

ÁRVORES BINÁRIAS DE BUSCA

Prof. Joaquim Uchôa
Profa. Juliana Greggi
Prof. Renato Ramos



- Visão geral
- Principais métodos
- Implementação da ABB
- Remoção de elementos

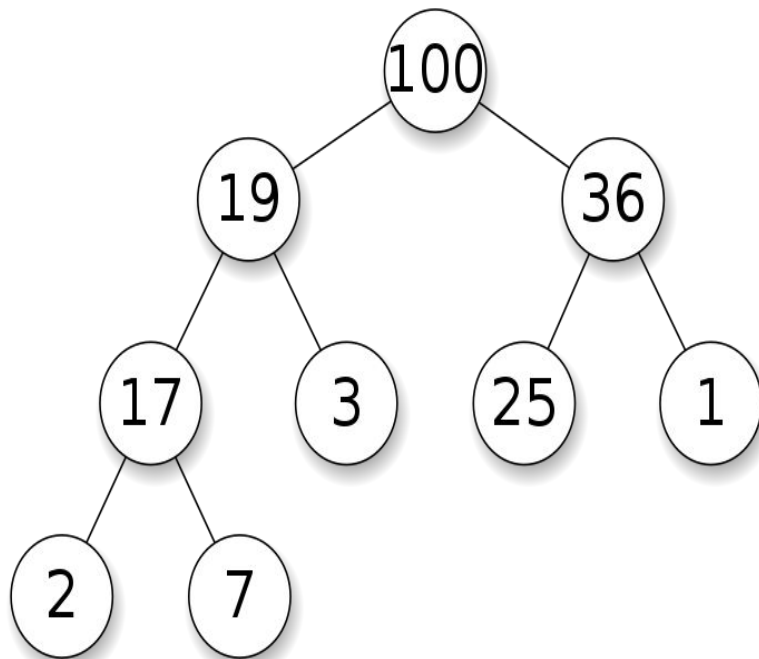
VISÃO GERAL



RELEMBRANDO CONCEITOS...

Uma árvore é um grafo conexo e sem ciclos.

Em uma árvore binária, cada nó possui zero, um ou dois filhos.



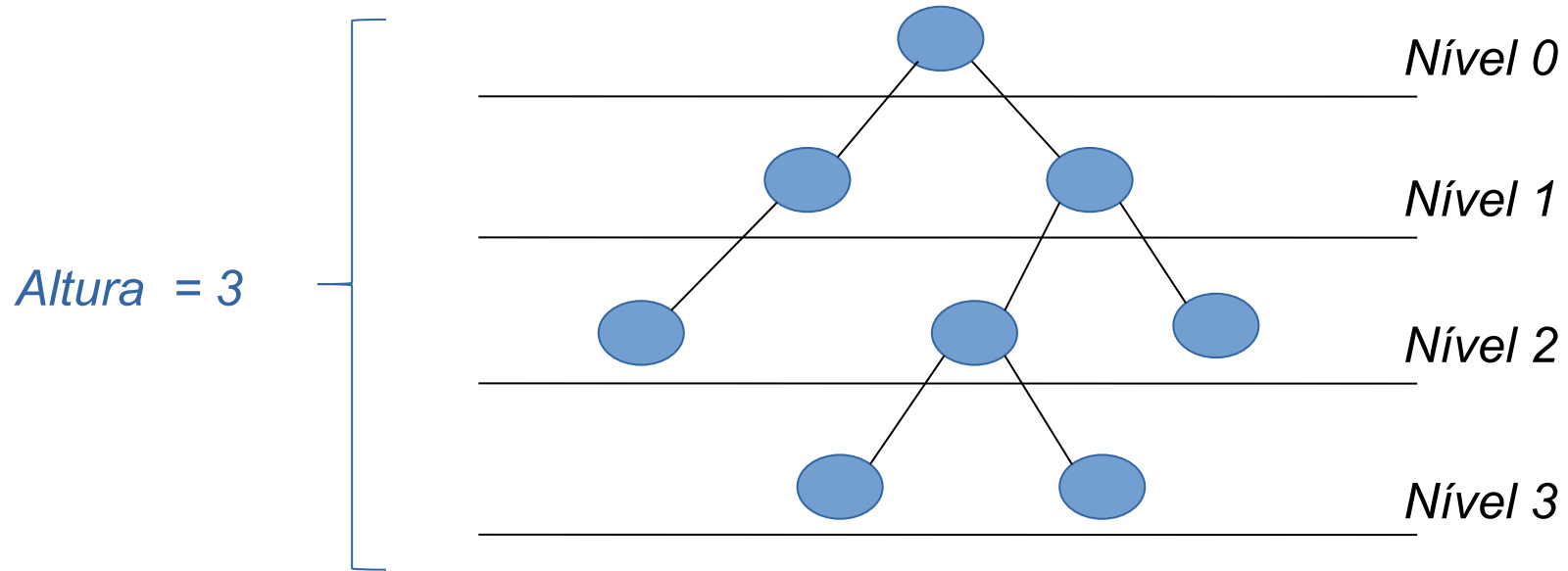
ÁRVORES BINÁRIAS

Em uma árvore binária, cada nó tem, no máximo, dois filhos. Assim, uma árvore binária pode ser definida como:

- Uma árvore vazia (ponteiro para a raiz nulo);
- Um nó raiz contendo duas subárvores, identificadas como subárvore direita (SD) e subárvore esquerda (SE).

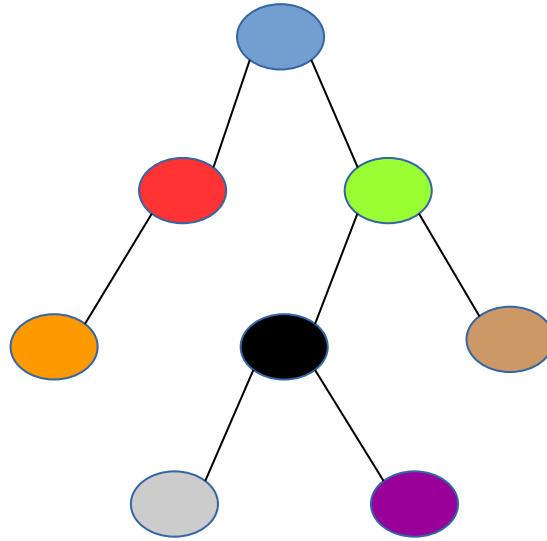
RELEMBRANDO CONCEITOS...

Níveis e altura de uma árvore



























PERCURSO EM ÁRVORES BINÁRIAS

- Pré-ordem: trata a raiz, a percorre a SE, percorre a SD;
- Em-ordem: percorre a SE, trata a raiz, percorre a SD;
- Pós-ordem: percorre a SE, percorre a SD e trata a raiz.



Sequência de visitação dos nós:

Pré-ordem								
Em-ordem								
Pós-ordem								

DIREÇÃO DO PERCORRIMENTO

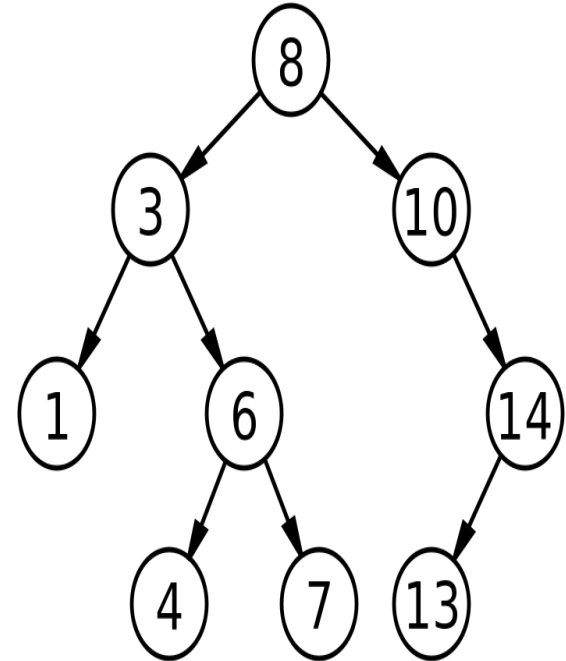
O percurso tradicional é feito geralmente da esquerda para a direita.

Para resolver alguns problemas, entretanto, pode ser necessário implementar o percurso à direita, em que nós à direita são visitados antes que nós à esquerda.

Nestes slides iremos abordar apenas o percurso tradicional, à esquerda.

É A TAL ÁRVORE BINÁRIA DE BUSCA?

Em uma árvore binária de busca (ABB), para cada nó, elementos maiores estão à sua direita e elementos menores à sua esquerda.



ÁRVORE BINÁRIA DE BUSCA (1/2)

Elementos da SE são menores que o nó raiz e os elementos da SD são maiores que o nó raiz. As subárvores são organizadas seguindo a mesma regra.

Árvores precisam estar balanceadas para que a busca seja eficiente: SE e SD devem ter o mesmo número de nós ou próximo disso;

ÁRVORE BINÁRIA DE BUSCA (2/2)

Otimização da busca de valores ordenados:

- Uma árvore com altura h pode ter, no máximo, $2^{h+1} - 1$ nós $\rightarrow O(2^h)$;
- Uma árvore com n nós tem altura mínima de $O(\log n)$.



Podemos alcançar qualquer um dos n nós em $O(\log n)$ passos em uma árvore balanceada.

PRINCIPAIS MÉTODOS



OPERAÇÕES BÁSICAS EM ÁRVORES BINÁRIAS

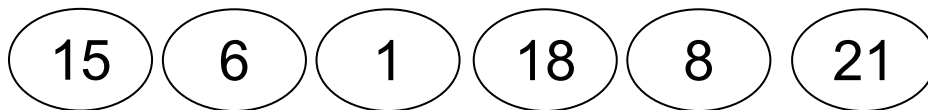
- Inserção
- Remoção
- Busca

Também podem ser implementados:

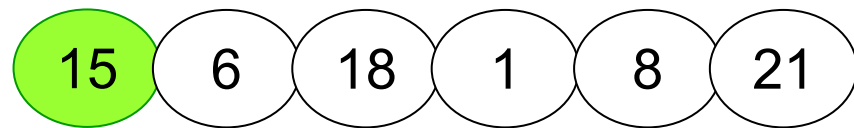
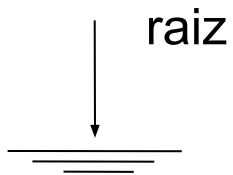
- Mínimo
- Máximo
- Percorrimento

EXEMPLO - INSERÇÃO

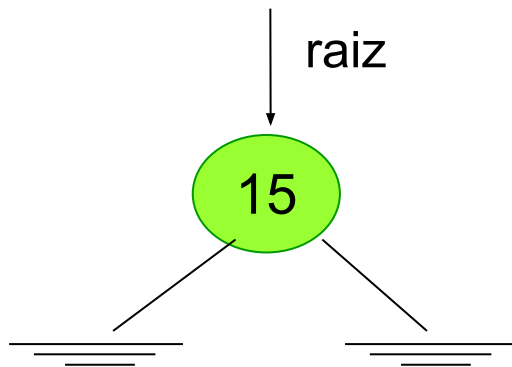
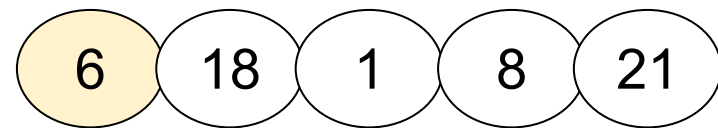
Considere o seguinte conjunto de nós a serem inseridos:



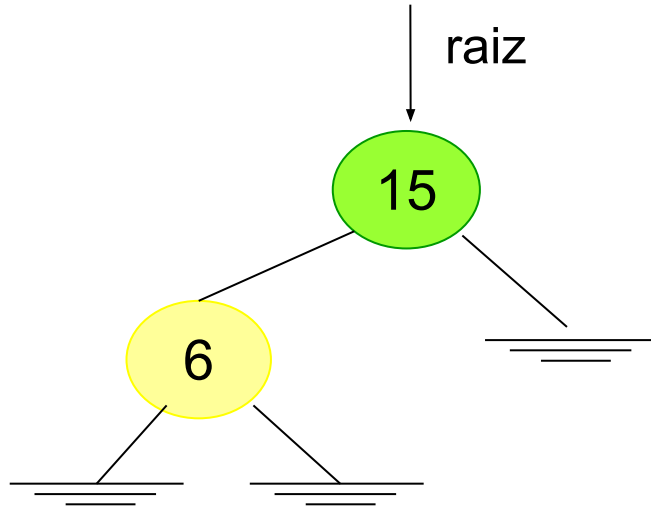
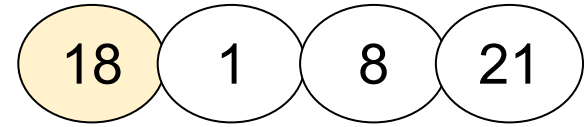
Inserção



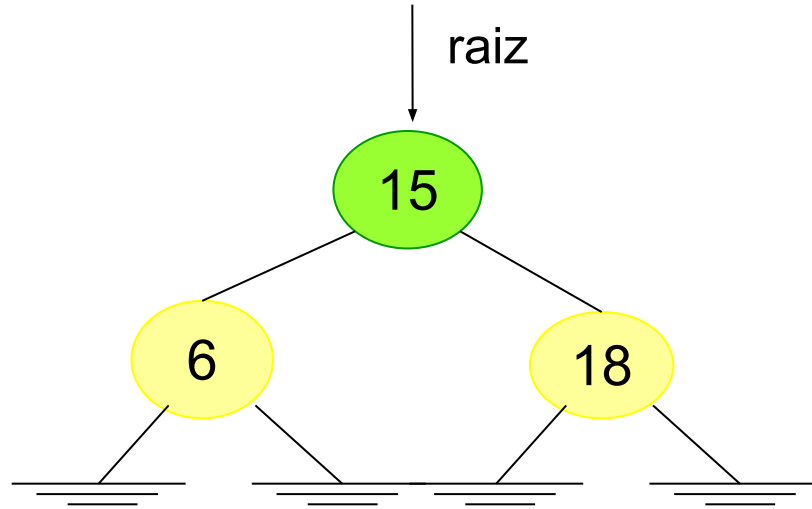
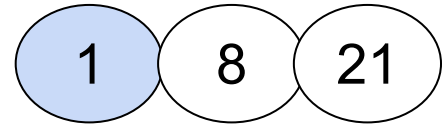
Inserção



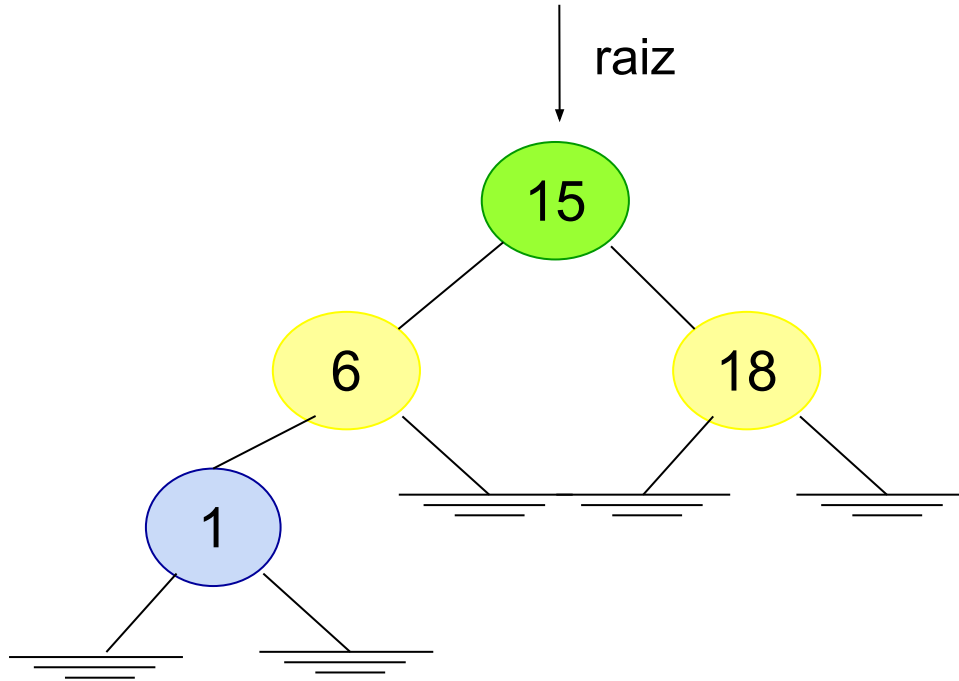
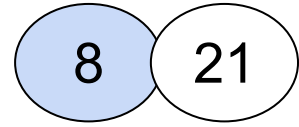
Inserção



Inserção

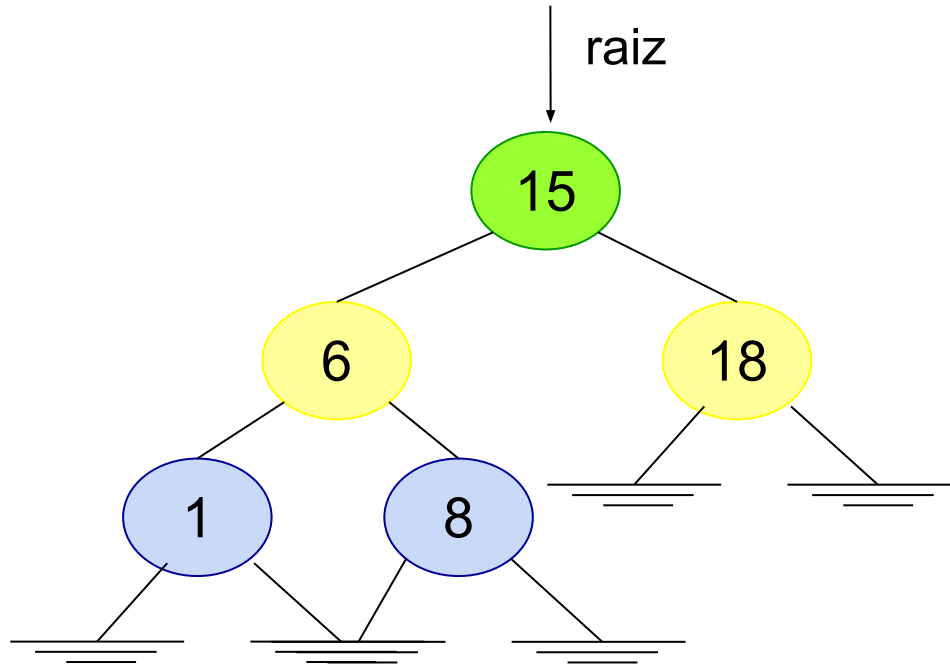


Inserção

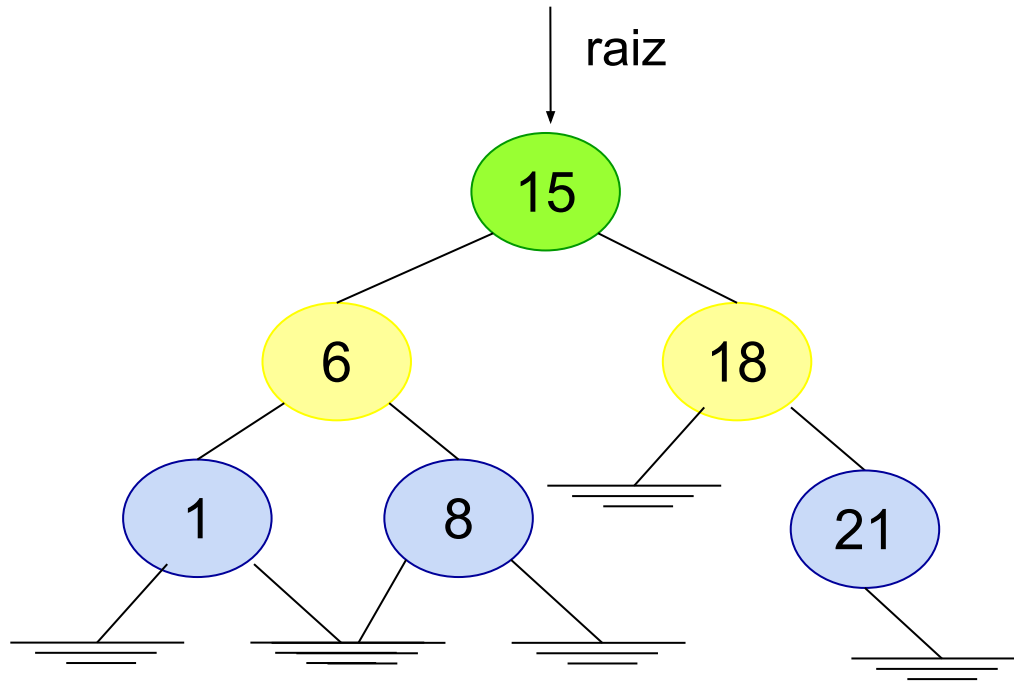


Inserção

21

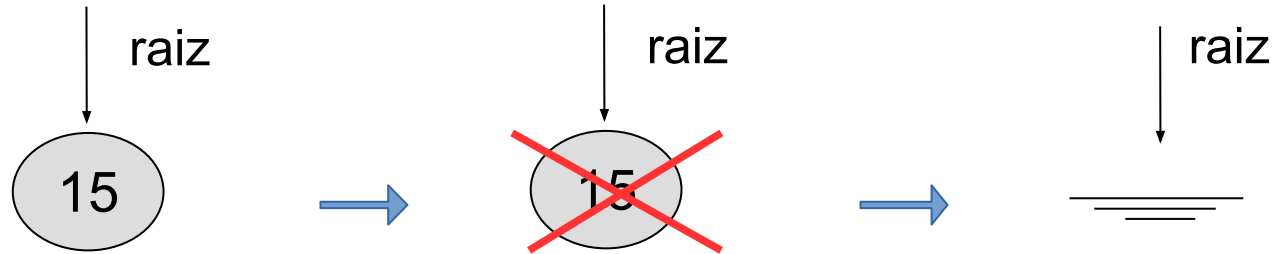


Inserção



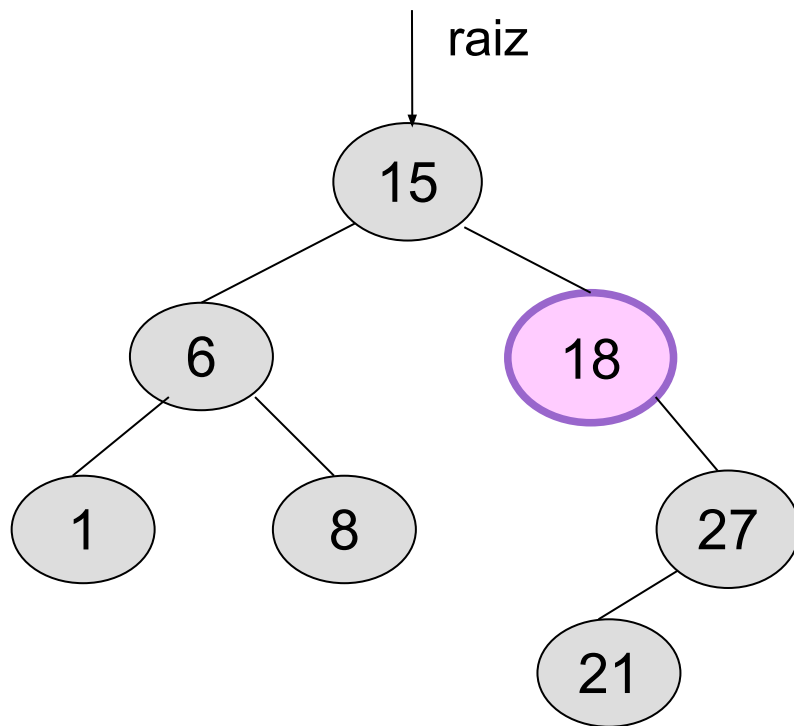
Remoção

1º caso - Remoção de item único
(raiz de subárvore)



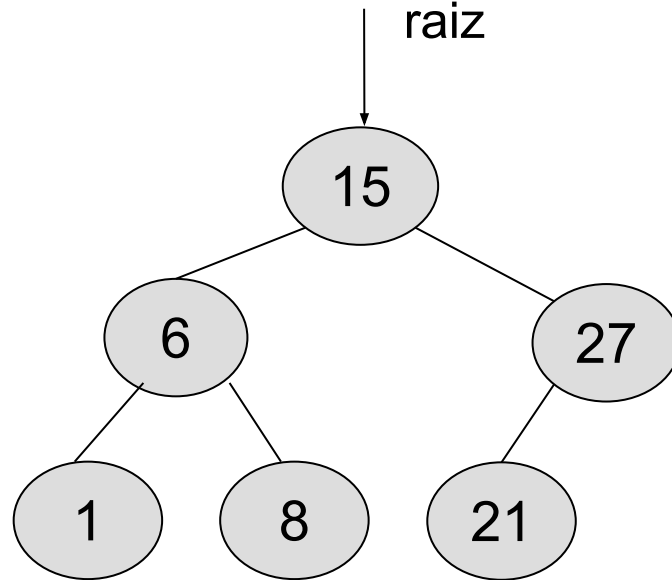
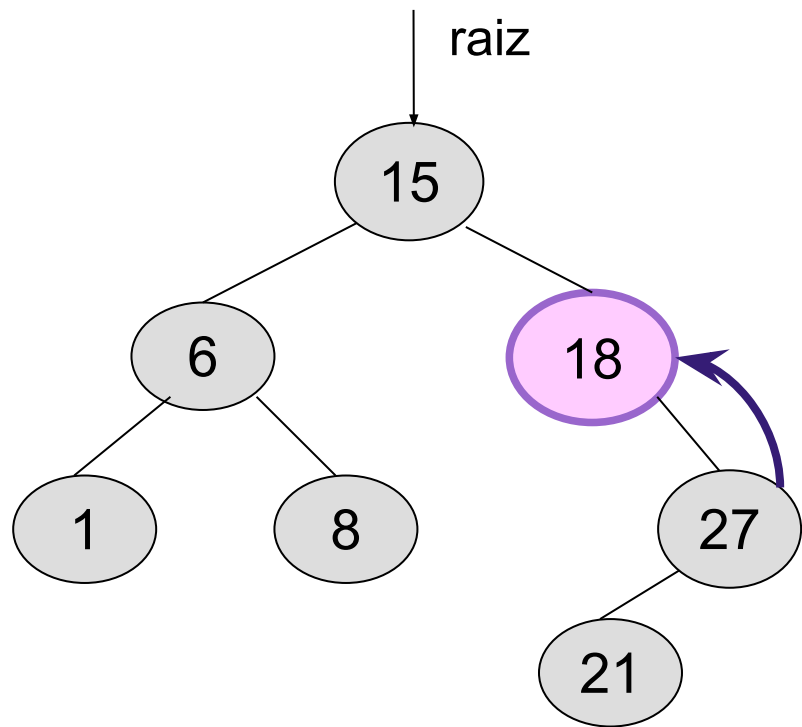
Remoção

2º caso – Remoção de subárvore com apenas um filho.



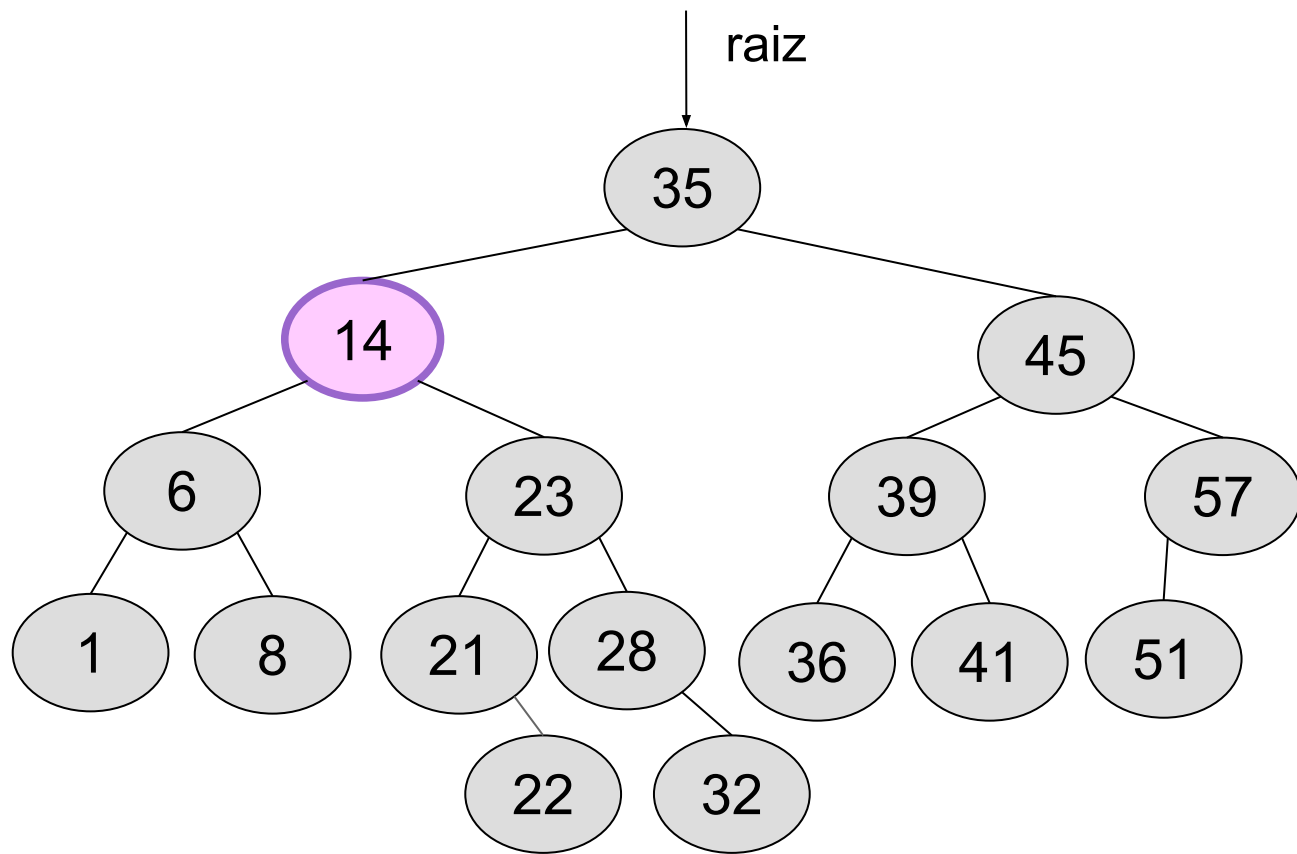
Remoção

2º caso – Remoção de subárvore com apenas um filho.



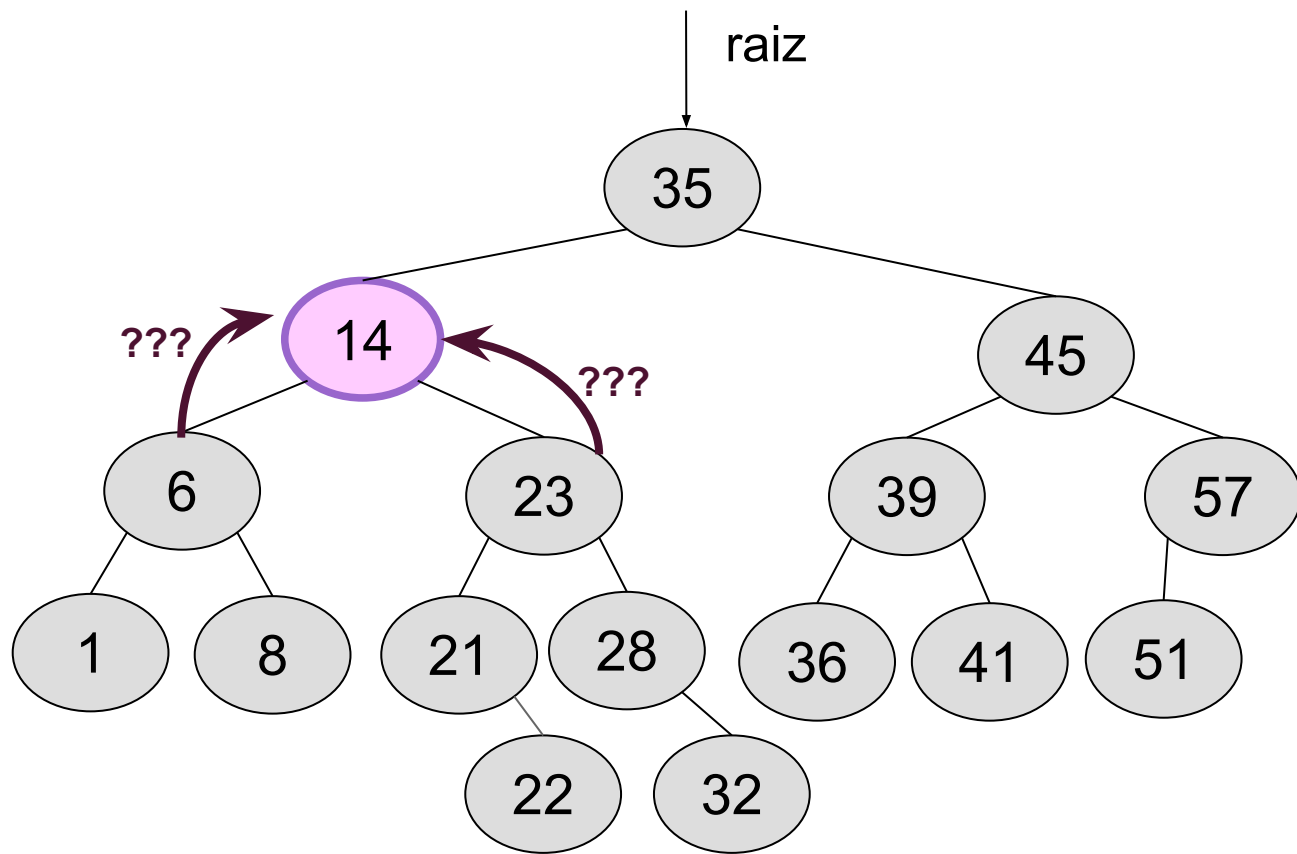
Remoção

3º caso – Remoção de subárvore com dois filhos (remover o 14).



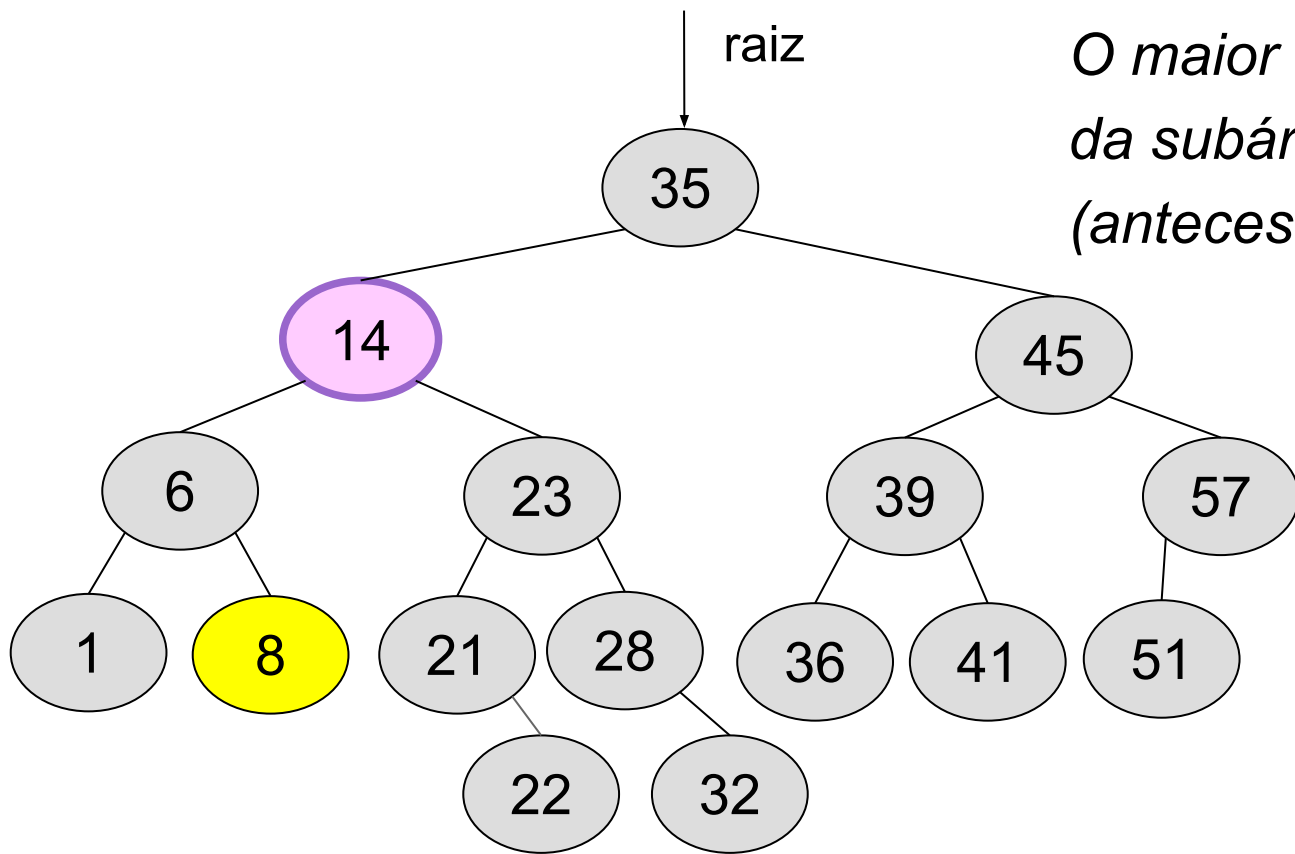
Remoção

3º caso – Remoção de subárvore com dois filhos (remover o 14).



Remoção

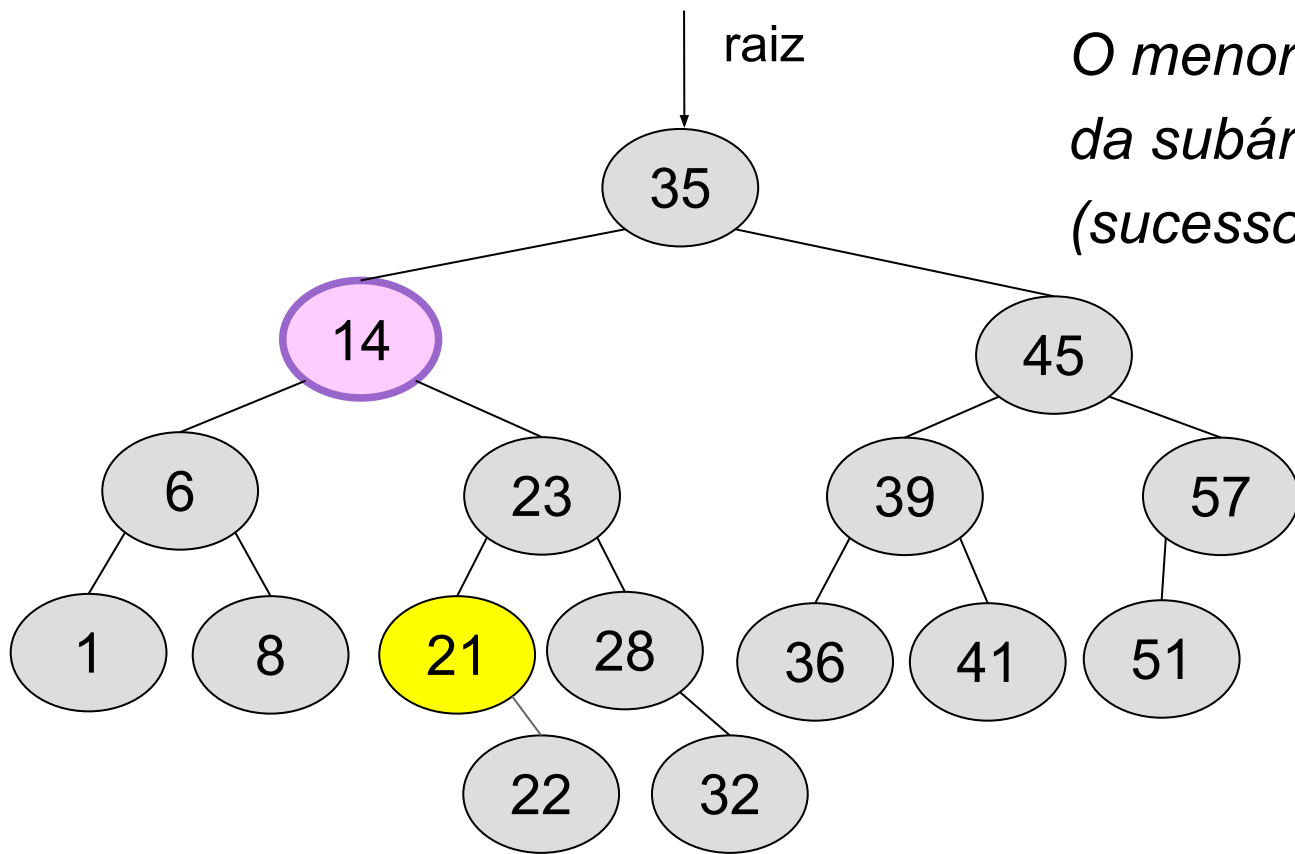
3º caso – Remoção de subárvore com dois filhos (remover o 14).



O maior elemento da subárvore esquerda (antecessor)

Remoção

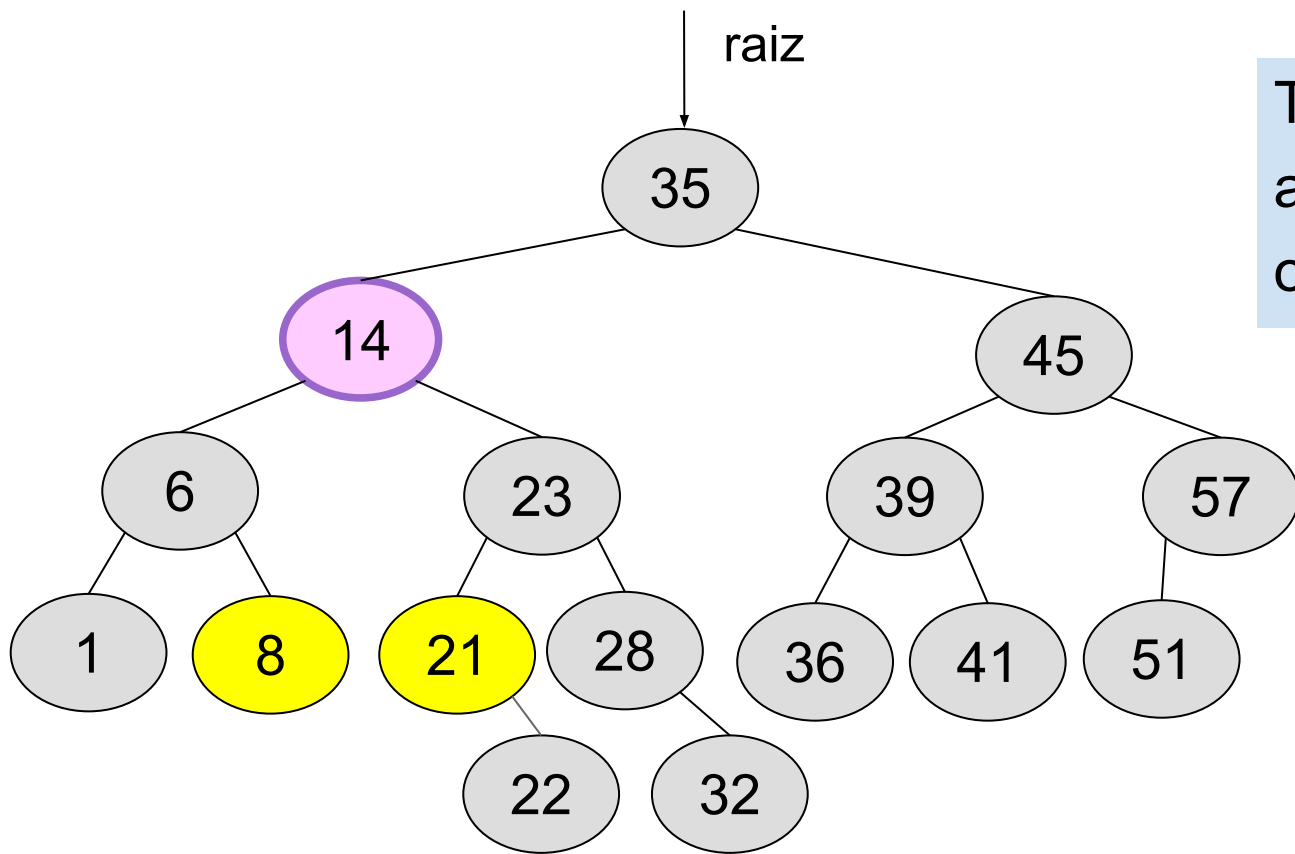
3º caso – Remoção de subárvore com dois filhos (remover o 14).



O menor elemento da subárvore direita (sucessor)

Remoção

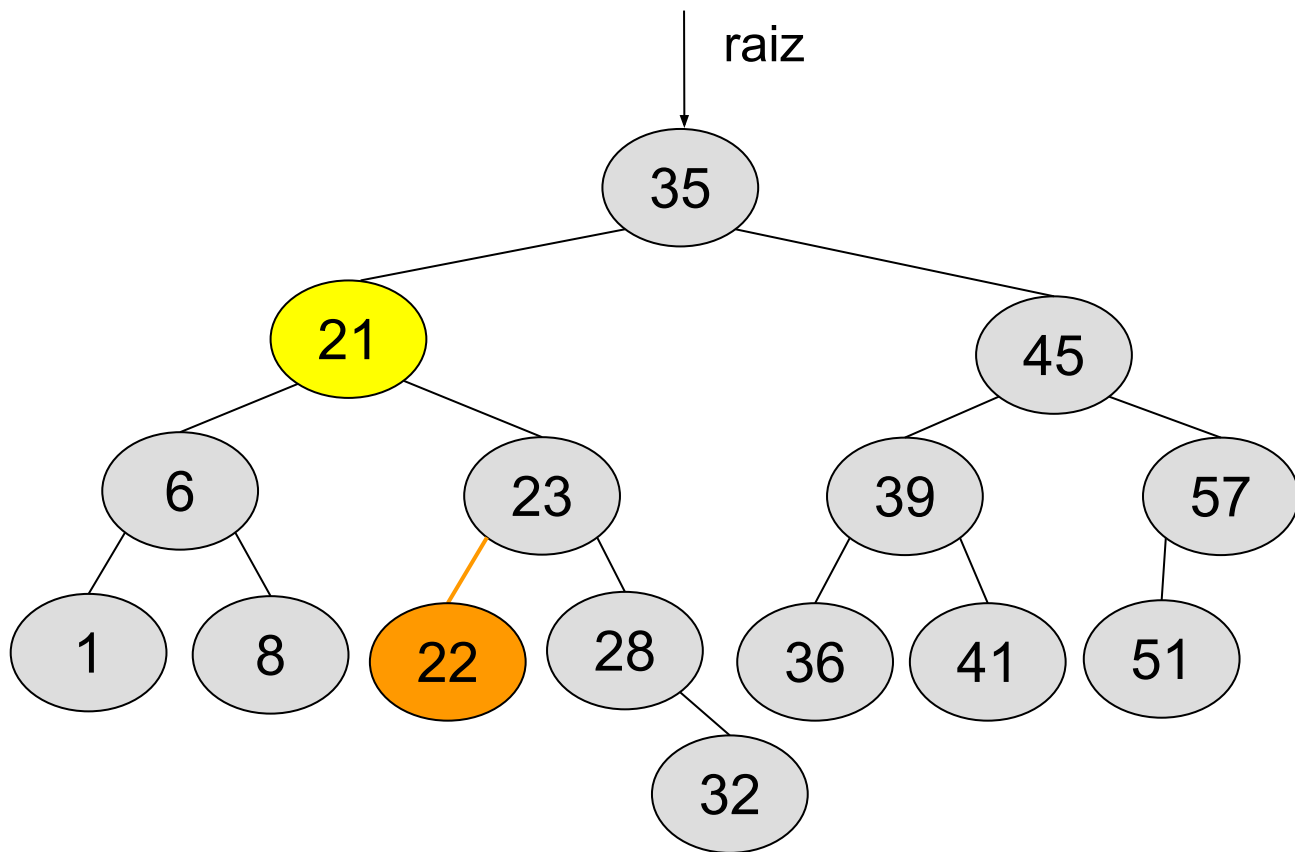
3º caso – Remoção de subárvore com dois filhos (remover o 14).



Troca nó com
antecessor
ou sucessor

Remoção

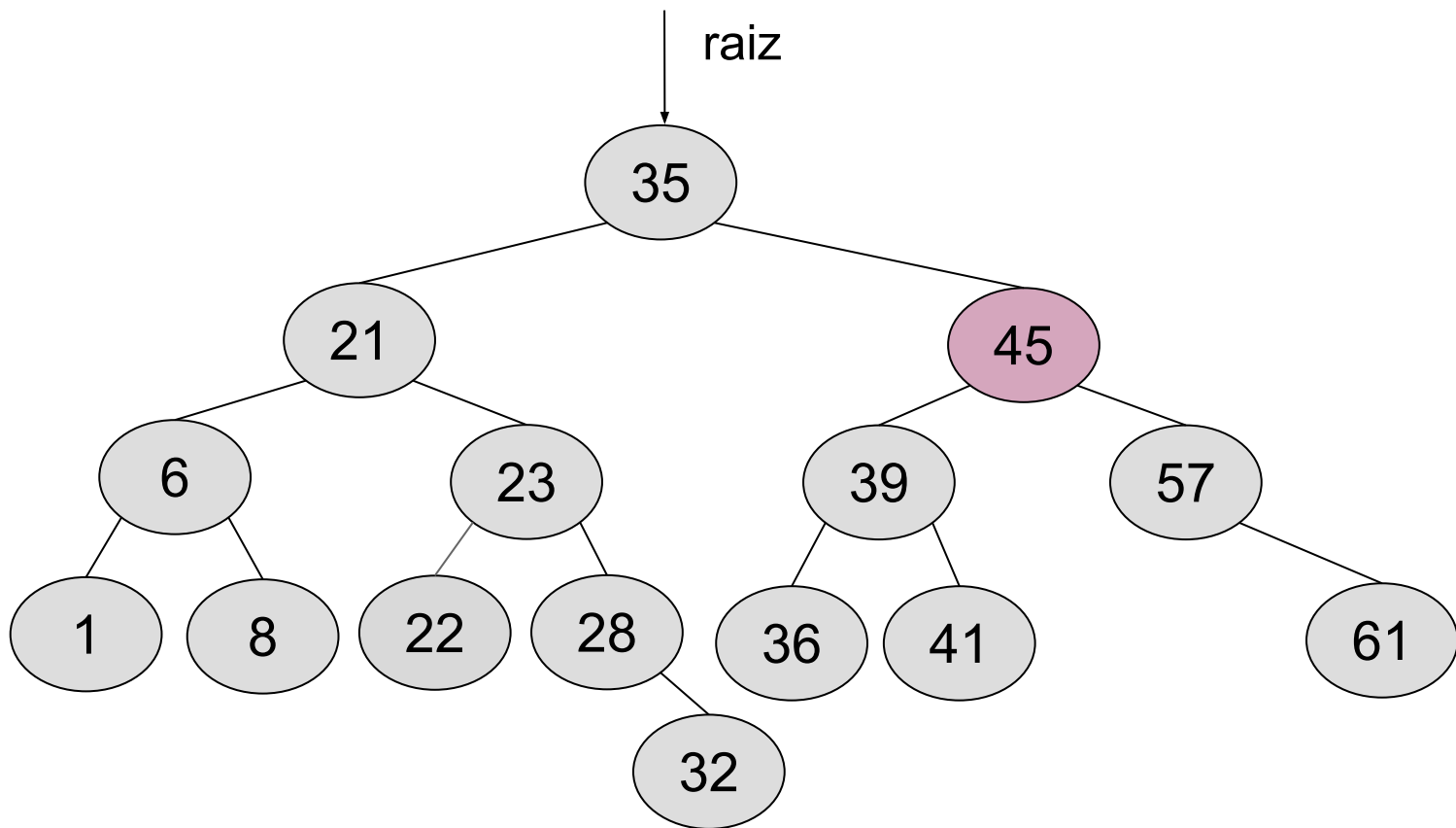
3º caso – Remoção de subárvore com dois filhos (remover o 14).



Houve
necessidade
de ajustar
filho à
esquerda do
sucessor

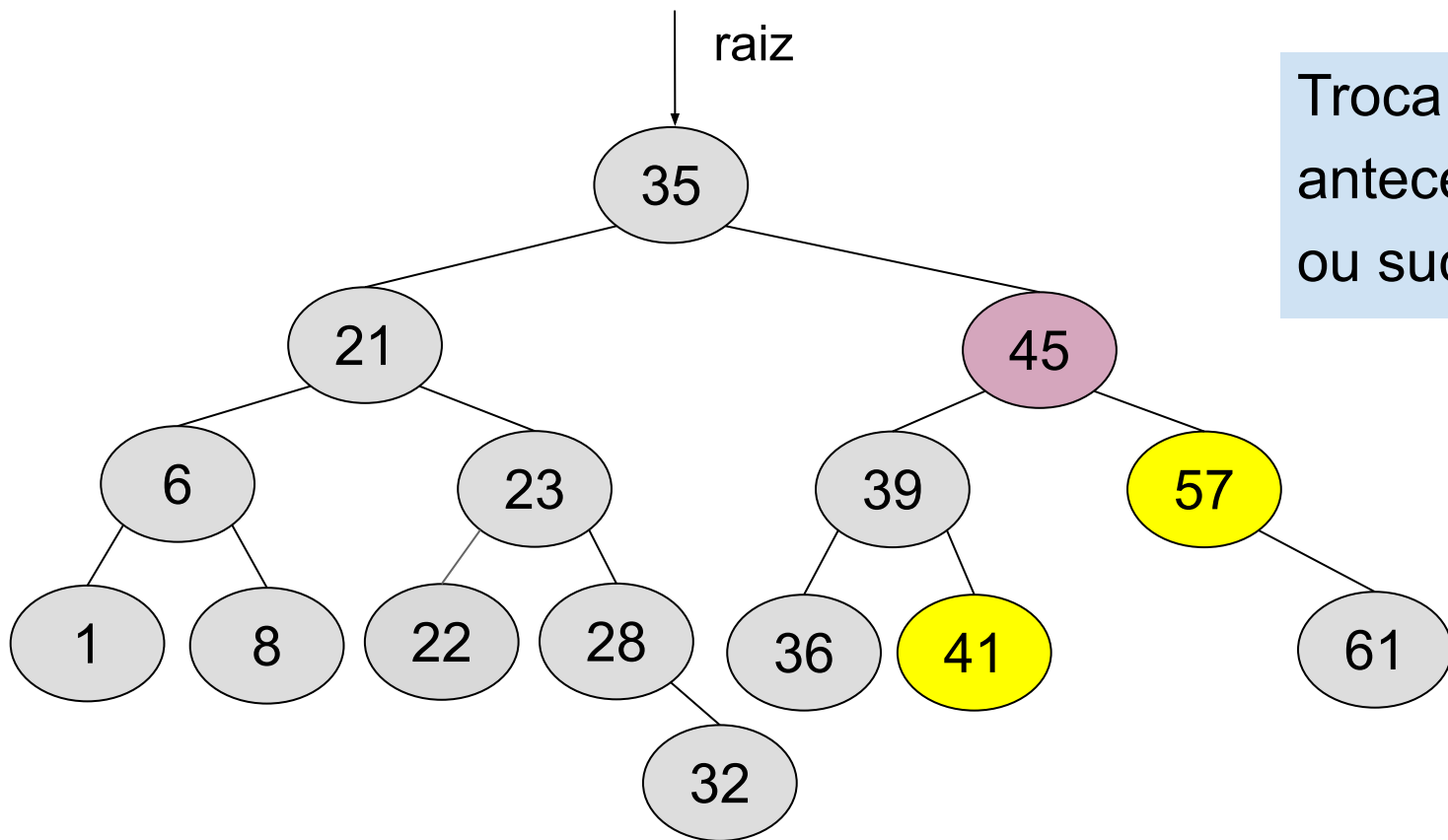
Remoção

3º caso – Remoção de subárvore com dois filhos (remover o 45).



Remoção

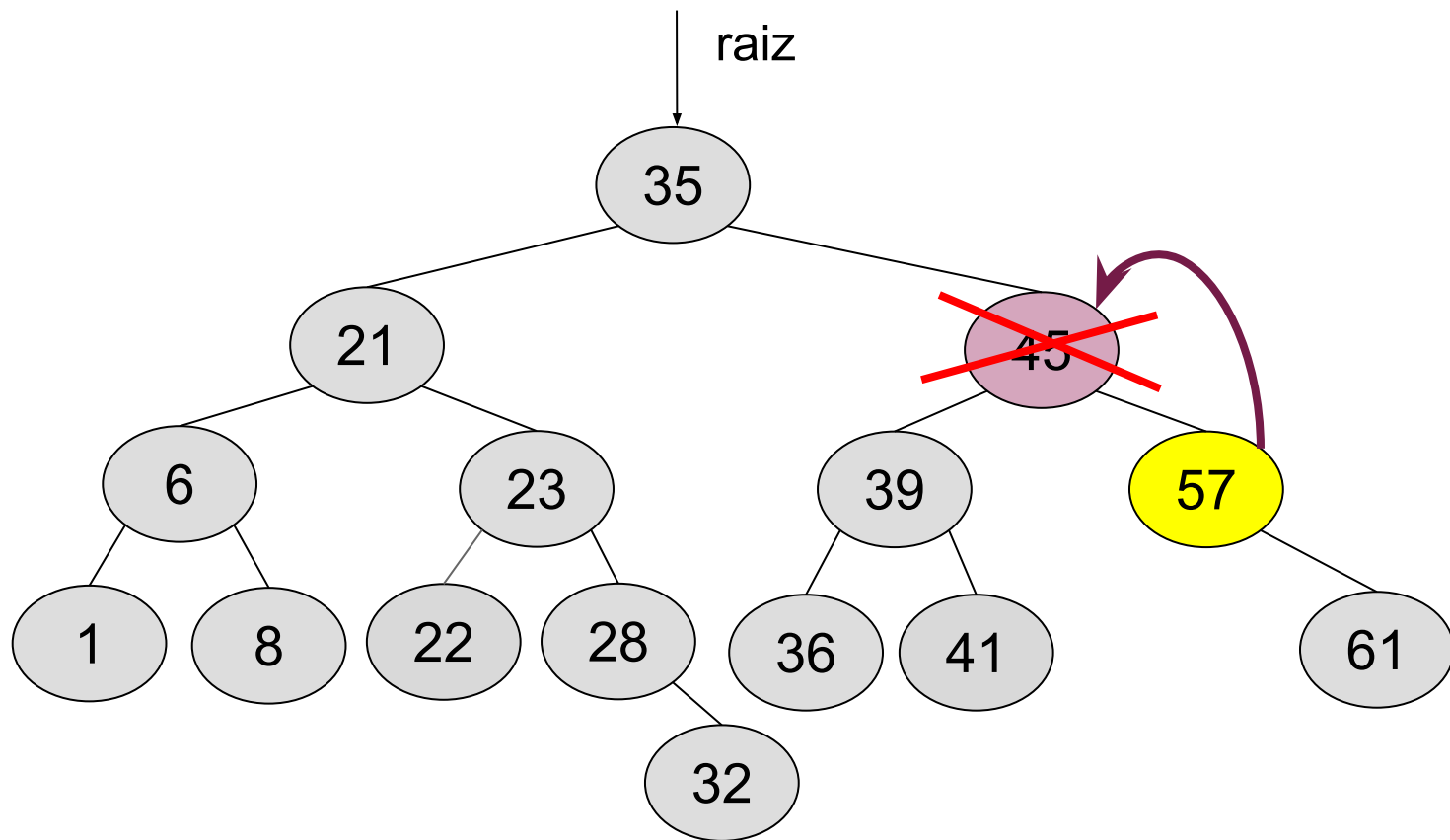
3º caso – Remoção de subárvore com dois filhos (remover o 45).



Troca nó com
antecessor
ou sucessor

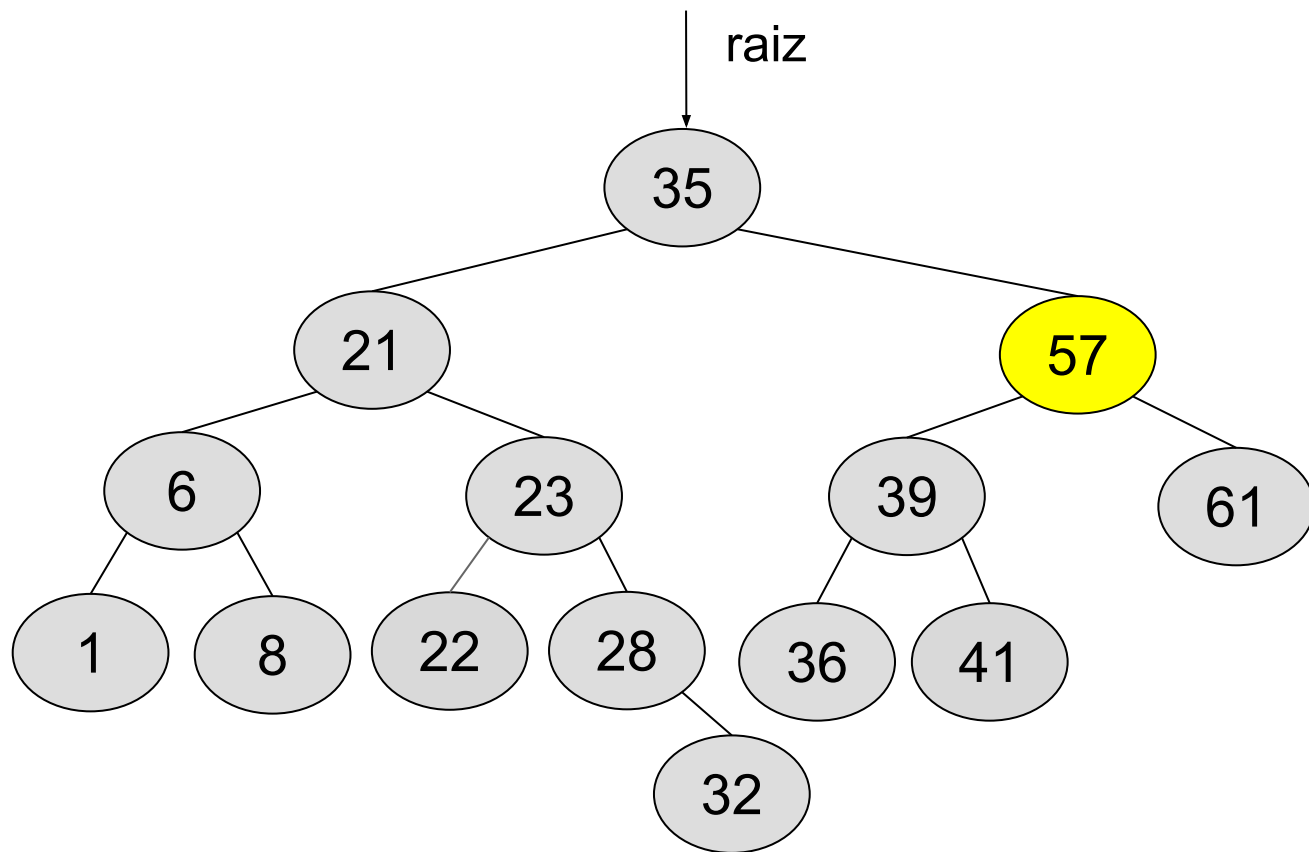
Remoção

3º caso – Remoção de subárvore com dois filhos (remover o 45).



Remoção

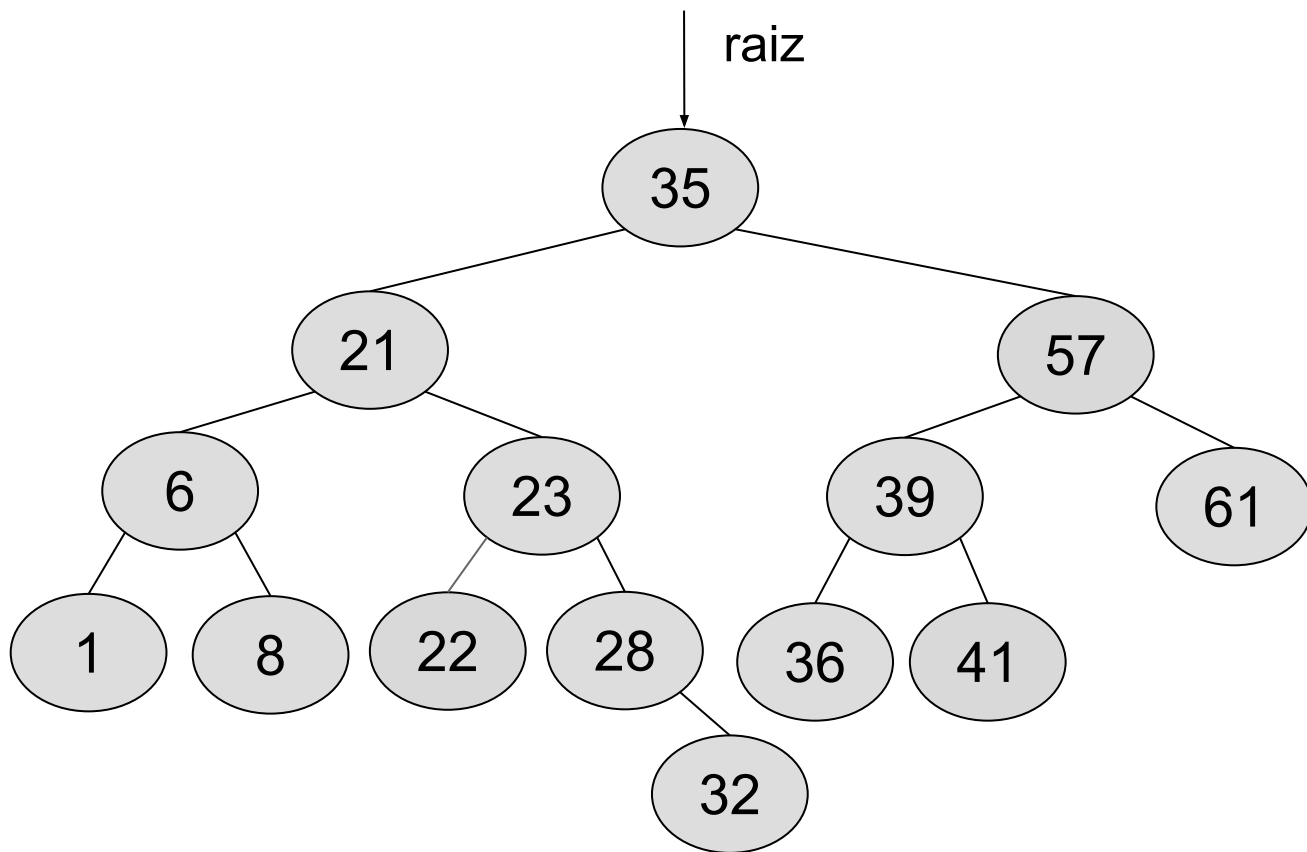
3º caso – Remoção de subárvore com dois filhos (remover o 45).



Busca

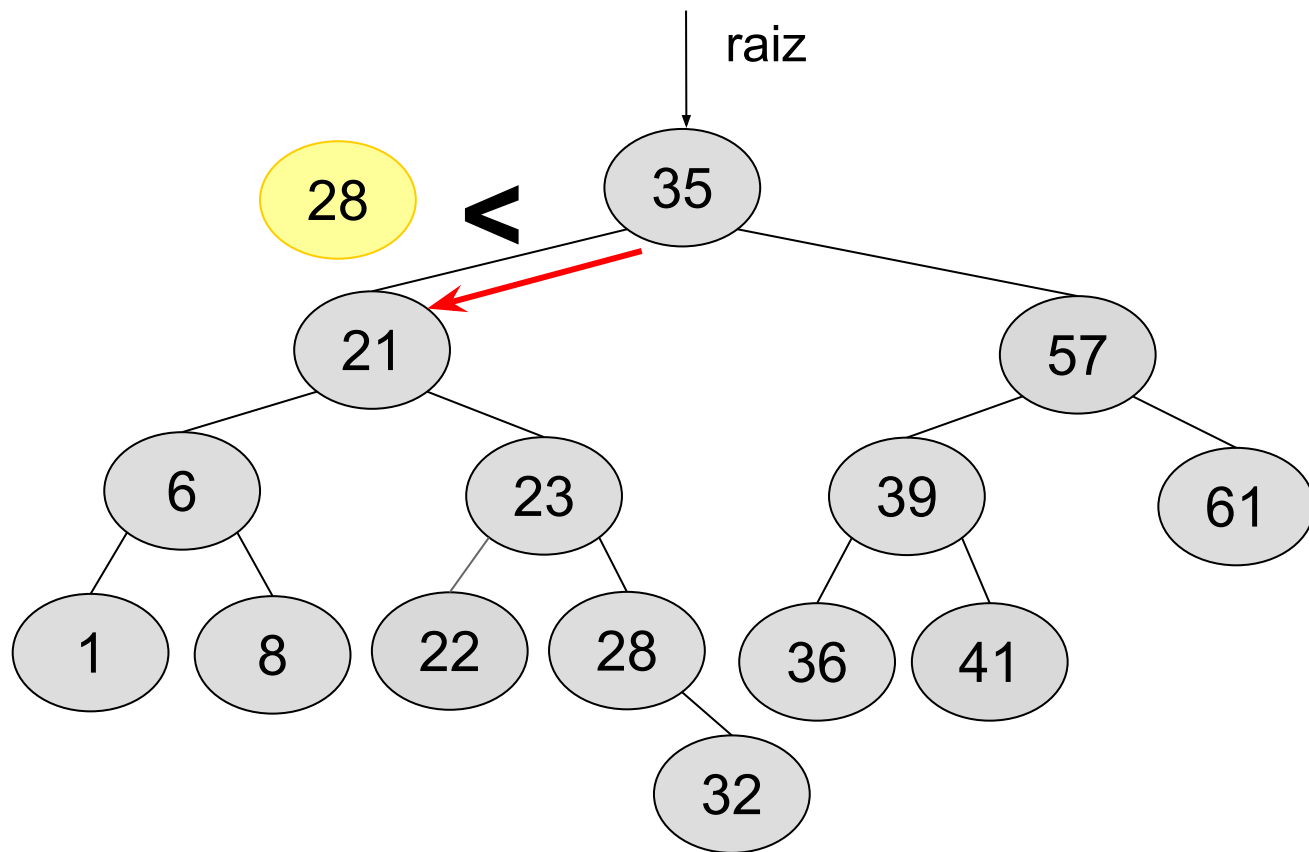
28

60



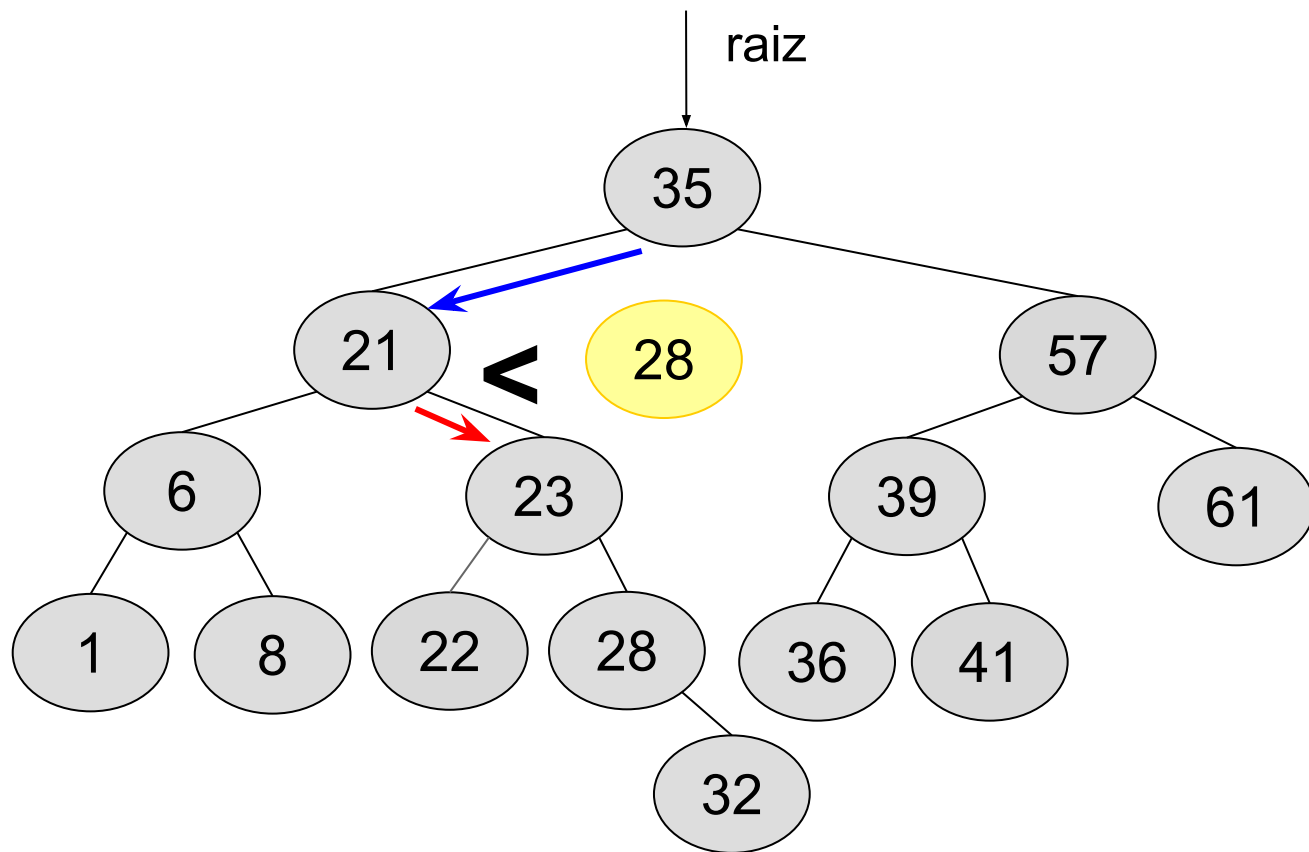
Busca

60



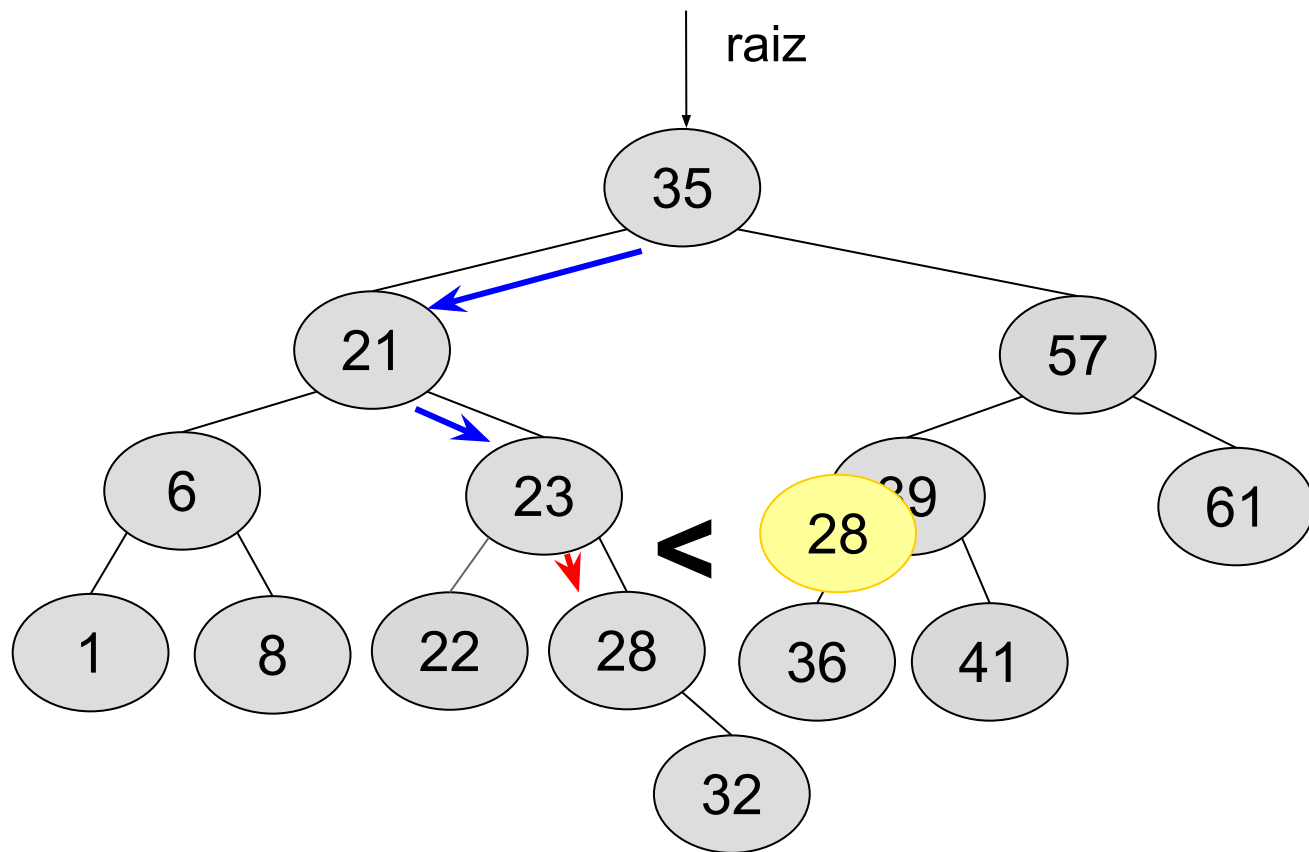
Busca

60



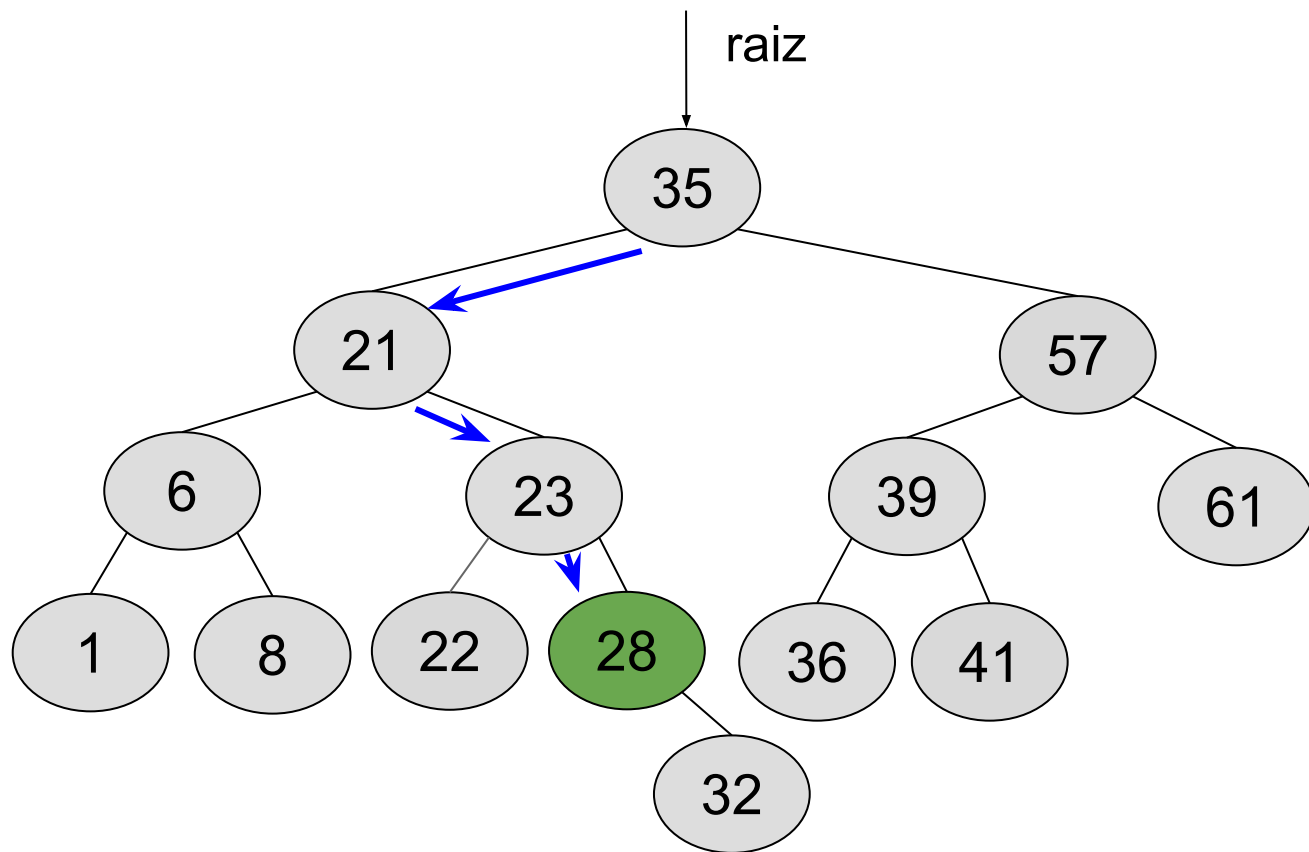
Busca

60

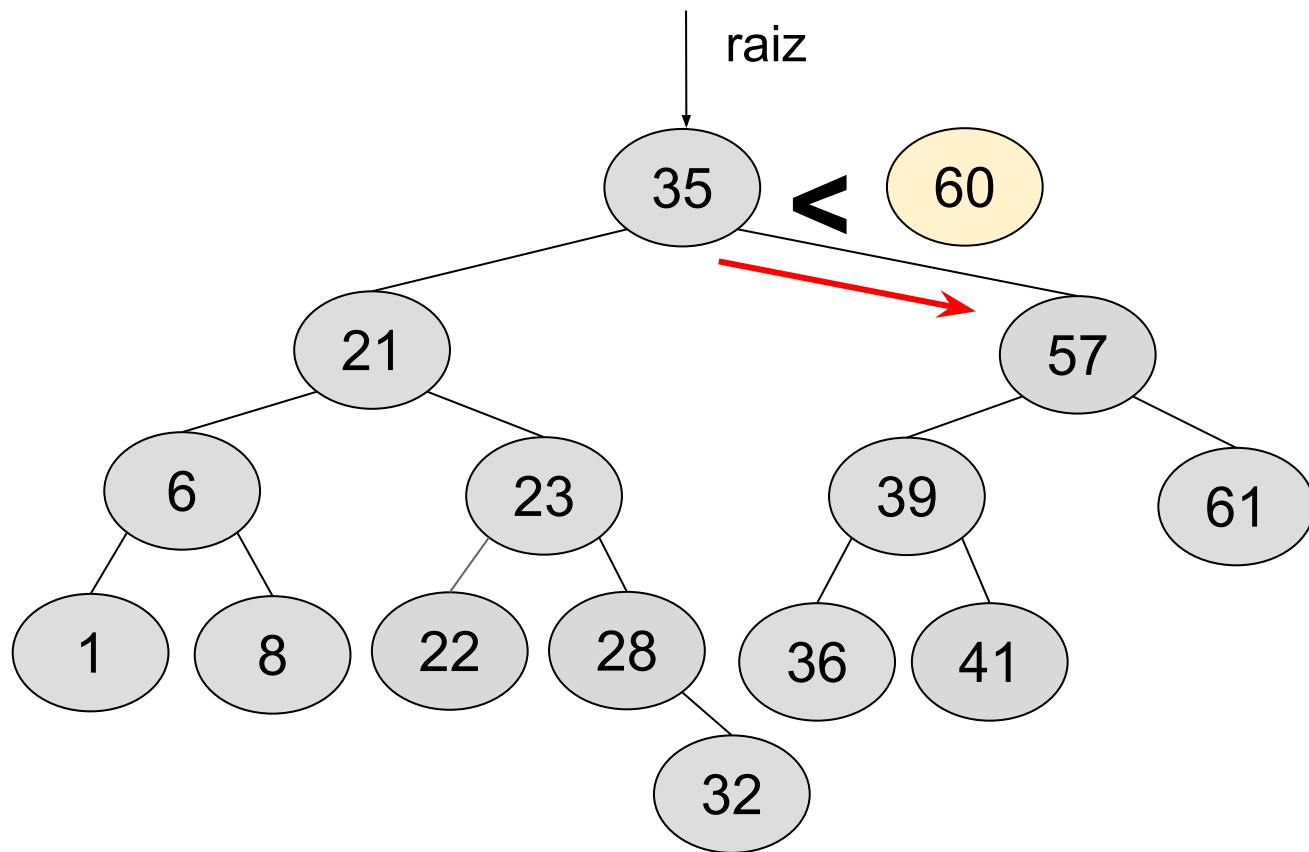


Busca

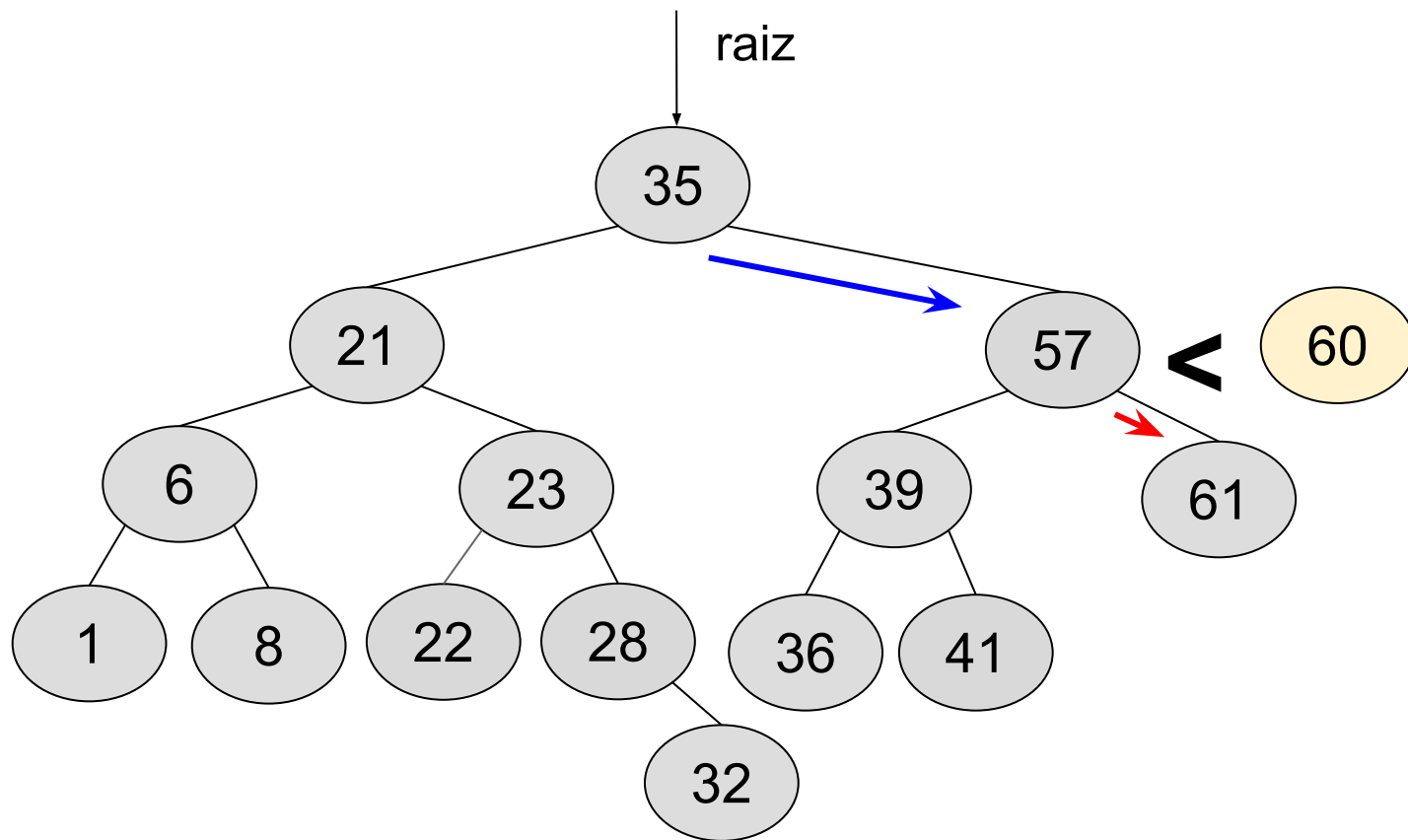
60



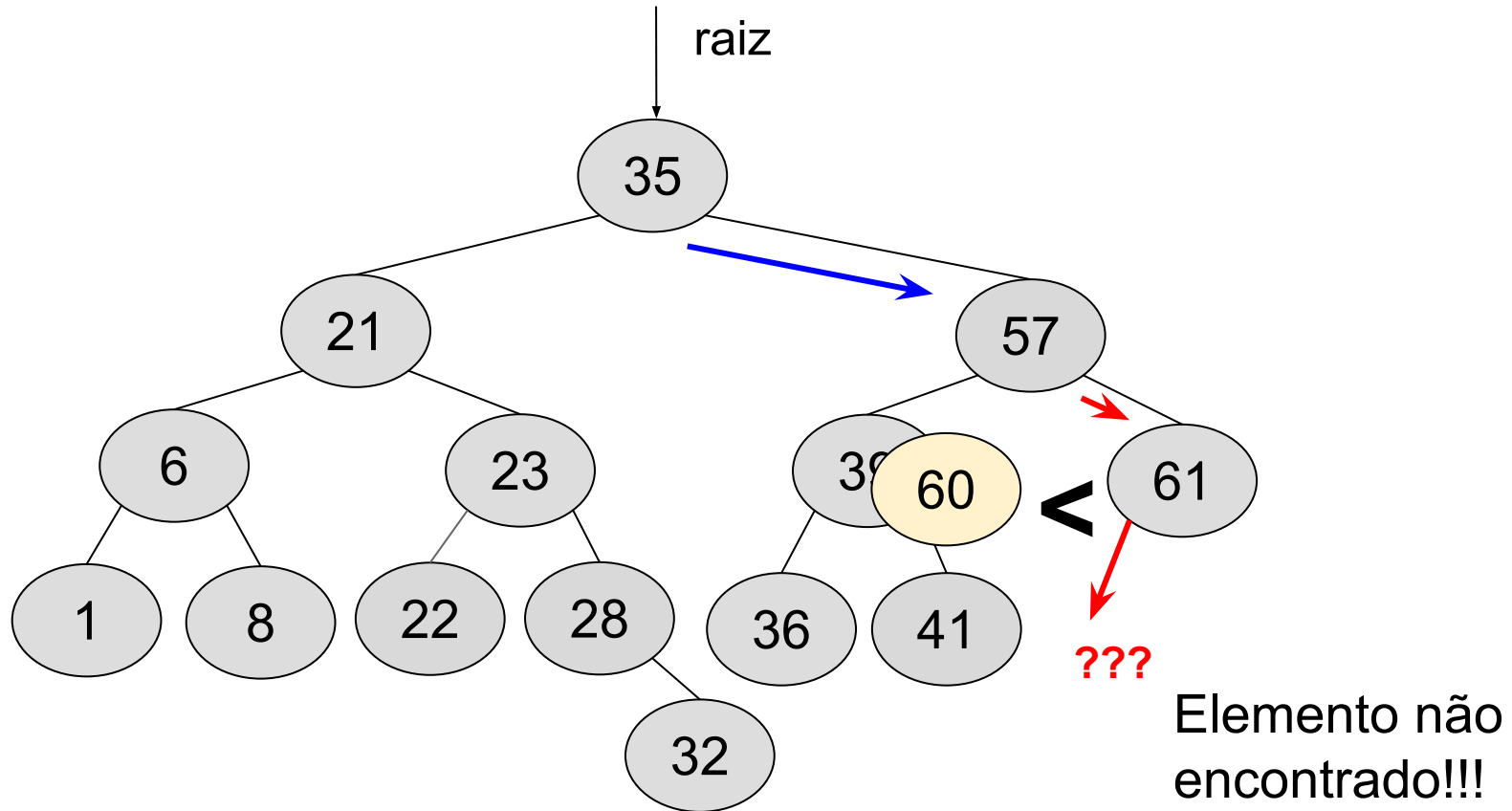
Busca



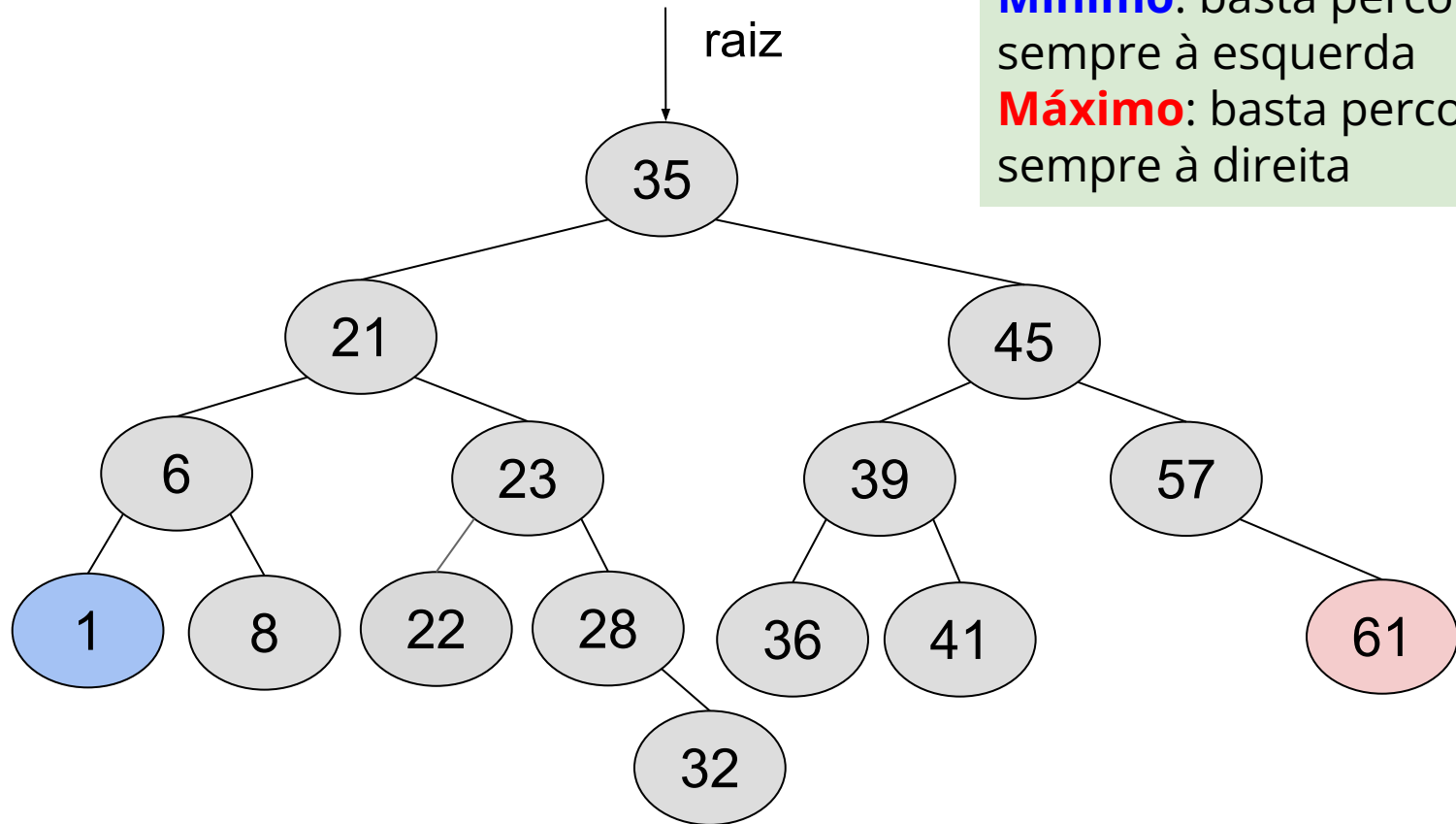
Busca



Busca



Mínimo e Máximo



Mínimo: basta percorrer sempre à esquerda
Máximo: basta percorrer sempre à direita

IMPLEMENTAÇÃO DA ABB



ABB - IMPLEMENTAÇÃO (1/4)

A implementação de uma ABB é feita geralmente utilizando-se nós ligados por indireção, com um ponteiro apontando para o nó raiz.

Algumas implementações armazenam em cada nó a sua altura ou informações adicionais para balanceamento.

Cada nó aponta para seus dois filhos (esquerdo e direito) e armazena um valor. Caso não possua um de seus filhos, ou ambos, então o respectivo ponteiro irá apontar para NULO.

ABB - IMPLEMENTAÇÃO (2/4)

Em algumas implementações, os nós também apontam para o nó pai (duplo encadeamento). Isso permite sair de um nó folha, por exemplo, e retornar à raiz, sem precisar usar recursão.

Iremos mostrar as implementações considerando o duplo encadeamento, para maior detalhamento. Recomendamos, entretanto, a implementação com encadeamento simples, por ser mais simples e eficiente, e mais fácil de ser mantida.

Como será verificado, **não é tarefa simples** manter o duplo encadeamento, que deve ser evitado, a menos que realmente necessário.

ABB - IMPLEMENTAÇÃO (3/4)

A maior parte dos métodos em uma ABB podem ser implementados de maneira recursiva ou iterativa. Em uma busca recursiva, por exemplo, cada nó verifica se não possui o valor e, em caso contrário, repassa a chamada para o filho adequado.

Como os nós costumam ser privados, métodos que os utilizam costumam ser auxiliares e privados ou movidos para uma classe que representa os nós da lista.

ABB - IMPLEMENTAÇÃO (4/4)

Entre os métodos básicos encontram-se inserção, remoção e busca. Além desses métodos, algumas implementações também disponibilizam métodos para encontrar o maior e o menor elemento.

Entre os métodos auxiliares, encontra-se mínimo e máximo (se não disponibilizados como básicos), que são necessários para a remoção. No caso da remoção iterativa, também é disponibilizado um método para copiar um nó para uma nova posição (transplanta).

CRIAÇÃO E DESTRUIÇÃO DA ABB - PSEUDOCÓDIGO

criarABB():

raiz \leftarrow NULO;

destruirABB():

// ou cada nó chama o destrutor de seus filhos, ou
// percorre-se a árvore pós-ordem destruindo os filhos
// antes do nó pai
destruirNohRecursivamente(raiz);

ABB - INSERÇÃO RECURSIVA - PSEUDOCÓDIGO - 1

inserirRecursivamente(umValor):

raiz \leftarrow inserirRecAux(raiz, umValor);

inserirRecAux(umNoh, umValor):

se (umNoh = NULO) {

 novo \leftarrow criar_noh(umValor); // cria um nó com o valor

 retornar novo;

}

senão { ...

ABB - INSERÇÃO RECURSIVA - PSEUDOCÓDIGO - 2

```
senão {  
    // não é folha nula, checa inserção à esquerda ou direita  
    se (umValor < umNoh.valor) {  
        umNoh.esquerdo ← inserirRecAux(umNoh.esquerdo, umValor);  
        // se for duplo encadeamento, acerta o nó pai  
        umNoh.esquerdo.pai ← umNoh;  
    } senão {  
        umNoh.direito ← inserirRecAux(umNoh.direito, umValor);  
        // se for duplo encadeamento, acerta o nó pai  
        umNoh.direito.pai ← umNoh;  
    }  
}  
retornar umNoh;
```

ABB - INSERÇÃO ITERATIVA - PSEUDOCÓDIGO - 1

inserirIterativamente(umValor):

novo ← criar_noh(umValor); // cria um nó com o valor

se (raiz = NULO) {

 raiz ← novo;

} senão {

 atual ← raiz;

 // percorre a árvore para encontrar o ponto de inserção

 // anterior irá marcar o ponto de inserção

 enquanto (atual ≠ NULO) {

ABB - INSERÇÃO ITERATIVA - PSEUDOCÓDIGO - 2

```
// percorre a árvore para encontrar o ponto de inserção
// anterior irá marcar o ponto de inserção
enquanto (atual ≠ NULO) {
    anterior ← atual;
    // use ≥ para permitir valores repetidos
    se (atual.valor > umValor) {
        atual ← atual.esquerdo; // segue pelo filho esquerdo
    } senão {
        atual ← atual.direito; // segue pelo filho direito
    }
}
```

ABB - INSERÇÃO ITERATIVA - PSEUDOCÓDIGO - 3

```
// encontrou o ponto, agora é inserir
novo.pai ← anterior;
se (anterior.valor > novo.valor) {
    anterior.esquerdo ← novo;
} senão {
    anterior.direito ← novo;
}
}
```

ABB - MÍNIMO - PSEUCÓDIGO

minimo():

```
se (raiz = NULO) {  
    geraErro("Árvore vazia");  
}  
senão {  
    nohMinimo ← minimoAux(raiz);  
    retornar nohMinimo.valor;  
}
```

minimoAux(raizSubArvore):

```
enquanto (raizSubArvore.esquerdo ≠ NULO) {  
    raizSubArvore ← raizSubArvore.esquerdo;  
}  
retornar raizSubArvore;
```


ABB - MÁXIMO - PSEUCÓDIGO

maximo():

```
se (raiz = NULO) {  
    geraErro("Árvore vazia");  
}  
senão {  
    nohMaximo ← maximoAux(raiz);  
    retornar nohMaximo.valor;  
}
```

maximoAux(raizSubArvore):

```
enquanto (raizSubArvore.direito ≠ NULO) {  
    raizSubArvore ← raizSubArvore.direito;  
}  
retornar raizSubArvore;
```

ABB - BUSCAAUX - PSEUDOCÓDIGO

buscaAux(umValor):

// método retorna um nó, por isso é auxiliar (privado)

atual ← raiz;

enquanto (atual ≠ NULO) {

 se (atual.valor = umValor) {

 retorna atual;

 } senão se (atual.valor > umValor) {

 atual ← atual->esquerdo;

 } senão {

 atual ← atual.direito;

 }

}

retorna atual; // retorna nulo quando não encontrado

ABB - BUSCA - PSEUDOCÓDIGO

busca(unValor):

```
nohComValor ← buscaAux(unValor);  
se (nohComValor = NULO){  
    informa("Não encontrado");  
} senão {  
    // efetua ação desejada,  
    // por exemplo retornar o registro completo  
    efetuaAcao(nohComValor);  
}
```

ABB - PERCORRER EM ORDEM - PSEUDOCÓDIGO

percorreEmOrdem():

```
percorreEmOrdemAux(raiz);
```

percorreEmOrdemAux(umNoh):

```
se (umNoh != NULL) {  
    percorreEmOrdemAux(umNoh.esquerdo);  
    efetuaAcao(umNoh); // impressão, etc.  
    percorreEmOrdemAux(umNoh.direito);  
}
```

ABB - PERCORRER PRÉ-ORDEM - PSEUDOCÓDIGO

percorrePreOrdem():

```
percorrePreOrdemAux(raiz);
```

percorrePreOrdemAux(umNoh):

```
se (umNoh != NULL) {  
    efetuaAcao(umNoh); // impressão, etc.  
    percorrePreOrdemAux(umNoh.esquerdo);  
    percorrePreOrdemAux(umNoh.direito);  
}
```

ABB - PERCORRER PÓS-ORDEM - PSEUDOCÓDIGO

percorrePosOrdem():

```
percorrePosOrdemAux(raiz);
```

percorrePosOrdemAux(umNoh):

```
se (umNoh != NULL) {  
    percorrePosOrdemAux(umNoh.esquerdo);  
    percorrePosOrdemAux(umNoh.direito);  
    efetuaAcao(umNoh); // impressão, etc.  
}
```

REMOÇÃO DE ELEMENTOS (VERSÃO ITERATIVA)



REMOÇÃO ITERATIVA

O procedimento iterativo de remoção consiste basicamente em encontrar o nó a ser removido e utilizar uma função auxiliar, transplanta, para auxiliar no processo de mudar um nó de posição.

Após o “transplante” do nó, os ajustes finais, incluindo um dos filhos do nó sendo removido são realizados.

ABB - TRANSPLANTA - PSEUDOCÓDIGO

transplanta(antigo, novo):

```
// transplanta muda uma subárvore com raiz em novo
// para o local onde antes estava o nó antigo
se (raiz = antigo) {
    raiz ← novo;
} senão se (antigo = antigo.pai.esquerdo) {
    antigo.pai.esquerdo ← novo;
} senão { // antigo = antigo.pai.direito
    antigo.pai.direito ← novo;
}
se (novo ≠ NULO) {
    novo.pai ← antigo.pai;
}
```

ABB - TRANSPLANTA - PSEUDOCÓDIGO

transplanta(antigo, novo):

// transplanta muda uma subárvore com raiz em novo

// para o local onde antes estava o nó antigo

se (raiz = antigo) {

 raiz \leftarrow novo;

} ...

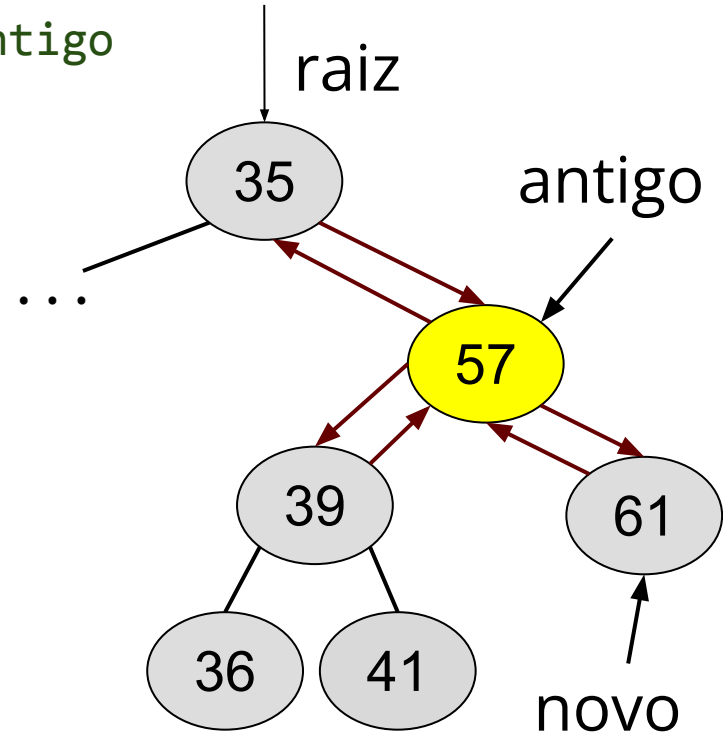


ABB - TRANSPLANTA - PSEUDOCÓDIGO

transplanta(antigo, novo):

```
// transplanta muda uma subárvore com raiz em novo
```

```
// para o local onde antes estava o nó antigo
```

```
se (raiz = antigo) {
```

```
raiz ← novo;
```

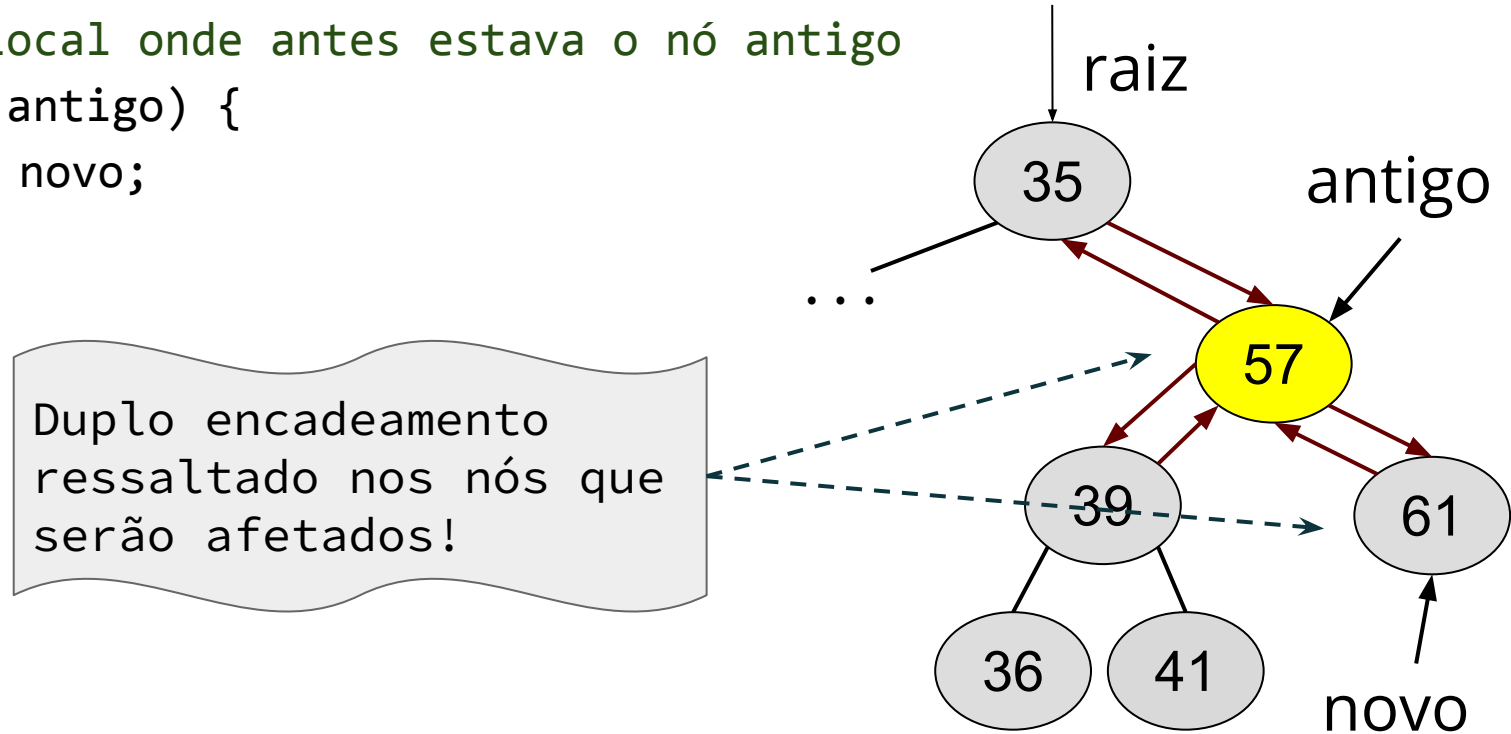
$$\} \dots$$


ABB - TRANSPLANTA - PSEUDOCÓDIGO

```
...senão se (antigo = antigo.pai.esquerdo) {  
    antigo.pai.esquerdo ← novo;  
}...
```

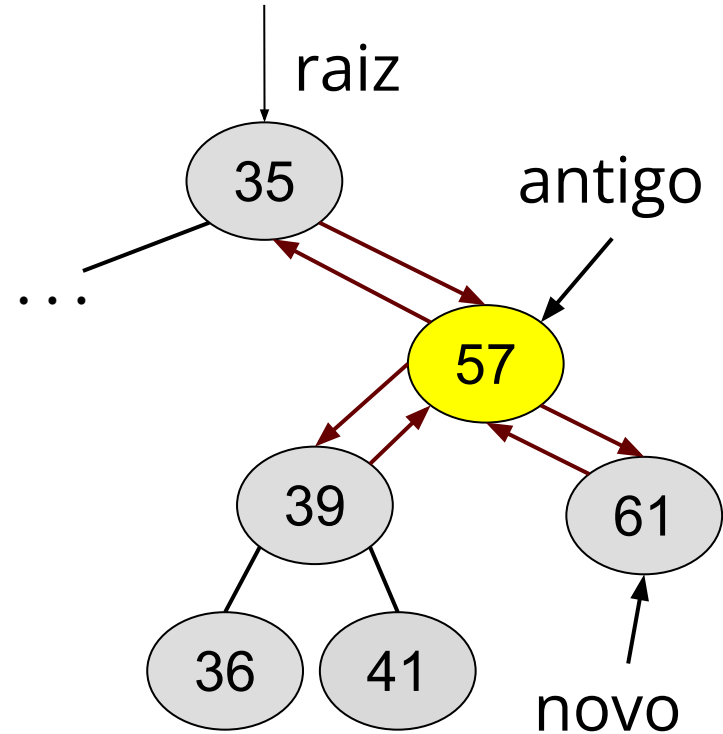


ABB - TRANSPLANTA - PSEUDOCÓDIGO

```
...senão se (antigo = antigo.pai.esquerdo) {  
    antigo.pai.esquerdo ← novo;  
}senão { // antigo = antigo.pai.direito  
    antigo.pai.direito ← novo;  
}
```

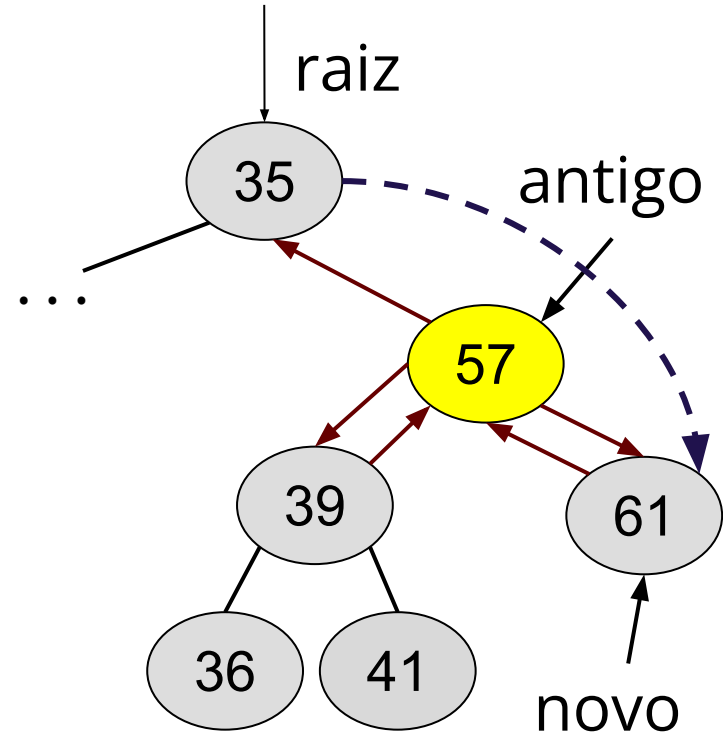


ABB - TRANSPLANTA - PSEUDOCÓDIGO

```
...senão se (antigo = antigo.pai.esquerdo) {  
    antigo.pai.esquerdo ← novo;  
}senão { // antigo = antigo.pai.direito  
    antigo.pai.direito ← novo;  
}  
  
se (novo ≠ NULO) {  
    novo.pai ← antigo.pai;  
}
```

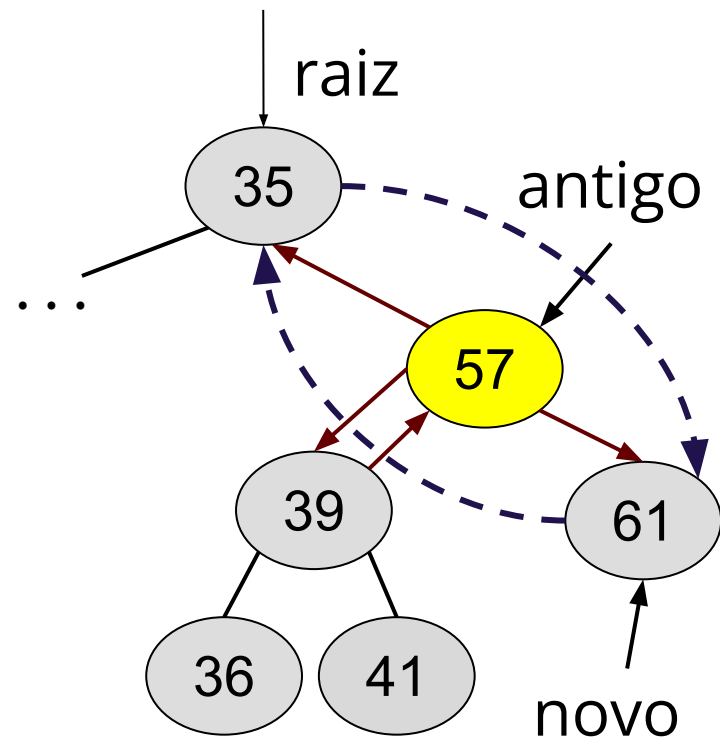


ABB - TRANSPLANTA - PSEUDOCÓDIGO

```
...senão se (antigo = antigo.pai.esquerdo) {
```

```

} se (Percebam que o nó antigo
    ficou solto na árvore, é
    necessário ajustá-lo, isso
    será feito pelo método
    remove().
    novo.pai ← antigo.pai
}

```

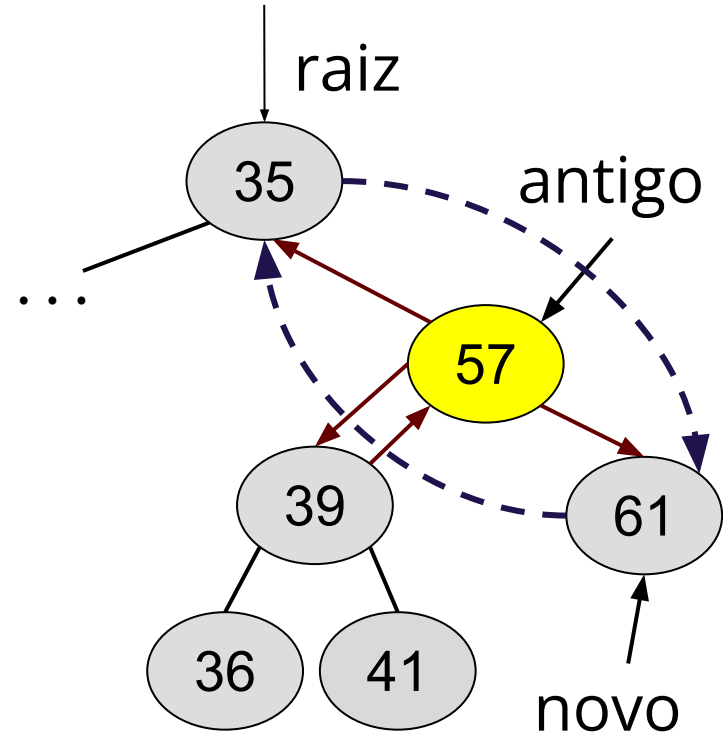


ABB - REMOÇÃO ITERATIVA - PSEUDOCÓDIGO 1

remove(umValor):

// primeiro, achamos o nó a remover na árvore

nohRemover ← buscaAux(umValor);

se (nohRemover = NULO)

 geraErro("Nó não encontrado!");

senão {

 se (nohRemover.esquerdo = NULO) {

 transplanta(nohRemover, nohRemover.direito);

 } senão se (nohRemover.direito = NULO) {

 transplanta(nohRemover, nohRemover.esquerdo);

 } senão { // nó tem dois filhos

 // podemos trocar pelo antecessor ou sucessor

 sucessor ← minimoAux(nohRemover.direito);

ABB - REMOÇÃO ITERATIVA - PSEUDOCÓDIGO - 2

```
sucessor ← minimoAux(nohRemover.direito);  
se (sucessor.pai ≠ nohRemover) {  
    transplanta(sucessor, sucessor.direito);  
    sucessor.direito ← nohRemover.direito;  
    sucessor.direito.pai ← sucessor;  
}  
transplanta(nohRemover, sucessor);  
sucessor.esquerdo ← nohRemover.esquerdo;  
sucessor.esquerdo.pai ← sucessor;  
}  
  
// ponteiros ajustados, apagamos o nó  
apagar(nohRemover);
```

ABB - REMOÇÃO ITERATIVA - PSEUDOCÓDIGO 1

remove(umValor):

// primeiro, achamos o nó a remover na árvore

nohRemover ← buscaAux(umValor);

se (nohRemover = NULO)

 geraErro(“Nó não encontrado!”);

senão {...

Considere que o valor a ser removido seja o 23

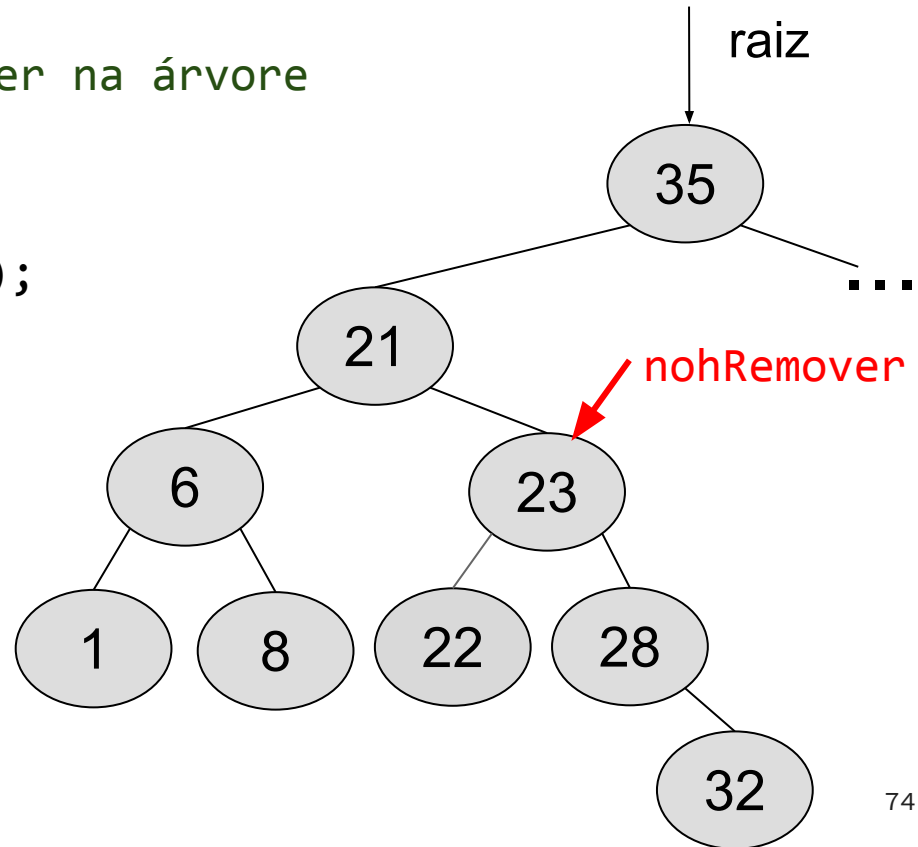


ABB - REMOÇÃO ITERATIVA - PSEUDOCÓDIGO 1

remove(umValor):

```
... senão {  
    se (nohRemover.esquerdo = NULO) {  
        transplanta(nohRemover, nohRemover.direito);  
    } senão ...
```

nohRemover.esquerdo
não é nulo

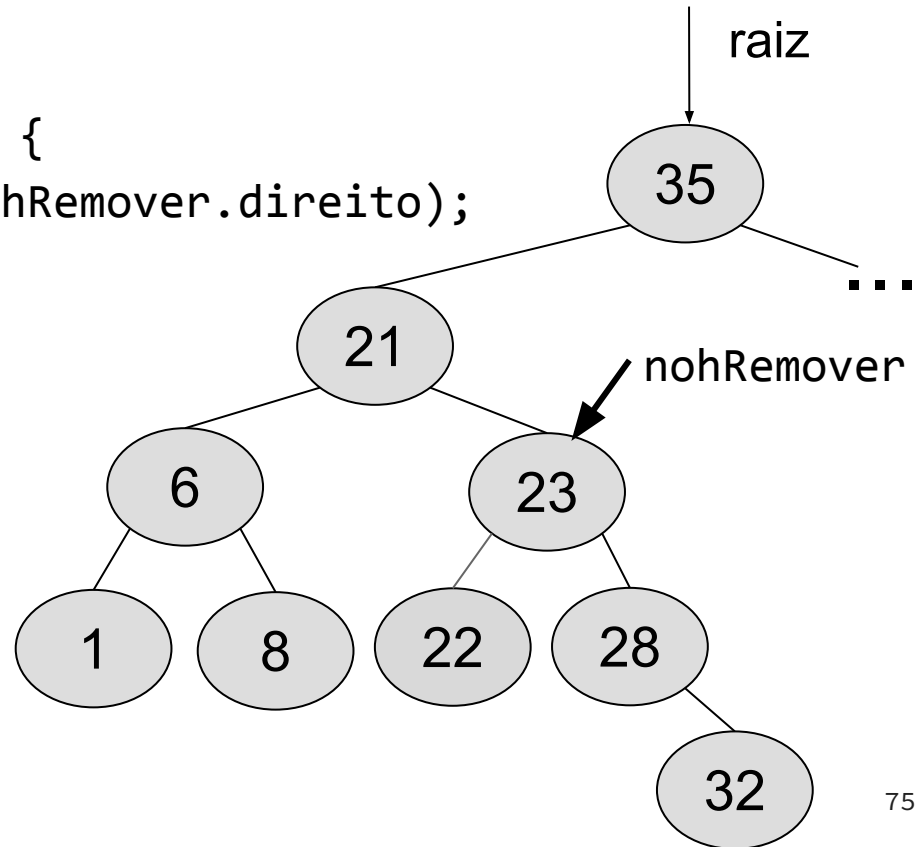


ABB - REMOÇÃO ITERATIVA - PSEUDOCÓDIGO 1

remove(umValor):

```
... senão se (nohRemover.direito = NULO) {  
    transplanta(nohRemover, nohRemover.esquerdo);  
} senão {...
```

nohRemover.direito
também não é nulo

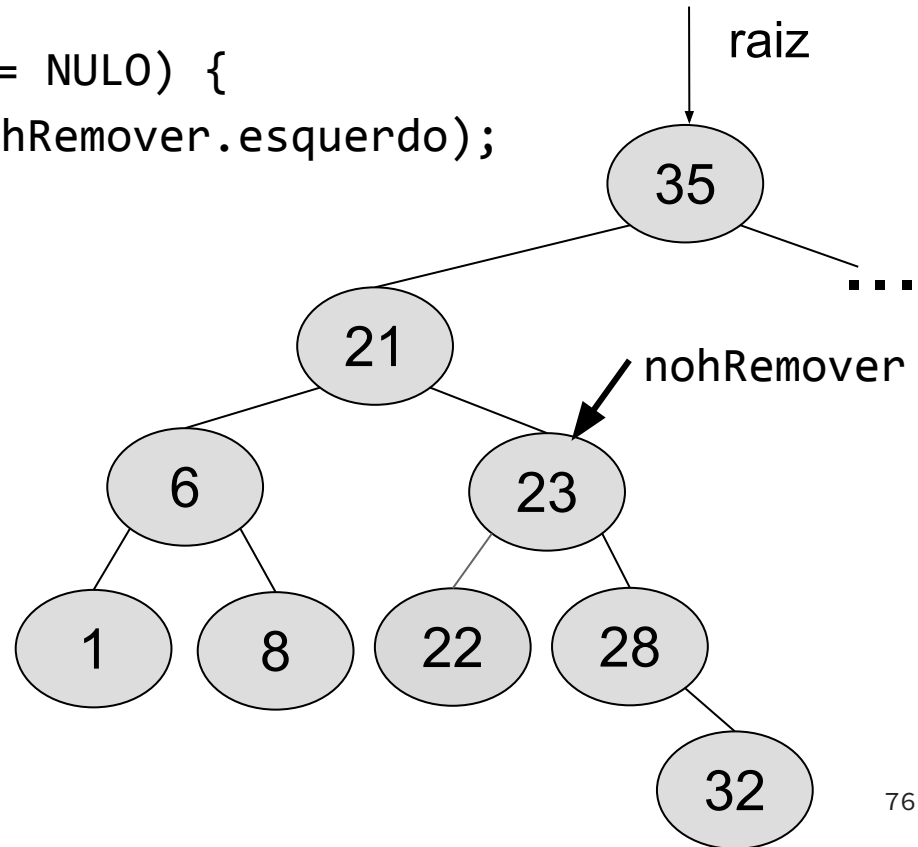


ABB - REMOÇÃO ITERATIVA - PSEUDOCÓDIGO 1

remove(umValor):

```
... senão { // nó tem dois filhos  
    // podemos trocar pelo antecessor ou sucessor  
    sucessor ← minimoAux(nohRemover.direito);  
    ...
```

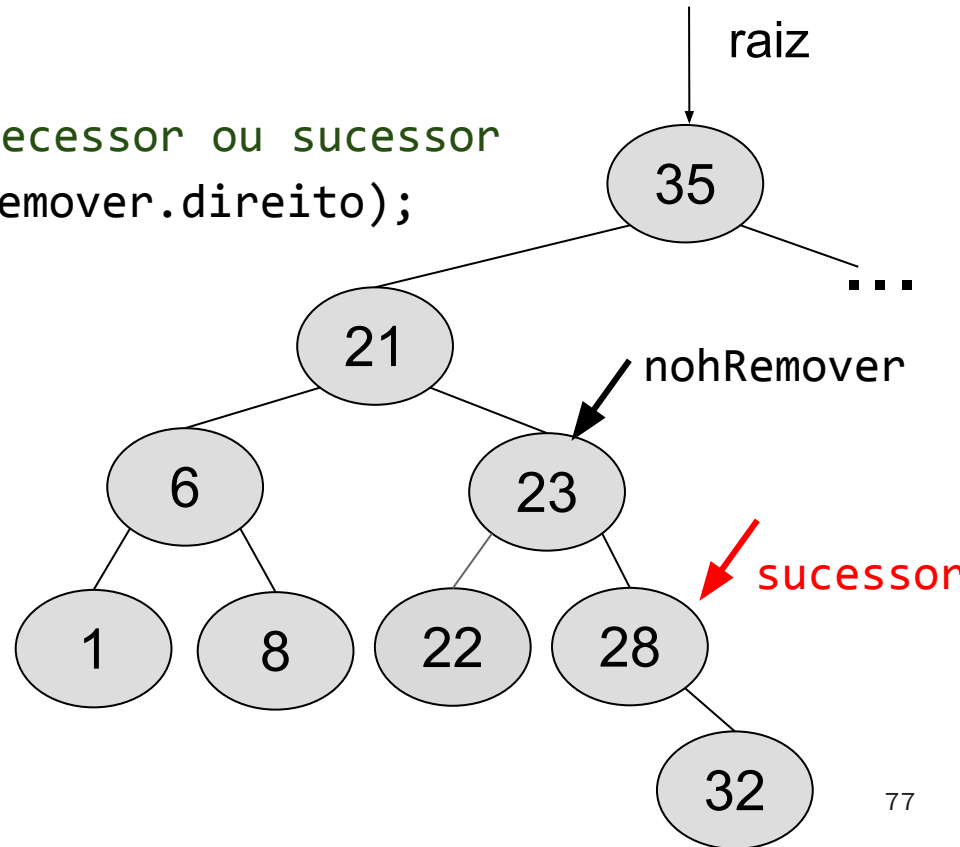


ABB - REMOÇÃO ITERATIVA - PSEUDOCÓDIGO 1

remove(unValor):

```
... se (sucessor.pai ≠ nohRemover) {  
    transplanta(sucessor, sucessor.direito);  
    sucessor.direito ← nohRemover.direito;  
    sucessor.direito.pai ← sucessor;  
}  
...
```

O nó pai do sucessor é
o próprio nó a ser
removido então...

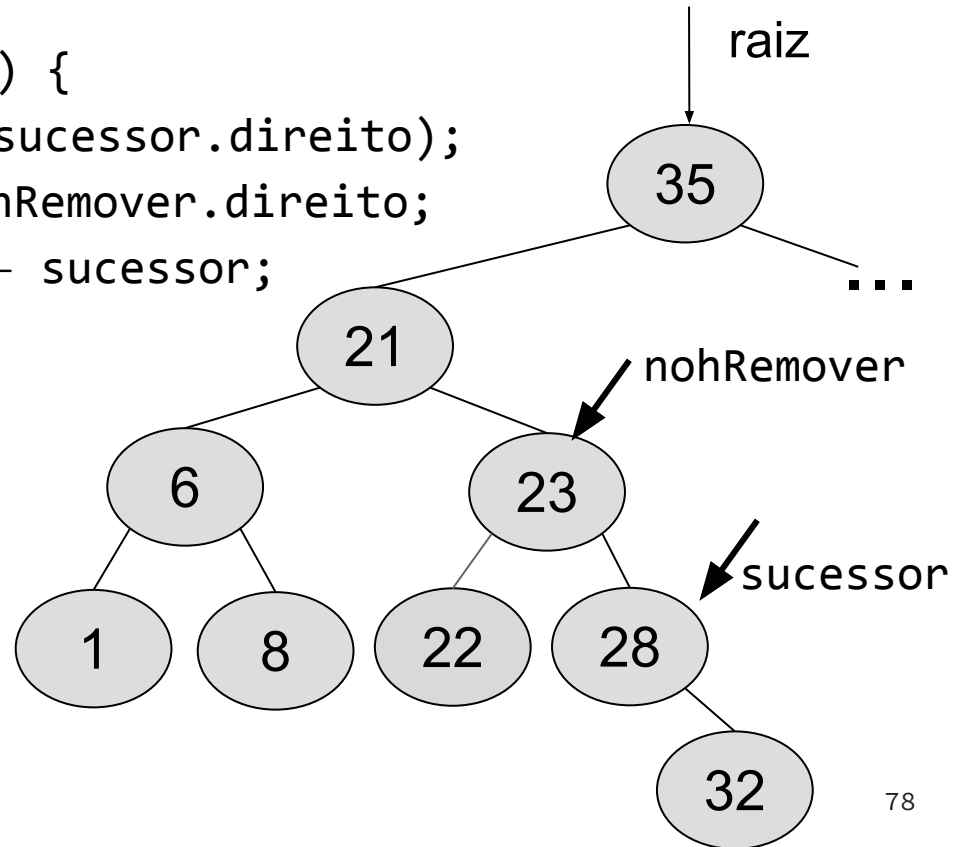


ABB - REMOÇÃO ITERATIVA - PSEUDOCÓDIGO 1

remove(umValor):

... transplanta(nohRemover, sucessor);

Ao chamar a transplanta para os nós *nohRemover* e *sucessor*, essa será a nova configuração da árvore.

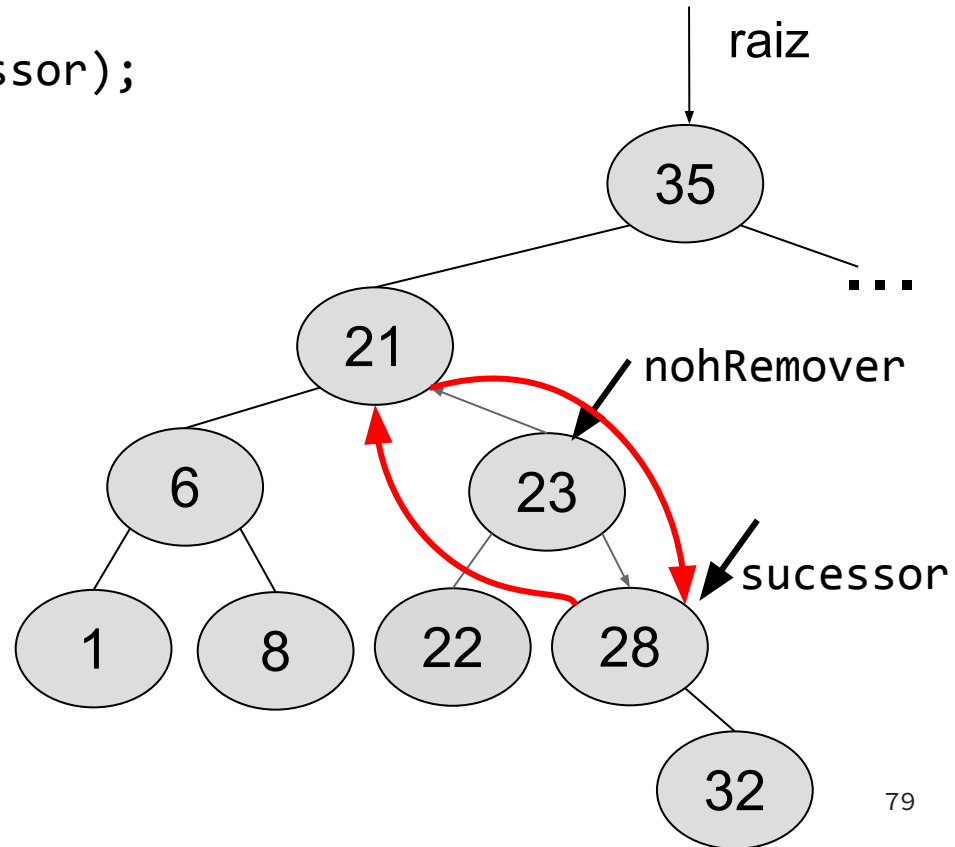


ABB - REMOÇÃO ITERATIVA - PSEUDOCÓDIGO 1

remove(umValor):

```
... sucessor.esquerdo ← nohRemover.esquerdo;  
    sucessor.esquerdo.pai ← sucessor;
```

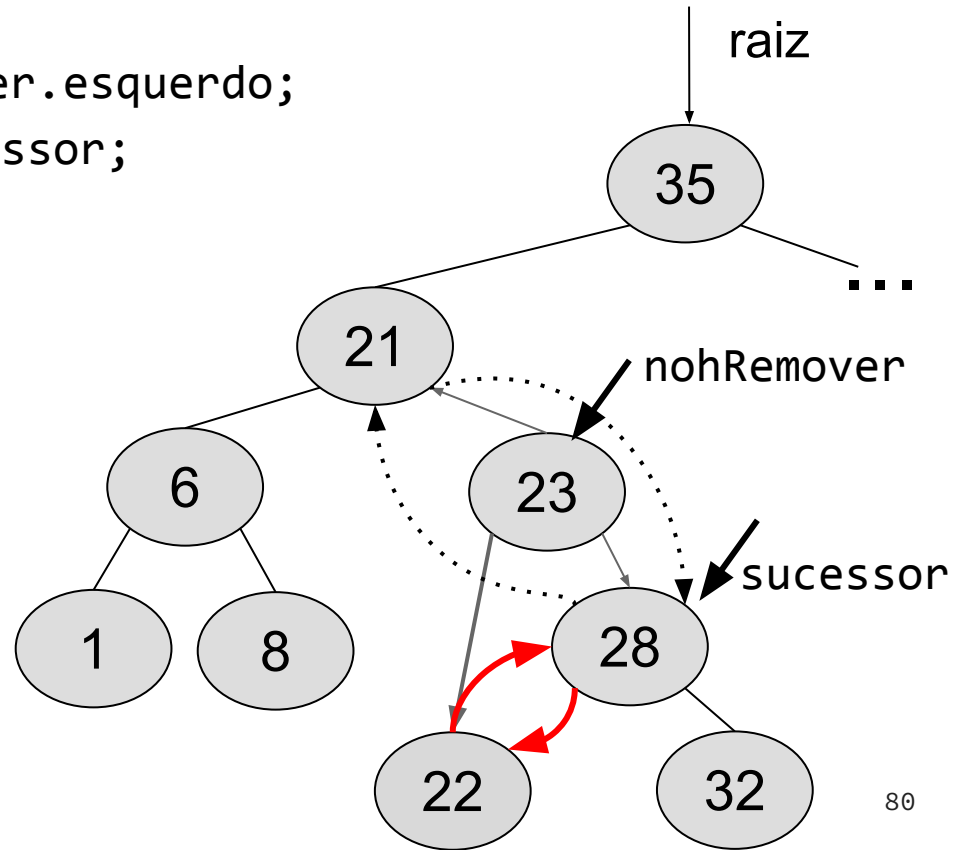


ABB - REMOÇÃO ITERATIVA - PSEUDOCÓDIGO 1

remove(umValor):

... apagar(nohRemover);

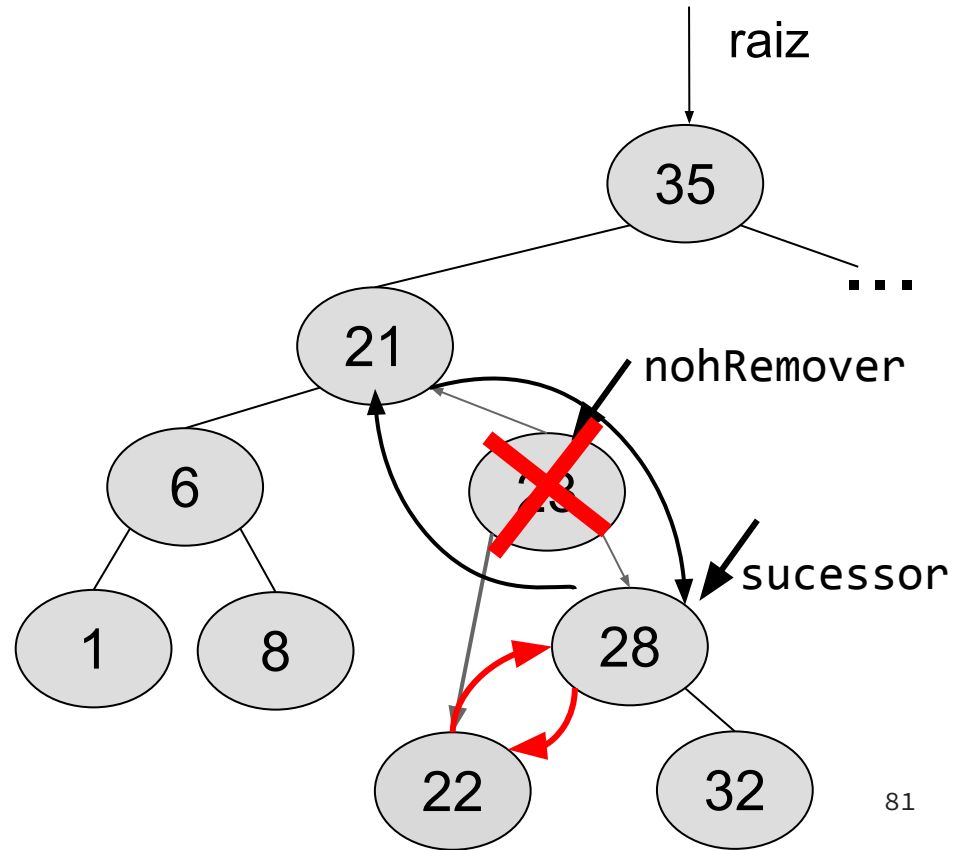


ABB - REMOÇÃO ITERATIVA - PSEUDOCÓDIGO 1

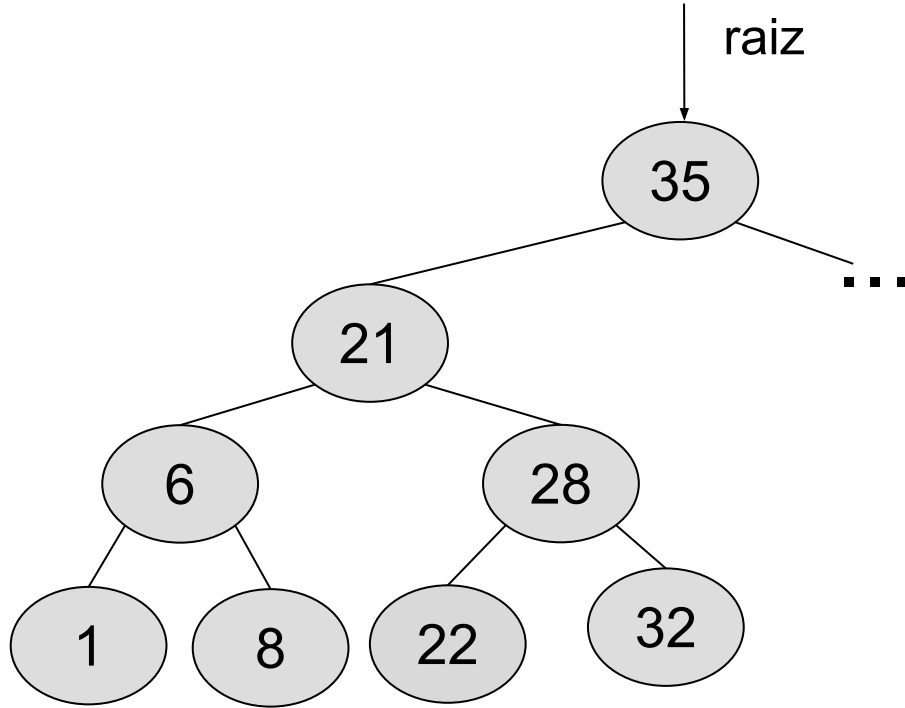


ABB - REMOÇÃO ITERATIVA - PSEUDOCÓDIGO 1

remove(umValor):

// primeiro, achamos o nó a remover na árvore

nohRemover ← buscaAux(umValor);

se (nohRemover = NULO)

 geraErro(“Nó não encontrado!”);

senão {...

Considere, agora, que o valor a ser removido seja o 45

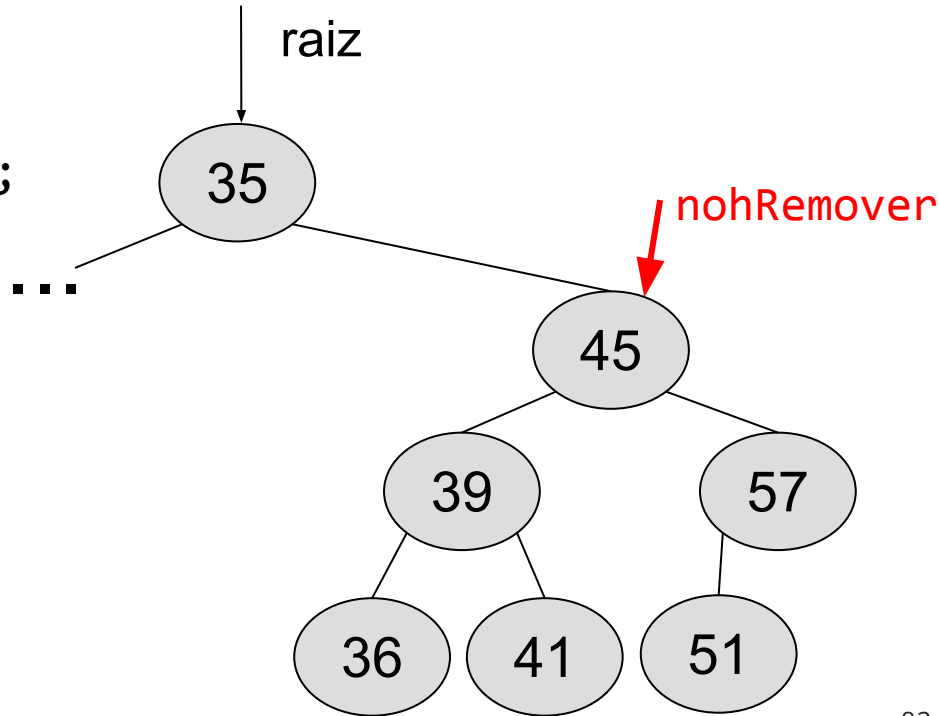


ABB - REMOÇÃO ITERATIVA - PSEUDOCÓDIGO 1

```
... se (nohRemover.esquerdo = NULO) {  
    transplanta(nohRemover, nohRemover.direito);  
}senão ...
```

nohRemover.esquerdo
não é nulo

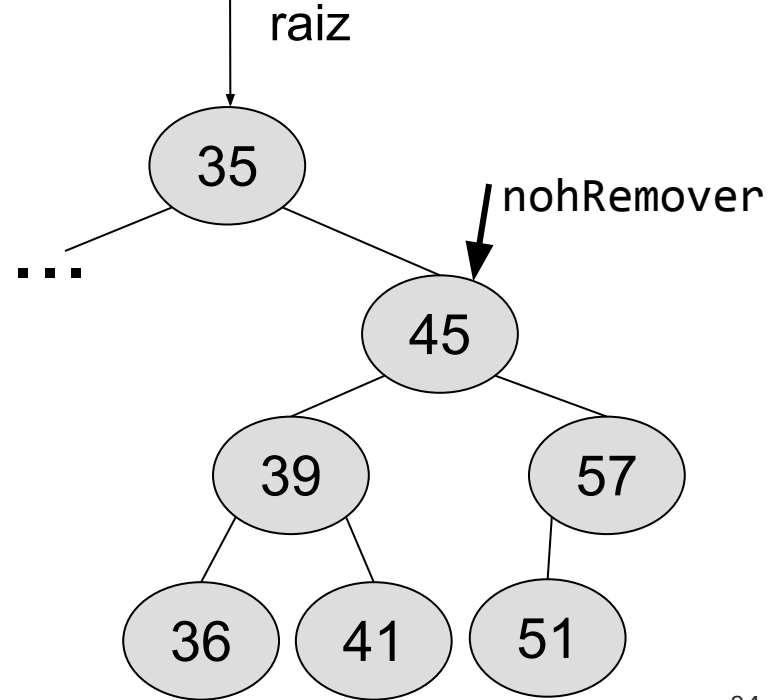


ABB - REMOÇÃO ITERATIVA - PSEUDOCÓDIGO 1

```
... senão se (nohRemover.direito = NULO) {  
    transplanta(nohRemover, nohRemover.esquerdo);  
} senão {...
```

nohRemover.direito
também não é nulo

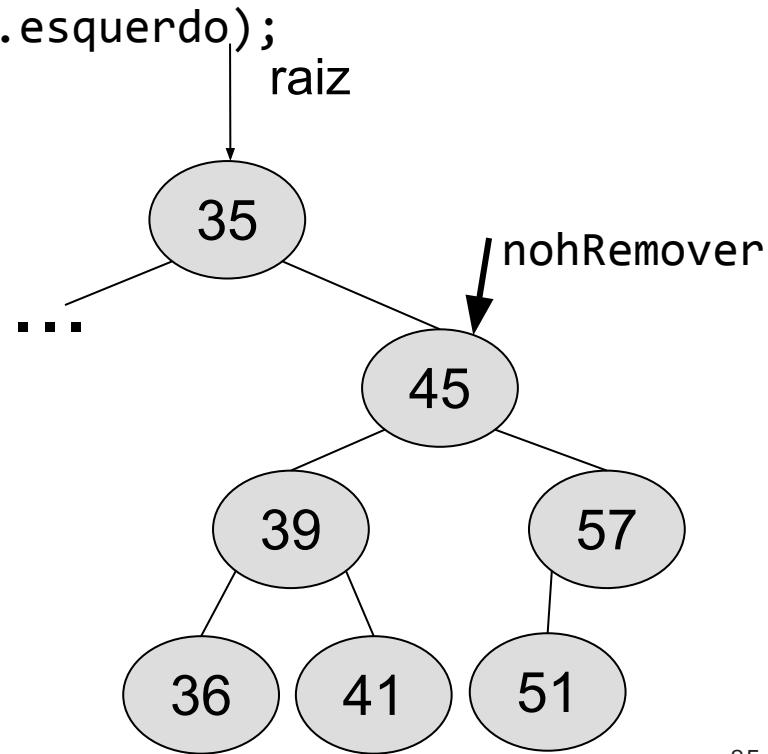


ABB - REMOÇÃO ITERATIVA - PSEUDOCÓDIGO 1

... senão { // nó tem dois filhos

// podemos trocar pelo antecessor ou sucessor

sucessor ← mínimoAux(nohRemover.direito);

...

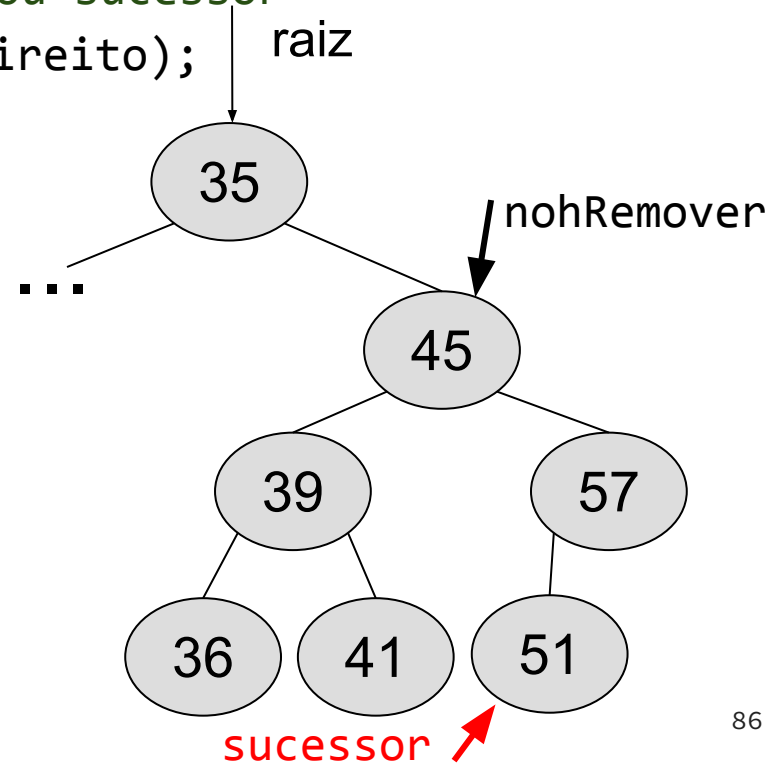
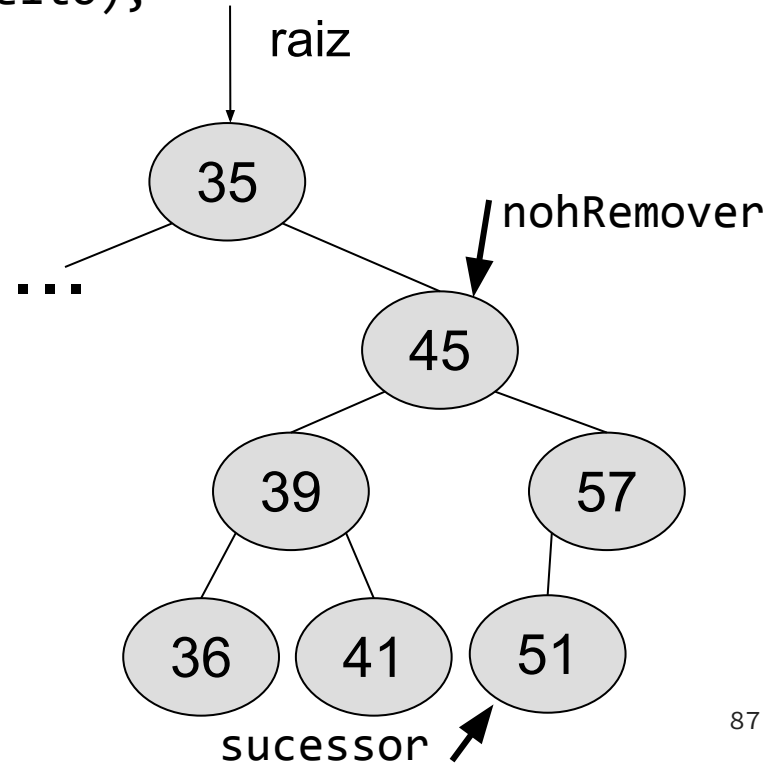


ABB - REMOÇÃO ITERATIVA - PSEUDOCÓDIGO 1

```
... se (sucessor.pai ≠ nohRemover) {  
    transplanta(sucessor, sucessor.direito);  
...}
```



O nó pai do sucessor
não é o nó a ser
removido então...

ABB - REMOÇÃO ITERATIVA - PSEUDOCÓDIGO 1

```
... se (sucessor.pai ≠ nohRemover) {  
    transplanta(sucessor, sucessor.direito);  
...  
...
```

Ao chamar a *transplanta* para os nós *sucessor* e *sucessor.direito*, essa será a nova configuração da árvore.

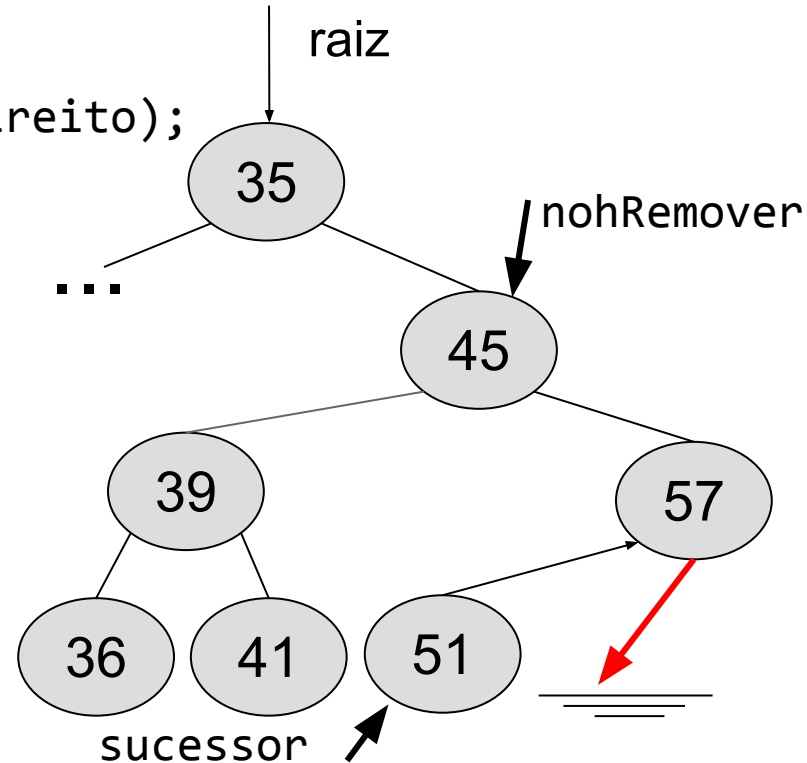


ABB - REMOÇÃO ITERATIVA - PSEUDOCÓDIGO 1

```
...    sucessor.direito ← nohRemover.direito;  
       sucessor.direito.pai ← sucessor;  
    }...
```

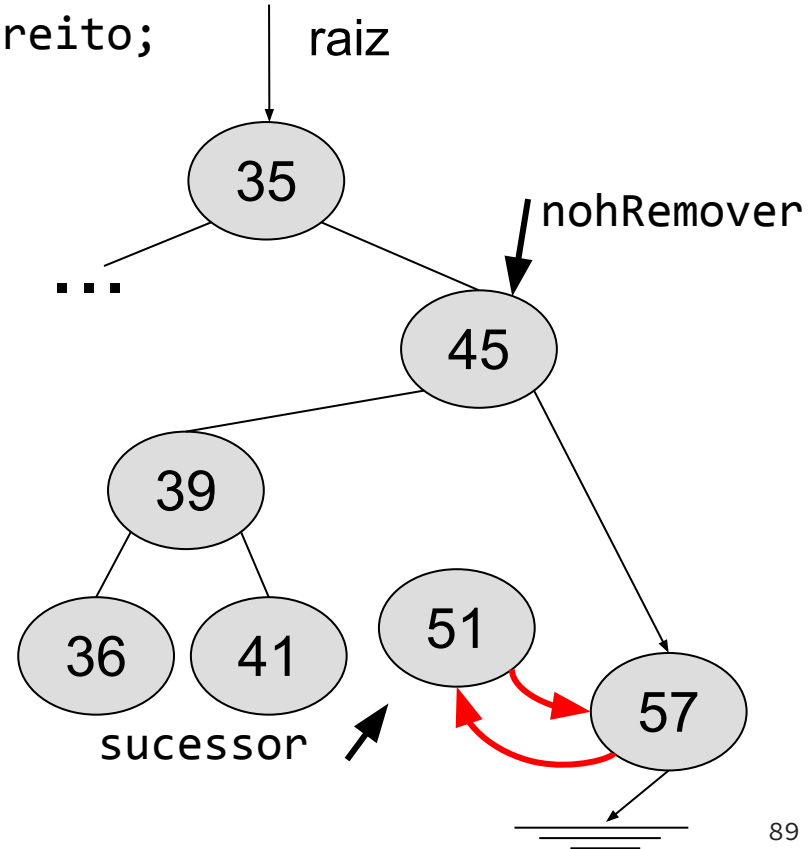


ABB - REMOÇÃO ITERATIVA - PSEUDOCÓDIGO 1

... transplanta(nohRemover, sucessor);

...

Ao chamar a transplanta para os nós *sucessor* e *nohRemover*, essa será a nova configuração da árvore.

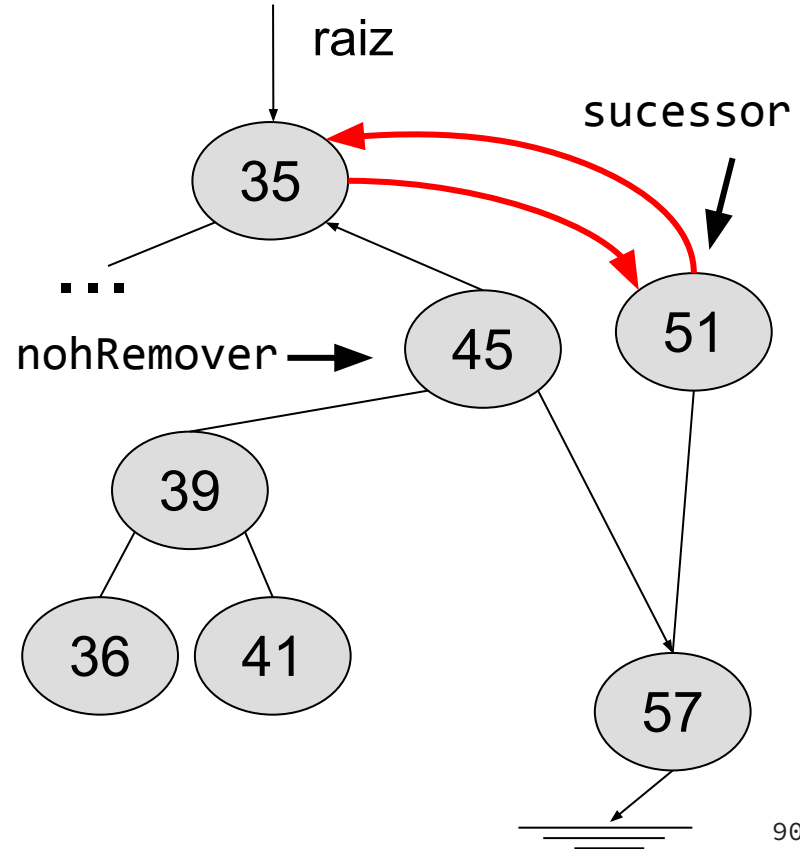


ABB - REMOÇÃO ITERATIVA - PSEUDOCÓDIGO 1

```
... sucessor.esquerdo ← nohRemover.esquerdo;  
sucessor.esquerdo.pai ← sucessor;  
...
```

Ao término desses ajustes, percebam que nohRemover foi isolado dos outros nós, podendo ser apagado sem problemas.

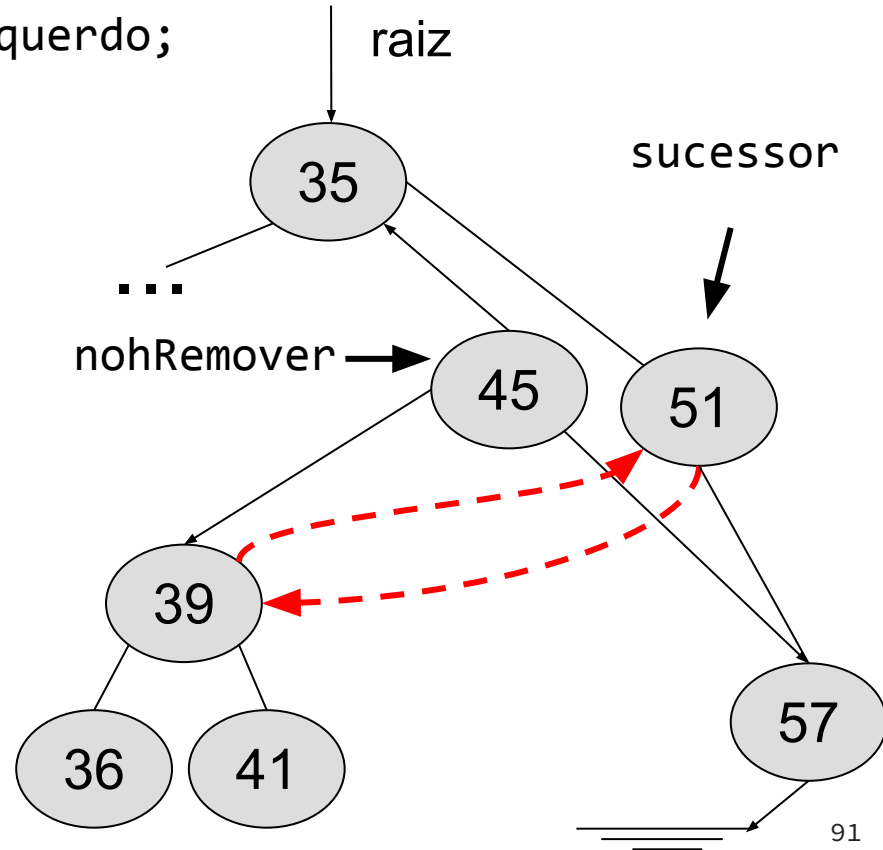


ABB - REMOÇÃO ITERATIVA - PSEUDOCÓDIGO 1

... apagar(nohRemover);

...

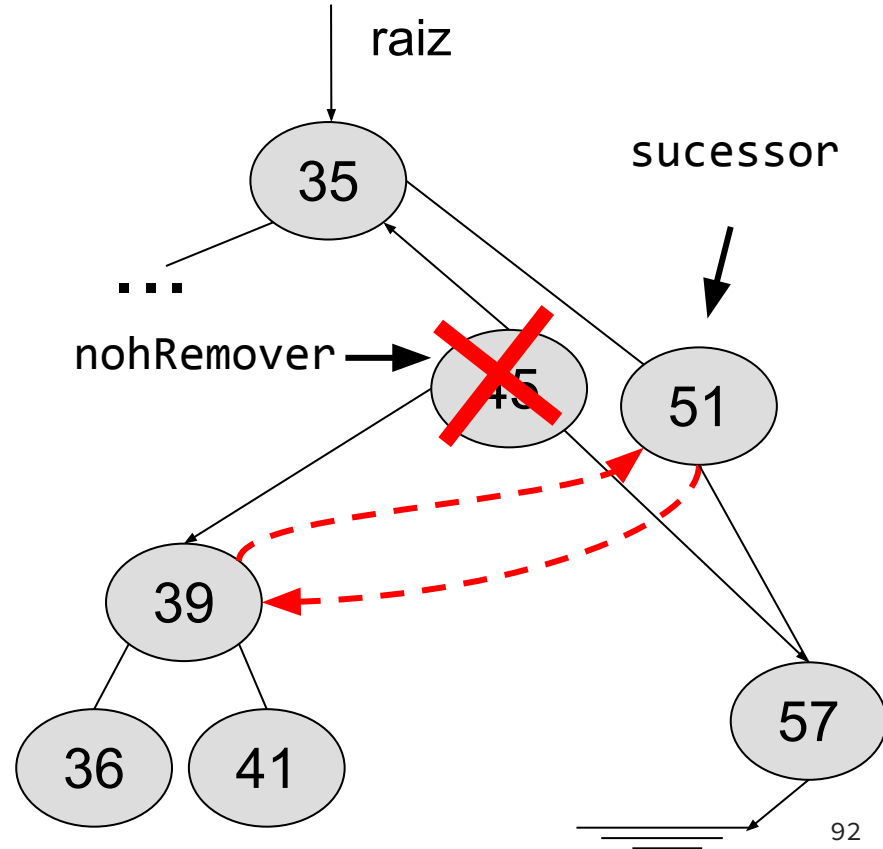
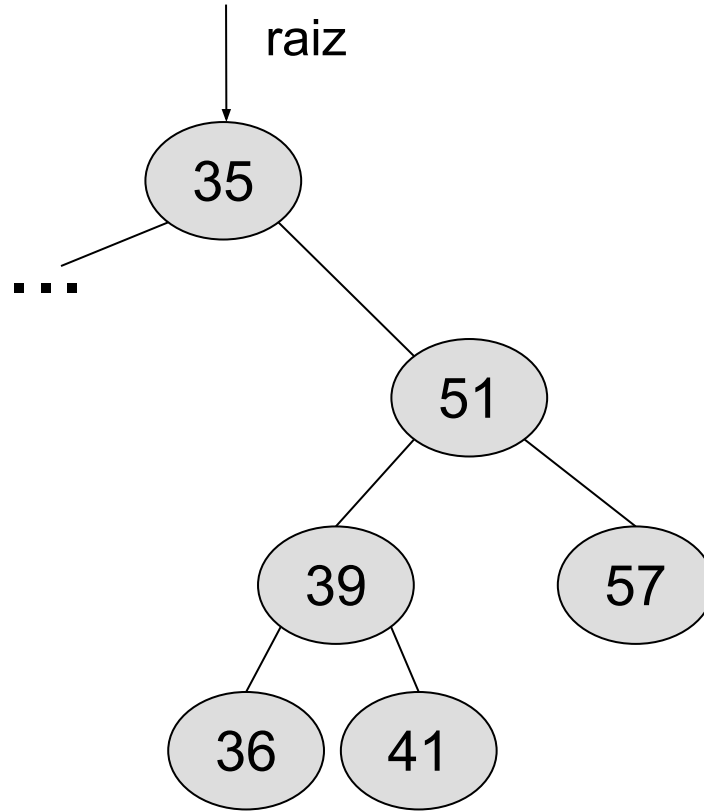


ABB - REMOÇÃO ITERATIVA - PSEUDOCÓDIGO 1



REMOÇÃO DE ELEMENTOS (VERSÃO RECURSIVA)



REMOÇÃO RECURSIVA (1/2)

É possível alterar a versão iterativa e adicionar recursão apenas na etapa de busca e ajuste dos nós. Um problema nessa versão seria o uso da transplanta, uma vez que esse método não é necessariamente recursivo.

Para a ABB tradicional isso não gera maiores problemas, entretanto isso atrapalha a implementação de árvores balanceadas que são modificações da ABB, como as árvores AVL e rubro-negras.

REMOÇÃO RECURSIVA (2/2)

Uma implementação totalmente recursiva deveria fazer a troca pelo sucessor recursivamente, no caminho do sucessor até o nó sendo removido.

Essa implementação existe e utiliza, ao invés da transplanta, um método para ir substituindo um nó pelo imediatamente menor, a `removeMenor()`.

Além disso, ela faz uso da `minimoAux()`, para encontrar o sucessor.

E O ANTECESSOR?

Estamos aqui considerando a troca pelo sucessor, obviamente também seria possível pelo antecessor. Nesse caso precisaríamos de uma `maximoAux()` e `removeMaior()`.

CADÊ MEU PAI?

Uma vantagem da implementação totalmente recursiva é a de não precisar de um apontador para o pai do nó, uma vez que o retorno ao nó pai é feito pela própria recursão.

Assim, as funções a seguir não fazem uso desse apontador. Por esse, e outros motivos, a versão recursiva de remoção em ABB deve ser preferida, a não ser que exista algum motivo para usar a versão iterativa.

ABB - REMOVEMENOR- PSEUDOCÓDIGO

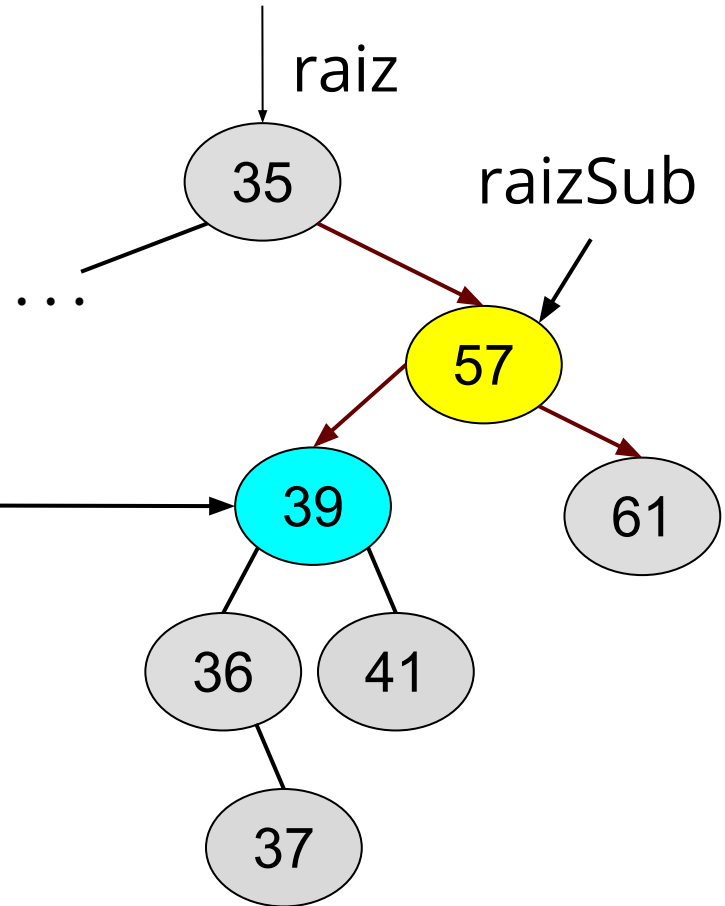
removeMenor(raizSub):

```
// procedimento auxiliar para remover o sucessor substituindo-o
// pelo seu filho à direita
se (raizSub.esquerdo = NULO) { // encontrou o sucessor
    retorna raizSub.direito;
} senão { // não achou ainda, desce mais na subárvore
    raizSub.esquerdo ← removeMenor(raizSub->esquerdo);
    retorna raizSub;
}
```

removeMenor(raizSub):

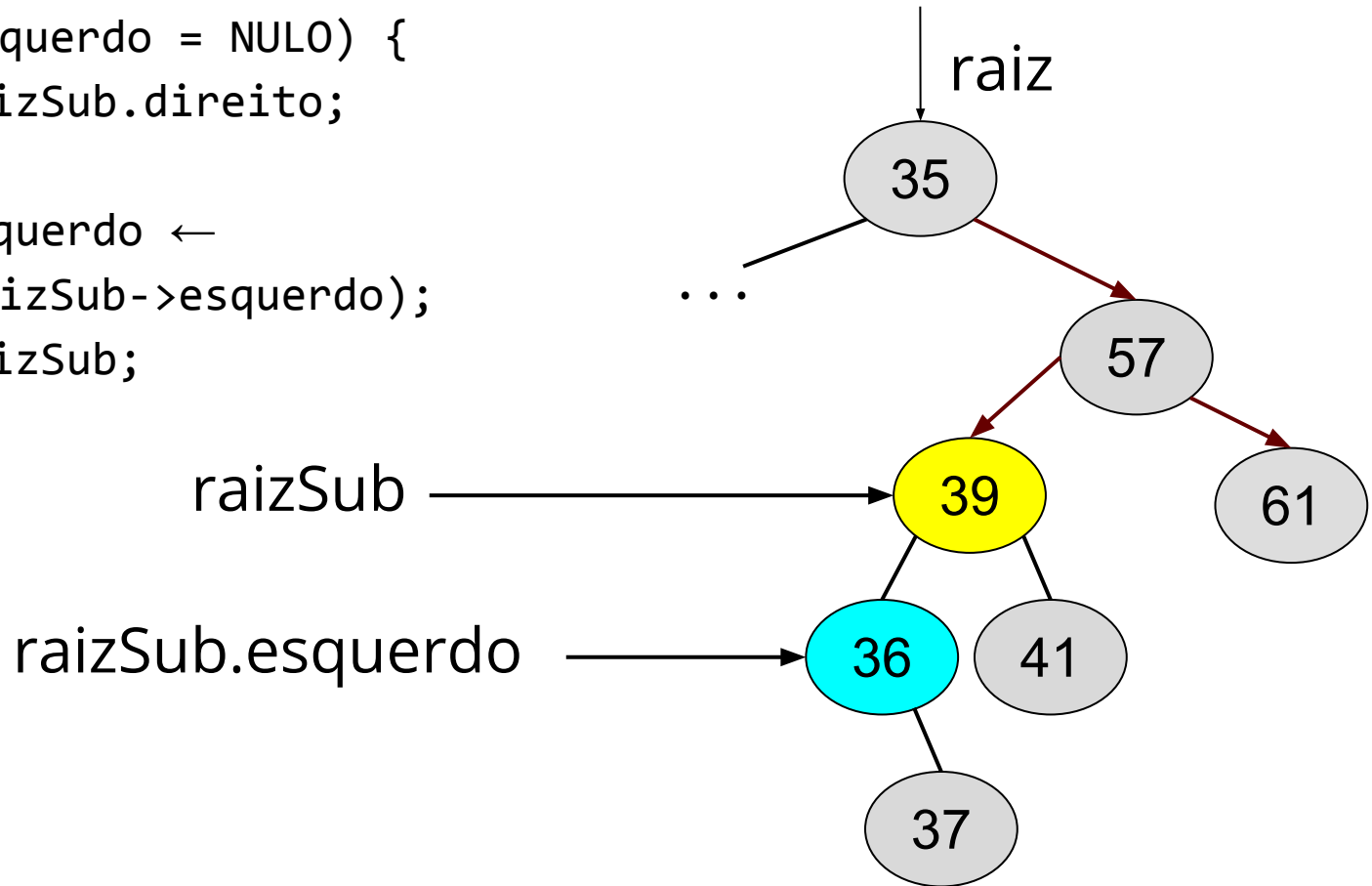
```
se (raizSub.esquerdo = NULO) {  
    retorna raizSub.direito;  
}  
senão {  
    raizSub.esquerdo ←  
    removeMenor(raizSub->esquerdo);  
    retorna raizSub;  
}
```

raizSub.esquerdo



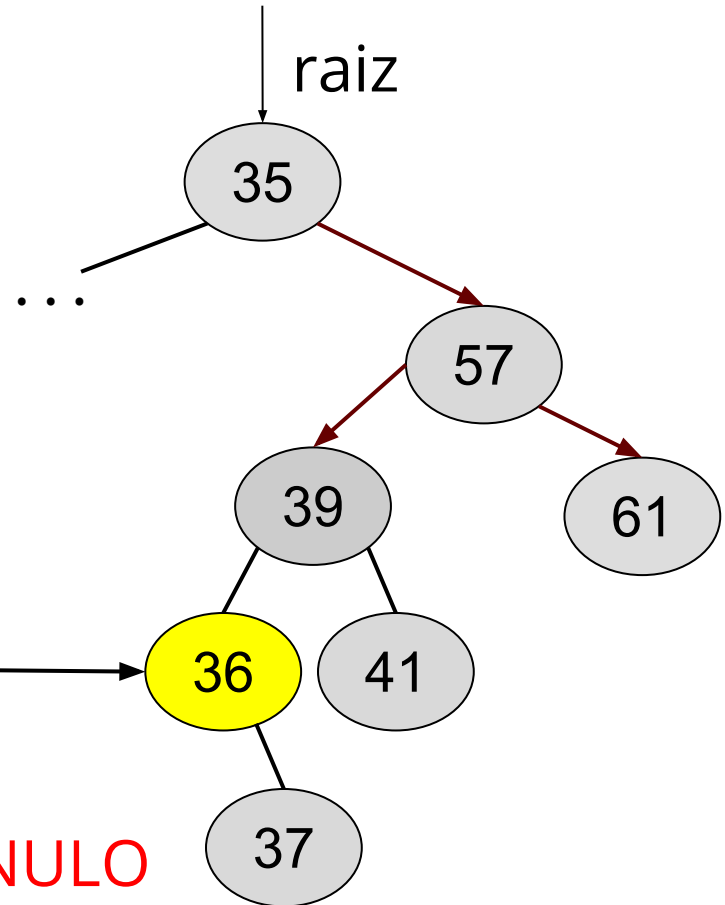
removeMenor(raizSub):

```
se (raizSub.esquerdo = NULO) {  
    retorna raizSub.direito;  
} senão {  
    raizSub.esquerdo ←  
removeMenor(raizSub->esquerdo);  
    retorna raizSub;  
}
```



removeMenor(raizSub):

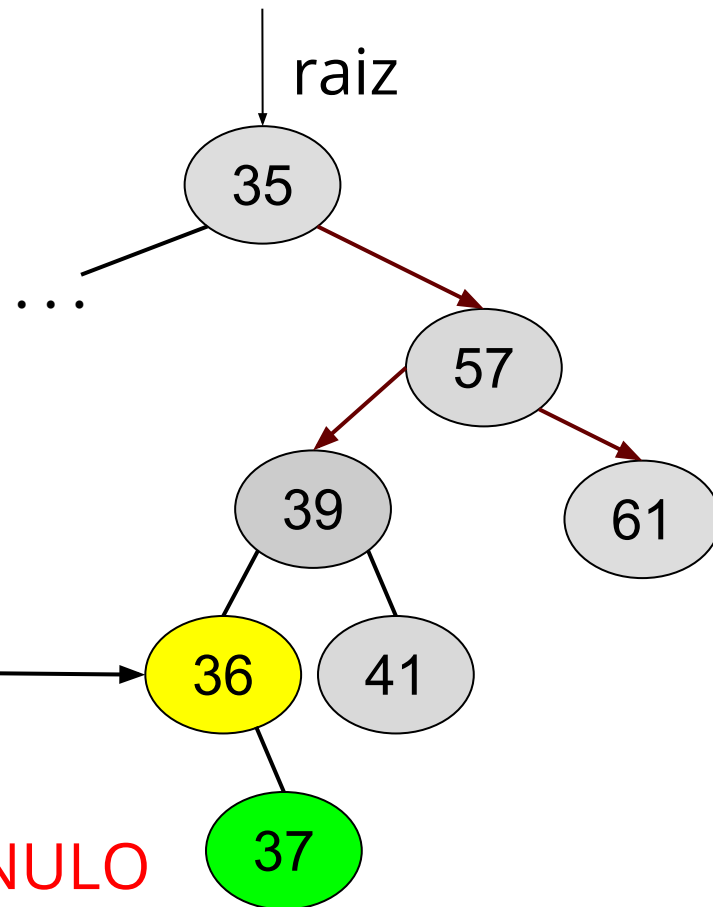
```
se (raizSub.esquerdo = NULO) {  
    retorna raizSub.direito;  
} senão {  
    raizSub.esquerdo ←  
removeMenor(raizSub->esquerdo);  
    retorna raizSub;  
}
```



removeMenor(raizSub):

```
se (raizSub.esquerdo = NULO) {  
    retorna raizSub.direito;  
}  
senão {  
    raizSub.esquerdo ←  
removeMenor(raizSub->esquerdo);  
    retorna raizSub;  
}
```

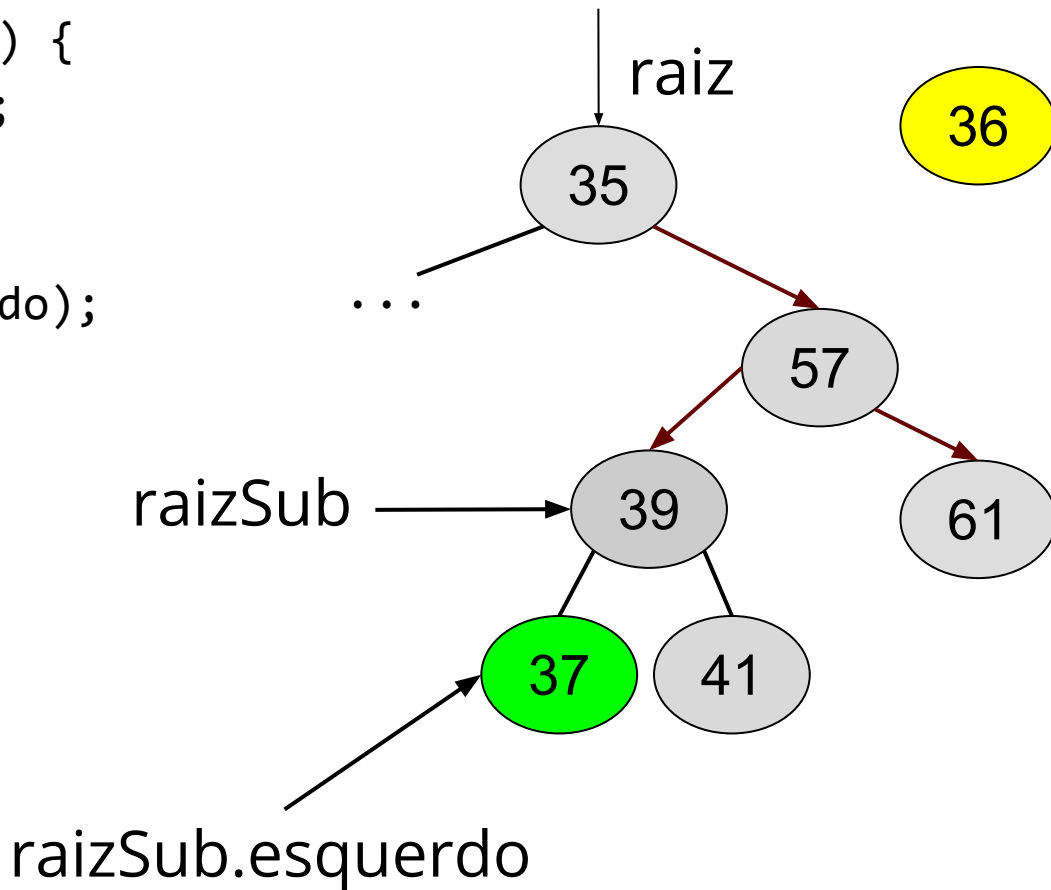
Encontrou o Sucessor, 36... A função retorna o 37 para a chamada no 39.



removeMenor(raizSub):

```
se (raizSub.esquerdo = NULO) {  
    retorna raizSub.direito;  
} senão {  
    raizSub.esquerdo ←  
removeMenor(raizSub->esquerdo);  
    retorna raizSub;  
}
```

**0 nó 39 irá
então alterar o
seu nó esquerdo,
do 36 para o 37.**



removeMenor(raizSub):

```
se (raizSub.esquerdo = NULO) {  
    retorna raizSub.direito;  
} senão {  
    raizSub.esquerdo ←  
removeMenor(raizSub->esquerdo);  
    retorna raizSub;  
}
```

O nó 36, terá sido atribuído antes à posição que se encontra o nó 57 (sendo excluído, por meio de uso da mínimoAux()).

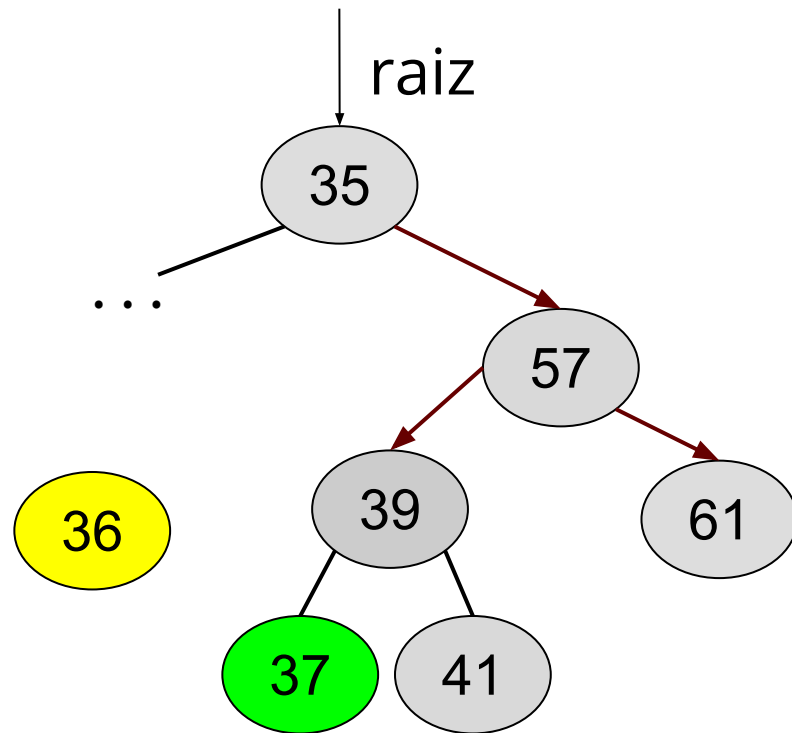


ABB - REMOÇÃO RECURSIVA 2- PSEUDOCÓDIGO - I

removerRecursivamente(umValor):

raiz \leftarrow removerRecAux(raiz, umValor);

removerRecAux(umNoh, umValor):

```
se (umNoh = NULO) {  
    geraErro("Nó não encontrado!");  
}
```

...

ABB - REMOÇÃO RECURSIVA 2 - PSEUDOCÓDIGO - II

```
novaRaizSubArvore ← umNoh;  
// valor menor que nó atual, vai para subárvore esquerda  
se ( umValor < umNoh.valor ) {  
    umNoh.esquerdo ← removerRecAux(umNoh.esquerdo, umValor);  
// valor maior que nó atual, vai para subárvore direita  
} senão se ( umValor > umNoh->valor ) {  
    umNoh.direito ← removerRecAux(umNoh.direito, umValor);  
// valor é igual ao armazenado no nó atual,  
// que deve ser apagado  
} senão { // umNoh é removido neste senão  
    // nó não tem filhos à esquerda  
    se (umNoh.esquerdo = NULO) {
```

ABB - REMOÇÃO RECURSIVA 2 - PSEUDOCÓDIGO - III

```
// nó não tem filhos à esquerda
se (umNoh.esquerdo = NULO) {
    novaRaizSubArvore ← umNoh.direito;
// nó não tem filhos à direita
} senão se (umNoh.direito = NULO) {
    novaRaizSubArvore ← umNoh.esquerdo;
} senão { // nó tem dois filhos
    // podemos trocar pelo antecessor ou sucessor
```

ABB - REMOÇÃO RECURSIVA 2 - PSEUDOCÓDIGO - III

```
// nó não tem filhos à esquerda
```

```
se (umNoh.esquerda == null)
```

```
    novaRaizSubArvore = umNoh;
```

```
// nó não tem filhos à direita
```

```
} senão se (umNoh.direita == null)
```

```
    novaRaizSubArvore = umNoh;
```

```
} senão { // nó tem dois filhos
```

```
    // podemos trocar pelo antecessor ou sucessor
```

Notem a ausência de duplo encadeamento.

ABB - REMOÇÃO RECURSIVA 2 - PSEUDOCÓDIGO - IV

```
} senão { // nó com dois filhos
    // troca o nó por seu sucessor
    novaRaizSubArvore ← minimoAux(umNoh.direito);
    // troca o sucessor por seu filho à direita
    novaRaizSubArvore.direito ←
removeMenor(umNoh.direito);
    // filho à esquerda de umNoh torna-se filho à esquerda
    // do sucessor
    novaRaizSubArvore.esquerdo ← umNoh.esquerdo;
}
// ponteiros ajustados, apagamos o nó
apagar(umNoh);
```

ABB - REMOÇÃO RECURSIVA 2 - PSEUDOCÓDIGO - V

} *// fim do senão para remoção de umNoh*

// devolve a nova raiz

retorna novaRaizSubArvore;

SOBRE O MATERIAL



SOBRE ESTE MATERIAL

Material produzido coletivamente, principalmente pelos seguintes professores do DCC/UFLA:

- Joaquim Quinteiro Uchôa
- Juliana Galvani Greggi
- Renato Ramos da Silva

Inclui contribuições de outros professores do setor de Fundamentos de Programação do DCC/UFLA.

Esta obra está licenciado com uma Licença [Creative Commons Atribuição 4.0 Internacional](https://creativecommons.org/licenses/by/4.0/).