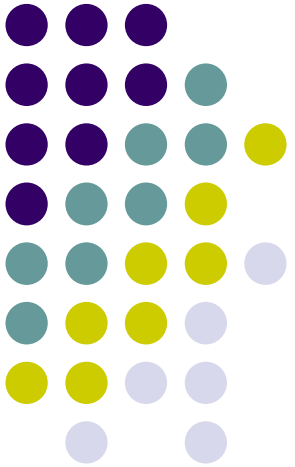
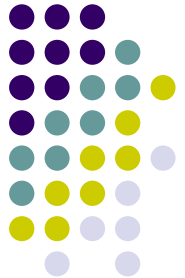
Hashing

Prof. Leandro C. Fernandes

Estruturas de Dados

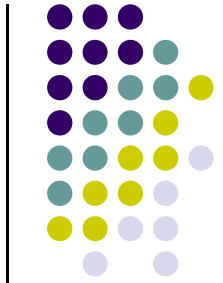


Hashing



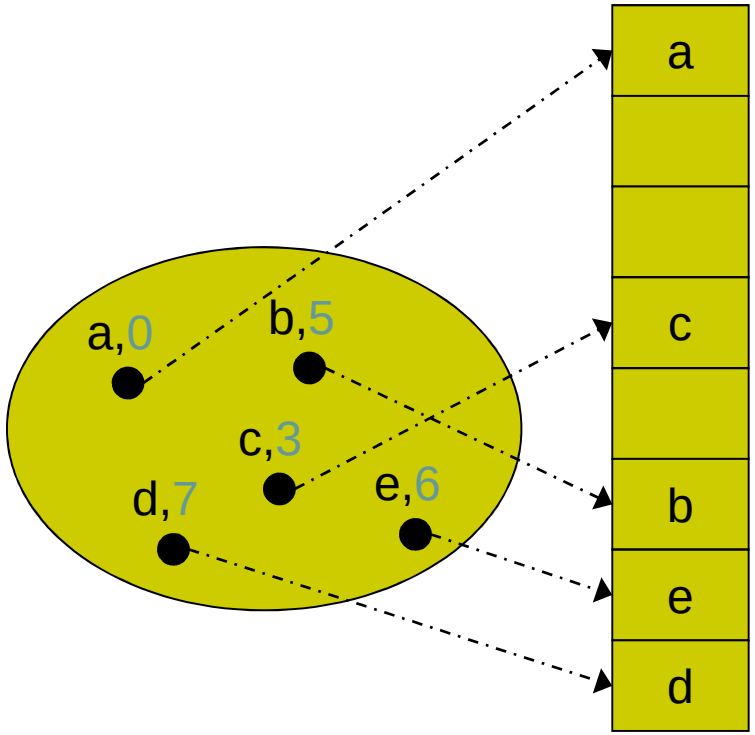
- “Hashing” é uma técnica que busca realizar as operações de inserção, remoção e busca em tempo constante, i.e., O(1).
- As outras estruturas que estudamos organizavam os dados segundo o valor relativo de cada chave entre as demais.
- A *tabela hash* considera somente o valor absoluto da chave, interpretando-o como um valor numérico utilizado para a indexação da informação.

A idéia de Hashing

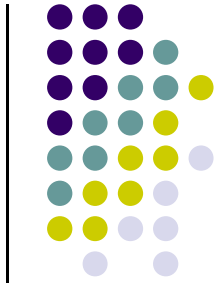


- Suponha que, para cada elemento do conjunto, seja possível extrair um valor inteiro diferente e, assim, utilizá-lo para determinar em qual posição esse elemento seria armazenado num vetor.

- A função que associa a cada elemento de um conjunto *U* um número que sirva de índice em uma tabela (armazenamento linear) é chamada *função hash*.

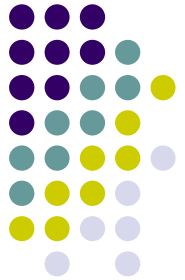


Aplicações



- O acesso a dados em tempo constante é característica muito importante quando se trabalha com armazenamento secundário em disco (onde o acesso a um determinado endereço é bastante lento).
- Algumas aplicações que são beneficiadas pelo seu uso direto são:
  - Bancos de dados;
  - Dicionários;
  - Tabelas de palavras reservadas de um compilador;
  - ...

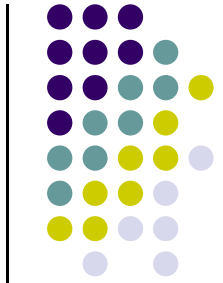
Operações em Tabela Hash



- Consiste nas operações típicas de uma estrutura de armazenamento, ou seja:
  - Inserção;
  - Remoção; e
  - Busca.

  ... além da função responsável por viabilizar todas as operações anteriores:
  - Função de Hash.

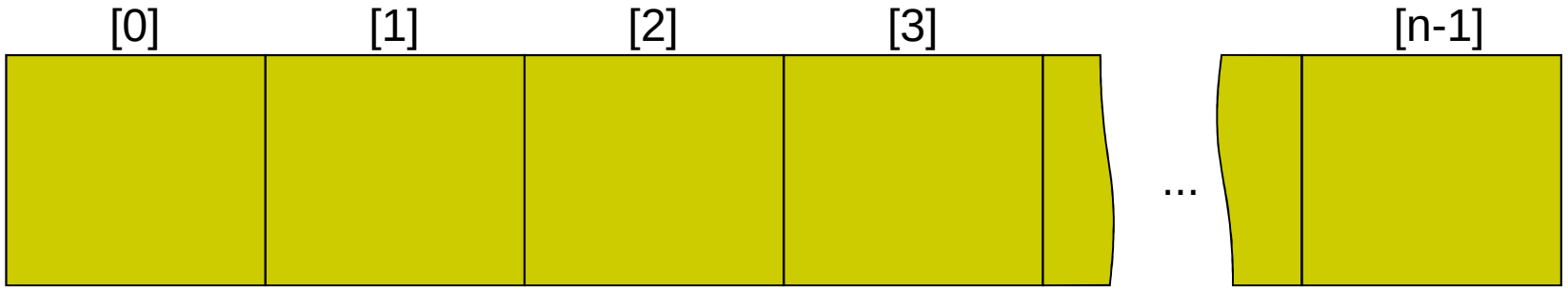
A função Hash



- Uma *função hash* adequada deve satisfazer as seguintes condições:
  - Ser simples de calcular
  - Assegurar que elementos distintos possuam índices distintos.
  - Gerar uma distribuição equilibrada para os elementos dentro da estrutura.

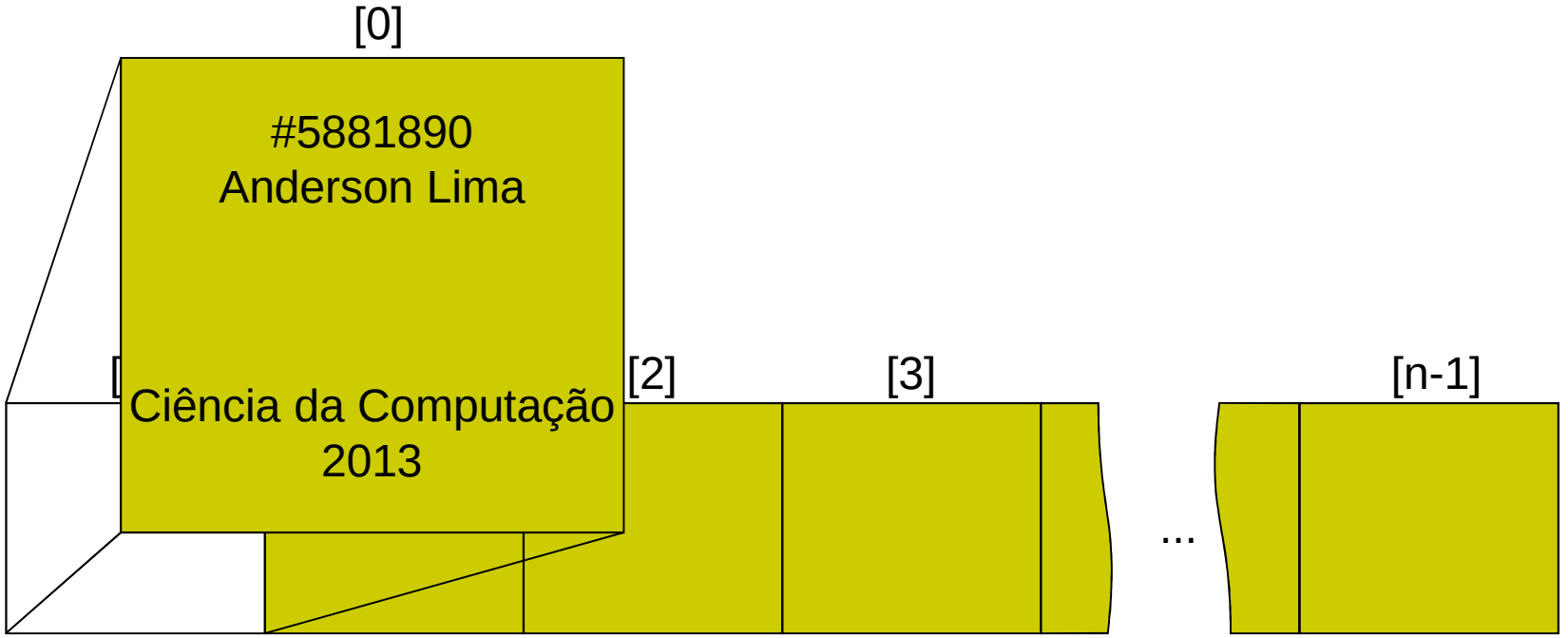
Exemplo

- Suponha um vetor para armazenar N registros de alunos. Cada registro contém o RA – Registro de Aluno (chave) e as demais informações do aluno.



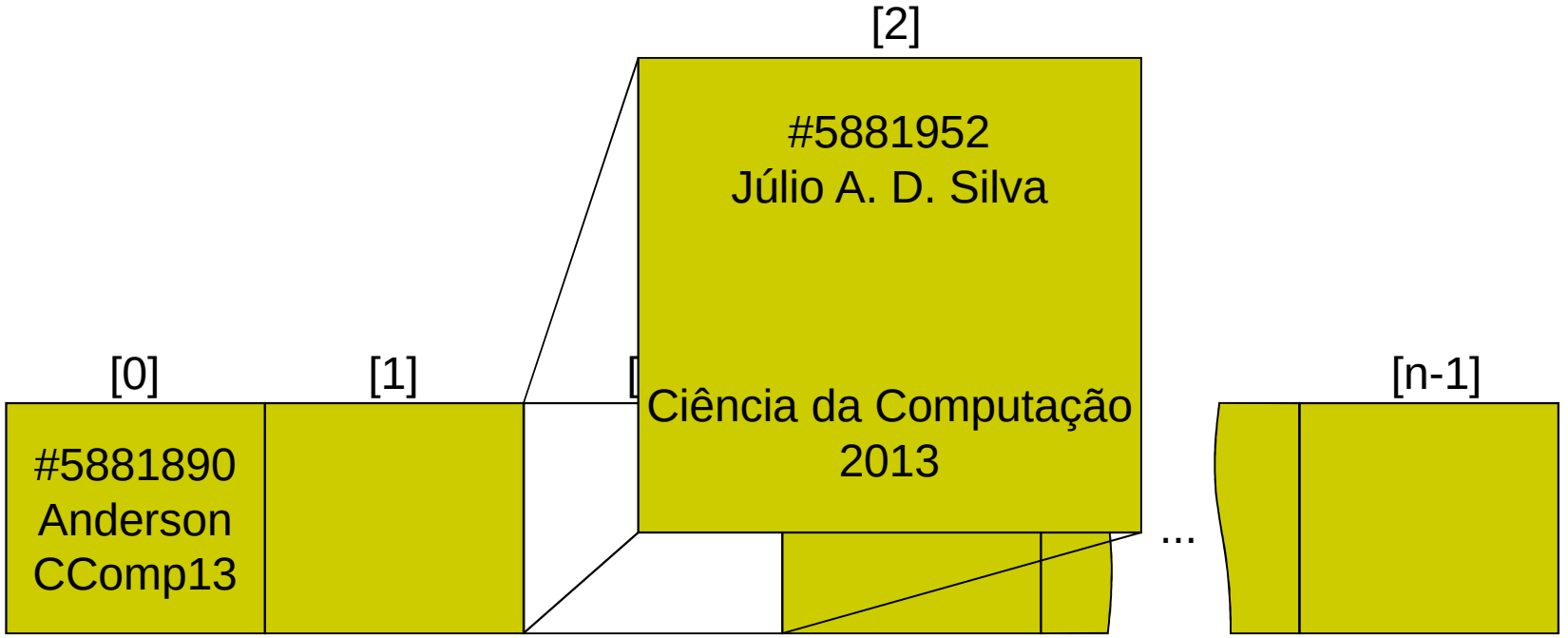
Exemplo

- Inserindo um novo dado: #5881890
- Hash(#5881890) = 0



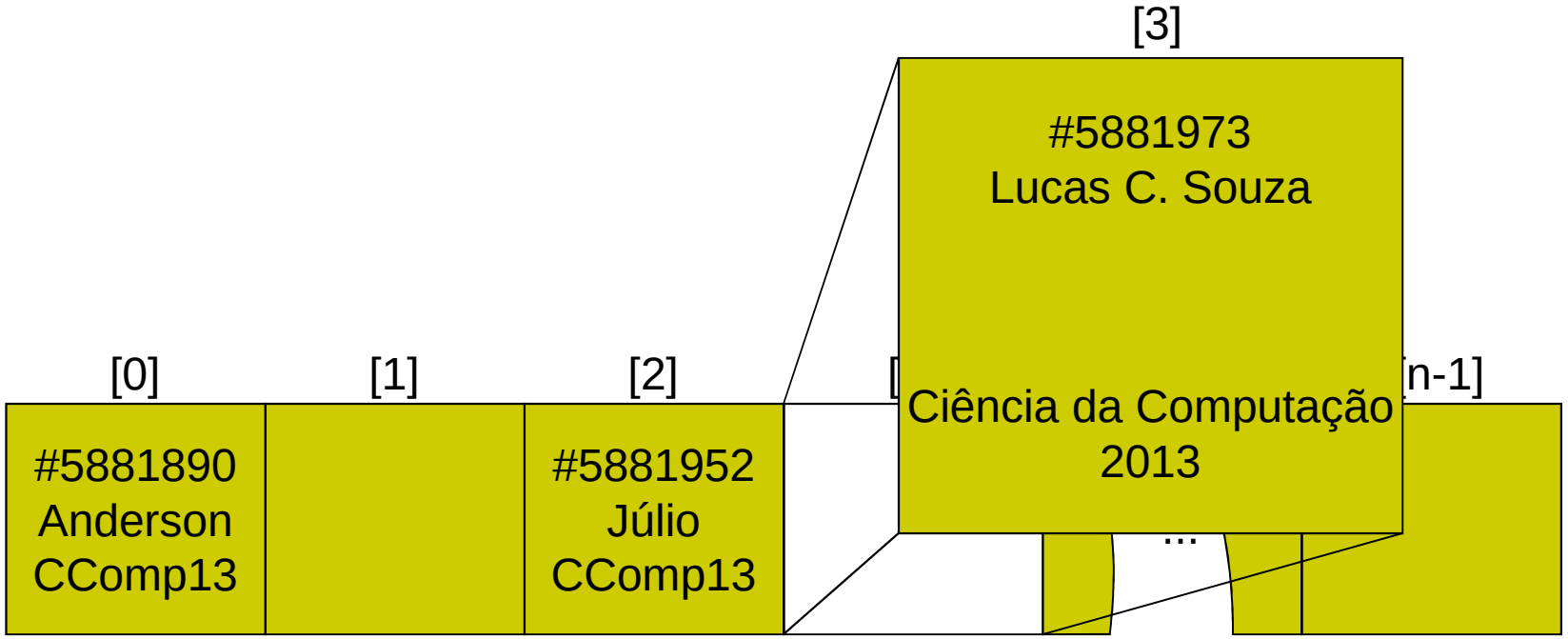
Exemplo

- Inserindo um novo dado: #5881952
- Hash(#5881952) = 2



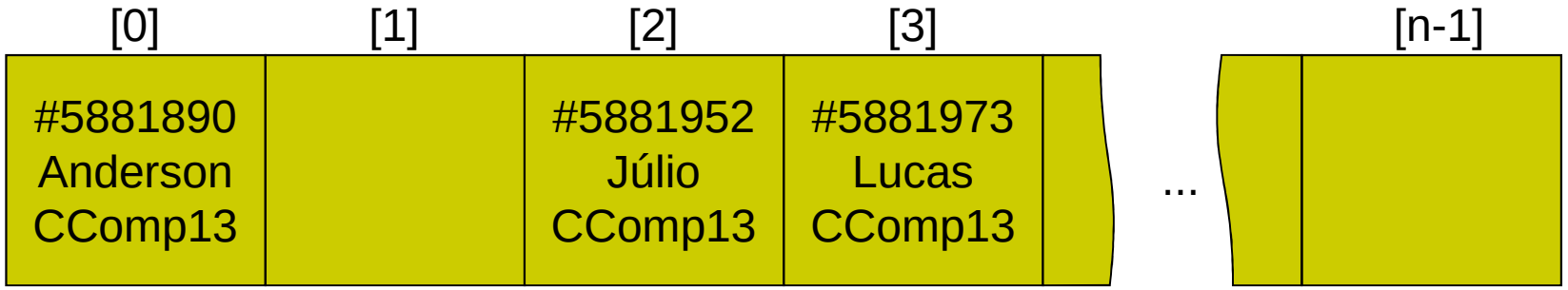
Exemplo

- Inserindo um novo dado: #5881973
- Hash(#5881973) = 3



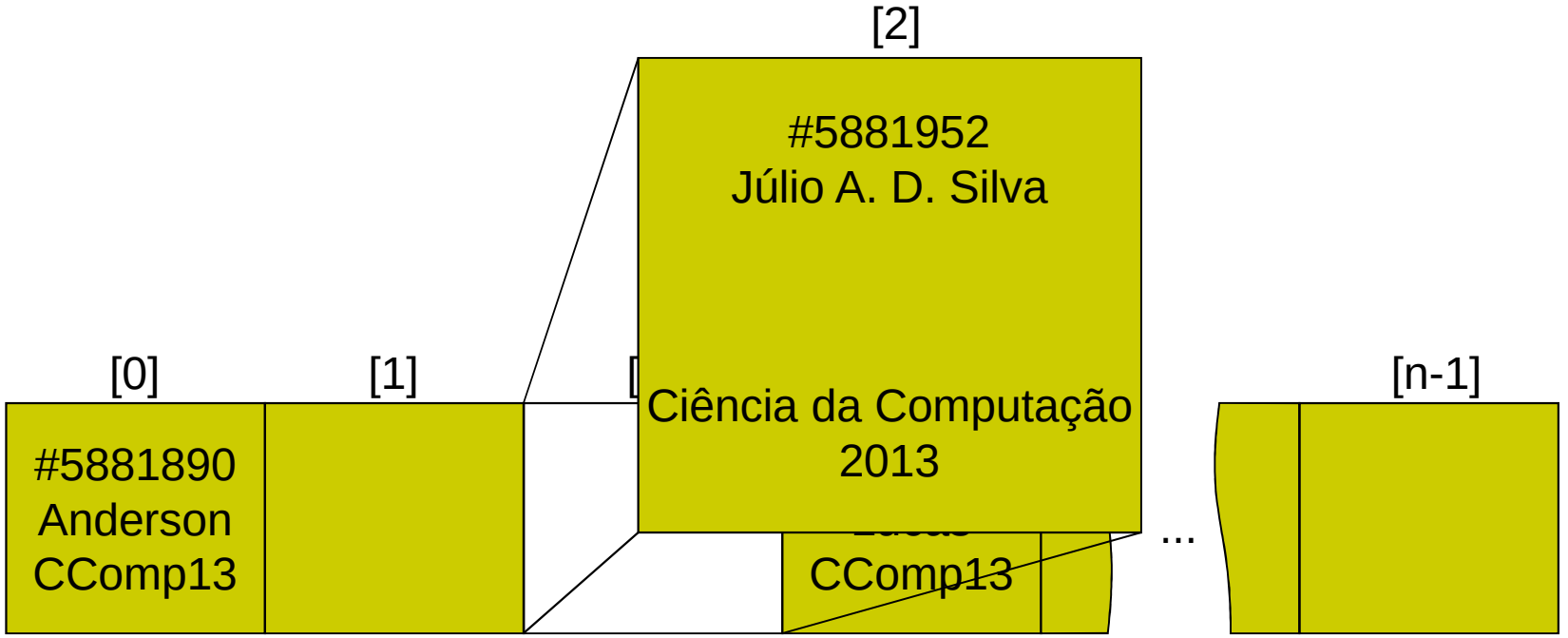
Exemplo

- Buscar pelo dado: #5881952
- Acessar: vetor[Hash(#5881952)]

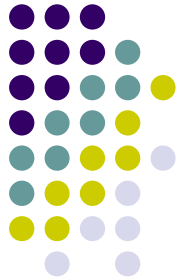


Exemplo

- Buscar pelo dado: #5881952
- Acessar: vetor[2]

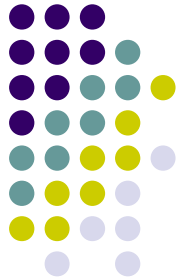


Escolhendo a função de Hashing



- Qual função devo utilizar?
  - Método da Divisão
  - Método da Dobra
  - Método da Multiplicação
  - Hashing universal
- Será que nunca vamos produzir chaves iguais para elementos distintos?
  - Tratamento de colisões

Método da Divisão



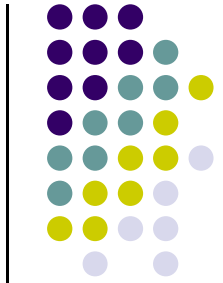
- Suponha que os elementos *x* que devem ser armazenados sejam números inteiros, uma função hash que pode ser utilizada para espalhar tais números em um array (tabela hash) é:

$$h(x) = x \bmod TableSize$$

onde *tablesize* é o tamanho do vetor.

Obs.: Utilizar *tablesize* como sendo um número primo minimiza o problema de elementos distintos podem receber o mesmo índice.

Implementação da Função de Hashing (Divisão)

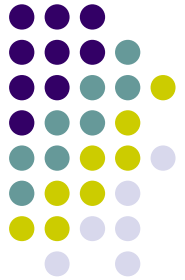


```
int hash(char *a, int stringsize) {
  int hashval, j;

  hashval = (int) a[0];
  for(j=1; j < stringsize; j++)
    hashval += (int) a[j];

  /* supondo que tablesize é global */
  return (hashval % tablesize);
}
```

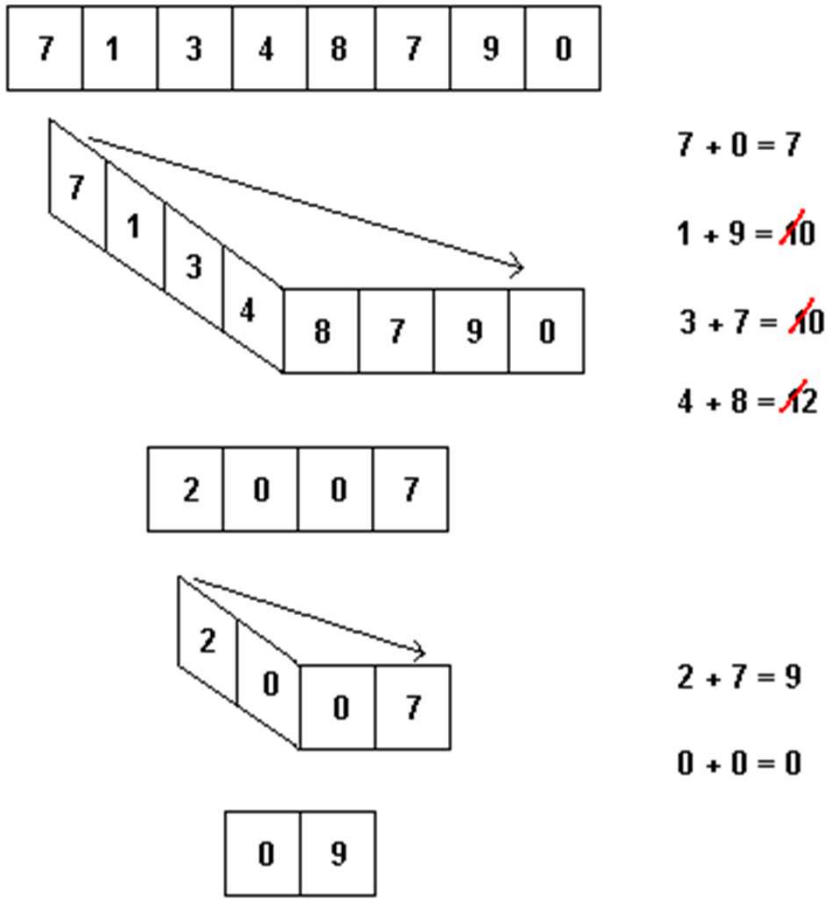
Método da Divisão



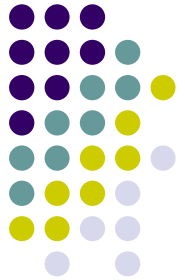
- Um dos problemas é que se *tablesize* é muito grande a função não distribui bem os elementos. Por exemplo:
  - Suponha *tablesize = 10.007* (número primo) e que todas as strings possuam no máximo oito caracteres.
  - Desde que o código ascii de um caractere é no máximo 127, a função acima assumirá valores entre 0 e 1016, não gerando uma distribuição equilibrada.
  - Uma dentre as possíveis soluções para esse problema seria elevar o valor da soma dos caracteres *ascii* ao quadrado e só então dividi-lo pelo tamanho da tabela hash.

Método da Dobra

- As chaves são interpretadas como uma seqüência de dígitos escritos num pedaço de papel.
- O método consiste em "dobrar" esse papel, de maneira que os dígitos se superponham sem levar em consideração o "vai um".
- O processo é repetido até que os dígitos formem um número menor que o tamanho da tabela hash.

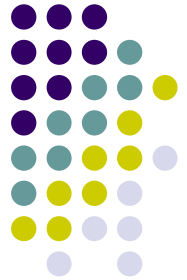


Método da Dobra



- É importante destacar que o método da dobra também pode ser usado com números binários. Nesse caso, ao invés da soma, deve-se realizar uma operação de "ou exclusivo", pois operações de "e" e de "ou" tendem a produzir endereços-base concentrados no final ou no início da tabela.

Método da Multiplicação

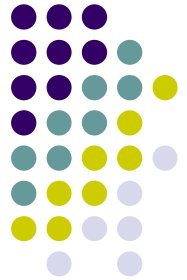


- Assuma que todas as chaves sejam inteiras, i.e., $m = 2^r$, e nosso computador tenha palavras de w-bit comprimento.
- Defina:

$$h(k) = (A * k \bmod 2^w) rsh (w - r)$$

onde rsh é a operação binária de shift a direita e A é um inteiro ímpar do intervalo $2^{w-1} < A < 2^w$.

Método da Multiplicação



- A chave *k* pode ser multiplicada por ela mesma ou por uma constante *A*, e ter o seu resultado armazenado em uma palavra de memória de comprimento igual a *w* bits.
- Considere que o número de bits necessários para endereçar uma chave na tabela hash seja ***r.***
- Dessa forma, para compor o endereço-base de uma chave, descartam-se os bits excessivos da extrema direita e da extrema esquerda da palavra de *S* bits calculada.

Exemplo

- Suponha a chave k = 1100
- Calculando:

  h(k) = k * k

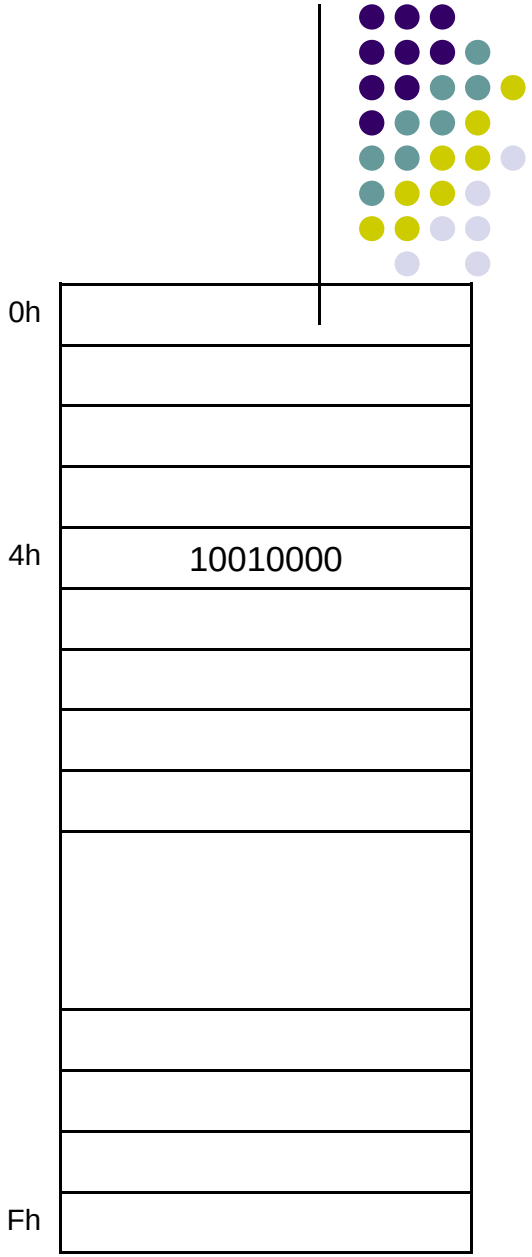
  = 1100 * 1100

  = 10010000

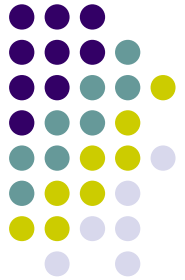
  h(k) = 0100 = 4d

  Desprezando os bits das extremidades, e considerando apenas o necessário para o endereçamento (i.e. 4 bits)

Obs.: Essa técnica é conhecida como "*meio do quadrado*".

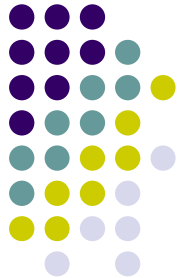


Método da Multiplicação



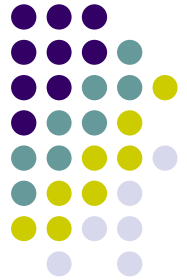
- Questões de desempenho:
  - A multiplicação pelo módulo de 2^w é rápida se comparada com a divisão.
  - O operador rsh é muito rápido.
- Distribuição adequada das chaves:
  - Não utilize um A muito próximo de 2^{w-1} ou 2^w, pois isso implica em um concentração das chaves nos extremos do vetor.

Hashing Universal



- Qualquer função hash está sujeita ao problema de criar chaves iguais para elementos distintos.
- Dependendo da entrada, algumas funções hash podem espalhar os elementos de forma mais ou menos equilibrada que outras.
- A idéia é, se pudermos utilizar diferentes funções hash teremos, em média, um espalhamento mais equilibrado.
- A estratégia é de escolher aleatoriamente (em tempo de execução) uma função hash a partir de um conjunto de funções cuidadosamente desenhado.

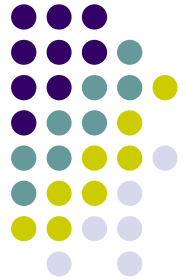
Hashing Universal



- Uma família de funções pode ser gerada da sequinte forma:
  - Suponha *tablesize* um número primo.
  - Decomponha a chave *k* em *r*+1 bytes (para o caso de strings, decompomos em caracteres ou substrings).
  - Seja a = {a_0, a_1, a_2, ..., a_r} uma seqüência de elementos escolhida randomicamente a partir de 0, 1, 2, ... , *m*-1.
- Definimos uma família de funções hashing como:

$$h_a(x) = \sum_{i=0}^{r} a_i * x_i \bmod TableSize$$

Densidade de Ocupação



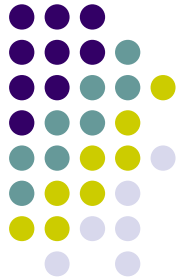
Def.: Razão entre o número de registros a serem armazenados (r) e o número de espaços de endereçamento disponíveis (N, assumindo um registro por endereço).

$$\alpha = \frac{r}{N}$$

- A densidade de ocupação proporciona uma medida da quantidade de espaço do arquivo que está sendo de fato utilizada, e é o único valor necessário para avaliar o desempenho de um espalhamento.

Obs.: Quanto maior a densidade, maior a chance de ocorrerem colisões quando um novo registro precisar ser adicionado!

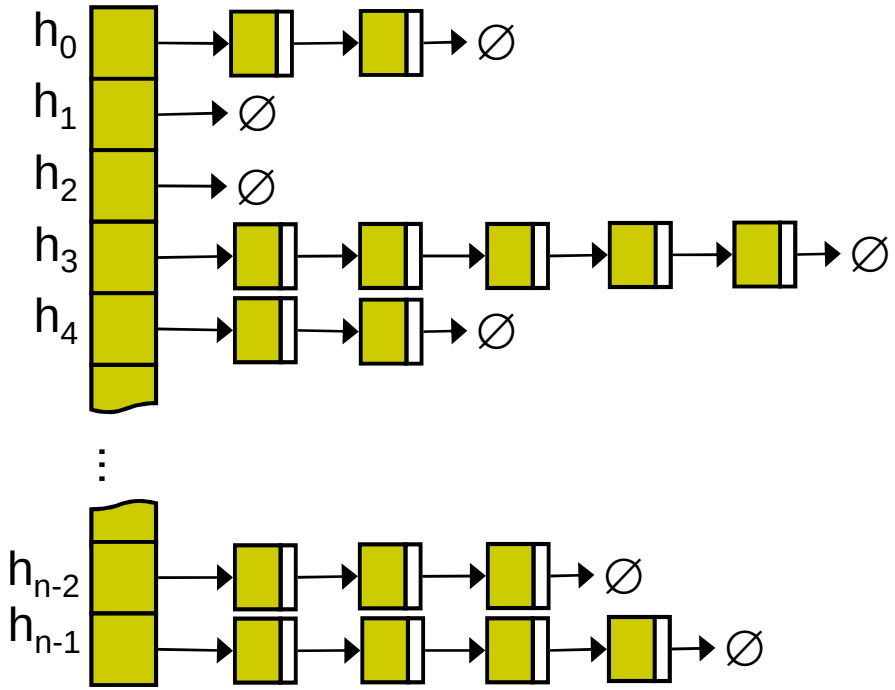
Colisões



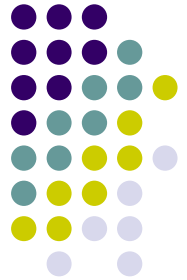
- Por mais que se tente encontrar uma função hash eficiente, em aplicações práticas é difícil conseguir evitar o problema de colisão de chaves.
- Existem várias formas de se resolver o problema de colisões:
  - Encadeamento Separado ou Externo (Separate Chaining)
  - Encadeamento Interno
  - Endereçamento Aberto (Open Addressing)
    - Espalhamento linear
    - Espalhamento quadrático
    - Espalhamento duplo

Encadeamento Separado ou Externo (Separate Chaining)

- No encadeamento separado resolvemos o problema das colisões colocando os elementos que possuem chaves iguais numa lista encadeada.



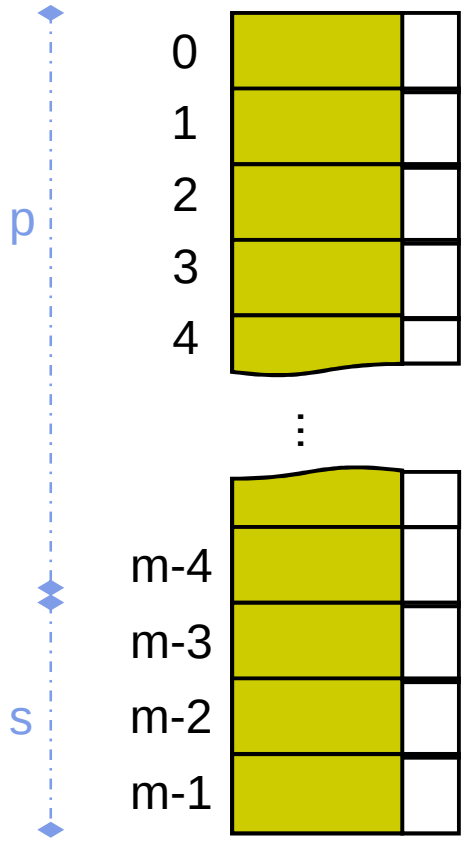
Encadeamento Separado ou Externo (Separate Chaining)



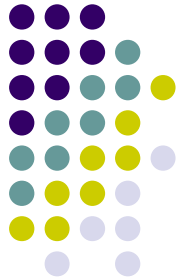
- Utilizando esta estratégia, não é difícil perceber que no melhor caso o tempo para inserir um novo elemento na tabela é O(1).
- Procurar por um elemento, leva tempo proporcional ao tamanho da lista armazenada em cada slot.
- Portanto, o tempo necessário para a recuperação de uma informação no pior caso é de O(*n*), onde *n* corresponde a qtde de elementos armazenados na lista.

Encadeamento Interno

- O encadeamento interno prevê a divisão da tabela T em duas zonas:
  - De endereços-base (de tamanho p); e
  - Reservada para colisões (de tamanho s).
- Assim, o tamanho da tabela hash é dado por:
  - M = P + S



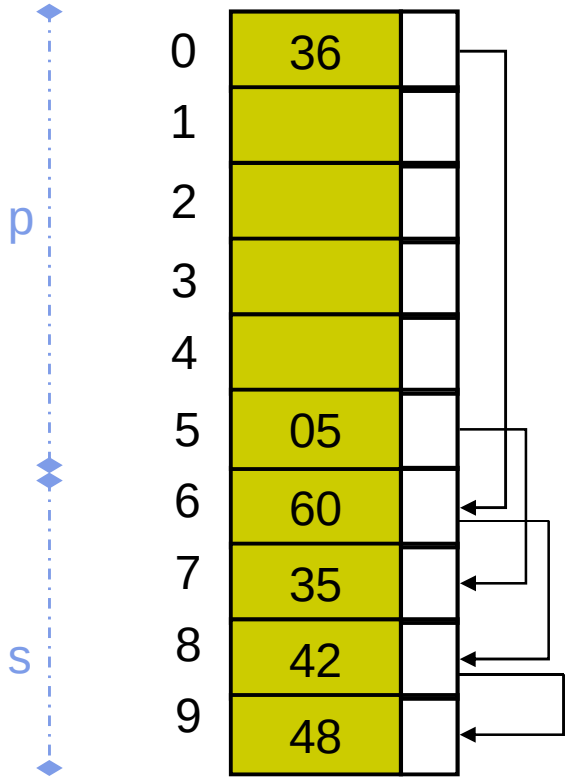
Encadeamento Interno



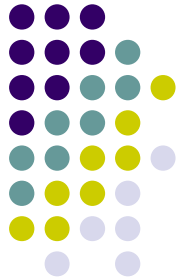
- A técnica recebe esse nome porque é um método que emprega listas encadeadas que compartilham o mesmo espaço de memória que a tabela de espalhamento.
- Os valores resultantes da função de hashing endereçam as posições correspondentes a zona de endereços-bases.
- Caso ocorra colisão, o elemento deverá ser armazenado na zona de colisão, correspondendo ao setor S da tabela

Encadeamento Interno

- Exemplo:
  - Suponha as entradas:
    36, 60, 05, 35, 42, 48
- Função de hash:
  - h(x) = x % 6
- O que aconteceria com a inclusão de números com endereço base iguais a 0 ou 5?
  - Provocará overflow, apesar de a zona de endereços-base ainda apresentar posições vazias.



Encadeamento Interno

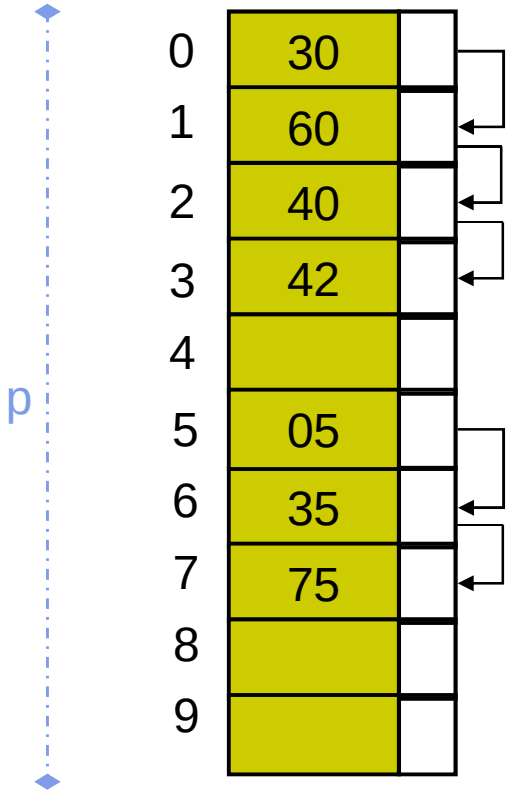


- O que fazer para evitar esta situação?
  - Se aumentarmos a zona de colisões, a eficiência da tabela hash diminui.
  - No caso limite, temos que o tempo médio de busca é *O*(n).
- Podemos também fazer com que um endereço possa ser tanto de base quanto de colisão.

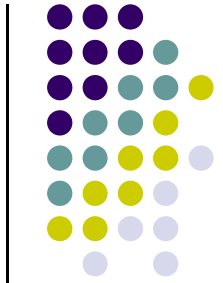
Encadeamento Interno

- Exemplo:
  - Suponha as entradas:

    30, 60, 05, 35, 40, 42, 75
- Função de hash:
  - h(x) = x % 10
- Onde se encontra a chave: #42 ?

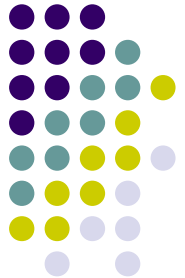


Encadeamento Interno



- Problemas:
  - Colisões secundárias
    - colisões provenientes da coincidência de endereços para chaves que não são sinônimas (ex. #42).
  - Remoção de um elemento
    - É preciso tomar cuidado neste procedimento para que chamada de busca e remoção futuras não retornem falsos resultados.
      - Pq isso seria um problema??

Endereçamento Aberto (Open Addressing)

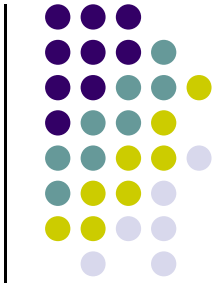


- Neste tipo de endereçamento todos os elementos são armazenados na própria tabela hash, isto é, não existem listas nem elementos armazenados fora da tabela.
- A vantagem é que a quantidade de memória utilizada para armazenar ponteiros é utilizada para aumentar o tamanho da tabela, possibilitando menos colisões e aumentando a velocidade de recuperação das informações.

Endereçamento Aberto (Open Addressing)

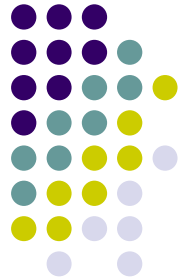
- Para inserir um novo elemento, calculamos a função de hash e caso haja colisão, examinamos sucessivamente a tabela até encontrarmos uma posição vazio.
- A busca por um elemento deve seguir a mesma seqüência de slots que o algoritmo de inserção.
- Para remover um elemento de um slot *i* devemos marcar *i* como REMOVIDO ao invés de VAZIO.
  - Não podemos simplesmente marcar tal slot como VAZIO, pois isto tornaria impossível recuperar qualquer elemento para o qual a inserção encontrou o slot *i* ocupado em algum momento.

Endereçamento Aberto (Open Addressing)



- Desempenho:
  - Após um certo número de inserções e eliminações, o desempenho das operações da tabela hash sofre uma queda substancial devido à falta de organização na estrutura.
- Sugestões:
  - pode-se reorganizar o arquivo armazenado; ou ainda
  - usar uma técnica diferente para tratamento de colisões.

Implementação: Open Addressing

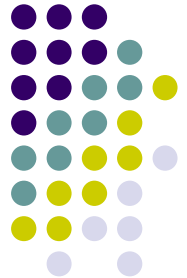


```
#define VAZIO     0
#define REMOVIDO  1
#define OCUPADO   2

typedef struct no no_hash;

struct no {
    tDado data;
    int state;
};
```

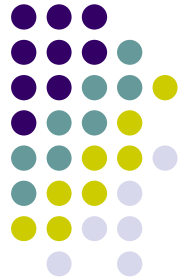
Implementação: Open Addressing



```
/* Cria a tabela hash */
no_hash *Cria_Hash(int m) {
  no_hash *temp;
  int i;

  temp = (no_hash *) malloc( m*sizeof(no_hash) );
  if (temp != NULL) {
    for(i=0; i<m; i++)
      temp[i].state = VAZIO;
    return temp;
  }
  else
    return NULL;
}
```

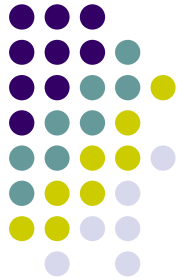
Implementação: Open Addressing



```
/* Calcula a função de hashing */
int hash(int k, int m, int i) {
    return ((k+i)% m);
}
```

- Os parâmetros da função de hash são:
  - K: chave
  - M: tamanho da tabela
  - i: contador de colisões

Implementação: Open Addressing

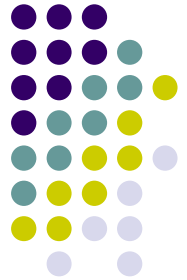


```
/* Insere um elemento k na tabela T de tamanho m */
int Insere_Hash(no_hash *T, int m, int k) {
  int j, i = 0;
  do {
    j = hash(k,m,i);
    if (T[j].state == VAZIO || T[j].state == REMOVIDO) {
       T[j].data = k;
       T[j].state = OCUPADO;
       return 1;
    }
    else
      i++;
  } while (i < m);
  return 0;
}
```

Implementação: Open Addressing

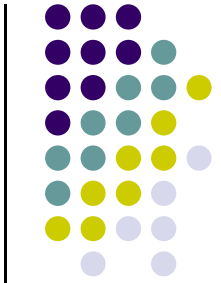
```
/* Busca um elemento k na tabela T de tamanho m */
int Busca_Hash(no_hash *T, int m, int k, int i) {
  int j;
  if (i < m) {
    j = hash(k,m,i);
    if (T[j].state == VAZIO)
       return -1;
    else
       if (T[j].state == REMOVIDO)
         return Busca_Hash(T,m,k,i+1);
       else
         if (T[j].data == k)
           return j;
         else
           return Busca_Hash(T,m,k,i+1);
    }
    return -1;
}
```

Implementação: Open Addressing



```
/* Remove um elemento k na tabela T
   de tamanho m */
int Remove_Hash(no_hash *T, int m, int k) {
  int i;
  i = Busca_Hash(T,m,k,0);
  if (i == -1)
    return -1;
  else {
    T[i].state = REMOVIDO;
    return 1;
  }
}
```

Espalhamento linear



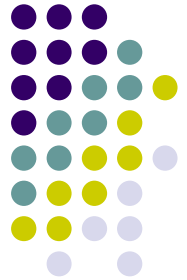
- O método utiliza a seguinte função hash:

$$h(x,i) = \left(h'(x) + i\right) \bmod TableSize$$

  supondo i = 0, ..., *TableSize*-1

- Esta função hash tenta espalhar os elementos seqüencialmente a partir de h(x).
- O espalhamento linear apresenta um problema conhecido como “agrupamento primário":
  - quando a tabela começa a ficar cheia, o tempo para incluir um novo elemento aumenta.

Espalhamento quadrático

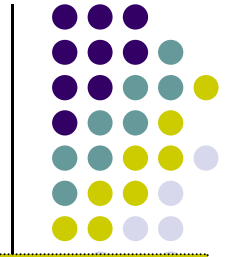


- O método utiliza a seguinte função hash:

$$h(x,i) = \left(h'(x) + c_1 i + c_2 i^2\right) \bmod TableSize$$

  supondo i = 0, ..., *TableSize*-1
- Esta função hash tenta espalhar os elementos segundo um comportamento quadrático de *i*. Note que para uma utilização completa da tabela precisamos restringir c_1, c_2 e *TableSize*.
- O espalhamento quadrático possui um problema conhecido como “agrupamento secundário"
  - Se $h(x_1,0) = h(x_2,0) \rightarrow h(x_1,i) = h(x_2,i)$

Espalhamento duplo



$h_1(x)$ = x mod *TableSize*
$h_2(x)$ = 1 + (*x* mod M)

Onde M é um pouco menor que *TableSize*

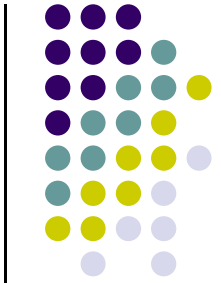
- O método utiliza a seguinte função

$$h(x,i) = \left(h_1(x) + i * h_2(x)\right) \bmod TableSize$$

onde $h_1(x)$ e $h_2(x)$ são funções hash auxiliares.

- Os elementos são distribuídos conforme $h_1(x)$, mas havendo colisão, os novos elementos são espalhados segundo $i * h_2(x)$.
- Espalhamento duplo é um dos melhores métodos para endereçamento aberto, pois as permutações produzidas são semelhantes às permutações aleatórias.

Métodos de espalhamento



- Examinar as chaves buscando um certo padrão.
  - Por exemplo, em chaves numéricas: os números de identificação dos funcionários de uma empresa podem estar ordenados de acordo com a ordem de entrada dos funcionários na empresa. Se uma parte das chaves mostrar um padrão, pode-se usar um função que extrai esta parte, e pode-se inclusive ter uma função uniforme neste caso.
- Separar partes da chave.
  - pode-se extrair dígitos de uma parte da chave e somar estes dígitos apenas. Este método destrói eventuais padrões nas chaves originais, mas em algumas circunstâncias pode preservar a separação entre certos subconjuntos das chaves que se espalham naturalmente.
- Dividir a chave por um número primo.
- Elevar a chave ao quadrado e considerar os dígitos intermediários do resultado.
- Mudar de base e dividir pelo espaço de endereçamento (considerando o módulo)

Obs.: *Os 3 primeiros métodos tentam explorar a ordem natural que pode existir entre as chaves. Os 2 últimos tentam explorar a aleatoriedade das chaves.*