MAC0323 Algoritmos e Estruturas de Dados II

Edição 2020 – 2

Fonte: ash.atozviews.com

Compacto dos melhores momentos

AULA 13

BSTs rubro-negras: `delete()`

Fonte: `.../only-one/red-leaves-black-tree/`

Referências: BSTs rubro-negras (PF); Balanced Search Trees (S&W); slides (S&W)

Remoção em árvore 2-3

No caminho até a chave a ser removida,
o algoritmo mantém a relação invariante
com respeito à árvore 2-3:

> *o nó sendo examinado é um 3-nó*
> *ou um 4-nó (temporário).*

`deleteMin`: Três casos

Caso 1: Nó esquerdo ou seu filho esquerdo é vermelho (ou seja, faz parte de um 3-nó ou 4-nó)

Prossigo com a remoção no nó da esquerda.

`deleteMin`: Três casos

Caso 1: Nó esquerdo ou seu filho esquerdo é vermelho (ou seja, faz parte de um 3-nó ou 4-nó)

Prossigo com a remoção no nó da esquerda.

Caso 2: Nó esquerdo e seu filho esquerdo são **negros** (ou seja, filho esquerdo é um 2-nó)

Caso 2.1: Se filho direito é um 3-nó, ajusto para transformar filho esquerdo em 3-nó.

`deleteMin`: Três casos

Caso 1: Nó esquerdo ou seu filho esquerdo é vermelho (ou seja, faz parte de um 3-nó ou 4-nó)

Prossigo com a remoção no nó da esquerda.

Caso 2: Nó esquerdo e seu filho esquerdo são **negros** (ou seja, filho esquerdo é um 2-nó)

Caso 2.1: Se filho direito é um 3-nó, ajusto para transformar filho esquerdo em 3-nó.

Caso 2.2: Se filho direito também é um 2-nó, formo 4-nó com filhos e o nó corrente.

Depois dos ajustes, prossigo com a remoção no nó da esquerda.

Caso 2

Nó esquerdo e seu filho esquerdo são **negros**

Caso 2.1: Se filho direito é um 3-nó,
ajusto para transformar filho esquerdo em 3-nó.

Caso 2.2: Se filho direito também é um 2-nó,
formo 4-nó com filhos e o nó corrente.

```
static Node moveRedLeft(Node h) {
    flipColors(h);
    if (isRed(h->right->left)) {
        h->right = rotateRight(h->right);
        h = rotateLeft(h);
        flipColors(h);
    }
    return h;
}
```

deleteMin()

Se ambos os filhos da raiz são **negros**
faz a raiz rubra.

```
void deleteMin() {
   if (r == NULL) return;
   if (!isRed(r->left) && !isRed(r->right))
      ▷ filho esquerdo da raiz é 2-nó
      r->color = RED;
   r = deleteMinTree(r);
   if (!isEmpty()) r->color = BLACK;
}
```

deleteMin()

```
static Node deleteMinTree(Node h) {
   if (h->left == NULL)
      return NULL;
   if (!isRed(h->left) && !isRed(h->left->left))
      ▷ filho esquerdo é 2-nó
      h = moveRedLeft(h);
   h->left = deleteMinTree(h->left);
   return balance(h);
}
```

deleteMin()

Volta o invariante rubro-**negro**.

```
static Node balance(Node h) {
   if (isRed(h->right))
      h = rotateLeft(h);
   if (isRed(h->left) && isRed(h->left->left))
      h = rotateRight(h);
   if (isRed(h->left) && isRed(h->right))
      flipColors(h);
   h->n = size(h->left) + size(h->right) + 1;
   return h;
}
```

AULA 14

delete()

Remoção de uma chave arbitrária.

```
void delete(Key key) {
   if (!isRed(r->left))
      r->color = RED;
   r = delete(r, key);
   if (!isEmpty())
      r->color = BLACK;
}
```

delete()

```
static Node delete(Node h, Key key) {
    if (compare(key, h->key) < 0) {
        if(!isRed(h->left) && !isRed(h->left->left))
            h = moveRedLeft(h);
        h->left = delete(h->left, key);
    }
    else {    ▷ chave está à direita
        if (isRed(h->left))
            h = rotateRight(h);
        if (compare(key, h->key) == 0
            && h->right == NULL)
            return NULL;
```

delete()

```
        if(!isRed(h->right) && !isRed(h->right->left))
            h = moveRedRight(h);
        if (compare(key, h->key) == 0) {
            Node x = min(h->right);
            h->key = x->key;
            h->val = x->val;
            h->right = deleteMin(h->right);
        }
        else h->right = delete(h->right, key);
    }
    return balance(h);
}
```

moveRedRight

Um pouco mais simples que o moveRedLeft.

```
static Node moveRedRight(Node h) {
    flipColors(h);
    if (isRed(h->left->left)) {
        h = rotateRight(h);
        flipColors(h);
    }
    return h;
}
```

Observação

BSTs são estruturas ordenadas.

Como as chaves de uma BST são comparáveis,
podemos perguntar pela chave mínima
e pela chave máxima.

Já fizemos isso.

Observação

`BST`s são estruturas ordenadas.

Como as chaves de uma `BST` são comparáveis,
podemos perguntar pela chave mínima
e pela chave máxima.

Já fizemos isso.

O **piso** (`floor()`) de uma chave `key` na `BST` é a maior chave da `BST` que é menor que ou igual a `key`.

Observação

BSTs são estruturas ordenadas.

Como as chaves de uma BST são comparáveis,
podemos perguntar pela chave mínima
e pela chave máxima.

Já fizemos isso.

O **piso** (`floor()`) de uma chave `key` na BST é a maior chave da BST que é menor que ou igual a `key`.

O **teto** (`ceiling()`) de uma chave `key` na BST é a menor chave da BST que é maior que ou igual a `key`.

Os métodos `min()`, `max()`, `floor()` e `ceiling()`
são exatamente os mesmos das BSTs ordinárias!

floor()

Devolve NULL se key não tem piso nesta BST.

```
Key floor(Key key) {
   Node x = floorTree(r, key);
   if (x == NULL) return NULL;
   return x->key;
}
```

floor()

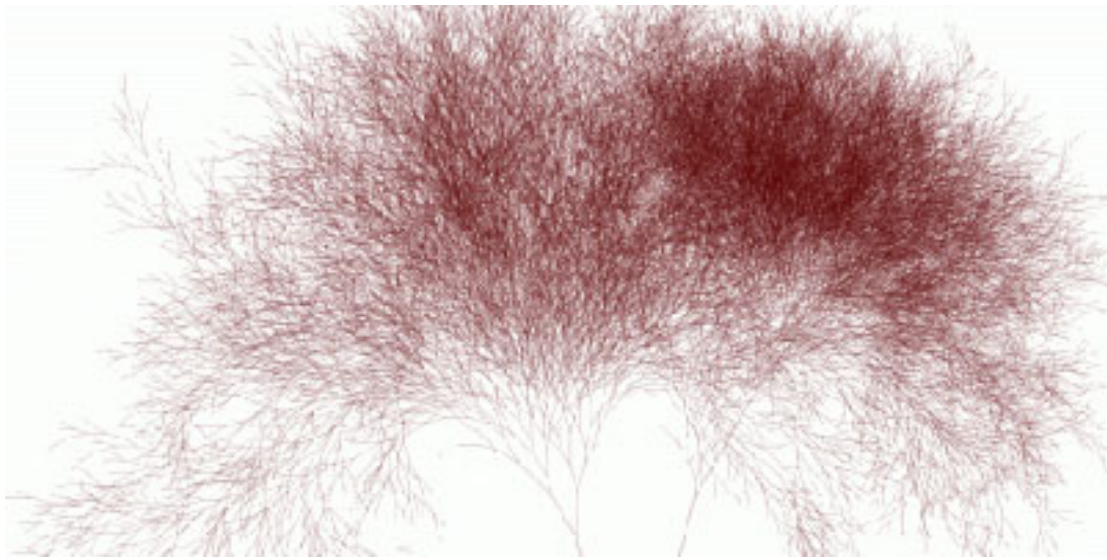
Devolve o nó que contém o piso
de key na subárvore com raiz x.
Devolve NULL se esse piso não existe.

```
static Node floorTree(Node x, Key key) {
   if (x == NULL) return NULL;
   int cmp = compare(key, x->key);
   if (cmp == 0) return x;
   if (cmp < 0)
      return floorTree(x->left, key);
   Node t = floorTree(x->right, key);
   if (t != NULL) return t;
   else return x;
}
```

Fonte: ash.atozviews.com

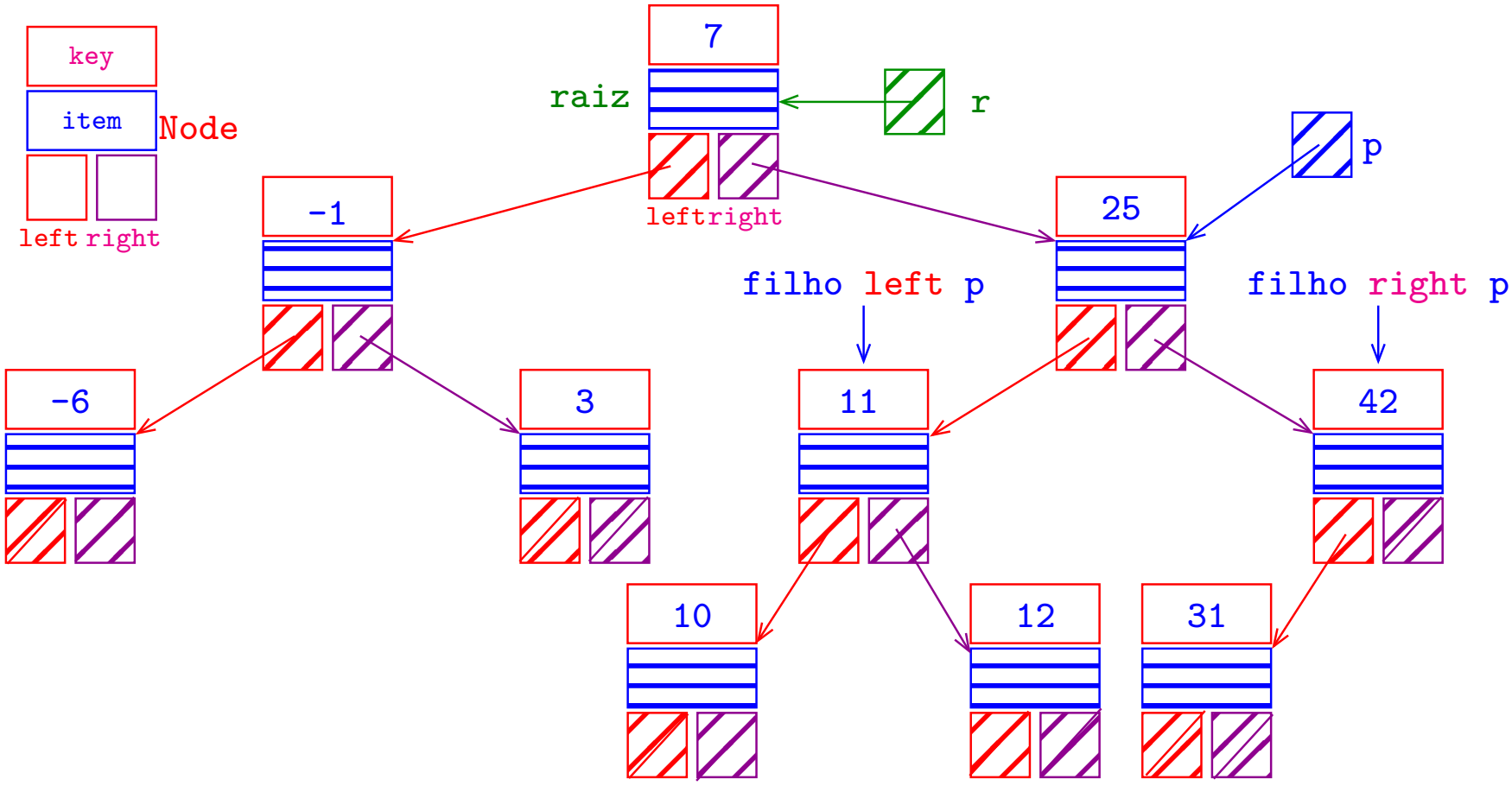
Compacto dos melhores momentos das últimas aulas

Árvores binárias de busca



Fonte: http://infosthetics.com/archives/

Árvore binárias de busca



in-ordem (e-r-d): -6 -1 3 7 10 11 12 25 31 42

Consumo de tempo

O consumo de tempo das operações `get()`, `put()` e `delete()` é, no pior caso, proporcional à altura da árvore.

Consumo de tempo no pior caso

No pior caso a altura de uma BST
é proporcional ao número `n` de nós BST.

Conclusão:

O consumo de tempo das operações
`get()`, `put()` e `delete()` em uma BST é,
no pior caso, proporcional ao número `n` de nós.

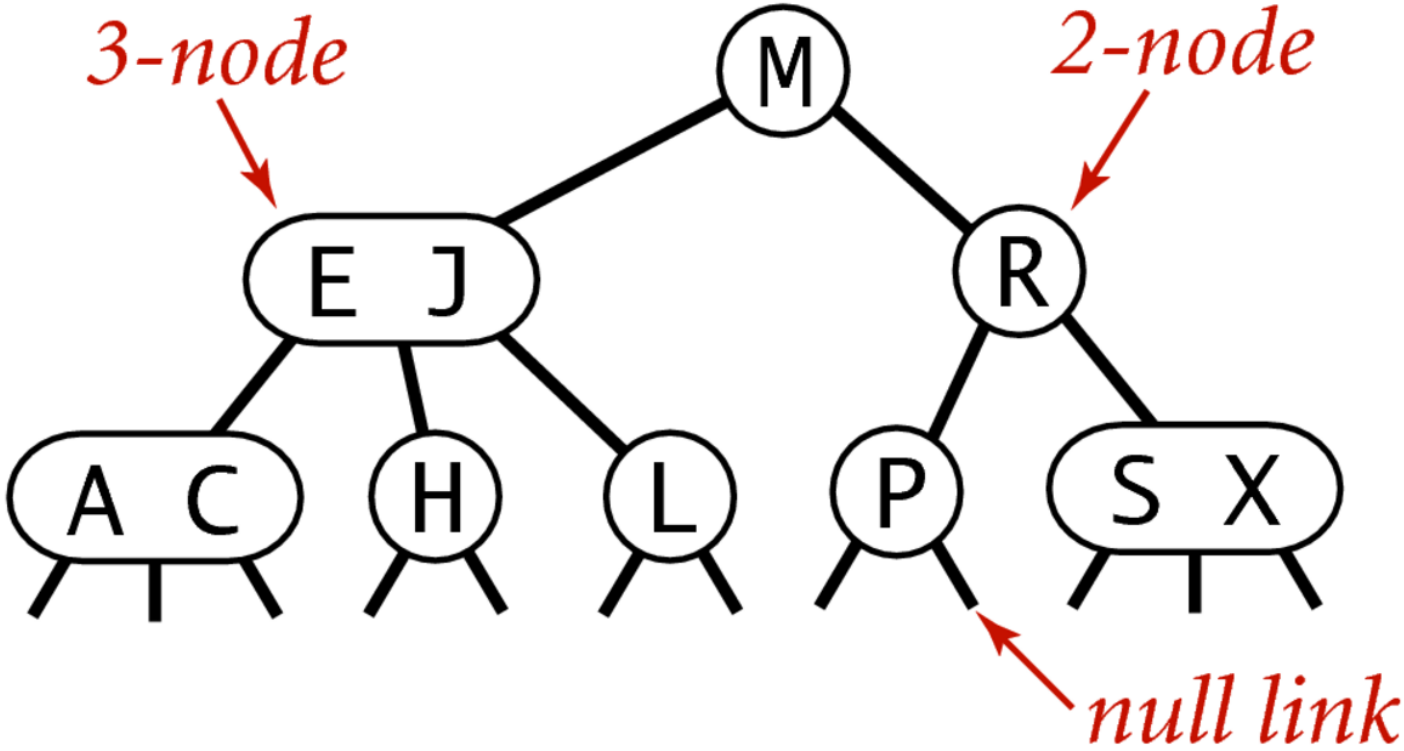
Consumo de tempo esperado

A altura esperada de BST aleatória
é aproximadamente $2 \lg n$.

Conclusão:

O consumo de tempo esperado das
operações `get()`, `put()` e `delete()` em
uma BST aleatória é proporcional $\lg n$,
onde n é o número de nós.

Árvore 2-3 de busca



Anatomy of a 2-3 search tree

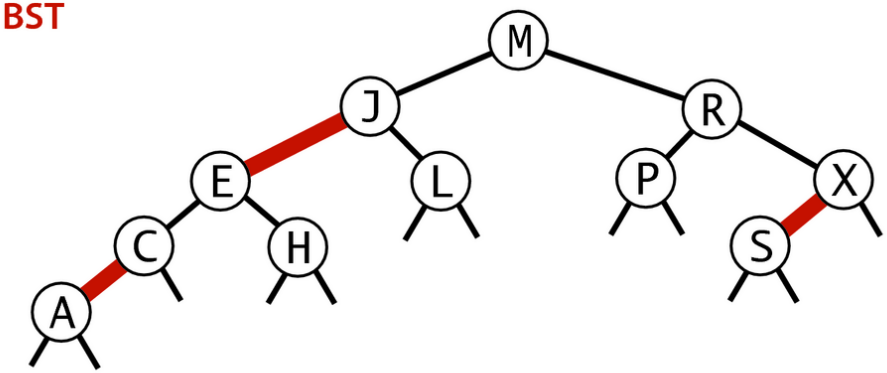
Fonte: algs4

Consumo de tempo

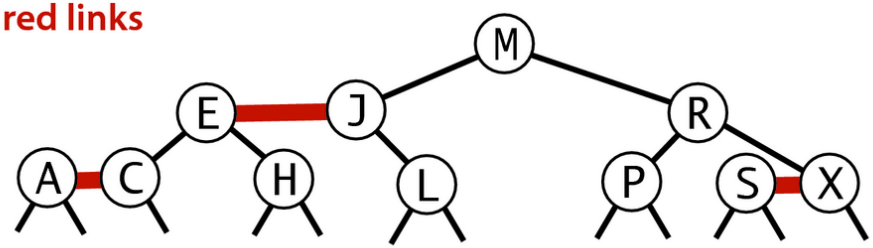
Numa árvore 2-3 com n nós, busca e inserção nunca visitam mais que $\lg(n+1)$. Cada visita faz no máximo 2 comparações de chaves.

BST rubro-negra

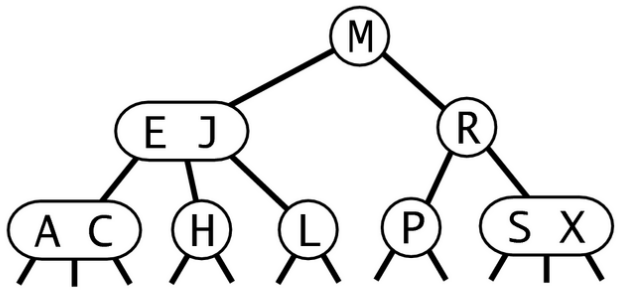
red-black BST



horizontal red links



2-3 tree

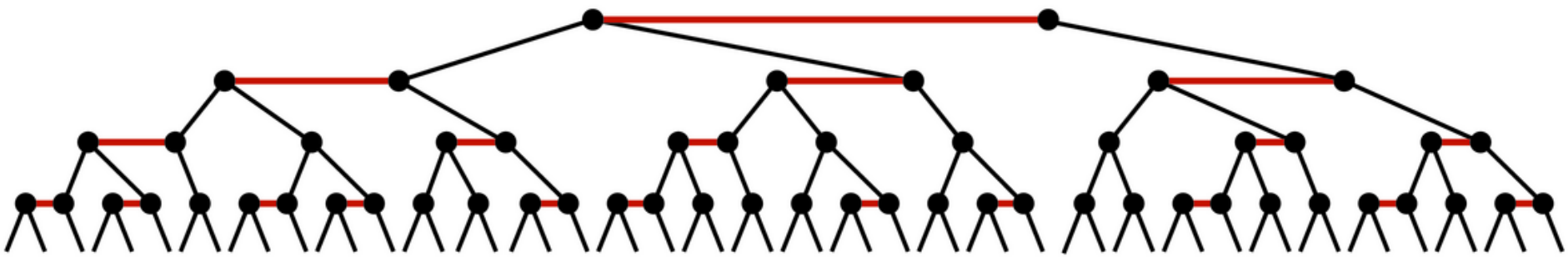


Fonte: algs4

1-1 correspondence between red-black BSTs and 2-3 trees

Árvore 2-3 para rubro-negra

Se os links rubros forem desenhados horizontalmente e depois contraídos, teremos uma árvore 2-3:



A red-black tree with horizontal red links is a 2-3 tree

Fonte: algs4

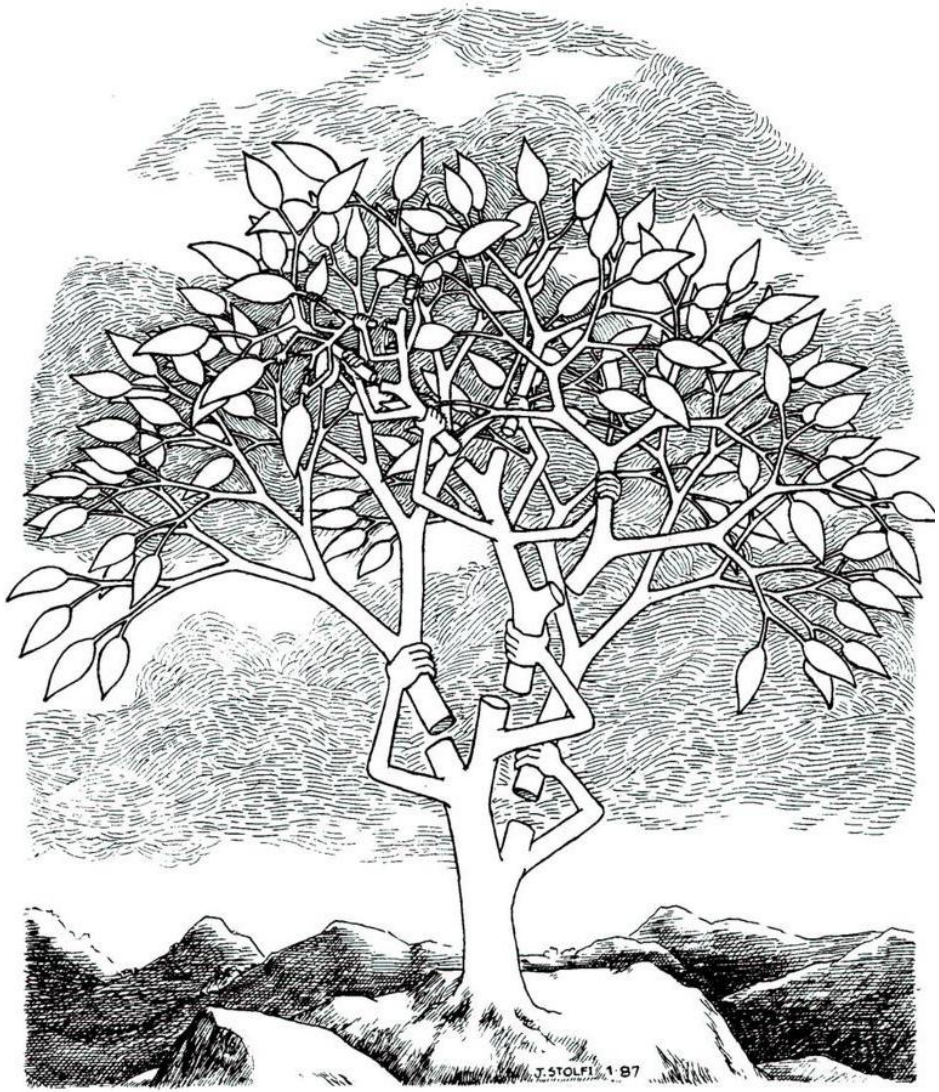
Consumo de tempo

A altura esperada de uma BST rubro-negra é aproximadamente $\leq 2 \lg n$.

Conclusão:

O consumo de tempo das operações `get()`, `put()` e `delete()` em uma BST rubro-negra é $O(\lg n)$.

Self-adjusting BSTs



Fonte: Jorge Stolfi

Self-adjusting BSTs

Uma `BST` com auto balanceamento/ajuste (*self-balancing*/*self-adjusting*) é uma ABB que **automaticamente** mantém a sua altura pequena diante de uma sequência de operações `put()`, `get()`, ...

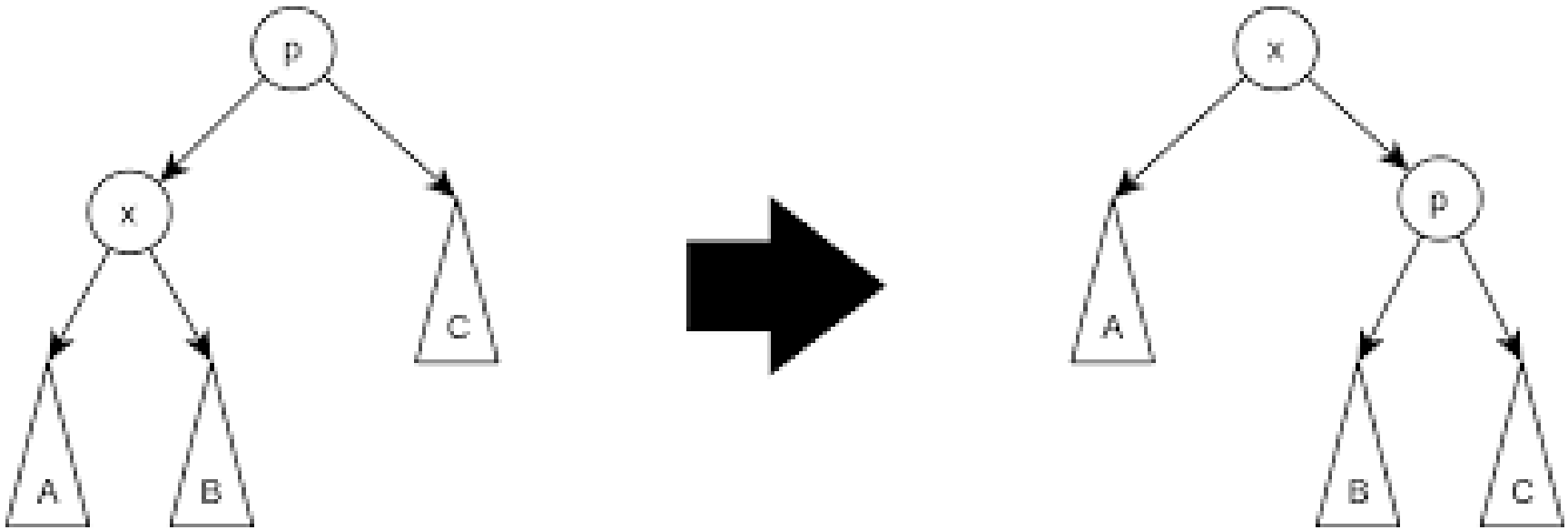
Árvores rubro-negras são `BST`s com auto balanceamento.

Splay trees

Uma **splay tree** é uma BST com
auto-balanceamento e com a propriedade adicional
que os elementos acessados recentemente são
rapidamente acessados.

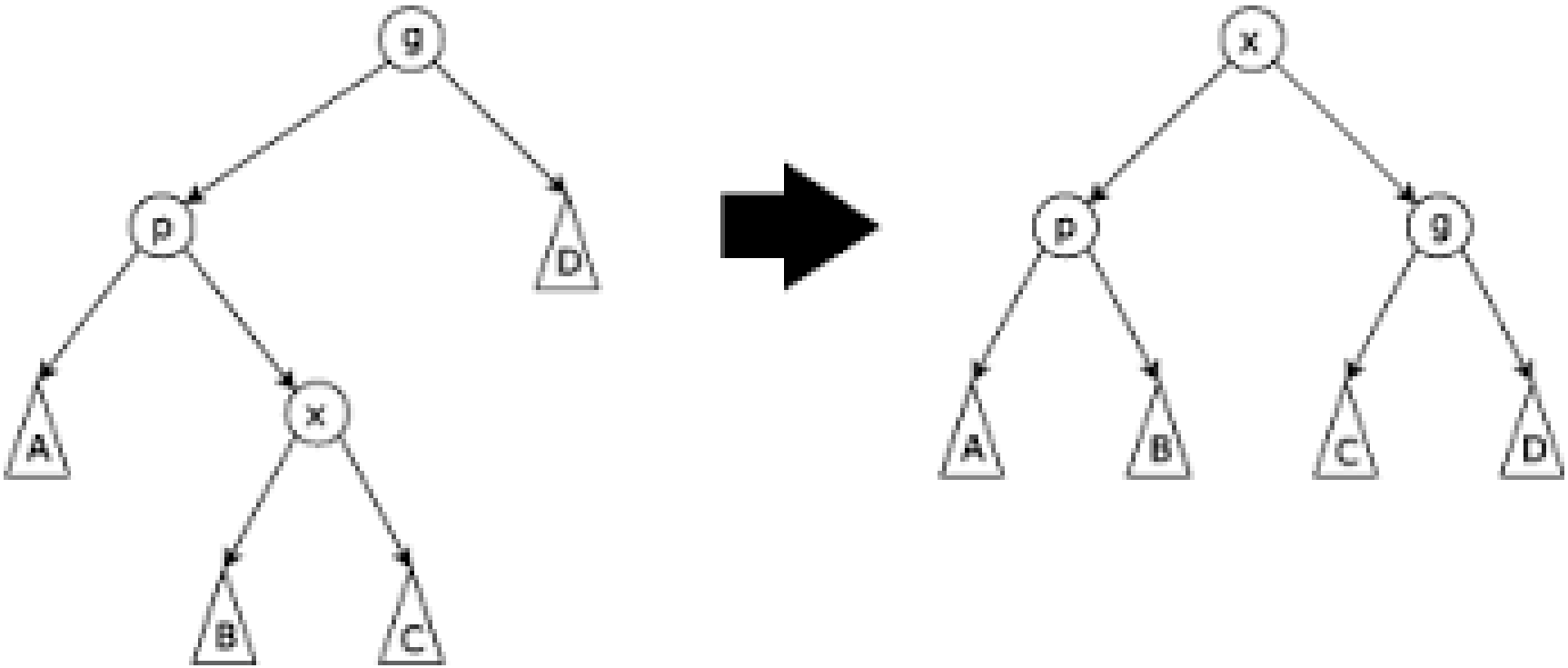
Splay trees implementam em BSTs
a política *move to front*.

Splaying: zig



Fonte: Wikipedia

Splaying: zig-zag



Fonte: Wikipedia

Resumo

estrutura	consumo de tempo `get()`, `put()`, ...	observação
Skip list	$O(\lg n)$	esperado
BST	$O(n)$	pior caso
BST-aleatória	$O(\lg n)$	esperado
2-3 ST	$O(\lg n)$	pior caso
RedBlack BST	$O(\lg n)$	pior caso
Splay BST	$O(\lg n)$	amortizado
Treap BST	$O(\lg n)$	esperado

n = número de nós na estrutura

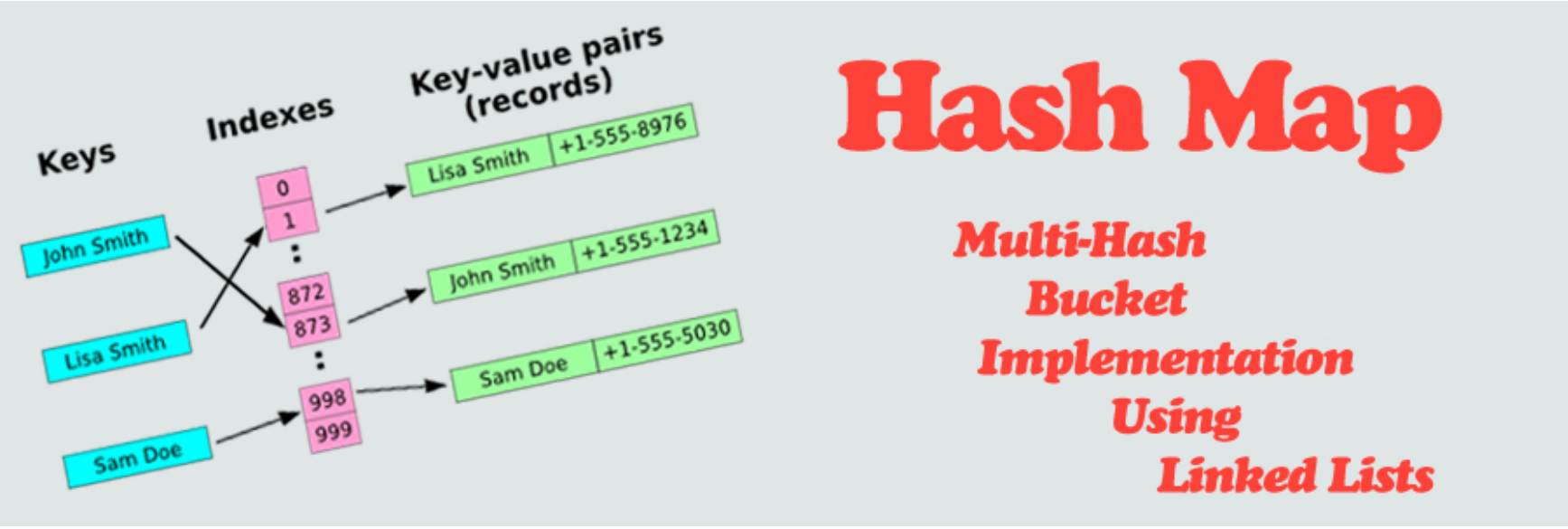
Mais experimentos ainda

Consumo de tempo para se criar um `ST` em que as chaves são as palavras em `les_miserables.txt` e os valores são o número de ocorrências.

estrutura	`ST`	tempo
vetor MTF	não-ordenada	7.6
vetor	ordenada	1.5
lista ligada MTF	não-ordenada	15.3
skiplist	ordenada	1.1
árvore binária de busca	ordenada	0.72
árvore rubro-**negra**	ordenada	0.76
splay tree	ordenada	0.68

Tempos em segundos obtidos com `StopWatch`.

Hashing



Fonte: http://programmingnotes.freeweq.com

Referências: Hashing (PF); Hash Tables (S&W); slides (S&W); Hashing Functions (S&W); CLRS, cap 12; TAOP, vol 3, cap. 6.4;

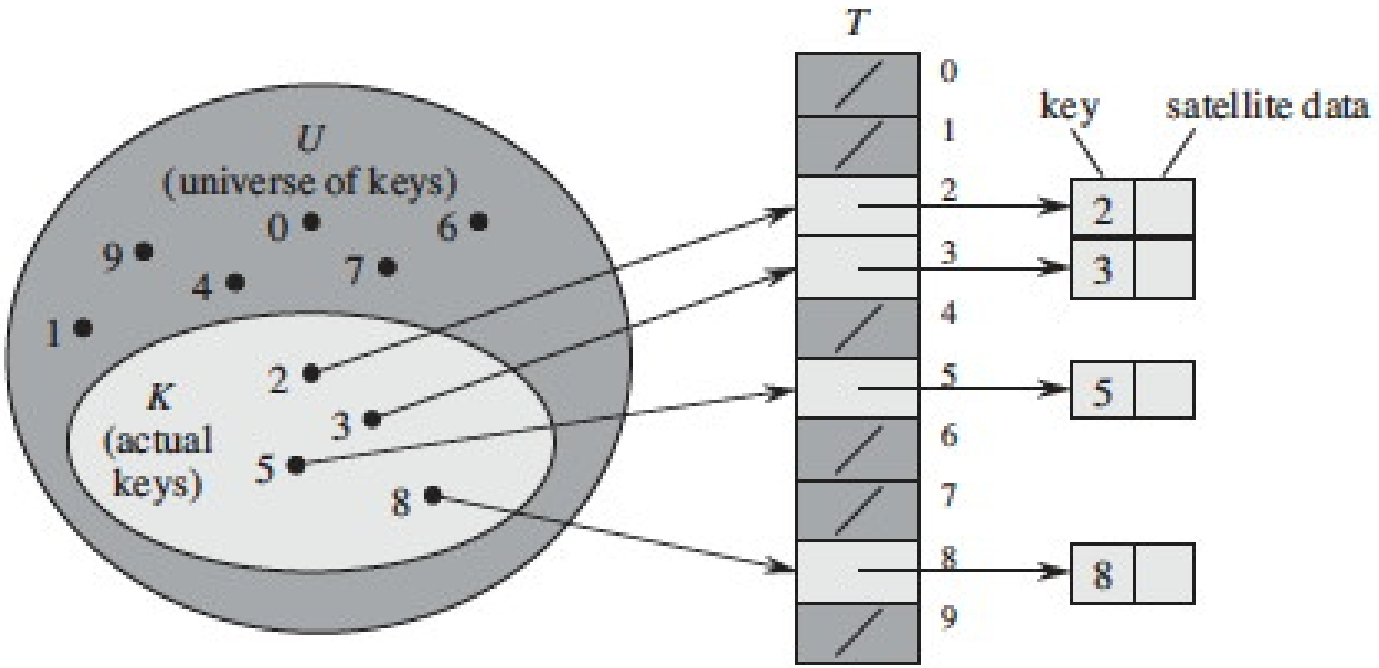
Endereçamento direto

Endereçamento direto (*directed-address*) é uma técnica que funciona bem quando o universo de chaves é razoavelmente pequeno.

Tabela indexada pelas chaves, uma posição para cada possível índice.

Cada posição armazena o valor correspondente a uma dada chave.

Endereçamento direto



Fonte: CLRS

Endereçamento direto

Implementação:

```
static Value *vals;

void DirectAddressInit(int m) {
   vals = mallocSafe(m * sizeof(Value));
   for (int i = 0; i < m; i++)
      vals[i] = NULL;
}
```

Endereçamento direto

```
Value get(Key key) {
   return vals[key];
}

void put(Key key, Value val) {
   vals[key] = val;
}

void delete(Key key) {
   vals[key] = NULL;
}
```

Consumo de tempo

Em uma tabela se símbolos com endereçamento direto o consumo de tempo de `get`(), `put`() e `delete`() é $O(1)$.

Maiores defeitos

Os maiores defeitos dessa implementação são:

- Em geral, as chaves não são inteiros não-negativos pequenos. . .
- desperdício de espaço: é possível que a maior parte da tabela fique vazia.

Tabelas de dispersão (*hash tables*)

Uma **tabela de dispensão** (= *hash table*) é uma maneira de organizar uma tabela de símbolos.

Inventadas para funcionar bem (em $\mathrm{O}(1)$) em média.

universo de chaves = conjunto de **todas** as possíveis chaves

chaves *realmente usadas* são, em geral, uma *parte pequena* do universo.

A tabela terá a forma `st`[0 . . `m`-1], onde `m` é o tamanho da tabela.

Funções de dispersão

Uma **função de dispersão** (= *hash function*) é uma maneira de mapear o universo de chaves no conjunto de índices da tabela.

A função de dispersão recebe uma chave `key` e retorna um número inteiro `h(key)` no intervalo 0 . . `m`-1.

O número `h(key)` é o **código de dispensão** (= *hash code*) da chave.

Queremos uma função de hashing que . . .

Queremos uma função de hashing que:

- possa ser calculada eficientemente (em $\mathrm{O}(1)$) e
- espalhe *bem* as chaves pelo intervalo $0 \,..\, \mathtt{m}-1$.

`Knuth, TAOC, pg. 514:`

> *"The verb 'to hash' means to chop something up to make a mess out of it; the idea in hashing is to scramble some aspects of the key and to use this partial information as basis for searching. . . "*

Funções injetoras...

Funções que associam chaves diferentes a inteiros diferentes são difíceis de se encontrar.

Mesmo se conhecêssemos as chaves de antemão!

Exemplo:

Existem $41^{31} \equiv 10^{50}$ funções de 31 elementos em 41 elementos e somente $41!/10! \equiv 10^{43}$ são injetoras: uma em cada 10 milhões!

Funções injetoras...

Funções que associam chaves diferentes a inteiros diferentes são difíceis de se encontrar.

Mesmo se conhecêssemos as chaves de antemão!

Mesmo se o tamanho da tabela for razoavelmente maior que o número de chaves.

O paradoxo do aniversário nos diz que se selecionarmos uniformemente ao acaso uma função que leva 23 chaves em uma tabela de tamanho 365, a probabilidade de que duas chaves sejam associadas a uma mesma posição é maior 0,5.

Conclusão: temos que conviver com **colisões**.

Função de hashing modular

Método da divisão (*division method*) ou hash modular: supondo que as chaves são inteiros positivos, podemos usar a função modular (resto da divisão por `m`):

```
static int hash(int key) {
   return key % m;
}
```

Função de hashing modular

Exemplos com `m` = 100 e com `m` = 97:

key	hash ($M = 100$)	hash ($M = 97$)
212	12	18
618	18	36
302	2	11
940	40	67
702	2	23
704	4	25
612	12	30
606	6	24
772	72	93
510	10	25
423	23	35
650	50	68
317	17	26
907	7	34
507	7	22
304	4	13
714	14	35
857	57	81
801	1	25
900	0	27
413	13	25

Fonte: `algs4`

Função de hashing modular

No caso de Strings, podemos iterar
hashing modular sobre os caracteres da string:

```
static int hash(String key) {
   int h = 0;
   for (int i = 0; i < strlen(key); i++)
      h = (31 * h + key[i]) % m;
   return h;
}
```

Função de hashing modular

Vantagens: rápida, faz apenas uma divisão.

Desvantagem:
devemos evitar certos valores para $\mathtt{m}$, por exemplo:

- se $\mathtt{m} = 2^{\mathtt{p}}$, então $\mathtt{h}(\mathtt{key})$ são os $\mathtt{p}$ bits menos significativos de $\mathtt{key}$.
- se a string de caracteres é interpretada como números na base $2^{\mathtt{p}}$, então $\mathtt{m} = 2^{\mathtt{p}} - 1$ é uma má escolha: permutações de caracteres são levadas ao mesmo valor de hash.

Um primo não "muito perto" de uma potência de 2 parece ser uma boa escolha para $\mathtt{m}$.

Função Multiplicativa

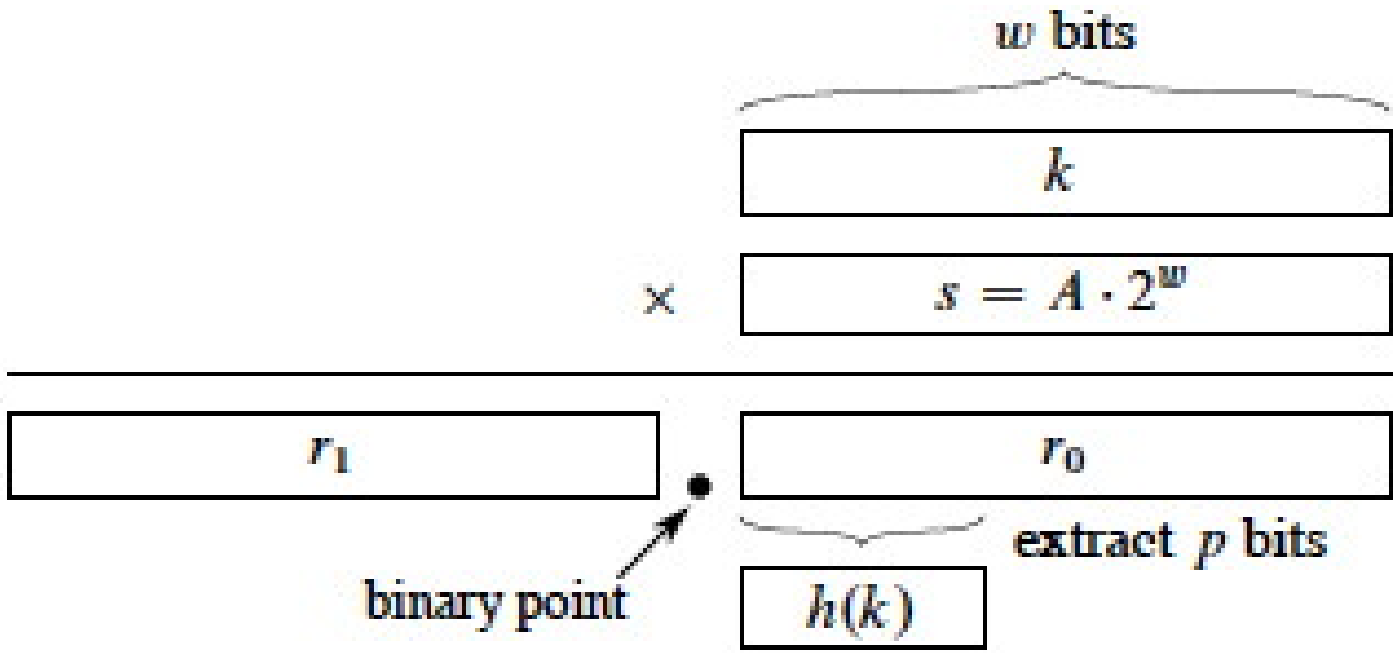
Método multiplicativo (*multiplicative method*):

- escolha uma constante `A` com $0 < \texttt{A} < 1$;
- multiplique `key` por `A`;
- extraia a parte fracional de `key` $\times$ `A`;
- multiplique a parte fracionária por `m`;
- o valor de hash é o chão dessa multiplicação.

Função Multiplicativa

Nesse caso, `m` é uma potência de 2.

Assim, `h(key)` contém os bits iniciais da metade menos significa de `key` $\times$ `A`.



Função Multiplicativa

Desvantagem: mais lenta que o hash modular

Vantagem: o valor de `m` não é crucial

O que Ubuntu tem a dizer...

http://releases.ubuntu.com/17.10/

MD5SUMS	2018-01-12 05:38	198
MD5SUMS-metalink	2018-01-12 05:38	213
MD5SUMS-metalink.gpg	2018-01-12 05:38	916
MD5SUMS.gpg	2018-01-12 05:38	916
SHA1SUMS	2018-01-12 05:38	222
SHA1SUMS.gpg	2018-01-12 05:38	916
SHA256SUMS	2018-01-12 05:38	294
SHA256SUMS.gpg	2018-01-12 05:38	916

https://en.wikipedia.org/wiki/MD5

https://en.wikipedia.org/wiki/SHA-2

O que Java tem a dizer

Em Java, toda classe tem
um método padrão `hashCode()`
que produz um inteiro entre -2^{31} e $2^{31} - 1$.

Exemplo:

```
String s = StdIn.readString();
int h = s.hashCode();
```

Boas e más funções de dispersão

Uma função só é eficiente se
espalha as chaves pelo intervalo de índices
de maneira *razoavelmente uniforme*.

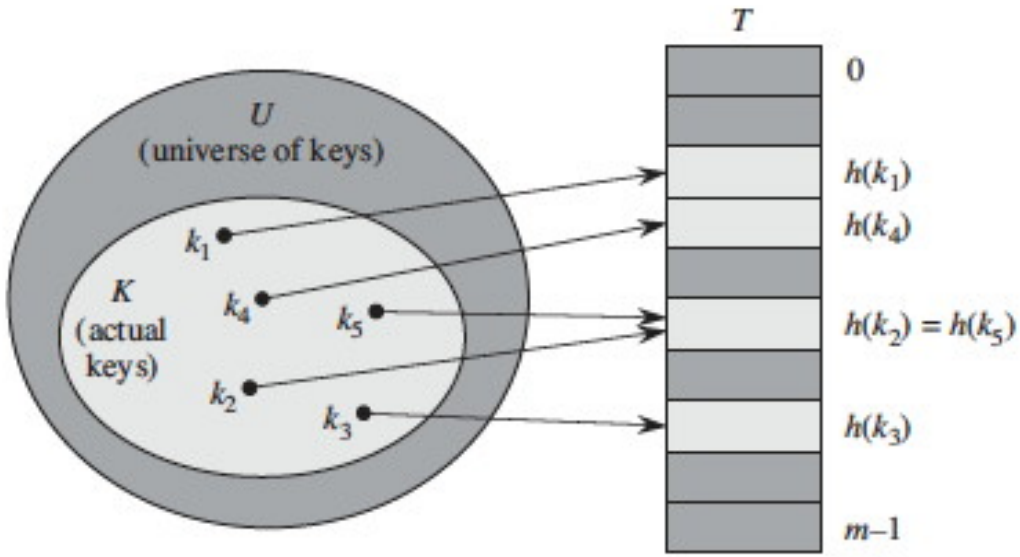
Por exemplos, se os dois últimos dígitos das chaves
não variam muito, então "`key % 100`" é uma
péssima função de dispersão.

Em geral é recomendável que
`m` seja um número primo.

Escolha de funções de dispersão é
uma combinação de estatística, probabilidade,
teoria dos números (primalidade), ...

Colisões

Como o número de chaves é em geral maior que `m`, é inevitável que a função de dispersão leve várias chaves diferentes no mesmo índice.



Fonte: CLRS

Colisões

Dizemos que há uma **colisão** quando
duas chaves diferentes são levadas no mesmo índice.

Algumas maneiras de tratar colisões:

- lista encadeadas (=*separating chaining*);
- sondagem linear (=*linear probing*);
  Também conhecido como *open addressing*.
- *double hashing (open addressing)*;