Pesquisa em memória primária: hashing

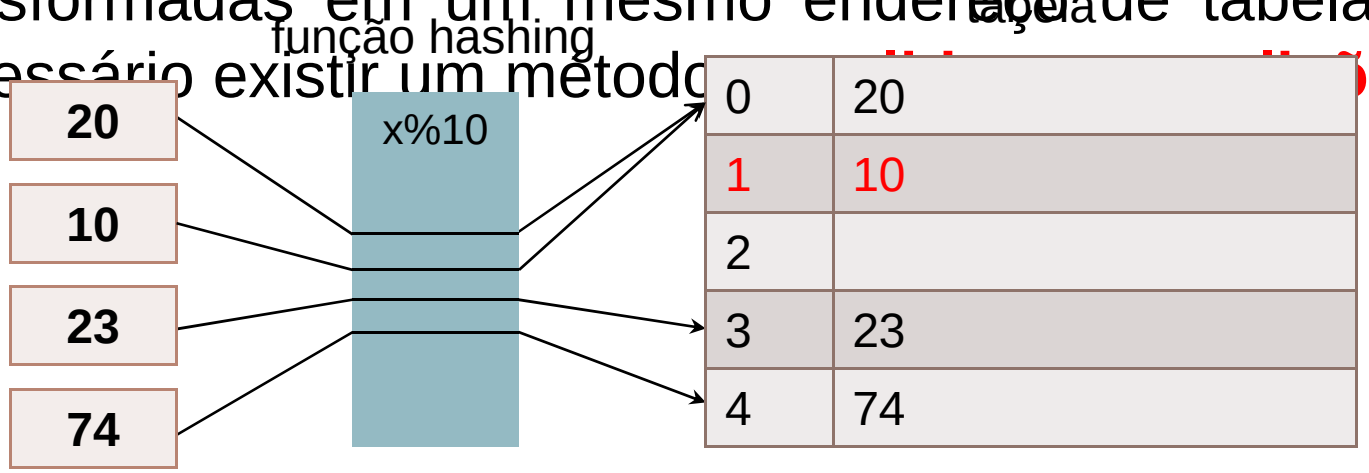
Algoritmos e Estruturas de Dados II

Hashing

- Algoritmos vistos efetuam comparações para localizar uma chave.
- Os registros armazenados em uma tabela são diretamente endereçados a partir de uma transformação aritmética sobre a chave de pesquisa.
- Busca por meio de operações aritméticas que transformam a chave em endereços em uma tabela.

Hashing

- Um método de pesquisa com o uso da transformação de chave é constituído de duas etapas principais:
  - Computar o valor da **função de transformação**, a qual transforma a chave de pesquisa em um endereço da tabela.
  - Considerando que duas ou mais chaves podem ser transformadas em um mesmo endereço de tabela, é necessário existir um método [illegible] **es**.



Hashing: colisões

- Qualquer que seja a função de transformação, algumas colisões irão ocorrer fatalmente, e tais colisões têm de ser resolvidas de alguma forma.
- Mesmo que se obtenha uma função de transformação que distribua os registros de forma uniforme entre as entradas da tabela, existe uma alta probabilidade de haver colisões.

Hashing: colisões

- O paradoxo do aniversário (Feller,1968, p. 33), diz que em um grupo de 23 ou mais pessoas, juntas ao acaso, existe uma chance maior do que 50% de que 2 pessoas comemorem aniversário no mesmo dia.
- Para hashing:
  - Tabela com 365 posições
  - Adição de 23 chaves
  - Chance > 50% de haver colisão

Hashing: colisões

- A probabilidade p de se inserir N itens consecutivos sem colisão em uma tabela de tamanho M é:

$$p = \frac{M-1}{M} \times \frac{M-2}{M} \times \ldots \times \frac{M-N+1}{M} = \prod_{i=1}^{N} \frac{M-i+1}{M} = \frac{M!}{(M-N)!M^N}$$

Hashing: colisões

- Seja uma tabela com 50.063.860 de posições:

Chaves	Chance de colisão	Fator de carga (N/M)
1000	0.995%	0.002%
2000	3.918%	0.004%
4000	14.772%	0.008%
6000	30.206%	0.012%
8000	47.234%	0.016%
10000	63.171%	0.020%
12000	76.269%	0.024%
14000	85.883%	0.028%
16000	92.248%	0.032%
18000	96.070%	0.036%
20000	98.160%	0.040%
22000	99.205%	0.044%

Função de Transformação

- Uma função de transformação deve mapear chaves em inteiros dentro do intervalo [0..M − 1], onde M é o tamanho da tabela.
- A função de transformação ideal é aquela que:
  - Seja simples de ser computada.
  - Para cada chave de entrada, qualquer uma das saídas possíveis é igualmente provável de ocorrer.
- Exemplo:
  - Usa o resto da divisão por M (onde k é a chave)

$$h(k) = k \% M$$

Chaves não Numéricas

- As chaves não numéricas devem ser transformadas em números:

$$\mathbf{H} = \sum_{i=1}^{n} \mathrm{Chave}[i] \times p[i],$$

- n é o número de caracteres da chave.
- Chave[i] corresponde à representação ASCII do i-ésimo caractere da chave.
- p[i] é um inteiro de um conjunto de pesos gerados randomicamente para 1 ≤ i ≤ n.

$$H = \sum_{i=1}^{n} Chave[i] \times 128^{n-i}$$

Chaves não Numéricas

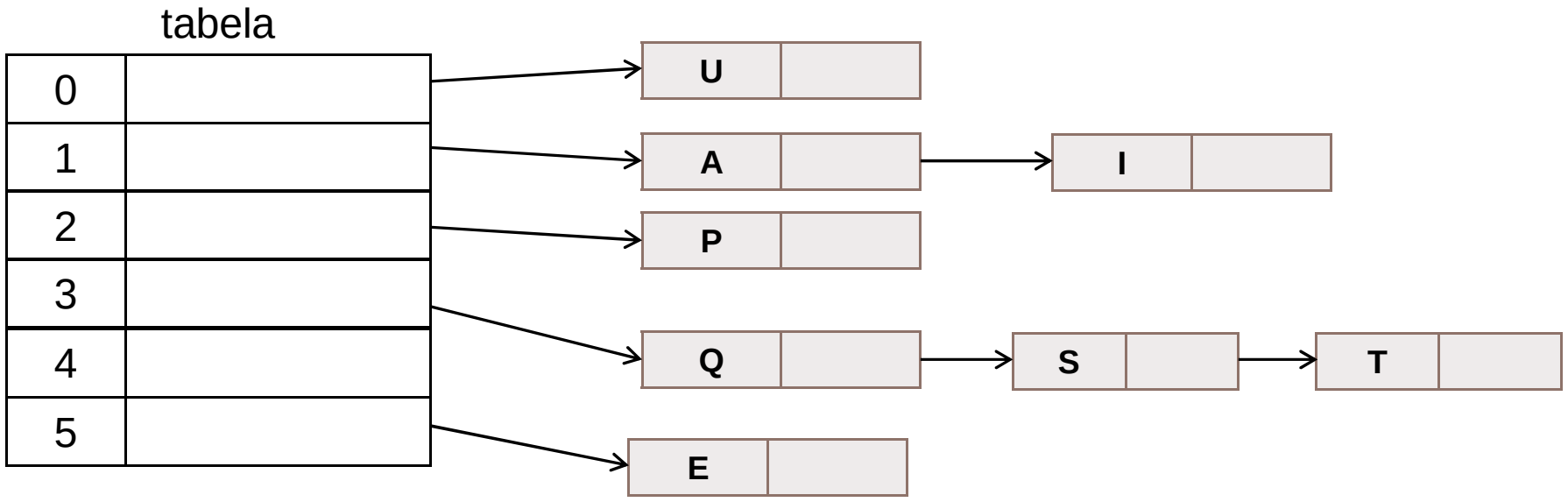
```
Indice h(TipoChave Chave, TipoPesos p) {
unsigned int Soma = 0;
int i;

   int comp = strlen(Chave);

   for (i = 0; i < comp; i++)
      Soma += (unsigned int) Chave[i] * p[i];

    return (Soma % M);
}
```

Resolução de Colisões

- Encadeamento
- Endereçamento aberto

Resolução de Colisões - Encadeamento

- Cria uma lista encadeada para cada endereço da tabela.
- Todas as chaves com mesmo endereço na tabela são encadeadas em uma lista linear.



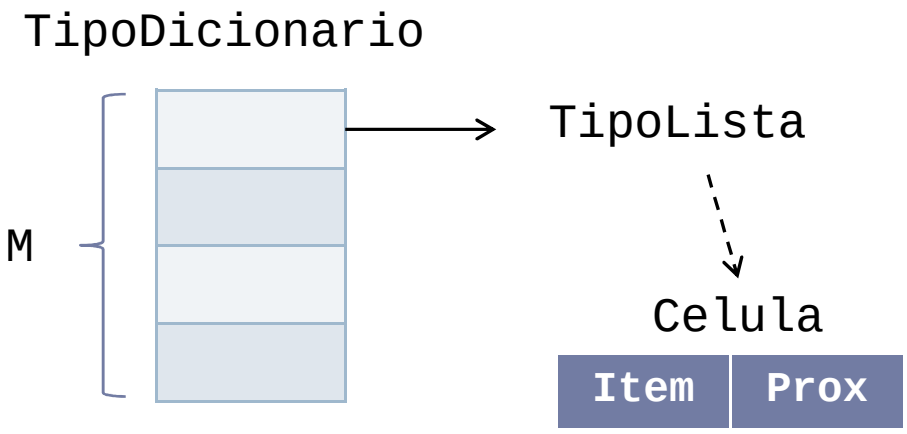
Resolução de Colisões – Encadeamento

- Exemplo: Considerando a i-ésima letra do alfabeto representada por i e a função de transformação h(Chave) = Chave mod M (M=10). Insira EXEMPLO na tabela.

Estrutura de Dicionário para Encadeamento

```
#define M 10
#define n 5
typedef char TipoChave[n];
typedef unsigned int TipoPesos[n];

typedef struct {
   /* outros componentes */
   TipoChave Chave;
} TipoItem;

typedef unsigned int Indice;
typedef struct _celula* Apontador;

typedef TipoLista TipoDicionario[M];
```

```
typedef struct _celula {
  TipoItem Item;
  Apontador Prox;
} Celula;

typedef struct {
   Celula *Primeiro, *Ultimo;
} TipoLista;
```



Estrutura de Dicionário para Encadeamento

```
void Inicializa(TipoDicionario T) {
int i;
  for (i = 0; i < M; i++) /* inicializa as M listas */
     FLVazia(&T[i]);
}
```

Estrutura de Dicionário para Encadeamento

```
Apontador Pesquisa(TipoChave Ch, TipoPesos p, TipoDicionario T){
/* Apontador de retorno aponta para o item anterior da lista */
Indice i;   Apontador Ap;

  i = h(Ch, p);
  if (Vazia(T[i])) return NULL;   /* Pesquisa sem sucesso */
  else  {
     Ap = T[i].Primeiro;
     while ((Ap->Prox->Prox != NULL) &&
       (strncmp(Ch, Ap->Prox->Item.Chave, sizeof(TipoChave))))
       { Ap = Ap->Prox;}
     if (!strncmp(Ch, Ap->Prox->Item.Chave, sizeof(TipoChave)))
          return Ap;
     else return NULL;   /* Pesquisa sem sucesso */
    }
}
```

Estrutura de Dicionário para Listas Encadeadas

```
void Insere(TipoItem x, TipoPesos p, TipoDicionario T) {
Indice i;
   if (Pesquisa(x.Chave, p, T) == NULL)
      /* insere do TAD Lista */
      Insere(x, &T[h(x.Chave, p)]);
   else
      printf("Registro ja  esta  presente\n");
}

void Retira(TipoItem x, TipoPesos p, TipoDicionario T) {
Apontador Ap = Pesquisa(x.Chave, p, T);

   if (Ap == NULL)
      printf(" Registro nao esta  presente\n");
   else
      Retira(Ap, &T[h(x.Chave, p)], &x); /*Retira do TAD Lista*/
}
```

Encadeamento: Análise

- Tamanho esperado de cada lista: N/M
  - Assumindo que qualquer item do conjunto tem igual probabilidade endereçado para qualquer entrada de T.
  - N: número de registros, M: tamanho da tabela
- Operações Pesquisa, Insere e Retira: O(1 + N/M)
  - 1: tempo para encontrar a entrada na tabela
  - N/M: tempo para percorrer a lista
- M ~ N, tempo se torna constante.

Resolução de Colisões: Endereçamento Aberto

- Quando o número de registros a serem armazenados na tabela puder ser previamente estimado, não há necessidade de se utilizar apontadores.

- Para tabela com tamanho M (M > N), pode-se utilizar os espaços vazios da própria tabela para resolver as colisões.

- **Endereçamento aberto**: chaves são armazenadas na própria tabela.

Resolução de Colisões: Endereçamento Aberto

- Quando encontra uma colisão, procura localizações alternativas (h_j).

- Hashing linear

$$h_j = (h(x) + j) \bmod M, \quad \text{para } 1 \leq j \leq M - 1$$

- Hashing quadrático

$$h_j = (h(x) + j^2) \bmod M$$

Endereçamento Aberto: Exemplo

- Se a i-ésima letra do alfabeto é representada pelo número i e a função de transformação

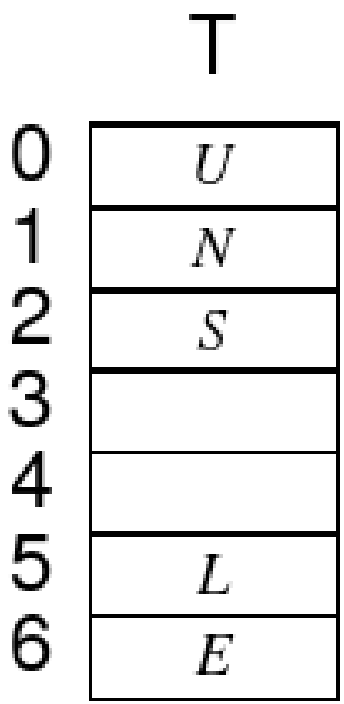
  h(Chave) = Chave mod M

é utilizada com M = 7

- O resultado da inserção das chaves L U N E S na tabela, usando hashing linear para resolver colisões é mostrado abaixo.

Endereçamento Aberto: Exemplo

```
h(L) = h(12) = 5
h(U) = h(21) = 0
h(N) = h(14) = 0
h(E) = h(5)  = 5
h(S) = h(19) = 5
```



Endereçamento Aberto: Implementação

```
#define Vazio           "!!!!!!!!!!"
#define Retirado        "**********"
#define M               7
#define n               11     /* Tamanho da chave */
typedef unsigned int Apontador;

typedef char TipoChave[n];
typedef unsigned TipoPesos[n];

typedef struct {
  /* outros componentes */
  TipoChave Chave;
} TipoItem;

typedef unsigned int Indice;
typedef TipoItem TipoDicionario[M];
```

Endereçamento Aberto: Implementação

```
void Inicializa(TipoDicionario T) {
int i;

   /* inicializa todas as posições como vazias */
   for (i = 0; i < M; i++)
      strcpy(T[i].Chave, Vazio);
}
```

Endereçamento Aberto: Implementação

```
Apontador Pesquisa(TipoChave Ch, TipoPesos p, TipoDicionario T) {
unsigned int  i = 0;
unsigned int  Inicial;

   Inicial = h(Ch, p);  /* transforma a chave */

   while ((strcmp(T[(Inicial + i) % M].Chave, Vazio) != 0) &&
          (strcmp(T[(Inicial + i) % M].Chave, Ch) != 0)  &&
          (i < M))
             i++;

   if (strcmp(T[(Inicial + i) % M].Chave, Ch) == 0)
      return ((Inicial + i) % M);
   else
      return M;   /* Pesquisa sem sucesso */
}
```

Endereçamento Aberto: Implementação

```
void Retira(TipoChave Ch, TipoPesos p, TipoDicionario T) {
Indice i;

   i = Pesquisa(Ch, p, T);

   if (i < M) /* registro encontrado */
      strcpy(T[i].Chave, Retirado);
   else
      printf("Registro nao esta presente\n");
}
```

Endereçamento Aberto: Implementação

```
void Insere(TipoItem x, TipoPesos p, TipoDicionario T) {
unsigned int i = 0;
unsigned int Inicial;

  if (Pesquisa(x.Chave, p, T) < M)  {
    printf("Elemento ja esta presente\n");
    return;
  }

  Inicial = h(x.Chave, p); /* transforma a chave */

  while ((strcmp(T[(Inicial + i) % M].Chave, Vazio) != 0) &&
         (strcmp(T[(Inicial + i) % M].Chave, Retirado) != 0) &&
         (i < M))
           i++;

  if (i < M)   {
     strcpy (T[(Inicial + i) % M].Chave, x.Chave);
     /* Copiar os demais campos de x, se existirem */
  }
  else printf("Tabela cheia\n");
}
```

Endereçamento Aberto: Análise

- Seja α = N / M o fator de carga da tabela. Conforme demonstrado por Knuth (1973), o custo de uma pesquisa com sucesso é

$$C(n) = \frac{1}{2}\left(1 + \frac{1}{1-\alpha}\right)$$

- O hashing linear sofre de um mal chamado agrupamento.
- Ocorre na medida em que a tabela começa a ficar cheia, pois a inserção de uma nova chave tende a ocupar uma posição na tabela que esteja contígua a outras posições já ocupadas, o que deteriora o tempo necessário para novas pesquisas.

Endereçamento Aberto: Análise

$$C(n) = \frac{1}{2}\left(1 + \frac{1}{1-\alpha}\right)$$

α = N / M	C(n)
0.1000	1.0556
0.2000	1.1250
0.3000	1.2143
0.4000	1.3333
0.5000	1.5000
0.6000	1.7500
0.7000	2.1667
0.8000	3.0000
0.9000	5.5000
0.9500	10.5000
0.9800	25.5000
0.9900	50.5000

Vantagens e Desvantagens da Transformação da Chave

- Vantagens
  - Alta eficiência no custo de pesquisa, que é O(1) para o caso médio.
  - Simplicidade de implementação.

- Desvantagens
  - Pior caso é O(N).