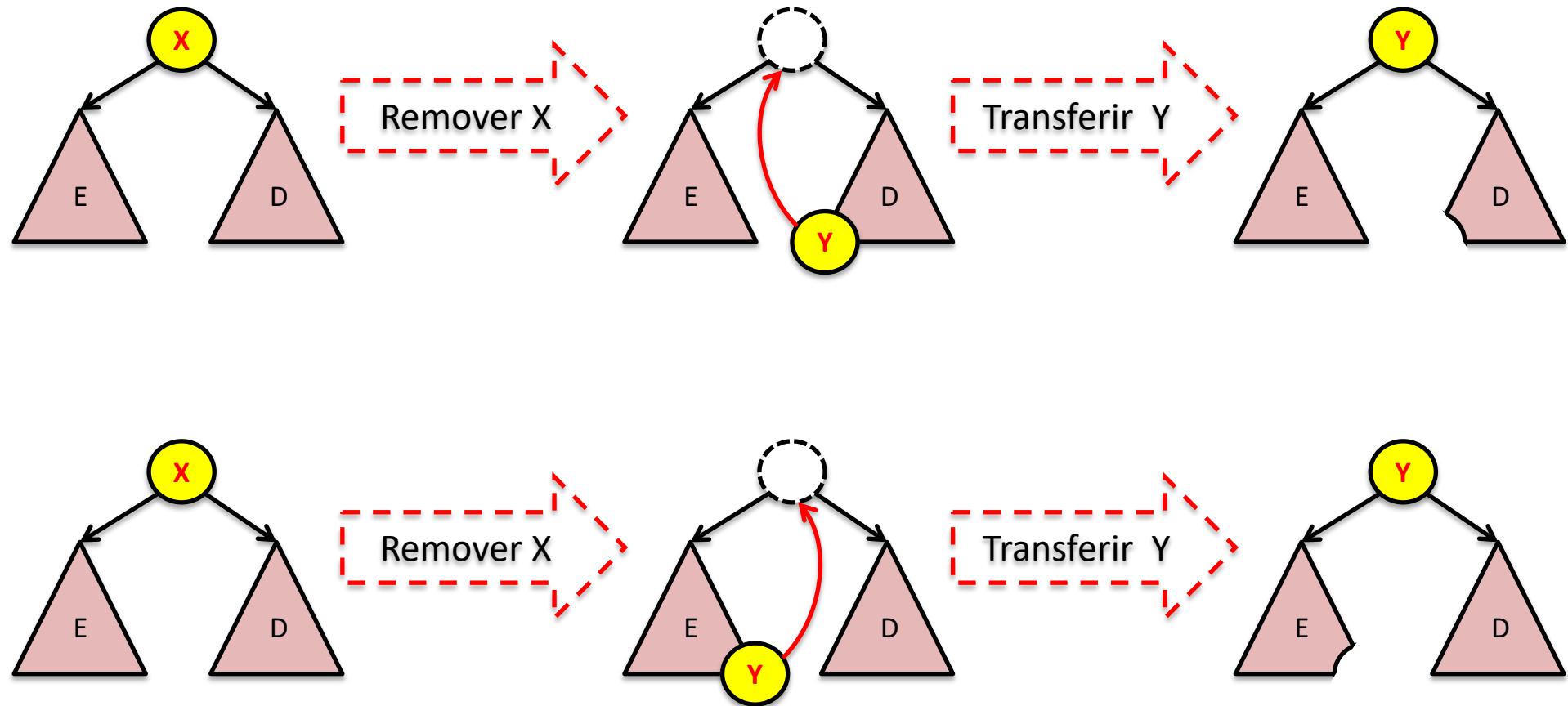
Árvores Binárias de Pesquisa



Remover X



Árvores Binárias de Pesquisa

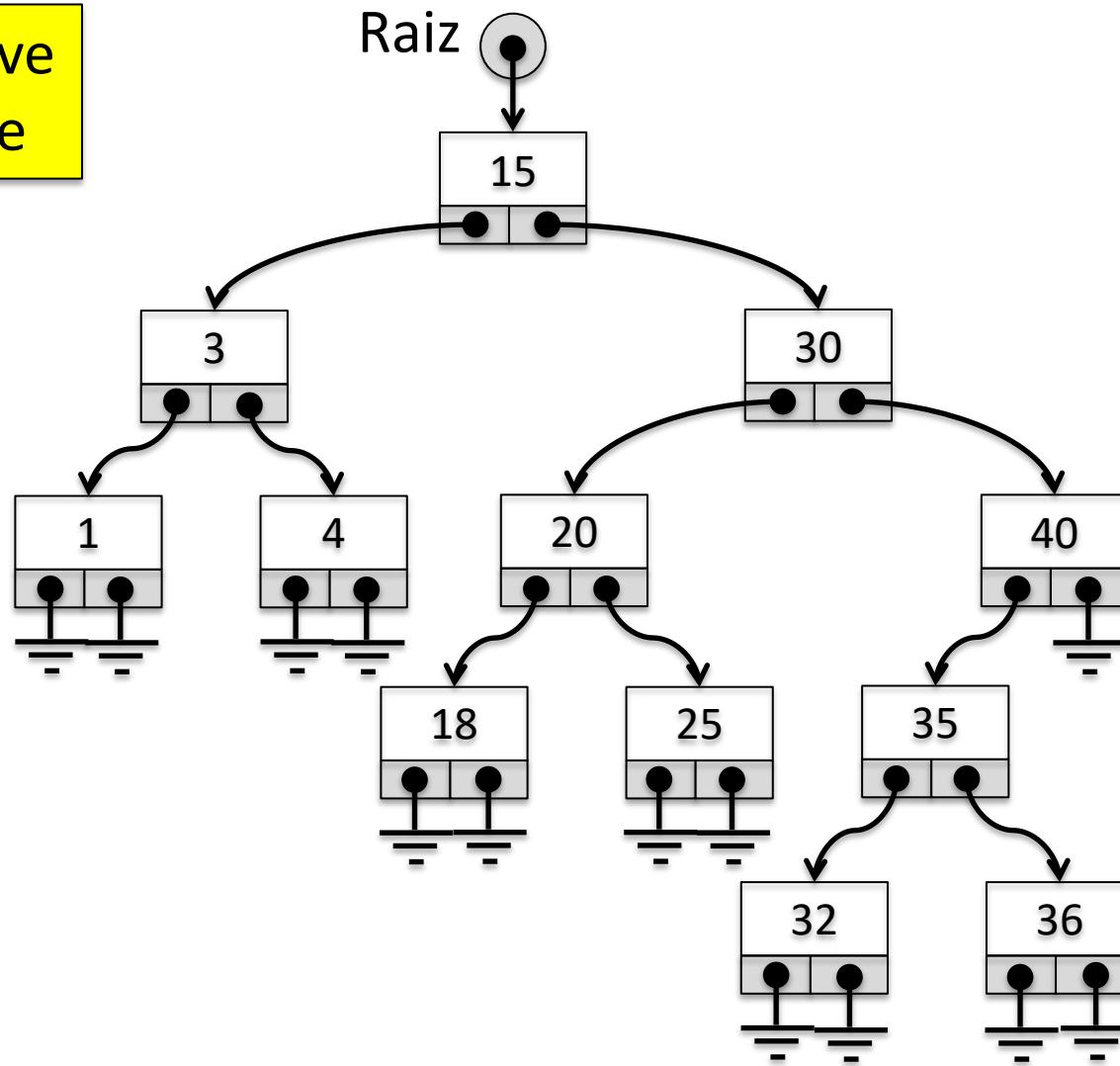


- Alguns comentários:

1. A retirada de um registro não é tão simples quanto a inserção
2. Se o nó que contém o registro a ser retirado possuir no máximo um descendente \Rightarrow a operação é simples
3. No caso do nó conter dois descendentes o registro a ser retirado deve ser primeiro:
 - substituído pelo registro mais à direita na subárvore esquerda
 - ou pelo registro mais à esquerda na subárvore direita

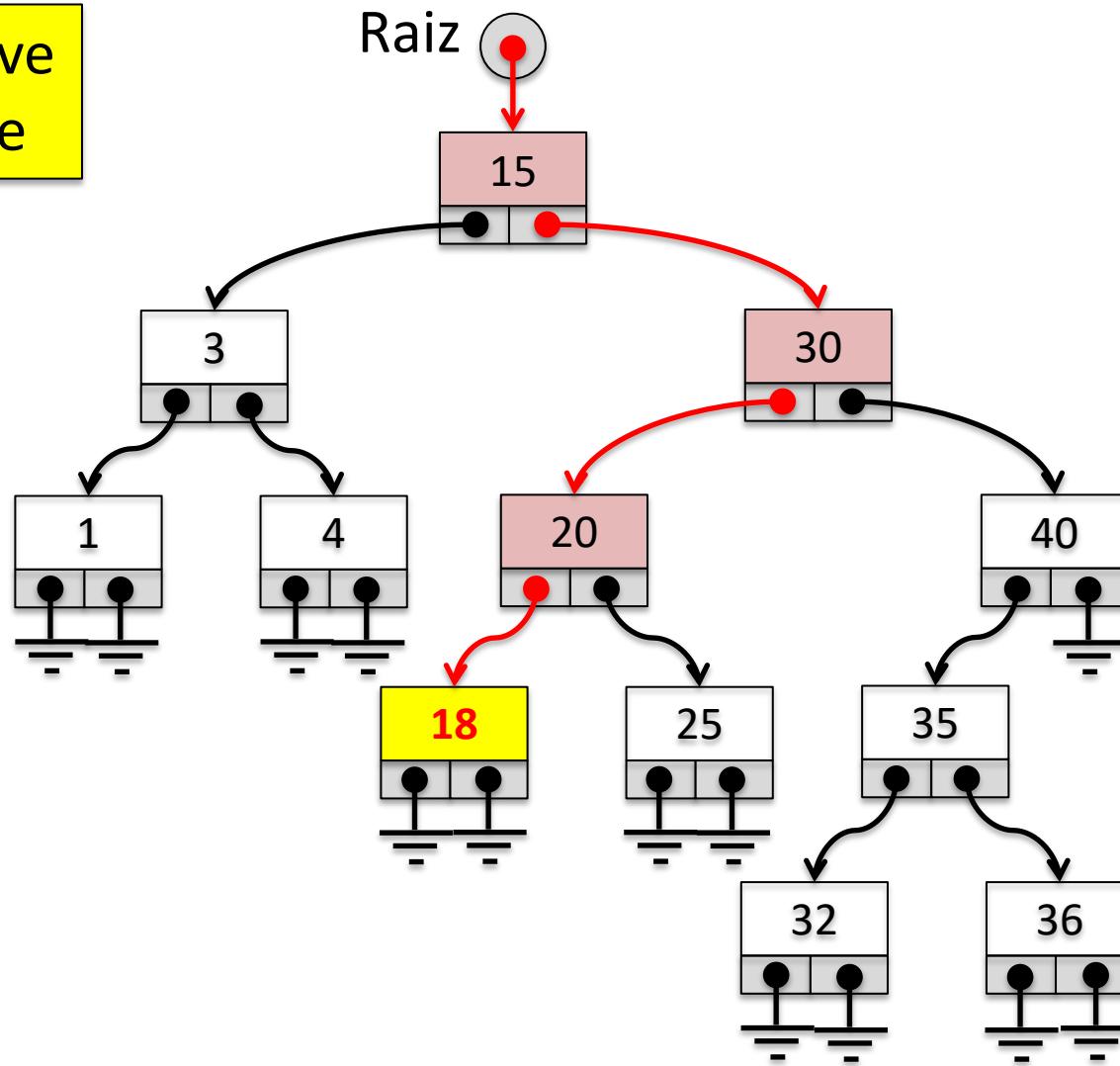
Árvores Binárias de Pesquisa

remover a chave
= 18 da árvore



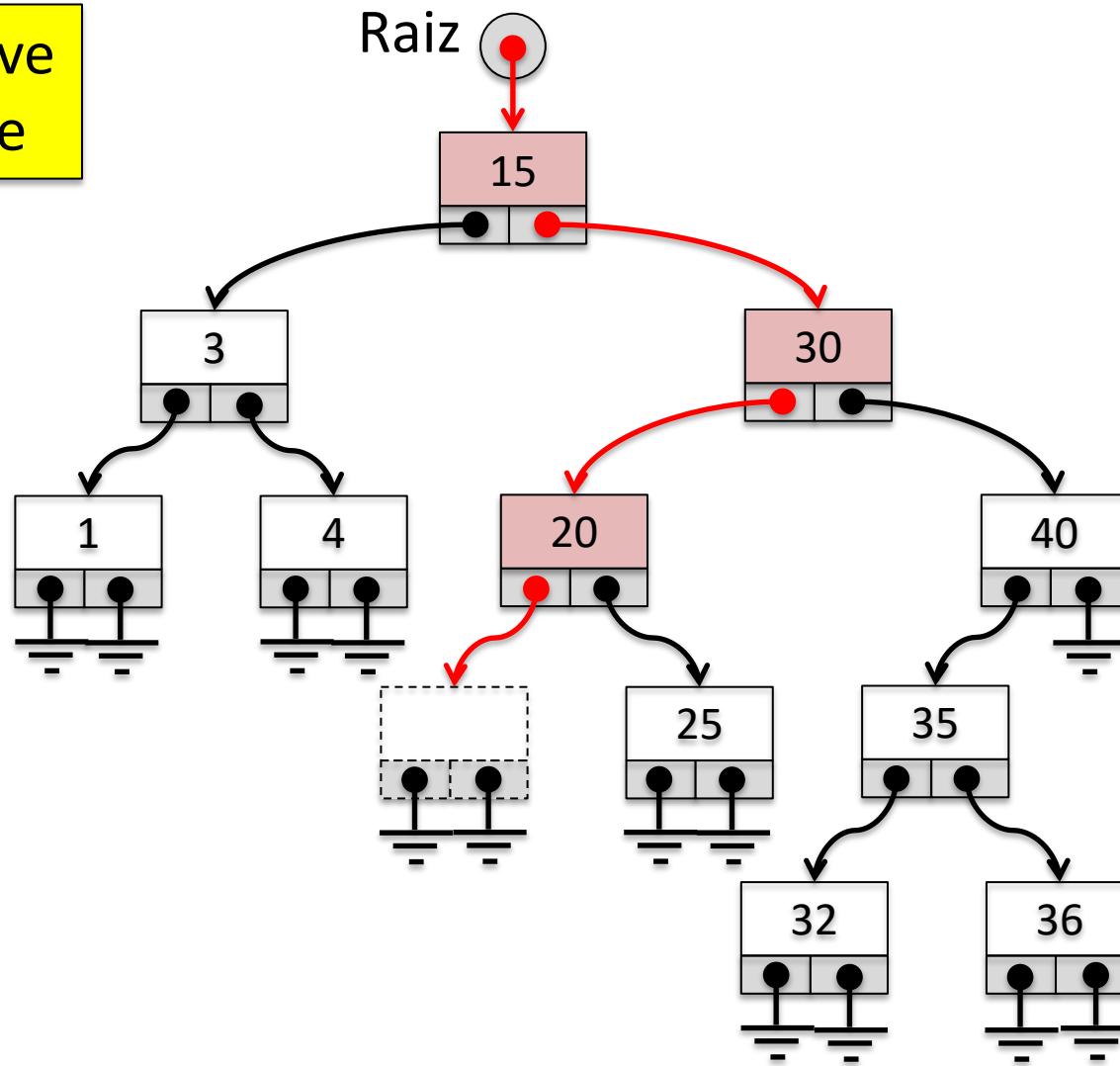
Árvores Binárias de Pesquisa

remover a chave
= 18 da árvore



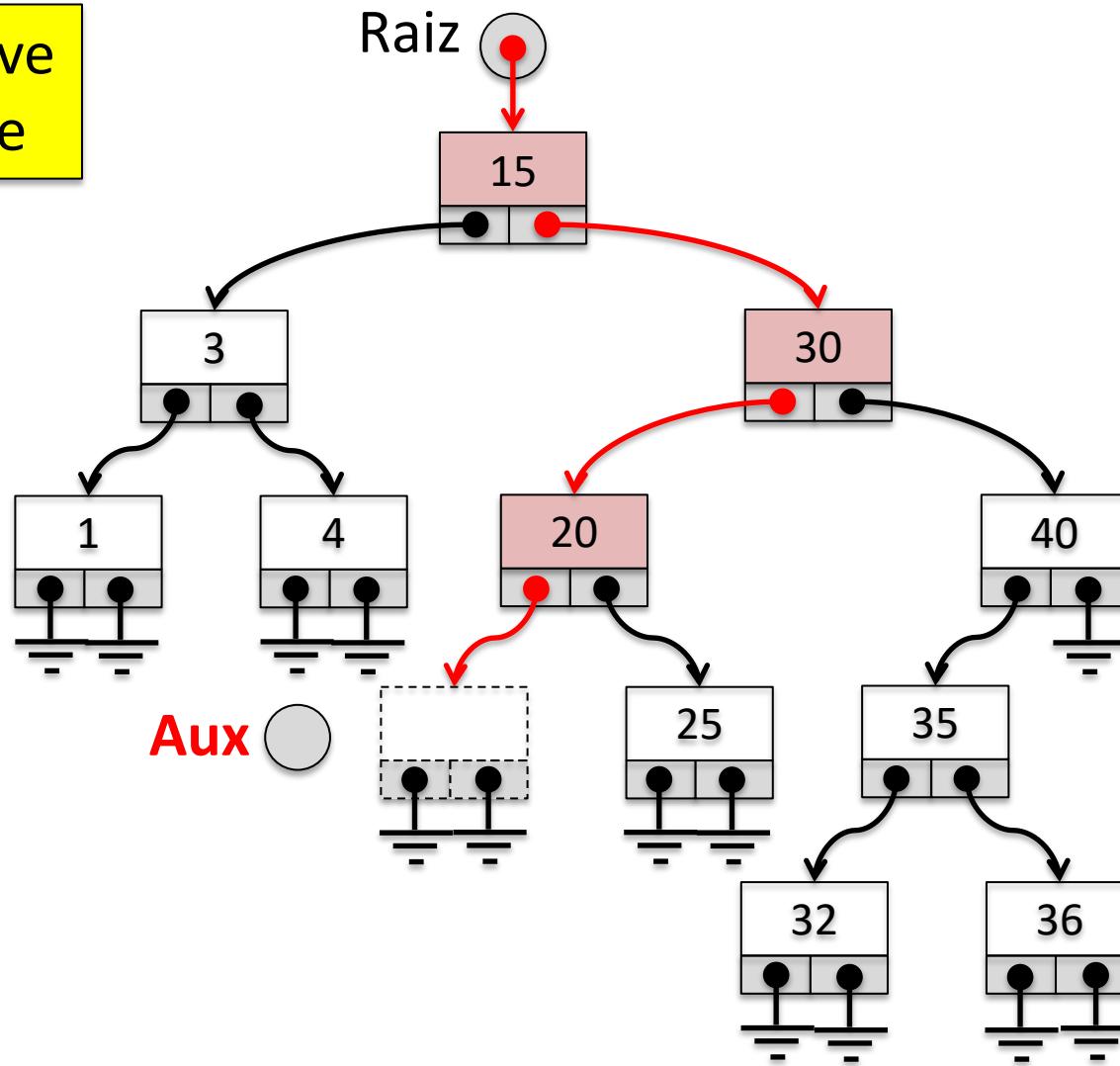
Árvores Binárias de Pesquisa

remover a chave
= 18 da árvore



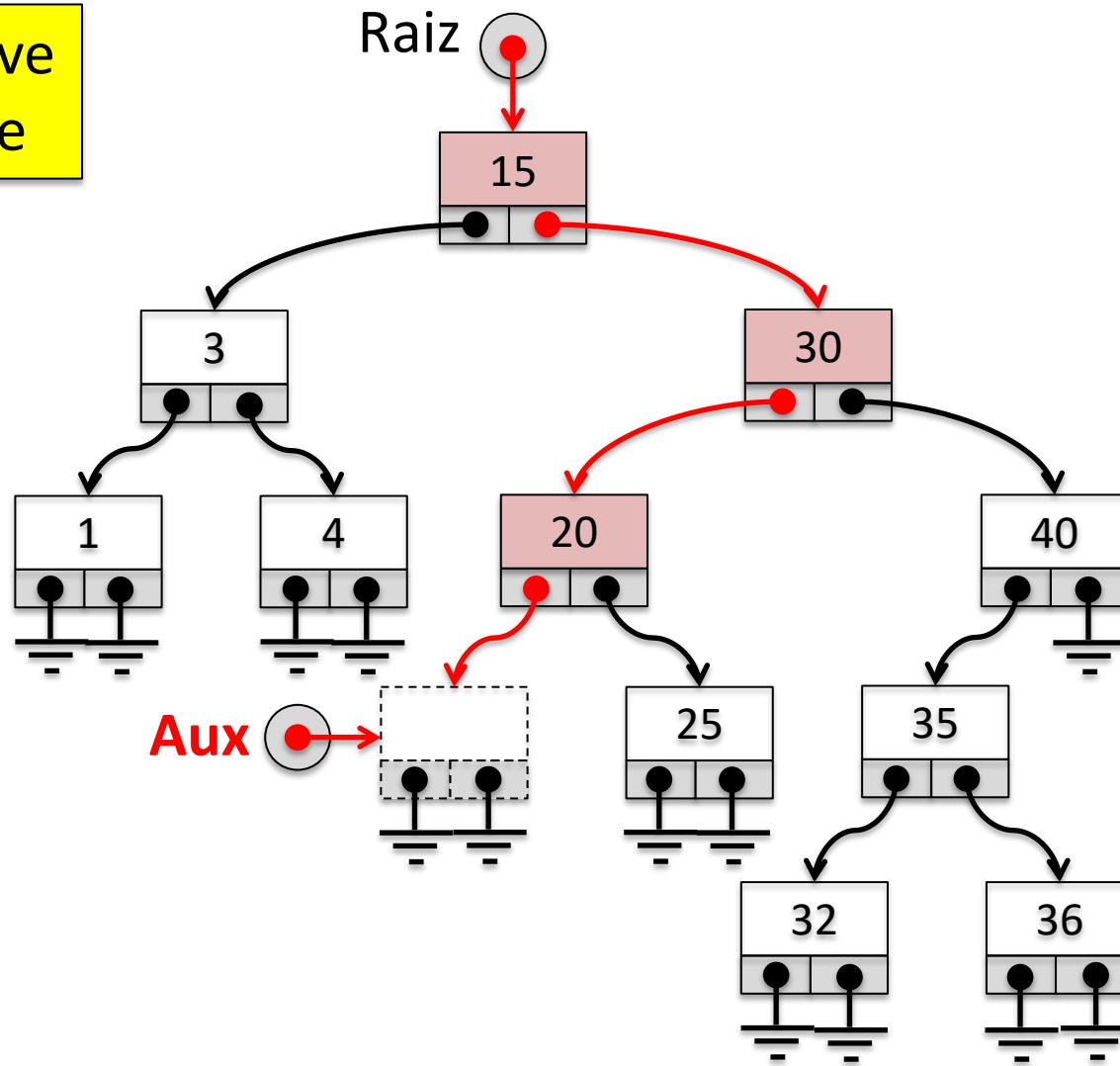
Árvores Binárias de Pesquisa

remover a chave
= 18 da árvore



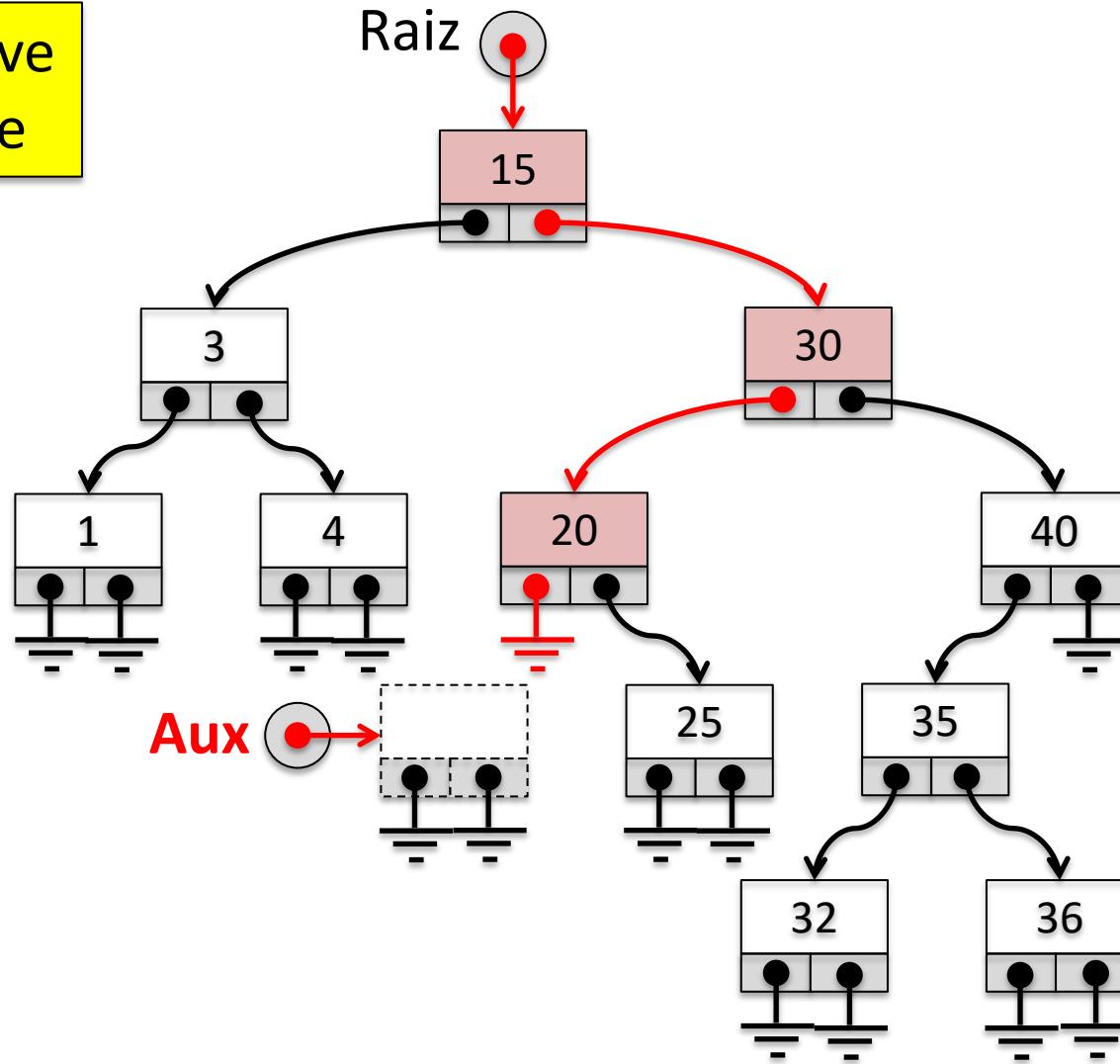
Árvores Binárias de Pesquisa

remover a chave
= 18 da árvore



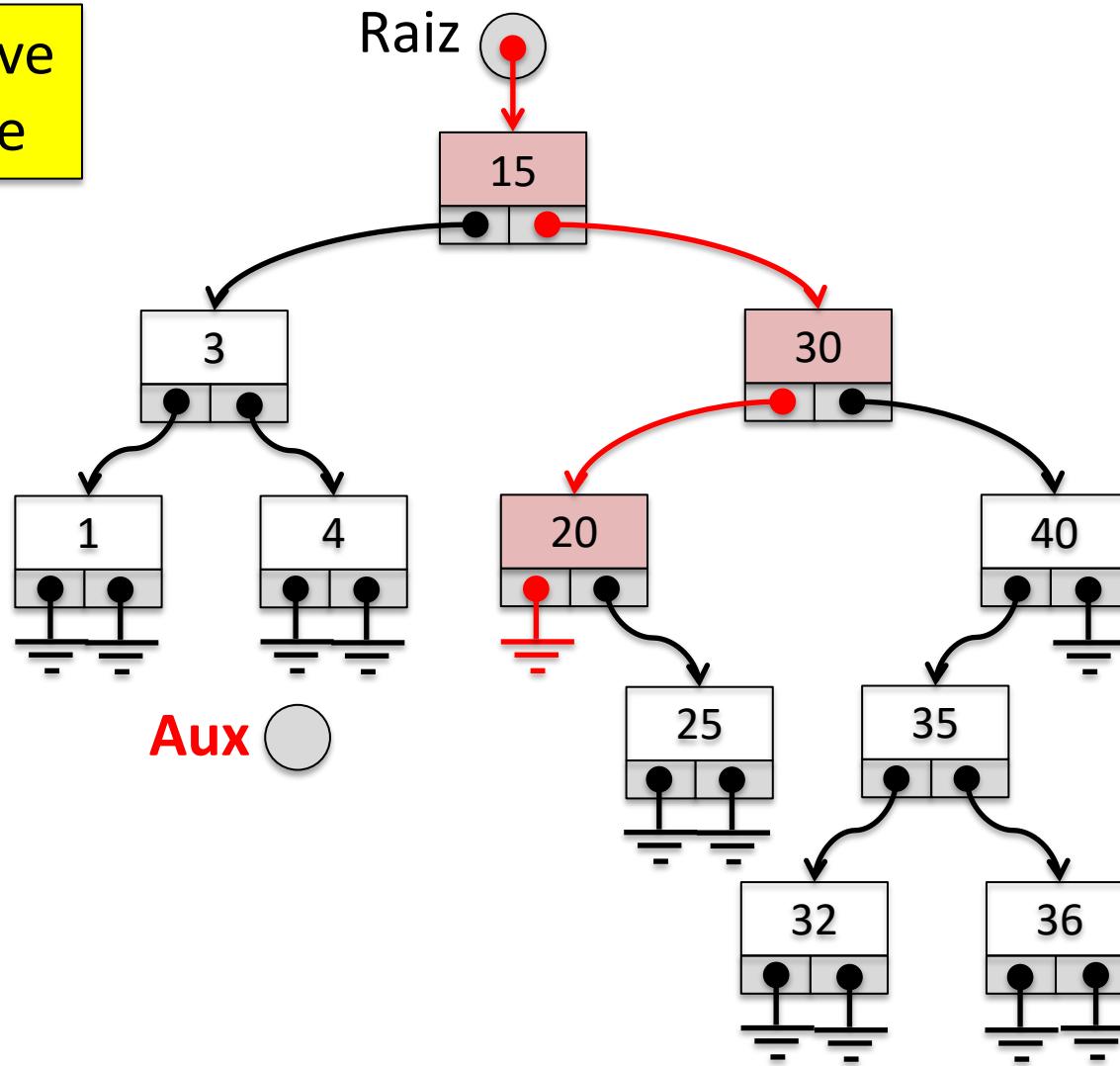
Árvores Binárias de Pesquisa

remover a chave
= 18 da árvore



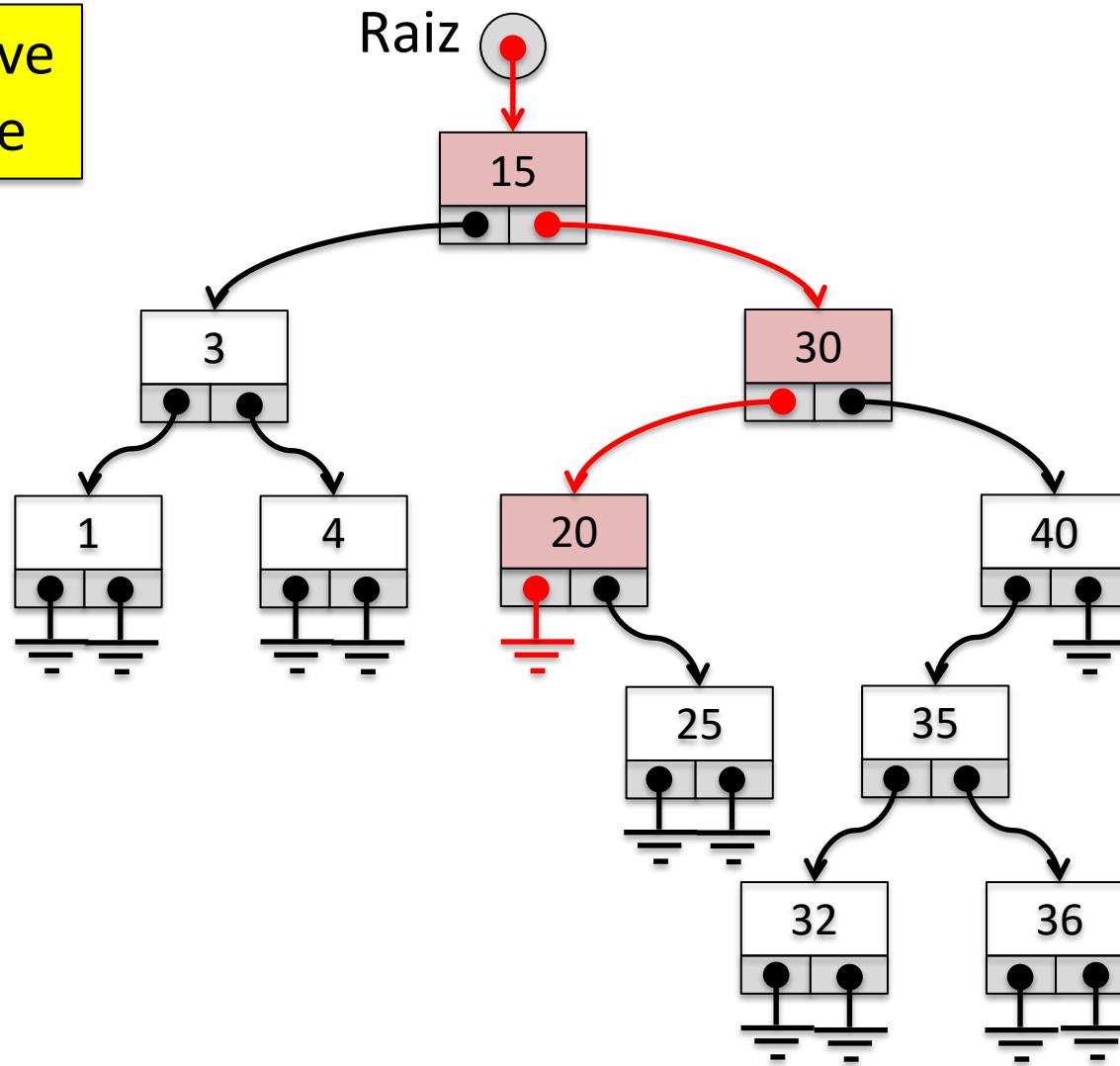
Árvores Binárias de Pesquisa

remover a chave
= 18 da árvore

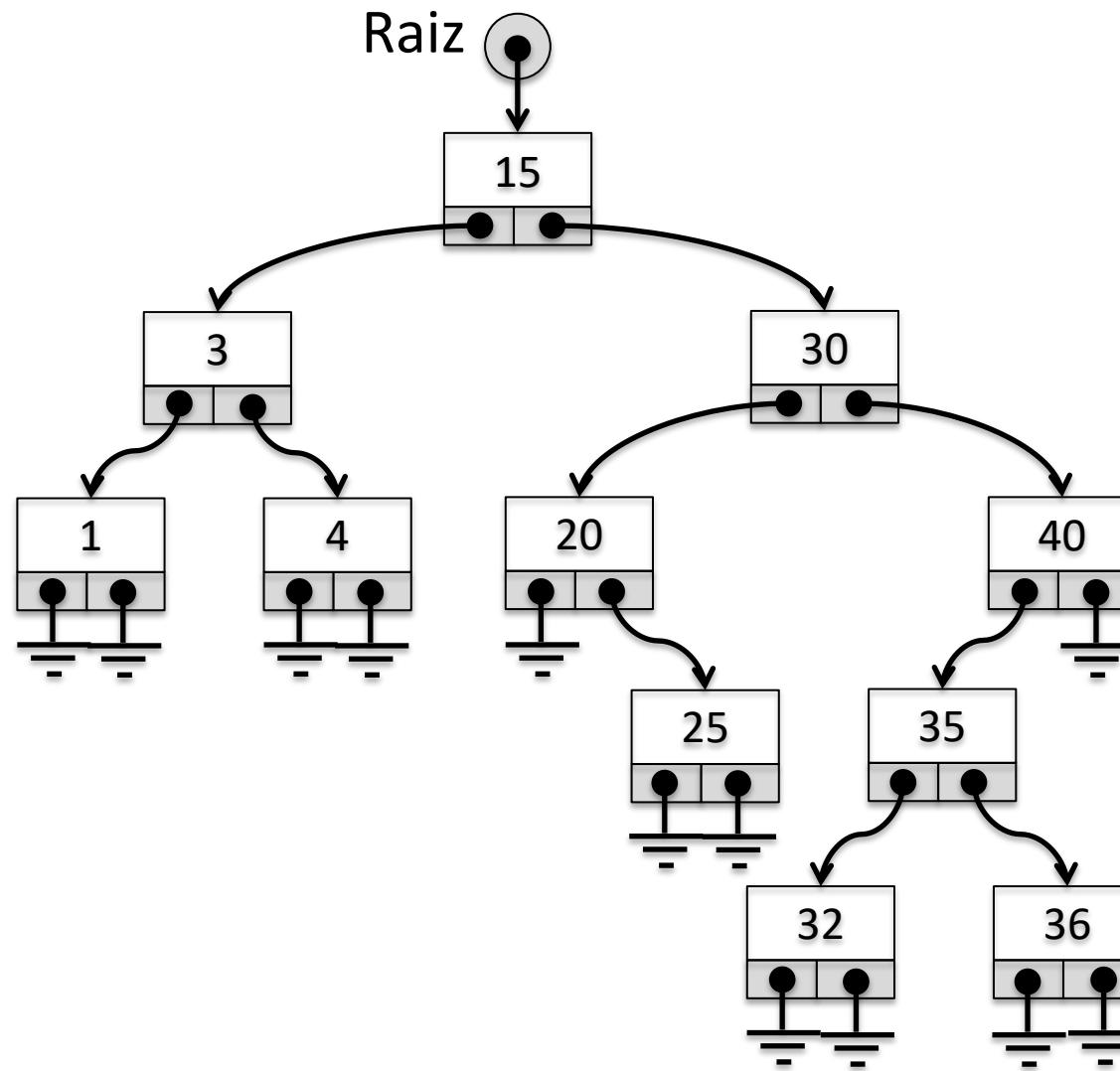


Árvores Binárias de Pesquisa

remover a chave
= 18 da árvore

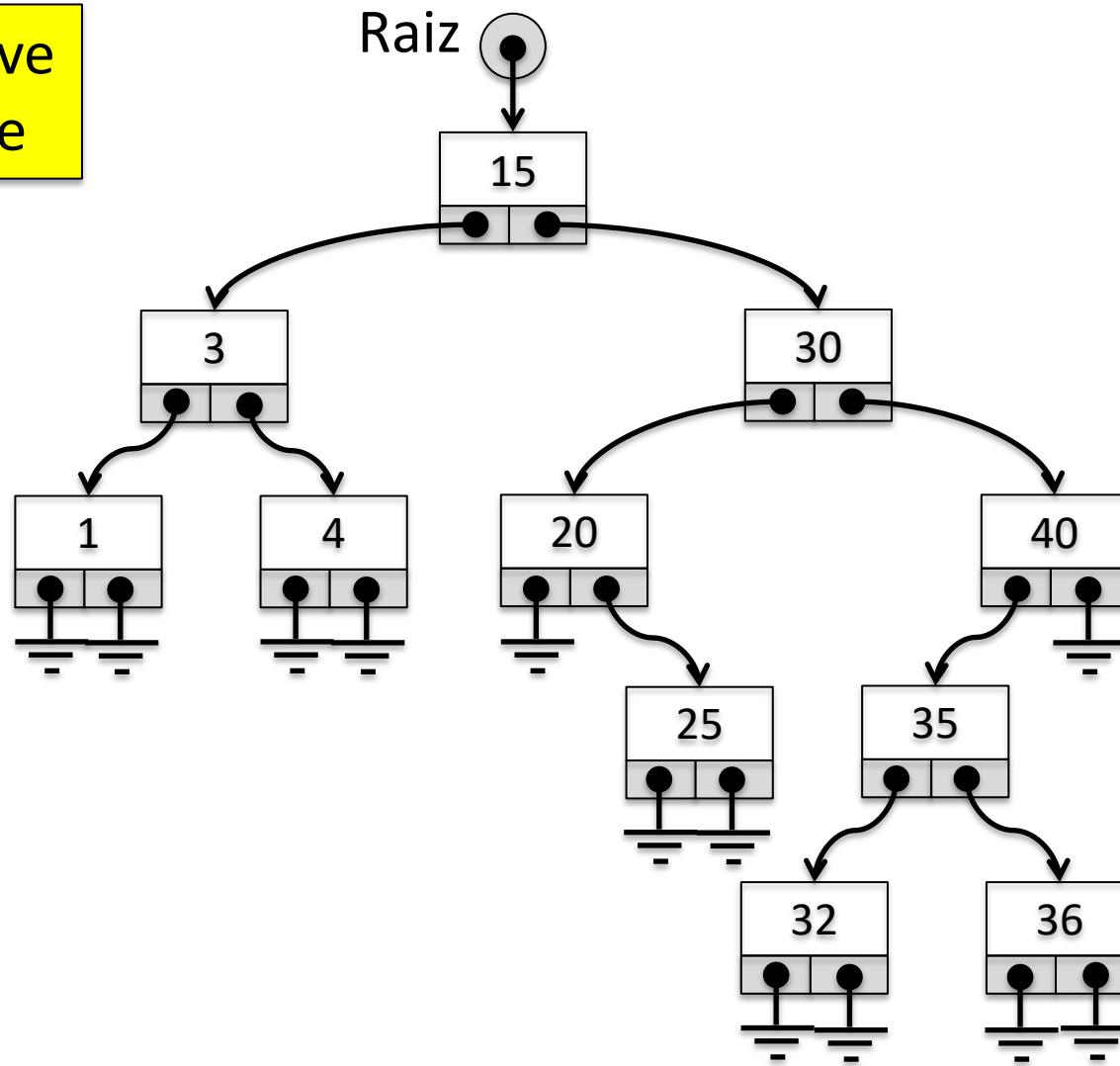


Árvores Binárias de Pesquisa



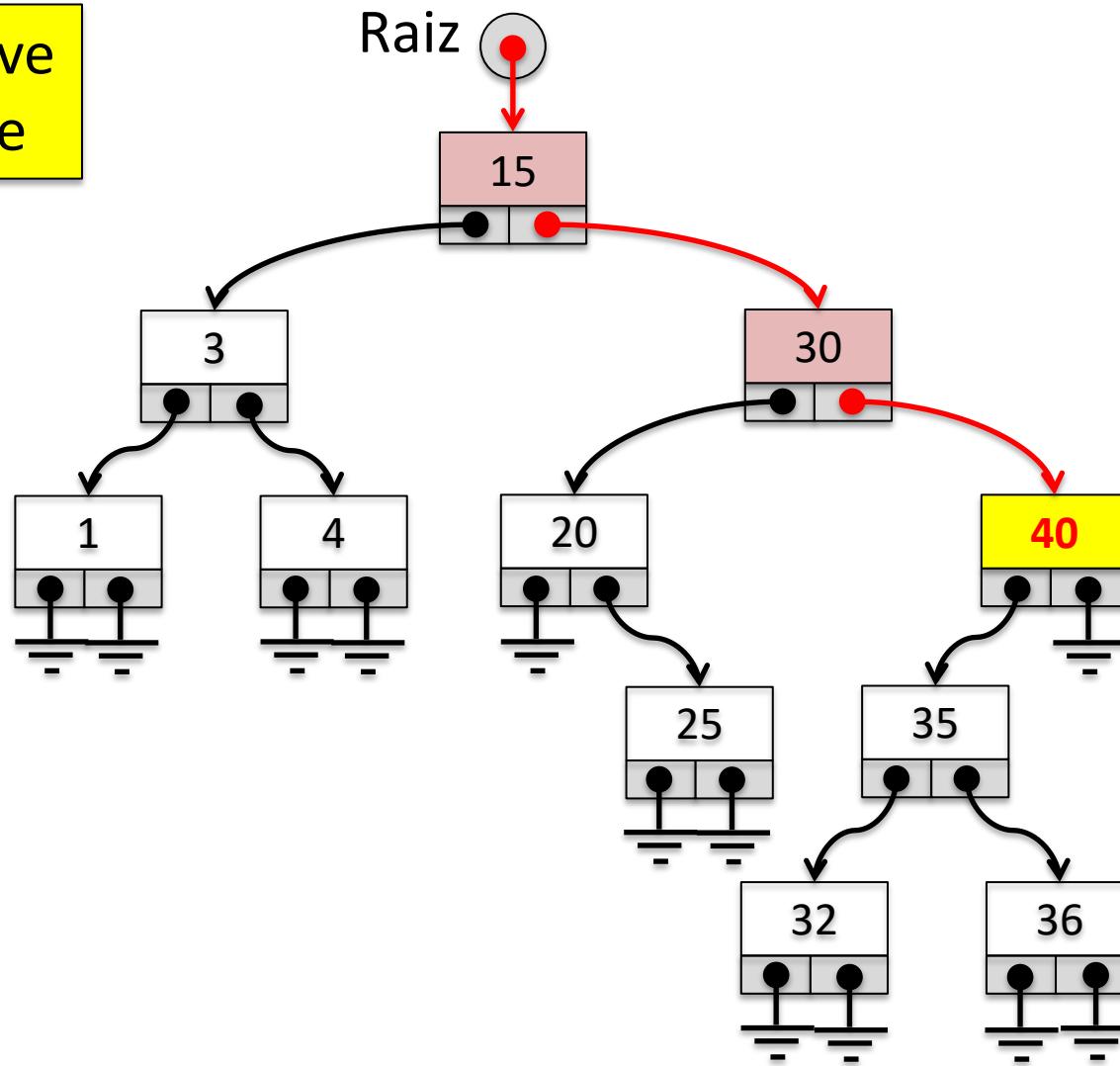
Árvores Binárias de Pesquisa

remover a chave
= 40 da árvore



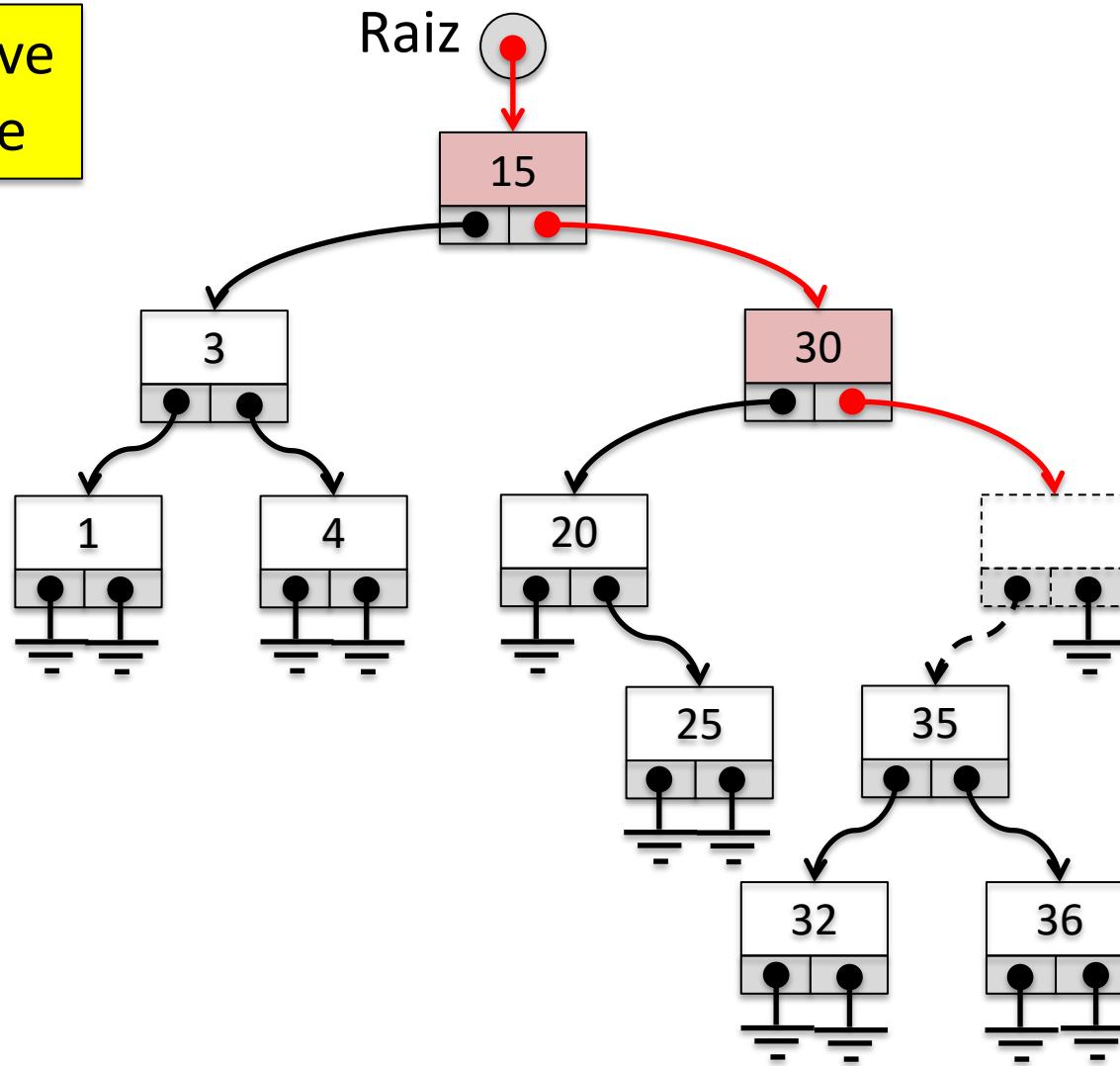
Árvores Binárias de Pesquisa

remover a chave
= 40 da árvore



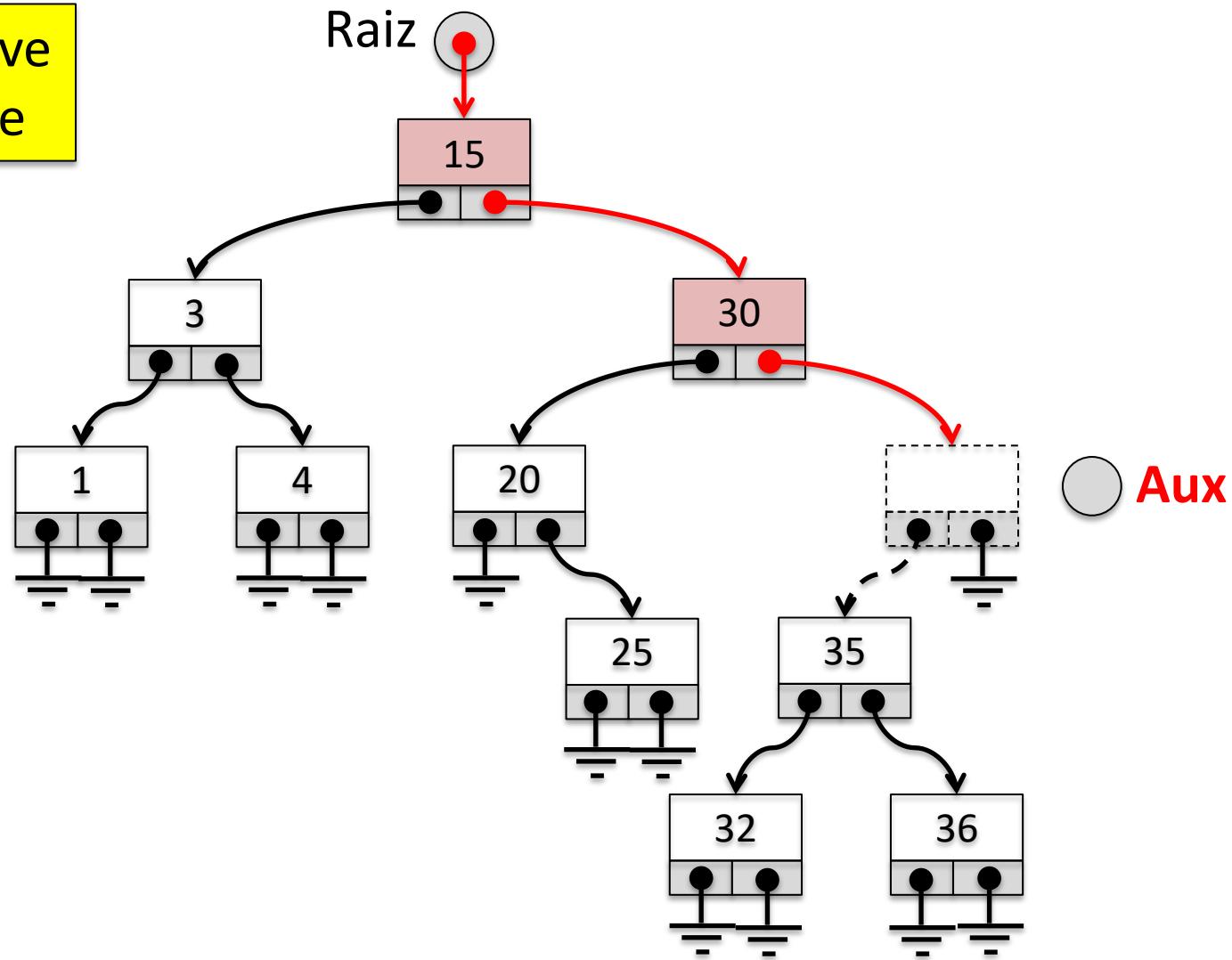
Árvores Binárias de Pesquisa

remover a chave
= 40 da árvore



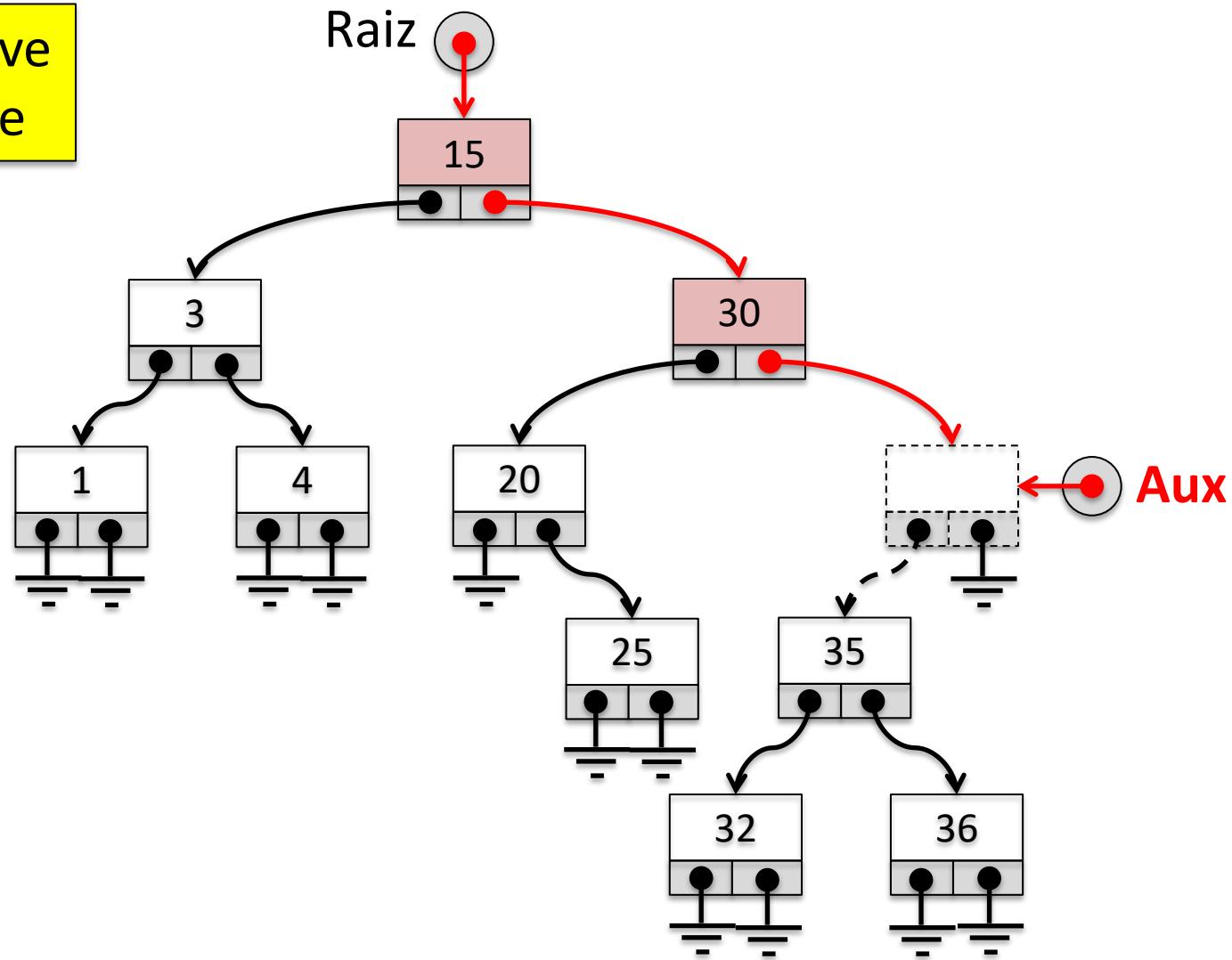
Árvores Binárias de Pesquisa

remover a chave
= 40 da árvore



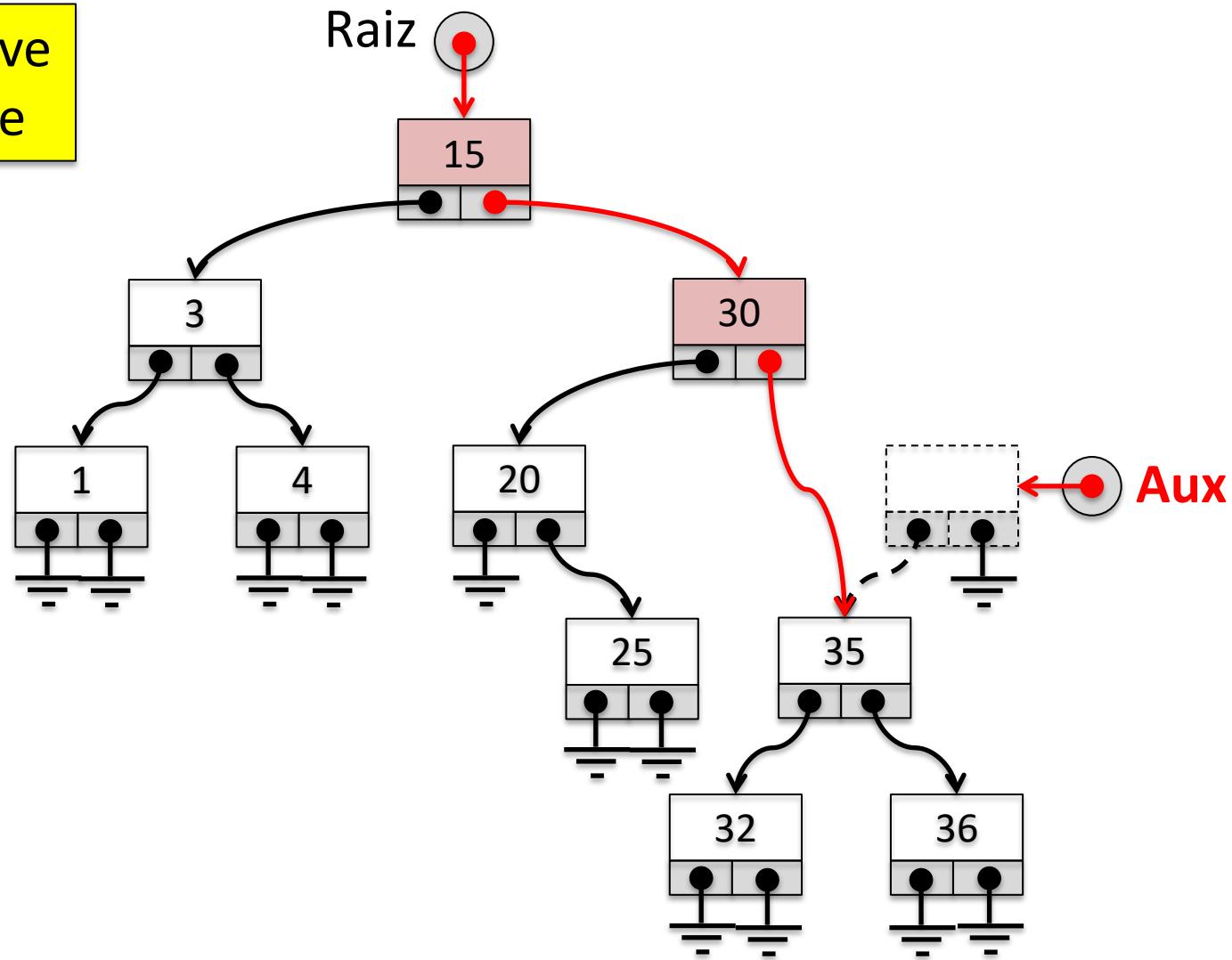
Árvores Binárias de Pesquisa

remover a chave
= 40 da árvore



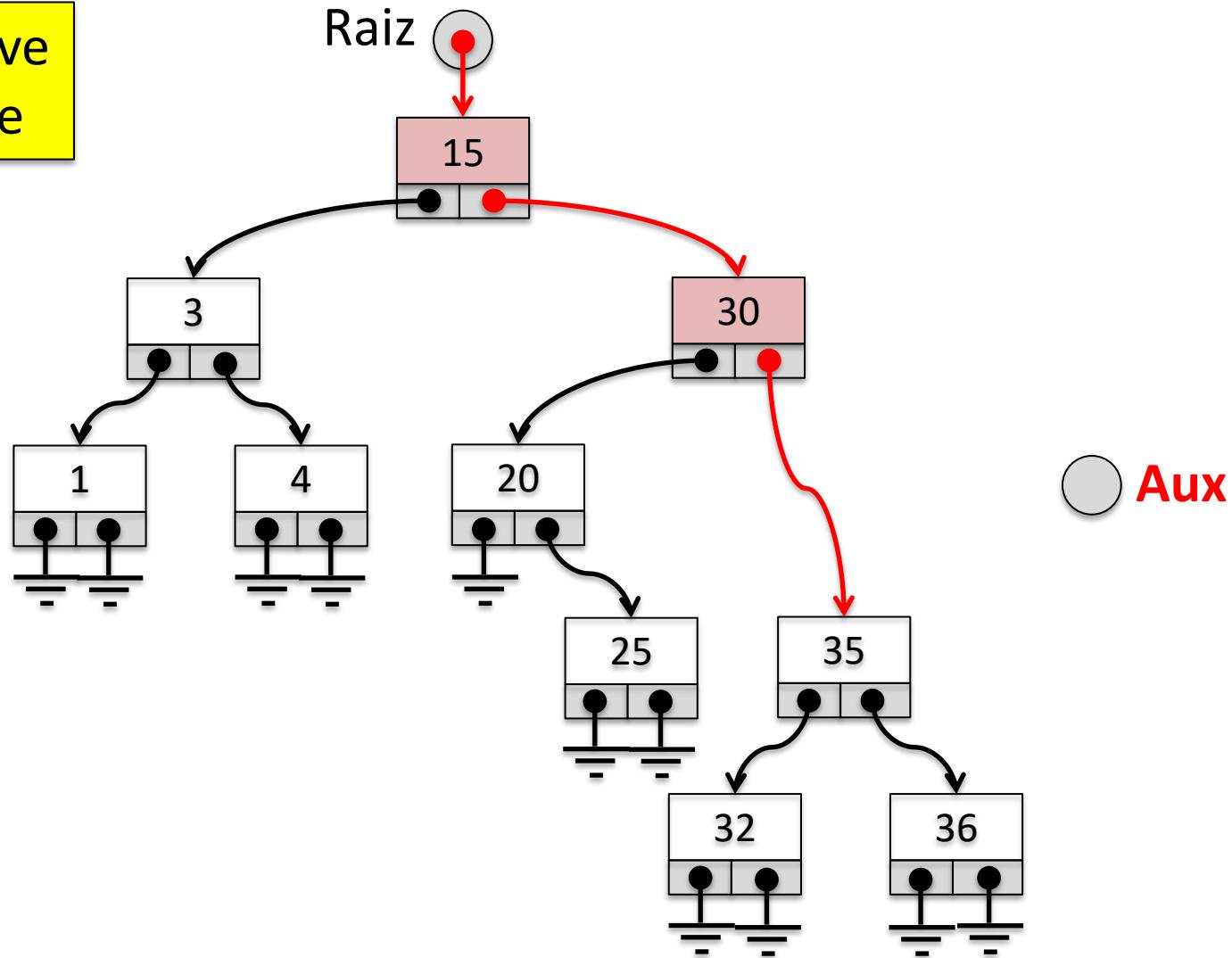
Árvores Binárias de Pesquisa

remover a chave
= 40 da árvore



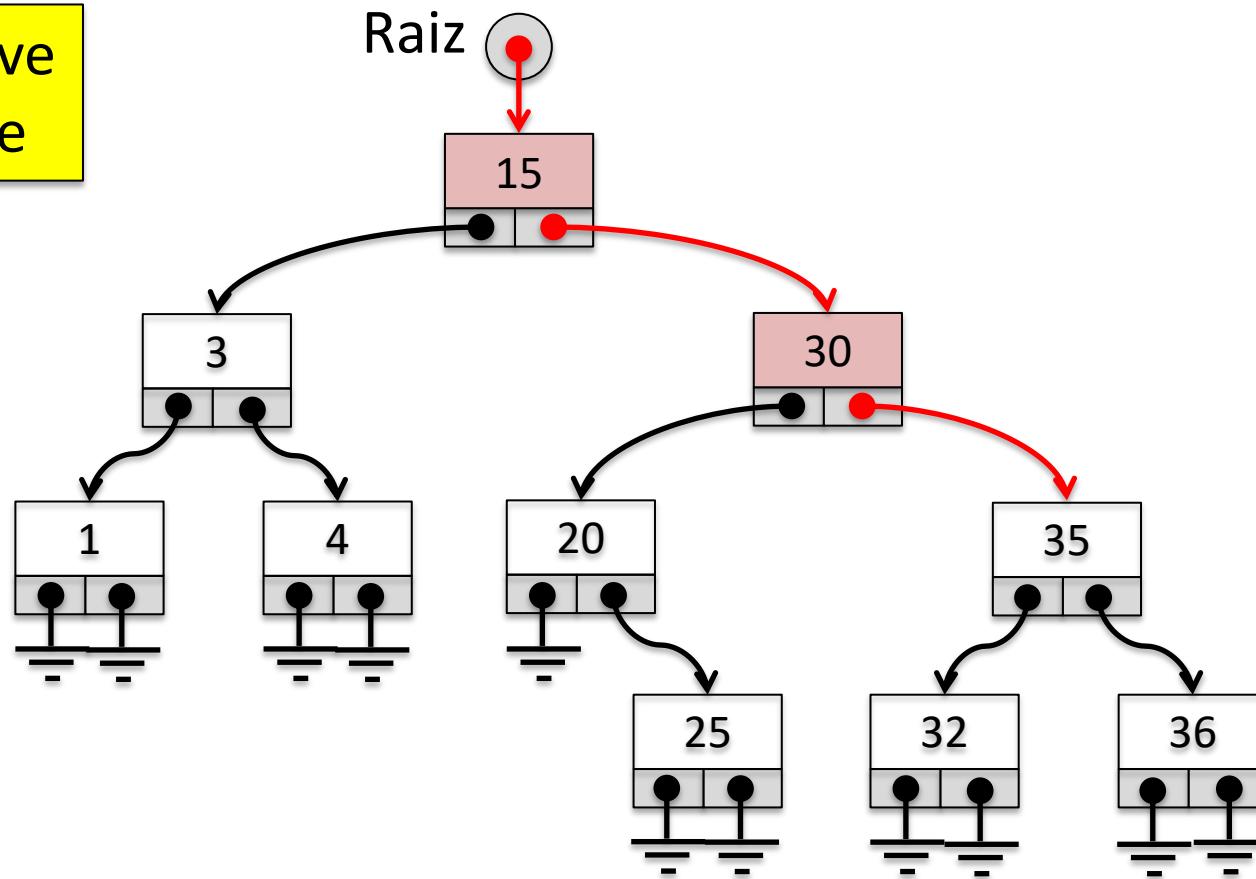
Árvores Binárias de Pesquisa

remover a chave
= 40 da árvore

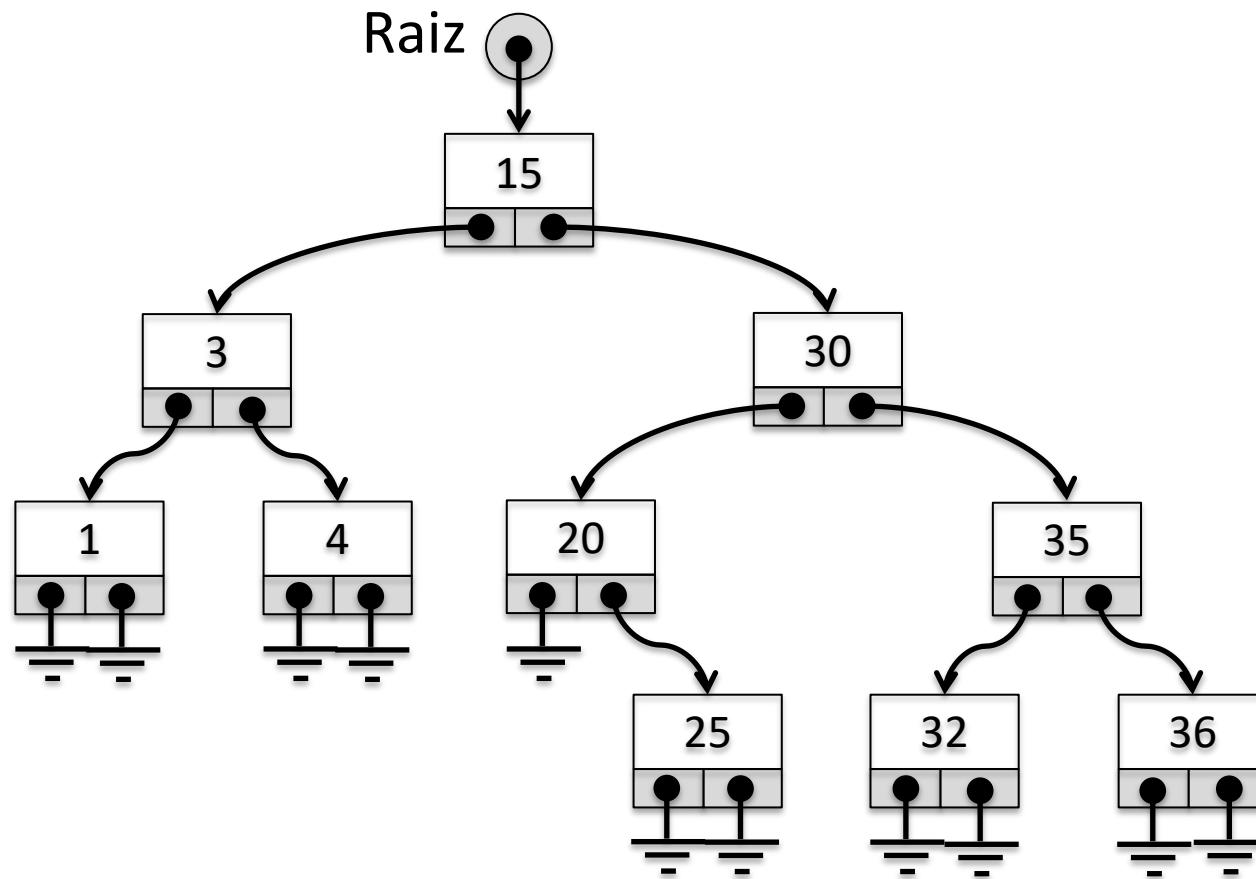


Árvores Binárias de Pesquisa

remover a chave
= 40 da árvore

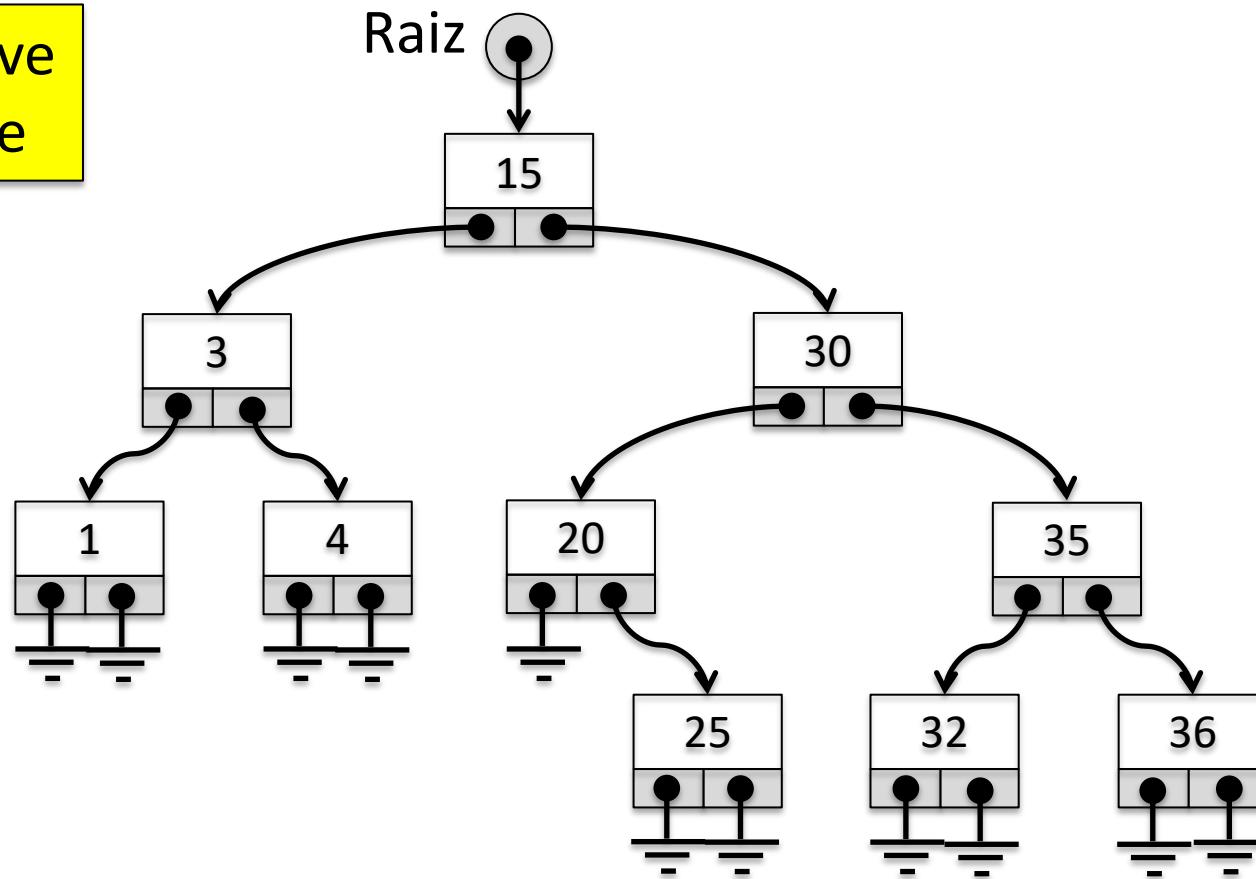


Árvores Binárias de Pesquisa



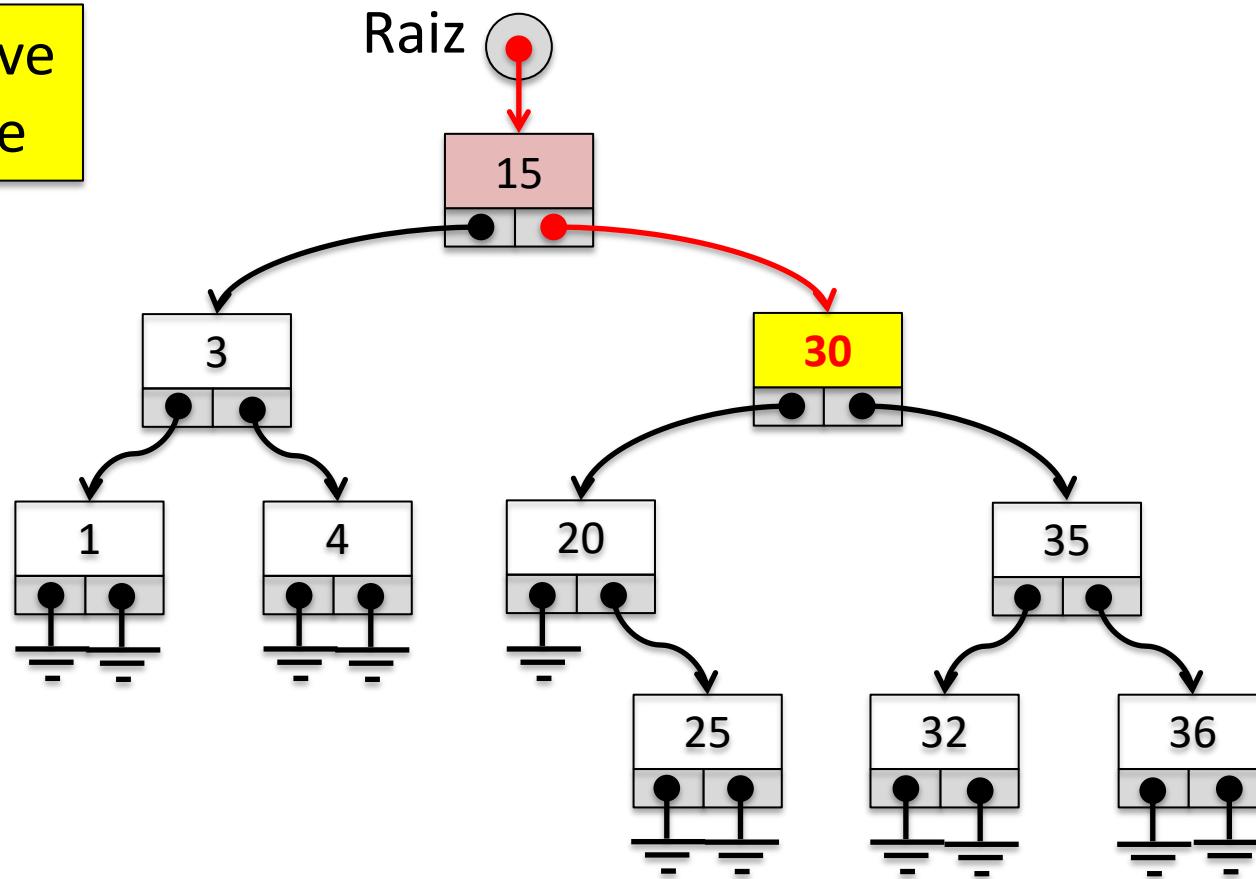
Árvores Binárias de Pesquisa

remover a chave
= 30 da árvore



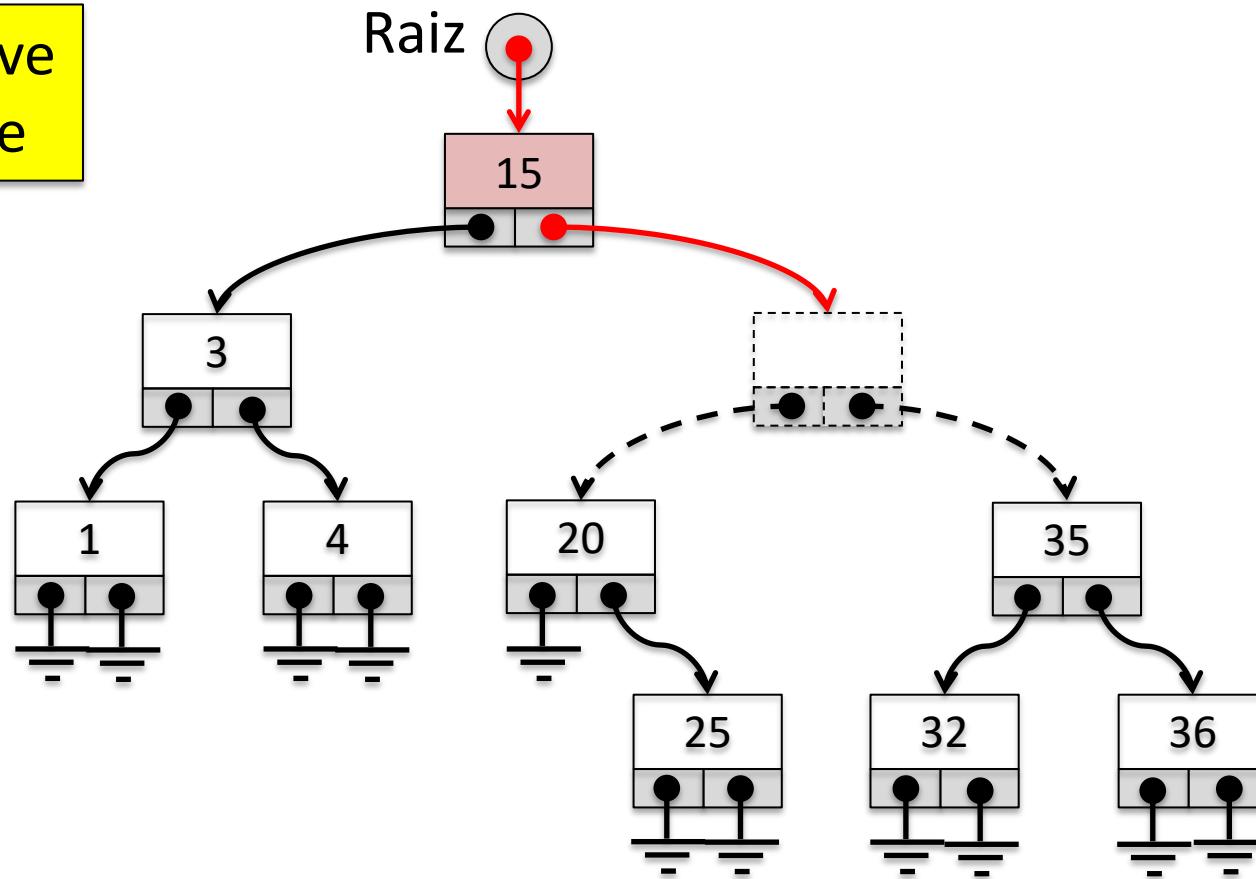
Árvores Binárias de Pesquisa

remover a chave
= 30 da árvore



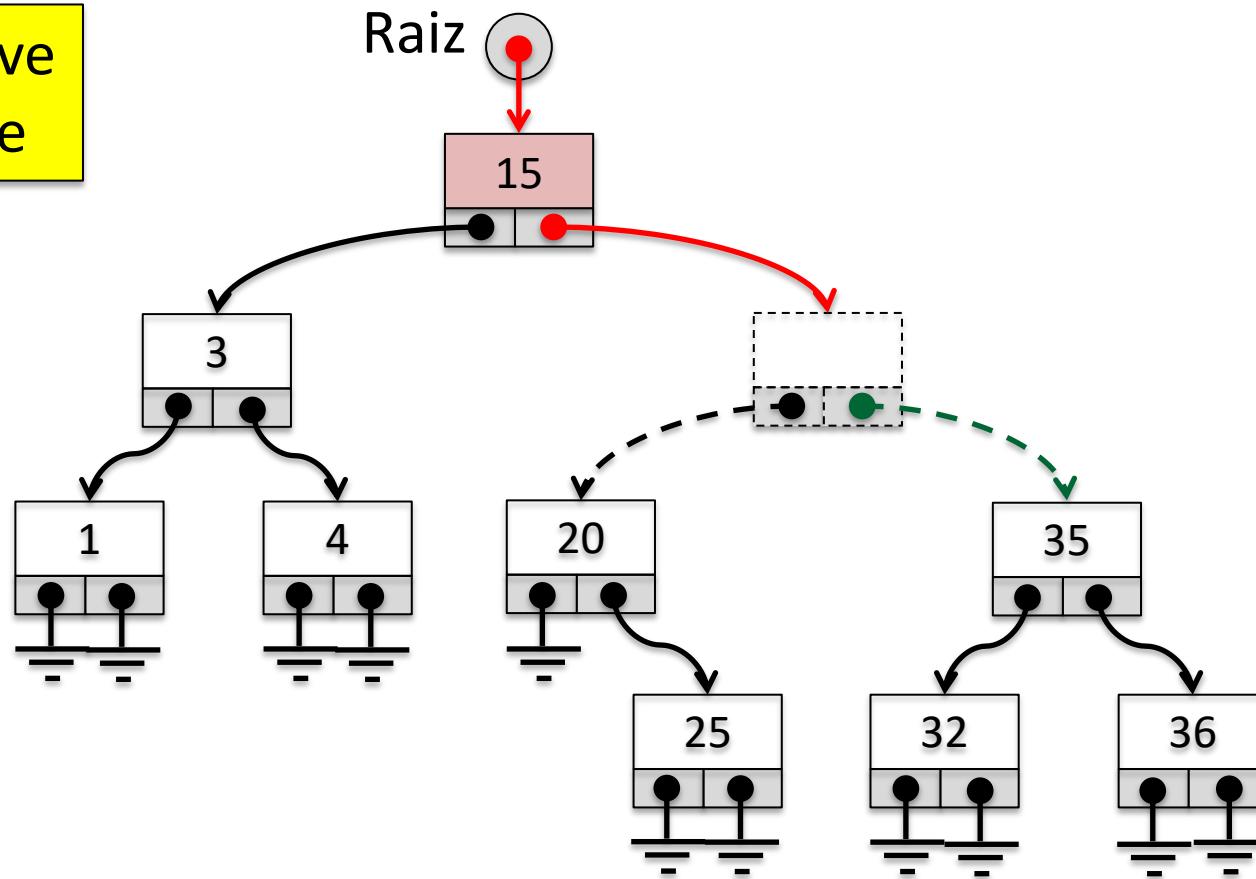
Árvores Binárias de Pesquisa

remover a chave
= 30 da árvore



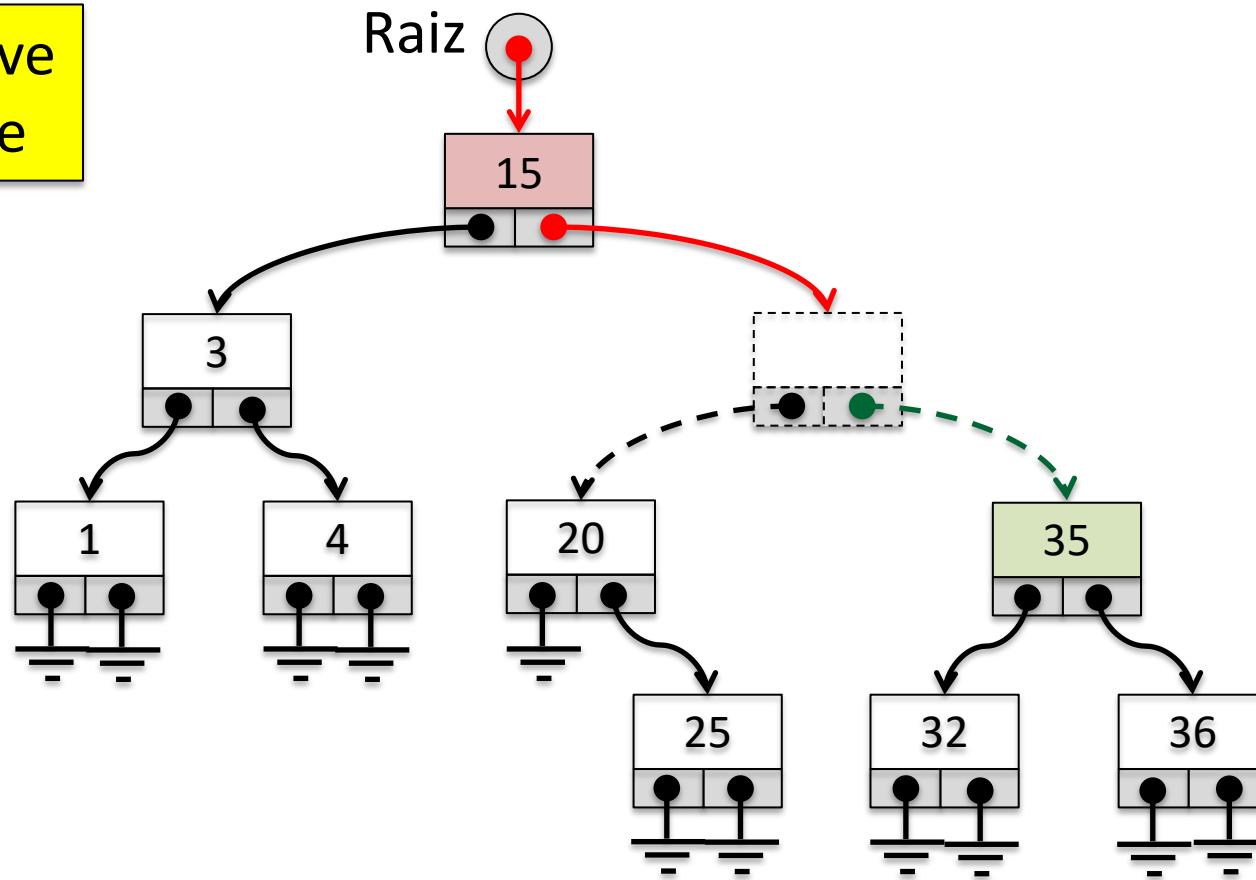
Árvores Binárias de Pesquisa

remover a chave
= 30 da árvore



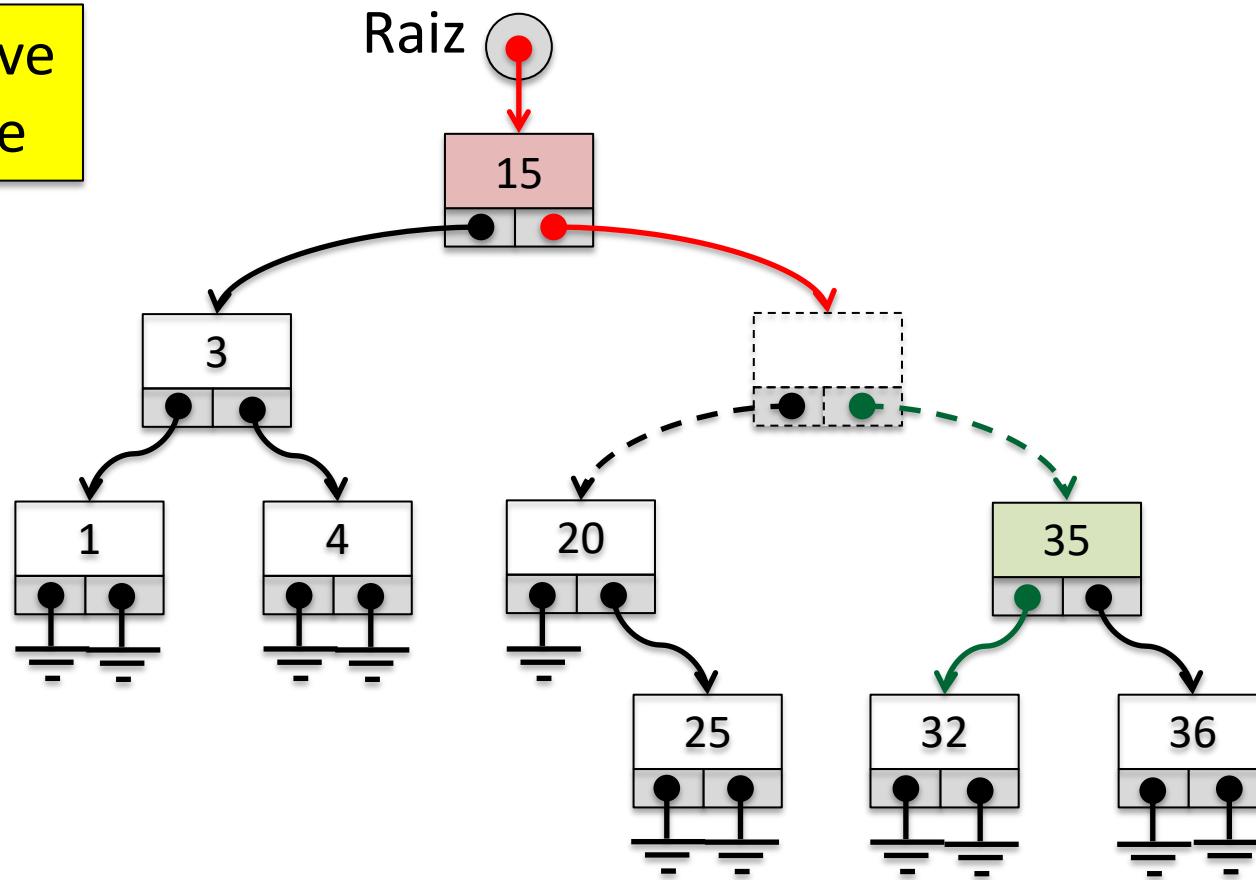
Árvores Binárias de Pesquisa

remover a chave
= 30 da árvore



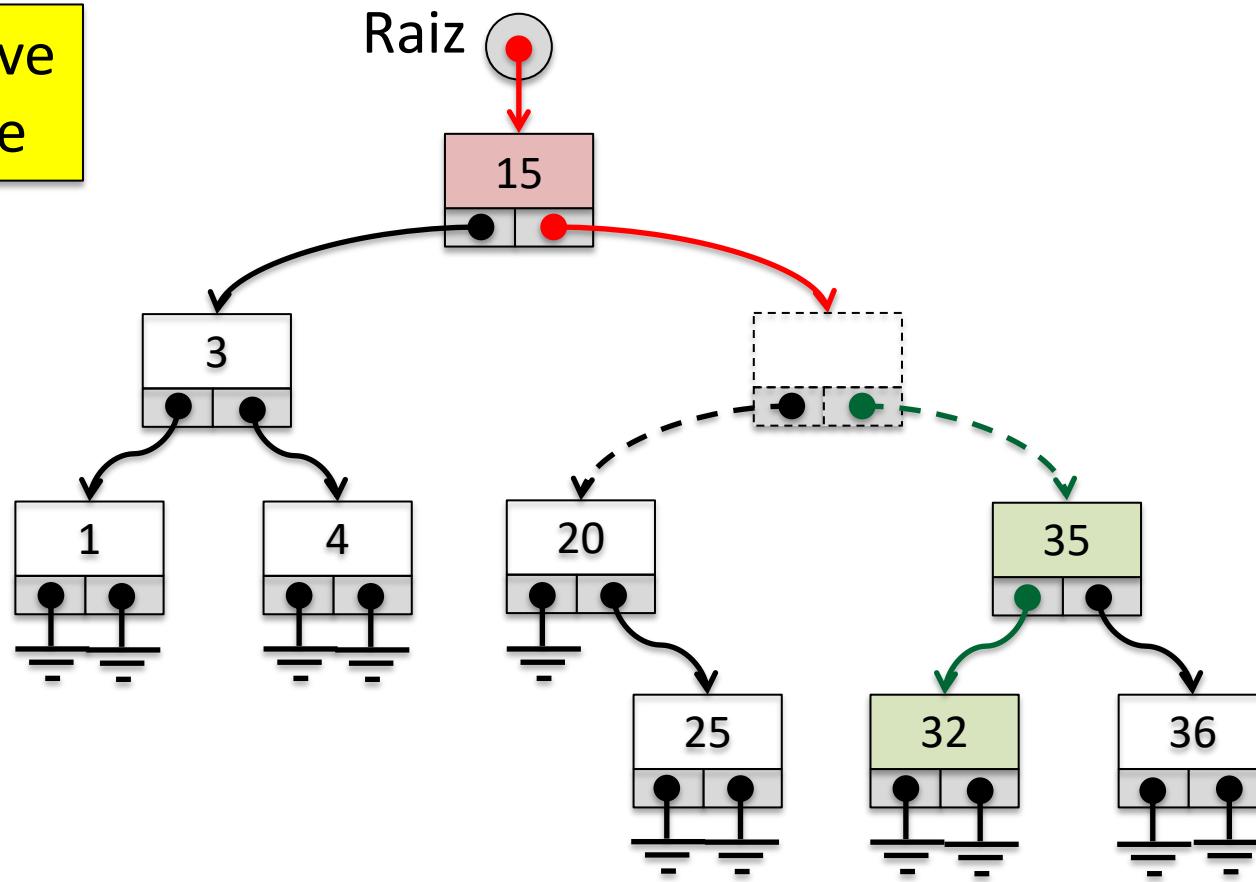
Árvores Binárias de Pesquisa

remover a chave
= 30 da árvore



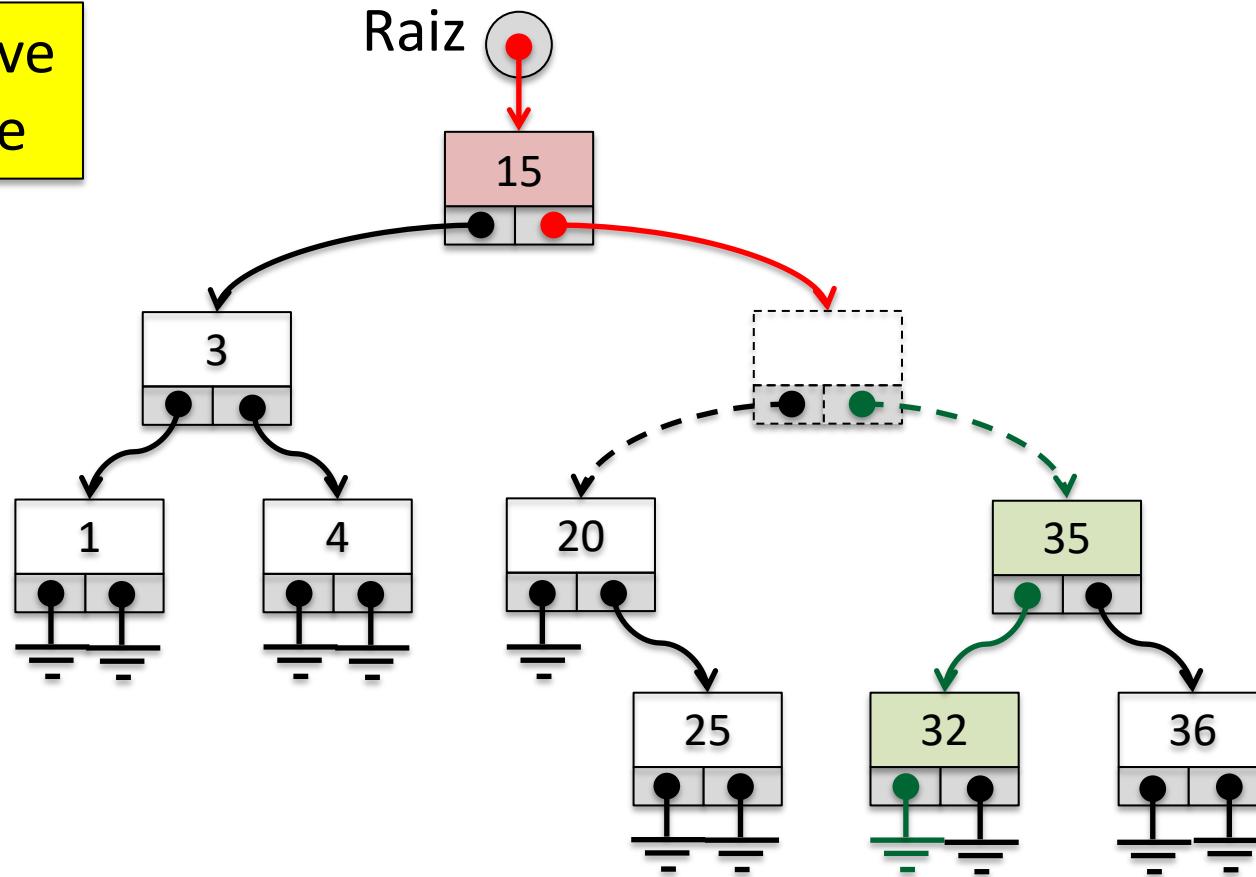
Árvores Binárias de Pesquisa

remover a chave
= 30 da árvore



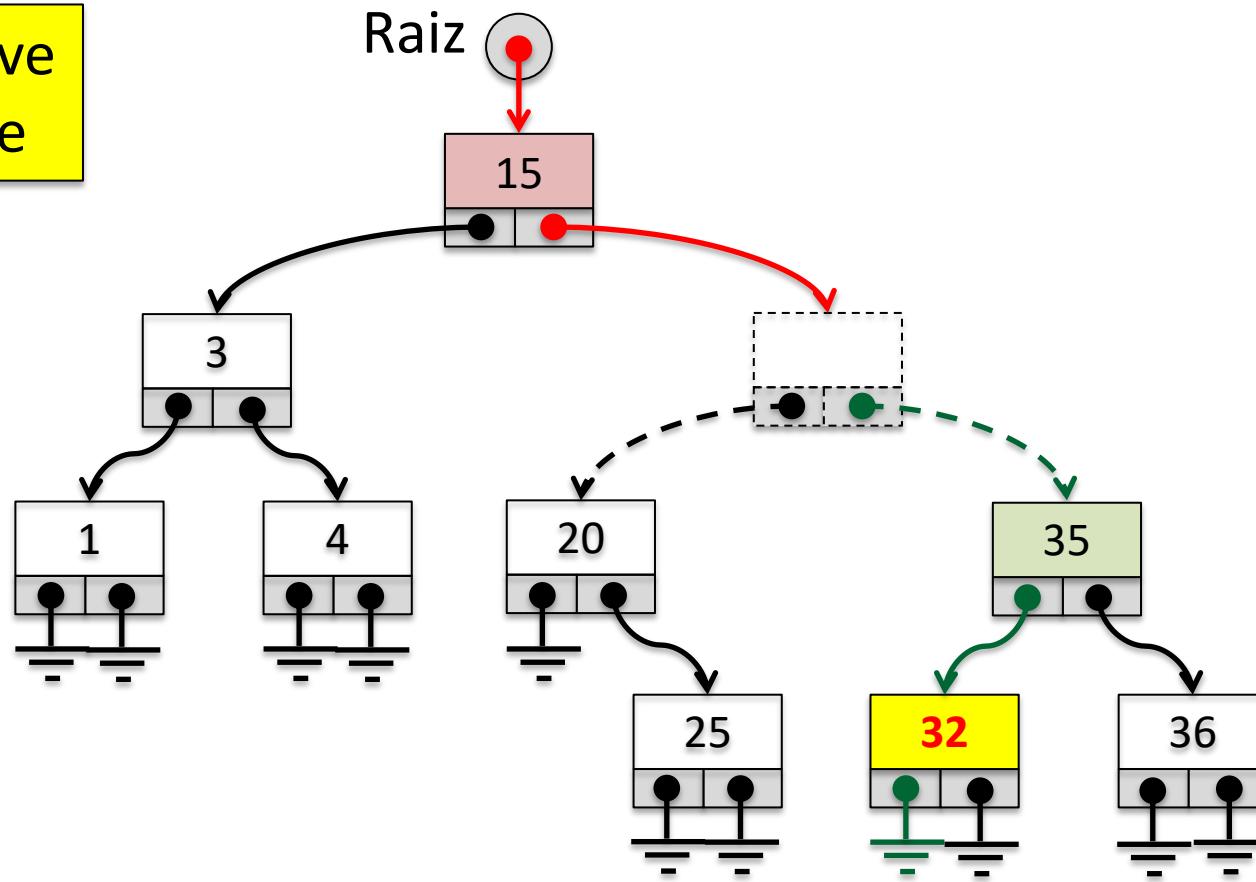
Árvores Binárias de Pesquisa

remover a chave
= 30 da árvore



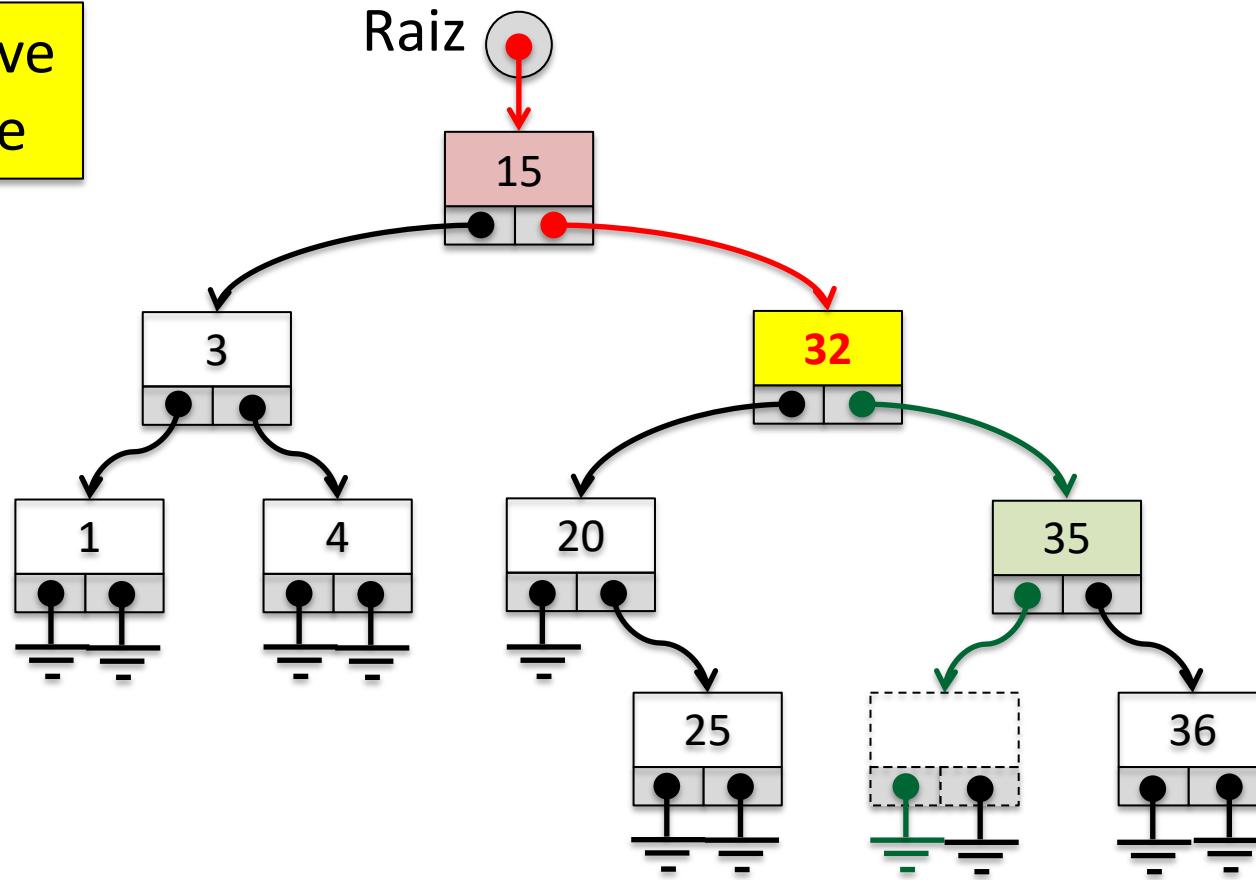
Árvores Binárias de Pesquisa

remover a chave
= 30 da árvore



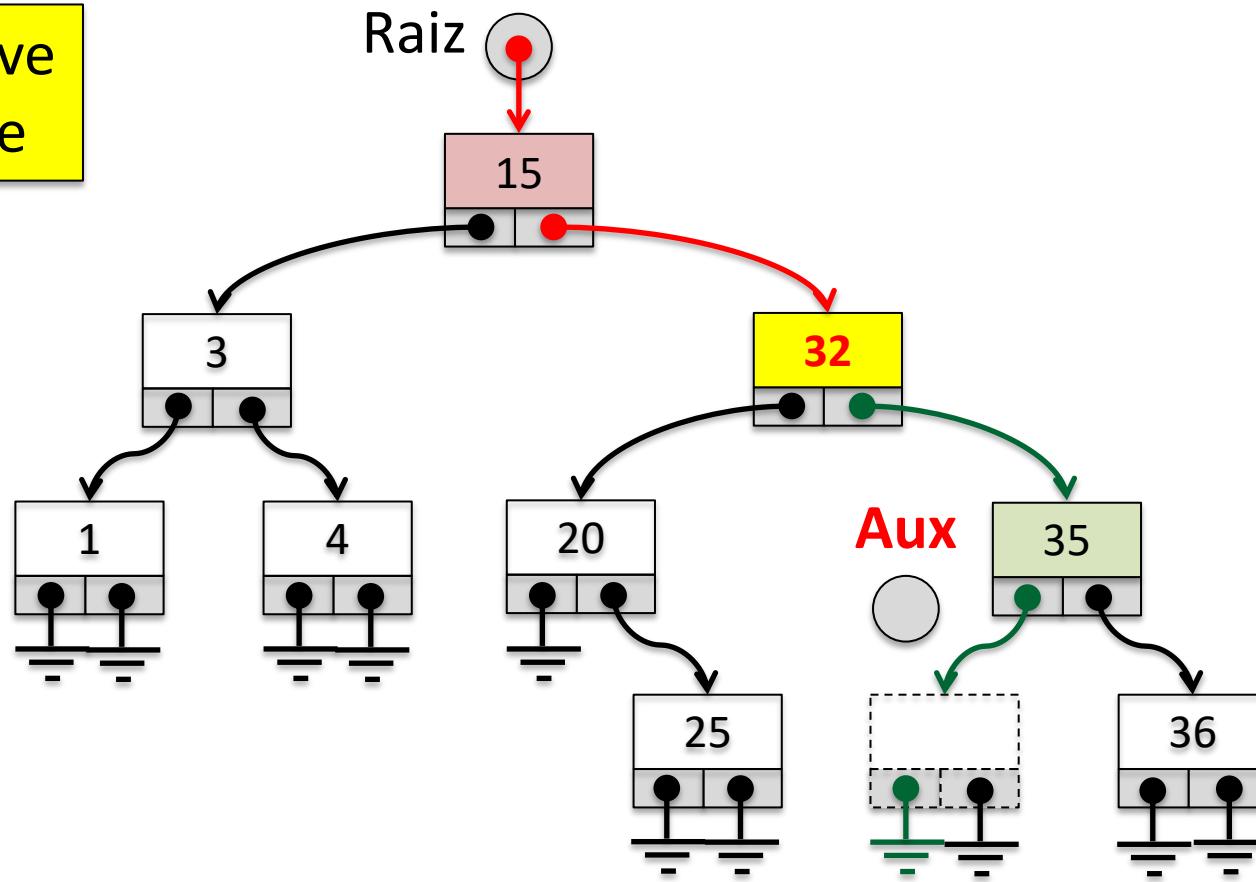
Árvores Binárias de Pesquisa

remover a chave
= 30 da árvore



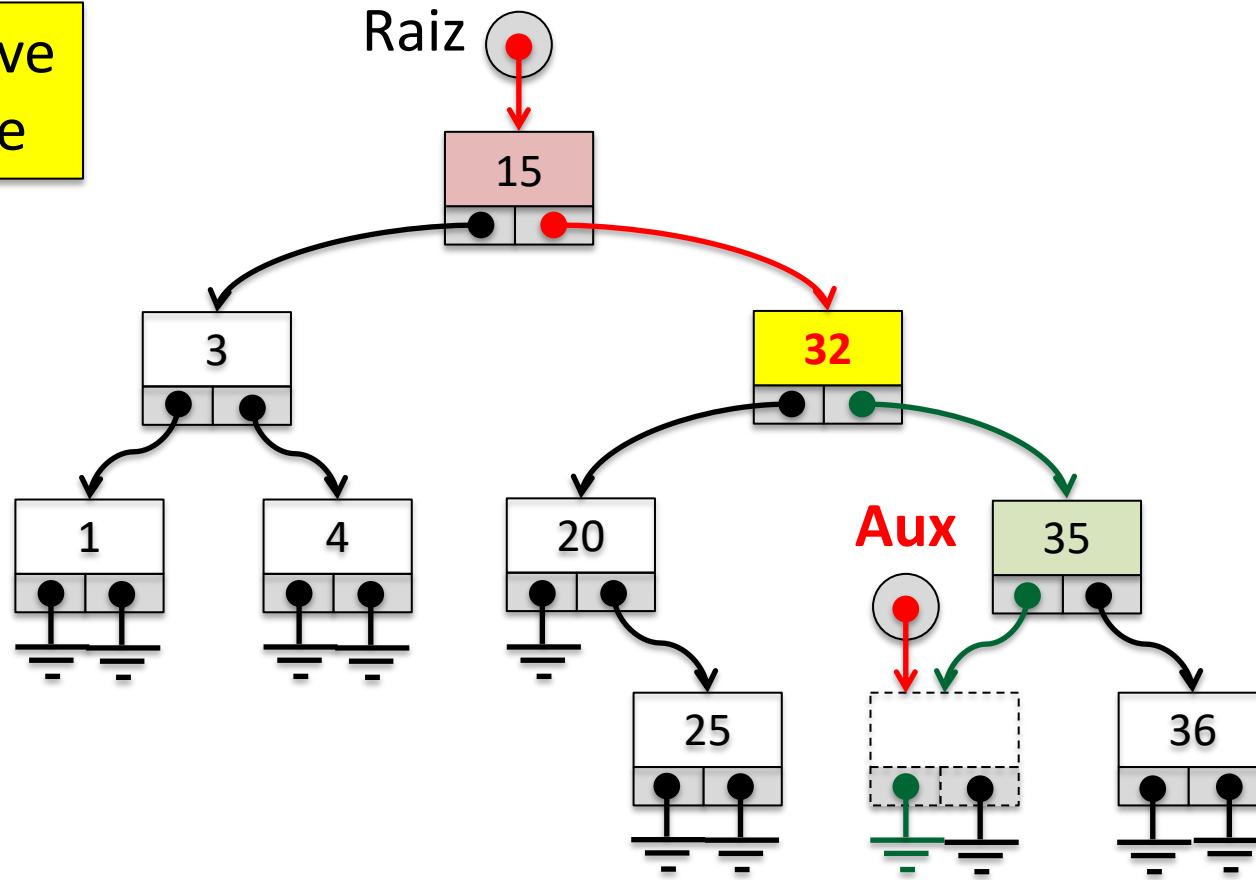
Árvores Binárias de Pesquisa

remover a chave
= 30 da árvore



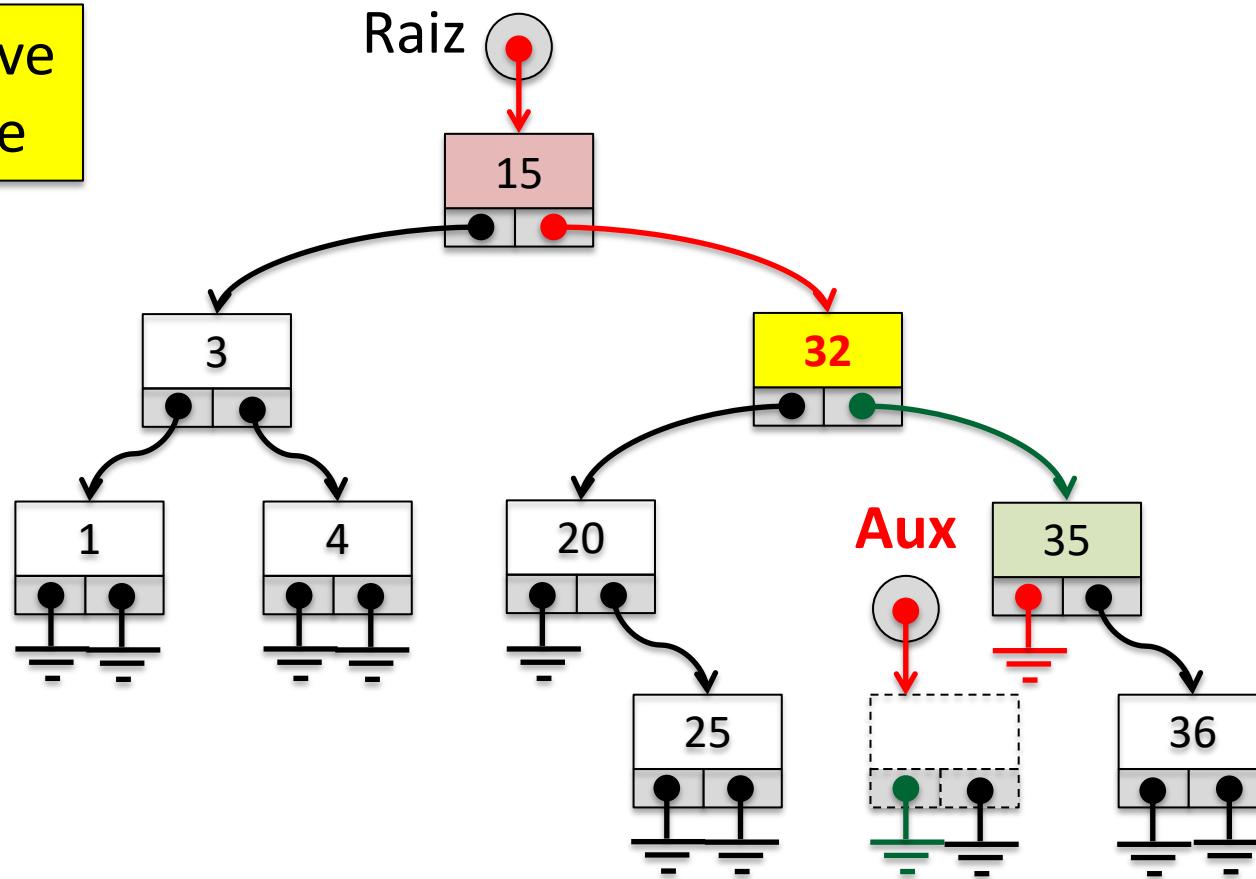
Árvores Binárias de Pesquisa

remover a chave
= 30 da árvore



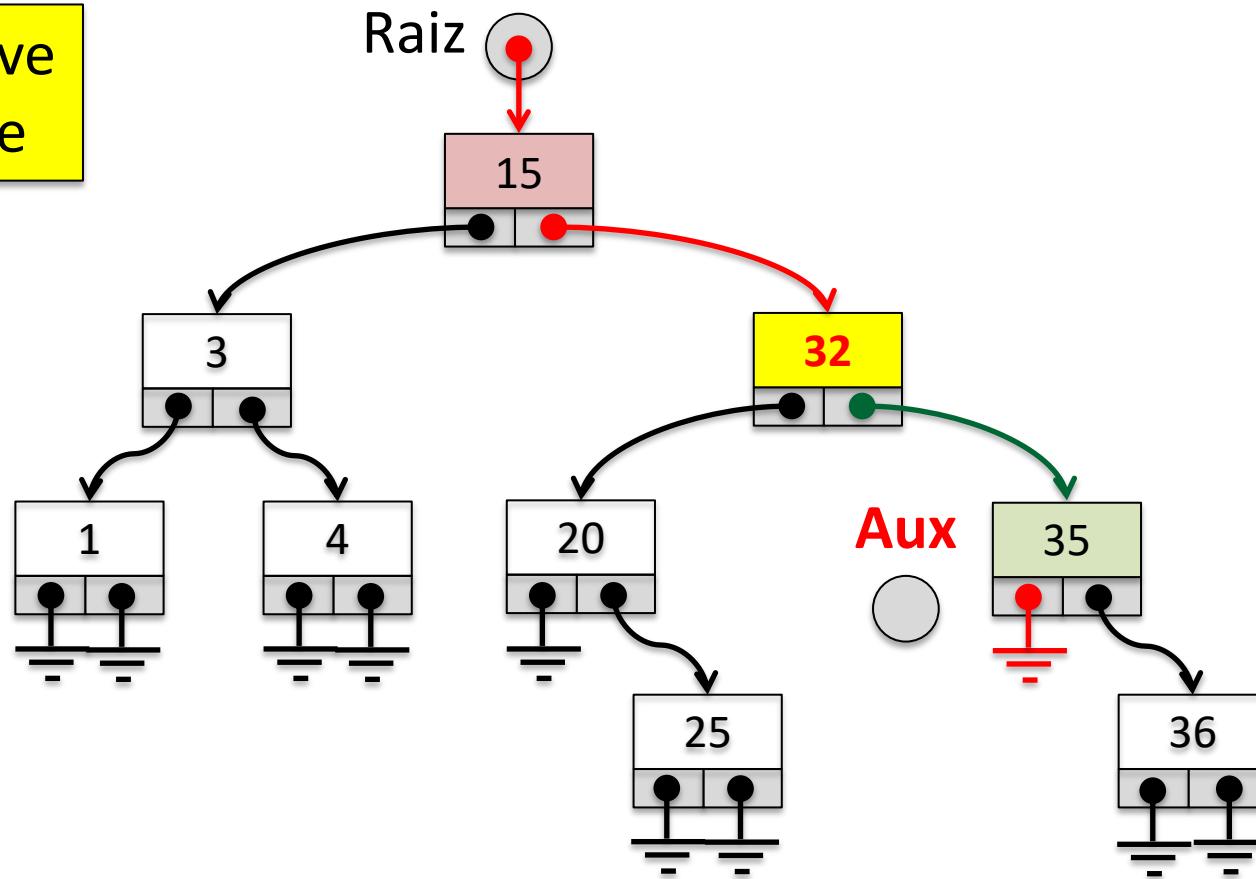
Árvores Binárias de Pesquisa

remover a chave
= 30 da árvore



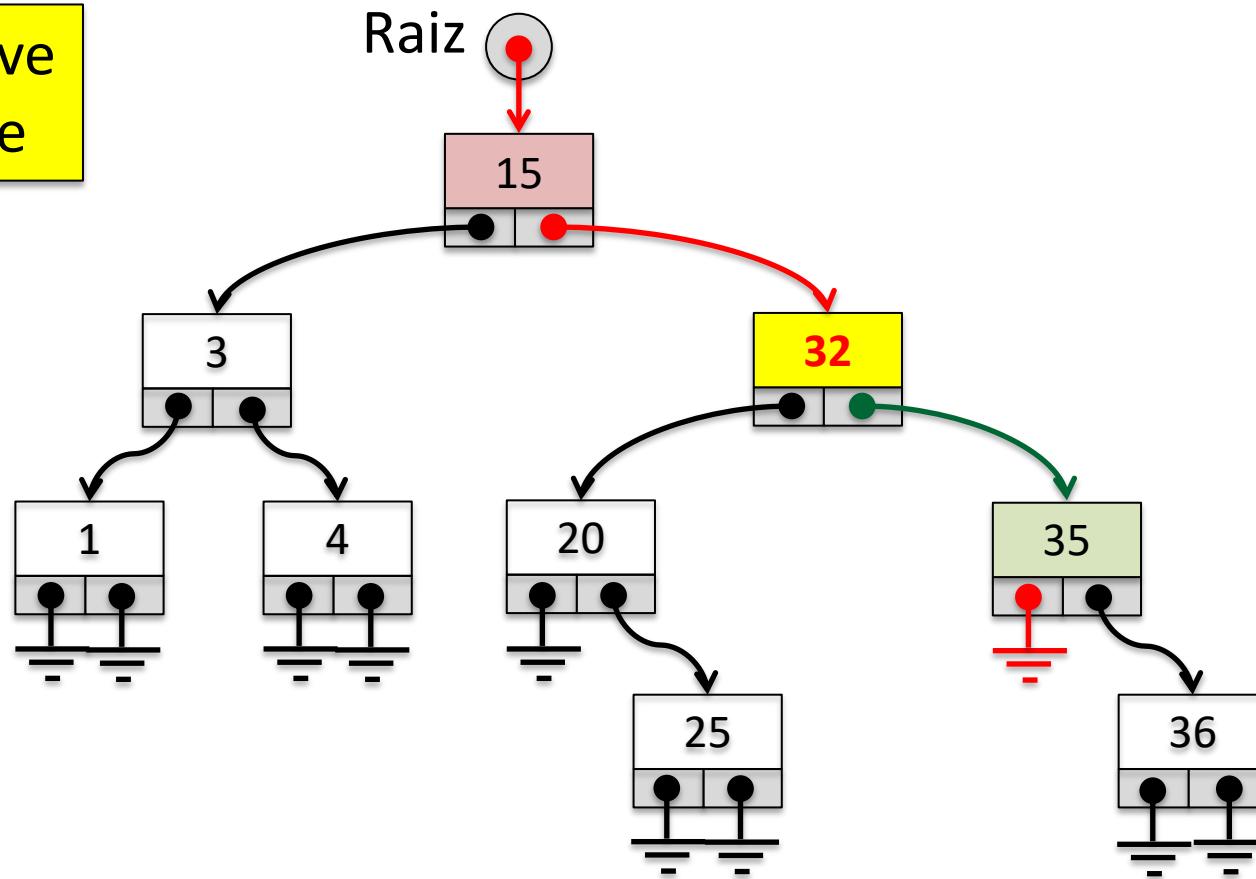
Árvores Binárias de Pesquisa

remover a chave
= 30 da árvore

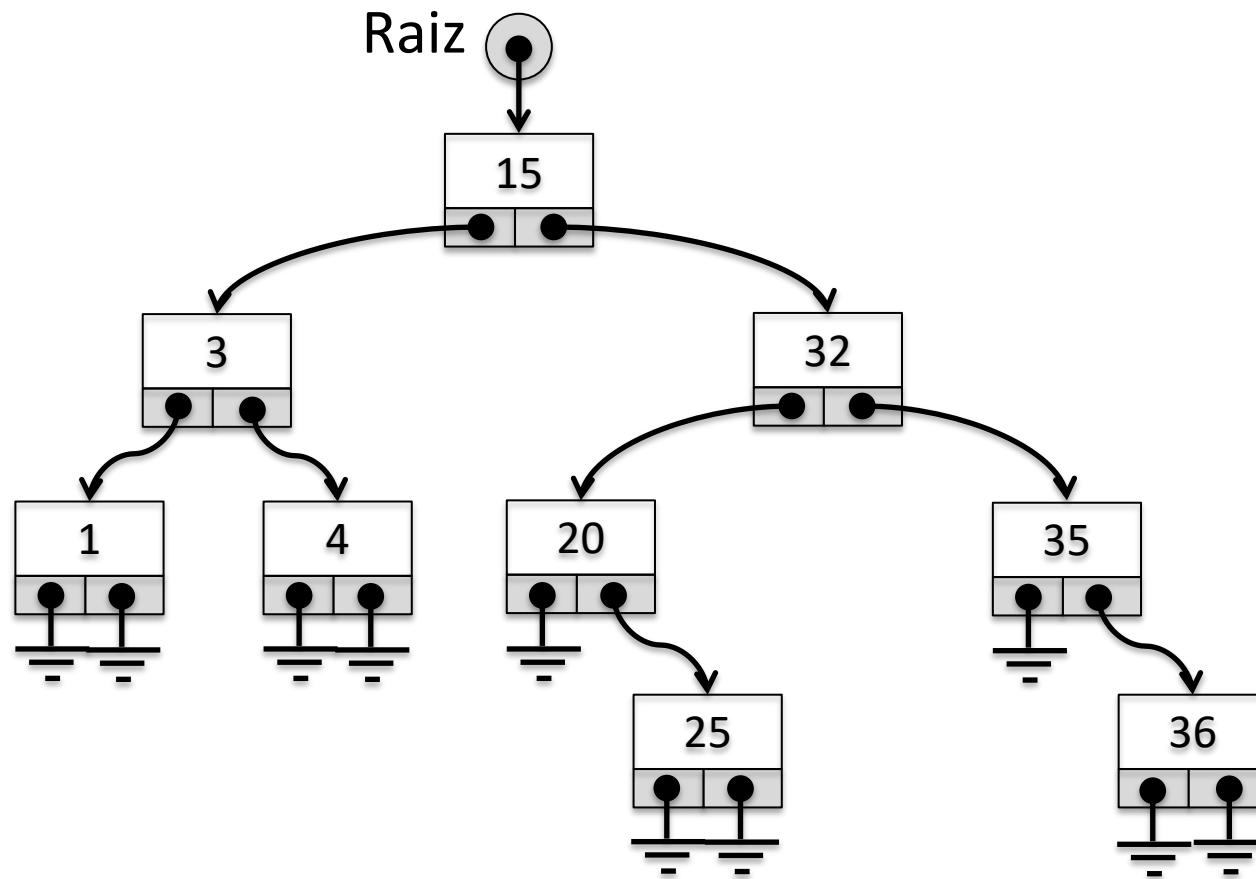


Árvores Binárias de Pesquisa

remover a chave
= 30 da árvore



Árvores Binárias de Pesquisa



Árvores Binárias de Pesquisa

```
int TABB_Remove(TABB *p, TChave x)
{
    TABB q;

    if (*p == NULL)
        return 0; // retorna 0 caso o item não esteja na arvore
    else if (x < (*p)->Item.Chave)
        return TABB_Remove(&(*p)->Esq, x);
    else if (x > (*p)->Item.Chave)
        return TABB_Remove(&(*p)->Dir, x);
    else {
        q = *p;
        if (q->Esq == NULL)
            *p = q->Dir;
        else if (q->Dir == NULL)
            *p = q->Esq;
        else // possui dois filhos
            TABB_Sucessor(&q, &q->Dir); // equivalente a Predecessor(&q, &q->Esq);
        free(q);
        return 1;
    }
}
```

Árvores Binárias de Pesquisa

```
int TABB_Remove(TABB *pRaiz, TChave x)
{
    TABB *p, q;

    p = pRaiz;
    while ((*p != NULL) && (x != (*p)->Item.Chave)) {
        if (x < (*p)->Item.Chave)
            p = &(*p)->Esq;
        else if (x > (*p)->Item.Chave)
            p = &(*p)->Dir;
    }
    if (*p != NULL) {
        q = *p;
        if (q->Esq == NULL)
            *p = q->Dir;
        else if (q->Dir == NULL)
            *p = q->Esq;
        else // possui dois filhos
            TABB_Sucessor(&q, &q->Dir); // equivalente a Predecessor(&q, &q->Esq);
        free(q);
        return 1;
    }
    return 0; // retorna 0 caso o item nao esteja na arvore
}
```

Árvores Binárias de Pesquisa

```
void TABB_Predecessor(TABB *q, TABB *r)
{
    if ((*r)->Dir != NULL)
        TABB_Predecessor(q, &(*r)->Dir);
    else {
        (*q)->Item = (*r)->Item;
        *q = *r;
        *r = (*r)->Esq;
    }
}

void TABB_Sucessor(TABB *q, TABB *r)
{
    if ((*r)->Esq != NULL)
        TABB_Sucessor(q, &(*r)->Esq);
    else {
        (*q)->Item = (*r)->Item;
        *q = *r;
        *r = (*r)->Dir;
    }
}
```

- O número de comparações em uma pesquisa com sucesso:
 - melhor caso : $C(n) = 1$
 - pior caso : $C(n) = n$
 - caso médio : $C(n) = \log_2 n$
- O tempo de execução dos algoritmos para árvores binárias de pesquisa dependem muito do formato das árvores, ou seja, se ela está **balanceada** ou não

- 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 $(n + 1)/2$.
- Para uma árvore de pesquisa aleatória, o número esperado de comparações para recuperar um registro qualquer é cerca de $1,39 \log_2 n$, apenas 39% pior que a árvore completamente balanceada

- Vantagens:

- No caso médio, o custo de pesquisa é da ordem de $\log n$
- Custo de inserção e remoção: $\log n$, para o caso médio
- Custo para obter os registros em ordem: n operações (melhor caso, pior caso e caso médio)

- Desvantagem:

- No pior caso, o custo tanto para a pesquisa quanto para a inserção e remoção é de n operações, i.e., $O(n)$

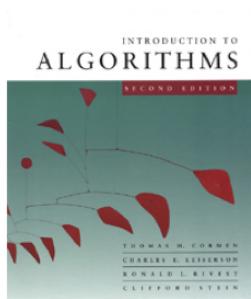
Exercício

- Desenhe a árvore binária de pesquisa obtida a partir da inserção das seguintes chaves:

K M L D F C A Y N E Z W B J

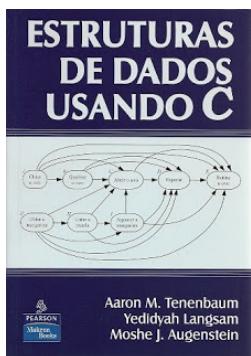
- Considere a árvore obtida e mostre como ela ficaria após a remoção das chaves abaixo, na seguinte ordem:

J A M



CORMEN, T. H.; LEISERSON, C. E.; RIVEST, R. L.; STEIN, C. **Introduction to Algorithms**. 3^a Edição. MIT Press, 2009. **Seções 12.1 a 12.3**

ZIVIANI, N. **Projeto de Algoritmos com Implementações em Pascal e C**. 3^a Edição. Cengage Learning, 2010. **Seção 5.3**



TENENBAUM, A. M.; LANGSAM, Y.; AUGENSTEIN, M. J. **Estruturas de Dados usando C**. Pearson Makron Books, 2008.
Capítulo 5