

INF1007: Programação 2

10 – Árvores Binárias



Tópicos Principais

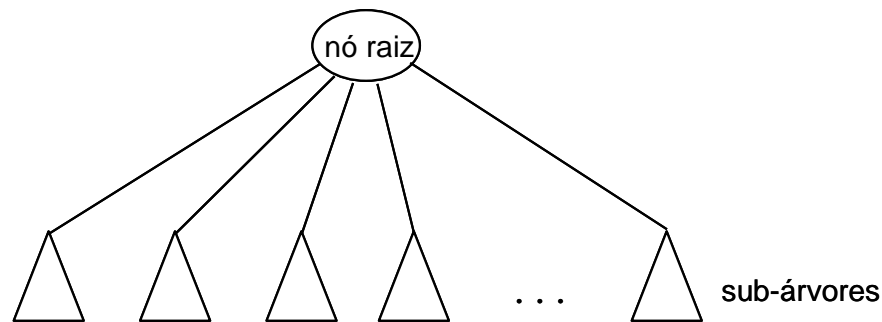
- Introdução
- Árvores binárias
 - Representação em C
 - Ordens de percurso em árvores binárias
 - Altura de uma árvore
- Árvores binária de busca (ABB)
- Funções para ABBs
 - Impressão
 - Busca

Tópicos Complementares

- Inserção em ABB
- Remoção em ABB

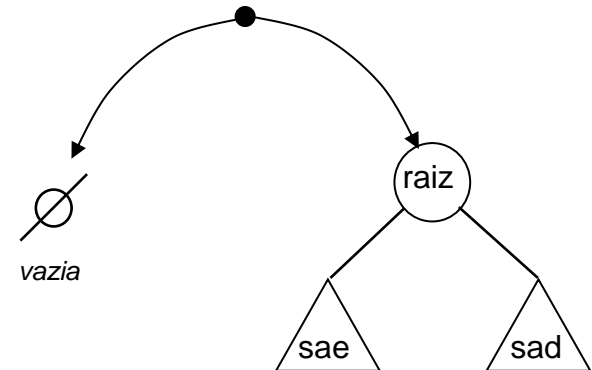
Introdução

- Árvore
 - um conjunto de nós tal que
 - existe um nó r , denominado *raiz*, com zero ou mais sub-árvores, cujas raízes estão ligadas a r
 - os nós raízes destas sub-árvores são os *filhos* de r
 - os *nós internos* da árvore são os nós com filhos
 - as *folhas* ou *nós externos* da árvore são os nós sem filhos



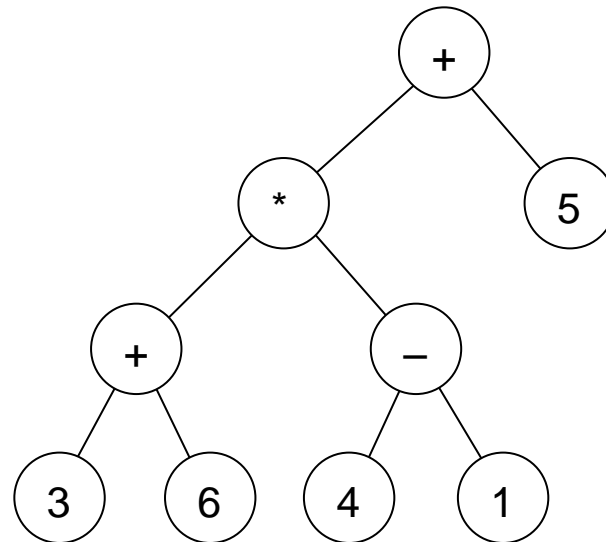
Árvores binárias

- um árvore em que cada nó tem zero, um ou dois filhos
- uma árvore binária é:
 - uma árvore vazia; ou
 - um nó raiz com duas sub-árvores:
 - a sub-árvore da direita (sad)
 - a sub-árvore da esquerda (sae)



Árvores binárias

- Exemplo
 - árvores binárias representando expressões aritméticas:
 - nós folhas representam operandos
 - nós internos operadores
 - exemplo: $(3+6)*(4-1)+5$



Árvores binárias - Implementação em C

- Representação de uma árvore:
 - através de um ponteiro para o nó raiz
- Representação de um nó da árvore:
 - estrutura em C contendo
 - a informação propriamente dita (exemplo: um caractere)
 - dois ponteiros para as sub-árvores, à esquerda e à direita

```
struct noArv {  
    char info;  
    struct noArv*  esq;  
    struct noArv*  dir;  
};
```

Árvores binárias - Implementação em C

- Interface do tipo abstrato Árvore Binária: [arv.h](#)

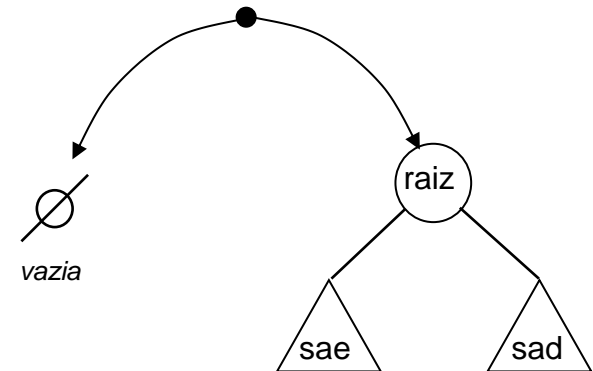
```
typedef struct noArv NoArv;  
  
NoArv* arv_criavazia (void);  
NoArv* arv_cria (char c, NoArv* e, NoArv* d);  
NoArv* arv_libera (NoArv* a);  
int  arv_vazia (NoArv* a);  
int  arv_pertence (NoArv* a, char c);  
void arv_imprime (NoArv* a);
```


Árvores binárias - Implementação em C

- implementação recursiva, em geral
- usa a definição recursiva da estrutura

Uma árvore binária é:

- uma árvore vazia; ou
- um nó raiz com duas sub-árvores:
 - a sub-árvore da direita (sad)
 - a sub-árvore da esquerda (sae)



Árvores binárias - Implementação em C

- função `arv_criavazia`
 - cria uma árvore vazia

```
NoArv* arv_criavazia (void)
{
    return NULL;
}
```

Árvores binárias - Implementação em C

- função `arv_cria`
 - cria um nó raiz dadas a informação e as duas sub-árvores, a da esquerda e a da direita
 - retorna o endereço do nó raiz criado

```
NoArv* arv_cria (char c, NoArv* sae, NoArv* sad)
{
    NoArv* p=(NoArv*)malloc(sizeof(NoArv)) ;
    if(p==NULL) exit(1);
    p->info = c;
    p->esq = sae;
    p->dir = sad;
    return p;
}
```

Árvores binárias - Implementação em C

- `criavazia` e `cria`
 - as duas funções para a criação de árvores representam os dois casos da definição recursiva de árvore binária:
 - uma árvore binária `NoArv* a;`
 - é vazia `a=arv_criavazia()`
 - é composta por uma raiz e duas sub-árvores `a=arv_cria(c,sae,sad);`

Árvores binárias - Implementação em C

- função `arv_libera`
 - libera memória alocada pela estrutura da árvore
 - as sub-árvores devem ser liberadas antes de se liberar o nó raiz
 - retorna uma árvore vazia, representada por `NULL`

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

Árvores binárias - Implementação em C

- função `arv_vazia`
 - indica se uma árvore é ou não vazia

```
int arv_vazia (NoArv* a)
{
    return a==NULL;
}
```

Árvores binárias - Implementação em C

- função `arv_pertence`
 - verifica a ocorrência de um caractere `c` em um de nós
 - retorna um valor booleano (1 ou 0) indicando a ocorrência ou não do caractere na árvore

```
int arv_pertence (NoArv* a, char c)
{
    if (arv_vazia(a))
        return 0; /* árvore vazia: não encontrou */
    else
        return a->info==c ||
               arv_pertence(a->esq,c) ||
               arv_pertence(a->dir,c);
}
```

Árvores binárias - Implementação em C

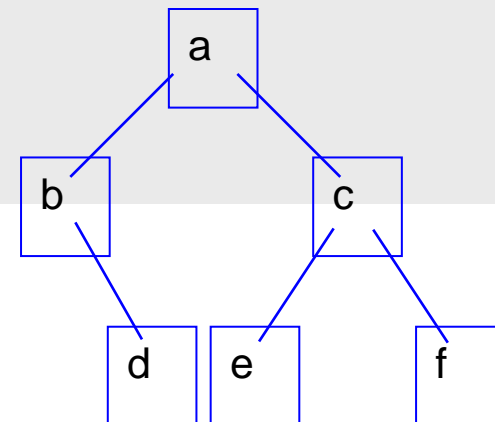
- função `arv_imprime`
 - percorre recursivamente a árvore, visitando todos os nós e imprimindo sua informação

```
void arv_imprime (NoArv* a)
{
    if (!arv_vazia(a)) {
        printf("%c ", a->info);    /* mostra raiz */
        arv_imprime(a->esq);        /* mostra sae */
        arv_imprime(a->dir);        /* mostra sad */
    }
}
```


Árvores binárias - Implementação em C

- Exemplo: <a <b <> <d <><>> > <c <e <><> > <f <><> > > >

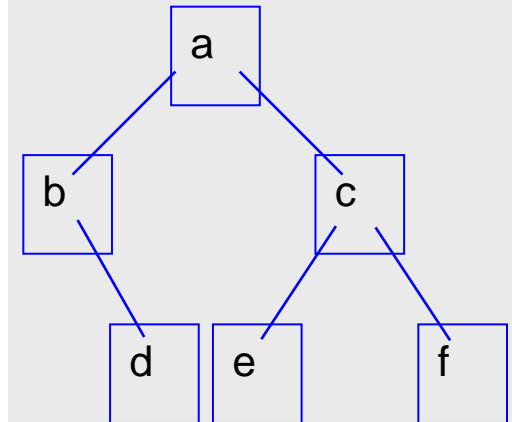
```
/* sub-árvore 'd' */
NoArv* a1= arv_cria('d',arv_criavazia(),arv_criavazia());
/* sub-árvore 'b' */
NoArv* a2= arv_cria('b',arv_criavazia(),a1);
/* sub-árvore 'e' */
NoArv* a3= arv_cria('e',arv_criavazia(),arv_criavazia());
/* sub-árvore 'f' */
NoArv* a4= arv_cria('f',arv_criavazia(),arv_criavazia());
/* sub-árvore 'c' */
NoArv* a5= arv_cria('c',a3,a4);
/* árvore 'a' */
NoArv* a = arv_cria('a',a2,a5 );
```



Árvores binárias - Implementação em C

- Exemplo: <a <b <> <d <><>> > <c <e <><> > <f <><> > > >

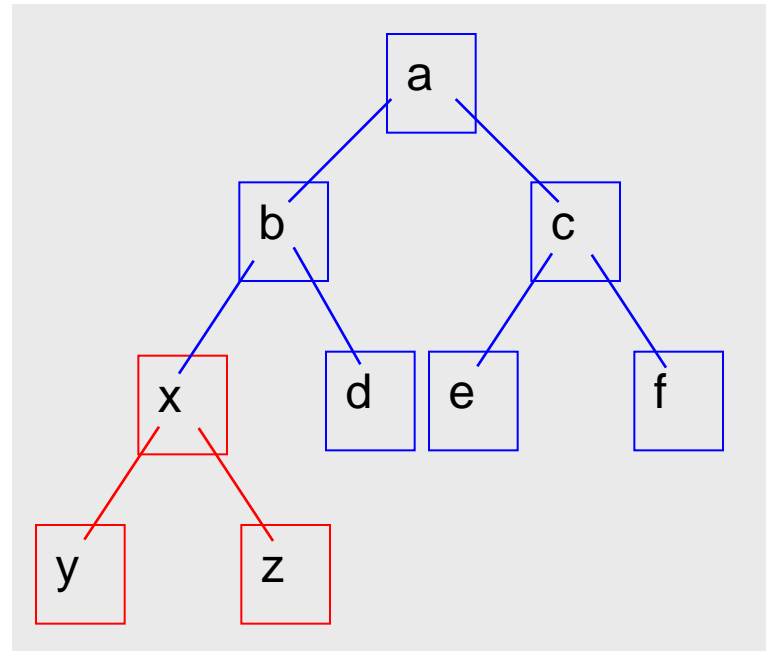
```
NoArv* a = arv_cria('a',  
    arv_cria('b',  
        arv_criavazia(),  
        arv_cria('d', arv_criavazia(), arv_criavazia())  
    ),  
    arv_cria('c',  
        arv_cria('e', arv_criavazia(), arv_criavazia()),  
        arv_cria('f', arv_criavazia(), arv_criavazia())  
    )  
);
```



Árvores binárias - Implementação em C

- Exemplo - acrescenta nós

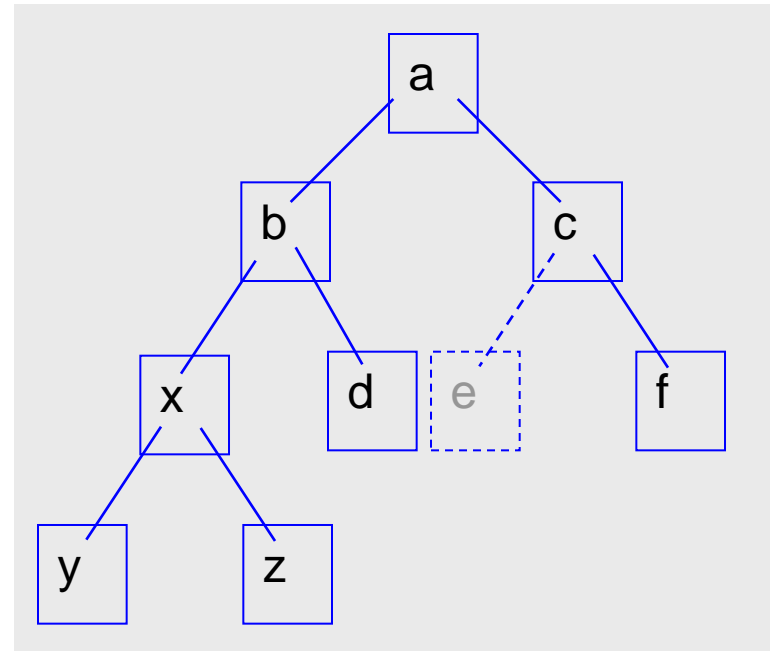
```
a->esq->esq =  
    arv_cria('x',  
        arv_cria('y',  
            arv_criavazia(),  
            arv_criavazia()),  
        arv_cria('z',  
            arv_criavazia(),  
            arv_criavazia())  
    );
```



Árvores binárias - Implementação em C

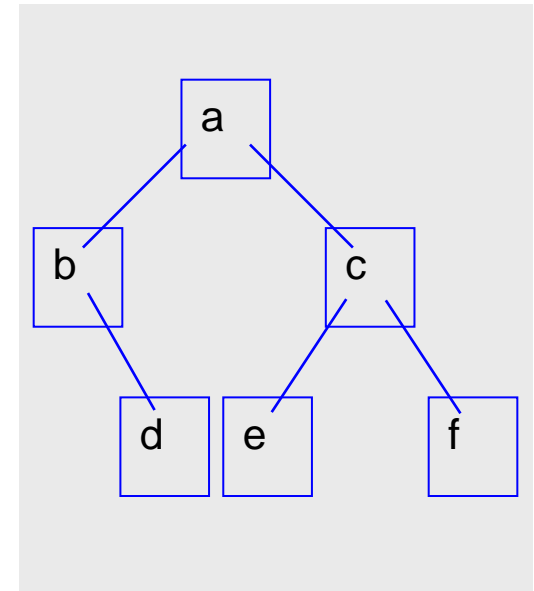
- Exemplo - libera nós

```
a->dir->esq = libera(a->dir->esq);
```



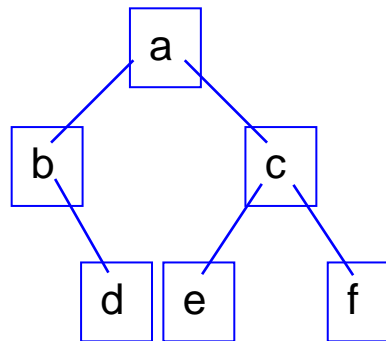
Árvores binárias - Ordens de percurso

- Ordens de percurso:
 - *pré-ordem*:
 - trata *raiz*, percorre *sae*, percorre *sad*
 - exemplo: a b d c e f
 - *ordem simétrica*:
 - percorre *sae*, trata *raiz*, percorre *sad*
 - exemplo: b d a e c f
 - *pós-ordem*:
 - percorre *sae*, percorre *sad*, trata *raiz*
 - exemplo: d b e f c a



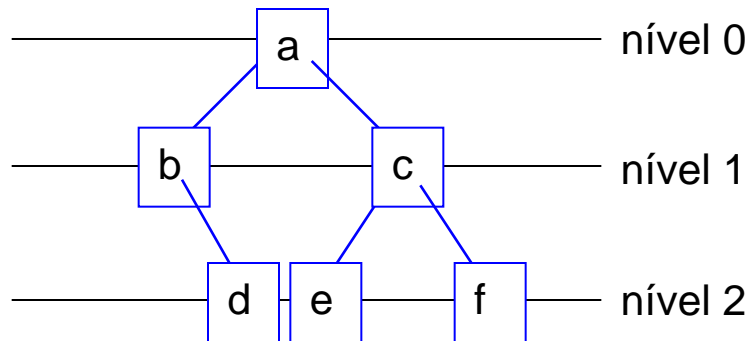
Árvores binárias - Altura

- Propriedade fundamental de árvores
 - só existe um caminho da raiz para qualquer nó
- Altura de uma árvore
 - comprimento do caminho mais longo da raiz até uma das folhas
 - a altura de uma árvore com um único nó raiz é zero
 - a altura de uma árvore vazia é -1
 - exemplo:
 - $h = 2$



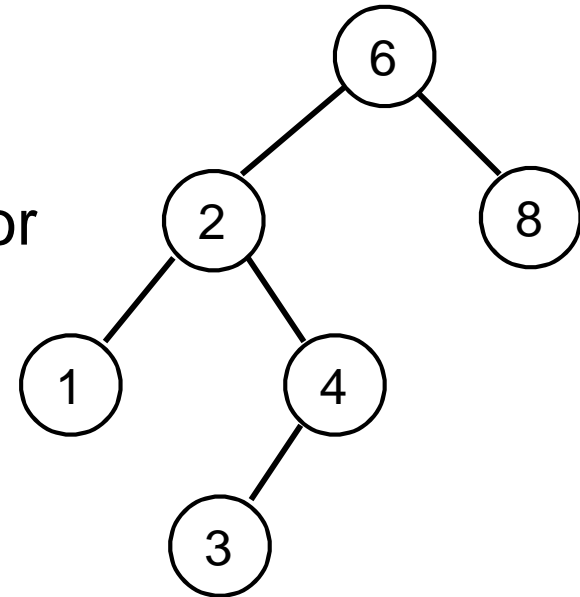
Árvores binárias - Altura

- Nível de um nó
 - a raiz está no nível 0, seus filhos diretos no nível 1, ...
 - o último nível da árvore é a altura da árvore



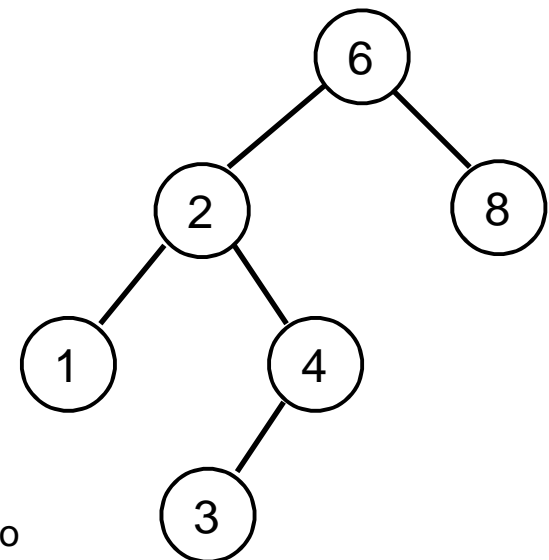
Árvore Binária de Busca (ABB)

- o valor associado à raiz é sempre maior que o valor associado a qualquer nó da sub-árvore à esquerda (*sae*) e
- o valor associado à raiz é sempre menor ou igual (para permitir repetições) que o valor associado a qualquer nó da sub-árvore à direita (*sad*)
- quando a árvore é percorrida em ordem simétrica (*sae - raiz - sad*), os valores são encontrados em ordem não decrescente



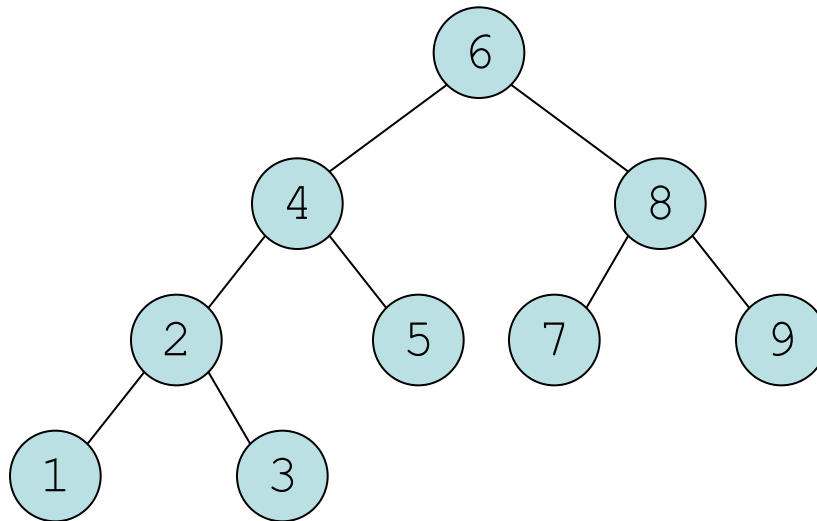
Pesquisa em Árvore Binária de Busca

- compare o valor dado com o valor associado à raiz
- se for igual, o valor foi encontrado
- se for menor, a busca continua na sae
- se for maior, a busca continua na sad



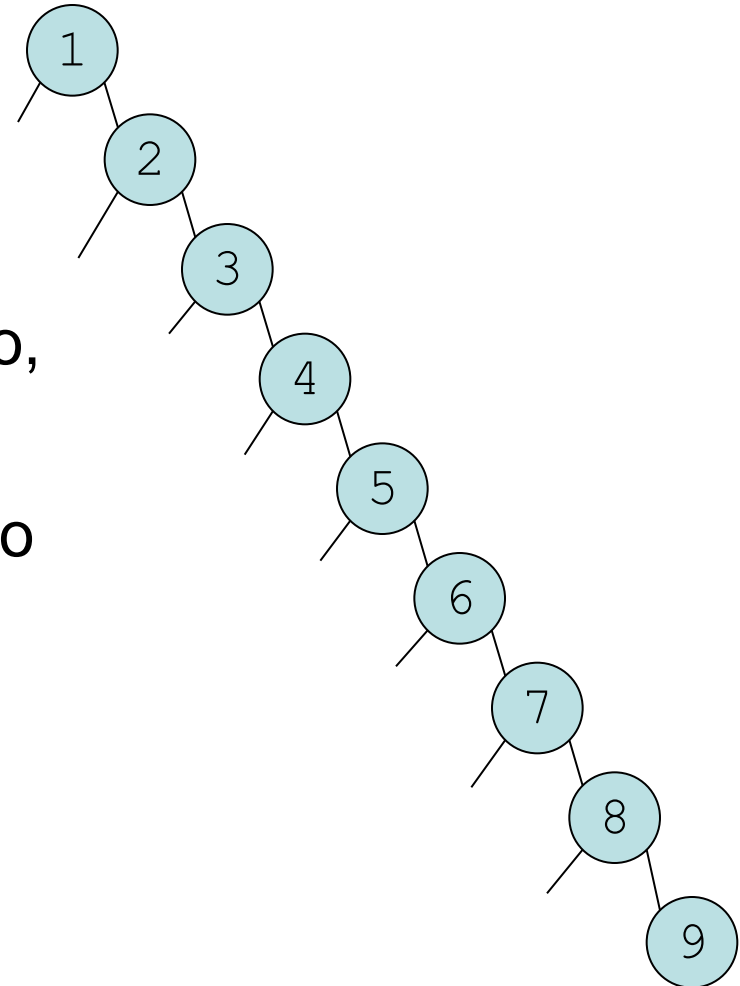
Árvore Binária de Busca Balanceadas

- os nós internos têm todos, ou quase todos, 2 filhos
- qualquer nó pode ser alcançado a partir da raiz em $O(\log n)$ passos



Árvore Binária de Busca Degenerada

- todos os nós têm apenas 1 filho, com exceção da (única) folha
- qualquer nó pode ser alcançado a partir da raiz em $O(n)$ passos



Tipo Árvore Binária de Busca

- árvore é representada pelo ponteiro para o nó raiz

```
struct noArv {  
    int info;  
    struct noArv* esq;  
    struct noArv* dir;  
};  
  
typedef struct noArv NoArv;
```

ABB: Criação

- árvore vazia representada por NULL:

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

ABB: Impressão

- imprime os valores da árvore em ordem crescente, percorrendo os nós em ordem simétrica

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

ABB: Busca

- explora a propriedade de ordenação da árvore
- possui desempenho computacional proporcional à altura ($O(\log n)$ para o caso de árvore balanceada)

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

ABB: Inserção

- recebe um valor v a ser inserido
- retorna o eventual novo nó raiz da (sub-)árvore
- para adicionar v na posição correta, faça:
 - se a (sub-)árvore for vazia
 - crie uma árvore cuja raiz contém v
 - se a (sub-)árvore não for vazia
 - compare v com o valor na raiz
 - insira v na sae ou na sad, conforme o resultado da comparação


```

NoArv* abb_inserere (NoArv* a, int v)
{
    if (a==NULL) {
        a = (NoArv*)malloc(sizeof(NoArv)) ;
        a->info = v;
        a->esq = a->dir = NULL;
    }

    else if (v < a->info)
        a->esq = abb_inserere(a->esq,v) ;

    else /* v >= a->info */
        a->dir = abb_inserere(a->dir,v) ;

    return a;

```

é necessário atualizar os ponteiros para as sub-árvores à esquerda ou à direita quando da chamada recursiva da função, pois a função de inserção pode alterar o valor do ponteiro para a raiz da (sub-)árvore.

cria

insere 6

insere 4

insere 8

insere 2

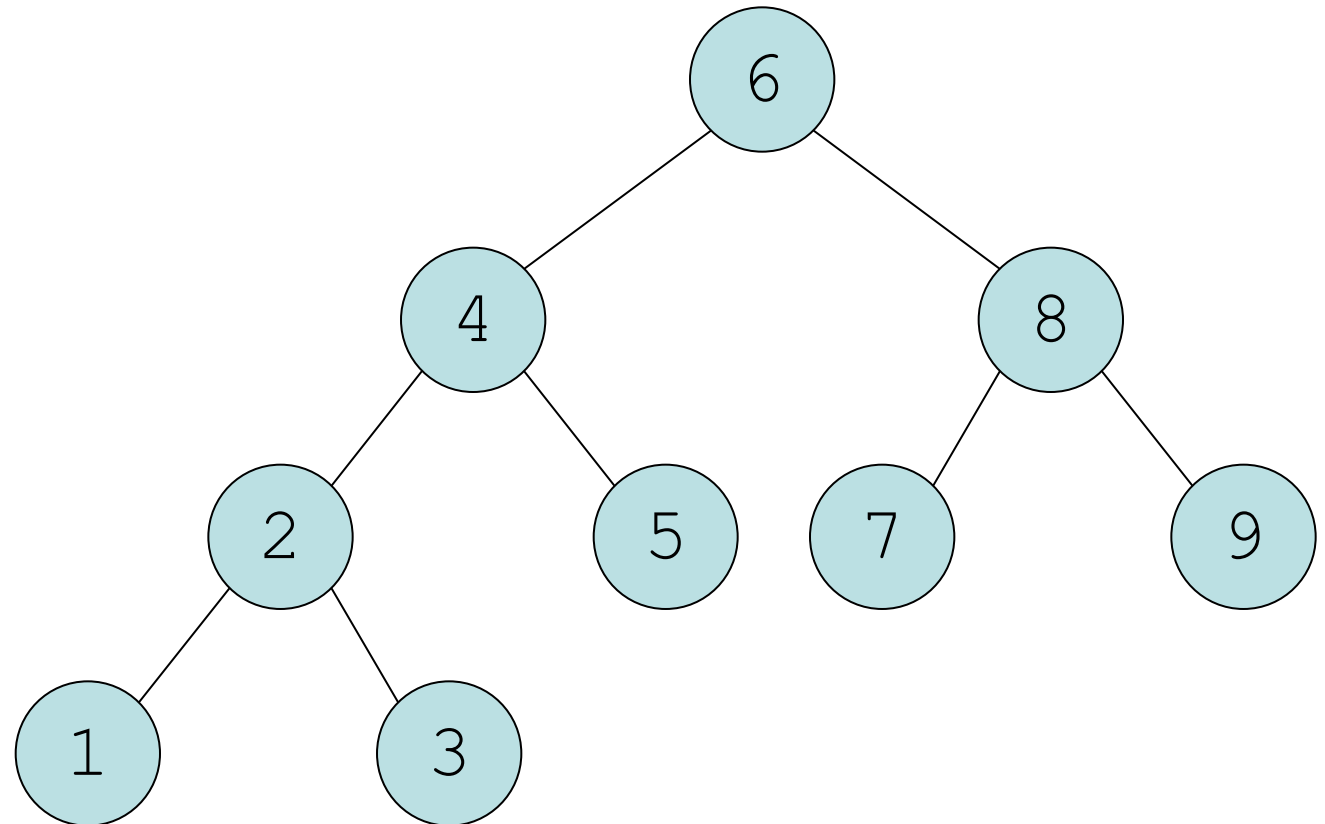
insere 5

insere 1

insere 3

insere 7

insere 9



cria

insere 6

insere 4

insere 8

insere 2

insere 5

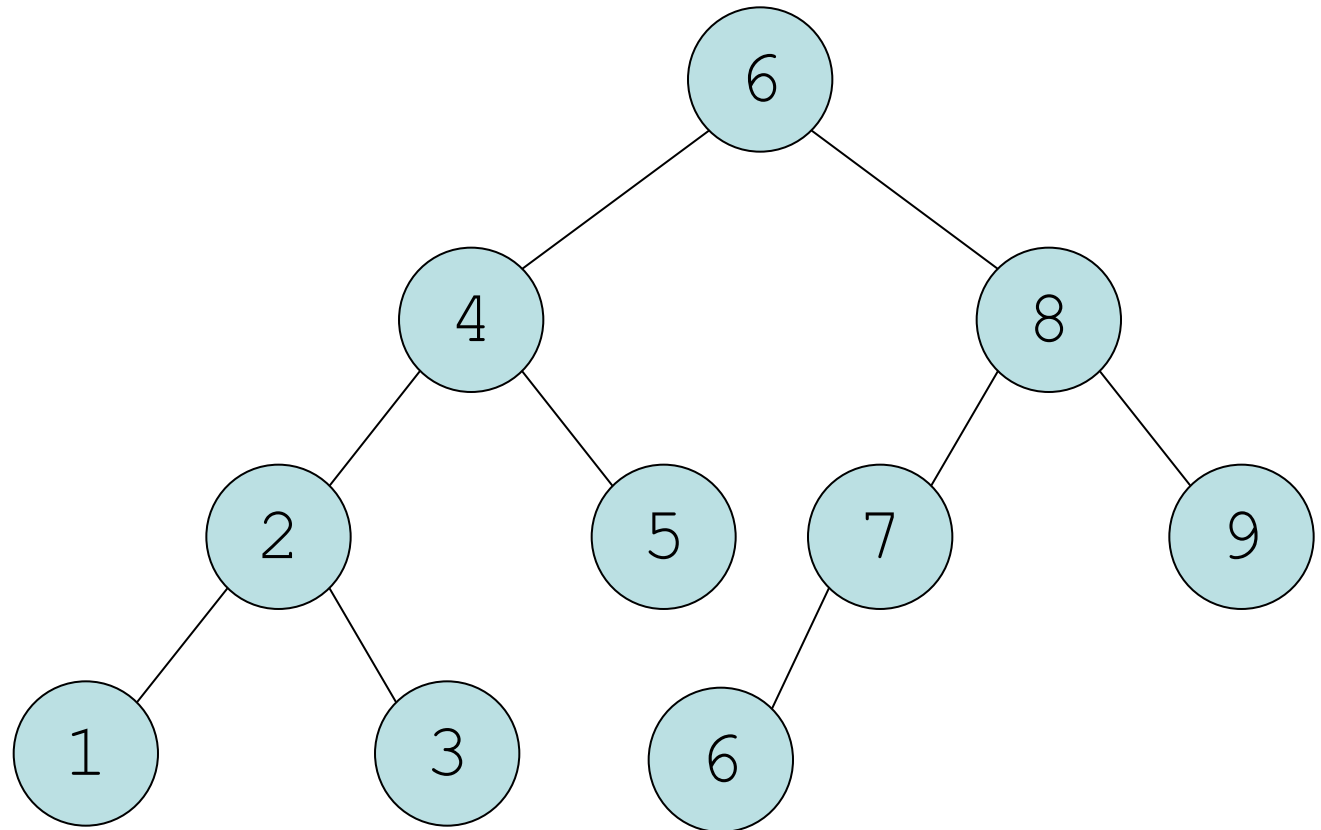
insere 1

insere 3

insere 7

insere 9

insere 6



a repetição está permitida!

```
NoArv* abb_inserere (NoArv* a, int v)
{
    if (a==NULL) {
        a = (NoArv*)malloc(sizeof(NoArv)) ;
        a->info = v;
        a->esq = a->dir = NULL;
    }

    else if (v < a->info)
        a->esq = abb_inserere(a->esq,v) ;

    else if (v > a->info)
        a->dir = abb_inserere(a->dir,v) ;

    return a;
}
```

ABB: Remoção

- recebe um valor v a ser retirado
- retorna a eventual nova raiz da árvore
- para remover v , faça:
 - se a árvore for vazia
 - nada tem que ser feito
 - se a árvore não for vazia
 - compare o valor armazenado no nó raiz com v
 - se for maior que v , retire o elemento da sub-árvore à esquerda
 - se for menor do que v , retire o elemento da sub-árvore à direita
 - se for igual a v , retire a raiz da árvore

ABB: Remoção

- para retirar a raiz da árvore, há 3 casos:
 - caso 1: a raiz que é folha
 - caso 2: a raiz a ser retirada possui um único filho
 - caso 3: a raiz a ser retirada tem dois filhos

ABB: Remoção de folha

- Caso 1: a raiz da sub-árvore é folha da árvore original
 - libere a memória alocada pela raiz
 - retorne a raiz atualizada, que passa a ser NULL

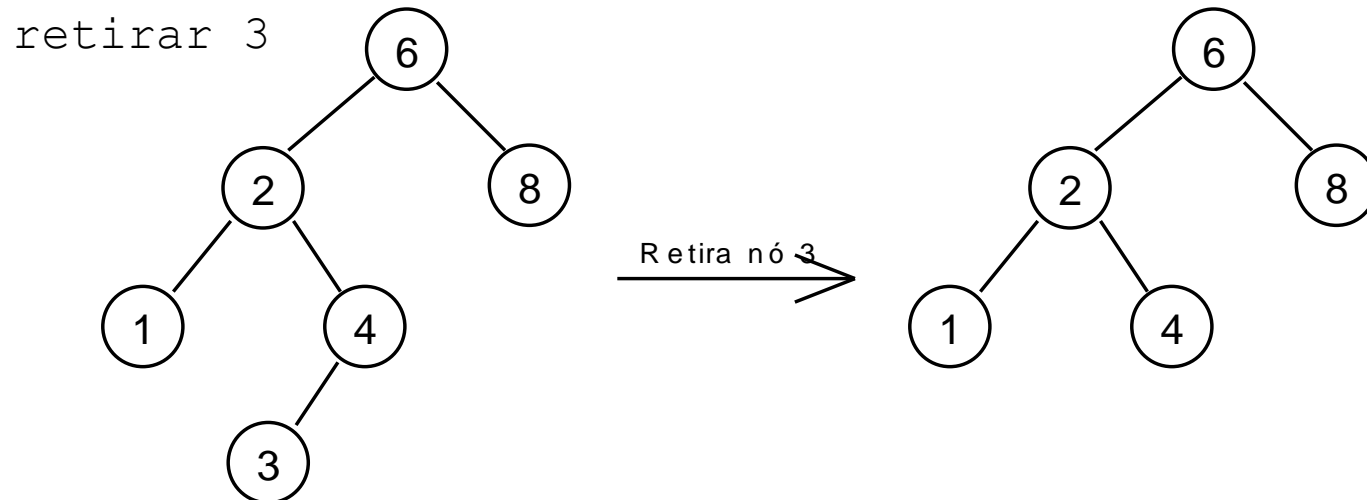


ABB: Remoção de pai de filho único

- Caso 2: a raiz a ser retirada possui um único filho
 - libere a memória alocada pela raiz
 - a raiz da árvore passa a ser o único filho da raiz

retirar 4

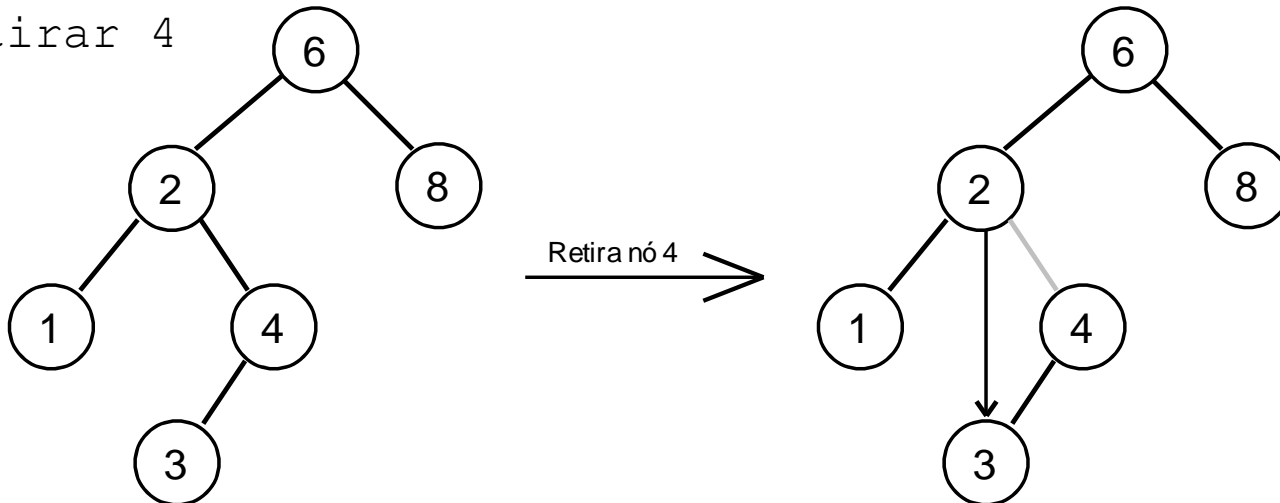
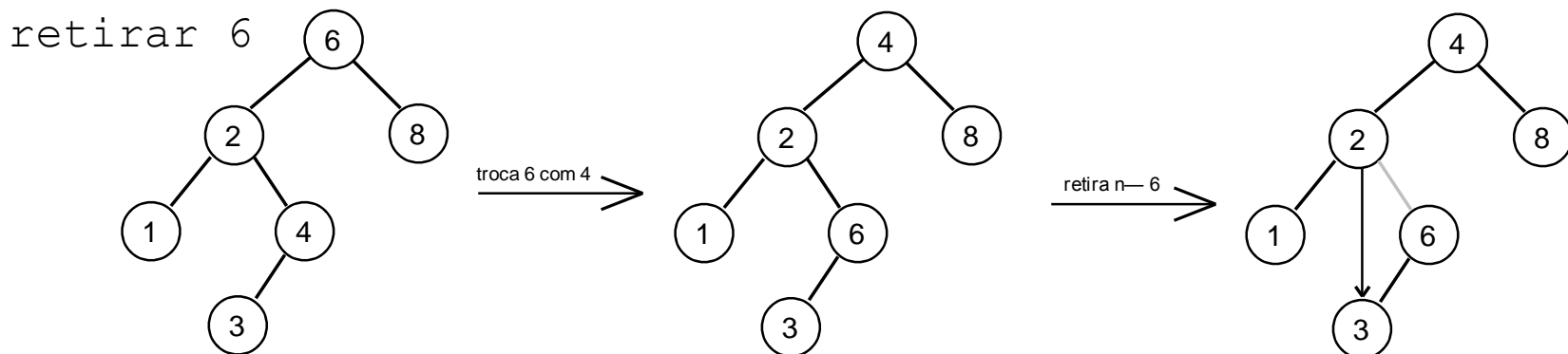


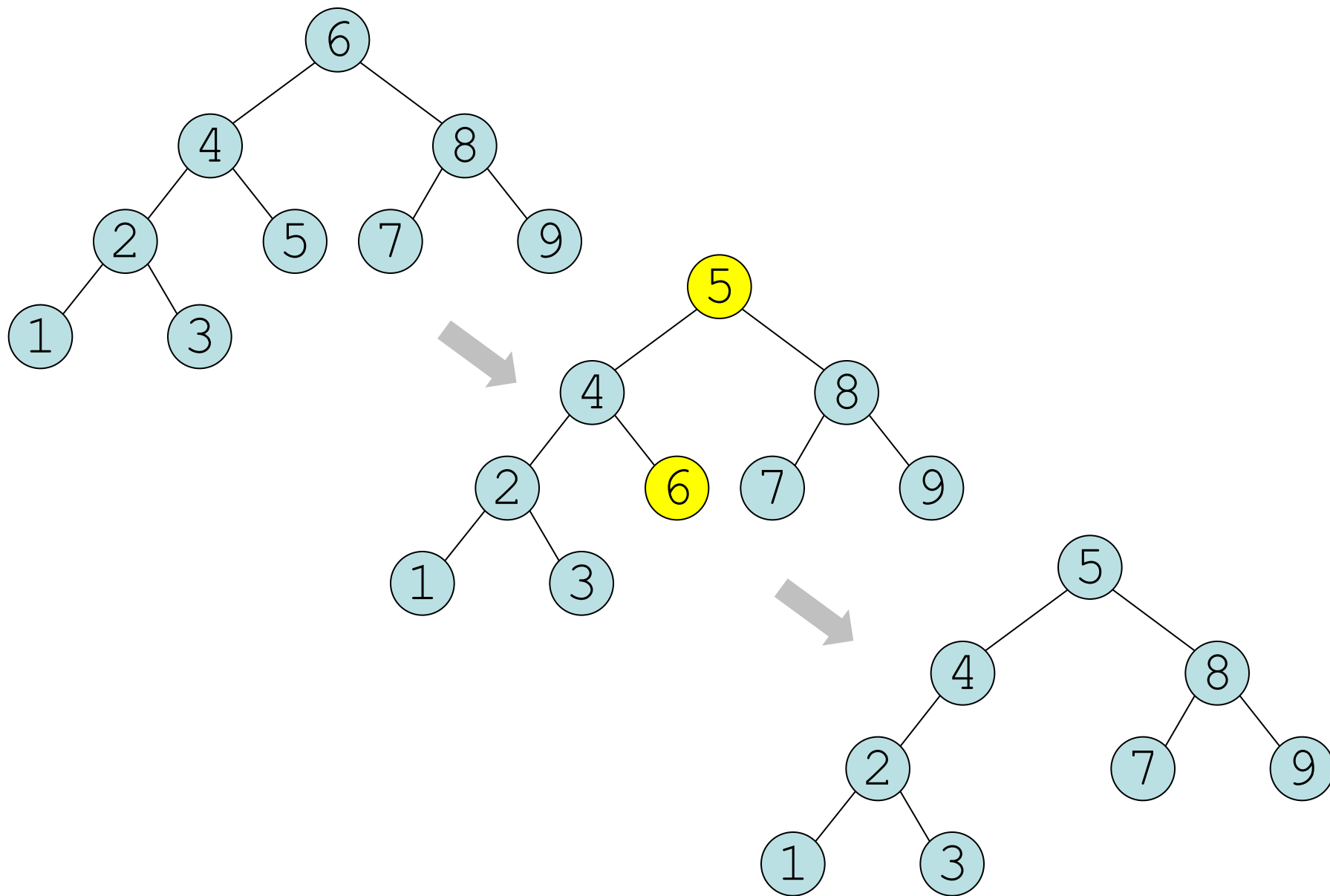
ABB: remoção de pai de dois filhos

- Caso 3: a raiz a ser retirada tem dois filhos
 - encontre o nó N que precede a raiz na ordenação (o elemento mais à direita da sub-árvore à esquerda)
 - troque o dado da raiz com o dado de N
 - retire N da sub-árvore à esquerda (que agora contém o dado da raiz que se deseja retirar)
 - retirar o nó N mais à direita é trivial, pois N é um nó folha ou N é um nó com um único filho (no caso, o filho da direita nunca existe)



```
NoArv* abb_retira (NoArv* r, int v)
{
    if (r == NULL)
        return NULL;
    else if (r->info > v)
        r->esq = abb_retira(r->esq, v);
    else if (r->info < v)
        r->dir = abb_retira(r->dir, v);
    else {
        /* achou o nó a remover */
        /* nó sem filhos */
        if (r->esq == NULL && r->dir == NULL) {
            free (r);
            r = NULL;
        }
        /* nó só tem filho à direita */
        else if (r->esq == NULL) {
            NoArv* t = r;
            r = r->dir;
            free (t);
        }
    }
}
```

```
/* só tem filho à esquerda */  
else if (r->dir == NULL) {  
    NoArv* t = r;  
    r = r->esq;  
    free (t);  
}  
/* nó tem os dois filhos */  
else {  
    NoArv* f = r->esq;  
    while (f->dir != NULL) {  
        f = f->dir;  
    }  
    r->info = f->info; /* troca as informações */  
    f->info = v;  
    r->esq = abb_retira(r->esq,v);  
}  
}  
return r;  
}
```



Referências

Waldemar Celes, Renato Cerqueira, José Lucas Rangel,
Introdução a Estruturas de Dados, Editora Campus
(2004)

Capítulo 13 – Árvores

Capítulo 17 – Busca