Fonte: ash.atozviews.com

Compacto dos melhores momentos

AULA 13

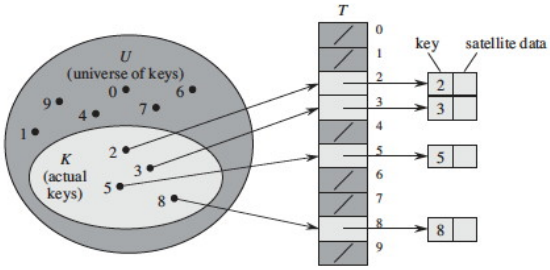
Endereçamento direto

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 arnazena 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:

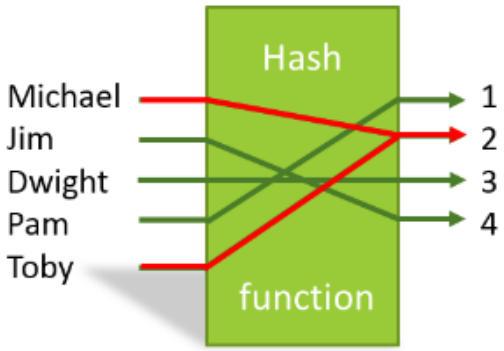
- possa ser calculada em $O(1)$ e
- espalhe *bem* as chaves pelo intervalo $0, \ldots, \mathtt{m}-1$.

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 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.

Conviver com colisões. . .



Fonte: `https://stackoverflow.com/`

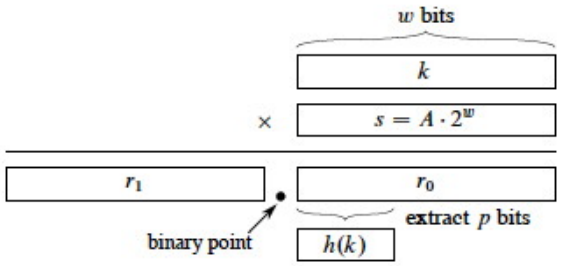
Hashing modular e multiplicativo

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`):

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

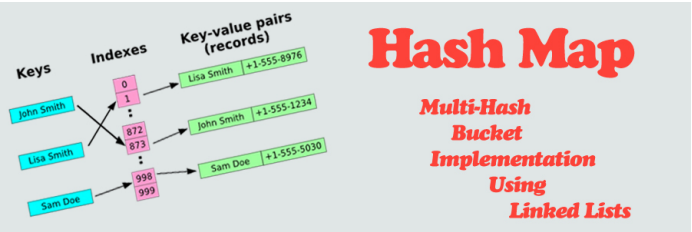
Hashing modular e multiplicativo

`h(key)` contém os bits iniciais da metade menos significa de `key` $\times$ `A` para uma constante $0 < \mathtt{A} < 1$.



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;

Colisões por listas encadeadas

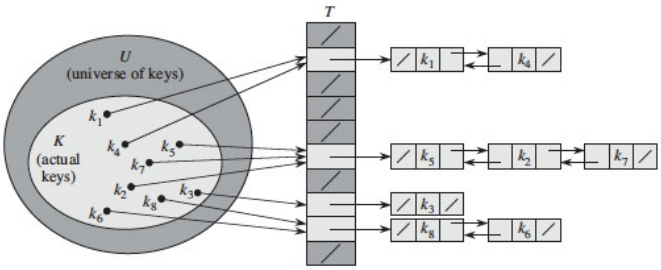
Uma solução popular para resolver colisões é conhecida como **separate chaining**:

> *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.

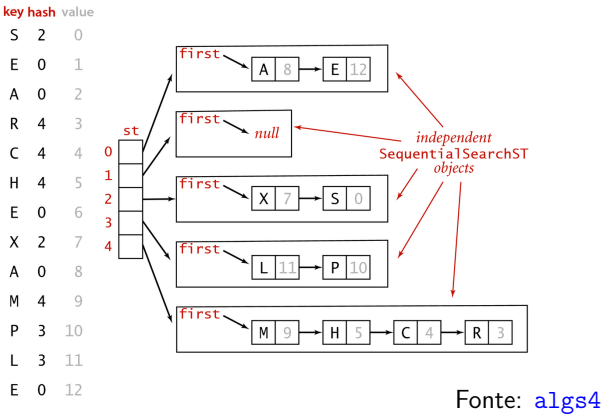
Se o número total de chaves usadas for n, o comprimento de cada lista deveria, idealmente, estar próximo de $\alpha = n/m$.
O valor α é chamado de **fator de carga** ((= *load factor*) da tabela.

Colisões por listas encadeadas



Fonte: CLRS

Colisões por listas encadeadas



Fonte: algs4

Hashing with separate chaining for standard indexing client

Colisões por listas encadeadas

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

```
public class SeparateChainingHashST<Key, Value>{
   private int n; // número de chaves

   private int m; // tam da tabela de hash

   // vetor de TSs
   private SequentialSearchST<Key,Value>[] st;
```

Colisões por listas encadeadas

```
public SeparateChainingHashST(int m) {
   this.m = m;
   st = (SequentialSearchST<Key,Value>[])
            new SequentialSearchST[m];
   for (int h = 0; h < m; h++)
       st[h]=new SequentialSearchST<Key,Value>();
}

public SeparateChainingHashST() {
   this(997); // tamanho de tabela default
}
```

Colisões por listas encadeadas

```
private int hash(Key key) {
  return (key.hashCode()&0x7fffffff)%m;
}
public Value get(Key key) {
  int h = hash(key);
  return st[h].get(key);
}
public void put(Key key, Value val) {
  int h = hash(key);
  st[h].put(key, val);
}
```

Rehashing

A seguir, dobramos o tamanho da tabela se $\alpha \geq 10$.

```
public void put(Key key, Value val) {
  if (n >= 10*m) resize(2*m);
  int h = hash(key);
  if (!st[h].contains(key)) n++;
  st[h].put(key, val);
}
```

Rehashing

```
private void resize(int chains) {
  SeparateChainingHashST<Key, Value> t;
  t= new Sep...HashST<Key,Value>(chains);
  for (int h = 0; h < m; h++) {
      for (Key key:  st[h].keys()) {
          t.put(key, st[h].get(key));
      }
  }
  this.m = t.m;
  this.n = t.n;
  this.st = t.st;
}
```

Rehashing

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

```
public void delete(Key key) {
  inth = hash(key);
  if (st[h].contains(key)) n--;
  st[h].delete(key);
  if (m > INIT_CAPACITY && n <= 2*m)
      resize(m/2);
}
```

Rehashing

Retorna as chaves da ST como iteráveis.

```
public Iterable<Key> keys() {
  Queue<Key> queue = new Queue<Key>();
  for (int h = 0; h < m; h++) {
      for (Key key:  st[h].keys())
          queue.enqueue(key);
  }
  return queue;
}
```

Hipótese do Hashing Uniforme

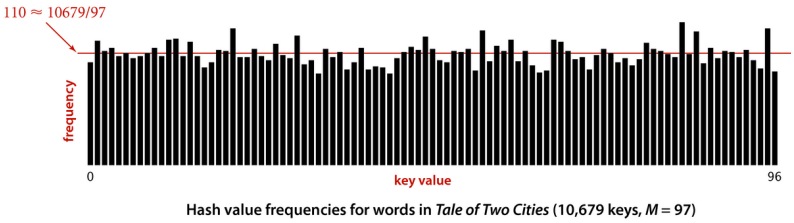
O comprimento médio das listas é $\alpha = n/m$.

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 $0..m$ 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 as cada chave `key` é escolhida de um universo `U` de acordo com uma distribuição de probabilidade Pr; ou seja, Pr(`key`) é a probabilidade de `key` ser escolhida. Então a hipótese do hashing uniforme nos diz que

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

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

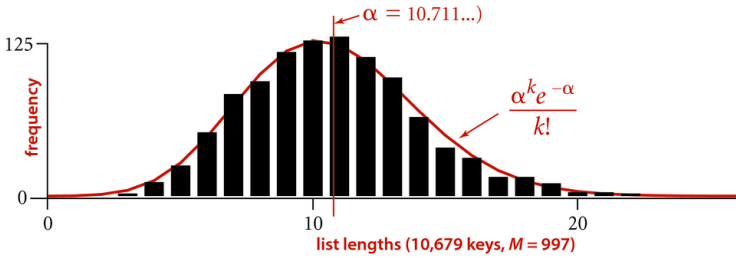
Hipótese do Hashing Uniforme

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

Exemplo: Se $\texttt{n}/\texttt{m} = 10$, 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`:



List lengths for `java FrequencyCounter 8 < tale.txt` using SeparateCha

Análise 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.

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 $\texttt{j} = 0, \ldots, \texttt{m}-1$ seja $\texttt{n}_\texttt{j}$ o comprimento da lista `st[j]`.

Logo, $\texttt{n} = \texttt{n}_1 + \texttt{n}_2 + \cdots + \texttt{n}_{\texttt{m}-1}$.

O valor esperado de $\texttt{n}_\texttt{j}$ é $\texttt{E}[\texttt{n}_\texttt{j}] = \alpha$.

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 malsucedida

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

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

Hash uniforme nos diz que $\Pr(h(key) = j) = 1/m$.

O comprimento esperado da lista st[h(key)] é α.

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

O termo "1" é devido ao consumo de tempo de h(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?

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 $j = 1, \ldots, n$ seja key_j a j-ésima chave inserida na ST.

Para todo i e j defina a variável aleatória indicadora:

$$X_{ij} = I_{i,j} = \begin{cases} 1, & \text{se } h(key_i) = h(key_j) \\ 0, & \text{caso contrário} \end{cases}$$

Hash uniforme: $\Pr(h(key_i) = h(key_j)) = 1/m$. Por quê?

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

Busca bem-sucedida

O número esperado de chaves examinadas em uma busca com sucesso é o número médio de chaves key_j inseridas depois de key_i **e** tal que $X_{ij} = 1$.

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

- o somatório interno conta as chaves key_j inseridas depois de key_i e que tem o mesmo valor de hash de key_i
- o "1" é pelo custo de examinar key_i
- o somatório mais externo faz a soma sobre todas a chaves
- $1/n$ é para a média

Busca bem-sucedida

Pela linearidade da esperança e . . . a média é

$$\begin{aligned} &= \frac{1}{n}\sum_{i=1}^{n}\Big(1 + \sum_{j=i+1}^{n} E[X_{ij}]\Big) \\ &= \frac{1}{n}\sum_{i=1}^{n}\Big(1 + \sum_{j=i+1}^{n} \frac{1}{m}\Big) \\ &= 1 + \frac{1}{nm}\sum_{i=1}^{n}(n-i) \\ &= 1 + \frac{1}{nm}\Big(n^2 - \frac{n(n+1)}{2}\Big) \end{aligned}$$

Busca bem-sucedida

Continuando . . .

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

Substituindo n/m pelo fator de carga α obtemos

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

Consumo de tempo

Seja n é o número de chaves e m é o tamanho da tabela.

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 $O(1 + n/m)$.

Consumo de tempo

Supondo 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 mais experimentos ainda

Consumo de tempo para se criar um ST em que a 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.