Tabelas de *hash*

Diego de Freitas Aranha – IC/UNICAMP

4 de setembro de 2007

Tabelas de *hash*

Tabelas de *hash* são úteis para implementar a funcionalidade de um dicionário T.

Dicionário T

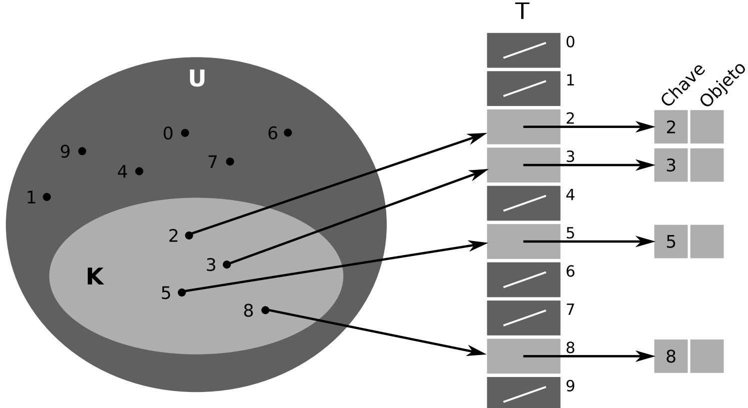
INSERIR(T, x): inserir elemento x no conjunto T;
REMOVER(T, x): remover elemento x do conjunto T;
BUSCAR(T, x): retornar elemento x no conjunto T, quando $x \in T$.

Exemplo: tabela de símbolos de um compilador;

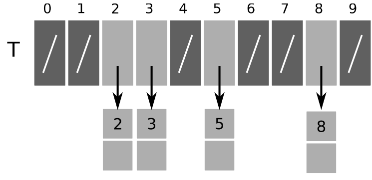
Objetivo: realizar operações em tempo constante!(tempo esperado)

Tabelas de endereçamento direto

- Cada elemento é identificado por uma **chave** em $\mathbb{N}$;
- Quando o universo de chaves $U = \{0, 1, \ldots, m-1\}$ é pequeno, a tabela pode ser implementada como um vetor. Cada posição representa uma chave de U e armazena um elemento x ou um ponteiro para x.



Tabelas de endereçamento direto - Algoritmos



INSERIR-ENDEREÇAMENTO-DIRETO(T, x)
    $T[key[x]] \leftarrow x$

REMOVER-ENDEREÇAMENTO-DIRETO(T, x)
    $T[key[x]] \leftarrow \text{NIL}$

BUSCAR-ENDEREÇAMENTO-DIRETO(T, k)
    return $T[k]$

Complexidade: as operações tomam tempo constante no pior caso.

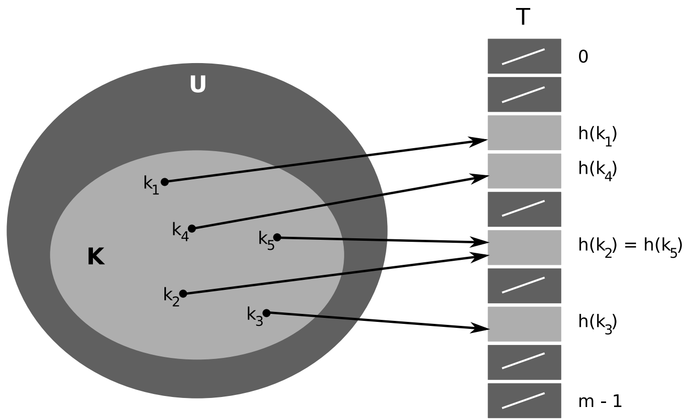
Problema

O universo de chaves pode ser muito grande ou esparso.

Solução: Utilizar uma **função de hash** h para mapear um elemento x à sua chave $k = h(x)$.

Tabelas de *hash*

A tabela é implementada como um vetor de m posições em que cada posição armazena um subconjunto de U.



Vantagem

Se K é o conjunto das chaves armazenadas, a tabela requer espaço $\Theta(|K|)$ ao invés de $\Theta(|U|)$.

Desvantagens

- **Colisão:** duas chaves podem ser mapeadas para a mesma posição!
- A busca na tabela requer $O(1)$ no caso médio, mas $O(n)$ no pior caso.

Como evitar colisões?

Problema

O número de colisões não pode ser muito grande.

O número de colisões depende de como a função de *hash* h espalha os elementos.

Solução: escolher uma função h determinística, mas com saída aparentemente aleatória.

Como evitar colisões?

Problema

O número de colisões não pode ser muito grande.

O número de colisões depende de como a função de *hash* h espalha os elementos.

Solução: escolher uma função h determinística, mas com saída aparentemente aleatória.

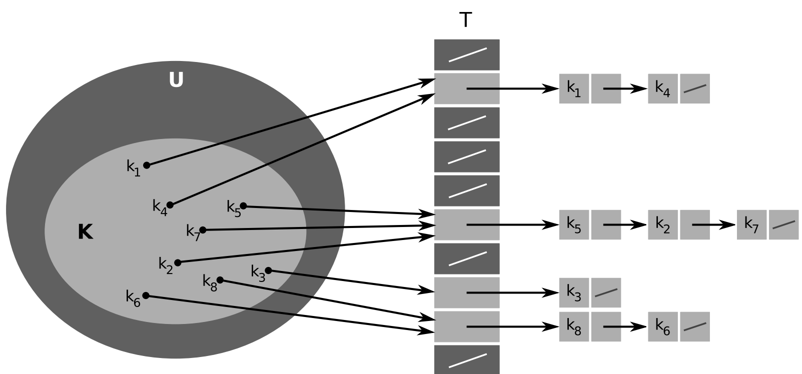
Problema

Como $|U| > m$, a escolha de h apenas minimiza o número de colisões.

Solução: tratar as colisões restantes de forma algorítmica, aplicando **encadeamento** ou **endereçamento aberto**.

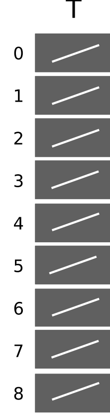
Encadeamento

Em uma tabela de *hash* encadeada, todos os elementos mapeados para uma mesma posição são armazenados em uma lista ligada.



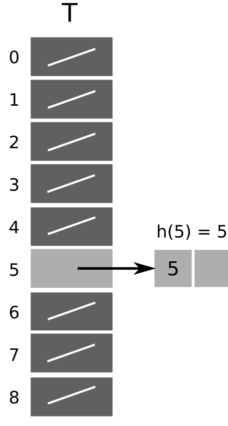
Inserção em tabela de *hash* encadeada

Inserção das chaves $\{5, 28, 19, 15, 20, 33\}$ em uma tabela com 9 posições, utilizando $h(k) = k \bmod 9$.



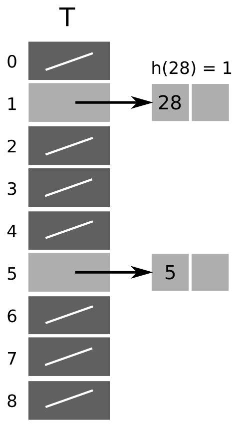
Inserção em tabela de *hash* encadeada

Inserção da chave 5 para $h(k) = k \bmod 9$.



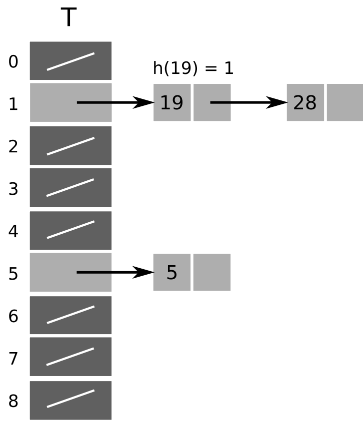
Inserção em tabela de *hash* encadeada

Inserção da chave 28 para $h(k) = k \bmod 9$.



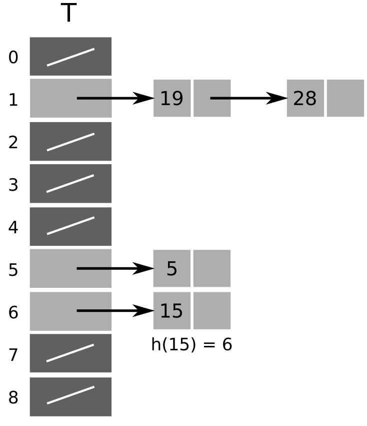
Inserção em tabela de *hash* encadeada

Inserção da chave 19 para $h(k) = k \bmod 9$.



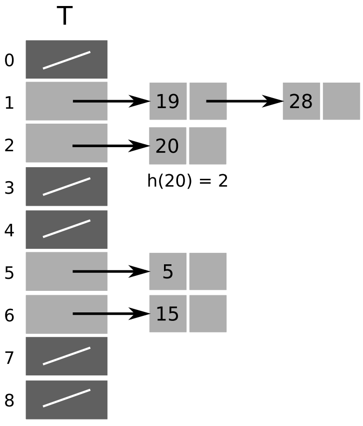
Inserção em tabela de *hash* encadeada

Inserção da chave 15 para $h(k) = k \bmod 9$.



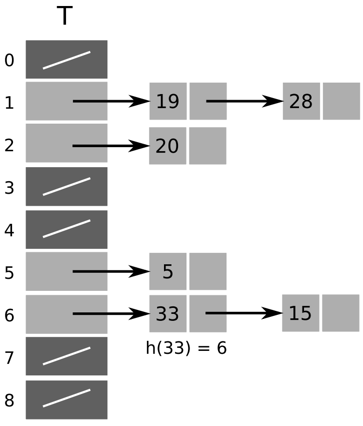
Inserção em tabela de *hash* encadeada

Inserção da chave 20 para $h(k) = k \bmod 9$.



Inserção em tabela de *hash* encadeada

Inserção da chave 33 para $h(k) = k \bmod 9$.



Encadeamento - Algoritmos

INSERIR-HASH-ENCADEADO(T, x)
Inserir x no início da lista $T[h(key[x])]$;

REMOVER-HASH-ENCADEADO(T, x)
Remover x da lista $T[h(key[x])]$;

BUSCAR-HASH-ENCADEADO(T, k)
Buscar por um elemento com chave k na lista $T[h(k)]$;

Encadeamento - Algoritmos

INSERIR-HASH-ENCADEADO(T, x)
Inserir x no início da lista $T[h(key[x])]$;

REMOVER-HASH-ENCADEADO(T, x)
Remover x da lista $T[h(key[x])]$;

BUSCAR-HASH-ENCADEADO(T, k)
Buscar por um elemento com chave k na lista $T[h(k)]$;

Complexidade

- **Inserção**: $O(1)$;
- **Remoção**: $O(1)$ com lista duplamente ligada.
- **Busca sem sucesso**: depende do comprimento de $T[h(k)]$;
- **Busca com sucesso**: depende do número de elementos antes de x em $T[h(key[x])]$;

Encadeamento - Análise

No pior caso, todos os elementos são mapeados para a mesma posição e a busca custa $\Theta(n)$ mais o cálculo de h.

Mapeamento uniforme simples

No caso médio, podemos assumir que um elemento pode ser mapeado para qualquer posição igualmente e que dois elementos são mapeados independentemente:

$$Pr\{h(k_i) = h(k_j)\} = \frac{1}{m}.$$

Encadeamento - Análise

Definição

Em uma tabela de *hash* com m posições que armazena n elementos, o **fator de carga** α é definido como $\frac{n}{m}$.

Seja n_j o comprimento da lista $T[j]$. O valor esperado de n_j é α. Assume-se que calcular $k = h(x)$ toma $O(1)$.

Encadeamento - Análise

Definição

Em uma tabela de *hash* com m posições que armazena n elementos, o **fator de carga** α é definido como $\frac{n}{m}$.

Seja n_j o comprimento da lista $T[j]$. O valor esperado de n_j é α. Assume-se que calcular $k = h(x)$ toma $O(1)$.

Complexidade

- **Busca sem sucesso**: examina-se toda a lista $T[k]$ com tamanho esperado α. A complexidade é $O(1 + \alpha)$.
- **Busca com sucesso**: examinam-se os elementos anteriores a x e o próprio x. Na média, examinam-se:

$$1 + \frac{1}{n}\sum_{i=1}^{n}\sum_{j=i+1}^{n} 1/m = O(1 + \frac{1}{n}\frac{1}{m}n^2) = O(1 + \alpha).$$

Encadeamento - Análise

- Se o número de posições é proporcional ao número de elementos, ou $n = O(m)$, o fator de carga é $\alpha = O(1)$ e a busca toma tempo constante no caso médio;
- Todas as operações tomam tempo constante em média.

Projeto de funções de *hash*

Problema

Funções de *hash* verdadeiramente aleatórias não podem ser implementadas com tempo constante.

Precisamos de uma função que pareça aleatória, ou seja, mapeie um elemento para cada posição com probabilidade próxima de $\frac{1}{m}$. Isto depende da distribuição das entradas.

Projeto de funções de *hash*

Exemplos:

- Se as entradas são valores reais uniformemente distribuídos no intervalo $[0, 1)$, podemos usar $h(k) = \lfloor km \rfloor$;
- Se as entradas são identificadores de um programa, h deve diminuir a probabilidade de elementos parecidos como "`pt`" e "`pts`" colidirem.

Heurísticas - Método da divisão

Método da divisão

A função h é definida como $h(k) = k \bmod m$.

A qualidade depende da escolha de m:

- Se $m = 2^p$, a função escolhe os *bits* menos significativos de k;
- Se m é um número primo não muito próximo de uma potência de 2, h considera mais *bits* de k.

Heurísticas - Método da divisão

Método da divisão

A função h é definida como $h(k) = k \bmod m$.

A qualidade depende da escolha de m:

- Se $m = 2^p$, a função escolhe os *bits* menos significativos de k;
- Se m é um número primo não muito próximo de uma potência de 2, h considera mais *bits* de k.

Exemplo: Para armazenar $n = 2000$ elementos em uma tabela de *hash* , onde uma busca sem sucesso pode visitar até 3 elementos, m deve ser primo e próximo de $\frac{2000}{3}$. Um bom valor para m é 701.

Heurísticas - Método da multiplicação

Método da multiplicação

A função h é definida como $h(k) = \lfloor m(kc - \lfloor kc \rfloor) \rfloor$, para uma constante $0 < c < 1$.

Neste caso, o valor de m não é crítico, mas a escolha de c depende das características da entrada. Knuth sugere $c = \frac{\sqrt{5}-1}{2}$.
Se $m = 2^p$, h pode ser implementada eficientemente com operações de *bit*.

Método da multiplicação

A função h é definida como $h(k) = \lfloor m(kc - \lfloor kc \rfloor) \rfloor$, para uma constante $0 < c < 1$.

Neste caso, o valor de m não é crítico, mas a escolha de c depende das características da entrada. Knuth sugere $c = \frac{\sqrt{5}-1}{2}$.
Se $m = 2^p$, h pode ser implementada eficientemente com operações de *bit*.

Exemplo: Se $k = 123456$, $m = 16384$ e a sugestão acima é seguida, $h(k) = 67$.

Mapeamento universal

Problema

Heurísticas são determinísticas e podem ser manipuladas de forma indesejada. Um adversário pode escolher as chaves de entrada para que todas colidam.

Solução: no início da execução, escolher aleatoriamente uma função de *hash* de uma classe $\mathcal{H}$ de funções determinísticas.

Mapeamento universal

Problema

Heurísticas são determinísticas e podem ser manipuladas de forma indesejada. Um adversário pode escolher as chaves de entrada para que todas colidam.

Solução: no início da execução, escolher aleatoriamente uma função de *hash* de uma classe $\mathcal{H}$ de funções determinísticas.

Definição

Uma classe $\mathcal{H}$ de funções de *hash* é **universal** se o número de funções $h \in \mathcal{H}$ em que $h(k_i) = h(k_j)$ é $\frac{|\mathcal{H}|}{m}$.

A colisão entre duas chaves k_i e k_j ocorre com probabilidade $\frac{1}{m}$, a mesma probabilidade de colisão se $h(k_i)$ e $h(k_j)$ fossem selecionados aleatoriamente em U.

O limite superior para o número esperado de colisões para cada chave k, baseando-se na escolha da função de *hash* é:

$$\sum_{l \in T, l \neq k} \frac{1}{m}.$$

- Se $k \notin T$, a lista $n_{h(k)}$ tem tamanho esperado $\frac{n}{m} = \alpha$;
- Se $k \in T$, a lista $n_{h(k)}$ tem tamanho esperado $1 + \frac{n-1}{m} < 1 + \alpha$.

Mapeamento universal - Análise

Teorema 11.3

Com mapeamento universal e encadeamento, o tamanho esperado de cada lista n_i é no máximo $1 + \alpha$.

Corolário 11.4

Com mapeamento universal e encadeamento, uma tabela com m posições realiza qualquer seqüência de n operações contendo $O(m)$ inserções em tempo esperado $\Theta(n)$.

Complexidade: as operações tomam tempo constante em média.

Mapeamento universal - Projeto

Seja p um número primo tal que $p > |U|$. Denota-se por $Z_p = \{0, 1, \ldots, p-1\}$ e $Z_p^* = Z_p - \{0\}$.

Teorema 11.5

A classe $\mathcal{H}_{p,m}$ de funções $\{h_{a,b} : a \in Z_p^*, b \in Z_p\}$ é universal para $h_{a,b}(k) = ((ak + b) \bmod p) \bmod m$.

Mapeamento universal - Projeto

Seja p um número primo tal que $p > |U|$. Denota-se por $Z_p = \{0, 1, \ldots, p-1\}$ e $Z_p^* = Z_p - \{0\}$.

Teorema 11.5

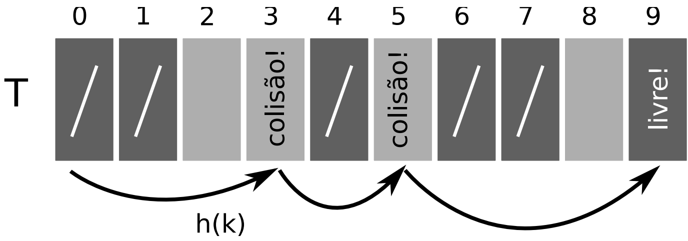
A classe $\mathcal{H}_{p,m}$ de funções $\{h_{a,b} : a \in Z_p^*, b \in Z_p\}$ é universal para $h_{a,b}(k) = ((ak + b) \bmod p) \bmod m$.

Exemplo: Se $p = 17$ e $m = 16$, temos $h_{3,4}(8) = 5$.

A classe tem $p(p-1)$ funções distintas. A universalidade segue das propriedades da redução módulo o número primo p.

Endereçamento aberto

Em uma tabela de *hash* com endereçamento aberto, todos os elementos são armazenados na tabela propriamente dita. O espaço gasto com encadeamento é economizado e a colisão é tratada com a busca de uma nova posição para inserção.
Naturalmente, o fator de carga não pode exceder o valor 1.



Endereçamento aberto - Algoritmos

Durante a inserção, uma seqüência de posições é testada até que uma posição livre seja encontrada. A função de *hash* é modificada para receber um argumento que armazena o número do teste.

Inserir-Hash-Aberto(T, x)
  $i \leftarrow 0$
  repeat $j \leftarrow h(k, i)$
    if $T[j] = \text{NIL}$ **then**
      $T[j] \leftarrow k$
      return j
    else $i \leftarrow i + 1$
  until $i = m$
  error overflow

Endereçamento aberto - Algoritmos

O algoritmo de busca percorre a mesma seqüência examinada pelo algoritmo de inserção quando k foi inserido.

```
BUSCAR-HASH-ABERTO(T, k)
    i ← 0
    repeat j ← h(k, i)
        if T[j] = k then
            return j
        i ← i + 1
    until T[j] = NIL or i = m
    return NIL
```

A remoção de elementos é difícil. Pode-se utilizar um valor especial DELETED para marcar elementos removidos, mas o custo da busca deixa de depender do fator de carga.

Heurísticas - Teste linear

A função de *hash* deve produzir como seqüência de teste uma permutação de $\{0, 1, \ldots, m-1\}$.

Teste linear

Seja h' uma função de *hash* auxiliar. A função h é definida como $h(k, i) = (h'(k) + i) \bmod m$.

Heurísticas - Teste linear

A função de *hash* deve produzir como seqüência de teste uma permutação de $\{0, 1, \ldots, m - 1\}$.

Teste linear

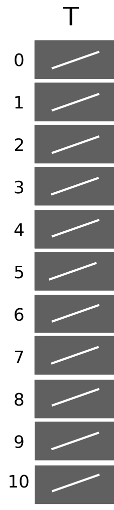
Seja h' uma função de *hash* auxiliar. A função h é definida como $h(k, i) = (h'(k) + i) \bmod m$.

Vantagem: fácil implementação.

Desvantagem: é suscetível a **agrupamento primário**. Ou seja, são construídas seqüências longas de posições ocupadas, o que degrada o desempenho da busca.

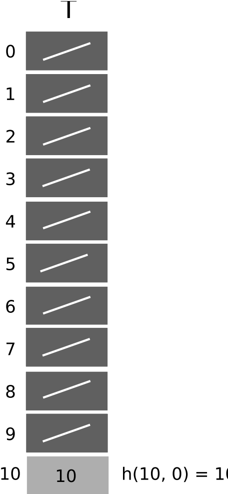
Inserção em tabela com endereçamento aberto

Inserção das chaves $\{10, 22, 31, 4, 15, 28, 59\}$ em uma tabela de tamanho 11 com teste linear e função $h(k, i) = (k + i) \bmod 11$.

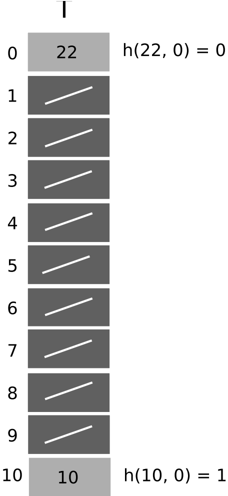


Endereçamento aberto e teste linear

Inserção da chave 10 para $h(k, i) = (k + i) \bmod 11$.

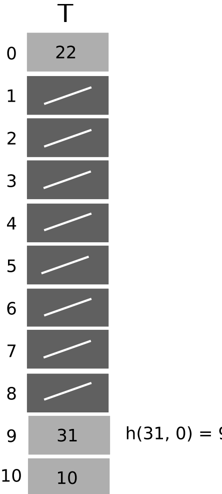


Inserção da chave 22 para $h(k, i) = (k + i) \bmod 11$.



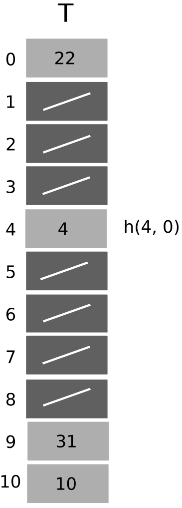
Endereçamento aberto e teste linear

Inserção da chave 31 para $h(k, i) = (k + i) \bmod 11$.



Endereçamento aberto e teste linear

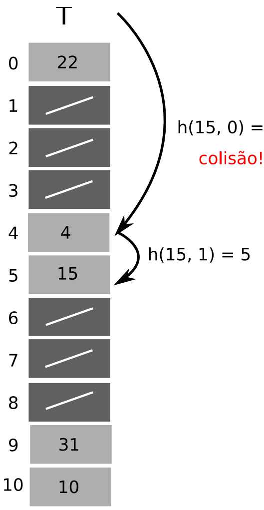
Inserção da chave 4 para $h(k, i) = (k + i) \bmod 11$.



h(4, 0) = 4

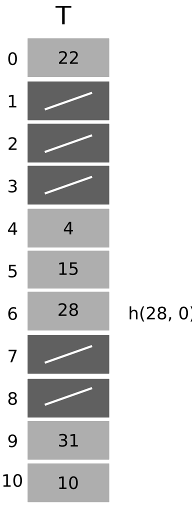
Endereçamento aberto e teste linear

Inserção da chave 15 para $h(k, i) = (k + i) \bmod 11$.



Endereçamento aberto e teste linear

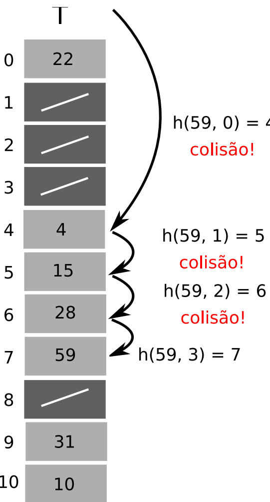
Inserção da chave 28 para $h(k, i) = (k + i) \bmod 11$.



h(28, 0) = 6

Endereçamento aberto e teste linear

Inserção da chave 59 para $h(k, i) = (k + i) \bmod 11$.



Heurísticas - Teste quadrático

Teste quadrático

Seja h' uma função de *hash* auxiliar, c_1 e c_2 constantes não-nulas. A função h é definida como $h(k, i) = (h'(k) + c_1 i + c_2 i^2) \bmod m$.

Heurísticas - Teste quadrático

Teste quadrático

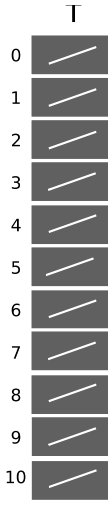
Seja h' uma função de *hash* auxiliar, c_1 e c_2 constantes não-nulas. A função h é definida como $h(k, i) = (h'(k) + c_1 i + c_2 i^2) \bmod m$.

Vantagem: é imune a agrupamento primário.

Desvantagem: é suscetível a **agrupamento secundário**. Ou seja, as seqüências de teste são idênticas para duas chaves k_i e k_j tais que $h'(k_i) = h'(k_j)$.

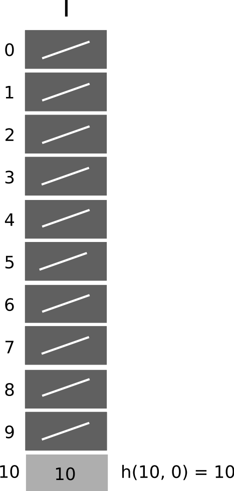
Inserção em tabela com endereçamento aberto

Inserção das chaves $\{10, 22, 31, 4, 15, 28, 59\}$ em uma tabela de tamanho 11 com teste quadrático e $h(k, i) = (k + i + 3i^2) \bmod 11$.



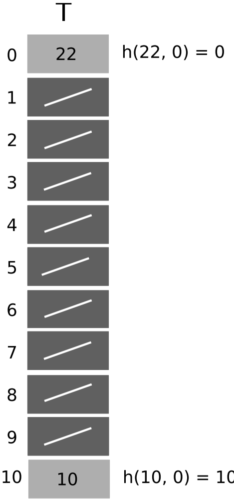
Endereçamento aberto e teste quadrático

Inserção da chave 10 para $h(k, i) = (k + i + 3i^2) \bmod 11$.



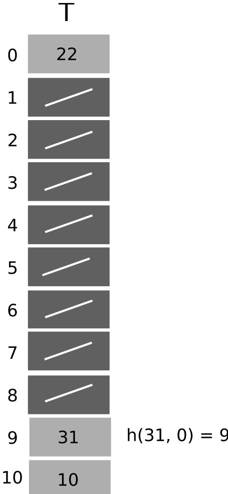
Endereçamento aberto e teste quadrático

Inserção da chave 22 para $h(k,i) = (k + i + 3i^2) \bmod 11$.



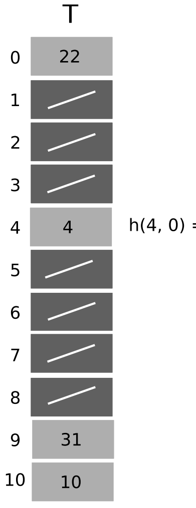
Endereçamento aberto e teste quadrático

Inserção da chave 31 para $h(k, i) = (k + i + 3i^2) \bmod 11$.



Endereçamento aberto e teste quadrático

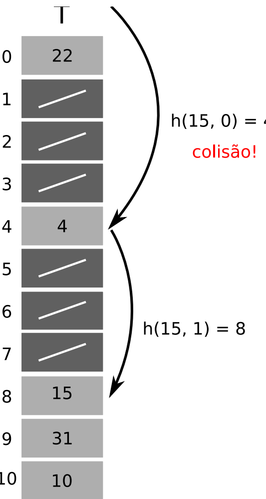
Inserção da chave 4 para $h(k, i) = (k + i + 3i^2) \bmod 11$.



h(4, 0) = 4

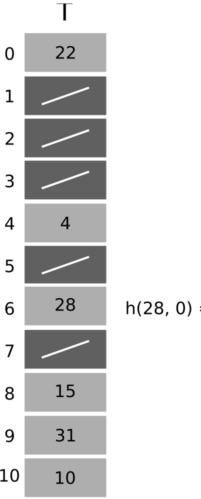
Endereçamento aberto e teste quadrático

Inserção da chave 15 para $h(k,i) = (k + i + 3i^2) \bmod 11$.



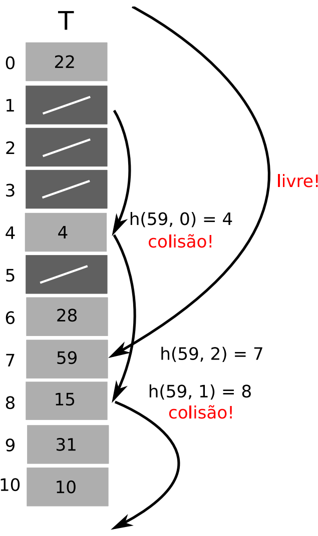
Endereçamento aberto e teste quadrático

Inserção da chave 28 para $h(k, i) = (k + i + 3i^2) \bmod 11$.



Endereçamento aberto e teste quadrático

Inserção da chave 59 para $h(k, i) = (k + i + 3i^2) \bmod 11$.



Heurísticas - Duplo mapeamento

Duplo mapeamento

Sejam h_1 e h_2 funções de *hash* auxiliares. A função h é definida como $h(k, i) = (h_1(k) + ih_2(k)) \bmod m$.

É preciso que $h_2(k)$ e m sejam relativamente primos:

- Escolhe-se $m = 2^p$ e $h_2(k)$ com saída sempre ímpar;
- Escolhe-se m primo e $h_2(k)$ com saída sempre menor que m.

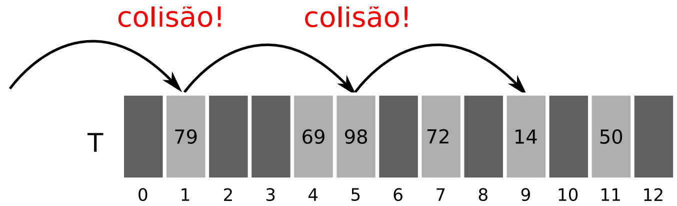
Heurísticas - Duplo mapeamento

Vantagem: o mapeamento duplo considera $\Theta(m^2)$ seqüências de teste, já que cada par $(h_1(k), h_2(k))$ produz uma nova seqüência. Teste linear ou quadrático apenas consideram $\Theta(m)$ seqüencias.

Desvantagem: projeto e implementação mais difícil.

Exemplo:
Sejam $h_1(k) = k \bmod 13$ e $h_2(k) = 1 + (k \bmod 11)$.

Então $h(14, 0) = 1$, $h(14, 1) = 5$ e $h(14, 2) = 9$.



Endereçamento aberto - Análise

Mapeamento uniforme

No caso médio, podemos assumir que cada chave é mapeada igualmente para cada uma das $m!$ seqüências de teste possíveis no conjunto $\{0, 1, \ldots, m-1\}$.

Mapeamento uniforme

No caso médio, podemos assumir que cada chave é mapeada igualmente para cada uma das $m!$ seqüências de teste possíveis no conjunto $\{0, 1, \ldots, m-1\}$.

Complexidade

- **Busca sem sucesso**: proporcional ao número de testes feitos. O primeiro teste sempre é feito e falha com probabilidade α. O segundo falha com probabilidade α^2, e assim por diante. O número de testes é limitado por:

$$\sum_{i=1}^{\infty} \alpha^{i-1} = \frac{1}{1-\alpha} = O(1).$$

Endereçamento aberto - Análise

Complexidade

- **Inserção**: consiste em encontrar uma posição desocupada (busca sem sucesso). Logo, a complexidade é $O(1)$ no caso médio.

Complexidade

- **Inserção**: consiste em encontrar uma posição desocupada (busca sem sucesso). Logo, a complexidade é $O(1)$ no caso médio.
- **Busca com sucesso**: a busca por uma chave k segue a mesma seqüência de teste percorrida na inserção de k. Se k foi a $(i+1)$-ésima chave inserida, o número esperado de testes em uma busca por k é limitado por $\frac{1}{1-i/m} = \frac{m}{m-i}$. A média sobre todas as n chaves é:

$$\frac{1}{n}\sum_{i=0}^{n-1}\frac{m}{m-i} \leq \frac{1}{\alpha} ln \frac{1}{1-\alpha} = O(1).$$

Mapeamento perfeito

Quando o conjunto de chaves K é estático, tabelas de *hash* podem ser usadas para obter alto desempenho mesmo no pior caso.

Exemplo: O conjunto de palavras reservadas de uma linguagem de programação é estático.

Mapeamento perfeito

Quando o conjunto de chaves K é estático, tabelas de *hash* podem ser usadas para obter alto desempenho mesmo no pior caso.

Exemplo: O conjunto de palavras reservadas de uma linguagem de programação é estático.

Definição

Uma técnica de mapeamento é chamada **perfeita** se o número de acessos à memória necessários para uma busca é $O(1)$ no pior caso.

Mapeamento perfeito

Problema

Projetar um esquema de mapeamento em que todas as buscas tomem tempo constante no pior caso.

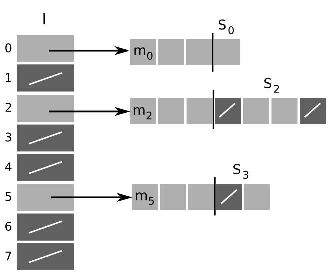
Solução: utilizar um esquema com mapeamento em dois níveis, e mapeamento universal em cada nível.

Mapeamento perfeito

O primeiro nível é idêntico a uma tabela de *hash* encadeada. As n chaves são mapeadas em m posições por uma função h universal.

No lugar da lista de chaves que colidem na posição j, utiliza-se uma **tabela de hash secundária** S_j com uma função associada h_j. A escolha de h_j permite impedir colisões no segundo nível.

Para isso, o tamanho m_j da tabela de *hash* S_j deve ser o quadrado da quantidade n_j de chaves mapeadas para a posição j.



Mapeamento perfeito - Projeto

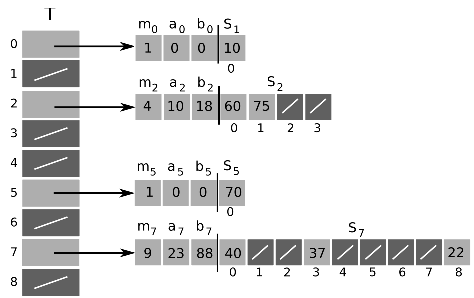
Pode-se escolher as funções tais que:

- A função h de primeiro nível seja escolhida de uma classe universal $\mathcal{H}_{p,m}$;
- As funções h_j sejam escolhidas de classes universais $\mathcal{H}_{p,m_j}$.

Inserção em tabela com mapeamento perfeito

Inserção das chaves $\{10, 22, 37, 40, 60, 70, 75\}$. Para a tabela externa, $m = 9$ e $h(k) = ((3k + 42) \bmod 101) \bmod 9$.

$h(10) = (72 \bmod 101) \bmod 9 = 0$, $h_0(10) = 0$;
$h(22) = (108 \bmod 101) \bmod 9 = 7$, $h_7(22) = (594 \bmod 101) \bmod 9 = 8$.
$h(37) = (153 \bmod 101) \bmod 9 = 7$, $h_7(7) = (249 \bmod 101) \bmod 9 = 3$.



Mapeamento perfeito - Análise

Teorema 11.9

Ao armazenar n chaves em uma tabela com tamanho $m = n^2$ e função h escolhida aleatoriamente de uma classe universal de funções, a probabilidade de haver qualquer colisão é menor que $\frac{1}{2}$.

O número esperado de colisões é igual ao produto entre o número de colisões possíveis e a probabilidade de colisão:

$$\binom{n}{2} \cdot \frac{1}{m} = \frac{n^2 - n}{2} \cdot \frac{1}{n^2} < \frac{1}{2}$$

Portanto, uma função de *hash* h pode ser cuidadosamente escolhida para não produzir colisões no conjunto estático K.

Mapeamento perfeito - Análise

Como n é grande, $m = n^2$ pode ser excessivo e esta abordagem é usada apenas nas tabelas secundárias. Cada tabela S_j tem tamanho $m_j = n_j^2$ e permite busca em tempo constante sem colisões.

Teorema 11.10

Armazenar n chaves em uma tabela de dois níveis com tamanhos $m = n$ e $m_j = n_j^2$ usando uma função de *hash* h escolhida aleatoriamente de uma classe universal requer memória $\Theta(n)$.

A quantidade de memória necessária para essa configuração tem como valor esperado:

$$\sum_{j=0}^{m-1} n_j^2 \leq 2n - 1 = \Theta(n)$$

Conclusões

- Tabelas de *hash* são os dicionários mais eficientes quando apenas inserção, remoção e busca precisam ser suportadas;
- Se mapeamento uniforme é utilizado, cada operação toma tempo constante no caso médio;
- Se mapeamento universal é utilizado, tempo constante é mantido mesmo sob atuação de adversários;
- Se características da entrada podem ser exploradas, funções de *hash* baseadas em heurísticas têm bom desempenho e fácil implementação;

Conclusões

- Para resolução de colisões, encadeamento é o método mais simples, mas gasta mais espaço;
- Endereçamento aberto tem implementação mais difícil ou que pode ser suscetível a efeitos de agrupamento.
- Mapeamento perfeito é ótimo para conjuntos estáticos de chaves e tem consumo de memória $\Theta(n)$.

Exercício 11.2-1

Suponha que uma função de *hash* é usada para mapear n chaves distintas em um vetor T de tamanho m. Assumindo mapeamento uniforme simples, qual o número esperado de colisões? Mais precisamente, qual é a cardinalidade esperada do conjunto $\{\{k, l\} : k \neq l \text{ e } h(k) = h(l)\}$?

Exercícios

Exercício 11.2-1

Suponha que uma função de *hash* é usada para mapear n chaves distintas em um vetor T de tamanho m. Assumindo mapeamento uniforme simples, qual o número esperado de colisões? Mais precisamente, qual é a cardinalidade esperada do conjunto $\{\{k, l\} : k \neq l \text{ e } h(k) = h(l)\}$?

Exercício 11.2-3

Se modificássemos o esquema de encadeamento para que cada lista fosse mantida em ordem, que impacto causaríamos ao tempo de execução de inserções, remoções e buscas?

Exercício 11.4-2

Escreva pseudocódigo para o algoritmo REMOVER-HASH-ABERTO e modifique o algoritmo INSERIR-HASH-ABERTO para levar em conta o valor especial DELETED.