Técnicas de Programação Avançada

Série 2 – Tabelas Hashing

Introdução

- Tabela Hash: permite realizar buscas eficientes em um conjunto de elementos armazenados em uma tabela (um arranjo) sem necessitar ordenar os elementos do arranjo.
- Mecanismos de inserção e recuperação do elemento
  - Dado um par {chave, elemento}, onde chave é a chave de busca e elemento é o elemento a ser armazenado:
    - Inserção:
      - Utilizando a chave, calcula-se o índice da posição na tabela onde o elemento deve ser inserido.
      - Inserir o elemento na posição calculada.
    - Recuperação
      - Utilizando a chave, calculo o índice onde o elemento está armazenado
      - Obtenho o elemento na posição do índice calculado.

Hashing

- Os registros armazenados em uma tabela são diretamente endereçados a partir de uma transformação aritmética sobre a chave de pesquisa.
- Hash significa:
  - Fazer picadinho de carne e vegetais para cozinhar.
  - Fazer uma bagunça. (Webster's New World Dictionary).
  - Espalhar x Transformar

Hashing – Funcionamento

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

Transformação de Chave

- O paradoxo do aniversário (Feller,1968), 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.
- Assim, se for utilizada uma função de transformação uniforme que enderece 23 chaves randômicas em uma tabela de tamanho 365, a probabilidade de que haja colisões é maior do que 50%.

Transformação de Chave (*Hashing*)

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

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

Transformação de Chave (*Hashing*)

- Alguns valores de p para diferentes valores de N, onde M = 365.

N	P
10	0,883
22	0,524
23	0,493
30	0,303

Para N pequeno a probabilidade p pode ser aproximada por $p \approx N(N-1))/730$. Por exemplo, para N = 10 então p ≈ 87,7%.

Hashtable: exemplo de Hash em Java

```
public class Elemento {
  private String nome;
  private  String telefone;
/* construtor e metodos get e set */ }

public static void main(String[] args) {
      Hashtable tab1 = new
Hashtable();
      Elemento e;
      e = new Elemento();
      e.setNome("Laura");
      e.setTelefone("54");
      tab1.put(e.getNome(),e);
      e = new Elemento();
       e.setNome("Luciana");
       e.setTelefone("33");
       tab1.put(e.getNome(),e);
       System.out.println("Informe o nome:");
       String nome =
InputHandler.readString();
       Elemento en = (Elemento)
tab1.get(nome);
       System.out.println(" O telefone de " +
nome + " eh " + en.getTelefone());
```

Funções 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.

Construindo uma tabela Hash com Vetores

- Um problema é que uma única chave pode gerar o mesmo índice.
  - Este problema é conhecido por colisão
- Por maior que seja a tabela, sempre há risco de colisão.
- Colisões deve ser tratadas

Construindo uma tabela Hash com Vetores

- A utilização de Hashing envolve:
  - Computar a função de transformação
  - Tratar colisões

Função Hash

- Usa o resto da divisão por M .

$$h(K) = K \% M \quad \text{(em linguagem C)}$$

onde K é um inteiro correspondente à chave.

Função Hash

- Cuidado na escolha do valor de M. M deve ser um número primo, mas não qualquer primo: devem ser evitados os números primos obtidos a partir de

$$b*i \pm j$$

onde b é a base do conjunto de caracteres (geralmente b = 64 para BCD, 128 para ASCII, 256 para EBCDIC, ou 100 para alguns códigos decimais), e i e j são pequenos inteiros.

Função Hash

As chaves não numéricas devem ser transformadas em números:

$$K = \sum_{i=1}^{n} \text{Chave}[i] \times p[i],$$

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 $1 \leq i \leq n$.

Transformações de Chaves não numéricas

Vantagem de se usar pesos: Dois conjuntos diferentes de pesos p1[i] e p2[i], $1 \leq i \leq n$, leva a duas funções de transformação h1(K) e h2(K) diferentes.

Transformações de Chaves não numéricas

Programa que gera um peso para cada caractere de uma chave constituída de n caracteres:

```
void GeraPesos(TipoPesos p)
{  /* Gera valores randomicos entre 1 e 10.000 */
  int i;
  struct timeval semente;
  /* Utilizar o tempo como semente para a funcao srand() */
  gettimeofday(&semente,NULL);
  srand((int)(semente.tv_sec + 1000000*semente.tv_usec));
  for (i = 0; i < n; i++)
     p[i] =  1+(int) (10000.0*rand()/(RAND_MAX+1.0));
}
```

Transformações de Chaves não numéricas

Implementação da função de transformação:

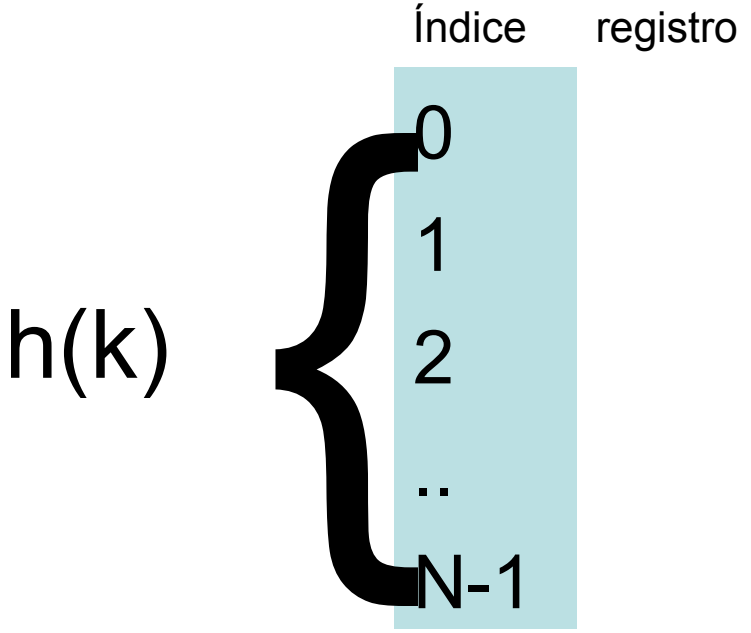
```
Indice h(TipoChave Chave, TipoPesos p)
{
  int i;
  unsigned int Soma = 0;
  int comp = strlen(Chave);

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

  return (Soma % M);
}
```

Construindo uma tabela Hash com Vetores

- O índice na tabela relativa a uma chave é obtida aplicando-se uma transformação aritmética (h) sobre a chave.
- Esta transformação deve gerar um valor que varie de 0 a N-1



Construindo uma tabela Hash com Vetores

- Uma função hash (h) deve:
  - Mapear chaves em inteiros entre 0 e N-1, onde N é o tamanho da tabela.
  - Ser de computação simples
  - Gerar entradas para a tabela com igual probabilidade

Construindo uma tabela Hash com Vetores

- Colisões
  - Ocorrem quando h(ki) = h(kj) para algum ki<>kj
  - Ou seja: duas chaves distintas endereçam para uma mesma entrada na tabela
  - Colisões são praticamente inevitáveis utilizando hashing (exceto utilizando hashing perfeito)

Construindo uma tabela Hash com Vetores

- Exemplo considerando que a chave de pesquisa é um string:

```
Int h (char * chave){
 int i,soma=0;
for(i=0;i<strlen(chave);i++) {
   soma+=abs(chave[i])*(i+1);
 }
 return soma%MAXTAB;
}
```

Construindo uma tabela Hash com Vetores

Considere uma tabela com registros contendo o nome e o telefone de uma pessoa:

```
#define MAX 1000
#define true 1
#define false 0
typedef struct { char nome[20];
                 char tel[20];
                 int ocupado;
                } Elemento;
Elemento tabHash[MAX];

InicializaTabHash(Elemento tabHash[], int size){
Int i;
for(i=0;i<size;i++)
   tabHash[i].ocupado = false;
}
```

Inserindo na Tabela

```
void  create(Elemento tabHash[], Elemento e) {
  int pos;
pos = h(e.nome);
If (tabHash[pos].ocupado == false) {
  tabHash[pos] = e;
  tabHash[pos].ocupado = true;
}
else{
  printf("ocorreu uma colisão");
}
}
```

Construindo uma tabela Hash com Vetores

Recuperando um elemento:

```
Elemento read (Elemento  tabHash[], char *chave) {
int pos;
pos = h(chave);
If (tabHash[pos].ocupado == true) {
    if ( (strcmp(tabHash[pos].nome, chave) ==0)
{
        return tabHash[pos];
   }
}
```

Atualizando um elemento

```
Elemento update (Elemento  tabHash[],Elemento e) {
int pos;
pos = h(e.nome);
If (tabHash[pos].ocupado == true)
    if ( (strcmp(tabHash[pos].nome, chave) ==0)
        tabHash[pos].telefone = e.telefone;
  }
```

Excluindo um elemento

```
Elemento delete (Elemento  tabHash[], char *chave) {
int pos;
pos = h(chave);
If (tabHash[pos].ocupado == true)
    if ( (strcmp(tabHash[pos].nome, chave) ==0)
        tabHash[pos].ocupado = false;
  }
```

Resolvendo colisões

- Listas Encadeadas
- Endereçamento Aberto

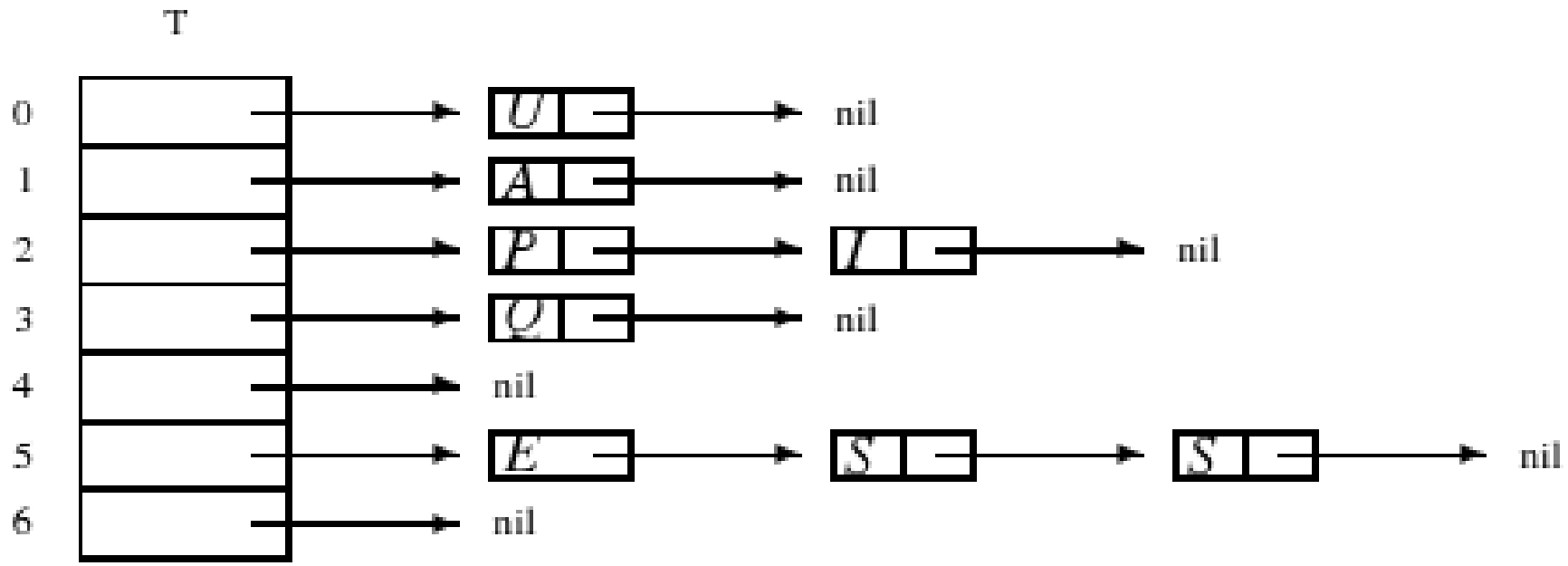
Listas Encadeadas

- Uma das formas de resolver as colisões é simplesmente construir uma lista linear encadeada para cada endereço da tabela. Assim, todas as chaves com mesmo endereço são encadeadas em uma lista linear.

Listas Encadeadas

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 para M = 7, o resultado da inserção das chaves P E S Q U I S A na tabela é o seguinte:

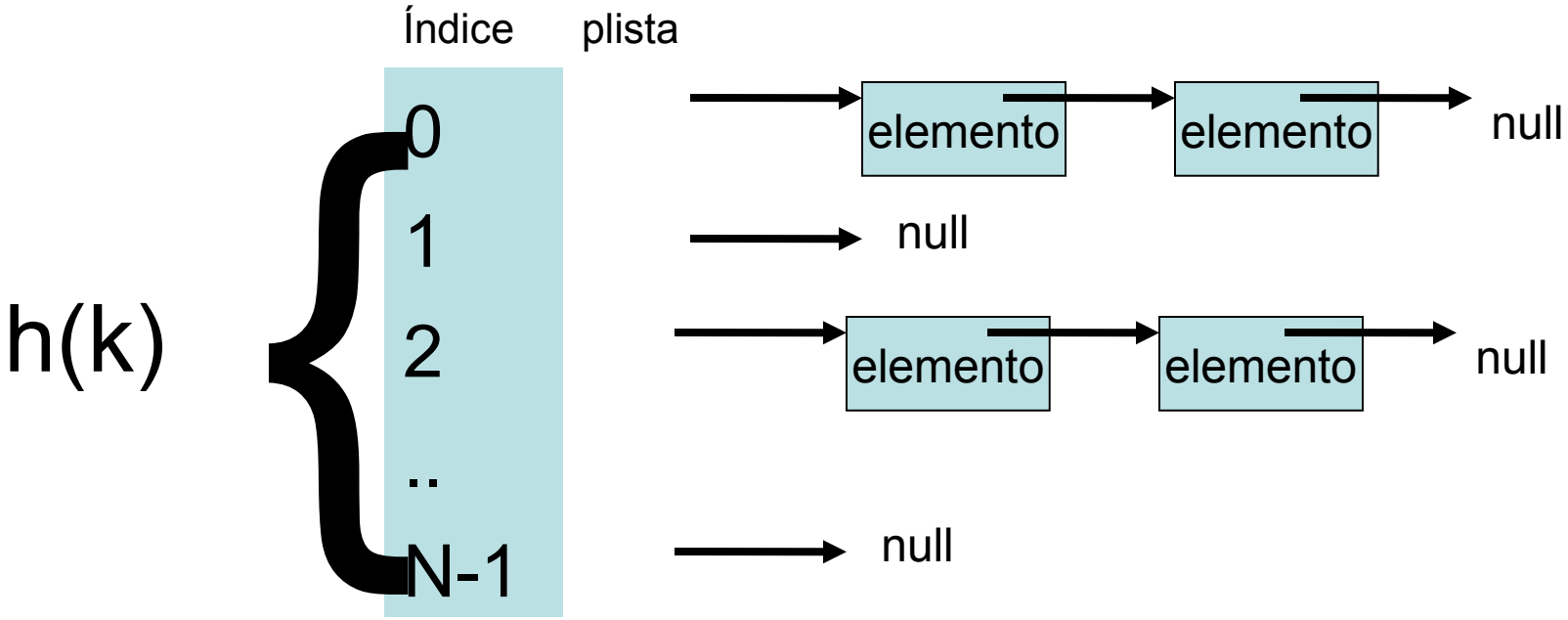
Por exemplo, h(A) = h(1) = 1, h(E) = h(5) = 5, h(S) = h(19) = 5, etc



Tratamento de Colisão

- Solução 1: Cada Entrada da tabela conterá uma lista encadeada
  - Todos os registros com chaves que endereçam para uma mesma entrada na tabela são guardadas em uma mesma lista linear
  - Análise: Considerando que qualquer item tem igual probabilidade de ser endereçado para qualquer entrada da Tabela temos que:
    - O comprimento esperado de cada lista é N/m, onde n é o número de itens e M o tamanho da tabela

Tratamento de Colisão



Tratamento de Colisão -- listas encadeadas

- Estruturas:

```
#define MAX 1000
typedef struct {
        char nome[20];
        char telefone[10];
        } Elemento;
typedef struct noLista{
            Elemento e;
            struct noLista* prox;
            }noLista;
typedef noLista * Lista;

typedef Lista *TabHash;
```

- Implementação das operações de inserção e pesquisa em uma ÚNICA LISTA:

```
void insereNaLista(Lista *l, Elemento e){
    ..
}
Elemento pesquisaNaLista(Lista l, char *chave){
    ..
}
```

Tratamento de Colisão -- listas encadeadas

Implementação das operações da tabela Hash

```
void inicializaTabHash(TabHash tab, int size) {
  int register i;
    *tab =   malloc(size*sizeof(Lista));
    for (i=0;i<size;i++) {
        (*tab)[i] = NULL;
    }
}
// funcao hash. Tomando MAX como tamanho da tabela
int h (char * chave){
int soma=0;
while (*chave) {
   soma+=abs(*chave);
   chave++;
 }
 return soma%MAX;
}
```

- Funções create e read:

```
// função create
void  create(TabHash tab, Elemento e) {
  int pos;
  pos = h(e.nome);
  insereNaLista(&tab[pos], e);
}

// função read
Elemento read (TabHash tab, char *chave) {
 int pos;
 pos = h(chave);
 return pesquisaNaLista(tab[pos],chave);

}
```

Tratamento de Colisão -- listas encadeadas

- Uso de listas encadeadas para tratamento da colisão
  - O acesso a um registro da tabela custa

$O(1+N/M)$

1 Representa o acesso a entrada da tabela

N/M é o tempo para percorrer a lista

Endereçamento Aberto

- Quando o número de registros a serem armazenados na tabela puder ser previamente estimado, então não haverá necessidade de usar apontadores para armazenar os registros.
- Existem vários métodos para armazenar N registros em uma tabela de tamanho $M > N$, os quais utilizam os lugares vazios na própria tabela para resolver as colisões.

Endereçamento Aberto

- Se o número de registros N a ser armazenado puder ser estimado e for menor que M pode-se usar os lugares vazios na tabela para alocar itens em colisão
- Neste método, quando uma chave é endereçada para uma entrada que já esteja ocupada, uma seqüência de localizações alternativas é definida
- Esta seqüência é geralmente obtido na vizinhança da posição calculada pela função Hash.

Endereçamento Aberto

Todas as chaves são armazenadas na própria tabela

Assim, toda posição da tabela ou possui um elemento ou está vazia.

A funções de inserção e busca devem procurar *sistematicamente* a posição associada à chave.

No lugar de seguir ponteiros o lugar a ser procurado é também calculado.

Os ponteiros economizados podem ser usados para aumentar o tamanho da tabela.

Endereçamento Aberto

- **No Endereçamento aberto** todas as chaves são armazenadas na própria tabela, sem o uso de apontadores explícitos.
- Existem várias propostas para a escolha de localizações alternativas. A mais simples é chamada de hashing linear, onde a posição hj na tabela é dada por:
- A escolha de localizações alternativas é chamada de **seqüência de sondagens**

$h_j = (h(x) + j) \bmod M$, para $1 \leq j \leq M - 1$.

Exemplo

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

$$h(Chave) = Chave \bmod M$$

é utilizada para $M = 7$,

- Então o resultado da inserção das chaves

L U N E S na tabela T, usando hashing linear para resolver colisões é:

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.

T

0	U
1	N
2	S
3	
4	
5	L
6	E

Endereçamento Aberto

A Sequência de sondagem é a sequência de posições da tabela determinada para a inserção ou pesquisa de uma chave k.

A sequência de sondagens deve ser uma permutação valida entre as M! Permutações possíveis de endereços da tabela.

O ideal é que a função hash gere sequências igualmente prováveis.

Endereçamento Aberto

Existem várias propostas para a escolha da sequência de sondagem. A mais simples é chamada de hashing linear, onde a posição hj na tabela é dada por:

$hj = (h(x) + j) \bmod M$, para $1 \leq j \leq M - 1$.

Onde h(x) é chamada de hash auxiliar.

Endereçamento aberto

Sondagem quadrática:

- Utiliza uma função hash na forma
- $h(k,i) = (h'(k) + c_1 i + c_2 i2) \bmod M$

- Onde h' é a função hash auxiliar, c1 e c2 != 0 são constantes auxiliares e i = 0,1..M-1
- Funciona melhor que a sondagem linear mas gera seqüências de sondagem sempre iguais quando a h' é igual para duas chaves.

Endereçamento Aberto

Hash duplo

- As sequências de sondagem simulam uma escolha aleatória
- h(k,i) =(h1(k) + I h2(k)) mod m
- onde h1 e h2 são funções hash auxiliares.
- As psições de sondagem sucessivas são deslocadas da posição anterior em função do valor de h2.
- Dessa forma, o deslocamento também depende da chave k.

Endereçamento Aberto

Os Valores de h2(k) e o tamanho da tabela devem ser primos entre si para que toda a tabela possa ser inspecionada (verificar como exercício).

Uma forma de assegurar esta propriedade é fazer m como uma pontência de 2 e projetar h2 para que seja impar. Ou, fazer m um número primo e fazer h2 retornar sempre um inteiro positivo menor que m.

Endereçamento Aberto

Hash duplo

- Ex: $h1(k) = k \bmod m$
- $h2(k) = 1 + (k \bmod m')$
- onde m é primo e m' é um número ligeiramente menor que m.

Endereçamento Aberto

Análise:

- Fator de carga ou ocupação: a = n/m
- No endereçamento aberto, n <=m, logo a é <=1.

- Número de sondagens em uma pesquisa mal sucedida é 1/(1-a).
- Número de sondagens em uma pesquisa bem sucedida é 1/a*ln(1/(1-a).

Análise

Hash Linear:

Conforme demonstrado por Knuth (1973), o número de sondagens com sucesso é:

$$1/2 * (1 + 1/(1-a))$$

- O hashing linear sofre de um mal chamado agrupamento(clustering) (Knuth, 1973, pp.520–521).

- Este fenômeno 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.

Vantagens e Desvantagens de Transformações de Chaves

Vantagens:

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

Desvantagens:

- Custo para recuperar os registros na ordem lexicográfica das chaves é alto, sendo necessário ordenar o arquivo.
- Pior caso é O(N)

Funções Hash

Hash ideal: Uma chave tomada ao acaso pode ser mapeada pela função com probabilidade igual para todas as m posições da tabela.

Situação dificil de se verificar → Não conhecemos a distribuição de probabilidade das chaves.

Obtem-se funções Hash por meio de Heurísticas

Funções Hash

Heurísticas:

- Basear-se em informações qualitativas sobre as chaves.
  - Ex: Tabelas de símbolos de um compilador possui determinados padrões para identificadores. Considerar esses padrões (ex. pt, pts, ptr) minimizando a chance de que os mesmos mapêem em um mesma posição.
- Deriva a função de um modo independente de qualquer padrão que venha a existir nos dados. Ex: Método da divisão.

Funções Hash

Interpretação das chaves como número naturais:

- Alternativa: Interpretar uma cadeia de caracteres considerando sua codificação e a base da codificação.
  - Ex: pt → (112,116) em ASCII (base 128):
    - S = 112*128 + 116 = 14452

Funções Hash

Método da Divisão

- h(k)=k mod m
  - Ex: se m = 12, e k = 100 → h=4
- Método rápido
- Deve-se evitar certos valores de m:
  - m não deve ser uma potência de 2.
  - Se m = 2p h(k) fica limitado aos números formados pelo grupo de p bits de mais baixa ordem da chave k.
  - Uma boa escolha é um número primo não muito próximo de uma potência de 2, pois a transformação irá considerar mais bits de k.

Funções Hash

Ex:

- n :: 2000 (aproximadamente)
- Aceita-se até 3 sondagens em uma pesquisa sem sucesso.
- Tratamento com lista encadeada.
- Então, 701 é uma boa escolha e h(k) = k mod 701

- Método da Multiplicação:

h(k) = (**(k A) mod 1)** m.

k.A mod 1 é a parte fracionária de k.A

0<A<1.

Sugestão de Knuth: A = (raiz(5) -1)/2 = 0,6180

Funções Hash

Hash Universal

- A função hash é escolhida aleatoriamente no início de cada execução, de forma que minimize/evite tendências das chaves
- Por exemplo, h(k)=((A*k + B) % P) % m
  - P é um número primo maior do que a maior chave k
  - A é uma constante escolhida aleatoriamente de um conjunto de constantes {0, 1, 2, ..., P-1} no início da execução
  - B é uma constante escolhida aleatoriamente de um conjunto de constantes {1, 2, ..., P-1} no início da execução
  - Diz-se que h representa uma coleção de funções universal

Funções Hash

Hash perfeito

- Quando não há colisão
- Aplicável em um cenário em que o conjunto de chaves é estático
  - Hashing em 2 níveis
  - No primeiro nível, uma primeira função hash universal é utilizada para encontrar a posição na tabela, sendo que cada posição da tabela contém uma outra tabela (ou seja, outro arranjo)
  - No segundo nível, Uma segunda função hash universal é utilizada para indicar a posição do elemento na tabela associada a posição da tabela do primeiro nível.

Funções Hash

Teorema: Se armazenamos n chaves em uma tabela com m = n2 usando uma função h escolhida ao acaso de uma família de funções hash universal, a probabilidade de haver quaisquer colisões é $< 1/2$.

O resultado deste teorema é usado para determinar o tamanho das tabelas secundárias:

- mj = nj2
- Faz-se o tamanho da tabela do primeiro nível igual a n.

Funções Hash

Realiza-se experiências com as chaves k (conjunto fixo) para assegurar uma escolha da função hash da família hash universal que garanta 0 colisões no segundo nível

- e assim garante-se um número constante de sondagens mesmo no pior caso.