Tabelas de símbolos com hashing

- Além do Sedgewick (**sempre** leiam o Sedgewick), veja
  - `http://www.ime.usp.br/~pf/mac0122-2002/aulas/hashing.html`

Implementações elementares de tabelas de símbolos

- Vetor indexado pelas chaves (que supomos serem inteiros pequenos)
  - Supomos que as chaves são inteiros não-negativos pequenos; colocamos item de chave i na posição $v[i]$ do vetor v.
  - Tempo de inserção, remoção e busca: $O(1)$

Tabelas de símbolos

```
/* prog12.1.c - ST.h */
void STinit(int);
 int STcount();
void STinsert(Item);
Item STsearch(Key);
void STdelete(Item);
Item STselect(int);
void STsort(void (*visit)(Item));
```

Vetores indexados por chaves

```
/* prog12.3.c */
static Item *st;
static int M = maxKey;

void STinit(int maxN)
  { int i;
    st = malloc((M+1)*sizeof(Item));
    for (i = 0; i <= M; i++) st[i] = NULLitem;
  }
```

Vetores indexados por chaves

```
[...]
void STinsert(Item item)
  { st[key(item)] = item; }

Item STsearch(Key v)
  { return st[v]; }

void STdelete(Item item)
  { st[key(item)] = NULLitem; }
[...]
```

Problema básico

- Em geral, as chaves não são inteiros não-negativos pequenos.
- *Funções de espalhamento (hashing)*: funções que transformam chaves genéricas em endereços (índices) de tabelas (vetores).

Funções de hashing

- Consideramos o caso em que as chaves são cadeias de caracteres.
- Usamos o valor ASCII de cada caracter
- Podemos interpretar strings como inteiros em base 128 (ou 256). Por exemplo: $\text{Google} = \text{G} \times 128^5 + \text{o} \times 128^4 + \text{o} \times 128^3 + \text{g} \times 128^2 + \text{l} \times 128^1 + \text{e} \times 128^0 = 71 \times 128^5 + 111 \times 128^4 + 111 \times 128^3 + 103 \times 128^2 + 108 \times 128^1 + 101 \times 128^0 = 2469572245093$
- Podemos reduzir $\mod M$ para obter valores entre 0 e $M-1$.

Método de Horner

$$71 \times 128^5 + 111 \times 128^4 + 111 \times 128^3 + 103 \times 128^2 + 108 \times 128^1 + 101 \times 128^0$$

Pode ser calculado como

$$= ((((71 \times 128 + 111) \times 128 + 111) \times 128 + 103) \times 128 + 108) \times 128 + 101$$

Função de hashing básica

```
/* prog14.1.c */
int hash(char *v, int M)
  { int h = 0, a = 127;
    for (; *v != '\0'; v++)
      h = (a*h + *v) % M;
    return h;
  }
```

- $a = 127$: "receita"
- Em geral, M deve ser um primo de tamanho apropriado

Função de hashing “universal”

```
/* prog14.2.c */
int hashU(char *v, int M)
  { int h, a = 31415, b = 27183;
    for (h = 0; *v != '\0'; v++, a = a*b % (M-1))
        h = (a*h + *v) % M;
    return h;
  }
```

Vetor indexado por chaves + funções de hashing

```
[...]
void STinsert(Item item)
  { st[h(key(item), M)] = item; }

Item STsearch(Key v)
  { return st[h(v, M)]; }
[...]
```

- Problema: *colisões*, isto é, chaves $x \neq y$ com $h(x, M) = h(y, M)$
- Uma possível solução: *encadeamento*, isto é, colocamos uma lista ligada em `st[i]` (temos `M` listas ligadas) [Veja `prog12.5.c` (implementação de TSs com listas ligadas não-ordenadas)]

Resolução de colisões por encadeamento

```
/* prog14.3.c */
static link *heads, z;
static int N, M;

void STinit(int max)
  { int i;
    N = 0; M = max/5;
    heads = malloc(M*sizeof(link));
    z = NEW(NULLitem, NULL);
    for (i = 0; i < M; i++) heads[i] = z;
  }
```

Resolução de colisões por encadeamento

```
/* prog14.3.c */
[...]
Item STsearch(Key v)
  { return searchR(heads[hash(v, M)], v); }

void STinsert(Item item)
  { int i = hash(key(item), M);
    heads[i] = NEW(item, heads[i]); N++; }

void STdelete(Item item)
  { int i = hash(key(item), M);
    heads[i] = deleteR(heads[i], item); }
```

Resolução de colisões por encadeamento

- Suponha que temos N itens e M listas. Então o tamanho médio de cada lista é $\lambda = N/M$.
- Em geral, escolhemos λ uma constante pequena.

Resolução de colisões por encadeamento

- Qual é a probabilidade de uma dada lista ter tamanho k? Sob a hipótese de que os valores da função de hashing são uniformes e independentes:

$$\mathbb{P}\left(\mathrm{Bi}(N, 1/M) = k\right) = \binom{N}{k} \left(\frac{1}{M}\right)^k \left(1 - \frac{1}{M}\right)^{N-k},$$

que, se $k = t\lambda$, é aproximadamente

$$\mathbb{P}(\mathrm{Po}(\lambda) = k) = \frac{1}{k!}\lambda^k e^{-\lambda} \leq \left(\frac{e\lambda}{t}\right)^t e^{-\lambda}.$$

- Para $\lambda = 20$ e $k = 40$ ($t = 2$), temos que esta probabilidade é algo como $.0006\%$!

Funções da interface difíceis de implementar com hashing

- `STselect()`
- `STsort()`

EP5b: Bônus

- EP5a: use tabelas de símbolos implementadas com ABBs (obrigatoriamente)
- EP5b: use tabelas de símbolos implementadas com tabelas de espalhamento (obrigatoriamente) [bônus]
- Compare a eficiência de seu EP5a e de seu EP5b e escreva um pequeno relatório.

Alternativas

- Linear probing
- Double hashing
- Tabelas de espalhamento dinâmicas