# **SBVORIN:** Organização e Recuperação da Informação

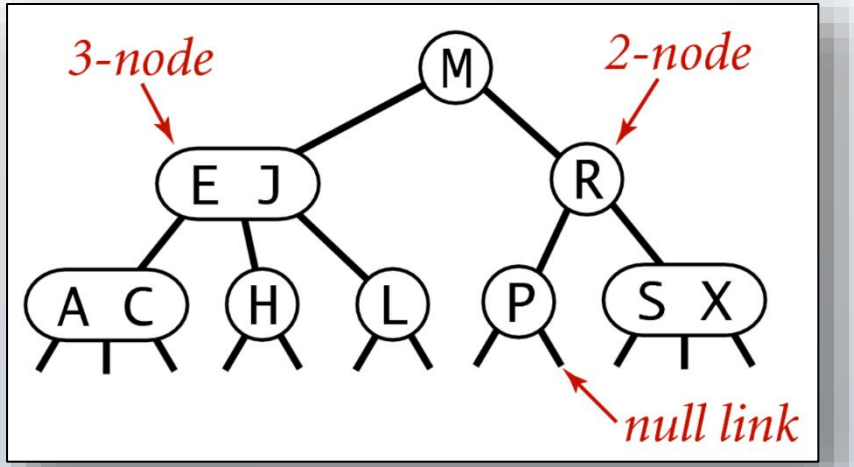
## Aula 06: Árvores Binárias de Busca Balanceadas: 2-3 *Search Tree*, Árvore Vermelho e Preto e B-*Trees*

Bacharelado em Ciência da Computação
Prof. Dr. David Buzatto

INSTITUTO FEDERAL DE EDUCAÇÃO, CIÊNCIA E TECNOLOGIA
SÃO PAULO
Campus São João da Boa Vista

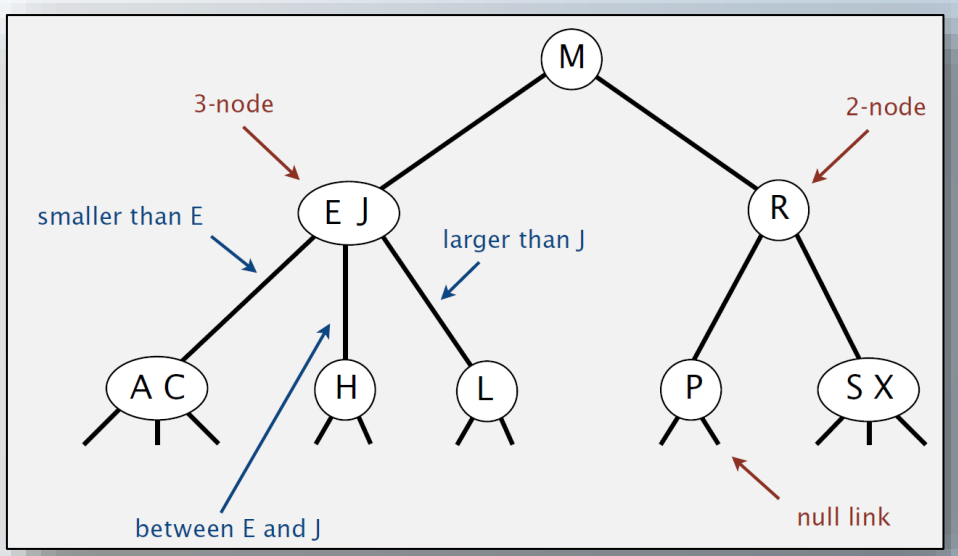
# 2-3 *Search Trees*
## Introdução

- Por que a altura de uma ABB aumenta?
  - Cada inserção (`put`) cria um novo nó, podendo aumentar a altura da árvore;
- **Solução:** colocar mais de uma chave em cada nó!
- **Difícil:** fazer isso de modo que nenhum nó tenha mais que 2 chaves.
  - A árvore, que era binária, torna-se ternária.



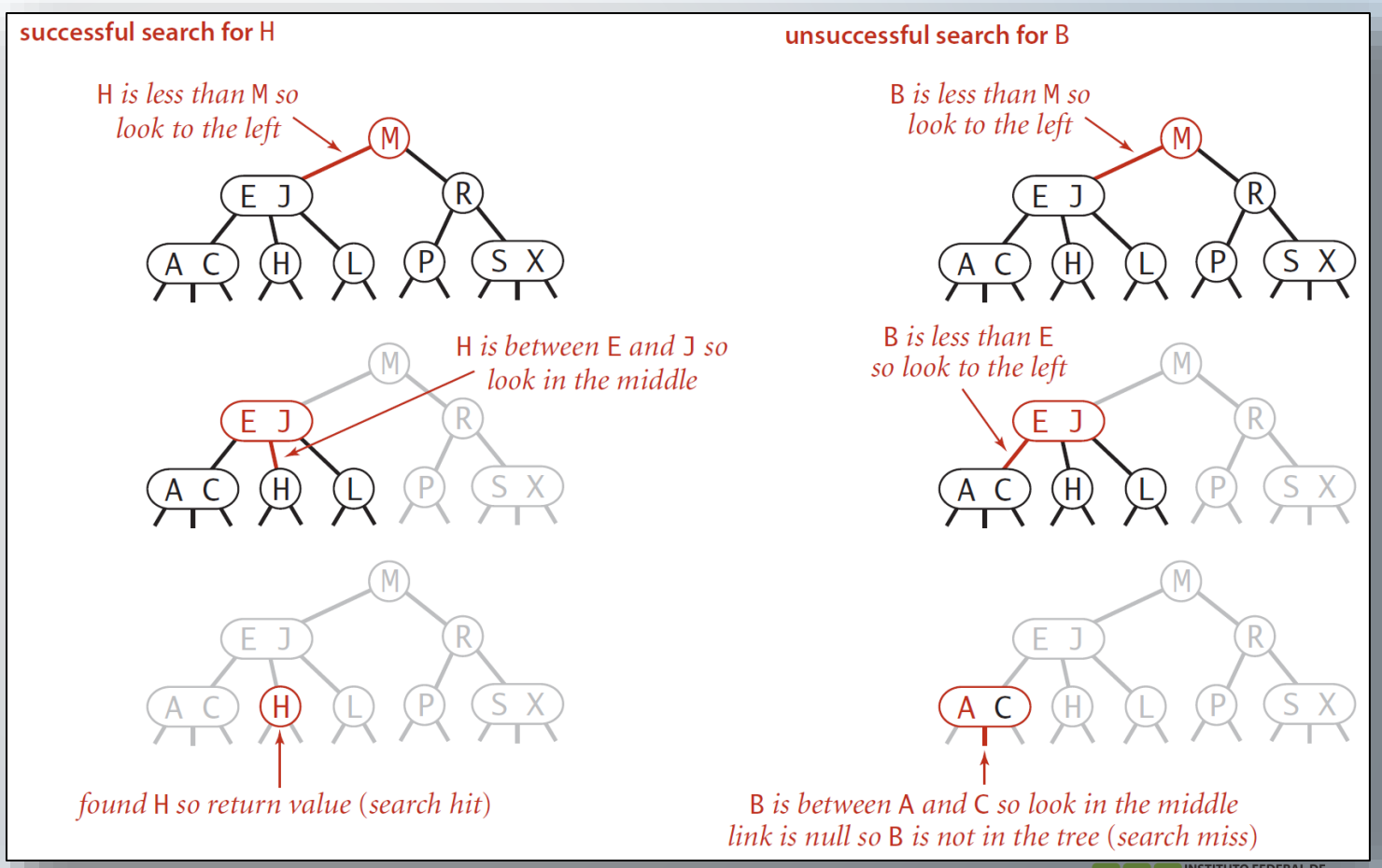
# 2-3 *Search Trees*
## Introdução

- Uma Árvore 2-3 de Busca (2-3 *Search Tree*) é uma ABB que possui:
  - Uma árvore vazia;
  - 2-node: um nó simples, que contém uma chave e duas referências(dois filhos):
    - Uma referência da esquerda para uma árvore 2-3 que tem chaves menores que a chave do nó e uma referência da direita para uma árvore 2-3 que tem chaves maiores;
  - 3-node: um nó duplo, que contém duas chaves e três referências (três filhos):
    - Uma referência da esquerda para uma árvore 2-3 que tem chaves menores; uma referência do meio para uma árvore 2-3 que tem chaves entre as duas chaves do nó; e uma referência da direita para uma árvore 2-3 que tem chaves maiores;
- Toda árvore 2-3 é perfeitamente balanceada:
  - Todos as referências nulas estão no mesmo nível.



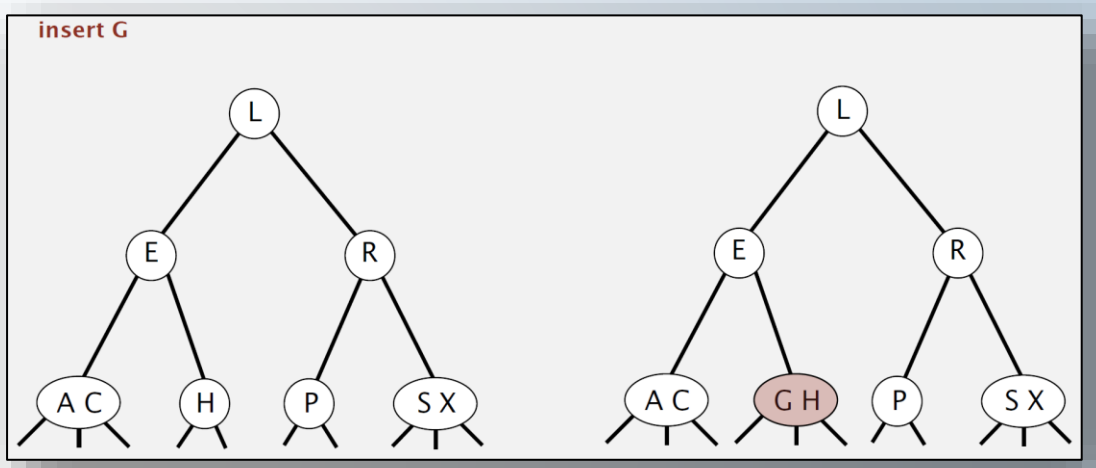
# 2-3 *Search Trees*
## Busca por Chave

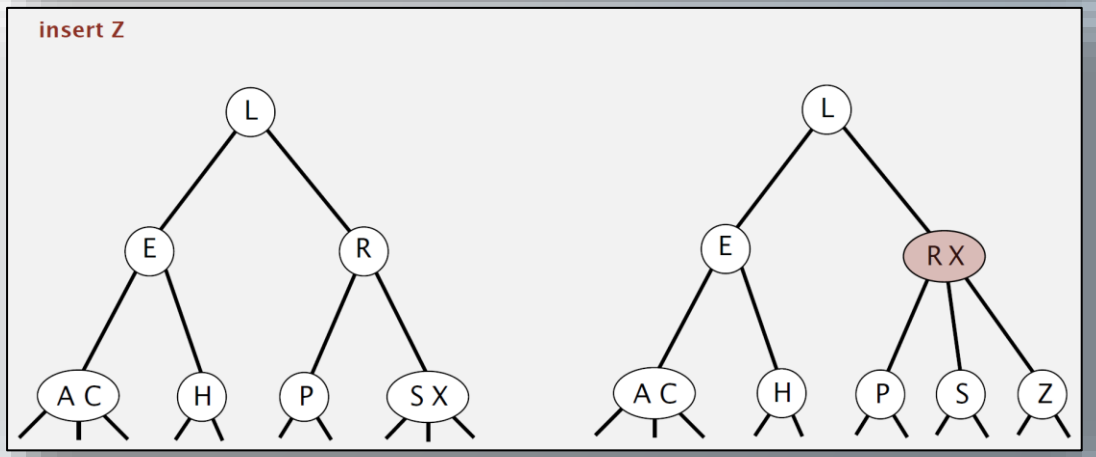
- Busca em uma Árvore 2-3 de Busca:
  - Comparar a chave pesquisada com as chaves do nó;
  - Encontra o intervalo contendo a chave pesquisada;
  - Segue pelo caminho associado ao intervalo recursivamente.



# 2-3 *Search Trees*
## Inserção de uma Nova Chave

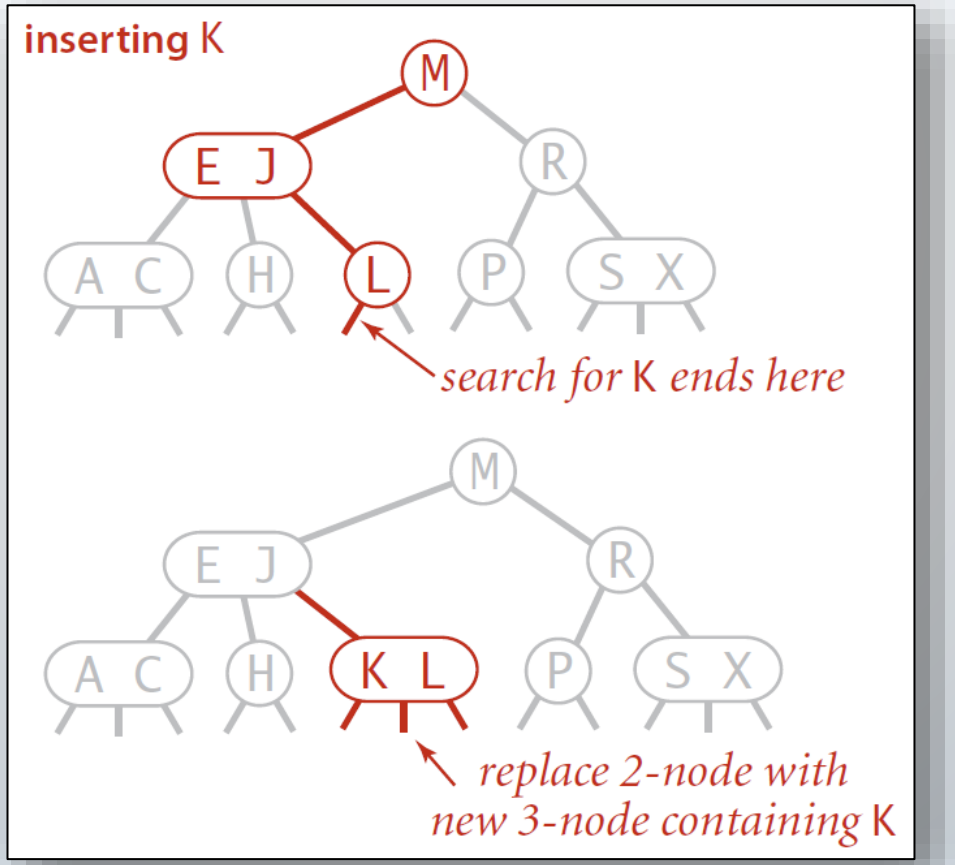
- Inserção em uma Árvore 2-3 de Busca:
  - Não esquecer o princípio base:
    - Toda Árvore 2-3 é perfeitamente balanceada, pois todas as referências nulas estão no mesmo nível;
  - Inserção em um 2-node:
    - Adiciona uma nova chave no 2-node para criar um 3-node;
  - Inserção em um 3-node:
    - Adiciona uma nova chave no 3-node para criar um nó 4-node temporariamente;
    - Movimenta a chave do meio do 4-node para o nó pai;
    - Se alcançar a raiz da árvore e for um nó 4-node, divida-o em três nós 2-node.



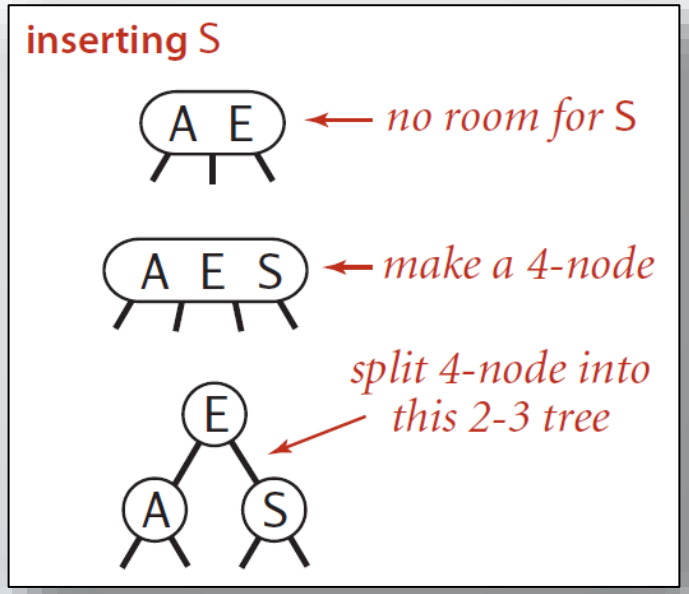
# 2-3 *Search Trees*
## Inserção de uma Nova Chave

- Inserção em uma Árvore 2-3 de Busca:
  - Exemplo de inserção em um nó simples.



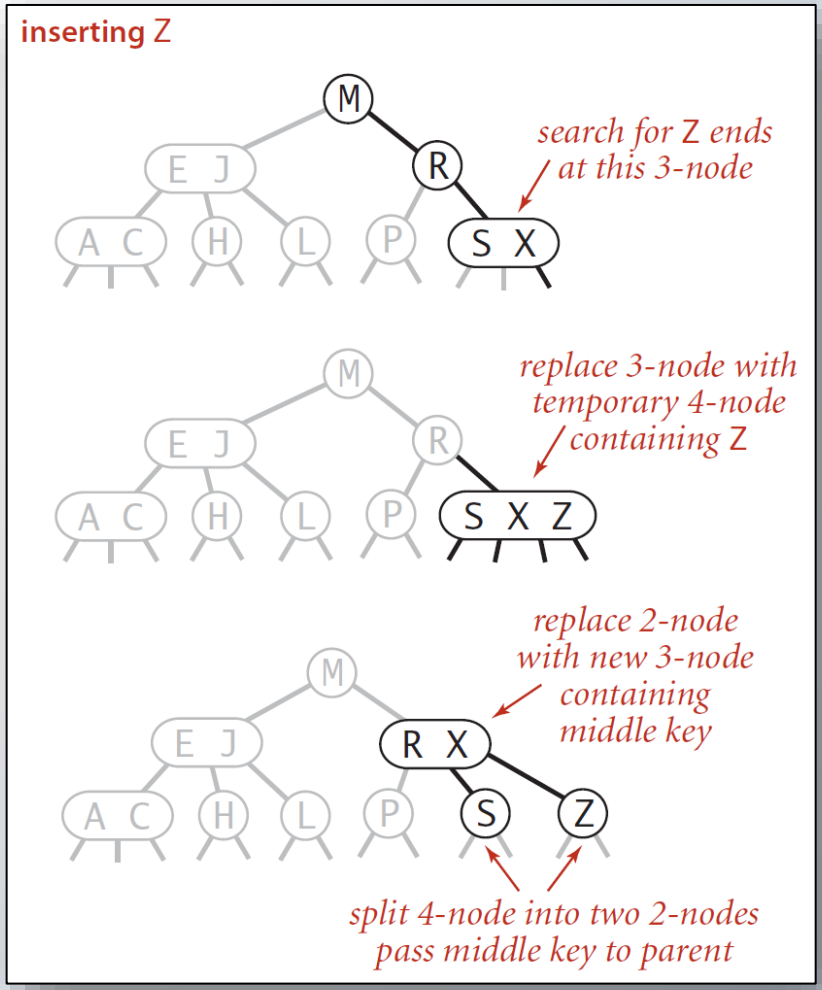
# 2-3 *Search Trees*
## Inserção de uma Nova Chave

- Inserção em uma Árvore 2-3 de Busca:
  - Exemplo de inserção em um nó duplo isolado;



# 2-3 *Search Trees*
## Inserção de uma Nova Chave

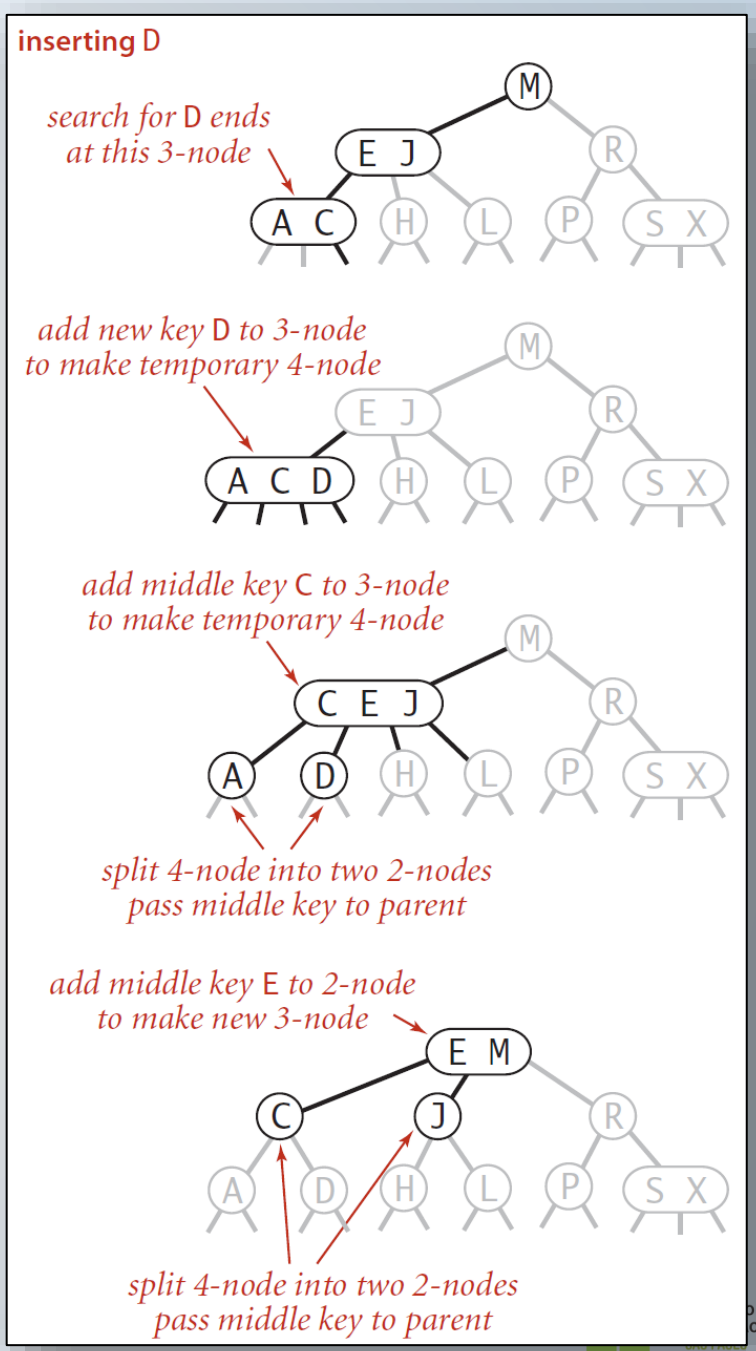
- Inserção em uma Árvore 2-3 de Busca:
  - Exemplo de inserção em um nó duplo cujo pai é um nó simples.



# 2-3 *Search Trees*

## Inserção de uma Nova Chave

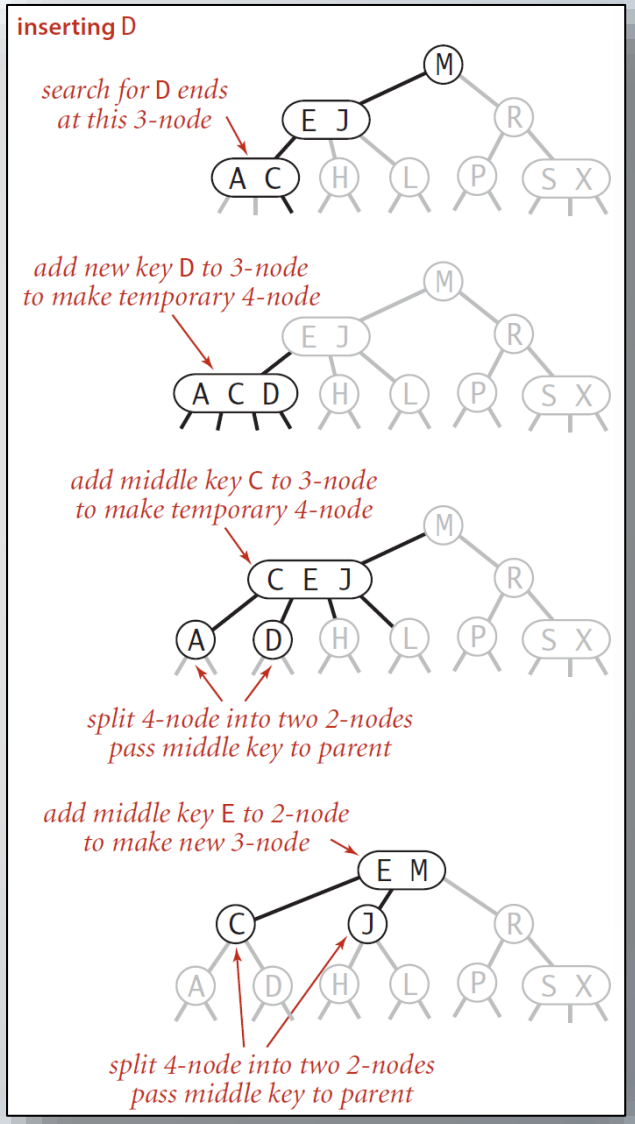
- Inserção em uma Árvore 2-3 de Busca:
  - Exemplo de inserção em um nó duplo cujo pai é um nó duplo.

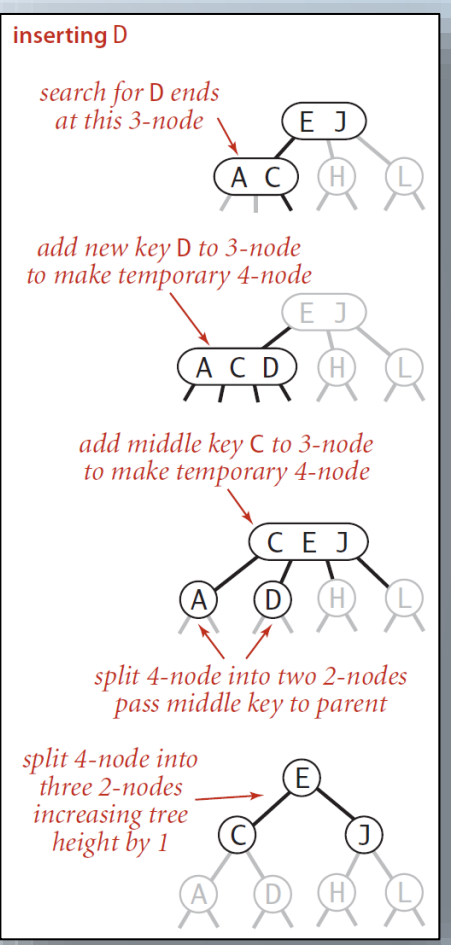


# 2-3 *Search Trees*

## Inserção de uma Nova Chave

- Inserção em uma Árvore 2-3 de Busca:
  - Exemplo de inserção em um nó duplo cujo pai é um nó duplo:
    - Repita a operação subindo em direção à raiz até encontrar um nó simples, sendo que nesse caso a altura não aumenta, ou até encontrar a raiz aumentando a altura.

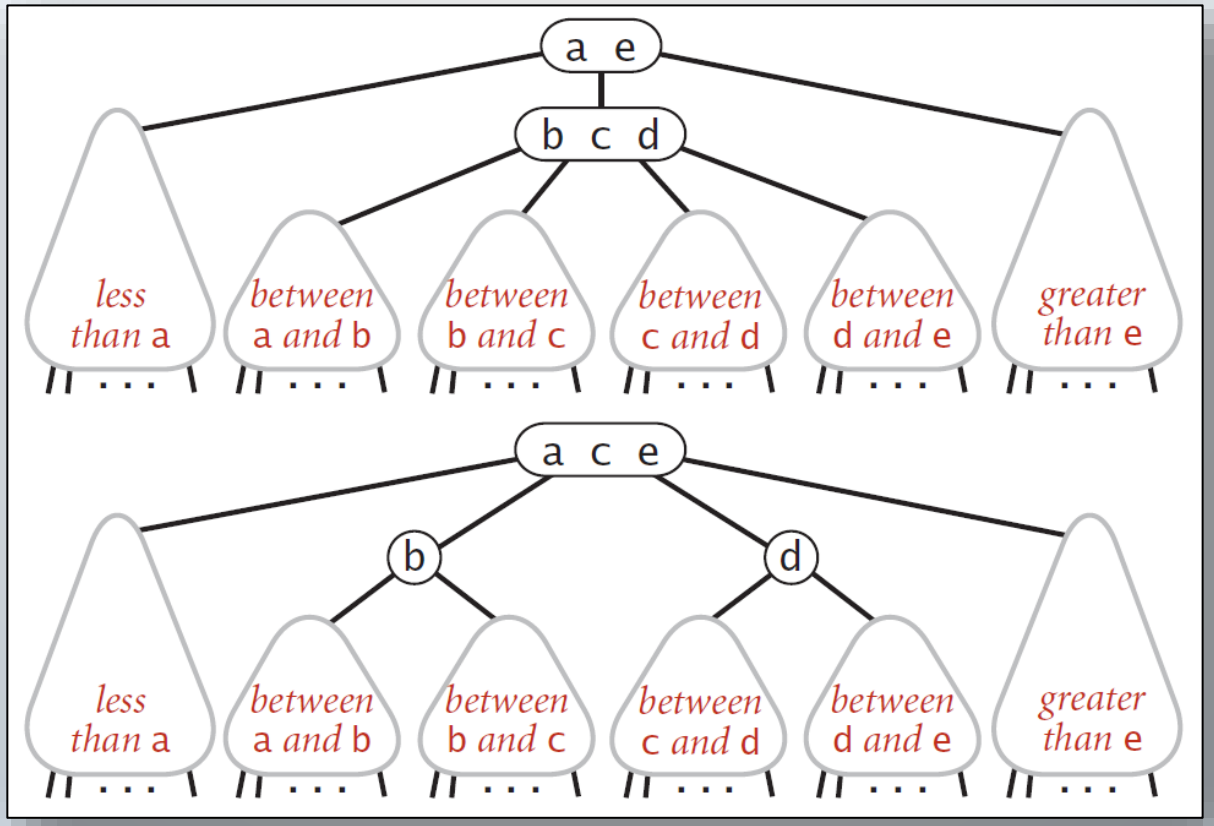


# 2-3 *Search Trees*

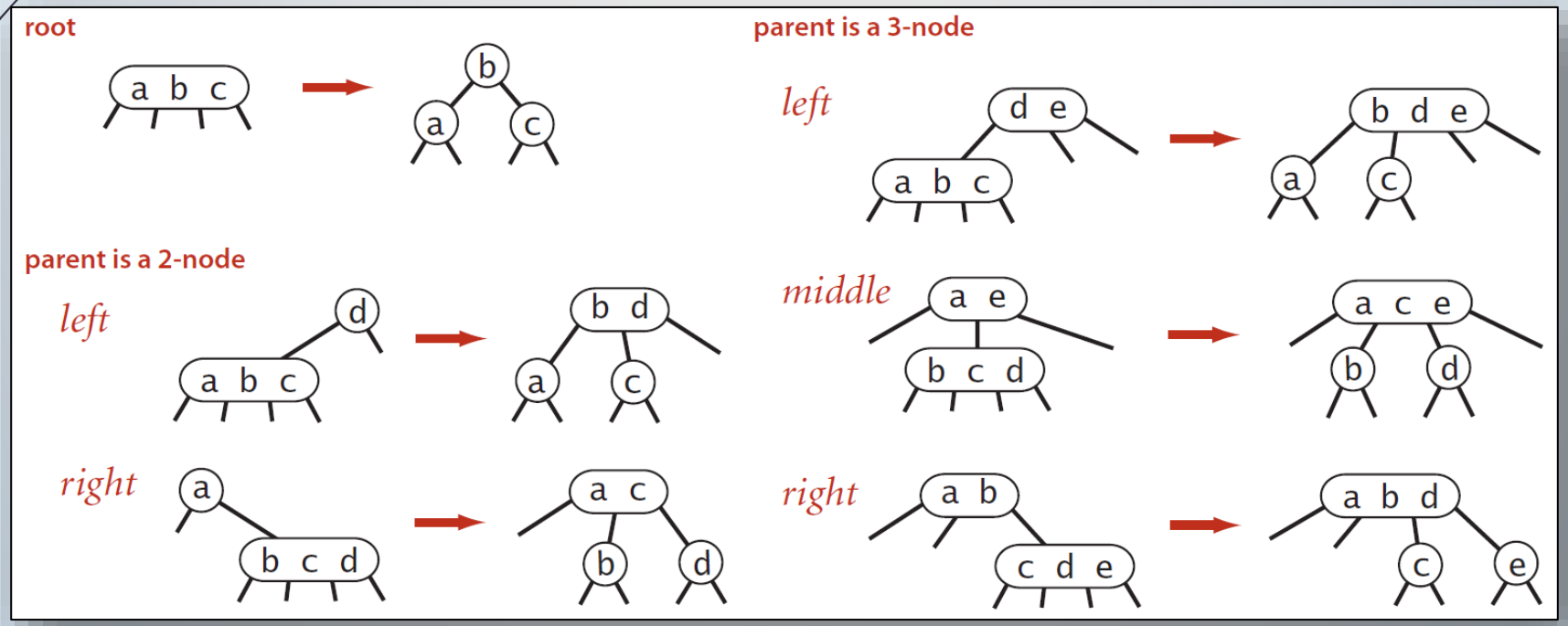
## Inserção de uma Nova Chave

- As transformações preservam as propriedades globais da árvore, ou seja, a árvore continua em ordem e perfeitamente balanceada;
  - Dividir um 4-node é uma transformação local: número constante de operações.



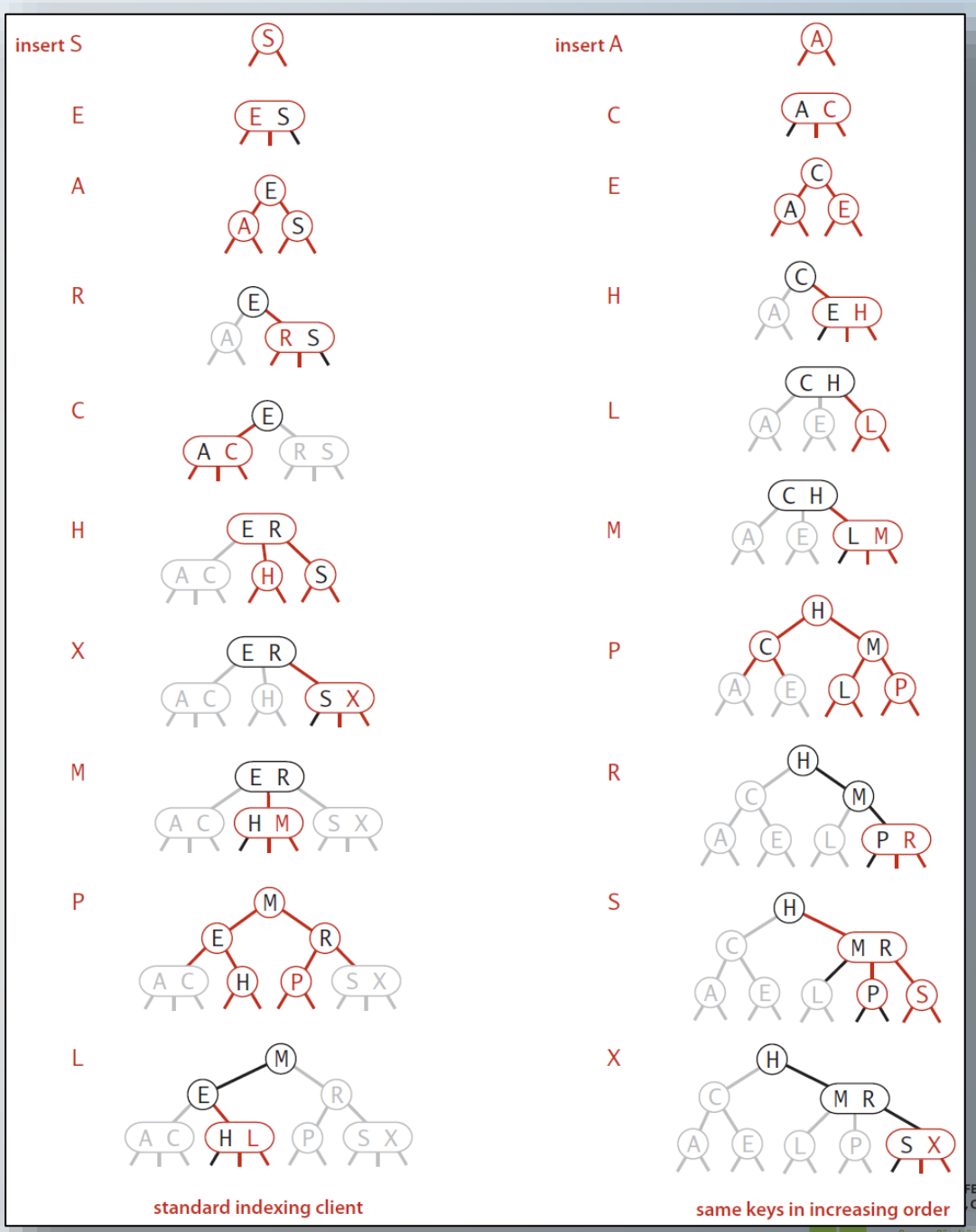
# 2-3 *Search Trees*
## Inserção de uma Nova Chave

- Propriedades globais de uma Árvore 2-3 de Busca:
  - Mantem ordem simétrica, ou seja, cada nó tem uma chave e a chave de cada nó é maior do que todas as chaves em sua subárvore esquerda e menor do que todas as chaves em sua subárvore direita e balanceamento perfeito, ou seja, todos os caminhos da raiz até uma referência nula tem o mesmo comprimento;
    - Cada transformação mantém a ordem simétrica e o balanceamento perfeito;



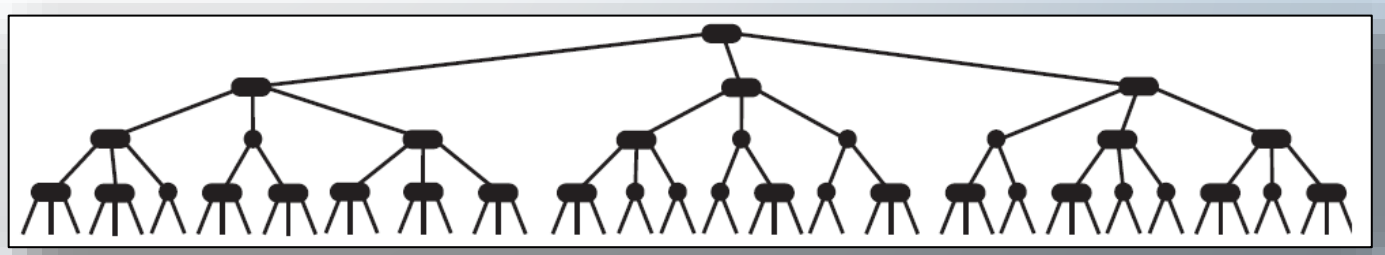
# 2-3 *Search Trees*

## Passo-a-passo da Construção



# 2-3 *Search Trees*
## Desempenho do Pior Caso

- Árvores 2-3 foram inventadas para garantir um bom desempenho no pior caso:
  - Se existe uma satisfação com um bom desempenho médio, basta usar uma ABB;
- Considere uma Árvore 2-3 com $n$ nós:
  - Se existem apenas nós simples, a altura da árvore será $\lfloor \lg n \rfloor$
  - Se existem apenas nós duplos, a altura da árvore será $\lfloor \log_3 n \rfloor$, que é igual a $\lfloor 0{,}63 \lg n \rfloor$
  - Conclusão: a altura nunca passa de $\lfloor \lg n \rfloor$
- Numa Árvore 2-3 com $n$ nós, a busca e a inserção nunca visitam mais que $\lg n$ nós, mesmo no pior caso:
  - Cada visita faz no máximo 2 comparações de chaves;
- Balanceamento perfeito:
  - Todos os caminhos da raiz até uma referência nula tem o mesmo comprimento.

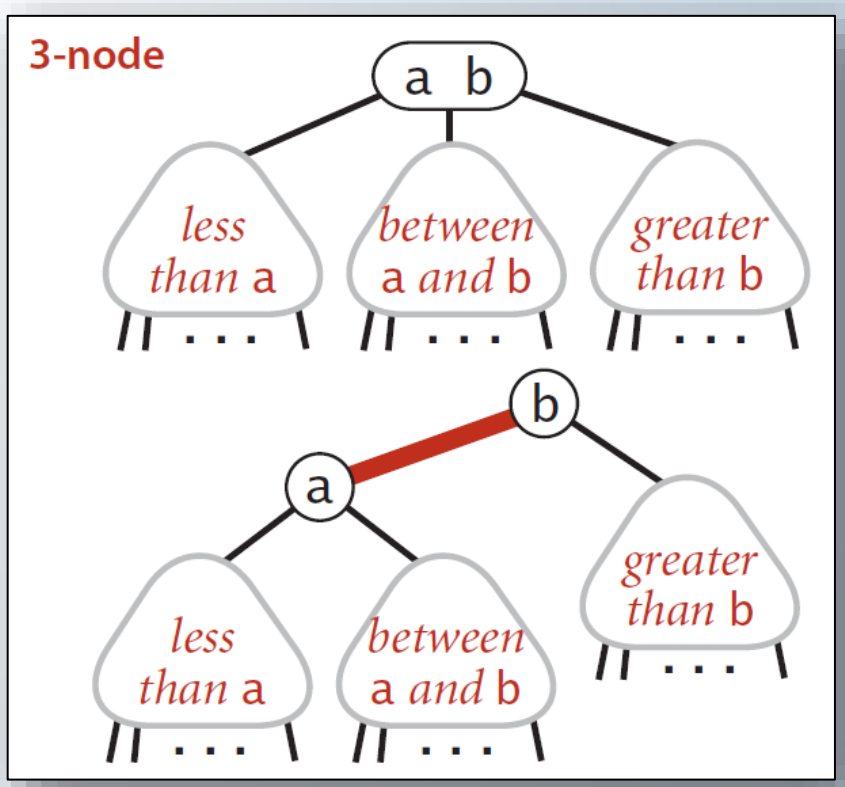


# 2-3 *Search Trees*

- Implementações diretas de Árvores 2-3 são complicadas:
  - Necessita de múltiplas comparações para descer a árvore;
  - É necessário subir de volta na árvore para dividir 4-nodes;
  - Ocorre um grande número de casos de divisão;
- Para implementar uma Árvore 2-3, deve-se utilizar nós de dois tipos:
  - Um com uma chave e duas referências;
  - Outro com duas chaves e três referências;
- Uma ideia melhor:
  - Usar ABBs para simular Árvores 2-3!

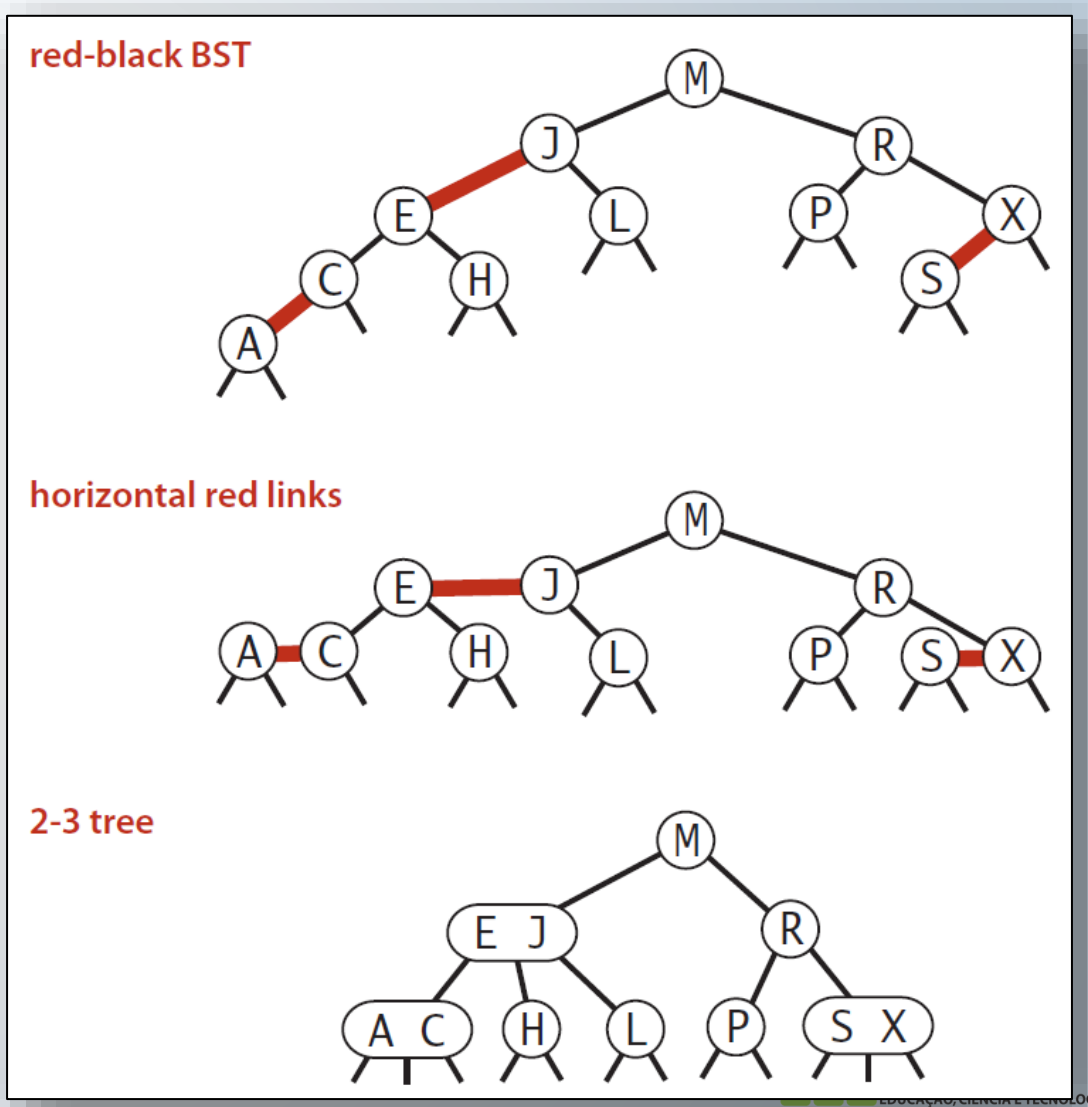
# Árvores Vermelho e Preto
## Introdução

- Uma Árvore Binária de Busca Vermelho e Preto / Rubro-Negras (Red-Black BST) é:
  - É uma ABB que simula uma Árvore 2-3;
  - Cada nó duplo da Árvore 2-3 é representado por dois nós simples ligados por uma referência vermelha;
  - São conhecidas como ABBs inclinadas à esquerda, pois as referências vermelhas são sempre inclinados para a esquerda;
- Representação de um nó duplo por meio de dois nós simples ligados por uma referência vermelha.



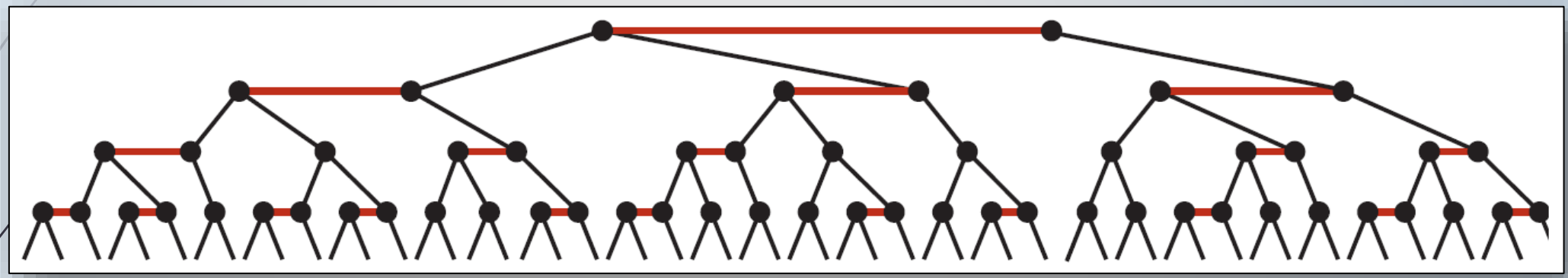
# Árvores Vermelho e Preto
## Propriedades

- Referências vermelhas se inclinam para a esquerda;
- Nenhum nó incide em duas referências vermelhas;
- Balanceamento preto perfeito:
  - Todo caminho da raiz até uma referência nula tem o mesmo número de referências pretas;
- Se as referências vermelhas forem desenhadas horizontalmente e depois contraídos, tem-se uma Árvore 2-3:



# Árvores Vermelho e Preto
## Propriedades

- Todos as referências nulas estão à mesma profundidade preta;



- A profundidade preta de um nó $x$ é o número de referências pretas no caminho da raiz até $x$.
- A altura preta da árvore é o máximo da profundidade preta de todos os nós.

# Árvores Vermelho e Preto
## Representação

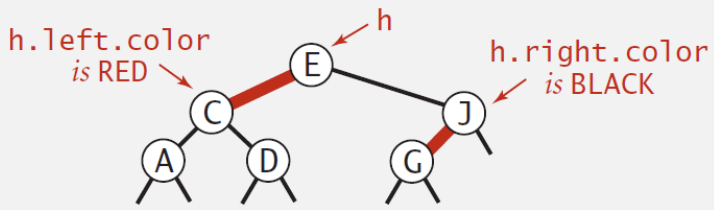
- É inconveniente armazenar a cor de uma referência na estrutura de dados, sendo mais simples armazenar essa informação nos nós como um atributo. A cor de um nó é a cor da única referência que entra nele.
- A raiz é considerada preta.

```
private static final boolean RED   = true;
private static final boolean BLACK = false;

private class Node
{
   Key key;
   Value val;
   Node left, right;
   boolean color;   // color of parent link
}

private boolean isRed(Node x)
{
   if (x == null) return false;
   return x.color == RED;
}
```

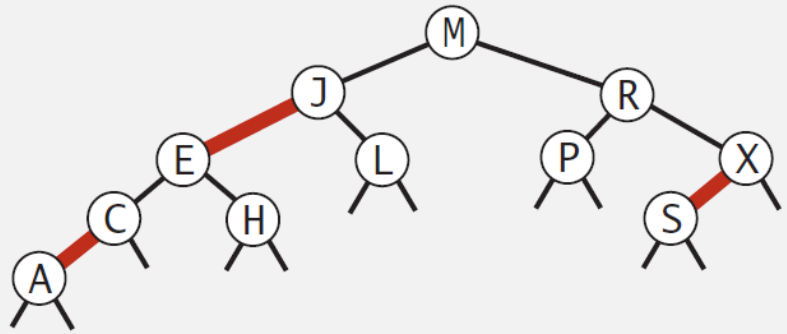
null links are black



# Árvores Vermelho e Preto
## Implementação da Busca

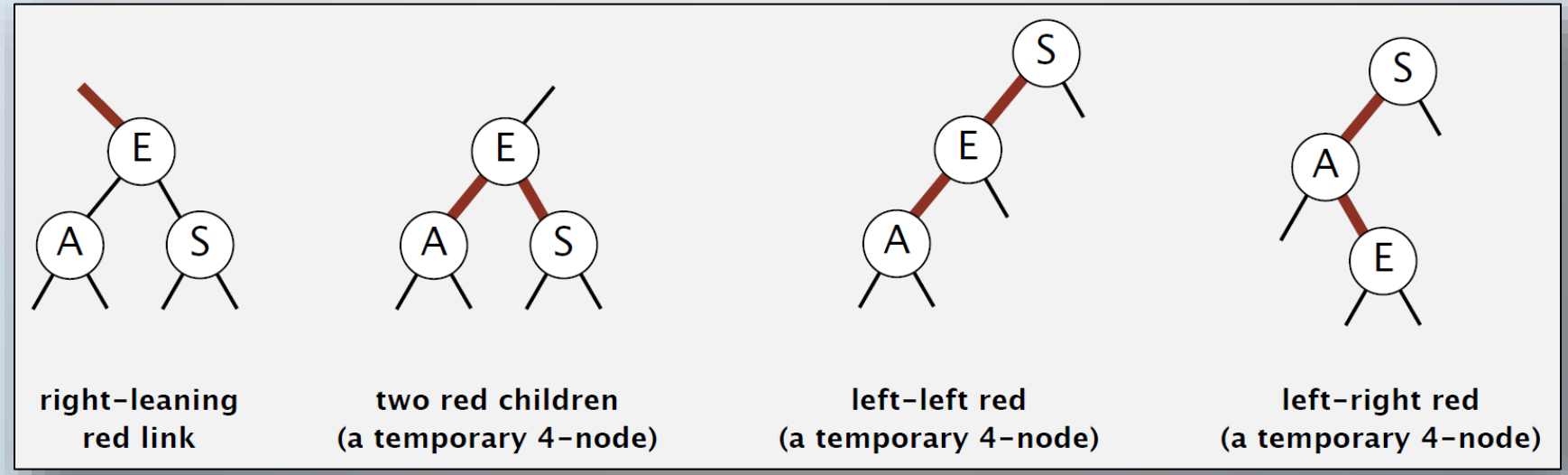
- O lógica de busca (`get`) para Árvores Vermelho e Preto é exatamente igual ao das ABBs comuns, ou seja, ignora-se as cores das referências
  - Por outro lado é mais rápida. Por quê?
    - As Árvores Vermelho e Preto tem um balanceamento melhor!

```
public Val get(Key key)
{
   Node x = root;
   while (x != null)
   {
      int cmp = key.compareTo(x.key);
      if      (cmp  < 0) x = x.left;
      else if (cmp  > 0) x = x.right;
      else if (cmp == 0) return x.val;
   }
   return null;
}
```



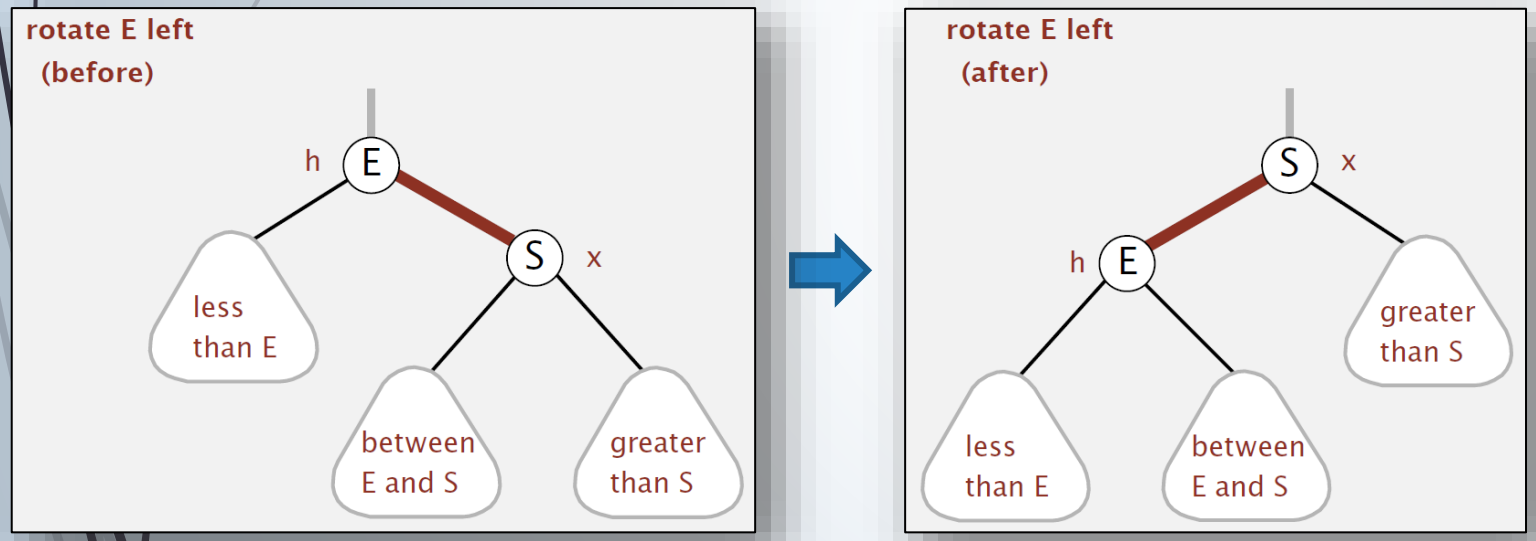
# Árvores Vermelho e Preto
## Implementação da Inserção

- O código de inserção (`put`) é complicado, pois depende de operações de rotação durante operações internas para manter:
  - A ordem simétrica;
  - O balanceamento preto perfeito;
- Durante uma operação de inserção, pode-se ter, temporariamente, uma referência vermelha inclinada para a direita ou duas referências vermelhas incidindo no mesmo nó;
- Para corrigir isso, utiliza-se de operações de rotações e troca de cores.



# Árvores Vermelho e Preto
## Implementação da Inserção - Rotações

- Rotação Esquerda (*Left Rotation*) ou Anti-Horária em torno de um nó $h$:
  - O filho direito de $h$ sobe e adota $h$ como seu filho esquerdo.



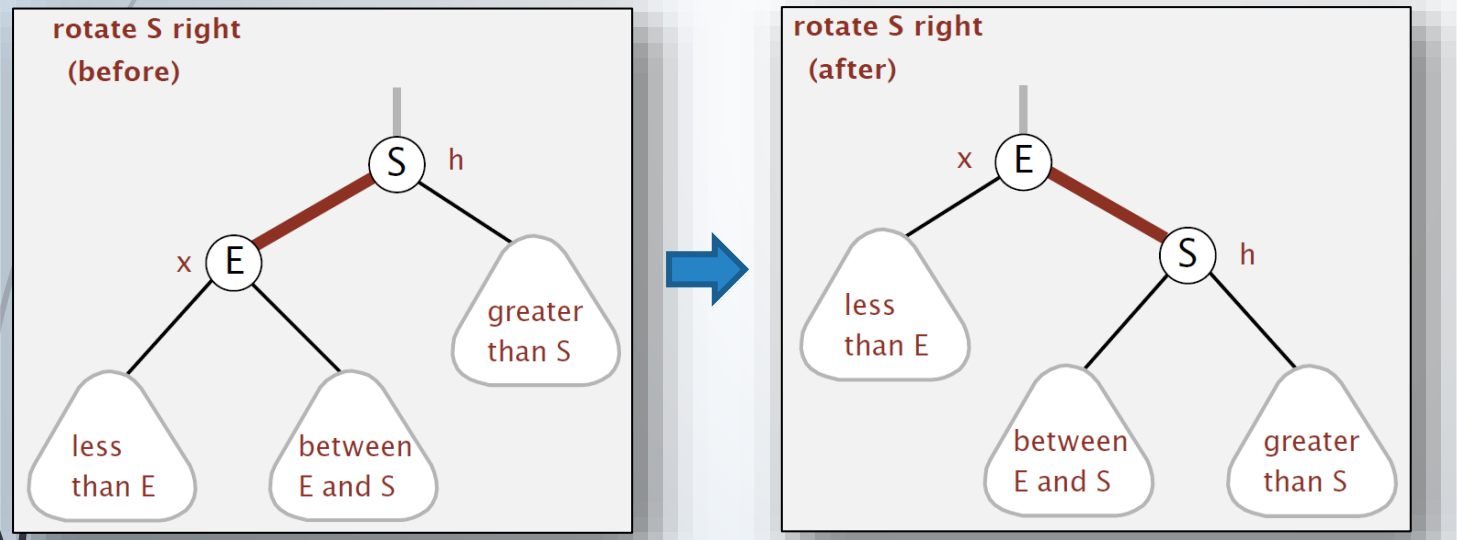




```
private Node rotateLeft(Node h)
{
   assert isRed(h.right);
   Node x = h.right;
   h.right = x.left;
   x.left = h;
   x.color = h.color;
   h.color = RED;
   return x;
}
```

# Árvores Vermelho e Preto
## Implementação da Inserção - Rotações

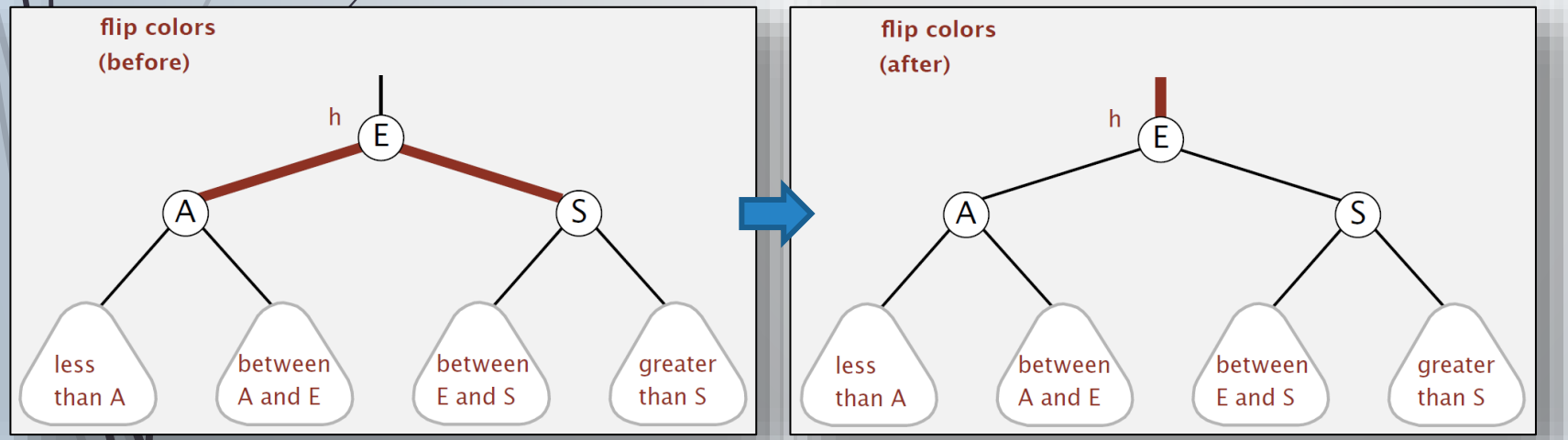
- Rotação Direita (*Right Rotation*) ou Horária em torno de um nó $h$:
  - O filho esquerdo de $h$ sobe e adota $h$ como seu filho direito.




```
private Node rotateRight(Node h)
{
   assert isRed(h.left);
   Node x = h.left;
   h.left = x.right;
   x.right = h;
   x.color = h.color;
   h.color = RED;
   return x;
}
```

# Árvores Vermelho e Preto
## Implementação da Inserção - Rotações

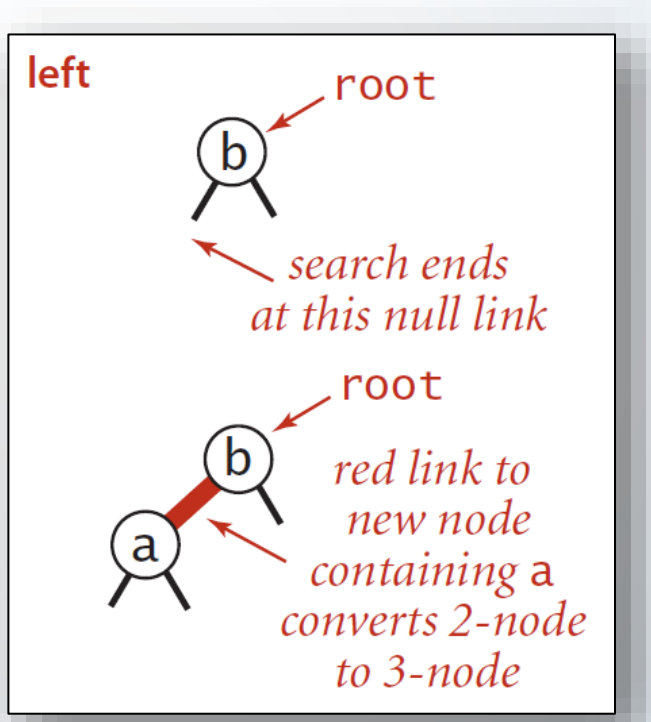
- Inversão de Cores (*Color Flip*) em torno de um nó $h$:
  - A inversão de cores implementa a divisão de um 4-node temporário na Árvore 2-3;
  - A cor vermelha é retirada das duas referências que partem de $h$ e passa para a referência que entra em $h$. Diz-se que a cor vermelha "passa para cima".

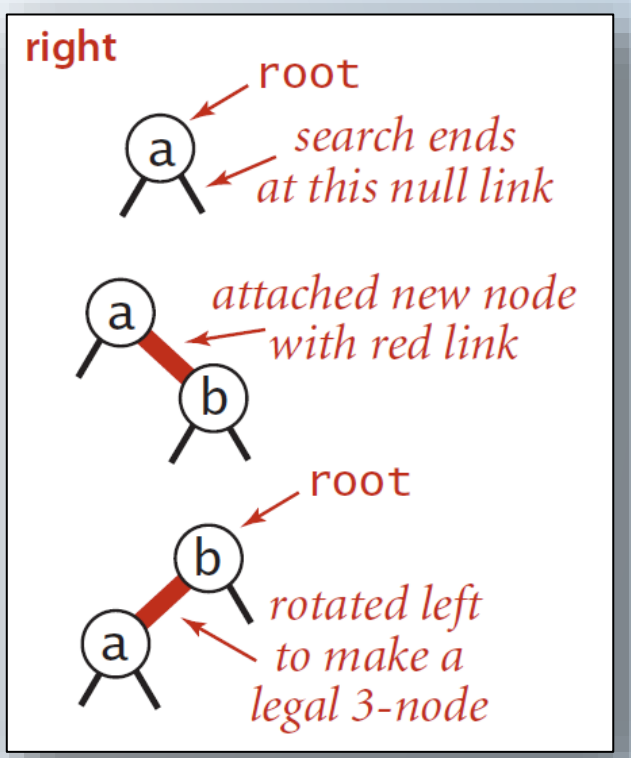



```
private void flipColors(Node h)
{
   assert !isRed(h);
   assert isRed(h.left);
   assert isRed(h.right);
   h.color = RED;
   h.left.color = BLACK;
   h.right.color = BLACK;
}
```

# Árvores Vermelho e Preto
## Inserção de um Novo Nó

- Um novo nó sempre se liga à árvore por uma referência vermelha.
- **Exemplo:** Suponha que a árvore só tem a raiz:
  - Se o novo nó é pendurado à esquerda da raiz, não é preciso fazer mais nada;
  - Se o novo nó é pendurado à direita da raiz, é preciso fazer uma rotação para a esquerda (anti-horária) em torno da raiz, ou seja, `root=rotateLeft(root)`, para que a árvore fique inclinada à esquerda:

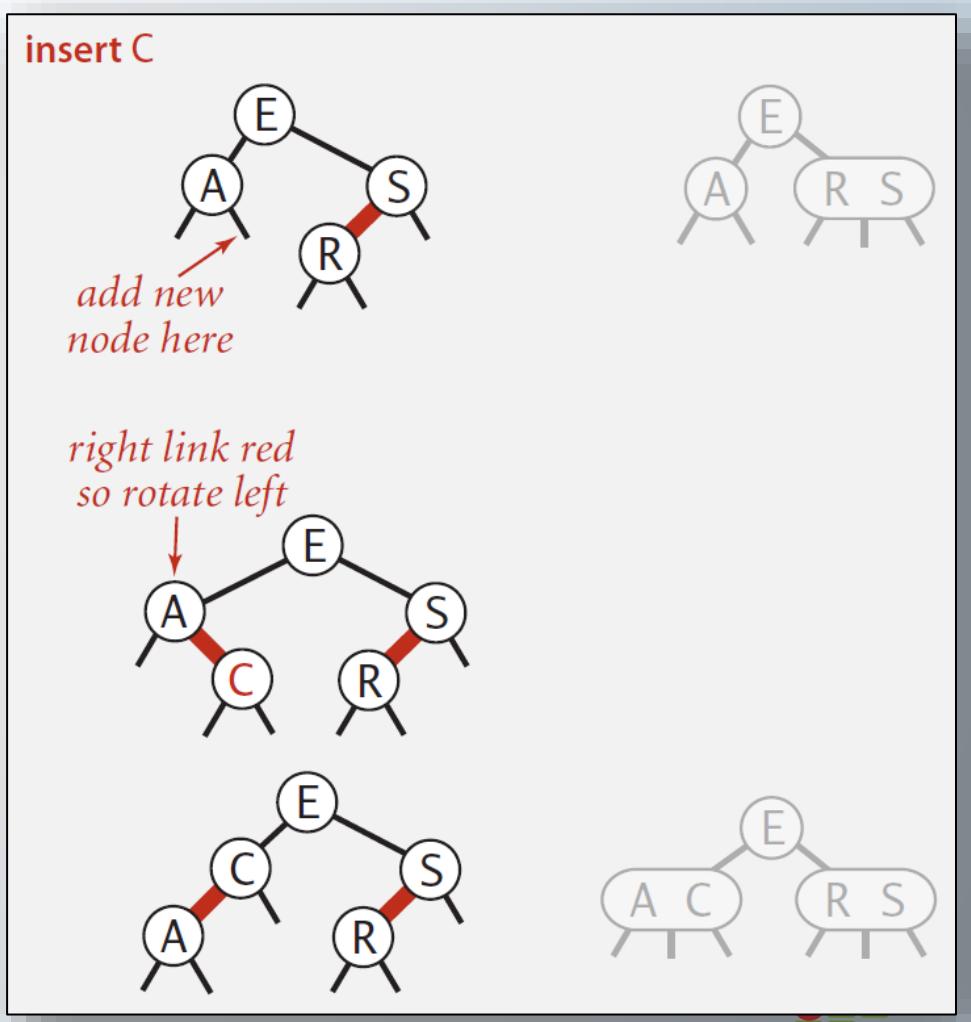


# Árvores Vermelho e Preto

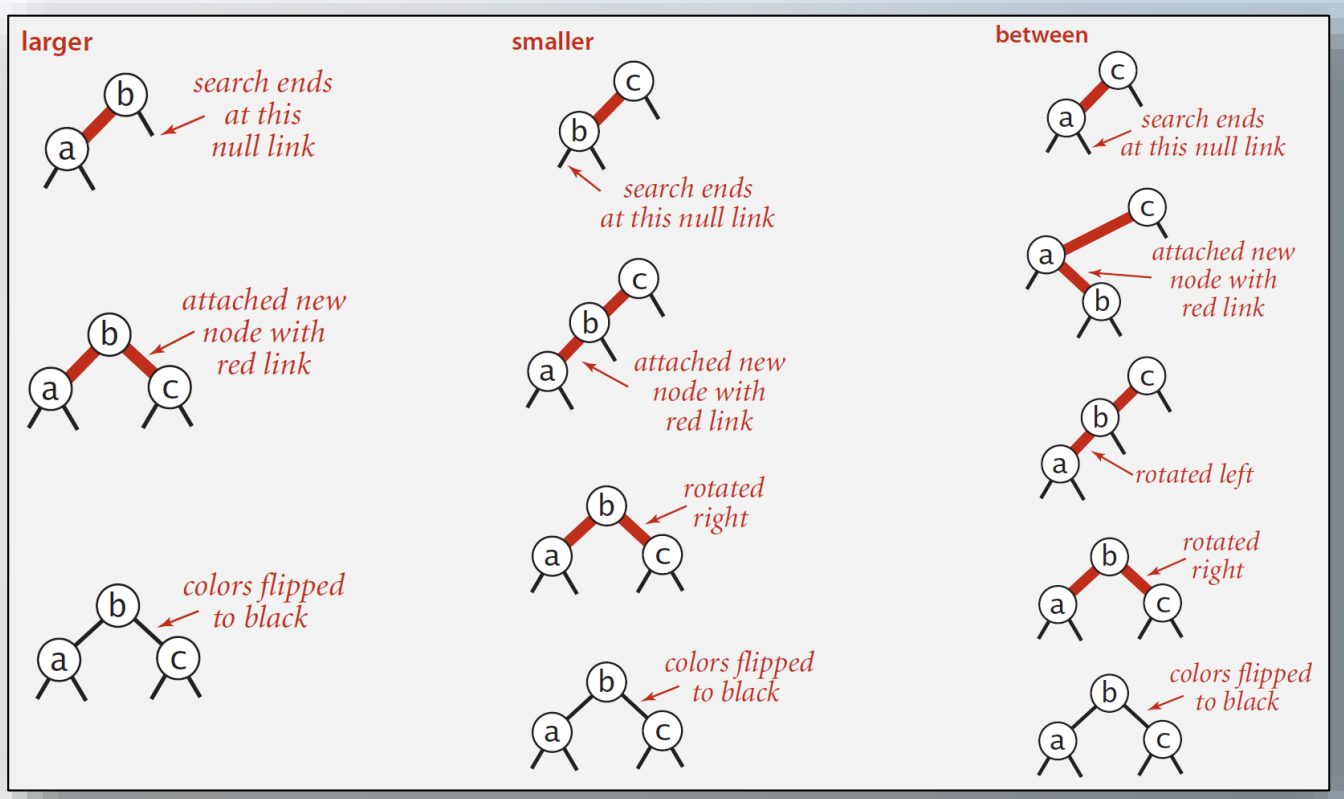
## Inserção de um Novo Nó

- **Caso 01:** Inserir em um 2-node:
  - Pendurar um novo nó em um nó simples, ou seja, um nó que não incide em uma referência vermelha;
  - Fazer a inserção padrão da ABB, colorindo a nova referência de vermelho;
  - Se a nova referência vermelha for para a direita, realizar a rotação esquerda.



# Árvores Vermelho e Preto
## Inserção de um Novo Nó

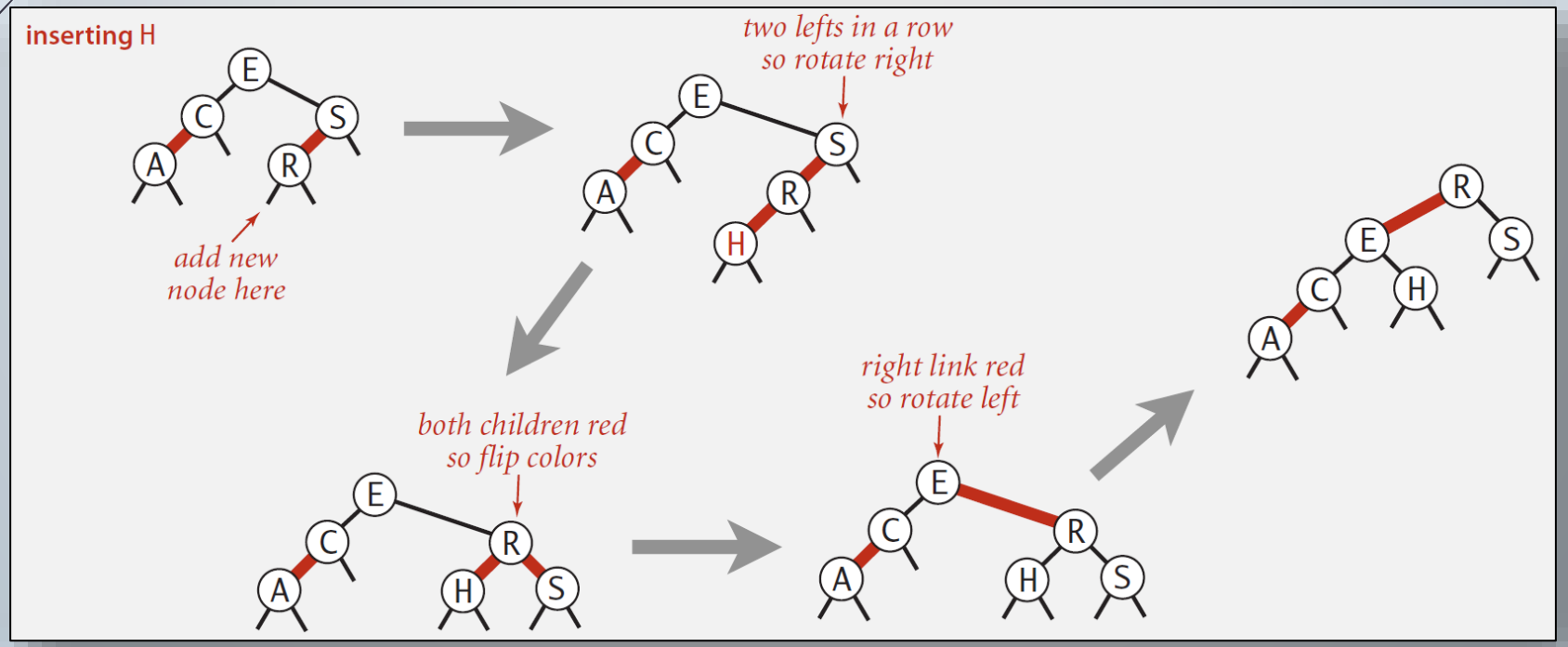
- Inserir uma árvore com dois nós:
  - Pendurar um novo nó em um dado nó duplo, isto é, em um par de nós ligados por uma referência vermelha;
  - Dependendo da chave, o novo nó pode ser pendurado em três lugares;
  - Em cada caso, pendure por uma referência vermelha e depois use rotações e inversão de cores para restabelecer as propriedades que caracterizam a Árvore Vermelho e Preta.




**Resumindo:** toda inserção de um novo nó precisa de zero, uma, ou duas rotações seguidas de uma inversão de cores.

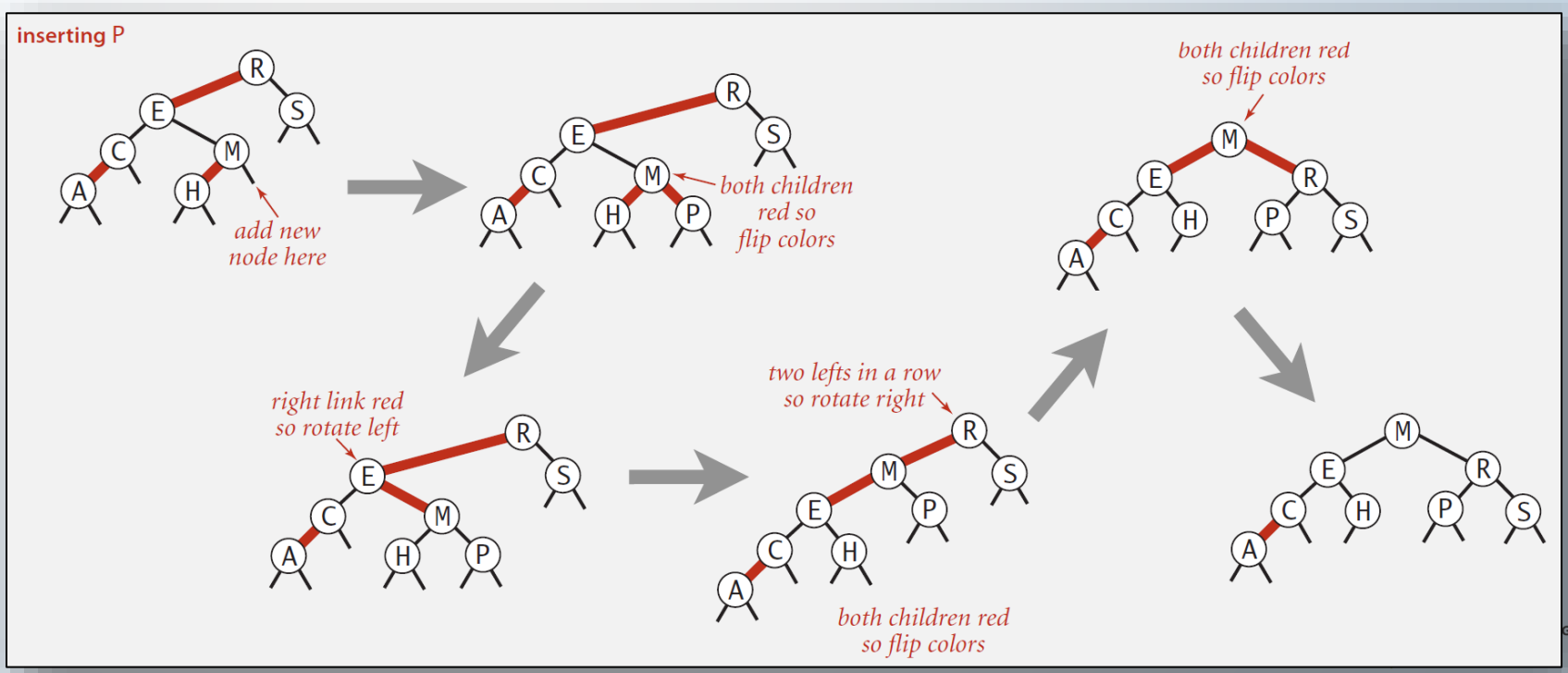
# Árvores Vermelho e Preto
## Inserção de um Novo Nó

- **Caso 02:** Inserir em um 3-node:
  - Fazer a inserção padrão da ABB colorindo a nova referência de vermelho;
  - Rotacionar para equilibrar o 4-node se necessário;
  - Inverter as cores para subir a referência vermelha para um nível acima.
  - Se necessário, rotacionar para inclinar para a esquerda.



# Árvores Vermelho e Preto
## Inserção de um Novo Nó

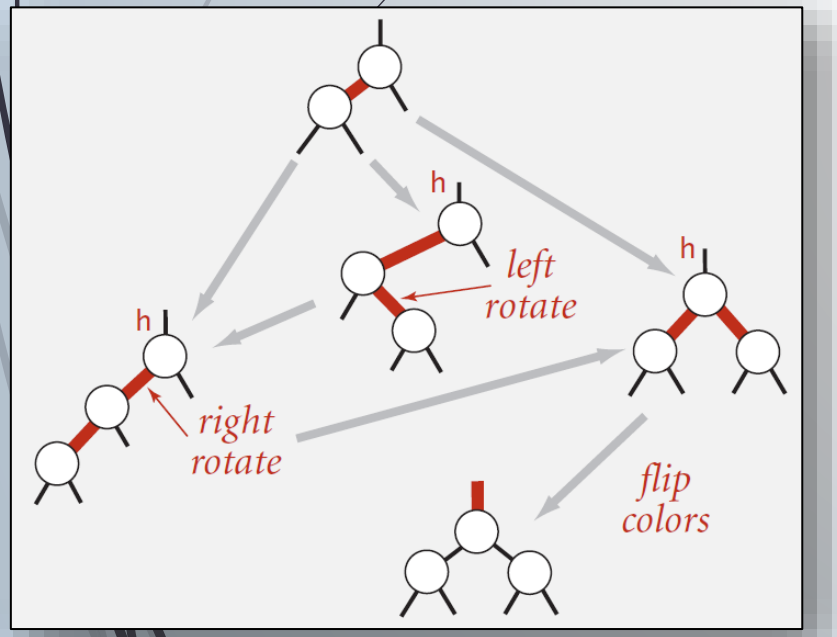
- **Caso 02:** Inserir em um 3-node:
  - Fazer a inserção padrão da ABB colorindo a nova referência de vermelho;
  - Rotacionar para equilibrar o 4-node se necessário;
  - Inverter as cores para subir a referência vermelha para um nível acima.
  - Se necessário, rotacionar para inclinar para a esquerda.
  - Repetir o **Caso 01** ou o **Caso 02** na árvore se necessário.



# Árvores Vermelho e Preto
## Implementação da Inserção

**put() é recursivo:** a operação desce pela árvore até achar o nó certo e depois volta, desempilhando as chamadas recursivas e fazendo as rotações e inversões de cores que forem necessárias.

- O método `put()` é o mesmo para todos os casos:
  - Se filho direito é vermelho e o filho esquerdo preto: Rotação Esquerda
  - Se filho esquerdo e neto esquerdo-esquerdo são vermelhos: Rotação Direita
  - Se dois filhos são vermelhos: Inversão de Cores




```
private Node put(Node h, Key key, Value val)
{
   if (h == null) return new Node(key, val, RED);
   int cmp = key.compareTo(h.key);
   if      (cmp  < 0) h.left  = put(h.left,  key, val);
   else if (cmp  > 0) h.right = put(h.right, key, val);
   else if (cmp == 0) h.val = val;

   if (isRed(h.right) && !isRed(h.left))      h = rotateLeft(h);
   if (isRed(h.left)  && isRed(h.left.left)) h = rotateRight(h);
   if (isRed(h.left)  && isRed(h.right))      flipColors(h);

   return h;
}
```

insert at bottom (and color it red)

lean left

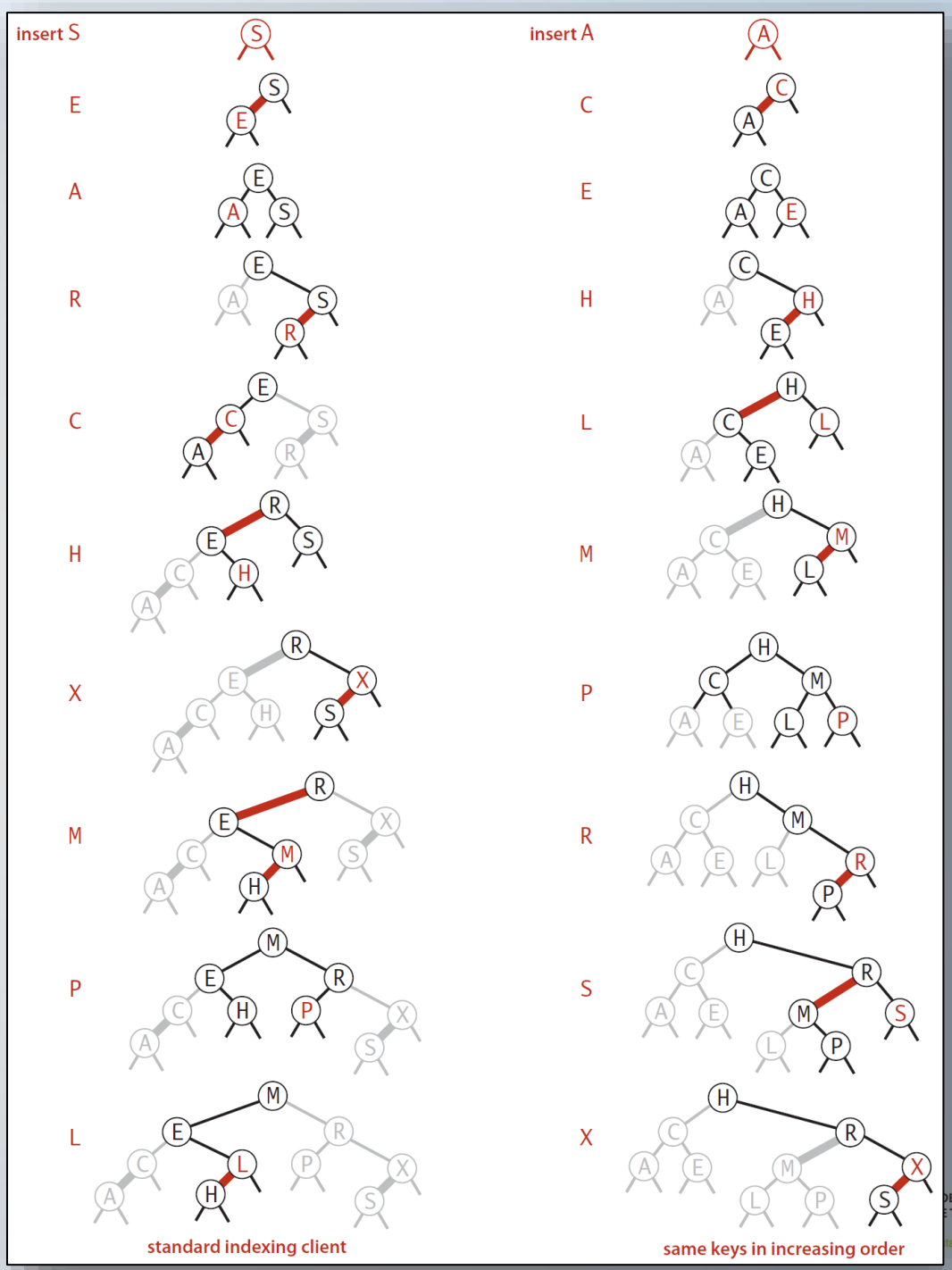
balance 4-node

split 4-node

only a few extra lines of code provides near-perfect balance

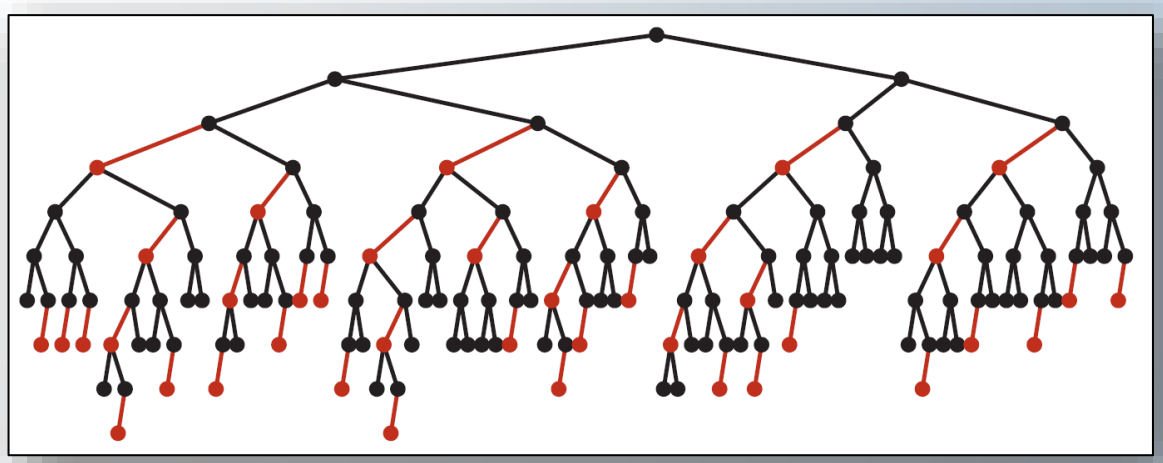
# Árvores Vermelho e Preto

## Passo-a-passo da Construção



# Árvores Vermelho e Preto
## Desempenho do Pior Caso

- A altura de uma Árvore Vermelho e Preta com $n$ nós é menor ou igual a $\lg n$
  - Refere-se a altura total, não da altura preta;
- Esboço da prova:
  - Pior caso é uma Árvore 2-3 que só tem nós duplos no caminho que desce pela borda esquerda da árvore;
  - Como nenhum nó incide em duas referências vermelhas, esse caminho pela borda extrema esquerda é apenas duas vezes mais longo que os caminhos que só passam por nós simples.



- Em cada operação sobre uma Árvore Vermelho e Preta com $n$ nós, o número de comparações entre chaves é limitado por $\lg n$ multiplicado por uma constante pequena, ou seja, o consumo de tempo de cada operação é logarítmico, mesmo no pior caso.

# B-*Trees*
## Introdução

- Árvores B (B-*Trees*) são estruturas usadas para implementar tabelas de símbolos muito grandes;
- Uma Árvore B pode ser vista como um índice -análogo ao índice de um livro- para uma coleção de pequenas tabelas de símbolos: o índice diz em qual das pequenas tabelas de símbolos está a chave que se deseja buscar.
  - Pode-se dizer que uma Árvore B é uma "tabela de símbolos de tabelas de símbolos".

| chave | valor |
|---|---|
| www.ebay.com | 66.135.192.87 |
| www.princeton.edu | 128.112.128.15 |
| www.cs.princeton.edu | 128.112.136.35 |
| www.harvard.edu | 128.103.60.24 |
| www.yale.edu | 130.132.51.8 |
| www.cnn.com | 64.236.16.20 |
| www.google.com | 216.239.41.99 |
| www.nytimes.com | 199.239.136.200 |
| www.apple.com | 17.112.152.32 |
| www.slashdot.org | 66.35.250.151 |
| www.espn.com | 199.181.135.201 |
| www.weather.com | 63.111.66.11 |
| www.yahoo.com | 216.109.118.65 |
| ... | ... |
| www.ime.usp.br | 143.107.45.37 |

# B-*Trees*
## Introdução

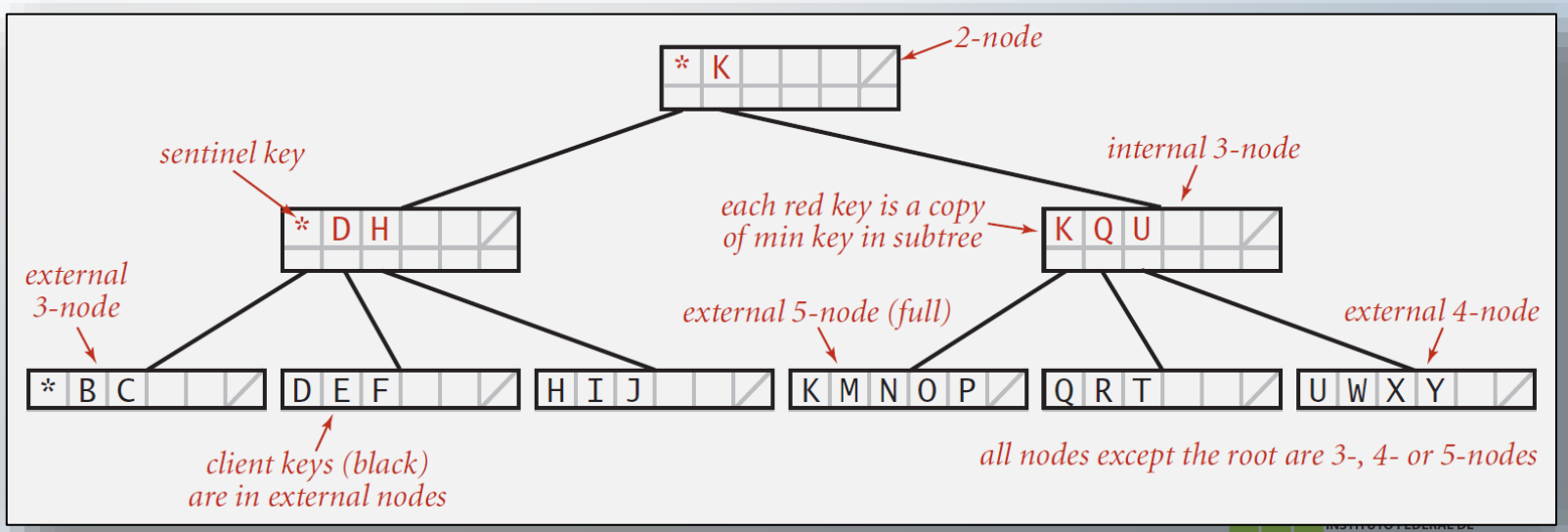
- Dois tipos de memória de um computador típico: a memória interna (ou principal) e a memória externa (ou secundária, usualmente um disco rígido ou memória de estado sólido)
  - A memória interna tem alta velocidade, mas pouca capacidade enquanto a memória externa tem baixa velocidade, mas grande capacidade;
- Dada uma tabela de símbolos ordenada $T$, imagine que $T$ tem um número enorme de chaves, tão grande que não cabe na memória rápida do computador:
  - Para manipular essa tabela de símbolos é preciso dividi-la em segmentos $T_1, T_2, \dots, T_n$, cada segmento sendo uma tabela de símbolos pequena o suficiente para caber com folga na memória rápida;
  - É natural definir os segmentos de modo que todas as chaves em um segmento sejam menores que todas as chaves no segmento seguinte;
- Para procurar uma chave nessa tabela de símbolos fracionada, precisa-se de um índice, ou seja, uma tabela de símbolos auxiliar que indique em qual dos segmentos $T_1, T_2, \dots, T_n$ está a chave procurada:
  - A tabela de símbolos auxiliar é interna e as tabelas de símbolos $T_1, T_2, \dots, T_n$ são externas.

# B-*Trees*
## Introdução

- A tabela de símbolos interna $A$ é organizada da seguinte maneira:
  - As chaves de $A$ são $k_1, k_2, \ldots, k_n$, sendo $k_i$ a menor chave de $T_i$;
  - O valor que $A$ associa com a chave $k_i$ é uma referência a $Ti$;
- Para procurar por uma chave $x$, escolha o maior $i$ tal que $k_i \leq x$, use $A$ para localizar $T_i$, e então procure por $x$ em $T_i$. Se um tal $i$ não existe, $x$ não está em $T$;
- O que acontece se $A$ for grande demais para caber na memória rápida? Em outras palavras, o que acontece se $A$ for muito maior que qualquer $T_i$?
  - Nesse caso, aplique a $A$ a mesma ideia de segmentação que aplicou-se a $T$ e repita, recursivamente, se necessário. Isso resulta em uma árvore cujos nós são tabelas de símbolos, originando assim o conceito de Árvore B;
- Convém impor restrições ao número de chaves em cada nó da árvore:
  - Escolha um inteiro positivo par $M$ e organize as coisas de modo que cada nó da árvore tenha no máximo $M - 1$ chaves e no mínimo $M/2$ chaves
    - A raiz é excepcional: basta que ela tenha duas ou mais chaves.

# B-*Trees*
## Exemplo com M = 6

- Exemplo de uma Árvore B com $M = 6$:
  - As chaves são `*, B, C, … , X, Y`
  - As chaves são comparadas em ordem alfabética, assim, * é a menor das chaves
    - Um nó com $m$ chaves é chamado m-nó;
- No exemplo utiliza-se um valor pequeno de $M$, como 4 ou 6:
  - Um valor mais realista pode ficar em torno de 1000;
  - Na prática, cada nó é um arquivo, armazenado na memória lenta, ou uma página na web por exemplo.



# B-*Trees*
## Definição

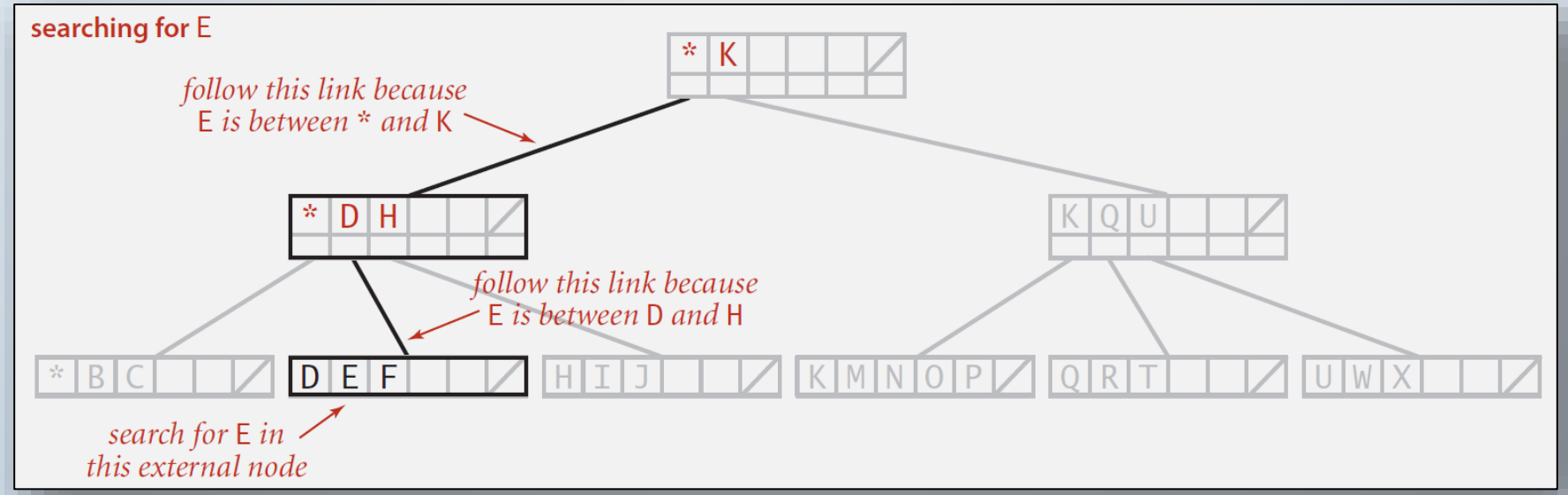
- **Definição:** Para qualquer inteiro positivo par $M$, uma Árvore B de ordem $M$ é uma árvore com as seguintes propriedades:
  - Cada nó contém no máximo $M - 1$ chaves;
  - A raiz contém, no mínimo, 2 chaves e cada um dos demais nós contém no mínimo $M/2$ chaves;
  - Cada nó que não seja uma folha tem um filho para cada uma de suas chaves;
  - Todos os caminhos da raiz até uma folha têm o mesmo comprimento, ou seja, é perfeitamente balanceada;
- Uma Árvore B de ordem 4 é essencialmente uma Árvore 2-3, embora existam algumas diferenças que abordaremos a seguir.

# B-*Trees*
## Aplicações

- Árvores B são a estrutura subjacente a muitos sistemas de arquivos e bancos de dados;
- Por exemplo:
  - O sistema de arquivos NTFS do Windows;
  - O sistema de arquivos HFS do macOS;
  - Os sistemas de arquivos ReiserFS, XFS, Ext3FS, JFS do Linux;
  - Os bancos de dados ORACLE, DB2, INGRES, SQL e PostgreSQL.

# B-*Trees*
## Buscar Chave

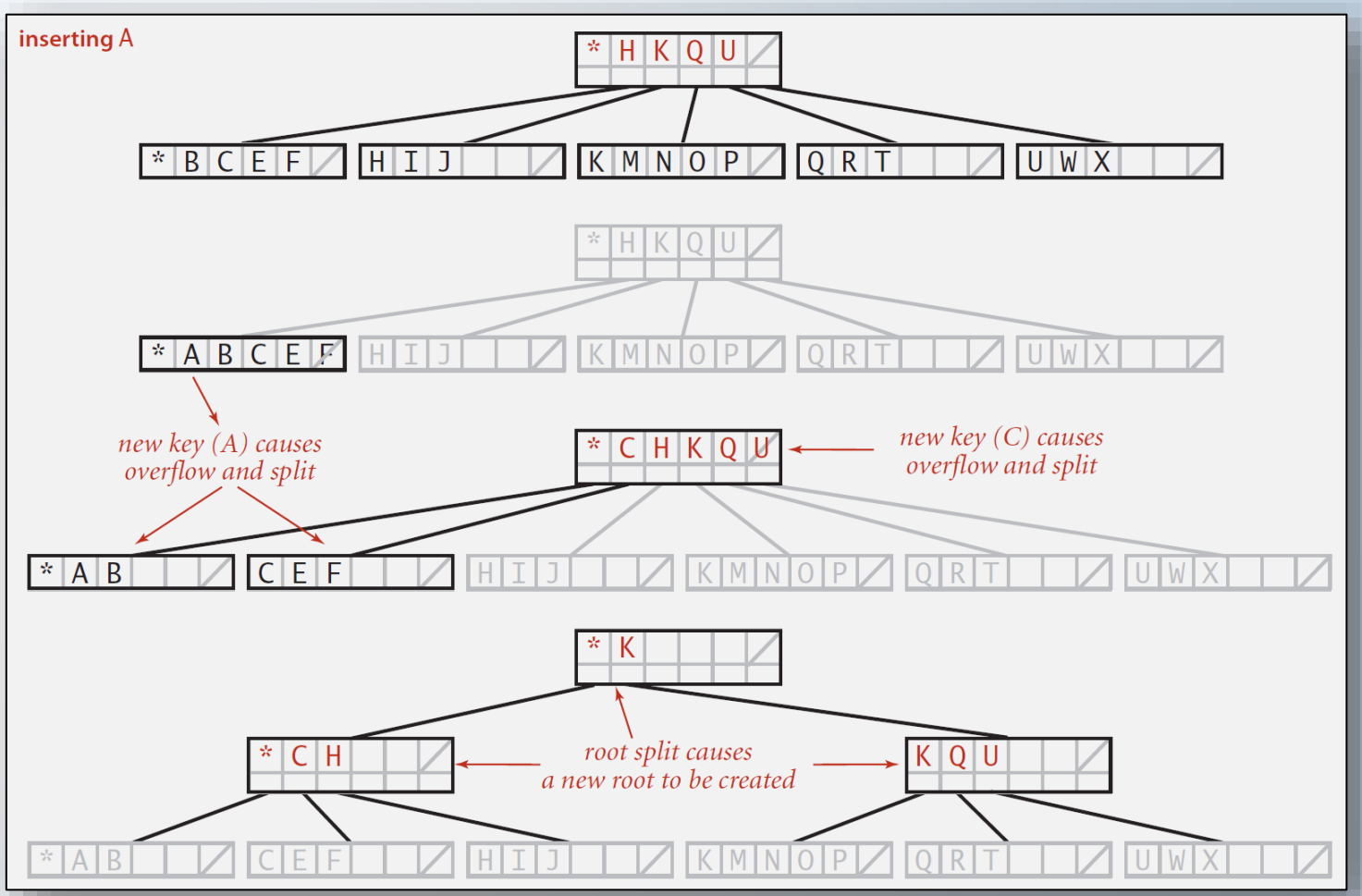
- Operação de Busca (semelhante à busca em ABBs):
  1. Inicia a pesquisa na raiz;
  2. Encontra o intervalo da chave pesquisada e caminha pela referência correspondente;
  3. A busca termina em um nó externo;
- **Exemplo:** Busca pela chave `E` em uma Árvore B de ordem 6 que armazena as chaves `*, B, C, … , W, X`



# B-*Trees*

## Inserir Nova Chave

- Operação de Inserção:
  1. Pesquisa pela nova chave;
  2. Insere na parte inferior da árvore;
  3. Divide os nós em $M$ pares `(chave, referência)` na subida da árvore;
- **Exemplo:** Inserção da chave uma Árvore B de ordem 6 que armazena as chaves `*, B, C, … , W, X`



# B-*Trees*
## Análise do Desempenho

- O consumo de tempo de cada operação na memória rápida é muito menor que o tempo necessário para trazer uma página da memória lenta para a rápida ou levar uma página da memória rápida para a lenta
  - Assim, faz sentido ignorar o custo das operações na memória rápida e contar apenas o número de sondagens;
  - Uma sondagem é o primeiro acesso a uma página durante uma busca ou inserção;
- **Proposição:** Uma busca ou inserção em uma Árvore B de ordem $M$ com $n$ chaves envolve entre $\log_m n$ e $\log_{\frac{m}{2}} n$ sondagens.
- **Prova:** O número de sondagens é igual à altura da árvore:
  - No melhor caso, todas as páginas internas têm $M - 1$ filhos;
  - No pior caso, todas as páginas internas, exceto a raiz, têm $M/2$ filhos;
- Se $M$ é da ordem de $1000$, por exemplo, a altura da árvore não passa de 4 se $n$ for menor que 62 bilhões.

# B-*Trees*
## Construindo uma Árvore Grande

- Evolução do conjunto de páginas externas de uma Árvore B de ordem 8:
  - Em cada linha da Figura é apresentado o resultado da inserção de uma chave em alguma página externa;
  - Cada pequena barra horizontal representa uma página externa: a porção preta da barra é a parte ocupada da página e a porção branca é a parte desocupada;
  - Cada barra vermelha representa uma página externa cheia que está prestes a receber uma nova chave e transbordar.
  - Começamos com uma só página externa (cheia) e terminamos com 22 páginas externas.

# Bibliografia

SEDGEWICK, R.; WAYNE, K. **Algorithms**. 4. ed. Boston: Pearson Education, 2011. 955 p.

GOODRICHM M. T.; TAMASSIA, R. **Estruturas de Dados & Algoritmos em Java**. Porto Alegre: Bookman, 2013. 700 p.

CORMEN, T. H.; LEISERSON, C. E.; RIVEST, R. L.; STEIN, C. **Algoritmos – Teoria e Prática**. 3. ed. São Paulo: GEN LTC, 2012. 1292 p.