

AULA 23

Tabela de símbolos

Uma **tabela de símbolos** (= *symbol table* = *dictionary*) é um conjunto de **objetos** (*itens*), cada um dotado de uma **chave** (= *key*) e de um **valor** (= *value*).

As chaves podem ser números inteiros ou *strings* ou outro tipo de dados.

Uma tabela de símbolos está sujeito a **dois tipos de operações**:

- ▶ **inserção**: consiste em introduzir um objeto na tabela;
- ▶ **busca**: consiste em encontrar um elemento que tenha uma dada chave.

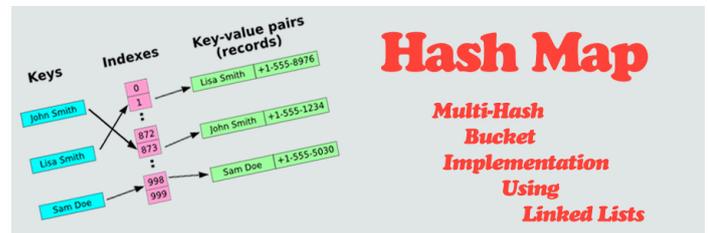
Um exemplo simples

Os **valores** serão nomes de pessoas (**Strings**), identificados por **números inteiros** que farão o papel das **chaves**.

Para cada número queremos saber o nome da pessoa que identificada por ele.

```
typedef char * String;  
typedef int Chave;  
typedef String Valor;
```

Tabelas de símbolos e de distribuição



Fonte: <http://programmingnotes.freeweq.com>

S 12.4, 12.5, 12.8, 14.1, 14.2

<http://www.ime.usp.br/~pf/.../symbol-table.html>

<http://www.ime.usp.br/~pf/.../symb-table.html>

Tabela de símbolos

Problema: Organizar uma **tabela de símbolos** de maneira que as operações de **inserção** e **busca** sejam **razoavelmente eficientes**.

Em geral, uma organização que permite **inserções** rápidas impede **buscas** rápidas e vice-versa.

Interface

Feita **sob medida**. **Não é** uma implementação **genérica**. Só admite uma tabela de símbolos.

```
void stInit(int);  
void stInsert(Chave, Valor);  
Valor stSearch(Chave);  
void stFree( );  
void stDelete(Chave);
```

Implementação com endereçamento direto

Tabela indexada pelas **chaves**, uma posição para cada possível **índice**.

Cada posição armazena o **valor** correspondente a uma dada **chave**.

```
static Valor *tab= NULL;
static int nChaves = 0;
static int M = 0; /* tam da tabela */
```

stInit

```
void
stInit(int max)
{
    int h;
    M = max;
    nChaves = 0;
    tab = mallocSafe(M * sizeof(Valor));
    for (h = 0; h < M; h++)
        tab[h] = NULL;
}
```

stDelete

```
void
stDelete(Chave chave)
{
    if (tab[chave])
    {
        free(tab[chave]);
        tab[chave] = NULL;
        nChaves--;
    }
}
```

Implementação com endereçamento direto

/* CopyString da biblioteca strlib de

```
Eric Roberts */
static String
copyString(String string)
{
    String str = NULL;
    int len = strlen(string) + 1;
    str = mallocSafe(len * sizeof(char));
    strncpy(str, string, len);
    return str;
}
```

stInsert e stSearch

```
void
stInsert(Chave chave, Valor valor)
{
    if (tab[chave]) free(tab[chave]);
    else nChaves++;
    tab[chave] = copyString(valor);
}

Valor
stSearch(Chave chave)
{
    return tab[chave];
}
```

stFree

```
void
stFree()
{
    int h;
    for (h = 0; h < M; h++)
        if (tab[h]) free(tab[h]);
    free(tab);
    tab = NULL;
    nChaves = 0;
    M = 0;
}
```

Consumo de tempo

Em uma **tabela se símbolos** com **endereçamento direto** o consumo de tempo de **stInsert**, **stSearch** e **stDelete** é **O(1)**.

O consumo de tempo de **stInit** e **stFree** é **O(M)**, onde **M** é o número de **chaves**.

Maiores defeitos

Os **maiores defeitos** dessa implementação são:

- ▶ Em geral, as **chaves não são inteiros** não-negativos pequenos. . .
- ▶ **despedício de espaço**: é possível que a maior parte da tabela fique vazia

Tabelas de dispersão (*hash tables*)

Uma **tabela de dispersão** (= *hash table*) é uma maneira de organizar uma tabela de símbolos.

Inventadas para funcionar bem em **média**.

universo de chaves = conjunto de **todas** as possíveis **chaves**

chaves realmente usadas são, em geral, uma **parte pequena** do **universo**.

A tabela terá a forma **tab[0..M-1]**, onde **M** é o **tamanho da tabela**.

Funções de dispersão

Uma **função de dispersão** (= *hash function*) é uma maneira de mapear o **universo de chaves** no conjunto de **índices** da tabela.

A **função de dispersão** recebe uma **chave** e retorna um número inteiro **h** no intervalo **0..M-1**.

O número **h** é o **código de dispersão** (= *hash code*) da chave.

Exemplo de um função de dispersão **modular**:

```
int hash(Chave chave, int M) {  
    return chave % M;  
}
```

Boas e más funções de dispersão

Uma função só é **eficiente** se **espalha** as chaves pelo intervalo de índices de maneira **razoavelmente uniforme**.

Por exemplos, se os **dois últimos dígitos** da chaves não variam muito, então "**chave% 100**" é um **péssima** função de dispersão.

É recomendável que **M** seja um número **primo**.

Escolha de funções de dispersão é uma **combinação** de **estatística**, **probabilidade**, **teoria dos números** (**primalidade**); . . . ;

$a = M*k_a + a \% M \rightarrow a \% M = a - M*k_a$

$b = M*k_b + a \% M \rightarrow b \% M = b - M*k_b$

$(a \% M) \% M = a \% M$

$(a + b) \% M = (a + b) - M*k_{ab}$
 $= (a + b) - M*(k_a + k_b + (a \% M + b \% M) / M)$
???
 $= (a - M*k_a) + (b - M*k_b)$
 $= a \% M + b \% M$

Funções de dispersão para strings

typedef String Chave;

Utilizando o valor ASCII de cada caractere, uma **string** pode ser interpretada como a representação em base 128 (ou 256) de um **número**:

"MAC0122" = 'M' × 128⁶ + 'A' × 128⁵ + 'C' × 128⁴
+ '0' × 128³ + '1' × 128² + '2' × 128¹
+ '2' × 128⁰
= 340901050997042

É recomendável que a base também seja um número **primo**.

$= ('2'*1 + 128*('2' + 128*('1' + 128*('0' + 128*(...)))) \% M$
???
 $= ('2'*1 + 128*('2' + 128*('1' + 128*('0' + 128*(...)%M)%M) \% M) \% M$

Função de dispersão básica

Para evitar *overflow* usamos o *método de Horne* e tomamos o resto da divisão após cada multiplicação:

```
int hash(Chave chave, int M)
{
    int i, h = 0;
    int primo = 127;
    for (i = 0; chave[i] != '\0'; i++)
        h = (h * primo + chave[i]) % M;
    return h;
}
```

◀ ▶ ↺ ↻ 🔍

Problema motivação

Usaremos o problema a seguir para exemplificar o uso de técnicas para tratar *colisões*.

Problema: Determinar o número de ocorrências de cada *palavra* em um *arquivo texto*. Em seguida, dizer *quantas vezes* cada *palavra* de uma lista de palavras ocorreu no *texto*.

Utilizaremos uma *Tabela de símbolos* para resolver o problema.

◀ ▶ ↺ ↻ 🔍

Interface

```
typedef char *String;
typedef String Chave;
typedef int Valor;

void stInit(int);
void stInsert(Chave, Valor);
Valor stSearch(Chave);
void stFree();
void stDelete(Chave);
```

◀ ▶ ↺ ↻ 🔍

Colisões

Como o número de *chaves* é em geral *maior* que *M*, é inevitável que a função de dispersão leve várias *chaves diferentes* no *mesmo índice*.

Dizemos que há uma *colisão* quando duas *chaves diferentes* são levadas no *mesmo índice*.

Algumas maneiras de tratar colisões:

- ▶ *lista encadeadas* (= *separating chaining*);
- ▶ *sondagem linear* (= *linear probing (open addressing)*);
- ▶ *double hashing* (*open addressing*);

◀ ▶ ↺ ↻ 🔍

Tabela de símbolos

Na *tabela de símbolos*, as *palavras* no *texto* farão o papel de *chaves*.

O *valor* de cada *chave* será o *número de vezes* que ela *ocorre* no *texto*.

A *tabela de símbolos* será implementada através de uma *tabela de dispersão* (= *hash table*).

◀ ▶ ↺ ↻ 🔍

Colisões por listas encadeadas

Uma solução popular para resolver *colisões* é conhecida como *separate chaining*:

para cada índice h da tabela há uma lista encadeada que armazena todos os objetos que a função de dispersão leva em h .

Essa solução é muito boa se cada uma das “*listas de colisão*” resultar *curta*.

Se o número total de *chaves* usadas for *N*, o comprimento de cada lista deveria, idealmente, estar próximo de $\alpha = N/M$.

O valor α é chamado de *fator de carga* (= *load factor*) da tabela.

◀ ▶ ↺ ↻ 🔍

Implementação

```
static int hash(Chave chave, int M);
typedef struct celTS CelTS;
struct celTS{
    Chave chave;
    Valor valor;
    CelTS *prox;
};
static CelTS **tab = NULL;
static int nChaves = 0;
static int M; /* tam. da tabela */
```

◀ ▶ ↺ ↻ 🔍

stInsert

```
void stInsert(Chave chave, Valor valor) {
    CelTS *p;
    int h = hash(chave,M);
    p = tab[h];
    while (p && strcmp(p->chave,chave))
        p = p->prox;
    if (p == NULL) {
        p = mallocSafe(sizeof *p);
        p->chave = copyString(chave);
        nChaves += 1;
        p->prox = tab[h];    tab[h] = p;
    }
    p->valor = valor;
}
```

◀ ▶ ↺ ↻ 🔍

stDelete

Rascunho:

```
void stDelete(Chave chave) {
    int h;
    h = hash(chave,M); /* codigo de hash */
    "tab[h] = buscaRemove(tab[h], chave);"
    /* precisa atualizar nChaves...*/
}
```

◀ ▶ ↺ ↻ 🔍

stInit

```
void stInit(int max)
{
    int h;
    M = max;
    nChaves = 0;
    tab = mallocSafe(M * sizeof(CelTS*));
    for (h = 0; h < M; h++)
        tab[h] = NULL;
}
```

◀ ▶ ↺ ↻ 🔍

stSearch

```
Valor stSearch(Chave chave) {
    CelTS *p = NULL;
    int h;
    h = hash(chave,M); /* codigo de hash */
    p = tab[h];
    while (p && strcmp(p->chave,chave))
        p=p->prox;
    if (p == NULL) return 0;
    return p->valor;
}
```

◀ ▶ ↺ ↻ 🔍

stFree

```
void stFree() {
    CelTS *p = NULL, *q = NULL;
    int h;
    for (h = 0; h < M; h++) {
        p = tab[h];
        while (p) {
            q = p;
            p = p->prox;
            free(q->chave);
            free(q);
        }
    }
    free(tab);    tab = NULL;    nChaves = 0;
}
```

◀ ▶ ↺ ↻ 🔍

Consumo de tempo

Seja N é o número de **chaves** e M é o tamanho da tabela.

Supondo que a função **hash** distribuía as chaves uniformemente em $[0..M-1]$, em uma **tabela de distribuição** com **listas encadeadas** o consumo de tempo de **stInsert**, **stSearch** e **stDelete** é $O(N/M)$.

◀ ▶ ↺ ↻ 🔍

Colisões por sondagem linear

Um outro método de resolução de **colisões** é conhecido como **sondagem linear** (= *linear probing*).

Todos os objetos são armazenados em um vetor **tab** $[0..M-1]$.

Quando ocorre uma **colisão**, procuramos a **próxima posição vaga** do vetor.

Digamos que a tabela tem M posições e contém N **chaves** num dados instante. O **fator de carga** $\alpha = N/M$ da tabela é menor do que 1.

Quanto maior o **fator de carga**, mais tempo as funções de busca e inserção vão consumir.

◀ ▶ ↺ ↻ 🔍

Implementação

```
#define LIVRE(h) (tab[h].chave == NULL)
#define INCR(h) (h = h == M-1? 0: h+1)

static int hash(Chave chave, int M);

typedef struct celTS CelTS;
struct celTS {
    Chave chave;
    Valor valor;
};

static CelTS *tab = NULL;
static int nChaves = 0;
static int M; /* tam. da tabela */
```

◀ ▶ ↺ ↻ 🔍

Consumo de tempo

Seja N é o número de **chaves** e M é o tamanho da tabela.

Em uma **tabela de distribuição** com **listas encadeadas** o consumo de tempo de **stInit** é $O(M)$ e o consumo de tempo de **stFree** é $O(M + N)$.

◀ ▶ ↺ ↻ 🔍

Interface

```
typedef char *String;
typedef String Chave;
typedef int Valor;

void stInit(int);
void stInsert(Chave, Valor);
Valor stSearch(Chave);
void stFree();
void stDelete(Chave);
```

◀ ▶ ↺ ↻ 🔍

stInit

```
void stInit(int max)
{
    int h;
    M = max;
    nChaves = 0;
    tab = mallocSafe(M * sizeof(CelTS));
    for (h = 0; h < M; h++)
        tab[h].chave = NULL;
}
```

◀ ▶ ↺ ↻ 🔍

