

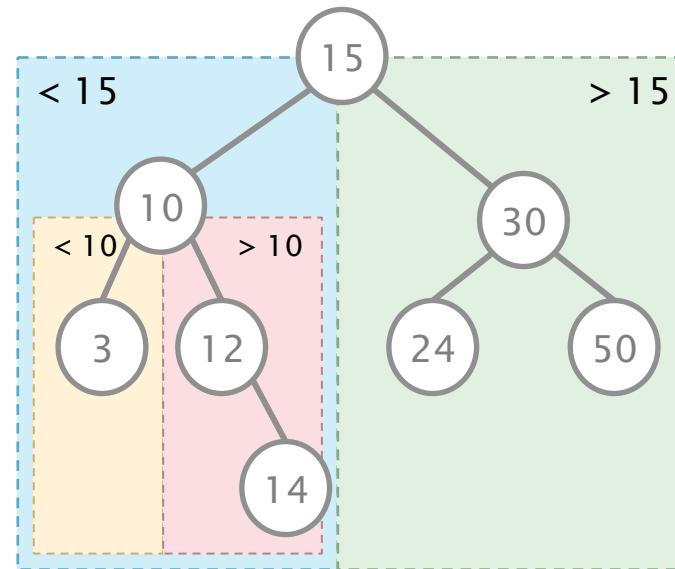


INF 1010

Estruturas de Dados Avançadas

Árvores Binárias de Busca

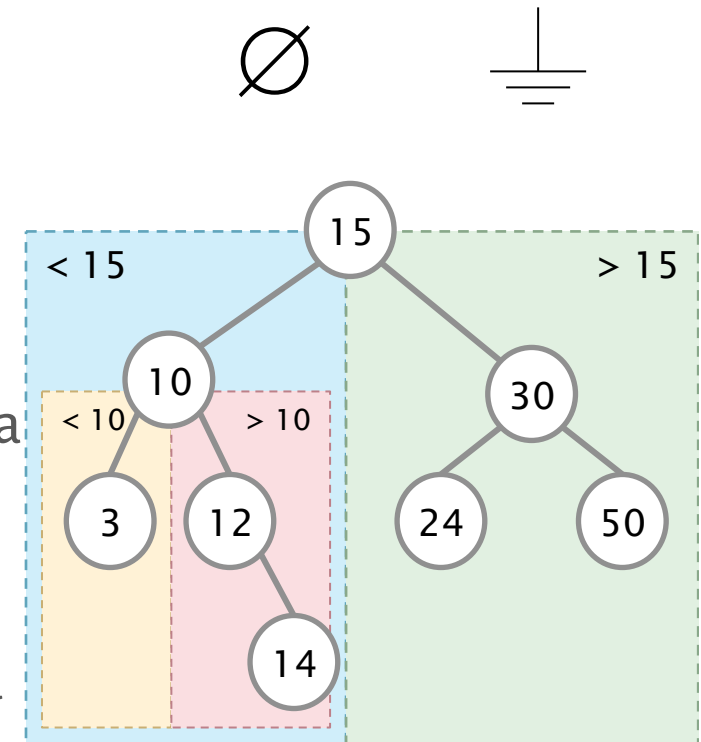
Árvores binárias de busca (ABB)



Definição de uma ABB

ABB é uma árvore binária vazia,
ou

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-árvores esquerda e direita são árvores binárias de busca



Uma implementação de ABB em C

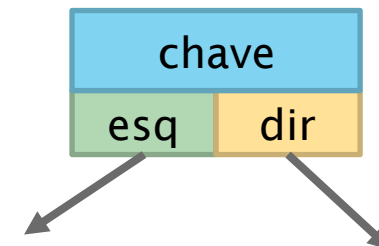
abb.h

```
typedef struct _abb Abb;  
  
Abb* abb_cria (void);  
Abb* abb_insere (Abb* raiz, int val);  
Abb* abb_busca (Abb* raiz, int val);  
. . .
```

```
Abb* abb_cria (void)  
{  
    return NULL;  
}
```

abb.c

```
#include "abb.h"  
  
struct _abb {  
    int chave;  
    Abb* esq;  
    Abb* dir;  
};
```



Outra implementação de ABB (com pai)

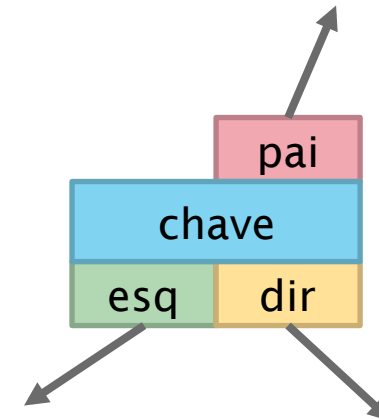
abb.h

```
typedef struct _abb Abb;  
  
Abb* abb_cria (void);  
Abb* abb_insere (Abb* raiz, int val);  
Abb* abb_busca (Abb* raiz, int val);  
. . .
```

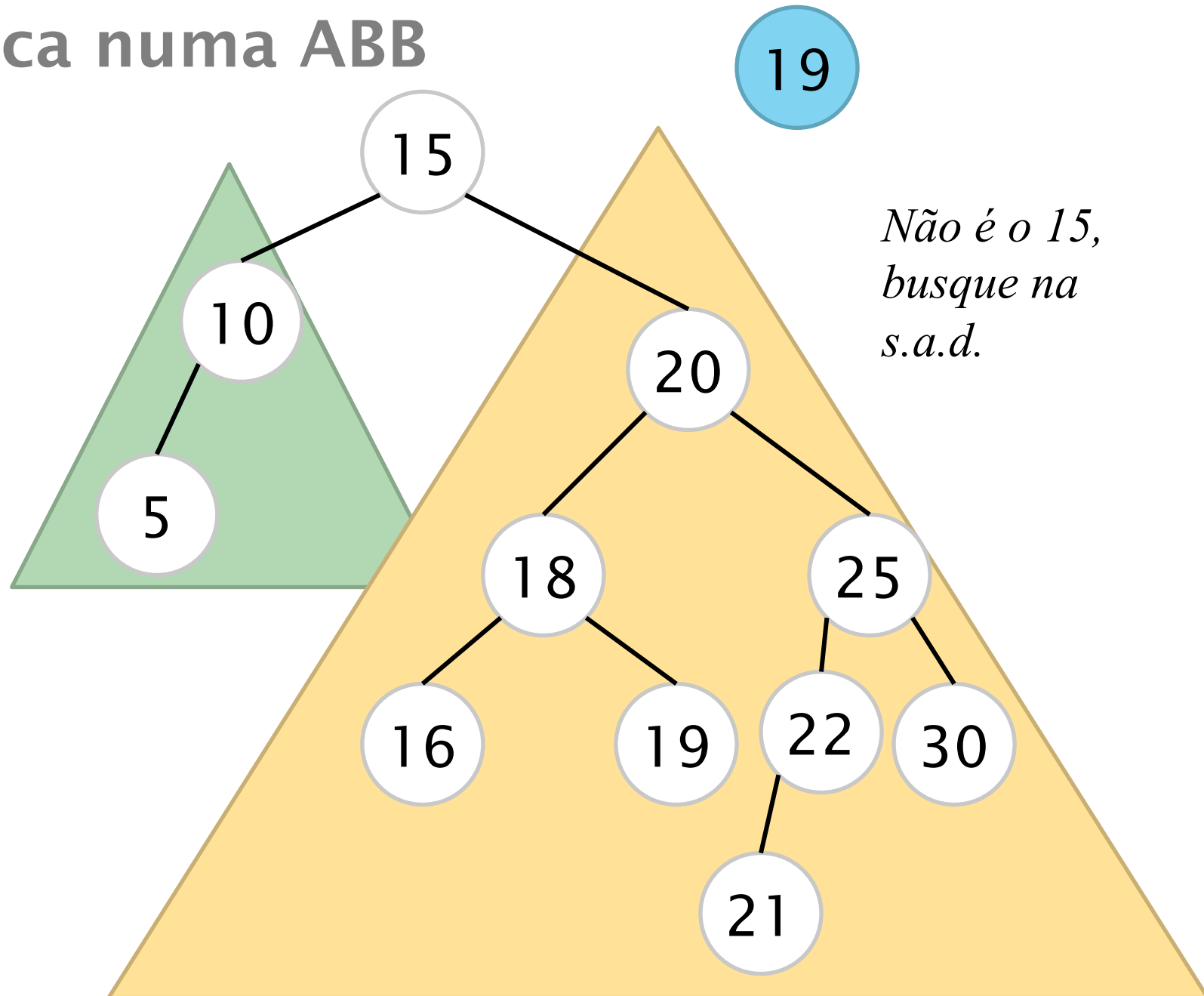
```
Abb* abb_cria (void)  
{  
    return NULL;  
}
```

abb.c

```
#include "abb.h"  
  
struct _abb {  
    int chave;  
    Abb* pai;  
    Abb* esq;  
    Abb* dir;  
};
```



Busca numa ABB



Algoritmo para busca em ABBs

```
Abb* abb_busca (Abb* raiz, int val);
```

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**

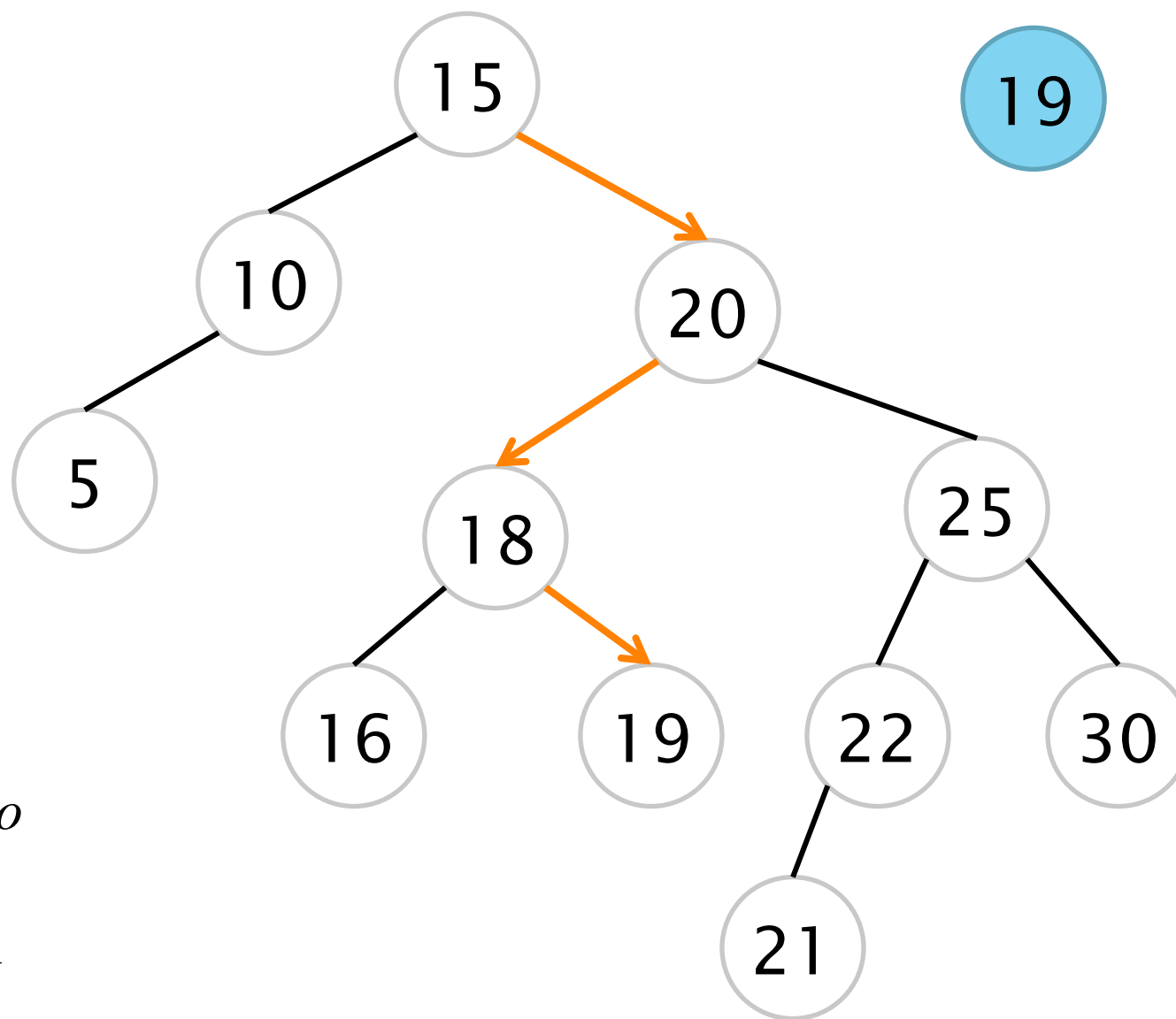
CC = caso contrário

Busca numa abb

```
Abb* abb_busca (Abb* r, int v)
{
    if (r == NULL)
        return NULL;
    else if (v < r->chave)
        return abb_busca (r->esq, v);
    else if (v > r->chave)
        return abb_busca (r->dir, v);
    else return r;
}
```

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**

Busca iterativa numa ABB



*Busca como
em lista,
com o prox
variando*

Busca iterativa numa ABB

```
Abb* abb_busca_iterativa(Abb* r, int val);
```

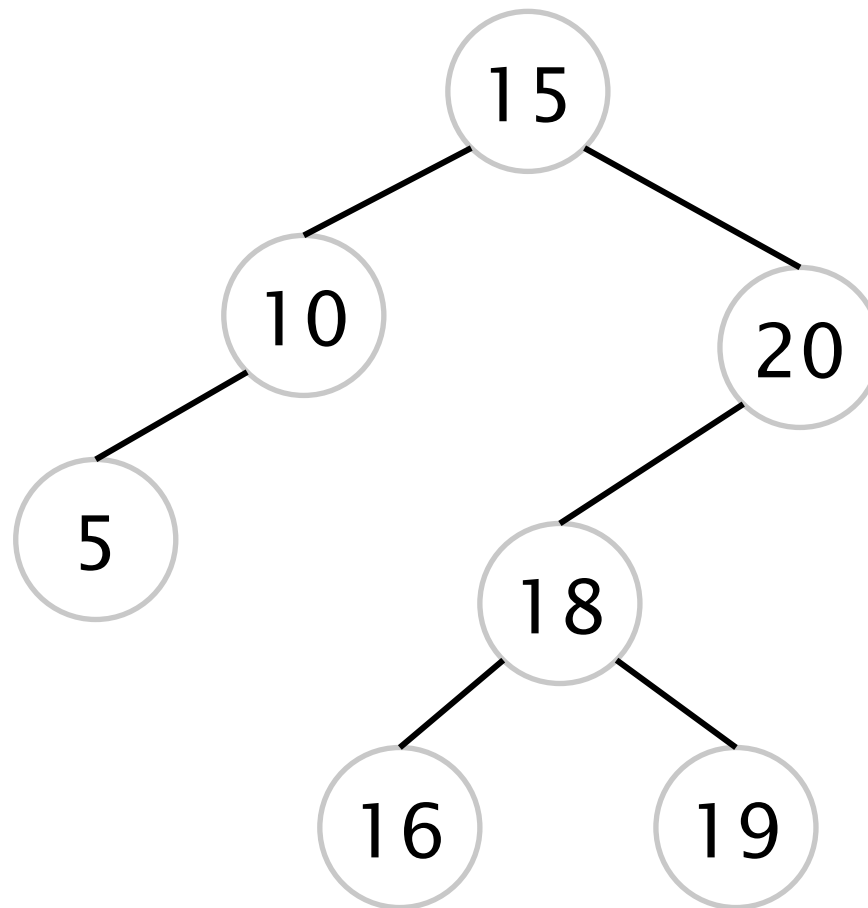
1. Comece a busca pelo nó raiz
2. Enquanto o nó não for **NULL** ou não contiver a chave procurada:
 1. Se a chave do nó for **maior** que a chave procurada vá para o filho à **direita**
 2. Se a chave do nó for **menor** que a chave procurada vá para o filho à **esquerda**
3. Retorne o nó corrente

Busca iterativa numa ABB

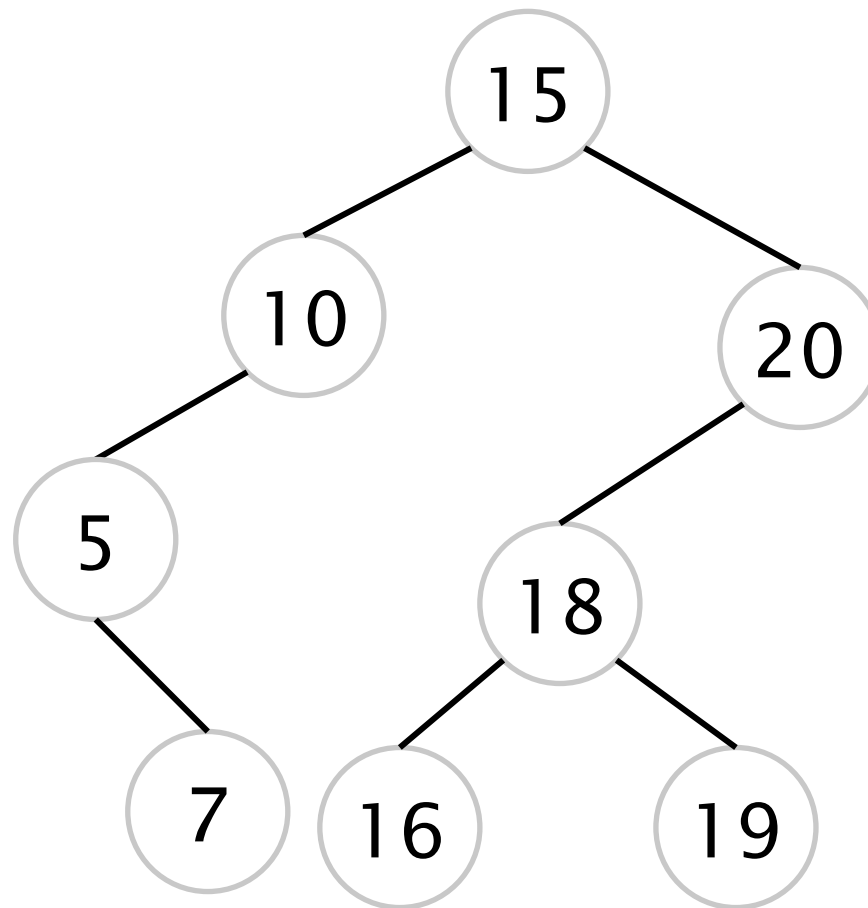
```
Abb* abb_busca_iterativa(Abb* r, int val)
{
    while (r!=NULL && r->chave != val)
    {
        if (val < r->chave)
            r=r->esq;
        else
            r=r->dir;
    }
    return r;
}
```

1. Comece a busca pelo nó raiz
2. Enquanto o nó não for **NULL** ou não contiver a chave procurada:
 1. Se a chave do nó for **maior** que a chave procurada vá para o filho à **direita**
 2. Se a chave do nó for **menor** que a chave procurada vá para o filho à **esquerda**
3. Retorne o nó corrente

Menor nó numa ABB



Menor nó numa ABB



Menor nó numa ABB

```
Abb* abb_min (Abb* r)
```

É o nó mais à esquerda da árvore

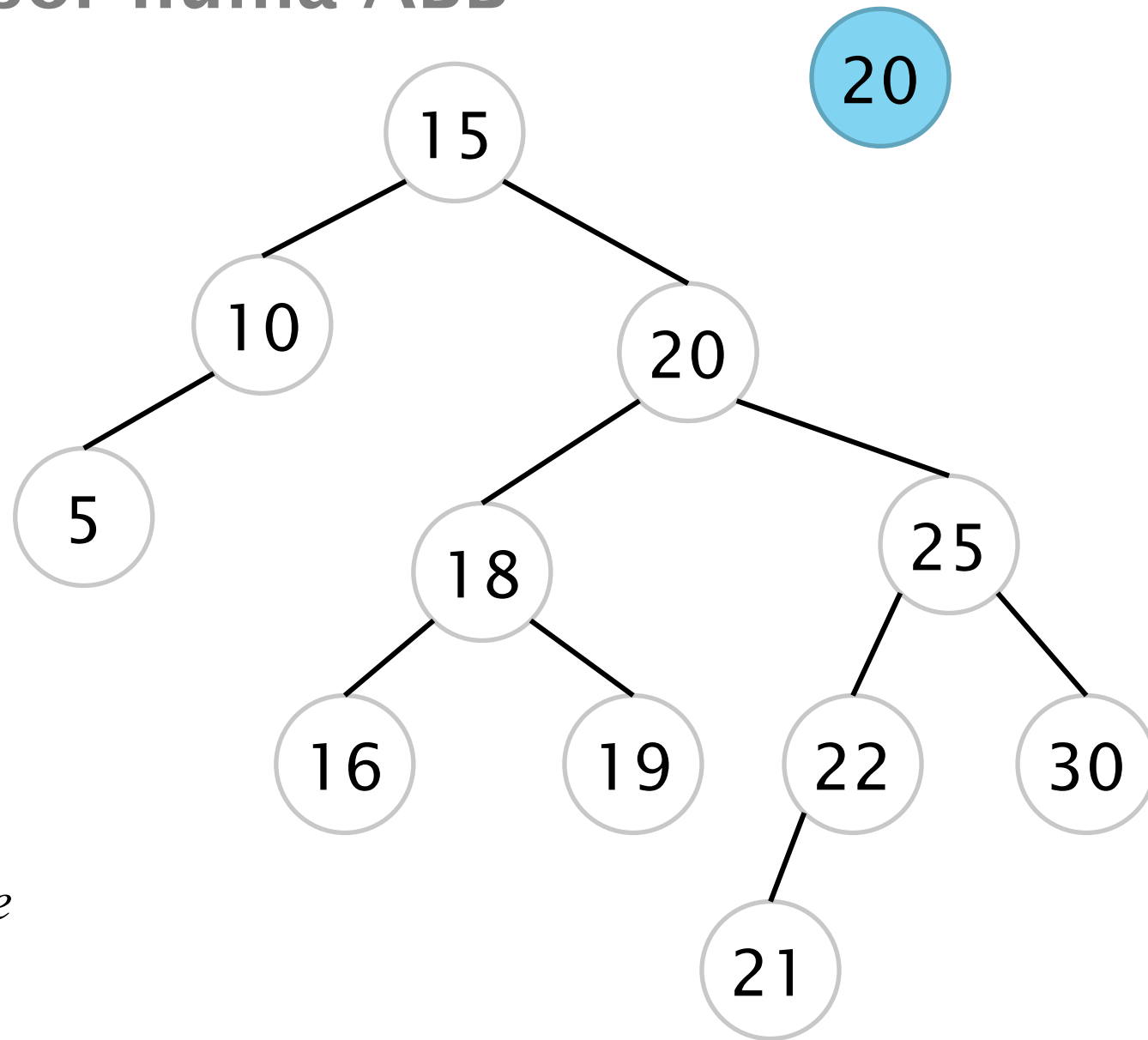
1. Começando pelo nó raiz
2. Se a árvore for vazia retorne NULL
3. Caso contrário, caminhe sempre à esquerda até enquanto o filho esquerdo não for NULL

Menor nó numa ABB

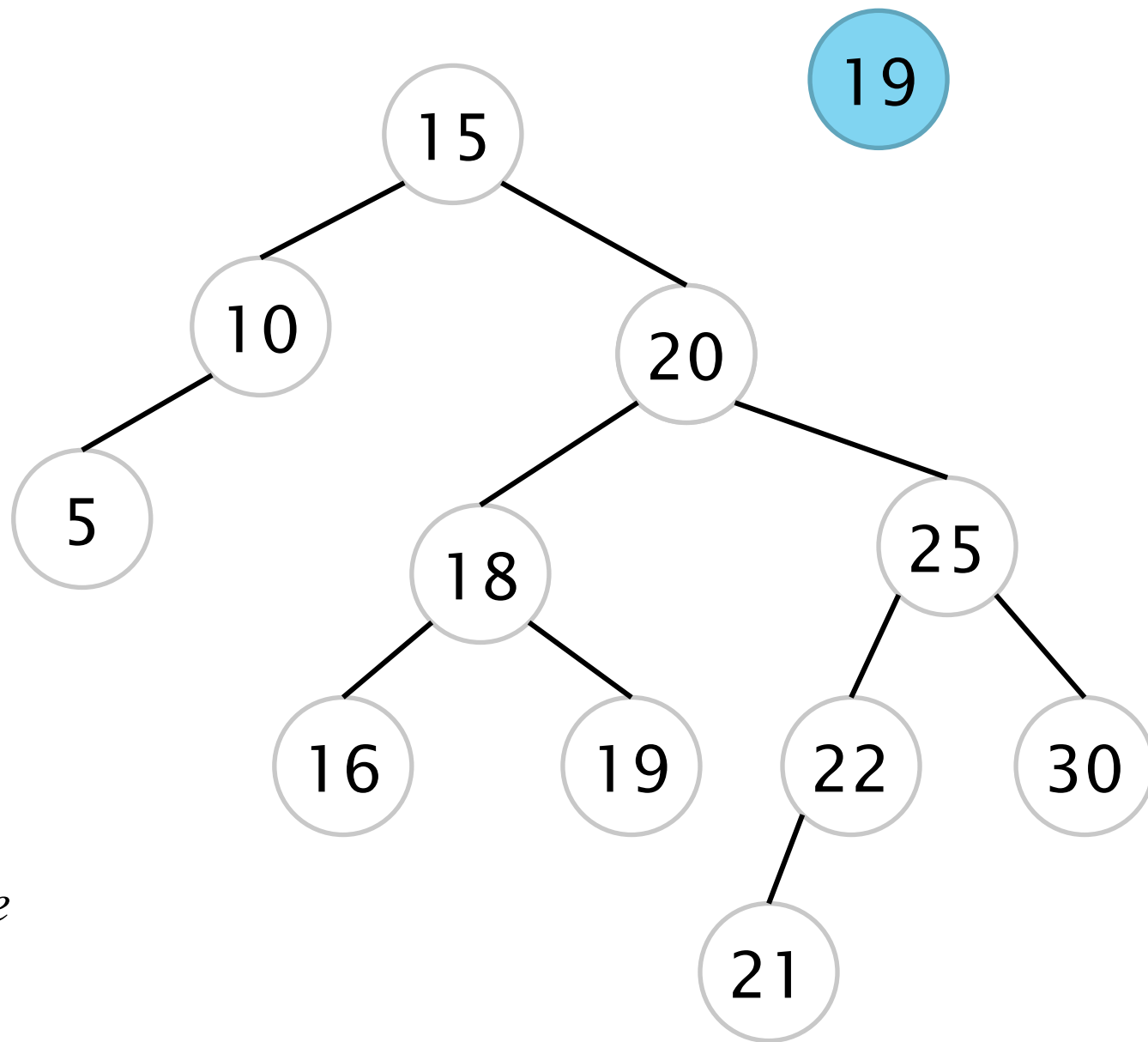
```
Abb* abb_min (Abb* r) {  
    if (r==NULL) return NULL;  
    while(r->esq!=NULL)  
        r=r->esq;  
    return r;  
}
```

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

Sucessor numa ABB



*Quem é o
sucessor de
20?*



*Quem é o
sucessor de
19?*

Sucessor numa ABB (prox)

```
Abb* abb_prox (Abb* r);
```

1. Se o nó for NULL return NULL
2. CC se o nó tiver uma sub-árvore à direita, retorne o mínimo dela
3. CC suba na árvore procurando o primeiro ancestral que seja maior que seu filho. Ou seja, o nó corrente vai estar na sua sub-árvore à esquerda. Se nessa busca você chegar na raiz (ancestral NULL), retorne NULL.

CC = caso contrário

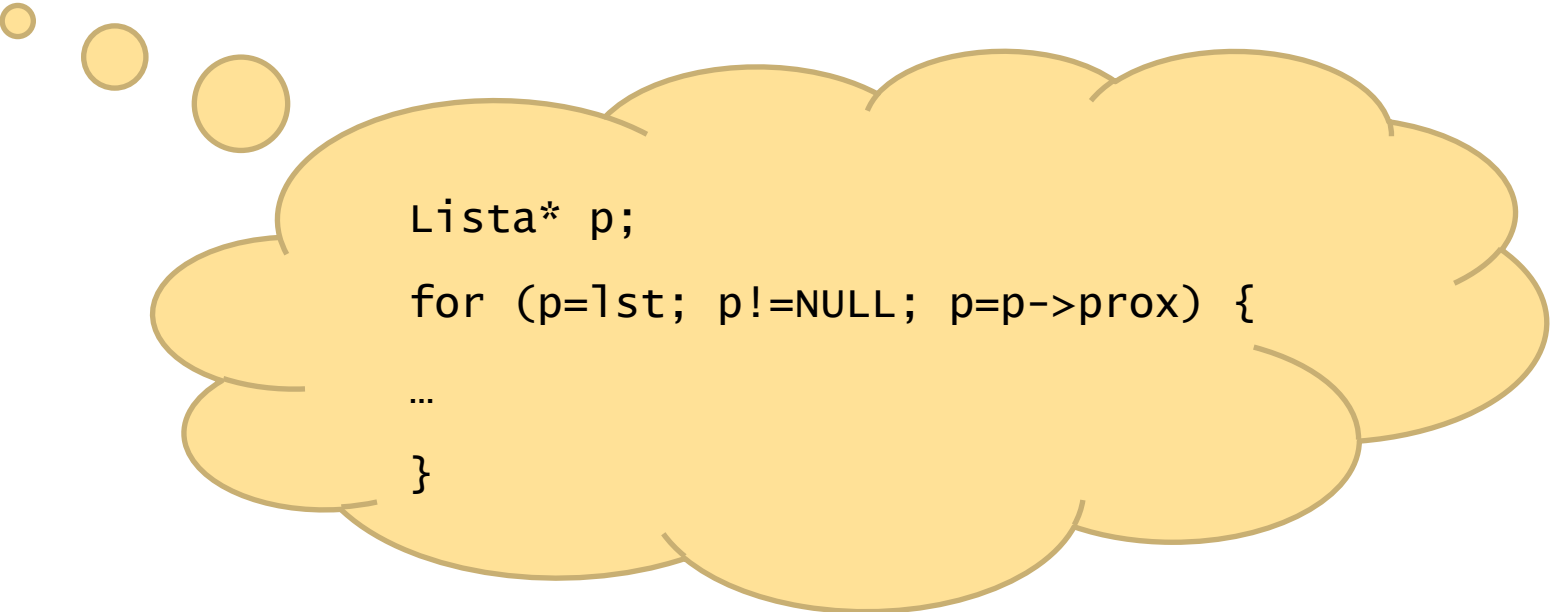
Sucessor numa ABB (prox)

```
Abb* abb_prox (Abb* r) {  
    if (r==NULL) return NULL;  
    else if (r->dir!=NULL)  
        /* retorna o menor da sad */  
        return abb_min(r->dir);  
    else {  
        /* retorna o ancestral mais próximo  
         * que seja maior que seu filho */  
        Abb* p = r->pai;  
        while ( p!=NULL && r==p->dir ) {  
            r = p;  
            p = p->pai;  
        }  
        return p;  
    }  
}
```

1. Se o nó for NULL
return NULL
2. CC se o nó tiver uma
sub-árvore à direita,
retorne o mínimo dela
3. CC suba na árvore
procurando o primeiro
ancestral que seja
maior que seu filho.
Ou seja, o nó corrente
vai estar na sua sub-
árvore à esquerda. Se
nessa busca você
chegar na raiz
(ancestral NULL),
retorne NULL.

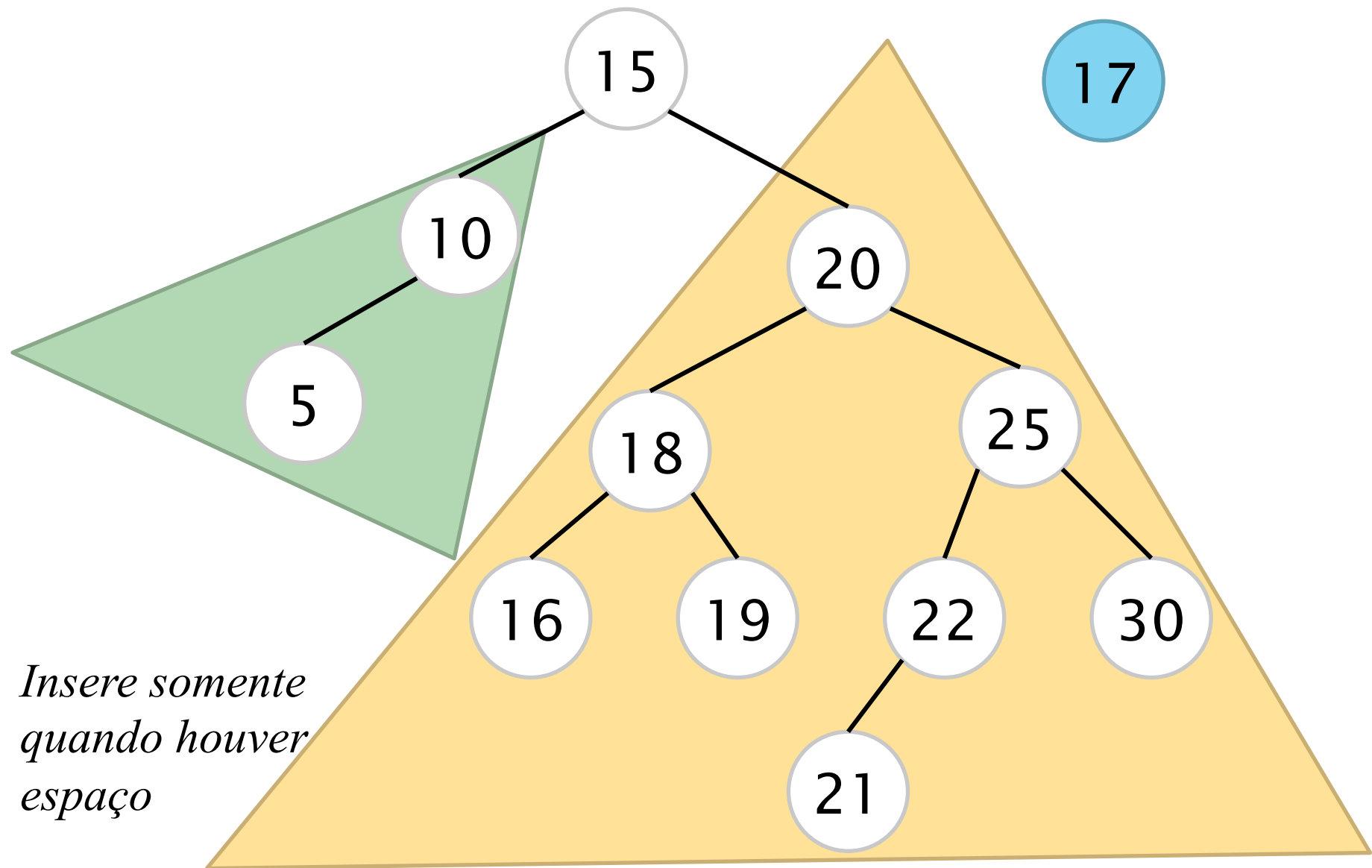
Percorrendo em ordem os nós de uma ABB

```
Abb* p;  
for (p=abb_min(arv); p!=NULL; p=abb_prox(p)) {  
...  
}
```

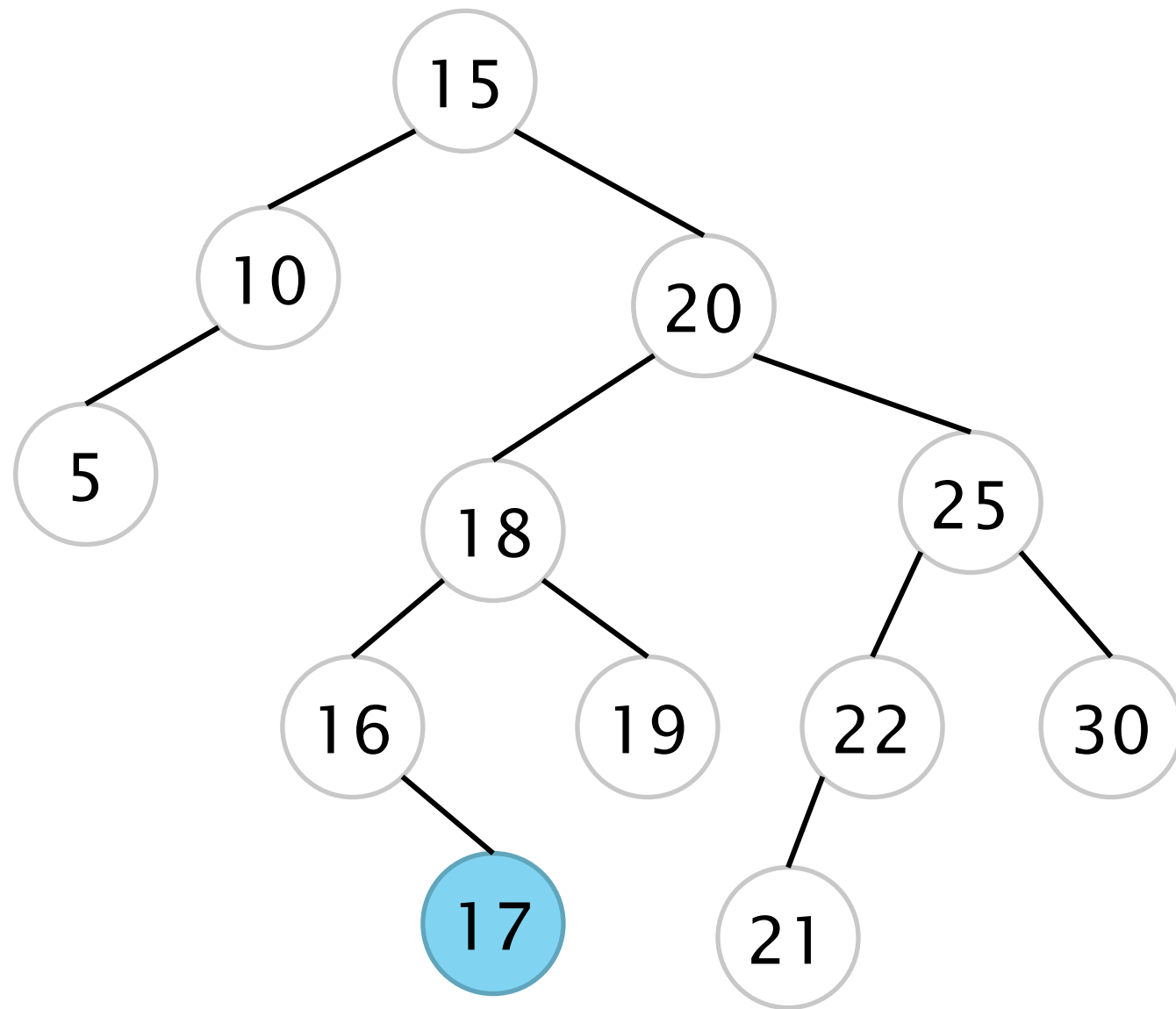


```
Lista* p;  
for (p=lst; p!=NULL; p=p->prox) {  
...  
}
```

Inserção numa ABB



Inserção numa ABB



Inserção numa abb

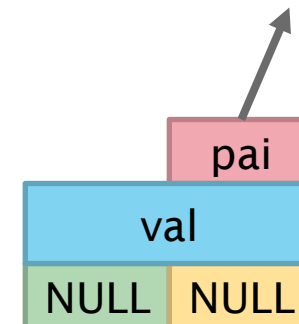
```
Abb* abb_insere (Abb* r, int val);
```

1. Começando pelo nó raiz
2. Se o nó for NULL crie um nó com a chave dada e retorne o endereço dele.
3. CC se a chave dada for **maior** que a chave corrente:
 1. o nó tiver uma sub-árvore à **direita**, insira nela a chave
 2. CC crie um nó com a chave dada e este será o filho à direita do nó corrente
4. CC se a chave dada for **menor** que a chave corrente:
 1. o nó tiver uma sub-árvore à **esquerda**, insira nela a chave
 2. CC crie um nó com a chave dada e este será o filho à esquerda do nó corrente
5. CC se a chave for **igual**, troque a informação associada à chave (na árvore de inteiros não faça nada)

Inserção numa abb

Primeiro faça uma função auxiliar para criar um nó

```
static Abb* cria_filho (Abb* pai, int val) {  
    Abb* no = (Abb*) malloc(sizeof(Abb));  
    no->chave = val;  
    no->pai=pai;  
    no->esq = no->dir = NULL;  
    return no;  
}
```



Inserção recursiva numa abb

```
Abb* abb_insere (Abb* r, int val){
    if (r==NULL)
        return cria_filho(r,val);
    else if (val < r->chave) {
        if (r->esq == NULL)
            r->esq = cria_filho(r,val);
        else
            r->esq = abb_insere(r->esq,val);

    }
    else if (val > r->chave) {
        if (r->dir == NULL)
            r->dir = cria_filho(r,val);
        else
            r->dir = abb_insere(r->dir,val);
    }
    return r;
}
```

Inserção iterativa numa abb

```
Abb* abb_insere_iterativa (Abb* r, int val);
```

1. Se a árvore for vazia crie um nó e retorne
2. CC 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**
4. Crie o nó com a chave dada e o coloque como filho do pai

Inserção iterativa numa abb

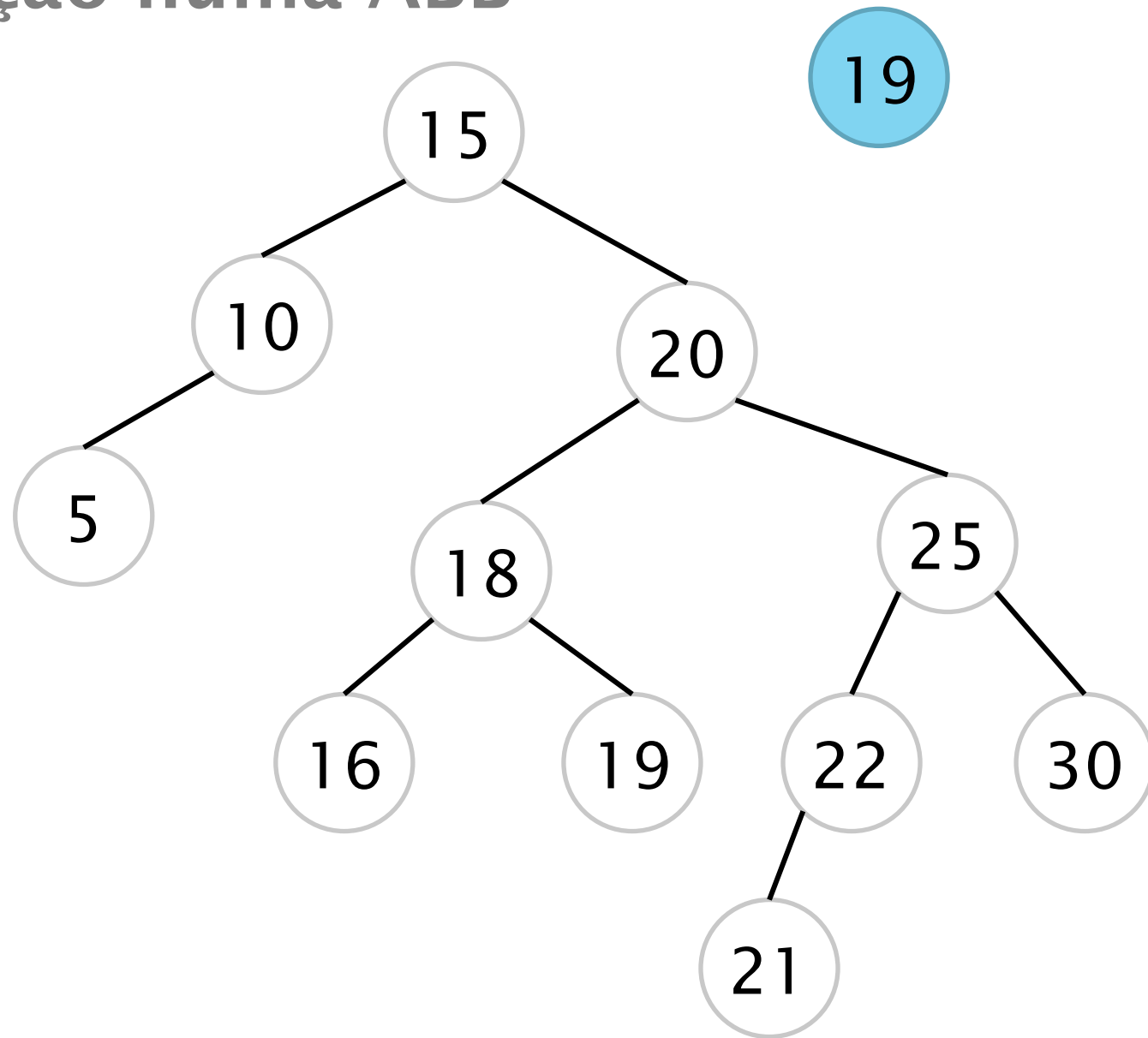
```
Abb* abb_insere_iterativa (Abb* r, int val){
    Abb* z = cria_filho(NULL, val);
    if (r == NULL) return z; /* a arvore era vazia */
    else {
        Abb* pai = NULL;
        Abb* x = r;
        while (x!=NULL) {
            pai=x;
            x = (val < x->chave)? x->esq : x->dir;
        }
        z->pai = pai;
        if (val < pai->chave)
            pai->esq = z;
        else
            pai->dir = z;
        return r;
    }
}
```

Remoção de um nó de uma abb

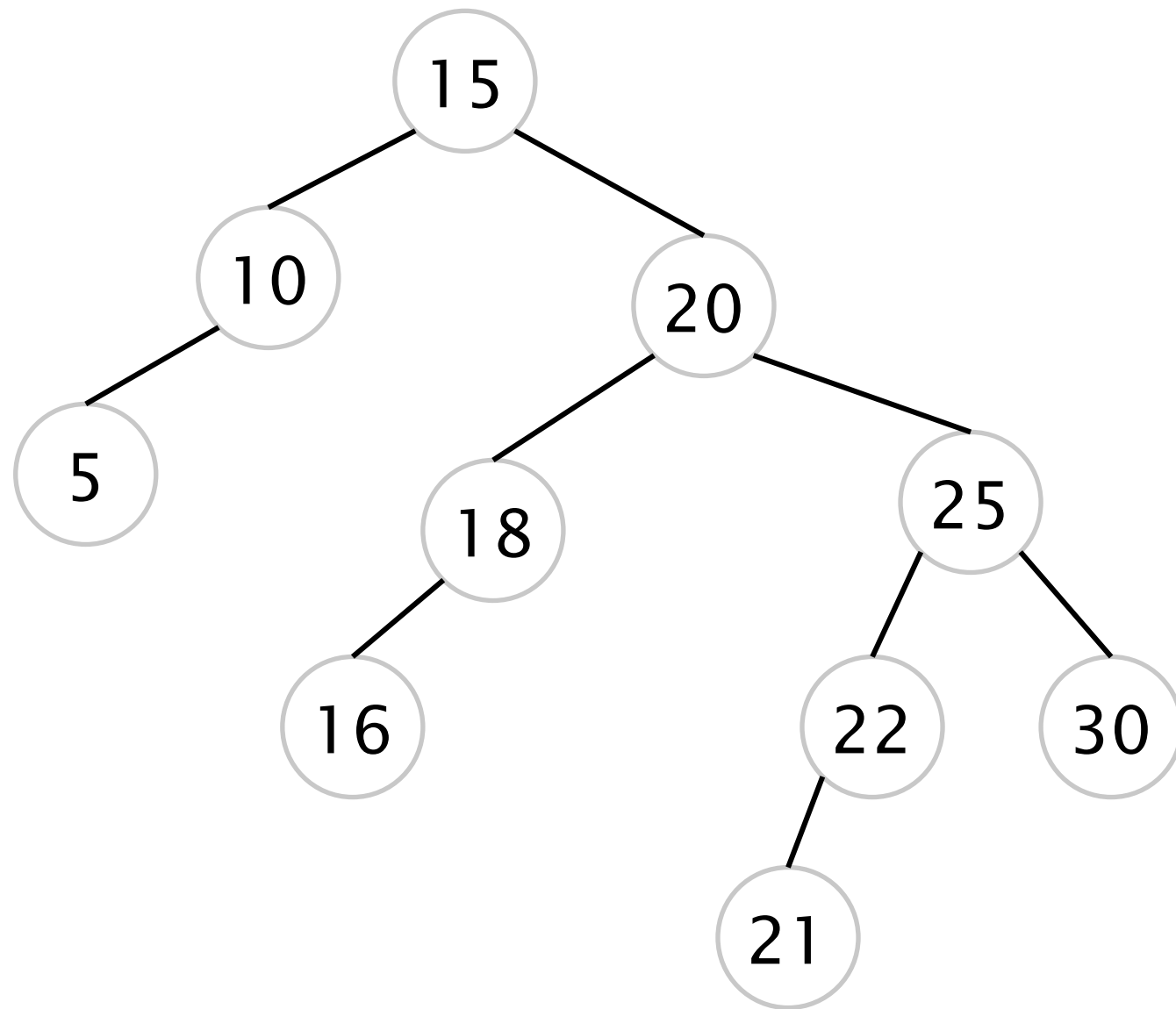
Três casos:

1. nó folha
2. nó possui uma sub-árvore
3. nó possui duas sub-árvores

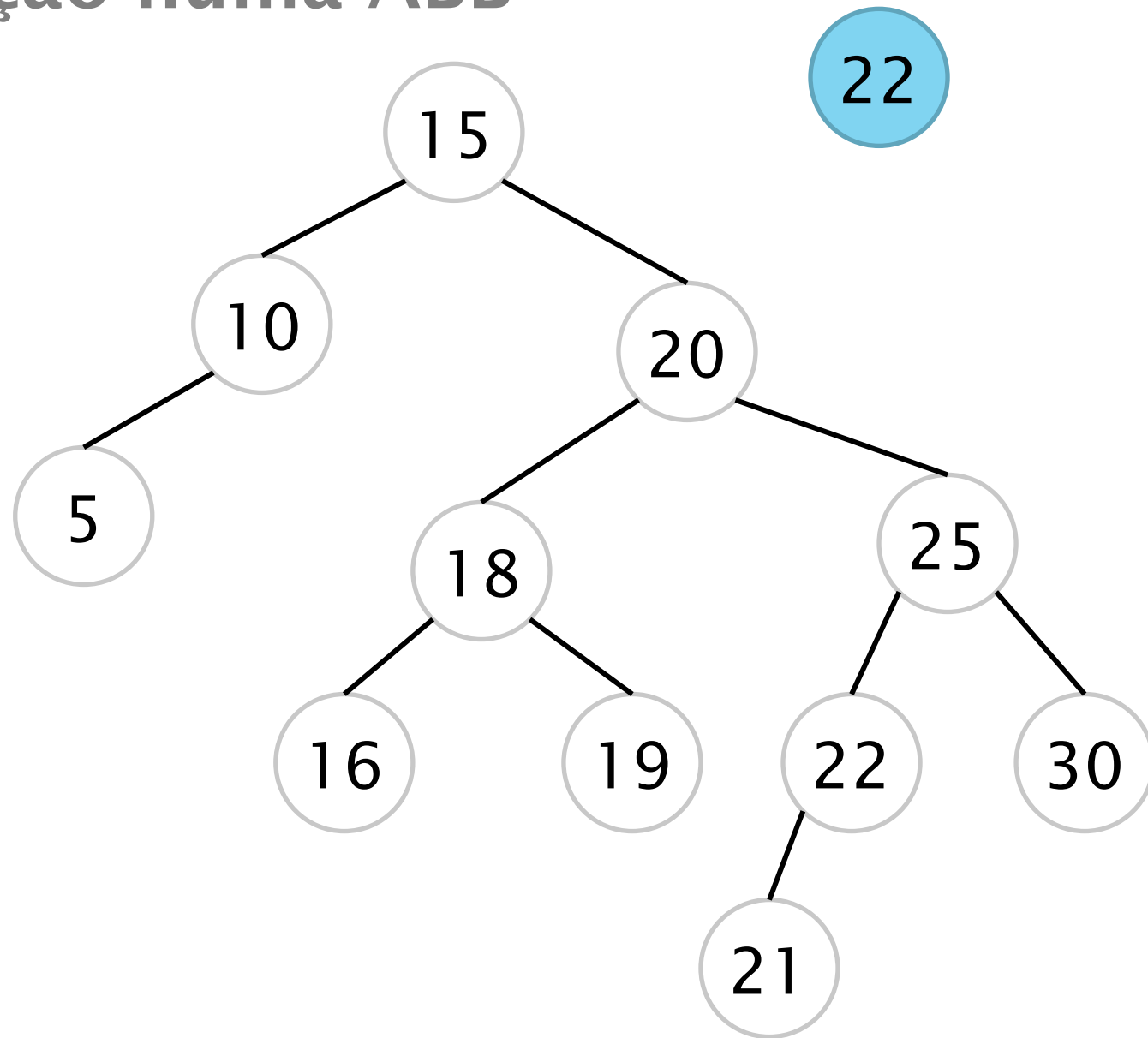
Remoção numa ABB



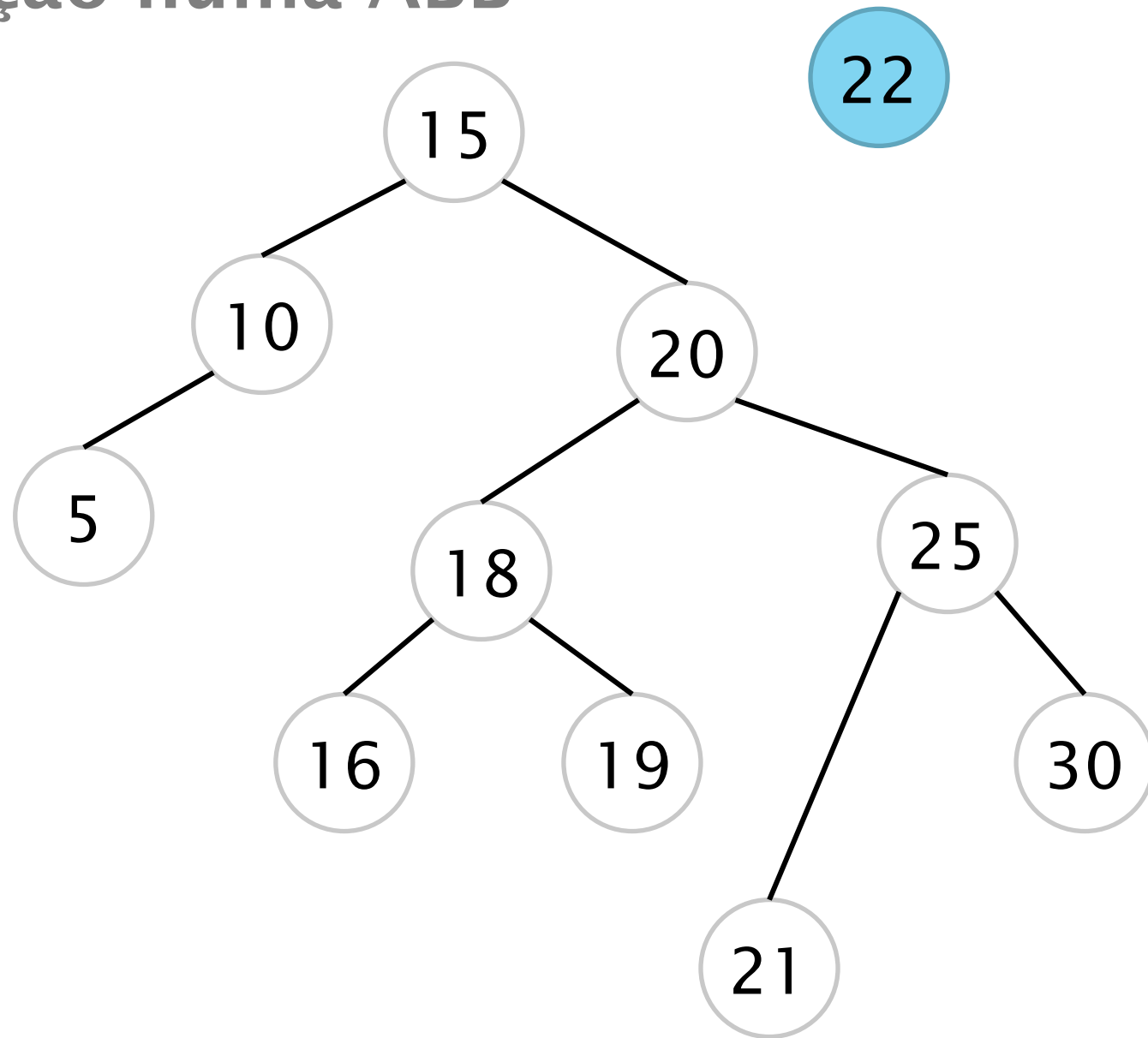
Remoção numa ABB



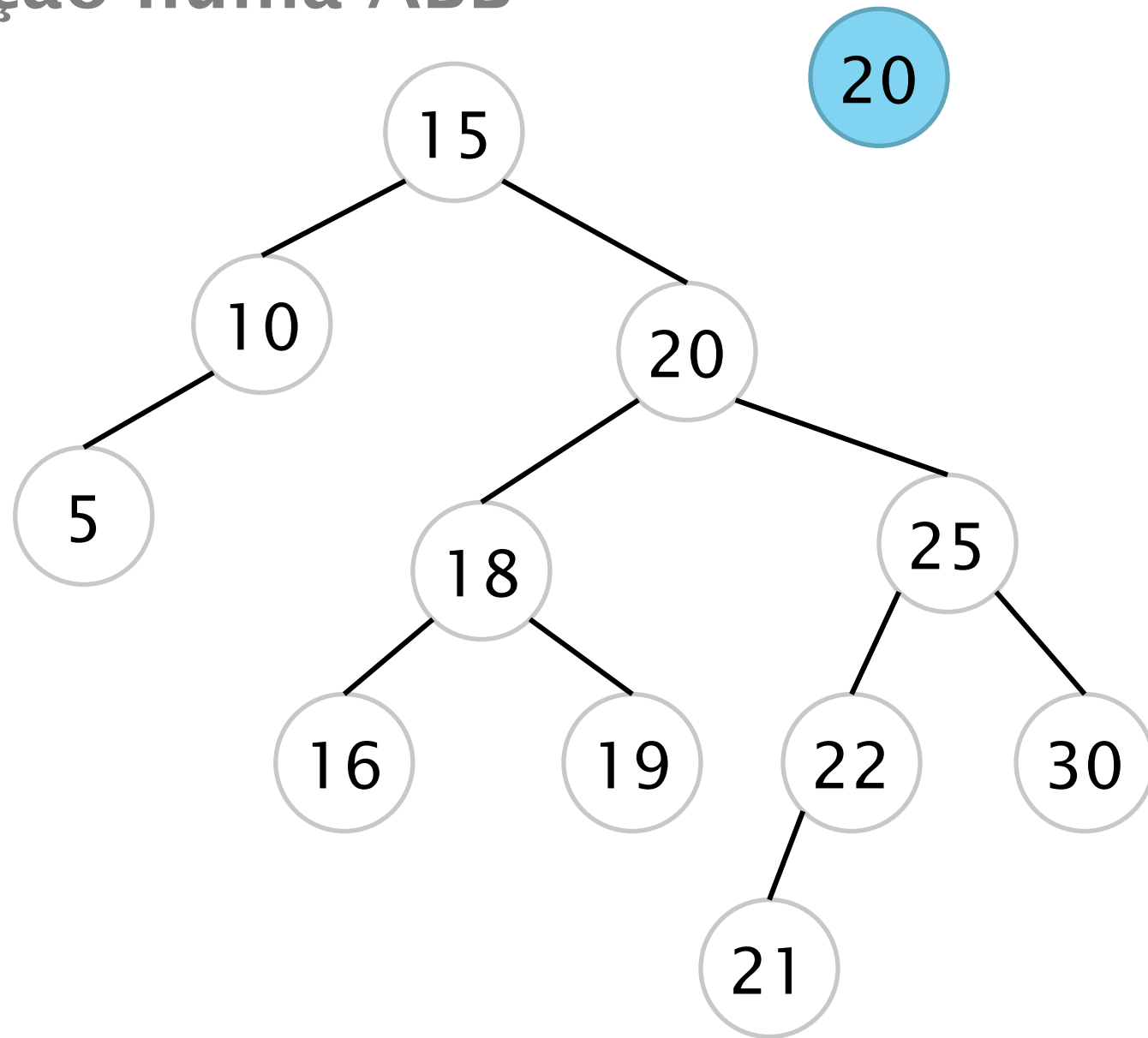
Remoção numa ABB



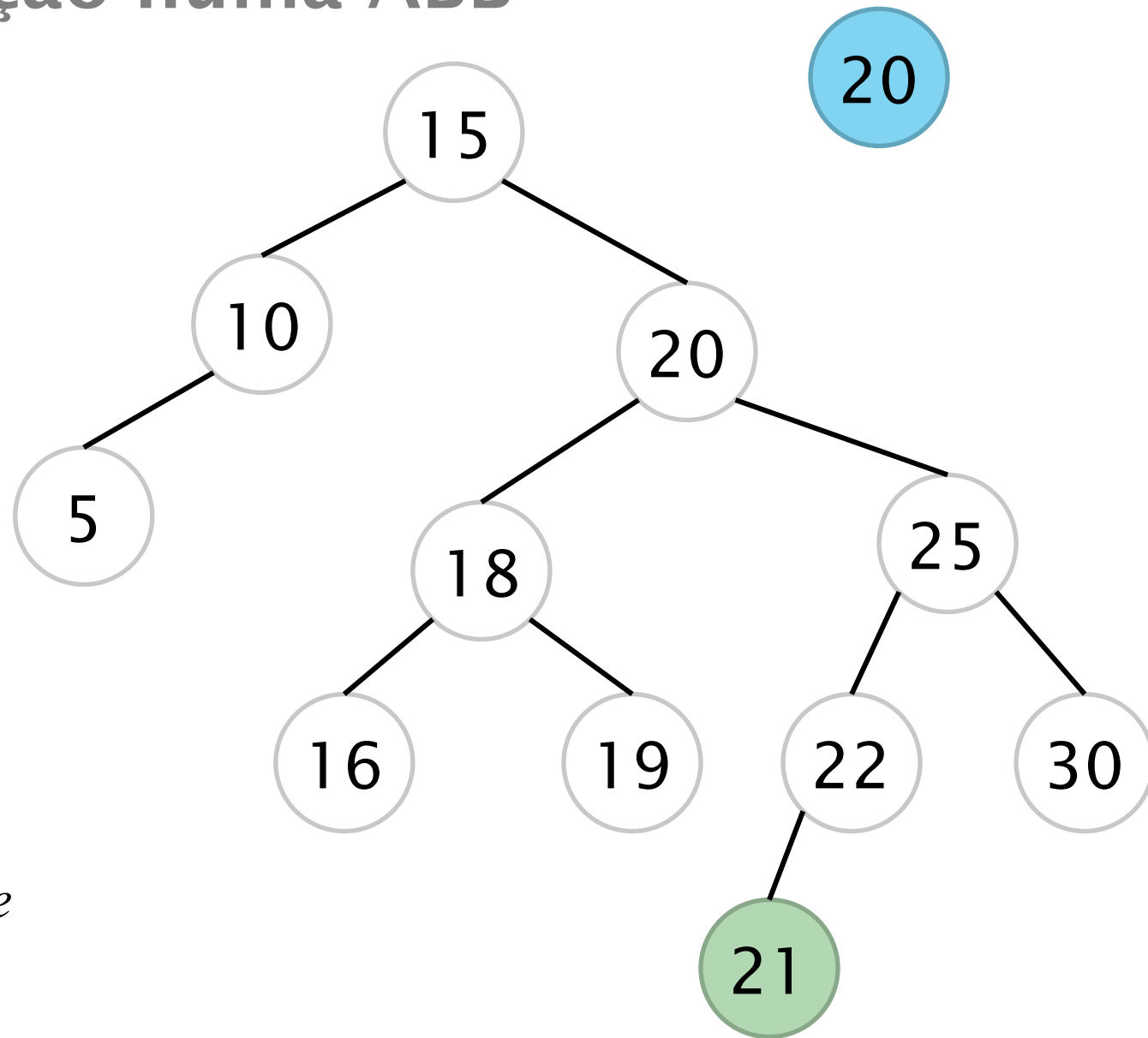
Remoção numa ABB



Remoção numa ABB

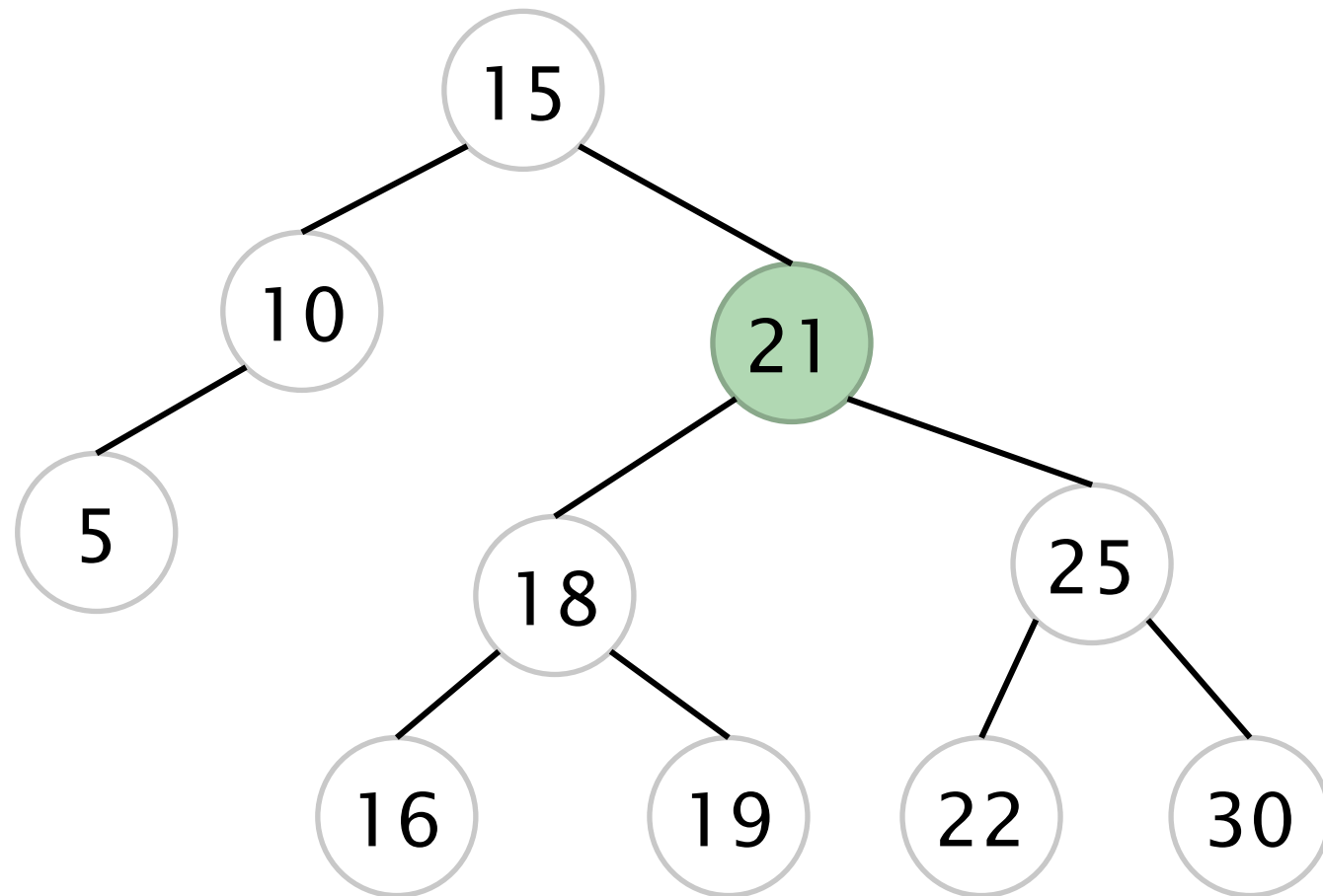


Remoção numa ABB

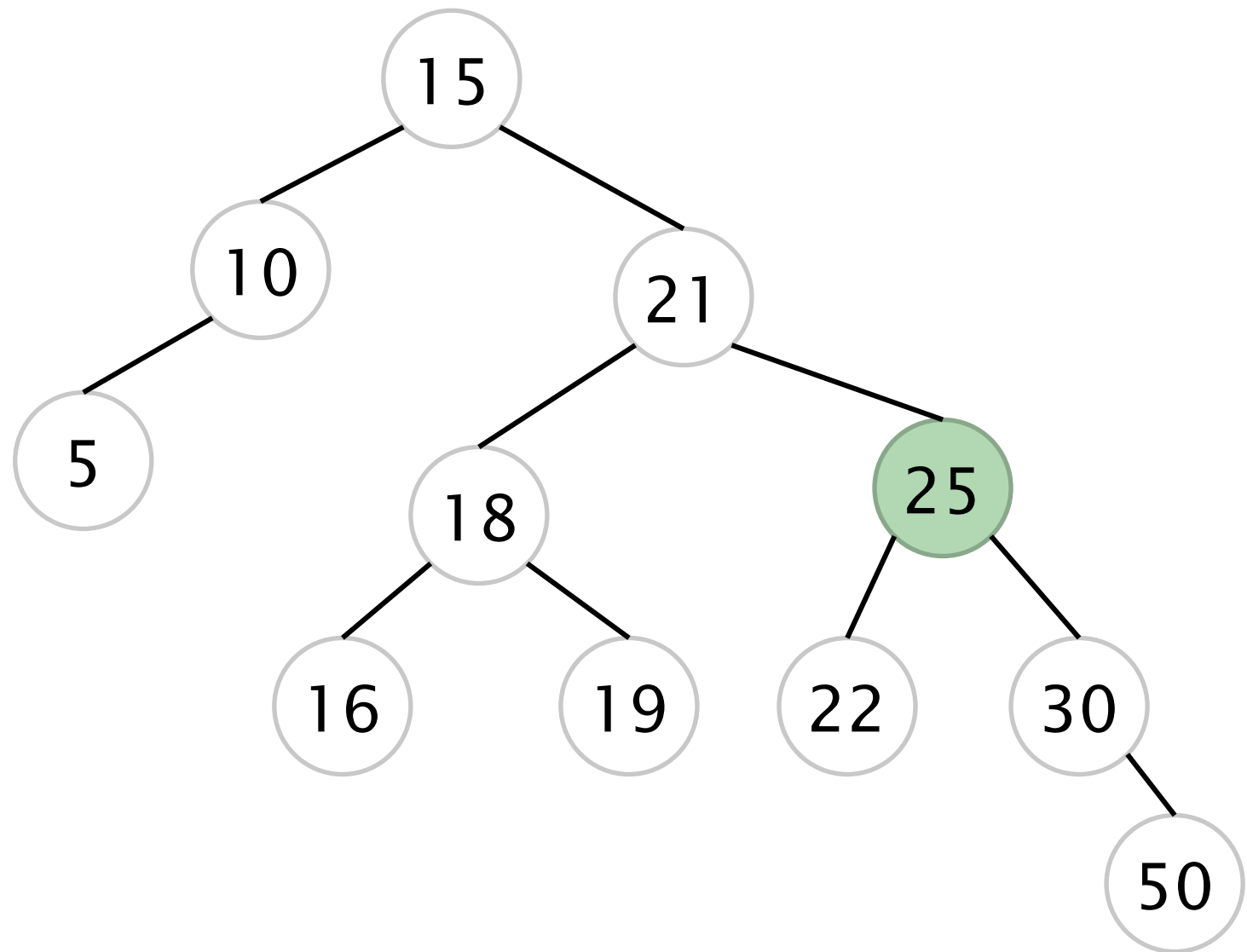


*Quem é o
sucessor de
20?*

Remoção numa ABB



Remoção numa ABB



Remoção de um nó de uma abb

Três casos:

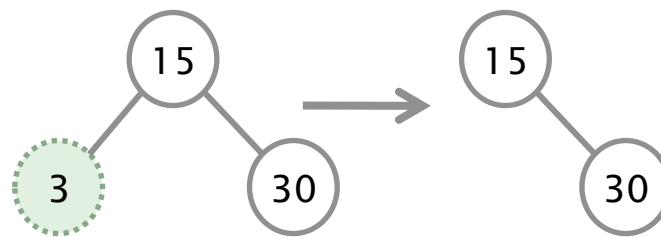


1 nó folha

simplesmente elimina o nó

2. nó possui uma sub-árvore

3. nó possui duas sub-árvores



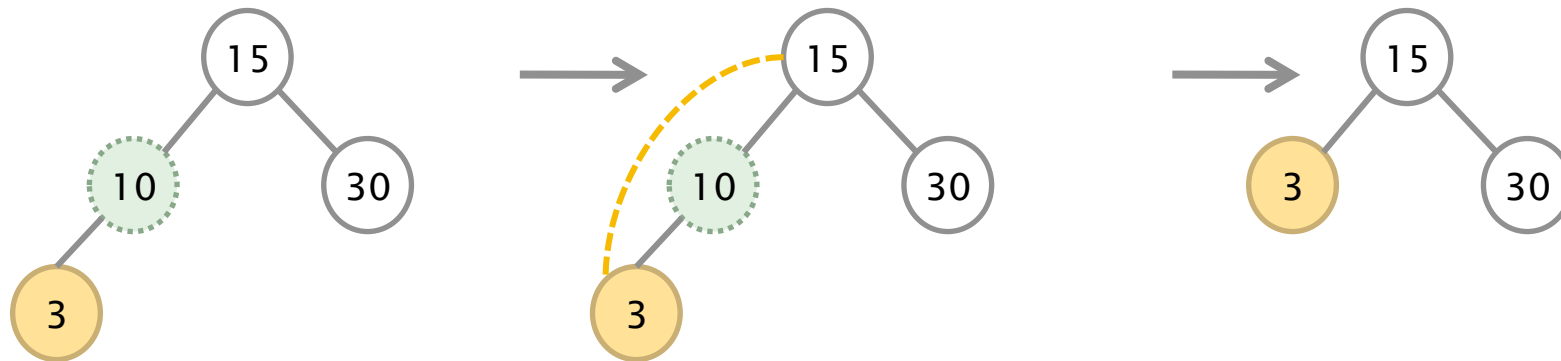
Remoção de um nó de uma abb

Três casos:

1. nó folha

2. nó possui uma sub-árvore [dois subcasos: sae; sad]
promove a sub-árvore

3. nó possui duas sub-árvores



Remoção de um nó de uma abb

Três casos:

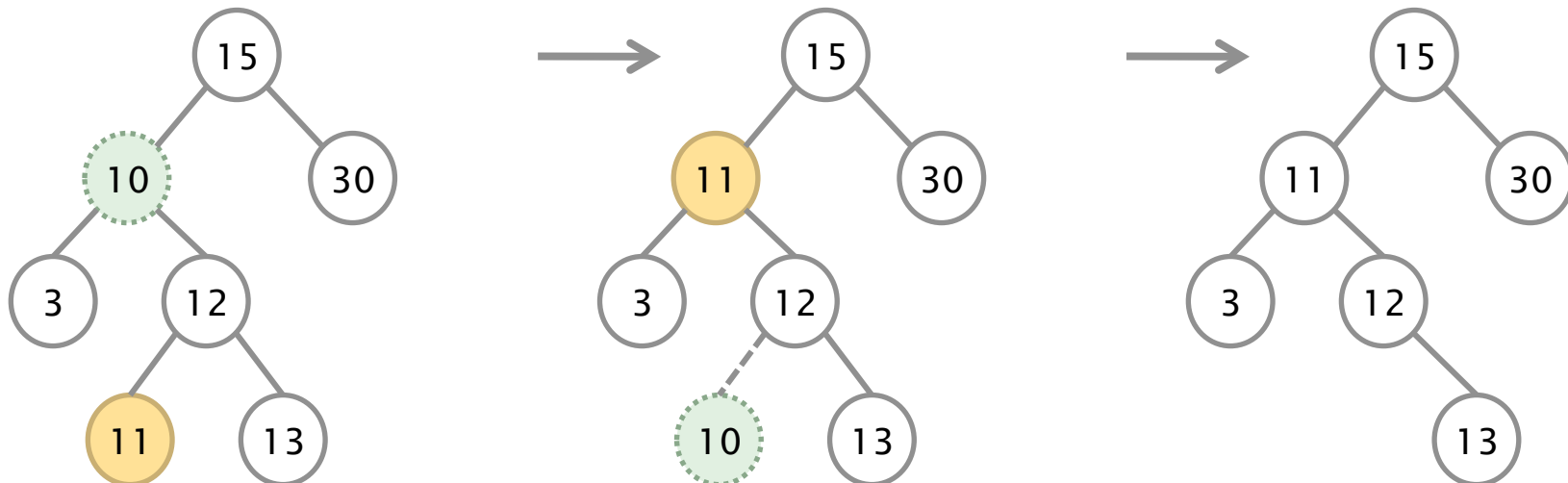
1. nó folha

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. CC 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 (obs)

1. O sucessor é sempre o nó de menor chave da sub-arvore à direita
2. Ao retirar um nó temos que atualizar os campos de seu pai e de seu filho para que eles se referenciem

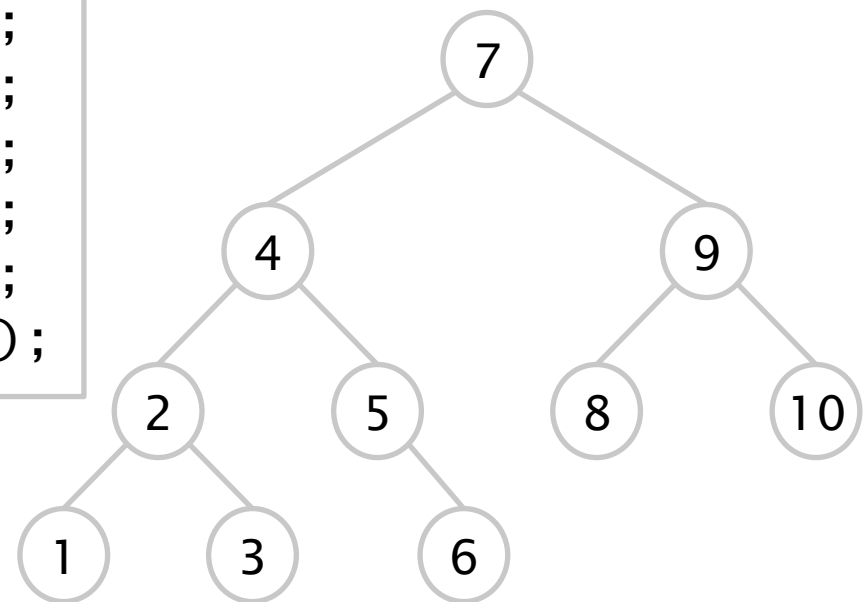
```

Abb* abb_retira (Abb* r, int chave){
    if (r == NULL) return NULL;
    else if (chave < r->chave)
        r->esq = abb_retira(r->esq, chave);
    else if (chave > r->chave)
        r->dir = abb_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 */
            Abb* t = r; r = r->dir; r->pai= t->pai; free (t);
        }
        else if (r->dir == NULL) { /* só tem filho à esquerda */
            Abb* t = r; r = r->esq; r->pai=t->pai; free (t);
        }
        else { /* nó tem os dois filhos: busca o sucessor */
            Abb* sucessor = r->dir;
            while (sucessor->esq != NULL) sucessor = sucessor->esq;
            r->chave = sucessor->chave; /* troca as chaves */
            sucessor->chave = chave;
            /* liga o filho do sucessor e o avo */
            if (sucessor->pai->esq == sucessor)
                sucessor->pai->esq = sucessor->dir; /* se sucessor for filho à esquerda */
            else
                sucessor->pai->dir = sucessor->dir; /* se sucessor for filho à direita */
            if (sucessor->dir!=NULL) sucessor->dir->pai=sucessor->pai;
            free(sucessor);
        }
    }
    return r;
}

```

Exemplo

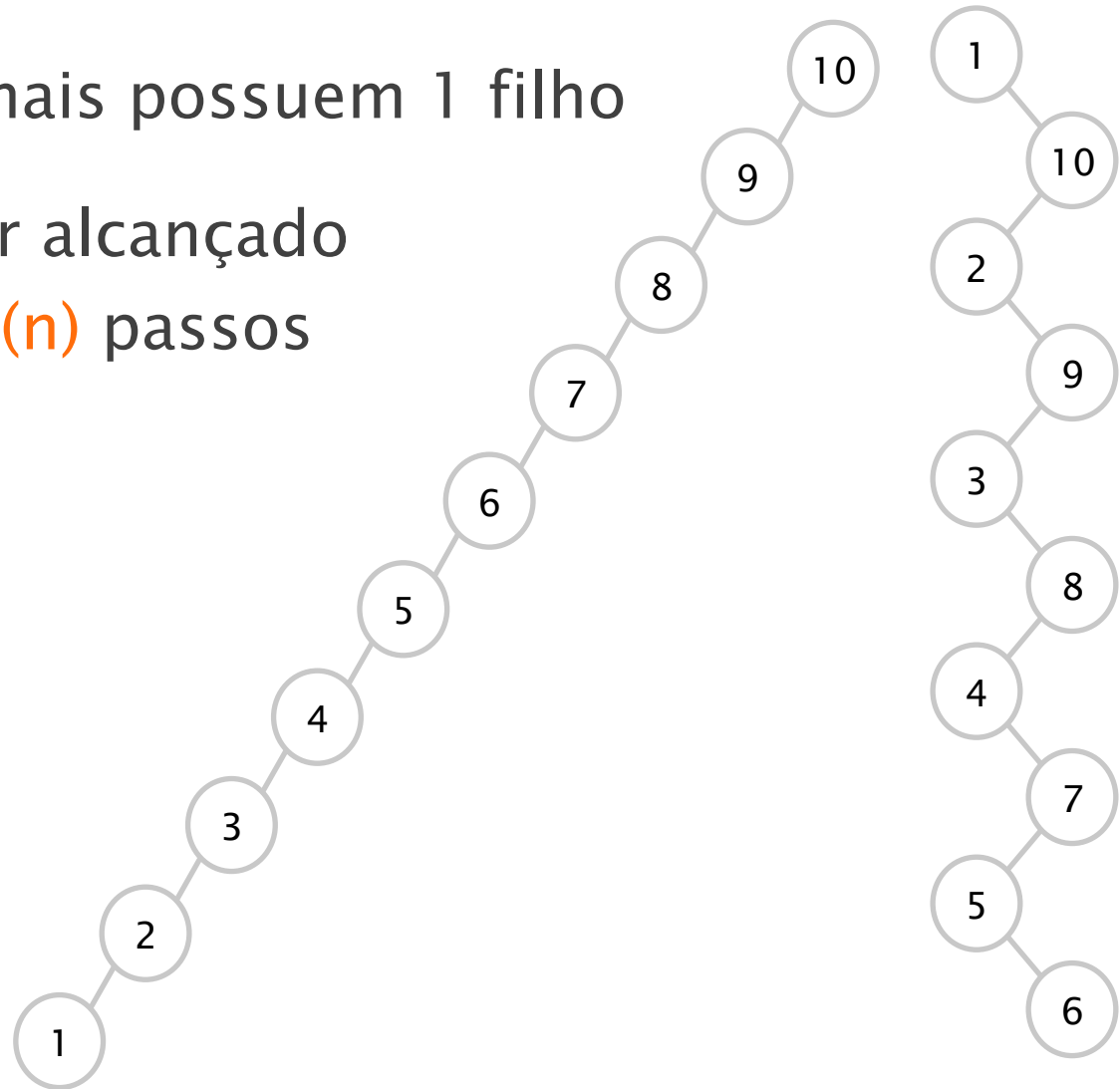
```
ArvBB *a;  
a = arvbb_cria_vazia();  
a = arvbb_insere_elemento(a,7);  
a = arvbb_insere_elemento(a,4);  
a = arvbb_insere_elemento(a,9);  
a = arvbb_insere_elemento(a,2);  
a = arvbb_insere_elemento(a,5);  
a = arvbb_insere_elemento(a,6);  
a = arvbb_insere_elemento(a,1);  
a = arvbb_insere_elemento(a,8);  
a = arvbb_insere_elemento(a,3);  
a = arvbb_insere_elemento(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

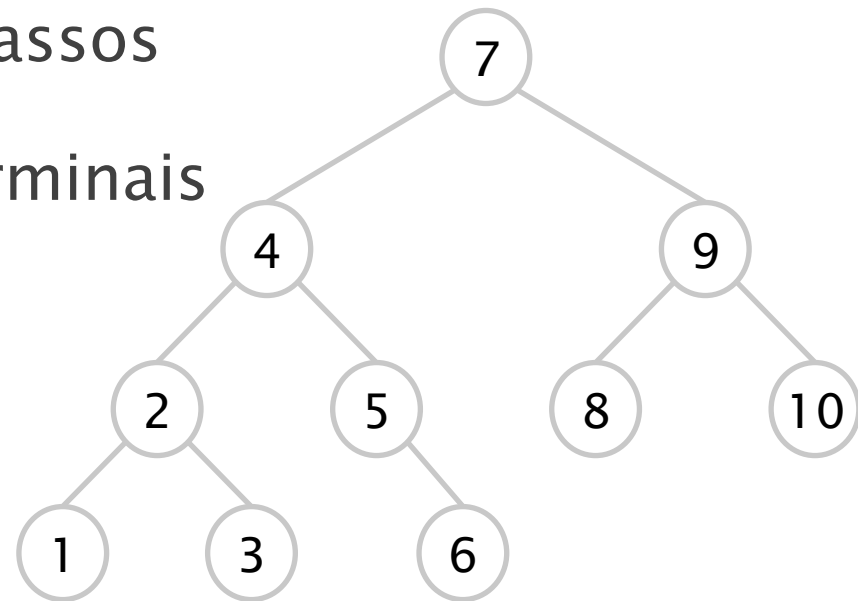
$$|h_e - h_d| \leq 1$$

h_e = altura da sub-árvore esquerda

h_d = 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



dúvidas?