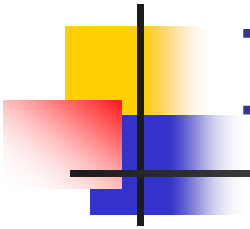
Hashing (Espalhamento)

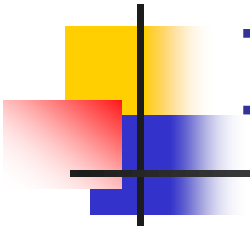
Estrutura de Dados II

Parte 1



Introdução

- Acesso a arquivo O(1)
  - Essa é a situação ideal, uma vez que o número de seeks é sempre o mesmo, não importa o quanto o arquivo possa crescer.
- Árvores-B levam a um acesso $O(\log_k N)$, sendo k proporcional ao tamanho da página
- Hashing: O(1)
  - Estático
  - Dinâmico



Introdução

- Assim, o hashing é uma técnica que agiliza o processo de busca de informações.
  - Tempo de busca: O(1)
- Além disso, é uma técnica que não requer ordenação, o que torna a inserção tão rápida quanto o acesso.

O que é Hashing?

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

$$\bigcup_{i=1}^{n} P_i = P \qquad \bigcap_{i=1}^{n} P_i = \varnothing$$

O que é Hashing?

- A correspondência unívoca entre os elementos do conjunto P e as N classes sugere a existência de um função h, através da qual é feito o particionamento.
- Tal função é chamada de **função de espalhamento** ou **função hashing**.

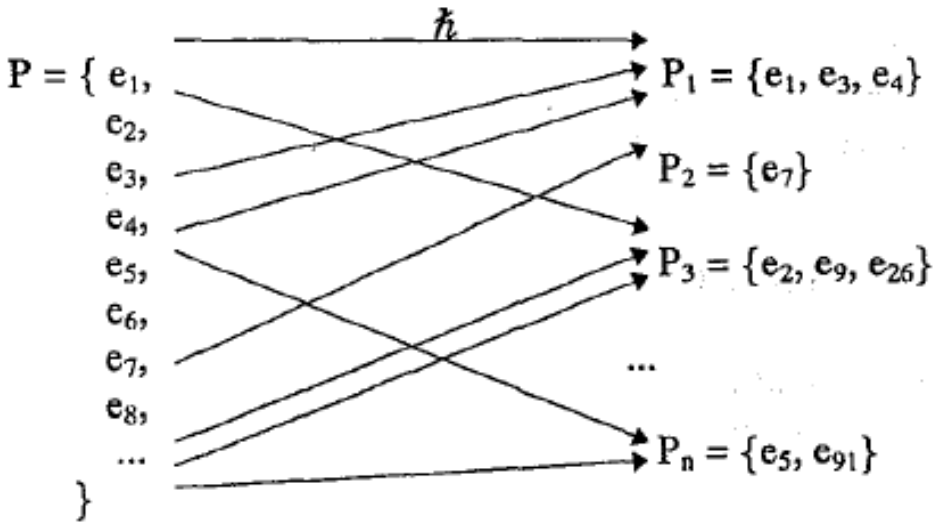
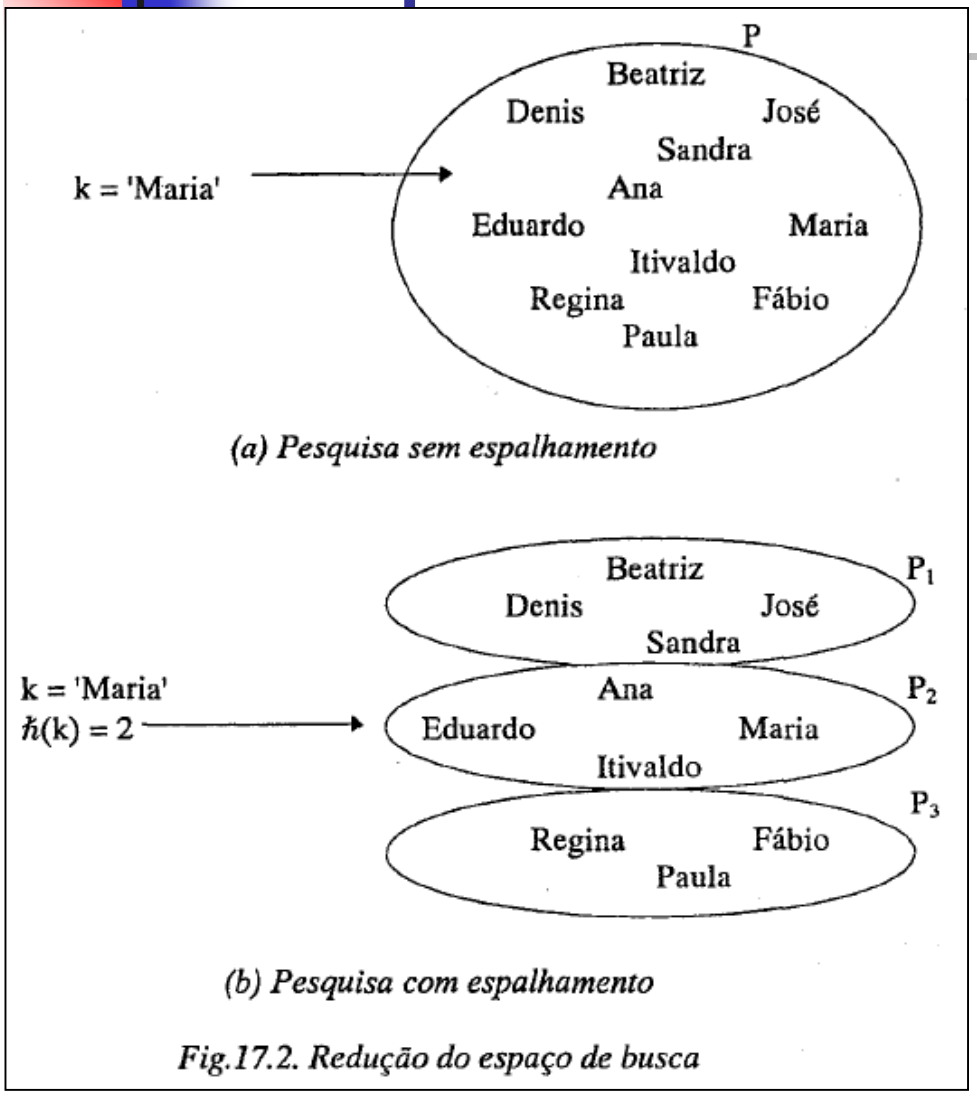
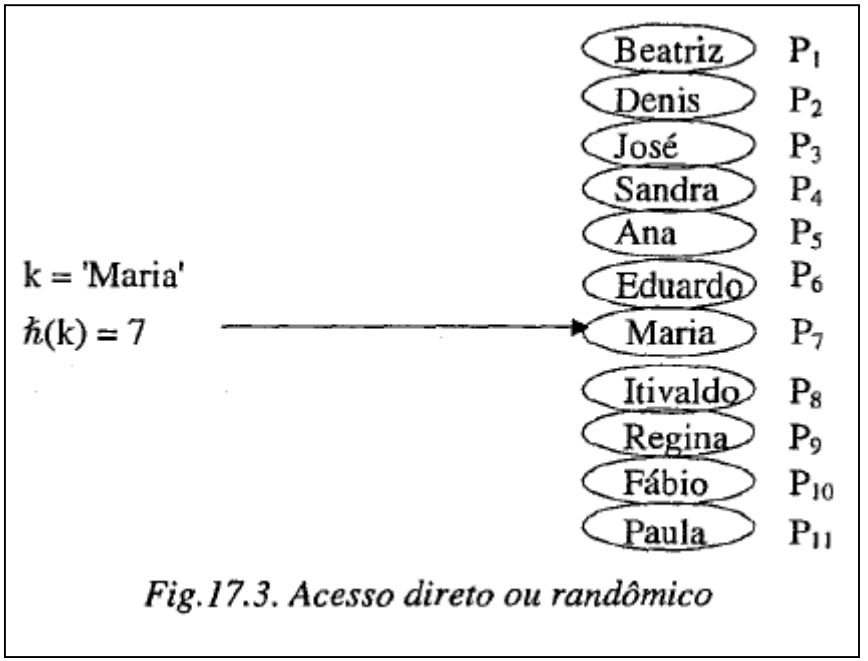
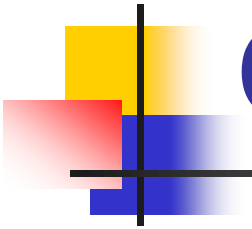


Fig.17.1. O espalhamento realizado pela função h

O que é Hashing?







O que é Hashing?

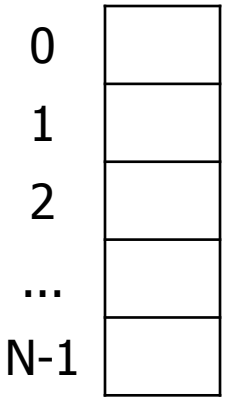
- Uma função hashing h(k) transforma uma chave k em um endereço.
- Este endereço é usado como base para o armazenamento e recuperação de registros.
- É similar a uma indexação, pois associa a chave ao endereço relativo do registro.

O que é Hashing?

EM MEMÓRIA

- Tabela Hashing
  - Estrutura de dados que implementa o espalhamento para aplicações em computador.

Tabela Hashing ou
Tabela de Espalhamento



O número de posições na tabela deve coincidir com o número de classes criadas pela função de espalhamento.

Hashing Estático: o tamanho do espaço de endereçamento não muda.

Hashing Dinâmico: O tamanho do espaço de endereçamento pode aumentar e diminuir.

O que é Hashing?

EM DISCO

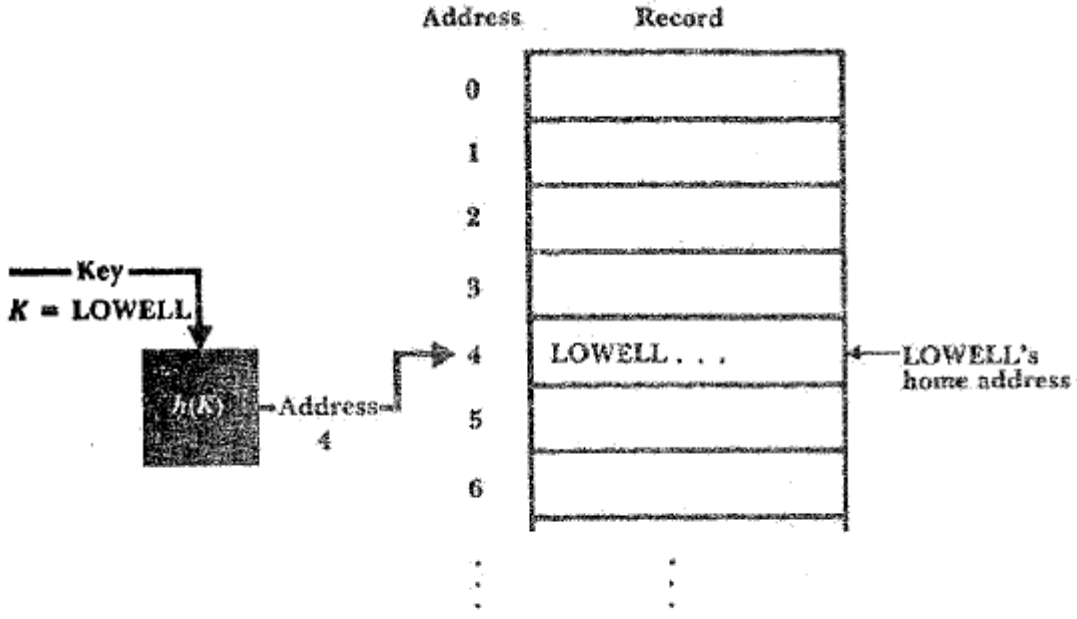


FIGURE 10.1 Hashing the key LOWELL to address 4.

Suponha que desejamos armazenar 75 registros em um arquivo, sendo que a chave para os registros é um campo sobrenome. Suponha que foi reservado espaço para manter 1.000 registros.

FUNÇÃO HASHING

TABLE 10.1 A simple hashing scheme

Name	ASCII Code for First Two Letters	Product	Home Address
BALL	66 65	66 × 65 = 4,290	290
LOWELL	76 79	76 × 79 = 6,004	004
TREE	84 82	84 × 82 = 6,888	888

Colisões

- Considerando o exemplo anterior, se calcularmos o endereço da chave OLIVIER, iremos produzir o endereço 004.
- Neste caso, ocorreu uma **colisão** com a chave LOWELL e dizemos que as chaves são **sinônimas**.

TABLE 10.1 A simple hashing scheme

Name	ASCII Code for First Two Letters	Product	Home Address
BALL	66 65	66 × 65 = 4,290	290
LOWELL	76 79	76 × 79 = 6,004	004
TREE	84 82	84 × 82 = 6,888	888

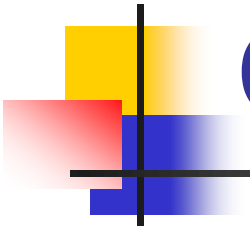
Colisões

- Colisões causam problemas, uma vez que não podemos colocar dois registros no mesmo espaço.
- Solução Ideal
  - Usar uma função de espalhamento perfeita, que não produza colisão.
  - Muito difícil (quase impossível) para mais de 500 chaves. Não vale a pena tentar.
  - Se tivermos uma aplicação especial que trabalha com menos que 500 registros fixos como, por exemplo, um compilador e a tabela de palavras reservadas de uma linguagem, pode valer a pena a busca pela função ideal.

Probabilidade de não ocorrência de colisão

$$\frac{N!}{(N-r)!N^r}$$

r=número chaves
N=número de entradas



Colisões

- A solução prática é tentar reduzir a um valor aceitável o número de colisões que podem ocorrer.
- Por exemplo, se uma entre 10 buscas resultar em colisão, então o número médio de acessos a disco necessários para recuperar um registro continua muito pequeno.

Um Algoritmo Simples de Espalhamento

- O objetivo ao se escolher um algoritmo de hashing é que a função "espalhe" as chaves da maneira mais uniforme possível entre os endereços disponíveis.
- Já vimos que a função anterior não faz isso, em parte porque usa só uma parte do valor da chave.

Um Algoritmo Simples de Espalhamento

- A função a seguir é bastante eficiente na maioria dos problemas.
- O algoritmo tem 3 passos:
  - Representar a chave numericamente (caso a chave não seja numérica).
  - Dobrar e somar (fold and add).
  - Dividir o resultado da soma pelo espaço de endereçamento para obter o endereço final. [a = s MOD N]

Um Algoritmo Simples de Espalhamento

- Passo 1: representar a chave numericamente (caso a chave não seja numérica).
  - Utilizar os códigos ASCII de cada um dos caracteres para formar um número.

LOWELL_____ = 76 79 87 69 76 76 32 32 32 32 32 32
L O W E L L |------Brancos------|

Um Algoritmo Simples de Espalhamento

FUNÇÕES HASHING

- Passo 2: Dobrar e somar (fold and add).
  - Somatório com dobra.

76 79 | 87 69 | 76 76 | 32 32 | 32 32 | 32 32

7.679 + 8.769 + 7.676 + 3.232 + 3.232 + 3.232 = 33.820

A dobra evita que padrões contendo permutações dos mesmos caracteres caiam no mesmo endereço. Exemplo: ABC, ACB, BAC, etc.

Um Algoritmo Simples de Espalhamento

- Passo 2: Dobrar e somar (fold and add).
  - 33.820 é maior que 32.767 (16 bits)
  - Considerando que as chaves contém apenas espaços e caracteres maiúsculos, maior parcela é ZZ (9.090)

```
 7679 + 8769 → 16448 → 16448 mod 19937 → 16448
16448 + 7676 → 24124 → 24124 mod 19937 →  4187
 4187 + 3232 →  7419 →  7419 mod 19937 →  7419
 7419 + 3232 → 10651 → 10651 mod 19937 → 10651
10651 + 3232 → 13883 → 13883 mod 19937 → 13883
```

```
FUNCTION hash(KEY,MAXAD)

    set SUM to 0
    set J to 0

    while (J < 12)
        set SUM to (SUM + 100*KEY[J] + KEY[J + 1]) mod 19937
        increment J by 2
    endwhile

    return (SUM mod MAXAD)

end FUNCTION
```

FIGURE 10.2 Function *hash(KEY,MAXAD)* uses folding and prime number division to compute a hash address.

Um Algoritmo Simples de Espalhamento

FUNÇÕES HASHING

- Passo 3: Dividir o resultado da soma pelo espaço de endereçamento para obter o endereço final.

$$[a = s \text{ MOD } N]$$

Gera um número entre 0 e (N-1)

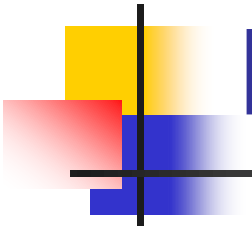
- Para N=100 (endereços entre 0 e 99)

$$a = 13.883 \text{ MOD } 100 = 83$$

Observação: recomenda-se que o valor escolhido para N seja um número primo. Isso porque a escolha de N influencia o espalhamento dos registros, e números primos tendem a distribuir os restos das divisões de maneira mais uniforme que os não primos.

No caso da chave LOWELL e do arquivo com 75 registros, se escolhermos N = 101 (o arquivo vai ficar 74.3% cheio), teremos a = 13.883 MOD 101 = 46 (RRN do registro).

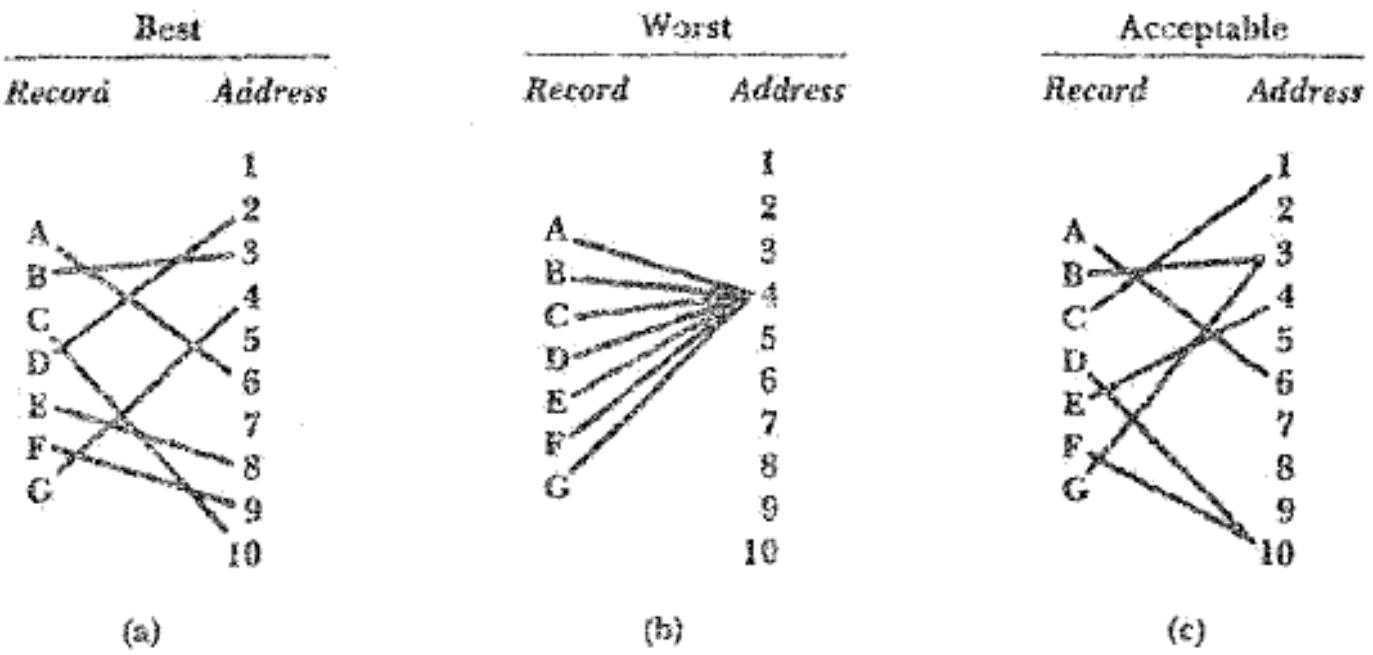
Distribuição de Registros entre Endereços



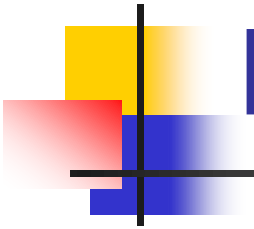
- A função de espalhamento perfeita (ou uniforme) para um dado conjunto de chaves seria uma que não produzisse nenhum sinônimo em um dado espaço de endereçamento.
- A pior função seria aquela que, qualquer que fosse a chave, geraria sempre o mesmo endereço (todas as chaves seriam sinônimas).
- Uma função aceitável é uma que gera poucos sinônimos.
  - Em geral, isto é mais provável de ocorrer se usarmos uma função que distribui as chaves aleatoriamente no conjunto de endereços disponíveis.

Distribuição de Registros entre Endereços

FIGURE 10.3 Different distributions. (a) No synonyms (uniform). (b) All synonyms (worst case). (c) A few synonyms.



Distribuição de Registros entre Endereços



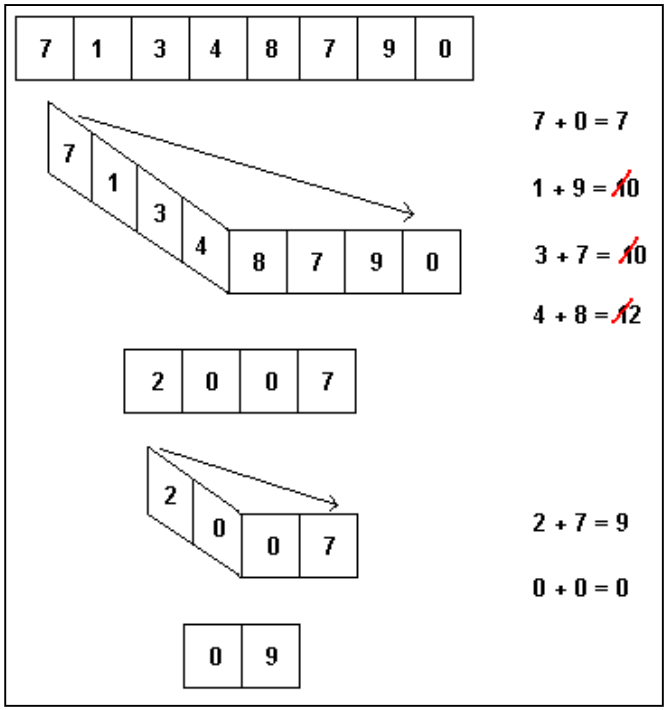
- Entretanto, existem situações em que é possível achar funções que produzem distribuições melhores que as aleatórias, uma vez que elas espalham as chaves entre os endereços disponíveis de maneira mais uniforme que uma distribuição aleatória, embora ainda produzam uma quantidade razoável de sinônimos.
- Experiência prática mostrou que as abordagens sugeridas a seguir funcionam bem, em circunstâncias adequadas.

Distribuição de Registros entre Endereços

FUNÇÕES HASHING

- Métodos de espalhamento potencialmente melhores que os aleatórios:
  - Método dobrar e somar
    - Este método destrói os padrões contidos nas chaves originais, mas em algumas circunstâncias pode preservar a separação entre certos sub-conjuntos das chaves que se espalham naturalmente.

O processo é repetido até que os dígitos formem um número menor que o tamanho da tabela hash.



Distribuição de Registros entre Endereços

FUNÇÕES HASHING

- Métodos de espalhamento potencialmente melhores que os aleatórios:
  - Método da divisão inteira.
    - Dividir a chave pelo número de endereços.
      - Escolher um número primo de endereços.
    - Preserva sequências de chaves consecutivas. Entretanto, se existem diversas sequências a divisão por um N pequeno pode causar muitas colisões.

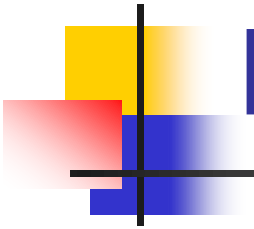
Os métodos até aqui apresentados são projetados para tirar vantagem da ordenação natural entre as chaves!!!

Distribuição de Registros entre Endereços

FUNÇÕES HASHING

- Métodos de espalhamento potencialmente melhores que os aleatórios [quando os anteriores não puderem ser utilizados, aleatorizar]:
  - Método do meio do quadrado.
    - Elevar a chave ao quadrado e considerar os dígitos intermediários do resultado (proporcional ao tamanho da tabela).
    - Exemplo: 453 → 453^2 → 205.209 → 52
      - Se estiver procurando por um endereço entre 0 e 99
    - Produzem resultados aleatórios justos
  - Mudar de base e dividir pelo espaço máximo de endereçamento (considerando o resto da divisão).
    - Exemplo: 453_{10} → 382_{11} → 382 MOD 99 → 85
      - Considerando 100 endereços de 0 a 99
    - Produzem resultados aleatórios justos

Distribuição de Registros entre Endereços



- Seria bom se existisse uma função hashing que garantisse uma distribuição melhor do que uma distribuição aleatória em todos os casos, mas não há.

- A distribuição gerada pela função hashing depende do conjunto de chaves a serem espalhadas e, portanto, é necessário levar o conjunto de chaves em consideração ao escolher a função.

Predizendo a Distribuição de Registros

- Densidade de Ocupação
  - Razão entre o número de registros a serem armazenados (r) e o número de espaços de endereçamento disponíveis (N, assumindo um registro por endereço).

$$Densidade\ de\ Ocupação = \frac{r}{N}$$

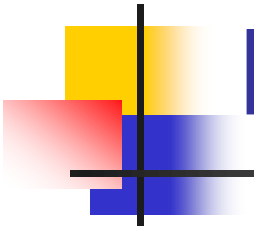
Predizendo a Distribuição de Registros

- Qual a probabilidade de um mesmo endereço possuir x registros associados depois da função ter sido aplicada aos r registros?

$$p(x) = \frac{(r/N)^x e^{-(r/N)}}{x!}$$

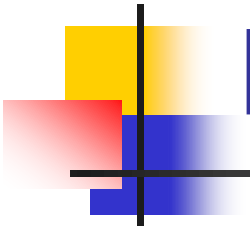
N = número de endereços disponíveis
r = número de registros a serem armazenados
x = número de registros associados a um mesmo endereço

Predizendo a Distribuição de Registros



- Exemplo: para 1.000 endereços e 1.000 registros, temos que:
  - p(0) = 0.368
    - probabilidade de um endereço ter nenhuma chave associada
  - p(1) = 0.368
    - probabilidade de um endereço ter uma chave associada
  - p(2) = 0.184
    - probabilidade de um endereço ter duas chaves associadas
  - p(3) = 0.061
    - probabilidade de um endereço ter três chaves associadas

Predizendo a Distribuição de Registros

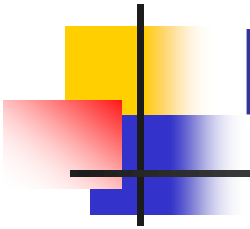


- Em geral, se existem N endereços, então o número esperado de endereços com x registros associados a ele é:

$$Np(x)$$

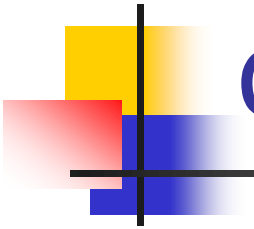
- Considerando o exemplo anterior:
  - 1.000*p(1) = 368
  - 1.000*p(2) = 184
  - 1.000*p(3) = 61

Predizendo a Distribuição de Registros



- Os 184 endereços com dois registros cada um representam um problema (colisão!!!).
  - Isso significa que 184 registros irão caber no endereço e 184 não.
  - Pior ainda no último caso: 61
    - 2*61 = 122 registros não irão caber

Quanto de memória extra deve ser utilizada?



- A fórmula p(x) permite prever o número de colisões para um arquivo cheio. Note que, na fórmula, a densidade de ocupação aparece duas vezes.
- A densidade de ocupação dá uma medida da quantidade de espaço do arquivo que está sendo de fato utilizada.
- É o único valor necessário para avaliar o desempenho de um espalhamento, assumindo que a função de espalhamento produz uma distribuição razoavelmente aleatória dos registros.
- Observe que quanto maior a densidade, maior a chance de ocorrerem colisões quando um novo registro precisar ser adicionado!!!

Quanto de memória extra deve ser utilizada?

- Exemplo:
  - Assumindo que apenas um registro pode ser associado a cada endereço, quantas colisões de registros são esperadas?
  - Supondo 1.000 endereços e 500 registros.

Densidade de ocupação = 50%

$$N*[1*p(2) + 2*p(3) + 3*p(4) + 4*p(5)] =$$
$$1.000*[1*0.0758 + 2*0.0126 + 3*0.0016 + 4*0.0002] = 107$$

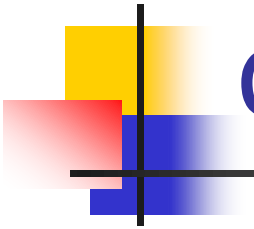
Porcentagem: 107/500 = 0.214 = 21.4%

Quanto de memória extra deve ser utilizada?

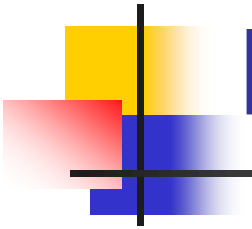
TABLE 10.2 Effect of packing density on the proportion of records not stored at their home addresses

Packing Density (%)	Synonyms as % of Records
10	4.8
20	9.4
30	13.6
40	17.6
50	21.4
60	24.8
70	28.1
80	31.2
90	34.1
100	36.8

Quanto de memória extra deve ser utilizada?



- Por outro lado, precisamos decidir quanto de espaço estamos dispostos a perder para reduzir o número de colisões.
- Queremos o menor número de colisões, mas não ao custo de precisar usar 2 discos para armazenar um arquivo que poderia caber em 1!



Resolução de Colisões

- Ainda que se tenha um ótimo algoritmo de espalhamento, é bastante provável que colisões ocorram.
- Portanto, qualquer solução que use espalhamento precisa incorporar algum mecanismo para tratar registros que não possam ser colocados no seu endereço-base.
- Existem várias maneiras de se tratar colisões.

http://www.ee.ryerson.ca/~courses/coe428/structures/hash.html
https://www.cs.usfca.edu/~galles/visualization/Algorithms.html
http://www.csanimated.com/animation.php?t=Hash_table

Resolução de Colisões

- Encadeamento Aberto ou
  Hashing Fechado/Interno
  - Re-espalhamento Linear (linear probing) ou
    Overflow Progressivo (Busca, Inserção, Remoção)
  - Re-espalhamento Quadrático
  - Espalhamento Duplo
- Encadeamento Interno ou
  Overflow Progressivo Encadeado
- Encadeamento ou
  Encadeamento com Área de Overflow Separada ou
  Hashing Aberto/Externo

Resolução de Colisões por Overflow Progressivo

- Uma técnica bastante simples mas que pode funcionar bem em muitas situações.
- O tratamento de uma colisão é feito procurando-se a próxima posição vazia depois do endereço-base da chave.

$$h(k) = k \text{ MOD } N$$

$$rh_j(h(k)) = (h(k) + j) \text{ MOD } N,\ 1 \leq j \leq N-1$$

Resolução de Colisões por Overflow Progressivo

$$h(k) = k \text{ MOD } N$$

$$rh_j(h(k)) = (h(k) + j) \text{ MOD } N,\ 1 \leq j \leq N-1$$

0	U
1	N
2	S
3	
4	
5	L
6	E

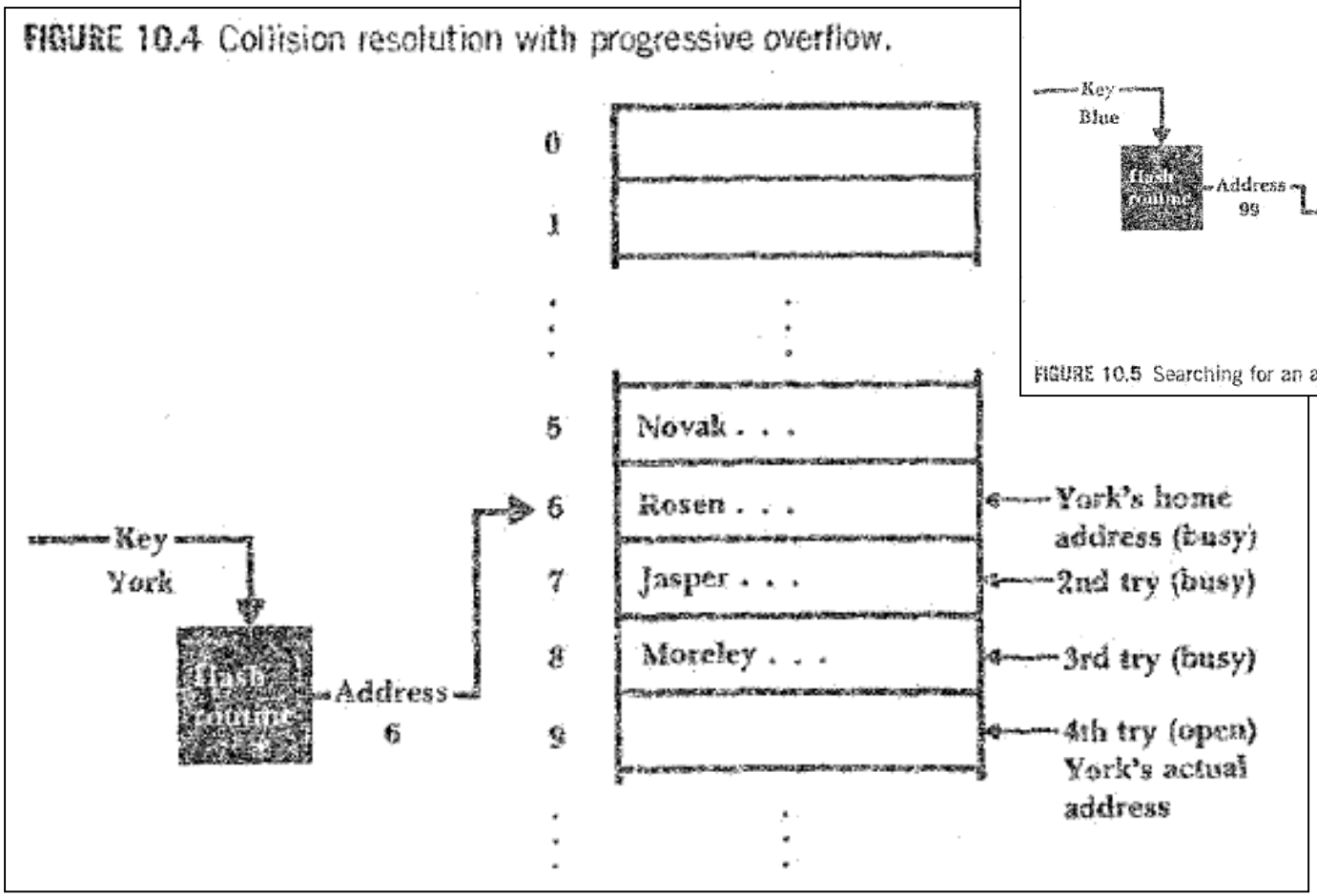
h(L) = h(12) = 12 MOD 7 = 5
h(U) = h(21) = 21 MOD 7 = 0

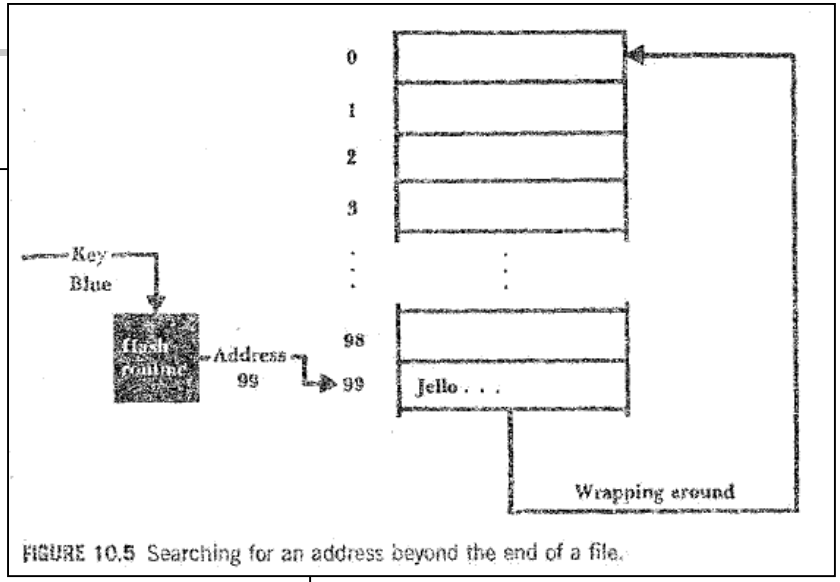
h(N) = h(14) = 14 MOD 7 = 0
$rh_1(h(14)) = (h(14) + 1) \text{ MOD } 7 = 1$

h(E) = h(5) = 5 MOD 7 = 5
$rh_1(h(5)) = (h(5) + 1) \text{ MOD } 7 = 6$

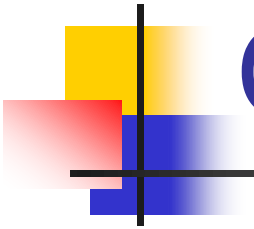
h(S) = h(19) = 19 MOD 7 = 5
$rh_1(h(19)) = (h(19) + 1) \text{ MOD } 7 = 6$
$rh_2(h(19)) = (h(19) + 2) \text{ MOD } 7 = 0$
$rh_3(h(19)) = (h(19) + 3) \text{ MOD } 7 = 1$
$rh_4(h(19)) = (h(19) + 4) \text{ MOD } 7 = 2$

Resolução de Colisões por Overflow Progressivo



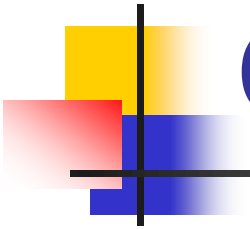


Resolução de Colisões por Overflow Progressivo



- Problema com a busca!!!
- O que acontece se ocorre a busca por um registro que nunca esteve no arquivo?
  - Se um endereço aberto é encontrado, a rotina de busca assume que o registro não está no arquivo.
  - Se o arquivo está cheio, a busca volta onde começou e conclui que o registro não está no arquivo. Torna a busca muito lenta!!!

Resolução de Colisões por Overflow Progressivo



- Ou seja, se ocorrerem muitas colisões, pode ocorrer um clustering (agrupamento) de chaves em uma certa área.
- Isso pode fazer com que sejam necessários muitos acessos para recuperar um certo registro.
- O problema vai ser agravado se a densidade de ocupação para o arquivo for alta.

Resolução de Colisões por Overflow Progressivo

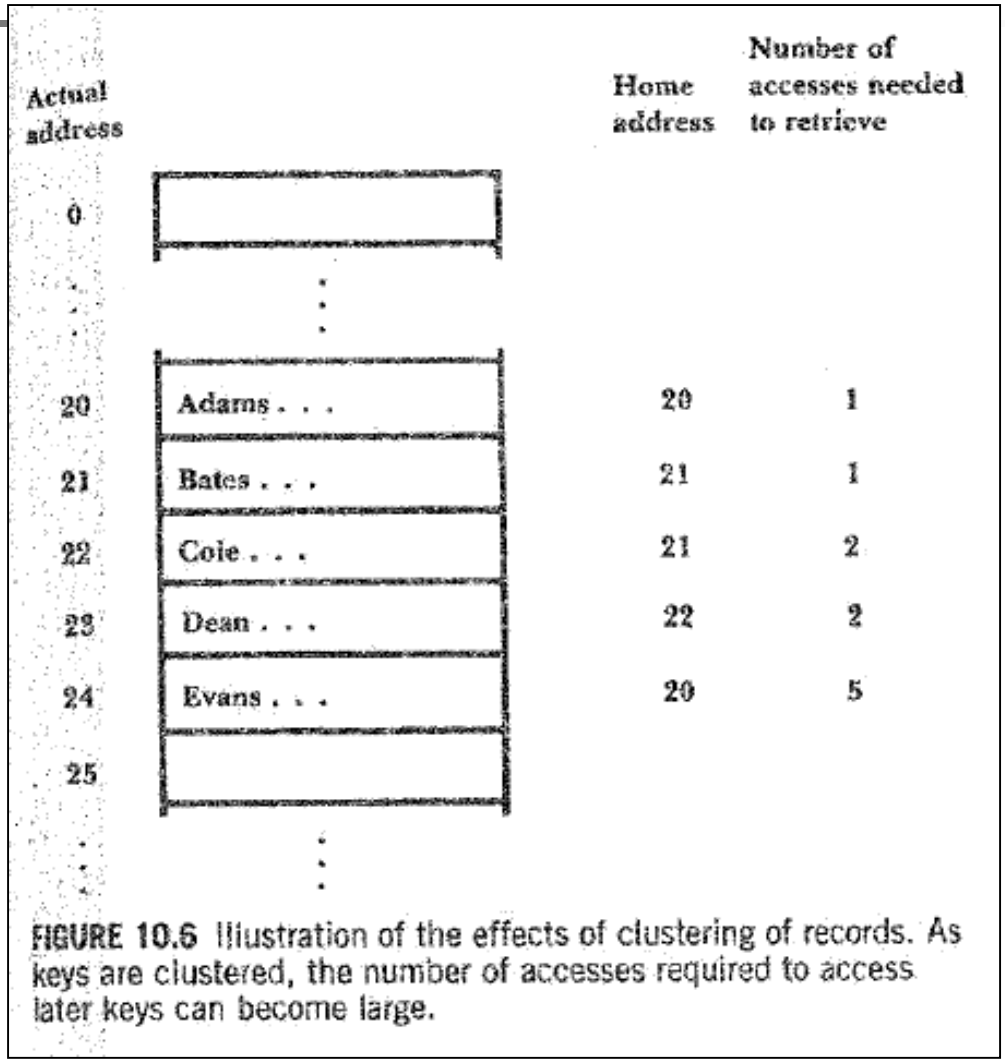
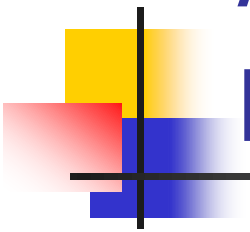


FIGURE 10.6 Illustration of the effects of clustering of records. As keys are clustered, the number of accesses required to access later keys can become large.

Armazenamento de Vários Registros por Endereço: Cestos (Buckets)



- Cestos permitem armazenar mais de um registro em um mesmo endereço.
  - Custo para recuperar um ou mais registros é o mesmo depois de localizado no disco. (idem árvore-B).
  - Escreve e lê um cesto por vez; menos acessos e menos colisões!!!
- Por exemplo, um cesto de tamanho dois permite alocar dois registros, e só existirá colisão quando um terceiro registro precisar ser alocado no mesmo endereço.
- Note que a fórmula da densidade de ocupação muda para r/bN, onde b é número de registros em um cesto e bN o número de localizações disponíveis.
- O uso de cestos melhora consideravelmente o desempenho do espalhamento, uma vez que aumenta a proporção de registros por endereço e diminui o número de colisões.

Armazenamento de Vários Registros por Endereço: Cestos (Buckets)

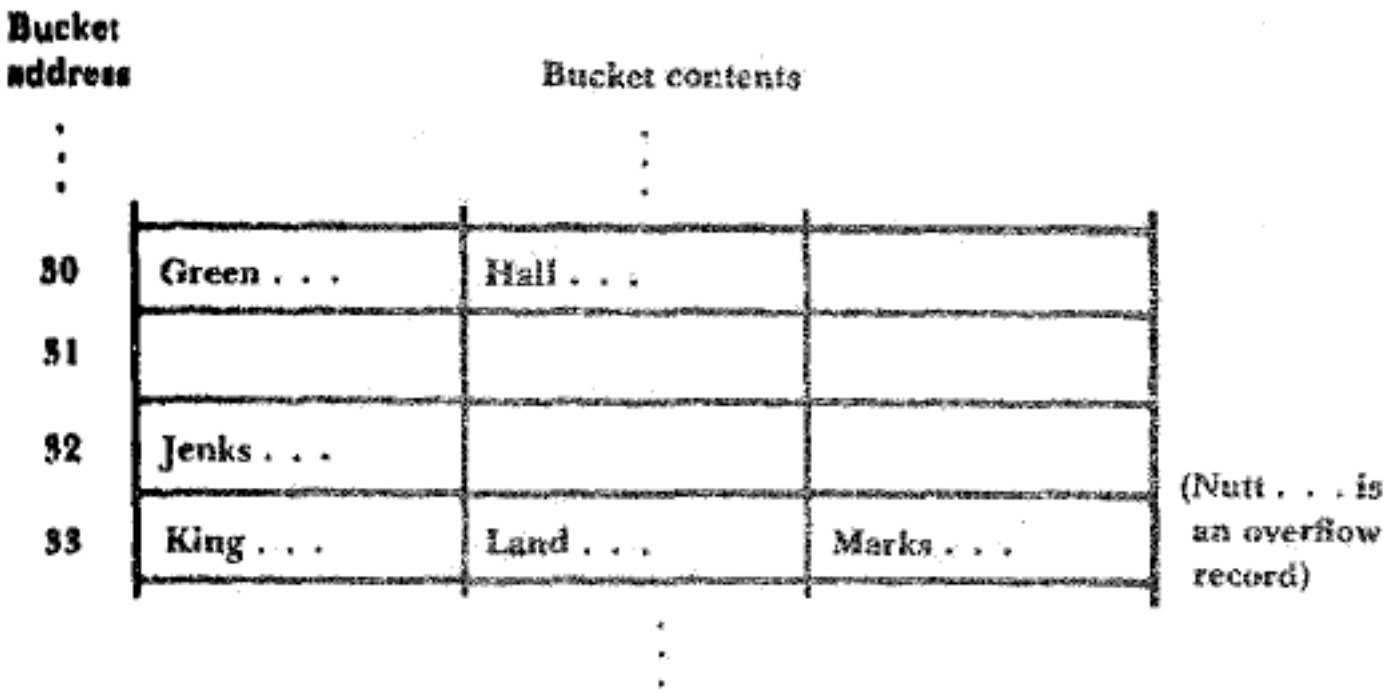


FIGURE 10.8 An illustration of buckets. Each bucket can hold up to three records. Only one synonym (Nutt) results in overflow.

Armazenamento de Vários Registros por Endereço: Cestos (Buckets)

Mesma Densidade de Ocupação	File without Buckets	File with Buckets
Number of records	$r = 750$	$r = 750$
Number of addresses	$N = 1{,}000$	$N = 500$
Bucket size	$b = 1$	$b = 2$
Packing density	0.75	0.75
Ratio of records to addresses	$r/N = 0.75$	$r/N = 1.5$

TABLE 10.3 Poisson distributions for two different file organizations

$p(x)$	File without Buckets ($r/N = 0.75$)	File with Buckets ($r/N = 1.5$)
$p(0)$	0.472	0.223
$p(1)$	0.354	0.335
$p(2)$	0.133	0.251
$p(3)$	0.033	0.126
$p(4)$	0.006	0.047
$p(5)$	0.001	0.014
$p(6)$	—	0.004
$p(7)$	—	0.001

Colisões – Sem Cestos
N*[1*p(2) + 2*p(3) + 3*p(4) + 4*p(5) + ...] = 222 = 29.6%

Colisões – Com cestos
N*[1*p(3) + 2*p(4) + 3*p(5) + 4*p(6) + ...] = 140 = 18.7%

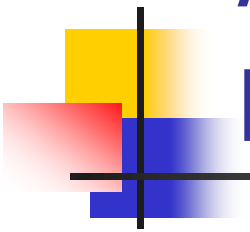
Redução de 37%
(37% de esforço a menos para localizar um registro que não se encontra em seu endereço)

Armazenamento de Vários Registros por Endereço: Cestos (Buckets)

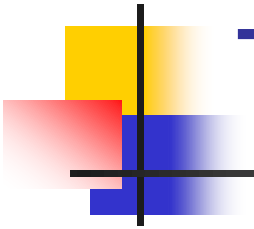
TABLE 10.4 Synonyms causing collisions as a percent of records for different packing densities and different bucket sizes

Packing Density (%)	Bucket Size				
	1	2	5	10	100
10	4.8	0.6	0.0	0.0	0.0
20	9.4	2.2	0.1	0.0	0.0
30	13.6	4.5	0.4	0.0	0.0
40	17.6	7.3	1.1	0.1	0.0
50	21.3	10.4	2.5	0.4	0.0
60	24.8	13.7	4.5	1.3	0.0
70	28.1	17.0	7.1	2.9	0.0
75	29.6	18.7	8.6	4.0	0.0
80	31.2	20.4	10.3	5.3	0.1
90	34.1	23.8	13.8	8.6	0.8
100	36.8	27.1	17.6	12.5	4.0

Armazenamento de Vários Registros por Endereço: Cestos (Buckets)

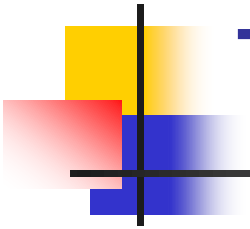


- Quão grande deve ser um cesto?
- Cuidados na escolha do tamanho dos cestos
  - Evite cestos maiores que uma trilha (ao menos que os registros sejam muito grandes).
    - Uma vez que o hashing requer a recuperação de um único registro por busca, qualquer tempo de transmissão adicional é desperdício.
  - O menor número pode ser o de um cluster.



Tópicos de Implementação

- Um arquivo hash é um arquivo de registros de tamanho fixo, cujos registros são acessados por RRN.
- Arquivos hash diferem dos arquivos até então discutidos em aulas passadas em dois aspectos:
  - Uma vez que a função hashing depende da existência de um número fixo de endereços disponíveis, o tamanho lógico do arquivo hash deve ser fixado antes do arquivo ser preenchido com registros, e deve permanecer fixo durante o tempo que a mesma função hashing é usada.

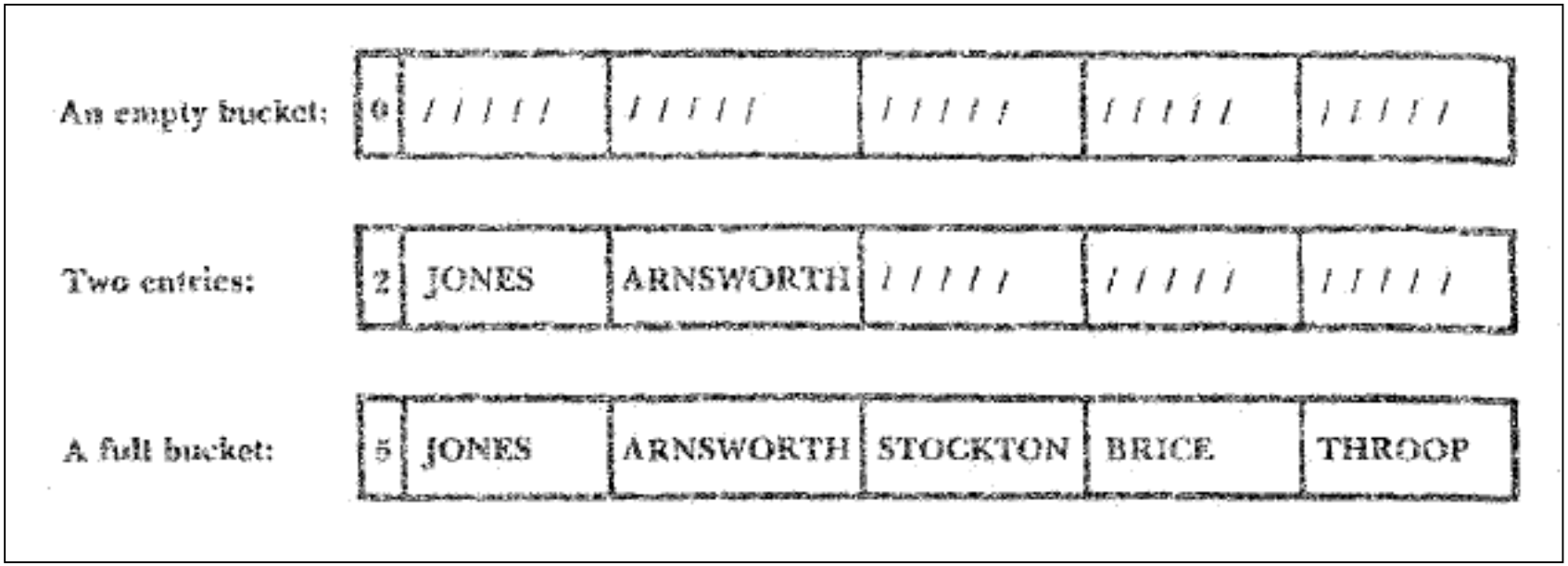


Tópicos de Implementação

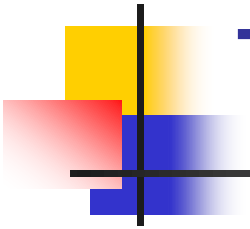
- Arquivos hash diferem dos arquivos até então discutidos em aulas passadas em dois aspectos:
  - Uma vez que o endereço RRN de um registro em um arquivo hash está unicamente relacionado com a sua chave, qualquer procedimento que adicione, remova, ou altere um registro não deve quebrar a ligação entre o registro e seu RRN. Se isso ocorrer, o registro não poderá mais ser recuperado por hashing.

Tópicos de Implementação

- Estrutura do Cesto

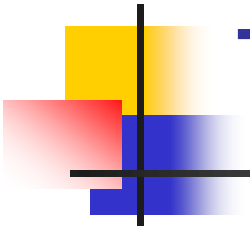


/////: indica se o slot do cesto está vazio



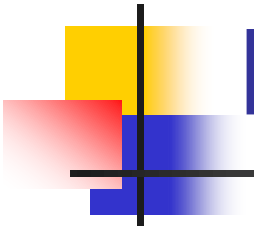
Tópicos de Implementação

- Inicialização do Arquivo
  - Uma vez que o tamanho lógico de um arquivo hash deve permanecer fixo, é necessário alocar espaço físico para o arquivo antes de começar armazenar os registros.
  - Isso geralmente é feito criando um arquivo de espaços vazios para todos os registros.
    - Aumenta a probabilidade dos registros serem armazenados próximos uns dos outros no disco, facilitando o processamento sequencial do arquivo.



Tópicos de Implementação

- Carregando um Arquivo Hash
  - Usar a função hashing para produzir o endereço de cada chave.
  - Em caso de colisão, utilizar o overflow progressivo.
  - Se não for permitido armazenar chaves duplicadas no arquivo, deverá ser implementado algum mecanismo de tratamento para este problema.



Remoções

- Deletar registros de um arquivo hash é mais complicado do que adicionar, uma vez que:
  - O slot liberado pela remoção não pode permitir o impedimento de buscas futuras.
  - Deverá ser possível reusar o slot liberado para adições futuras.
- Quando o overflow progressivo é utilizado, a busca por um registro termina se um endereço vazio é encontrado. Assim, não podemos deixar um endereço vazio de forma a quebrar a busca inapropriadamente.

Remoções

Record	Home address	Actual address
Adams	5	5
Jones	6	6
Morris	6	7
Smith	5	8

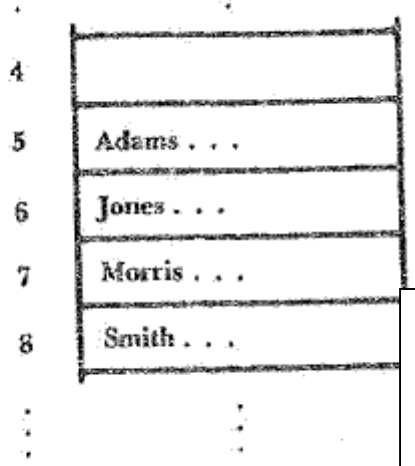


FIGURE 10.9 File organization before deletions.

Buscar por Smith

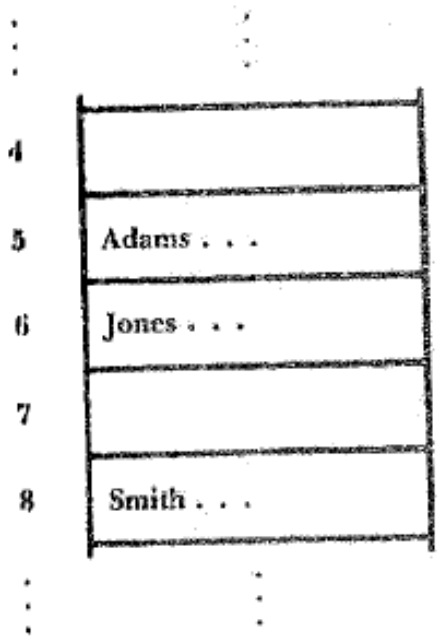
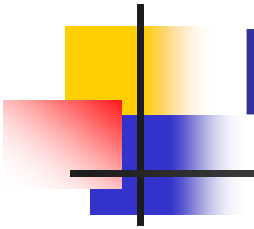


FIGURE 10.10 The same organization as in Fig. 10.9, with Morris deleted.



Remoções

- Uma solução para esse problema é utilizar uma marca especial chamada tombstone (por exemplo, #####), para identificar vagas nas quais registros foram eliminados.
- Deste modo:
  - O espaço vago não determina o término da pesquisa pelo registro.
  - O espaço vago pode ser utilizado para adições futuras.

Remoções

Record	Home address	Actual address
Adams	5	5
Jones	6	6
Morris	6	7
Smith	5	8

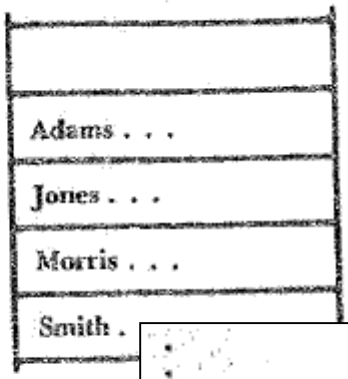


FIGURE 10.9 File organization before deletions.

Buscar por Smith

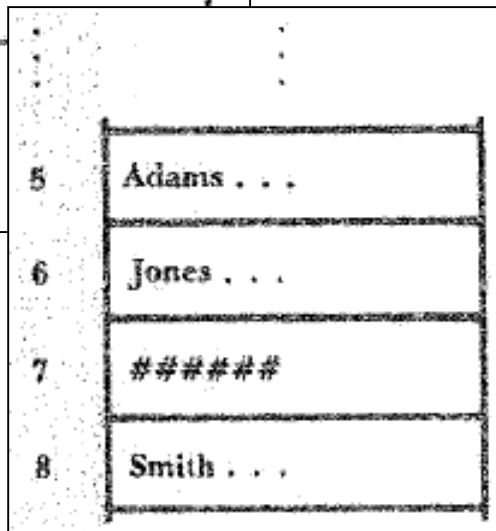


FIGURE 10.11 The same file as in Fig. 10.9 after the insertion of a tombstone for Morris.

Implicações na Inserção

- Assim, durante a inserção de um registro, é possível que o mesmo seja inserido tanto quando um slot vazio for encontrado ///// tanto quando um slot liberado for encontrado #####.
- Problema???

Inserir Smith

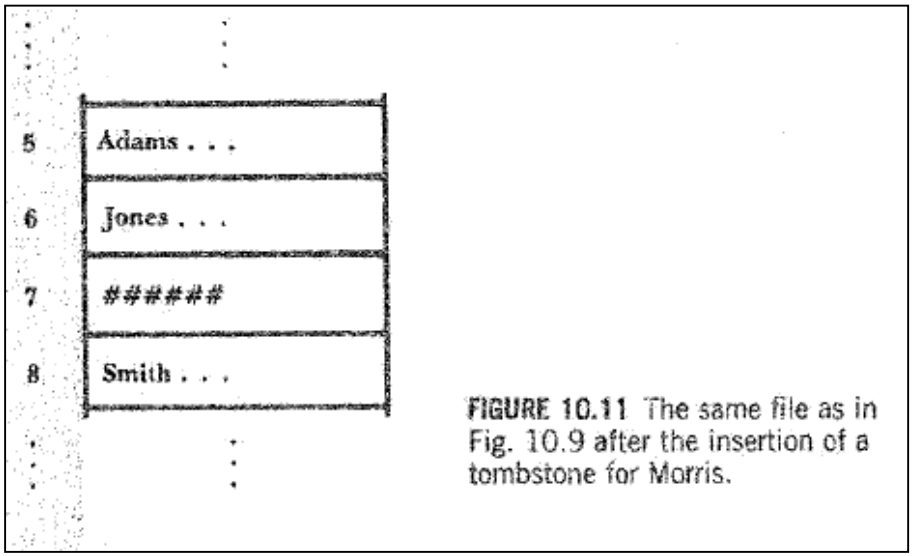
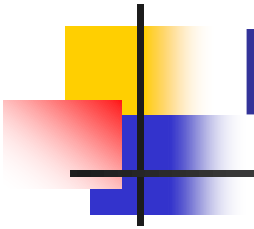


FIGURE 10.11 The same file as in Fig. 10.9 after the insertion of a tombstone for Morris.

É necessário examinar todo o agrupamento de chaves contíguas para se certificar que a chave a ser inserida não existe no arquivo e aí então voltar para a primeira posição disponível e realizar a inserção



Efeitos no Desempenho

- Após um certo número de inserções e eliminações o desempenho piora.
  - Espera-se encontrar muitos tombstones ocupando lugares que seriam ocupados por registros cujos endereços base os precedem (ver fig. anterior)
- Após um certo limite, toda a organização está comprometida.
- Soluções:
  - Reorganizar localmente após cada eliminação.
  - Reorganizar completamente o arquivo.
  - Usar outra técnica para tratar as colisões.

Resolução de Colisões

- Encadeamento Aberto ou
  Hashing Fechado/Interno
  - Re-espalhamento Linear (linear probing) ou
    Overflow Progressivo (Busca, Inserção, Remoção)
    - Vantagem: simplicidade.
    - Desvantagem: agrupamento primário; com estrutura cheia a busca fica lenta; dificulta inserções e remoções.
  - Re-espalhamento Quadrático
  - Espalhamento Duplo
- Encadeamento Interno ou
  Overflow Progressivo Encadeado
- Encadeamento ou
  Encadeamento com Área de Overflow Separada ou
  Hashing Aberto/Externo

Resolução de Colisões por Re-espalhamento Quadrático

$$h(k) = k \text{ MOD } N$$

$$rh_j(h(k)) = (h(k) + j^2) \text{ MOD } N,\ 1 \leq j \leq N-1$$

Tenta diminuir o agrupamento primário gerado pelo overflow progressivo.

Entretanto, gera o agrupamento secundário.

$h(x_1,0) = h(x_2,0) \rightarrow h(x_1,i) = h(x_2,i)$

0	U
1	N
2	S
3	
4	
5	L
6	E

h(L) = h(12) = 12 MOD 7 = 5
h(U) = h(21) = 21 MOD 7 = 0

h(N) = h(14) = 14 MOD 7 = 0
$rh_1(h(14)) = (h(14) + 1^2) \text{ MOD } 7 = 1$

h(E) = h(5) = 5 MOD 7 = 5
$rh_1(h(5)) = (h(5) + 1^2) \text{ MOD } 7 = 6$

h(S) = h(19) = 19 MOD 7 = 5
$rh_1(h(19)) = (h(19) + 1^2) \text{ MOD } 7 = 6$
$rh_2(h(19)) = (h(19) + 2^2) \text{ MOD } 7 = 2$

Resolução de Colisões por Espalhamento Duplo

$$h(k) = k \text{ MOD } N$$

$$h_2(k) = 1 + (K \text{ MOD } (N-1))$$

$$rh_j(h(k)) = (h(k) + j*h_2(k)) \text{ MOD } N,\ 1 \leq j \leq N-1$$

0	U
1	
2	
3	N
4	E
5	L
6	S

h(L) = h(12) = 12 MOD 7 = 5
h(U) = h(21) = 21 MOD 7 = 0

h(N) = h(14) = 14 MOD 7 = 0
$rh_1(h(14)) = (h(14) + 1*h_2(14)) \text{ MOD } 7 = (0 + 3) \text{ MOD } 7 = 3$

h(E) = h(5) = 5 MOD 7 = 5
$rh_1(h(5)) = (h(5) + 1*h_2(5)) \text{ MOD } 7 = (5 + 6) \text{ MOD } 7 = 4$

h(S) = h(19) = 19 MOD 7 = 5
$rh_1(h(19)) = (h(19) + 1*h_2(19)) \text{ MOD } 7 = (5 + 2) \text{ MOD } 7 = 0$
$rh_2(h(19)) = (h(19) + 2*h_2(19)) \text{ MOD } 7 = (5 + 4) \text{ MOD } 7 = 2$
$rh_3(h(19)) = (h(19) + 3*h_2(19)) \text{ MOD } 7 = (5 + 6) \text{ MOD } 7 = 4$
$rh_4(h(19)) = (h(19) + 4*h_2(19)) \text{ MOD } 7 = (5 + 8) \text{ MOD } 7 = 6$

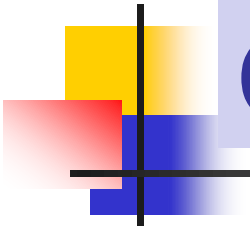
Evita agrupamento. É um dos melhores métodos para endereçamento aberto, pois as permutações produzidas são semelhantes às permutações aleatórias.

Entretanto, os endereços podem estar muito distantes um do outro, provocando acessos adicionais.

Resolução de Colisões

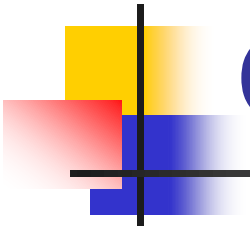
- Encadeamento Aberto ou
  Hashing Fechado/Interno
  - Re-espalhamento Linear (linear probing) ou
    Overflow Progressivo (Busca, Inserção, Remoção)
    - Vantagem: simplicidade.
    - Desvantagem: agrupamento primário; com estrutura cheia a busca fica lenta; dificulta inserções e remoções.
  - Re-espalhamento Quadrático
  - Espalhamento Duplo
- Encadeamento Interno ou
  Overflow Progressivo Encadeado
- Encadeamento ou
  Encadeamento com Área de Overflow Separada ou
  Hashing Aberto/Externo

Resolução de Colisões por Overflow Progressivo Encadeado



- É um método para resolver problemas de colisões mediante o emprego de listas encadeadas que compartilham o mesmo espaço de memória que a tabela de espalhamento.
- É similar ao Overflow Progressivo, com a diferença de que os sinônimos são encadeados na própria área de endereçamento.
  - Para cada conjunto de sinônimos existe uma lista encadeada conectando seus registros, e é essa lista que é percorrida quando um registro é procurado.

Resolução de Colisões por Overflow Progressivo Encadeado



- A vantagem do Overflow Progressivo Encadeado sobre o Overflow Progressivo é que somente os registros com chaves sinônimas são acessados em um determinada busca.
- Entretanto, tem como desvantagem a necessidade da existência de um campo de ligação, a fim de garantir que sinônimos sejam encontrados.

olisões por
essivo Encadeado

FIGURE 10.12 Hashing with progressive overflow.

Key	Home address	Actual address	Search length
Adams	20	20	1
Bates	21	21	1
Cole	20	22	3
Dean	21	23	3
Evans	24	24	1
Flint	20	25	6

Average search length = (1 + 1 + 3 + 3 + 1 + 6)/6 = 2.5

Buscar Flint

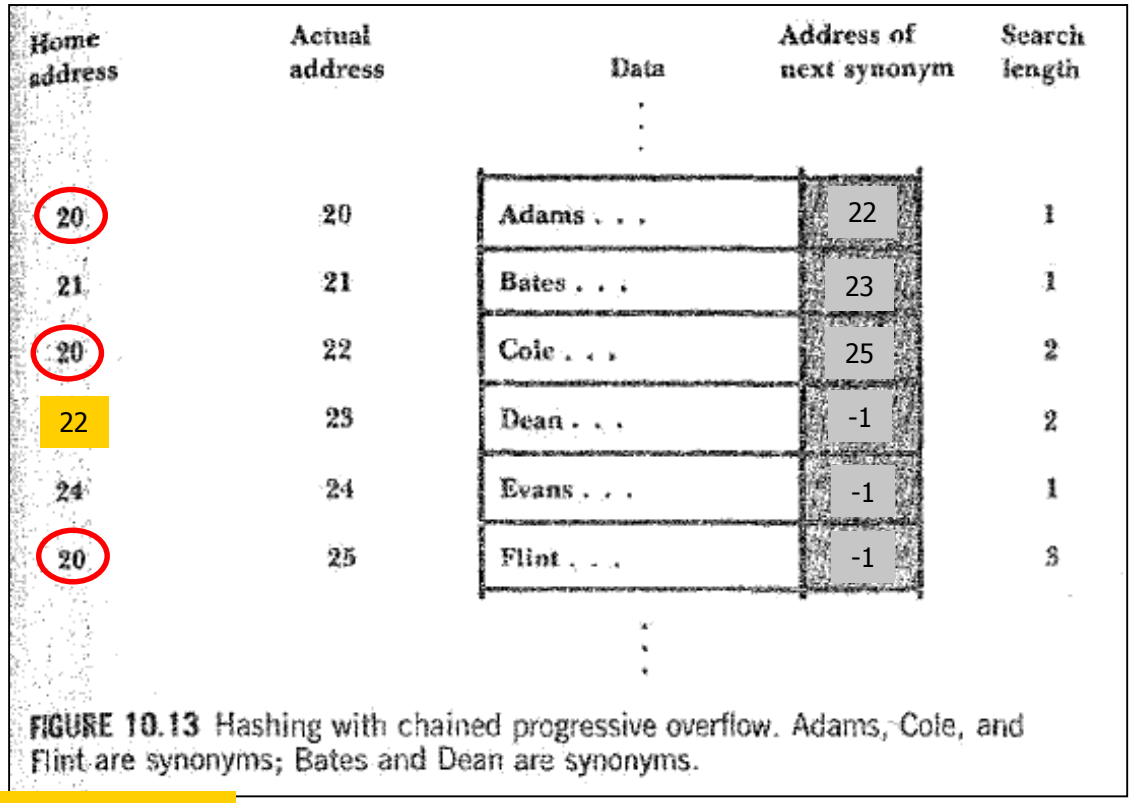
Home address	Actual address	Data	Address of next synonym	Search length
20	20	Adams . . .	22	1
21	21	Bates . . .	23	1
20	22	Cole . . .	25	2
21	23	Dean . . .	-1	2
24	24	Evans . . .	-1	1
20	25	Flint . . .	-1	3

FIGURE 10.13 Hashing with chained progressive overflow. Adams, Cole, and Flint are synonyms; Bates and Dean are synonyms.

Resolução de Colisões por Overflow Progressivo Encadeado

Se o endereço de Dean for 22 ao invés de 21, o que fazer?

Problema: tratamento especial de chaves que não podem ocupar seu endereço base pois este já está ocupado por um registro de outra lista.

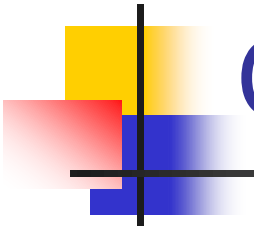


Home address	Actual address	Data	Address of next synonym	Search length
20	20	Adams . . .	22	1
21	21	Bates . . .	23	1
20	22	Cole . . .	25	2
22	23	Dean . . .	-1	2
24	24	Evans . . .	-1	1
20	25	Flint . . .	-1	3

FIGURE 10.13 Hashing with chained progressive overflow. Adams, Cole, and Flint are synonyms; Bates and Dean are synonyms.

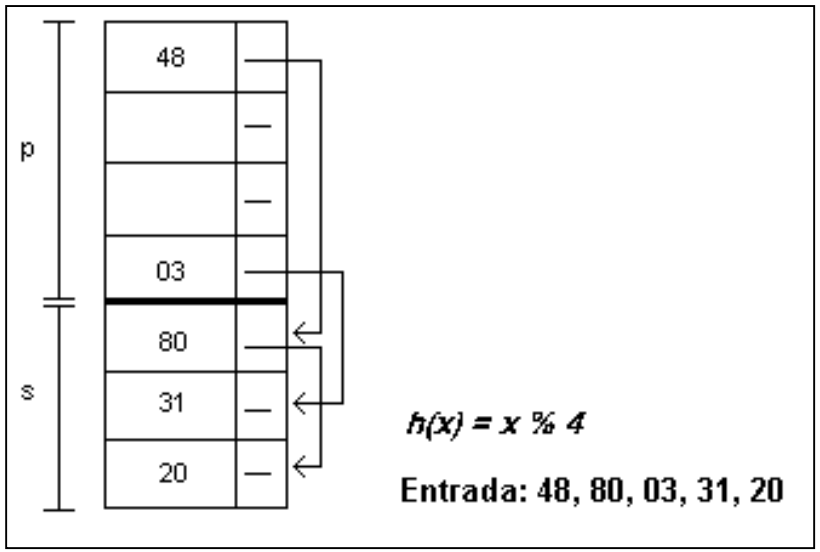
Informações sobre técnicas para manutenção dos campos de ligação ver Knuth 1973, The Art of Programming, vol. 3, Searching and Sorting.

Resolução de Colisões por Overflow Progressivo Encadeado

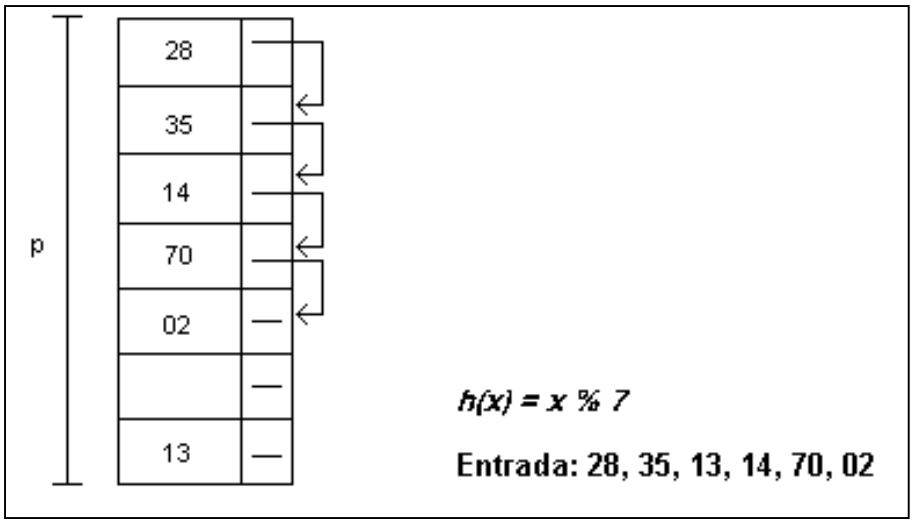


- Solução: two-pass loading
  - carga de todas as chaves no endereço-base num primeiro passo;
  - carga das chaves de colisão no segundo passo.
  - Resolve no carregamento da lista.
  - Não resolve se forem necessárias inserções e eliminações.
- Outra solução: divisão da tabela (vide a seguir)

Resolução de Colisões por Overflow Progressivo Encadeado



Podemos prever a divisão da tabela T em duas zonas, uma de endereços-base, de tamanho p, e outra reservada para colisões, de tamanho s. Naturalmente, p+s = N, onde N é o tamanho da tabela hashing.



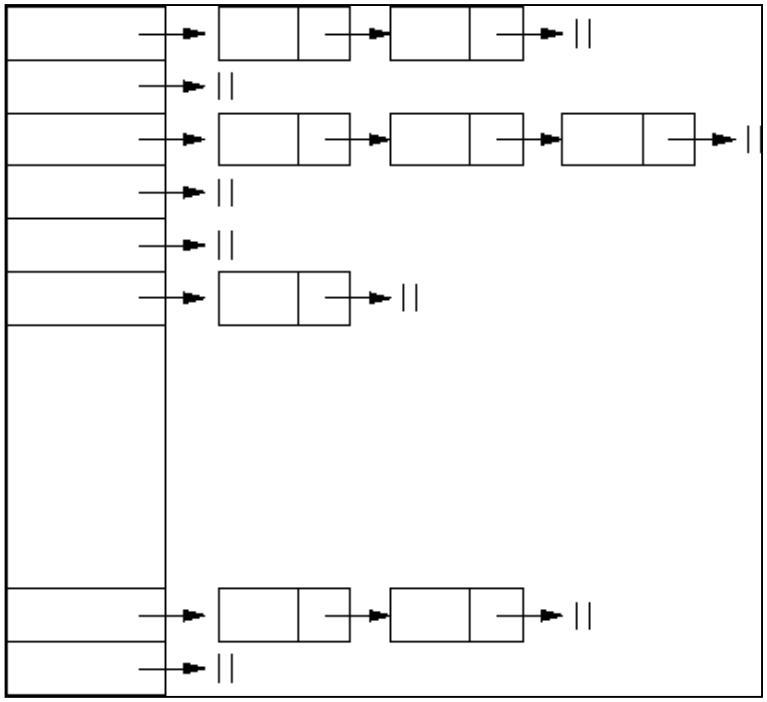
Podemos também fazer com que um endereço possa ser tanto de base quanto de colisão.

Resolução de Colisões

- Encadeamento Aberto ou
  Hashing Fechado/Interno
  - Re-espalhamento Linear (linear probing) ou
    Overflow Progressivo (Busca, Inserção, Remoção)
    - Vantagem: simplicidade.
    - Desvantagem: agrupamento primário; com estrutura cheia a busca fica lenta; dificulta inserções e remoções.
  - Re-espalhamento Quadrático
  - Espalhamento Duplo
- Encadeamento Interno ou
  Overflow Progressivo Encadeado
- Encadeamento ou
  Encadeamento com Área de Overflow Separada ou
  Hashing Aberto/Externo

Resolução de Colisões por Encadeamento com Área de Overflow Separada

- No encadeamento colocamos os elementos que possuem chaves iguais em uma lista encadeada.



Resolução de Colisões por Encadeamento com Área de Overflow Separada

A área de overflow pode estar também no final do arquivo (menor tempo de acesso).

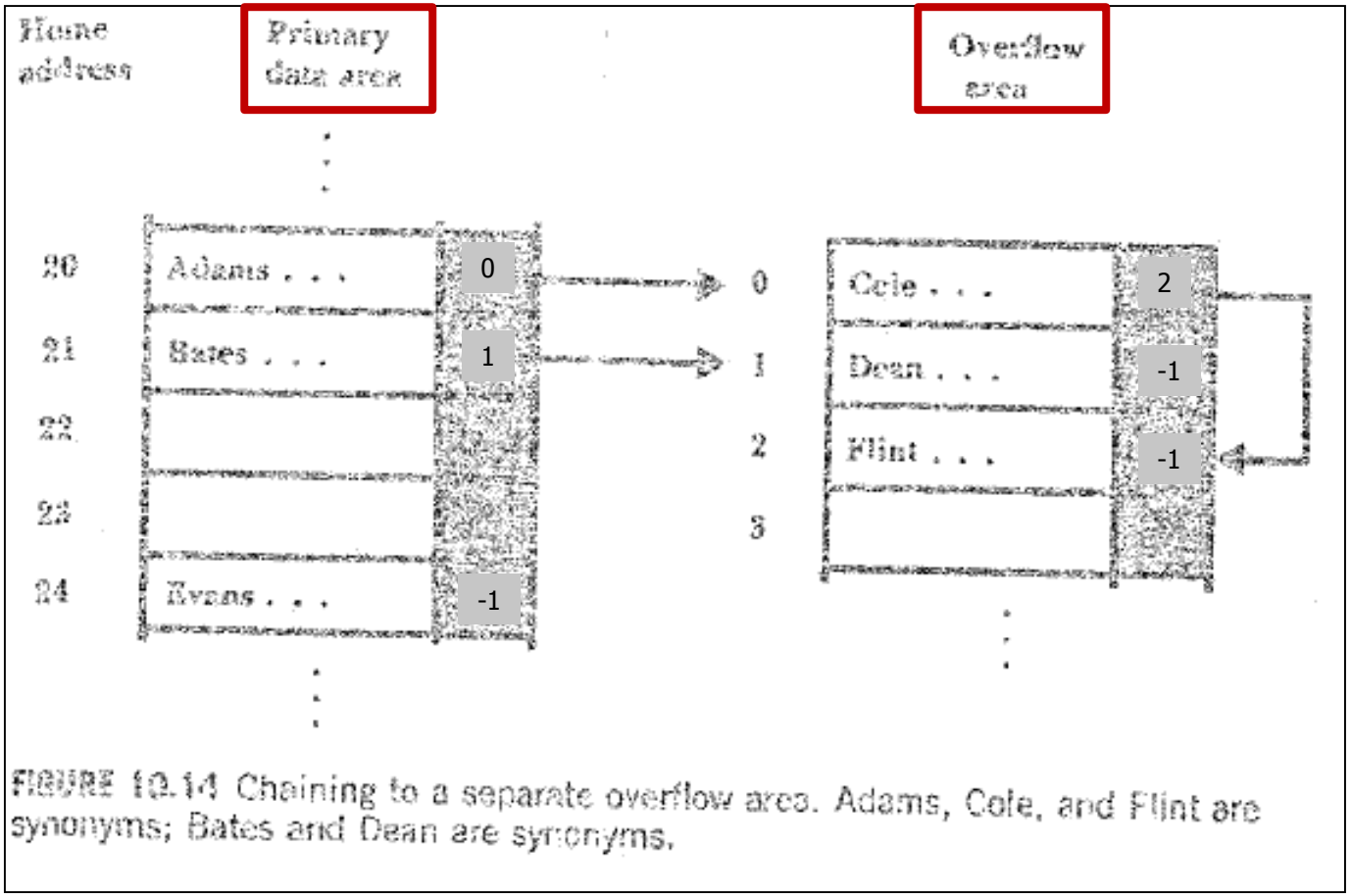
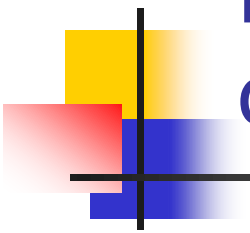


FIGURE 10.14 Chaining to a separate overflow area. Adams, Cole, and Flint are synonyms; Bates and Dean are synonyms.

Resolução de Colisões por Encadeamento com Área de Overflow Separada



- É uma forma de evitar que os registros de overflow ocupem endereços base (caso apresentado).
- Em comparação com o anterior, facilita o processo de remoção e inserção, já que ocupa uma área separada.
- Além disso, a tabela pode receber mais itens mesmo quando um endereço já foi ocupado.
  - Permite armazenar um conjunto de informações de tamanho, potencialmente, ilimitado.

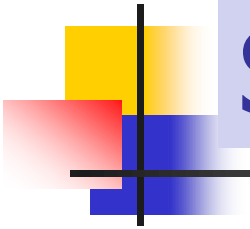
Resolução de Colisões por Encadeamento com Área de Overflow Separada

- Vantagens
  - Mantém todos os endereços base não utilizados disponíveis para inserções futuras.
  - Se a área primária é alocada de forma que o tamanho do cesto seja suficientemente grande para permitir poucos overflows, a área de overflow pode ser um simples arquivo entry-sequenced com cestos de tamanho um. O espaço é alocado somente quando necessário.
  - Resolve o problema de manutenção como, por exemplo, o reajuste depois de cada eliminação.
- Necessário quando a densidade de ocupação é maior que 1 (100%), ou seja, quando existem mais registros do que endereços base.
  - Quando o arquivo começa a crescer e ultrapassa a quantidade de endereços base inicial.

Resolução de Colisões por Encadeamento com Área de Overflow Separada

- Desvantagens
  - Espaço extra para as listas.
  - Listas longas implicam em muito tempo gasto na busca.
    - Se as listas estiverem ordenadas reduz-se o tempo de busca.
      - Entretanto, custo extra com a ordenação.

Scatter Tables



- Quando, ao invés do registro todo, o arquivo na realidade guarda apenas as chaves e sua posição nos arquivos de dados, o hashing gera, na realidade, um índice, e o arquivo relacionado é chamado **scatter table**.
- Nesse caso, o hashing é utilizado como um mecanismo de busca.
- **Vantagem**: suporta registros de tamanho variado!!!

Scatter Tables

O arquivo pode ser implementado de diversas formas: ordenado, entry-sequenced, etc.

Nesse caso, o tempo de **busca** no **índice** para encontrar a chave é **O(1)**!

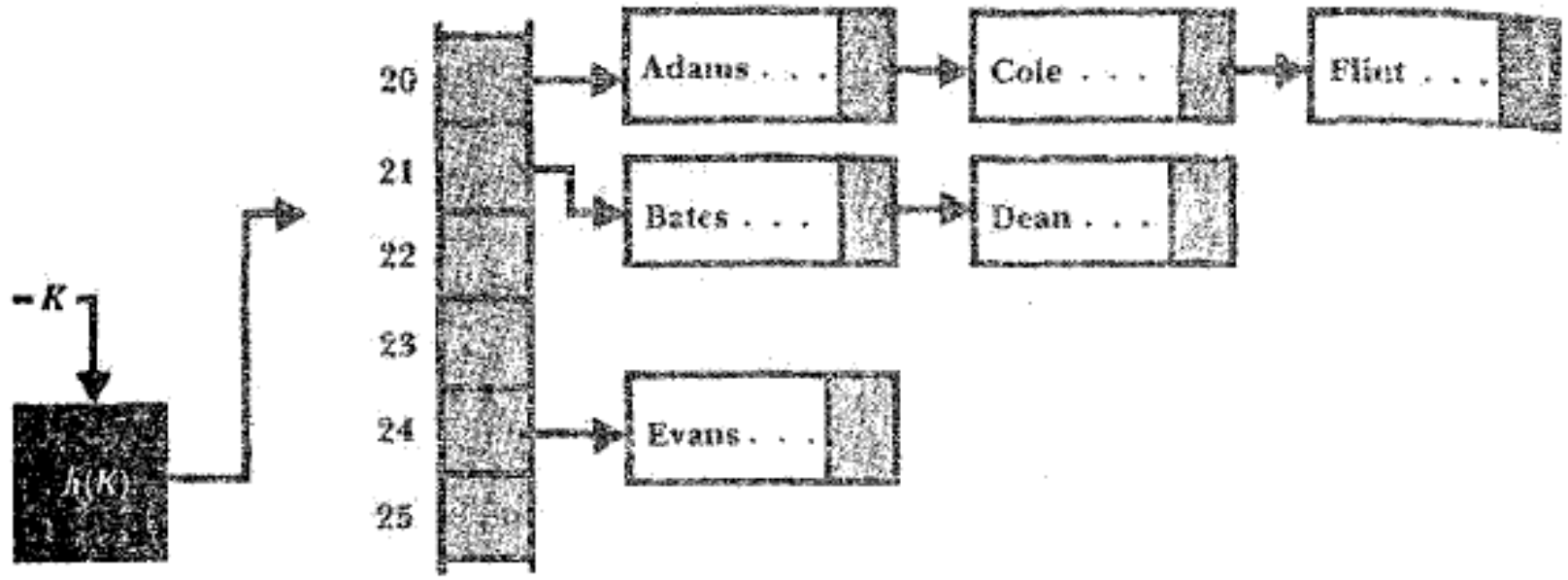
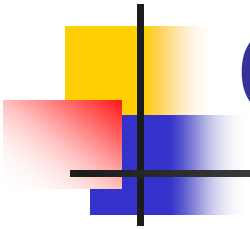
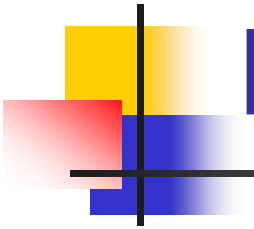


FIGURE 10.15 Example of a scatter table structure. Because the hashed part is an index, the data file may be organized in any way that is appropriate.



Considerações

- **Conclusão:** Para trabalhar com hashing é necessário:
  - Um função de espalhamento;
  - Um mecanismo para tratamento de colisões.
- Desempenho
  - Melhor caso: O(1)
  - Pior caso: O(n)
- Porque os Bancos de Dados implementam, em geral, árvore-B ao invés de hashing?
  - Não permite recuperar/imprimir todos os elementos em ordem de chave nem tampouco outras operações que exijam sequência dos dados.
  - Não permite operações do tipo recuperar o elemento com a maior ou menor chave, nem o antecessor e o sucessor de uma determinada chave.



Bibliografia

- FOLK, M.; ZOELLICK, B. File Structures, Second Edition. Addison-Wesley, 1992.
- FOLK, M.; ZOELLICK, B.; RICCARDI, G. File Structures: An Object-Oriented Approach Using C++. Third Edition. Addison-Wesley, 1998.
- Pereira, S. L. Estruturas de dados fundamentais : conceitos e aplicações. 2000.
- Ziviani, N. Projeto de algoritmos: com implementações em Pascal e C. 2004.
- Tenenbaum, A. M.; Langsan, Y.; Augenstein, M. Estruturas de dados usando C. 1995.
- CORMEN, T. H.; LEISERSON, C. E.; RIVEST, R. L. Introduction to Algorithms. The MIT Press. Mc Graw-Hill, 1990.