Hashtables - Tabelas de dispersão

Tabelas e funções de dispersão.
Factor de carga, colisões e tipos de dispersão:
- encadeamento separado (cadeias explícitas)
- endereçamento aberto:
  - -acesso linear, quadrático e com duplo hash

Rehashing

20-Maio-2021

Hash Tables -“Tabelas de Dispersão”

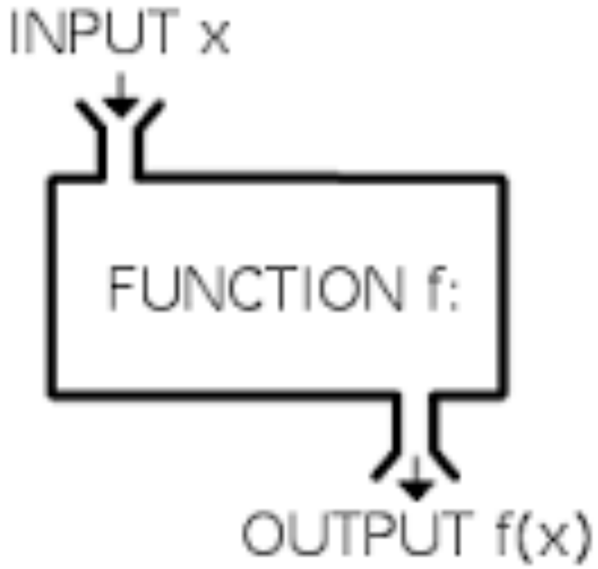
- Do inglês "Hash"
  - Pode ser traduzido por “picado/mastigado”
  - eventualmente outros significados…
  - Endereçamento por cálculo
- Ideia /GOAL:
  - Fazer pesquisa, inserção e remoção em $T = O(1)$

Hash tables

- Usa-se um array, com os elementos indexados por uma função da chave
  - Idealmente, a chave seria o índice
  - No entanto, não vai ser tão simples, porque:
    - o domínio das chaves é potencialmente infinito
    - o tamanho do vector será sempre limitado

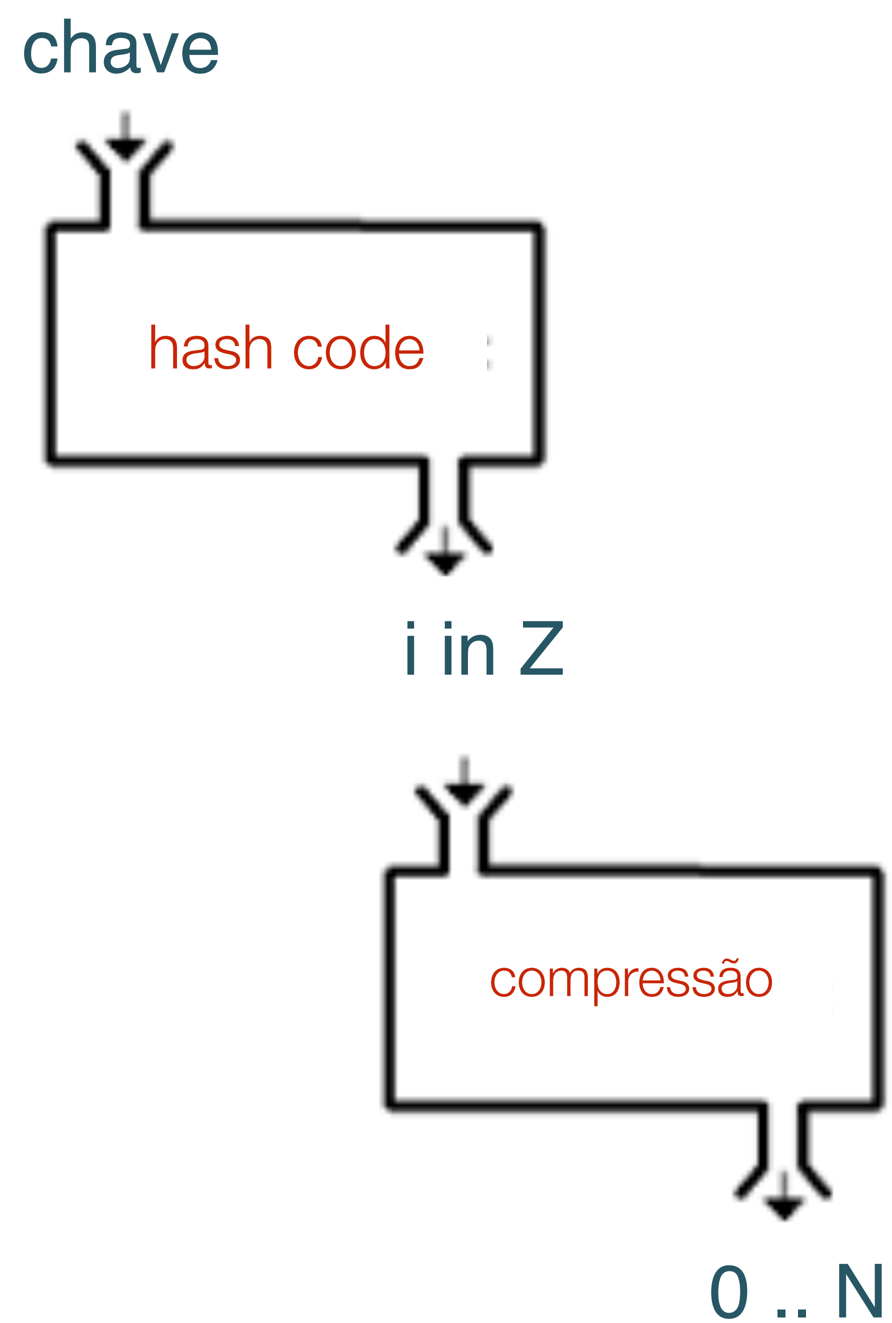
Hash tables: Exemplo

- Função f?

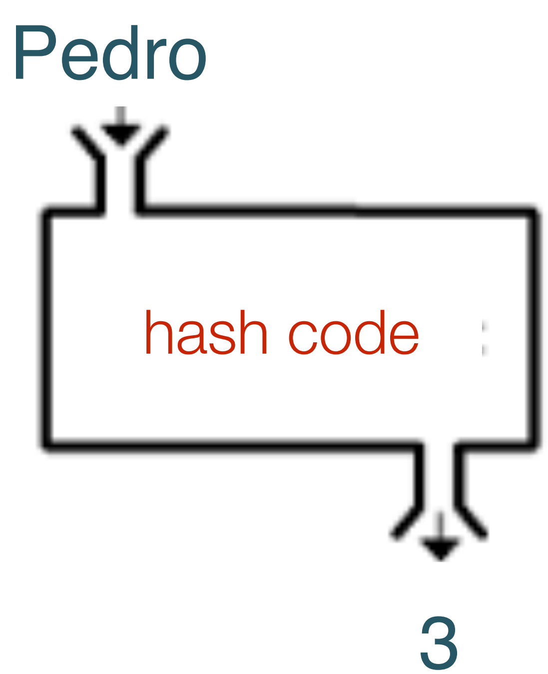


Hash tables: Exemplo

- `Função hash?`



Utilização



	nome	telefone
0	Ana	915636758
1		
2	João	917344121
3	Pedro	961124839
4		
5		
6		
7		
8		
9		

Formulação do Problema

- É preciso
  - Definir a função de hash, usada para gerar índices a partir de chaves
  - Garantir que esta é adequada às caracteristicas
    - Do domínio (e distribuição) das chaves
    - Do tamanho do vector
  - Saber o que fazer quando duas chaves dão o mesmo índice (colisão)

Escolha da Função de Hash

- Escolha determinante(para o quê!?)
- Se domínio das chaves são inteiros
  - Pode-se usar simplesmente:
    - $hash(key) = Key \,\%\, TableSize$
    - Problema:

```
Index HashInt( int Key, int TableSize ){
        return Key % TableSize;
}
```

    - Tabela de tamanho 10 e o dígito das unidades das chaves é o mesmo;
      - Solução: assegurar que o tamanho da tabela é um número primo

Escolha função de hash

- Se domínio das chaves são strings..
  - Procurar funções que produzam uma boa distribuição em termos do tamanho da tabela
- Se quero converter uma String num inteiro, uma primeira ideia será?
  - Uma opção será somar o ASCII dos caracteres da String

```
Index HashSt1( const char *Key, int TableSize ){
    unsigned int HashVal = 0;

    while( *Key != '\0' )
        HashVal += *Key++;

    return HashVal % TableSize;
}
```

Funções de Hash(Strings)

- hash(“Ana”,10007)= ASCII(‘A')+ASCII('n')+ASCI('a')=65+110+97=272%10007=272

Problema:

- Como ASCII(c)≤127, se as strings tiverem no máximo 8 caracteres o hash(s)≤127*8=1016.
- Usando uma tabela com um tamanho muito superior a este valor não obtemos uma boa distribuição.
- Porquê?????

Funções de Hash(Strings)

- O tamanho da tabela dependerá da quantidade de informação que pretendemos armazenar, e não da função de hash usada para os acessos

- Solução?

  - Atribuir um peso a cada caracter da String :

$$hash(Key) = \sum_i p^i Key[i]$$

  - Por exemplo usando peso=27 (letras do alfabeto) e tomando só os três primeiros caracteres

```
Index HashSt2( const char *Key, int TableSize ){
    return ( Key[0] + 27 * Key[1] + 729 * Key[2] )% TableSize;
}
```

Hashcode no Java

- Existem $26^3 = 17576$ possíveis palavras com 3 letras, mas num dicionário Inglês só 2851 correspondem a palavras num dicionário em Inglês. Mesmo que não existissem colisões, só uma pequena % da tabela seria acedida($2851/10007 \simeq 28\,\%$) e portanto para valores muito maiores que 2851, esta função não serve….

- Uma função que envolva todos os caracteres da String, e um peso apropriado(fácil de calcular) e alguma matemática resolvem o assunto:

$$hashcode(Key) = Key[0].32^{N-1} + Key[1].32^{N-2} + \ldots + Key[N-1]$$

  - E usando a regra de Horner que diz que outra forma de calcular $h_k = k_1 + 27k_2 + 27^2k_3$ pode ser calculado por $h_k = (k_3 * 27 + k2) * 27 + k_1$

Função de hash para Strings

- Como multiplicar por 32, é equivalente a shiftar 5 bits para a esquerda, usamos o 32 e não o 27…

```
Index HashSt3( const char *Key, int TableSize ){
    unsigned int HashVal = 0;
    while( *Key != '\0' )
        HashVal = ( HashVal << 5 ) + *Key++;

    return HashVal % TableSize;
}
```

Colisões

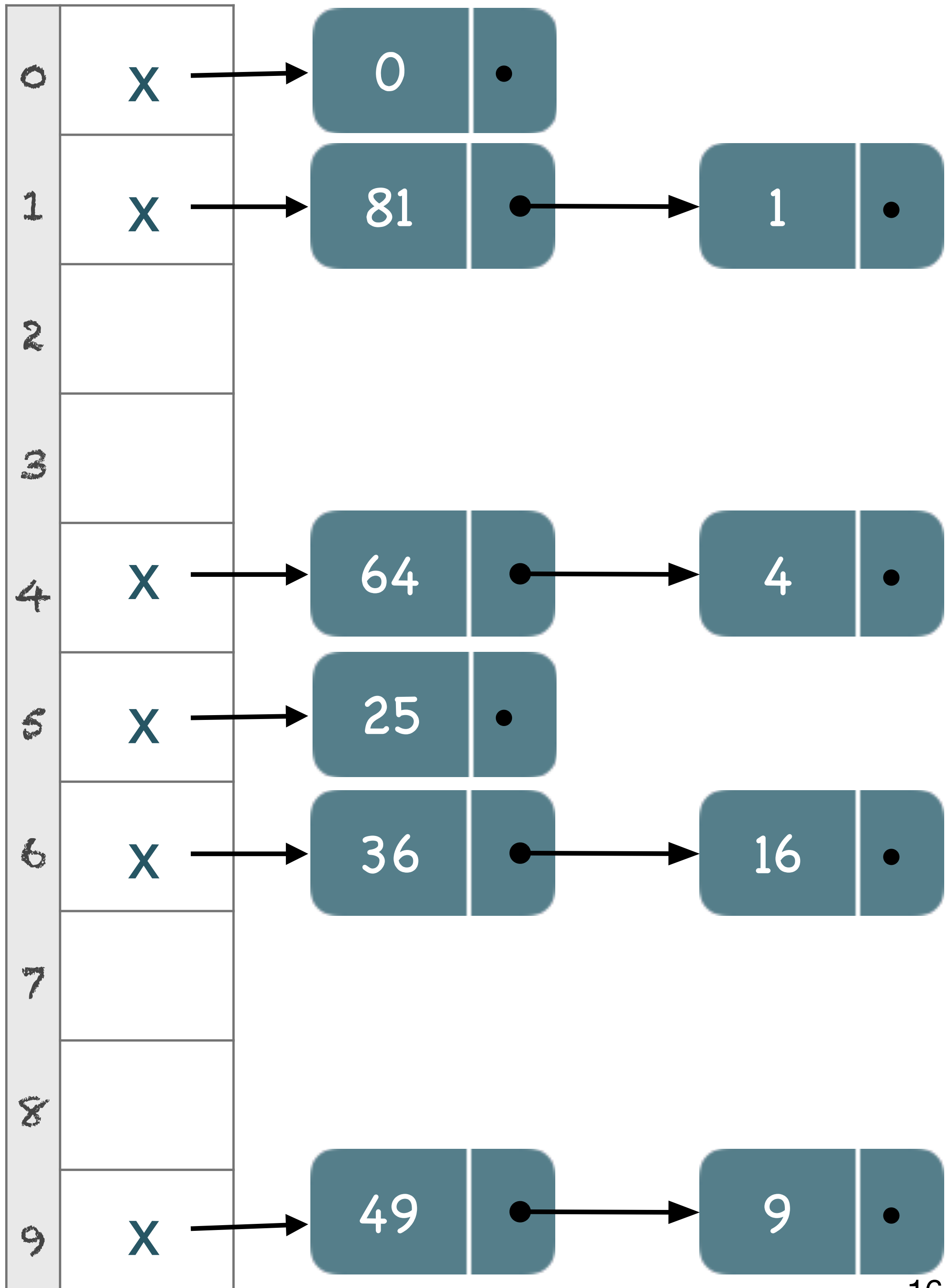
- Definição: duas chaves distintas resultam no mesmo índice
- Várias abordagens são possíveis, sendo que as mais básicas são:
  - Hashing Aberto(“Cadeias separadas”)
  - Hashing fechado(“Endereçamento aberto”)

Hashing aberto
(cadeias separadas)

- A pesquisa/inserção/remoção fazem-se:
  - Aplicando a função de hash às chaves
  - A operação a realizar (pesquisa/inserção/remoção) é feita numa cadeia externa: lista, árvore, array, tabela de hash, etc.
  - Convém que as cadeias sejam curtas

Exemplo Hashing aberto

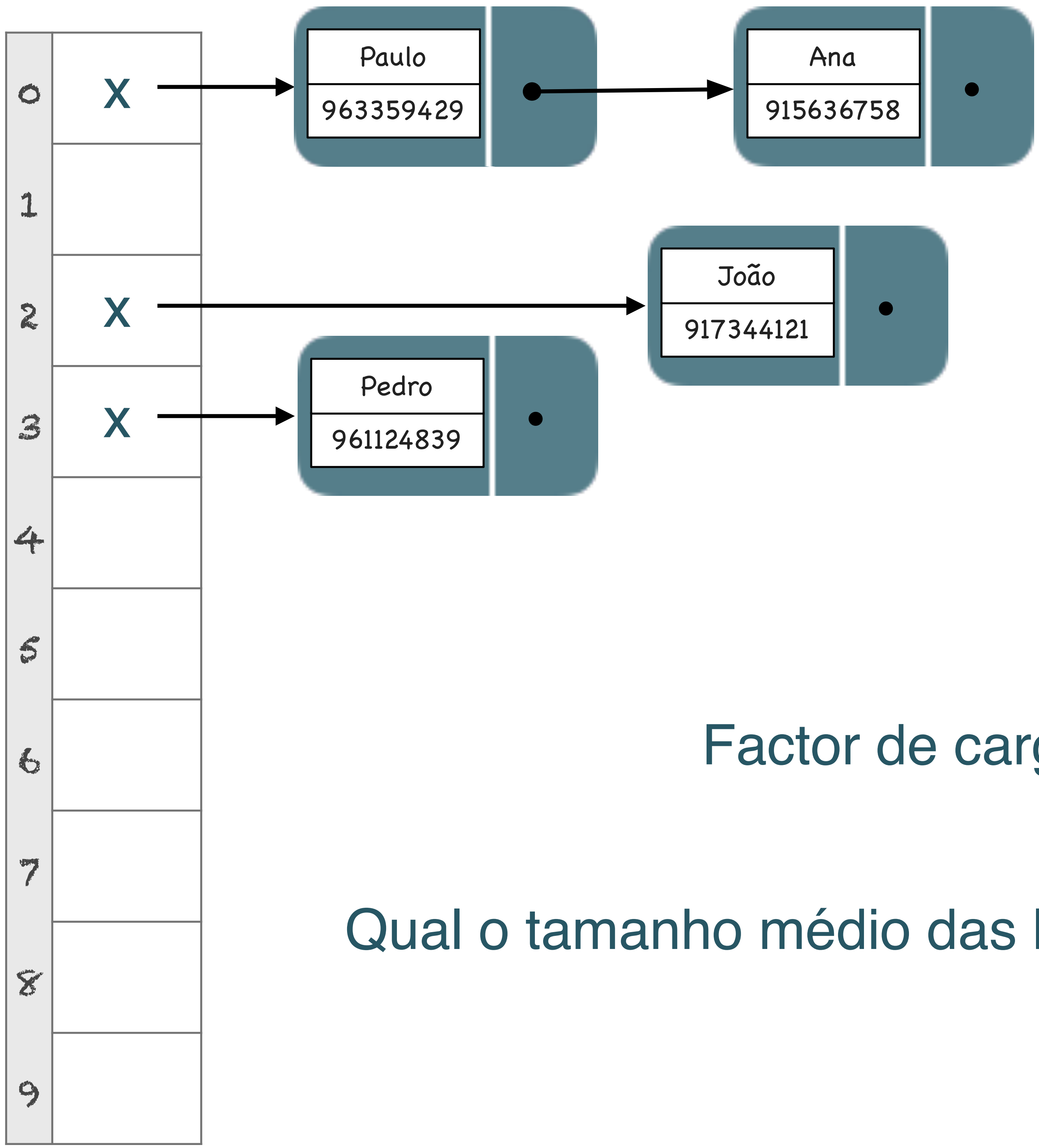
- Queremos inserir numa tabela de hash, os 10 primeiros quadrados perfeitos
  - 0,1,4,9,16,25,36,49,64,81, numa tabela de tamanho 10, usando como função de hash, hash(x)=x % 10
  - Usem-se listas ligadas para as cadeias externas, e insira-se à cabeça nas listas, porque…



Hash Aberto

- Define-se factor de carga λ como sendo
  - λ = N/M , para
    - N = nº de chaves inseridas
    - M = tamanho do vector (deve ser primo)

Exemplo:Encadeamento



hash(915636758)=0

hash(961124839)=3

hash(963359429)=0

hash(917344121)=2

Factor de carga? $\lambda = 4/10 = 0.4$

Qual o tamanho médio das listas? λ

Pesquisa/Inserção/Remoção

- O tamanho (médio) das listas é λ;
- Uma Pesquisa/Inserção/Remoção requer:
  - Cálculo do hashing da chave (O(1));
  - Pesquisar/Inserir/Remover na correspondente lista o objecto pretendido:
    - Pesquisa: $O(\lambda)$;
    - Inserção: cabeça O(1),cauda $O(\lambda)$
    - Remoção: $O(\lambda)$

Hashing Aberto

- Resumindo:
  - Importante é o factor de carga, não o tamanho da tabela, assumindo uma “boa” distribuição pela função de hash!
    - Se $\lambda<1$, as operações são $O(1)$
- Desvantagens do hashing aberto:
  - Existência de uma estrutura extra para resolver as colisões, com consequências no espaço ocupado;

Hashing fechado (endereçamento aberto)

- Inserem-se na própria tabela, não em cadeias externas
- Como dispor chaves com mesmo valor de hash? Isto é: Como se resolvem as colisões?
  - Tentam-se os índices: $h_0(Key)$, $h_1(Key)$, $h_2(Key)$... até que algum destes índices esteja livre
  - $h_i(Key) = hash(Key) + f(i)$, com $f(0) = 0$
    - para garantir que a 1ª tentativa só depende da função de hash

Hashing Fechado

- No endereçamento aberto os acessos são calculados(endereçamento por cálculo) através da fórmula $h_i(Key) = hash(Key) + f(i)$

- O acesso diz-se linear se f é linear

  - Usualmente $f(i) = i$

- Acesso quadrático se f é quadrática

  - Usualmente $f(i) = i^2$

- Duplo hashing, f é uma função de hash

  - Usualmente $f(i) = hash_2(i)$

Exemplo

- Inserir 66;47;87;90;126;140;145;153;177;285;393;395;467;566; 620;735; numa tabela de hash com 20 posições, usando:
  - Como função de hash:
    - $h(x) = x \ mod \ 20$
    - como resolvedor de colisões:
    - o acesso linear e $f(i) = i$

h0(66)=	hash(66) = 66 % 20 = 6	😀
h0(47)=	hash(47) = 47 % 20 = 7	😀
h0(87)=	hash(87) = 87 % 20 = 7	😡
h1(87)=	hash(87) + f(1) = 7 + 1 =8	😀
h0(90)=	hash(90) = 90 % 20 = 10	😀
h0(126)=	hash(126)= 126 % 20 = 6	😡
h1(126)=	hash(126)+ f(1) = 6 + 1 = 7	😡
h2(126)=	hash(126)+ f(2) = 6 + 2 = 8	😡
h3(126)=	hash(126)+ f(3) = 6 + 3 = 9	😀
h0(140)=	hash(140) = 140 % 20 = 0	😀
h0(145)=	hash(145) = 145 % 20 = 5	😀
h0(153)=	hash(153) = 153 % 20 = 13	😀
h0(177)=	hash(177) = 177 % 20 = 17	😀

0	140
1	
2	
3	
4	
5	145
6	66
7	47
8	87
9	126
10	90
11	
12	
13	153
14	
15	
16	
17	177
18	
19	

h0(285) = hash(285) = 285 % 20 = 5 😡

h1(285) = 5 + 1 = 6 😡

...

h6(285) = 5 + 6 = 11 😀

h0(393) = hash(393)= 393 % 20 = 13 😡

h1(393) = 13 + 1 = 14 😀

h0(395) = hash(395)= 395 % 20 = 15 😀

h0(467) = hash(467)= 467 % 20 = 7 😡

...

h5(467) = 7 + 5 = 12 😀

h0(566) = hash(566)= 467 % 20 = 6 😡

...

h10(566)= 6 + 10 = 16 😀

h0(620)= hash(620)= 620 % 20 = 0 😡

h1(620)= 0 + 1 = 1 😀

h0(735)= hash(735)= 675 % 20 = 15 😡

h3(735)= 15 + 3 = 18 😀

0	140
1	620
2	
3	
4	
5	145
6	66
7	47
8	87
9	126
10	90
11	285
12	467
13	153
14	393
15	395
16	566
17	177
18	735
19	

Acesso Linear

- À medida que a carga aumenta, vão-se formando "blocos" de células ocupadas: "Primary Clustering";
- É sempre possível (desde que exista espaço) encontrar local para inserir um elemento mas os acessos/tentativas necessárias vão aumentando.
    - A nova entrada vai por sua vez, aumentar o tamanho do cluster;
- Quando λ> 0.6 os acessos crescem muito, perde-se T=O(1)

Acesso Linear

- Como pesquisar um elemento na tabela?
  - Calcular o $hash(chave)$ e tentar as iteradas h_0, h_1 ,... até encontrar a chave ou a entrada estar vaga.
    - Exemplo:
      - procurar chave 130 na tabela do exemplo anterior
        - hash(130)=130 % 20=10

0	140
1	620
2	
3	
4	
5	145
6	66
7	47
8	87
9	126
10	90
11	285
12	467
13	153
14	393
15	395
16	566
17	177
18	735
19	

Não está

Acesso Linear

- Como remover?
  - Hipótese: procurar elemento e removê-lo directamente
  - Contra-Exemplo:
    - remover 467 (hash(467)=7)
    - procurar 566 (hash(566)=6)
- Solução:
  - "lazy deletion"

0	140
1	620
2	
3	
4	
5	145
6	66
7	47
8	87
9	126
10	90
11	285
12	467
13	153
14	393
15	395
16	566
17	177
18	735
19	

Não está

está,está !

Acesso Quadrático

- Inserir 65;76;47;87;77 numa tabela de hash com 11 posições, usando o acesso quadrático como resolvedor de colisões.

```
h0(65)=  hash(65)= 65 % 11 = 10
h0(76)=  hash(76)= 76 % 11 = 10
h1(76)=   hash(76)+f(1)=10+1² =
         = 11 % 11= 0
h0(47)=   h0(47)= hash(47) = 3
```

0	76
1	
2	
3	47
4	
5	
6	
7	
8	
9	
10	65

Acesso quadrático

λ=5/11

h0(87) = hash(87) = 10 😡

h1(87) = hash(87)+f(1)=10+1 = 11

11 % 11 = 0 😡

h2(87) = hash(87) + 2^2= 10 + 4 =

14 % 11 = 3 😡

h3(87) = 10 + 3^2= 19 % 11 = 8 😀

h0(77) = hash(77) = 11 % 11 = 0 😡

h1(77) = hash(77)+f(1)=11+1^2=

12 % 11 = 1 😀

0	76
1	77
2	
3	47
4	
5	
6	
7	
8	87
9	
10	65

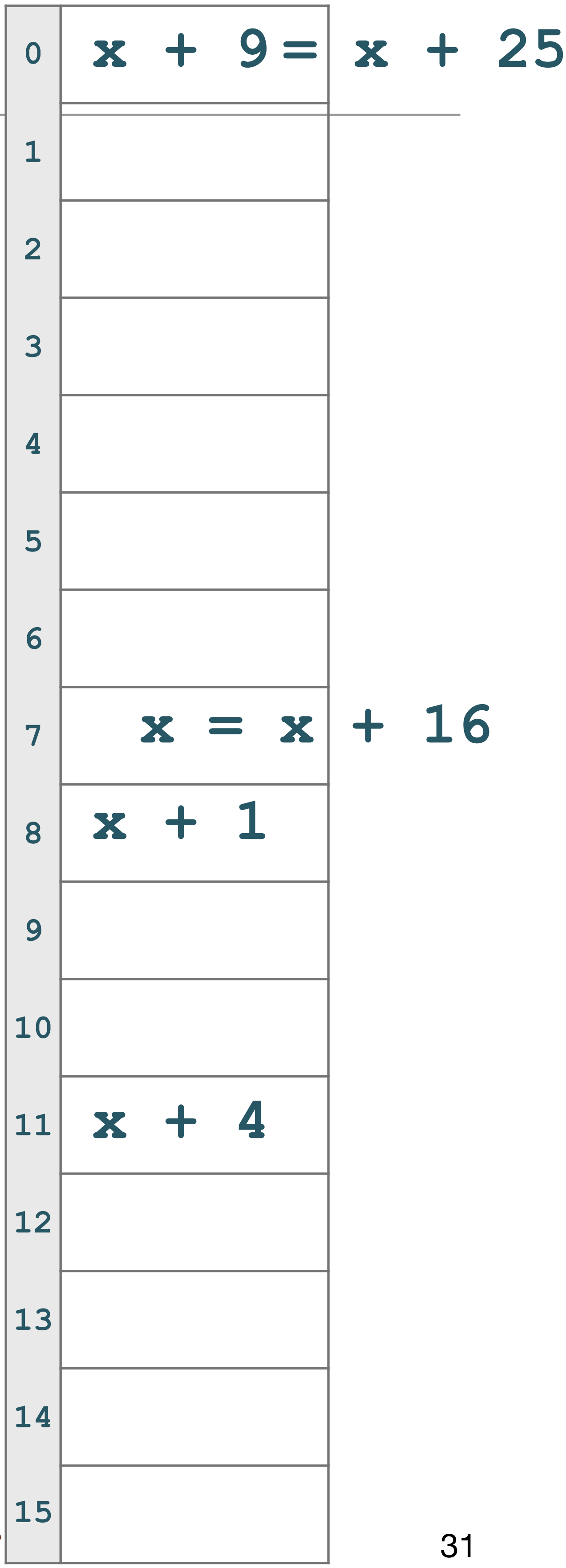
Acesso Quadrático

- Que garantias podem ser dadas, de que existindo entradas livres, este tipo de acesso encontra uma? (Diferente do linear!)

  - Se mais de metade da tabela estiver livre, e o tamanho da tabela for um número primo, é possível demonstrar que o acesso quadrático "encontra" uma célula livre para inserir;

- Exemplo: Numa tabela tamanho 16 (não é primo!), as alternativas estão às distâncias 1, 4 e 9. Estando preenchidas, as iteradas hi(chave) não produzem outras alternativas!

$1^2 = 1, 2^2=4, 3^2=9, 4^2=16, 5^2=25, 6^2=36, 7^2=49, 8^2=64,$ …



Acesso Quadrático

- Apesar de eliminar o primary clustering, o acesso quadrático gera outro tipo de clustering: “secondary clustering”: Todos os elementos que “hasham” no mesmo sítio, acessam as mesmas alternativas.

Duplo hashing

- Usa-se uma segunda função de hash para os acessos. Geralmente
  - $f(i) = i \, . \, hash_2(x)$
  - Os acessos ficam agora às distâncias:
    - $hash_2(x)$, $2.hash_2(x)$, $3.hash_2(x)$, ….
- Resolve o secondary clustering...
- Usual tomar $hash_2(x) = R - (x \; mod \; R)$ para R primo menor que o tamanho da tabela

DUPLO HASHING

- Numa tabela de tamanho 10, inserir 89,18, 49, 58, 69, 60, 23, sendo o acesso de duplo hashing (hash2(x)= 7 - (x mod 7)) e usando como 1ª função de hash, hash(x)= x mod 10.

h0(89)= hash(89)=9 😀

h0(18)= hash(18)=8 😀

h0(49)= hash(49)=9 😡

h1(49)= hash(49)+ 1.hash2(49)

h1(49)= 9+1.(7-(49 %7))= 16 =6 😀

0	
1	
2	
3	
4	
5	
6	49
7	
8	18
9	89

Duplo hashing

```
h0(58)=  hash(58)=8                              😡
h1(58)=  hash(58)+ 1.hash2(58)
h1(58)=  8+1.(7-(58 % 7))
h1(58)=  8 + 1.(7-2)= 3                          😀
h0(69)=  hash(69)=9                              😡
h1(69)=  hash(69)+ 1.hash2(69)
h1(69)=  9+1.(7-(69 % 7))=9 + 1.(7-6)
h1(69)=  10 % 10 = 0                             😀
```

0	69
1	
2	
3	58
4	
5	
6	49
7	
8	18
9	89

Duplo Hashing

```
h0(60)= hash(60)=0
h1(60)= hash(60)+ 1.hash2(60)
h1(60)= 0+1.(7-(60 % 7))=0 + 1.(3)= 3
h2(60)= hash(60)+ 2.hash2(60)
h2(60)= 0 + 2.(3)= 6
h3(60)= hash(60)+ 3.hash2(60)
h3(60)= 0 + 3.(3) = 9
h4(60)= hash(60)+ 4.hash2(60)
h4(60)= 0 + 4.(3)=12 = 2
```

0	69
1	
2	60
3	58
4	
5	
6	49
7	
8	18
9	89

DUPLO HASHING

```
h0(23)=  hash(23)=3                               😡
h1(23)=  hash(23)+ 1.hash2(23)
h1(23)=  3+1.(7-(23 % 7))=3 + 1.(5)= 8            😡
h2(23)=  hash(23)+ 2.hash2(23)
h2(23)=  3 + 2.(5)=13 =3                          😡
h3(23)=  hash(23)+ 3.hash2(23)
h3(23)=  3 + 3.(5)=18 =8                          😡
h4(23)=  hash(23)+ 4.hash2(23)
h4(23)=  3 + 4.(5)=23 = 3                         😡
```

0	69
1	
2	60
3	58
4	
5	
6	49
7	
8	18
9	89

Estou a repetir!

Porquê?

Rehashing

- Como os casos anteriores, só que:
  - se λ ultrapassar um determinado valor (por exemplo 0,5), "duplica-se" o tamanho da tabela
  - Procurar manter tamanho da tabela primo
  - Inserção tem T=O(1) amortizado
    - Ocasionalmente será O(N)
  - Mantém λ baixo, portanto poucos acessos

Rehasing: Exemplo

- Acesso quadrático inseriu-se 65;76;47;87;77; $\lambda=5/11 = 0.45$
- queremos ainda inserir 88, $\lambda=6/11 > 0.5$
- Então faça-se rehash
    - Menor primo >2*11=23, será o tamanho da próxima tabela
    - Temos que calcular os novos endereços de todos os elementos da tabela antiga na nova tabela, já que não serão iguais

0	76
1	77
2	
3	47
4	
5	
6	
7	
8	87
9	
10	65

Rehashing: Exemplo

0	76
1	77
2	
3	47
4	
5	
6	
7	
8	87
9	
10	65

$h0(76) = \quad hash(76) = 76 \ \% \ 23 = 7$

$h0(77) = \quad hash(77) = 77 \ \% \ 23 = 8$

$h0(47) = \quad hash(47) = 47 \ \% \ 23 = 1$

$h0(87) = \quad hash(87) = 87 \ \% \ 23 = 18$

$h0(65) = \quad hash(65) = 65 \ \% \ 23 = 19$

$h0(88) = \quad hash(88) = 88 \ \% \ 23 = 19$

$h1(88) = \quad 19 + 1^2 = 20$

0	
1	47
2	
3	
4	
5	
6	
7	76
8	77
9	
10	
11	
12	
13	
14	
15	
16	
17	
18	87
19	65
20	88
21	
22	

Exercício

- Desenhe numa tabela de tamanho (N=11), os resultados de inserir as chaves do array abaixo, usado para função de hash, hash1(x)=(d_0+d_n)%N, para do e dn os dígitos menos e mais significativos de x, respectivamente, e assumindo que as colisões são resolvidas usando:

  A. Hashing fechado de acesso quadrático

  B. Hashing fechado com duplo hashing usando como segunda função de hash, hash2(x)=3-(x%3)

5	1	18	30	19	7	39	0	8	2	3