

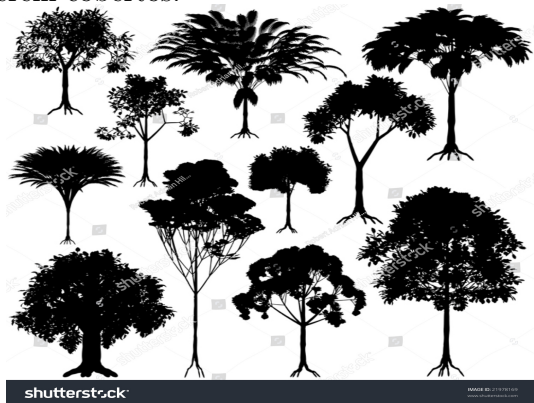
# Onde estamos ...

# Capítulo 06 – Árvores

Pontos fundamentais a serem cobertos:

mento

- 1 Contexto e motivação
- 2 Definição
- 3 Implementações
- 4 Exercícios



# Definição

- Uma árvore é uma estrutura hierárquica composta por nós e ligações entre eles
- Pode ser vista como um grafo acíclico
- Cada nó possui somente um pai e zero ou mais filhos
- Muitas definições ...

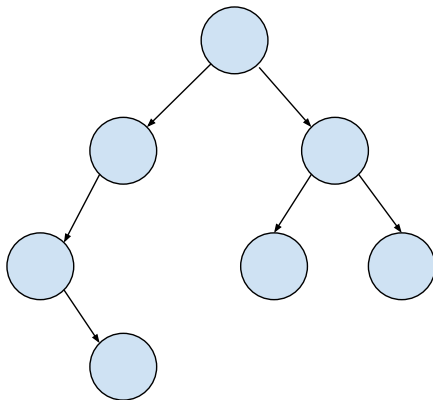


Figura 1: Exemplo de uma árvore

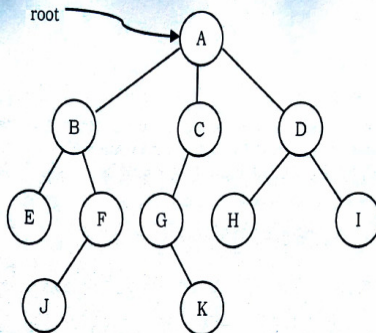
# Roadmap para estudo

mantendo um *foco*:

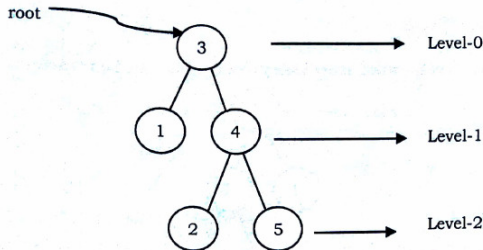
- ❶ Conceitos de árvores genéricas etc ...
- ❷ Árvores Binárias
- ❸ Árvore Binária de Busca
- ❹ Árvore AVL (iniciais dos nomes: *Georgii Adelson-Velsky et Evguenii Landis (en), qui l'ont publié en 1962 sous le titre An Algorithm for the Organization of Information*)
- ❺ Projeto 10% :
  - Implemente uma AVL;
  - Leia um conjunto de dados numéricos contidos no arquivo fornecido (um valor por linha: string, int, float, char), inserindo-os sequencialmente na AVL implementada;
  - Imprima o percurso (valores dos nós) em pré-ordem, em-ordem e pós-ordem, além da altura da árvore
  - Com a altura dará para ver se a AVL está OK!
- ❻ Vídeos bem legais no Youtube da UNIVEST

# Características – Requisitos

- Como é um nó?
- Qual o grau de um nó?
- Como devem estar estruturados os valores dos nós?
- O que é uma chave do nó?
- O que é a altura?
- Nível?
- Caminhos



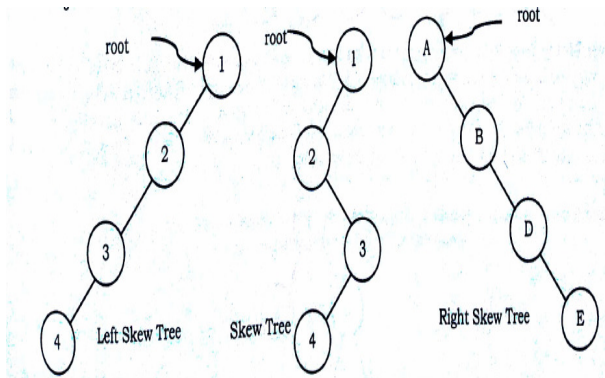
- The *root* of a tree is the node with no parents. There can be at most one root node in a tree (node *A* in the above example).
- An *edge* refers to the link from parent to child (all links in the figure).
- A node with no children is called *leaf node* (*E, J, K, H* and *I*).
- Children of same parent are called *siblings* (*B, C, D* are siblings of *A*, and *E, F* are the siblings of *B*).
- A node *p* is an *ancestor* of node *q* if there exists a path from *root* to *q* and *p* appears on the path. The node *q* is called a *descendant* of *p*. For example, *A, C* and *G* are the ancestors of *K*.
- The set of all nodes at a given depth is called the *level* of the tree (*B, C* and *D* are the same level). The *root* node is at level zero.



- The *depth* of a node is the length of the path from the root to the node (depth of *G* is 2, *A* – *C* – *G*).
- The *height* of a node is the length of the path from that node to the deepest node. The height of a tree is the length of the path from the root to the deepest node in the tree. A (rooted) tree with only one node (the root) has a height of zero. In the previous example, the height of *B* is 2 (*B* – *F* – *J*).
- *Height of the tree* is the maximum height among all the nodes in the tree and *depth of the tree* is the maximum depth among all the nodes in the tree. For a given tree, depth and height returns the same value. But for individual nodes we may get different results.
- The size of a node is the number of descendants it has including itself (the size of the subtree *C* is 3).
- If every node in a tree has only one child (except leaf nodes) then we call such trees *skew trees*. If every node has only left child then we call them *left skew trees*. Similarly, if every node has only right child then we call them *right skew trees*.



# Glossário – 03



# Árvores Genéricas

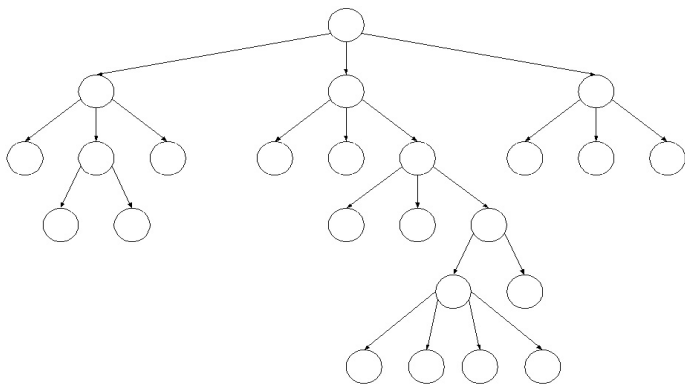


Figura 2: Usando os problemas de árvores genéricas para apresentar Árvores Binárias (AB)

# Transformando uma Árvore Genérica em Binária

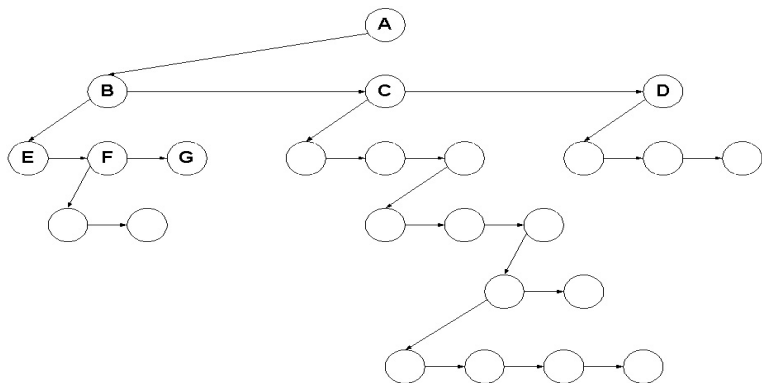


Figura 3: Usando os problemas de árvores genéricas para apresentar Árvores Binárias(AB)

# Representação Computacional de uma Árvore Genérica

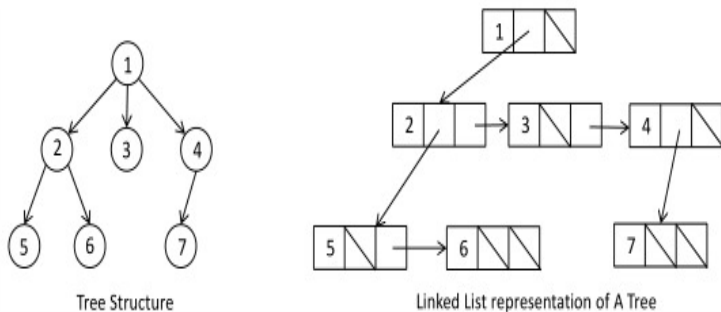
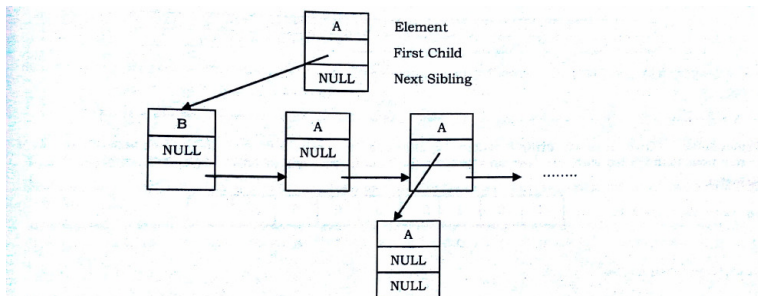


Figura 4: Felizmente há um algoritmo que transforme Árvore Genéricas em Binárias (AB)

# Representação Computacional de uma Árvore Genérica



Based on this discussion, the tree node declaration for general tree can be given as:

```
struct TreeNode {  
    int data;  
    struct TreeNode *firstChild;  
    struct TreeNode *nextSibling;  
};
```

**Note:** Since we are able to convert any generic tree to binary representation; in practice we use binary trees. We can treat all generic trees with a first child/next sibling representation as binary trees.

Figura 5: Veja a *struct* ... lembra o quê?

- Área de compiladores: análise sintática
- Buscas com complexidade na ordem de:  $O(\log n)$
- Na área de IA para construção de árvores de decisão: mineração de dados (*big data*)
- Organização de taxonomias de conhecimento
- Estruturas hierárquicas em geral

# Aplicação de Árvores

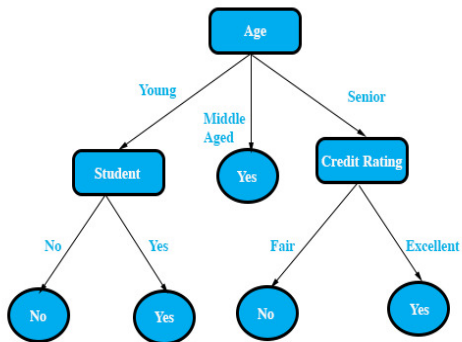
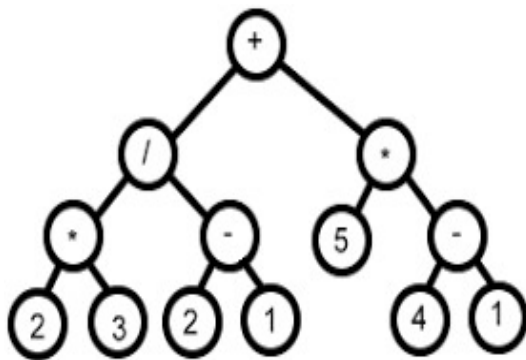


Figura 6: Árvore de Decisão

# Aplicação de Árvores



Expression tree for  $2*3/(2-1)+5*(4-1)$

Figura 7: Árvores de expressões – binárias – novamente



# Árvores Binárias de Buscas - ABBs

- Árvores Genéricas (AGs): foram utilizadas para motivação as ABBs
- Computacionalmente, as ABBs tem um interesse maior que as AGs
- Assim, se inicia com ABBs, e seus algoritmos serão adaptados as AGs

# Árvores Binárias de Buscas - ABBs

- Árvores Genéricas (AGs): foram utilizadas para motivação as ABBs
- Computacionalmente, as ABBs tem um interesse maior que as AGs
- Assim, se inicia com ABBs, e seus algoritmos serão adaptados as AGs

## Definição:

Árvore onde cada nó possui até 2 filhos. O filho da esquerda só pode conter chaves menores do que a do pai, enquanto que o filho da direita só comporta chaves maiores do que a do pai.

# Árvore Binária de Busca

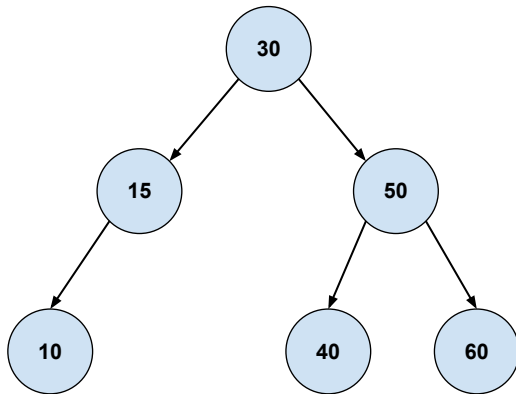
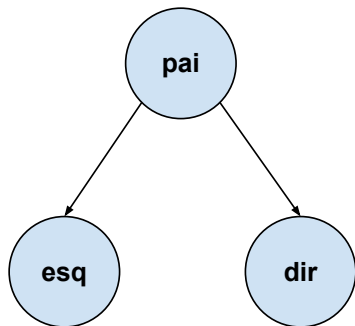


Figura 8: Exemplo de árvore binária de busca

# Árvore Binária de Busca



```
struct ArvoreBinaria {  
    struct ArvoreBinaria* esq;  
    struct ArvoreBinaria* dir;
```

```
    // contém a chave e os dados satélite  
    Tipoltem valor;  
};
```

Figura 9: Estrutura básica / nó

# Operações Básicas

## Operações Básicas

- Inserção
- Visitas na árvore (percorrer os nós)
- Busca
- Remoção – faltando

## Usos Comuns

- Dicionários / vetores associativos
- Filas de prioridades

## Alguns pontos **atenção**:

- Muitos conceitos importantes
- Algoritmos *pequenos* para ABs usando a recursividade
- A recursividade tem um custo de espaço computacional (conteúdo ilustrado ao longo do curso)
- Alternativa e uma estrutura fundamental em ABs: **pilhas**
- Empilhar/Desempilhar uma sequência de nós numa busca ou meta
- ABs  $\neq$  ABB (são diferentes)
- Exemplo: o conceito de **chave** é onipresente nas ABBs
- Exemplifique esta diferença e avance após esta figura no seu caderno

# Complexidade Computacional

- Quando a árvore está balanceada todas as três operações podem ser implementadas com complexidade computacional igual a  $O(\log n)$
- Faça o desenho e verifique o crescimento exponencial de nós na árvore, e sua função inversa é um logaritmo na base 2 (estude isto!)
- No pior caso (desbalanceamento) estas operações possuem complexidade  $O(n)$  [?].
- *Quizz*: uma árvore binária com  $n$  nós, quantas topologias de árvores distintas podemos formar?

# Complexidade Computacional

- Quando a árvore está balanceada todas as três operações podem ser implementadas com complexidade computacional igual a  $O(\log n)$
- Faça o desenho e verifique o crescimento exponencial de nós na árvore, e sua função inversa é um logaritmo na base 2 (estude isto!)
- No pior caso (desbalanceamento) estas operações possuem complexidade  $O(n)$  [?].
- *Quiz*: uma árvore binária com  $n$  nós, quantas topologias de árvores distintas podemos formar?
- Comece com  $n = 1$ ,  $n = 2$ ,  $n = 3$ , ...



# Complexidade Computacional

- Quando a árvore está balanceada todas as três operações podem ser implementadas com complexidade computacional igual a  $O(\log n)$
- Faça o desenho e verifique o crescimento exponencial de nós na árvore, e sua função inversa é um logaritmo na base 2 (estude isto!)
- No pior caso (desbalanceamento) estas operações possuem complexidade  $O(n)$  [?].
- *Quizz*: uma árvore binária com  $n$  nós, quantas topologias de árvores distintas podemos formar?
- Comece com  $n = 1$ ,  $n = 2$ ,  $n = 3$ , ...
- R:  $2^n - n$

# Árvore Binária de Busca - Inserção

```
INSERÇÃO(ARVORE, ITEM) {  
    SE ARVORE == NULO  
        ARVORE->ITEM = ITEM  
        return  
  
    SE ITEM->CHAVE < ARVORE->CHAVE  
        SE ARVORE->ESQ = NULO ENTÃO  
            ARVORE->ESQ = ARVORE(ITEM)  
        SENÃO  
            INSERÇÃO(ARVORE->ESQ, ITEM)  
    SENÃO  
        SE ARVORE->DIR = NULO ENTÃO  
            ARVORE->DIR = ARVORE(ITEM)  
        SENÃO  
            INSERÇÃO(ARVORE->DIR, ITEM)  
}
```

- Se a árvore estiver vazia em  $t_0$  (☺ ☹), esta vira uma ABB ao final
- Vá ao fonte implementado – insere nó

# Árvore Binária de Busca - Inserção

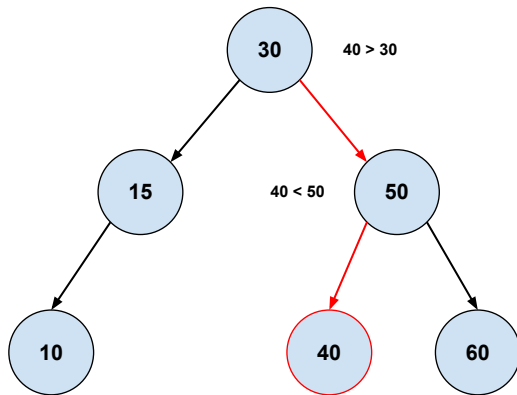


Figura 10: Exemplo de inserção da chave 40

# Árvore Binária de Busca - Inserção

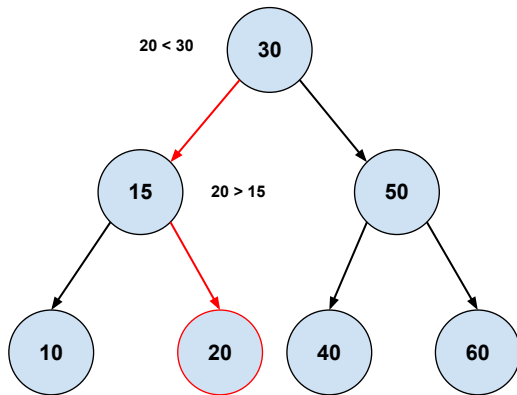


Figura 11: Exemplo de inserção da chave 20

## Exercícios – Inserção

Seguindo o algoritmo acima de inserção, construa as árvores binárias para seguintes entradas:

- $30 \rightarrow 20 \rightarrow 40 \rightarrow 10 \rightarrow 25 \rightarrow 35 \rightarrow 50$
- $30 \rightarrow 40 \rightarrow 20 \rightarrow 35 \rightarrow 25 \rightarrow 10 \rightarrow 50$
- $50 \rightarrow 40 \rightarrow 20 \rightarrow 35 \rightarrow 25 \rightarrow 10 \rightarrow 30$
- $50 \rightarrow 25 \rightarrow 20 \rightarrow 35 \rightarrow 40 \rightarrow 10 \rightarrow 30$
- $10 \rightarrow 20 \rightarrow 25 \rightarrow 30 \rightarrow 35 \rightarrow 40 \rightarrow 50$
- Que reflexões?

# Percorrendo Árvores Binárias

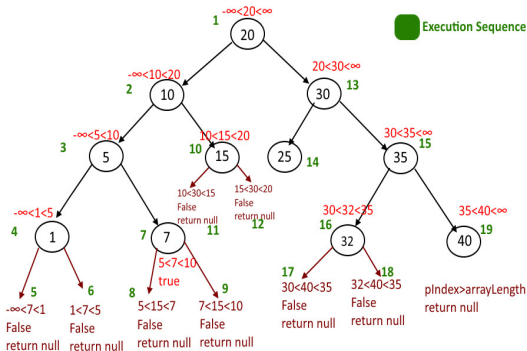
- Uma operação muito comum é percorrer uma árvore binária, o que significa passar por todos os nós, pelo menos uma vez.
- O conceito de visitar significa executar uma operação com a informação armazenada no nó, por exemplo, imprimir seu conteúdo.
- Na operação de percorrer a árvore pode-se passar por alguns nós mais de uma vez, sem porém visitá-los.
- Uma árvore é uma estrutura não seqüencial, diferentemente de uma lista, por exemplo. Não existe ordem natural para percorrer árvores e portanto podemos escolher diferentes maneiras de percorrê-las.
- Iremos estudar três métodos para percorrer árvores.
- Todos estes três métodos podem ser definidos recursivamente e se baseiam em **três operações básicas**: visitar a raiz, percorrer a subárvore da esquerda e percorrer a subárvore da direita.
- A única diferença entre estes métodos é a ordem em que estas operações são executadas.

O primeiro método, conhecido como percurso em pré-ordem, implica em executar recursivamente os três passos na seguinte ordem:

- 1 Visitar a raiz (imprimir esta);
- 2 Percorrer a subárvore da esquerda em pré-ordem;
- 3 Percorre a subárvore da direita em pré-ordem.

# Exemplo em pré-ordem

`int[] preOrder = { 20, 10, 5, 1, 7, 15, 30, 25, 35, 32, 40 };`





## Outros métodos:

### In-ordem:

De modo recursivo, execute os 3 passos que se seguem:

- 1 Percorrer a subárvore da esquerda em pré-ordem;
- 2 Visitar a raiz (imprimir esta);
- 3 Percorre a subárvore da direita em pré-ordem.

### Pós-ordem:

De modo recursivo, execute os 3 passos que se seguem:

- 1 Percorrer a subárvore da esquerda em pré-ordem;
- 2 Percorre a subárvore da direita em pré-ordem;
- 3 Visitar a raiz (imprimir esta);

# Resumo das Estratégias

The order in which the nodes are visited during a tree traversal can be easily determined by imagining there is a “flag” attached to each node, as follows:



To traverse the tree, collect the flags:

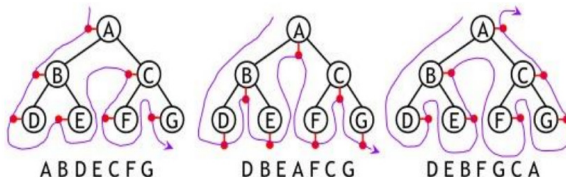


Figura 12: Resumo de *transversal tree* – percorrer árvores – reflexões

- 1 Para as árvores construídas no exercício, mostre a sequência dos nós em: *pré-ordem*, *in-ordem* e *pós-ordem*

- 1 Para as árvores construídas no exercício, mostre a sequência dos nós em: *pré-ordem*, *in-ordem* e *pós-ordem*
- 2 Usando *pós-ordem* e uma pilha, verifique como seria para implementar um algoritmo que fizesse um *pretty-tree* de uma ABB

- ① Para as árvores construídas no exercício, mostre a sequência dos nós em: *pré-ordem*, *in-ordem* e *pós-ordem*
- ② Usando *pós-ordem* e uma pilha, verifique como seria para implementar um algoritmo que fizesse um *pretty-tree* de uma ABB
- ③ Ainda sobre o algoritmo do *pós-ordem*, qual outro método importante que se utiliza a mesma idéia?

# Árvore Binária de Busca – Busca

```
BUSCA(ARVORE, CHAVE) {  
    SE ARVORE = NULO  
        return NULO  
  
    SE ARVORE->CHAVE = CHAVE  
        return ARVORE  
  
    SE CHAVE < ARVORE->CHAVE  
        return BUSCA(ARVORE->ESQ, CHAVE)  
    SENÃO  
        return BUSCA(ARVORE->DIR, CHAVE)  
}
```

- Aqui há o método iterativo também implementado
- Importante pois é a base do conceito de percurso em ABBs

# Árvore Binária de Busca – Remoção

A remoção de um nó se enquadra em um dos seguintes casos:

- ① Remoção de um nó folha (nenhum filho)
- ② Remoção de um nó com somente um filho
- ③ Remoção de um nó com dois filhos
- ④ Faltam as figuras ainda .... vários slides

## Remoção em Figuras – Caso 1

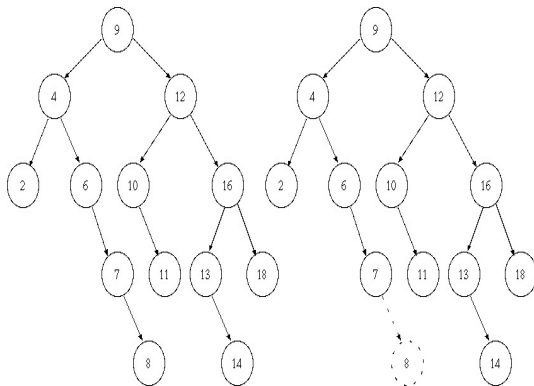


Figura 13: Remoção de uma folha – temp figure

Requisito: ser uma ABB!



## Remoção em Figuras – Caso 2

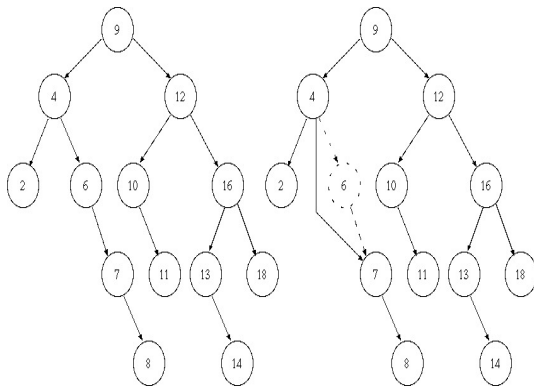


Figura 14: Remoção de um nó sem um dos filhos – esquerda ou direita igual a NULL

Requisito: ser uma ABB!

## Remoção em Figuras – Caso 3

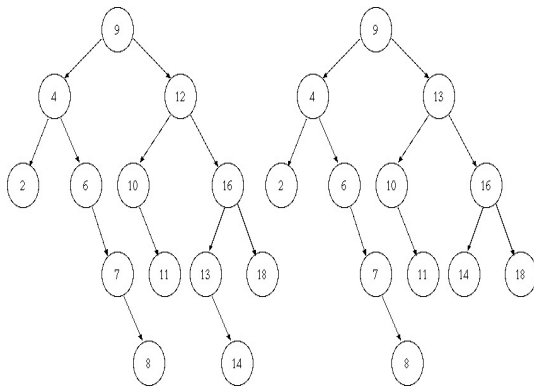


Figura 15: Remoção de um nó com sub-árvores em ambos os lados: remova o nó 12

Requisito: ser uma ABB!

## Exemplificando – uma raiz sem um dos filhos

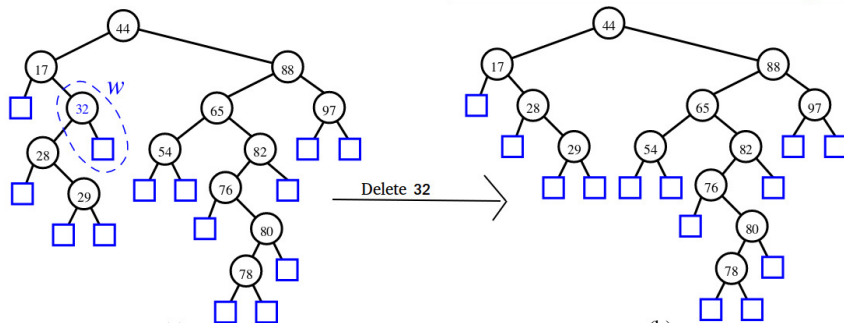


Figura 16: Quadrado **AZUL** é NULL

Requisito: ser uma ABB!

# Exemplificando a Remoção – uma raiz com dois filhos

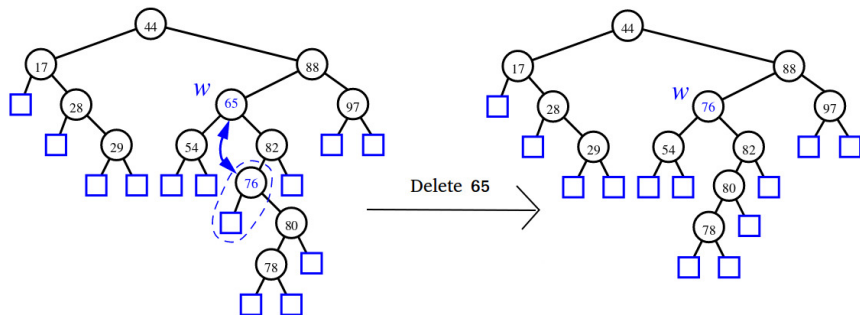


Figura 17: Quadrado **AZUL** é NULL

- A nova raiz será a menor chave à esquerda, da sub-árvore a direita do nó que se deseja excluir!
- Requisito: ser uma ABB!

# Ainda a Remoção

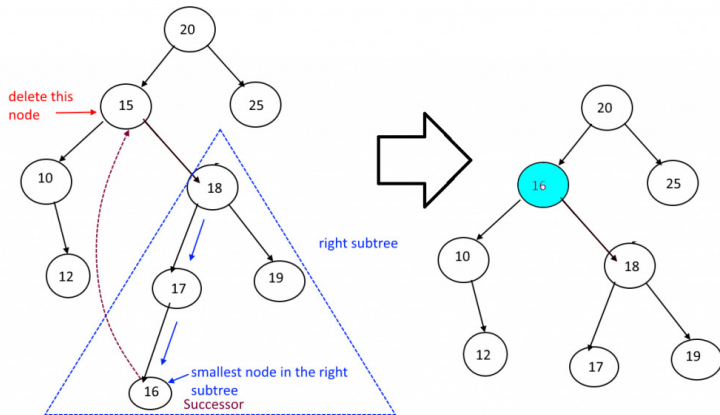


Figura 18: A idéia vale para os dois lados: é o menor à esquerda da sub-árvore a direita

Requisito: ser uma ABB!

- A remoção é a base das partes mais difíceis de ABs
- Pois o próximo assunto é *balanceamento*
- Implemente este método

- Uma árvore binária de busca balanceada garante operações de busca, inserção e remoção com complexidade  $O(\log n)$ , onde  $n$  é o número de nós, o que a torna atrativa para diversas aplicações.
- Determinadas sequências de inserções ou remoções podem fazer com que uma ABB fique desbalanceada, tornando suas operações  $O(n)$ .
- Para verificar se uma árvore (ABB) está desbalanceada é necessário algumas **métricas**!

- Uma árvore binária de busca balanceada garante operações de busca, inserção e remoção com complexidade  $O(\log n)$ , onde  $n$  é o número de nós, o que a torna atrativa para diversas aplicações.
- Determinadas sequências de inserções ou remoções podem fazer com que uma ABB fique desbalanceada, tornando suas operações  $O(n)$ .
- Para verificar se uma árvore (ABB) está desbalanceada é necessário algumas **métricas**!
- Por exemplo: número de nós por sub-árvores, alturas, etc



# Cálculo da Altura

A altura de um nó ou da árvore é a maior profundidade de uma de suas sub-árvores a esquerda ou direita. As folhas tem altura 0.

Um algoritmo é dado por:

```
ALTURA(ARVORE) {  
    SE ARVORE = NULO  
        return -1  
  
    A1 = ALTURA(ARVORE->DIR)  
    A2 = ALTURA(ARVORE->ESQ)  
  
    return maior(A1, A2) + 1  
}
```

Lembre: a raiz da árvore tem a maior altura!

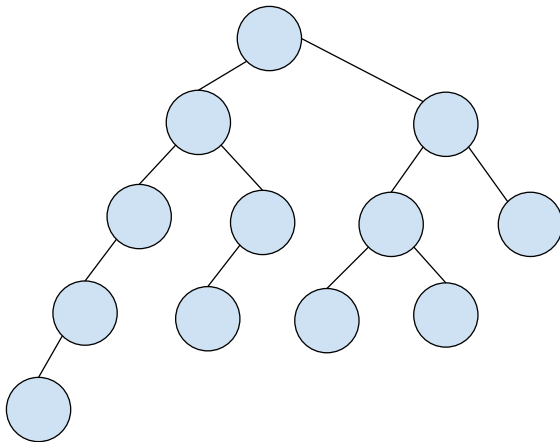


Figura 19: Exercício: determine a altura de cada subárvore.

## Cálculo da Altura

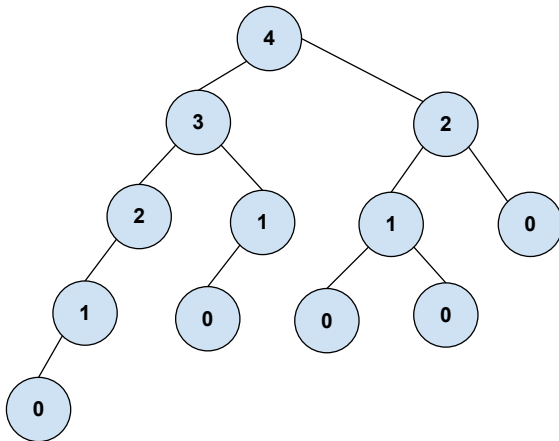


Figura 20: Resposta do exercício.

# Cálculo do Fator de Balanceamento

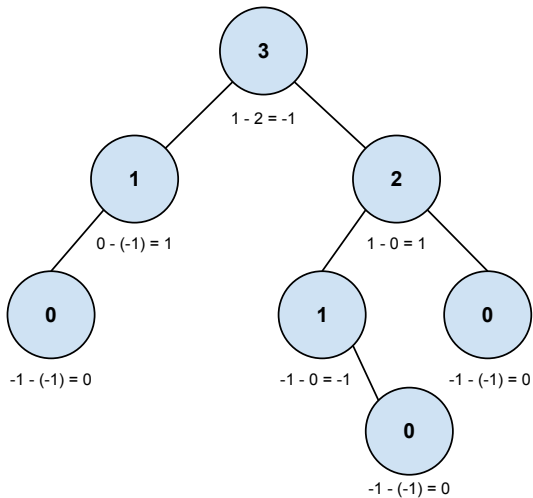
O fator de balanceamento de uma árvore é a diferença entre as alturas de suas sub-árvores a esquerda ou direita.

Um algoritmo é dado por:

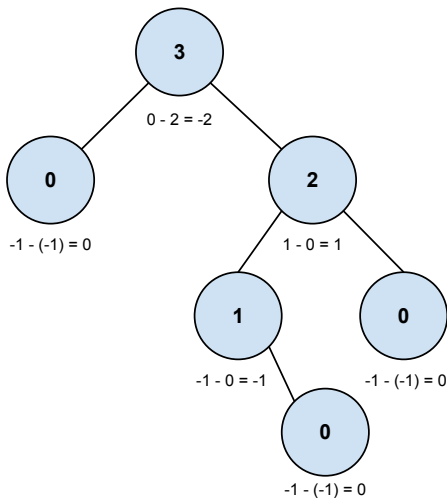
```
FB(ARVORE) {  
    A1 = ALTURA(ARVORE->ESQ)  
    A2 = ALTURA(ARVORE->DIR)  
    return A1 - A2  
}
```

- Uma ABB está balanceada quando cada nó possui um FB igual a **-1, 0** ou **1**
- A diferença das alturas entre dois nós no mesmo nível é um valor absoluto
- Uma inserção ou remoção pode tornar uma árvore desbalanceada, necessitando de **rotações** para o seu balanceamento

# Exemplo de ABB Balanceada



# Exemplo de ABB Desbalanceada



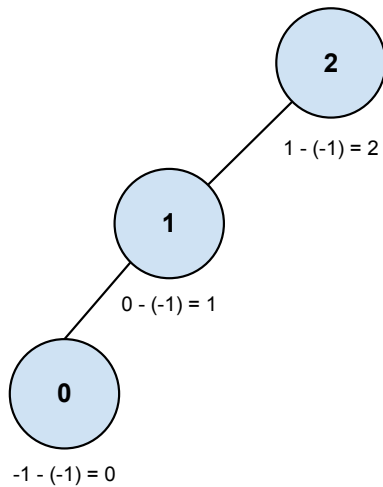
# Operação de rotação

```
ROTACAO_DIREITA(RAIZ) {  
    PIVO      = RAIZ->ESQ  
    RAIZ->ESQ = PIVO->DIR  
    PIVO->DIR = RAIZ  
    RAIZ      = PIVO  
}
```

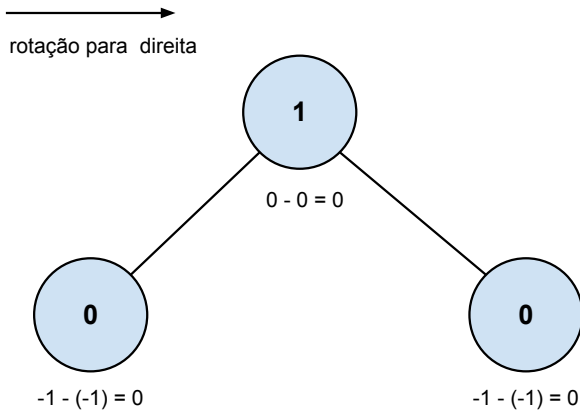
```
ROTACAO_ESQUERDA(RAIZ) {  
    PIVO      = RAIZ->DIR  
    RAIZ->DIR = PIVO->ESQ  
    PIVO->ESQ = RAIZ  
    RAIZ      = PIVO  
}
```



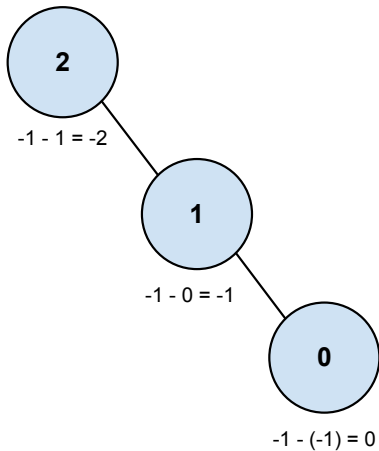
# Rotação para Direita



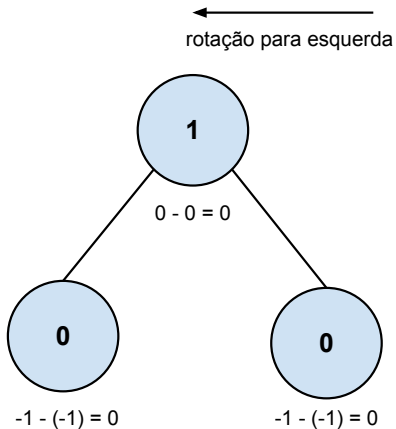
# Rotação para Direita



# Rotação para Esquerda



# Rotação para Esquerda



- **AVL** desenvolvida por G. M. Adelson-Velskii and E. M. Landis
- Critério do fator de balanceamento de  $-1$  a  $+1$  deve ser seguido em cada nó da árvore
- Garante o balanceamento da árvore ao realizar rotações após cada inserção ou remoção na ABB

# Balanceamento – Inserção

```
BALANCEAMENTO(RAIZ) {  
    SE FB(RAIZ) = -2 ENTÃO  
        SE FB(RAIZ->DIR) = -1 ENTÃO  
            ROTACAO_ESQUERDA(RAIZ)  
        SENÃO  
            ROTACAO_DIREITA(RAIZ->DIR)  
            ROTACAO_ESQUERDA(RAIZ)  
    SENÃO SE FB(RAIZ) = 2 ENTÃO  
        SE FB(RAIZ->ESQ) = 1 ENTÃO  
            ROTACAO_DIREITA(RAIZ)  
        SENÃO  
            ROTACAO_ESQUERDA(RAIZ->DIR)  
            ROTACAO_DIREITA(RAIZ)  
}
```

# Balanceamento - Inserção

- Para que a árvore tenha um bom desempenho, é essencial que o balanceamento seja calculado eficientemente, isto é, sem a necessidade de percorrer toda a árvore após cada modificação
- Manter a árvore estritamente balanceada após cada modificação tem seu preço (desempenho). Árvores AVL são utilizadas normalmente onde o número de consultas é muito maior do que o número de inserções e remoções e quando a localidade de informação não é importante

## rotações necessárias nas AVLs

A cada inserção há um rebalanceamento. Após uma inserção (vai valer para exclusão) os seguintes casos violam uma AVL. Logo, seguem os balanceamentos:

- Caso 1: uma inserção na sub-árvore à esquerda do filho à esquerda de X (simples)
- Caso 2: uma inserção na sub-árvore à direita do filho à esquerda de X (simples)
- Caso 3: uma inserção na sub-árvore à esquerda do filho à direita de X (duplas)
- Caso 4: uma inserção na sub-árvore à direita do filho à direita de X (duplas)



# Rotação para Esquerda

# Árvore de Espalhamento

- Uma alternativa de melhorar os procedimentos de AVL
- Reestrutura a árvore em cada operação de inserção, busca ou remoção por meio de operações de rotação
- Nome original: *splay tree* [?]. Não confundir com a Árvore N-Ária de Espalhamento (ANE) criada por professores da UDESC

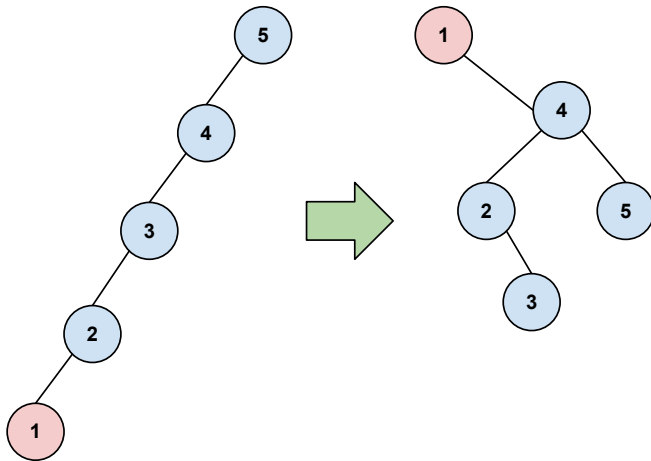
# Árvore de Espalhamento

- Evita a repetição de casos ruins [ $O(n)$ ] devido ao seu rebalanceamento natural
- Não realiza o cálculo de fatores de balanceamento, simplificando sua implementação
- Pior caso para uma operação se mantém  $O(n)$ , mas, ao considerar uma cadeia de operações, *garante* uma complexidade amortizada de  $O(\log n)$  para suas operações básicas

# Árvore de Espalhamento

- Se baseia na operação de espalhamento, que utiliza rotações para mover uma determinada chave até a raiz
- A sua complexidade  $O(\log n)$  em uma análise amortizada é garantida pelas rotações efetuadas, o que a difere do uso simples de heurísticas como o *mover para a raiz*

## Exemplo - Espalhamento pela chave 1



# Operações Básicas

**Espalhamento** Move a chave desejada para a raiz por uma sequência bem definida de operações de rotação

**Busca** Busca uma chave na árvore

**Inserção** Insere uma nova chave na árvore

**Remoção** Remove uma chave da árvore

- Uma árvore de espalhamento é uma árvore binária de busca válida, logo operações como os percursos (pré, em e pós ordem) são idênticas as operações em uma ABB
- As operações de inserção, busca e remoção podem ser definidas com base na operação de espalhamento

# Árvore de Espalhamento - Busca

```
BUSCA(RAIZ, CHAVE) {  
    return ESPALHAMENTO(RAIZ, CHAVE)  
}
```



# Árvore de Espalhamento - Inserção

```
INSERE(RAIZ, CHAVE) {  
    INSERE_ABB(RAIZ, CHAVE)  
    return ESPALHAMENTO(RAIZ, CHAVE)  
}
```

# Árvore de Espalhamento - Remoção

```
REMOVE(RAIZ, CHAVE) {  
    RAIZ = ESPALHAMENTO(RAIZ, CHAVE)  
  
    SE RAIZ->DIR ENTÃO  
        AUX = ESPALHAMENTO(RAIZ->DIR, CHAVE)  
        AUX->ESQ = RAIZ->ESQ  
    SENÃO  
        AUX = RAIZ->ESQ  
  
    return AUX  
}
```

# Estratégias de Espalhamento

Duas estratégias:

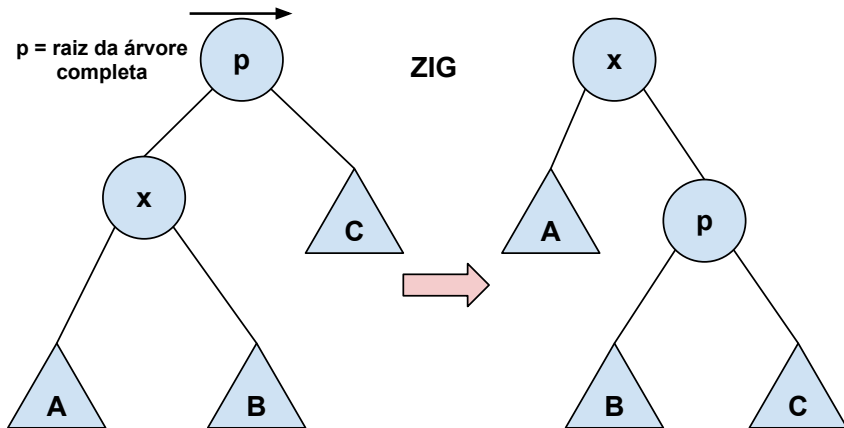
**Bottom-Up** Parte do nó acessado e o movimenta para a raiz da árvore por meio de rotações

**Top-Down** Parte do nó raiz, rotacionando e *removendo do caminho* os nós entre a raiz e o nó desejado, armazenando-os em duas árvores auxiliares, remontando a árvore completa na sua etapa final.

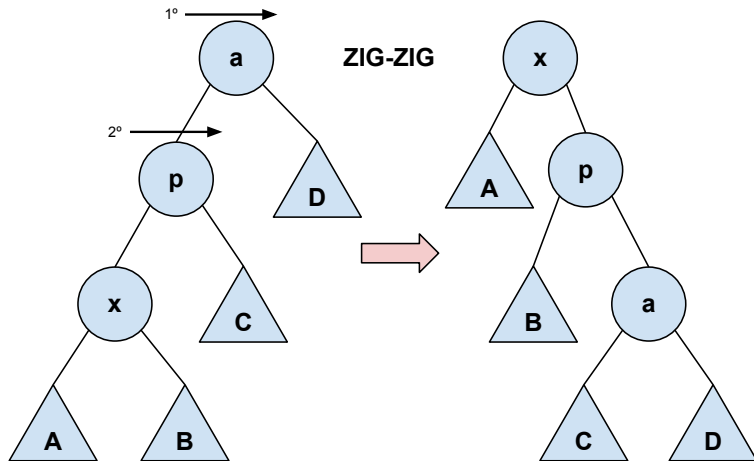
# Espalhamento Bottom-Up

- Na estratégia Bottom-Up, a operação de espalhamento realiza rotações subindo gradativamente de níveis, a partir da chave desejada
- Enquanto a chave não estiver na raiz, deve-se verificar qual o caso aplicável (ZIG, ZIG-ZIG ou ZIG-ZAG) e realizar as rotações necessárias

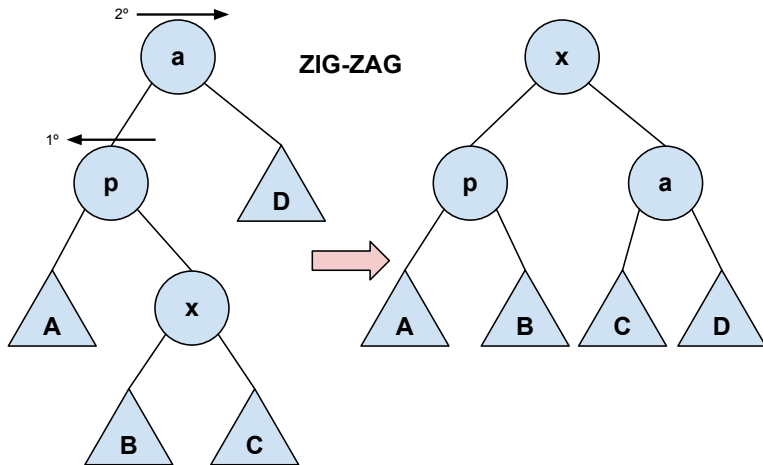
## Caso 1: ZIG



## Caso 2: ZIG-ZIG



## Caso 3: ZIG-ZAG

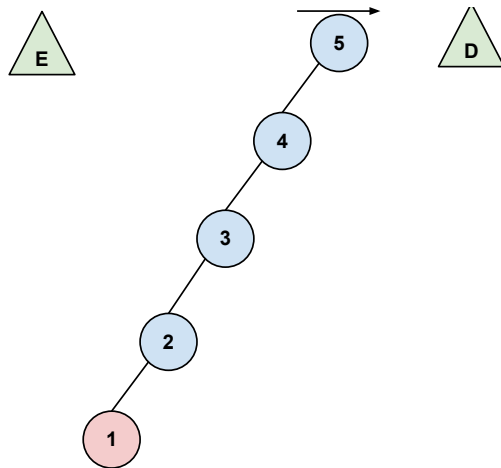


# Espalhamento Top-Down

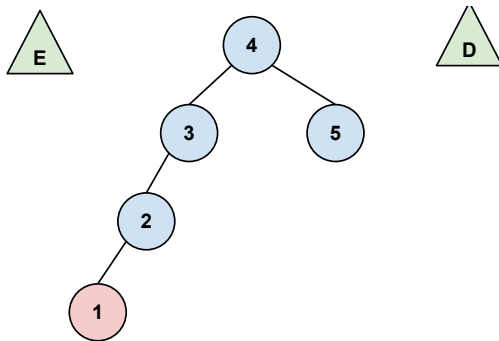
- Na estratégia Top-Down as chaves que estão no caminho da chave desejada para a raiz são rotacionadas e removidas para árvores auxiliares seguindo uma sequência de operações bem definidas
- Quando a chave desejada chega até a raiz, a árvore é remontada pelo retorno das chaves removidas



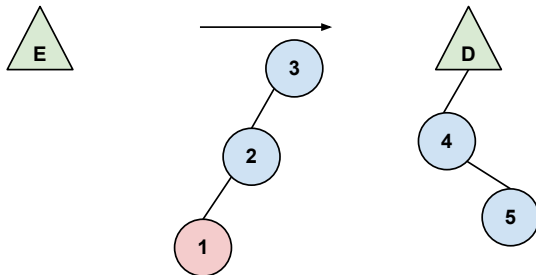
## Exemplo: Top-Down 1/6



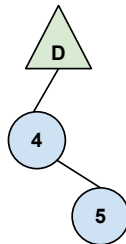
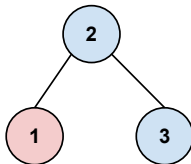
## Exemplo: Top-Down 2/6



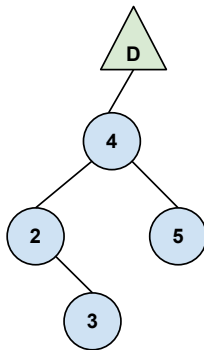
## Exemplo: Top-Down 3/6



## Exemplo: Top-Down 4/6



## Exemplo: Top-Down 5/6



## Exemplo: Top-Down 6/6

