Introdução - Conceitos Básicos

- Conjunto de registros ou arquivos ⇒ tabelas
- **Tabela:**

  associada a entidades de vida curta, criadas na memória interna durante a execução de um programa.
- **Arquivo:**

  geralmente associado a entidades de vida mais longa, armazenadas em memória externa.
- **Distinção não é rígida:**

  tabela: arquivo de índices

  arquivo: tabela de valores de funções.

Introdução - Conceitos Básicos

- Estudo de como recuperar informação a partir de uma grande massa de informação previamente armazenada.
- A informação é dividida em **registros**.
- Cada registro possui uma chave para ser usada na pesquisa.
- **Objetivo da pesquisa:**

  Encontrar uma ou mais ocorrências de registros com chaves iguais à **c**have de pesquisa.
- **Pesquisa com sucesso** X **Pesquisa sem sucesso**.

Conteúdo do Capítulo

5.1 Pesquisa Sequencial

5.2 Pesquisa Binária

5.3 Árvores de Pesquisa

- 5.3.1 Árvores Binárias de Pesquisa sem Balanceamento
- 5.3.2 Árvores Binárias de Pesquisa com Balanceamento
  - ∗ Árvores SBB

5.4 Pesquisa Digital

- 5.4.1 Trie
- 5.4.2 Patricia

5.5 Transformação de Chave (*Hashing*)

- 5.5.1 Funções de Transformação
- 5.5.2 Listas Encadeadas
- 5.5.3 Endereçamento Aberto
- 5.5.4 *Hashing* Perfeito com ordem Preservada
- 5.5.5 *Hashing* Perfeito Usando Espaço Quase Ótimo

Pesquisa em Memória Primária*

Última alteração: 31 de Agosto de 2010

*Transparências elaboradas por Fabiano C. Botelho, Israel Guerra e Nivio Ziviani

Pesquisa Sequencial

- **Método de pesquisa mais simples:** a partir do primeiro registro, pesquise sequencialmente até encontrar a chave procurada; então pare.
- Armazenamento de um conjunto de registros por meio do tipo estruturado arranjo:

```
const MAXN = 10;
type TipoRegistro = record
                      Chave: TipoChave;
                      { outros componentes }
                    end;
     TipoIndice   = 0..MAXN;
     Tipotabela   = record
                      Item: array [TipoIndice] of TipoRegistro;
                      n   : TipoIndice;
                    end;
```


Dicionário

- Nome comumente utilizado para descrever uma estrutura de dados para pesquisa.
- **Dicionário** é um **tipo abstrato de dados** com as operações:
  1. Inicializa
  2. Pesquisa
  3. Insere
  4. Retira
- Analogia com um dicionário da língua portuguesa:
  - Chaves $\Longleftrightarrow$ palavras
  - Registros $\Longleftrightarrow$ entradas associadas com cada palavra:
    - pronúncia
    - definição
    - sinônimos
    - outras informações

Algoritmos de Pesquisa $\Rightarrow$ Tipos Abstratos de Dados

- É importante considerar os algoritmos de pesquisa como **tipos abstratos de dados**, com um conjunto de operações associado a uma estrutura de dados, de tal forma que haja uma independência de implementação para as operações.
- **Operações mais comuns:**
  1. Inicializar a estrutura de dados.
  2. Pesquisar um ou mais registros com determinada chave.
  3. Inserir um novo registro.
  4. Retirar um registro específico.
  5. Ordenar um arquivo para obter todos os registros em ordem de acordo com a chave.
  6. Ajuntar dois arquivos para formar um arquivo maior.

Escolha do Método de Pesquisa mais Adequado a uma Determinada Aplicação

- **Depende principalmente:**
  1. Quantidade dos dados envolvidos.
  2. Arquivo estar sujeito a inserções e retiradas frequentes.

     Se conteúdo do arquivo é estável é importante minimizar o tempo de pesquisa, sem preocupação com o tempo necessário para estruturar o arquivo

Pesquisa Sequencial: Análise

- Pesquisa com sucesso:

$$\begin{aligned} \text{melhor caso} &: C(n) = 1 \\ \text{pior caso} &: C(n) = n \\ \text{caso médio} &: C(n) = (n+1)/2 \end{aligned}$$

- Pesquisa sem sucesso:

$$C'(n) = n + 1.$$

- O algoritmo de pesquisa sequencial é a **melhor escolha** para o problema de pesquisa em tabelas com até **25 registros**.

Pesquisa Sequencial

- Utilização de um registro **sentinela** na posição zero do **array**:
  1. Garante que a pesquisa sempre termina:
     se o índice retornado por Pesquisa for zero, a pesquisa foi sem sucesso.
  2. Não é necessário testar se $i > 0$, devido a isto:
     - o anel interno da função Pesquisa é extremamente simples: o índice i é decrementado e a chave de pesquisa é comparada com a chave que está no registro.
     - isto faz com que esta técnica seja conhecida como **pesquisa sequencial rápida**.

Pesquisa Sequencial

- Pesquisa retorna o índice do registro que contém a chave x;
- Caso não esteja presente, o valor retornado é zero.
- A implementação não suporta mais de um registro com uma mesma chave.
- Para aplicações com esta característica é necessário incluir um argumento a mais na função Pesquisa para conter o índice a partir do qual se quer pesquisar.

Pesquisa Sequencial

```
procedure Inicializa (var T: Tipotabela);
begin T.n := 0;
end; { Inicializa }

function Pesquisa (x: TipoChave; var T: Tipotabela): TipoIndice;
var i: integer;
begin T.Item[0].Chave := x;
      i := T.n + 1;
      repeat i := i - 1; until T.Item[i].Chave = x;
      Pesquisa := i;
end; { Pesquisa }

procedure Insere (Reg: TipoRegistro; var T: Tipotabela);
begin if T.n = MAXN
      then writeln('Erro: tabela cheia')
      else begin T.n := T.n + 1; T.Item[T.n] := Reg; end;
end; { Insere }
```

Árvores de Pesquisa

- A árvore de pesquisa é uma estrutura de dados muito eficiente para armazenar informação.
- Particularmente adequada quando existe necessidade de considerar todos ou alguma combinação de:
  1. Acesso direto e sequencial eficientes.
  2. Facilidade de inserção e retirada de registros.
  3. Boa taxa de utilização de memória.
  4. Utilização de memória primária e secundária.

Pesquisa Binária: Análise

- A cada iteração do algoritmo, o tamanho da tabela é dividido ao meio.
- **Logo:** o número de vezes que o tamanho da tabela é dividido ao meio é cerca de $\log n$.
- **Ressalva:** o custo para manter a tabela ordenada é alto:
  a cada inserção na posição p da tabela implica no deslocamento dos registros a partir da posição p para as posições seguintes.
- Consequentemente, a pesquisa binária não deve ser usada em aplicações muito dinâmicas.

Algoritmo de Pesquisa Binária

```
function Binaria (x: TipoChave; var T: TipoTabela): TipoIndice;
var i, Esq, Dir: TipoIndice;
begin
  if T.n = 0
  then Binaria := 0
  else begin
       Esq := 1; Dir := T.n;
       repeat
         i := (Esq + Dir) div 2;
         if x > T.Item[i].Chave
         then Esq := i+1
         else Dir := i-1;
       until (x = T.Item[i].Chave) or (Esq > Dir);
       if x = T.Item[i].Chave then Binaria := i else Binaria := 0;
       end;
end; { Binaria }
```

Pesquisa para a chave G:

1	2	3	4	5	6	7	8
A	B	C	***D***	E	F	G	H
				E	***F***	G	H
						G	H

Pesquisa Binária

- **Pesquisa em tabela pode ser mais eficiente $\Rightarrow$ Se registros forem mantidos em ordem**
- Para saber se uma chave está presente na tabela
  1. Compare a chave com o registro que está na posição do meio da tabela.
  2. **Se** a chave é menor **então** o registro procurado está na primeira metade da tabela
  3. **Se** a chave é maior **então** o registro procurado está na segunda metade da tabela.
  4. Repita o processo até que a chave seja encontrada, ou fique apenas um registro cuja chave é diferente da procurada, significando uma pesquisa sem sucesso.

Procedimento para Pesquisar na Árvore Uma Chave x

- Compare-a com a chave que está na raiz.
- Se x é menor, vá para a subárvore esquerda.
- Se x é maior, vá para a subárvore direita.
- Repita o processo recursivamente, até que a chave procurada seja encontrada ou um nó folha é atingido.
- Se a pesquisa tiver sucesso o conteúdo retorna no próprio registro x.

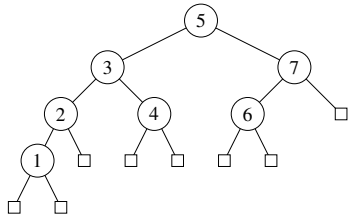
```
procedure Pesquisa (var x: TipoRegistro; var p: TipoApontador);
begin if p = nil
      then writeln ('Erro: TipoRegistro nao esta presente na arvore')
      else if x.Chave < p^.Reg.Chave
           then Pesquisa (x, p^.Esq)
           else if x.Chave > p^.Reg.Chave
                then Pesquisa (x, p^.Dir) else x := p^.Reg;
end; { Pesquisa }
```


Implementação do Tipo Abstrato de Dados Dicionário usando a Estrutura de Dados Árvore Binária de Pesquisa

Estrutura de dados:

```
type TipoChave       = integer;
     TipoRegistro    = record
                         Chave: TipoChave;
                         { outros componentes }
                       end;
     TipoApontador   = ^TipoNo;
     TipoNo          = record
                         Reg: TipoRegistro;
                         Esq, Dir: TipoApontador;
                       end;
     TipoDicionario  = TipoApontador;
```

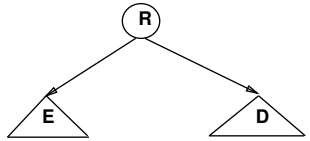

Árvores Binárias de Pesquisa sem Balanceamento



- O **nível** do nó raiz é 0.
- Se um nó está no nível i então a raiz de suas subárvores estão no nível $i + 1$.
- A **altura** de um nó é o comprimento do caminho mais longo deste nó até um nó folha.
- A altura de uma árvore é a altura do nó raiz.

Árvores Binárias de Pesquisa sem Balanceamento

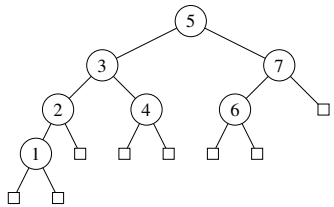
- Para qualquer nó que contenha um registro



Temos a relação invariante

1. Todos os registros com chaves menores estão na subárvore à esquerda.
2. Todos os registros com chaves maiores estão na subárvore à direita.

Exemplo da Retirada de um Registro da Árvore



Assim: para retirar o registro com chave 5 na árvore basta trocá-lo pelo registro com chave 4 ou pelo registro com chave 6, e então retirar o nó que recebeu o registro com chave 5.

Procedimento para Retirar x da Árvore

- Alguns comentários:
  1. A retirada de um registro não é tão simples quanto a inserção.
  2. Se o nó que contém o registro a ser retirado possui no máximo um descendente $\Rightarrow$ a operação é simples.
  3. No caso do nó conter dois descendentes o registro a ser retirado deve ser primeiro:
     - substituído pelo registro mais à direita na subárvore esquerda;
     - ou pelo registro mais à esquerda na subárvore direita.

Procedimentos para Inicializar e Criar a Árvore

```
procedure Inicializa (var Dicionario: TipoDicionario);
begin Dicionario := nil;
end; { Inicializa }

program CriaArvore;
type TipoChave = integer;
{-- Entra aqui a definição dos tipos mostrados no slide 18 --}
var Dicionario: TipoDicionario;
    x           : TipoRegistro;
{-- Entram aqui os procedimentos Insere e Inicializa --}
begin Inicializa (Dicionario);
      read (x.Chave);
      while x.Chave > 0 do begin
        Insere (x, Dicionario);
        read (x.Chave);
        end;
      end.
```


Procedimento para Inserir na Árvore

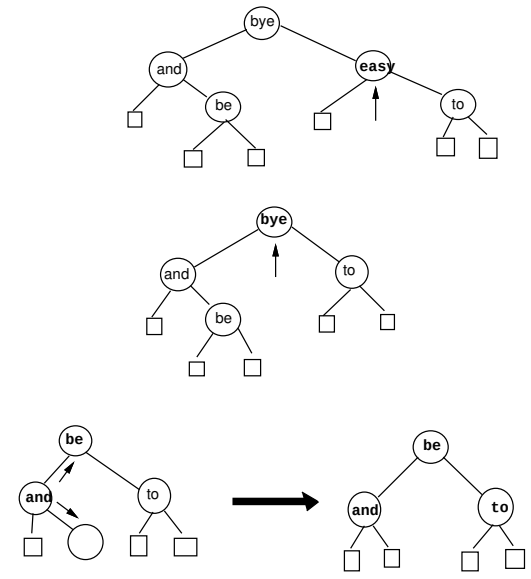
- Atingir um apontador nulo em um processo de pesquisa significa uma pesquisa sem sucesso.
- O apontador nulo atingido é o ponto de inserção.

```
procedure Insere (x: TipoRegistro; var p: TipoApontador);
begin
  if p = nil
  then begin
       new (p);
       p^.Reg := x;  p^.Esq := nil;  p^.Dir := nil;
       end
  else if x.Chave < p^.Reg.Chave
       then Insere (x, p^.Esq)
       else if x.Chave > p^.Reg.Chave
            then Insere (x, p^.Dir)
            else writeln ('Erro: Registro ja existe na arvore')
end; { Insere }
```

Caminhamento Central

- Após construída a árvore, pode ser necessário percorrer todos os registros que compõem a tabela ou arquivo.
- Existe mais de uma ordem de **caminhamento** em árvores, mas a mais útil é a chamada ordem de **caminhamento central**.
- O caminhamento central é mais bem expresso em termos recursivos:
  1. caminha na subárvore esquerda na ordem central;
  2. visita a raiz;
  3. caminha na subárvore direita na ordem central.
- Uma característica importante do caminhamento central é que os nós são visitados de forma ordenada.

Outro Exemplo de Retirada de Nó



Procedimento para Retirar x da Árvore

```
begin {—— Retira——}
  if p = nil
  then writeln ('Erro: Registro nao esta na arvore')
  else if x.Chave < p^.Reg.Chave
       then Retira (x, p^.Esq)
       else if x.Chave > p^.Reg.Chave
            then Retira (x, p^.Dir)
            else if p^.Dir = nil
                 then begin Aux := p;  p := p^.Esq;  dispose(Aux); end
                 else if p^.Esq = nil
                      then begin Aux:=p;  p:=p^.Dir;  dispose(Aux); end
                      else Antecessor (p, p^.Esq);
end; { Retira }
```

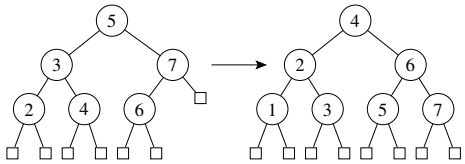
- **Obs.:** proc. recursivo Antecessor só é ativado quando o nó que contém registro a ser retirado possui 2 descendentes. Solução usada por Wirth, 1976, p.211.

Procedimento para Retirar x da Árvore

```
procedure Retira (x: TipoRegistro; var p: TipoApontador);
var Aux: TipoApontador;

  procedure Antecessor (q: TipoApontador; var r: TipoApontador);
  begin
    if r^.Dir <> nil
    then Antecessor (q, r^.Dir)
    else begin
         q^.Reg := r^.Reg;
         q := r;  r := r^.Esq;
         dispose (q)
         end;
  end; { Antecessor }
```

Árvores Binárias de Pesquisa com Balanceamento

- Árvore completamente balanceada ⇒ nós externos aparecem em no máximo dois níveis adjacentes.
- Minimiza tempo médio de pesquisa para uma distribuição uniforme das chaves, onde cada chave é igualmente provável de ser usada em uma pesquisa.
- Contudo, custo para manter a árvore completamente balanceada após cada inserção é muito alto.
- Para inserir a chave 1 na árvore à esquerda e obter a árvore à direita é necessário movimentar todos os nós da árvore original.



Análise

1. Para obter o pior caso basta que as chaves sejam inseridas em ordem crescente ou decrescente. Neste caso a árvore resultante é uma lista linear, cujo número médio de comparações é $(n+1)/2$.
2. Para uma **árvore de pesquisa randômica** o número esperado de comparações para recuperar um registro qualquer é cerca de $1,39 \log n$, apenas 39% pior que a árvore completamente balanceada.

- Uma árvore A com n chaves possui $n+1$ nós externos e estas n chaves dividem todos os valores possíveis em $n+1$ intervalos. Uma inserção em A é considerada *randômica* se ela tem probabilidade igual de acontecer em qualquer um dos $n+1$ intervalos.
- Uma *árvore de pesquisa randômica* com n chaves é uma árvore construida através de n inserções randômicas sucessivas em uma árvore inicialmente vazia.

Análise

- O número de comparações em uma pesquisa com sucesso:

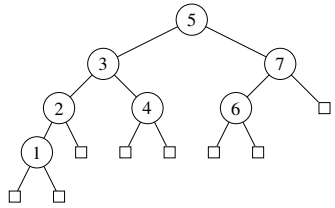
  melhor caso : $C(n) = O(1)$
  pior caso : $C(n) = O(n)$
  caso médio : $C(n) = O(\log n)$

- O tempo de execução dos algoritmos para árvores binárias de pesquisa dependem muito do formato das árvores.

Caminhamento Central

```
procedure Central (p: TipoApontador);
begin
  if p <> nil
  then begin
       Central (p^.Esq);
       writeln (p^.Reg.Chave);
       Central (p^.Dir);
       end;
end; { Central }
```

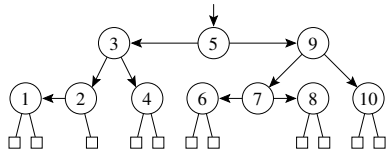
- Percorrer a árvore usando caminhamento central recupera, na ordem: 1, 2, 3, 4, 5, 6, 7.



Árvores SBB

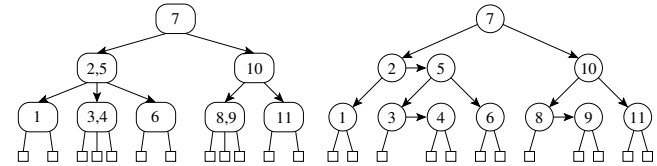
- Árvore 2-3 ⇒ **árvore B binária** (assimetria inerente)
  1. Apontadores à esquerda apontam para um nó no nível abaixo.
  2. Apontadores à direita podem ser verticais ou horizontais.

  Eliminação da assimetria nas árvores B binárias ⇒ árvores B binárias simétricas (*Symmetric Binary B-trees* – SBB)
- **Árvore SBB** tem apontadores verticais e horizontais, tal que:
  1. todos os caminhos da raiz até cada nó externo possuem o mesmo número de apontadores verticais, e
  2. não podem existir dois apontadores horizontais sucessivos.



Árvores SBB

- Árvores B ⇒ estrutura para memória secundária. (Bayer R. e McCreight E.M., 1972)
- **Árvore 2-3** ⇒ caso especial da árvore B.
- Cada nó tem duas ou três subárvores.
- Mais apropriada para memória primária.
- **Exemplo:** Uma árvore 2-3 e a árvore B binária correspondente



Uma Forma de Contornar este Problema

- **Comprimento do caminho interno:** corresponde à soma dos comprimentos dos caminhos entre a raiz e cada um dos nós internos da árvore.
- Por exemplo, o comprimento do caminho interno da árvore à esquerda na figura do slide 31 é $8 = (0 + 1 + 1 + 2 + 2 + 2)$.

Uma Forma de Contornar este Problema

- Procurar solução intermediária que possa manter árvore "quase-balanceada", em vez de tentar manter a árvore completamente balanceada.
- **Objetivo:** Procurar obter bons tempos de pesquisa, próximos do tempo ótimo da árvore completamente balanceada, mas sem pagar muito para inserir ou retirar da árvore.
- **Heurísticas:** existem várias heurísticas baseadas no princípio acima.
- Gonnet e Baeza-Yates (1991) apresentam algoritmos que utilizam vários critérios de balanceamento para árvores de pesquisa, tais como restrições impostas:
  - na diferença das alturas de subárvores de cada nó da árvore,
  - na redução do **comprimento do caminho interno**
  - ou que todos os nós externos apareçam no mesmo nível.

Procedimentos Auxiliares para Árvores SBB

```
procedure DD (var Ap: TipoApontador);
var Ap1: TipoApontador;
begin
  Ap1 := Ap^.Dir;          Ap^.Dir := Ap1^.Esq;  Ap1^.Esq := Ap;
  Ap1^.BitD := Vertical; Ap^.BitD := Vertical; Ap := Ap1;
end; { DD }

procedure DE (var Ap: TipoApontador);
var Ap1, Ap2: TipoApontador;
begin
  Ap1 := Ap^.Dir;          Ap2 := Ap1^.Esq;       Ap1^.BitE := Vertical;
  Ap^.BitD := Vertical; Ap1^.Esq := Ap2^.Dir; Ap2^.Dir := Ap1;
  Ap^.Dir := Ap2^.Esq;  Ap2^.Esq := Ap;        Ap := Ap2;
end; { DE }
```


Procedimentos Auxiliares para Árvores SBB

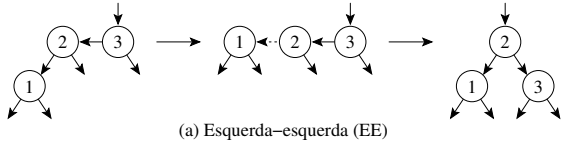
```
procedure EE (var Ap: TipoApontador);
var Ap1: TipoApontador;
begin
  Ap1 := Ap^.Esq;          Ap^.Esq := Ap1^.Dir;  Ap1^.Dir := Ap;
  Ap1^.BitE := Vertical; Ap^.BitE := Vertical; Ap := Ap1;
end; { EE }

procedure ED (var Ap: TipoApontador);
var Ap1, Ap2: TipoApontador;
begin
  Ap1 := Ap^.Esq;          Ap2 := Ap1^.Dir;       Ap1^.BitD := Vertical;
  Ap^.BitE := Vertical; Ap1^.Dir := Ap2^.Esq; Ap2^.Esq := Ap1;
  Ap^.Esq := Ap2^.Dir;  Ap2^.Dir := Ap;        Ap := Ap2;
end; { ED }
```


Estrutura de Dados Árvore SBB para Implementar o Tipo Abstrato de Dados Dicionário

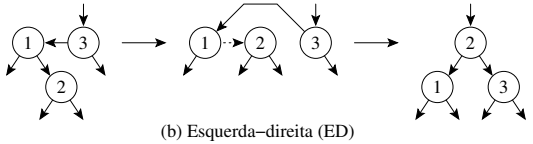
```
type TipoChave        = integer;
     TipoRegistro     = record
                          Chave: TipoChave
                          { outros componentes }
                        end;
     TipoInclinacao = (Vertical, Horizontal);
     TipoApontador  = ^TipoNo;
     TipoNo = record
                Reg        : TipoRegistro;
                Esq, Dir   : TipoApontador;
                BitE, BitD : TipoInclinacao
              end;
     TipoDicionario = TipoApontador;
```


Transformações para Manutenção da Propriedade SBB

- O algoritmo para árvores SBB usa transformações locais no caminho de inserção ou retirada para preservar o balanceamento.
- A chave a ser inserida ou retirada é sempre inserida ou retirada após o apontador vertical mais baixo na árvore.
- Nesse caso podem aparecer dois apontadores horizontais sucessivos, sendo necessário realizar uma transformação:



(a) Esquerda–esquerda (EE)



(b) Esquerda–direita (ED)

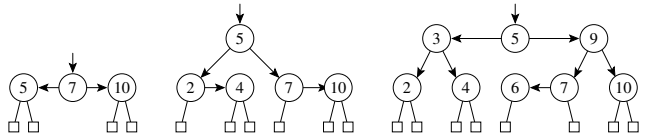
Procedimento Retira

- Retira contém um outro procedimento interno de nome IRetira.
- IRetira usa 3 procedimentos internos:EsqCurto, DirCurto, Antecessor.
  - EsqCurto (DirCurto) é chamado quando um nó folha que é referenciado por um apontador vertical é retirado da subárvore à esquerda (direita) tornando-a menor na altura após a retirada;
  - Quando o nó a ser retirado possui dois descendentes, o procedimento Antecessor localiza o nó antecessor para ser trocado com o nó a ser retirado.

Exemplo

Inserção de uma sequência de chaves em uma árvore SBB:

1. Árvore à esquerda é obtida após a inserção das chaves 7, 10, 5.
2. Árvore do meio é obtida após a inserção das chaves 2, 4 na árvore anterior.
3. Árvore à direita é obtida após a inserção das chaves 9, 3, 6 na árvore anterior.




```
procedure Inicializa (var Dicionario: TipoDicionario);
begin
  Dicionario := nil;
end; { Inicializa }
```


Procedimento para Inserir na Árvore SBB

```
    else
    if x.Chave > Ap^.Reg.Chave
    then begin
         IInsere (x, Ap^.Dir, Ap^.BitD, Fim);
         if not Fim
         then if Ap^.BitD = Horizontal
              then begin
                   if Ap^.Dir^.BitD = Horizontal
                   then begin DD (Ap); IAp := Horizontal; end
                   else if Ap^.Dir^.BitE = Horizontal
                        then begin DE (Ap); IAp := Horizontal; end;
                   end
              else Fim := true;
         end
    else begin
         writeln ('Erro: Chave ja esta na arvore');
         Fim := true;
         end;
  end; { IInsere }
begin { Insere }
  IInsere (x, Ap, IAp, Fim);
end; { Insere }
```


Procedimento para Inserir na Árvore SBB

```
procedure Insere (x: TipoRegistro; var Ap: TipoApontador);
var Fim: boolean; IAp: TipoInclinacao;
  procedure IInsere(x: TipoRegistro; var Ap: TipoApontador; var IAp: TipoInclinacao; var Fim: boolean);
  begin
    if Ap = nil
    then begin
         new (Ap); IAp := Horizontal;  Ap^.Reg := x;
         Ap^.BitE := Vertical;  Ap^.BitD := Vertical;
         Ap^.Esq := nil;  Ap^.Dir := nil;  Fim := false;
         end
    else
    if x.Chave < Ap^.Reg.Chave
    then begin
         IInsere (x, Ap^.Esq, Ap^.BitE, Fim);
         if not Fim
         then if Ap^.BitE = Horizontal
              then begin
                   if Ap^.Esq^.BitE = Horizontal
                   then begin EE (Ap); IAp := Horizontal; end
                   else if Ap^.Esq^.BitD = Horizontal
                        then begin ED (Ap); IAp := Horizontal; end;
                   end
              else Fim := true;
         end
```

Procedimento para Retirar da Árvore SBB

```
begin { IRetira }
  if Ap = nil then begin writeln ('Chave nao esta na arvore'); Fim := true; end
  else if x.Chave < Ap^.Reg.Chave
       then begin IRetira (x, Ap^.Esq, Fim); if not Fim then EsqCurto (Ap, Fim); end
       else if x.Chave > Ap^.Reg.Chave
       then begin IRetira (x, Ap^.Dir, Fim); if not Fim then DirCurto (Ap, Fim); end
       else begin { Encontrou chave }
            Fim := false; Aux := Ap;
            if Aux^.Dir = nil
            then begin Ap := Aux^.Esq; dispose (Aux); if Ap <> nil then Fim := true; end
            else if Aux^.Esq = nil
                 then begin Ap := Aux^.Dir; dispose (Aux); if Ap <> nil then Fim := true; end
                 else begin Antecessor (Aux, Aux^.Esq, Fim); if not Fim then EsqCurto (Ap, Fim); end;
            end;
end; { IRetira }
begin { Retira }
  IRetira (x, Ap, Fim)
end; { Retira }
```


Procedimento para Retirar da Árvore SBB – Antecessor

```
procedure Antecessor (q: TipoApontador; var r: TipoApontador; var Fim: boolean);
begin
  if r^.Dir <> nil
  then begin
       Antecessor (q, r^.Dir, Fim);
       if not Fim then DirCurto (r, Fim);
       end
  else begin
       q^.Reg := r^.Reg; q := r;
       r := r^.Esq;      dispose (q);
       if r <> nil then Fim := true;
       end;
end; { Antecessor }
```


Procedimento para Retirar da Árvore SBB – DirCurto

```
procedure DirCurto (var Ap: TipoApontador; var Fim: boolean);
var Ap1: TipoApontador;
begin { Folha direita retirada => arvore curta na altura direita }
  if Ap^.BitD = Horizontal
  then begin Ap^.BitD := Vertical; Fim := true; end
  else if Ap^.BitE = Horizontal
       then begin
            Ap1:=Ap^.Esq; Ap^.Esq:=Ap1^.Dir; Ap1^.Dir:=Ap; Ap:=Ap1;
            if Ap^.Dir^.Esq^.BitD = Horizontal
            then begin ED (Ap^.Dir); Ap^.BitD := Horizontal; end
            else if Ap^.Dir^.Esq^.BitE = Horizontal
                 then begin EE (Ap^.Dir); Ap^.BitD := Horizontal; end;
            Fim := true;
            end
       else begin
            Ap^.BitE := Horizontal;
            if Ap^.Esq^.BitD = Horizontal
            then begin ED (Ap); Fim := true; end
            else if Ap^.Esq^.BitE = Horizontal then begin EE (Ap); Fim := true; end;
            end;
end; { DirCurto }
```


Procedimento para Retirar da Árvore SBB

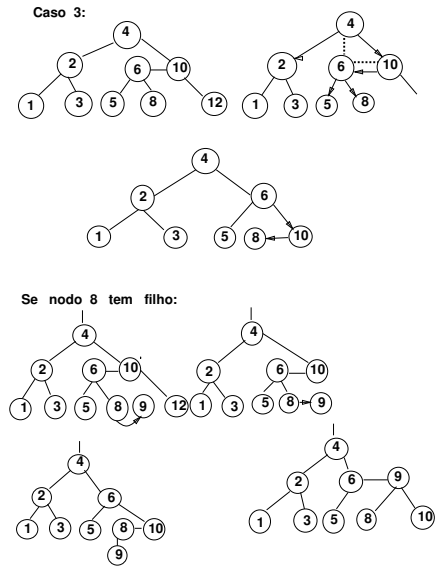
```
procedure Retira (x: TipoRegistro; var Ap: TipoApontador);
var Fim: boolean;
procedure IRetira(x:TipoRegistro;var Ap:TipoApontador;var Fim:boolean);
var Aux: TipoApontador;
procedure EsqCurto (var Ap: TipoApontador; var Fim: boolean);
var Ap1: TipoApontador;
begin { Folha esquerda retirada => arvore curta na altura esquerda }
  if Ap^.BitE = Horizontal
  then begin Ap^.BitE := Vertical; Fim := true; end
  else if Ap^.BitD = Horizontal
       then begin
            Ap1:=Ap^.Dir; Ap^.Dir:=Ap1^.Esq; Ap1^.Esq:=Ap; Ap:=Ap1;
            if Ap^.Esq^.Dir^.BitE = Horizontal
            then begin DE (Ap^.Esq); Ap^.BitE := Horizontal; end
            else if Ap^.Esq^.Dir^.BitD = Horizontal
                 then begin DD (Ap^.Esq); Ap^.BitE := Horizontal; end;
            Fim := true;
            end
       else begin
            Ap^.BitD := Horizontal;
            if Ap^.Dir^.BitE = Horizontal
            then begin DE (Ap); Fim := true; end
            else if Ap^.Dir^.BitD = Horizontal then begin DD (Ap); Fim := true; end;
            end;
end; { EsqCurto }
```

Análise

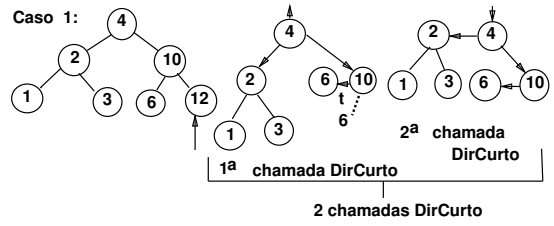
- Nas árvores SBB é necessário distinguir dois tipos de **alturas**:
  1. **Altura vertical** $h \rightarrow$ necessária para manter a altura uniforme e obtida através da contagem do número de apontadores verticais em qualquer caminho entre a raiz e um nó externo.
  2. **Altura** $k \rightarrow$ representa o número máximo de comparações de chaves obtida através da contagem do número total de apontadores no maior caminho entre a raiz e um nó externo.
- A altura k é maior que a altura h sempre que existirem apontadores horizontais na árvore.
- Para uma árvore SBB com n nós internos, temos que

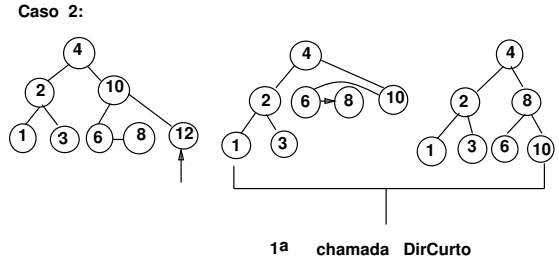
$$h \leq k \leq 2h.$$

Exemplo: Retirada de Nós da Árvore SBB

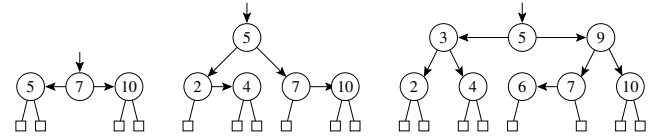


Exemplo: Retirada de Nós da Árvore SBB

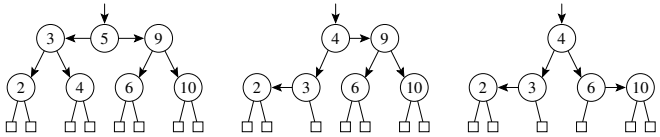




Exemplo



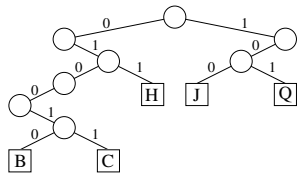
- A árvore à esquerda abaixo é obtida após a retirada da chave 7 da árvore à direita acima.
- A árvore do meio é obtida após a retirada da chave 5 da árvore anterior.
- A árvore à direita é obtida após a retirada da chave 9 da árvore anterior.



Exemplo

- Dada as chaves de 6 bits:

$$\begin{aligned} \mathrm{B} &= 010010 \\ \mathrm{C} &= 010011 \\ \mathrm{H} &= 011000 \\ \mathrm{J} &= 100001 \\ \mathrm{M} &= 101000 \end{aligned}$$



Trie

- Uma trie é uma árvore M-ária cujos nós são vetores de M componentes com campos correspondentes aos dígitos ou caracteres que formam as chaves.
- Cada nó no nível i representa o conjunto de todas as chaves que começam com a mesma sequência de i dígitos ou caracteres.
- Este nó especifica uma ramificação com M caminhos dependendo do $(i+1)$-ésimo dígito ou caractere de uma chave.
- Considerando as chaves como sequência de bits (isto é, $M = 2$), o algoritmo de pesquisa digital é semelhante ao de pesquisa em árvore, exceto que, em vez de se caminhar na árvore de acordo com o resultado de comparação entre chaves, caminha-se de acordo com os bits de chave.

Pesquisa Digital

- Pesquisa digital é baseada na representação das chaves como uma sequência de caracteres ou de dígitos.
- Os métodos de pesquisa digital são particularmente vantajosos quando as chaves são grandes e de **tamanho variável**.
- Um aspecto interessante quanto aos métodos de pesquisa digital é a possibilidade de localizar todas as ocorrências de uma determinada cadeia em um texto, com tempo de resposta logarítmico em relação ao tamanho do texto.
  - **Trie**
  - **Patrícia**

Análise

- De fato Bayer (1972) mostrou que

$$\log(n+1) \leq k \leq 2\log(n+2) - 2.$$

- Custo para manter a propriedade SBB $\Rightarrow$ Custo para percorrer o caminho de pesquisa para encontrar a chave, seja para inserí-la ou para retirá-la.
- **Logo:** O custo é $O(\log n)$.
- Número de comparações em uma pesquisa com sucesso é:

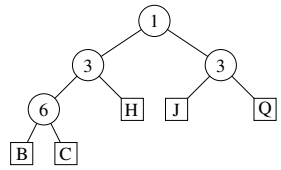
  melhor caso : $C(n) = O(1)$
  pior caso : $C(n) = O(\log n)$
  caso médio : $C(n) = O(\log n)$

- **Observe:** Na prática o caso médio para C_n é apenas cerca de 2% pior que o C_n para uma árvore completamente balanceada, conforme mostrado em Ziviani e Tompa (1982).

Mais sobre Patricia

- O algoritmo para construção da árvore Patricia é baseado no método de pesquisa digital, mas sem o inconveniente citado para o caso das tries.
- O problema de caminhos de uma só direção é eliminado por meio de uma solução simples e elegante: cada nó interno da árvore contém o índice do bit a ser testado para decidir qual ramo tomar.
- **Exemplo:** dada as chaves de 6 bits:

$$\begin{aligned} B &= 010010 \\ C &= 010011 \\ H &= 011000 \\ J &= 100001 \\ Q &= 101000 \end{aligned}$$



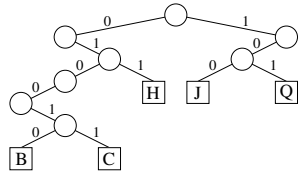
Patricia - Practical Algorithm To Retrieve Information Coded In Alphanumeric

- Criado por Morrison D. R. 1968 para aplicação em recuperação de informação em arquivos de grande porte.
- Knuth D. E. 1973 $\rightarrow$ novo tratamento algoritmo.
- Reapresentou-o de forma mais clara como um caso particular de pesquisa digital, essencialmente, um caso de árvore trie binária.
- Sedgewick R. 1988 apresentou novos algoritmos de pesquisa e de inserção baseados nos algoritmos propostos por Knuth.
- Gonnet, G.H e Baeza-Yates R. 1991 propuzeram também outros algoritmos.

Considerações Importantes sobre as Tries

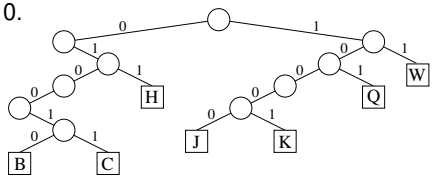
- O formato das tries, diferentemente das árvores binárias comuns, não depende da ordem em que as chaves são inseridas e sim da estrutura das chaves através da distribuição de seus bits.
- **Desvantagem:**
  - Uma grande desvantagem das tries é a formação de caminhos de uma só direção para chaves com um grande número de bits em comum.
  - **Exemplo:** Se duas chaves diferirem somente no último bit, elas formarão um caminho cujo comprimento é igual ao tamanho delas, não importando quantas chaves existem na árvore.
  - Caminho gerado pelas chaves B e C.

Inserção das Chaves W e K na Trie Binária



Faz-se uma pesquisa na árvore com a chave a ser inserida. Se o nó externo em que a pesquisa terminar for vazio, cria-se um novo nó externo nesse ponto contendo a nova chave. Exemplo: a inserção da chave W = 110110.

Se o nó externo contiver uma chave cria-se um ou mais nós internos cujos descendentes conterão a chave já existente e a nova chave. Exemplo: inserção da chave K = 100010.



Funções Auxiliares

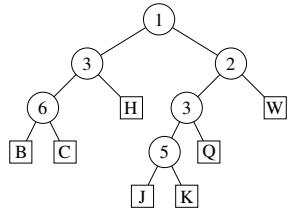
```
function Bit (i: TipoIndexAmp; k: TipoChave): TipoDib;
{ Retorna o i–esimo bit da chave k a partir da esquerda }
var c, j: integer;
begin
  if i = 0
  then Bit := 0
  else begin
       c := ord (k);
       for j := 1 to D − i do c := c div 2;
       Bit := c mod 2;
       end;
end; { Bit }

function EExterno (p: TipoArvore): boolean;
begin { Verifica se p^ nodo externo }
  EExterno := p^.nt = Externo;
end; { EExterno }
```


Estrutura de Dados

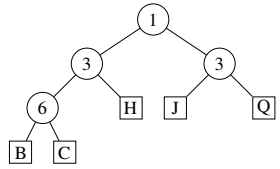
```
const D = 8; { depende de TipoChave }
type TipoChave     = char; { a definir, dependendo da aplicacao }
     TipoIndexAmp  = 0..D;
     TipoDib       = 0..1;
     TipoNo        = (Interno, Externo);
     TipoArvore    = ^TipoPatNo;
     TipoPatNo     = record
                       case nt: TipoNo of
                         Interno:(Index:TipoIndexAmp; Esq, Dir:TipoArvore);
                         Externo:(Chave:TipoChave);
                     end;
```

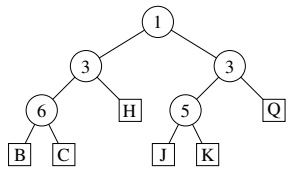

Inserção da Chave W

- A inserção da chave W = 110110 ilustra um outro aspecto.
- Os bits das chaves K e W são comparados a partir do primeiro para determinar em qual índice eles diferem (nesse casod os de índice 2).
- **Portanto:** o ponto de inserção agora será no caminho de pesquisa entre os nós internos de índice 1 e 3.
- Cria-se aí um novo nó interno de índice 2, cujo descendente direito é um nó externo contendo W e cujo descendente esquerdo é a subárvore de raiz de índice 3.



Inserção da Chave K





- Para inserir a chave K = 100010 na árvore à esquerda, a pesquisa inicia pela raiz e termina quando se chega ao nó externo contendo J.
- Os índices dos bits nas chaves estão ordenados da esquerda para a direita. Bit de índice 1 de K é 1 → a subárvore direita Bit de índice 3 → subárvore esquerda que neste caso é um **nó externo**.
- Chaves J e K mantêm o padrão de bits 1x0xxx, assim como qualquer outra chave que seguir este caminho de pesquisa.
- Novo nó interno repõe o nó J, e este com nó K serão os nós externos descendentes.
- O índice do novo nó interno é dado pelo 1° bit diferente das 2 chaves em questão, que é o bit de índice 5. Para determinar qual será o descendente esquerdo e o direito, verifique o valor do bit 5 de ambas as chaves.

Descrição Informal do Algoritmo de Inserção

- Continuação:
  3. Se a raiz da subárvore corrente for um nó interno, vai-se para a subárvore indicada pelo bit da chave k de índice dado pelo nó corrente, de forma recursiva.
  4. Depois são criados um nó interno e um nó externo: o primeiro contendo o índice i e o segundo, a chave k. A seguir, o nó interno é ligado ao externo pelo apontador de subárvore esquerda ou direita, dependendo se o bit de índice i da chave k seja 0 ou 1, respectivamente.
  5. O caminho de inserção é percorrido novamente de baixo para cima, subindo com o par de nós criados no Passo 4 até chegar a um nó interno cujo índice seja menor que o índice i determinado no Passo 2. Este é o ponto de inserção e o par de nós é inserido.

Descrição Informal do Algoritmo de Inserção

- Cada chave k é inserida de acordo com os passos abaixo, partindo da raiz:
  1. Se a subárvore corrente for vazia, então é criado um nó externo contendo a chave k (isto ocorre somente na inserção da primeira chave) e o algoritmo termina.
  2. Se a subárvore corrente for simplesmente um nó externo, os bits da chave k são comparados, a partir do bit de índice imediatamente após o último índice da sequência de índices consecutivos do caminho de pesquisa, com os bits correspondentes da chave k' deste nó externo até encontrar um índice i cujos bits difiram. A comparação dos bits a partir do último índice consecutivo melhora consideravelmente o desempenho do algoritmo. Se todos forem iguais, a chave já se encontra na árvore e o algoritmo termina; senão, vai-se para o Passo 4.

Algoritmo de Pesquisa

```
procedure Pesquisa (k: TipoChave; t: TipoArvore);
begin
  if EExterno (t)
  then if k = t^.Chave
       then writeln ('Elemento encontrado')
       else writeln ('Elemento nao encontrado')
  else if Bit (t^.Index, k) = 0
       then Pesquisa (k, t^.Esq)
  else Pesquisa (k, t^.Dir)
end; { Pesquisa }
```


Procedimentos para Criar Nós Interno e Externo

```
function CriaNodoInt(i: integer; var Esq, Dir: TipoArvore): TipoArvore;
var p: TipoArvore;
begin
  new (p, Interno);
  p^.nt := Interno;
  p^.Esq := Esq; p^.Dir := Dir;
  p^.Index := i; CriaNodoInt := p;
end; { CriaNodoInt }

function CriaNodoExt (k: TipoChave): TipoArvore;
var p: TipoArvore;
begin
  new (p, Externo);
  p^.nt := Externo;
  p^.Chave := k;
  CriaNodoExt := p;
end; { CriaNodoExt }
```

Transformação de Chave (*Hashing*)

- 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 (*Hashing*)

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

Algoritmo de inserção

```
begin
  if t = nil
  then Insere := CriaNodoExt (k)
  else begin
       p := t;
       while not EExterno (p) do
         begin if Bit (p^.Index, k) = 1 then p := p^.Dir else p := p^.Esq; end;
       i := 1; { acha o primeiro bit diferente }
       while (i <= D) and (Bit (i, k) = Bit (i, p^.Chave)) do i := i+1;
       if i > D
       then begin
            writeln ('Erro: chave ja esta na arvore'); Insere := t;
            end
       else Insere := InsereEntre (k, t, i);
  end;
end; { Insere }
```


Algoritmo de inserção

```
function Insere (k: TipoChave; var t: TipoArvore): TipoArvore;
var p: TipoArvore; i: integer;
  function InsereEntre (k: TipoChave; var t: TipoArvore; i: integer): TipoArvore;
  var p: TipoArvore;
  begin
    if EExterno (t) or (i < t^.Index)
    then begin { cria um novo no externo }
         p := CriaNodoExt (k);
         if Bit (i, k) = 1
         then InsereEntre := CriaNodoInt (i, t, p)
         else InsereEntre := CriaNodoInt (i, p, t);
         end
    else begin
         if Bit (t^.Index, k) = 1
         then t^.Dir := InsereEntre (k, t^.Dir, i)
         else t^.Esq := InsereEntre (k, t^.Esq, i);
         InsereEntre := t;
         end;
  end; { InsereEntre }
```

Método mais Usado

- Usa o resto da divisão por M.

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

  onde K é um inteiro correspondente à chave.

- **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.

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:**
  1. Seja simples de ser computada.
  2. Para cada chave de entrada, qualquer uma das saídas possíveis é igualmente provável de ocorrer.

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 $p \approx \frac{N(N-1))}{730}$. Por exemplo, para $N = 10$ então $p \approx 87,7\%$.

Transformação de Chave (*Hashing*)

- 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.
- 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%.
- 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 Chaves Não Numéricas: Nova Versão

Implementação da função *hash* de Zobrist:

- Para obter h é necessário o mesmo número de adições da função do programa anterior, mas nenhuma multiplicação é efetuada.
- Isso faz com que h seja computada de forma mais eficiente.
- Nesse caso, a quantidade de espaço para armazenar h é $O(n \times |\Sigma|)$, onde $|\Sigma|$ representa o tamanho do alfabeto, enquanto que para a função do programa anterior é $O(n)$.

```
function h (Chave: TipoChave; p: TipoPesos): TipoIndice;
var i, Soma: integer; { Funcao h do Zobrist}
begin
  Soma := 0;
  for i := 1 to N do Soma := Soma + p[i, ord(Chave[i])];
  h := Soma mod M;
end; { h }
```


Transformação de Chaves Não Numéricas: Nova Versão

- Modificação no cálculo da função h para evitar a multiplicação da representação ASCII de cada caractere pelos pesos (Zobrist 1990).
  - Este é um caso típico de troca de espaço por tempo.
- Um peso diferente é gerado randomicamente para cada um dos 256 caracteres ASCII possíveis na $i-$ésima posição da chave, para $1 \leq i \leq n$.

```
const TAMALFABETO = 256;
type TipoPesos = array[1..N, 1..TAMALFABETO] of integer;
procedure GeraPesos (var p: TipoPesos);
var i, j: integer;
begin
  randomize;
  for i:= 1 to N do
    for j := 1 to TAMALFABETO do
      p[i, j] := trunc(1000000 * random + 1);
end; {GeraPesos}
```


Transformação de Chaves Não Numéricas

```
type TipoPesos = array [1..N] of integer;
procedure GeraPesos (var p: TipoPesos);
var i: integer;
begin
  randomize;
  for i:= 1 to N do p[i] := trunc (1000000 * random + 1);
end;

type TipoChave  = packed array [1..N] of char;
     TipoIndice = 0..M - 1;
function h (Chave: TipoChave; p: TipoPesos): TipoIndice;
var i, Soma: integer;
begin
  Soma := 0;
  for i := 1 to N do Soma := Soma + ord (Chave[i]) * p[i];
  h := Soma mod M;
end; { h }
```


Transformação de Chaves Não Numéricas

- 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 $1 \leq i \leq n$.
- Vantagem de usar pesos: Dois conjuntos diferentes de $p_1[i]$ e $p_2[i]$, $1 \leq i \leq n$, leva a duas funções $h_1(K)$ e $h_2(K)$ diferentes.

Operações do Dicionário Usando Listas Encadeadas

```
procedure Insere (x: TipoItem; var p: TipoPesos; var T: TipoDicionario);
begin
  if Pesquisa (x.Chave, p, T) = nil
  then Ins (x, T[h(x.Chave, p)])
  else writeln ('Registro ja esta presente')
end; { Insere }

procedure Retira (x: TipoItem; var p: TipoPesos; var T: TipoDicionario);
var Ap: TipoApontador;
begin
  Ap := Pesquisa (x.Chave, p, T);
  if Ap = nil
  then writeln ('Registro nao esta presente')
  else Ret (Ap, T[h(x.Chave, p)], x)
end; { Retira }
```


Operações do Dicionário Usando Listas Encadeadas

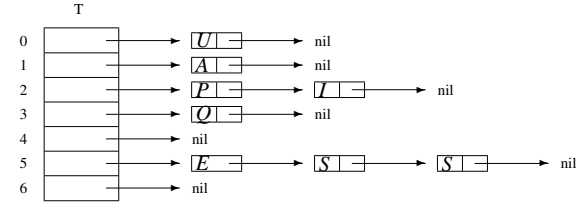
```
procedure Inicializa (var T: TipoDicionario);
var i: integer;
begin for i := 0 to M − 1 do FLVazia (T[i]) end;

function Pesquisa (Ch: TipoChave; var p: TipoPesos; var T: TipoDicionario): TipoApontador;
{—Obs.: Apontador de retorno aponta para o item anterior da lista—}
var i : TipoIndice; Ap: TipoApontador;
begin
  i := h (Ch, p);
  if Vazia (T[i])
  then Pesquisa := nil { Pesquisa sem sucesso }
  else begin
       Ap := T[i].Primeiro;
       while (Ap^.Prox^.Prox <> nil) and (Ch <> Ap^.Prox^.Item.Chave) do Ap := Ap^.Prox;
       if Ch = Ap^.Prox^.Item.Chave
       then Pesquisa := Ap
       else Pesquisa := nil { Pesquisa sem sucesso }
       end
end; { Pesquisa }
```


Estrutura do Dicionário para Listas Encadeadas

```
type TipoChave      = packed array [1..N] of char;
     TipoItem       = record
                        Chave: TipoChave
                        { outros componentes }
                      end;
     TipoIndice     = 0..M − 1;
     TipoApontador  = ^TipoCelula;
     TipoCelula     = record
                        Item: TipoItem;
                        Prox: TipoApontador
                      end;
     TipoLista      = record
                        Primeiro: TipoApontador;
                        Ultimo  : TipoApontador
                      end;
     TipoDicionario = array [TipoIndice] of TipoLista;
```


Listas Encadeadas

- Uma das formas de resolver as **colisões** é construir uma lista linear encadeada para cada endereço da tabela. Assim, todas as chaves com mesmo endereço são encadeadas em uma lista linear.
- **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 $M = 7$, o resultado da inserção das chaves $P\ E\ S\ Q\ U\ I\ S\ A$ na tabela é o seguinte:
  - $h(A) = h(1) = 1$, $h(E) = h(5) = 5$, $h(S) = h(19) = 5$, e assim por diante.



Estrutura do Dicionário Usando Endereçamento Aberto

```
const VAZIO = '!!!!!!!!!!';
      RETIRADO = '**********';
      M = 7;
      N = 10; { Tamanho da chave }

type TipoApontador   = integer;
     TipoChave       = packed array [1..N] of char;
     TipoItem        = record
                         Chave: TipoChave
                         { outros componentes }
                       end;
     TipoIndice      = 0..M − 1;
     TipoDicionario  = array [TipoIndice] of TipoItem;
```


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, usando *hashing linear* para resolver colisões é mostrado abaixo.
- Por exemplo, $h(L) = h(12) = 5$, $h(U) = h(21) = 0$, $h(N) = h(14) = 0$, $h(E) = h(5) = 5$, e $h(S) = h(19) = 5$.

T

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

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**. (Knuth, 1973, p.518)
- 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 h_j na tabela é dada por:

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

Análise

- Assumindo que qualquer item do conjunto tem igual probabilidade de ser endereçado para qualquer entrada de T, então o comprimento esperado de cada lista encadeada é N/M, onde N representa o número de registros na tabela e M o tamanho da tabela.
- **Logo:** as operações Pesquisa, Insere e Retira custam $O(1 + N/M)$ operações em média, onde a constante 1 representa o tempo para encontrar a entrada na tabela e N/M o tempo para percorrer a lista. Para valores de M próximos de N, o tempo se torna constante, isto é, independente de N.

Análise

- Seja $\alpha = N/M$ 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**(***clustering***) (Knuth, 1973, pp.520–521).
- Esse 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.
- Entretanto, apesar do *hashing linear* ser um método relativamente pobre para resolver colisões os resultados apresentados são bons.
- O melhor caso, assim como o caso médio, é $O(1)$.

Operações do Dicionário Usando Endereçamento Aberto

```
procedure Retira (Ch: TipoChave; var p: TipoPesos; var T: TipoDicionario);
var i: integer;
begin
  i := Pesquisa (Ch, p, T);
  if i < M
  then T[i].Chave := RETIRADO
  else writeln ('Registro nao esta presente')
end; { Retira }
```


Operações do Dicionário Usando Endereçamento Aberto

```
procedure Insere (x: TipoItem; var p: TipoPesos; var T: TipoDicionario);
var i, Inicial: integer;
begin
  if Pesquisa (x.Chave, p, T) < M
  then writeln ('Elemento ja esta presente')
  else begin
       Inicial := h (x.Chave, p);
       i := 0;
       while ((T[(Inicial + i) mod M].Chave <> VAZIO) and
          (T[(Inicial + i) mod M].Chave <> RETIRADO)) and
          (i < M) do i := i + 1;
       if i < M
       then T[(Inicial + i) mod M] := x
       else writeln (' Tabela cheia')
       end;
end; { Insere }
```


Operações do Dicionário Usando Endereçamento Aberto

```
procedure Inicializa (var T: TipoDicionario);
var i: integer;
begin
  for i := 0 to M − 1 do T[i].Chave := VAZIO
end; { Inicializa }

function Pesquisa (Ch: TipoChave; var p: TipoPesos;
                   var T: TipoDicionario): TipoApontador;
var i, Inicial: integer;
begin
  Inicial := h (Ch, p);  i := 0;
  while (T[(Inicial + i) mod M].Chave <> VAZIO) and
        (T[(Inicial + i) mod M].Chave <> Ch) and (i < M) do i := i + 1;
  if T[(Inicial + i) mod M].Chave = Ch
  then Pesquisa := (Inicial + i) mod M
  else Pesquisa := M { Pesquisa sem sucesso }
end; { Pesquisa }
```

Algoritmo de Czech, Havas e Majewski

- Czech, Havas e Majewski (1992, 1997) propõem um método elegante baseado em **grafos randômicos** para obter uma função de transformação perfeita com ordem preservada.
- A função de transformação é do tipo:

$$hp(x) = (g[h_0(x)] + g[h_1(x)] + \ldots + g[h_{r-1}(x)]) \bmod N,$$

  na qual $h_0(x), h_1(x), \ldots, h_{r-1}(x)$ são r funções não perfeitas descritas pelos programas dos slides 77 ou 79, x é a chave de busca, e g um arranjo especial que mapeia números no intervalo $0 \ldots M-1$ para o intervalo $0 \ldots N-1$.

Vantagens e Desvantagens de Uma Função de Transformação Perfeita Mínima

- Nas aplicações em que necessitamos apenas recuperar o registro com informação relacionada com a chave e a pesquisa é sempre com sucesso, não há necessidade de armazenar a chave, pois o registro é localizado sempre a partir do resultado da função de transformação.
- Não existem colisões e não existe desperdício de espaço pois todas as entradas da tabela são ocupadas. Uma vez que colisões não ocorrem, cada chave pode ser recuperada da tabela com um único acesso.
- Uma função de transformação perfeita é específica para um conjunto de chaves conhecido. Em outras palavras, ela não pode ser uma função genérica e tem de ser pré-calculada.
- A desvantagem no caso é o espaço ocupado para descrever a função de transformação hp.

Hashing Perfeito com Ordem Preservada

- Se $h(x_i) = h(x_j)$ se e somente se $i = j$, então não há colisões, e a função de transformação é chamada de **função de transformação perfeita** ou função *hashing* perfeita(hp).
- Se o número de chaves N e o tamanho da tabela M são iguais ($\alpha = N/M = 1$), então temos uma **função de transformação perfeita mínima**.
- Se $x_i \leq x_j$ e $hp(x_i) \leq hp(x_j)$, então a ordem lexicográfica é preservada. Nesse caso, temos uma **função de transformação perfeita mínima com ordem preservada**.

Vantagens e Desvantagens de Transformação da Chave

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 $O(N)$.

Problema Resolvido Pelo Algoritmo

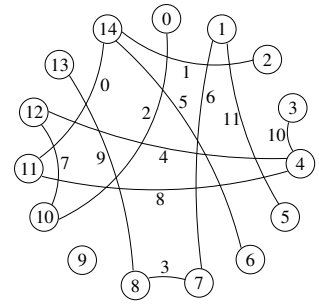
- Um **h**ipergrafo ou $r-$grafo é um grafo não direcionado no qual cada aresta conecta r vértices.
- Dado um hipergrafo não direcionado acíclico $G_r = (V, A)$, onde $|V| = M$ e $|A| = N$, encontre uma atribuição de valores aos vértices de G_r tal que a soma dos valores associados aos vértices de cada aresta tomado módulo N é um número único no intervalo $[0, N-1]$.
- A questão principal é como obter uma função g adequada. A abordagem mostrada a seguir é baseada em hipergrafos acíclicos randômicos.

Exemplo (Obs.: Existe Erro na Tab.5.3(a), pag.205 do livro)

Chave x	$h_0(x)$	$h_1(x)$	$hp(x)$
jan	11	14	0
fev	14	2	1
mar	0	10	2
abr	8	7	3
mai	4	12	4
jun	14	6	5
jul	1	7	6
ago	12	10	7
set	11	4	8
out	8	13	9
nov	3	4	10
dez	1	5	11

- **Chaves:** 12 meses do ano abreviados para os três primeiros caracteres.
- Vamos utilizar um hipergrafo acíclico com $r = 2$ (ou 2-grafo), onde cada aresta conecta 2 vértices.
- Usa duas funções de transformação universais $h_0(x)$ e $h_1(x)$.
- **Objetivo:** obter uma função de transformação perfeita hp de tal forma que o i-ésimo mês é mantido na $(i-1)$-ésima posição da tabela *hash*.

Grafo Acíclico Randômico Gerado



- O problema de obter a função g é equivalente a encontrar um hipergrafo acíclico contendo M vértices e N arestas.
- Os vértices são rotulados com valores no intervalo $0 \ldots M-1$
- Arestas são definidas por $(h_1(x), h_2(x))$ para cada uma das N chaves x.
- Cada chave corresponde a uma aresta que é rotulada com o valor desejado para a função hp perfeita.
- Os valores das duas funções $h_1(x)$ e $h_2(x)$ definem os vértices sobre os quais a aresta é incidente.

Obtenção da Função g a Partir do Grafo Acíclico

Passo importante: conseguir um arranjo g de vértices para inteiros no intervalo $0 \ldots N-1$ tal que, para cada aresta $(h_0(x), h_1(x))$, o valor de $hp(x) = g(h_0(x)) + g(h_1(x))) \bmod N$ seja igual ao rótulo da aresta.

- O primeiro passo é obter um hipergrafo randômico e verificar se ele é acíclico.
- O Programa 7.10 do Capítulo 7 do livro para verificar se um hipergrafo é acíclico é baseado no fato de que um r-grafo é **a**cíclico se e somente se a remoção repetida de arestas contendo vértices de grau 1 elimina todas as arestas do grafo.

Programa para Obter Função de Transformação Perfeita

```
Program ObtemHashingPerfeito;
begin
  Ler conjunto de N chaves;
  Ler o valor de M;
  Ler o valor de r;
  repeat
    Gera os pesos p_0[i], p_1[i], ..., p_{r-1}[i]
      para 1 ≤ i ≤ MAXTAMCHAVE;
    Gera o hipergrafo G_r = (V, A);
    GrafoAciclico (G_r, L, GAciclico)
  until GAciclico;
  Atribuig (G, L, g);
  Retorna p_0[i], p_1[i], ..., p_{r-1}[i] e g;
end.
```

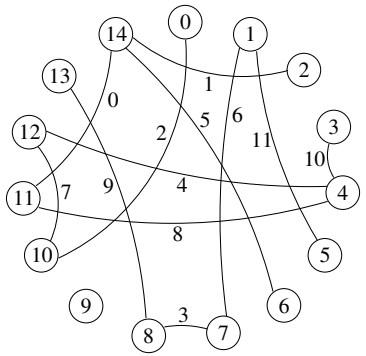
- Gera hipergrafos randômicos iterativamente e testa se o grafo gerado é acíclico.
- Cada iteração gera novas funções $h_0, h_1, \ldots, h_{r-1}$ até que um grafo acíclico seja obtido.
- A função de transformação perfeita é determinada pelos pesos $p_0, p_1, \ldots, p_{r-1}$, e pelo arranjo g.

Rotula Grafo e Atribui Valores para o Arranjo g

```
Procedure Atribuig(var Grafo: TipoGrafo; var L: TipoArranjoArestas; var g: Tipog);
var i, u, Soma: integer; v: TipoValorVertice; a: TipoAresta;
begin
  for i := Grafo.NumVertices − 1 downto 0 do g[i] := INDEFINIDO;
  for i := Grafo.NumArestas − 1 downto 0 do
    begin
    a := L[i];  Soma := 0;
    for v := Grafo.r − 1 downto 0 do
      if g[a.Vertices[v]] = INDEFINIDO
      then begin u := a.Vertices[v]; g[u] := Grafo.NumArestas; end
      else Soma := Soma + g[a.Vertices[v]];
    g[u] := a.Peso − Soma;
    if g[u] < 0 then g[u] := g[u] + (Grafo.r−1) * Grafo.NumArestas;
    end;
end;
```

- Todas as entradas do arranjo g são feitas igual a $Indefinido = -1$.
- Atribua o valor N para $g[v_{j+1}], \ldots, g[v_{r-1}]$ que ainda estão indefinidos e faça $g[v_j] = (i_a - \sum_{v_i \in a \wedge g[v_i] \neq -1} g[v_i]) \bmod N$.

Algoritmo para Obter g no Exemplo dos 12 Meses



Chave x	$h_0(x)$	$h_1(x)$	$hp(x)$
jan	11	14	0
fev	14	2	1
mar	0	10	2
abr	8	7	3
mai	4	12	4
jun	14	6	5
jul	1	7	6
ago	12	10	7
set	11	4	8
out	8	13	9
nov	3	4	10
dez	1	5	11

v	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14
$g[v]$	3	3	1	2	8	8	5	3	12	-1	11	12	8	9	0

Obtenção da Função g a Partir do Grafo Acíclico

Algoritmo:

1. O Programa 7.10 retorna os índices das arestas retiradas no arranjo $\mathcal{L} = (2, 1, 10, 11, 5, 9, 7, 6, 0, 3, 4, 8)$. O arranjo $\mathcal{L}$ indica a ordem de retirada das arestas.
2. As arestas do arranjo $\mathcal{L}$ devem ser consideradas da direita para a esquerda, condição suficiente para ter sucesso na criação do arranjo g.
3. O arranjo g é iniciado com -1 em todas as entradas.
4. A aresta $a = (4, 11)$ de índice $i_a = 8$ é a primeira a ser processada. Como inicialmente $g[4] = g[11] = -1$, fazemos $g[11] = N$ e $g[4] = i_a - g[11] \bmod N = 8 - 12 \bmod 12 = 8$.
5. Para a próxima aresta $a = (4, 12)$ de índice $i_a = 4$, como $g[4] = 8$, temos que $g[12] = i_a - g[4] \bmod N = 4 - 8 \bmod 12 = 8$, e assim sucessivamente até a última aresta de $\mathcal{L}$.

Estruturas de dados (3)

```
{ -- Tipos usados em Fila do Programa 3.17 -- }
TipoItem   = record
               Chave: TipoValorVertice;
               { outros componentes }
             end;
TipoFila   = record
               Item   : array [1..MaxTam] of TipoItem;
               Frente: TipoApontador;
               Tras   : TipoApontador;
             end;
TipoPesos       = array [1..MAXTAMCHAVE] of integer;
TipoTodosPesos = array [Tipor] of Tipopesos;
Tipog           = array[0..MAXNUMVERTICES] of integer;
TipoChave       = packed array[1..MAXTAMCHAVE] of char;
TipoConjChaves = array[0..MAXNUMCHAVES] of TipoChave;
TipoIndice      = TipoValorVertice;
```


Estruturas de dados (1)

```
const
  MAXNUMVERTICES = 100000; {--No. maximo de vertices--}
  MAXNUMARESTAS = 100000; {--No. maximo de arestas--}
  MAXR = 5;
  MAXTAMPROX = MAXR*MAXNUMARESTAS;
  MAXTAM = 1000; {--Usado Fila--}
  MAXTAMCHAVE = 6; {--No. maximo de caracteres da chave--}
  MAXNUMCHAVES = 100000; {--No. maximo de chaves lidas--}
  INDEFINIDO = -1;
type
  {-- Tipos usados em GrafoListaInc do Programa 7.25 --}
  TipoValorVertice     = -1..MAXNUMVERTICES;
  TipoValorAresta      = 0..MAXNUMARESTAS;
  Tipor                = 0..MAXR;
  TipoMaxTamProx       = -1..MAXTAMPROX;
  TipoPesoAresta       = integer;
  TipoArranjoVertices = array[Tipor] of TipoValorVertice;
```


Estruturas de dados (2)

```
TipoAresta          = record
                        Vertices: TipoArranjoVertices;
                        Peso     : TipoPesoAresta;
                        end;
TipoArranjoArestas  = array[TipoValorAresta] of TipoAresta;
TipoGrafo =
  record
    Arestas        : TipoArranjoArestas;
    Prim           : array[TipoValorVertice] of TipoMaxTamProx;
    Prox           : array[TipoMaxTamProx] of TipoMaxTamProx;
    ProxDisponivel: TipoMaxTamProx;
    NumVertices    : TipoValorVertice;
    NumArestas     : TipoValorAresta;
    r              : Tipor;
  end;
TipoApontador = integer;
```


Gera um Grafo sem Arestas Repetidas e sem *Self-Loops*

```
procedure GeraGrafo (var ConjChaves      : TipoConjChaves;
                     N                   : TipoValorAresta;
                     M                   : TipoValorVertice;
                     r                   : Tipor;
                     var Pesos           : TipoTodosPesos;
                     var NgrafosGerados: integer;
                     var Grafo           : TipoGrafo);
{ Gera um grafo sem arestas repetidas e sem selfloops }
var i, j: integer; Aresta: TipoAresta; GrafoValido: boolean;
  function VerticesIguais (Aresta: TipoAresta): boolean;
  var i, j: integer;
  begin VerticesIguais := false;
        for i := 0 to Grafo.r - 2 do
          for j := i + 1 to Grafo.r - 1 do
            if Aresta.Vertices[i] = Aresta.Vertices[j]
            then VerticesIguais := true;
  end;
```

Programa Principal para Gerar Arranjo g (3)

```
write ('Grafo aciclico com arestas retiradas:');
for i := 0 to Grafo.NumArestas - 1 do write (L[i].Peso:3); writeln;
  Atribuig (Grafo, L, g);
  writeln (ArqSaida, N, '  (N)');
  writeln (ArqSaida, M, '  (M)');
  writeln (ArqSaida, r, '  (r)');
  for j := 0 to Grafo.r - 1 do
    begin
    for i := 1 to MAXTAMCHAVE do write (ArqSaida, Pesos[j][i],' ');
    for i := 1 to MAXTAMCHAVE do write (Pesos[j][i],' ');
    writeln (ArqSaida, '  (p',j:1,')'); writeln ('  (p',j:1,')');
    end;
  for i := 0 to M - 1 do write (ArqSaida, g[i],' ');
  for i := 0 to M - 1 do write (g[i],' ');
  writeln (ArqSaida, '  (g)'); writeln ('  (g)');
  writeln (ArqSaida,'No. grafos gerados por GeraGrafo:',NGrafosGerados);
  close (ArqSaida); close (ArqEntrada);
end. { ObtemHashPerfeito }
```


Programa Principal para Gerar Arranjo g (2)

```
begin { -ObtemHashPerfeito- }
  randomize; {--Inicializa random para 2^32 valores--}
  write ('Nome do arquivo com chaves a serem lidas: ');
  readln (NomeArq); assign (ArqEntrada, NomeArq);
  write ('Nome do arquivo para gravar experimento: '); readln(NomeArq);
  assign (ArqSaida, NomeArq); reset (ArqEntrada); rewrite (ArqSaida);
  NGrafosGerados := 0; i := 0; readln (ArqEntrada, N, M, r);
  while (i < N) and (not eof(ArqEntrada)) do
    begin readln (ArqEntrada, ConjChaves[i]); i := i + 1; end;
  if (i<>N) then begin writeln('Erro: entrada com menos que ',N,' itens'); exit; end;
  repeat
    GeraGrafo(ConjChaves,N,M,r,Pesos,NgrafosGerados,Grafo); ImprimeGrafo(Grafo);
    write('prim: ');
    for i:=0 to Grafo.NumVertices-1 do write(Grafo.Prim[i]:3); writeln;
    write('prox: ');
    for i:=0 to Grafo.NumArestas*Grafo.r-1 do write(Grafo.prox[i]:3); writeln;
    GrafoAciclico (Grafo, L, GAciclico);
  until GAciclico;
```


Programa Principal para Gerar Arranjo g (1)

```
program RotulaGrafoAciclico;
{--Entram aqui as estruturas de dados dos slides 104, 105, 106--}
var M                    : TipoValorVertice;
    N                    : TipoValorAresta;
    r                    : Tipor;
    Grafo                : TipoGrafo;
    L                    : TipoArranjoArestas;
    GAciclico            : boolean;
    g                    : Tipog;
    Pesos                : TipoTodosPesos;
    i, j, NGrafosGerados : integer;
    ConjChaves           : TipoConjChaves;
    ArqEntrada, ArqSaida : text;
    NomeArq              : string [100];
{--Entram aqui os operadores do Programa 3.18--}
{--Entram aqui os operadores do slide 77--}
{--Entram aqui os operadores do Programa 7.26--}
{--Entram aqui VerticeGrauUm e GrafoAciclico do Programa 7.10--}
```


Gera um Grafo sem Arestas Repetidas e sem *Self-Loops*

```
begin { -GeraGrafo- }
  repeat
    GrafoValido := true;      Grafo.NumVertices := M;
    Grafo.NumArestas := N;    Grafo.r := r;
    FGVazio (Grafo);          NGrafosGerados := 0;
    for j := 0 to Grafo.r - 1 do GeraPesos (Pesos[j]);
    for i := 0 to Grafo.NumArestas - 1 do
      begin
      Aresta.Peso := i;
      for j := 0 to Grafo.r-1 do Aresta.Vertices[j]:=h(ConjChaves[i], Pesos[j]);
    if VerticesIguais (Aresta) or ExisteAresta (Aresta, Grafo)
      then begin GrafoValido := false; break; end
      else InsereAresta (Aresta, Grafo);
      end;
    NGrafosGerados := NGrafosGerados + 1;
  until GrafoValido;
end; { GeraGrafo }
```

Teste para a Função de Transformação Perfeita (3)

```
begin
  write ('Nome do arquivo com chaves a serem lidas: '); readln (NomeArq);
  assign (ArqEntrada, NomeArq); reset (ArqEntrada);
  readln (ArqEntrada, N); readln (ArqEntrada, M); readln (ArqEntrada, r);
  for j := 0 to r - 1 do
    begin
    for i := 1 to MAXTAMCHAVE do read (ArqEntrada, Pesos[j][i]);
    readln (ArqEntrada);
    end;
  for i := 0 to M-1 do read (ArqEntrada, g[i]); readln (ArqEntrada);
  readln (Chave);
  while Chave <> 'aaaaaa' do
    begin writeln (hp(Chave, r, Pesos, g)); readln (Chave); end;
  close (ArqEntrada);
end. { hashingperfeito }
```


Teste para a Função de Transformação Perfeita (2)

```
var
  M          : TipoValorVertice;
  N          : TipoValorAresta;
  r          : Tipor;
  g          : Tipog;
  Pesos      : TipoTodosPesos;
  i, j       : integer;
  ConjChaves : TipoConjChaves;
  NomeArq    : string [100];
  Chave      : TipoChave;
  ArqEntrada : text;
{ Entra aqui a funcao hash universal do slide 77 }
{ Entra aqui a funcao hash perfeita do slide 112 }
```


Teste para a Função de Transformação Perfeita (1)

```
program HashingPerfeito;
const
  MAXNUMVERTICES = 100000; {--Numero maximo de vertices--}
  MAXNUMARESTAS = 100000; {--Numero maximo de arestas--}
  MAXR = 5;
  MAXTAMCHAVE = 6; {--Numero maximo de caracteres da chave--}
  MAXNUMCHAVES = 100000; {--Numero maximo de chaves lidas--}
type
  TipoValorVertice = -1..MAXNUMVERTICES;
  TipoValorAresta  = 0..MAXNUMARESTAS;
  Tipor            = 0..MAXR;
  TipoPesos        = array [1..MAXTAMCHAVE] of integer;
  TipoTodosPesos   = array [Tipor] of Tipopesos;
  Tipog            = array[0..MAXNUMVERTICES] of integer;
  TipoChave        = packed array[1..MAXTAMCHAVE] of char;
  TipoConjChaves   = array[0..MAXNUMCHAVES] of TipoChave;
  TipoIndice       = 0..MAXNUMARESTAS;
```


Função de Transformação Perfeita

```
function hp (Chave    : TipoChave;
             r        : Tipor;
             var Pesos: TipoTodosPesos;
             var g    : Tipog): TipoIndice;
var i, v: integer;
begin
  v := 0;
  for i := 0 to r - 1 do v := v + g[h(Chave, Pesos[i])];
  hp := v mod N;
end; { hp }
```

Análise: Espaço Ocupado para Descrever a Função

- O número de *bits* por chave para descrever a função é uma medida de complexidade de espaço importante.
- Como cada entrada do arranjo g usa $\log N$ *bits*, a complexidade de espaço do algoritmo é $O(\log N)$ *bits* por chave, que é o espaço para descrever a função.
- De acordo com Majewski, Wormald, Havas e Czech (1996), o limite inferior para descrever uma função perfeita com ordem preservada é $\Omega(\log N)$ *bits* por chave, o que significa que o algoritmo que acabamos de ver é ótimo para essa classe de problemas.
- Na próxima seção vamos apresentar um algoritmo de *hashing* perfeito sem ordem preservada que reduz o espaço ocupado pela função de transformação de $O(\log N)$ para $O(1)$.

Análise: Influência do Valor de r

- O grande inconveniente de usar $M = 2,09N$ é o espaço necessário para armazenar o arranjo g.
- Uma maneira de aproximar o valor de M em direção ao valor de N é usar 3-grafos, onde o valor de M pode ser tão baixo quanto $1,23N$.
- Logo, o uso de 3-grafos reduz o custo de espaço da função, mas requer o cômputo de mais uma função de transformação auxiliar h_2.
- O problema tem naturezas diferentes para $r = 2$ e $r > 2$:
  - Para $r = 2$, a probabilidade P_{r_a} varia continuamente com c.
  - Para $r > 2$, se $c \leq c(r)$, então P_{r_a} tende para 0 quando N tende para ∞; se $c > c(r)$, então P_{r_a} tende para 1.
  - Logo, um 3-grafo é obtido em média na primeira tentativa quando $c \geq 1,23$.
- Obtido o hipergrafo, o procedimento Atribuig é determinístico e requer um número linear de passos.

Análise: Influência do Valor de r

- Segundo Czech, Havas e Majewski (1992, 1997), quando $M = cN$, $c > 2$ e $r = 2$, a probabilidade P_{r_a} de gerar aleatoriamente um 2-grafo acíclico $G_2 = (V, A)$, para $N \to \infty$, é:

$$P_{r_a} = e^{\frac{1}{c}} \sqrt{\frac{c-2}{c}}.$$

- Quando $c = 2,09$ temos que $P_{r_a} = 0,33$. Logo, o número esperado de iterações para gerar um 2-grafo acíclico é $1/P_{r_a} = 1/0,33 \approx 3$.
- Logo, aproximadamente 3 grafos serão testados em média.
- O custo para gerar cada grafo é linear no número de arestas do grafo.
- O procedimento GrafoAciclico para verificar se um hipergrafo é acíclico tem complexidade $O(|V| + |A|)$.
- Logo, a complexidade de tempo para gerar a função de transformação é proporcional ao número de chaves N, desde que $M > 2N$.

Análise

- **A questão crucial é:** quantas interações são necessárias para se obter um hipergrafo $G_r = (V, A)$ que seja acíclico?
- A resposta a esta questão depende dos valores de r e M escolhidos no primeiro passo do algoritmo.
- Quanto maior o valor de M, mais esparso é o grafo e, consequentemente, mais provável que ele seja acíclico.

Hashing Perfeito Usando Espaço Quase Ótimo

Ainda no passo (a) de geração do hipergrafo:

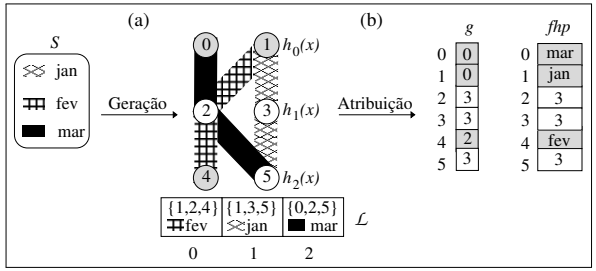
- Testa se o hipergrafo randômico resultante contém ciclos por meio da retirada iterativa de arestas de grau 1, conforme mostrado no Programa 7.10.
- As arestas retiradas são armazenadas em $\mathcal{L}$ na ordem em que foram retiradas.
- A primeira aresta retirada foi $\{1, 2, 4\}$, a segunda foi $\{1, 3, 5\}$ e a terceira foi $\{0, 2, 5\}$. Se terminar com um grafo vazio, então o grafo é acíclico, senão um novo conjunto de funções h_0, h_1 and h_2 é escolhido e uma nova tentativa é realizada.

Hashing Perfeito Usando Espaço Quase Ótimo

No passo (a) de geração do hipergrafo:

- Utiliza três funções h_0, h_1 and h_2, com intervalos $\{0, 1\}$, $\{2, 3\}$ e $\{4, 5\}$, respectivamente, cujos intervalos não se sobrepõem e por isso o grafo é 3-partido.
- Funções constroem um mapeamento do conjunto S de chaves para o conjunto A de arestas de um r-grafo acíclico $G_r = (V, A)$, onde $r = 3$, $|V| = M = 6$ e $|E| = N = 3$.
- No exemplo, "jan" é rótulo da aresta $\{h_0(\text{“jan”}), h_1(\text{“jan”}), h_2(\text{“jan”})\} = \{1, 3, 5\}$, "fev" é rótulo da aresta $\{h_0(\text{“fev”}), h_1(\text{“fev”}), h_2(\text{“fev”})\} = \{1, 2, 4\}$, e "mar" é rótulo da aresta $\{h_0(\text{“mar”}), h_1(\text{“mar”}), h_2(\text{“mar”})\} = \{0, 2, 5\}$.

Hashing Perfeito Usando Espaço Quase Ótimo



(a) Para $S = \{\text{jan}, \text{fev}, \text{mar}\}$, gera um 3-grafo 3-partido acíclico com $M = 6$ vértices e $N = 3$ arestas e um arranjo de arestas $\mathcal{L}$ obtido no momento de verificar se o hipergrafo é acíclico.

(b) Constrói função *hash* perfeita que transforma o conjunto S de chaves para o intervalo $[0, 5]$, representada pelo arranjo $g : [0, 5] \to [0, 3]$ de forma a atribuir univocamente uma aresta a um vértice.

Hashing Perfeito Usando Espaço Quase Ótimo

- Algoritmo proposto por Botelho (2008): obtem função *hash* perfeita com número constante de *bits* por chave para descrever a função.
- O algoritmo gera a função em tempo linear e a avaliação da função é realizada em tempo constante.
- Primeiro algoritmo prático descrito na literatura que utiliza $O(1)$ *bits* por chave para uma função *hash* perfeita mínima sem ordem preservada.
- Os métodos conhecidos anteriormente ou são empíricos e sem garantia de que funcionam bem para qualquer conjunto de chaves, ou são teóricos e sem possibilidade de implementação prática.
- O algoritmo utiliza **h**ipergrafos ou $r-$grafos randômicos r-partidos. Isso permite que r partes do vetor g sejam acessadas em paralelo.
- As funções mais rápidas e mais compactas são obtidas para hipergrafos com $r = 3$.

Valor das Variáveis na Execução do Programa

i	a	v	Visitado	u	j	Soma
2	$\{0,2,5\}$	2	False $\rightarrow$ True	5	2	0
		1	False $\rightarrow$ True	2	1	0
		0	False $\rightarrow$ True	0	0	0
1	$\{1,3,5\}$	2	True	-	-	3
		1	False $\rightarrow$ True	3	1	3
		0	False $\rightarrow$ True	1	0	3
0	$\{1,2,4\}$	2	False $\rightarrow$ True	4	2	0
		1	True	4	2	0
		0	True	4	2	3

- No exemplo, a primeira aresta considerada em $\mathcal{L}$ é $a = \{h_0(\text{“mar”}), h_1(\text{“mar”}), h_2(\text{“mar”})\} = \{0,2,5\}$. A Tabela mostra os valores das varáveis envolvidas no comando:

  for v := Grafo.r - 1 **downto** 0 **do**

Rotula Grafo e Atribui Valores para o Arranjo g

```
Procedure Atribuig (var Grafo: TipoGrafo; var L: TipoArranjoArestas; var g: Tipog);
var i, j, u, Soma: integer; v: TipoValorVertice;  a: TipoAresta;
    Visitado: array[0..MAXNUMVERTICES] of boolean;
begin
  for i := Grafo.NumVertices − 1 downto 0 do
    begin g[i] := grafo.r;  Visitado[i] := false;  end;
  for i := Grafo.NumArestas − 1 downto 0 do
    begin
    a := L[i];  Soma := 0;
    for v := Grafo.r − 1 downto 0 do
      if not Visitado[a.Vertices[v]]
      then begin Visitado[a.Vertices[v]] := true; u := a.Vertices[v]; j := v; end
      else Soma := Soma + g[a.Vertices[v]];
    g[u] := (j − Soma) mod grafo.r;
    end;
end; { −Fim Atribuig− }
```


Hashing Perfeito Usando Espaço Quase Ótimo

- O programa no slide seguinte mostra o procedimento para obter o arranjo g considerando um hipergrafo $G_r = (V, A)$.
- As estruturas de dados são as mesmas dos slides 104, 105 e 106.
- Para valores $0 \leq i \leq M-1$, o passo começa com $g[i] = r$ para marcar cada vértice como não atribuído e com $Visitado[i] = false$ para marcar cada vértice como não visitado.
- Seja j, $0 \leq j < r$, o índice de cada vértice u de uma aresta a.
- A seguir, para cada aresta $a \in \mathcal{L}$ da direita para a esquerda, percorre os vértices de a procurando por vértices u em a não visitados, faz $Visitado[u] = true$ e para o último vértice u não visitado faz $g[u] = (j - \sum_{v \in a \wedge Visitado[v]=true} g[v]) \bmod r$.

Hashing Perfeito Usando Espaço Quase Ótimo

No passo (b) de atribuição:

- Produz uma função *hash* perfeita que transforma o conjunto S de chaves para o intervalo $[0, M-1]$, sendo representada pelo arranjo g que armazena valores no intervalo $[0,3]$.
- O arranjo g permite selecionar um de três vértices de uma dada aresta, o qual é associado a uma chave k.

Obtem a Função Hash Perfeita

```
function hp (Chave: TipoChave; r: Tipor;
            var Pesos: TipoTodosPesos; var g: Tipog): TipoIndice;
var i, v: integer;  a: TipoArranjoVertices;
begin
  v := 0;
  for i := 0 to r − 1 do
    begin
    a[i] := h (Chave, Pesos[i]);
    v := v + g[a[i]];
    end;
  v := v mod r;
  hp := a[v];
end; { hp }
```

- O procedimento recebe a chave, o valor de r, os pesos para a função h do Programa 3.18 e o arranjo g, e segue a equação do slide 130 para descobrir qual foi o vértice da aresta escolhido para a chave.

Obtem a Função Hash Perfeita

- A partir do arranjo g podemos obter uma função *hash* perfeita para uma tabela com intervalo $[0, M-1]$.
- Para uma chave $k \in S$ a função hp tem a seguinte forma:

$$hp(k) = h_{i(k)}(k), \text{ onde } i(k) = (g[h_0(k)] + g[h_1(k)] + \ldots + g[h_{r-1}(k)]) \bmod r$$

- Considerando $r = 3$, o vértice escolhido para uma chave k é obtido por uma das três funções, isto é, $h_0(k)$, $h_1(k)$ ou $h_2(k)$.
- Logo, a decisão sobre qual função $h_i(k)$ deve ser usada para uma chave k é obtida pelo cálculo $i(k) = (g[h_0(k)] + g[h_1(k)] + g[h_2(k)]) \bmod 3$.
- No exemplo da Figura, a chave "jan" está na posição 1 da tabela porque $(g[1] + g[3] + g[5]) \bmod 3 = 0$ e $h_0(\text{"jan"}) = 1$. De forma similar, a chave "fev" está na posição 4 da tabela porque $(g[1] + g[2] + g[4]) \bmod 3 = 2$ e $h_2(\text{"fev"}) = 4$, e assim por diante.

Atribui Valores para g Usa Apenas 1 *Byte* por Entrada

- Como somente um dos quatro valores 0, 1, 2, ou 3 é armazenado em cada entrada de g, 2 *bits* são necessários.
- Na estrutura de dados do slide 106 o tipo do arranjo g é **integer**.
- Agora o comando

  Tipog = **array**[0..MAXNUMVERTICES] **of integer**;

  muda para

  Tipog = **array**[0..MAXNUMVERTICES] **of byte**;

Valor das Variáveis na Execução do Programa

- O comando após o anel:

  g[u] := (j - Soma) mod Grafo.r;

  faz $g[0] = (0 - 0) \mod 3 = 0$.
- Igualmente, para a aresta seguinte de $\mathcal{L}$ que é $a = \{h_0(\text{"jan"}), h_1(\text{"jan"}), h_2(\text{"jan"})\} = \{1, 3, 5\}$, o comando após o anel faz $g[1] = (0 - 3) \mod 3 = -3$.
- O comando a seguir:

  while g[u] < 0 do g[u] := g[u] + Grafo.r;

  irá fazer $g[1] = g[1] + 3 = -3 + 3 = 0$.
- Finalmente, para a última aresta em $\mathcal{L}$ que é $a = \{h_0(\text{"fev"}), h_1(\text{"fev"}), h_2(\text{"fev"})\} = \{1, 2, 4\}$, o comando após o anel faz $g[4] = (2 - 3) \mod 3 = -1$. faz $g[4] = g[4] + 3 = -1 + 3 = 2$.

Atribui Valores para g Usando 2 *Bits* por Entrada

```
Procedure Atribuig (var Grafo: TipoGrafo; var L: TipoArranjoArestas; var g: Tipog);
var i, j, u: integer; v: TipoValorVertice; a: TipoAresta; Soma: integer;
    valorg2bits: integer; Visitado: array[0..MAXNUMVERTICES] of boolean;
begin
  if (grafo.r <= 3) { valores de 2 bits requerem r <= 3}
  then begin
       for i := Grafo.NumVertices − 1 downto 0 do
         begin AtribuiValor2Bits (g, i, grafo.r); Visitado[i] := false; end;
       for i := Grafo.NumArestas − 1 downto 0 do
         begin
         a := L[i]; Soma := 0;
         for v := Grafo.r − 1 downto 0 do
           if not Visitado[a.Vertices[v]]
           then begin Visitado[a.Vertices[v]]:=true; u:=a.Vertices[v]; j:=v; end
           else Soma := Soma + ObtemValor2Bits (g, a.Vertices[v]);
         valorg2bits := (j − Soma) mod grafo.r; AtribuiValor2Bits (g, u, valorg2bits);
         end;
       end;
end; { Atribuig }
```


Como Empacotar Um Valor de g em Apenas 2 *Bits*

- Exemplo de "shl": $b_0, b_1, b_2, b_3, b_4, b_5, b_6, b_7$ shl 6 $= b_6, b_7, 0, 0, 0, 0, 0, 0)$.
- Exemplo de "shr": $b_0, b_1, b_2, b_3, b_4, b_5, b_6, b_7$ shr 6 $= 0, 0, 0, 0, 0, 0, b_0, b_1)$.
- Na chamada do procedimento AtribuiValor2Bits, consideremos a atribuição de $\mathrm{Valor} = 2$ na posição $\mathrm{Indice} = 4$ de g (no caso, $g[4] = 2$):
  - No primeiro comando o *byte* que vai receber $\mathrm{Valor} = 2 = (10)_2$ é determinado por i $=$ Indice div 4 $=$ 4 div 4 $=$ 1 (segundo *byte*).
  - Posição dentro do *byte* a seguir: Pos = Indice mod 4 = 4 mod 4 = 0 (os dois *bits* menos significatios do *byte*).
  - A seguir, Pos = Pos $*$ 2 porque cada valor ocupa 2 *bits* do *byte*. A seguir, **n**ot (3 shl Pos) $=$ **n**ot $((00000011)_2$ shl 0) $= (11111100)_2$. Logo, g[i] **a**nd $(11111100)_2$ zera os 2 *bits* a atribuir.
  - Finalmente, o comando g[i] **o**r (Valor shl Pos) realiza a atribuição e o *byte* fica como $(\mathrm{XXXXXX10})_2$, onde X representa 0 ou 1.

Como Empacotar Quatro Valores de g em um *Byte*

```
{−− Assume que as entradas de 2 bits do vetor g foram iniciadas com o valor 3 −−}
procedure AtribuiValor2Bits (var g : Tipog; Indice: integer; Valor : byte);
var i, Pos : integer;
begin
  i   := Indice div 4; Pos := (Indice mod 4);
  Pos := Pos * 2; { Cada valor ocupa 2 bits }
  g[i] := g[i] and (not(3 shl Pos)); { zera os dois bits a atribuir }
  g[i] := g[i] or (Valor shl Pos);   { realiza a atribuicao }
end; {AtribuiValor2Bits}
function ObtemValor2Bits (var g : Tipog; Indice: Integer): byte;
var i, Pos: Integer;
begin
  i := Indice div 4; Pos := (Indice mod 4);
  Pos := Pos * 2; { Cada valor ocupa 2 bits }
  ObtemValor2Bits := (g[i] shr Pos) and 3;
end; {ObtemValor2Bits}
```


Como Empacotar Quatro Valores de g em um *Byte*

- Para isso foram criados dois procedimentos:
  - AtribuiValor2Bits: atribui o $i-$ésimo valor de g em uma das quatro posições do *byte* apropriado.
  - ObtemValor2Bits: retorna o $i-$ésimo valor de g.
- Agora o tipo do arranjo g permanece como *byte*, mas o comando

  Tipog = **array**[0..MAXNUMVERTICES] **of byte**;

  muda para

  const MAXGSIZE = Trunc((MAXNUMVERTICES + 3)/4)

  Tipog = **array**[0..MAXGSIZE] **of byte**;

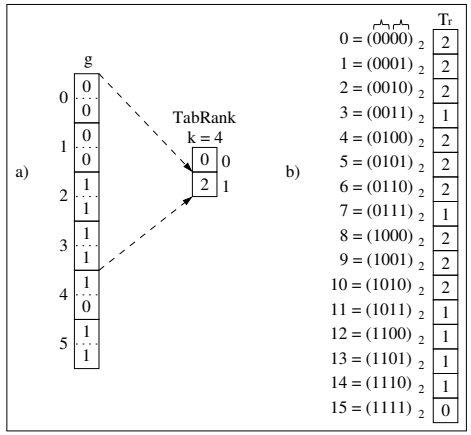
  onde MAXGSIZE indica que o arranjo Tipog ocupa um quarto do espaço e o *byte* passa a armazenar 4 valores.

Implementação da Tabela $TabRank$

- TabRank armazena em cada entrada o número total de valores de *2 bits* diferentes de $r = 3$ até cada k-ésima posição do arranjo g.
- No exemplo consideramos $k = 4$. Assim, existem 0 valores até a posição 0 e 2 valores até a posição 4 de g.

```
Procedure GeraTabRank (var g: Tipog; Tamg: TipoValorVertice;
                       k: TipoK; var TabRank: TipoTabRank);
var i, Soma: Integer;
begin
  Soma := 0;
  for i := 0 to Tamg − 1 do
    begin
    if (i mod k = 0) then TabRank[i div k] := Soma;
    if (ObtemValor2Bits(g,i) <> NAOATRIBUIDO) then Soma := Soma + 1;
    end;
end; { GeraTabRank }
```

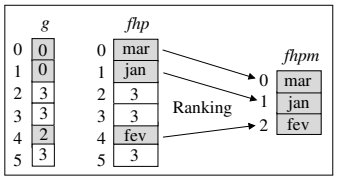

Implementação da Função $Rank$



- A função $rank$ usa um algoritmo proposto por Pagh (2001).
- Usa ϵM *bits* adicionais, $0 < \epsilon < 1$, para armazenar o $rank$ de cada $k-$ésimo índice de g em TabRank, onde $k = \lfloor \log(M)/\epsilon \rfloor$.
- Para uma avaliação de $rank(u)$ em $O(1)$, é necessário usar uma tabela T_r auxiliar.

Implementação da Função $Rank$

- Para obter uma função *hash* perfeita mínima precisamos reduzir o intervalo da tabela de $[0, M-1]$ para $[0, N-1]$.
- Vamos utilizar uma **estrutura de dados sucinta**, acompanhada de um algoritmo eficiente para a operação de pesquisa.
- $rank$: $[0, M-1] \rightarrow [0, N-1]$: conta o número de posições atribuidas antes de uma dada posição v em g em tempo constante.
- O passo de *ranking* constrói a estrutura de dados usada para computar a função $rank : [0, 5] \rightarrow [0, 2]$ em tempo $O(1)$. Por exemplo, $rank(4) = 2$ porque os valores de $g[0]$ e $g[1]$ são diferentes de 3.



Função de Transformação Perfeita Usando 2 $Bits$

```
function hp (Chave: TipoChave; r: Tipor;
             var Pesos: TipoTodosPesos; var g: Tipog): TipoIndice;
var i, v: integer;  a: TipoArranjoVertices;
begin
  v := 0;
  for i := 0 to r − 1 do
    begin
    a[i] := h (Chave, Pesos[i]);
    v := v + ObtemValor2Bits(g, a[i]);
    end;
  v := v mod r;
  hp := a[v];
end; { hp }
```

- Basta substituir no programa do slide 131 o comando
  v := v + g[a[i]];
  pelo comando
  v := v + ObtemValor2Bits(g, a[i]);

Função de Transformação Perfeita Usando 2 $Bits$

```
function hpm (Chave: TipoChave; r: Tipor; var Pesos: TipoTodosPesos; var g: Tipog;
              var Tr: TipoTr; k: TipoK; var TabRank: TipoTabRank): TipoIndice;
var i, j, u, Rank, Byteg: TipoIndice;
begin
  u := hp (Chave, r, Pesos, g);
  j := u div k;        Rank := TabRank[j];
  i := j * k;          j := i;
  Byteg := j div 4;  j := j + 4;
  while (j < u) do
    begin Rank := Rank + Tr[g[Byteg]]; j := j + 4;  Byteg := Byteg + 1; end;
  j := j - 4;
  while (j < u) do
    begin
    if (ObtemValor2Bits (g,j) <> NAOATRIBUIDO) then Rank := Rank+1;
    j := j + 1;
    end;
  hpm := Rank;
end; { hpm }
```


Função de Transformação Perfeita Usando 2 $Bits$

- A função *hash* perfeita mínima:

$$hpm(x) = rank(hp(x))$$

- Quanto maior for o valor de k mais compacta é a função *hash* perfeita mínima resultante. Assim, os usuários podem permutar espaço por tempo de avaliação variando o valor de k na implementação.
- Entretanto, o melhor é utilizar valores para k que sejam potências de dois (por exemplo, $k = 2^{b_k}$ para alguma constante b_k), o que permite trocar as operações de multiplicação, divisão e módulo pelas operações de deslocamento de *bits* à esquerda, à direita, e "*and*" binário, respectivamente.
- O valor $k = 256$ produz funções compactas e o número de *bits* para codificar k é $b_k = 8$.

Implementação da Tabela T_r

```
Procedure GeraTr (var Tr: TipoTr);
var i, j, v, Soma: Integer;
begin
  Soma := 0;
  for i := 0 to MAXTRVALUE do
    begin
    Soma := 0;  v := i;
    for j := 1 to 4 do
      begin
      if ((v and 3) <> NAOATRIBUIDO) then Soma := Soma + 1;
      v := v shr 2;
      end;
    Tr[i] := Soma;
    end;
end; { GeraTr }
```


Implementação da Tabela T_r

- Para calcular o $rank(u)$ usando as tabelas TabRank e T_r são necessários dois passos:
  - Obter o *rank* do maior índice precomputado $v \leq u$ em TabRank.
  - Usar T_r para contar número de vértices atribuídos de v até $u - 1$.
- Na figura do slide 138 T_r possui 16 entradas necessárias para armazenas todas as combinações possíveis de 4 *bits*.
- Por exemplo, a posição 0, cujo valor binário é $(0000)_2$, contém dois valores diferentes de $r = 3$; na posição 3, cujo valor binário é $(0011)_2$, contém apenas um valor diferente de $r = 3$, e assim por diante.
- Cabe notar que cada valor de $r \geq 2$ requer uma tabela T_r diferente.
- O procedimento a seguir considera que T_r é indexada por um número de 8 *bits* e, portanto, MaxTrValue = 255. Além disso, no máximo 4 vértices podem ser empacotados em um *byte*, razão pela qual o anel interno vai de 1 a 4.

Análise de Espaço da Função *Hash* Perfeita Mínima hpm

- Espaço para g é 2.46 *bits* por chave.
- Espaço para a tabela TabRank:

$$|g| + |\text{TabRank}| = 2cn + 32 * (cn/k),$$

assumindo que cada uma das cn/k entradas da tabela TabRank armazena um inteiro de 32 *bits* e que cada uma das cn entradas de g armazena um inteiro de 2 *bits*. Se tomarmos $k = 256$, teremos:

$$2cn + (32/256)cn = (2 + 1/8)cn = (2 + \epsilon)cn, \text{ para } \epsilon = 1/8 = 0.125.$$

- Logo, o espaço total é $(2 + \epsilon)cn$ *bits*. Usando $c = 1{,}23$ e $\epsilon = 0{,}125$, a função *hash* perfeita mínima necessita aproximadamente $2{,}62$ *bits* por chave para ser armazenada.

Análise de Espaço da Função *Hash* Perfeita hp

- Como somente quatro valores distintos são armazenados em cada entrada de g, são necessários 2 *bits* por valor.
- Como o tamanho de g para um 3-grafo é $M = cN$, onde $c = 1{,}23$, o espaço necessário para armazenar o arranjo g é de $2cn = 2{,}46$ *bits* por entrada.

Análise de Tempo

- O Programa 7.10 para verificar se um hipergrafo é acíclico do tem complexidade $O(|V| + |A|)$. Como $|A| = O(|V|)$ para grafos esparsos como os considerados aqui, a complexidade de tempo para gerar a função de transformação é proporcional ao número de chaves N.
- O tempo necessário para avaliar a função hp do slide 130 envolve a avaliação de três funções *hash* universais, com um custo final $O(1)$.
- O tempo necessário para avaliar a função hpm do slide 142 tem um custo final $O(1)$, utilizando uma estrutura de dados sucinta que permite computar em $O(1)$ o número de posições atribuidas antes de uma dada posição em um arranjo.
- A tabela T_r permite contar o número de vértices atribuídos em $\epsilon \log M$ *bits* com custo $O(1/\epsilon)$, onde $0 < \epsilon < 1$.
- Mais ainda, a avaliação da função $rank$ é muito eficiente já que tanto TabRank quanto T_r cabem inteiramente na memória *cache* da CPU.

Análise de Tempo (Botelho 2008)

- Quando $M = cN$, $c > 2$ e $r = 2$, a probabilidade P_{r_a} de gerar aleatoriamente um 2-grafo bipartido acíclico, para $N \to \infty$, é:

$$P_{r_a} = \sqrt{1 - \left(\frac{2}{c}\right)^2}.$$

- Quando $c = 2,09$, temos que $P_{r_a} = 0,29$ e o número esperado de iterações para gerar um 2-grafo bipartido acíclico é $1/P_{r_a} = 1/0,29 \approx 3,45$.
- Isso significa que, em média, aproximadamente $3,45$ grafos serão testados antes que apareça um 2-grafo bipartido acíclico.
- Quando $M = cN$, $c \geq 1{,}23$ e $r = 2$, um 3-grafo 3-partido acíclico é obtido em 1 tentativa com probabilidade tendendo para 1 quando $N \to \infty$.
- Logo, o custo para gerar cada grafo é linear no número de arestas do grafo.

Análise de Espaço da Função *Hash* Perfeita Mínima hpm

- Mehlhorn (1984) mostrou que o limite inferior para armazenar uma função *hash* perfeita mínima é $N \log e + O(\log N) \approx 1{,}44N$. Assim, o valor de aproximadamente $2{,}62$ *bits* por chave é um valor muito próximo do limite inferior de aproximadamente $1{,}44$ *bits* por chave para essa classe de problemas.
- Esta seção mostra um algoritmo prático que reduziu a complexidade de espaço para armazenar uma função *hash* perfeita mínima de $O(N \log N)$ *bits* para $O(N)$ *bits*. Isso permite o uso de *hashing* perfeito em aplicações em que antes não eram consideradas uma boa opção.
- Por exemplo, Botelho, Lacerda, Menezes e Ziviani (2009) mostraram que uma função *hash* perfeita mínima apresenta o melhor compromisso entre espaço ocupado e tempo de busca quando comparada com todos os outros métodos de *hashing* para indexar a memória interna para conjuntos estáticos de chaves.