Estruturas de Dados Avançadas (INF1010)

Tabelas de Dispersão

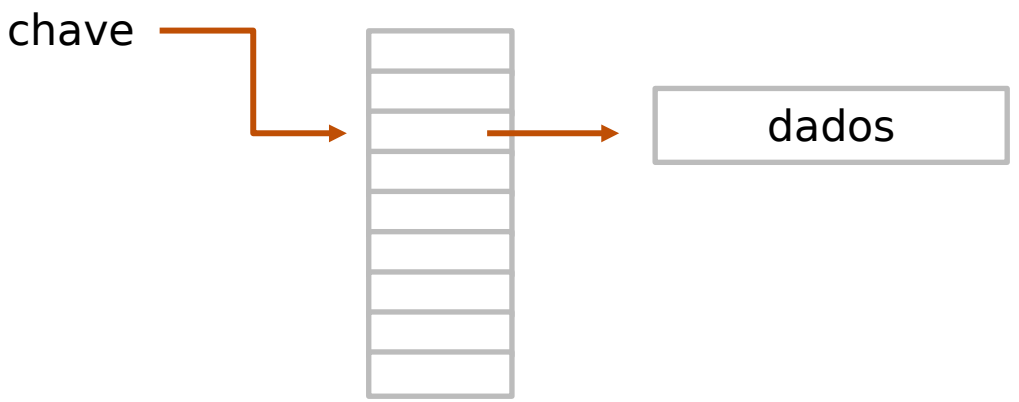
Tabelas de Dispersão (*Hash Tables*)

Uma outra implementação para **Mapa** (chave → valor)

Buscar elemento com complexidade constante O(1)

- estratégia: uso de *array*
- tradeoff: memória x tempo de acesso

acesso direto

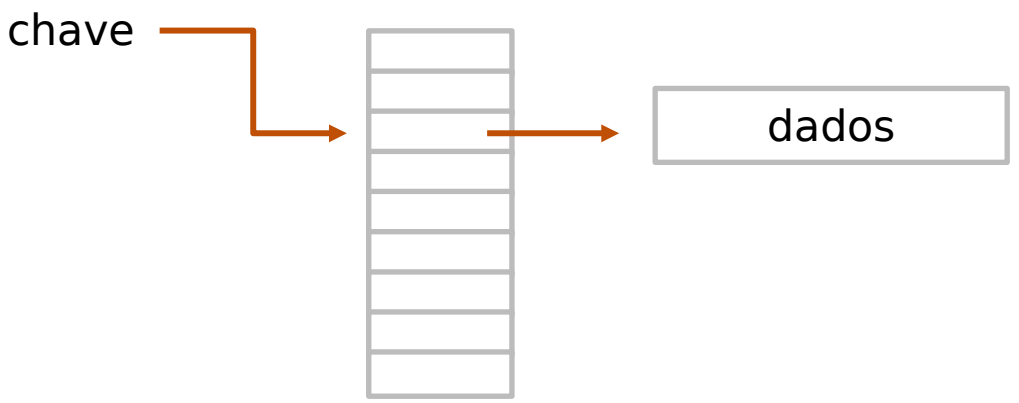


Acesso Direto

E se:

- as chaves não formam um conjunto contínuo de inteiros?
- temos muitas chaves?
- as chaves nem são inteiros?

acesso direto



Tabelas de Dispersão: ideia

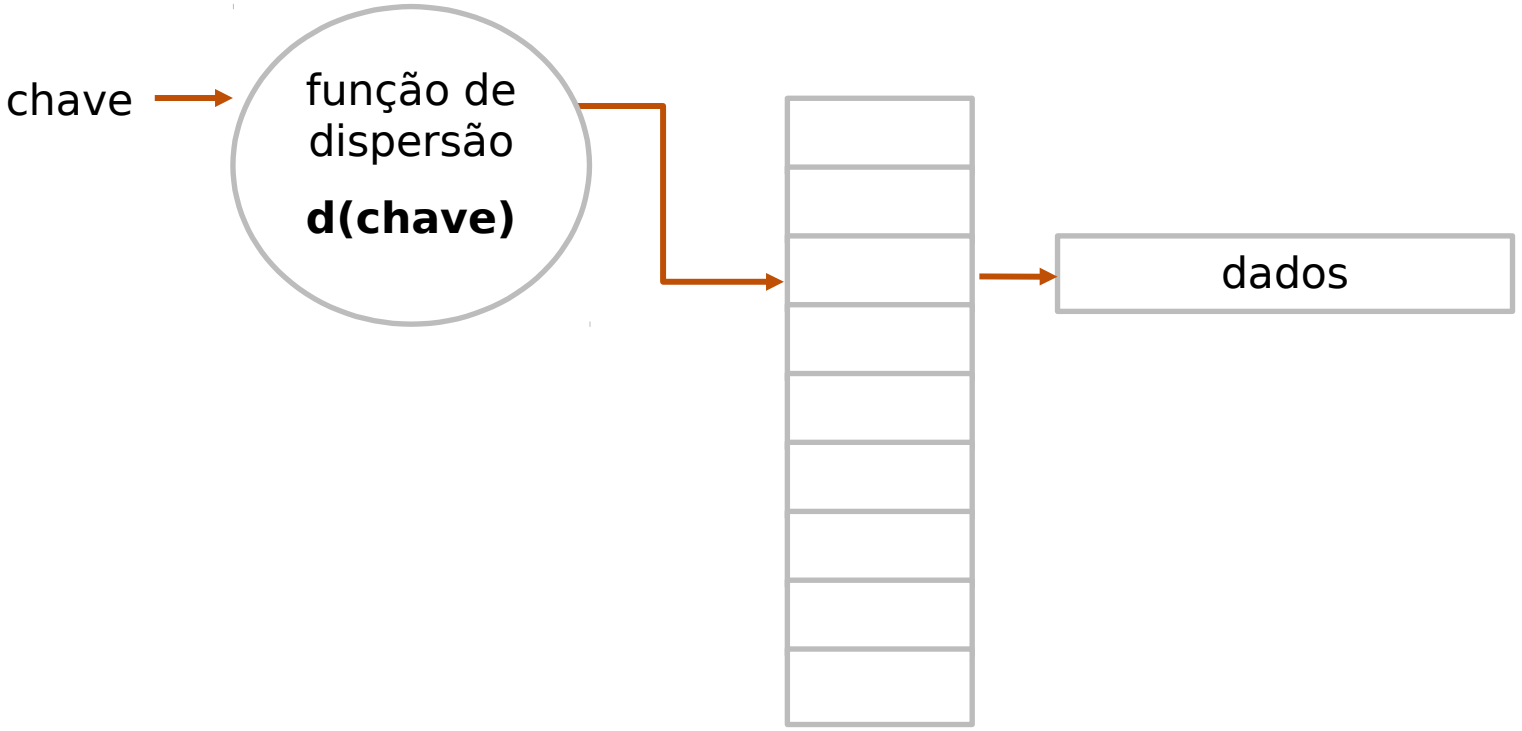
Mapeamento de chaves a posições do *array*

Tabelas de Dispersão: ideia

Mapeamento de chaves a posições do *array*

Funções de Dispersão

Função de dispersão (função de *hash*) mapeia uma chave em um índice

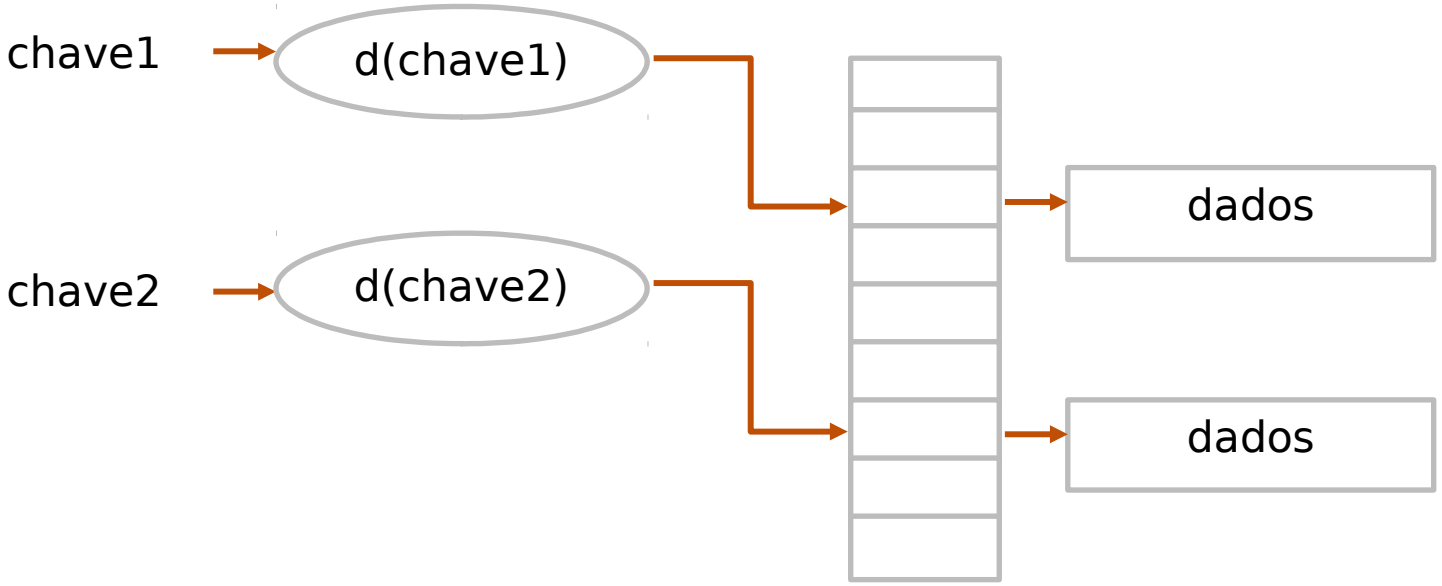


Boas Funções de Dispersão

Uma boa função de dispersão

- tem cálculo eficiente (“barato”)
- tem poucas “colisões” (espalha bem as chaves de busca)

  → uma colisão requer procedimento adicional para encontrar o elemento



Funções de Dispersão comuns

Chaves inteiras:

- resto da divisão (módulo N)
  - **N** número primo, ou sem fatores primos menores que 20, para um melhor espalhamento
- mid-square
  - **q** = chave^2
  - toma-se **x** bits do “meio” de **q**
- folding
  - chave é particionada em blocos de bits
  - soma ou multiplicação desses blocos

Chaves não numéricas

Cadeias de caracteres (*strings*)

- soma dos códigos ASCII
- soma com *shitfs*

```
unsigned int luaS_hash (const char *str, size_t l, unsigned int seed) {

  unsigned int h = seed ^ (unsigned int) l;
  size_t step = (l >> LUAI_HASHLIMIT) + 1;

  for (; l >= step; l -= step)
    h ^= ((h<<5) + (h>>2) + (unsigned char) (str[l - 1]));

  return h;
}
```

Colisões

Chaves diferentes mapeadas para um mesmo índice

- boas funções de dispersão minimizam ocorrência de colisões

Tamanho da tabela de dispersão deve ter folga em relação ao número de elementos armazenados

- tabela não deve ter taxa de ocupação maior que 75%
- uma taxa de 50% geralmente tem bons resultados
- uma taxa menor que 25% pode representar gasto excessivo de memória

Tratamento de Colisões

Encadeamento

- exterior → uso de listas encadeadas
- interior (*linear probing*) → busca linear

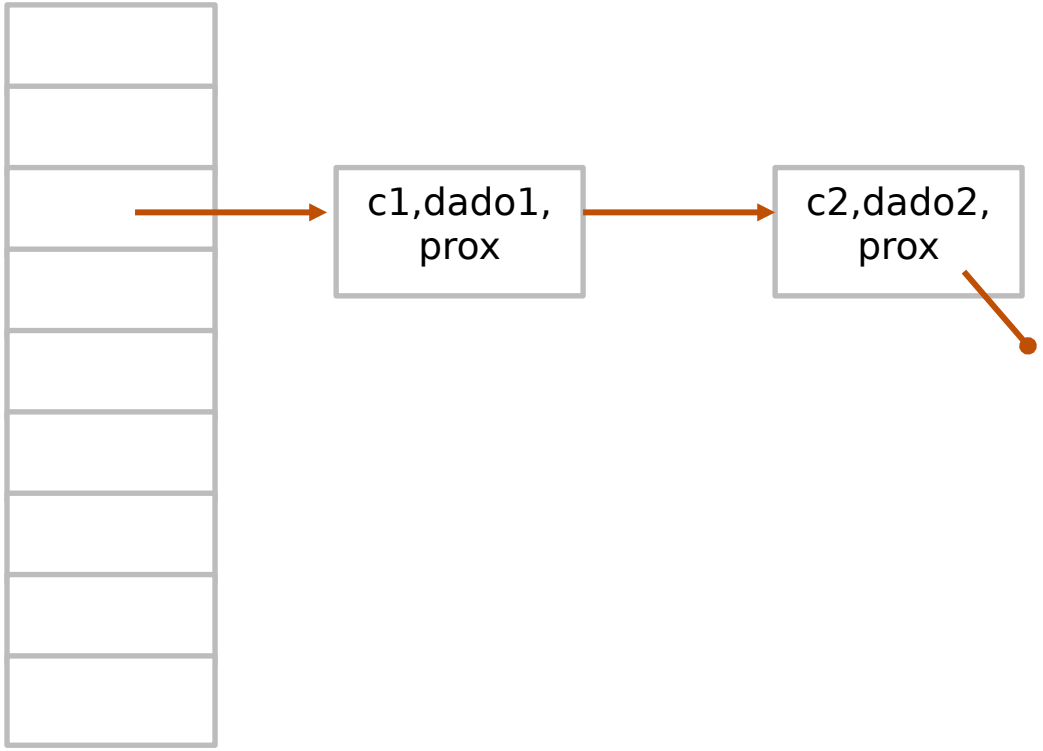
Uso de uma segunda função de dispersão

Encadeamento Exterior

Taxa de ocupação baixa → complexidade média de busca é O(1)

Desvantagem: requer estruturas externas

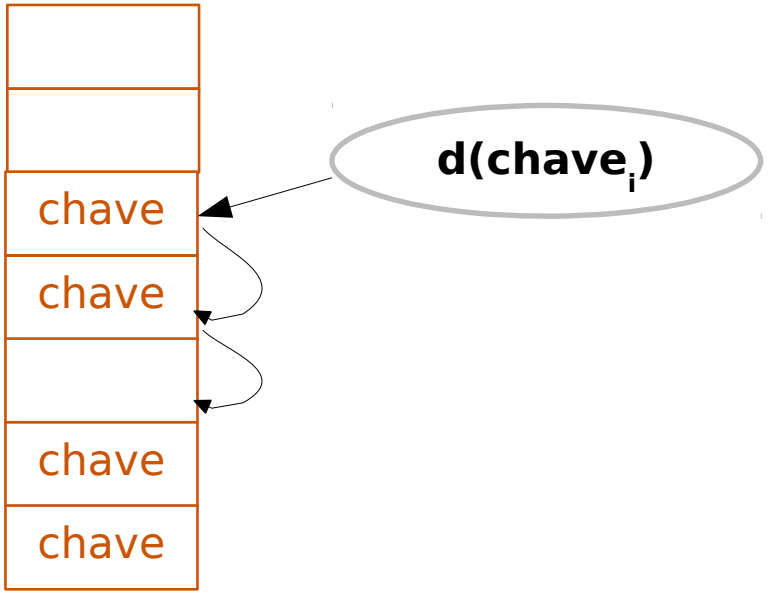
- alocação dinâmica de memória



Encadeamento Interior

Procura linear de uma posição livre a partir da posição h(x)

- incremento circular
- função de busca deve procurar a partir de h(x) uma posição com a chave dada
- importância da densidade de preenchimento!

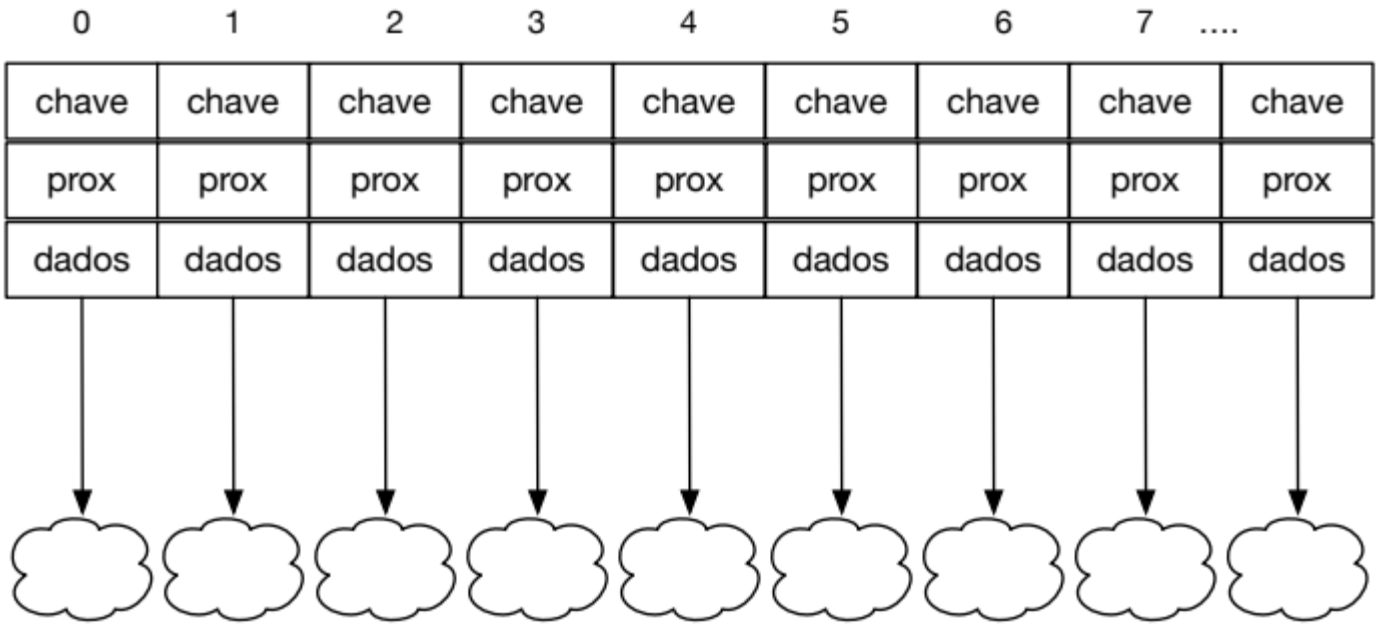


Segunda Função de Dispersão

Variante do encadeamento interior

- para evitar a concentração de posições ocupadas
- tamanho do “passo” para procura de posição livre é determinado por uma segunda função **d’(chave)**

Implementação: Encadeamento Interior

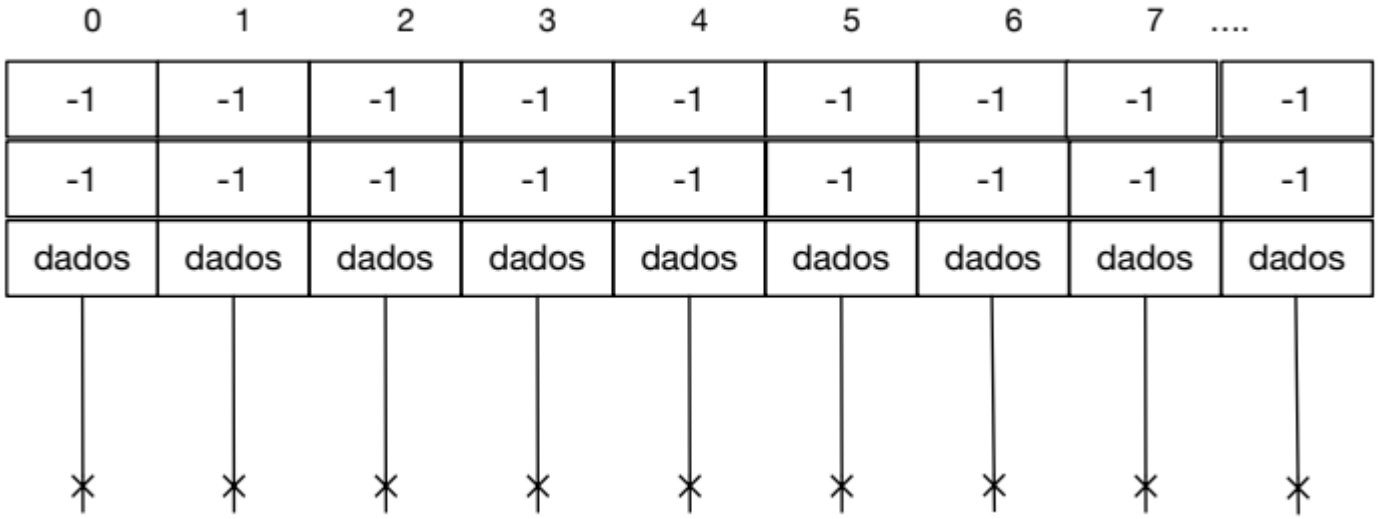


eficiência x espaço

```
typedef struct {
  int chave;
  int dados;
  int prox;
} ttabpos;
```

```
struct smapa {
  int tam;
  int ocupadas;
  ttabpos *tabpos;
};
```

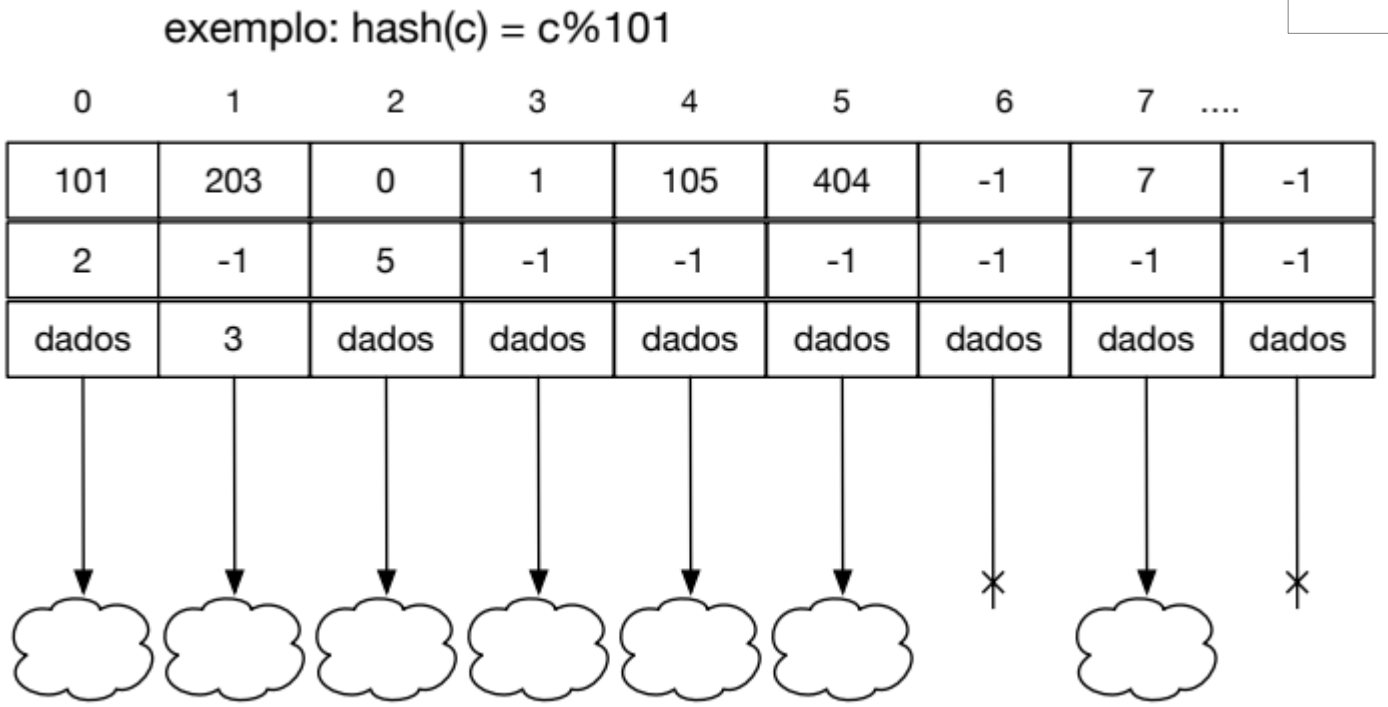
Criação do Mapa




```
Mapa* cria () {
  int i;
  Mapa* m = (Mapa*) malloc (sizeof(Mapa));
  if (m==NULL) {printf("erro na alocação! \n"); exit(1);}
  m->tabpos = (ttabpos*) malloc (TAMINICIAL*sizeof(ttabpos));
  if (m->tabpos==NULL) {printf("erro na alocação! \n"); exit(1);}
  m->tam = TAMINICIAL;
  m->ocupadas = 0;
  for (i=0;i<TAMINICIAL;i++) {
    m->tabpos[i].chave = -1;
    m->tabpos[i].prox = -1;
  }
  return m;
}
```

Exemplo

hash(101) = 0
hash(0) = 0
hash(404) = 0



Inserção

Mapa *insere(Mapa *m, int chave, int dados);

- posição livre: insere nessa posição
- posição ocupada: procura posição livre e trata conflito com a chave **c1** presente

Inserção

Mapa *insere(Mapa *m, int chave, int dados);

- posição livre: insere nessa posição
- posição ocupada: procura posição livre e trata conflito com a chave **c1** presente

Conflito primário: hash(c1) = hash(chave)

Inserção

Mapa *insere(Mapa *m, int chave, int dados);

- posição livre: insere nessa posição
- posição ocupada: procura posição livre e trata conflito com a chave **c1** presente

Conflito primário: hash(c1) = hash(chave)

- encadeia item (na posição livre) na “cadeia” de **c1**

Inserção

Mapa *insere(Mapa *m, int chave, int dados);

- posição livre: insere nessa posição
- posição ocupada: procura posição livre e trata conflito com a chave **c1** presente

Conflito primário: hash(c1) = hash(chave)

- encadeia item (na posição livre) na “cadeia” de **c1**

Conflito secundário: hash(c1) ≠ hash(chave)

Inserção

Mapa *insere(Mapa *m, int chave, int dados);

- posição livre: insere nessa posição
- posição ocupada: procura posição livre e trata conflito com a chave **c1** presente

Conflito primário: hash(c1) = hash(chave)

- encadeia item (na posição livre) na “cadeia” de **c1**

Conflito secundário: hash(c1) ≠ hash(chave)

- “expulsa” c1, que deve ser movido para a posição livre, encadeado na sua cadeia, se existir
- novo item ocupa a antiga posição de c1

Tabela Cheia

```
struct smapa {
  int tam;
  int ocupadas;
  ttabpos *tabpos;
};
```

idealmente mantém máximo de ocupação (75%)

```
Mapa* insere (Mapa* m, int chave, int dados) {

  if (m->ocupadas > 0.75*m->tam) redimensiona(m);
  int pos = hash(m, chave);
  …
```

a função de hash utiliza o tamanho!

Redimensionamento

```
static void redimensiona (Mapa* m) {
  int i;
  int tamanterior = m->tam;
  ttabpos* anterior = m->tabpos;

  /* aumenta o tamanho */
  m->tam = 1.947*m->tam;

  m->tabpos = (ttabpos*) malloc (m→tam*sizeof(ttabpos));

  /* inicialização da nova tabela */
  …

  /* e os elementos já inseridos na tabela anterior? */
  …


  free (anterior);
}
```