Técnicas de Busca e Ordenação

Árvores de Busca Balanceadas

Departamento de Informática (DI)
Centro Tecnológico (CT)
Universidade Federal do Espírito Santo (UFES)

(Material baseado nos slides dos Professores Eduardo Zambon e Giovanni Comarela)

Introdução

- Como já visto na aula anterior, uma estrutura muito usada é a árvore binária de busca *(BST – binary search tree)*.
- O desempenho das operações na BST está diretamente ligado à altura da árvore.
- **Aula de hoje**: mostrar variações de árvores com altura auto-ajustável (balanceadas).
- **Objetivos**: compreender as aplicações e o funcionamento de árvores 2-3, árvores rubro-negras e árvores B.

Referências

Chapter 13 – Balanced Trees
R. Sedgewick

Implementações de tabelas de símbolos: sumário

implementation	guarantee			average case			ordered ops?
	search	insert	delete	search hit	insert	delete	
sequential search (unordered list)	N	N	N	$\frac{1}{2} N$	N	$\frac{1}{2} N$	
binary search (ordered array)	$\lg N$	N	N	$\lg N$	$\frac{1}{2} N$	$\frac{1}{2} N$	✔
BST	N	N	N	$1.39 \lg N$	$1.39 \lg N$	$\sqrt{N}$	✔
goal	$\log N$	$\log N$	$\log N$	$\log N$	$\log N$	$\log N$	✔

Desafio: Desempenho logarítmico garantido para todas as operações.

Part I

Árvores 2-3

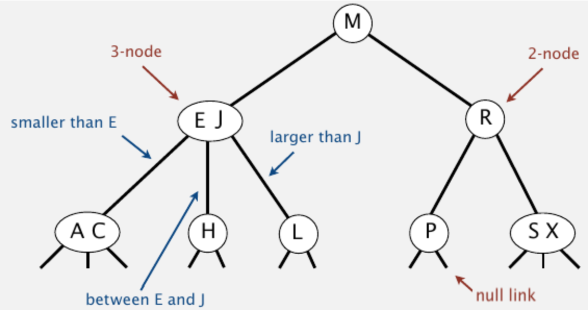
Árvore 2-3 (*2-3 tree*)

Permite 1 ou 2 chaves por nó.

- *2-node*: uma chave, dois filhos.
- *3-node*: duas chaves, três filhos.

Ordem simétrica: Caminhamento *in-order* visita as chaves em ordem crescente.

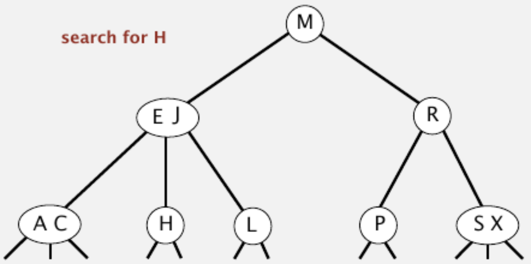
Balanceamento perfeito: Todo caminho da raiz para as folhas tem o mesmo tamanho. (Como manter?)



Árvore 2-3: busca

Busca:

- Compara chave buscada com as chaves no nó.
- Determina o intervalo contendo a chave buscada.
- Percorre recursivamente o ponteiro correspondente.

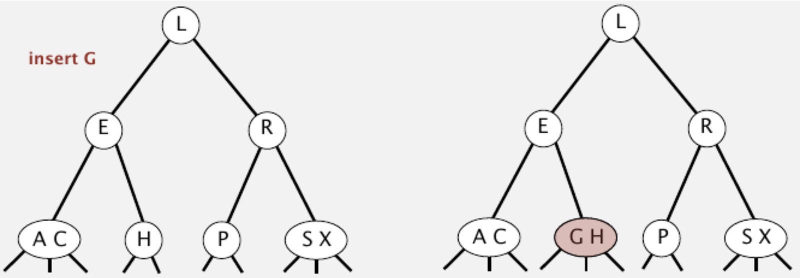


Veja o vídeo `33Demo23Tree.mov` entre 00:00 a 00:13 s.

Árvore 2-3: inserção

Inserção em um *2-node* na base:

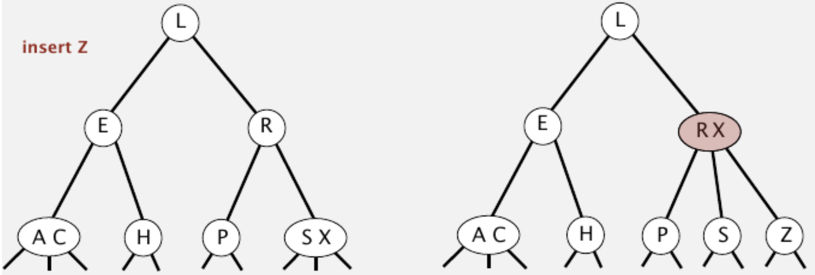
- Adicione a nova chave em um *2-node* para formar um *3-node*.



Veja o vídeo `33Demo23Tree.mov` entre 00:13 a 00:18 s.

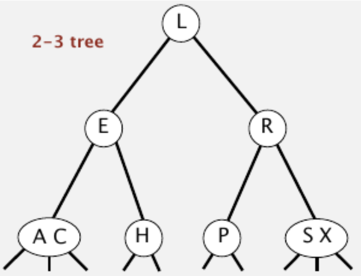
Inserção em um *3-node* na base:

- Adicione a nova chave em um *3-node* para formar um *4-node* temporário.
- Mova a chave do meio do *4-node* para o pai.
- Continue a propagação para cima, conforme necessário.
- Se chegar na raiz e é um *4-node*, divida em três *2-nodes*.



Veja o vídeo `33Demo23Tree.mov` entre 00:18 a 00:41 s.

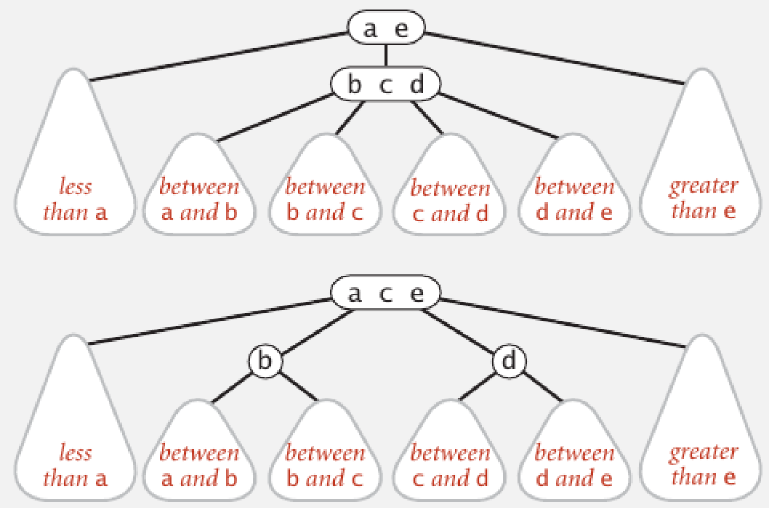
Árvore 2-3: construção



Veja o vídeo `33Demo23Tree.mov` de 00:41 s em diante.

Transformações locais em uma árvore 2-3

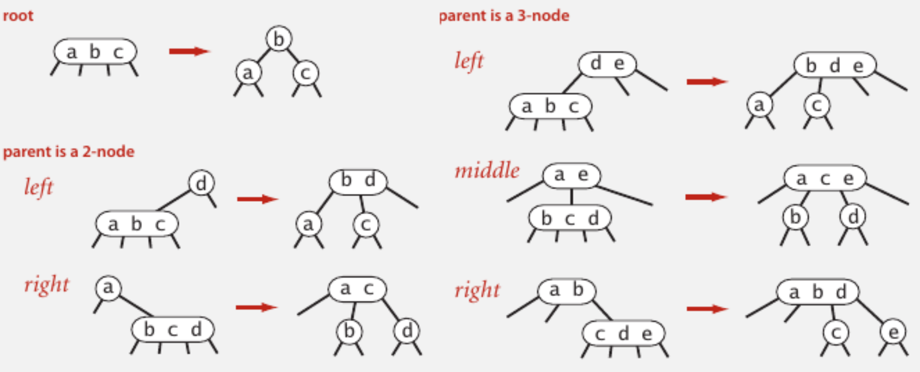
Dividir um *4-node* é uma operação local: número constante de operações.



Propriedades globais em uma árvore 2-3

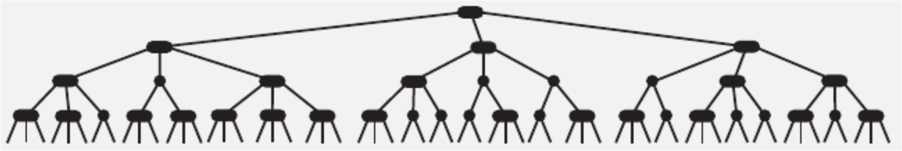
Invariantes: Manter a ordenação e o balanceamento.

Argumento: transformações na árvore mantém os invariantes.



Árvore 2-3: desempenho

Balanceamento perfeito: Todo caminho da raiz para as folhas tem o mesmo tamanho.



Altura da árvore.

- Pior caso: $\lg N$ (todos *2-nodes*).
- Melhor caso: $\log_3 N \approx .631 \lg N$ (todos *3-nodes*).
- Altura para 1 milhão (10^6) de nós: entre 12 e 20.
- Altura para 1 bilhão (10^9) de nós: entre 18 e 30.

Conclusão: Desempenho logarítmico garantido para as ops.

Implementações de tabelas de símbolos: sumário

implementation	guarantee			average case			ordered ops?
	search	insert	delete	search hit	insert	delete	
sequential search (unordered list)	N	N	N	$\frac{1}{2} N$	N	$\frac{1}{2} N$	
binary search (ordered array)	$\lg N$	N	N	$\lg N$	$\frac{1}{2} N$	$\frac{1}{2} N$	✔
BST	N	N	N	$1.39 \lg N$	$1.39 \lg N$	$\sqrt{N}$	✔
2-3 tree	$c \lg N$	$c \lg N$	$c \lg N$	$c \lg N$	$c \lg N$	$c \lg N$	✔

constant c depend upon implementation

Árvore 2-3: implementação?

Implementação direta de uma árvore 2-3 é complicada:

- Manter tipos diferentes de nós é complicado.
- Múltiplas comparações para descer na árvore.
- Subir na árvore para dividir *4-nodes*.
- Vários casos distintos de divisão.

Código de fantasia:

```
void put(Key key, Value val) {
    // Search key in tree, starting from root.
    // Let n be the node where key is to be inserted.
    if      (n == NULL)    { n = make_2node(key, val); }
    else if (is_2node(n)) { n = make_3node(n, key, val); }
    else if (is_3node(n)) { n = make_4node(n, key, val); }
    while (is_4node(n)) {
        n = split(n);
    }
}
```

Conclusão: possível de implementar mas há um jeito melhor.

Part II

Árvores rubro-negras

Como implementar árvores 2-3 com árvores binárias?

Desafio: como representar um *3-node*?

Método 1: BST comum.

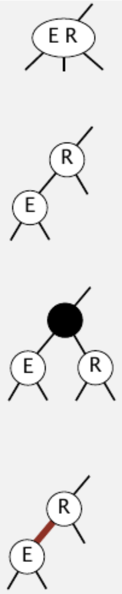
- Não há como distinguir um *3-node* de um *2-node*.
- Não dá para mapear de volta da BST para a árvore 2-3.

Método 2: BST comum com nós “cola”.

- Muito desperdício de espaço.
- Código fica mais complicado.

Método 3: BST comum com arcos “cola” vermelhos.

- Amplamente utilizado na prática.
- Restrição arbitrária: só arcos da esquerda são vermelhos.



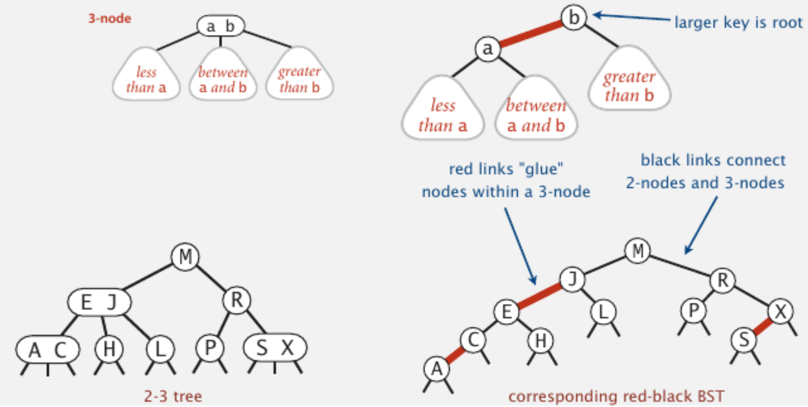
Árvores rubro-negras: história

A evolução de uma estrutura de dados:

- Bayer (1972): inventou a *symmetric binary B-tree*, AKA árvores 2-3-4.
- Guibas & Sedgewick (1978): derivaram a árvore rubro-negra (*red-black tree*) da árvore 2-3-4.
  - Sedgewick: rubro-negra por causa das melhores cores da impressora laser disponível do Xerox PARC.
  - Guibas: rubro-negra por causa das cores das canetas para desenhar.
- Andersson (1993): *right-leaning trees* para simplificar inserção e remoção.
- Cormen *et al* (2001): simplificou o algoritmo de remoção.
- Sedgewick (2007): *left-leaning red-black (LLRB) trees* simplificou ainda mais as operações da árvore.

LLRB BSTs

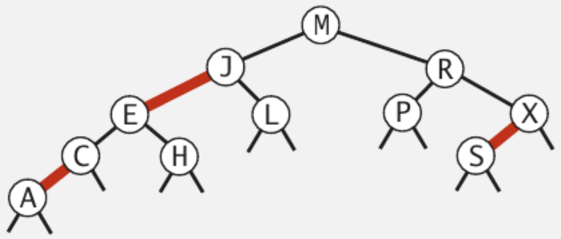
- Represente uma árvore 2-3 como uma BST.
- Use arcos vermelhos à esquerda para representar *3-nodes*.
- (Versão da árvore rubro-negra original tem uma associação com árvore 2-3-4.)



Uma definição equivalente

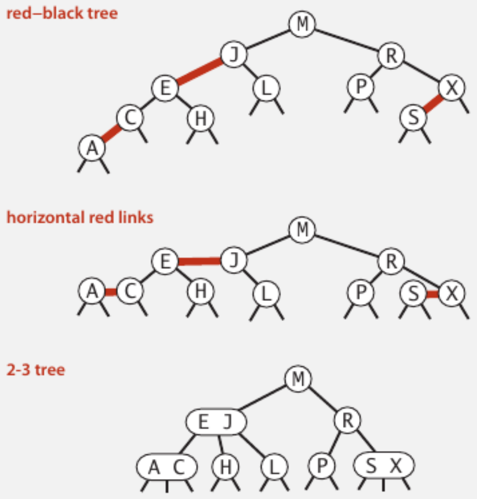
Uma árvore LLRB é uma BST aonde:

- Nenhum nó tem dois arcos vermelhos ligados a ele.
- Todo caminho da raiz até às folhas passa pela mesma quantidade de arcos de cor preta.
- Somente arcos da esquerda podem ser vermelhos.



LLRB BSTs: correspondência com árvores 2-3

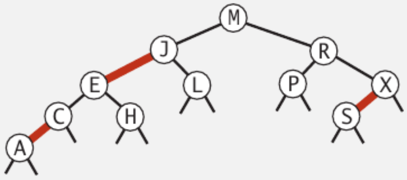
Propriedade fundamental: correspondência 1-1 entre árvores 2-3 e LLRB.



LLRB BSTs: busca

Observação: a busca é idêntica à BST padrão (ignora a cor).
Mas executa mais rápido porque a árvore é balanceada.

```
Value search(RBT *n, Key key) {
  while (n != NULL) {
    int cmp;
    cmp = compare(key, n->key);
    if      (cmp < 0) n = n->l;
    else if (cmp > 0) n = n->r;
    else          return n->val;
  }
  return NULL_Value;
```

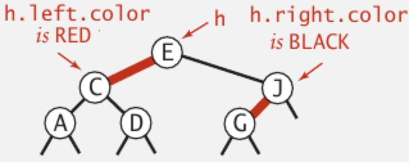


Obs.: Maioria das operações (*floor, traverse, etc*) também são idênticas.

LLRB BSTs: representação

Cada nó é apontado por exatamente um nó (o pai) $\Rightarrow$ podemos codificar a cor do arco nos nós filhos.

```
#define RED      true
#define BLACK    false

typedef struct node RBT;

struct node {
    Key key;
    Value val;
    bool color;
    RBT *l, *r;
};
```



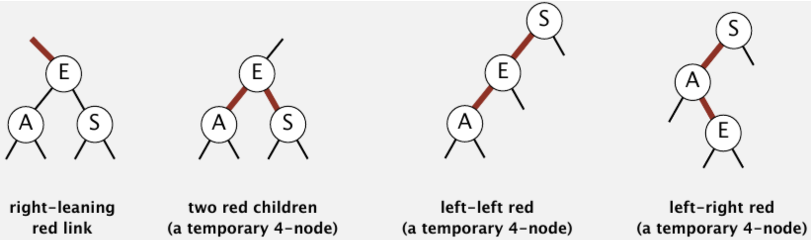

```
bool is_red(RBT *x) {
  if (x == NULL) return BLACK;
  return x->color; //RED == true
}
```

Inserção em LLRB BSTs: visão geral

Estratégia básica: Manter a correspondência de 1–1 com as árvores 2-3.

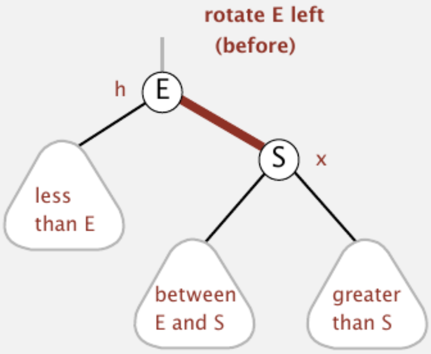
Durante operações internas, manter:

- Ordenação simétrica.
- Balanceamento.



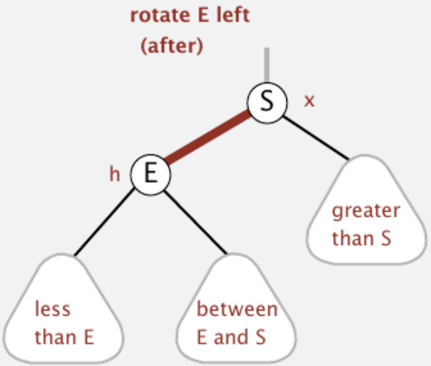
Como? Aplicar operações fundamentais: rotações e coloração.

Rotação à esquerda: Mudar a orientação de um arco vermelho que está (temporariamente) à direita para ficar à esquerda.



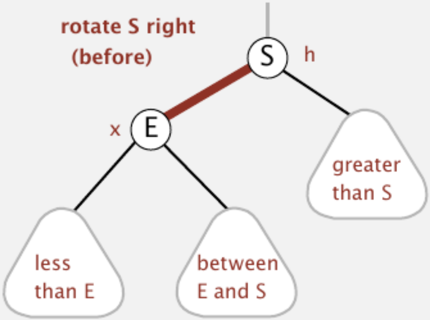

```
RBT* rotate_left(RBT *h) {
    RBT *x = h->r;
    h->r = x->l;
    x->l = h;
    x->color = x->l->color;
    x->l->color = RED;
    return x;
}
```

Rotação à esquerda: Mudar a orientação de um arco vermelho que está (temporariamente) à direita para ficar à esquerda.




```
RBT* rotate_left(RBT *h) {
    RBT *x = h->r;
    h->r = x->l;
    x->l = h;
    x->color = x->l->color;
    x->l->color = RED;
    return x;
}
```

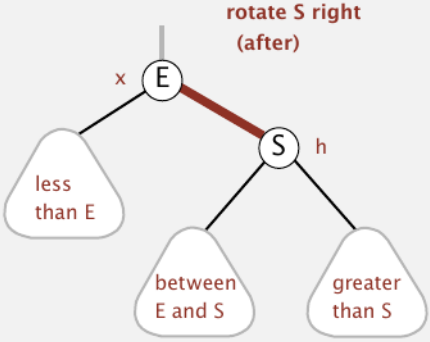
Rotação à direita: Mudar a orientação de um arco vermelho à esquerda para ficar (temporariamente) à direita.




```
RBT* rotate_right(RBT *h) {
    RBT *x = h->l;
    h->l = x->r;
    x->r = h;
    x->color = x->r->color;
    x->r->color = RED;
    return x;
}
```

LLRB BSTs: operações elementares

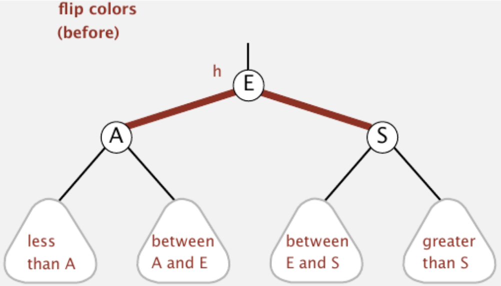
Rotação à direita: Mudar a orientação de um arco vermelho à esquerda para ficar (temporariamente) à direita.




```
RBT* rotate_right(RBT *h) {
    RBT *x = h->l;
    h->l = x->r;
    x->r = h;
    x->color = x->r->color;
    x->r->color = RED;
    return x;
}
```

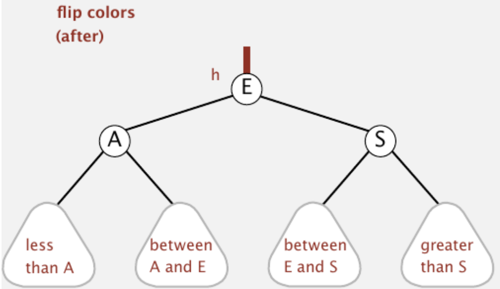
LLRB BSTs: operações elementares

Color flip: Mudar a coloração dos arcos para dividir um *4-node* temporário.




```
void flip_colors(RBT *h) {
    h->color = RED;
    h->l->color = BLACK;
    h->r->color = BLACK;
}
```

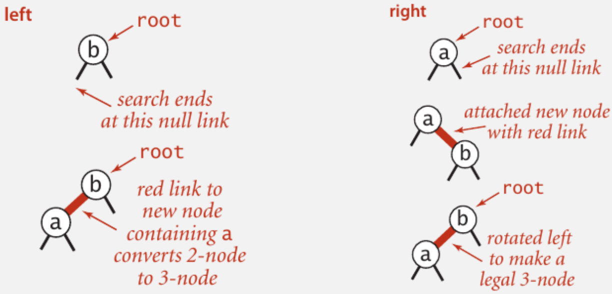
Color flip: Mudar a coloração dos arcos para dividir um *4-node* temporário.




```
void flip_colors(RBT *h) {
    h->color = RED;
    h->l->color = BLACK;
    h->r->color = BLACK;
}
```

Inserção em LLRB BSTs

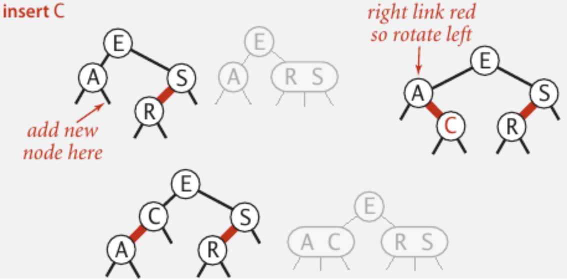
Aquecimento 1: Inserir em uma árvore com exatamente um nó.



Inserção em LLRB BSTs

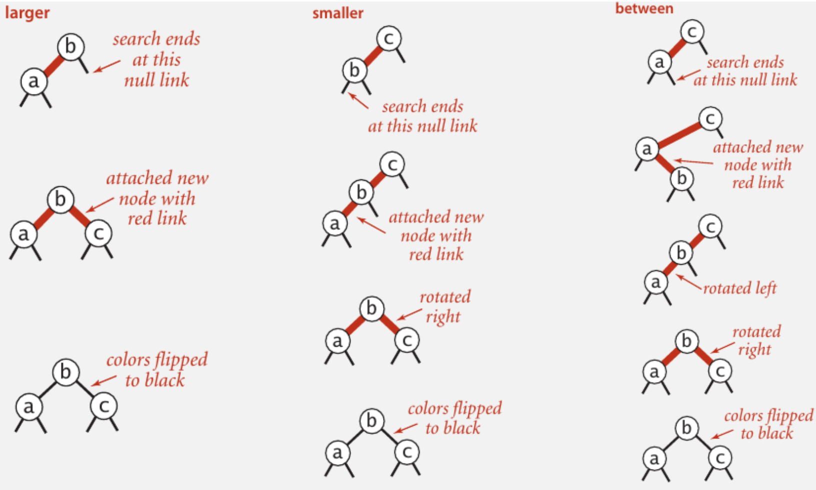
Caso 1: Inserir em um *2-node* nas folhas.

- Faça a inserção como na BST padrão, pinte o novo arco de vermelho.
- (Mantém a ordenação simétrica e balanceamento.)
- Se o novo arco vermelho está à direita, faça rotação à esquerda.
- (Re-estabelece os invariantes de coloração.)



Inserção em LLRB BSTs

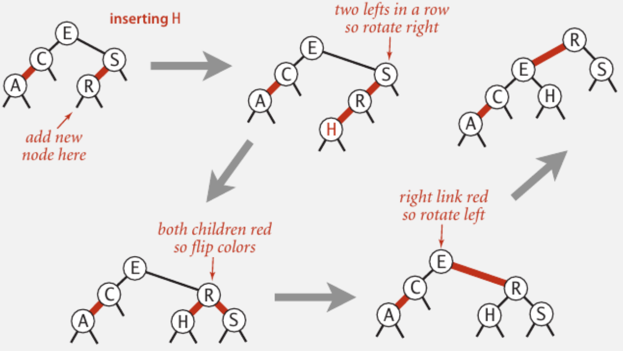
Aquecimento 2: Inserir em uma árvore com exatamente 2 nós.



Inserção em LLRB BSTs

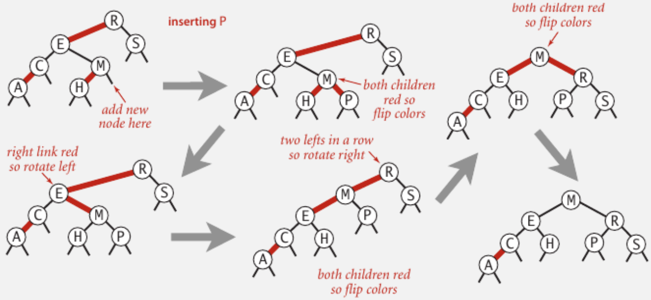
Caso 2: Inserir em um *3-node* nas folhas.

- Faça a inserção como na BST padrão, pinte o novo arco de vermelho.
- Se necessário, rotacione para equilibrar o *4-node*.
- Troque as cores para propagar o arco vermelho para cima.
- Se necessário, rotacione para tender à esquerda.

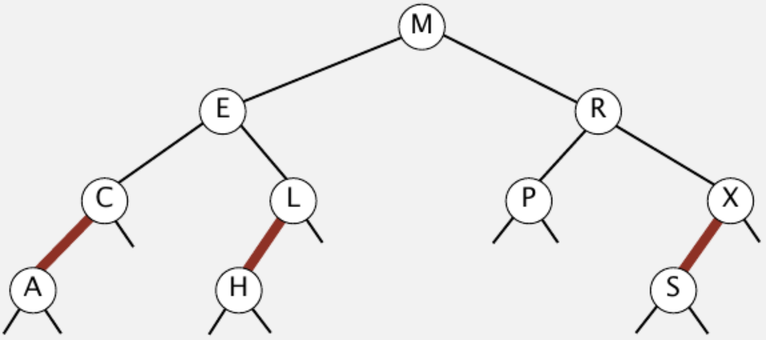


Caso 2: Inserir em um *3-node* nas folhas.

- Faça a inserção como na BST padrão, pinte o novo arco de vermelho.
- Se necessário, rotacione para equilibrar o *4-node*.
- Troque as cores para propagar o arco vermelho para cima.
- Se necessário, rotacione para tender à esquerda.
- Repita os casos 1 ou 2 para cima na árvore (se precisar).

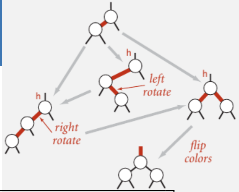


LLRB BSTs: construção



Veja o vídeo `33DemoRedBlackBST.mov`.

Inserção em LLRB BSTs: implementação

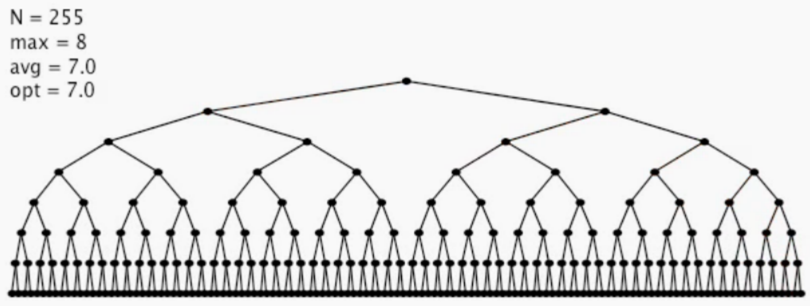


Mesmo código para todos os casos!

```
RBT* RBT_insert(RBT *h, Key key, Value val) {
  // Insert at bottom and color it red.
  if (h == NULL) { return create_node(key, val, RED); }

  int cmp = compare(key, h->key);
  if      (cmp < 0) { h->l = RBT_insert(h->l, key, val); }
  else if (cmp > 0) { h->r = RBT_insert(h->r, key, val); }
  else /*cmp == 0*/ { h->val = val; }

  // Lean left.
  if (is_red(h->r) && !is_red(h->l))     { h = rotate_left(h); }
  // Balance 4-node.
  if (is_red(h->l) &&  is_red(h->l->l)) { h = rotate_right(h); }
  // Split 4-node.
  if (is_red(h->l) &&  is_red(h->r))     { flip_colors(h); }

  return h;
}
```

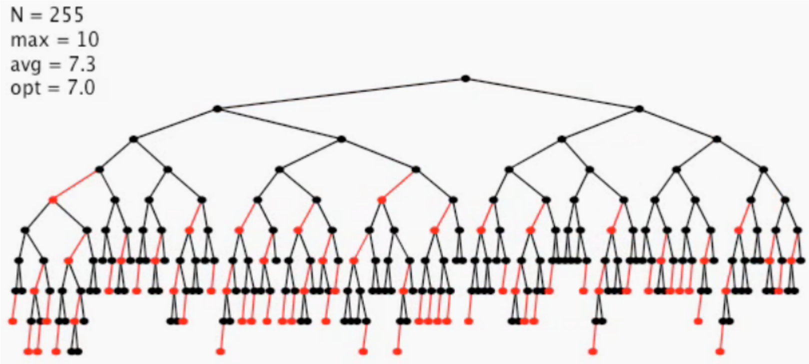
Inserção em LLRB BSTs: visualização

Árvore após 255 inserções ordenadas (cresc. ou decresc.)



Inserção em LLRB BSTs: visualização

Árvore após 255 inserções aleatórias.



Implementações de tabelas de símbolos: sumário

implementation	guarantee			average case			ordered ops?
	search	insert	delete	search hit	insert	delete	
sequential search (unordered list)	N	N	N	½ N	N	½ N	
binary search (ordered array)	$\lg N$	N	N	$\lg N$	½ N	½ N	✔
BST	N	N	N	$1.39 \lg N$	$1.39 \lg N$	$\sqrt{N}$	✔
2-3 tree	$c \lg N$	$c \lg N$	$c \lg N$	$c \lg N$	$c \lg N$	$c \lg N$	✔
red-black BST	$2 \lg N$	$2 \lg N$	$2 \lg N$	$1.0 \lg N$ *	$1.0 \lg N$ *	$1.0 \lg N$ *	✔

* exact value of coefficient unknown but extremely close to 1

Part III

Árvores B

Modelo de um sistema de arquivos

- Página (*page*): Bloco contíguo de dados (ex., um arquivo ou bloco de 4096-bytes).
- Sonda (*probe*): primeiro acesso a uma página (ex., carregar do disco para a memória).

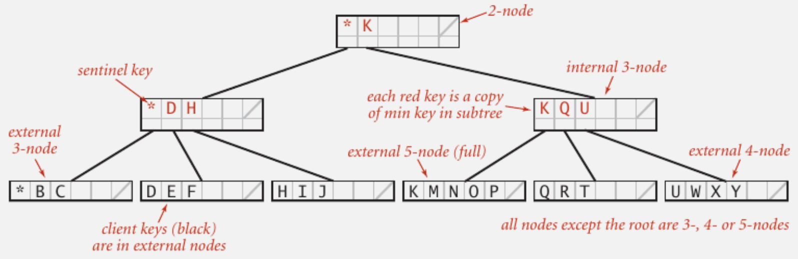
- Propriedade: Tempo necessário para uma sonda é muito maior que o tempo de acessar o dado em uma página.
- (Continua valendo mesmo com SSDs: HD $>>>$ RAM.)

- Modelo de custo: número de *probes*.
- Objetivo: Acessar os dados (aleatoriamente) com um número mínimo de *probes*.

Árvores B (*B-trees* – Bayer-McCreight, 1972)

Árvore B: Generaliza árvores 2-3 permitindo até $M - 1$ chaves por nó.

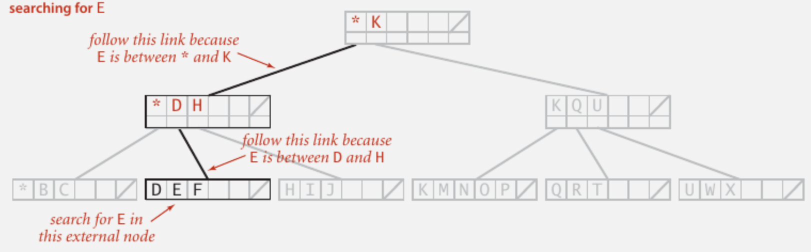
- Escolha M o maior possível para que tudo caiba em uma página. (Ex., $M = 1024$.)
- Pelo menos duas chaves na raiz.
- Pelo menos $M/2$ chaves nos demais nós.
- Nós folha contém as todas as chaves.
- Nós internos copiam as chaves para guiar a busca.



Anatomy of a B-tree set (M = 6)

Busca em uma árvore B

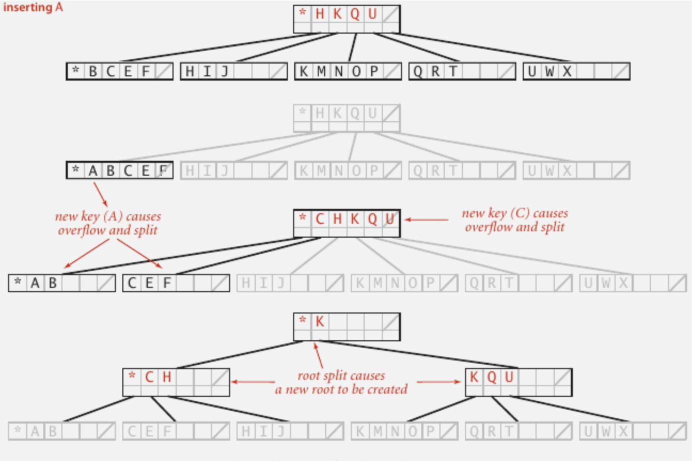
- Comece na raiz.
- Encontre o intervalo da chave buscada e tome o *link* correspondente.
- Busca termina em um nó folha.



Searching in a B-tree set (M = 6)

Inserção em uma árvore B

- Busque pela nova chave.
- Insira em uma folha.
- Divida os nós com *M* chaves subindo pela árvore.



Inserting a new key into a B-tree set

Balanceamento em uma árvore B

Proposição: Uma busca ou inserção em uma árvore B de ordem M com N chaves leva entre $\log_{M-1} N$ e $\log_{M/2} N$ *probes*.

Na prática: Número de *probes* é no máximo 4.

$M = 1024$, $N = 62$ bilhões $\Rightarrow \log_{M/2} N \leq 4$.

Otimização básica: Sempre manter a página raiz na memória.

Construindo uma árvore B grande

Árvores balanceadas na natureza selvagem

Árvores rubro-negras são amplamente utilizadas como tabelas de símbolos de sistemas.

- Java: `TreeMap`, `TreeSet`, `HashMap` (Java 8+).
- C++ STL: `map`, `multimap`, `multiset`.
- Linux: *Completely Fair Scheduler*, `linux/rbtree.h`

Variantes de árvores B: árvores B+, árvores B*, etc.

Árvores B (e variantes) são amplamente utilizadas em sistemas de arquivos e bancos de dados.

- Windows: NTFS.
- Mac: HFS, HFS+.
- Linux: ReiserFS, XFS, Ext3FS, JFS.
- Bancos de dados: ORACLE, DB2, INGRES, PostgreSQL.