Estruturas de Dados 1

481440

Julho/2018

Mario Liziér
lizier@ufscar.br

ABB – *Selection/Rank*

- Considerando uma sequência ordenada, temos:
  - *Selection(**n**)* : retorna o elemento da posição de número ***n*** da sequência ordenada
  - *Rank*(***x***) : retorna o número de chaves menores que o elemento ***x***
  - Exemplos:
    - *Select*(3) = D
    - *Select*(9) = J
    - *Rank*(C) = 2
    - *Rank*(F) = 5

	0	1	2	3	4	5	6	7	8	9
Sequência de chaves:	A	B	C	D	E	F	G	H	I	J

ABB – Seleção (*Selection*)

- *Selection(i)* : retorna a *i-ésima* chave do conjunto

```
T select(ABB *arvore, int k) {
    assert( k >= 0 && k < size(arvore->raiz) );
    return select_node(arvore->raiz, k)->data;
}

struct node* select_node(struct node* p, int k) {
    if (!p) // nunca deve acontecer
        return 0;
    int t = size(p->esquerda);
    if (t > k)
        return select_node(p->esquerda, k);
    else if (t < k)
        return select_node(p->direita, k-t-1);
    else
        return p;
}
```

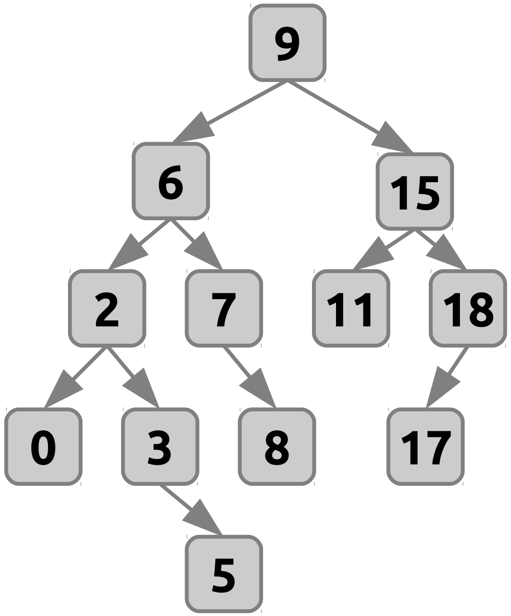


ABB – *Rank*

- *rank(key)* : retorna a posição da chave *key* no conjunto (número de chaves menores que *key*)

```
int rank(ABB *arvore, T key) {
    return rank_node(key, arvore->raiz);
}

int rank_node(T key, struct node* p) {
    if (!p)
        return 0;
    else if(p->data.key > key)
        return rank_node(key, p->esquerda);
    else if (p->data.key < key)
        return size(p->esquerda) +
                    rank_node(key, p->direita) + 1;
    else
        return size(p->esquerda);
}
```

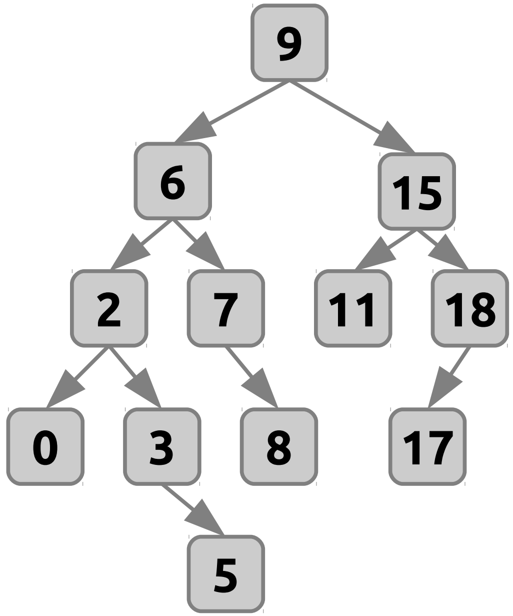


ABB – Busca por intervalo (*range search*)

- Procurar por todos os nós dentro de um intervalo
  - *rangeSearch*(3,7)
    - Elementos: 3, 5, 6 e 7
  - *rangeSearch*(4,11)
    - Elementos: 5, 6, 7, 8, 9 e 11
  - *rangeSearch*(10, 14)
    - Elemento: 11

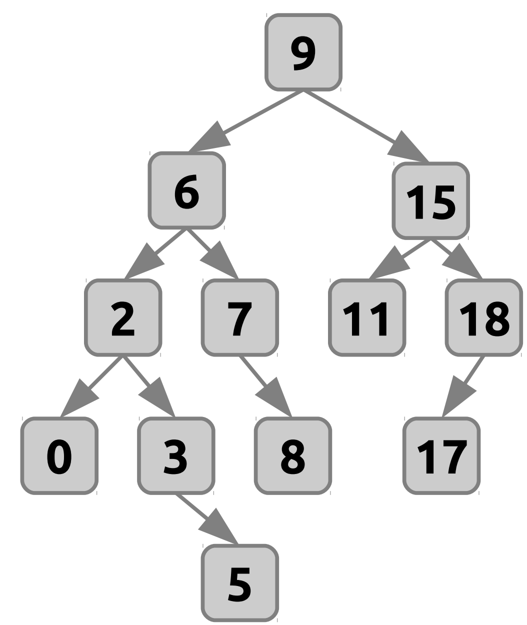
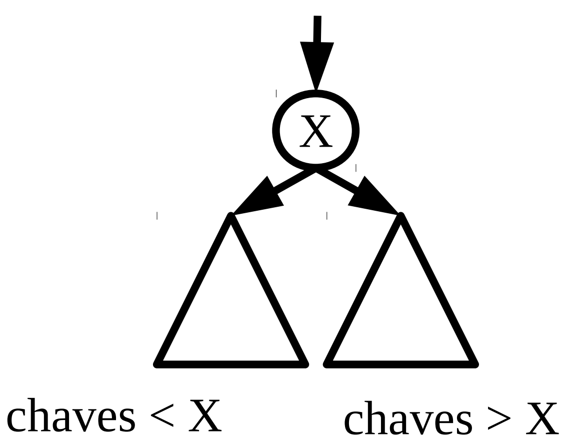


ABB – Busca por intervalo (*range search*)

- Busca pelo intervalo:
  - Caso 1: Todo o intervalo que procuramos está contido em uma subárvore (esquerda ou direita)
    - Igualdades incluem o nó
  - Caso 2: O intervalo é particionado pelas duas subárvores



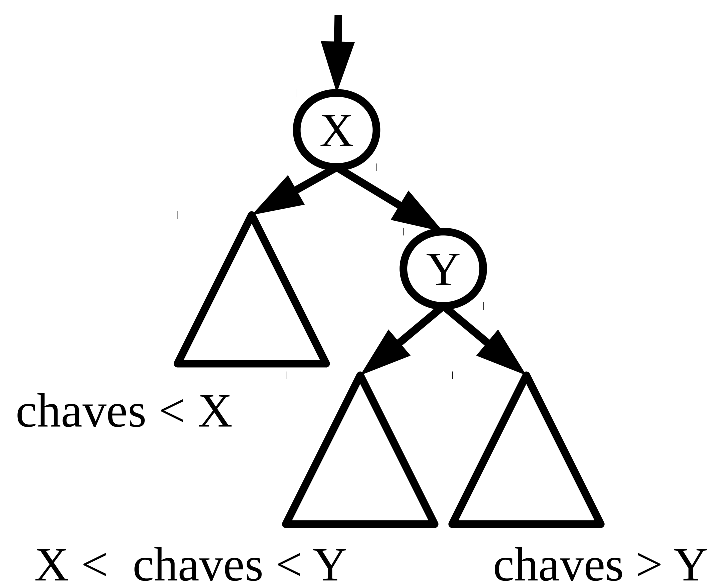


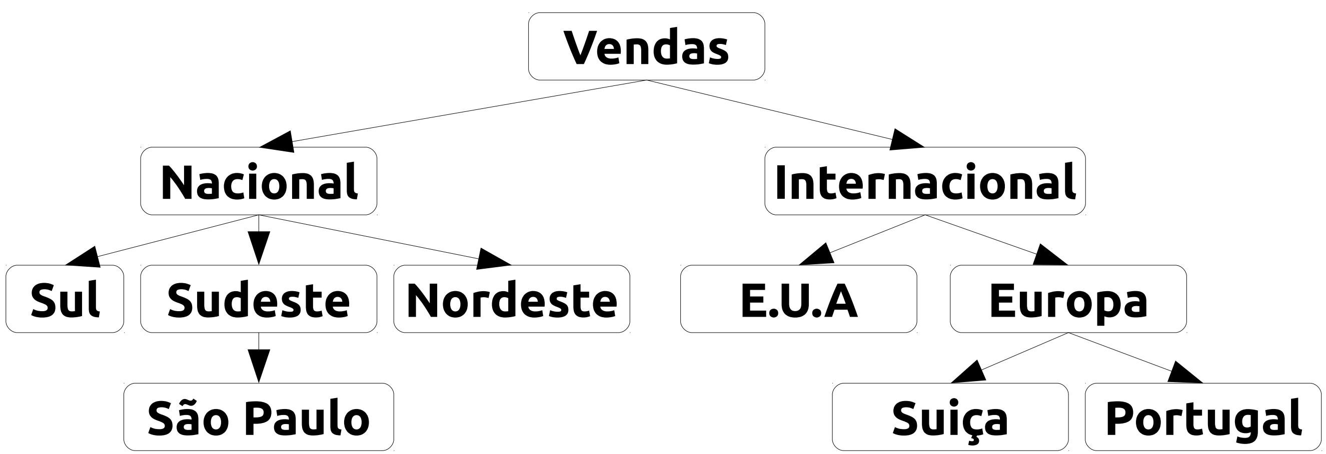
ABB – Busca por intervalo (*range search*)

- Percurso em-ordem
  - Apenas no intervalo em questão (as recursões não são realizadas se a subárvore não possa conter elementos do intervalo)

```
void rangeSearch(ABB *arvore, queue *q, T menor, T maior) {
     clear(q);
    rangeSearch_private(arvore->raiz, q, menor, maior);
}

void rangeSearch_private(struct node* p, queue *q, T menor, T maior) {
    if (!n)
        return;

    if( menor < p->data )
        rangeSearch_private( p->esquerda, q, menor, maior);

    if( menor <= p->data && maior >= p->data )
        push( q, p->data );

    if( maior > p->data )
        rangeSearch_private( p->direita, q, menor, maior);
}
```

Exercício

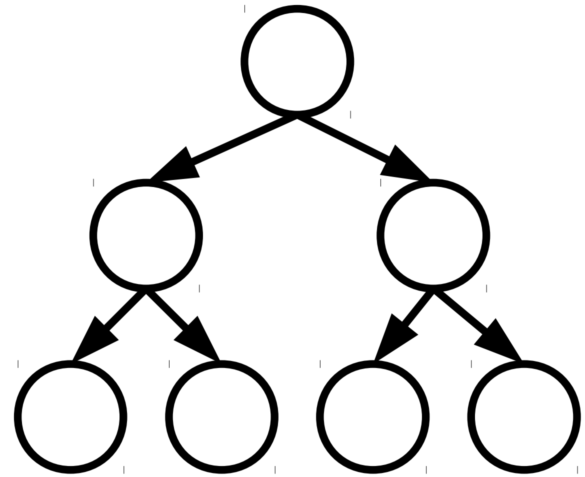
- O modelo "<u>por parênteses e com indentação</u>" representa uma árvore utilizando:
  - uma linha por nó,
  - indentando cada nó conforme a sua profundidade e
  - colocando cada subnível entre parênteses.
- Um exemplo de árvore representada por este modelo:



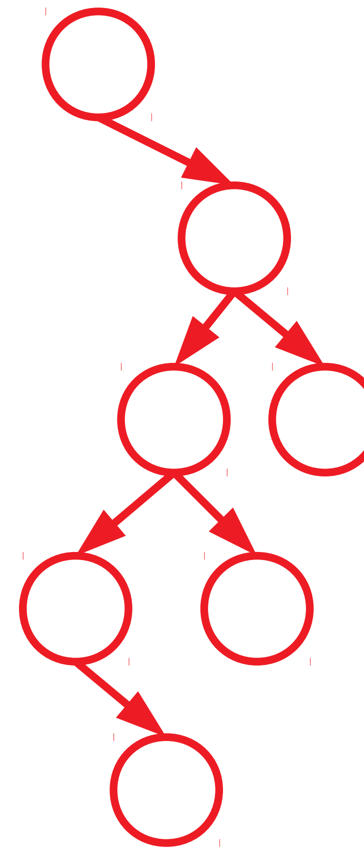

```
Vendas (
   Nacional (
      Sul
      Sudeste (
         São Paulo
      )
      Nordeste
   )
   Internacional (
      E.U.A.
      Europa (
         Suiça
         Portugal
      )
   )
)
```

- Faça um algoritmo que dado uma árvore qualquer, exiba na tela seguindo o modelo de representação descrito.

Árvores



- Buscas: O(h)
- Busca por intervalo: O(h+k)
- Inserção e remoção: O(h)
- Maior/Menor: O(h)
- Teto/Chão: O(h)
- *Selection*/*Rank*: O(h)
- Percursos: O(h+k)
- Profundidade: O(h)



- A altura da árvore é importante!

Árvore

- O que é uma altura ruim? e boa?
  - $\log N < h < N$
  - O ideal é termos a menor altura possível: $\log N$
- Precisamos então ter um critério para detectar (e medir) o quão ruim/boa é a altura de uma determinada árvore
- Critério de balanceamento:
  - Árvores AVL
  - Árvores Vermelha e Preta (ou rubro-negra ou *red-black* ou RB)

Árvore AVL

- Autores: Adel'son-Vel'skii e Landis (1962)
- Árvore Binária de Busca
- Algoritmos de inserção e remoção preservam o balanceamento da árvore
  - Logo, define um critério de balanceamento
- Campo adicional por nó:
  - 2-bits para armazenar a diferença das alturas das subárvores

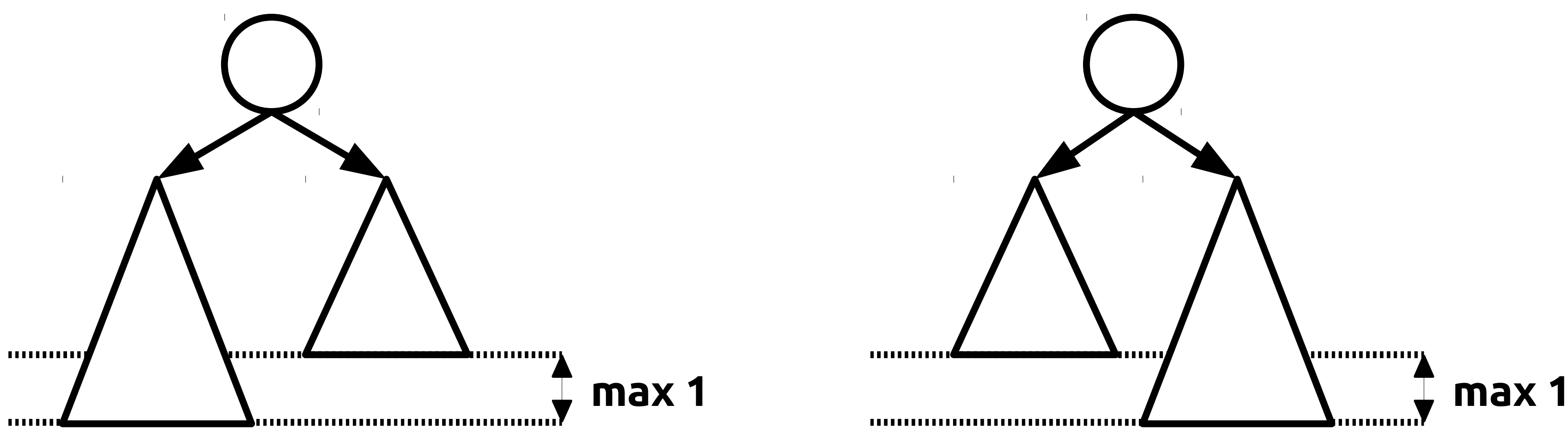
```
struct node {
   T data;
   struct node *esquerda, *direita;
   int bal; // na verdade precisamos de 2-bits
};
```

Critério de balanceamento AVL

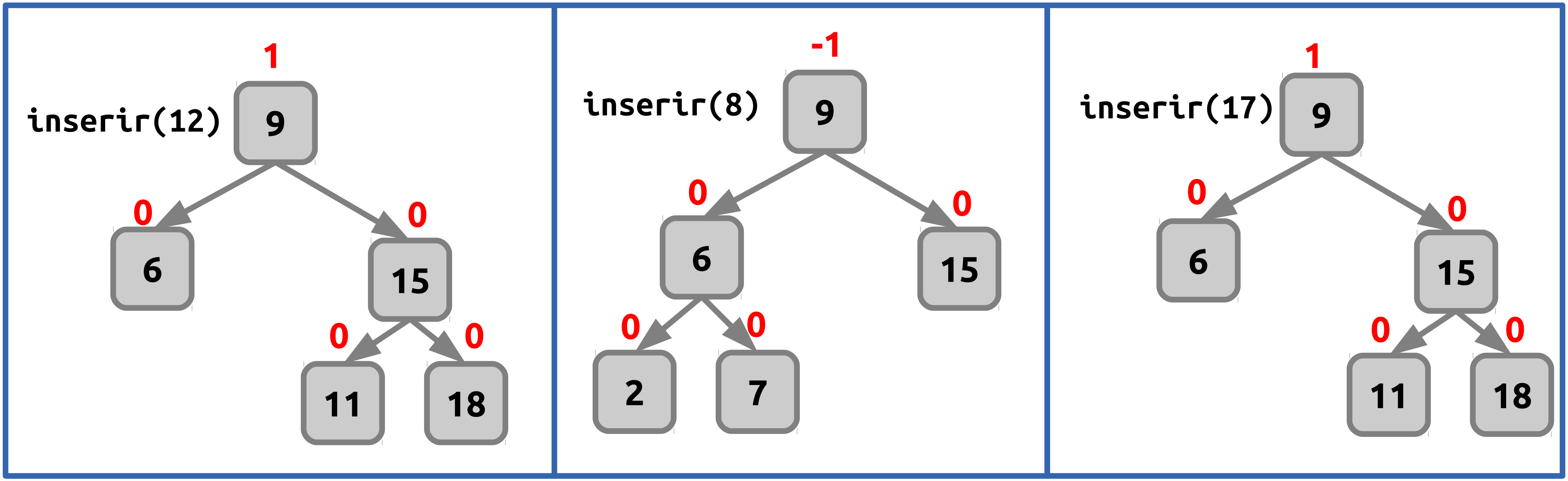
- Uma árvore é dita ser balanceada quando:
  - A diferença das alturas das subárvores

    $$bal = h_{direita} - h_{esquerda}$$

    de qualquer nó é igual a -1, 0 ou 1
  - Ou seja, a maior subárvore de um nó pode ter no máximo 1 nível a mais que a outra subárvore

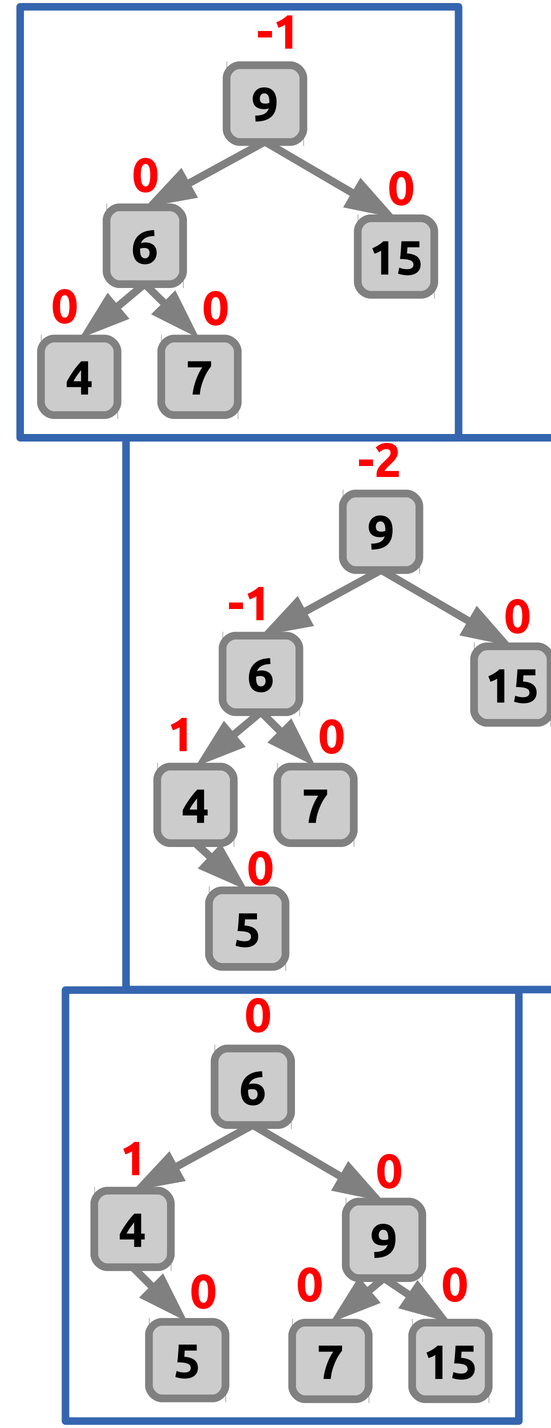


Desbalanceamento



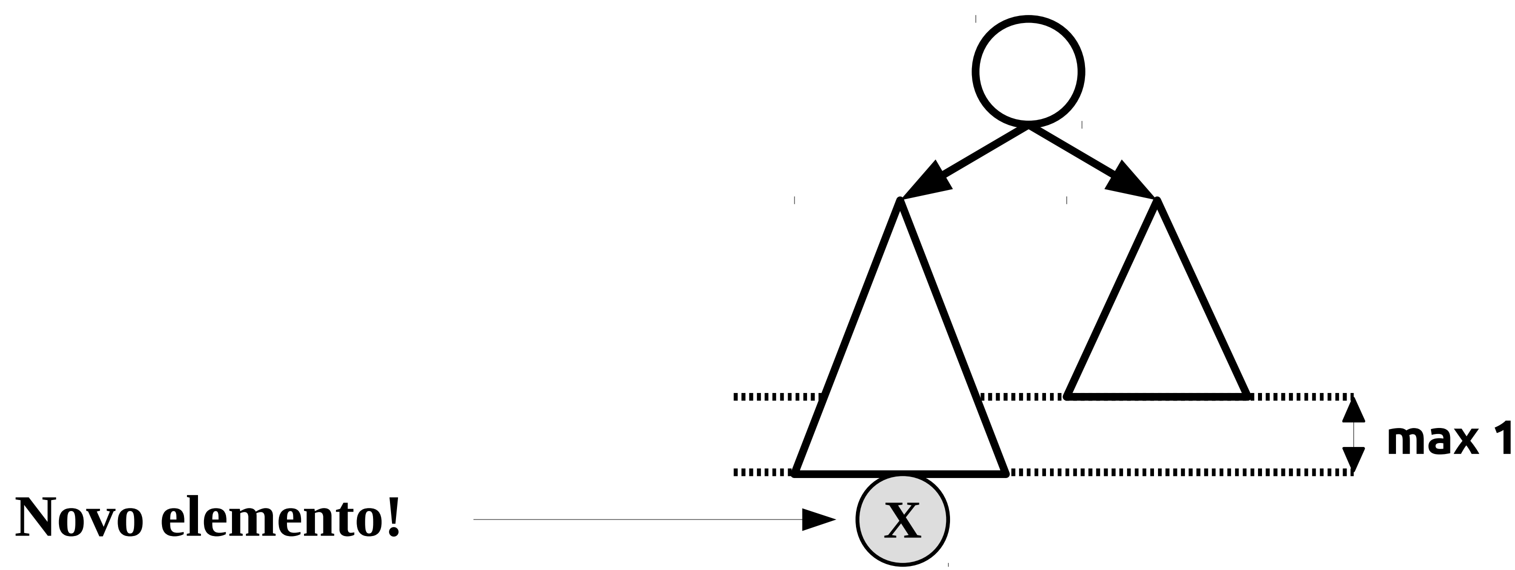
AVL - Inserção

- Inserir como em uma árvore binária de busca (ABB)
- Na "volta" da recursão verificamos os nós que violam o balanceamento AVL, pois apenas subárvores que contém o elemento inserido tiveram sua altura alterada
- Correção dos nós desbalanceados:
  - Apenas ancestrais podem estar desbalanceados
  - Identificação dos 3 nós "**primários**"
  - Quatro casos distintos:
    - ***EE***, ***DD***, ***ED*** e ***DE***

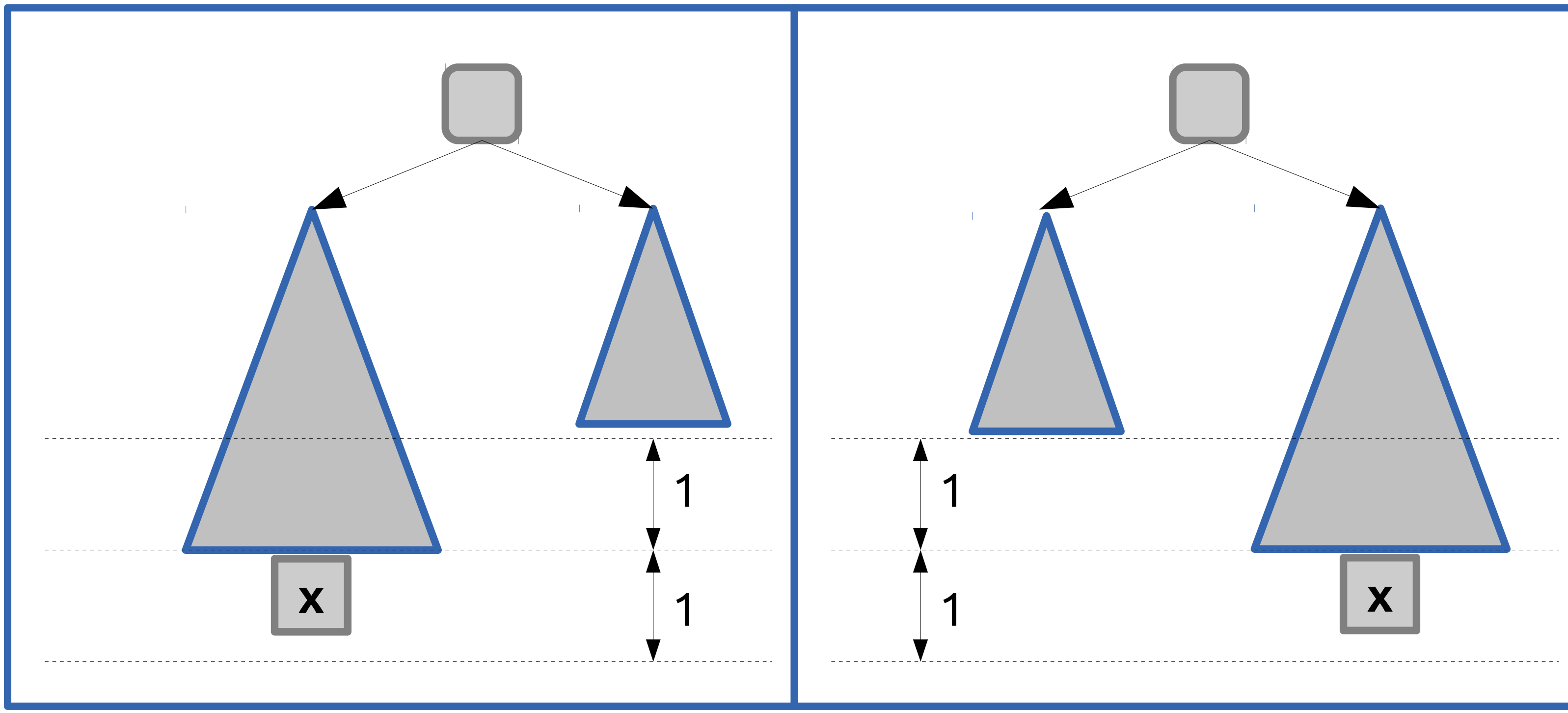


AVL - Inserção

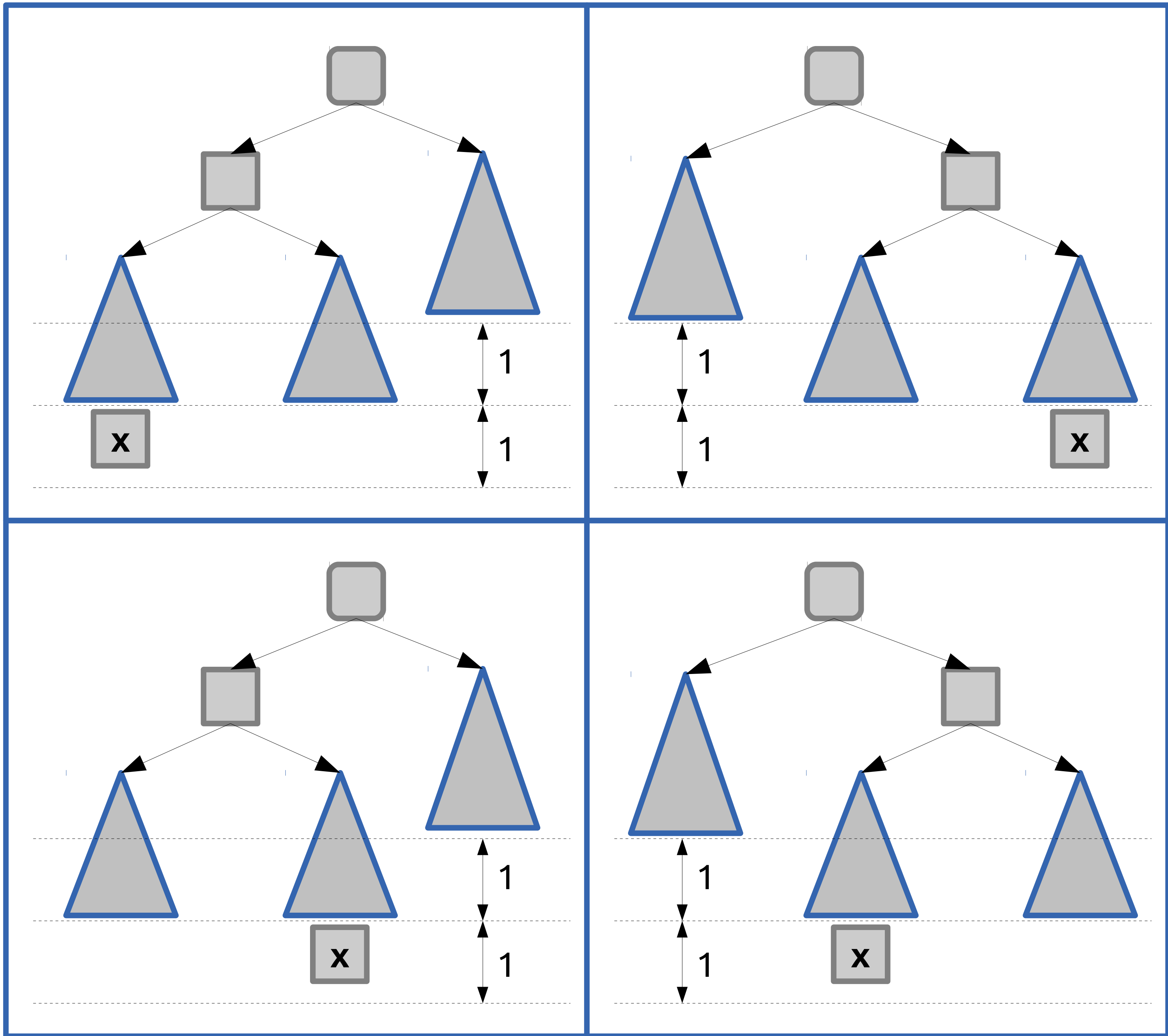
- Quando uma árvore fica desbalanceada?
  - Diferença se torna 2 ou -2
  - Isso só ocorre se a inserção aumentar a altura da subárvore que já era mais alta!



Casos que precisamos analisar:



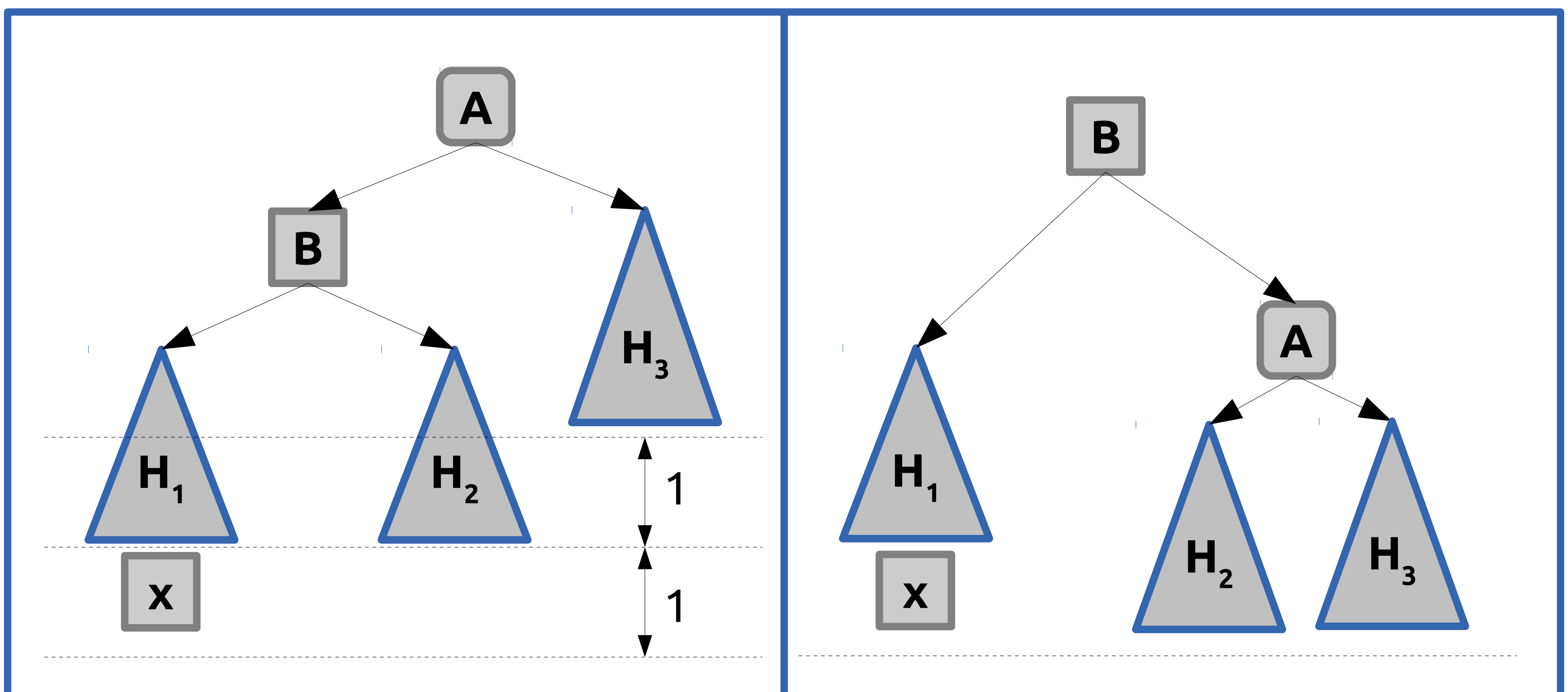
Casos que precisamos analisar:



AVL - Inserção

- Caso ***EE***:

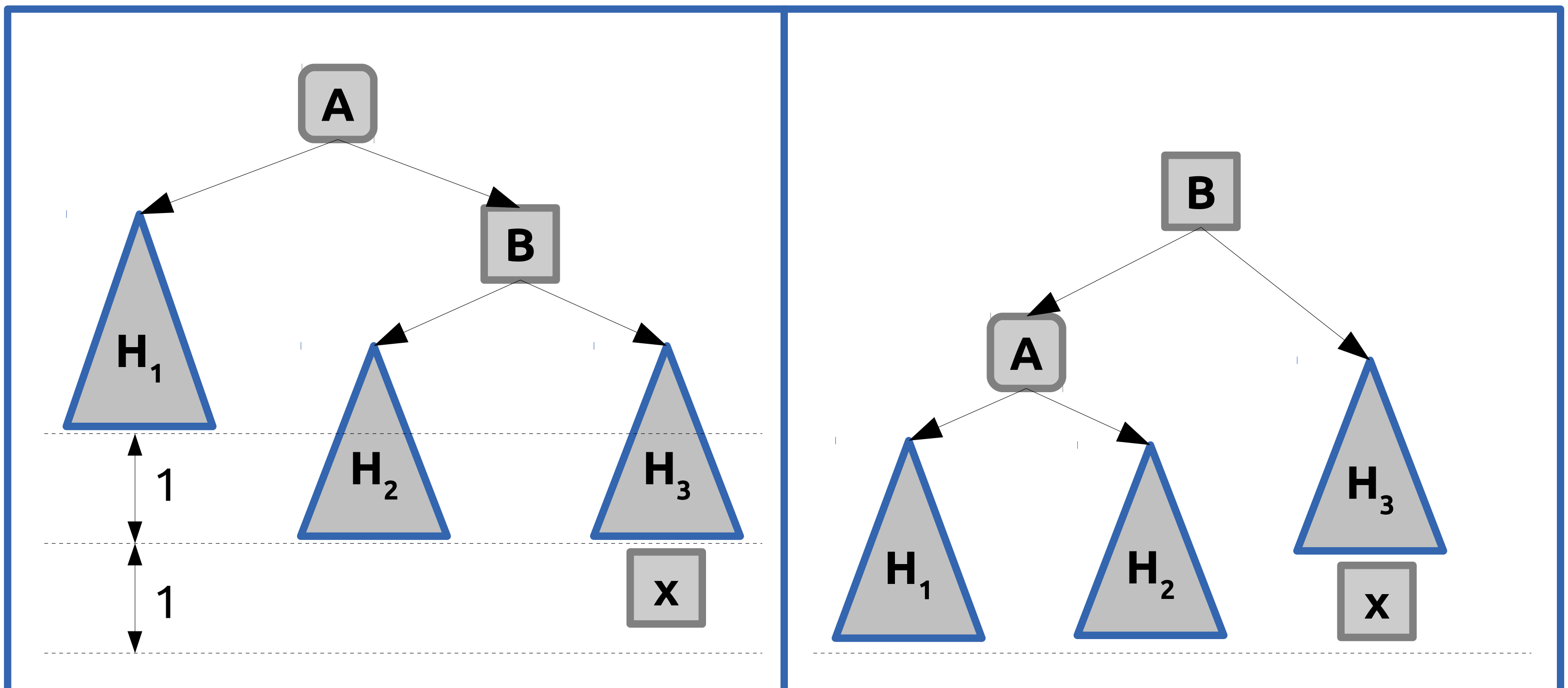
```
struct node* rotEE( struct node* A ) {
    struct node* B = A->esquerda;
    A->esquerda = B->direita;
    B->direita = A;
    return B;
}
```



AVL - Inserção

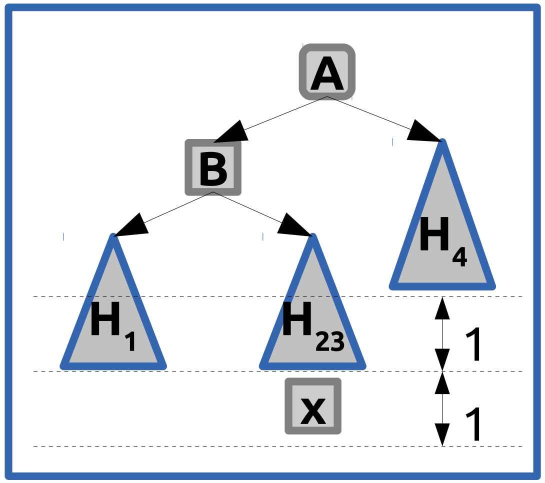
- Caso ***DD***:

```c
struct node* rotDD( struct node* A ) {
    struct node* B = A->direita;
    A->direita = B->esquerda;
    B->esquerda = A;
    return B;
}
```

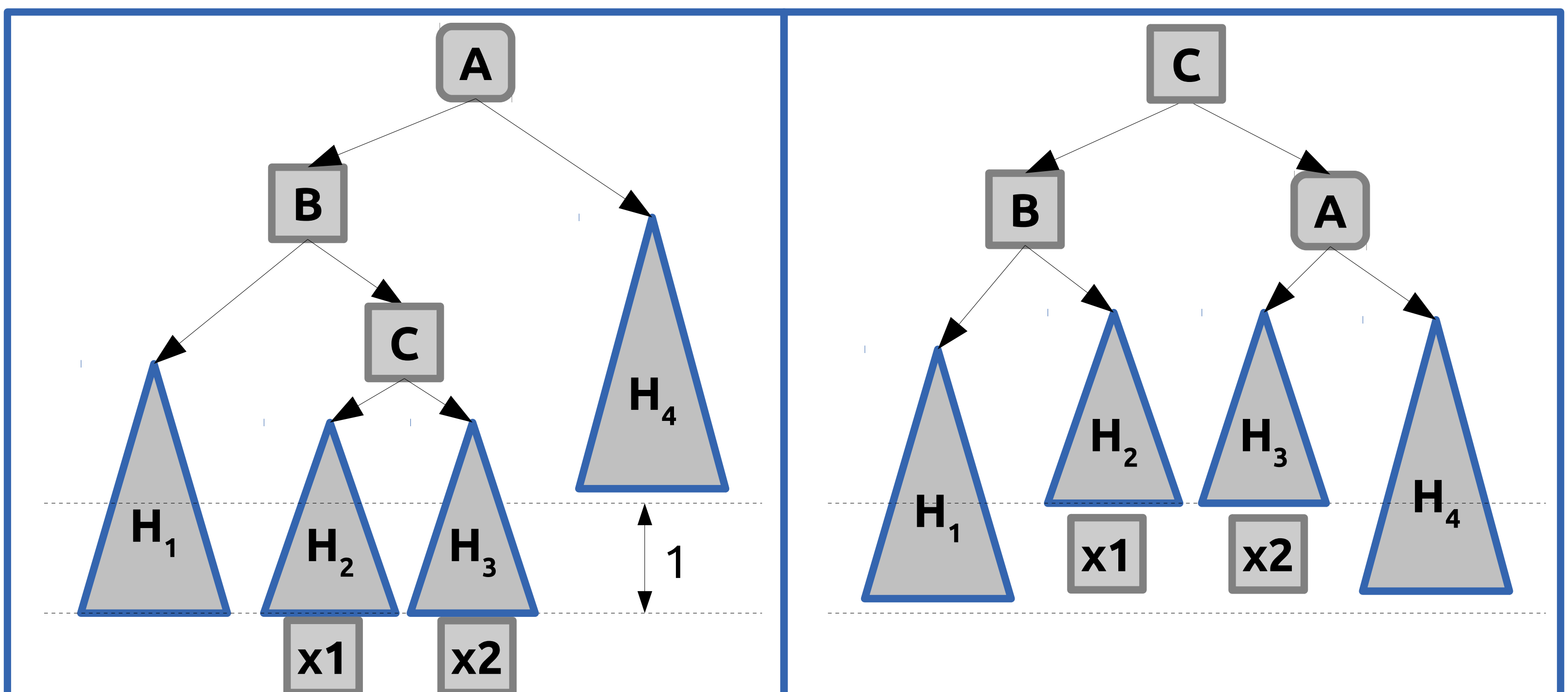


AVL - Inserção

- Caso *ED*:




```c
struct node* rotED( struct node* A ) {
    struct node* B = A->esquerda;
    struct node* C = B->direita;
    B->direita = C->esquerda;
    C->esquerda = B;
    A->esquerda = C->direita;
    C->direita = A;
    return C;
}
```

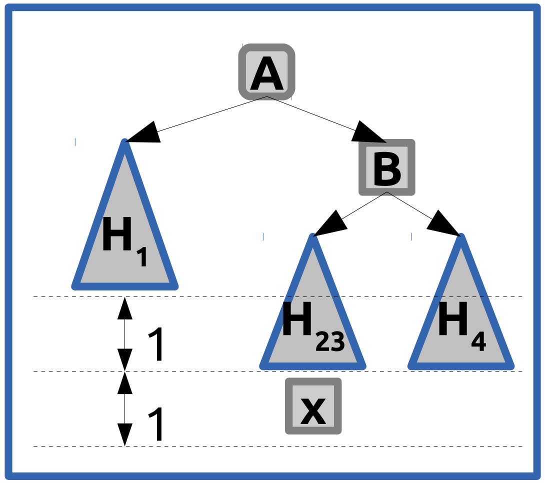


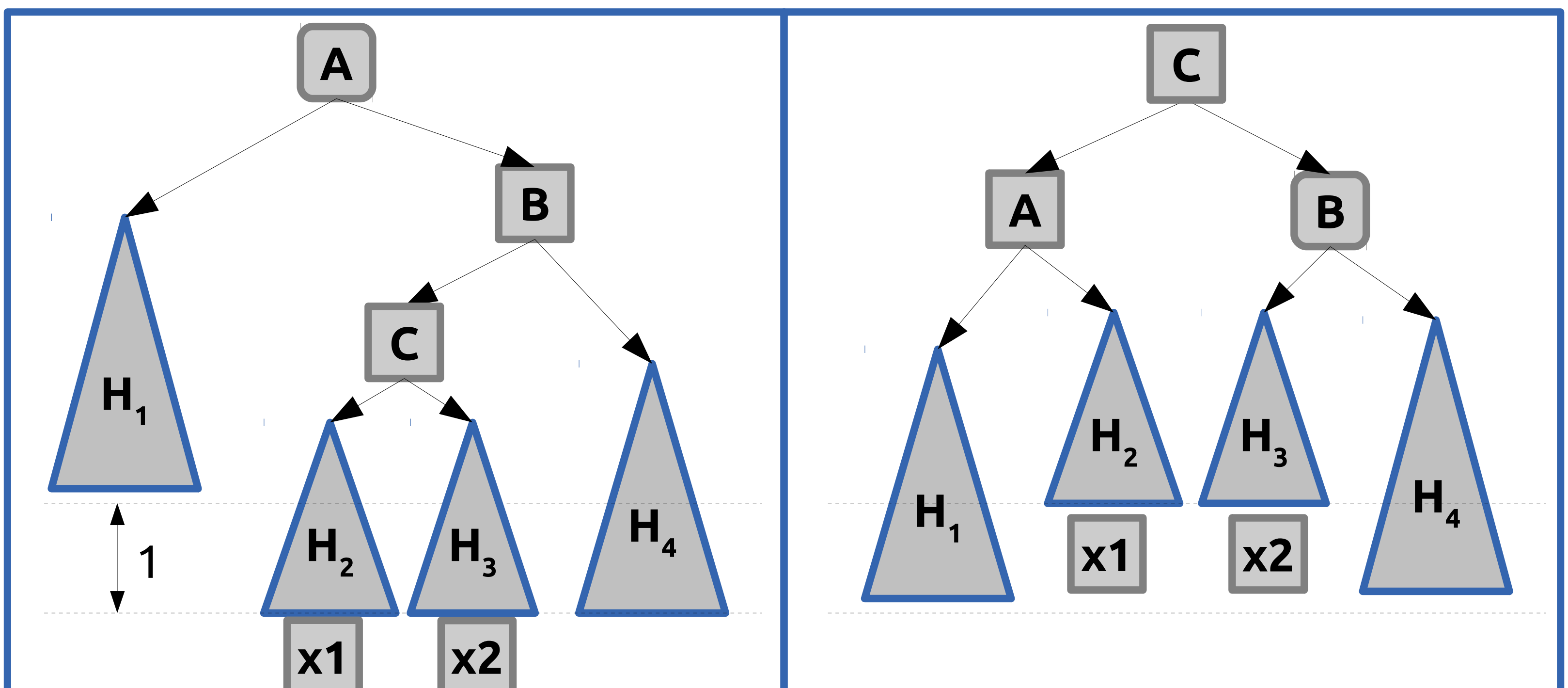
x1 ou x2

AVL - Inserção

- Caso ***DE***:

```
struct node* rotDE( struct node* A ) {
    struct node* B = A->direita;
    struct node* C = B->esquerda;
    B->esquerda = C->direita;
    C->direita = B;
    A->direita = C->esquerda;
    C->esquerda = A;
    return C;
}
```





x1 ou x2

AVL - Inserção

- Calcular a altura da subárvore em todo nó é muito custoso
- Logo, precisamos utilizar um novo campo em cada nó
- Precisamos apenas a diferença (e não o valor da altura)
  - Temos assim apenas 3 valores possíveis (2 *bits*)
- Com esta diferença conseguimos diferenciar cada um dos casos!
- Na volta da recursão a altura é verificada:
  - Ou a altura é mantida, ou a altura aumenta
  - Pela recursão sabemos qual lado aumentou/manteve
  - Quando a diferença entre as alturas for -2 ou 2, detectamos um desbalanceamento

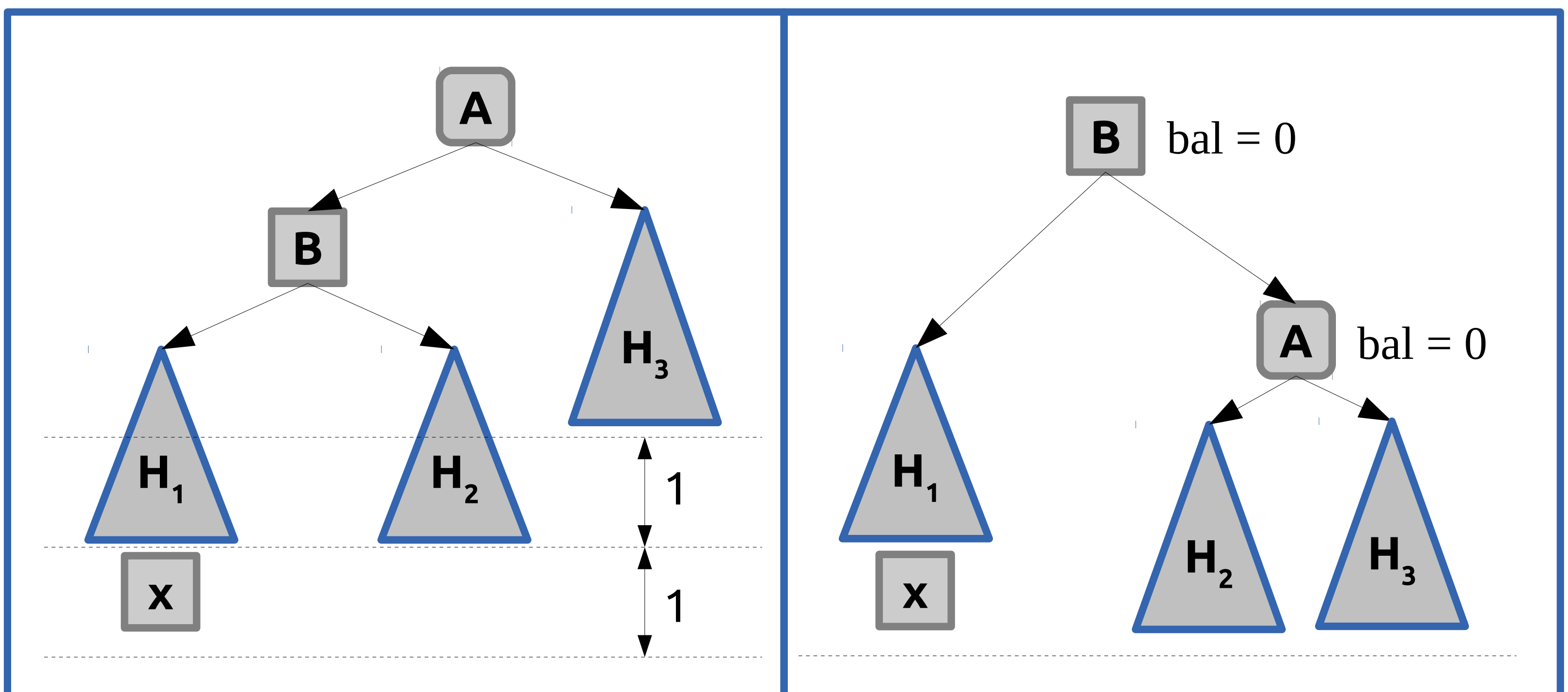
AVL - Inserção

- Vamos atualizar o campo do balanceamento em cada caso
  - Rotações ***EE***, ***DD***, ***ED*** e ***DE***
  - Volta da recursão
- Precisamos detectar quando a altura da árvore muda e quando não muda
- Incluir o algoritmo de busca

AVL - Inserção

- Caso *EE*:

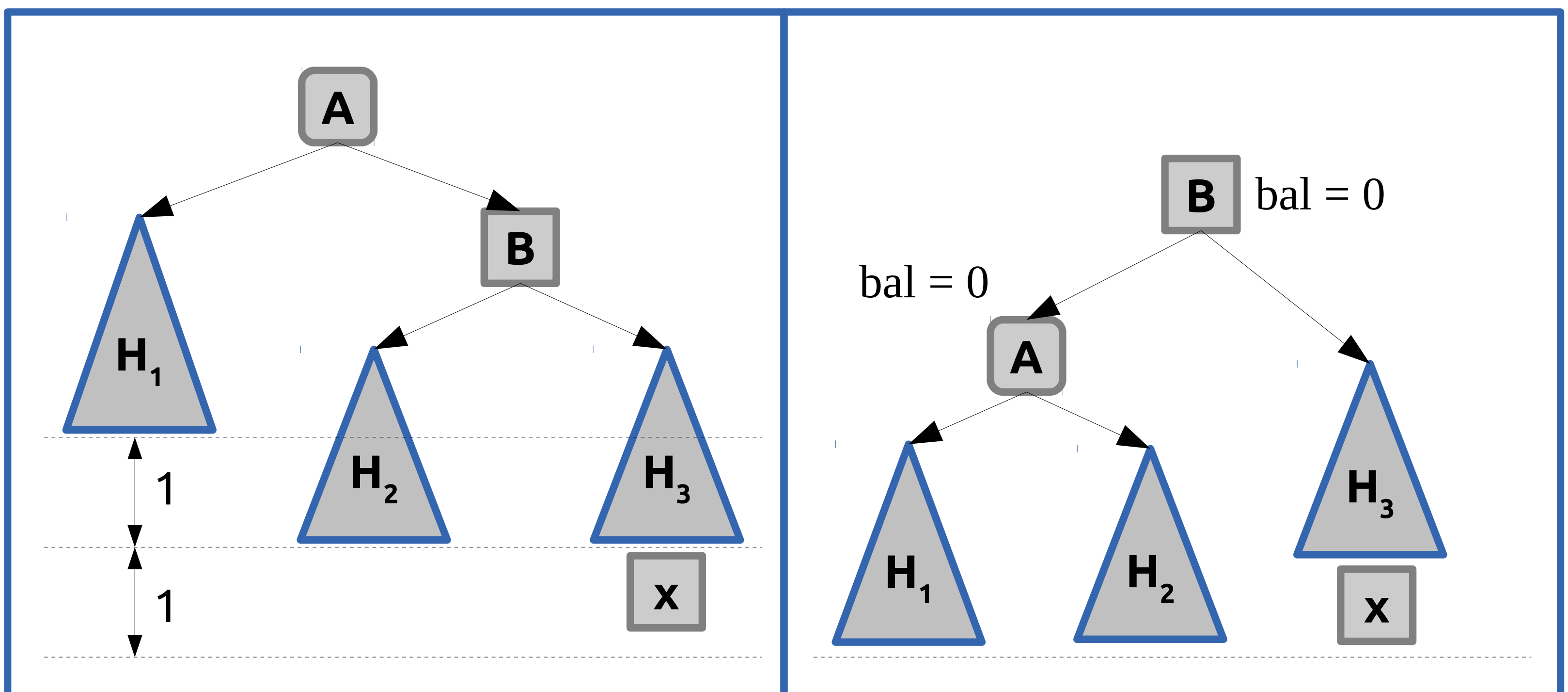
```
struct node* rotEE( struct node* A ) {
    struct node* B = A->esquerda;
    A->esquerda = B->direita;
    B->direita = A;
    A->bal = 0;
    B->bal = 0;
    return B;
}
```



AVL - Inserção

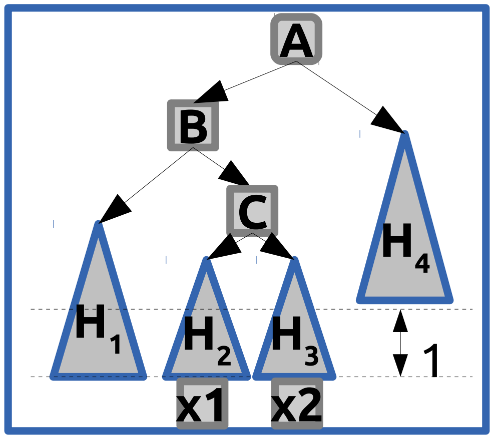
- Caso ***DD***:

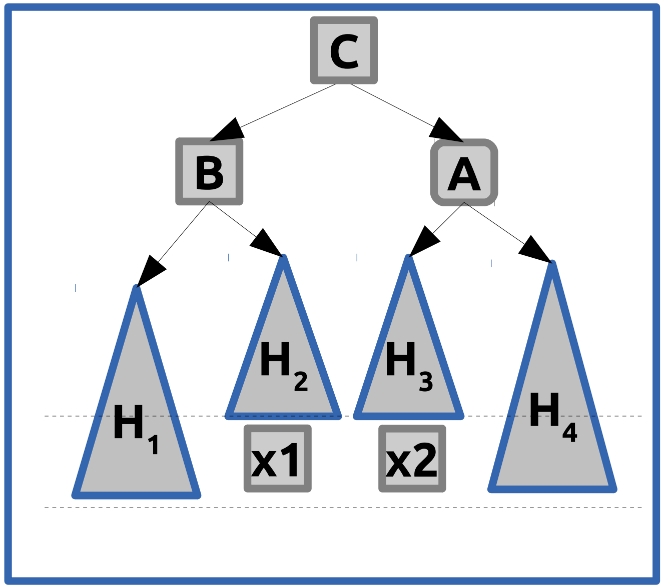
```
struct node* rotDD( struct node* A ) {
    struct node* B = A->direita;
    A->direita = B->esquerda;
    B->esquerda = A;
    A->bal = 0;
    B->bal = 0;
    return B;
}
```



AVL - Inserção

- Caso ***ED***:





x1

x2

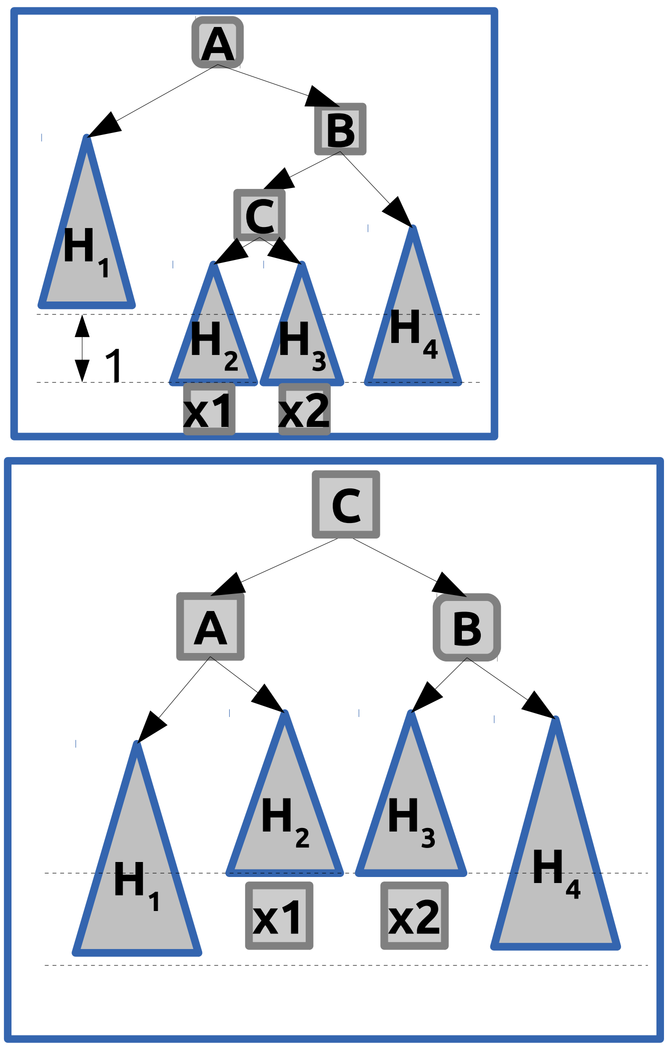
próprio C
(raiz)

```
struct node* rotED( struct node* A ) {
    struct node* B = A->esquerda;
    struct node* C = B->direita;
    B->direita = C->esquerda;
    C->esquerda = B;
    A->esquerda = C->direita;
    C->direita = A;
    if( C->bal == -1 ) {
        A->bal = 1;
        B->bal = 0;
        C->bal = 0;
    } else if( C->bal == 1 ) {
        A->bal = 0;
        B->bal = -1;
        C->bal = 0;
    } else { // C->bal == 0
        A->bal = 0;
        B->bal = 0;
    }
    return C;
}
```

x1 ou x2

AVL - Inserção

- Caso ***DE***:



x1

x2

próprio C
(raiz)

```
struct node* rotDE( struct node* A ) {
    struct node* B = A->direita;
    struct node* C = B->esquerda;
    B->esquerda = C->direita;
    C->direita = B;
    A->direita = C->esquerda;
    C->esquerda = A;
    if( C->bal == -1 ) {
        A->bal = 0;
        B->bal = 1;
        C->bal = 0;
    } else if( C->bal == 1 ) {
        A->bal = -1;
        B->bal = 0;
        C->bal = 0;
    } else { // C->bal == 0
        A->bal = 0;
        B->bal = 0;
    }
    return C;
}
```

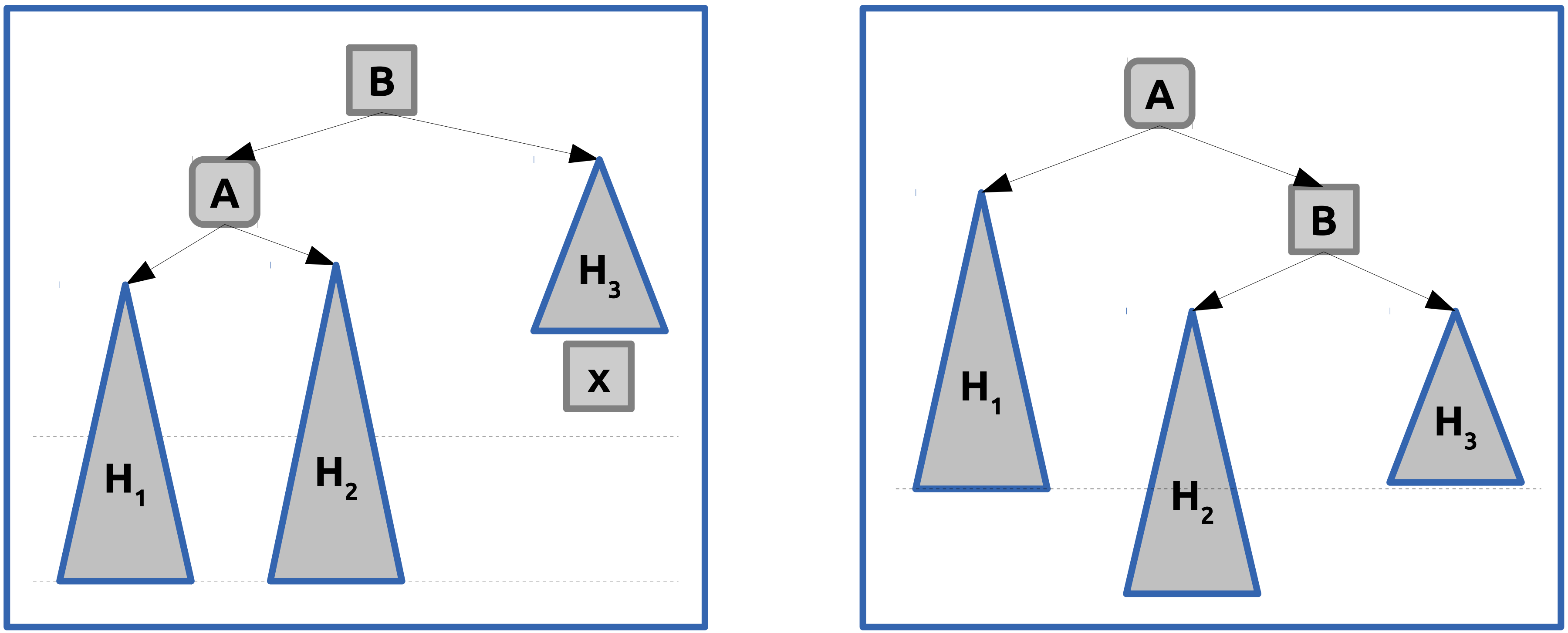
x1 ou x2

AVL - Remoção

- Passos:
  - Remover como em uma árvore BB;
  - Na “volta” da recursão verificamos nós que violam o balanceamento AVL;
  - Correção dos nós desbalanceados (só podem ser os ancestrais)
  - Diferenças com a inserção:
    - Uma situação a mais no ***EE*** e no ***DD***
    - Alguns casos alteram a altura da subárvore

AVL - Remoção

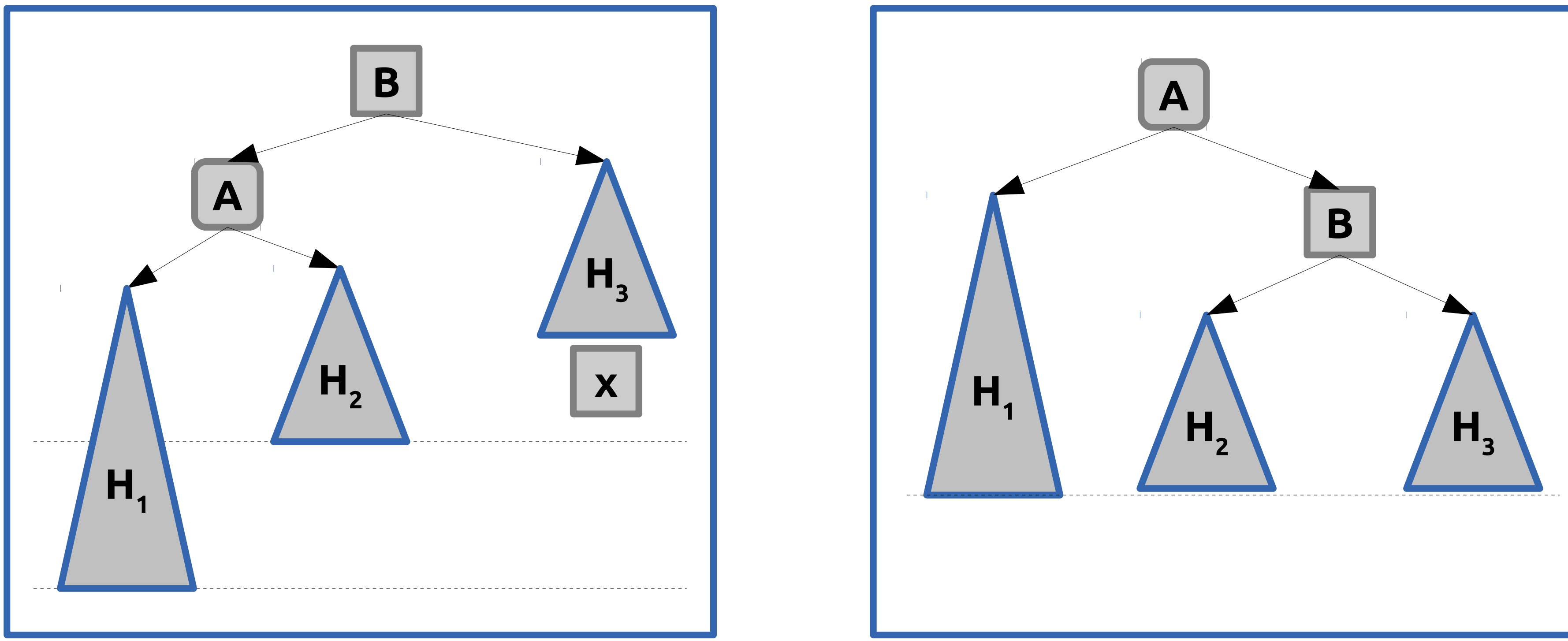
- Caso ***EE***:



mudouAltura = false

AVL - Remoção

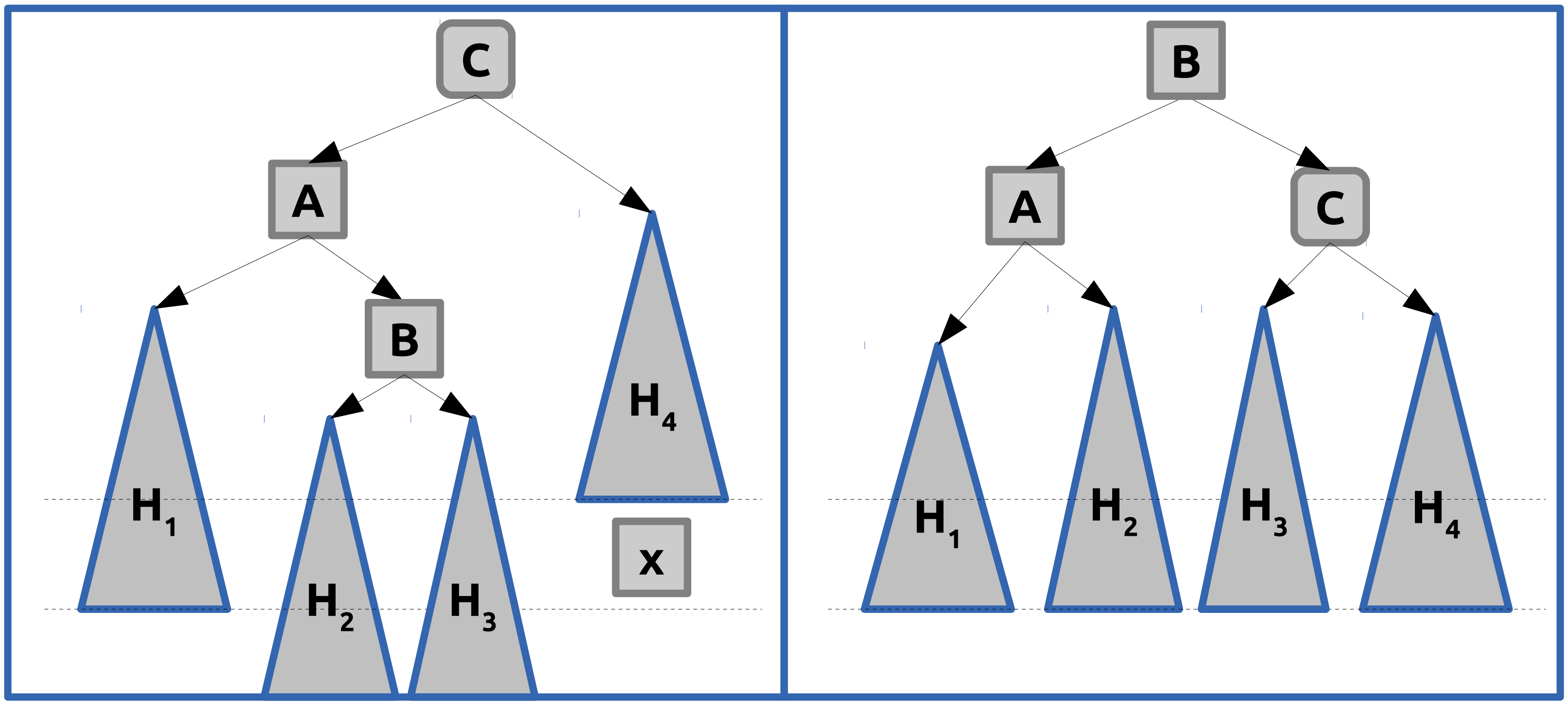
- Caso ***EE***:



mudouAltura = true

AVL - Remoção

- Caso *ED*:



mudouAltura = true

AVL - Remoção

- Casos:
  - ***DD*** e ***DE***

- Outras situações de configuração não alteram os casos

- ***MudouAltura*** não é sempre falso depois das rotações
  - Implicações?

AVL
Remoção

```c
struct node* rotEEremove(struct node* p, int *mudouAltura) {
    struct node* A = p->esquerda;
    p->esquerda = A->direita;
    A->direita = p;
    if(A->bal == 0) {
        A->bal = 1;
        p->bal = -1;
        *mudouAltura = 0;
    } else {
        A->bal = 0;
        p->bal = 0;
        mudouAltura = 1;
    }
    return A;
}
```

```c
struct node* rotEDremove(struct node* n, int *mudouAltura) {
    *mudouAltura = 1;
    return rotED(p);
}
```

AVL
Remoção

```c
struct node* rotDDremove(struct node* p, int *mudouAltura) {
    struct node *B = p->direita;
    p->direita = B->esquerda;
    B->esquerda = p;
    if(B->bal == 0) {
        B->bal = -1;
        p->bal = 1;
        *mudouAltura = 0;
    } else {
        B->bal = 0;
        p->bal = 0;
        *mudouAltura = 1;
    }
    return B;
}
```

```c
struct node* rotDEremove(struct node* p, int *mudouAltura) {
    *mudouAltura = 0;
    return rotDE(p);
}
```

Referências:

- Livro do Cormen
- Livro do Goodrich
- Livro do Robert Sedgewick
  - https://algs4.cs.princeton.edu