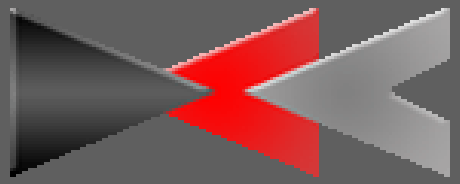
DCC107

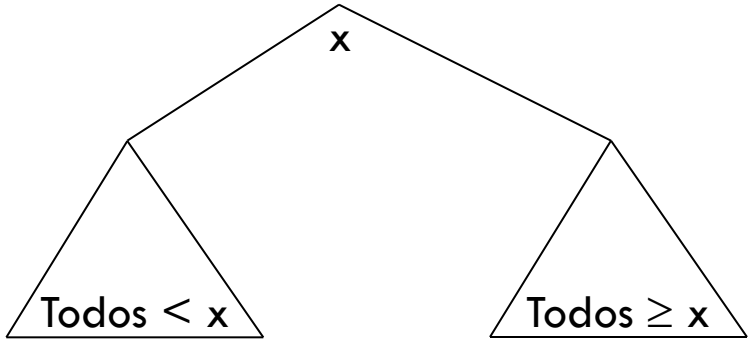


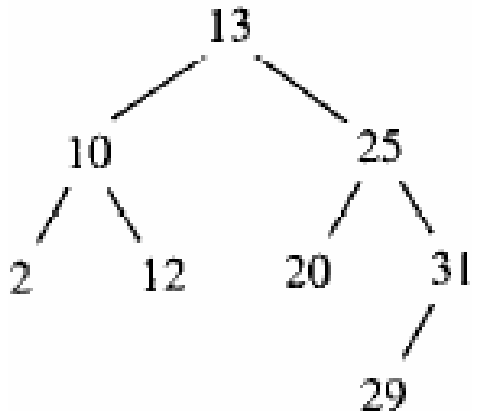
Laboratório de Programação II

Árvores Binárias de Busca

Árvores Binárias de Busca

- Propriedade fundamental:
  - Valor da chave da raiz é:
    - Maior que o valor da chave da subárvore da esquerda;
    - Menor que o valor da chave da subárvore da direita.
  - A subárvore da esquerda e subárvore da direita obedecem a propriedade fundamental.





Árvores Binárias de Busca

- Estrutura:

```
struct arv {
    int info;
    struct arv *esq;
    struct arv *dir;
};
typedef struct arv Arv;
```

- Inicialização:

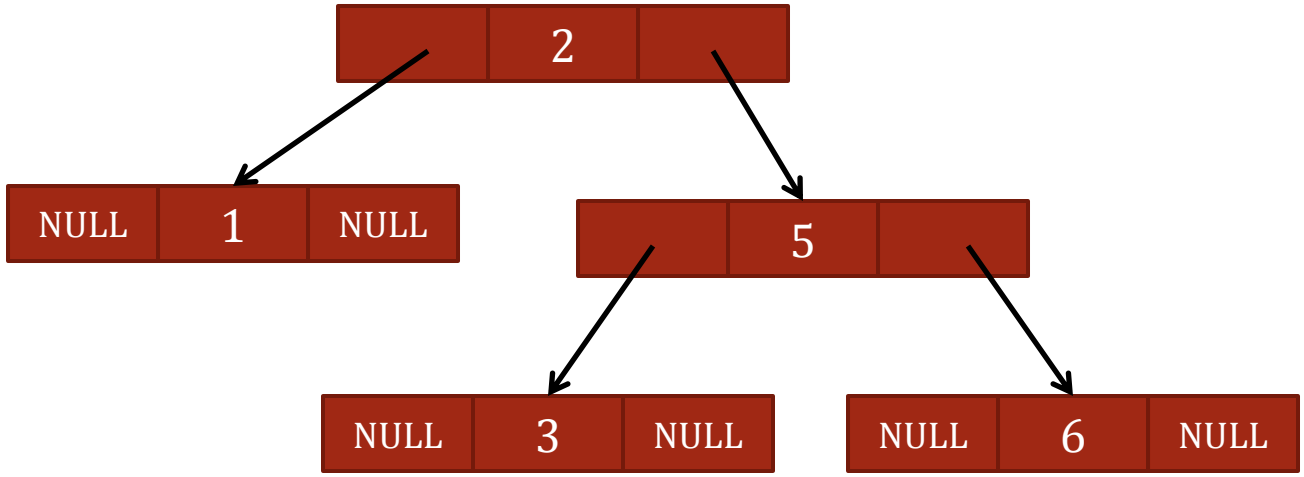
```
Arv* inicializa()
{
    return NULL;
}
```

Árvores Binárias de Busca

- Operações que são interessantes:
  - **busca**: busca um elemento na árvore;
  - **inserção**: insere um novo elemento na árvore;
  - **remoção**: retira um elemento da árvore.
- pois exploram a propriedade de ordenação das árvores binárias de busca.

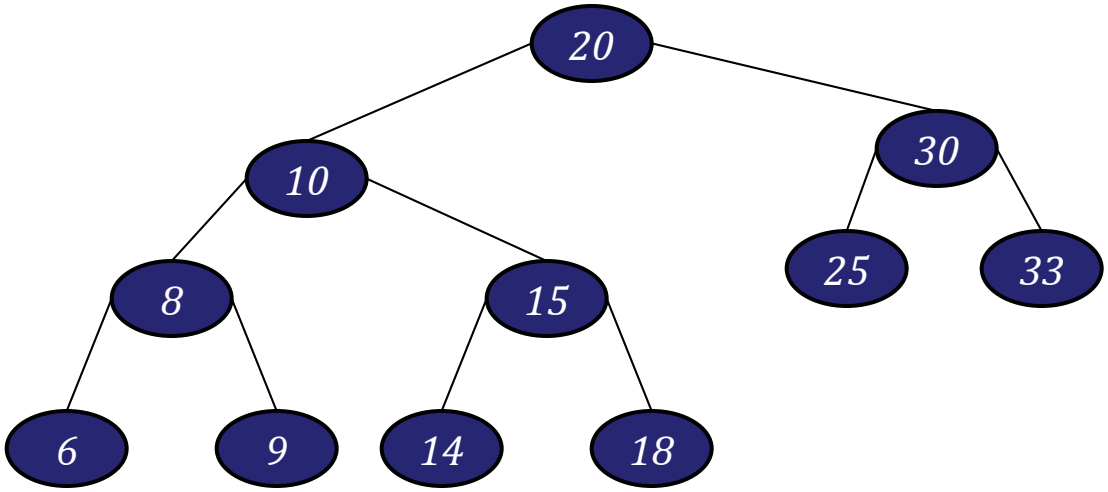
Árvores Binárias de Busca

- Busca “simplificada”:
  - Compara-se o valor com a raiz:
    - Se igual, achou;
    - Se maior, então buscar na subárvore da direita (SAD);
    - Se menor, então buscar na subárvore da esquerda (SAE).



Árvores Binárias de Busca

- Busca:
  - A partir da árvore binária a seguir, apresentar os caminhos percorridos para encontrar os nós:
    - 18
    - 9
    - 33
    - 35
    - 7



Árvores Binárias de Busca

- Busca:
  - Complexidade:
    - O número de comparações é proporcional a altura **h** da árvore: **O(h)**;
    - A altura da árvore é dada por $\log_2 N$;
    - Portanto, o número de operações é $O(\log_2 N)$, onde **N** é o número de nós da árvore.

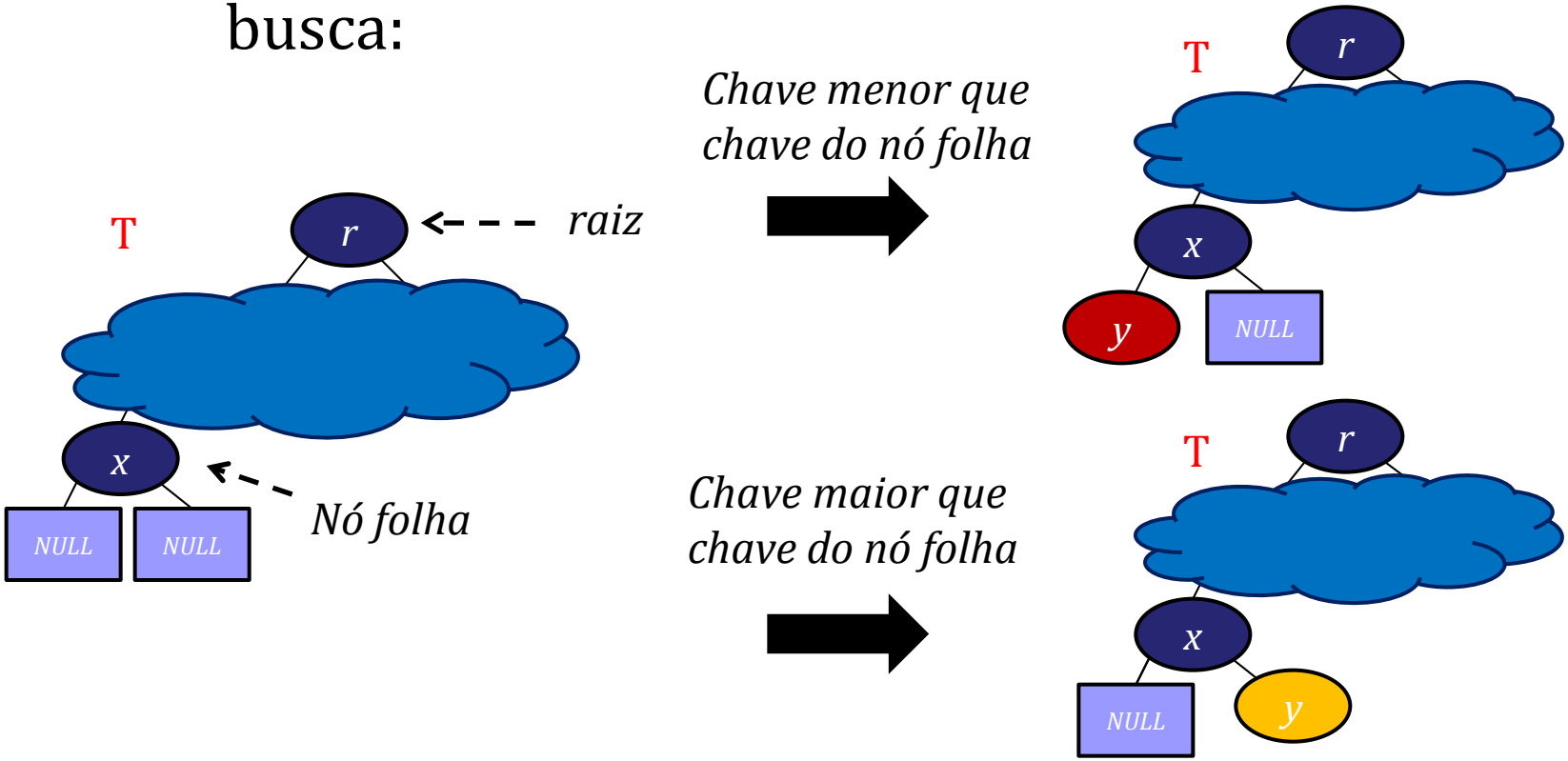
Árvores Binárias de Busca

- Busca:

```
Arv* busca(Arv *r, int v)
{
    if(r == NULL) //árvore está vazia
        return NULL;
    else if(v < r->info) //busca na subárvore à esquerda
        return busca(r->esq, v);
    else if(v > r->info) //busca na subárvore à direita
        return busca(r->dir, v);
    else
        return r;
}
```

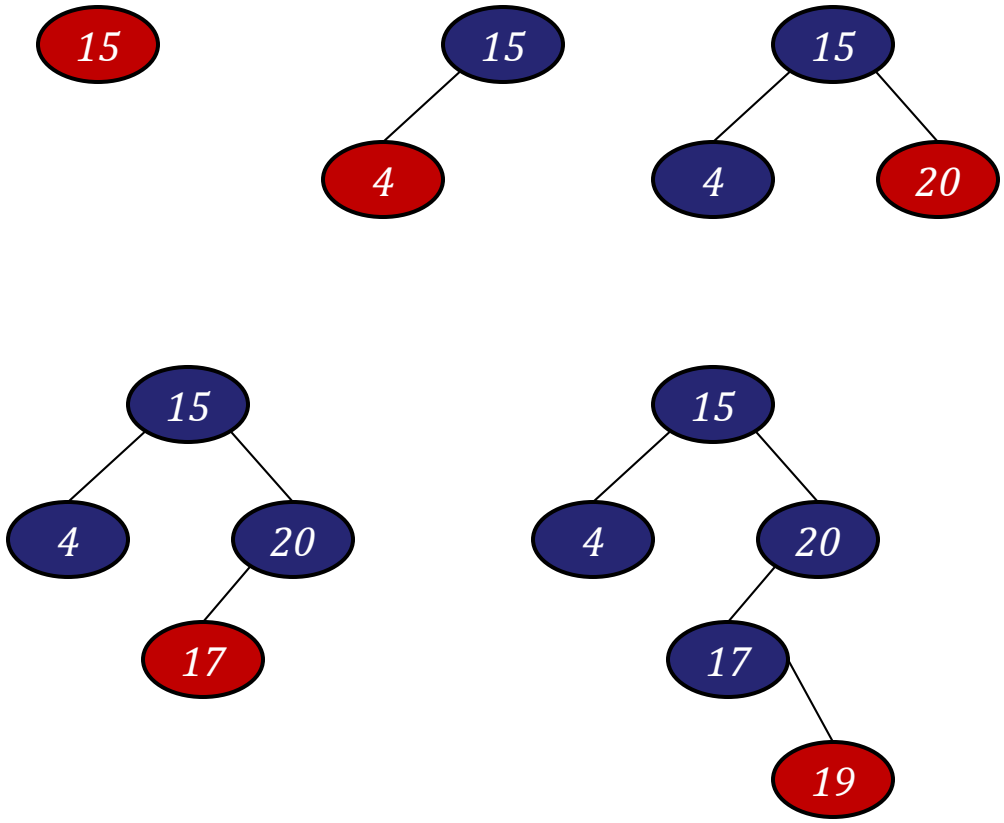
Árvores Binárias de Busca

- Inserção:
  - Inserir chave em novo nó no filho do nó folha mantendo a propriedade da árvore binária de busca:



Árvores Binárias de Busca

- Inserção (15, 4, 20, 17, 19):



Árvores Binárias de Busca

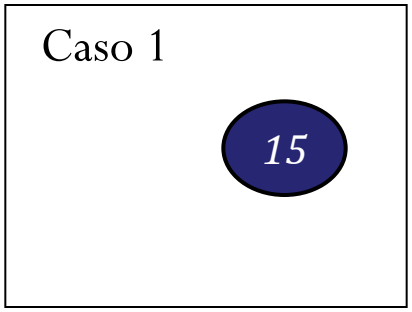
- Inserção:

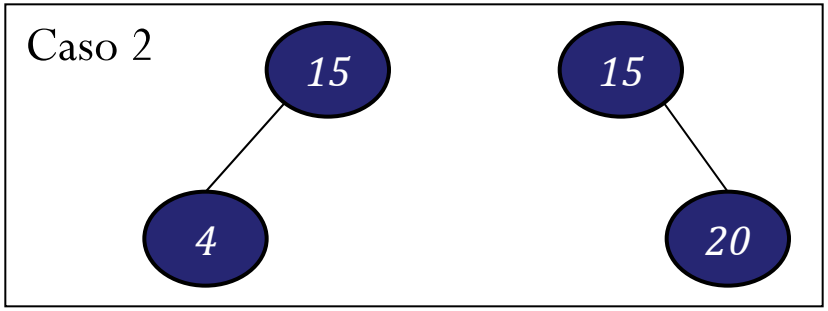
```
Arv* insere(Arv *a, int v)
{
    if (a == NULL)
    {
        a = (Arv*) malloc(sizeof(Arv));
        a->info = v;
        a->esq = a->dir = NULL;
    }
    else if(v < a->info)
        a->esq = insere(a->esq, v);
    else /* v > a->info */
        a->dir = insere(a->dir, v);
    return a;
}
```

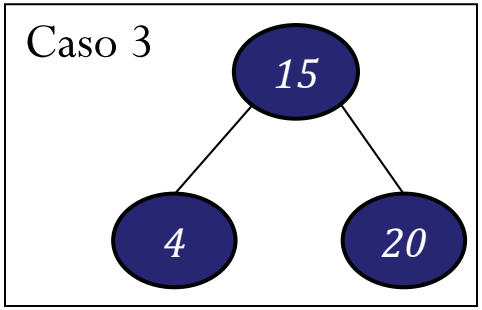
Árvores Binárias de Busca

- Remoção:
  - Nível de complexidade depende da posição do nó a ser removido, pois, após a remoção, a árvore deve preservar sua propriedade fundamental;
  - Mais difícil remover nó que tem duas subárvores do que remover nó folha;
  - Há três casos possíveis:
    - nó é uma folha (não tem filhos);
    - nó tem 1 filho;
    - nó tem 2 filhos.

Árvores Binárias de Busca

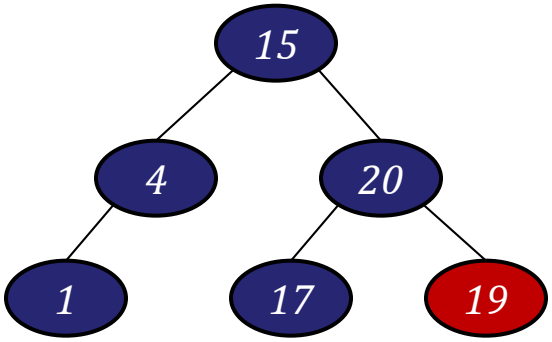


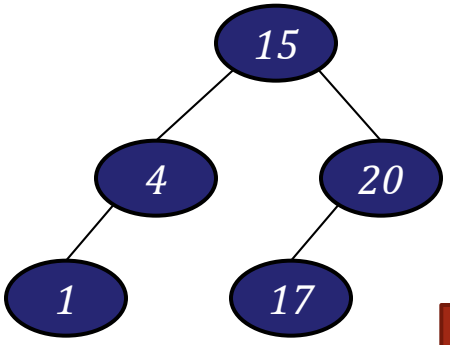




Árvores Binárias de Busca

- Remoção de nó folha:
  - Caso mais fácil de tratar;
  - O ponteiro apropriado de seu nó pai é ajustado para **`NULL`** e o nó é removido por desaloque (apagado da memória);
  - Exemplo: remover nó com conteúdo 19:

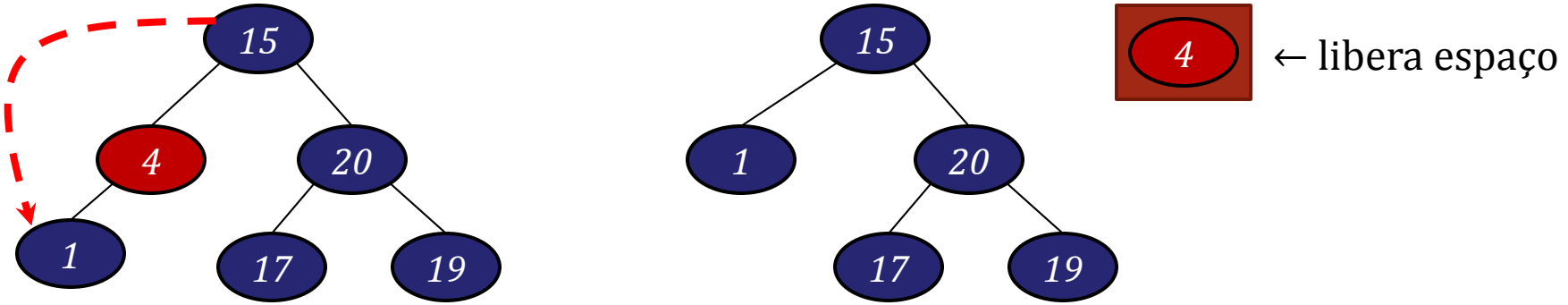




← libera espaço

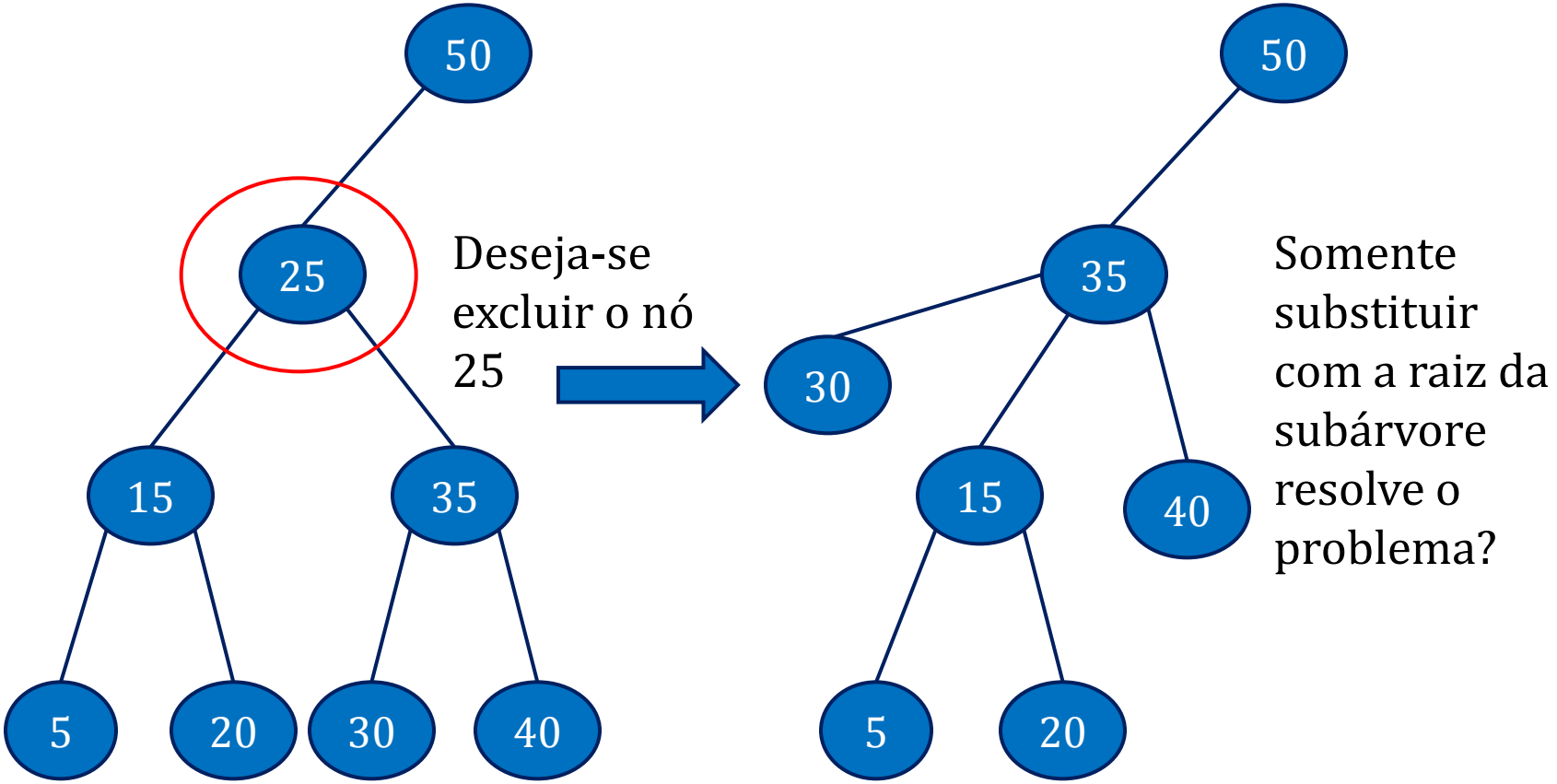
Árvores Binárias de Busca

- Remoção de nó com 1 filho:
  - Ponteiro do pai do nó a ser removido é reajustado para apontar para o filho do nó a ser removido, ou seja, o pai vai apontar para o neto (que passa a ser filho).
  - Desse modo:
    - Filhos do nó são elevados em 1 nível;
    - Bisnetos perdem um grau de descendência em suas designações de parentesco.
  - Exemplo: Remover nó 4:



Árvores Binárias de Busca

- Remoção de nó com 2 filhos:



Árvores Binárias de Busca

- Remoção de nó com 2 filhos:
  - Nó a ser removido tem dois filhos:
    - Remover fazendo junção (*merge*). Não será visto.
    - Remover fazendo cópia.

Árvores Binárias de Busca

- Remoção por cópia (Thomas Hibbard e Donald Knuth):
  - Substituir o nó pelo menor nó à direita e ajustar ponteiros;
  - Busca do substituto:
    - ir para direita;
    - procurar elemento mais à esquerda (menor filho à direita);
    - guardar antecessor, para descobrir pai do substituto
    - Se a direita estiver nula, o substituto será o próximo à esquerda.

Árvores Binárias de Busca

- Remoção por cópia (Thomas Hibbard e Donald Knuth):
  - Ajuste dos ponteiros:
    - ajustes no campo “esq” do pai do substituto e no campo “dir” do substituto (dispensáveis se nó à direita do removido é exatamente o substituto);
    - campo que referencia nó removido conterá substituto encontrado;
    - campo “esq” do substituto aponta para SAE do nó removido.

Árvores Binárias de Busca

- Remoção:

```
Arv* retira(Arv *r, int v)
{
    if (r == NULL)
        return NULL;
    else if(v < r->info)
        r->esq = retira(r->esq, v);
    else if(v > r->info)
        r->dir = retira(r->dir, v);
    else
    { /* achou o elemento */
        if(r->esq == NULL && r->dir == NULL)
        { /* elemento sem filhos */
            free(r);
            r = NULL;
        }
        //continua...
```

Árvores Binárias de Busca

- Remoção:

```
        else if(r->esq == NULL)
        { /* só tem filho à direita */
            Arv *t = r;
            r = r->dir;
            free(t);
        }
        else if(r->dir == NULL)
        { /* só tem filho à esquerda */
            Arv *t = r;
            r = r->esq;
            free(t);
        }
        //continua...
```

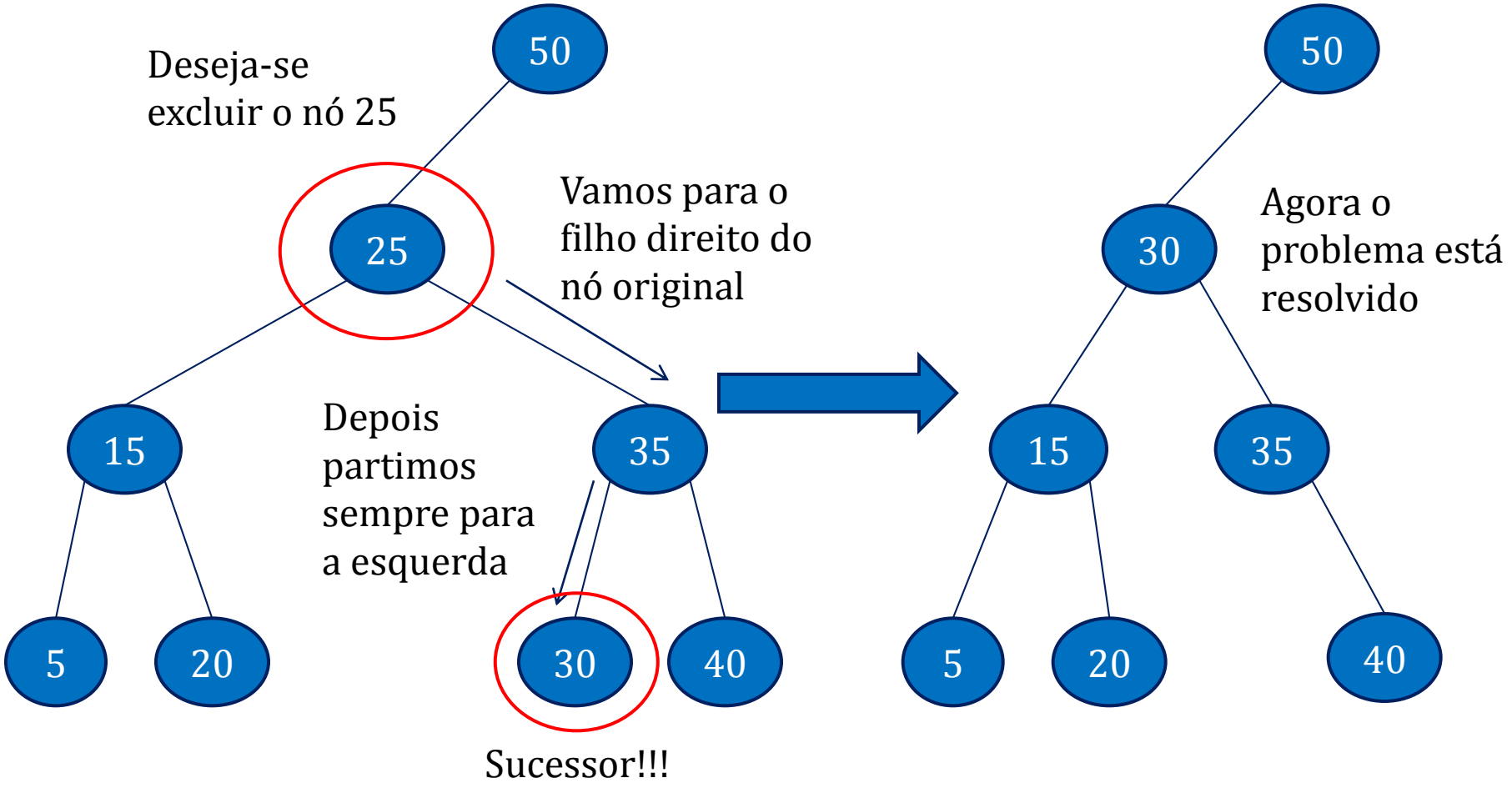
Árvores Binárias de Busca

- Remoção:

```
        else
        { /* tem os dois filhos */
            Arv *pai = r;
            Arv *f = r->dir;
            while(f->esq != NULL)
            {
                pai = f;
                f = f->esq;
            }
            r->info = f->info; /* troca as informações */
            f->info = v;
            r->dir = retira(r->dir, v);
        }
    }
    return r;
}
```

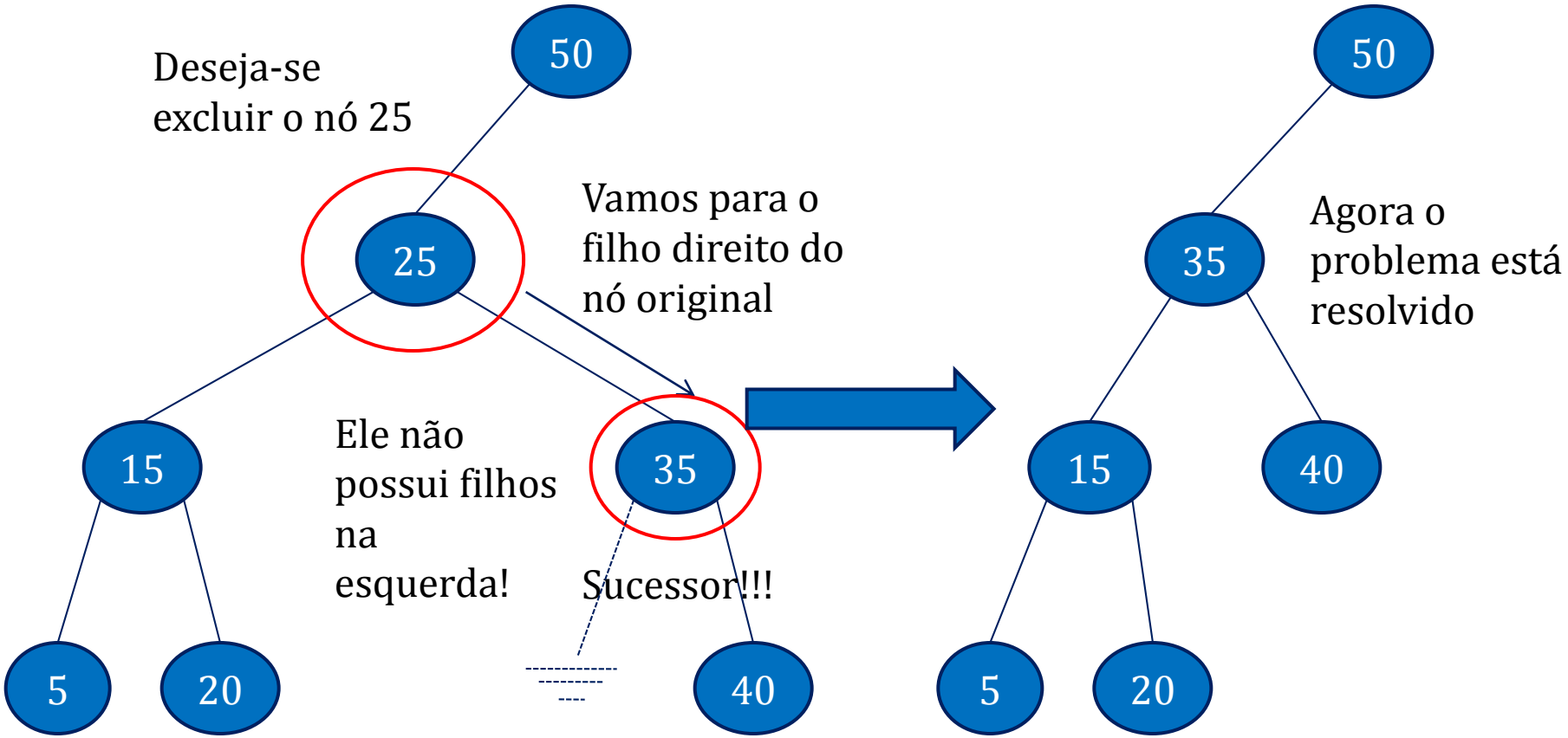
Árvores Binárias de Busca

- Remoção de nó com 2 filhos:



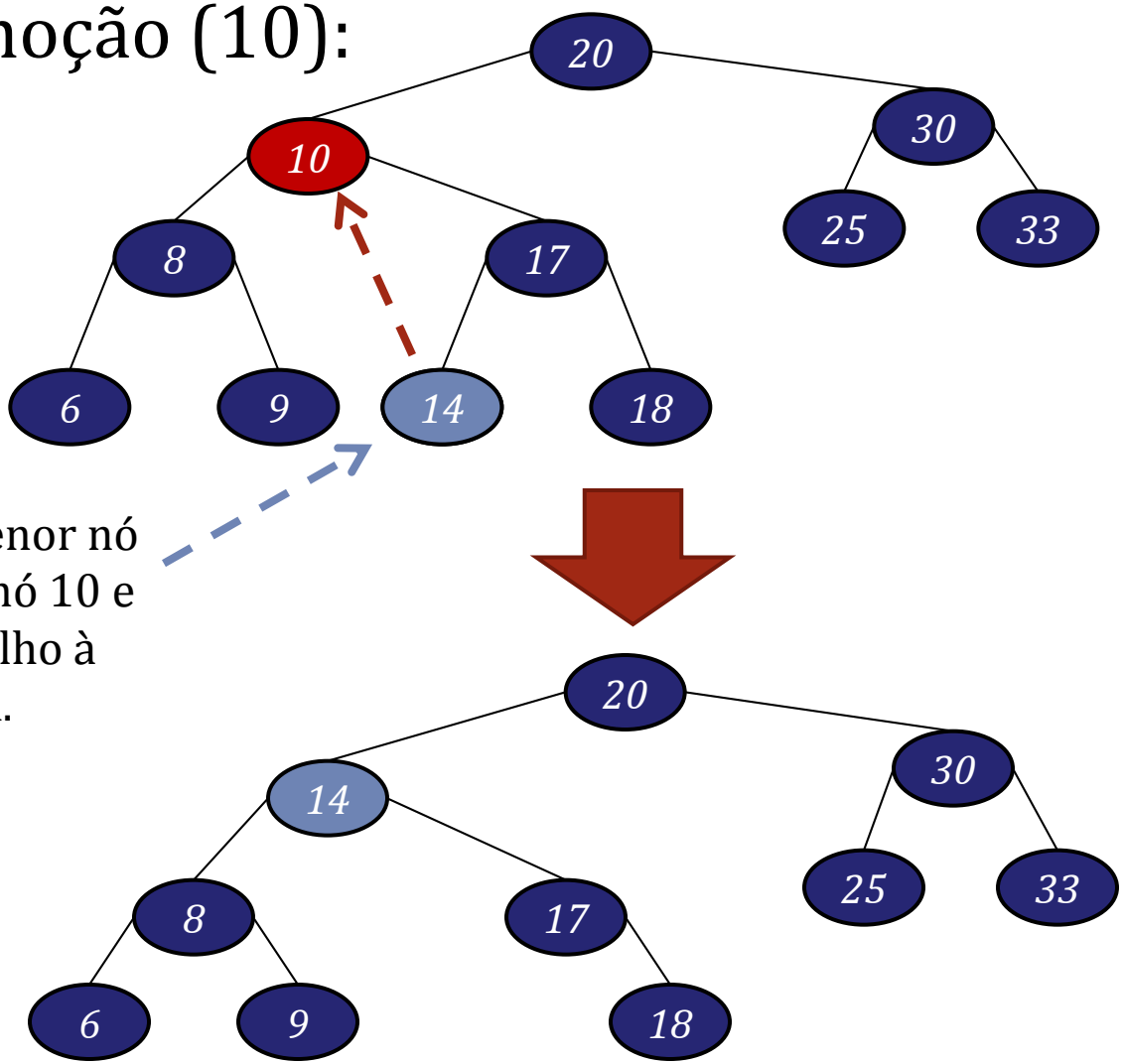
Árvores Binárias de Busca

- E quando chegarmos ao filho direito do nó original e ele não tiver filhos do lado esquerdo?



Árvores Binárias de Busca

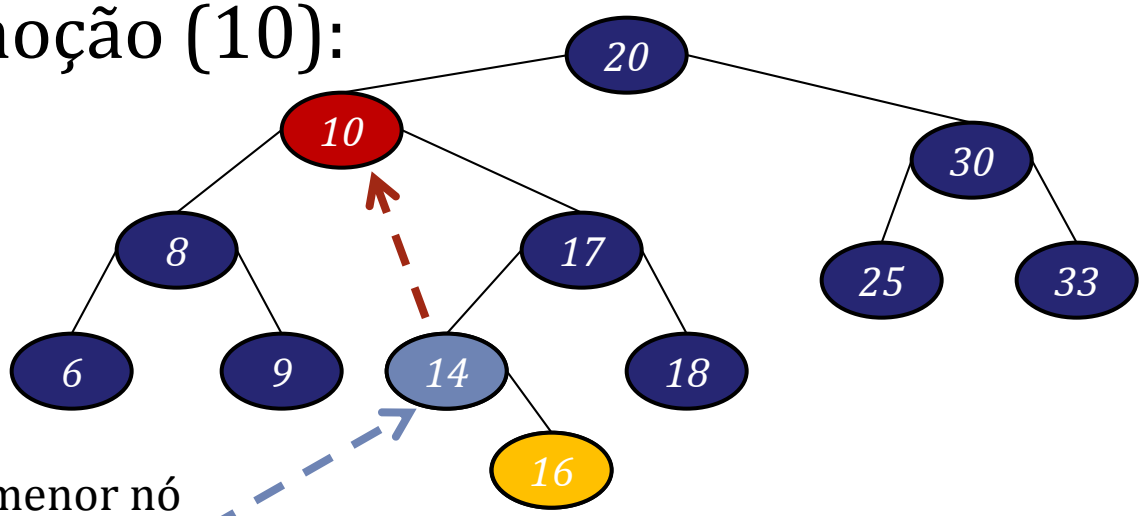
- Remoção (10):



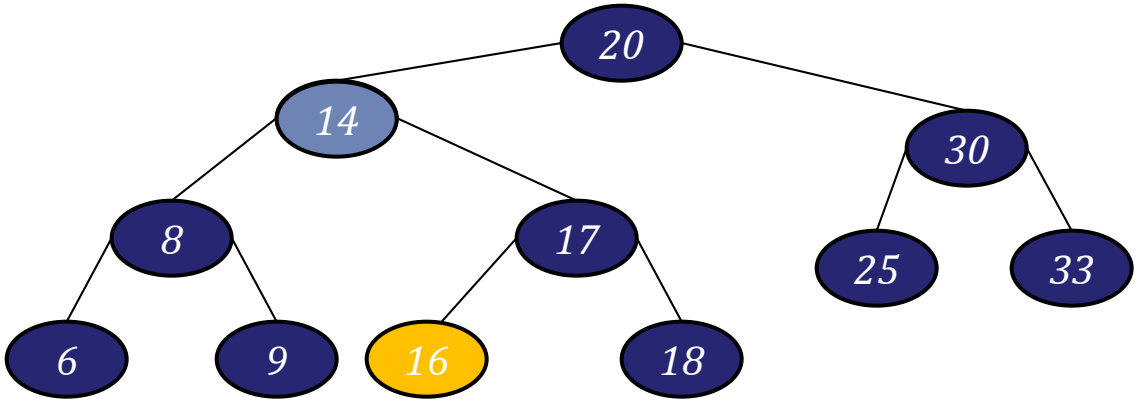
Nó 14 é o menor nó à direita do nó 10 e **não tem** filho à direita.

Árvores Binárias de Busca

- Remoção (10):

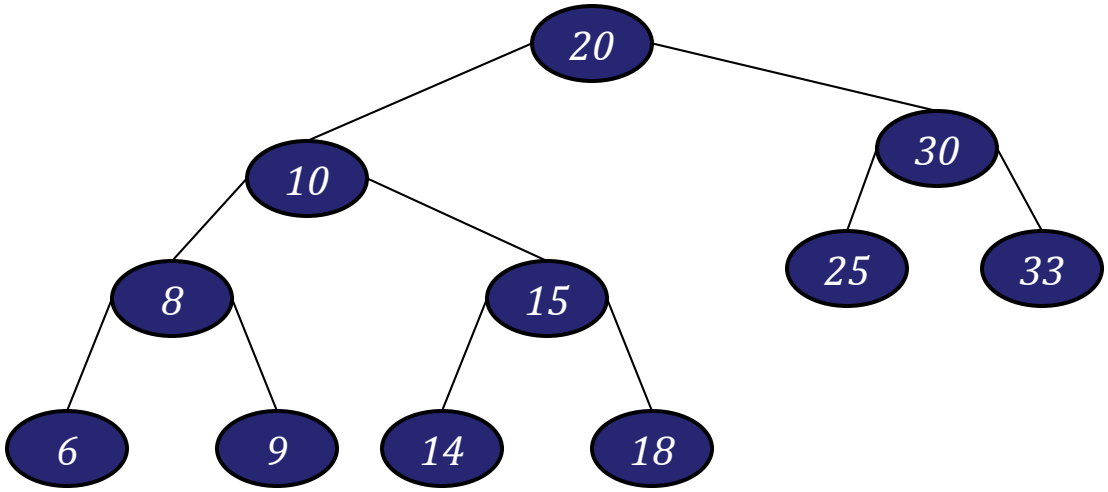


Nó 14 é o menor nó à direita do nó 10 e **tem** filho à direita.



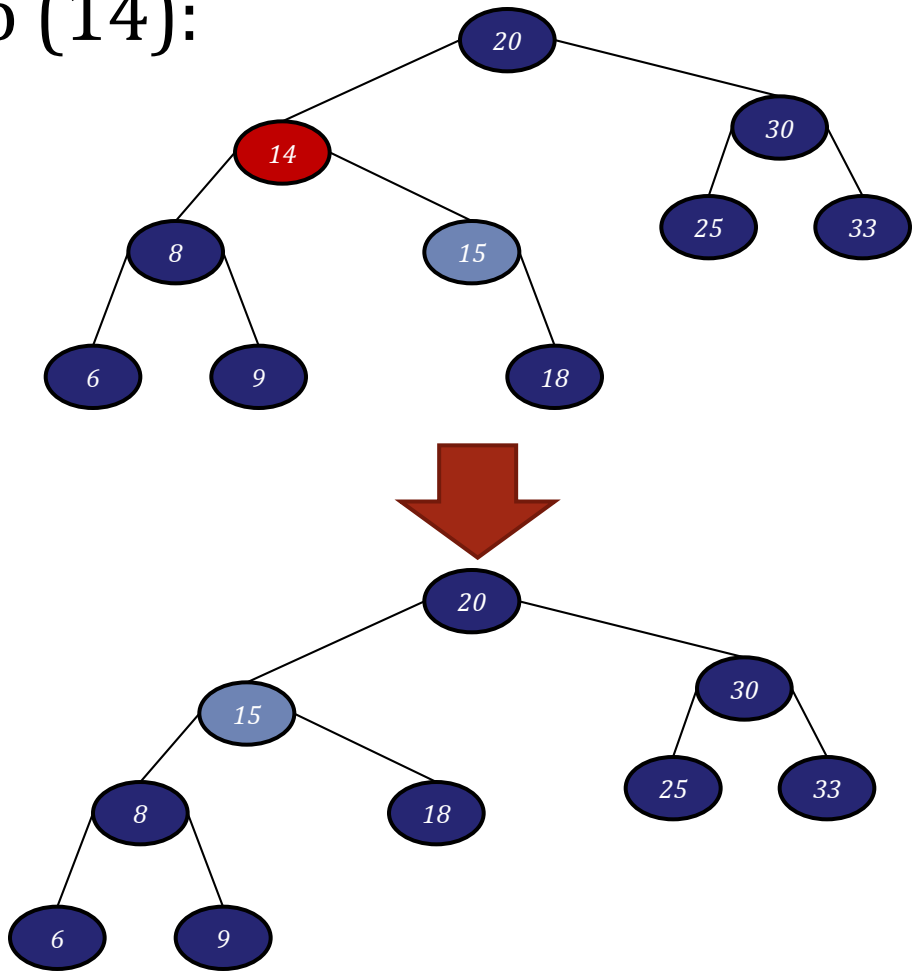
Árvores Binárias de Busca

- Exemplo Remoção:
  - Considere a seguinte árvore binária de busca:



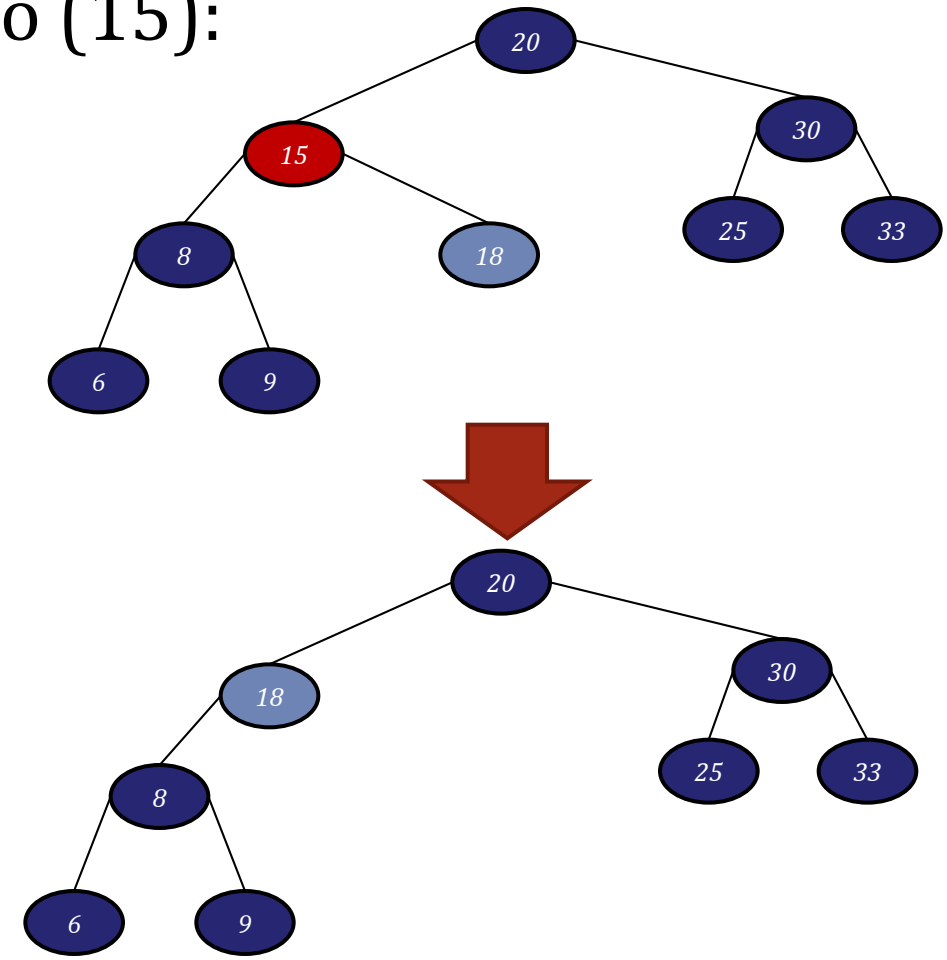
Árvores Binárias de Busca

- Remoção (14):



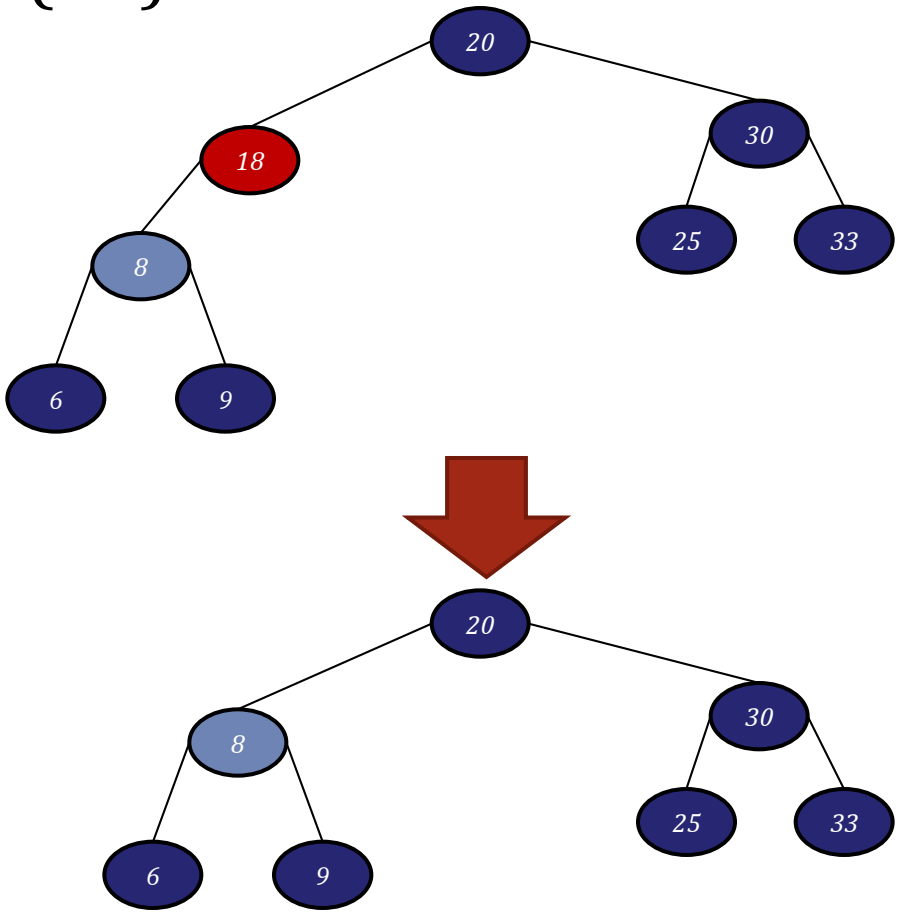
Árvores Binárias de Busca

- Remoção (15):



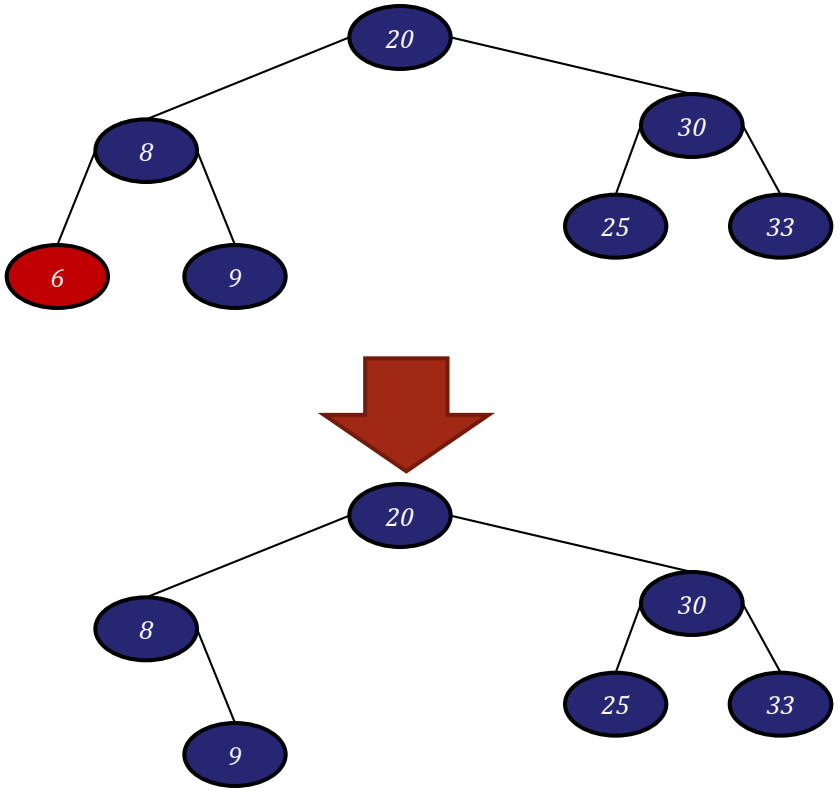
Árvores Binárias de Busca

- Remoção (18):



Árvores Binárias de Busca

- Remoção (6):



Árvores Binárias de Busca

- Libera:

```
int vazia(Arv *a)
{
    if(a == NULL)
        return 1;
    else
        return 0;
}

Arv* libera(Arv *a)
{
    if(!vazia(a))
    {
        libera(a->esq); /* libera sae */
        libera(a->dir); /* libera sad */
        free(a); /* libera raiz */
    }
    return NULL;
}
```

Árvores Binárias de Busca

- Em-Ordem:

```
void imprimeemordem(Arv *a)
{
     if(a != NULL)
     {
          imprimeemordem(a->esq);
          printf("%d\n",a->info);
          imprimeemordem(a->dir);
     }
}
```

Árvores Binárias de Busca

- Pré-Ordem:

```
void imprimepreordem(Arv *a)
{
    if(a != NULL)
    {
        printf("%d\n",a->info);
        imprimepreordem(a->esq);
        imprimepreordem(a->dir);
    }
}
```

Árvores Binárias de Busca

- Pós-Ordem:

```
void imprimeposordem(Arv *a)
{
    if(a != NULL)
    {
        imprimeposordem(a->esq);
        imprimeposordem(a->dir);
        printf("%d\n",a->info);
    }
}
```

Exercícios

- Fazer uma função para encontrar, e retornar, o **maior** elemento de uma árvore binária de busca;
- Fazer uma função para encontrar, e retornar, o **menor** elemento de uma árvore binária de busca;
- Fazer um procedimento para remover o **maior** elemento de uma árvore binária de busca;
- Fazer um procedimento para remover o **menor** elemento de uma árvore binária de busca.

Exercícios

- Alterar o(s) procedimento(s) de remoção de nó com dois filhos considerando, agora, o maior elemento da subárvore à esquerda como o elemento a ser “trocado” com o nó a ser removido.

Exercícios

- Uma árvore binária de busca é considerada balanceada se sua altura **h** é próxima de $\log_2(n)$, aonde **n** é o número de nós. Fazer uma função para verificar se uma dada árvore binária de busca está balanceada. Considere uma árvore balanceada se $h < \log_2(n) + 1$.

```
#include <math.h>

double log2(double n)
{
    return log(n) / log(2);
}
```

Exercícios

- Considere uma árvore binária de busca onde as informações armazenadas em cada nó são números inteiros. Implementar a função **Arv* podar(int c, Arv *a)** que cria e retorna uma nova árvore binária de busca na qual a raiz é um ponteiro para o nó cujo valor é **c**. O nó que tem o valor **c** e suas subárvores à esquerda e à direita devem ser removidos da árvore original **a**. Veja o exemplo abaixo: