Algoritmos de pesquisa

Tabelas de dispersão/Hash

Tabelas de dispersão/Hash

Introdução

➔ **Motivação:**

Considerar o problema de pesquisar um determinado valor num vetor (*array*).

- Se o *vetor* não está ordenado, a pesquisa requer O(n) de complexidade.
- Se o *vetor* está ordenado, pode-se fazer a pesquisa binária que requer uma complexidade de O(log n).
- Não parece haver melhor maneira de resolver o problema com custos melhorados.

Tabelas de dispersão/Hash

Introdução

➔ Ideia:

Poderia haver maneira de resolver o problema em O(1).

- Se o *vetor* estiver organizado de uma determinada maneira.

➔ Solução:

Arranjar uma *função mágica* que, dado um determinado valor a pesquisar, nos diga exatamente a posição exata no vetor.

➔ Esta função é chamada ***função de dispersão/hash***.

Tabelas de dispersão/Hash

Introdução

➔ Complexidade ideal – O(1)

- *Vetor* A[i] de 0 a 65.535 (16 bits) inicializado a zeros
- Cada inserção de um número k seria: A[k]++
- Cada remoção de um número k seria: A[k]--
- A[k] representa o número de vezes que o número k foi inserido
- Para procurar temos que fazer *find(k)*; se A[k] > 0 está encontrado

Tabelas de dispersão/Hash

Introdução

➔ Problemas encontrados

- Tamanho do vetor

  Se o inteiro for 16 bits dá um vetor com 65.536, mas se tivermos um com 32 bits dá 4.294.967.296, o que é incomportável.

- O uso de *strings*

  Se tivermos *strings* em vez de inteiros não dá para indexar num vetor, porque A[“Ana”] não existe.

Tabelas de dispersão/Hash

Introdução

➔ Resolução do problema dos *vetores* grandes

- Usar uma função que mapeie grandes números ou *strings* transformados em números mais pequenos ou manejáveis
- Esta função é chamada de *Hash* ou de *dispersão*

Tabelas de dispersão/Hash

Tabela de Hash

- As *tabelas de hash* são um tipo de estruturação criado para o armazenamento de informação, e são
  - uma forma extremamente simples,
  - fácil de se implementar e,
  - intuitiva para se organizar grandes quantidades de dados.
- Possui como ideia central a divisão do universo de dados a ser organizado em subconjuntos mais facilmente geríveis.
- A estruturação da informação em *tabelas de hash* visa principalmente permitir armazenar e procurar rapidamente grande quantidade de dados.

Tabelas de dispersão/Hash

Tabela de Hash

➔ As tabelas de *hash* são constituídas por 2 conceitos fundamentais:

- Tabela de *Hash*: estrutura que permite o acesso aos subconjuntos.
- Função de *Hash*: função que realiza um mapeamento entre os valores de chaves e as entradas na tabela.

Tabelas de dispersão/Hash

Tabela de Hash

➔ Criar um critério simples para dividir este universo em subconjuntos com base nalguma qualidade do domínio das chaves

- Possuir um índice que permita encontrar o início do subconjunto certo, depois de calcular o valor de hash.
- Isto é a ***tabela de hash***.

➔ Saber em qual subconjunto procurar e colocar uma chave

- indicar quantos subconjuntos se pretende
- criar uma regra de cálculo que, dada uma chave, determine em que subconjunto se deve procurar pelos dados com esta chave ou colocar estes dados (caso seja um novo elemento).
- Isto é chamado de ***função de hash***.

Tabelas de dispersão/Hash

Tabela de Hash

➔ Gerir estes subconjuntos bem menores com um método simples:

- Possuir uma estrutura ou um conjunto de estruturas de dados para os subconjuntos.
- Existem duas filosofias: *hashing* fechado (ou de endereçamento aberto) ou o *hashing* aberto (ou encadeado).

➔ Alguns dos problemas que se colocam quando usamos tabelas de *hash* são:

- determinar uma função de *hash* que minimize o número de colisões;
- obter os mecanismos eficientes para tratar as colisões.

Tabelas de dispersão/Hash

Funções de *Hash* - exemplos

➔ Números ***x*** (chave) de 0 a 99 (dois dígitos)

➔ ***tam*** tamanho da tabela

➔ Pode-se construir uma função que coloque ***x*** no *vetor* em termos do algarismo das dezenas.

f = x / tam (/ = divisão inteira)

➔ Ou pode-se construir uma função que coloque ***x*** no *vetor* em termos do algarismo das unidades.

f = x % tam (% = resto da divisão)

Tabelas de dispersão/Hash

Função de *Hash*

➔ Exemplo de uma *função de Hash:*

```
int  hash (int  key, int  tam) {
        return (key % tam);
}
```

➔ Objetivos das *funções de Hash:*

- deve ser eficiente.
- deve distribuir todos os elementos uniformemente por todas as posições da tabela.

Tabelas de dispersão/Hash

Função de *Hash* para *strings* - exemplos

➔ Método normal:

```
int  hash (char  key[], int  tam) {
        int  valHash = 0, k = 0;
        while (key[k] != '\0') {
                valHash = valHash + int(key[k]);
                k++;
        }
        return (valHash % tam);
}
```

Tabelas de dispersão/Hash

Função de *Hash* para *strings*

➔ Usando a regra de Horner:

```
int  hash (char  key[], int  tam) {
        int  valHash = 0, a = 127, k = 0;
        while (key[k] != '\0') {
                valHash = valHash * a + int(key[k]);
                k++;
        }
        return (valHash % tam);
}
```

Tabelas de dispersão/Hash

Colisões

- Quando a função de *Hash* é imperfeita poderão existir dois valores a serem colocados na mesma posição do *vetor*. A isto chama-se uma **colisão**.
- As colisões são normalmente tratadas como quem chega primeiro serve-se; isto é, o primeiro elemento a chegar a uma posição fica com ela.
- Terá de se arranjar uma solução eficiente para se determinar o que se deve fazer com o segundo valor que deveria ser colocado na mesma posição.

Tabelas de dispersão/Hash

Colisões

➔ O que fazer quando dois valores diferentes tentam ocupar a mesma posição?

- Solução 1 (**hashing fechado** ou **endereçamento aberto**): procura a partir dessa posição uma posição vazia.
- Solução 2: usar uma segunda função de *hash*, uma terceira, uma quarta, …
- Solução 3 (**hashing aberto** ou **encadeamento separado**): usa a posição do vetor como ***cabeça*** (head) de uma lista que vai conter todas as colisões dessa posição.

Tabelas de dispersão/Hash

Factor de ocupação da tabela λ

➔ n = número de elementos da tabela.

➔ dim = dimensão da tabela.

➔ λ = n / dim

➔ λ deve ser inferior a 1.

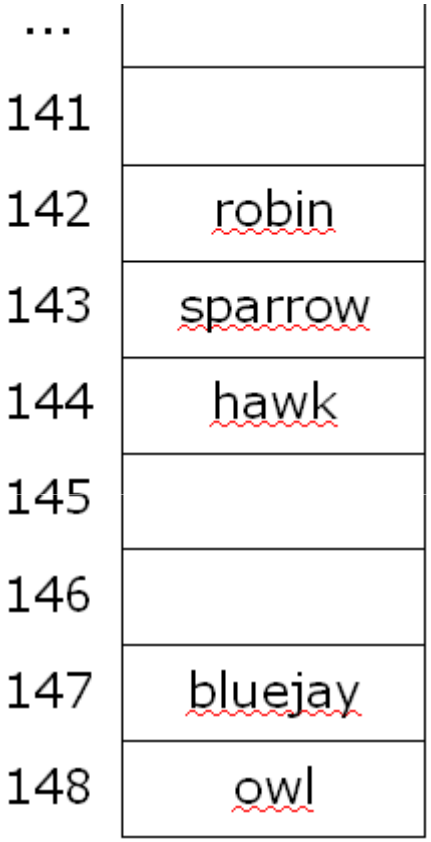
Tabelas de dispersão/Hash

Hashing fechado

➔ Suponha que quer adicionar *seagull* à tabela de Hash.

➔ Suponha também que:

- hash(seagull) = 143
- table[143] está ocupada
- table[143] != *seagull*
- table[143+1] está ocupada
- table[143+1] != *seagull*
- table[143+2] está vazia



Tabelas de dispersão/Hash

Hashing fechado

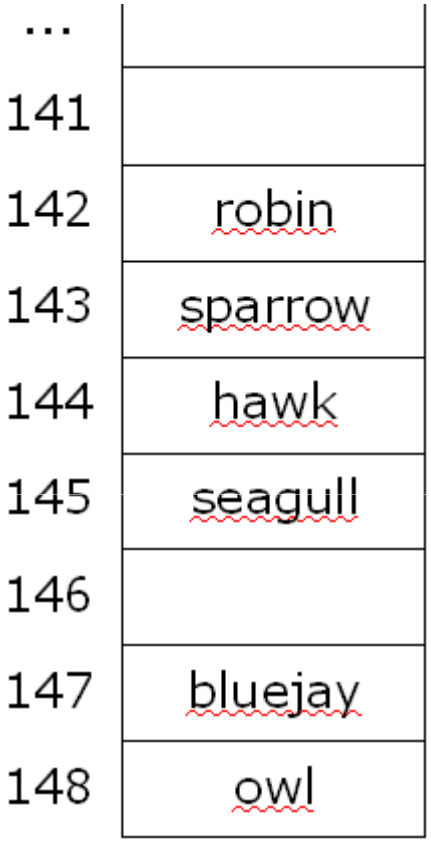
➔ Suponha que quer adicionar *seagull* à tabela de *Hash*.

➔ Suponha também que:

- hash(seagull) = 143
- table[143] está ocupada
- table[143] != *seagull*
- table[143+1] está ocupada
- table[143+1] != *seagull*
- table[143+2] está vazia

➔ Então, colocar *seagull* em 145.

➔ Foi utilizada a **sondagem linear.**



Tabelas de dispersão/Hash

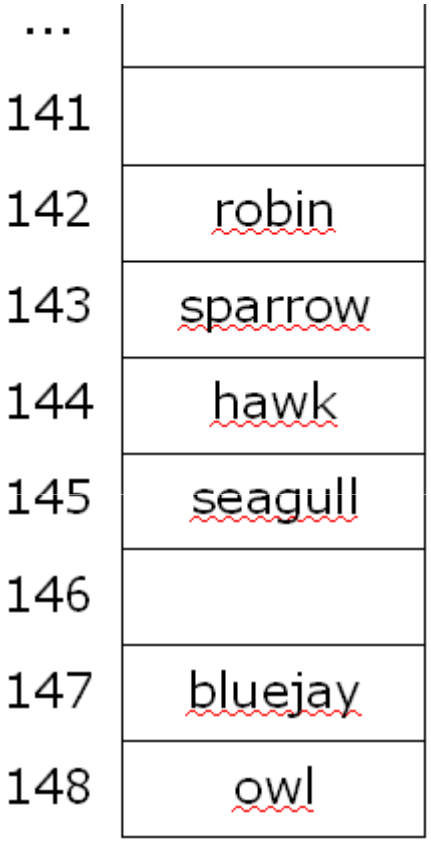
Hashing fechado

➔ Suponha que quer adicionar hawk à tabela de Hash.

➔ Suponha também que:

- hash(hawk) = 143
- table[143] está ocupada
- table[143] != hawk
- table[143+1] está ocupada
- table[143+1] == hawk
- O hawk já está na tabela.

➔ Então, não fazer nada.



Tabelas de dispersão/Hash

Hashing fechado

➔ Suponha que quer adicionar *cardinal* à tabela de *Hash*.

➔ Suponha também que:

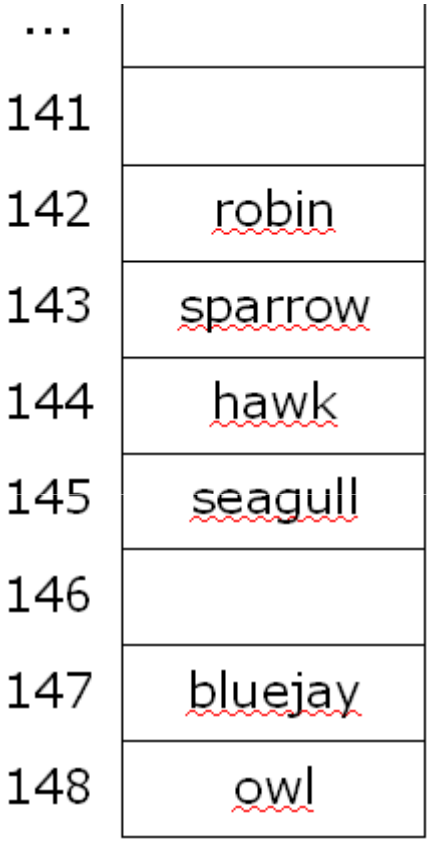
- hash(*cardinal*) = 147
- a última posição é 148
- table[147] está ocupada
- table[148] está ocupada

➔ Solução

- Tratar a tabela como circular:

  a seguir a 148 vem o 0.

- Assim, *cardinal* vai para a posição 0.



Tabelas de dispersão/Hash

Hashing fechado

➔ Problemas da sondagem linear

- Existência de blocos contíguos de posições ocupadas com grandes dimensões (em geral quando $\lambda > 0,5$) — **Clusters**.
- Quanto maiores os blocos ficam, mais tendência têm para crescer.
- A sondagem é igual para elementos que colidem.

Tabelas de dispersão/Hash

Hashing fechado

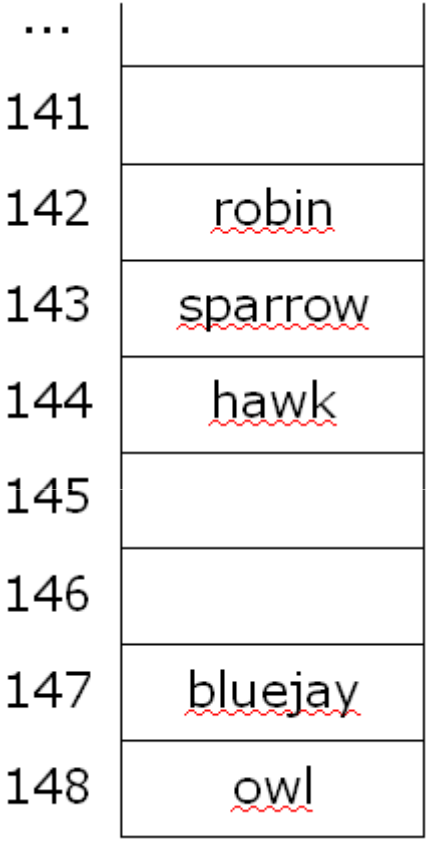
➔ Suponha que quer adicionar *seagull* à tabela de *Hash*.

➔ Suponha também que:

- hash(*seagull*) = 143
- table[143] não está vazia
- table[143] != *seagull*
- table[143+1^2] != *seagull*
- table[143+2^2] != *seagull*
- table[143+3^2] != está vazia

➔ Então, colocar *seagull* em 152.

➔ Foi utilizada a **sondagem quadrática**.



Tabelas de dispersão/Hash

Hashing fechado

➔ Problemas da sondagem quadrática

- Cada sondagem tenta uma nova posição. Se tivermos o tamanho do *vetor,* tam = 16, calha sempre nas posições 1, 2, 4, 9 entrando num ciclo infinito.

Tabelas de dispersão/Hash

Hashing fechado

➔ Resolução do problema

- ***tam*** tem de ser número primo
- O fator **λ** tem que ser >= 0,5
- Ao atingir λ = 0.5 pode-se aumentar a tabela para o próximo número primo, de seguida pega-se nos valores da tabela e passam-se para a nova aplicando de novo a função de *hash(x)*
- A este método chama-se o método de **rehash(x)**.

Tabelas de dispersão/Hash

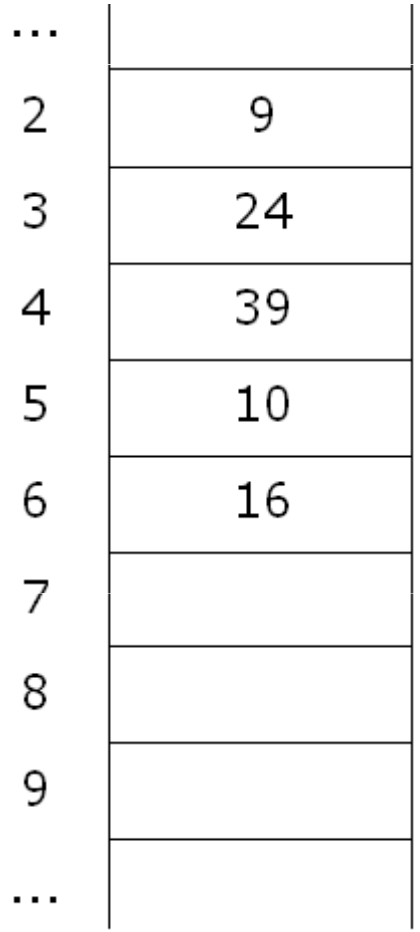
Hashing fechado

➔ Problema da remoção.

➔ Suponha também que:

- hash(9) = 2
- hash(24) = 3
- hash(39) = 4
- hash(10) = 3
- hash(16) = 2

➔ Remove(24)



Tabelas de dispersão/Hash

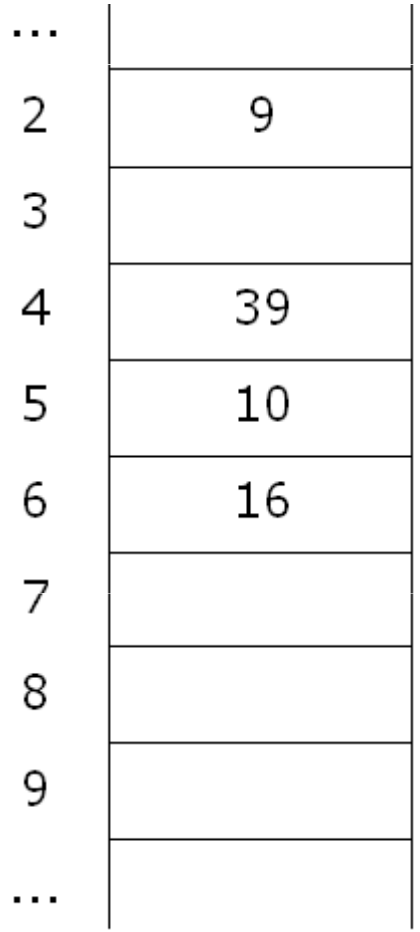
Hashing fechado

➔ Não existe nada na posição 3.

➔ hash(10) = 3.

➔ Logo, número 10 não existe no vetor.

O que é **falso**.



Tabelas de dispersão/Hash

Hashing fechado - resolução do problema

➔ Cria-se uma estrutura que assinale cada posição:

- Livre (vazia e nunca ocupada)
- Ocupada
- Removida (vazia mas já ocupada)

➔ Pesquisar/remover:

- Termina quando encontra o objeto
- Termina quando encontra uma posição livre
- Continua a pesquisa quando encontra uma posição removida

➔ Inserir (insere o objeto quando encontra):

- uma posição assinalada como livre ou removida.

Tabelas de dispersão/Hash

Hashing aberto

- A estruturação dos dados é talvez a forma mais intuitiva de se implementar o conceito de *Hashing*
- Consiste em ter um vetor de apontadores, com dimensão N, em que cada elemento do vetor contém uma ligação para uma lista dos elementos a guardar
- A pesquisa de um elemento efetua-se da seguinte forma:
  - A partir de uma chave, calcular qual o elemento do vetor é a cabeça da lista que se pretende
  - Usando um qualquer algoritmo para o efeito, pesquisar o elemento dentro daquela lista.

Tabelas de dispersão/Hash

Hashing aberto – Exemplo 1

➔ Armazenar os elementos: 19, 26, 33, 70, 79, 103 e 110.

➔ Usando a função de *hash*: **hash(x) = x % 10**

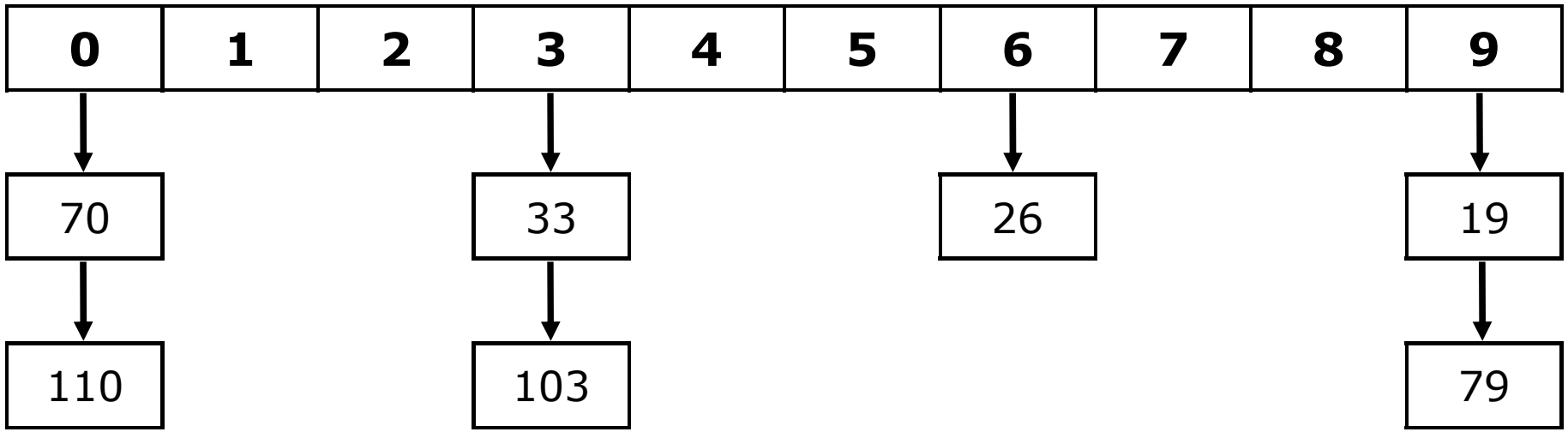
➔ Os valores das funções de *hash* são os seguintes:

- hash(19) = 9
- hash(26) = 6
- hash(33) = 3
- hash(70) = 0
- hash(79) = 9
- hash(103) = 3
- hash(110) = 0

Tabelas de dispersão/Hash

Hashing aberto – Exemplo 1

➔ A tabela de *hash* é a seguinte:



Tabelas de dispersão/Hash

Hashing aberto – Exemplo 2

- Cada elemento do vetor é um apontador para uma lista de estruturas com o mesmo valor da função de *hash*.
- Supondo que o vetor tem tamanho **tam** = 13 e que a função de *hash* é a que consta na tabela que se segue

Chave	A,B	C,D	E,F	G,H	I,J	K,L	M,N	O,P	Q,R	S,T	U,V	X,Y	W,Z
Hash	0	1	2	3	4	5	6	7	8	9	10	11	12

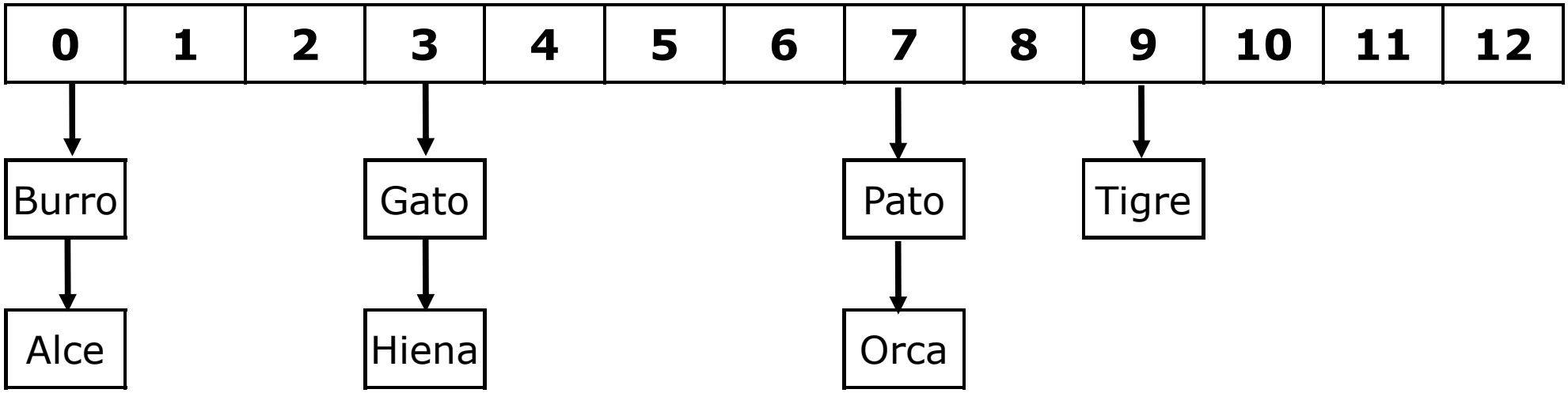
calcular os valores de *hash* para as seguintes chaves:

Alce, Burro, Gato, Hiena, Pato, Orca e Tigre.

Tabelas de dispersão/Hash

Hashing aberto – Exemplo 2

➔ A tabela de *hash* é a seguinte:



Tabelas de dispersão/Hash

Hashing aberto - propriedades

- ➔ Reduz o número de comparações, quando comparado com a pesquisa sequencial
- ➔ Necessidade de espaço em memória para armazenar o vetor de listas

Tabelas de dispersão/Hash

Vantagens e desvantagens

➔ Dispersão aberta (DA) vs. Fechada (DF)

- Suporta a primitiva de remoção
- As listas de colisão não se cruzam
- O erro por defeito do pré-dimensionamento não é tão grave

➔ Dispersão fechada vs. aberta

- Se a tabela estiver em ficheiro poupam-se muitos acessos com a sondagem linear, porque em geral os elementos que colidem estão fisicamente próximos.

Tabelas de dispersão/Hash

Vantagens e desvantagens

➔ Vantagens da Dispersão

- A complexidade das primitivas suportadas (pesquisa, inserção e remoção na DA), no caso esperado é constante.
- A técnica é eficiente e é só depende do fator de ocupação e da qualidade das funções (na DF).

➔ Problemas da Dispersão

- Não é uma estrutura dinâmica (o redimensionamento é necessário)
- Não suporta primitivas que se baseiam em relações de ordem dos elementos (mínimo, máximo, percurso ordenado).
- A complexidade das primitivas suportadas (pesquisa, inserção e remoção na DA), no pior caso é linear no nº de elementos da tabela.