

INF 1010

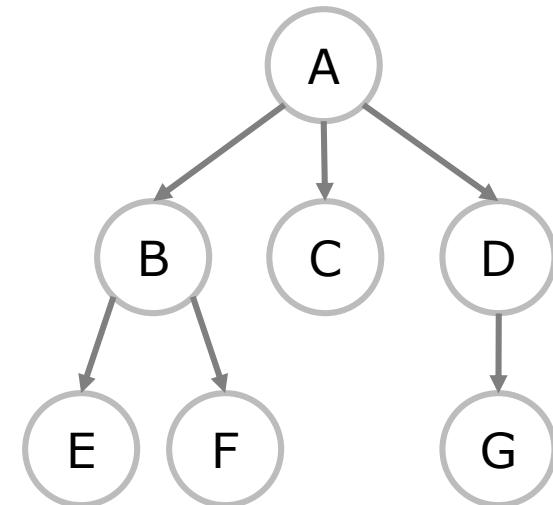
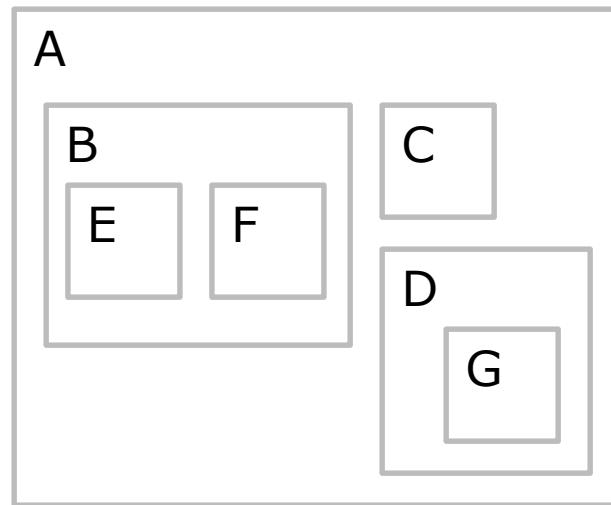
Estruturas de Dados Avançadas

Árvores e Árvores binárias

Árvore

- estrutura hierárquica:

- A
- ...B
-E
-F
- ...C
- ...D
-G



- $(A (B (E, F)), C, (D (G)))$

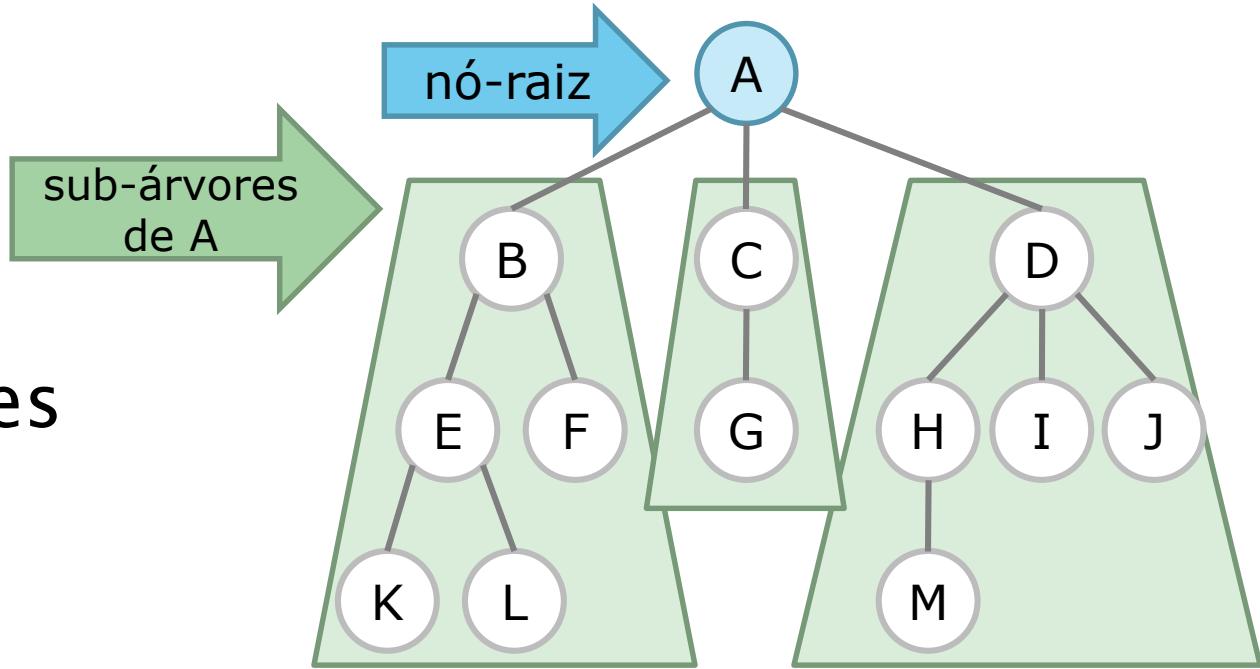
Árvore - definições

árvore:

- nó raiz
- sub-árvores

nó:

- informação
- ramos



Árvore - definições

grau de um nó:

- número de sub-árvores do nó

grau de A = 3

grau de B = 2

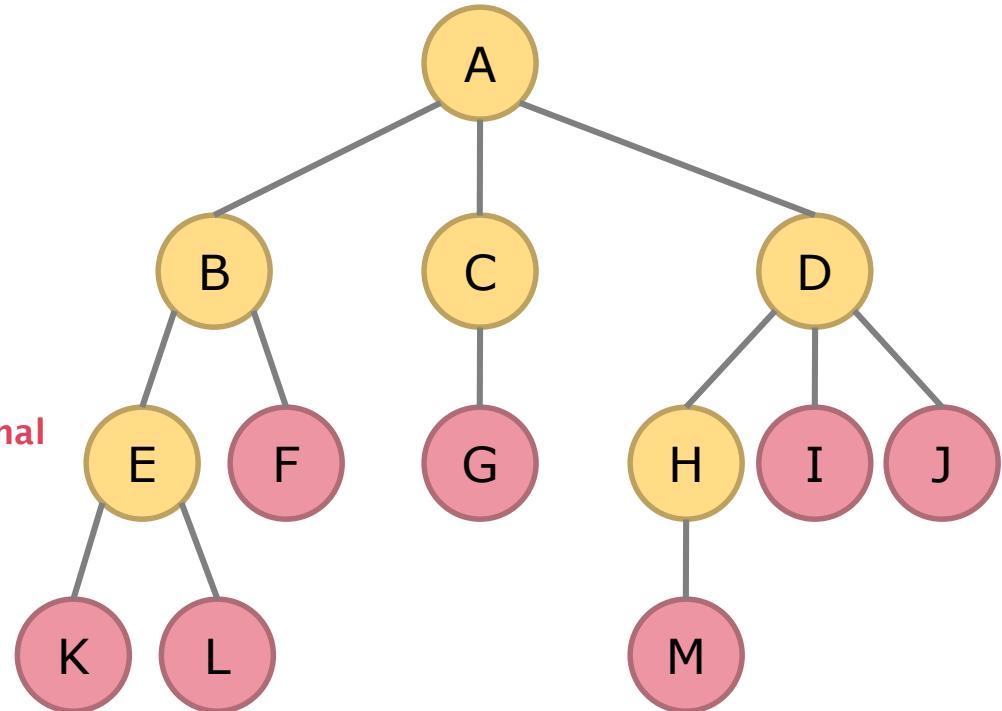
grau de F = 0

se grau = 0

- nó é chamado de **folha** ou **terminal**
- { F, G, I, J, K, L, M }

se grau > 0

- nó é chamado de **não-folha** ou **não-terminal**
- { A, B, C, D, E, H }



grau da árvore

- maior dentre os graus dos nós
- grau da árvore de exemplo = 3



nó não-terminal



folha ou nó terminal

Árvore - definições

filhos de A

- raízes das sub-árvores de A
- {B, C, D}

pai/progenitor de B: A

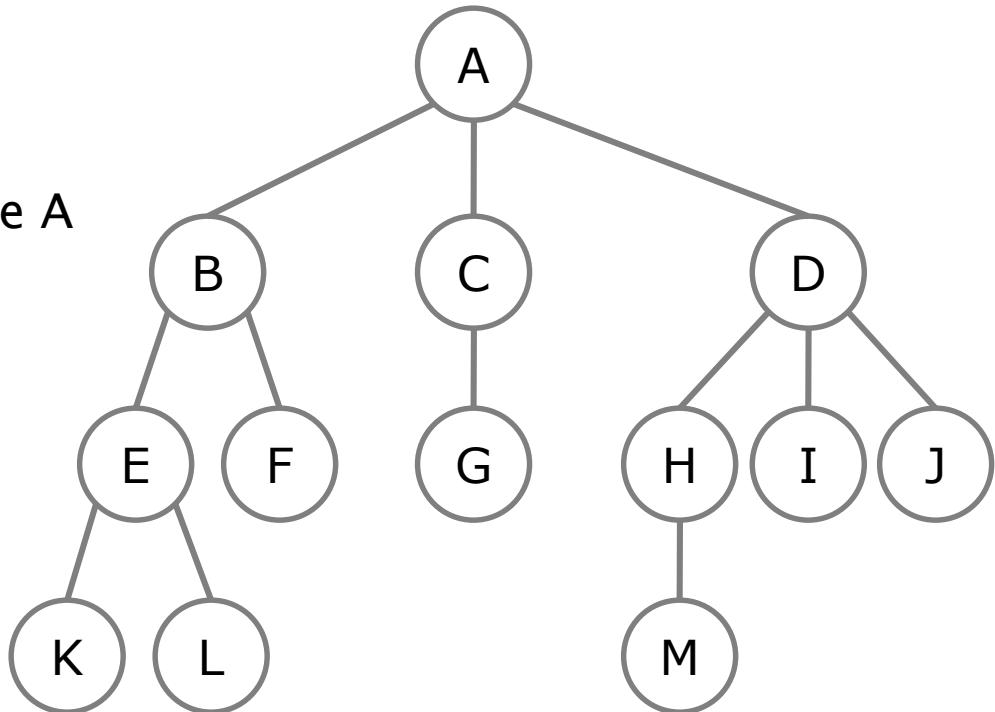
- X é pai dos seus filhos

irmãos

- nós que têm um mesmo pai
- {B, C, D}; {E, F}; {H, I, J}; {K, L}

ancestrais de K

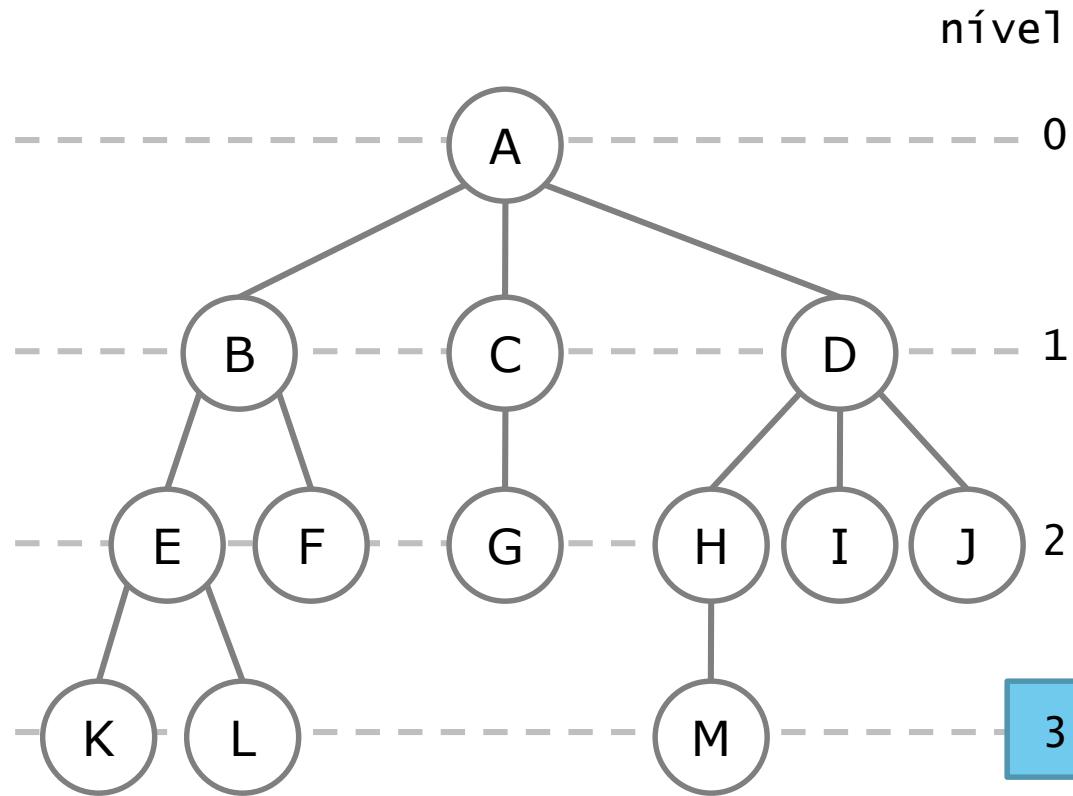
- nós no caminho da raiz até K
- {A, B, E}



Árvore - definições

nível (de um nó)

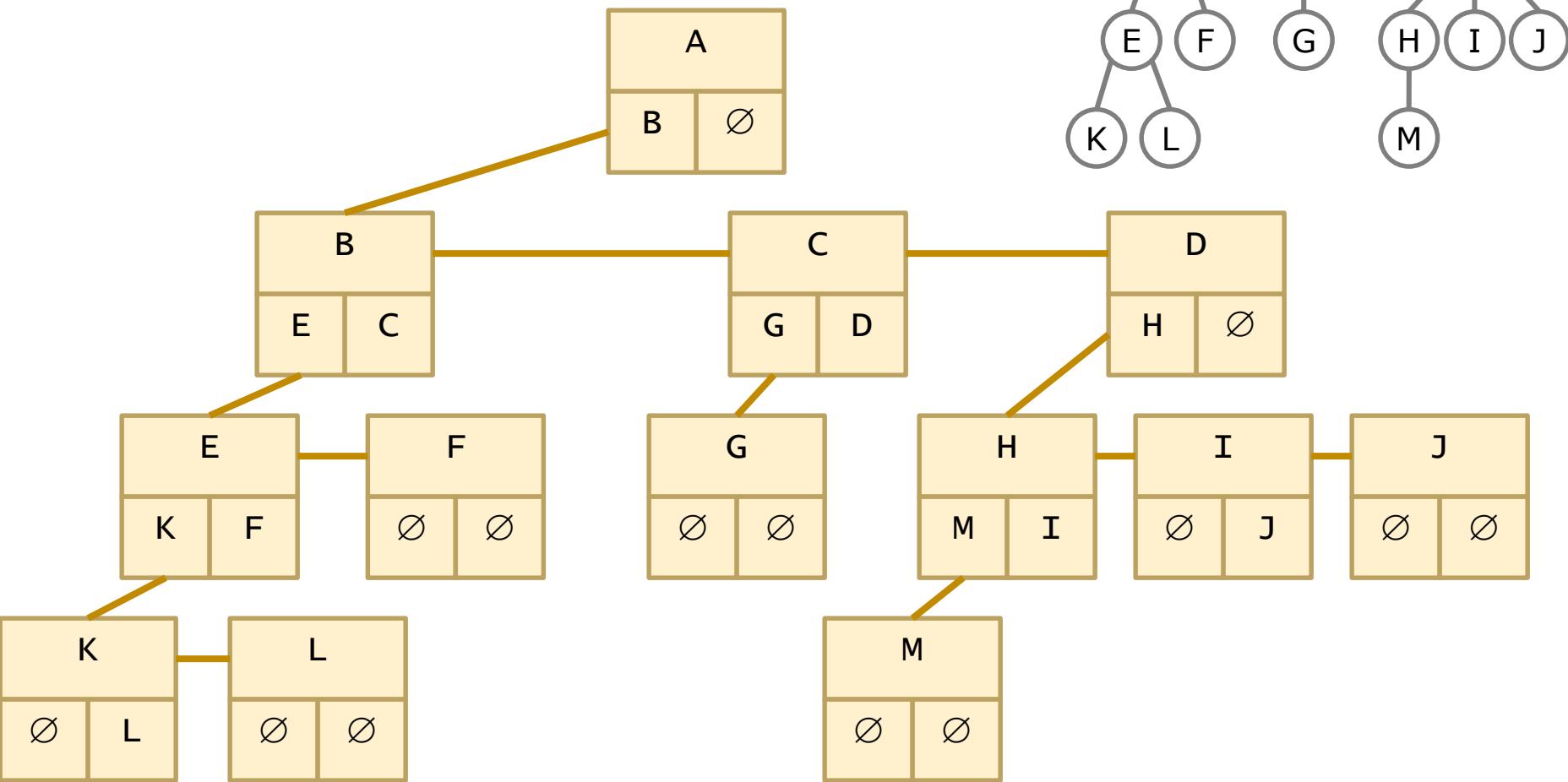
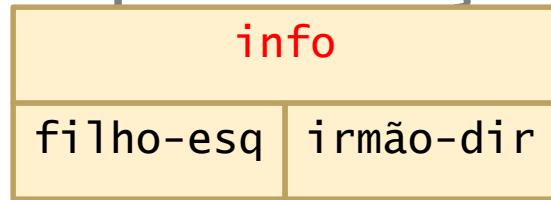
- raiz tem nível 0
- se nó X tem nível n , seus filhos têm nível $n+1$



altura ou profundidade (da árvore)

- maior nível dentre todos os nós
- No exemplo: $h = 3$

Representação de uma árvore

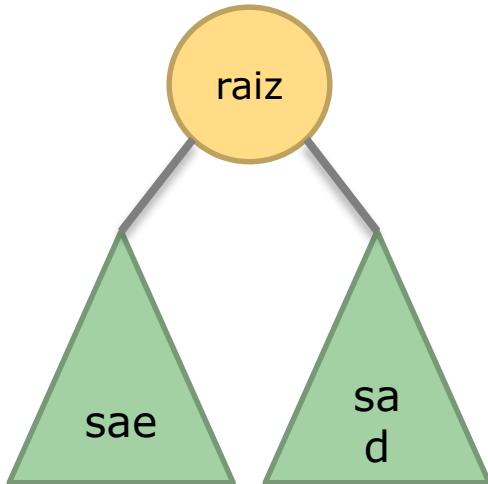


Árvore binária - definições

- \emptyset (árvore vazia)
- {raiz, sub-árvore esquerda , sub-árvore direita}, onde sae e sad são conjuntos disjuntos

\emptyset

ou

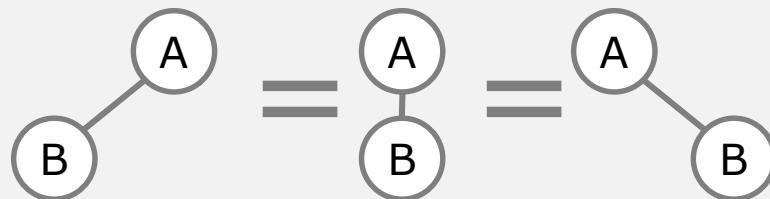


```
/* nó da árvore binária */  
struct arvbin {  
    char info;  
    struct arvbin *esq;  
    struct arvbin *dir;  
};
```

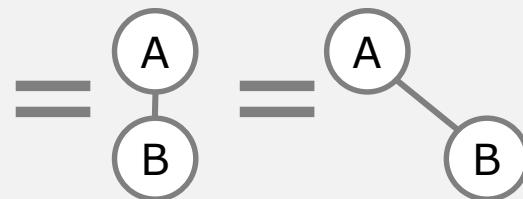
Árvore binária - definições

Nota: árvore binária não é árvore comum

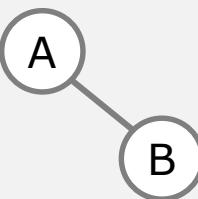
- árvore



(A (B))



(A (B))

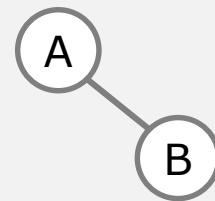


(A (B))

- árvore binária



(A (B) ())



(A () (B))

Árvore binária - conceitos

número máximo de nós no nível i : $n_i = 2^i$

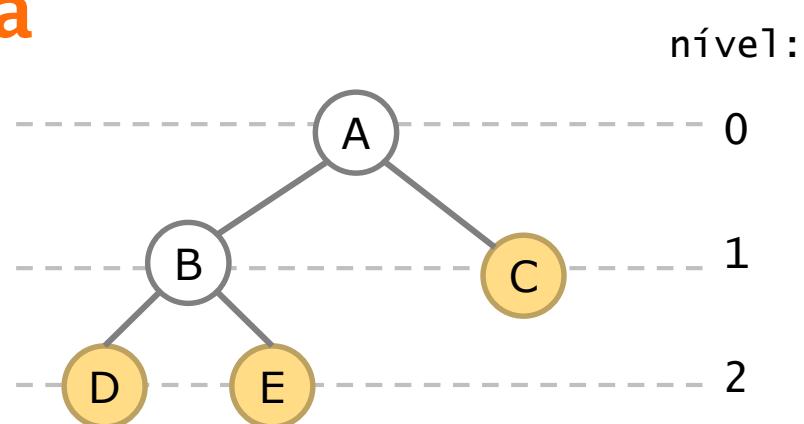
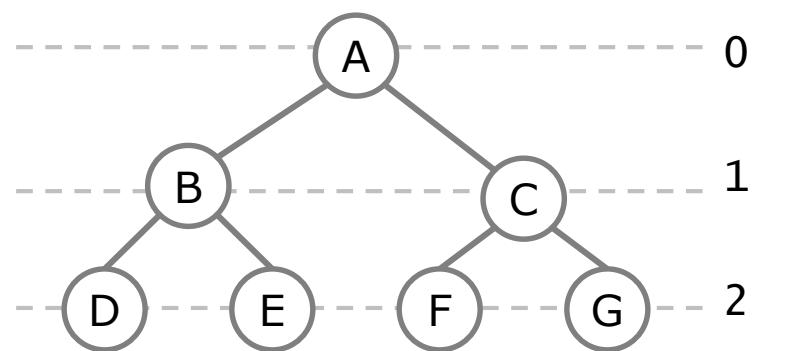
número máximo de nós na árvore de altura k :

- $n_{\max} = 2^k + \dots + 2^2 + 2 + 1 = 2^{k+1} - 1$

Árvore binária - conceitos

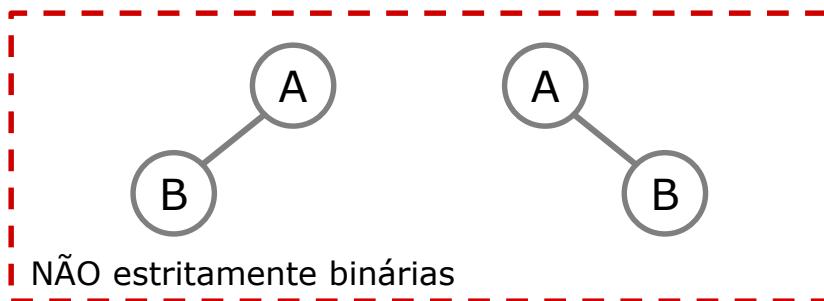
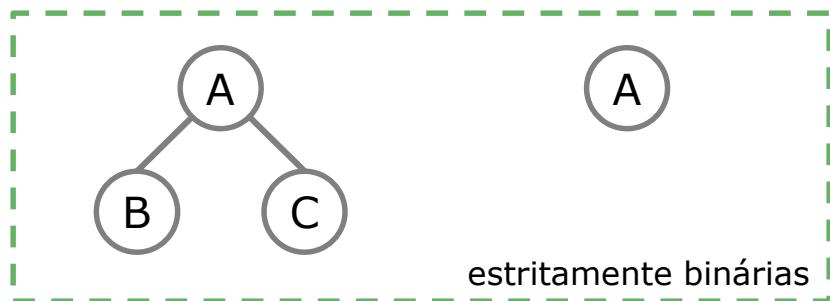
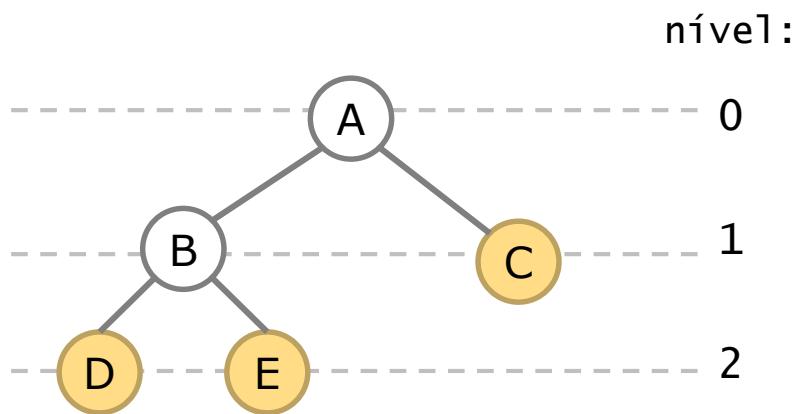
Árvore binária **cheia** de altura k:

- árvore com $2^{k+1} - 1$ nós
- Exemplo: para $k = 2$, árvore binária cheia possui $2^3 - 1 = 7$ nós
- Árvore binária **completa**
 - toda folha está no último ou penúltimo nível



Árvore binária – conceitos (cont.)

- Árvore **estritamente binária**
 - cada nó tem 0 ou dois filhos



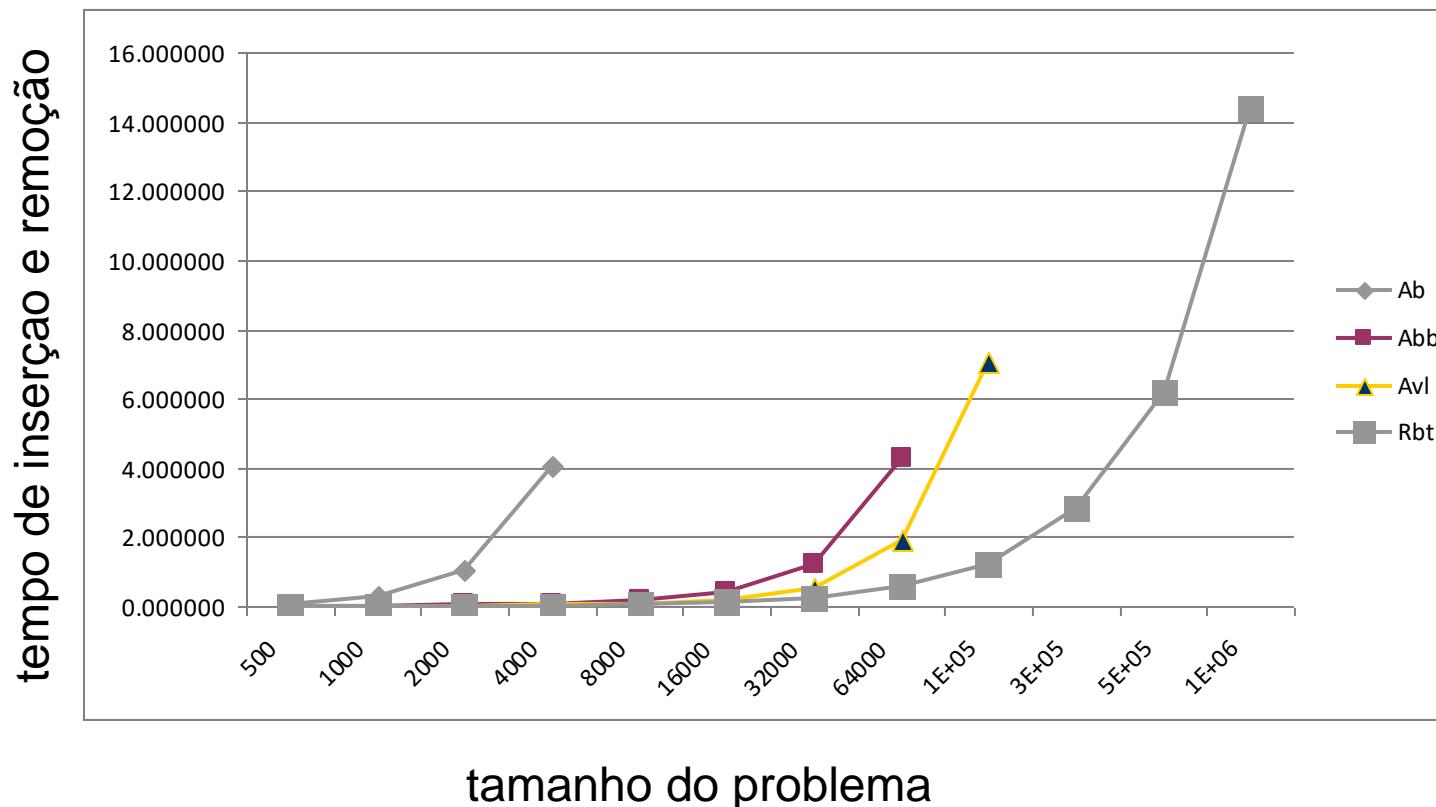
O que vamos estudar de árvores binárias?

Ab = árvore binária

Abb = árvore binária de busca

Avl = árvore binária de busca, balanceada

Rbt = árvores rubro-negras ou vermelho e preto (red-black tree)



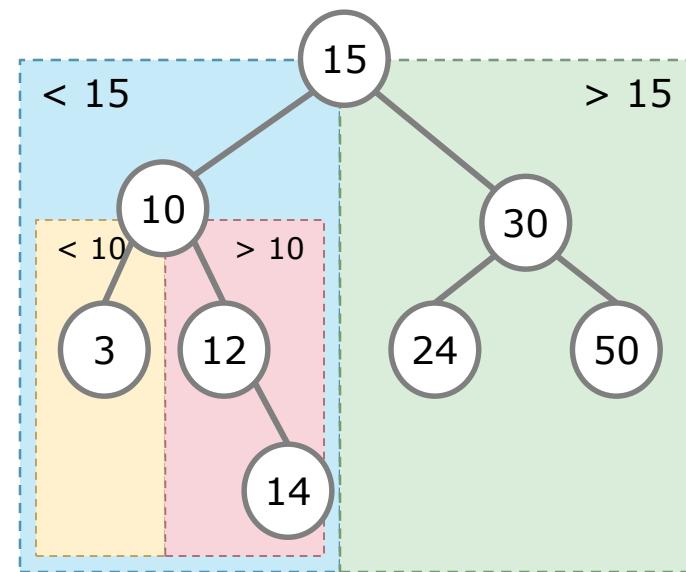
Definição de ABBs

- Uma ABB é uma árvore binária vazia, ou uma árvore tal que

1. cada nó possui uma chave
2. as chaves na sub-árvore esquerda (se houver) são menores do que a chave da raiz
3. as chaves na sub-árvore direita (se houver) são maiores do que a chave da raiz
4. as sub-árvore esquerda e direita são árvores binárias de busca

\emptyset

$\underline{\underline{}}$



Implementação de ABBs em C (sem ponteiro para o pai)

abb.h

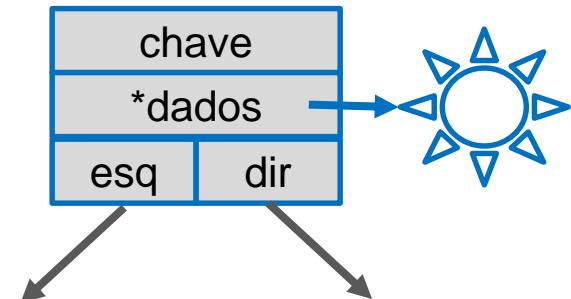
```
typedef struct smapa Mapa;  
  
Mapa* cria (void);  
Mapa* insere (Mapa* raiz, int chave,  
tdados *novosdados);  
tdados *busca (Mapa *raiz, int chave);
```

```
Mapa* cria(void)  
{  
    return NULL;  
}
```

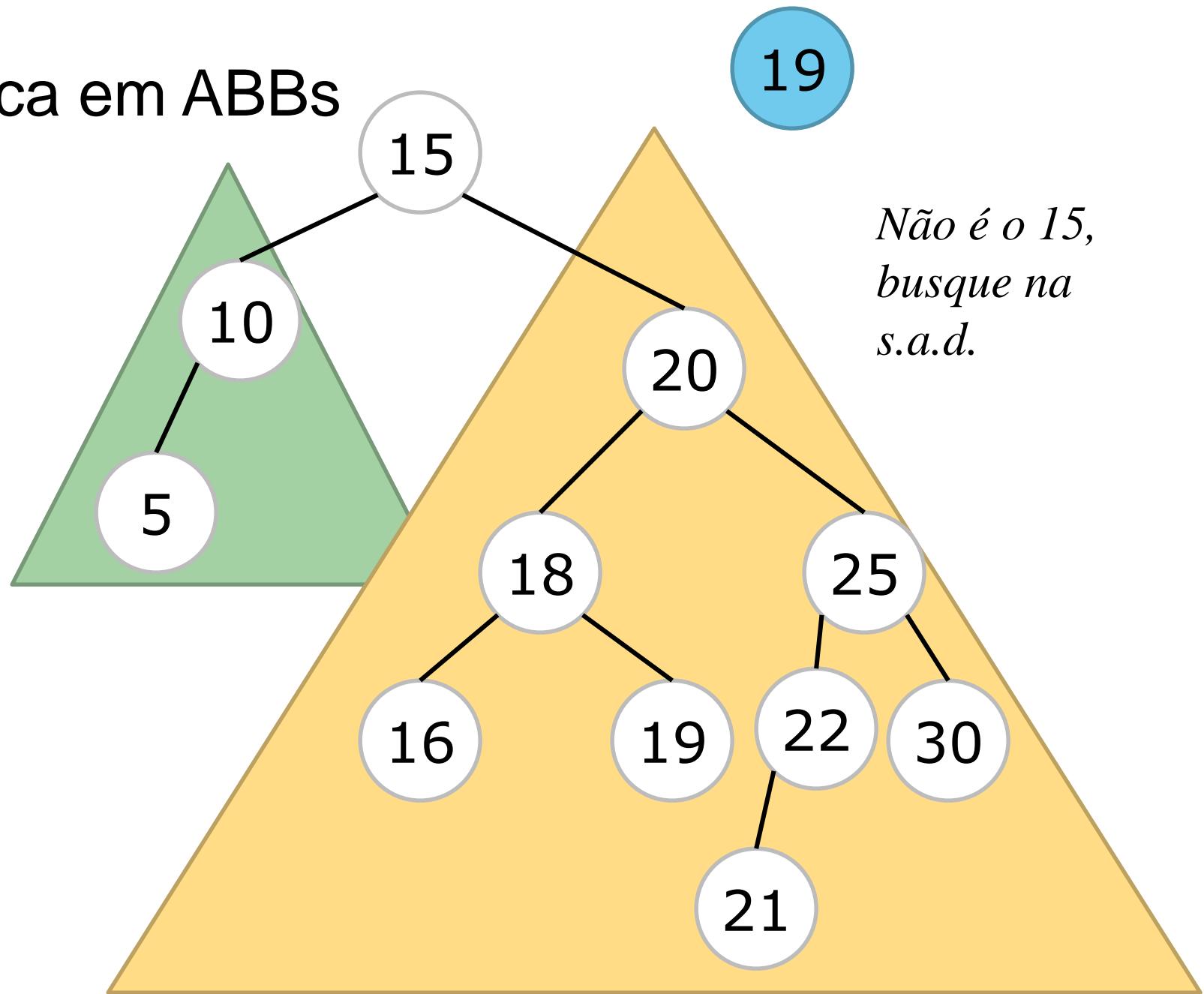
abb.c

```
struct smapa {  
    int chave;  
    tdados *dados;  
    Mapa* esq;  
    Mapa* dir;  
};
```

06/04/2022



Busca em ABBs



Busca em ABs

```
tdados* busca (Mapa* raiz, int chave);
```

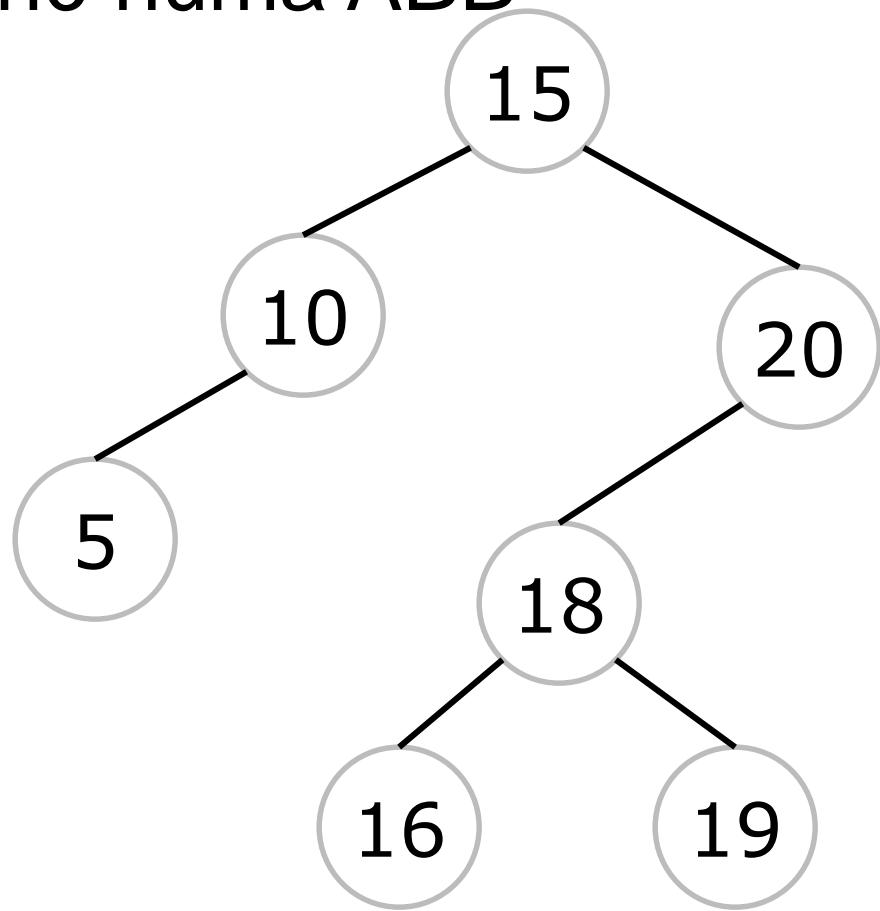
1. Comece a busca pelo nó raiz
2. Se a árvore for **vazia** então retorne **NULL**
3. Se a chave procurada for **menor** que a chave do nó,
procure na sub-árvore à esquerda e
responda com a resposta que você receber
4. Se a chave procurada for **maior** que a chave do nó,
procure na sub-árvore à direita e
responda com a resposta que você receber
5. Se for **igual** responda com o **endereço do nó**

Busca em ABBs - Algoritmo Recursivo

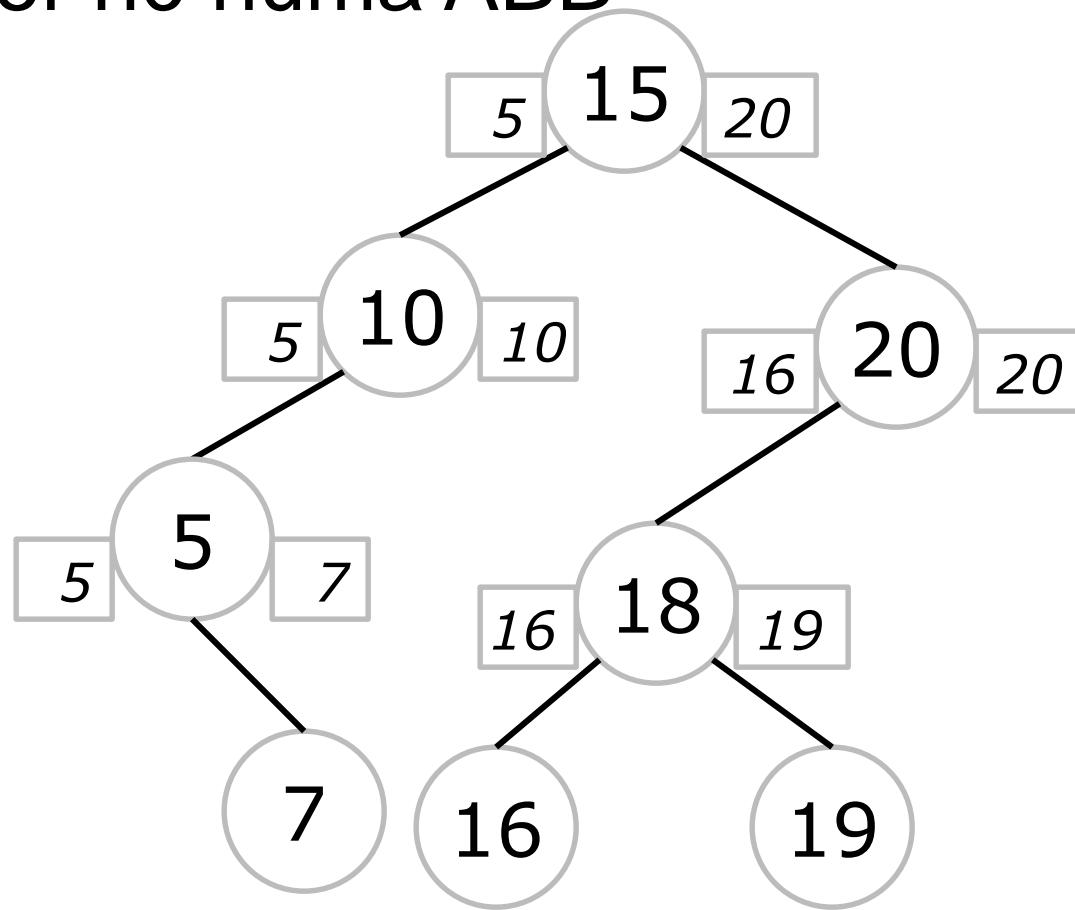
```
tdados* busca (Mapa* r, int c)
{
    if (r == NULL)
        return NULL;
    else if (c < r->chave)
        return busca (r->esq, c);
    else if (c > r->chave)
        return busca (r->dir, c);
    else return r->dados;
}
```

1. Comece a busca pelo nó raiz
2. Se a árvore for **vazia** retorne **NULL**
3. CC se a chave procurada for **menor** que a chave do nó, **procure na sub-árvore à esquerda** e responda com a resposta que você receber
4. CC se a chave procurada for **maior** que a chave do nó, **procure na sub-árvore à direita** e responda com a resposta que você receber
5. CC se for **igual** responda com o **endereço do nó**

Menor nó numa ABB



Menor nó numa ABB

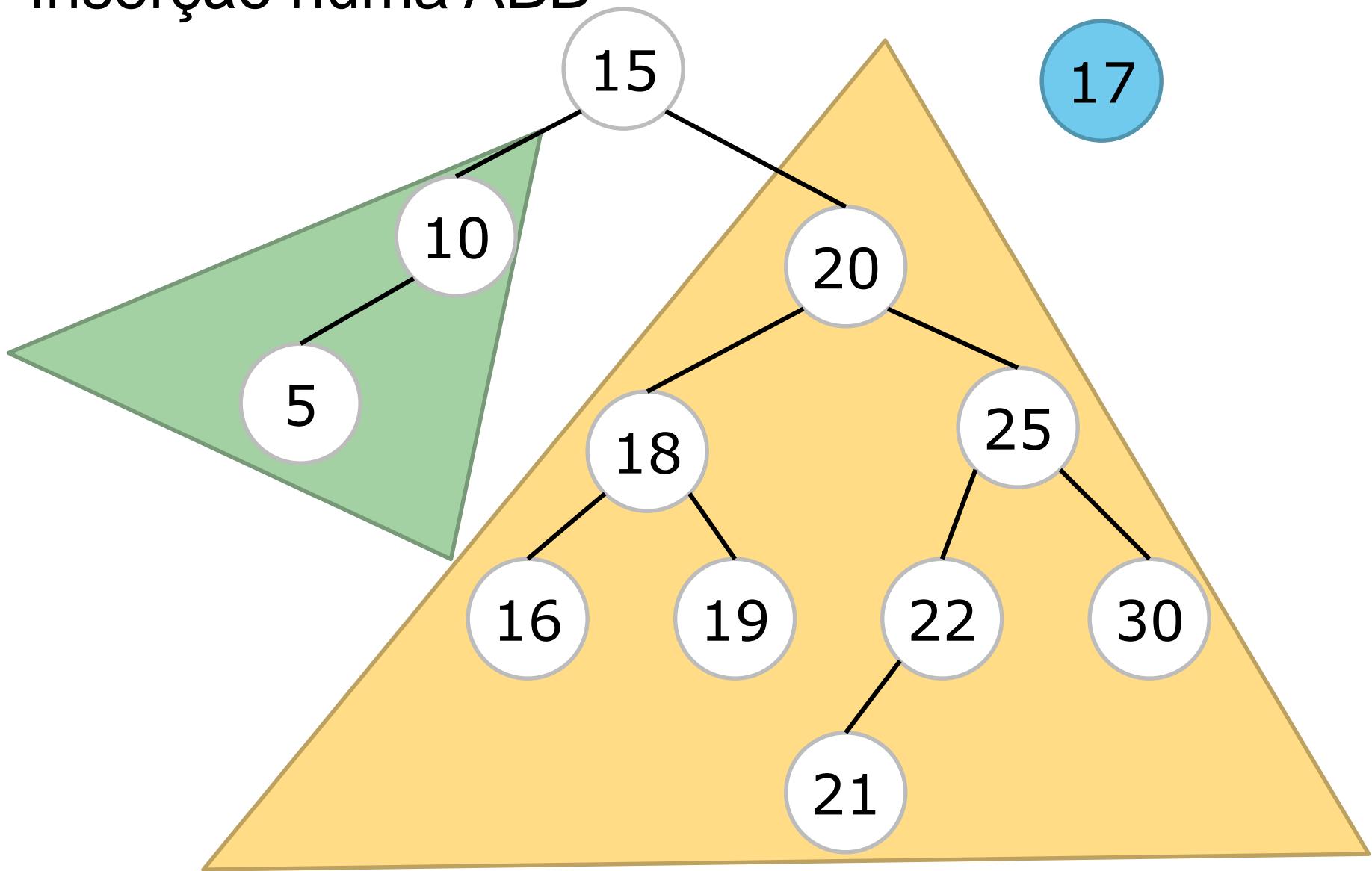


Menor nó numa ABB

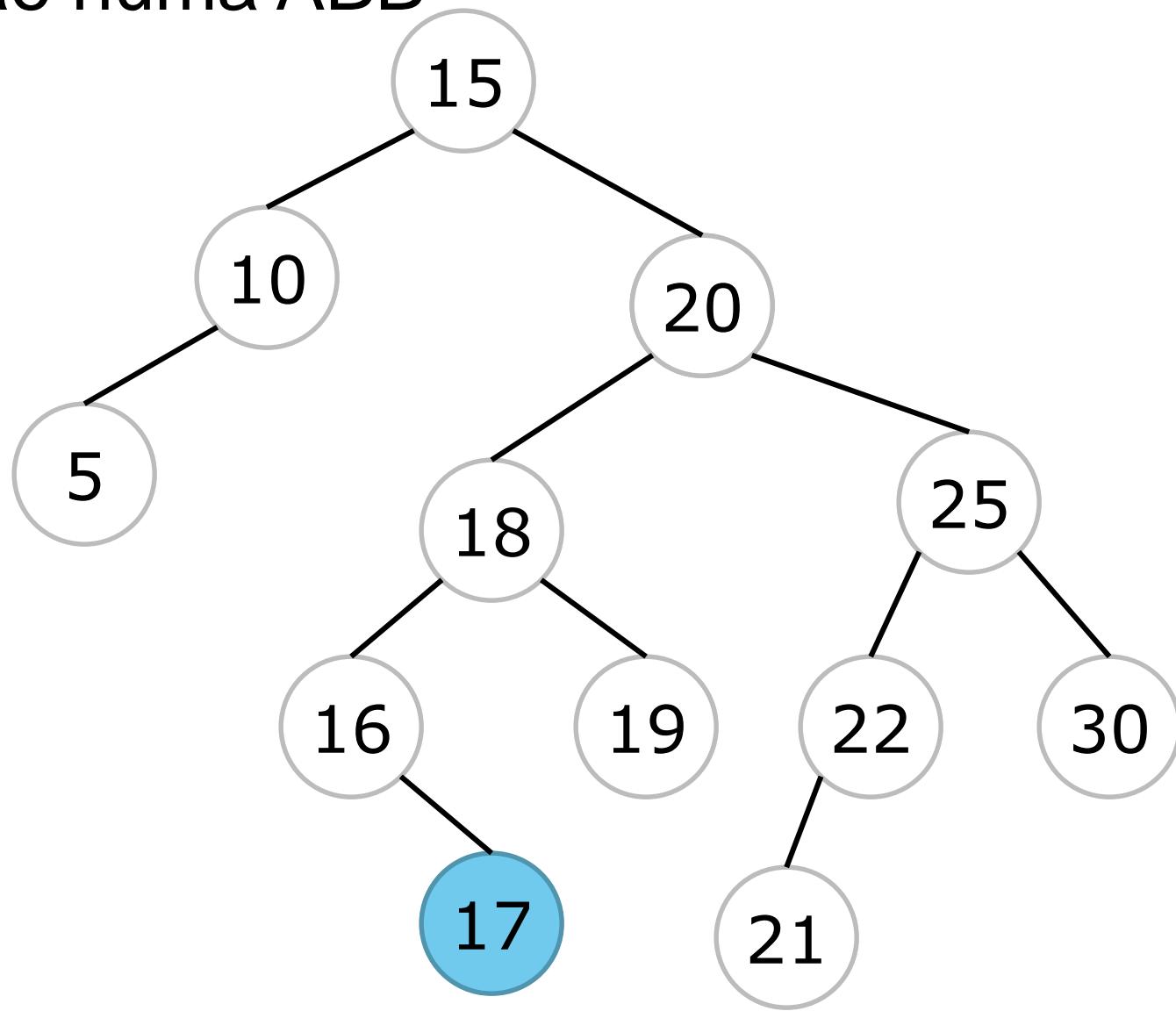
```
Mapa* min (Mapa* r)
```

- É o nó mais à esquerda da árvore
 - 1. Começando pelo nó raiz
 - 2. Se a árvore for vazia retorne NULL
 - 3. Caso contrário, caminhe sempre à esquerda enquanto o filho esquerdo não for NULL

Inserção numa ABB



Inserção numa ABB



Inserção numa ABB

```
Mapa* insere (Mapa* r, int chave, tdados *dados);
```

Inserção recursiva numa ABB

- Faça uma função auxiliar para criar um nó

```
static Mapa* cria_no (int c, tdados* d) {  
    Mapa* no = (Mapa*) malloc(sizeof(Mapa));  
    no->chave = c;  
    no->dados = d;  
    no->esq = no->dir = NULL;  
    return no;  
}
```

Inserção recursiva numa ABB

```
Mapa* insere (Mapa* r, int c, tdados *dados){  
    if (r==NULL)  
        return cria_no(c, dados);  
    else if (val < r->chave)  
        r->esq = insere(r->esq,c,dados);  
    else /* if (val > r->chave) */ {  
        r->dir = insere(r->dir,c,dados);  
    }  
    return r;  
}
```

Inserção iterativa numa ABB

```
Mapa* insere_iterativa (Mapa* r, int val,  
tdados *dados);
```

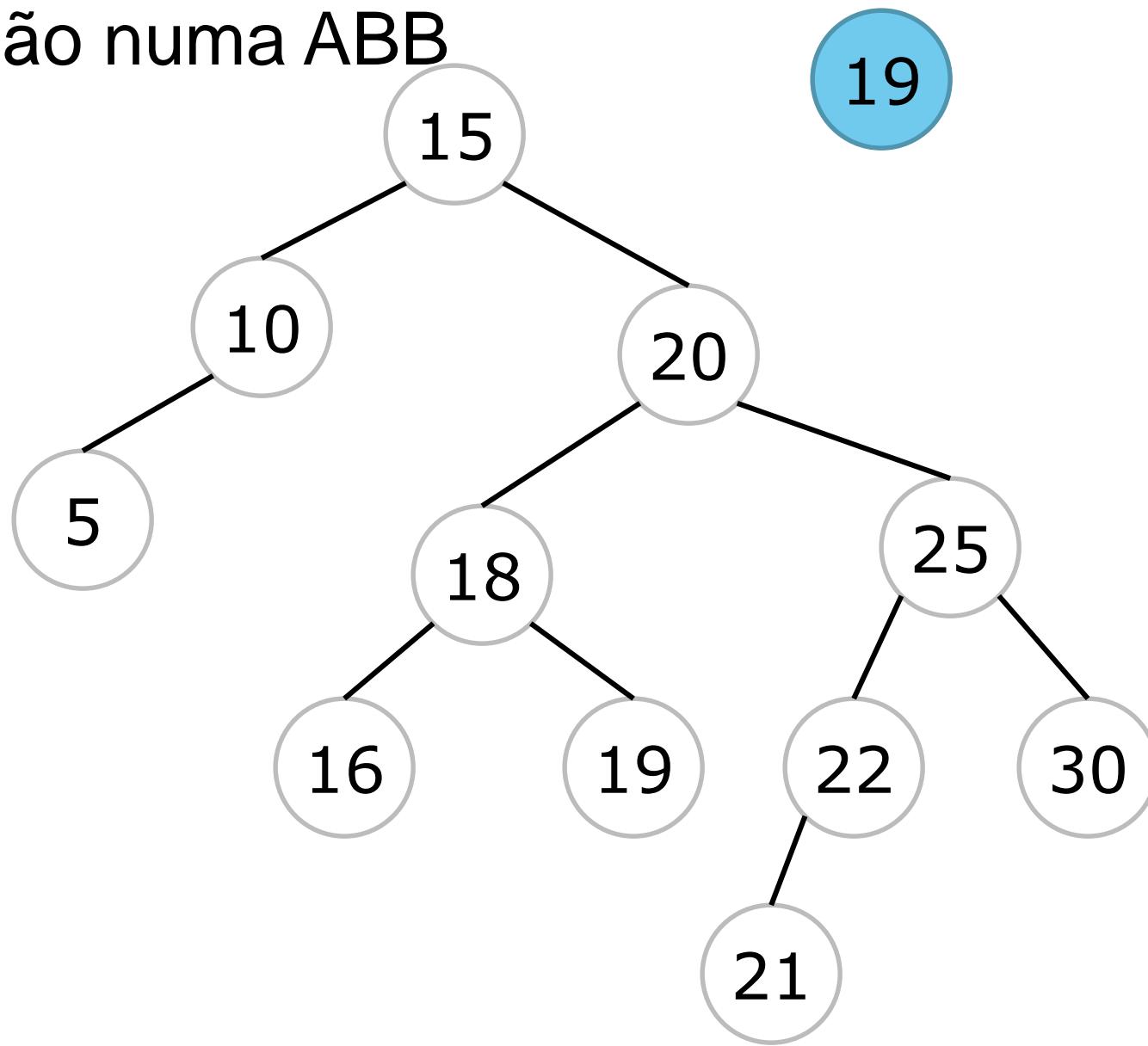
1. Se a árvore for vazia, então crie um nó e retorne
2. Senão, comece a busca pelo nó raiz, desça na árvore mantendo o nó anterior (pai)
3. Enquanto o nó não for **NULL** ou não contiver a chave dada:
 1. Se a chave do nó for **maior** que a chave dada, vá para o filho à **direita**
 2. Se a chave do nó for **menor** que a chave dada, vá para o filho à **esquerda**

Inserção iterativa numa ABB

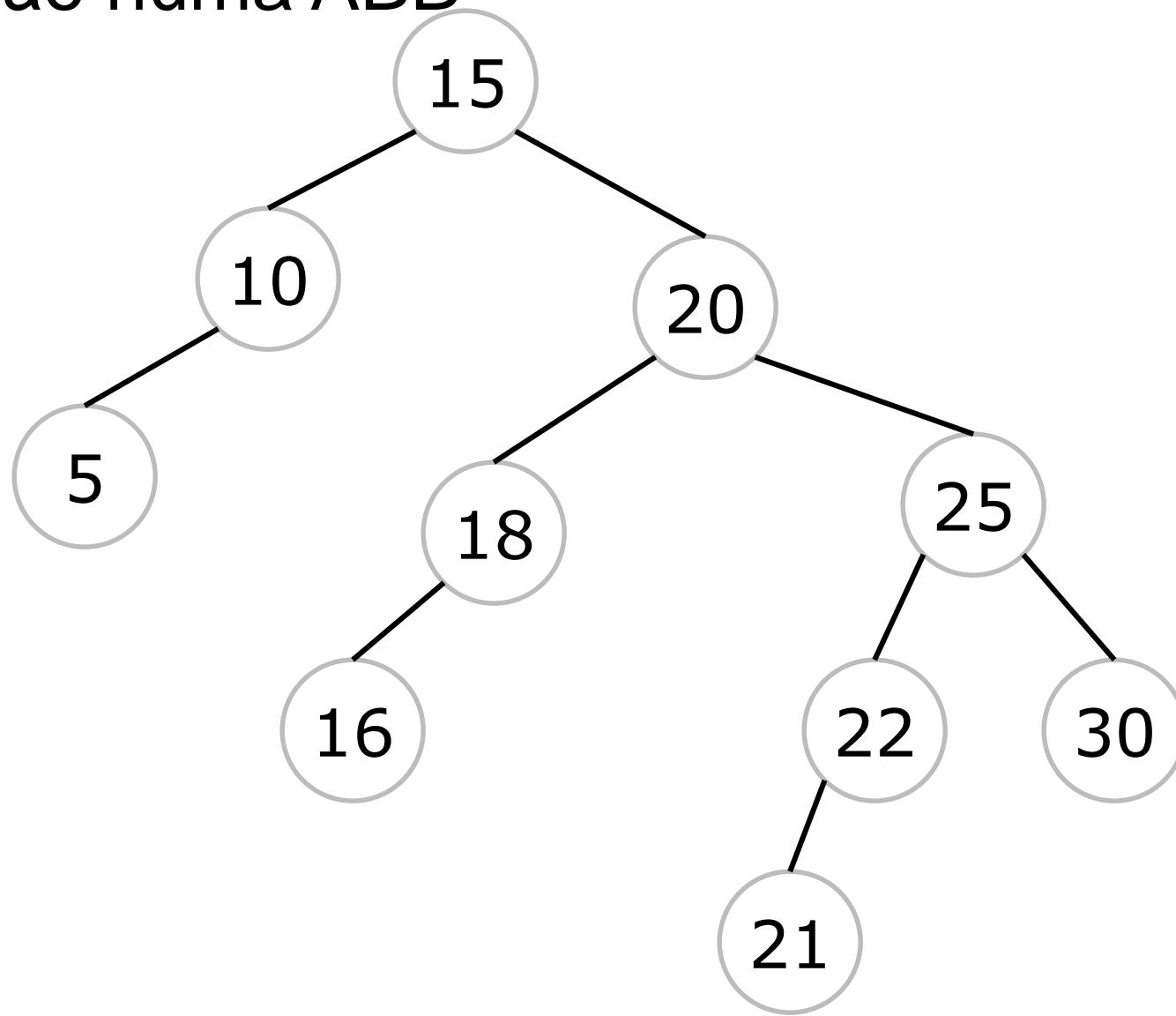
Remoção de um nó de uma ABB

- Três casos:
 1. Nó folha
 2. Nó possui uma sub-árvore
 3. Nó possui duas sub-árvore

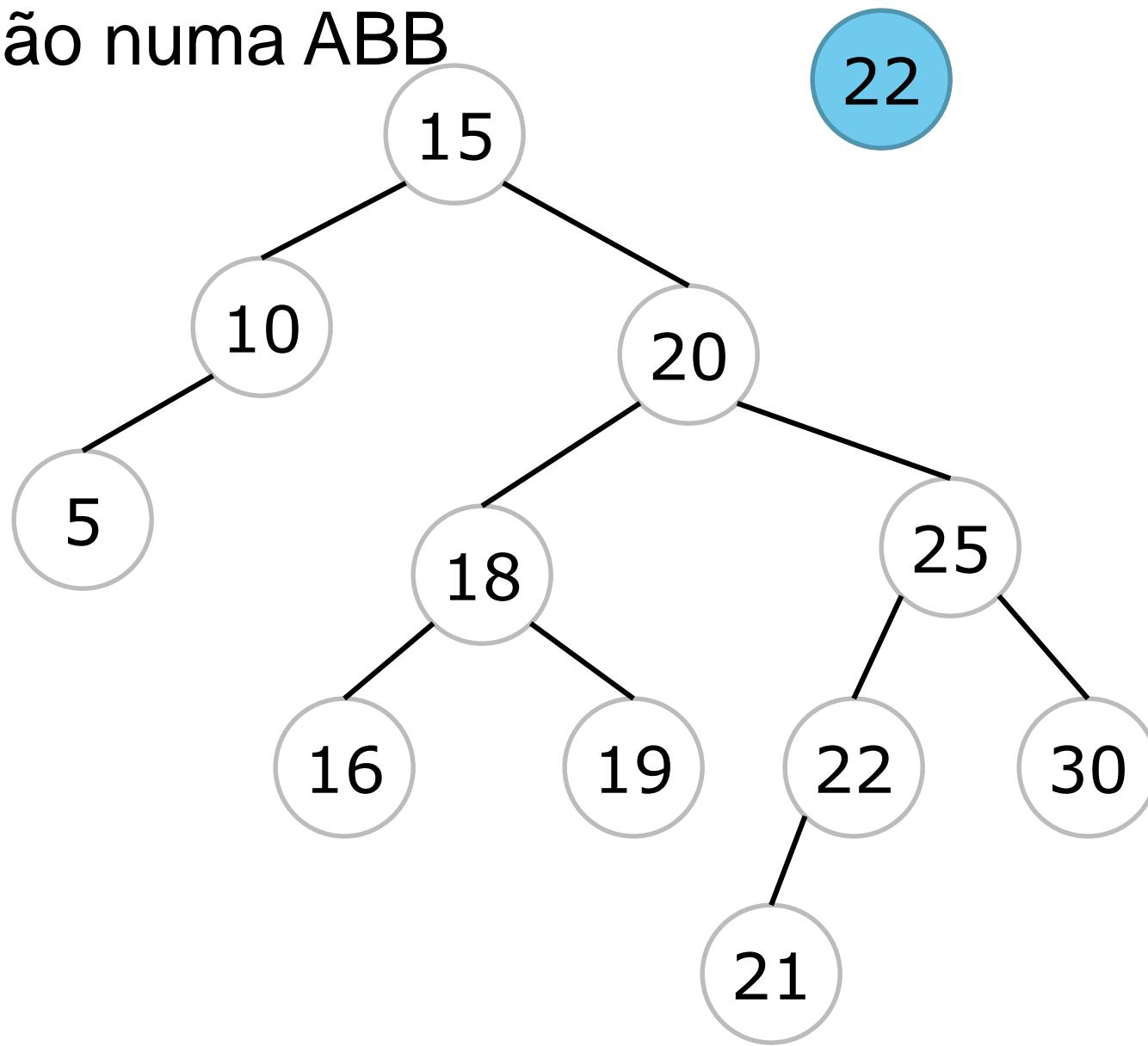
Remoção numa ABB



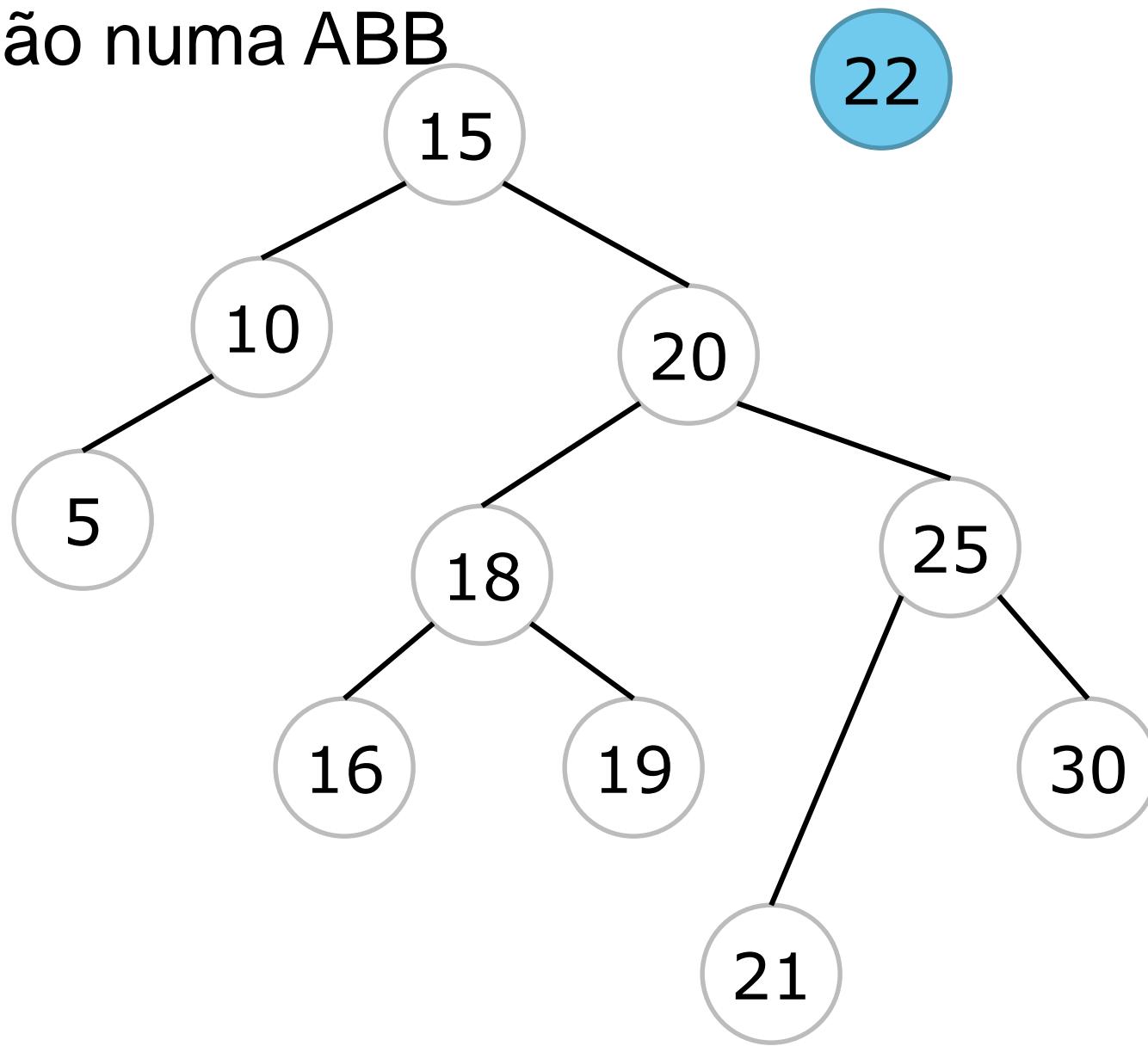
Remoção numa ABB



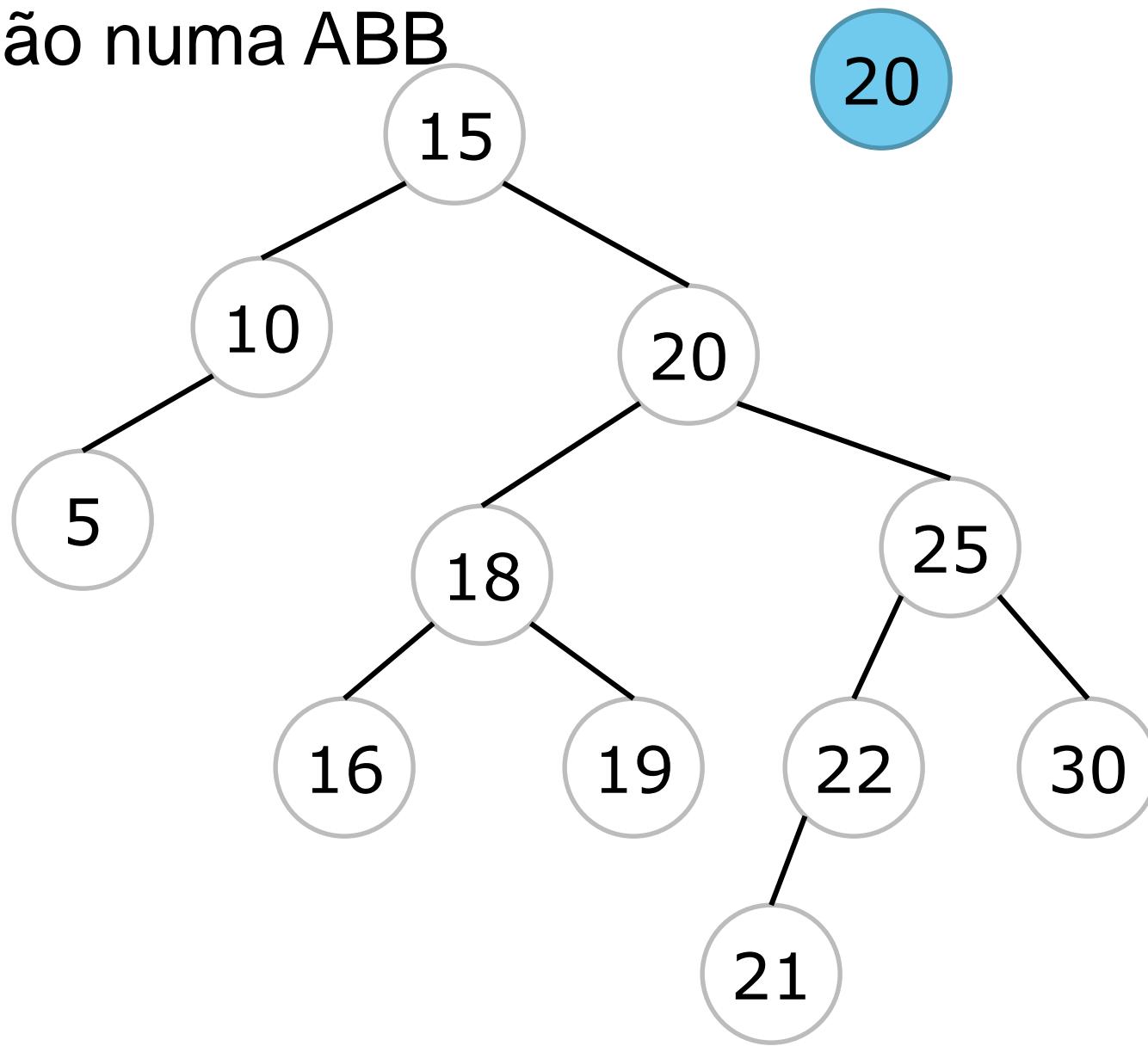
Remoção numa ABB



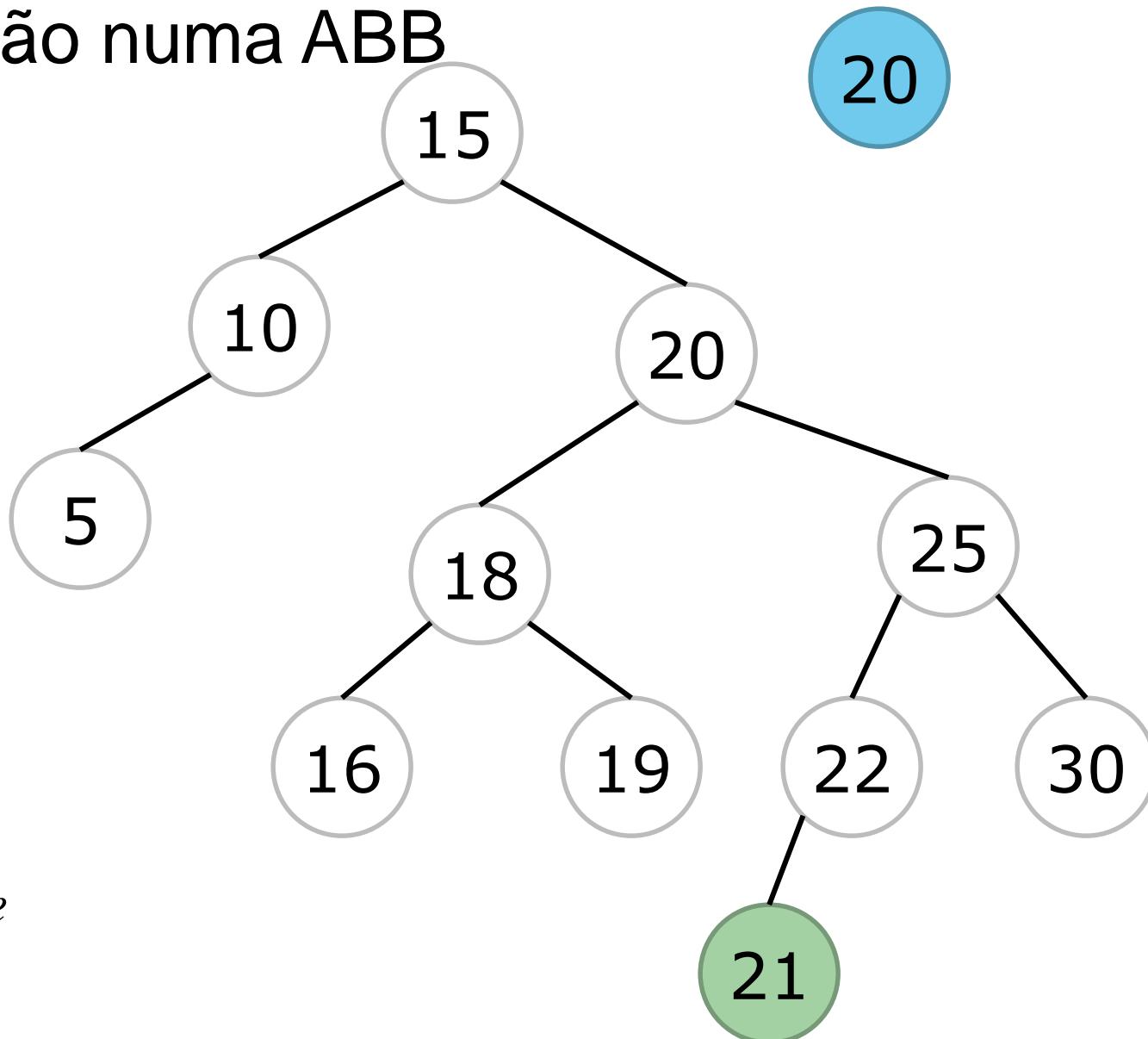
Remoção numa ABB



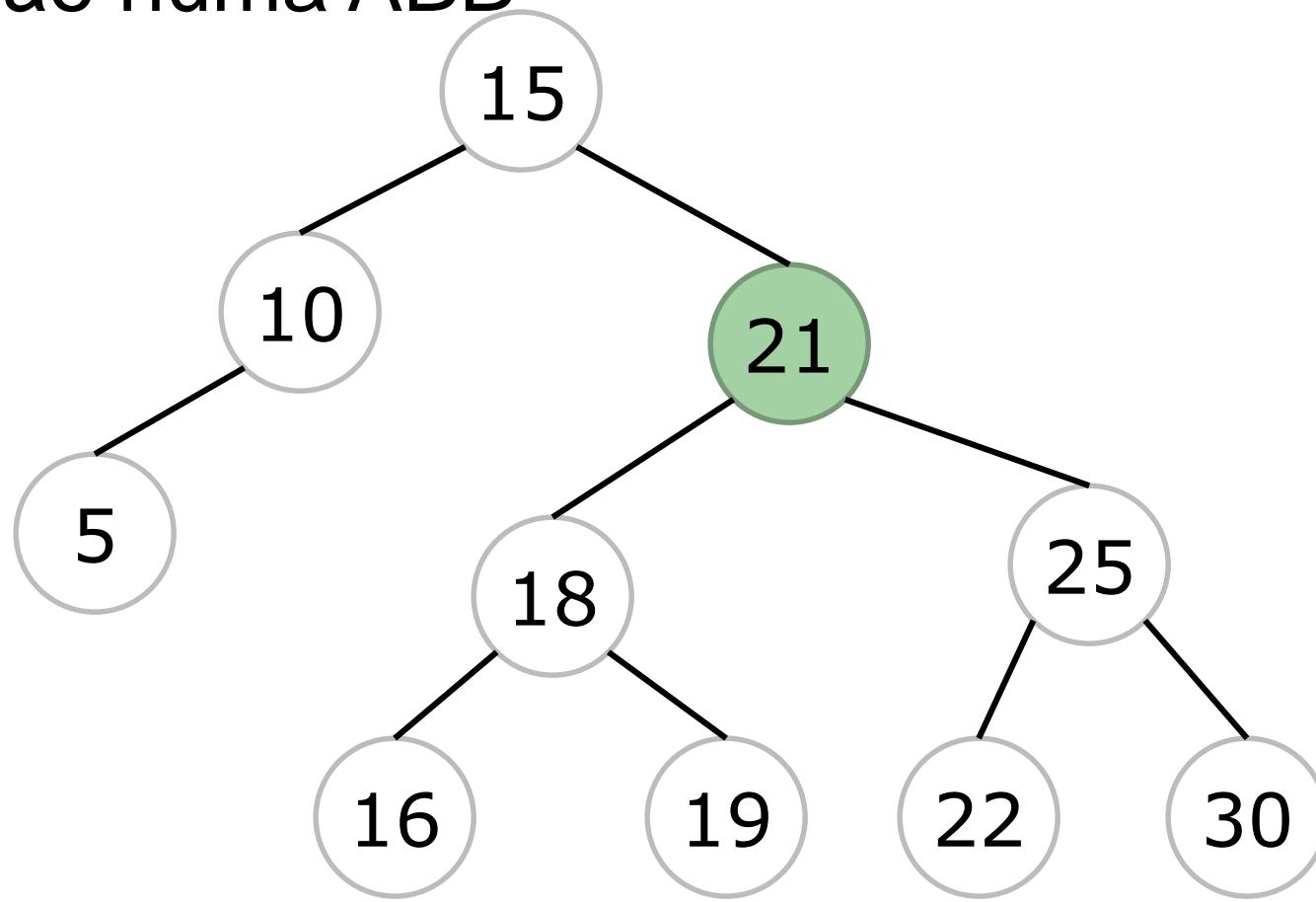
Remoção numa ABB



Remoção numa ABB



Remoção numa ABB



Remoção de um nó de uma ABB

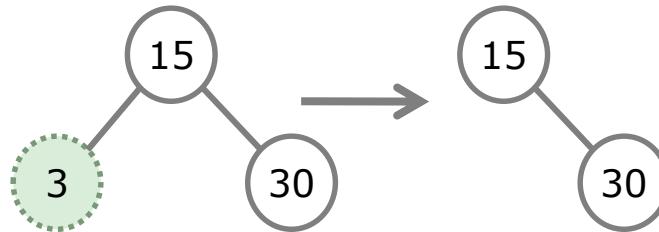
- Três casos:

1. nó folha

- simplesmente elimina o nó

2. nó possui uma sub-árvore

3. nó possui duas sub-árvores



Remoção de um nó de uma ABB

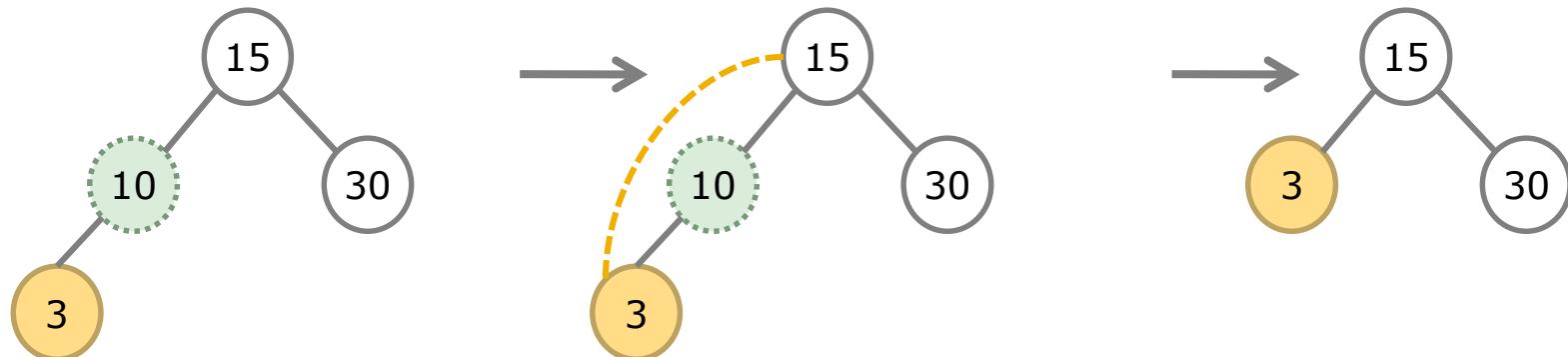
- Três casos:

1. nó folha

2. nó possui uma sub-árvore [dois subcasos: sae, sad]

- promove a sub-árvore

3. nó possui duas sub-árvore



Remoção de um nó de uma ABB

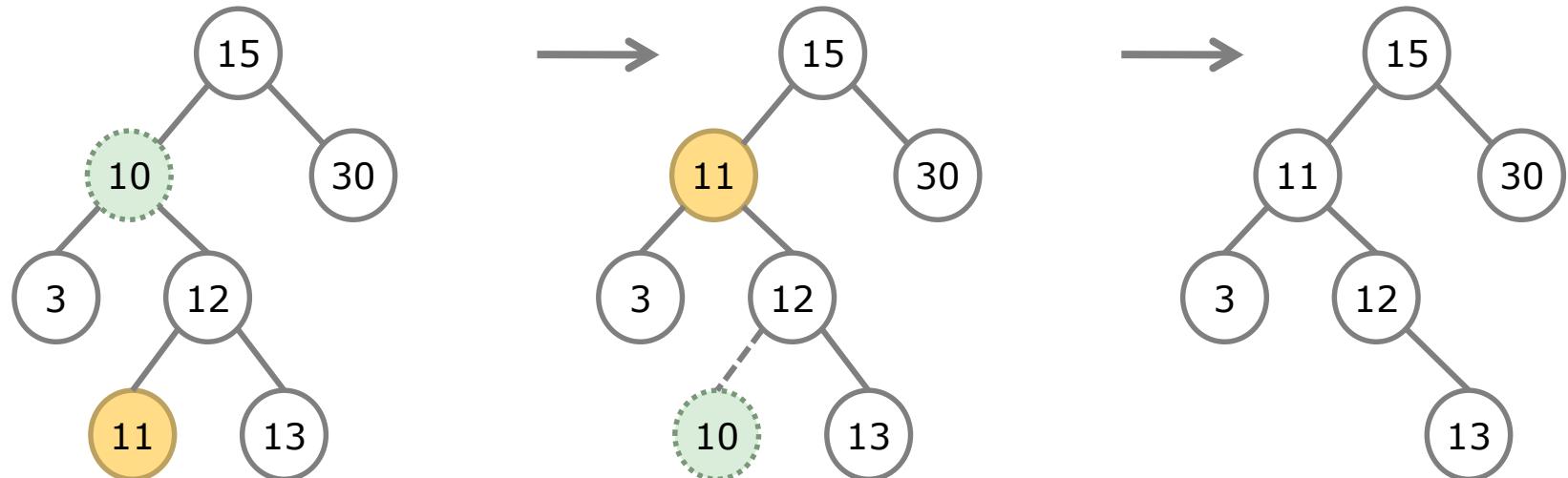
- Três casos:

1. nó folha

2. nó possui uma sub-árvore

3 → nó possui duas sub-árvores

1. coloque a informação do sucessor no nó
2. remova o sucessor

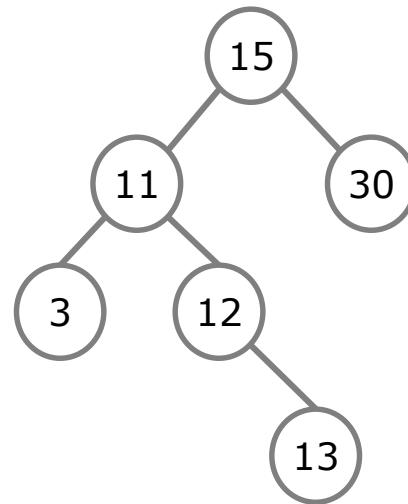


Remoção de um nó numa ABB

1. Ache o nó a ser removido
2. Se ele tiver um ou menos filhos,
faça a ligação avô-neto
3. Se ele tiver dois filhos,
procure o sucessor,
troque a info do nó pela do seu sucessor.
Apague o sucessor.

Remoção de um nó numa ABB

O sucessor é sempre o nó de menor chave da sub-arvore à direita



```

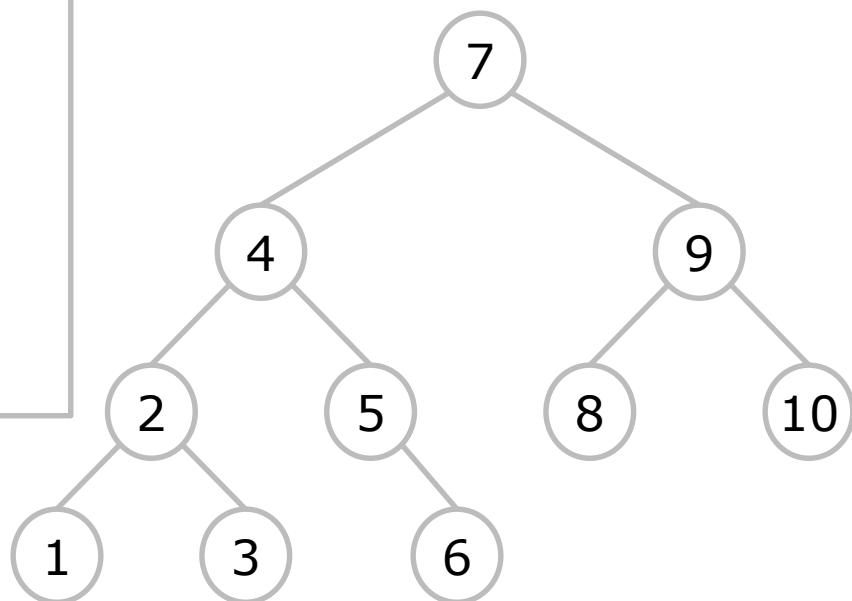
Mapa* retira (Mapa* r, int chave){  ESBOÇO
    Mapa *sucessor, *temp;

    if (r == NULL) return NULL;
    else if (chave < r->chave)
        r->esq = retira(r->esq, chave);
    else if (chave > r->chave)
        r->dir = retira(r->dir, chave);
    else {          /* achou o nó a remover */
        if (r->esq == NULL && r->dir == NULL) { /* nó sem filhos */
            free (r); r = NULL;
        }
        else if (r->esq == NULL) {/* nó só tem filho à direita */
            Mapa* t = r; r = r->dir; free (t);
        }
        else if (r->dir == NULL) {/* só tem filho à esquerda */
            Mapa* t = r; r = r->esq; free (t);
        }
        else {/* nó tem os dois filhos: busca o sucessor */
            sucessor = r->dir;
            ...
        }
    }
    return r;
}

```

balanceamento

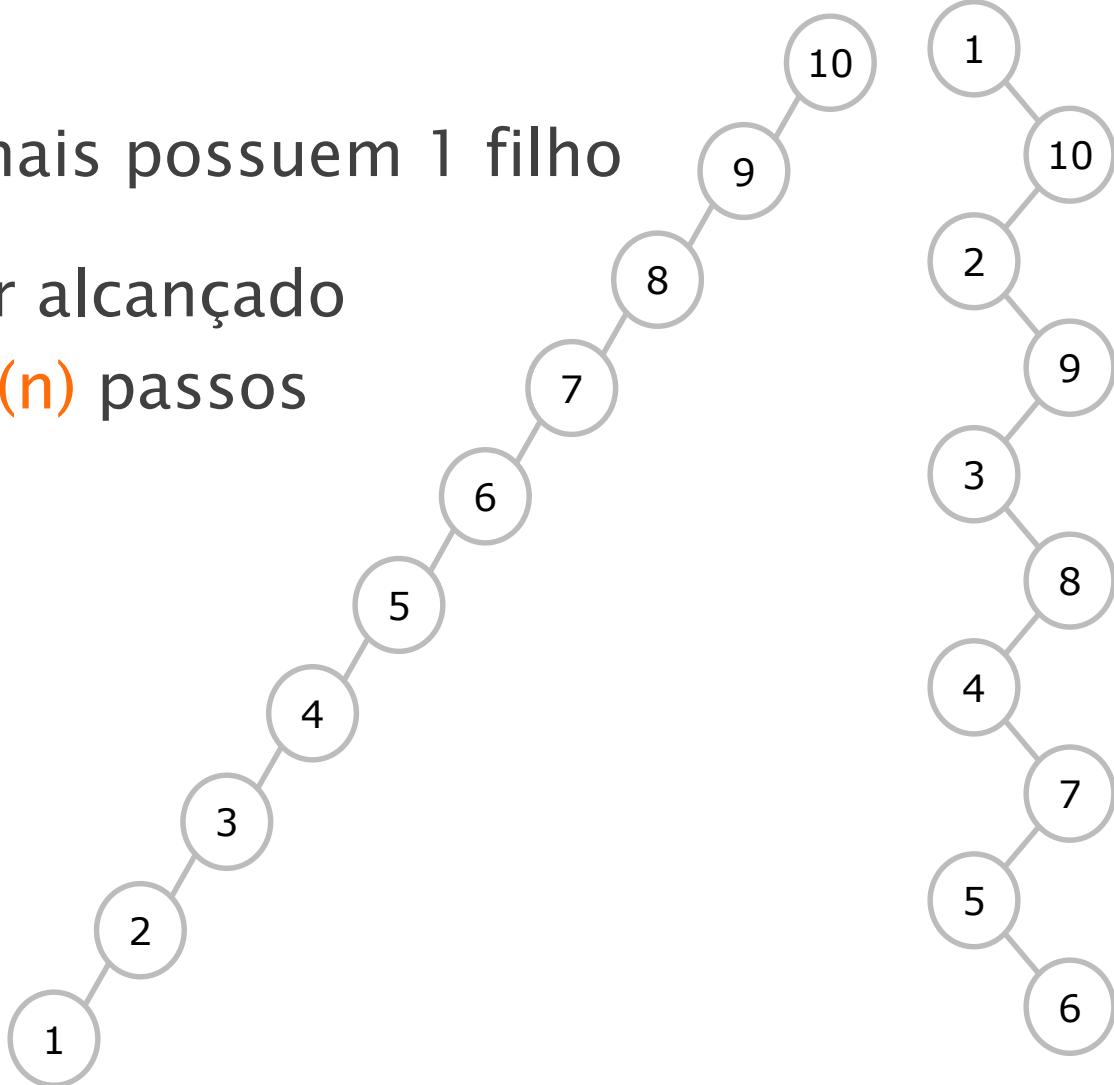
```
Mapa*a;  
a = cria_vazia();  
a = insere (a,7,_);  
a = insere (a,4._);  
a = insere (a,9._);  
a = insere (a,2._);  
a = insere (a,5._);  
a = insere (a,6._);  
a = insere (a,1,_);  
a = insere (a,8,_);  
a = insere (a,3,_);  
a = insere (a,10,_);
```



Árvore binária de busca degenerada

todos nós não terminais possuem 1 filho

qualquer nó pode ser alcançado
a partir da raiz em $O(n)$ passos



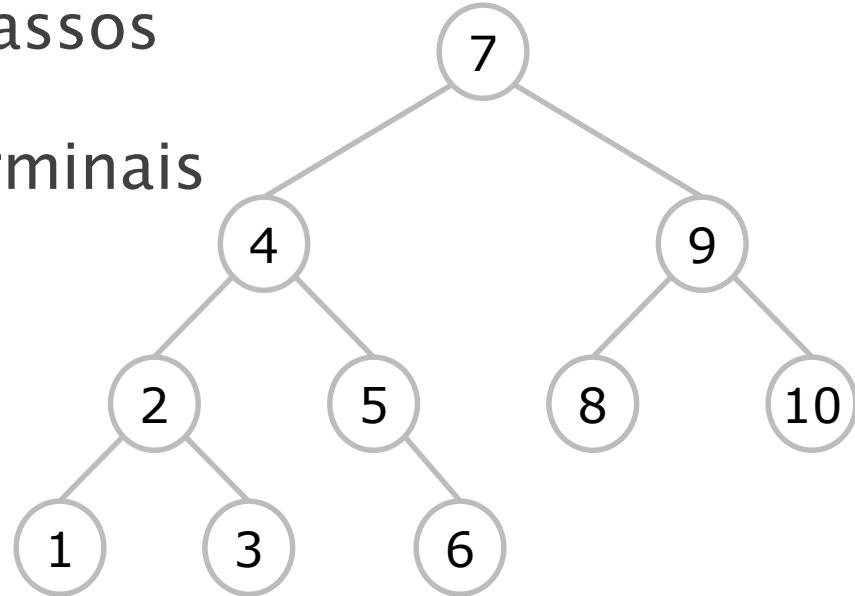
Árvore binária de busca balanceada

$$|he-hd| \leq 1$$

- he = altura da sub-árvore esquerda
- hd = altura da sub-árvore direita

qualquer nó pode ser alcançado a partir da raiz em $O(\log(n))$ passos

(quase) todos os nós não terminais têm dois filhos



Exercícios sobre árvores:

1. Percorrer a árvore em pré-ordem, pós-ordem e ordem simétrica (um dos percursos)
2. Criar a árvore binária e inserir nó na árvore (sempre completa ou cheia) a partir de números gerados aleatoriamente e a partir de um vetor previamente inicializado (neste caso para testes não precisa ser completa ou cheia)
3. Verificar se existe nó com uma dada chave na árvore

Percorso – pré-ordem

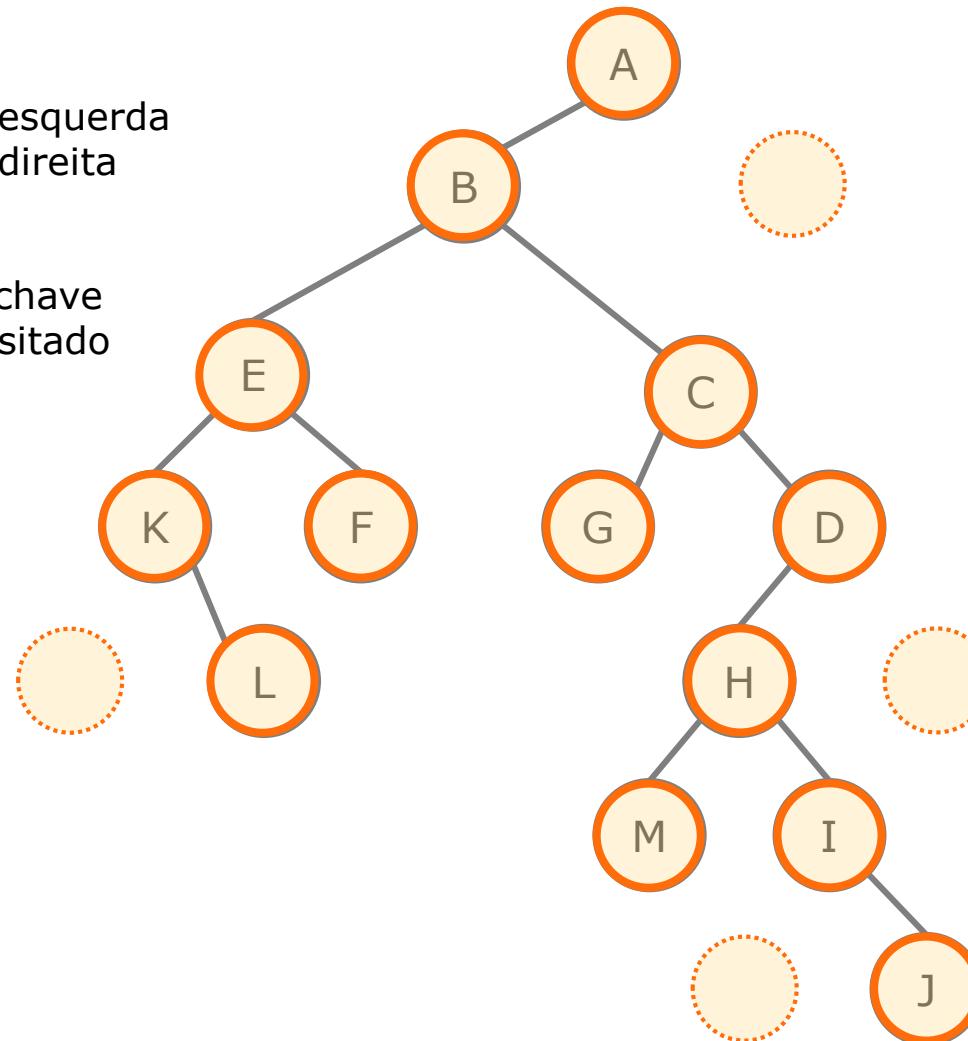
Visita raiz

Visita sub-árvore esquerda

Visita sub-árvore direita

Obs:

Visitar = exibir a chave
do nó visitado



A	B	E	K	*	L	F	C	G	D	H	M	I	*	J	*	*
---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---

Percurso – pós-ordem

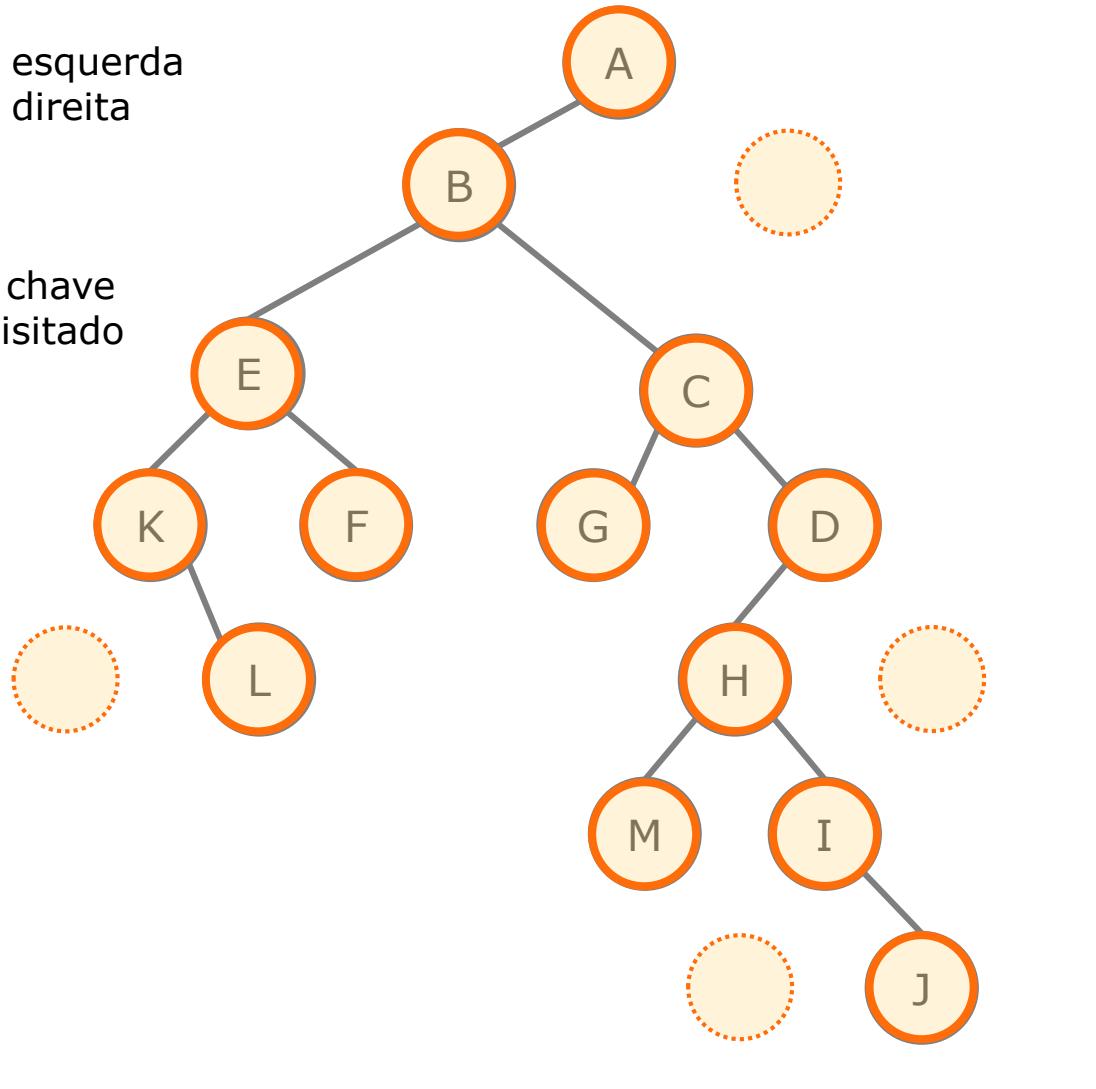
Visita sub-árvore esquerda

Visita sub-árvore direita

Visita raiz

Obs:

Visitar = exibir a chave
do nó visitado



*	L	K	F	E	G	M	*	J	I	H	*	D	C	B	*	A
---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---

Percurso – ordem simétrica

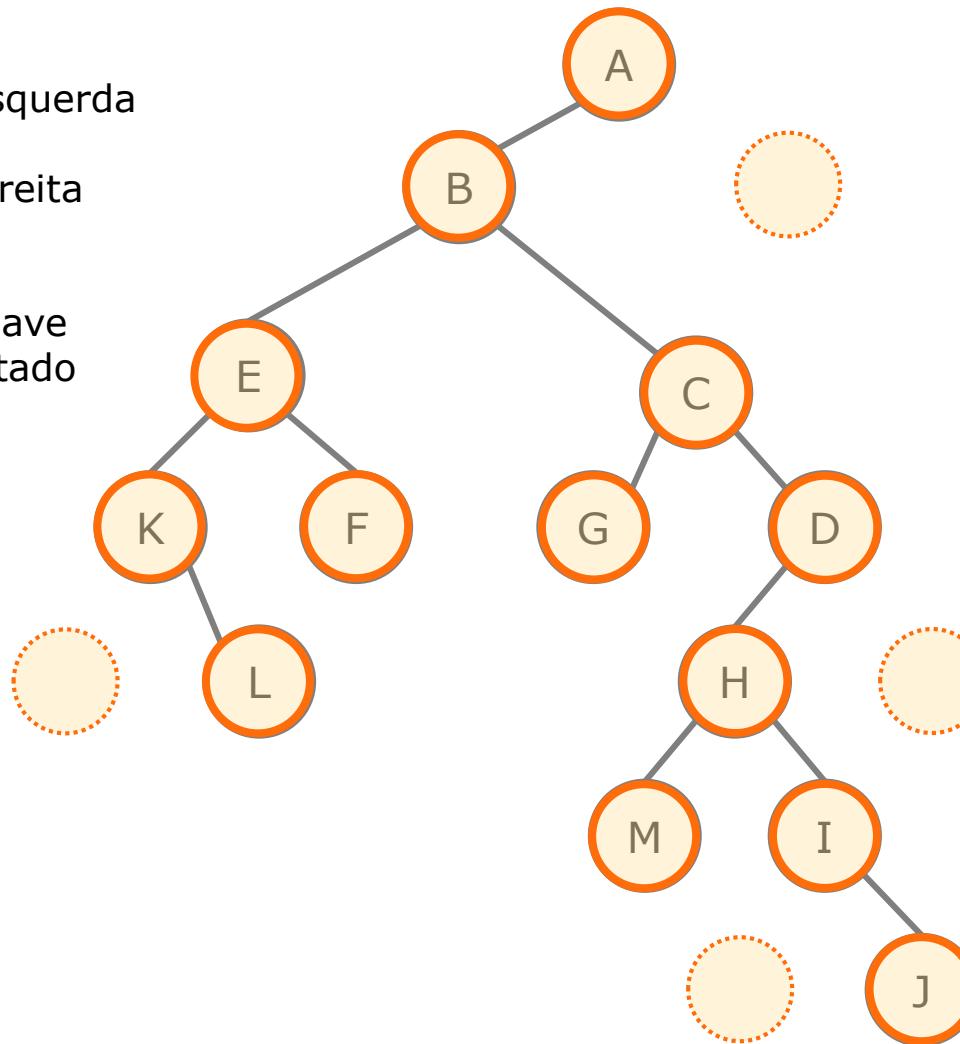
Visita sub-árvore esquerda

Visita raiz

Visita sub-árvore direita

Obs:

Visitar = exibir a chave
do nó visitado



Exercícios adicionais sobre árvores:

1. Excluir nó da árvore
2. Calcular a altura dos nós de uma árvore binária dada , armazenando o valor da altura, no nó.
3. Achar o maior e o menor elemento da árvore.
4. Trocar as sub-árvores esquerda e direita de todos os nós da árvore
5. Percorrer a árvore em nível (sugestão: usar uma fila)
6. Indicar o pai de cada nó, armazenando a informação no nó (ponteiro para o nó pai)
7. Calcular o número de nós da sub-árvore de cada nó, armazenando este valor no nó correspondente.

Estrutura de dados:

```
Struct nodo {  
    int chave;  
    struct nodo * esq;  
    struct nodo * dir;  
};  
Typedef struct nodo Nodo;
```

Variantes na estrutura a seu critério, em função do problema. Ex:

- Incluir campo ponteiro para o nó pai
- Incluir campo altura do nó
- Se a informação do nó for uma estrutura de dados (e não um valor de chave) alterar para um ponteiro para estrutura

```
void pre(Nodo * p) {  
    visita(p);  
    if (p->esq != NULL)  
        pre(p->esq);  
    if (p->dir != NULL)  
        pre(p->dir);  
}
```