MAC0323 Algoritmos e Estruturas de Dados II

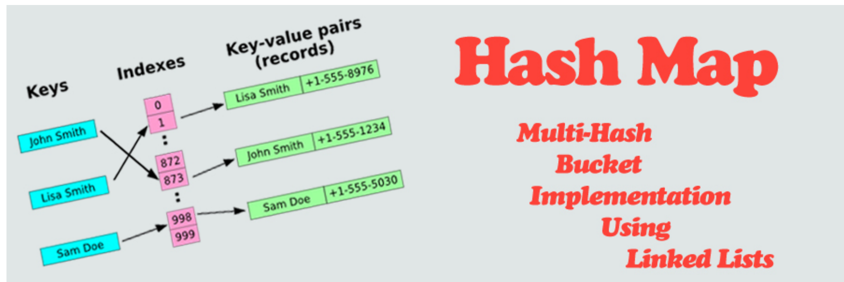
Edição 2020 – 2

Fonte: ash.atozviews.com

Compacto dos melhores momentos

AULA 14

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;

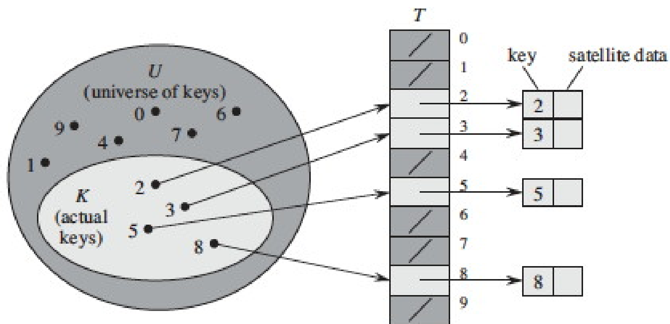
Endereçamento direto

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

Consumo de tempo

Em uma ST 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

Hash tables

Inventadas para funcionar em $O(1)$ ... em média.

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

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

Hash functions

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

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

Queremos uma função de hashing que:

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

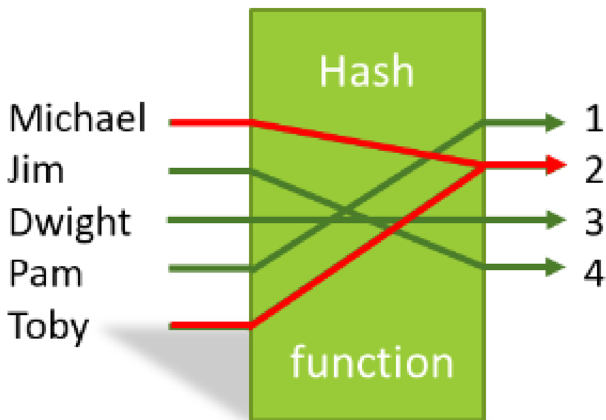
Perfeição é difícil. . .

Perfect hashing: funções que associam chaves diferentes a inteiros diferentes são difíceis de se encontrar mesmo conhecendo as chaves de antemão!

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 que 0,5.

AULA 15

Conviver com colisões. . .



Fonte: https://stackoverflow.com/

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 $\mathtt{m} = 100$ e com $\mathtt{m} = 97$:

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
701	1	22
418	18	30

Fonte: `algs4`

Função de hashing modular

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

```
static int hash(char *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 m, por exemplo:

- se $m = 2^p$, então h(key) são os p bits menos significativos de key.
- se a string de caracteres é interpretada como números na base 2^p, então $m = 2^p - 1$ 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 m.

Função Multiplicativa

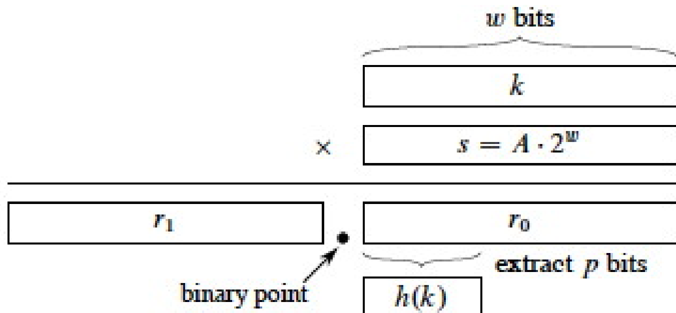
Método multiplicativo (*multiplicative method*):

- escolha uma constante A com $0 < A < 1$;
- 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.

$$h(key) = \lfloor (A * key \bmod 1) * m \rfloor$$

Função Multiplicativa

Nesse caso, se m é uma potência de 2, então h(key) conteria 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 exemplo, se os dois últimos dígitos das chaves
não variam muito, então "`key % 100`" é uma
péssima função de dispersão.

Boas e más funções de dispersão

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

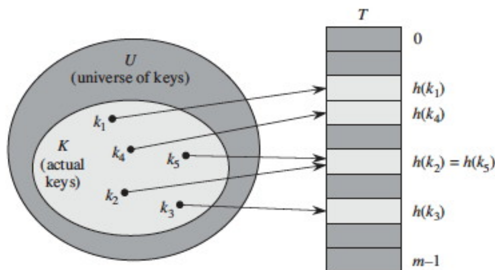
Por exemplo, se os dois últimos dígitos das chaves
não variam muito, então "`key % 100`" é uma
péssima função de dispersão.

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:

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

Colisões por listas encadeadas

Uma solução popular para resolver colisões é conhecida como **separate chaining**:

> *para cada índice h da tabela, há uma lista encadeada que armazena todos os objetos que a função de dispersão leva em h.*

Essa solução é muito boa se cada uma das "listas de colisão" resultar curta.

Colisões por listas encadeadas

Uma solução popular para resolver colisões
é conhecida como **separate chaining**:

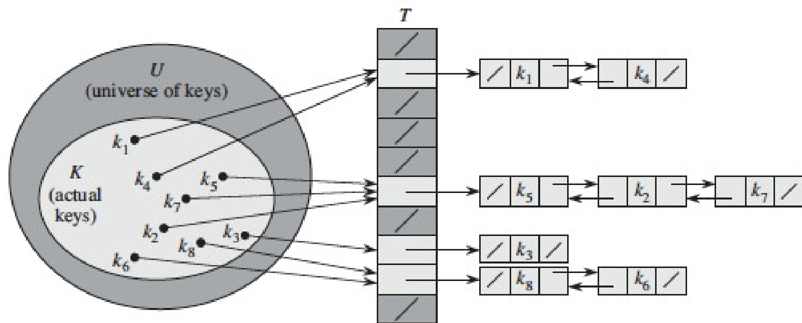
> *para cada índice h da tabela, há uma lista encadeada que armazena todos os objetos que a função de dispersão leva em h.*

Essa solução é muito boa se
cada uma das "listas de colisão" resultar curta.

Se o número total de chaves usadas for $\mathtt{n}$,
o comprimento de cada lista deveria,
idealmente, estar próximo de $\alpha = \mathtt{n}/\mathtt{m}$.

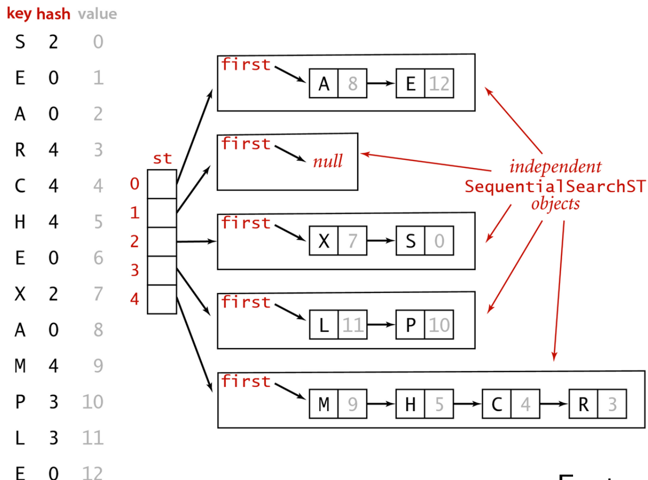
O valor α é chamado de **fator de carga**
($=$ *load factor*) da tabela.

Colisões por listas encadeadas



Fonte: CLRS

Colisões por listas encadeadas



Hashing with separate chaining for standard indexing client

Fonte: `algs4`

Colisões por listas encadeadas

SequentialSearchST: implementação de tabela de símbolos em uma lista ligada não ordenada.

```
static int n; /* número de chaves */
static int m; /* tam. da tabela de hash */
typedef struct pair Item;
struct pair {
   Key key;
   Value val;
   Item *next;
}
/* vetor da tabela de símbolos */
static Item **st;
```

Colisões por listas encadeadas

```
SeparateChainingHashSTInit(int size) {
   n = 0;
   m = size;
   st = mallocSafe(m * sizeof(*Item));
   for (int h = 0; h < m; h++)
      st[h] = NULL;
}
```

`hashCode(key)`: devolve a própria `key` ou
um número obtido da `key` se ela não for numérica

Colisões por listas encadeadas

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

Value getST(Key key) {
    int h = hash(key);
    return get(st[h], key);          /* busca na lista */
}

static void insertST(Key key, Value val) {
    int h = hash(key);
    put(st[h], key, val);
}
```

Rehashing

Ao inserir,
dobramos o tamanho da tabela se $\alpha \geq 10$.

```
void putST(Key key, Value val) {
    if (n >= 10*m) resize(2*m);
    int h = hash(key);
    if (get(st[h], key) == NULL) n++;
    put(st[h], key, val); /* insere ou altera valor */
}
```

Rehashing

```
static void resize(int size) {
   Item *p, *t = st; int h, aux = m;
   m = size;
   st = mallocSafe(size * sizeof(*Item));
   for (h = 0; h < m; h++) st[h] = NULL;
   for (h = 0; h < aux; h++)
      for (p = t[h]; p != NULL; p = p->next)
         insertST(p->key, p->val);
   freeST(t,aux);      /* libera a tabela velha */
}
```

`insertST(key, val)`: insere sem mexer no `n` nem fazer redimensionamento.

Rehashing

Redimensiona a ST se $\alpha \leq 2$.

```
void deleteST(Key key) {
   int h = hash(key);
   if (get(st[h], key)) {
      n--;
      delete(st[h], key);
      if (m > INIT_CAPACITY && n <= 2*m)
         resize(m/2);
   }
}
```

INIT_CAPACITY: guarda a capacidade inicial
e não deixa diminuir além disso.

Rehashing

Retorna uma fila com todas as chaves da ST.

```
Queue keys() {
   Item *p;
   Queue queue = queueInit(n);
                                   /* fila de Keys */
   for (int h = 0; h < m; h++)
      for (p = st[h]; p != NULL; p = p->next)
         enqueue(queue, p->key);
   return queue;
}
```

Hipótese do Hashing Uniforme

O comprimento médio das listas é $\alpha = \mathtt{n}/\mathtt{m}$.

Poderíamos ter uma lista muito longa
e todas as demais muito curtas . . .

Hipótese do Hashing Uniforme

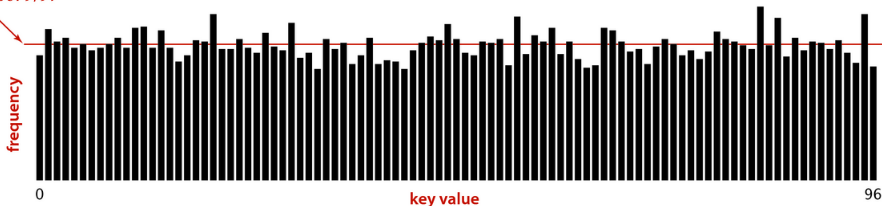
O comprimento médio das listas é $\alpha = \mathtt{n}/\mathtt{m}$.

Poderíamos ter uma lista muito longa
e todas as demais muito curtas ...

Para eliminar essa possibilidade,
precisamos saber ou supor algo sobre os dados.

Hipótese do Hashing Uniforme: Vamos supor que nossas funções de hashing distribuem as chaves pelo intervalo de inteiros $0\,..\,\mathtt{m}$-1 de maneira uniforme (todos os valores hash igualmente prováveis) e independente.

Hipótese do Hashing Uniforme



Hash value frequencies for words in *Tale of Two Cities* (10,679 keys, $M = 97$)

Fonte: algs4

Hipótese do Hashing Uniforme

Isso significa que se cada chave $\texttt{key}$ é escolhida de um universo $\texttt{U}$ de acordo com uma distribuição de probabilidade Pr (ou seja, $\mathsf{Pr}(\texttt{key})$ é a probabilidade de $\texttt{key}$ ser escolhida), então a hipótese do hashing uniforme nos diz que

$$\sum_{\texttt{key}:\texttt{h}(\texttt{key})=\texttt{j}} \mathsf{Pr}(\texttt{key}) = \frac{1}{\texttt{m}}$$

para $\texttt{j} = 0, 1, 2, \ldots, \texttt{m} - 1$.

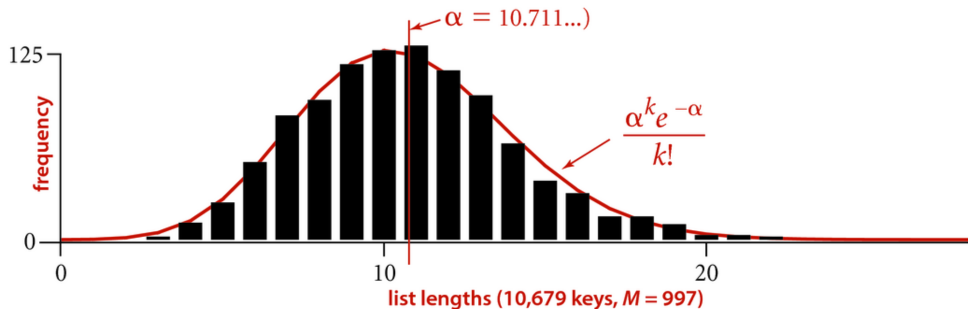
Hipótese do Hashing Uniforme

Proposição: Em uma tabela de hash encadeada com $\mathtt{m}$ listas e $\mathtt{n}$ chaves, se vale a hipótese do hashing uniforme, a probabilidade de que o número de chaves em cada lista não passa de $\alpha = \mathtt{n}/\mathtt{m}$ multiplicado por uma pequena constante é muito próxima de 1.

Exemplo: Se $\mathtt{n}/\mathtt{m} = 10$, a probabilidade de que uma lista tenha comprimento maior que 20 é inferior a 0,8%.

Hipótese do Hashing Uniforme

No gráfico, a altura de cada barra sobre o ponto `k` do eixo horizontal dá o número de listas que têm comprimento `k`:



Análise do separate chaining

Qual é o consumo de tempo de `get`(*key*)*?*

Análise é em termos do fator de carga $\alpha = \texttt{n}/\texttt{m}$
onde `n` é o número de itens na tabela
e `m` é o número de listas.

O fator de carga α é o número médio de itens por lista.

Análise do separate chaining

Qual é o consumo de tempo de `get(key)`*?*

Análise é em termos do fator de carga $\alpha = \mathtt{n}/\mathtt{m}$
onde `n` é o número de itens na tabela
e `m` é o número de listas.

O fator de carga α é o número médio de itens por lista.

O pior caso ocorre quando todas as `n` chaves vão para mesma lista.

Consumo de tempo médio depende de quão bem a função de hash `h()` distribui as chaves.

Consumo de tempo médio

A análise do consumo de tempo se apoia em uma suposição de **uniform hashing**.

Para $j = 0, \ldots, m-1$,
seja n_j o comprimento da lista $st[j]$.

Logo, $n = n_1 + n_2 + \cdots + n_{m-1}$.

O valor esperado de n_j é $E[n_j] = \alpha$.

Consumo de tempo médio

A análise do consumo de tempo se apoia em uma suposição de **uniform hashing**.

Para $j = 0, \ldots, m-1$, seja n_j o comprimento da lista $st[j]$.

Logo, $n = n_1 + n_2 + \cdots + n_{m-1}$.

O valor esperado de n_j é $E[n_j] = \alpha$.

Supondo que $h(key)$ é computada em tempo $O(1)$, o tempo gasto por `get(key)` depende do comprimento da lista `st[h(key)]`.

Busca mal sucedida

Considere dois casos:
busca mal sucedida (= `key` não está na `ST`) e
busca bem sucedida (= `key` está na `ST`).

Na busca mal sucedida
percorremos a lista `st[h[key]]` até o final.

Busca mal sucedida

Considere dois casos:
busca mal sucedida (= key não está na ST) e
busca bem sucedida (= key está na ST).

Na busca mal sucedida
percorremos a lista st[h[key]] até o final.

Hash uniforme nos diz que $\Pr[\mathtt{h}(\mathtt{key}) = \mathtt{j}] = 1/\mathtt{m}$.

O comprimento esperado da lista $\mathtt{st}[\mathtt{h}(\mathtt{key})]$ é α.

Busca mal sucedida

Considere dois casos:
busca mal sucedida (= key não está na ST) e
busca bem sucedida (= key está na ST).

Na busca mal sucedida
percorremos a lista st[h[key]] até o final.

Hash uniforme nos diz que $\Pr[\mathtt{h}(\mathtt{key}) = \mathtt{j}] = 1/\mathtt{m}$.

O comprimento esperado da lista $\mathtt{st}[\mathtt{h}(\mathtt{key})]$ é α.

Logo, o consumo de tempo médio de uma busca de uma chave key que não está em st[] é $\mathrm{O}(1+\alpha)$.

O termo "1" vem do consumo de tempo de $\mathtt{h}(\mathtt{key})$.

Busca bem sucedida

Suporemos que o elemento `key` procurado é igualmente provável de ser qualquer elemento na `ST`.

O número de chaves examinadas por `get(key)` é 1 mais o número de elementos na lista `st[h(key)]` antes de `key`.

Todos esses elementos foram inseridos na `ST` depois de `key`. Por quê?

Busca bem sucedida

Suporemos que o elemento `key` procurado é igualmente provável de ser qualquer elemento na `ST`.

O número de chaves examinadas por `get(key)` é 1 mais o número de elementos na lista `st[h(key)]` antes de `key`.

Todos esses elementos foram inseridos na `ST` depois de `key`. Por quê?

Precisamos encontrar, para cada `key` na `ST`, o número médio de elementos inseridos em `st[h(key)]` depois de `key`.

Esse é um trabalho para variáveis indicadoras!

Busca bem sucedida

Para $\texttt{j} = 1, \ldots, \texttt{n}$,
seja $\texttt{key}_\texttt{j}$ a $\texttt{j}$-ésima chave inserida na $\texttt{ST}$.

Para todo $\texttt{i}$ e $\texttt{j}$,
defina a variável aleatória indicadora:

$$X_{\texttt{ij}} = \texttt{I}_{\texttt{i},\texttt{j}} = \begin{cases} 1, & \text{se } \texttt{h}(\texttt{key}_\texttt{i}) = \texttt{h}(\texttt{key}_\texttt{j}) \\ 0, & \text{caso contrário} \end{cases}$$

Busca bem sucedida

Para $\texttt{j} = 1, \ldots, \texttt{n}$,
seja $\texttt{key}_\texttt{j}$ a $\texttt{j}$-ésima chave inserida na $\texttt{ST}$.

Para todo $\texttt{i}$ e $\texttt{j}$,
defina a variável aleatória indicadora:

$$X_{\texttt{ij}} = \texttt{I}_{\texttt{i},\texttt{j}} = \begin{cases} 1, & \text{se } \texttt{h}(\texttt{key}_\texttt{i}) = \texttt{h}(\texttt{key}_\texttt{j}) \\ 0, & \text{caso contrário} \end{cases}$$

Hash uniforme: $\text{Pr}(\texttt{h}(\texttt{key}_\texttt{i}) = \texttt{h}(\texttt{key}_\texttt{j})) = 1/\texttt{m}$.
Por quê?

Portanto, $\texttt{E}[X_{\texttt{ij}}] = 1/\texttt{m}$.

Busca bem sucedida

O número esperado de chaves examinadas em uma busca com sucesso é o número médio de chaves $\texttt{key}_\texttt{j}$ inseridas depois de $\texttt{key}_\texttt{i}$ **e** tais que $X_{\texttt{ij}} = 1$.

$$\texttt{E}\Big[\frac{1}{\texttt{n}}\sum_{\texttt{i}=1}^{\texttt{n}}\Big(1 + \sum_{\texttt{j}=\texttt{i}+1}^{\texttt{n}} X_{\texttt{ij}}\Big)\Big]$$

- o somatório interno conta as chaves $\texttt{key}_\texttt{j}$ inseridas depois de $\texttt{key}_\texttt{i}$ e que têm o mesmo valor de hash de $\texttt{key}_\texttt{i}$

Busca bem sucedida

O número esperado de chaves examinadas em uma busca com sucesso é o número médio de chaves $\texttt{key}_\texttt{j}$ inseridas depois de $\texttt{key}_\texttt{i}$ **e** tais que $X_{\texttt{ij}} = 1$.

$$\texttt{E}\Big[\frac{1}{\texttt{n}}\sum_{\texttt{i}=1}^{\texttt{n}}\Big(1 + \sum_{\texttt{j}=\texttt{i}+1}^{\texttt{n}} X_{\texttt{ij}}\Big)\Big]$$

- o somatório interno conta as chaves $\texttt{key}_\texttt{j}$ inseridas depois de $\texttt{key}_\texttt{i}$ e que têm o mesmo valor de hash de $\texttt{key}_\texttt{i}$
- o "1" é pelo custo de examinar $\texttt{key}_\texttt{i}$
- o somatório mais externo faz a soma sobre todas a chaves
- $1/\texttt{n}$ é para a média

Busca bem sucedida

Pela linearidade da esperança ... a média é

$$= \frac{1}{n}\sum_{i=1}^{n}\left(1 + \sum_{j=i+1}^{n} E[X_{ij}]\right)$$

$$= \frac{1}{n}\sum_{i=1}^{n}\left(1 + \sum_{j=i+1}^{n} \frac{1}{m}\right)$$

$$= 1 + \frac{1}{n\,m}\sum_{i=1}^{n}(n - i)$$

$$= 1 + \frac{1}{n\,m}\left(n^2 - \frac{n(n+1)}{2}\right)$$

Busca bem sucedida

Continuando ...

$$\begin{aligned} &= 1 + \frac{\mathtt{n} - 1}{2\,\mathtt{m}} \\ &= 1 + \frac{\mathtt{n}}{2\,\mathtt{m}} - \frac{\mathtt{n}}{2\,\mathtt{m}\,\mathtt{n}} \end{aligned}$$

Substituindo $\mathtt{n}/\mathtt{m}$ pelo fator de carga α obtemos

$$\begin{aligned} &= 1 + \frac{\alpha}{2} - \frac{\alpha}{2\mathtt{n}} \\ &= \mathrm{O}(1 + \alpha) \end{aligned}$$

Consumo de tempo

Seja $\mathtt{n}$ é o número de chaves
e $\mathtt{m}$ é o tamanho da tabela.

Supondo que a função hash distribuia
as chaves uniformemente em $[0\,..\,\mathtt{m}\text{-}1]$,
em uma tabela de distribuição com
listas encadeadas, o consumo de tempo de
get(), put() e delete() é $O(1 + \mathtt{n}/\mathtt{m})$.

Consumo de tempo

> Supondo que a função `hash` distribuia as chaves uniformemente em [0 . . m-1], em uma tabela de distribuição com listas encadeadas, o consumo de tempo de `get()`, `put()` e `delete()` é $O(1 + \alpha)$.

Se $n \leq c\,m$ para alguma constante c, ou seja, $n = O(m)$, então α é $O(1)$ e portanto $O(1 + \alpha)$ é **constante**.

Mais experimentos ainda

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

estrutura	ST	tempo
vetor	ordenada	1.5
skiplist	ordenada	1.1
árvore rubro-**negra**	ordenada	0.76
árvore binária de busca	ordenada	0.72
splay tree	ordenada	0.68
hash. encadeamento	não-ordenada	0.61
hash. encadeamento+MTF	não-ordenada	0.56

Tempos em segundos obtidos com `StopWatch`.