# Resumo AED II Métodos Hash – Profº Helton

***BoaNoche Development Group™***

**Tabela Hash Ordenada**

Consiste em manter todas as chaves que colidem na mesma posição em ordem decrescente. Se estivermos procurando uma chave c = T[i] | c < k, significa que k não está na tabela.

**Melhoras com gasto adicional de memória**

- **Bit visitado:**

  Gasto um bit a mais para cada posição da tabela. Todos os bits são inicializados com zero e atualizados para 1 quando na inserção de uma chave k, essa chave colide com alguma posição já ocupada.

  Toda vez que uma busca é executada e o bit adicional do n-ésimo hash é igual a zero, significa que ninguém colidiu com ele, inclusive a chave que está sendo buscada, logo ele não se encontra na tabela.

- **Método predictor:**

  Gasto um inteiro a mais para cada posição da tabela.

  Definimos uma função prb(j,k) que compute diretamente a posição do j-ésimo rehash da chave k.

  Exemplo com rehash linear:

  prb(0,k) = h(k);

  prb(1,k) = (h(k) + c)+Ts;

  prb(2,k) = (h(k) + 2c)+Ts;

  prb(j,k) =(h(k) + jc)+Ts;

  O campo adicional será chamado de predictor prd inicializado com zero. Supondo que a chave k está sendo inserida e j é o menor inteiro tal que a operação prd[prb(j,k)] = 0 depois de mais alguns rehash k1 é inserida na posição prb(p,k1).

  Prd[prb(j,k1)] é utilizado para p-j.

  Numa busca, quando a posição prb(j,k1) for diferente de k1, a próxima posição examinada é prb(j + prd[prb(j+k1)] ou prb (p, k1).

- **Hashing encadeado**

  Gasto de um inteiro a mais para cada posição da memória para marcar a próxima posição disponível na tabela.

  Não há função de rehash, a busca é orientada por uma lista encadeada na qual o campo auxiliar serve de indicador de próximo.

  No caso de colisão, a lista é seguida até o fim.

  A chave é colocada na posição de marcador de disponível; a última posição da lista é atualizada para o lugar que a chave for inserida. Analogamente, o marcador de disponível é atualizado.

**Escolhendo funções hash:** Depender de todos os bits da chave e sensível a permutações.

- **Método da divisão**

  H(k) = k%Ts

- **Método da multiplicação**

  Consiste em escolher um número real entre 0 e 1 e definir h(k) como floor(m*frac(c,k)), onde m é o tamanho da tabela, frac() é a parte fracionada de um número real e flor é a parte inteira.

  Resultados teóricos mostram que valores de c com boas propriedades são
  c = 0,6180339887 ou c = 0,381966011.

- **Método do quadrado médio**

  Consiste em elevar a chave ao quadrado e selecionar alguns dígitos do "meio" do resultado.

- **Método de dobra:** funciona bem para Ts = 2^k

  Converte-se a chave para a sua representação binária. Digamos que o resultado foi 01011 10010 10110 e que k=5. Faremos uma operação de "ou exclusivo" com os blocos de tamanho k.

  01011

  10010

  <u>10110</u>

  01111

- **Chaves alfanuméricas**

  O mais comum é transformar cada letra em um número na tabela ASCII e escolher uma função qualquer.
  Outra possibilidade é interpretar cada letra com um digito na base 26
  Exemplo: bola = b o l a
  1*26^3 14*26^2 12*26^1 0*26^0

**Hash extensível**

- A função Hash não varia.
- O número de buckets varia.
- À medida que os buckets se enchem, estes se duplicam, e o diretório de buckets se duplica.
- Se um único bucket duplica, o diretório todo de buckets duplica.
- Dois ponteiros do diretório podem apontar para o mesmo bucket,
- Somente duplicam os buckets que ficam cheios.
- Registros em buckets duplicados podem ser facilmente localizados através do novo ponteiro no diretório de buckets.
- Possui uma melhor ocupação dos buckets, apenas divide o bucket apropriado.
- Possui um custo I/O que o Hash Linear para seleções com igualdade, em caso de distribuição não uniforme.

- **Inserção:** se nível global = d.
  - Calcula h(k);
  - Considera a entrada m do diretório, onde m = número de correspondentes aos d últimos dígitos da representação binária de h(k);
  - Dirige-se ao bucket indicado;
  - Se o bucket estiver cheio e nível local = d:
    - Divide o bucket e duplica o diretório de buckets;
  - Se o bucket estiver cheio e nível local < d:
    - Divide o bucket, mas não duplica diretório.
- **Comparação com Hash estático:**
  - Sem overflow: hash estático tem custo de 1 I/O
  - Com overflow: hash estático tem custo dependente do número de páginas de overflow.
  - Hash extensível: no máximo 2 I/O.
- **Possíveis problemas:**
  - Distribuição tendenciosa dos valores h(k): muitos em um único bucket.
    - Ajustar função h(k) de modo a ter uma distribuição uniforme.
  - Colisão: quando há muitas entradas <k,*> com o mesmo h(k) que não cabem numa página.
    - Páginas de overflow são utilizadas.

**Hash Linear**

- Função hash varia.
- Não há diretório de buckets.
- Pode haver páginas de overflow a medida que os buckets enchem.
- Regularmente, a função hash se modifica e páginas de overflow são realocadas.
- Possui um custo I/O menor que do Hash Extensível, para seleções com igualdade, em caso de distribuição uniforme.

**Colisões**

Maneiras de reduzir o número de colisões:

- Representar a chave numericamente: (Caso a chave não seja numérica). Utilizando a Tabela ASCII, pode-se gerar um código para a chave. Ex:

  LOWELL$$$$$$ = 76 79 87 69 76 76 32 32 32 32 32 32

- Subdividir o número em partes e somar

  Se utilizarmos a palavra LOWELL$$$$$$ e separarmos em pares as partes ficaram da respectiva maneira:

  76 79 | 87 69 | 76 76 | 32 32 | 32 32 | 32 32

  A soma das partes resultará em 33820;

  Agora supondo que possuímos um Hash com 32767 posições está soma resultará em um overflow, pois 33820 > 32767.

  Para contornar este problema, basta encontrar o conjunto que resulte no maior conjunto possível e subtrair este valor do número de índices de seu hash. Por exemplo:

  O maior valor do conjunto de pares de char é 9090, que corresponde ao conjunto "ZZ", logo sua função hash será:

  Hash = Chave%(32767-9090)

- **Utilizar Buckets**

  Armazenar mais de um elemento por chave, aumentando assim a densidade de ocupação e diminuindo o número de colisões. Obs.: O tamanho do bucket deve ser menor que uma trilha.

**Matemática aplicada a HASH**

- Predizer a distribuição de registro

  Para calcular a probabilidade de que um mesmo endereço tenha X registro, utilizamos a fórmula de Poisson.

$$p(x) = \frac{(r/N)^x e^{-r/N}}{x!} \qquad \text{(distribuição de Poisson)}$$

Onde:

N = número de endereços disponíveis

R = número de registros armazenados

- Predizer o Número esperado de endereços com X registro

$$N \cdot p(x)$$

Onde:

N = número de endereços disponíveis

P(x) = A probabilidade de um mesmo endereço tenha X registros associados

- Densidade de Ocupação: Razão entre o número de registro a serem armazenados (r) e o número de espaços de endereçamento disponíveis (N, assumindo um registro por endereço)

$$\frac{r}{N}$$

A densidade de ocupação do Hash Bucket seria de:

$$\frac{r}{b.N}$$

Onde b representa a capacidade do Bucket.