

-
-
-
-
-
-
-

Árvores binárias de pesquisa (mais): AVL, Vermelho-Preto, Splay

Algoritmos e Estruturas de Dados

2020/2021



FEUP

-
-
-
-
-
-
-
-

Árvores AVL

- Árvore de pesquisa binária
 - árvores podem ser desequilibradas
 - operações de inserção e eliminação de elementos são de complexidade linear no pior caso, quando árvore degenera em lista
- Árvores equilibradas
 - a diferença das alturas das sub-árvores de cada nó não pode exceder 1
 - evitam casos degenerados
 - garantem $O(\log N)$ para operações de inserção, remoção e pesquisa
- Árvores AVL
 - árvores de pesquisa binária
 - árvores equilibradas

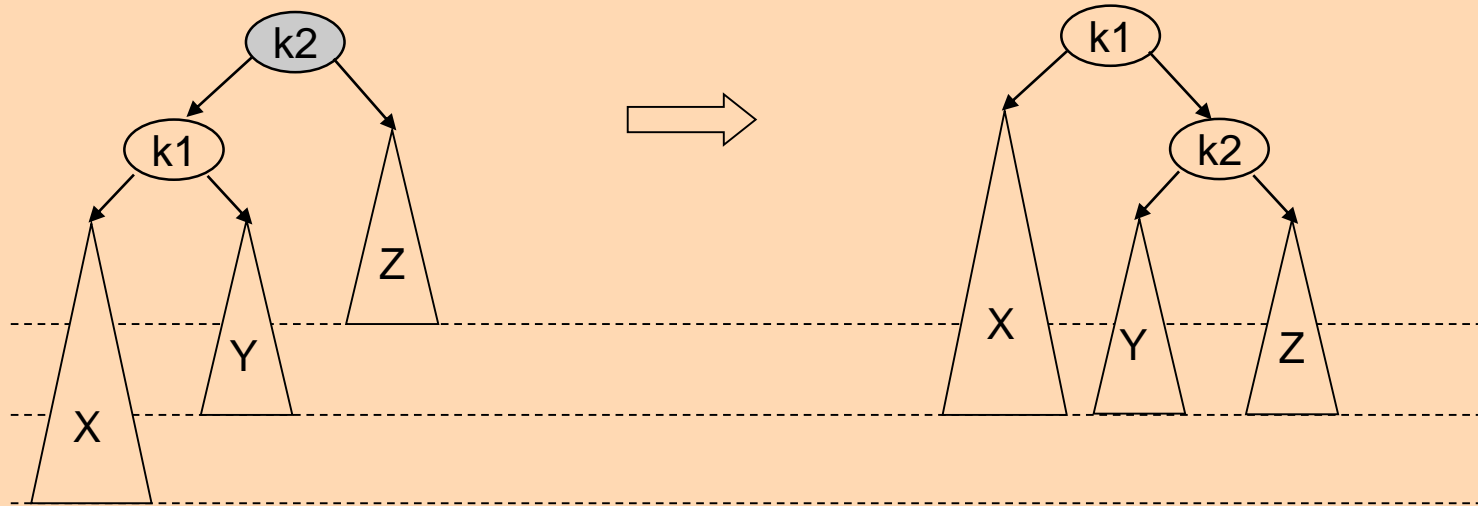
Árvores AVL

- Inserção de um elemento
 - inserção pode destruir o equilíbrio de alguns nós da árvore
 - após uma inserção, só os nós no caminho da raiz ao ponto de inserção podem ter a condição de equilíbrio alterada.
 - É necessário reequilibrar
 - reequilibrar o nó mais profundo onde surge desequilíbrio
 - toda a árvore resulta equilibrada
 - Seja ***K*** o nó a reequilibrar devido a inserção em:
 1. árvore esquerda do filho esquerdo de ***K***
 2. árvore direita do filho esquerdo de ***K***
 3. árvore esquerda do filho direito de ***K***
 4. árvore direita do filho direito de ***K***
 - casos 1 e 4 são simétricos; casos 2 e 3 são simétricos



Árvores AVL

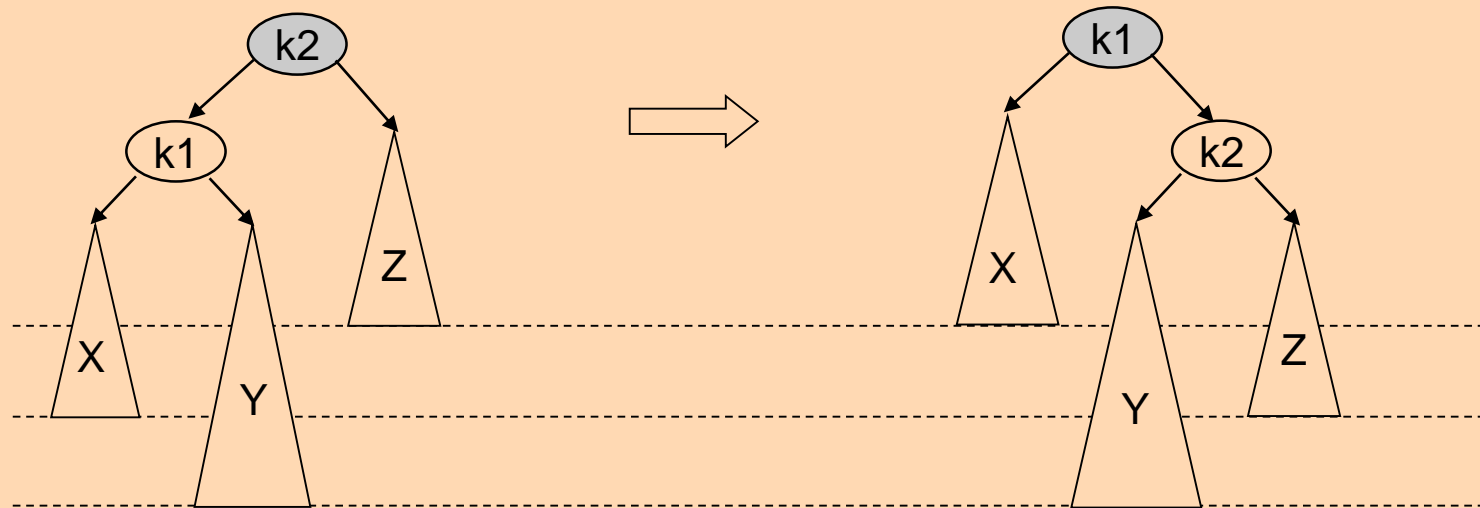
- Rotação simples (para os casos 1 e 4)



- *antes da rotação:*
 - **k2** é nó mais profundo onde falha o equilíbrio
 - sub-árvore esquerda está 2 níveis abaixo da direita
- *depois:*
 - **k1** e **k2** passam a ter sub-árvores da mesma altura
 - problema fica resolvido com uma só operação

Árvores AVL

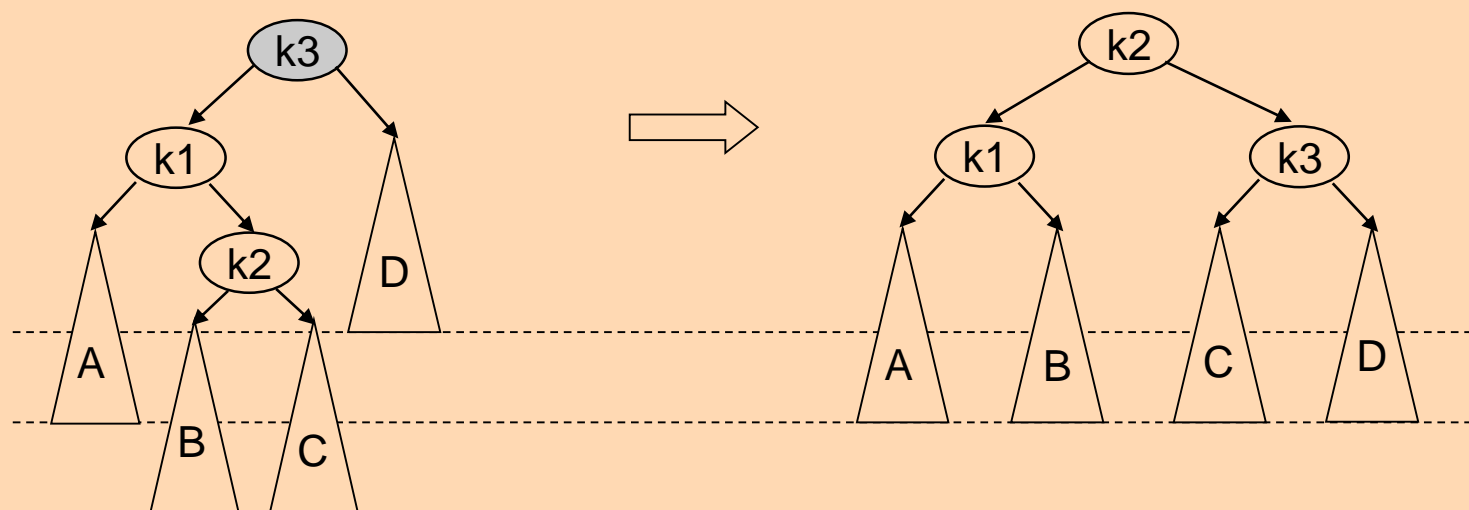
- Rotação simples não resolve os casos 2 e 3



- *antes da rotação:*
 - sub-árvore Y está a 2 níveis de diferença de Z
- *depois:*
 - sub-árvore Y está a 2 níveis de diferença de X

Árvores AVL

- Rotação dupla (para os casos 2 e 3)



- *antes da rotação:*
 - uma (e só uma) das sub-árvores B ou C está a 2 níveis de diferença de D
- Rotação dupla pode ser vista como uma sequência de 2 rotações simples
 - rotação entre o filho e o neto do nó
 - rotação entre o nó e o seu novo filho

Árvores AVL

- *Nó* da árvore AVL

```
template <class Comparable>
class AVLNode {
    Comparable element;
    AVLNode *left, *right;
    int height;
public:
    AVLNode(const Comparable &e, AVLNode *esq = 0,
            AVLNode *dir = 0, int h = 0): element(e),
            left(esq), right(dir), height(h) {}
    friend class AVLTree<Comparable>;
};
```

```
template <class Comparabe>
int AVLTree<Comparable>::height (AVLNode<Comparable> *t) const
{
    return t==NULL ? -1 : t->height;
}
```



Árvores AVL: implementação

- classe **AVLTree** : inserção

```
template <class Comparable>
void AVLTree<Comparable>:: insert(const Comparable & x,
                                   AVLNode<Comparable> * & t)
{
    if ( t == NULL)
        t = new AVLNode<Comparable>(x, NULL, NULL);
    else if ( x < t->element )
    {
        insert(x, t->left);
        if ( height(t->left) - height(t->right) == 2 )
            if ( x < t->left->element )
                rotateWithLeftChild(t);
            else
                doubleWithLeftChild(t);
    }
    // continua
```



Árvores AVL: implementação

- classe **AVLTree** : inserção

```
// continuação
else if ( t->element < x )
{
    insert(x, t->right);
    if ( height(t->right) - height(t->left) == 2 )
        if ( t->right->element < x )
            rotateWithRightChild(t);
        else
            doubleWithRightChild(t);
}
else
    ; // nó repetido, não fazer nada
t->height = max ( height(t->left), height(t->right) ) +1;
}
```

Árvores AVL: implementação

- classe **AVLTree** : rotação

```
template <class Comparable>
void AVLTree<Comparable>::
rotateWithLeftChild(AVLNode<Comparable> * & k2)
{
    AVLNode<Comparable> *k1 = k2->left;
    k2->left = k1->right;
    k1->right = k2;
    k2->height = max ( height(k2->left), height(k2->right) ) + 1;
    k1->height = max ( height(k1->left), height(k1->right) ) + 1;
    k2 = k1;
}
```

Árvores AVL: implementação

- classe **AVLTree** : rotação

```
template <class Comparable>
void AVLTree<Comparable>::
rotateWithRightChild(AVLNode<Comparable> * & k2)
{
    AVLNode<Comparable> *k1 = k2->right;
    k2->right = k1->left;
    k1->left = k2;
    k2->height = max ( height(k2->left), height(k2->right) ) + 1;
    k1->height = max ( height(k1->left), height(k1->right) ) + 1;
    k2 = k1;
}
```

Árvores AVL: implementação

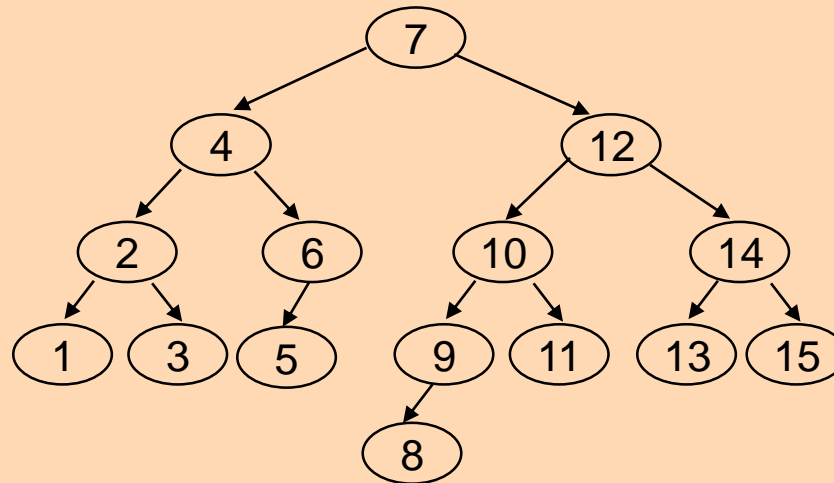
- classe **AVLTree** : rotação dupla

```
template <class Comparable>
void AVLTree<Comparable>::
doubleWithLeftChild(AVLNode<Comparable> * & k)
{
    rotateWithRightChild(k->left);
    rotateWithLeftChild(k);
}
```

```
template <class Comparable>
void AVLTree<Comparable>::
doubleWithRightChild(AVLNode<Comparable> * & k)
{
    rotateWithLeftChild(k->right);
    rotateWithRightChild(k);
}
```

Árvores AVL

- Construir a árvore AVL que resulta da inserção da seguinte sequência de valores:
 - 1, 2, 3, 4, 5, 6, 7, 15, 14, 13, 12, 11, 10, 9, 8



- Remoção de um elemento
 - remoção pode destruir o equilíbrio de alguns nós da árvore
 - reequilibrar a árvore (como no caso da inserção)

Árvores AVL

- classe **AVLTree** : remoção

```
template <class Comparable>
void AVLTree<Comparable>::
remove(const Comparable & x, AVLNode<Comparable> * & t)
{
    if ( t == NULL ) return; // não existe
    if ( x < t->element ) {
        remove(x, t->left);
        if ( height(t->right) - height(t->left) == 2 )
            if ( height(t->right->left) <= height(t->right->right) )
                rotateWithRightChild(t);
            else
                doubleWithRightChild(t);
    }
    // continua
```

Árvores AVL

- classe **AVLTree** : remoção

```
// continuação
else if ( t->element < x ) {
    remove(x, t->right);
    if ( height(t->left) - height(t->right) == 2 )
        if ( height(t->left->right) <= height(t->left->left) )
            rotateWithLeftChild(t);
        else
            doubleWithLeftChild(t);
}
else if ( t->left == NULL || t->right == NULL ) {
    AVLNode<Comparable> * oldNode = t;
    t = ( t->left != NULL ) ? t->left : t->right;
    delete oldNode;
}
// continua
```

Árvores AVL

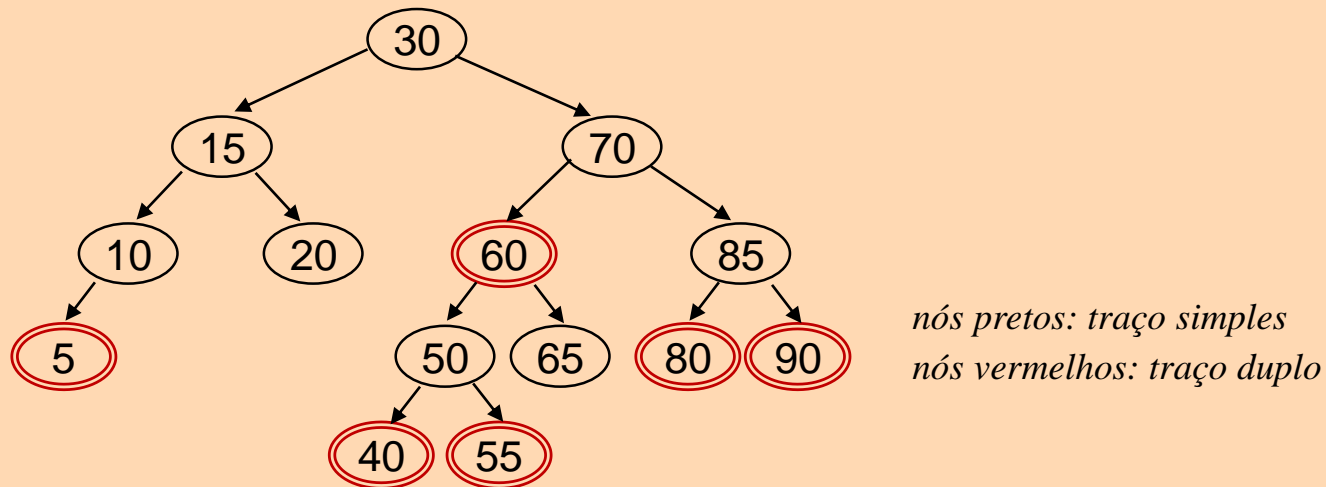
- classe **AVLTree** : remoção

```
// continuação
else {          // nó tem 2 filhos
    t->element = findMin(t->right)->element;
    remove(t->element, t->right);
    if ( height(t->left) - height(t->right) == 2 )
        if ( height(t->left->right) <= height(t->left->left) )
            rotateWithLeftChild(t);
        else
            doubleWithLeftChild(t);
}
if (t)
    t->height = max ( height(t->left), height(t->right) ) + 1;
}
```


Árvores VP

- Árvore Vermelho-Preto
 - alternativa à árvore AVL
 - operações possuem complexidade $O(\log N)$
- Propriedades de uma árvore VP
 1. cada nó é colorido como vermelho ou preto
 2. a raiz é preta
 3. se um nó é vermelho, os seus filhos são pretos
 4. qualquer caminho de um nó até uma subárvore vazia contém o mesmo número de nós pretos
 - Altura de uma árvore VP é no máximo $= 2 \times \log(N+1)$
 - garante que operação de pesquisa é de ordem logarítmica

Árvores VP



Operação mais complexa: inserção de um novo elemento

- o novo elemento é folha e é vermelho
- se pai é preto, terminar (ex: inserção do elemento 25)
- se pai é vermelho, corrigir a árvore (mudança de cor e/ou rotações)



Árvores VP

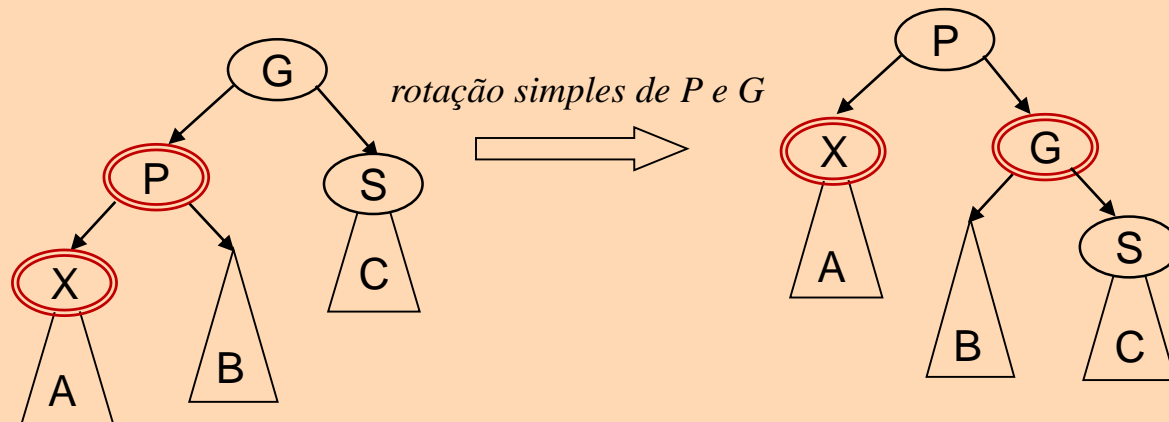
Inserção de um novo elemento X (quando pai é vermelho) **

- se irmão do pai (tio) é preto ou nulo: rotação simples ou dupla, seguida de mudança de cor

Rotação simples, quando:

- X é filho esquerdo e neto esquerdo, ou
- X é filho direito e neto direito

Mudança de cor: pai e antigo avô



Árvores VP

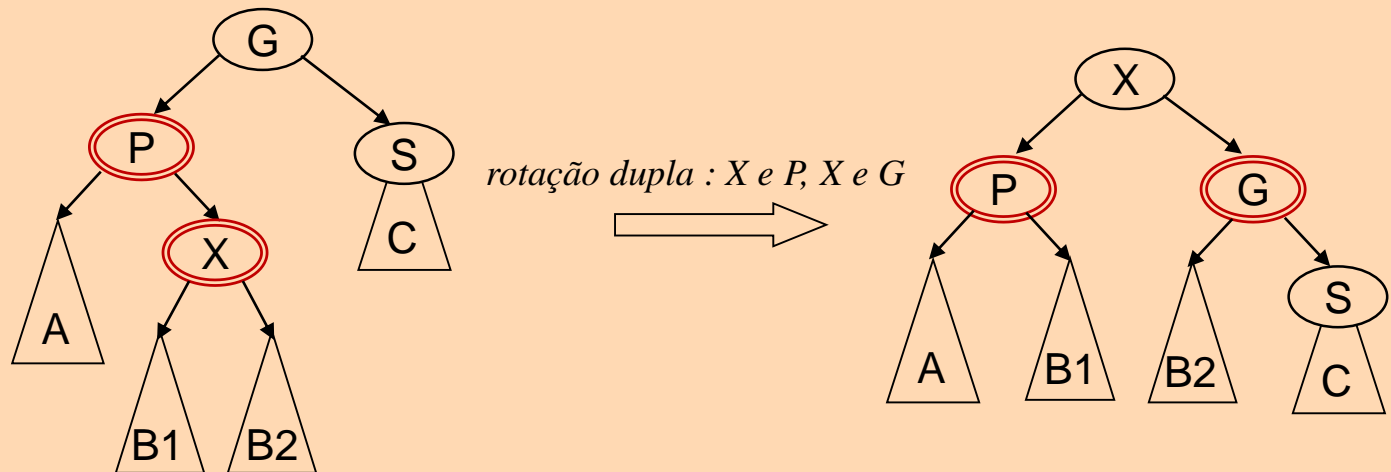
Inserção de um novo elemento X (quando pai é vermelho) **

- se irmão do pai (tio) é preto ou nulo: rotação simples ou dupla, seguida de mudança de cor

Rotação dupla, quando:

- se X é filho esquerdo e neto direito, ou
- se X é filho direito e neto esquerdo

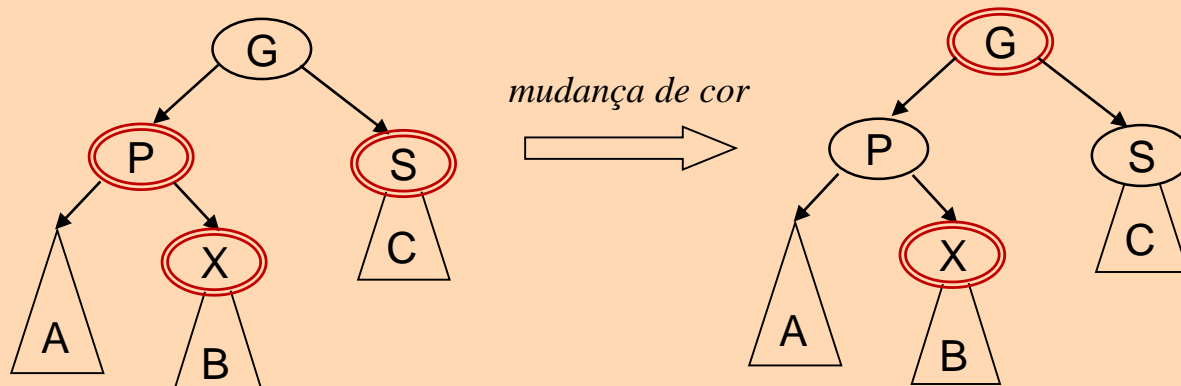
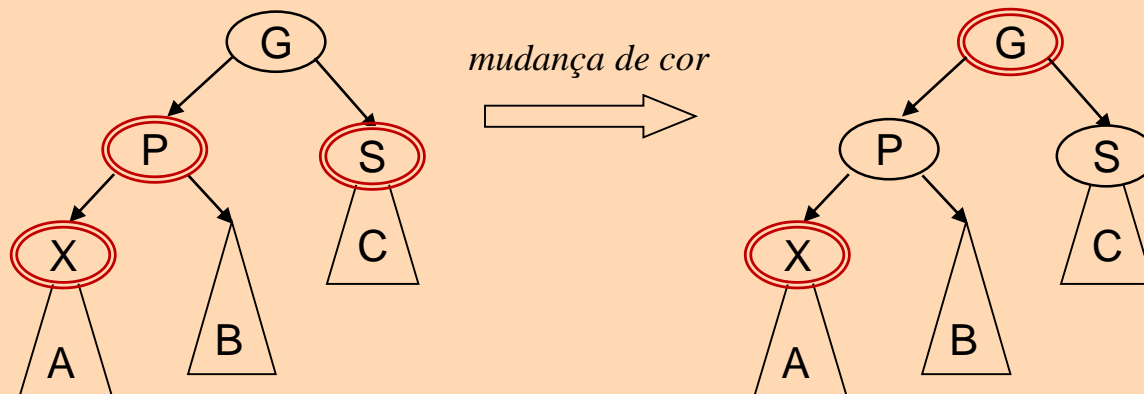
Mudança de cor: nó e antigo avô



Árvores VP

Inserção de um novo elemento X (quando pai é vermelho) **

- se irmão do pai é vermelho: mudança de cor
 - pai e tio passam para preto, e avô para vermelho

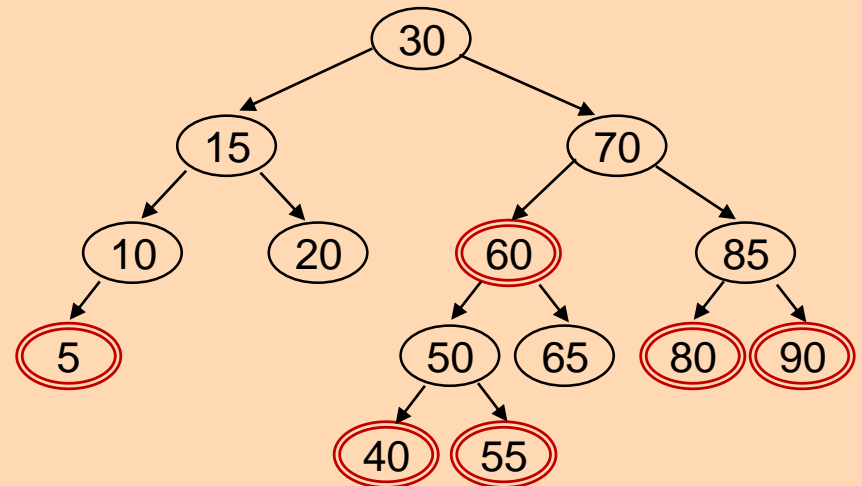


Árvores VP

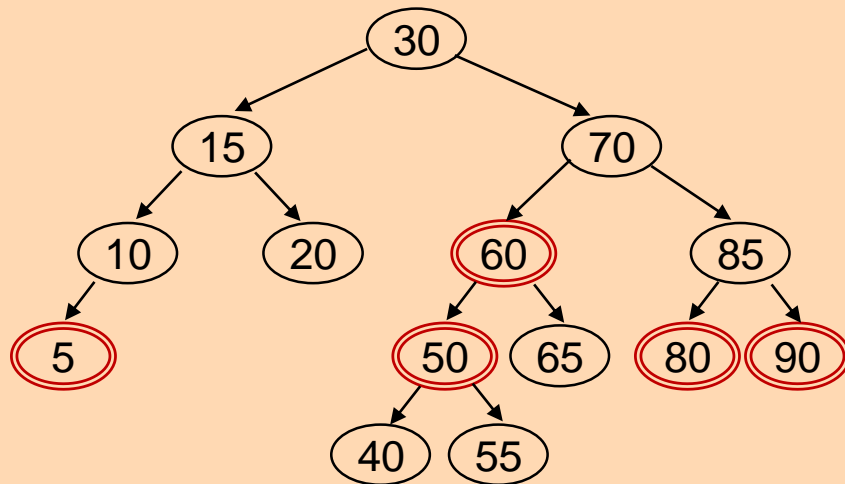
Inserção de um novo elemento (*Top-Down*) **

- novo elemento é vermelho, e é folha
- começar na raiz, comparando valor com os nós por onde passa para escolher a subárvore onde inserir o novo elemento
- quando um nó N tem dois filhos vermelhos
 - tornar N vermelho e os dois filhos pretos
 - se pai de N (P) é vermelho, ocorre violação da condição VP
 - neste caso, efectuar rotação simples ou dupla com mudança de cor

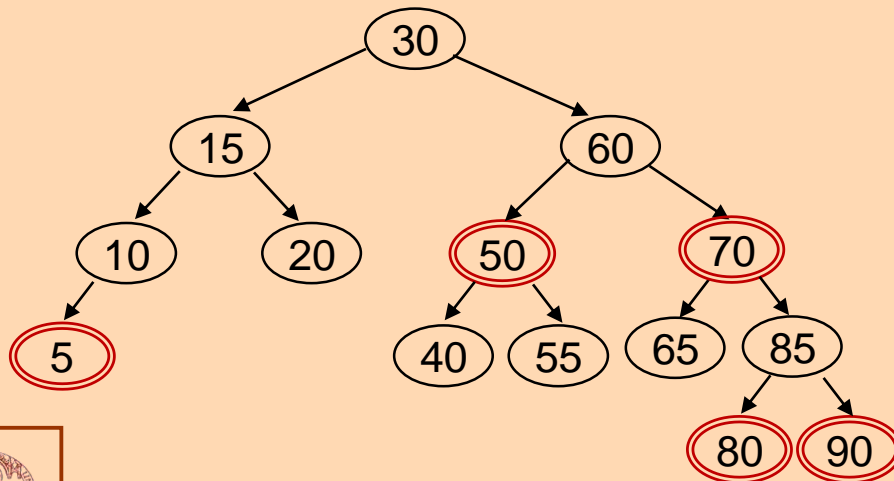
- ex: inserir valor 45
 - passa por 30, 70, 60,
 - nó 50 tem dois filhos V
 - então, efectuar mudança de cor



Árvores VP



- agora, 50 e 60 são ambos V (não pode ser)
- efectuar rotação simples entre 60 e 70 com mudança de cor



- continuar a procurar posição do novo elemento
- colocar 45 como folha V, e filho direito de 40
- como pai é preto, terminar

Árvores VP

Remoção de um elemento (*Top-Down*) **

- remoção é sempre realizada em uma folha
- se nó tem 2 filhos, ou apenas filho direito, substituir pelo menor da subárvore direita, e eliminar esse nó (tem no máximo 1 filho)
- se nó tem apenas filho esquerdo, substituir pelo maior da subárvore esquerda, e eliminar esse nó
- eliminação de uma folha vermelha, é trivial
- eliminação de uma folha preta, é mais complicado

Devemos garantir que folha a eliminar é vermelha !

- Solução: ao percorrer a árvore, manter o nó em análise como vermelho



Árvores VP

Remoção de um elemento (*Top-Down*) **

- Se raiz tem 2 filhos pretos, mudar raiz para vermelho, X é filho correspondente
- Senão, X é a raiz

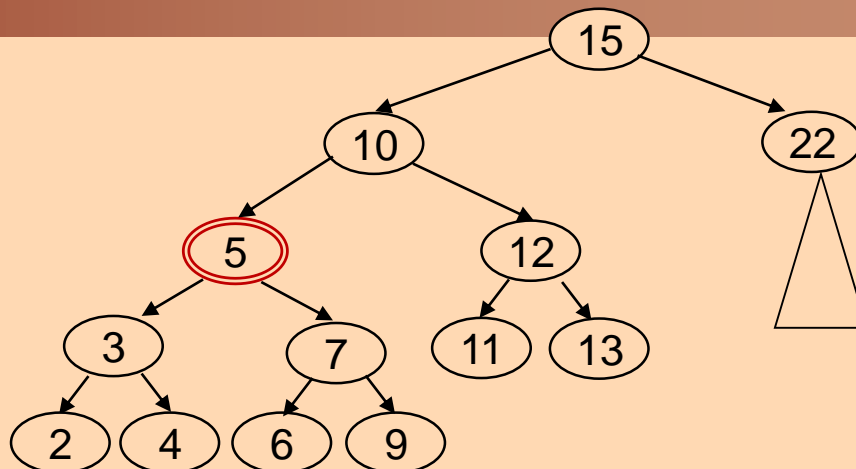
nota: X e irmão de X (Y) são pretos

- se X tem 2 filhos pretos:
 - se Y tem 2 filhos pretos, alterar as cores de X, Y, e o pai (P). Continuar
 - se Y tem pelo menos um filho vermelho (S), efetuar i) ou ii) conforme S e continuar:
 - i. rotação simples (Y, P), recolorir X, Y, P, S
 - ii. rotação dupla (S, Y, P), recolorir X, P
- se X tem pelo menos 1 filho vermelho, continuar na subárvore correspondente:
 - se novo X é o filho vermelho, continuar
 - se novo X é o filho preto, novo Y é vermelho, e novo P é preto: rotação simples de novo Y e novo P. Recolorir nós rodados.
- Quando encontrar X, remover
- Colocar a raiz na cor preto

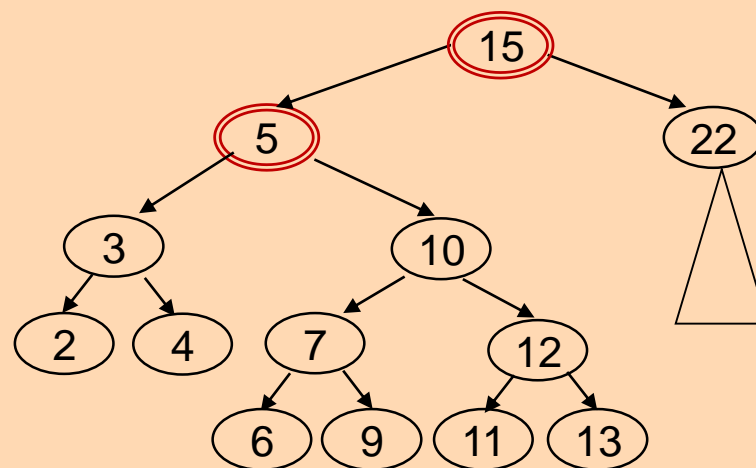
Árvores VP

**

- ex: remover 11 na árvore apresentada

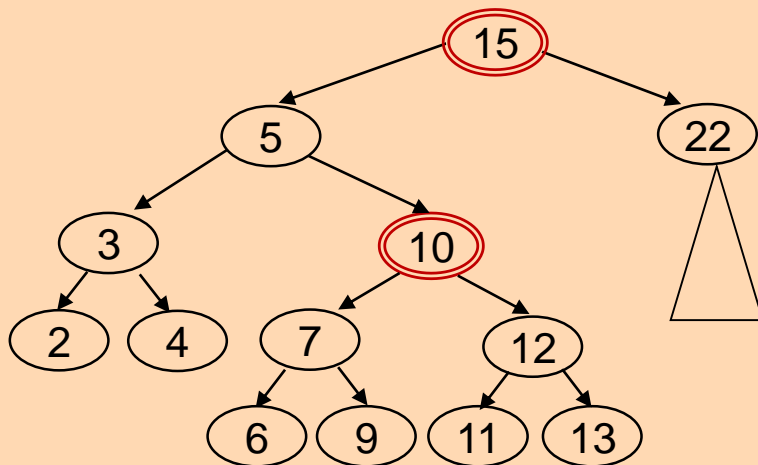


- mudar raiz para vermelho
- ir para X=10
- Tem 1 filho vermelho, ir para X=12
- X=12 é filho preto, rotação simples de Y=5 e P=10

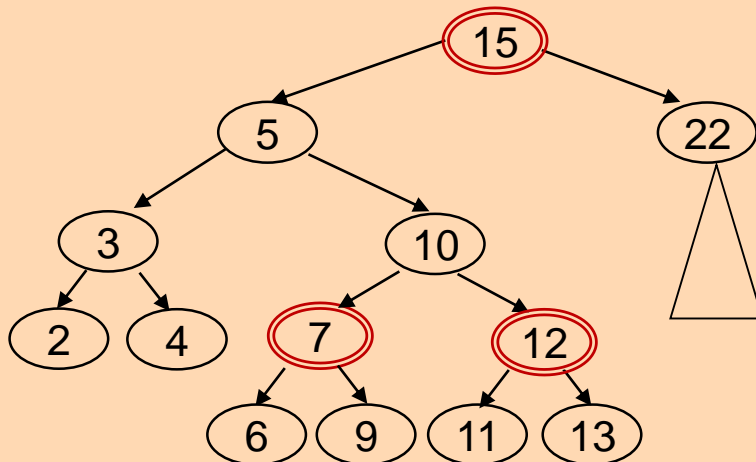


Árvores VP

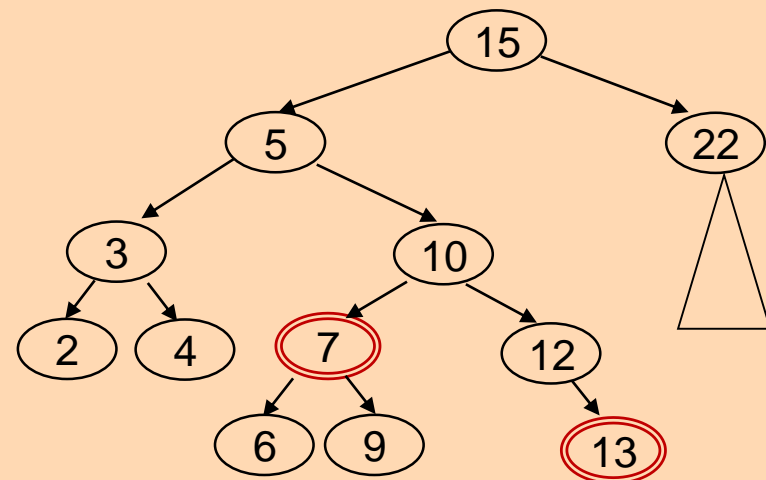
- recolorir nós rodados: 5 e 10 **



- X=12, tem 2 filhos pretos e Y=7 também: alterar cores de X, Y e P (12, 7 e 10)



- ir para X=11
- X=11 não tem filhos (equivalente a ter 2 filhos pretos), Y também: alterar cores de X, Y e P (11, 13 e 12)
- remover X=11 (que é vermelho)
- colorir a raiz a preto



Árvores VP (Standard Template Library - STL)

- class *set*
- Alguns métodos:
 - void **clear**();
 - std::pair<iterator,bool> **insert**(const value_type& value);
 - iterator **erase**(const_iterator position);
 - iterator **find**(const Key& key)
 - ...



Árvores Splay

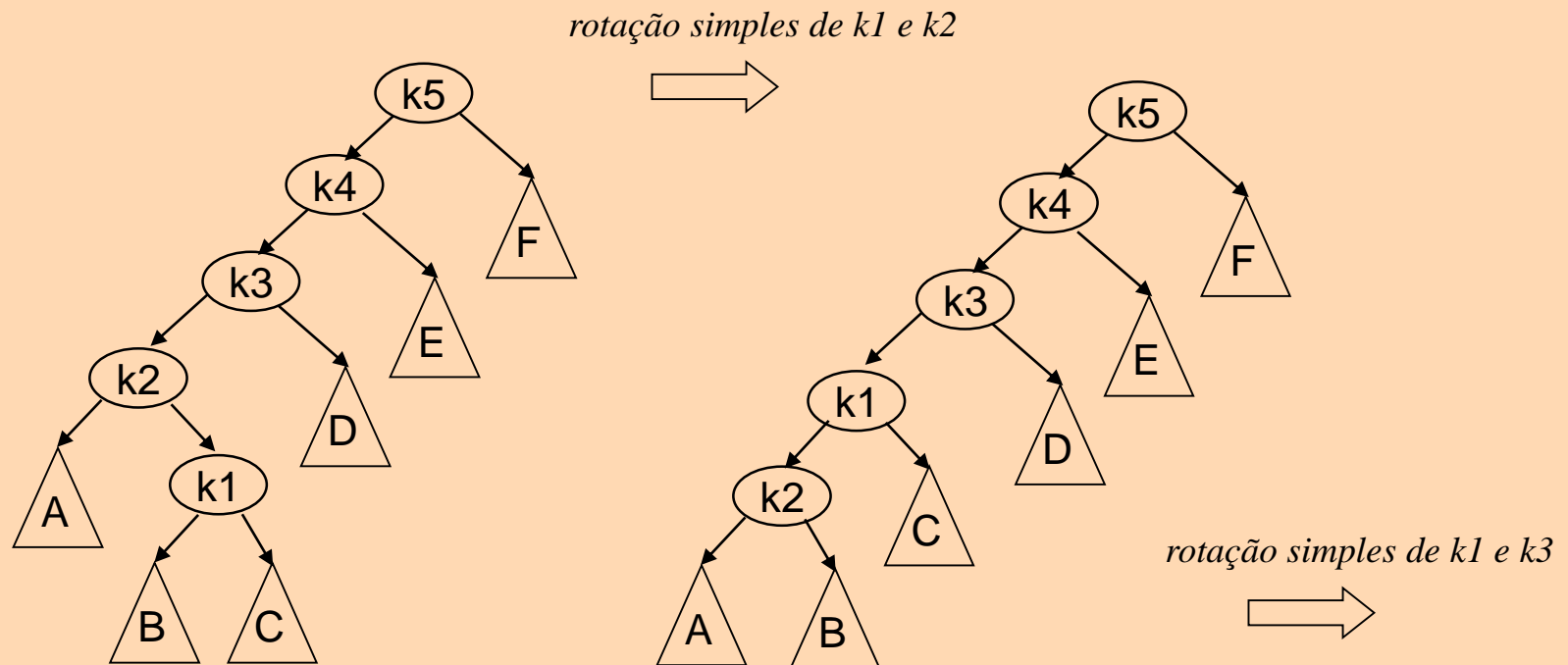
- Nas árvores AVL, as pesquisas frequentes a um mesmo elemento são penalizadas se este estiver a uma grande profundidade
 - em certas aplicações, quando um elemento é pesquisado uma vez, é muito provável que seja acedido de novo
 - seria bom que os elementos acedidos com frequência fossem “puxados” para a raiz da árvore
- Solução: *Árvores Splay*

Árvores Splay

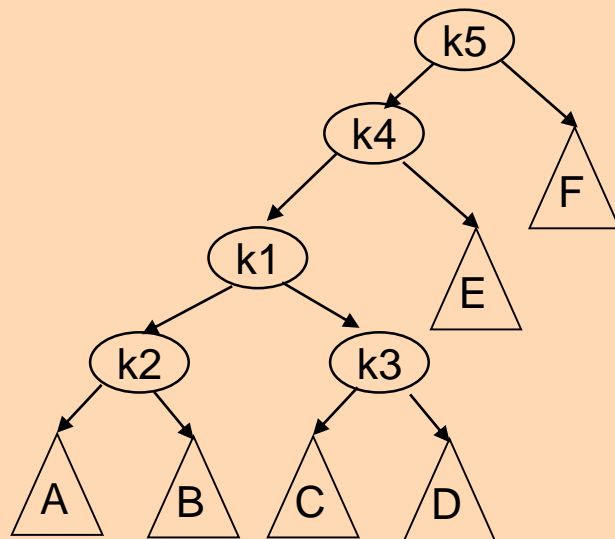
- Árvores mais simples que AVL
 - não força o equilíbrio
 - não mantém informação da altura
- Ajusta a estrutura da árvore à frequência de acesso aos dados
 - cada nó acedido é puxado para a raiz através de uma sequência de rotações
 - junto à raiz estão os elementos mais usados
 - os elementos mais inativos ficam mais “longe” da raiz
- ex: registos de doentes num hospital
 - podem estar no fundo da árvore, se os doentes não estiverem internados
 - passam para a raiz no momento do internamento
 - vão afundando se não voltarem a ser acedidos

Árvores Splay

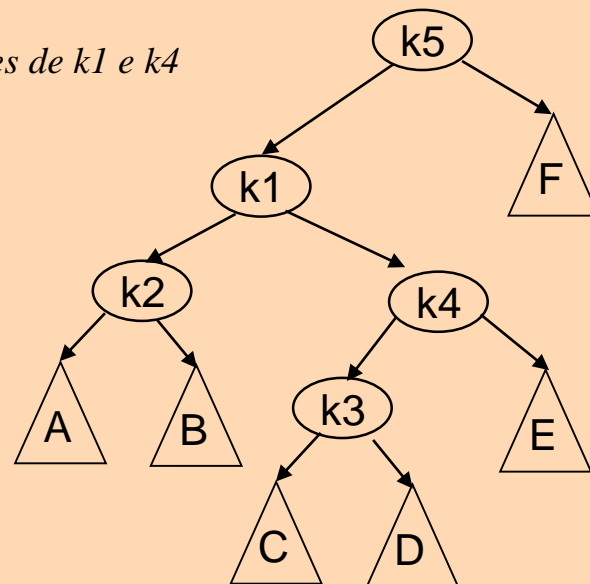
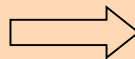
- Uma ideia simples (não resulta)
 - efectuar rotações simples
- *Exemplo: aceder ao nó k1*



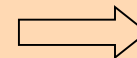
Árvores Splay



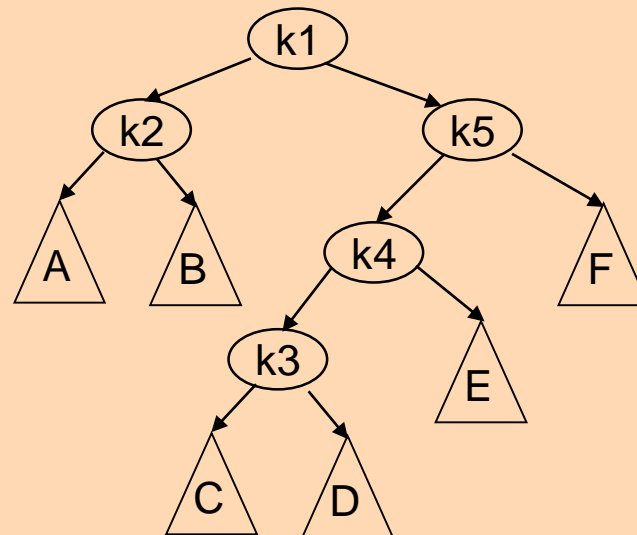
rotação simples de k1 e k4



rotação simples de k1 e k5



Árvores Splay



- O nó k3 está quase à mesma profundidade que k1 inicialmente
- uma visita a k3 seria também pesada, e afundaria outro nó
- **solução não serve**

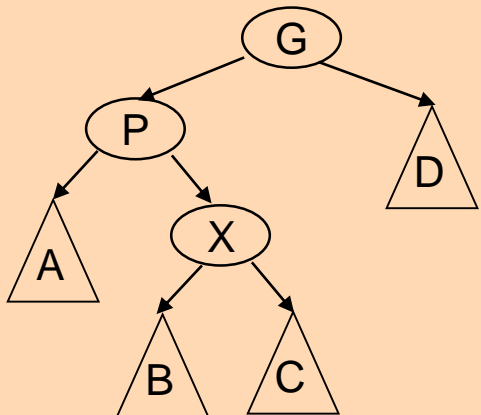
Árvores Splay

Splaying

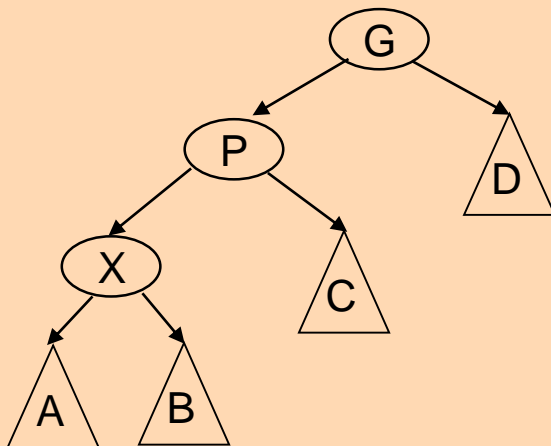
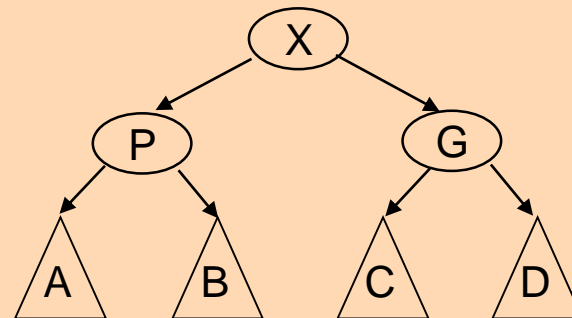
- Rotações ascendentes desde o nó acedido (X) até à raiz
- Se pai de X é raiz : rotação simples de X e raiz
- Senão, X possui um pai (P) e um avô (G)
 - X é filho direito (esquerdo) de P , e P é filho esquerdo (direito) de G : zig-zag
 - X é filho direito (esquerdo) de P , e P é filho direito (esquerdo) de G : zig-zig
 - zig-zag é uma rotação dupla AVL (duas rotações: 1ª rotação é de X e P ; 2ª rotação é de X e G)
 - zig-zig é específico do “splay” (duas rotações: 1ª rotação é de P e G ; 2ª rotação é de X e P)



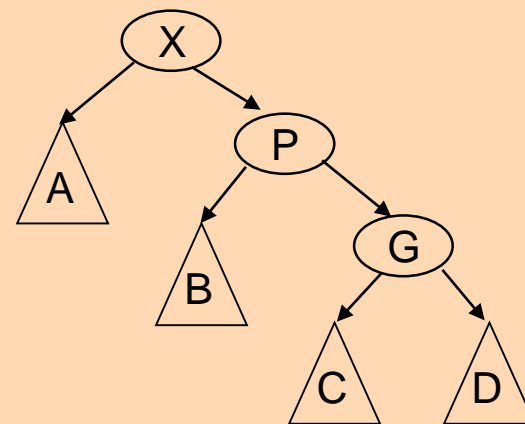
Árvores Splay



zig-zag



zig-zig

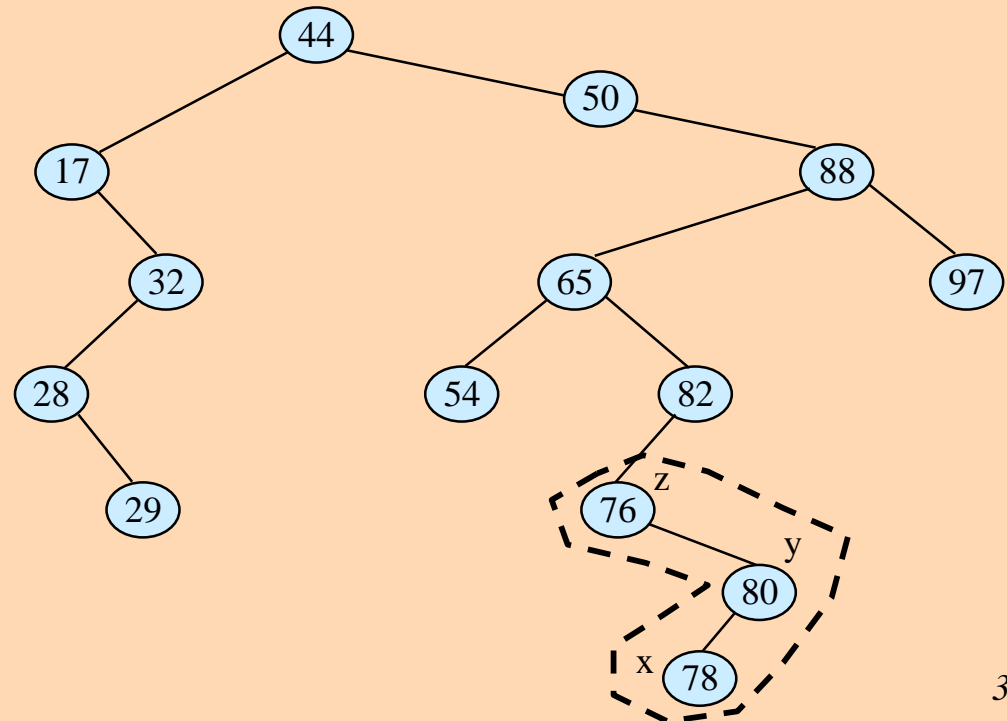


Árvores Splay

Inserção

- Inserir novo valor como na BST
- Se inserção com sucesso, efetuar operação “splaying” ao novo nó
- Se inserção com insucesso, efetuar operação “splaying” ao nó que provocou insucesso

Ex: inserir(78)



Árvores Splay

Remoção

- Remover valor como na BST
- Se remoção com sucesso (e não é raiz), efetuar operação “splaying” ao pai do nó removido
- Se remoção com insucesso, efetuar operação “splaying” ao último nó acedido

Pesquisa

- Se pesquisa com sucesso, efetuar operação “splaying” ao nó encontrado
- Se pesquisa com insucesso, efetuar operação “splaying” ao último nó acedido

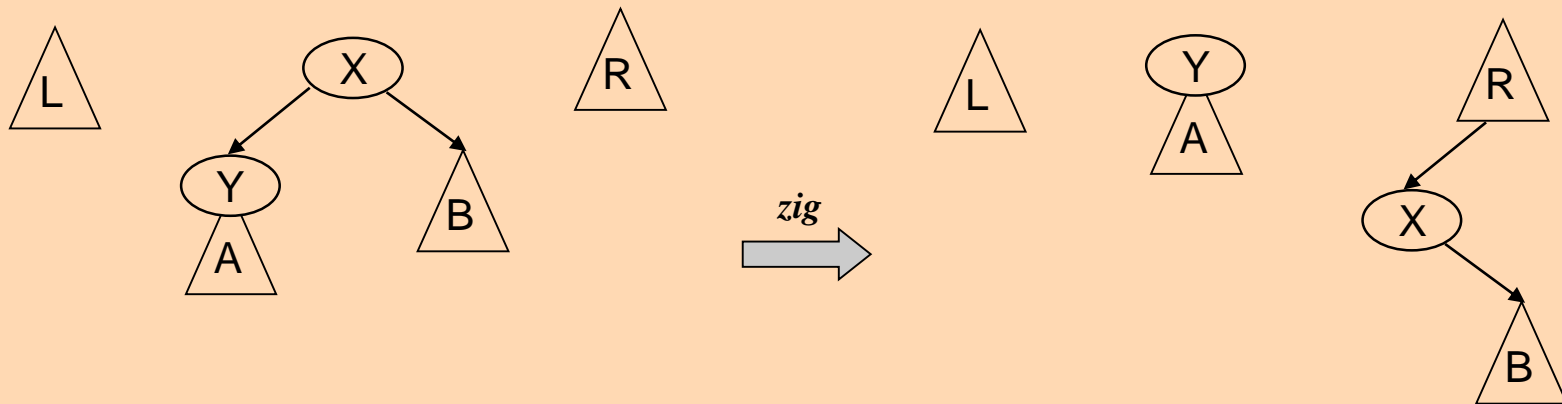
Árvores Splay Top-Down

Porque a análise “bottom-up” é difícil de implementar

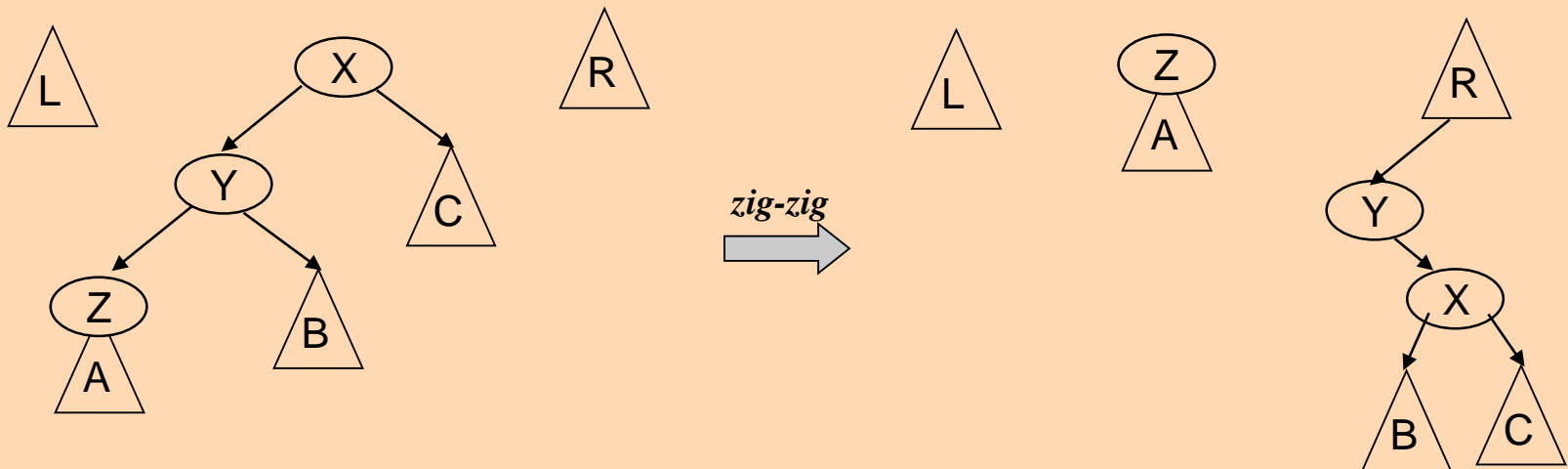
- Operação de Splay usa 3 árvores
 - árvore do meio: árvore com raiz no nó corrente
 - árvore da esquerda L: nós menores que X que não estão na subárvore de X
 - árvore da direita R: nós maiores que X que não estão na subárvore de X
 - inicialmente: L e R estão vazias e X é raiz da árvore

à medida que se desce na árvore, analisando dois níveis de cada vez, encontra-se um par de nós

Árvores Splay Top-Down

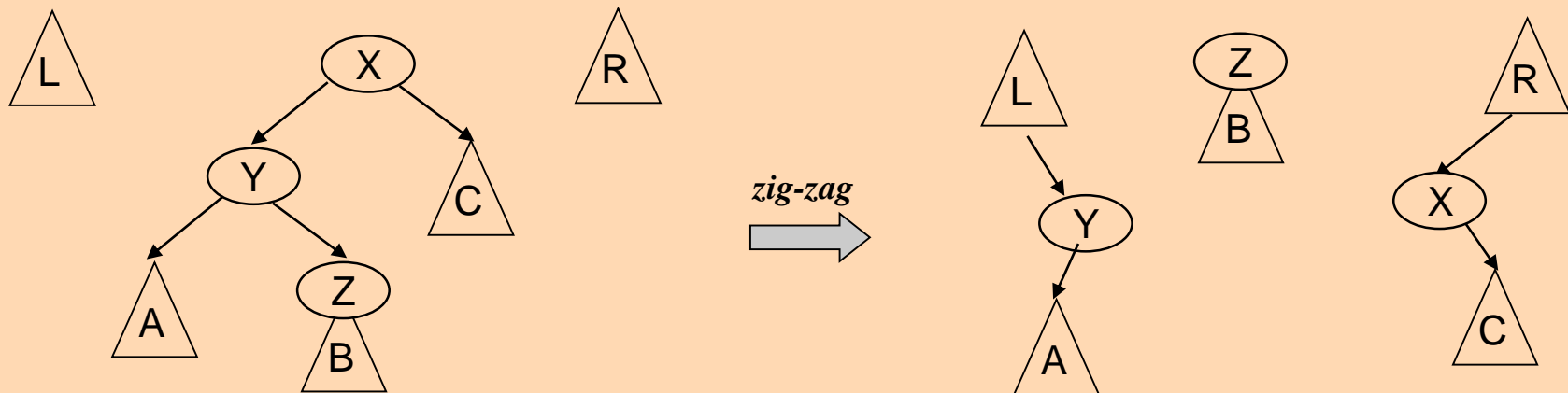


X é o menor elemento na árvore da direita (R): fácil inserir no futuro

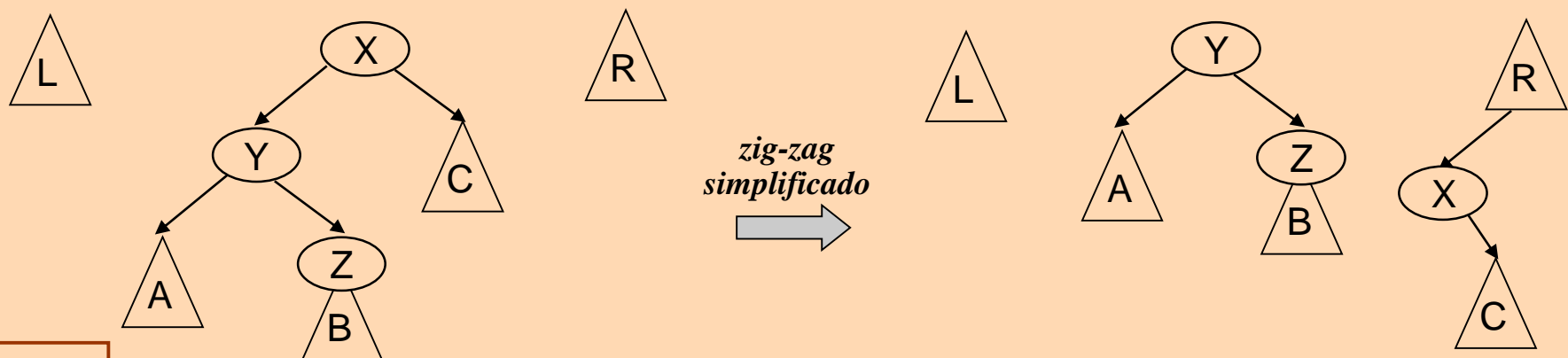


Y é o menor elemento na árvore da direita (R): fácil inserir no futuro

Árvores Splay Top-Down



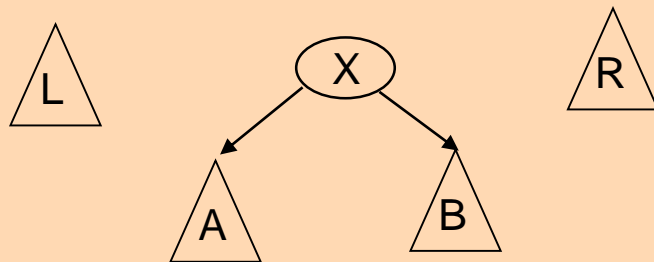
X é o menor elemento na árvore da direita (R): fácil inserir no futuro
Y é o maior elemento na árvore da esquerda (L): fácil inserir no futuro



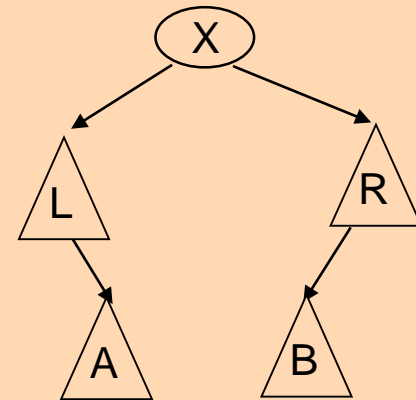
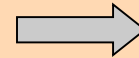
X é o menor elemento na árvore da direita (R): fácil inserir no futuro



Árvores Splay Top-Down



união das árvores



- Exercício: procurar o valor 19

