Estruturas de Dados II

Espalhamento (ou hashing)

Seja P o conjunto que contém os elementos de um determinado universo. Chamamos espalhamento (ou hashing) o particionamento de P em um número finito de classes P_1, P_2, P_3, ..., P_n, n>1.

A correspondência unívoca entre os elementos do conjunto P e as *n* classes sugere a existência de uma função ***h***, através da qual é feito o particionamento.

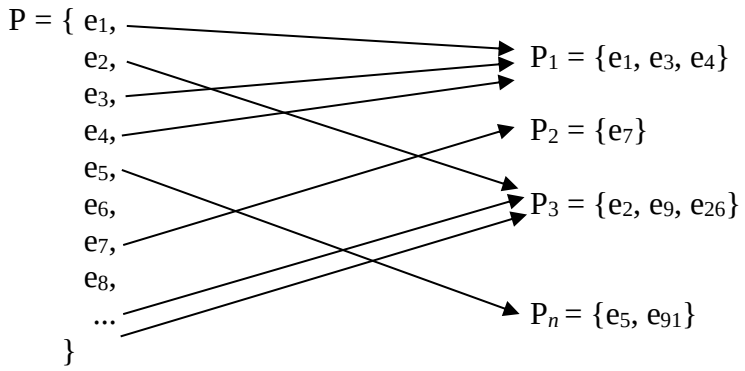


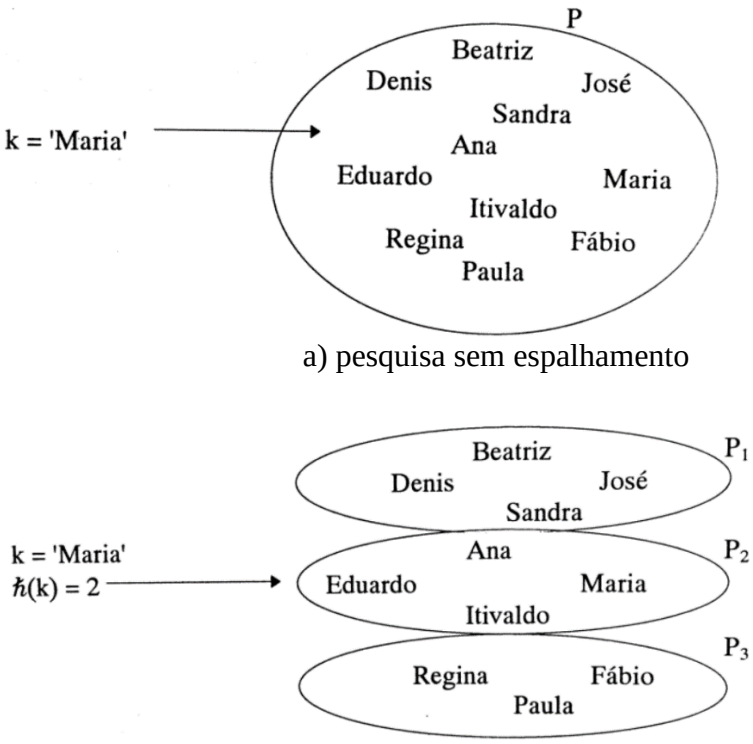
Figura 1. O espalhamento realizado pela função ***h***.

A função ***h***: P→[1..*n*], que leva cada elemento de P à sua respectiva classe, é chamada função de espalhamento ou **função de hashing**. Sendo *k* um elemento qualquer do conjunto P, o seu valor de hashing, ***h*(*k*)**, determina que classe ele pertence. Pode-se garantir que:

$$k \in P_{h(k)}$$

Aplicabilidade do Espalhamento

Em termos práticos, a grande vantagem do espalhamento está no fato de que, dado um elemento *k* de um conjunto P, o valor de hashing ***h*(*k*)** pode ser calculado em tempo constante, fornecendo imediatamente a classe de partição de P em que o elemento se encontra. Se considerarmos P uma coleção de elementos a ser pesquisada, é fácil perceber que o processo será muito mais eficiente se a pesquisa for restrita a uma pequena parte do conjunto P (uma única classe). Suponha o caso de procurar o nome ‘Maria’ em um conjunto de nomes próprios:



a) pesquisa sem espalhamento

b) pesquisa com espalhamento

Figura 2. Redução do espaço de busca.

O espalhamento pode ser usado como uma técnica de redução do espaço de busca; o processo de pesquisa será tão mais eficiente quanto menores forem as partições. Se o espalhamento for perfeito, então o valor de hashing calculado dará imediatamente a localização do elemento desejado; neste caso, temos um **acesso direto** ou randômico, como também é chamado, sendo esse o tipo de busca mais eficiente que dispomos.

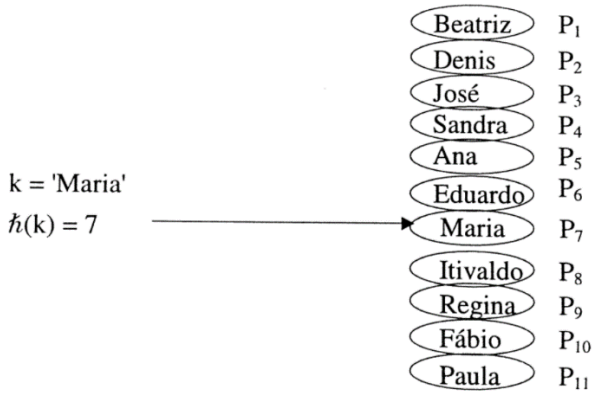


Figura 3. Acesso direto ou randômico.

Tabelas de Espalhamento

Tabela de espalhamento é a estrutura de dados que implementa o espalhamento para aplicações em computador. Ela pode ser representada por um vetor onde cada posição, denominada **encaixe**, mantém uma classe de partição. O número de encaixes na tabela deve coincidir com o número de classes criadas pela função de espalhamento. Considerando que um conjunto P seja espalhado em P_1, P_2, P_3, ..., P_n classes distintas, um

vetor T[1..*n*] pode representar o espalhamento de maneira satisfatória, bastando associar a cada elemento T[*i*] uma classe P_i, $1 \leq i \leq n$ correspondente.

Admitindo a existência de elementos sinônimos em P, é de se esperar que durante o espalhamento ocorra pelo menos uma colisão, ou seja, é possível que tenhamos que armazenar um elemento numa posição da tabela que já encontre ocupada por um outro valor. Uma forma de resolver este problema é chamada de **espalhamento externo**.

O espalhamento externo parte do princípio de que existirão muitas colisões durante o carreamento das chaves numa tabela e que, na verdade, cada encaixe T[*i*] armazenará não um único elemento, mas uma coleção de elementos sinônimos. Como a quantidade de elementos sinônimos em cada classe que pode variar bastante, a lista encadeada será útil. No espalhamento externo, a tabela de hashing será um vetor cujos elementos são ponteiros para listas encadeadas que representam as classes do espalhamento.

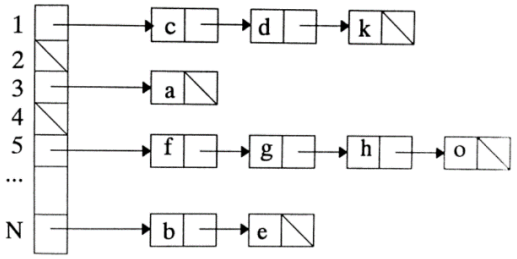


Figura 4. Uma tabela de espalhamento ou hashing.

Funções de Espalhamento

Espalhamento pode ser visto como um método de busca que permite acessar dados diretamente através de uma função que transforma uma chave *k* em um endereço físico, relativo ou absoluto, ***h*(*k*)**.

Uma função de espalhamento ideal seria aquela capaz de mapear *n* chaves em exatamente *n* endereços, sem a ocorrência de colisões. Considerando-se que existem n^n formas possíveis de atribuir *n* chaves a *n* endereços, a probabilidade de se obter o espalhamento perfeito é mínima ($n!/n^n$). Na prática, devemos nos contentar com funções capazes de fazer um mapeamento razoável, distribuindo as *n* chaves de maneira mais ou menos uniforme entre os endereços.

O Método da Divisão Inteira

O método da divisão consiste basicamente em realizar uma divisão inteira e tomar o seu resto. Exemplo: vamos espalhar as chaves 54, 21, 15, 46, 7, 33, 78, 9, 14, 62, 95 e 87 numa tabela contendo N=5 encaixes. A função definida a seguir, que transforma as chaves em endereços relativos, está baseada no método da divisão:

```
function dh(chv: integer): integer;
begin
  dh:= (chv mod N) + 1;
end;
```

A seguir, os valores de hashing gerados pela função:

- dh(54) = (54 mod 5) + 1 = 5
- dh(21) = (21 mod 5) + 1 = 2
- dh(15) = (15 mod 5) + 1 = 1
- dh(46) = (46 mod 5) + 1 = 2
- dh(7) = (7 mod 5) + 1 = 3
- dh(33) = (33 mod 5) + 1 = 4
- dh(78) = (78 mod 5) + 1 = 4
- dh(9) = (9 mod 5) + 1 = 5
- dh(14) = (14 mod 5) + 1 = 5
- dh(62) = (62 mod 5) + 1 = 3
- dh(95) = (59 mod 5) + 1 = 1
- dh(87) = (87 mod 5) + 1 = 3

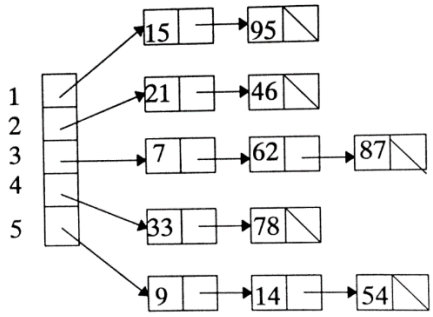


Figura 5. Espalhamento de chaves numéricas.

Transformações de Chaves Alfanuméricas

Uma forma simples de se transformar uma chave alfanumérica num valor numérico consiste em considerar cada caractere da chave com um valor inteiro (correspondente ao seu código ASCII) e realizar uma soma com todos eles:

```
function adh(chv: string): integer;
var i, soma: integer;
begin
  soma:=0;
  for i:=1 to length(chv) do
    soma:= soma + ord(chv[i]);
  adh:= (soma mod N) + 1;
end;
```

Vamos supor que pretendemos espalhar em uma tabela com N=7 encaixes as chaves Thais, Edu, Bia, Neusa, Lucy, Rose, Yara, Decio e Sueli. Calculando a função adh() para cada uma das chaves, obtemos os valores a seguir:

- adh('Thais') = 2
- adh('Edu') = 7
- adh('Bia') = 3
- adh('Neusa') = 5
- adh('Lucy') = 1
- adh('Rose') = 4
- adh('Yara') = 6

- adh(‘Decio’) = 2
- adh(‘Sueli’) = 4

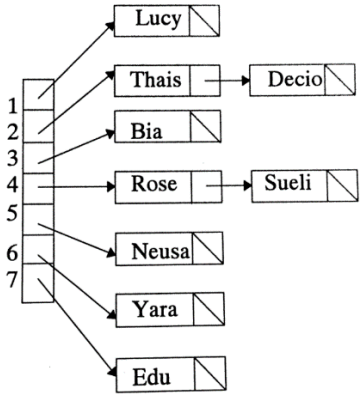


Figura 6. Espalhamento de chaves alfanuméricas.

Considere agora um conjunto de chaves alfanuméricas onde cada uma delas é apenas uma permutação dos mesmos caracteres básicos; por exemplo: ABC, ACB, BAC, BCA, CAB e CBA. Nesse caso, se estas chaves forem espalhadas pela função adh(), numa tabela com N=7 encaixes, teremos:

- adh(‘ABC’) = 3
- adh(‘ACB’) = 3
- adh(‘BAC’) = 3
- adh(‘BCA’) = 3
- adh(‘CAB’) = 3
- adh(‘CBA’) = 3

Claramente, se as chaves são todas compostas pelos mesmos caracteres, e a transformação da chave é baseada na soma dos seus “caracteres”, então todas as chaves serão mapeadas para a mesma classe (não há espalhamento).

Para resolver esse problema, vamos usar um algoritmo bastante eficiente para a transformação de chaves alfanuméricas denominado **somatório com deslocamentos**. Esse algoritmo associa a cada caractere da chave uma quantidade de bits que deve ser deslocada à esquerda no seu código ASCII, antes de ser adicionado à soma total. As quantidades a serem deslocadas à esquerda variam de 0 a 7, de forma cíclica, conforme esquematizado:

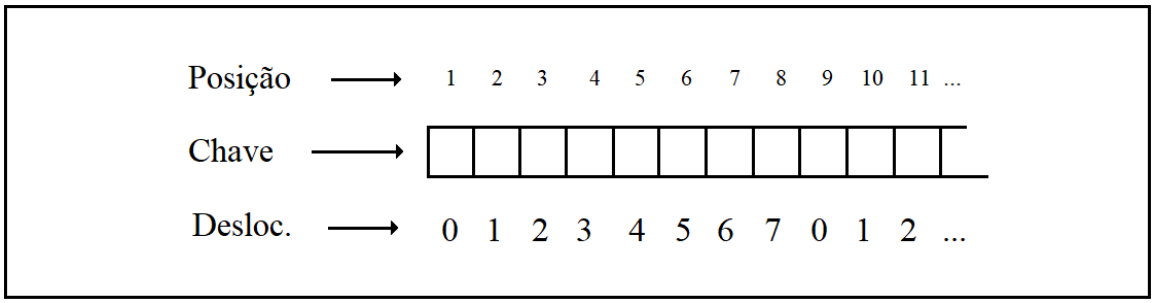


Figura 7. Quantidade de bits a ser deslocada para cada posição da chave.

No somatório com deslocamento, o valor numérico associado a um caractere x dentro da chave dependerá não somente do seu código, mas também da posição que ele ocupa dentro da chave.
Seu valor será:

ord(X) deslocado à esquerda de [(**posição-1**) **mod 8**] bits

Para entender melhor como esse algoritmo funciona, vamos tomar como exemplo, o código ASCII da letra A (65) na sua representação binária:

ord('A') = $(0\ 1\ 0\ 0\ 0\ 0\ 0\ 1)_B$

Se a letra A aparecer na primeira posição da chave, então o seu valor numérico será o seu próprio código ASCII, pois, para a primeira posição, o deslocamento é 0 e nada será alterado. Porém, se ela aparecer na segunda posição, seu código deverá ser deslocado de 1 bit à esquerda; se aparecer na terceira, 2 bits; na quarta, 3 e assim sucessivamente:
1ª posição: $(0\ 1\ 0\ 0\ 0\ 0\ 0\ 1)_B$ shl 0 = $(0\ 1\ 0\ 0\ 0\ 0\ 0\ 1)_B$ = 65
2ª posição: $(0\ 1\ 0\ 0\ 0\ 0\ 0\ 1)_B$ shl 1 = $(1\ 0\ 0\ 0\ 0\ 0\ 1\ 0)_B$ = -126
3ª posição: $(0\ 1\ 0\ 0\ 0\ 0\ 0\ 1)_B$ shl 2 = $(0\ 0\ 0\ 0\ 0\ 1\ 0\ 0)_B$ = 4
4ª posição: $(0\ 1\ 0\ 0\ 0\ 0\ 0\ 1)_B$ shl 3 = $(0\ 0\ 0\ 0\ 1\ 0\ 0\ 0)_B$ = 8
5ª posição: $(0\ 1\ 0\ 0\ 0\ 0\ 0\ 1)_B$ shl 4 = $(0\ 0\ 0\ 1\ 0\ 0\ 0\ 0)_B$ = 16
6ª posição: $(0\ 1\ 0\ 0\ 0\ 0\ 0\ 1)_B$ shl 5 = $(0\ 0\ 1\ 0\ 0\ 0\ 0\ 0)_B$ = 32
7ª posição: $(0\ 1\ 0\ 0\ 0\ 0\ 0\ 1)_B$ shl 6 = $(0\ 1\ 0\ 0\ 0\ 0\ 0\ 0)_B$ = 64
8ª posição: $(0\ 1\ 0\ 0\ 0\ 0\ 0\ 1)_B$ shl 7 = $(1\ 0\ 0\ 0\ 0\ 0\ 0\ 0)_B$ = -128
9ª posição: $(0\ 1\ 0\ 0\ 0\ 0\ 0\ 1)_B$ shl 0 = $(0\ 1\ 0\ 0\ 0\ 0\ 0\ 1)_B$ = 65

Obs: shl é um comando da linguagem Pascal para se deslocar bits de um valor inteiro à esquerda.

A função de hashing para chaves alfanuméricas com permutação usa o somatório com deslocamentos e o método da divisão, em conjunto:

```
function sdh(chv: string): integer;
var i, soma: integer;
begin
  soma:=0;
  for i:=1 to length(chv) do
    soma:= soma + ord(chv[i]) shl ((i-1) mod 8);
  sdh:= (abs(soma) mod N) + 1;
end;
```

Agora sim as chaves com permutação podem ser espalhadas numa tabela com 7 encaixes:

- sdh('ABC') = 4
- sdh('ACB') = 2
- sdh('BAC') = 3
- sdh('BCA') = 6
- sdh('CAB') = 7
- sdh('CBA') = 5

Um Algoritmo Simples de Espalhamento

O objetivo é que a função “espalhe” as chaves da maneira mais uniforme possível entre os endereços disponíveis.

O algoritmo tem 3 passos:

1. representação da chave numericamente (caso a chave não seja numérica). Por exemplo, usa os códigos ASCII dos caracteres (todos) para formar um número. Por exemplo:
   LOWELL$$$$$$ = 76 79 87 69 76 76 32 32 32 32 32 32

2. subdivisão e soma: subdivide o número em partes e soma as partes. No caso, subdivimos o número que descreve a chave em pares de códigos ASCII, que são tratados como valores inteiros, ao invés de caracteres:
   76 79 | 87 69 | 76 76 | 32 32 | 32 32 | 32 32

   Estes números inteiros podem ser somados, mas antes precisamos considerar o problema causado pela precisão limitada da máquina: em alguns sistemas, o maior inteiro permitido é 32.767 (15 bits), de maneira que se a soma resultar em um valor maior do que este podemos ter um *overflow*, ou um valor resultante negativo. Por exemplo, somando os primeiros 5 números o resultado é 30.588. Somando o último, 3.232, o resultado será maior que o máximo (33.820), causando um *overflow*.

3. divisão do resultado da soma pelo espaço de endereçamento para obter o endereço final. O objetivo deste passo é reduzir o tamanho do número gerado no passo anterior de forma que ele fique no intervalo de posições endereçáveis do arquivo. Isto pode ser feito dividindo o resultado pelo número total de endereços disponíveis, e tomando o resto da divisão inteira, que será interpretado como o endereço base para o registro. Ou seja, se ***s*** é o resultado da soma, ***n*** é o número de endereços do arquivo, então ***a*** é o endereço, sendo que:
   $$a = (s \text{ MOD } n) + 1$$
   O resto da divisão é um número entre 1 e *n*.

O método do meio do quadrado

O meio do quadrado é um método bastante usado para realizar espalhamentos uniformes. Dada uma chave inteira k, com $|k|$ bits, o método consiste em calcular k^2 e tomar $m<|k|$ bits do meio do quadrado. Com m bits, temos 2^m valores possíveis, logo, os valores gerados pela função estarão na faixa $[0..2^{m-1}]$ e o número de encaixes n da tabela a ser usada com esse método deve ser uma potência de 2 ($n=2^m$ ou $m=\log_2 n$). Observe no quadro a seguir, o espalhamento realizado pela função mq() que toma $m=3$ bits do meio do quadrado.

chave k	k^2	configuração binária de k^2	mq(k)
7	49	0000000 **000** 110001	0
12	144	0000000 **010** 010000	2
15	225	0000000 **011** 100001	3
22	484	0000000 **111** 100100	7
25	625	0000001 **001** 110001	1
37	1369	0000010 **101** 011001	5
48	2304	0000100 **100** 000000	4
55	3025	0000101 **111** 010001	7
59	3481	0000110 **110** 011001	6
60	3600	0000111 **000** 010000	0

Aplicação com arquivo

Função **HASH**
Endereço = F(chave)

No exemplo,
F(chave)=Numero mod 13

Área de Dados (primária)

	NÚMERO	NOME	SALÁRIO	STATUS	ELO
0	1950	Enio	$ 900	T	16
1	1600	Eber	$ 800	F	0
2					0
3					0
4	3150	Rui	$ 550	T	0
5	2150	Flavio	$ 420	F	0
6	1450	Diogo	$ 1200	T	0
7					0
8	1100	Antonio	$ 750	T	0
9	1400	Claudio	$ 600	T	0
10	1050	Afonso	$ 3200	T	17
11					0
12	1000	Ademar	$ 1200	T	14

Área de Overflow

	NÚMERO	NOME	SALÁRIO	STATUS	ELO
13	1700	Edson	$ 500	T	0
14	2300	Gerson	$ 800	T	0
15	3000	Ivan	$ 630	T	13
16	2600	Luis	$ 5000	T	0
17	2766	Pedro	$ 3000	T	15

COLISÃO utilizando tratamento por encadeamento (utilização da área de extensão (*overflow*)): Os registros que colidem em um mesmo endereço são armazenados em uma área de extensão (*overflow*) e inseridos na lista encadeada correspondente na área principal (atualização do campo ELO).
INSERÇÃO: é feita através da posição determinada para o armazenamento, através da função HASH.
REMOÇÃO: em geral é lógica.

Opções a serem construídas:
- Insere
- Altera
- Consulta
- Exclui
- Reorganiza (exclui todos os registros marcados com status false)
- Relatório (Exibe todos os registros, inclusive o número do registro no arquivo e o campo elo).

Referência

PEREIRA, Silvio do Lago, **Estruturas de dados fundamentais: conceitos e aplicações**. São Paulo: Érica, 2008.