ÁRVORES BINÁRIAS DE BUSCA

Vanessa Braganholo

Estruturas de Dados e Seus Algoritmos

REFERÊNCIA

Szwarcfiter, J.; Markezon, L. Estruturas de Dados e seus Algoritmos, 3a. ed. LTC. Cap. 4

BUSCA

Diversas aplicações precisam buscar um determinado valor em um conjunto de dados

Essa busca deve ser feita da forma mais eficiente possível

Árvores binárias possibilitam buscas com eficiência

Exemplo: buscar dados de uma pessoa que possui um determinado CPF

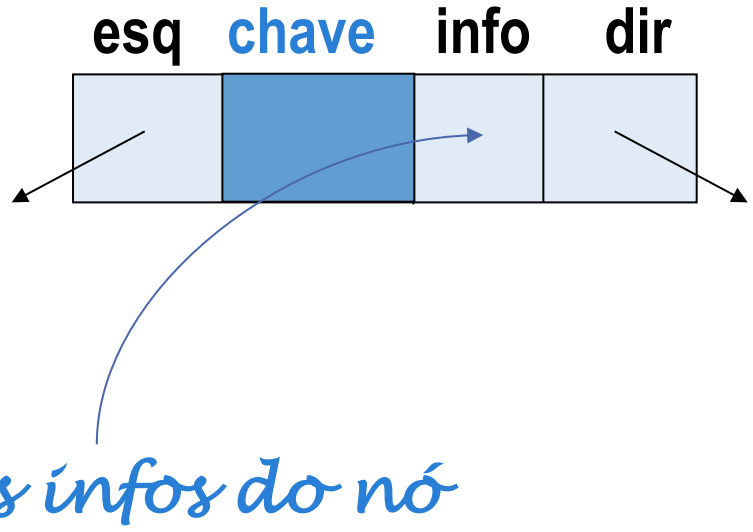
Dados das pessoas são armazenados numa árvore binária de busca

CPF funciona como "chave", pois é único para cada pessoa (não existem duas pessoas com o mesmo CPF)

ÁRVORES BINÁRIAS DE BUSCA

Apresentam uma relação de ordem entre os nós

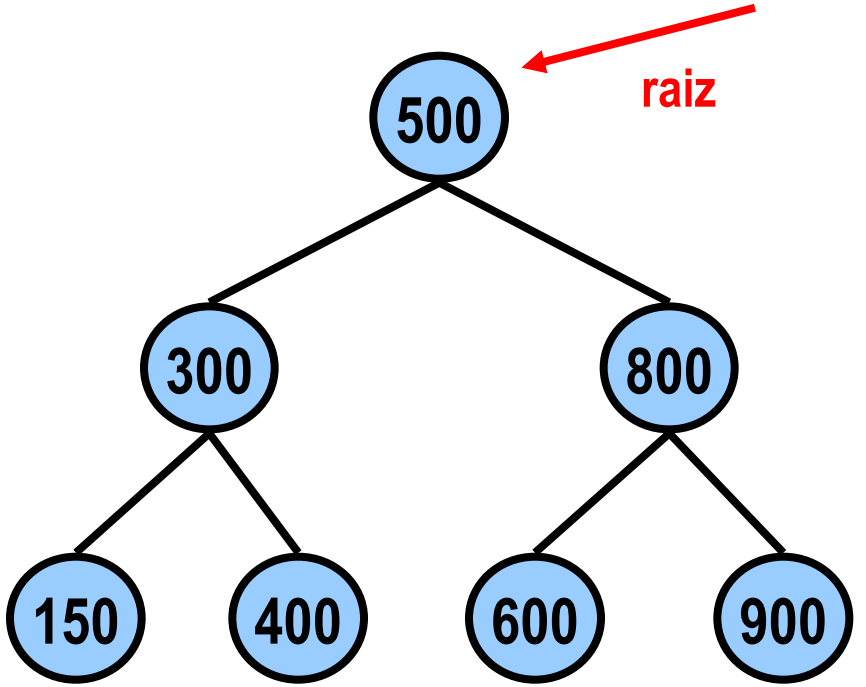
Ordem é definida pela chave



ÁRVORES BINÁRIAS DE BUSCA

Uma árvore binária T é uma árvore binária de busca se:

- Chaves da subárvore esquerda de T são menores do que chave da raiz de T; e
- Chaves da subárvore da direita de T são maiores do que a chave da raiz de T; e
- Subárvores da esquerda e da direita de T são árvores binárias de busca

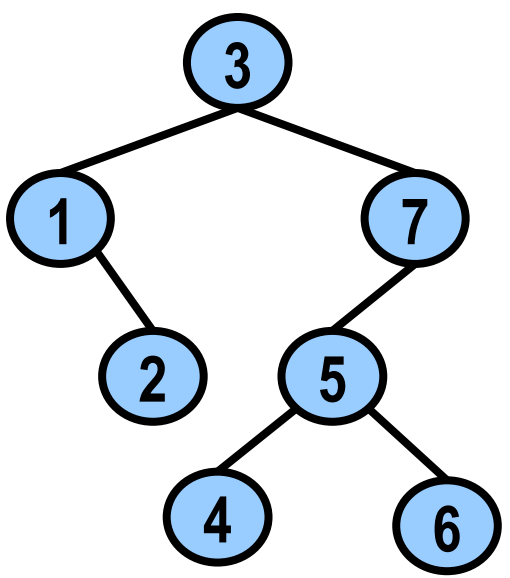


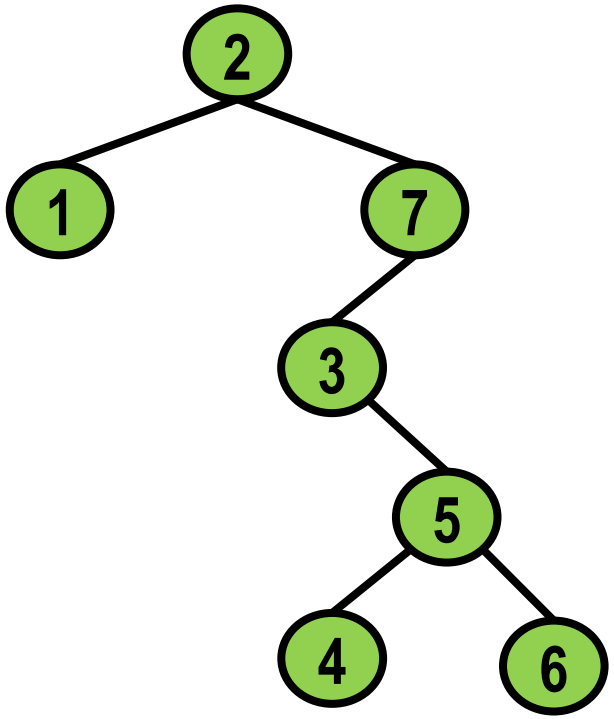
ÁRVORES BINÁRIAS DE BUSCA

Para um mesmo conjunto de chaves, existem várias árvores binárias de busca possíveis

Exemplos para o conjunto de chaves:

{1, 2, 3, 4, 5, 6, 7}





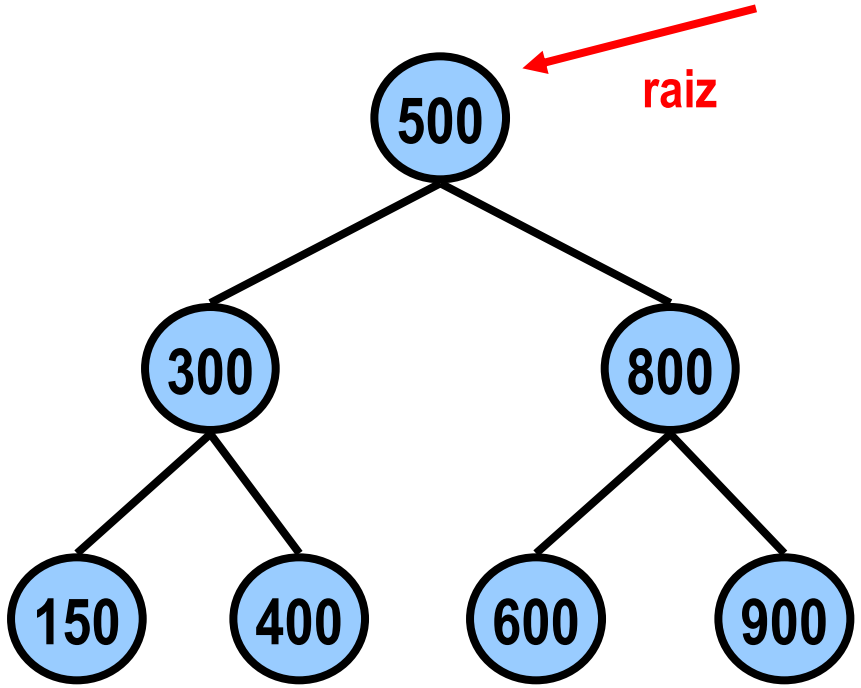
OPERAÇÕES

Buscar nó com determinada chave

Inserir novo nó

Remover nó

Operações devem preservar a ordem entre os nós!



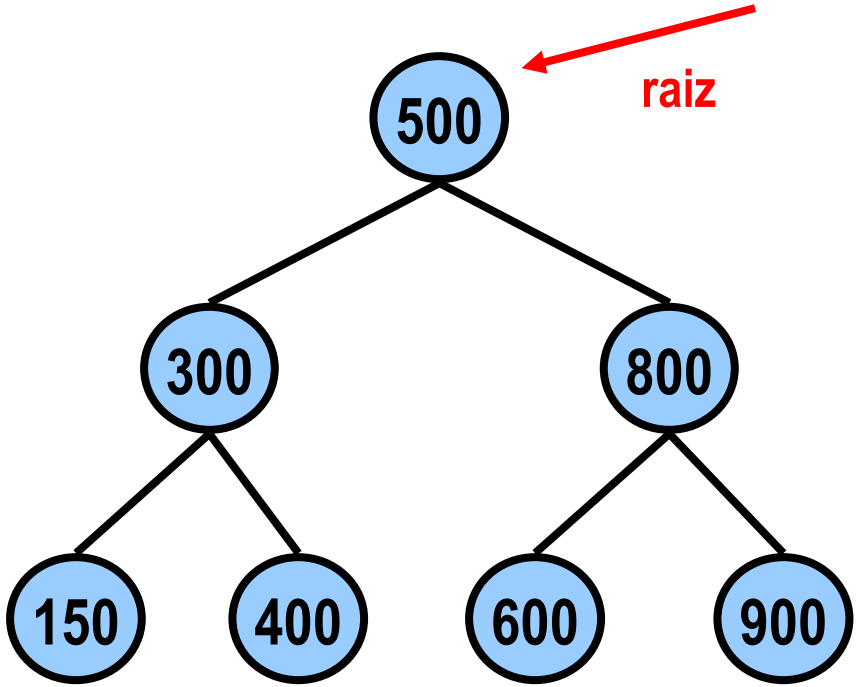
BUSCA POR NÓ COM CHAVE X

Em qualquer nó:

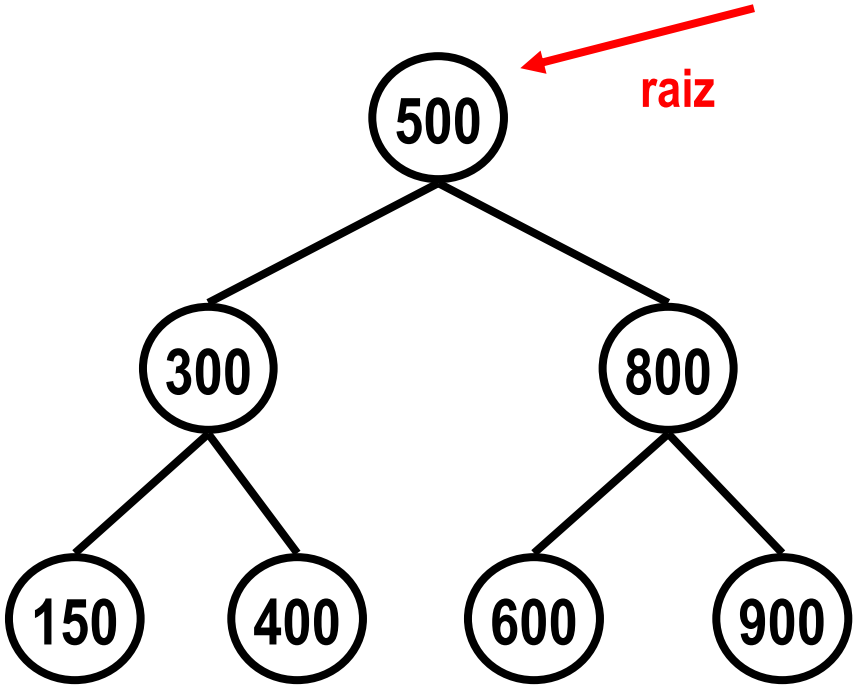
X = Chave

X > Chave

X < Chave



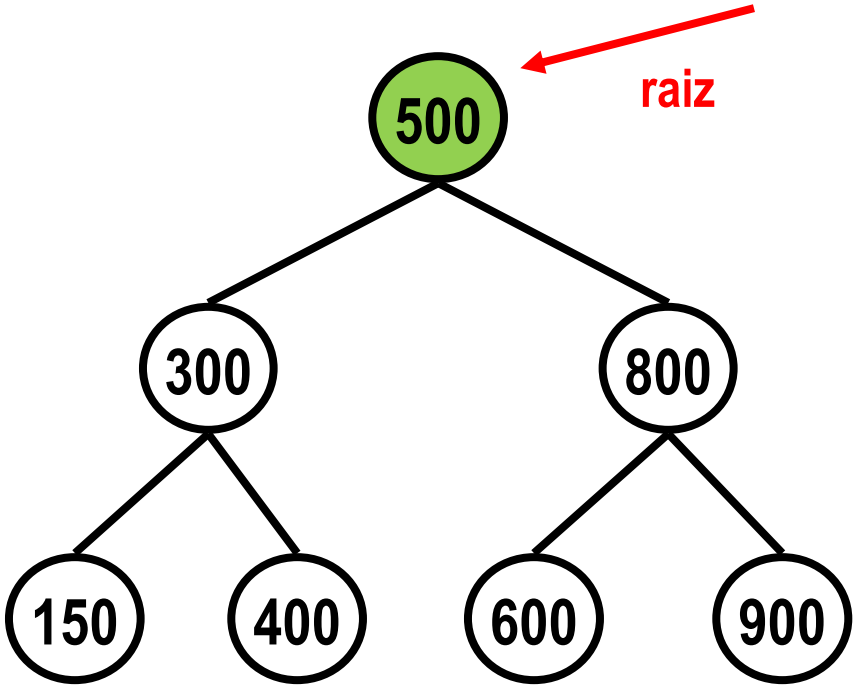
EXEMPLO: BUSCAR POR 600



EXEMPLO: BUSCAR POR 600

600 > 500

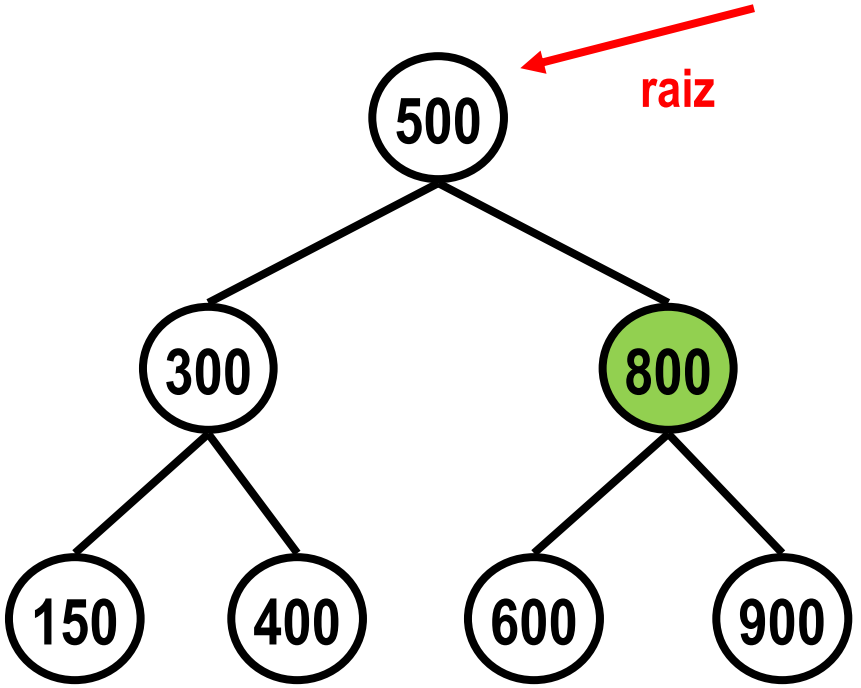
Ir para direita



EXEMPLO: BUSCAR POR 600

600 < 800

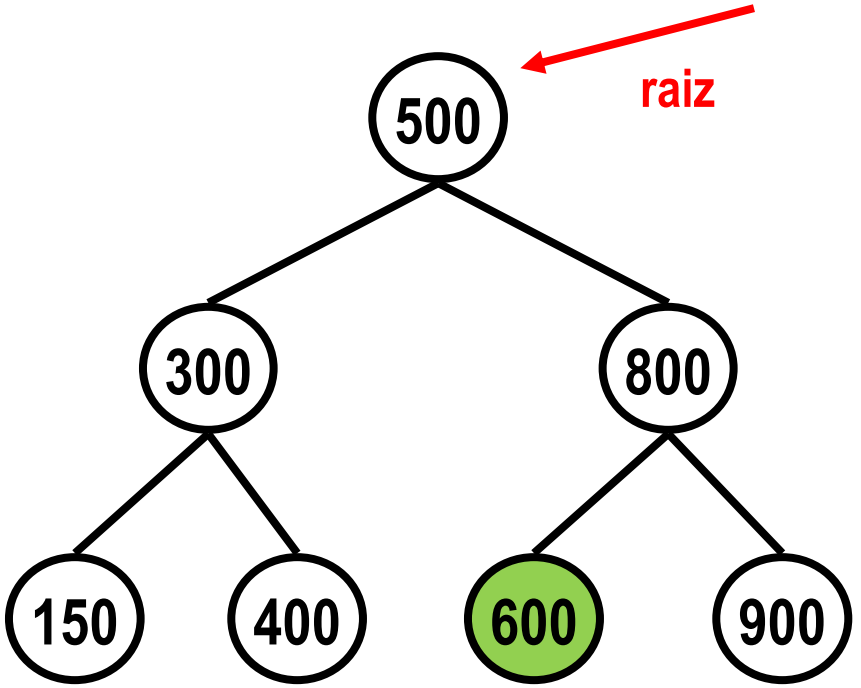
Ir para a esquerda



EXEMPLO: BUSCAR POR 600

600 = 600

Achou



EXEMPLO: BUSCAR POR 200

200 < 500

Ir para esquerda

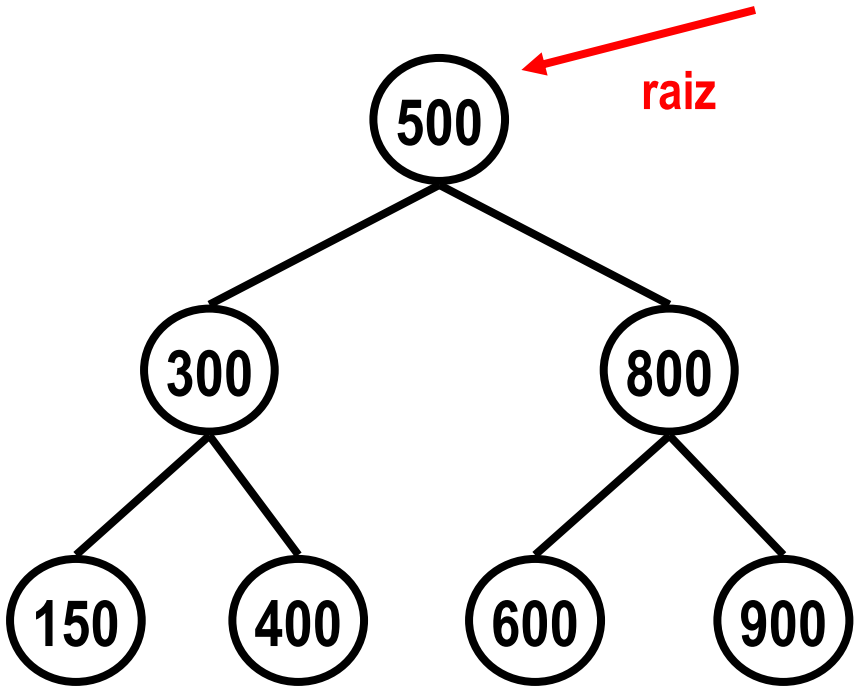
200 < 300

Ir para esquerda

200 > 150

Ir para a direita

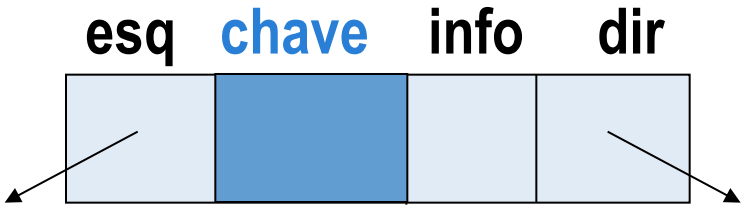
NULL: chave não encontrada



BUSCA POR NÓ COM DETERMINADA CHAVE

```
/* representação dos nós de a */
typedef struct sNoA {
   char info;
   int chave;
   struct sNoA* esq;
   struct sNoA* dir;
} TNoA;

TNoA* busca(TNoA *no, int chave) {
//Recebe endereço da raiz e chave procurada.
Se encontrar, retorna ponteiro p/ nó
encontrado.
//Caso contrário, retorna NULO
}
```



Fazer agora!

IMPLEMENTAÇÃO ITERATIVA

```
TNoA* busca(TNoA *no, int chave) {
    TNoA *aux = no;
    while (aux != NULL) {
        if (aux->chave == chave )
            return aux; //achou retorna o ponteiro para o nó
        else
        if (aux->chave > chave)
            aux = aux->esq;
        else
            aux = aux->dir;
    }
    return NULL; //não achou, retorna null
}
```

IMPLEMENTAÇÃO RECURSIVA

```
TNoA* buscaRecursiva(TNoA *no, int chave) {
    if (no == NULL)
        return NULL;
    else if (no->chave == chave)
        return no;
    else if (no->chave > chave)
        return buscaRecursiva(no->esq, chave);
    else
        return buscaRecursiva(no->dir, chave);
}
```

COMPLEXIDADE

Em cada chamada da função busca, é efetuado um número constante de operações.

A complexidade da busca é igual ao número de chamadas da função.

No pior caso (quando chave buscada está na folha), a complexidade é a altura da árvore.

Complexidade mínima **de pior caso** ocorre para árvore completa, onde altura é log n (n é o número de nós da árvore).

Portanto, o ideal é que a árvore binária de busca seja **o mais balanceada possível**.

INSERÇÃO

Se a árvore for vazia, instala o novo nó na raiz

Se não for vazia, compara a chave com a chave da raiz:

- se for menor, instala o nó na sub-árvore da esquerda
- caso contrário, instala o nó na sub-árvore da direita

IMPORTANTE: nó é sempre inserido como uma folha

INSERÇÃO

Se a árvore for vazia, instala o novo nó na raiz

Se não for vazia, compara a chave com a chave da raiz:

- se for menor, instala o nó na sub-árvore da esquerda
- caso contrário, instala o nó na sub-árvore da direita

Ordem de Inserção:
500 – 800 – 300 - 400

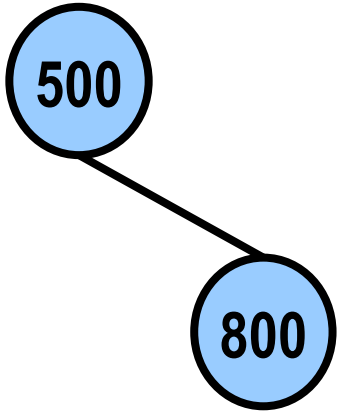
INSERÇÃO

Se a árvore for vazia, instala o novo nó na raiz

Se não for vazia, compara a chave com a chave da raiz:

- se for menor, instala o nó na sub-árvore da esquerda
- caso contrário, instala o nó na sub-árvore da direita

Ordem de Inserção:
500 – 800 – 300 - 400



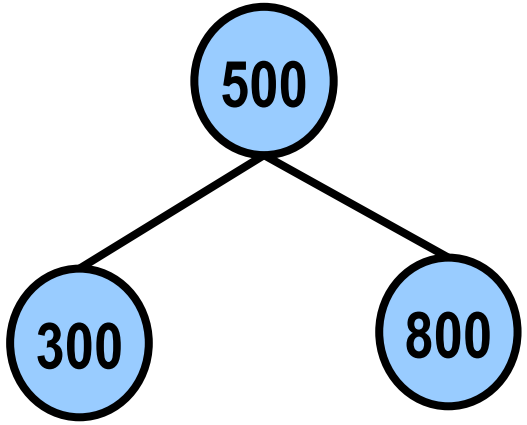
INSERÇÃO

Se a árvore for vazia, instala o novo nó na raiz

Se não for vazia, compara a chave com a chave da raiz:

- se for menor, instala o nó na sub-árvore da esquerda
- caso contrário, instala o nó na sub-árvore da direita

Ordem de Inserção:
500 – 800 – 300 - 400



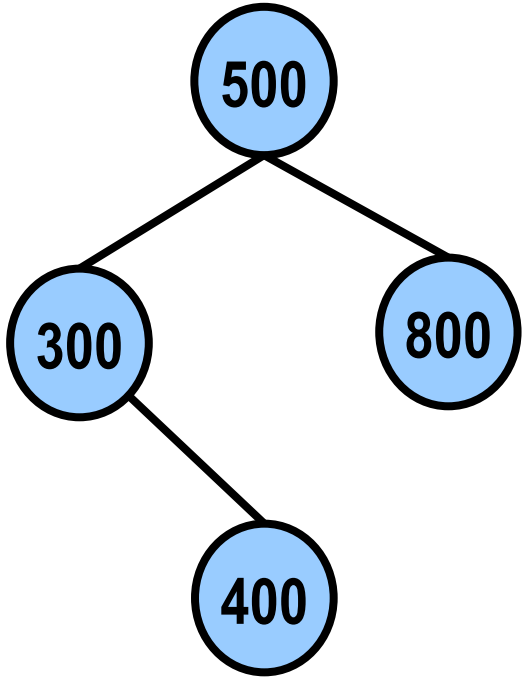
INSERÇÃO

Se a árvore for vazia, instala o novo nó na raiz

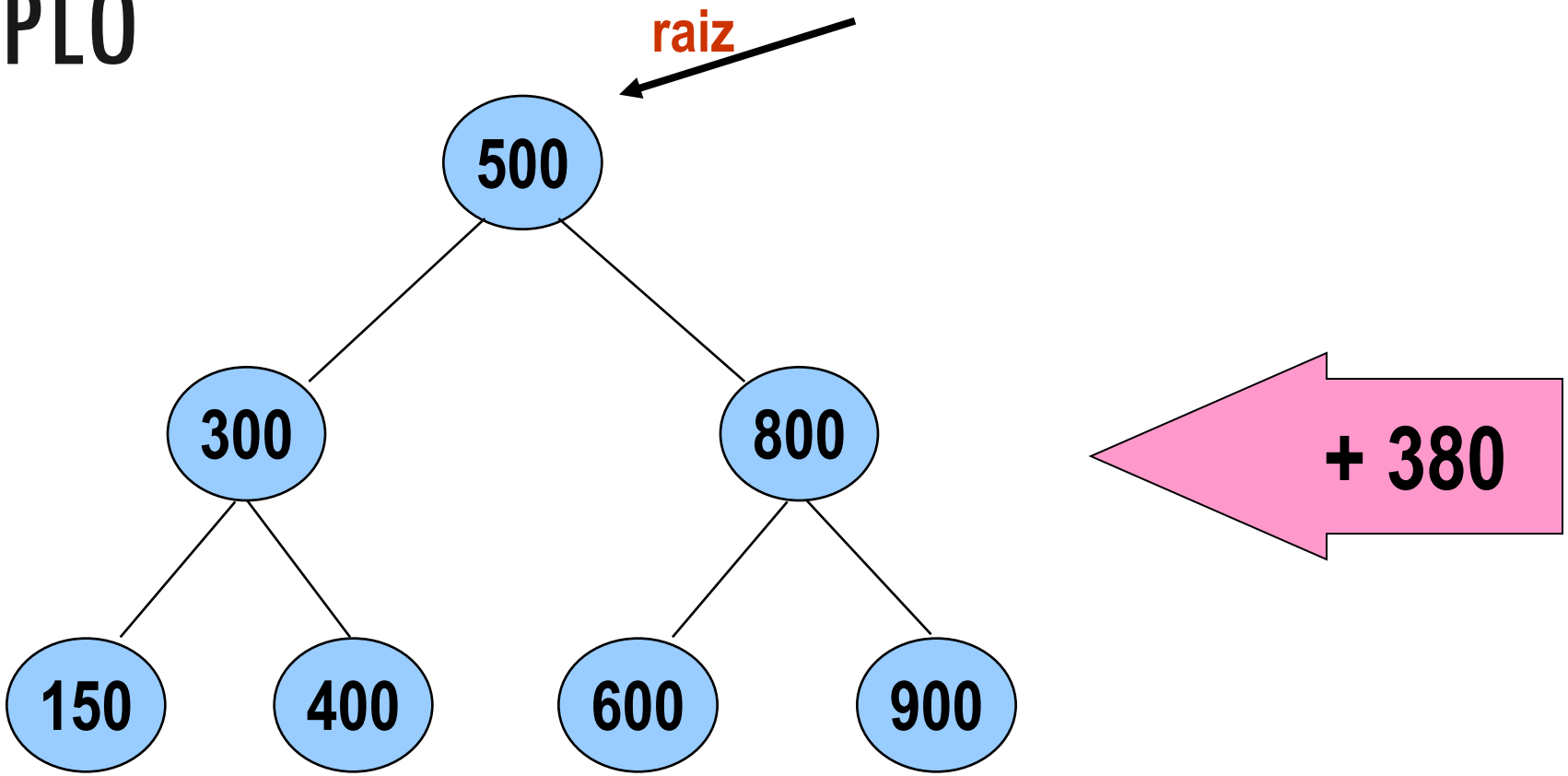
Se não for vazia, compara a chave com a chave da raiz:

- se for menor, instala o nó na sub-árvore da esquerda
- caso contrário, instala o nó na sub-árvore da direita

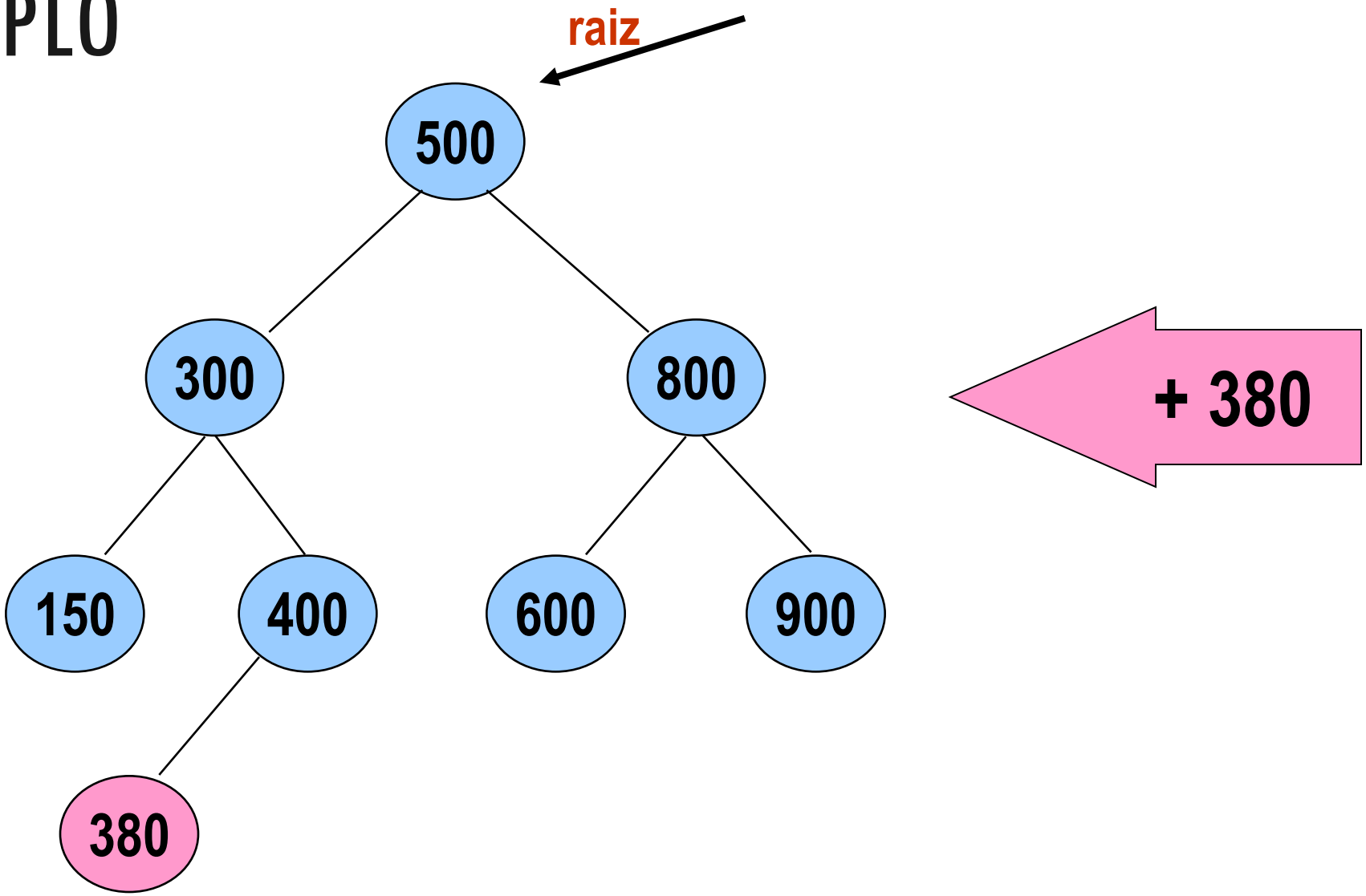
Ordem de Inserção:
500 – 800 – 300 - 400



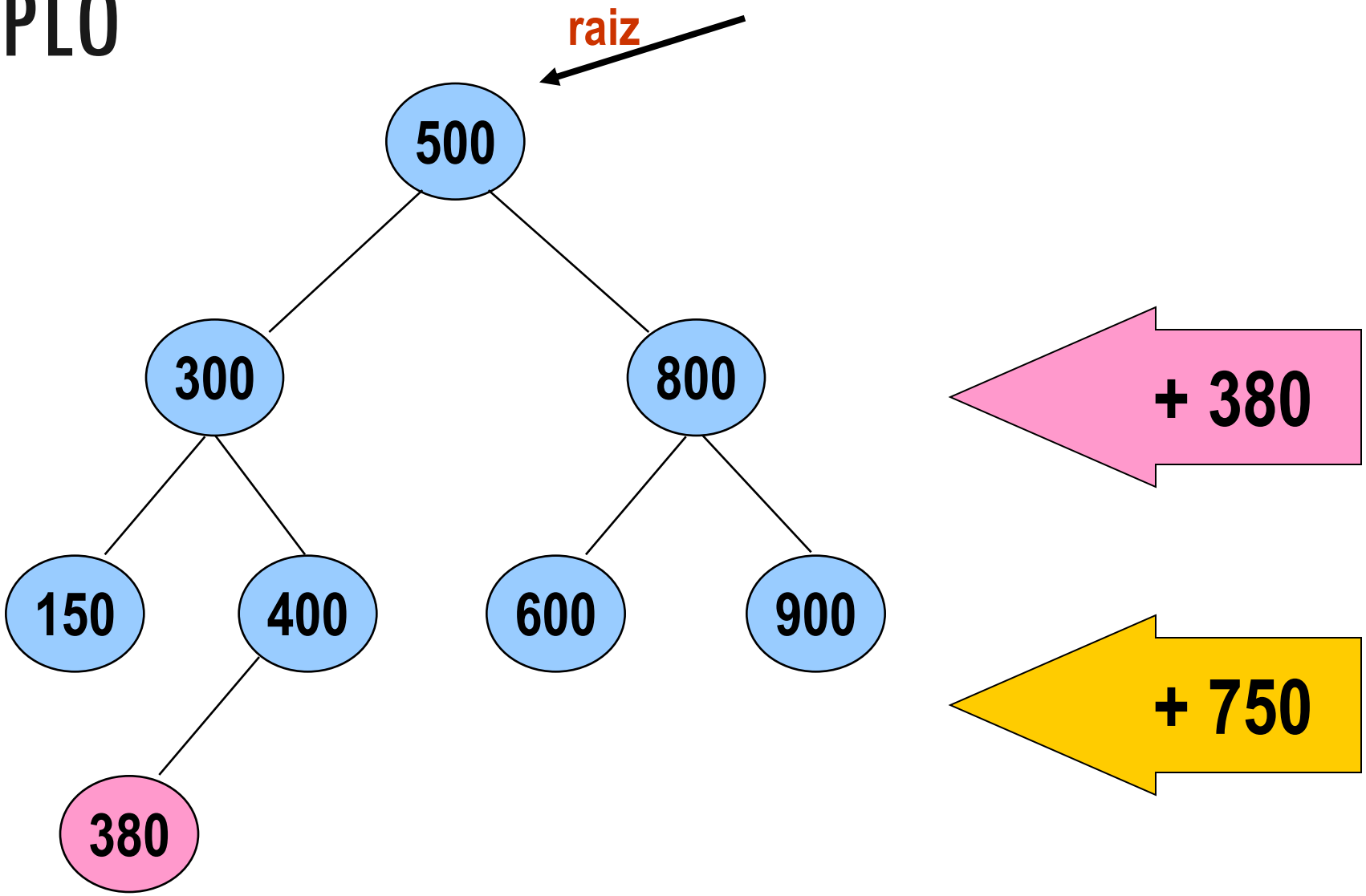
EXEMPLO



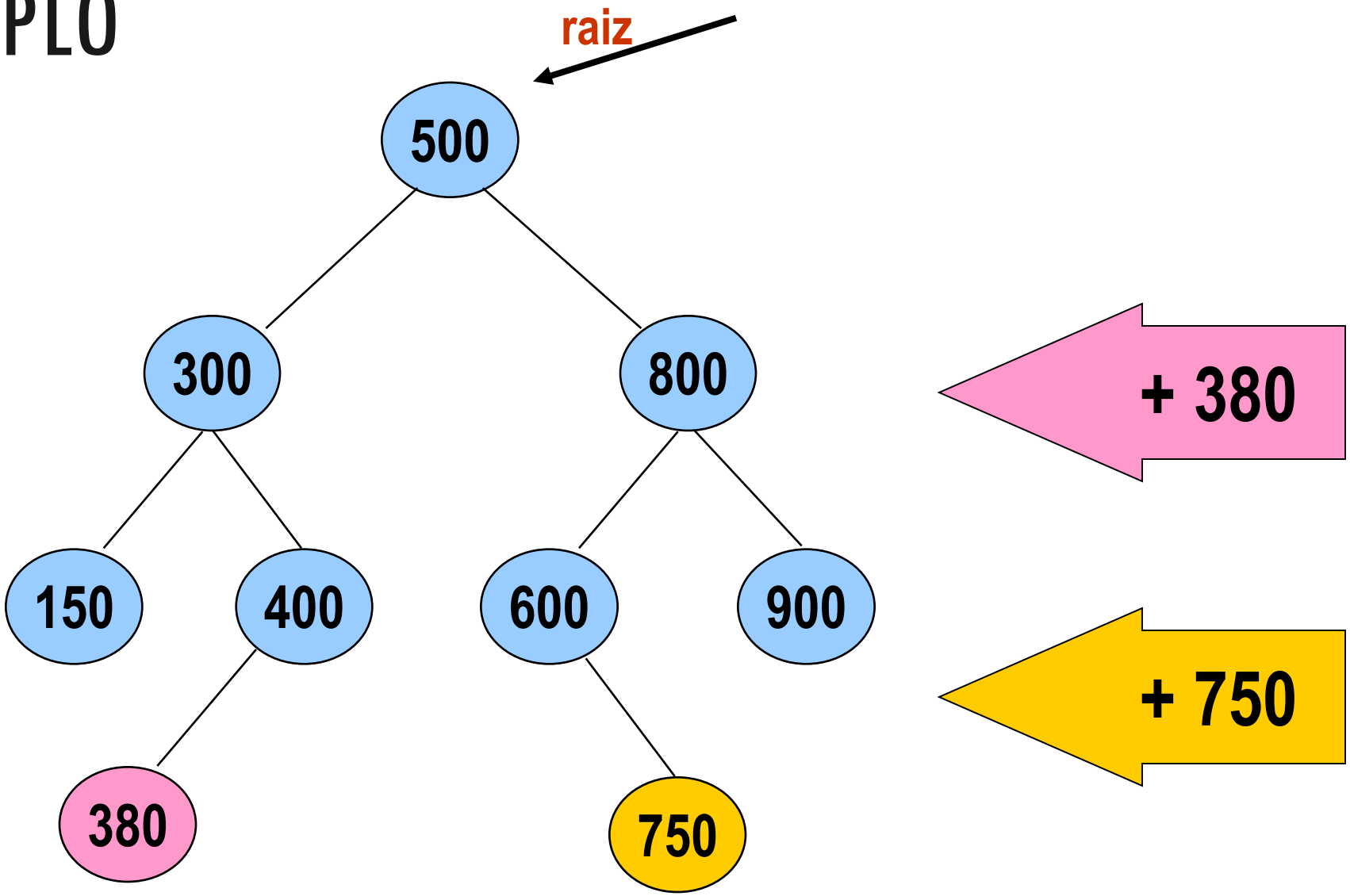
EXEMPLO



EXEMPLO



EXEMPLO



EXERCÍCIOS

1. Inserir em uma ABB inicialmente vazia, os seguintes valores:

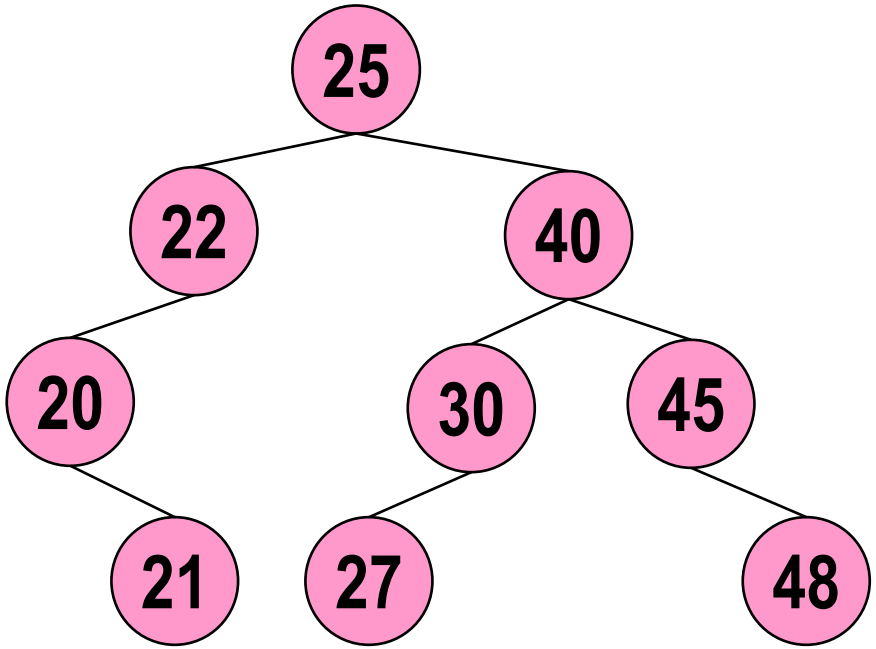
   25, 22, 40, 30, 45, 27, 20, 21, 48

2. Inserir em uma ABB inicialmente vazia, os seguintes valores:

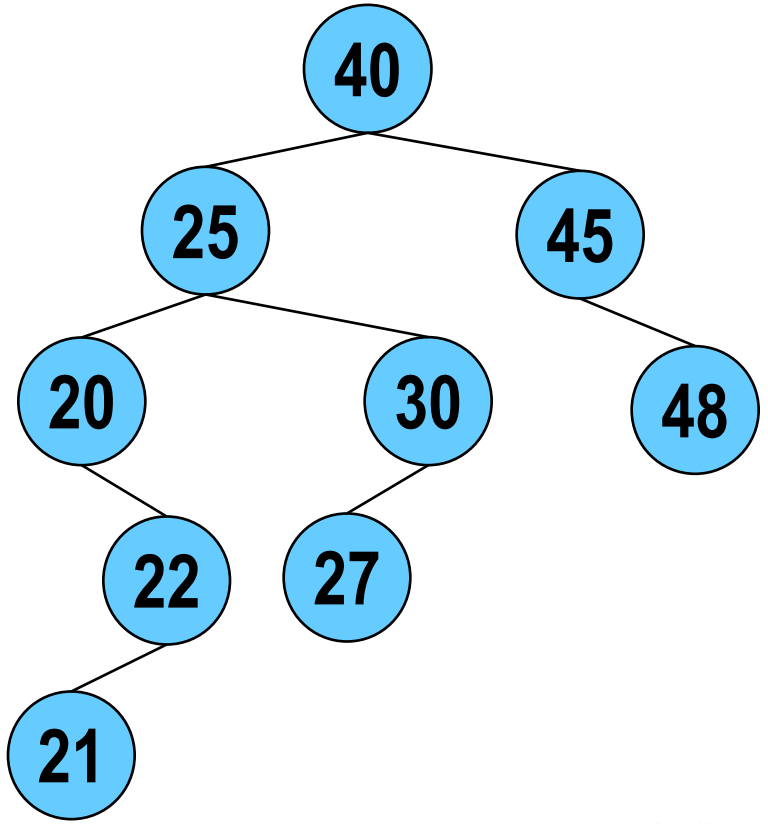
   40, 25, 20, 30, 45, 27, 22, 21, 48

INSERÇÃO

25 22 40 30 45 27 20 21 48



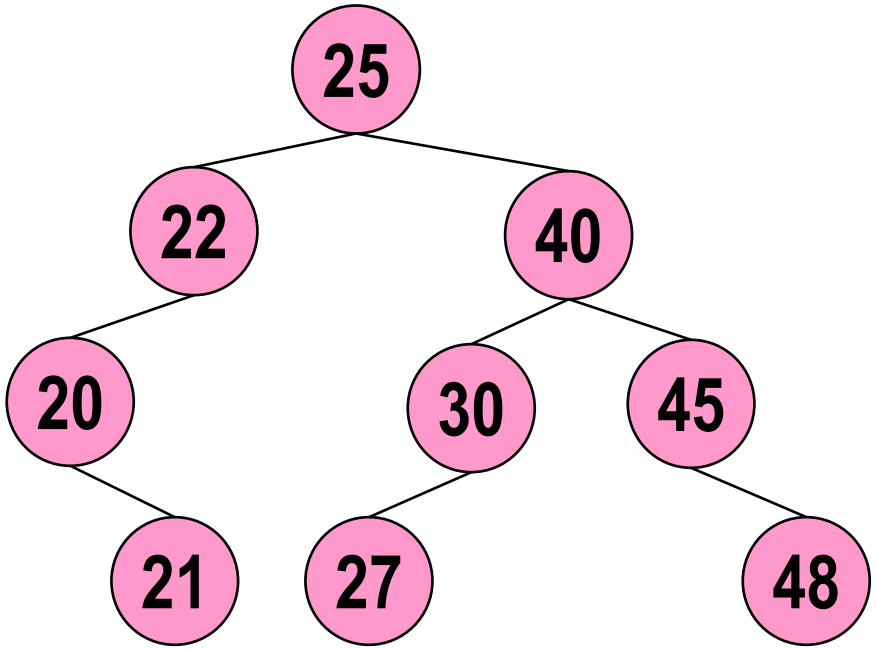
40 25 20 30 45 27 22 21 48



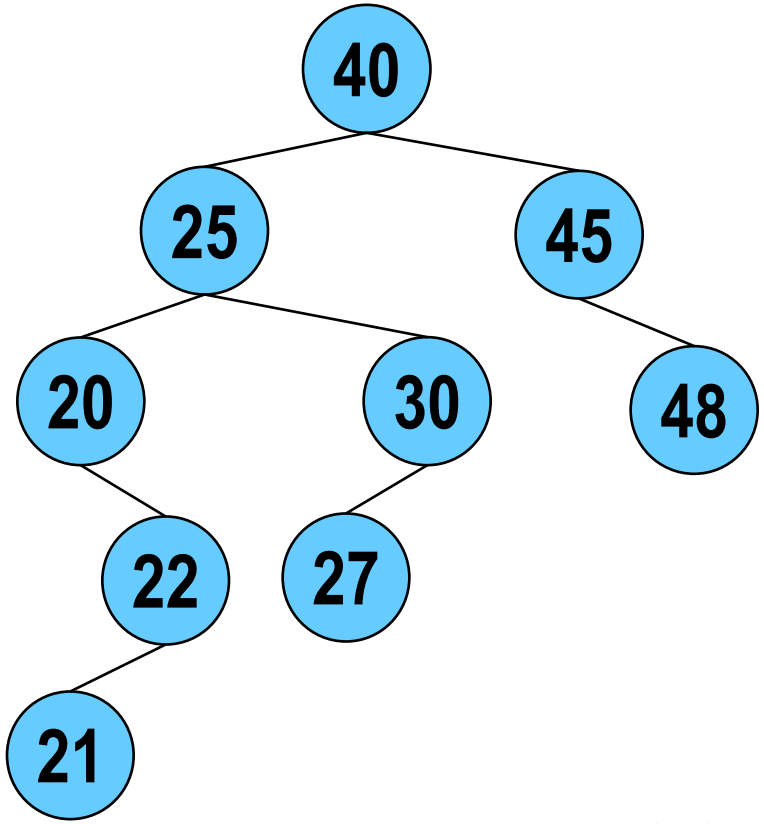
INSERÇÃO

A árvore gerada depende da ordem de inserção dos nós

25 22 40 30 45 27 20 21 48



40 25 20 30 45 27 22 21 48



IMPLEMENTAÇÃO DE INSERÇÃO

```
TNoA *insere(TNoA *no, int chave) {
    if (no == NULL) {
        no = (TNoA *) malloc(sizeof(TNoA));
        no->chave = chave;
        no->esq = NULL;
        no->dir = NULL;
    } else if (chave < (no->chave))
               no->esq = insere(no->esq, chave);
           else if (chave > (no->chave))
                   no->dir = insere(no->dir, chave);
                else {
                   printf("Inserção inválida! "); // chave já existe
                   exit(1);
                }
    return no;
}
```

IMPLEMENTAÇÃO DE INSERÇÃO

```
TNoA *insere(TNoA *no, int chave) {
    if (no == NULL) {
        no = (TNoA *) malloc(sizeof(TNoA));
        no->chave = chave;
        no->esq = NULL;
        no->dir = NULL;
    } else if (chave < (no->chave))
              no->esq = insere(no->esq, chave);
           else if (chave > (no->chave))
                   no->dir = insere(no->dir, chave);
                else {
                   printf("Inserção inválida! "); // chave já existe
                   exit(1);
                }
    return no;
}
```

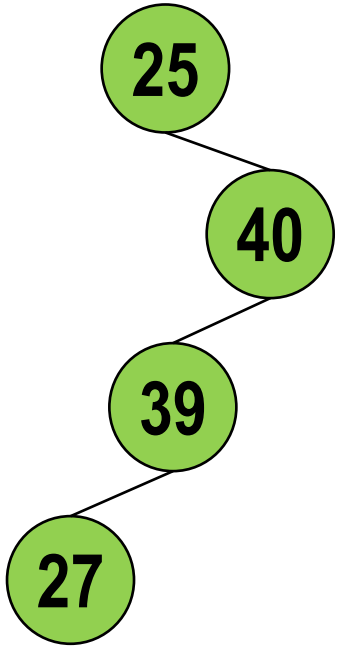
Árvore Binária de Busca não pode ter chave duplicada

PROBLEMA

A ordem em que as chaves são inseridas numa árvore de busca binária pode fazer com que uma árvore se deteriore, ficando com altura muito grande.

Exemplo:

25 40 39 27



CRIAÇÃO DE ÁRVORE BINÁRIA DE BUSCA MAIS BALANCEADA POSSÍVEL

Sabendo disso, é possível reordenar as chaves de entrada de forma a obter uma árvore o mais balanceada possível.

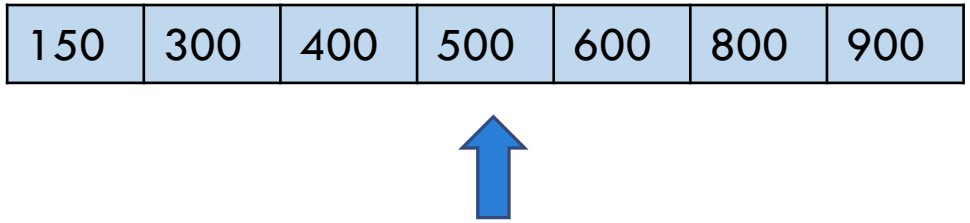
Algoritmo:

- Seja v um vetor ORDENADO contendo as chaves a serem inseridas
- Inserir a chave do meio
- Chamar recursivamente para os dois pedaços que sobraram

CRIAÇÃO DE ÁRVORE BINÁRIA DE BUSCA MAIS BALANCEADA POSSÍVEL

Algoritmo:

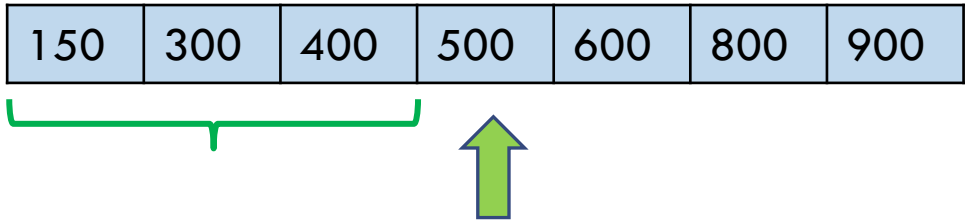
- Seja v um vetor ORDENADO contendo as chaves a serem inseridas
- Inserir a chave do meio
- Chamar recursivamente para os dois pedaços que sobraram (esquerda e direita)



CRIAÇÃO DE ÁRVORE BINÁRIA DE BUSCA MAIS BALANCEADA POSSÍVEL

Algoritmo:

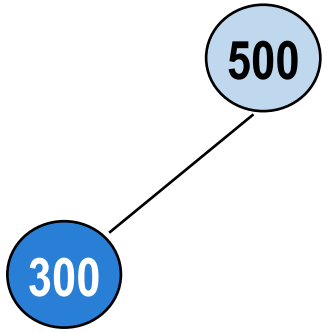
- Seja v um vetor ORDENADO contendo as chaves a serem inseridas
- Inserir a chave do meio
- Chamar recursivamente para os dois pedaços que sobraram (esquerda e direita)

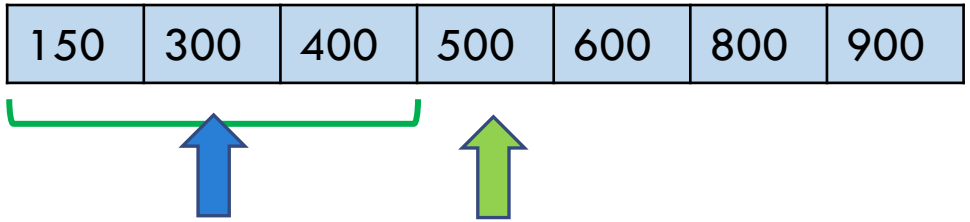


CRIAÇÃO DE ÁRVORE BINÁRIA DE BUSCA MAIS BALANCEADA POSSÍVEL

Algoritmo:

- Seja v um vetor ORDENADO contendo as chaves a serem inseridas
- Inserir a chave do meio
- Chamar recursivamente para os dois pedaços que sobraram (esquerda e direita)

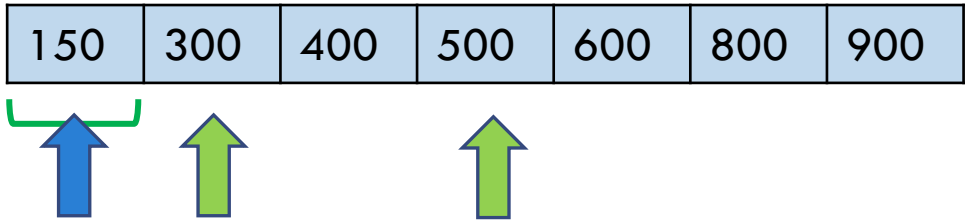


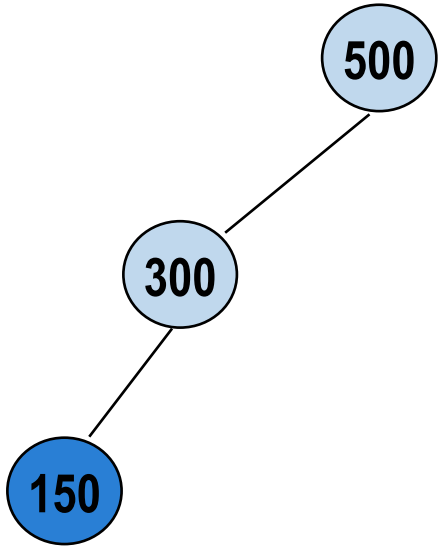


CRIAÇÃO DE ÁRVORE BINÁRIA DE BUSCA MAIS BALANCEADA POSSÍVEL

Algoritmo:

- Seja v um vetor ORDENADO contendo as chaves a serem inseridas
- Inserir a chave do meio
- Chamar recursivamente para os dois pedaços que sobraram (esquerda e direita)

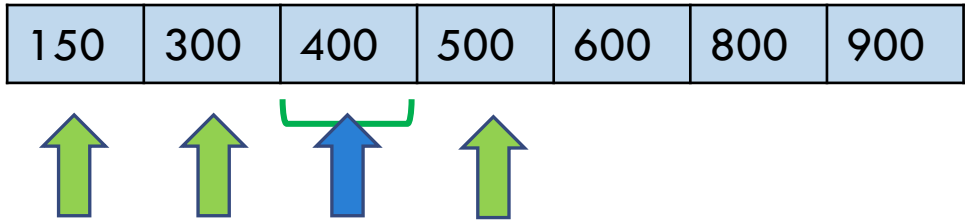


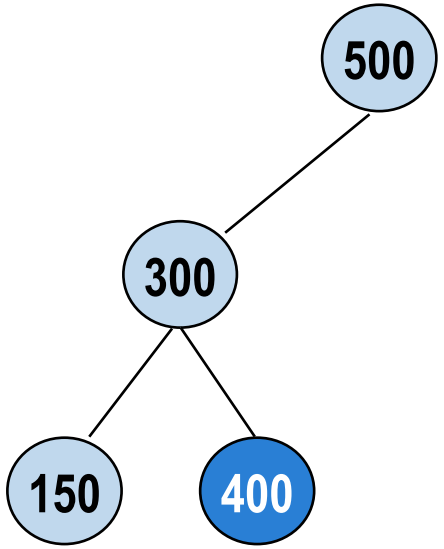


CRIAÇÃO DE ÁRVORE BINÁRIA DE BUSCA MAIS BALANCEADA POSSÍVEL

Algoritmo:

- Seja v um vetor ORDENADO contendo as chaves a serem inseridas
- Inserir a chave do meio
- Chamar recursivamente para os dois pedaços que sobraram (esquerda e direita)

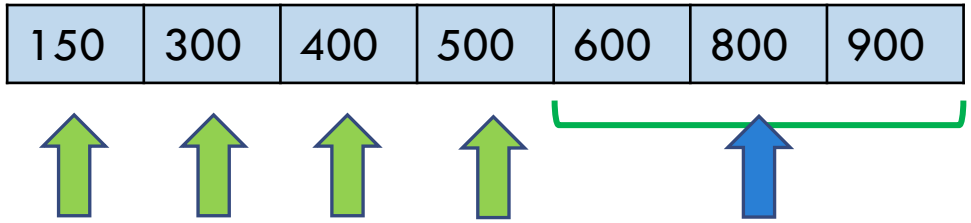


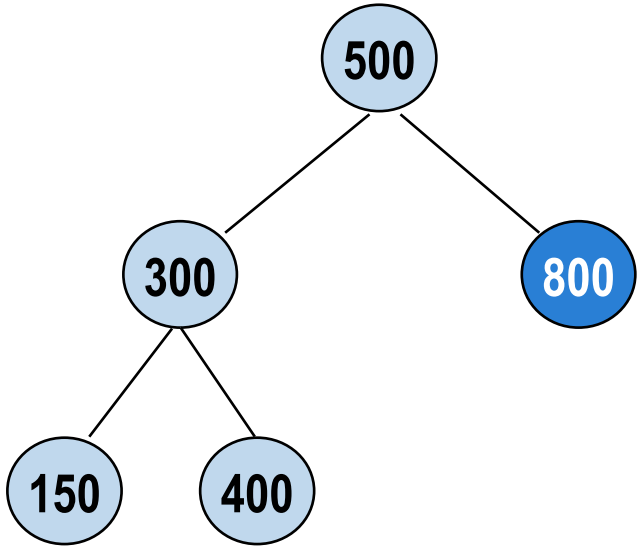


CRIAÇÃO DE ÁRVORE BINÁRIA DE BUSCA MAIS BALANCEADA POSSÍVEL

Algoritmo:

- Seja v um vetor ORDENADO contendo as chaves a serem inseridas
- Inserir a chave do meio
- Chamar recursivamente para os dois pedaços que sobraram (esquerda e direita)

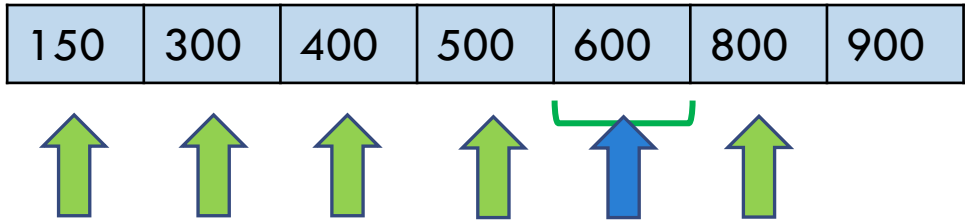


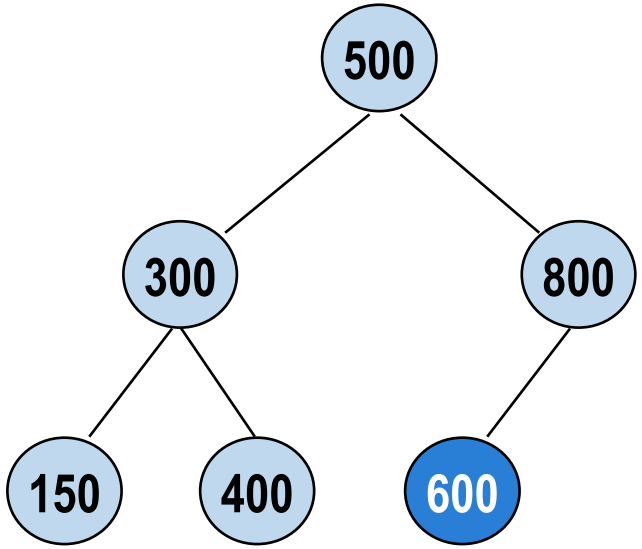


CRIAÇÃO DE ÁRVORE BINÁRIA DE BUSCA MAIS BALANCEADA POSSÍVEL

Algoritmo:

- Seja v um vetor ORDENADO contendo as chaves a serem inseridas
- Inserir a chave do meio
- Chamar recursivamente para os dois pedaços que sobraram (esquerda e direita)

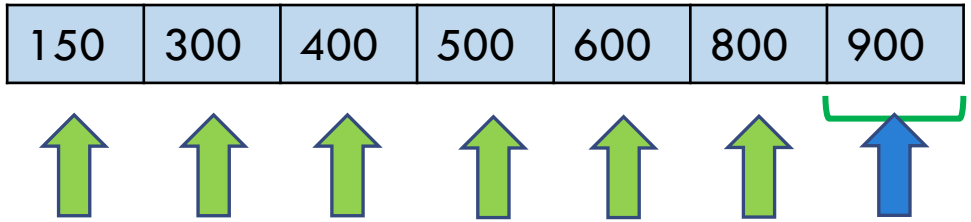


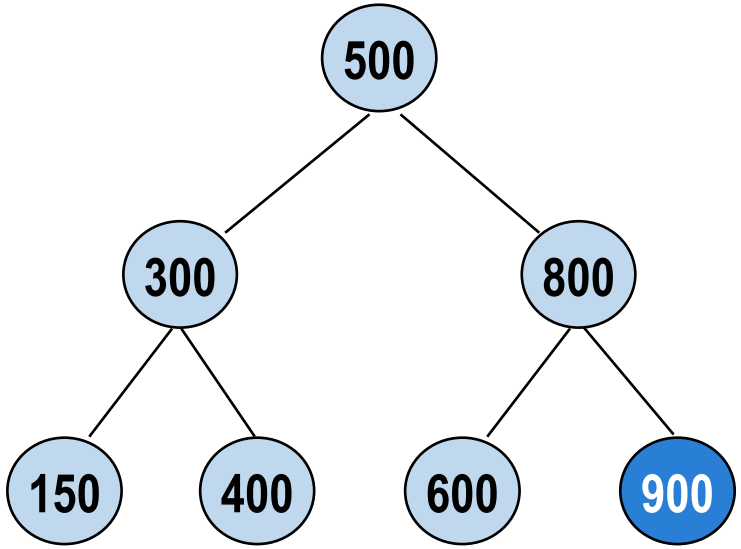


CRIAÇÃO DE ÁRVORE BINÁRIA DE BUSCA MAIS BALANCEADA POSSÍVEL

Algoritmo:

- Seja v um vetor ORDENADO contendo as chaves a serem inseridas
- Inserir a chave do meio
- Chamar recursivamente para os dois pedaços que sobraram (esquerda e direita)

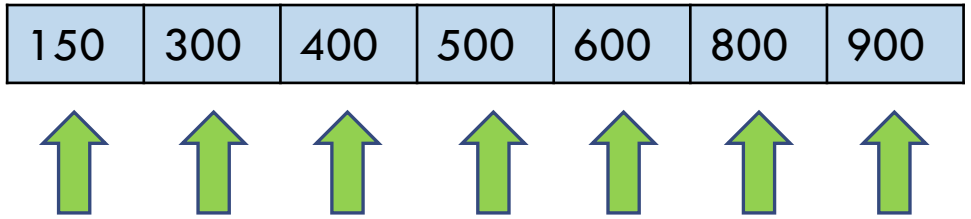


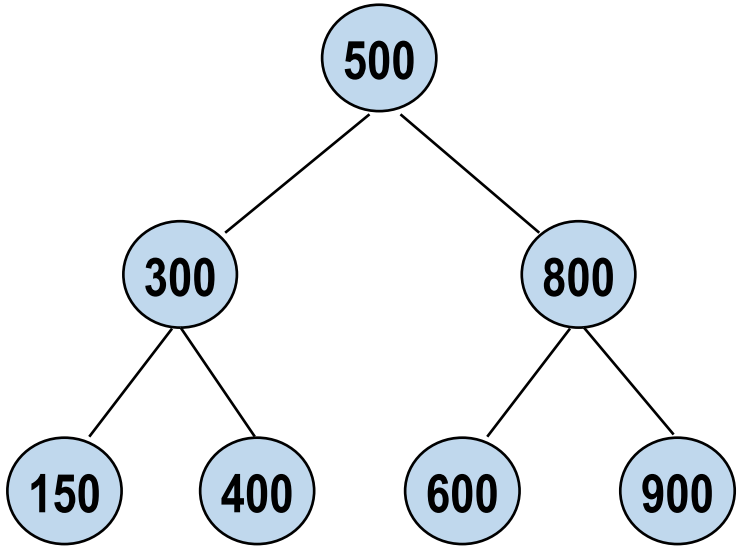


CRIAÇÃO DE ÁRVORE BINÁRIA DE BUSCA MAIS BALANCEADA POSSÍVEL

Algoritmo:

- Seja v um vetor ORDENADO contendo as chaves a serem inseridas
- Inserir a chave do meio
- Chamar recursivamente para os dois pedaços que sobraram (esquerda e direita)



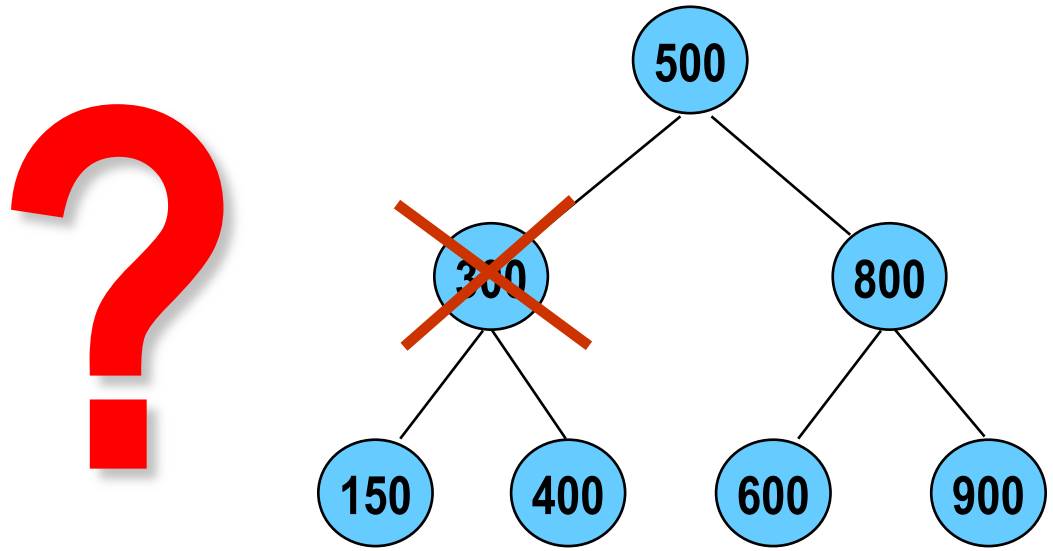


IMPLEMENTAÇÃO

```
void criaArvoreBalanceada(TNoA *raiz, int v[], int inicio, int fim) {
    if (inicio <= fim) {
        int meio = (inicio + fim) / 2;
        raiz = insere(raiz, v[meio]);
        //constroi subárvores esquerda e direita
        criaArvoreBalanceada(raiz, v, inicio, meio - 1);
        criaArvoreBalanceada(raiz, v, meio + 1, fim);
    }
}

int main(void) {
    int tam = 7;
    int v[] = {150, 300, 400, 500, 600, 800, 900};
    TNoA *raiz;
    raiz = NULL;
    criaArvoreBalanceada(raiz,v,0,tam-1);
    imprime(raiz, 0);
};
```

EXCLUSÃO



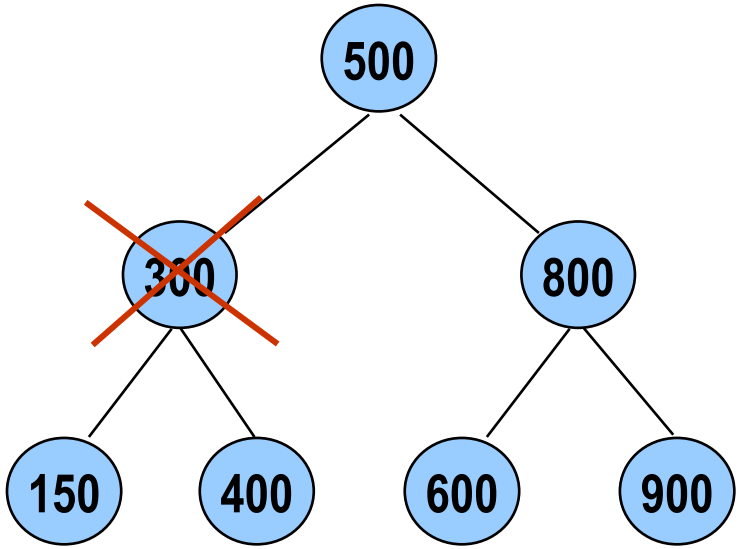
Fonte de Referência:
Celes, W., Cerqueira, R., Rangel, J.L.
Introdução a Estruturas de Dados,
Campus, 1a Edição, 2004.

EXCLUSÃO

Exclusão Física

3 casos

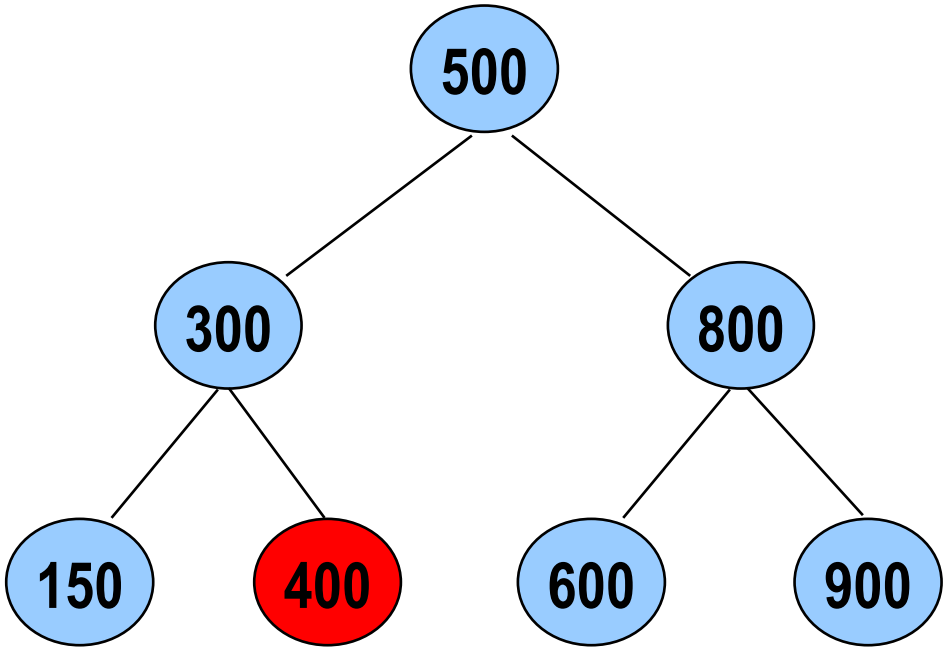
- Nó é folha
- Nó não folha
  - Possui uma subárvore
  - Possui duas subárvores



EXCLUSÃO – CASO 1: NÓ FOLHA

Quando o nó a ser excluído é uma folha, basta removê-lo

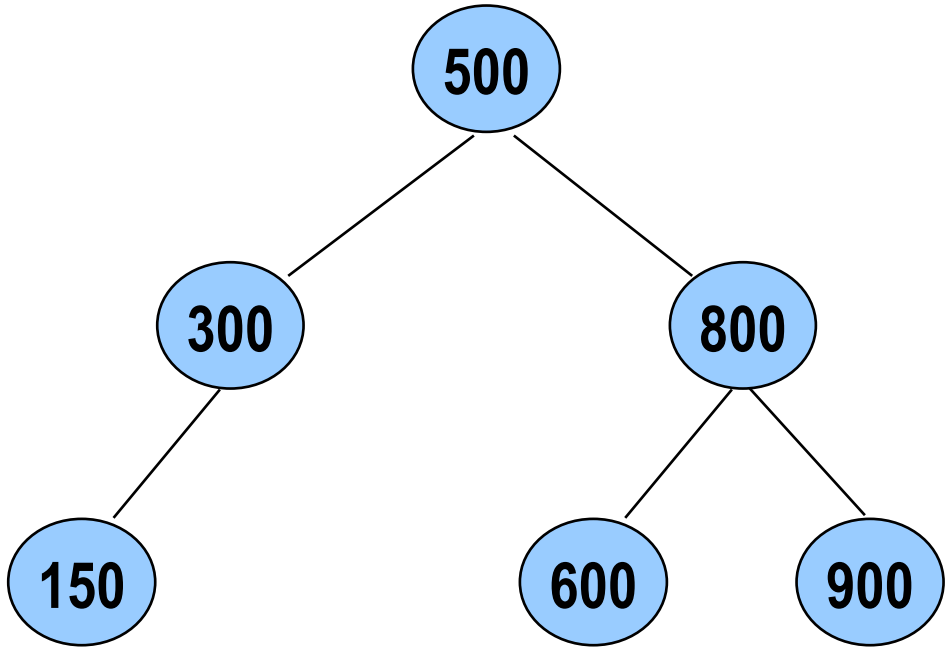
(lembrar de desalocar memória)



EXCLUSÃO – CASO 1: NÓ FOLHA

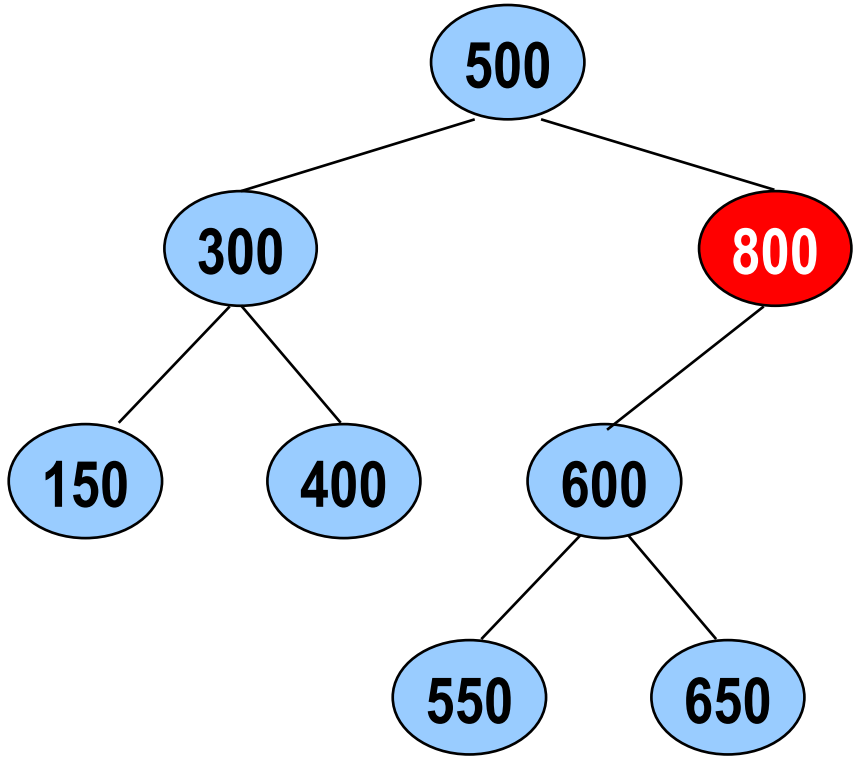
Quando o nó a ser excluído é uma folha, basta removê-lo

(lembrar de desalocar memória)



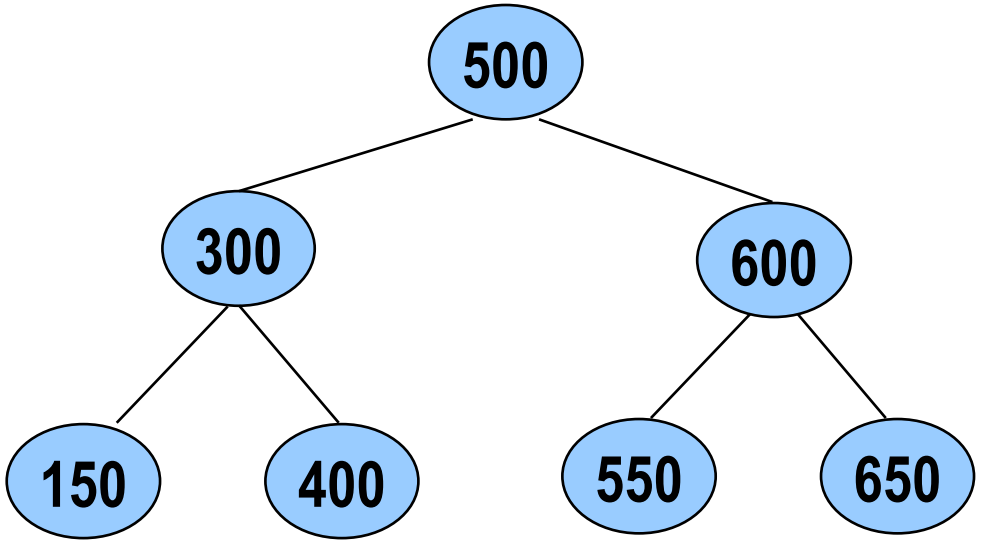
EXCLUSÃO – CASO 2: NÓ INTERNO COM APENAS UMA SUBÁRVORE

Raiz da subárvore passa a ocupar o lugar do nó excluído



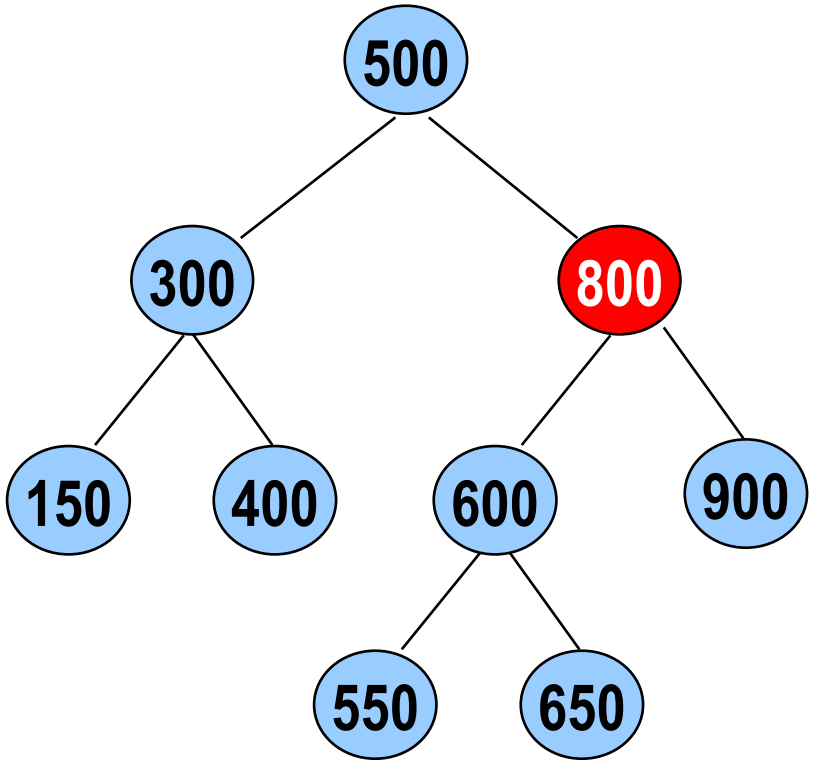
EXCLUSÃO – CASO 2: NÓ INTERNO COM APENAS UMA SUBÁRVORE

Raiz da subárvore passa a ocupar o lugar do nodo excluído



EXCLUSÃO – CASO 3: NÓ POSSUI 2 SUBÁRVORES

Reestruturar a árvore

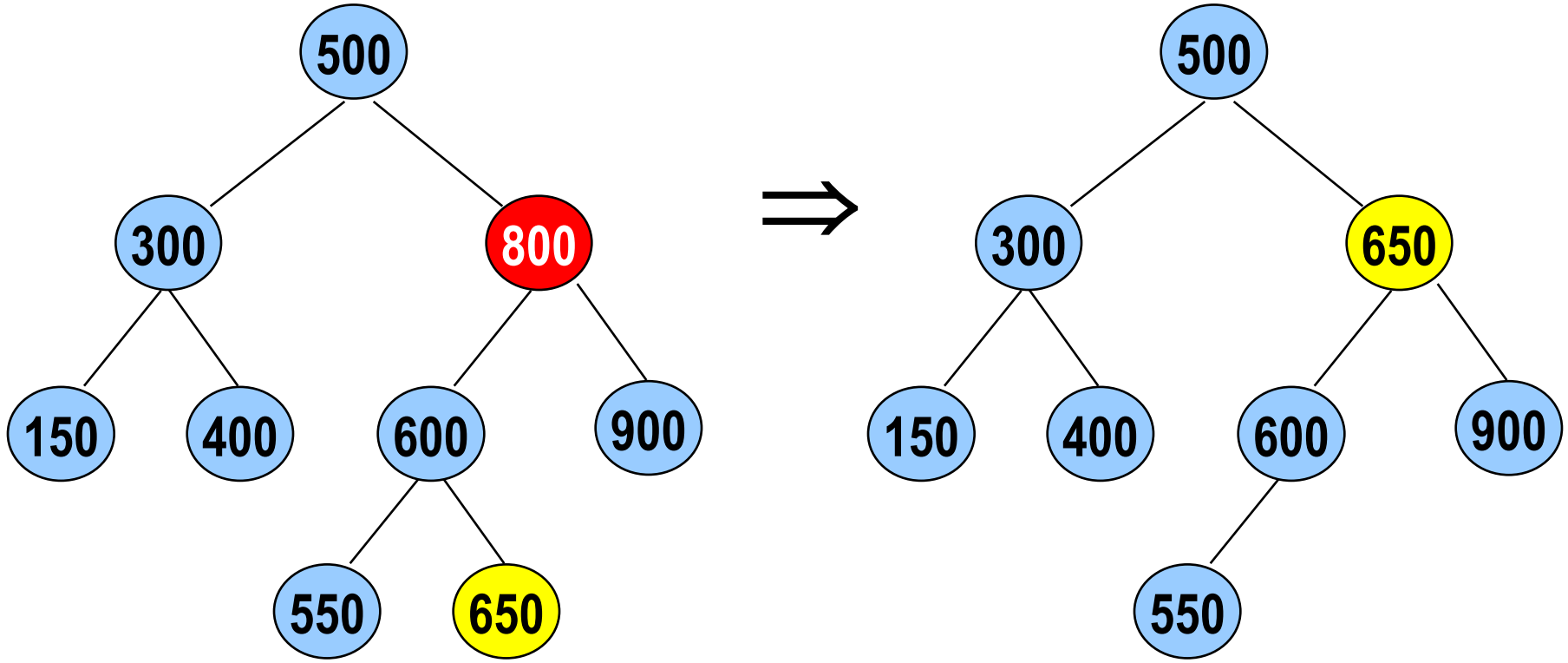


EXCLUSÃO – CASO 3: NÓ POSSUI 2 SUBÁRVORES

Estratégia Recursiva

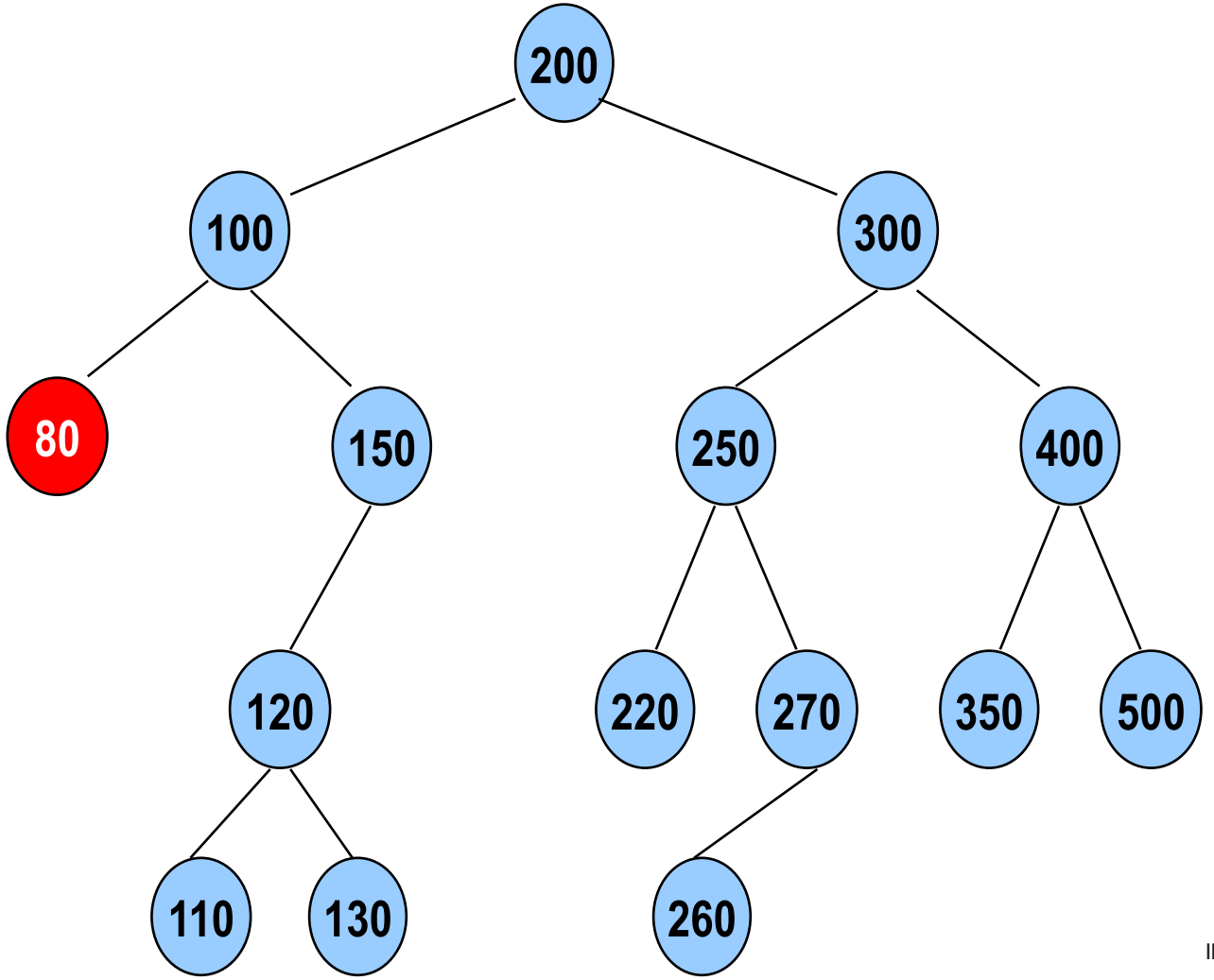
- Trocar o valor do nó a ser removido com
  - valor do nó que tenha a maior chave da sua subárvore à esquerda (será o que adotaremos em aula); OU
  - valor do nó que tenha menor chave da sua subárvore à direita
- Ir à subárvore onde foi feita a troca e remover o nó

EXCLUSÃO – CASO 3: NÓ POSSUI 2 SUBÁRVORES



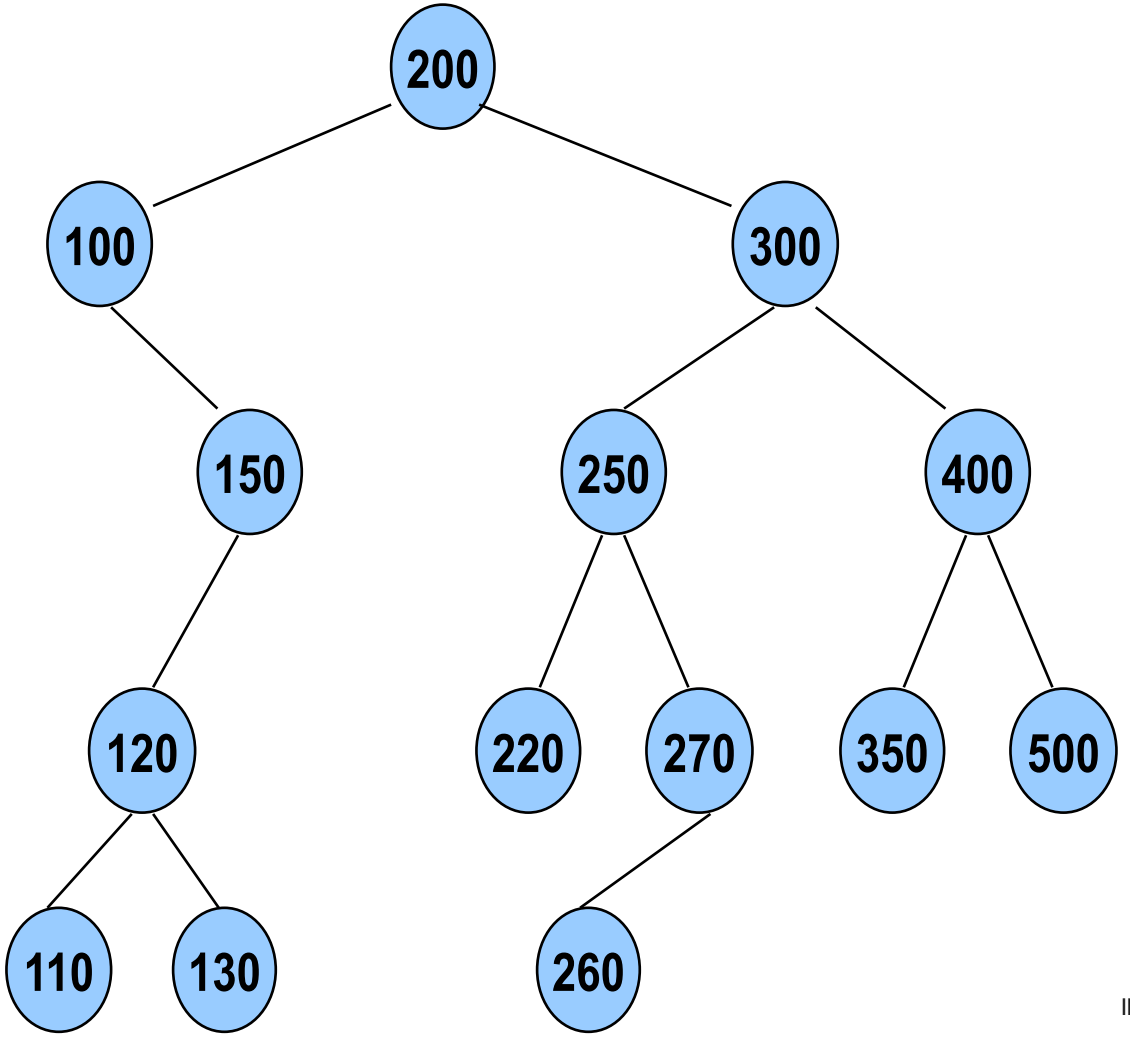
EXCLUSÃO DO NÓ DE CHAVE 80

1°. Caso: nó folha

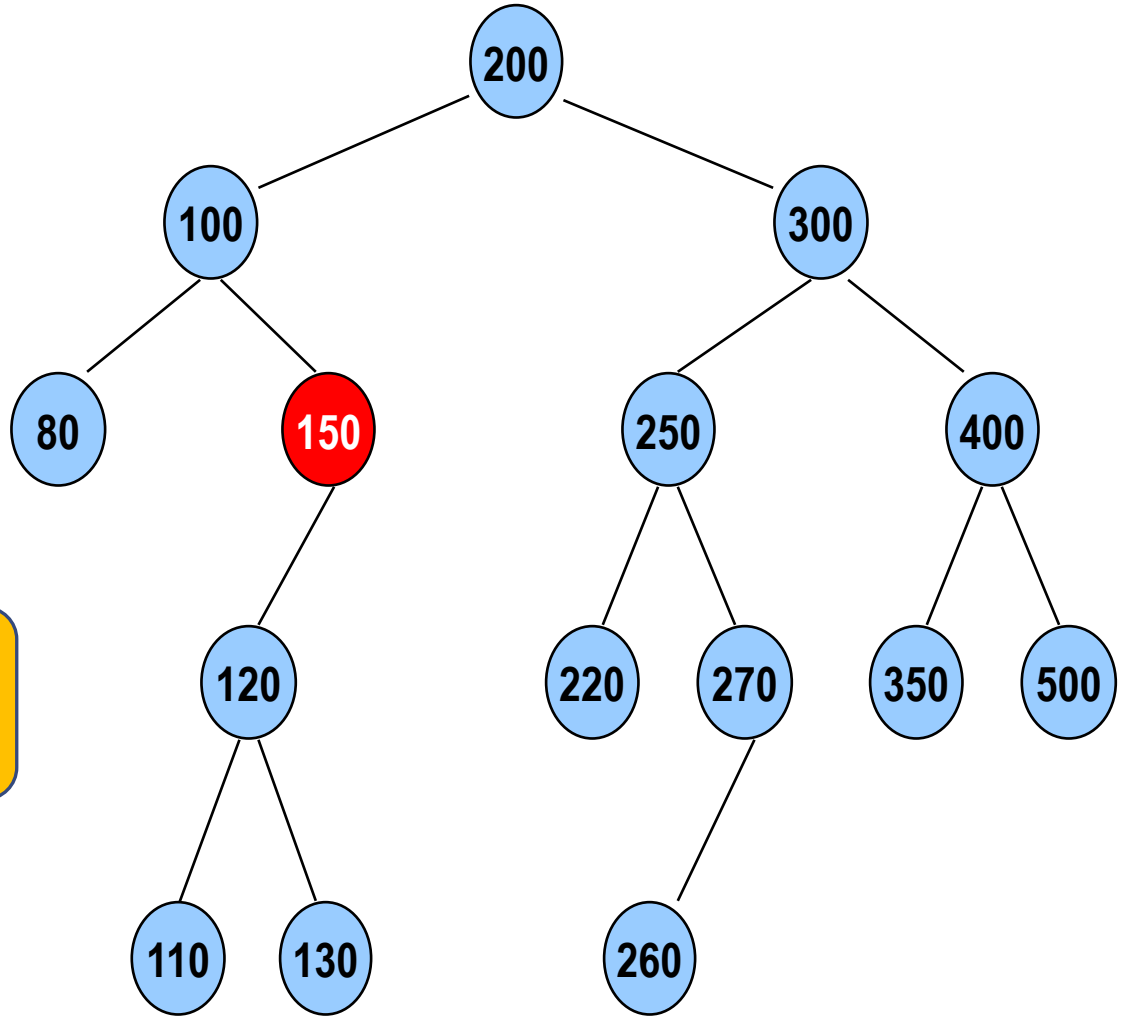


EXCLUSÃO DO NÓ DE CHAVE 80

1°. Caso: nó folha

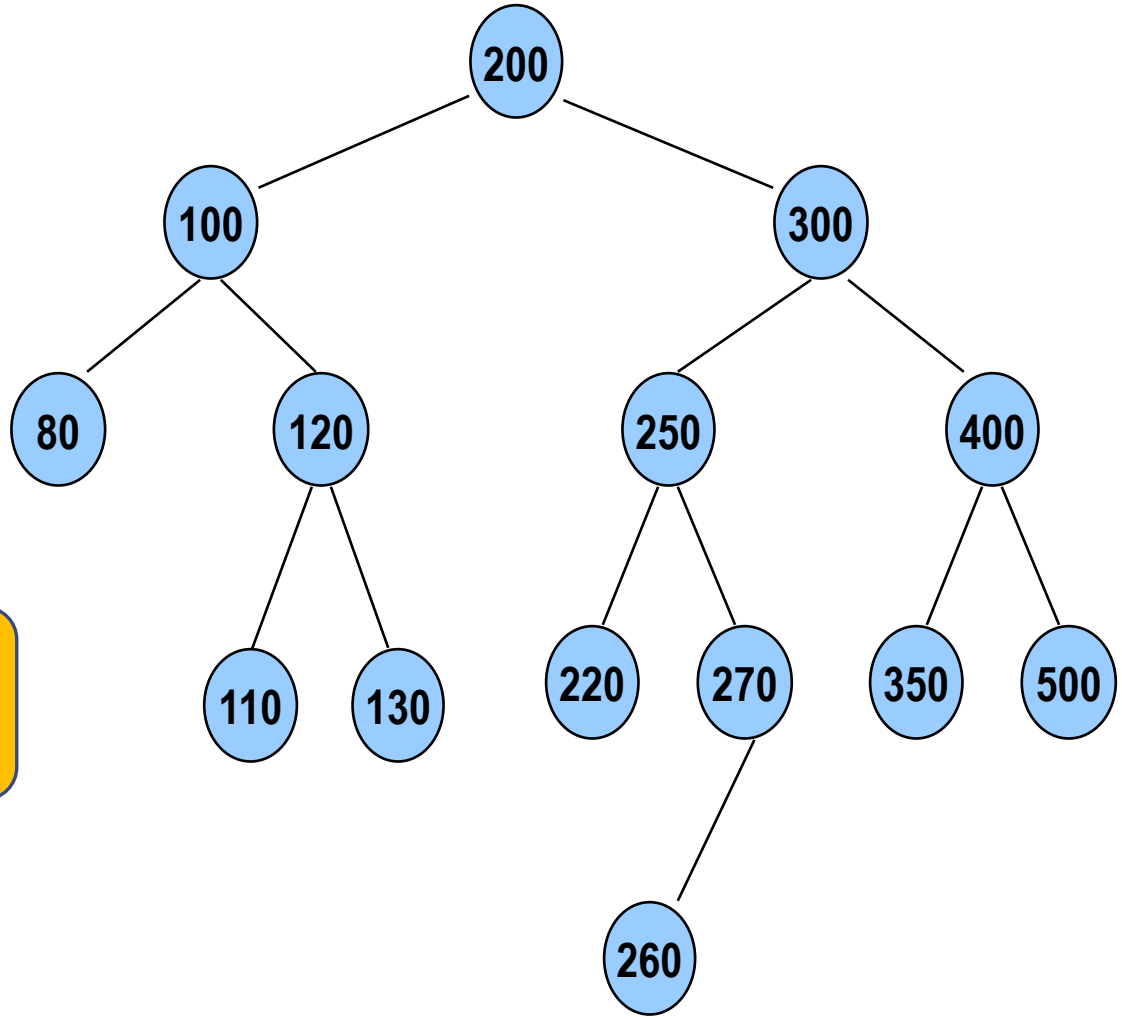


EXCLUSÃO DO NÓ DE CHAVE 150



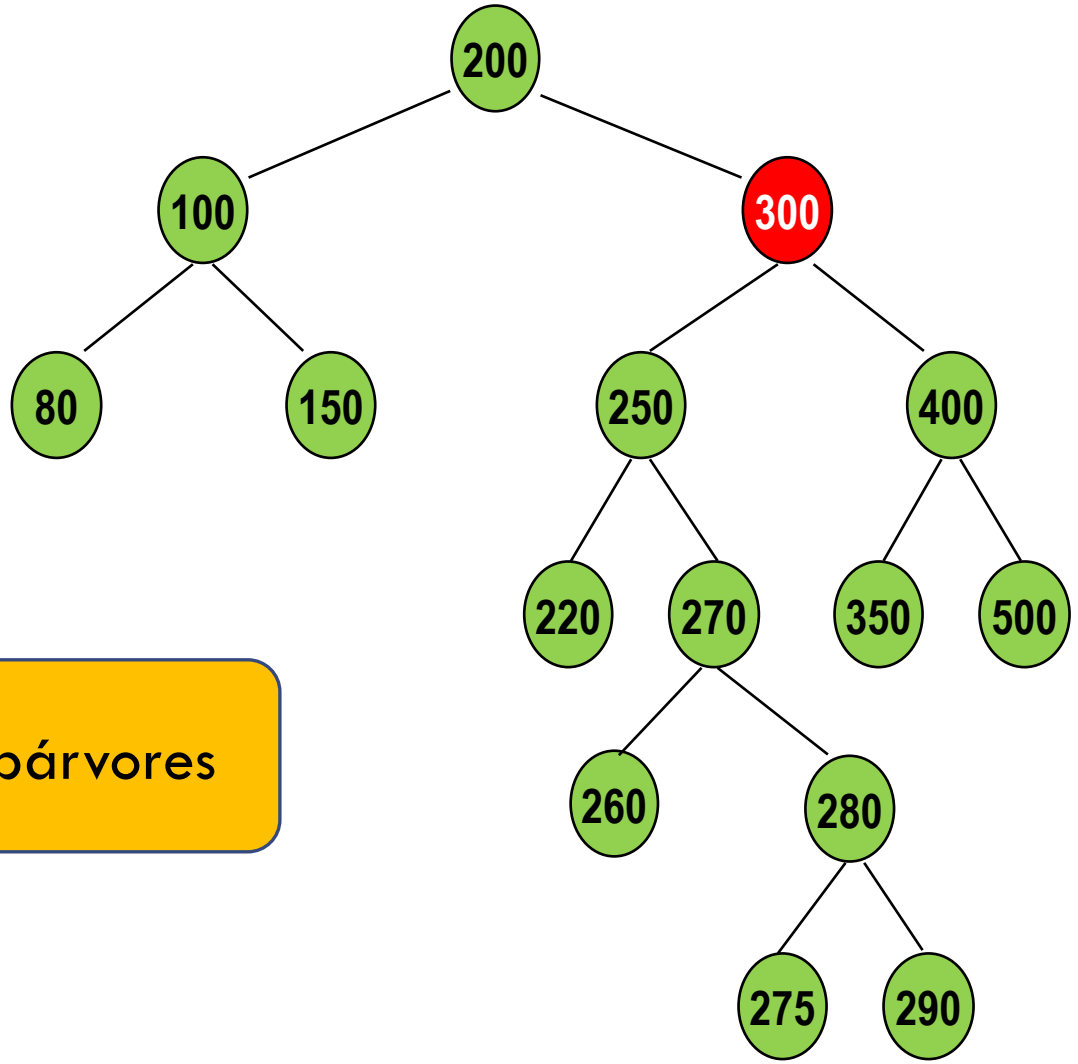
2°. Caso: nó com apenas 1 subárvore

EXCLUSÃO DO NÓ DE CHAVE 150



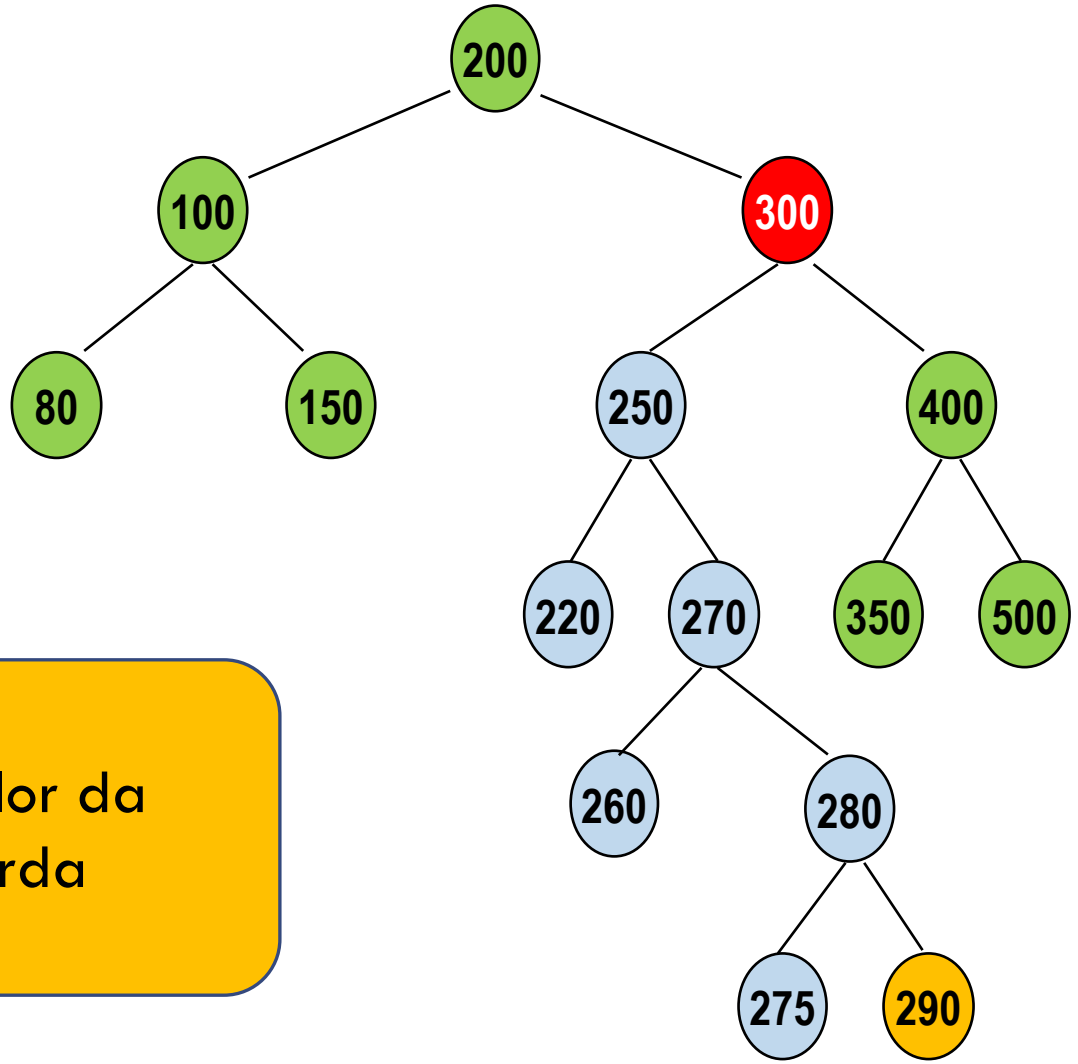
2°. Caso: nó com apenas 1 subárvore

EXCLUSÃO DO NÓ DE CHAVE 300



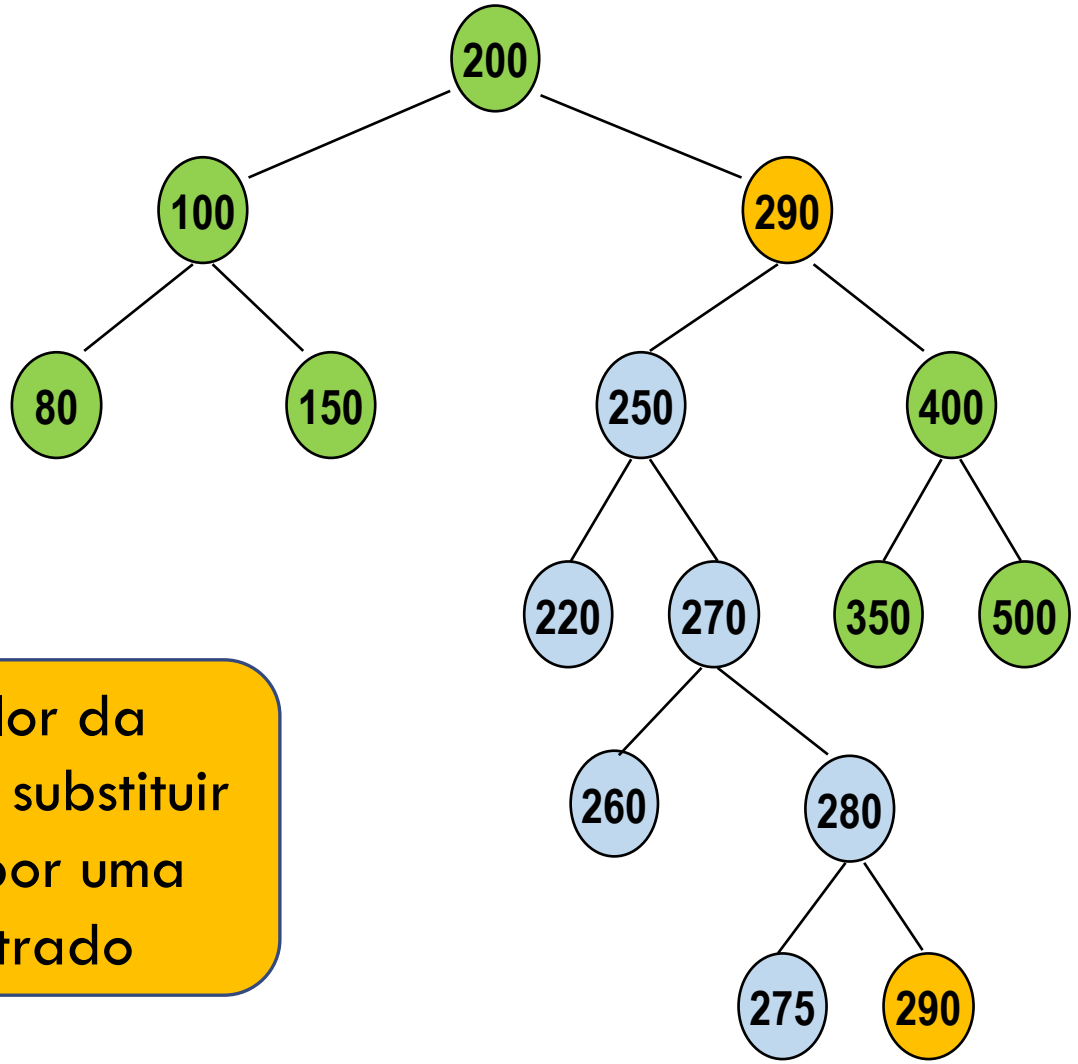
3°. Caso: nó com 2 subárvores

EXCLUSÃO DO NÓ DE CHAVE 300



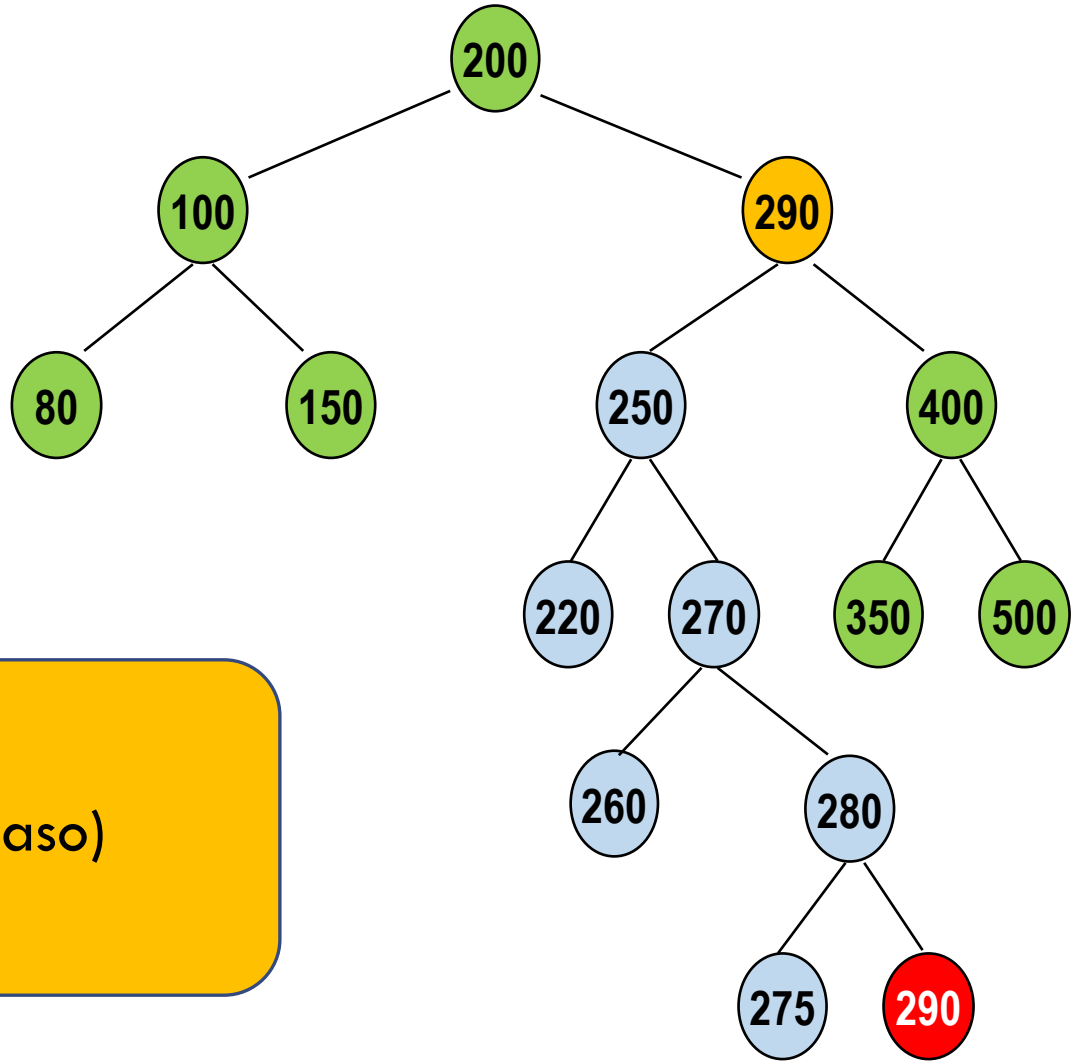
Encontrar maior valor da subárvore esquerda

EXCLUSÃO DO NÓ DE CHAVE 300



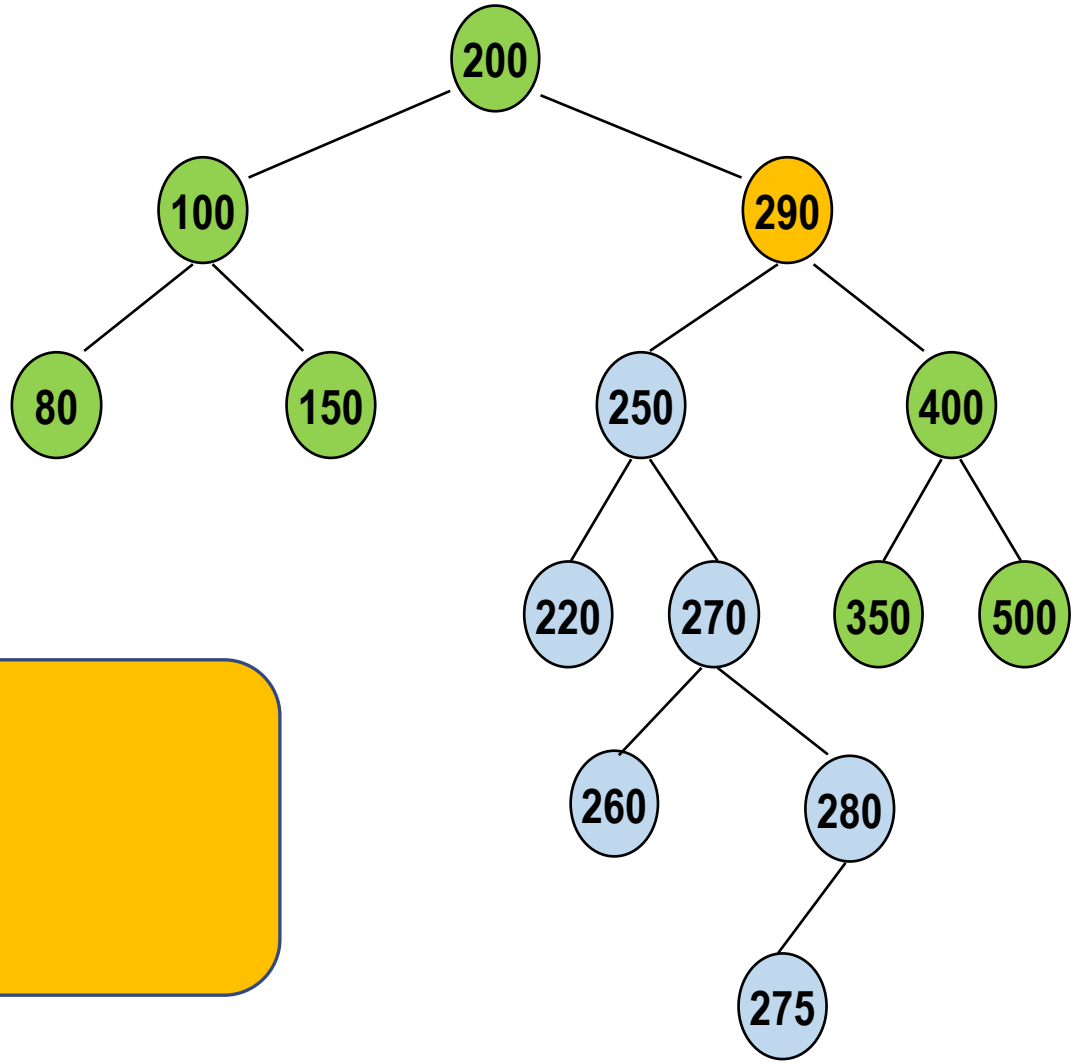
Encontrar maior valor da subárvore esquerda e substituir o nó a ser excluído por uma cópia do nó encontrado

EXCLUSÃO DO NÓ DE CHAVE 300



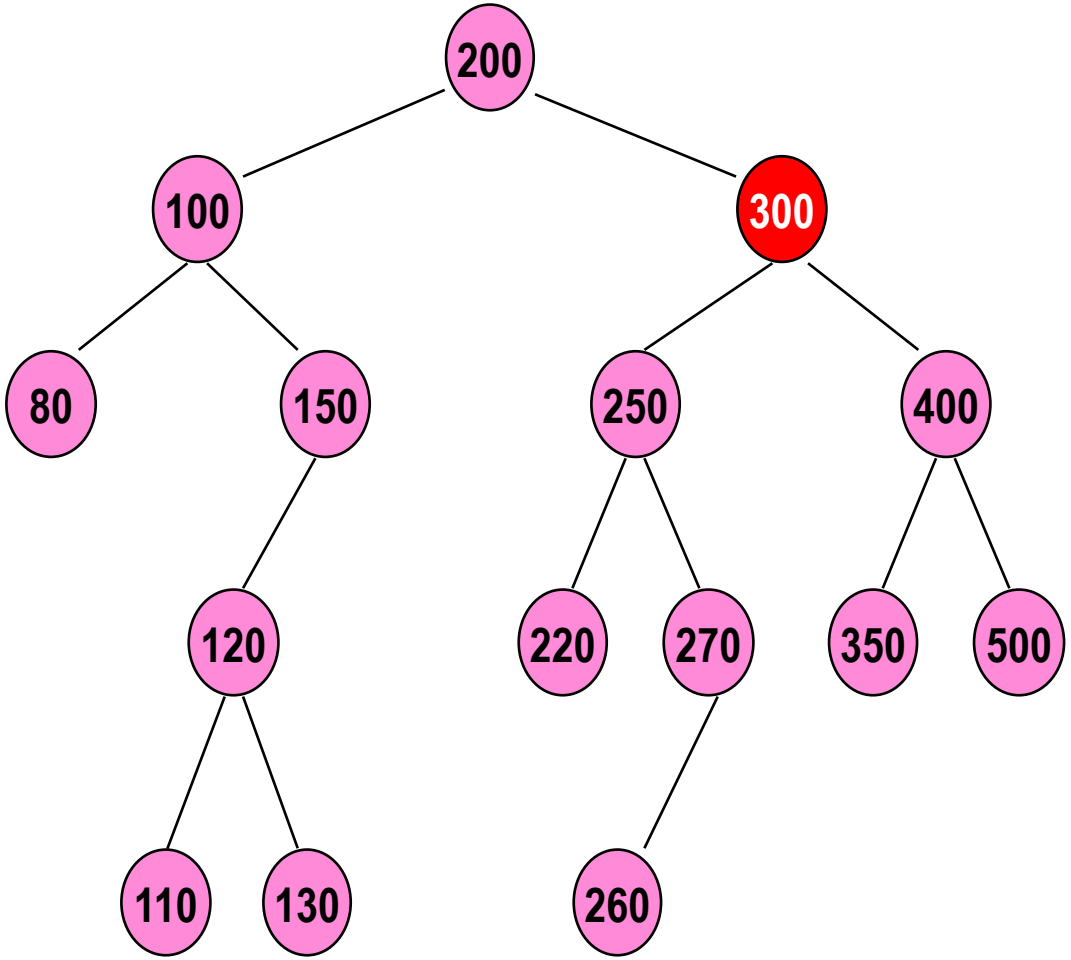
Excluir 290 (1°. Caso)

EXCLUSÃO DO NÓ DE CHAVE 300



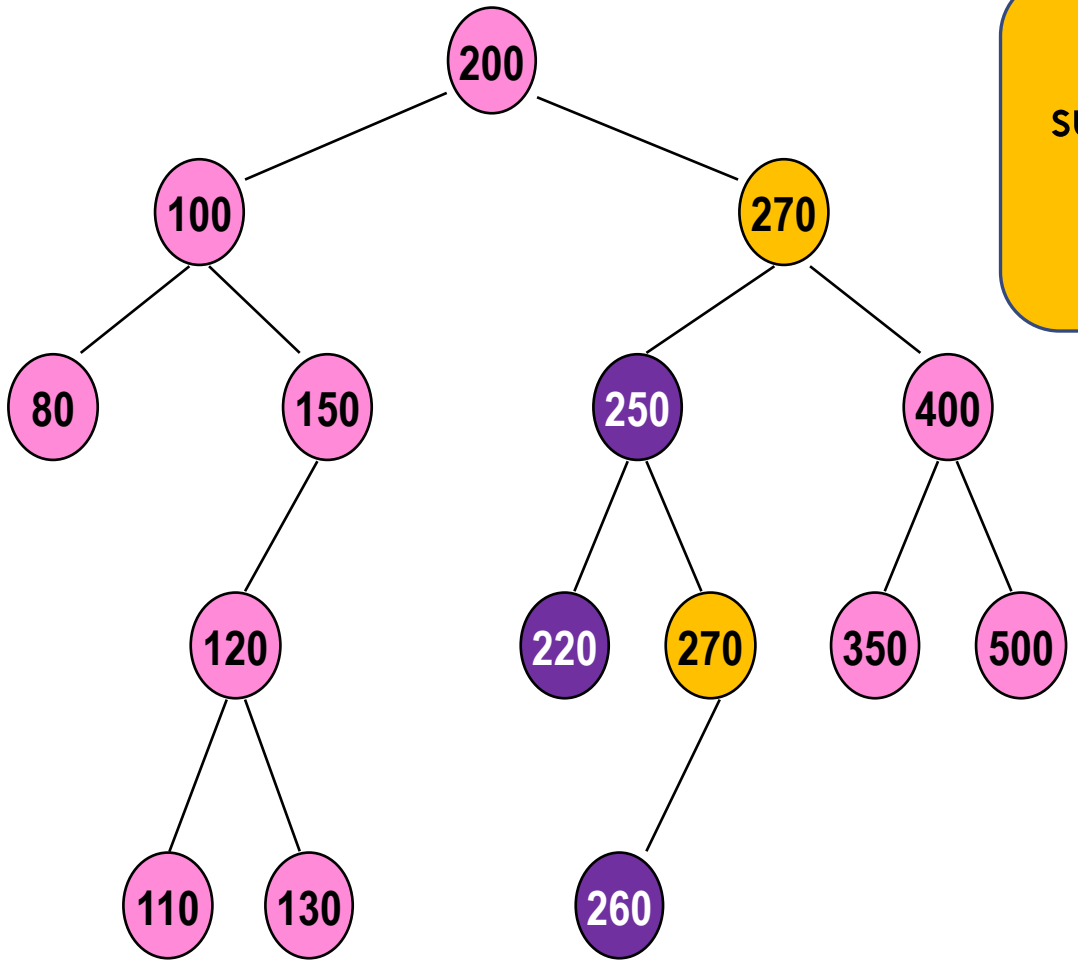
FIM

EXCLUSÃO DO NÓ DE CHAVE 300 (OUTRA ÁRVORE)



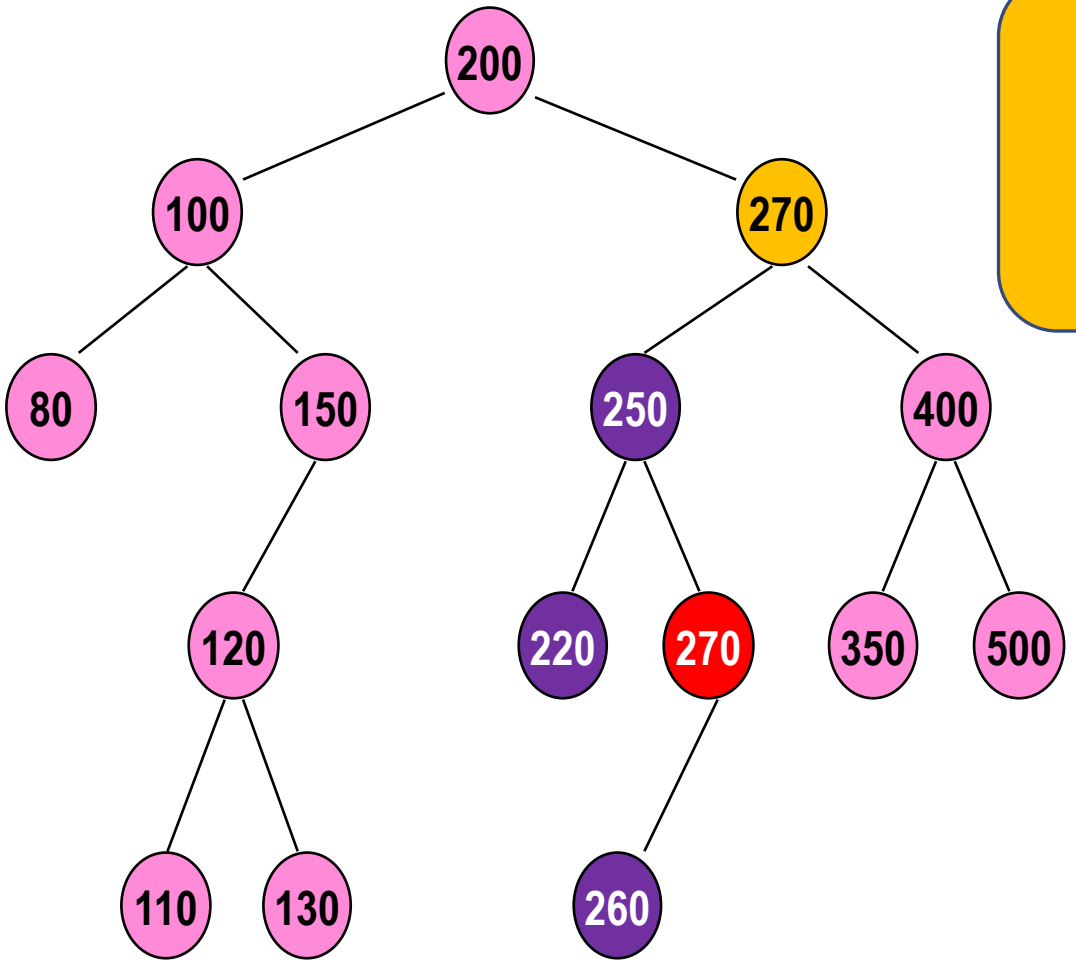
3°. Caso: nó com 2 subárvores

EXCLUSÃO DO NÓ DE CHAVE 300 (OUTRA ÁRVORE)



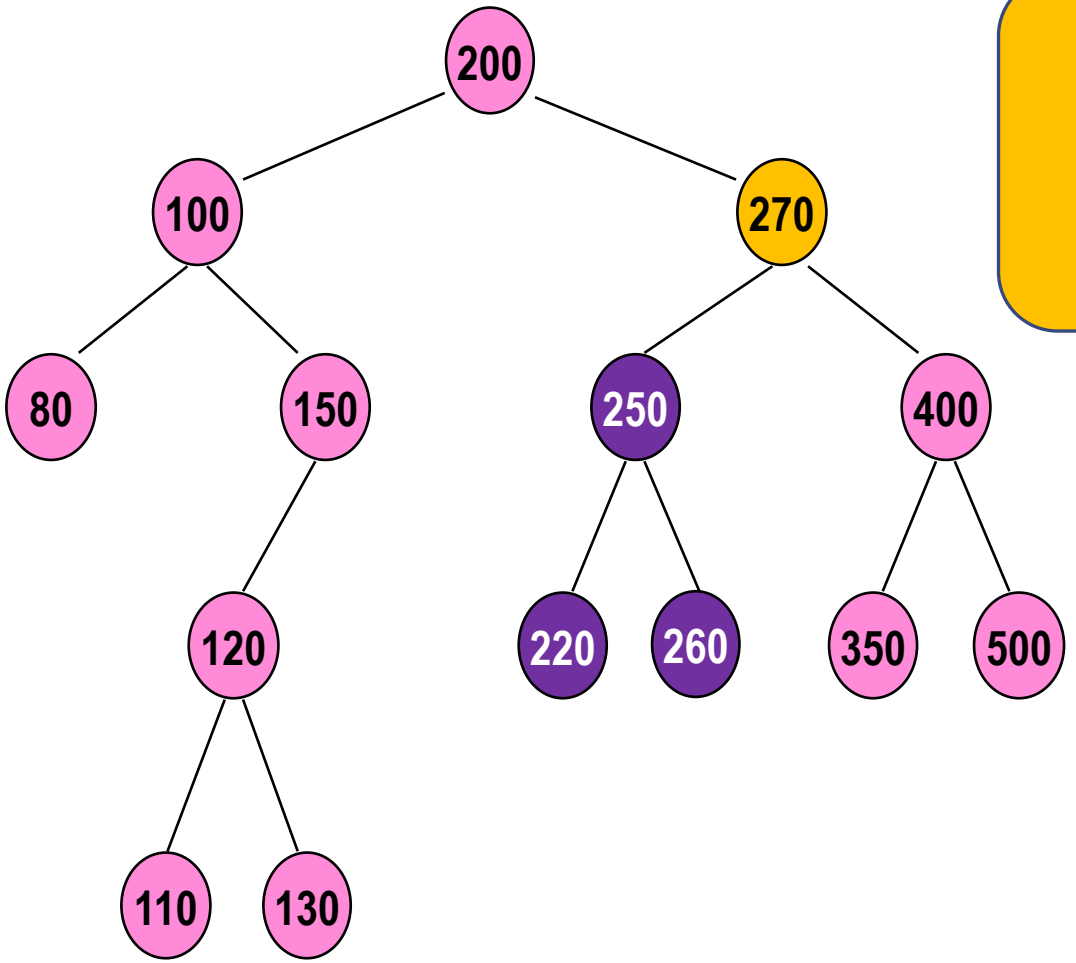
Encontrar maior valor da subárvore esquerda e substituir o nó a ser excluído por uma cópia do nó encontrado

EXCLUSÃO DO NÓ DE CHAVE 300 (OUTRA ÁRVORE)



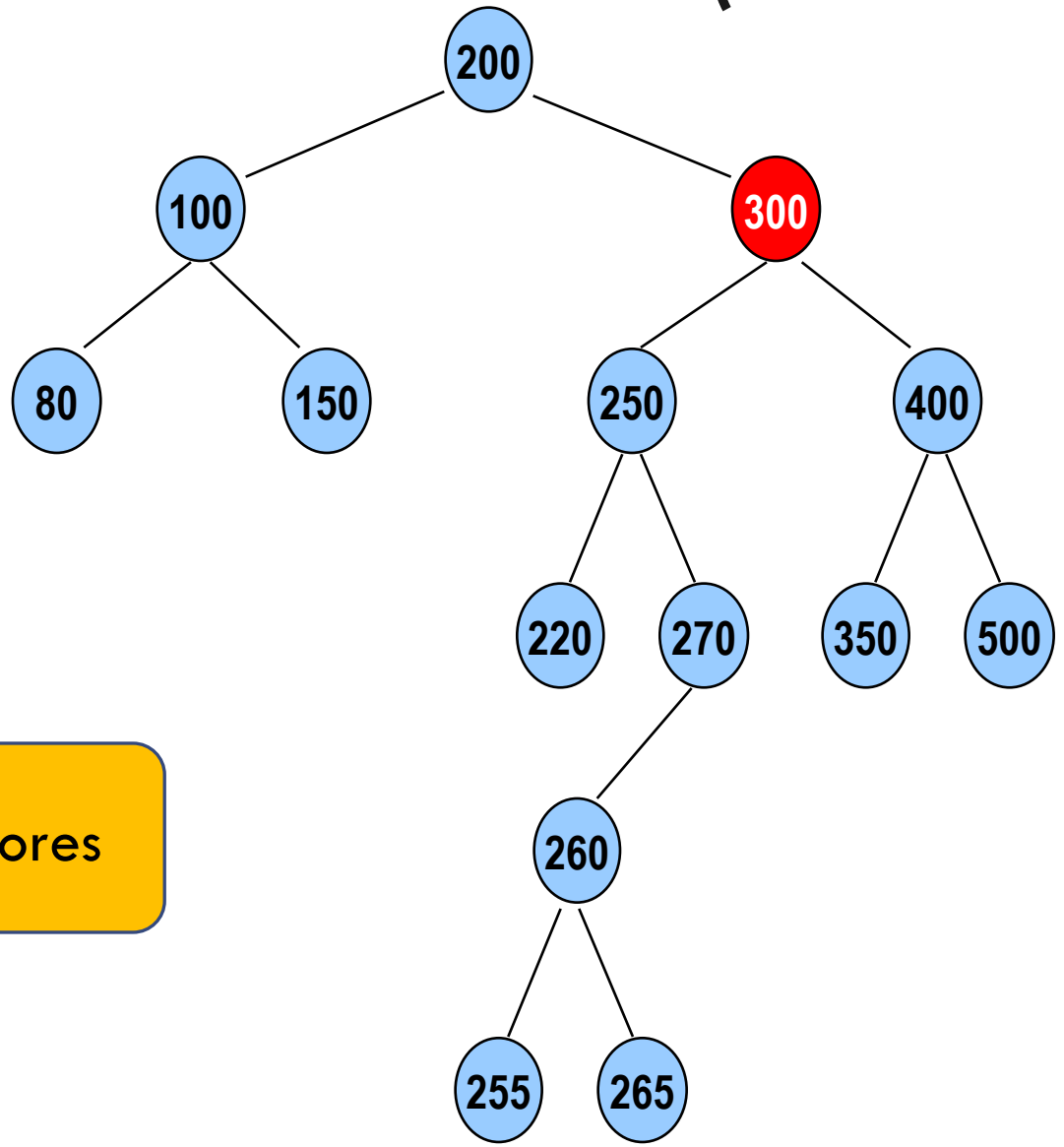
Excluir 270 (2°. Caso)

EXCLUSÃO DO NÓ DE CHAVE 300 (OUTRA ÁRVORE)



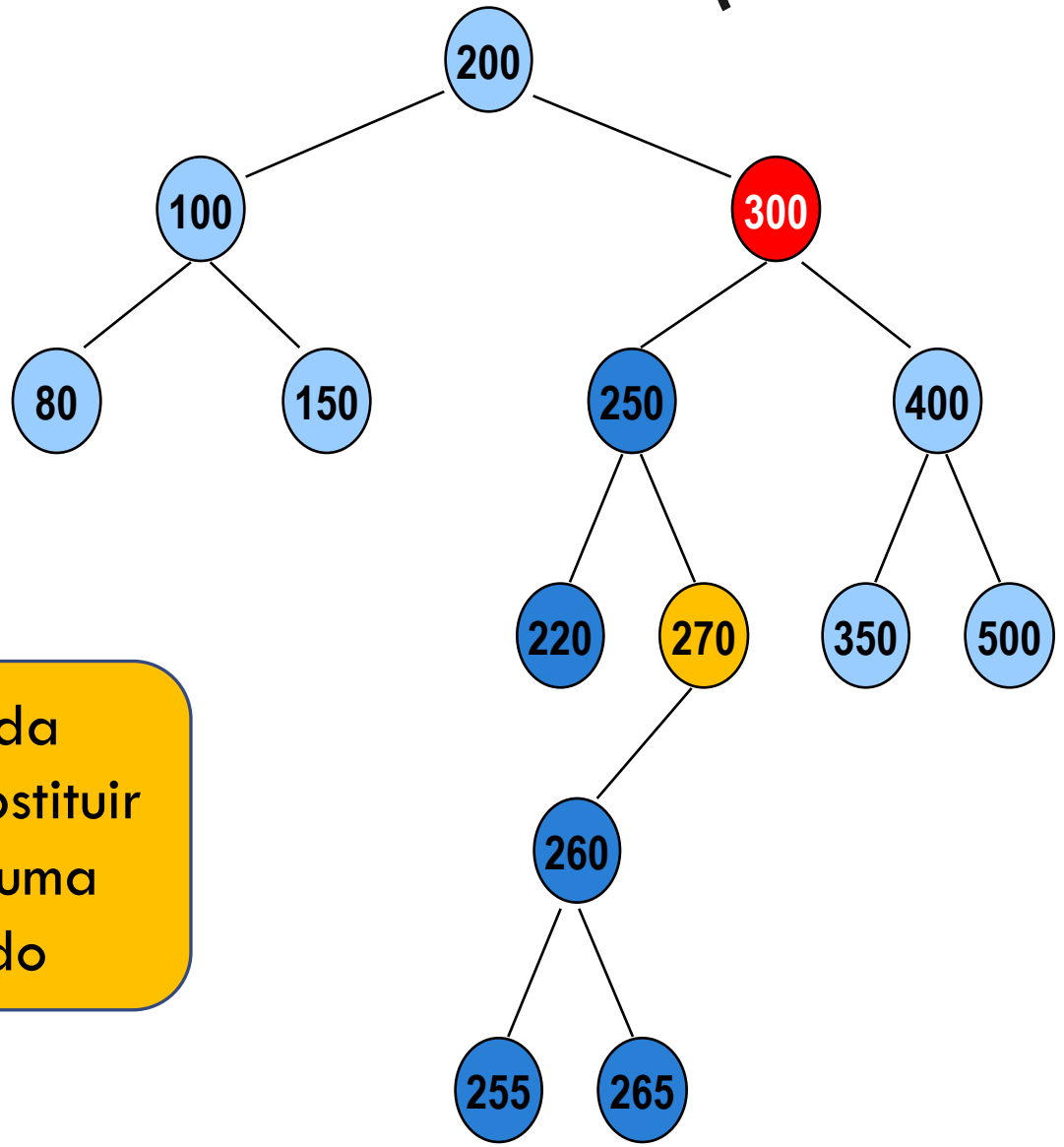
FIM

EXCLUSÃO DE NÓ DE CHAVE 300 (TERCEIRO EXEMPLO)



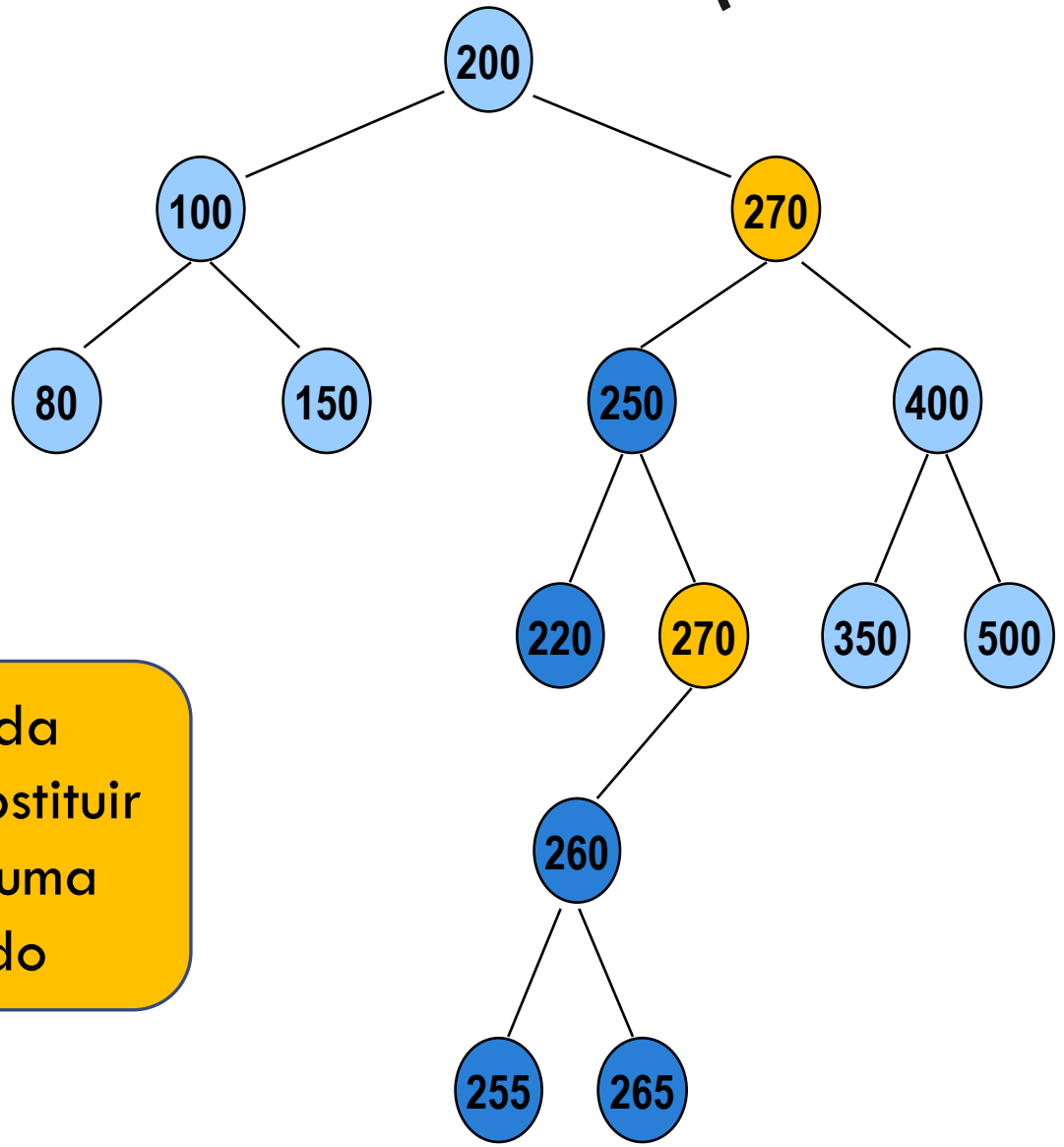
3°. Caso: nó com 2 subárvores

EXCLUSÃO DE NÓ DE CHAVE 300 (TERCEIRO EXEMPLO)



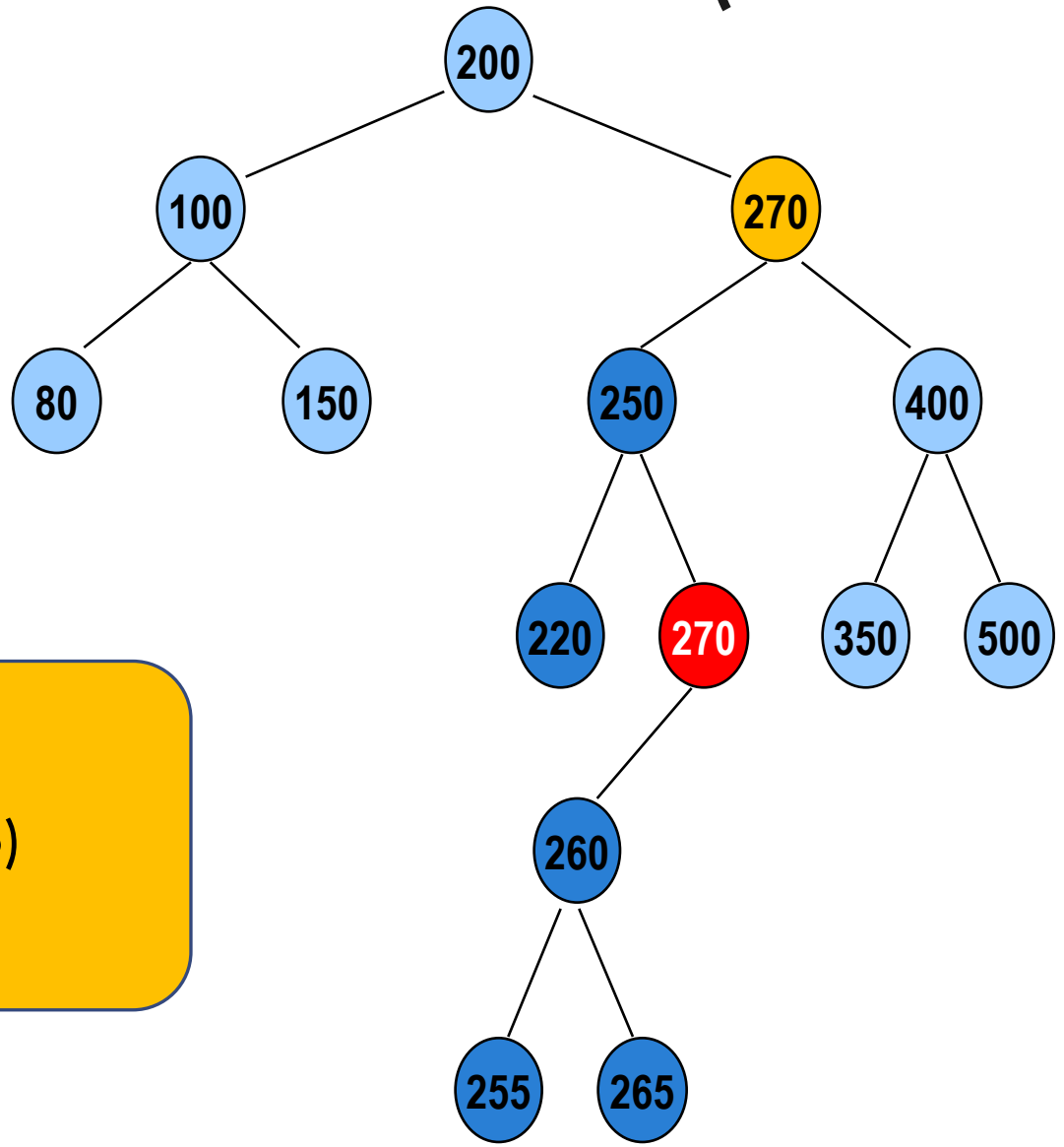
Encontrar maior valor da subárvore esquerda e substituir o nó a ser excluído por uma cópia do nó encontrado

EXCLUSÃO DE NÓ DE CHAVE 300 (TERCEIRO EXEMPLO)



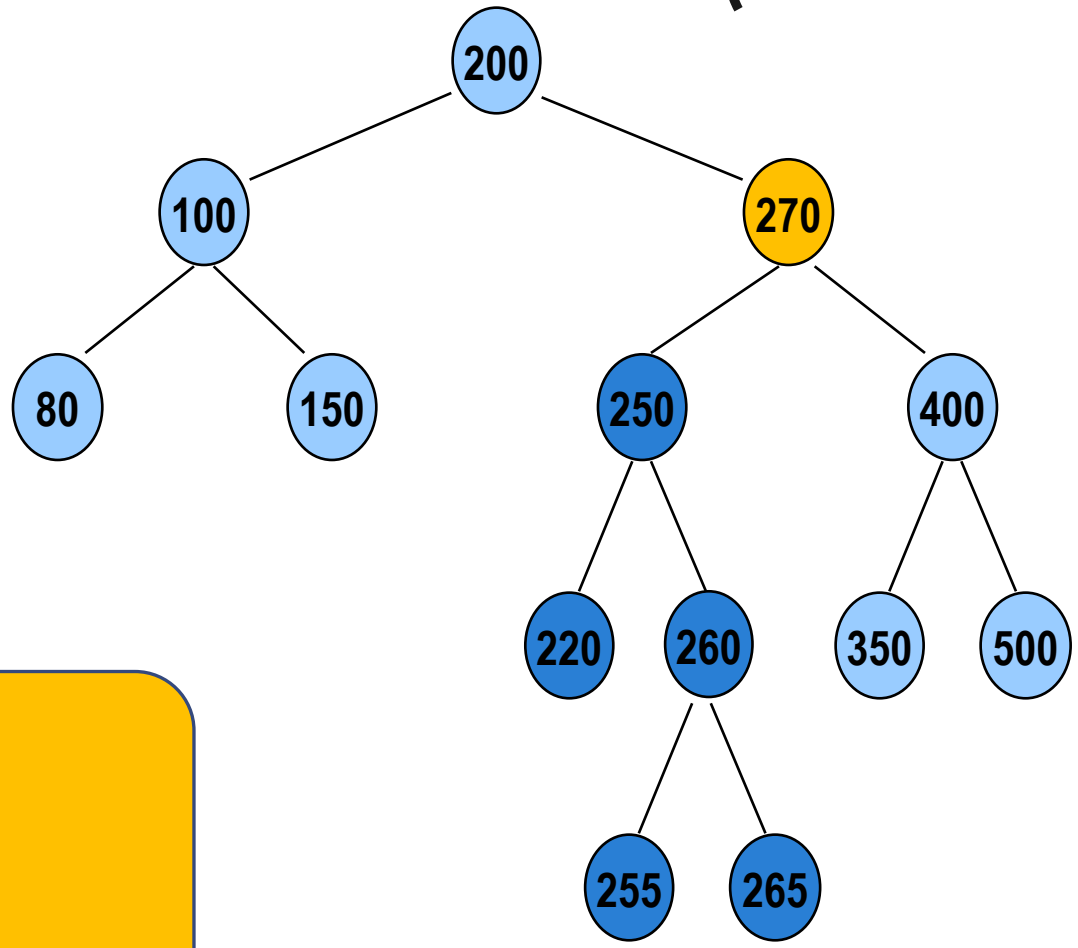
Encontrar maior valor da subárvore esquerda e substituir o nó a ser excluído por uma cópia do nó encontrado

EXCLUSÃO DE NÓ DE CHAVE 300 (TERCEIRO EXEMPLO)



Excluir 270 (2°. Caso)

EXCLUSÃO DE NÓ DE CHAVE 300 (TERCEIRO EXEMPLO)

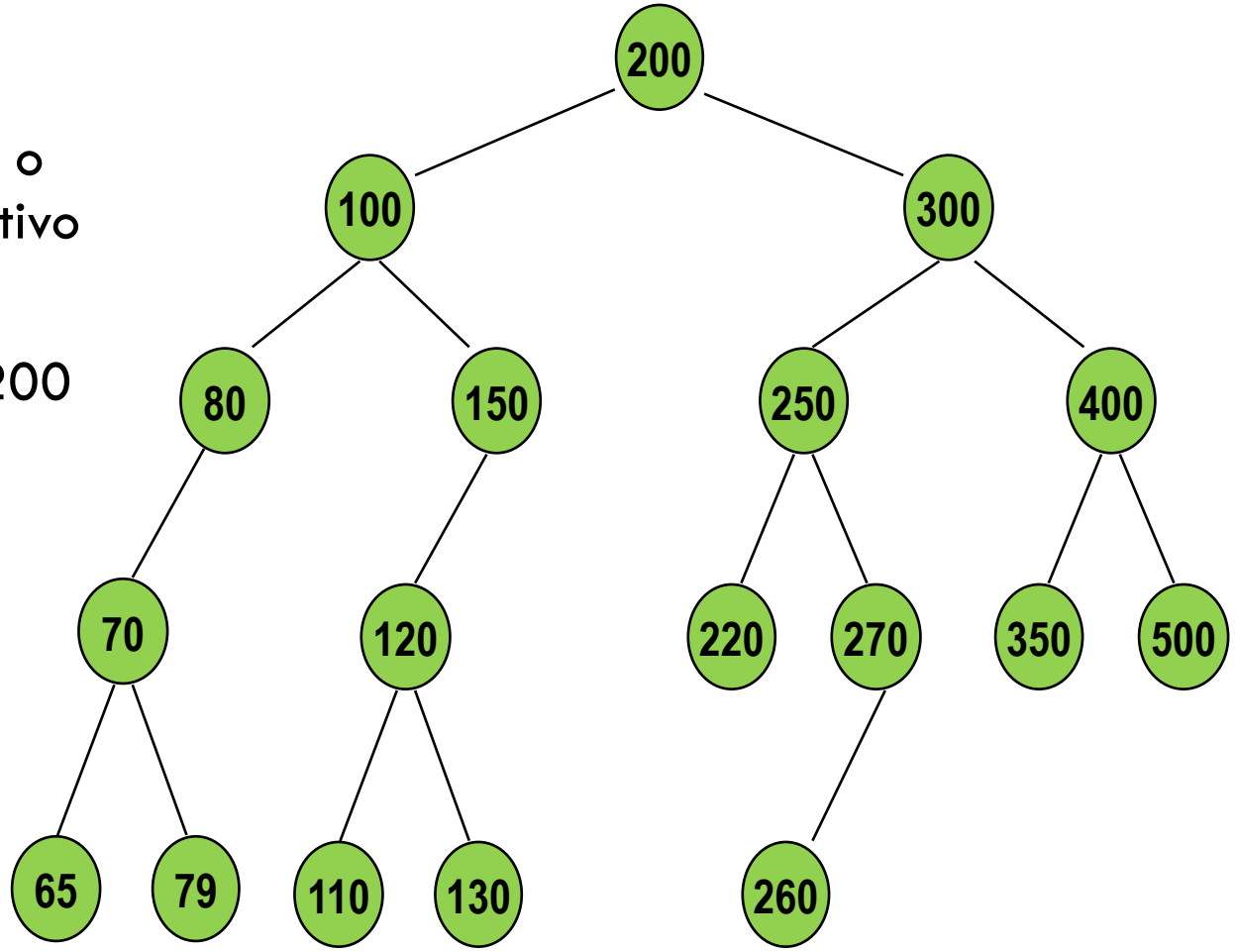


FIM

EXERCÍCIO

Dada a árvore a seguir, executar o procedimento de exclusão cumulativo dos seguintes nós:

100 – 150 – 80 – 270 – 400 – 200



CONSIDERAÇÕES FINAIS

Para grandes volumes de dados, árvores binárias de busca não são as alternativas mais eficientes.

Ao longo da disciplina veremos outras alternativas para buscas eficientes em grandes volumes de dados (Tabelas Hash, Árvores B, Árvores B+).

AGRADECIMENTOS

Material baseado nos slides de Renata Galante, UFRGS