Pesquisa em Memória Primária*

Última alteração: 7 de Setembro de 2010

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

Pesquisa em Memória Primária

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**.

Introdução - Conceitos Básicos

- Conjunto de registros ou arquivos $\Rightarrow$ 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.

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

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.

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

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:

```
#define MAXN 10
typedef long TipoChave;
typedef struct TipoRegistro {
  TipoChave Chave;
  /* outros componentes */
} TipoRegistro;
typedef int TipoIndice;
typedef struct TipoTabela {
  TipoRegistro Item[MAXN + 1];
  TipoIndice n;
} TipoTabela;
```

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

```
void Inicializa(TipoTabela *T)
{ T->n = 0; }

TipoIndice Pesquisa(TipoChave x, TipoTabela *T)
{ int i;
  T->Item[0].Chave = x;  i = T->n + 1;
  do {i--;} while (T->Item[i].Chave != x);
  return i;
}

void Insere(TipoRegistro Reg, TipoTabela *T)
{ if (T->n == MAXN)
  printf("Erro : tabela cheia\n");
  else { T->n++; T->Item[T->n] = Reg; }
}
```

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

```
TipoIndice Binaria(TipoChave x, TipoTabela *T)
{ TipoIndice i, Esq, Dir;
  if (T->n == 0)
  return 0;
  else
  { Esq = 1;
    Dir = T->n;
    do
      { i = (Esq + Dir) / 2;
        if (x > T->Item[i].Chave)
        Esq = i + 1;
        else Dir = i - 1;
      } while (x != T->Item[i].Chave && Esq <= Dir);
    if (x == T->Item[i].Chave) return i; else return 0;
  }
}
```

Pesquisa para a chave G:

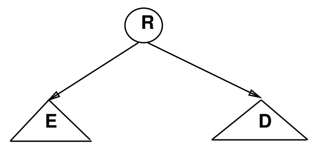
1	2	3	4	5	6	7	8
A	*B*	*C*	**D**	*E*	*F*	*G*	*H*
				E	**F**	*G*	*H*
						G	*H*

Pesquisa Binária

- **Pesquisa em tabela pode ser mais eficiente ⇒ 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.

Árvores Binárias de Pesquisa sem Balanceamento

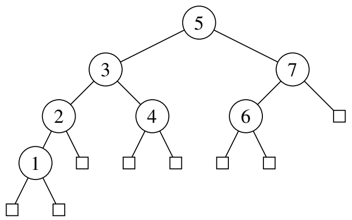
- Para qualquer nó que contenha um registro



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.

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.

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

Estrutura de dados:

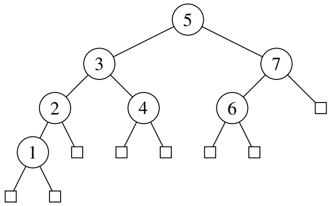
```
typedef long TipoChave;
typedef struct TipoRegistro {
  TipoChave Chave;
  /* outros componentes */
} TipoRegistro;
typedef struct TipoNo * TipoApontador;
typedef struct TipoNo {
  TipoRegistro Reg;
  TipoApontador Esq, Dir;
} TipoNo;
```


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.

```
void Pesquisa(TipoRegistro *x, TipoApontador *p)
{ if (*p == NULL)
  { printf("Erro: Registro nao esta presente na arvore\n"); return; }
  if (x->Chave < (*p)->Reg.Chave)
  { Pesquisa(x, &(*p)->Esq); return; }
  if (x->Chave > (*p)->Reg.Chave) Pesquisa(x, &(*p)->Dir);
  else *x = (*p)->Reg;
}
```

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

```
void Inicializa(TipoApontador *Dicionario)
{ *Dicionario = NULL; }
end; { Inicializa }

{-- Entra aqui a definição dos tipos mostrados no slide 18 --}
{-- Entram aqui os procedimentos Insere e Inicializa --}
int main(int argc, char *argv[])
{ TipoDicionario Dicionario; TipoRegistro x;
  Inicializa(&Dicionario);
  scanf("%d%*[^\n]", &x.Chave);
  while(x.Chave > 0)
    { Insere(x,&Dicionario);
      scanf("%d%*[^\n]", &x.Chave);
    }
}
```


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.

```
void Insere(TipoRegistro x, TipoApontador *p)
{ if (*p == NULL)
  { *p = (TipoApontador)malloc(sizeof(TipoNo));
    (*p)->Reg = x; (*p)->Esq = NULL; (*p)->Dir = NULL;
    return;
  }
  if (x.Chave < (*p)->Reg.Chave)
  { Insere(x, &(*p)->Esq); return; }
  if (x.Chave > (*p)->Reg.Chave)
  Insere(x, &(*p)->Dir);
  else printf("Erro : Registro ja existe na arvore\n");
}
```

Procedimento para Retirar x da Árvore

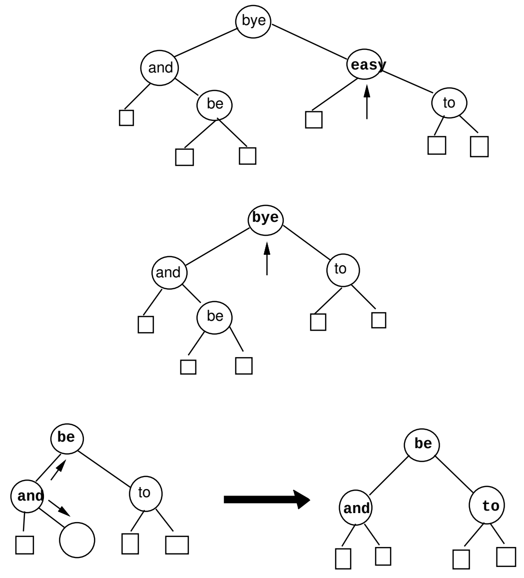
```
void Antecessor(TipoApontador q, TipoApontador *r)
{ if ((*r)->Dir != NULL)
  { Antecessor(q, &(*r)->Dir);
    return;
  }
  q->Reg = (*r)->Reg;
  q = *r;
  *r = (*r)->Esq;
  free(q);
}
```


Procedimento para Retirar x da Árvore

```
void Retira(TipoRegistro x, TipoApontador *p)
{ TipoApontador Aux;
  if (*p == NULL) { printf("Erro : Registro nao esta na arvore\n"); return; }
  if (x.Chave < (*p)->Reg.Chave) { Retira(x, &(*p)->Esq); return; }
  if (x.Chave > (*p)->Reg.Chave) { Retira(x, &(*p)->Dir); return; }
  if ((*p)->Dir == NULL)
  { Aux = *p; *p = (*p)->Esq;
    free(Aux); return;
  }
  if ((*p)->Esq != NULL) { Antecessor(*p, &(*p)->Esq); return; }
  Aux = *p; *p = (*p)->Dir;
  free(Aux);
}
```

- **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.

Outro Exemplo de Retirada de Nó

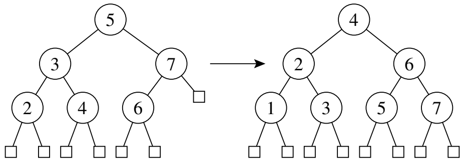


Caminhamento Central

- 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.

Árvores Binárias de Pesquisa com Balanceamento

- Árvore completamente balanceada $\Rightarrow$ 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:

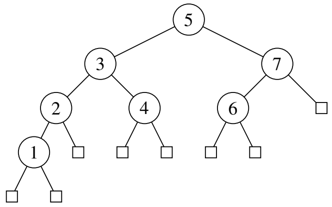
  melhor caso : $C(n) = O(1)$
  pior caso : $C(n) = O(n)$
  caso médio : $C(n) = O(\log n)$

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

Caminhamento Central

```
void Central(TipoApontador p)
{ if (p == NULL) return;
  Central(p->Esq);
  printf("%ld\n", p->Reg.Chave);
  Central(p->Dir);
}
```

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



Uma Forma de Contornar este Problema

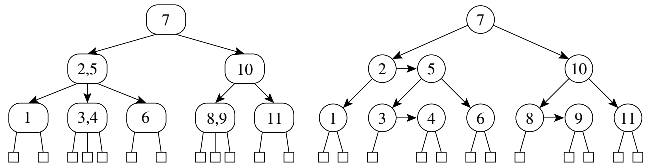
- Procurar solução intermediária que possa manter árvore “quase-balanceada”, em vez de tentar manter a árvore completamente balanceada.
- **Objetivo:** Procurar obter bons tempos de pesquisa, próximos do tempo ótimo da árvore completamente balanceada, mas sem pagar muito para inserir ou retirar da árvore.
- **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.

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 $8 = (0 + 1 + 1 + 2 + 2 + 2)$.

Árvores SBB

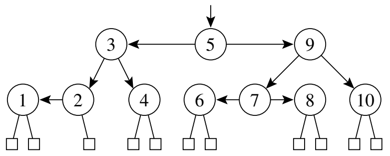
- Árvores B ⇒ estrutura para memória secundária. (Bayer R. e McCreight E.M., 1972)
- **Árvore 2-3** ⇒ caso especial da árvore B.
- Cada nó tem duas ou três subárvores.
- Mais apropriada para memória primária.
- **Exemplo:** Uma árvore 2-3 e a árvore B binária correspondente



Árvores SBB

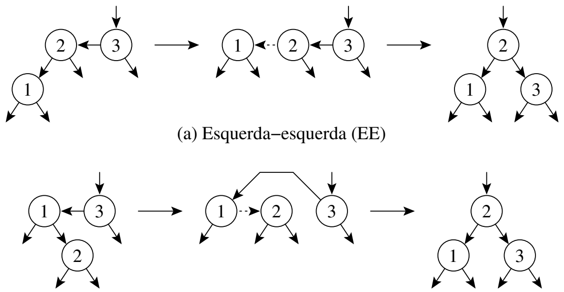
- Árvore 2-3 ⇒ **árvore B binária** (assimetria inerente)
  1. Apontadores à esquerda apontam para um nó no nível abaixo.
  2. Apontadores à direita podem ser verticais ou horizontais.

  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.



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)

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

```
typedef int TipoChave;
typedef struct TipoRegistro {
  /* outros componentes */
  TipoChave Chave;
} TipoRegistro;
typedef enum {
  Vertical, Horizontal
} TipoInclinacao;
typedef struct TipoNo* TipoApontador;
typedef struct TipoNo {
  TipoRegistro Reg;
  TipoApontador Esq, Dir;
  TipoInclinacao BitE, BitD;
} TipoNo;
```


Procedimentos Auxiliares para Árvores SBB

```
void EE(TipoApontador *Ap)
{ TipoApontador Ap1;
  Ap1 = (*Ap)->Esq; (*Ap)->Esq = Ap1->Dir; Ap1->Dir = *Ap;
  Ap1->BitE = Vertical; (*Ap)->BitE = Vertical; *Ap = Ap1;
}

void ED(TipoApontador *Ap)
{ TipoApontador Ap1, Ap2;
  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;
}
```


Procedimentos Auxiliares para Árvores SBB

```
void DD(TipoApontador *Ap)
{ TipoApontador Ap1;
  Ap1 = (*Ap)->Dir; (*Ap)->Dir = Ap1->Esq; Ap1->Esq = *Ap;
  Ap1->BitD = Vertical; (*Ap)->BitD = Vertical; *Ap = Ap1;
}

void DE(TipoApontador *Ap)
{ TipoApontador Ap1, Ap2;
  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;
}
```

Procedimento para Inserir na Árvore SBB

```
void IInsere(TipoRegistro x, TipoApontador *Ap,
             TipoInclinacao *IAp, short *Fim)
{ if (*Ap == NULL)
  { *Ap = (TipoApontador)malloc(sizeof(TipoNo));
    *IAp = Horizontal;  (*Ap)->Reg = x;
    (*Ap)->BitE = Vertical;  (*Ap)->BitD = Vertical;
    (*Ap)->Esq = NULL; (*Ap)->Dir = NULL; *Fim = FALSE;
    return;
  }
  if (x.Chave < (*Ap)->Reg.Chave)
  { IInsere(x, &(*Ap)->Esq, &(*Ap)->BitE, Fim);
    if (*Fim) return;
    if ((*Ap)->BitE != Horizontal) { *Fim = TRUE; return; }
    if ((*Ap)->Esq->BitE == Horizontal)
    { EE(Ap); *IAp = Horizontal; return; }
    if ((*Ap)->Esq->BitD == Horizontal) { ED(Ap); *IAp = Horizontal; }
    return;
  }
```

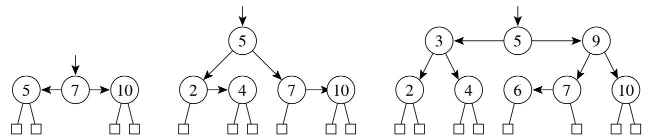

Procedimento para Inserir na Árvore SBB

```
  if (x.Chave <= (*Ap)->Reg.Chave)
  { printf("Erro: Chave ja esta na arvore\n");
    *Fim = TRUE;
    return;
  }
  IInsere(x, &(*Ap)->Dir, &(*Ap)->BitD, Fim);
  if (*Fim) return;
  if ((*Ap)->BitD != Horizontal) { *Fim = TRUE;  return; }
  if ((*Ap)->Dir->BitD == Horizontal)
  { DD(Ap); *IAp = Horizontal; return;}
  if ((*Ap)->Dir->BitE == Horizontal) { DE(Ap); *IAp = Horizontal; }
}

void Insere(TipoRegistro x, TipoApontador *Ap)
{ short Fim;  TipoInclinacao IAp;
  IInsere(x, Ap, &IAp, &Fim);
}
```


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.



```
void Inicializa(TipoApontador *Dicionario)
{ *Dicionario = NULL; }
```


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.

Procedimento para Retirar da Árvore SBB

```
void EsqCurto(TipoApontador *Ap, short *Fim)
{ /* Folha esquerda retirada => arvore curta na altura esquerda */
  TipoApontador Ap1;
  if ((*Ap)->BitE == Horizontal)
  { (*Ap)->BitE = Vertical; *Fim = TRUE; return; }
  if ((*Ap)->BitD == Horizontal)
  { Ap1 = (*Ap)->Dir; (*Ap)->Dir = Ap1->Esq; Ap1->Esq = *Ap; *Ap = Ap1;
    if ((*Ap)->Esq->Dir->BitE == Horizontal)
    { DE(&(*Ap)->Esq); (*Ap)->BitE = Horizontal;}
    else if ((*Ap)->Esq->Dir->BitD == Horizontal)
         { DD(&(*Ap)->Esq); (*Ap)->BitE = Horizontal; }
    *Fim = TRUE; return;
  }
  (*Ap)->BitD = Horizontal;
  if ((*Ap)->Dir->BitE == Horizontal) { DE(Ap); *Fim = TRUE; return; }
  if ((*Ap)->Dir->BitD == Horizontal) { DD(Ap); *Fim = TRUE; }
}
```


Procedimento para Retirar da Árvore SBB – DirCurto

```
void DirCurto(TipoApontador *Ap, short *Fim)
{ /* Folha direita retirada => arvore curta na altura direita */
  TipoApontador Ap1;
  if ((*Ap)->BitD == Horizontal)
  { (*Ap)->BitD = Vertical; *Fim = TRUE; return; }
  if ((*Ap)->BitE == Horizontal)
  { Ap1 = (*Ap)->Esq; (*Ap)->Esq = Ap1->Dir; Ap1->Dir = *Ap; *Ap = Ap1;
    if ((*Ap)->Dir->Esq->BitD == Horizontal)
    { ED(&(*Ap)->Dir); (*Ap)->BitD = Horizontal; }
    else if ((*Ap)->Dir->Esq->BitE == Horizontal)
         { EE(&(*Ap)->Dir); (*Ap)->BitD = Horizontal;}
    *Fim = TRUE; return;
  }
  (*Ap)->BitE = Horizontal;
  if ((*Ap)->Esq->BitD == Horizontal) { ED(Ap); *Fim = TRUE; return; }
  if ((*Ap)->Esq->BitE == Horizontal) { EE(Ap); *Fim = TRUE; }
}
```


Procedimento para Retirar da Árvore SBB – Antecessor

```
void Antecessor(TipoApontador q, TipoApontador *r, short *Fim)
{ if ((*r)->Dir != NULL)
  { Antecessor(q, &(*r)->Dir, Fim);
    if (!*Fim) DirCurto(r, Fim); return;
  }
  q->Reg = (*r)->Reg; q = *r; *r = (*r)->Esq; free(q);
  if (*r != NULL) *Fim = TRUE;
}
```


Procedimento para Retirar da Árvore SBB

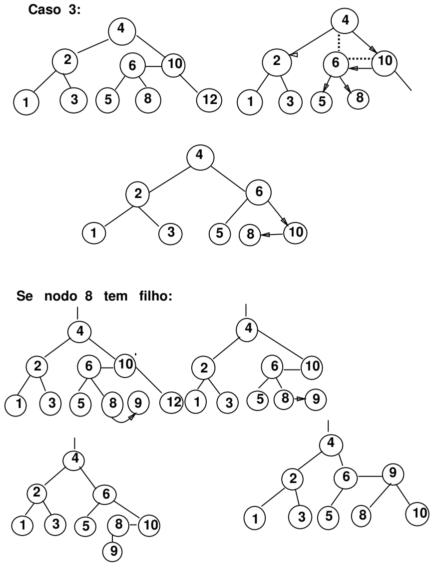
```
void IRetira(TipoRegistro x, TipoApontador *Ap, short *Fim)
{ TipoNo *Aux;
  if (*Ap == NULL) { printf("Chave nao esta na arvore\n"); *Fim = TRUE; return; }
  if (x.Chave < (*Ap)->Reg.Chave)
  { IRetira(x, &(*Ap)->Esq, Fim); if (!*Fim) EsqCurto(Ap, Fim); return; }
  if (x.Chave > (*Ap)->Reg.Chave)
  { IRetira(x, &(*Ap)->Dir, Fim);
    if (!*Fim) DirCurto(Ap, Fim); return;
  }
  *Fim = FALSE; Aux = *Ap;
  if (Aux->Dir == NULL)
  { *Ap = Aux->Esq; free(Aux);
    if (*Ap != NULL) *Fim = TRUE; return;
  }
  if (Aux->Esq == NULL)
  { *Ap = Aux->Dir; free(Aux);
    if (*Ap != NULL) *Fim = TRUE; return;
  }
  Antecessor(Aux, &Aux->Esq, Fim);
  if (!*Fim) EsqCurto(Ap, Fim); /* Encontrou chave */
}
void Retira(TipoRegistro x, TipoApontador *Ap)
{ short Fim; IRetira(x, Ap, &Fim); }
```

Análise

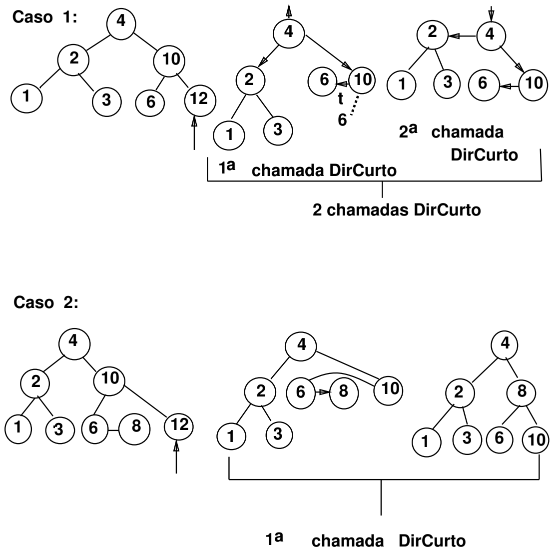
- Nas árvores SBB é necessário distinguir dois tipos de **alturas**:
  1. **Altura vertical** $h \rightarrow$ necessária para manter a altura uniforme e obtida através da contagem do número de apontadores verticais em qualquer caminho entre a raiz e um nó externo.
  2. **Altura** $k \rightarrow$ representa 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

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

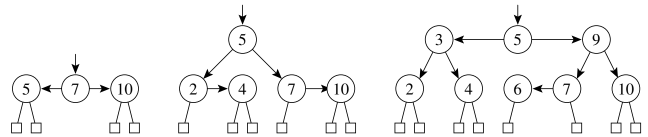
Exemplo: Retirada de Nós da Árvore SBB



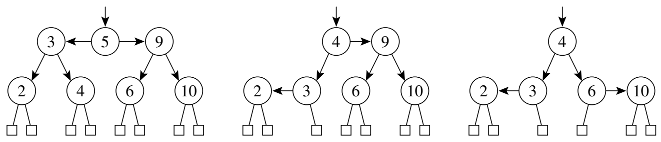
Exemplo: Retirada de Nós da Árvore SBB



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.



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).

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**

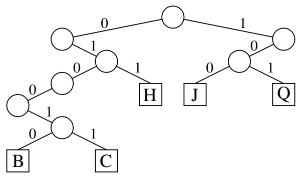
Trie

- 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.

Exemplo

- Dada as chaves de 6 bits:

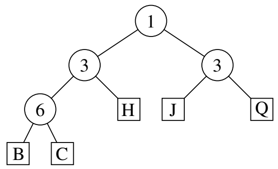
$$\begin{aligned} B &= 010010 \\ C &= 010011 \\ H &= 011000 \\ J &= 100001 \\ M &= 101000 \end{aligned}$$



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:

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



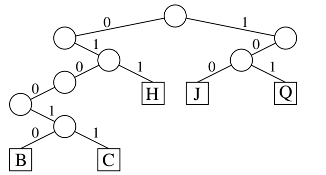
Patricia - Practical Algorithm To Retrieve Information Coded In Alphanumeric

- 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

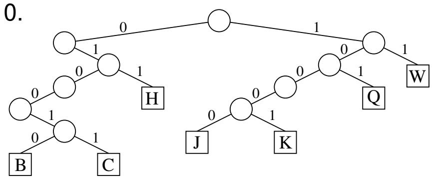
- O formato das tries, diferentemente das árvores binárias comuns, não depende da ordem em que as chaves são inseridas e sim da estrutura das chaves através da distribuição de seus bits.
- **Desvantagem:**
  - Uma grande desvantagem das tries é a formação de caminhos de uma só direção para chaves com 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

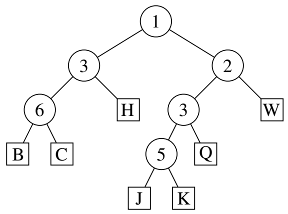
```
TipoDib Bit(TipoIndexAmp i, TipoChave k)
{ /* Retorna o i-esimo bit da chave k a partir da esquerda */
  int c, j;
  if (i == 0)
  return 0;
  else { c = k;
         for (j = 1; j <= D - i; j++) c /= 2;
         return (c & 1);
       }
}

short EExterno(TipoArvore p)
{ /* Verifica se p^ e nodo externo */
  return (p->nt == Externo);
}
```


Estrutura de Dados

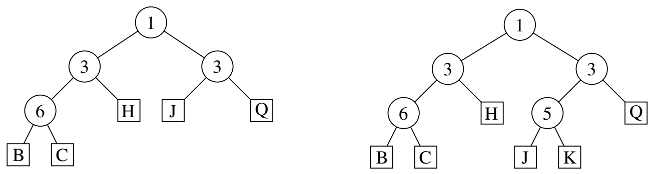
```
#define D 8   /* depende de TipoChave */
typedef unsigned char TipoChave; /* a definir, depende da aplicacao */
typedef unsigned char TipoIndexAmp;
typedef unsigned char TipoDib;
typedef enum {
  Interno, Externo
} TipoNo;
typedef struct TipoPatNo* TipoArvore;
typedef struct TipoPatNo {
  TipoNo nt;
  union {
    struct {
      TipoIndexAmp Index;
      TipoArvore Esq, Dir;
    } NInterno ;
    TipoChave Chave;
  } NO;
} TipoPatNo;
```


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 $\rightarrow$ a subárvore direita Bit de índice 3 $\rightarrow$ 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^o 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.

Procedimentos para Criar Nós Interno e Externo

```
TipoArvore CriaNoInt(int i, TipoArvore *Esq, TipoArvore *Dir)
{ TipoArvore p;
  p = (TipoArvore)malloc(sizeof(TipoPatNo));
  p->nt = Interno; p->NO.NInterno.Esq = *Esq;
  p->NO.NInterno.Dir = *Dir; p->NO.NInterno.Index = i;
  return p;
}

TipoArvore CriaNoExt(TipoChave k)
{ TipoArvore p;
  p = (TipoArvore)malloc(sizeof(TipoPatNo));
  p->nt = Externo; p->NO.Chave = k; return p;
}
```


Algoritmo de Pesquisa

```
void Pesquisa(TipoChave k, TipoArvore t)
{ if (EExterno(t))
  { if (k == t->NO.Chave)
    printf("Elemento encontrado\n");
    else printf("Elemento nao encontrado\n");
    return;
  }
  if (Bit(t->NO.NInterno.Index, k) == 0)
  Pesquisa(k, t->NO.NInterno.Esq);
  else Pesquisa(k, t->NO.NInterno.Dir);
}
```


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.

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.

Algoritmo de inserção

```
TipoArvore InsereEntre(TipoChave k, TipoArvore *t, int i)
{ TipoArvore p;
  if (EExterno(*t) || i < (*t)->NO.NInterno.Index)
  { /* cria um novo no externo */
    p = CriaNoExt(k);
    if (Bit(i, k) == 1)
    return (CriaNoInt(i, t, &p));
    else return (CriaNoInt(i, &p, t));
  }
  else
  { if (Bit((*t)->NO.NInterno.Index, k) == 1)
    (*t)->NO.NInterno.Dir = InsereEntre(k,&(*t)->NO.NInterno.Dir,i);
    else
    (*t)->NO.NInterno.Esq = InsereEntre(k,&(*t)->NO.NInterno.Esq,i);
    return (*t);
  }
}
```


Algoritmo de inserção

```
TipoArvore Insere(TipoChave k, TipoArvore *t)
{ TipoArvore p; int i;
  if (*t == NULL) return (CriaNoExt(k));
  else
  { p = *t;
    while (!EExterno(p))
      { if (Bit(p->NO.NInterno.Index, k) == 1) p = p->NO.NInterno.Dir;
        else p = p->NO.NInterno.Esq;
      }
    /* acha o primeiro bit diferente */
    i = 1;
    while ((i <= D) & (Bit((int)i, k) == Bit((int)i, p->NO.Chave)))
      i++;
    if (i > D) { printf("Erro: chave ja esta na arvore\n"); return (*t); }
    else return (InsereEntre(k, t, i));
  }
}
```


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)

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*)

- 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 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 $p \approx 87,7\%$.

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.

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.

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 $O(n)$.

```
typedef char TipoChave[N];

TipoIndice h(TipoChave Chave, TipoPesos p)
{ int i; unsigned int Soma = 0;
  int comp = strlen(Chave);
  for (i = 0; i < comp; i++) Soma += p[i][(unsigned int)Chave[i]];
  return (Soma % M);
}
```


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 **t**roca 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 $1 \leq i \leq n$.

```
#define TAMALFABETO 256
typedef unsigned TipoPesos[N][TAMALFABETO];
void GeraPesos(TipoPesos p) /* Gera valores randomicos entre 1 e 10.000 */
{ int i, j; struct timeval semente; /* Utilizar o tempo como semente */
  gettimeofday(&semente, NULL);
  srand((int)(semente.tv_sec + 1000000 * semente.tv_usec));
  for (i = 0; i < N; i++)
    for (j = 0; j < TAMALFABETO; j++)
      p[i][j] = 1 + (int)(10000.0 * rand() / (RAND_MAX + 1.0));
}
```


Transformação de Chaves Não Numéricas

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

typedef char TipoChave[N];
TipoIndice h(TipoChave Chave, TipoPesos p)
{ int i; unsigned int Soma = 0;
  int comp = strlen(Chave);
  for (i = 0; i < comp; i++) Soma += (unsigned int)Chave[i] * p[i];
  return (Soma % M);
}
```


Transformação de Chaves Não Numéricas

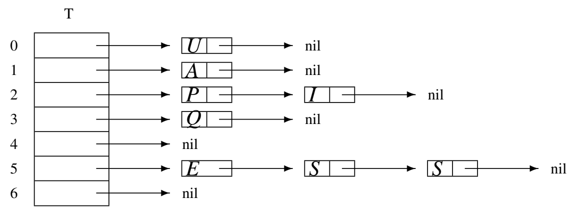
- As chaves não numéricas devem ser transformadas em números:

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

- 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.

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 para Listas Encadeadas

```
typedef char TipoChave[N];
typedef unsigned TipoPesos[N][TAMALFABETO];
typedef struct TipoItem {
  /* outros componentes */
  TipoChave Chave;
} TipoItem;
typedef unsigned int TipoIndice;
typedef struct TipoCelula* TipoApontador;
typedef struct TipoCelula {
  TipoItem Item;
  TipoApontador Prox;
} TipoCelula;
typedef struct TipoLista {
  TipoCelula *Primeiro, *Ultimo;
} TipoLista;
typedef TipoLista TipoDicionario[M];
```


Operações do Dicionário Usando Listas Encadeadas

```
void Inicializa(TipoDicionario T)
{ int i;
  for (i = 0; i < M; i++) FLVazia(&T[i]);
}

TipoApontador Pesquisa(TipoChave Ch, TipoPesos p, TipoDicionario T)
{ /* TipoApontador de retorno aponta para o item anterior da lista */
  TipoIndice i; TipoApontador Ap;
  i = h(Ch, p);
  if (Vazia(T[i])) return NULL; /* Pesquisa sem sucesso */
  else
  { Ap = T[i].Primeiro;
    while (Ap->Prox->Prox != NULL &&
        strncmp(Ch, Ap->Prox->Item.Chave, sizeof(TipoChave)))
      Ap = Ap->Prox;
    if (!strncmp(Ch, Ap->Prox->Item.Chave, sizeof(TipoChave))) return Ap;
    else return NULL; /* Pesquisa sem sucesso */
  }
}
```


Operações do Dicionário Usando Listas Encadeadas

```
void Insere(TipoItem x, TipoPesos p, TipoDicionario T)
{
  if (Pesquisa(x.Chave, p, T) == NULL)
  Ins(x, &T[h(x.Chave, p)]);
  else printf(" Registro ja esta presente\n");
}

void Retira(TipoItem x, TipoPesos p, TipoDicionario T)
{
  TipoApontador Ap;
  Ap = Pesquisa(x.Chave, p, T);
  if (Ap == NULL)
  printf(" Registro nao esta presente\n");
  else Ret(Ap, &T[h(x.Chave, p)], &x);
}
```

Estrutura do Dicionário Usando Endereçamento Aberto

```
#define VAZIO   "!!!!!!!!!!"
#define RETIRADO  "**********"
#define M  7
#define N  11   /* Tamanho da chave */

typedef unsigned int TipoApontador;
typedef char TipoChave[N];
typedef unsigned TipoPesos[N];
typedef struct TipoItem {
  /* outros componentes */
  TipoChave Chave;
} TipoItem;
typedef unsigned int TipoIndice;
typedef TipoItem TipoDicionario[M];
```


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.

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 $O(N)$.

Análise

- Seja $\alpha = N/M$ o fator de carga da tabela. Conforme demonstrado por Knuth (1973), o custo de uma pesquisa com sucesso é

$$C(n) = \frac{1}{2}\left(1 + \frac{1}{1-\alpha}\right) \cdot$$

- 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

```
void Insere(TipoItem x, TipoPesos p, TipoDicionario T)
{ unsigned int i = 0; unsigned int Inicial;
  if (Pesquisa(x.Chave,p,T) < M) { printf("Elemento ja esta presente\n"); return; }
  Inicial = h(x.Chave, p);
  while (strcmp(T[(Inicial + i) % M].Chave,VAZIO) != 0 &&
         strcmp(T[(Inicial + i) % M].Chave, RETIRADO) != 0 && i < M) i++;
  if (i < M)
  { strcpy(T[(Inicial + i) % M].Chave, x.Chave);
    /* Copiar os demais campos de x, se existirem */ }
  else printf(" Tabela cheia\n");
}
void Retira(TipoChave Ch, TipoPesos p, TipoDicionario T)
{ TipoIndice i;
  i = Pesquisa(Ch, p, T);
  if (i < M)
  memcpy(T[i].Chave, RETIRADO, N);
  else printf("Registro nao esta presente\n");
}
```


Operações do Dicionário Usando Endereçamento Aberto

```
void Inicializa(TipoDicionario T)
{ int i;
 for (i = 0; i < M; i++) memcpy(T[i].Chave, VAZIO, N);
}

TipoApontador Pesquisa(TipoChave Ch, TipoPesos p, TipoDicionario T)
{ unsigned int i = 0; unsigned int Inicial;
  Inicial = h(Ch, p);
  while (strcmp(T[(Inicial + i) % M].Chave,VAZIO) != 0 &&
         strcmp (T[(Inicial + i) % M].Chave, Ch) != 0 && i < M)
    i++;
  if (strcmp( T[(Inicial + i) % M].Chave, Ch) == 0)
  return ((Inicial + i) % M);
  else return M; /* Pesquisa sem sucesso */
}
```

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 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.

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 $0 \ldots N-1$.

Problema Resolvido Pelo Algoritmo

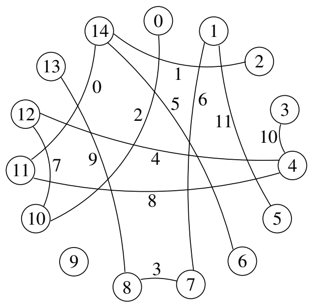
- Um **h**ipergrafo ou $r-$grafo é um grafo não direcionado no qual cada aresta conecta r vértices.
- Dado um hipergrafo não direcionado acíclico $G_r = (V, A)$, onde $|V| = M$ e $|A| = N$, encontre uma atribuição de valores aos vértices de G_r tal que a soma dos valores associados aos vértices de cada 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.

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 (1)

```
#define MAXNUMVERTICES 100000 /*—No. maximo de vertices—*/
#define MAXNUMARESTAS 100000 /*—No. maximo de arestas—*/
#define MAXR 5
#define MAXTAMPROX MAXR*MAXNUMARESTAS
#define MAXTAM 1000 /*—Usado Fila—*/
#define MAXTAMCHAVE 6 /*—No. maximo de caracteres da chave—*/
#define MAXNUMCHAVES 100000 /*—No. maximo de chaves lidas—*/
#define INDEFINIDO -1

typedef int TipoValorVertice;
typedef int TipoValorAresta;
typedef int Tipor;
typedef int TipoMaxTamProx;
```


Programa para Obter Função de Transformação Perfeita

```
int main()
{ Ler um conjunto de N chaves;
  Escolha um valor para M;
  do
    { Gera os pesos p1[i] e p2[i]
          para 1 <= i <= MAXTAMCHAVE
      Gera o grafo G = (V, A);
      Atribuig(G, g, GrafoRotulavel);
    } while (!GrafoRotulavel);
  Retorna  p1[i] e p2[i] e g;
}
```

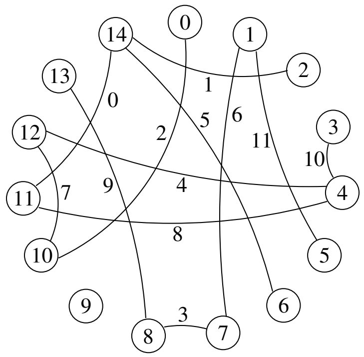
- Gera hipergrafos randômicos iterativamente e testa se o grafo gerado é acíclico.
- Cada iteração gera novas funções $h_0, h_1, \ldots, h_{r-1}$ até que um grafo acíclico seja obtido.
- A função de transformação perfeita é determinada pelos pesos $p_0, p_1, \ldots, p_{r-1}$, e pelo arranjo g.

Rotula Grafo e Atribui Valores para o Arranjo g

```
void Atribuig (TipoGrafo *Grafo,
               TipoArranjoArestas L,
               Tipog g)
{ int i, u, Soma; TipoValorVertice v;  TipoAresta a;
  for (i = Grafo->NumVertices - 1; i >= 0; i--) g[i] = INDEFINIDO;
  for (i = Grafo->NumArestas - 1; i >= 0; i--)
    { a = L[i]; Soma = 0;
      for (v = Grafo->r - 1; v >= 0; v--)
        { if (g[a.Vertices[v]] == INDEFINIDO) { u = a.Vertices[v]; g[u] = Grafo->NumArestas; }
          else Soma += g[a.Vertices[v]];
        }
        g[u] = a.Peso - Soma;
        if (g[u] < 0) g[u] = g[u]+(Grafo->r-1)*Grafo->NumArestas;
    }
}
```

- 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

Estruturas de dados (2)

```
typedef int TipoPesoAresta;
typedef TipoValorVertice TipoArranjoVertices[MAXR];
typedef struct TipoAresta {
  TipoArranjoVertices Vertices;
  TipoPesoAresta Peso;
} TipoAresta;
typedef TipoAresta TipoArranjoArestas[MAXNUMARESTAS];
typedef struct TipoGrafo {
  TipoArranjoArestas Arestas;
  TipoValorVertice Prim[MAXNUMVERTICES];
  TipoMaxTamProx Prox[MAXTAMPROX];
  TipoMaxTamProx ProxDisponivel;
  TipoValorVertice NumVertices;
  TipoValorAresta NumArestas;
  Tipor r;
} TipoGrafo;
```


Estruturas de dados (3)

```
typedef int TipoApontador;
typedef struct {
  TipoValorVertice Chave;
  /* outros componentes */
} TipoItem;
typedef struct {
  TipoItem Item[MAXTAM + 1];
  TipoApontador Frente, Tras;
} TipoFila;
typedef int TipoPesos[MAXTAMCHAVE];
typedef TipoPesos TipoTodosPesos[MAXR];
typedef int Tipog[MAXNUMVERTICES];
typedef char TipoChave[MAXTAMCHAVE];
typedef TipoChave TipoConjChaves[MAXNUMCHAVES];
typedef TipoValorVertice TipoIndice;
static TipoValorVertice M;
static TipoValorAresta N;
```


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

```
void GeraGrafo (TipoConjChaves   ConjChaves,
                TipoValorAresta  N,
                TipoValorVertice M,
                Tipor            r,
                TipoTodosPesos   Pesos,
                int              *NGrafosGerados,
                TipoGrafo        *Grafo)
{ /* Gera um grafo sem arestas repetidas e sem self-loops */
  int i, j;  TipoAresta Aresta;  int GrafoValido;

  inline int VerticesIguais (TipoAresta *Aresta)
  { int i, j;
    for (i = 0; i < Grafo->r - 1; i++)
    { for (j = i + 1; j < Grafo->r; j++)
      { if (Aresta->Vertices[i] == Aresta->Vertices[j])
        return TRUE;
      }
    }
```


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

```
  do
  { GrafoValido = TRUE; Grafo->NumVertices = M;
    Grafo->NumArestas = N; Grafo->r = r;
    FGVazio (Grafo); *NGrafosGerados = 0;
    for (j = 0; j < Grafo->r; j++) GeraPesos (Pesos[j]);
    for (i = 0; i < Grafo->NumArestas; i++)
      { Aresta.Peso = i;
        for (j = 0; j < Grafo->r; j++)
          Aresta.Vertices[j] = h (ConjChaves[i], Pesos[j]);
        if (VerticesIguais (&Aresta) || ExisteAresta (&Aresta, Grafo))
        { GrafoValido = FALSE;  break;  }
        else InsereAresta (&Aresta, Grafo);
      }
    ++(*NGrafosGerados);
  } while(!GrafoValido);
} /* Fim GeraGrafo */
```

Programa Principal para Gerar Arranjo g (1)

```
{—— Entram aqui as estruturas de dados dos slides 103, 104, 105 ——}
{—— 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 ——}
int main(){
  TipoR r; TipoGrafo Grafo; TipoArranjoArestas L; short GAciclico;
  Tipog g; TipoTodosPesos Pesos; int i, j; int NGrafosGerados;
  TipoConjChaves ConjChaves; FILE *ArqEntrada;
  FILE *ArqSaida; char NomeArq[100];
  printf ("Nome do arquivo com chaves a serem lidas: ");
  scanf("%s*[^\n]", NomeArq); printf("NomeArq = %s\n", NomeArq);
  ArqEntrada = fopen(NomeArq, "r");
  printf ("Nome do arquivo para gravar experimento: ");
  scanf("%s*[^\n]", NomeArq); printf("NomeArq = %s\n", NomeArq);
  ArqSaida = fopen(NomeArq, "w"); NGrafosGerados = 0; i = 0;
  fscanf(ArqEntrada, "%d%d%d*[^\n]", &N, &M, &r);
  Ignore(ArqEntrada, '\n'); printf("N=%d, M=%d, r=%d\n", N, M, r);
```


Programa Principal para Gerar Arranjo g (2)

```
while ((i < N) && (!feof(ArqEntrada)))
  { fscanf(ArqEntrada,"%s*[^\n]", ConjChaves[i]);
    Ignore(ArqEntrada, '\n'); printf("Chave[%d]=%s\n", i, ConjChaves[i]);
    i++;
  }
if (i != N)
  { printf("Erro: entrada com menos do que ', N, ' elementos.\n");
    exit(-1);
  }
  do{ GeraGrafo (ConjChaves, N, M, r, Pesos, &NGrafosGerados, &Grafo);
      ImprimeGrafo (&Grafo); /*—Imprime estrutura de dados—*/
      printf ("prim: ");
      for (i = 0; i < Grafo.NumVertices; i++) printf("%3d ", Grafo.Prim[i]);
      printf("\n"); printf ("prox: ");
      for (i = 0; i < Grafo.NumArestas * Grafo.r; i++)
          printf("%3d ", Grafo.Prox[i]);
      printf("\n"); GrafoAciclico (&Grafo, L, &GAciclico);
    } while (!GAciclico);
```


Programa Principal para Gerar Arranjo g (3)

```
printf ("Grafo aciclico com arestas retiradas:");
  for(i = 0; i < Grafo.NumArestas; i++) printf("%3d ", L[i].Peso);
  printf("\n");
  Atribuig (&Grafo, L, g);
  fprintf(ArqSaida, "%d  (N)\n", N);
  fprintf(ArqSaida, "%d  (M)\n", M);
  fprintf(ArqSaida, "%d  (r)\n", r);
  for (j = 0; j < Grafo.r; j++)
  { for (i = 0; i < MAXTAMCHAVE; i++)
      fprintf(ArqSaida, "%d ", Pesos[j][i]);
      fprintf(ArqSaida, "  (p%d)\n", j);
  }
  for (i = 0; i < M; i++) fprintf(ArqSaida, "%d ", g[i]);
  fprintf(ArqSaida, "  (g)\n");
  fprintf(ArqSaida, "No. grafos gerados por GeraGrafo:%d\n",
          NGrafosGerados);
  fclose (ArqSaida); fclose (ArqEntrada); return 0;
}
```


Função de Transformação Perfeita

```
TipoIndice hp (TipoChave Chave,
               TipoR r,
               TipoTodosPesos Pesos,
               Tipog g)
{ int i, v;
  v = 0;
  for (i = 0; i < r; i++) v += g[h(Chave, Pesos[i])];
  return (v % N);
} /* hp */
```

Teste para a Função de Transformação Perfeita (1)

```
#define MAXNUMVERTICES 100000 /*—No. maximo de vertices—*/
#define MAXNUMARESTAS 100000 /*—No. maximo de arestas—*/
#define MAXR 5
#define MAXTAMCHAVE 6 /*—No. maximo de caracteres da chave—*/
#define MAXNUMCHAVES 100000 /*—No. maximo de chaves lidas—*/
typedef int TipoValorVertice;
typedef int TipoValorAresta;
typedef int Tipor;
typedef int TipoPesos[MAXTAMCHAVE];
typedef TipoPesos TipoTodosPesos[MAXR];
typedef int Tipog[MAXNUMVERTICES];
typedef char TipoChave[MAXTAMCHAVE];
typedef TipoChave TipoConjChaves[MAXNUMCHAVES];
typedef TipoValorVertice TipoIndice;
static TipoValorVertice M;
static TipoValorAresta N;
```


Teste para a Função de Transformação Perfeita (2)

```
/** Entra aqui a funcao hash universal do slide 77 **/
/** Entra aqui a funcao hash perfeita do slide 111 **/
int main()
{ Tipor r; Tipog g; TipoTodosPesos Pesos; int i, j;
  TipoConjChaves ConjChaves;
  FILE *ArqChaves; FILE *ArqFHPM;
  char NomeArq[100]; TipoChave Chave;
  inline short VerificaFHPM()
  { short TabelaHash[MAXNUMVERTICES];
    int   i, indiceFHPM;
    for (i = 0; i < N; i++) TabelaHash[i] = FALSE;
    for (i = 0; i < N; i++)
      { indiceFHPM = hp (ConjChaves[i], r, Pesos, g);
        if ((TabelaHash[indiceFHPM])||(indiceFHPM >= N)) return FALSE;
        TabelaHash[indiceFHPM] = TRUE;
      }
    return TRUE;
  }
```


Teste para a Função de Transformação Perfeita (3)

```
printf ("Nome do arquivo com chaves a serem lidas: ");
  scanf("%s*[^\n]", NomeArq);
  printf("NomeArq = %s\n", NomeArq);
  ArqChaves = fopen(NomeArq, "r");
  fscanf(ArqChaves, "%d%d%d*[^\n]", &N, &M, &r);
  Ignore(ArqChaves, '\n');
  printf("N=%d, M=%d, r=%d\n", N, M, r);
  i = 0;
  while ((i < N) && (!feof(ArqChaves)))
  { fscanf(ArqChaves, "%s*[^\n]", ConjChaves[i]);
    Ignore(ArqChaves, '\n');
    printf("Chave[%d]=%s\n", i, ConjChaves[i]);
    i++;
  }
```


Teste para a Função de Transformação Perfeita (4)

```
  if (i != N)
  { printf("Erro: entrada com menos do que ', N, ' elementos.\n");
    exit(-1);
  }
  printf ("Nome do arquivo com a funcao hash perfeita: ");
  scanf("%s*[^\n]", NomeArq);
  printf("NomeArq = %s\n", NomeArq);
  ArqFHPM = fopen(NomeArq, "rb");
  fscanf(ArqFHPM, "%d*[^\n]", &N); Ignore(ArqFHPM, '\n');
  fscanf(ArqFHPM, "%d*[^\n]", &M); Ignore(ArqFHPM, '\n');
  fscanf(ArqFHPM, "%d*[^\n]", &r); Ignore(ArqFHPM, '\n');
  printf("N=%d, M=%d, r=%d\n", N, M, r);
  for (j = 0; j < r; j++)
  { for (i = 0; i < MAXTAMCHAVE; i++)
      fscanf(ArqFHPM, "%d*[^%d\n]", &Pesos[j][i]);
    Ignore(ArqFHPM, '\n');
    printf("\n");
```

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.

Teste para a Função de Transformação Perfeita (6)

```
  while (strcmp(Chave, "aaaaaa") != 0)
  { printf ("FHPM: %d\n", hp(Chave, r, Pesos, g));
    printf("Chave: ");
    scanf("%s*[^\n]", Chave);
  }
  fclose (ArqChaves);
  fclose (ArqFHPM);
  return 0;
}
```


Teste para a Função de Transformação Perfeita (5)

```
    for (i = 0; i < MAXTAMCHAVE; i++)
      printf("%d ", Pesos[j][i]);
    printf("  (p%d)\n", j);
  }
  for (i = 0; i < M; i++)
    fscanf(ArqFHPM, "%d*[%d\n]", &g[i]);
  Ignore(ArqFHPM, '\n');
  for (i = 0; i < M; i++) printf("%d ", g[i]);
  printf("  (g)\n");
  if (VerificaFHPM())
  printf ("FHPM foi gerada com sucesso\n");
  else printf ("FHPM nao foi gerada corretamente\n");
  printf("Chave: ");
  scanf("%s*[^\n]", Chave);
```

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.

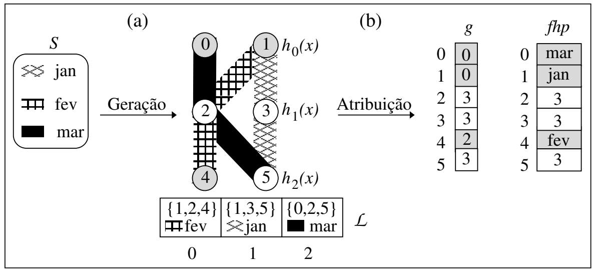
Análise: Espaço Ocupado para Descrever a Função

- 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 $O(1)$.

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 $r = 3$.

Hashing Perfeito Usando Espaço Quase Ótimo



(a) Para $S = \{\mathrm{jan}, \mathrm{fev}, \mathrm{mar}\}$, 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

- 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 103, 104 e 105.
- 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.

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 $\{h_0(\text{"mar"}), h_1(\text{"mar"}), h_2(\text{"mar"})\} = \{0, 2, 5\}$.

Rotula Grafo e Atribui Valores para o Arranjo g

```
void Atribuig (TipoGrafo *Grafo, TipoArranjoArestas L, Tipog g)
{ int i, j, u, Soma;  TipoValorVertice v;  TipoAresta a;
  unsigned char Visitado[MAXNUMVERTICES];
  for (i = Grafo->NumVertices - 1; i >= 0; i--)
    { g[i] = Grafo->r; Visitado[i] = FALSE; }
  for (i = Grafo->NumArestas - 1; i >= 0; i--)
    { a = L[i]; Soma = 0;
      for (v = Grafo->r - 1; v >= 0; v--)
        { if (!Visitado[a.Vertices[v]])
          { Visitado[a.Vertices[v]] = TRUE;
            u = a.Vertices[v]; j = v;
          }
          else Soma += g[a.Vertices[v]];
        }
      g[u] = (j - Soma) % Grafo->r;
      while (g[u] < 0) g[u] += Grafo->r;
    }
}
```


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**

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 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 105 o tipo do arranjo g é **integer**.

- Agora o comando

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

  muda para

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

Como Empacotar Quatro Valores de g em um *Byte*

```
/* Assume que todas as entradas de 2 bits do vetor */
/* g foram inicializadas com o valor 3            */
void AtribuiValor2Bits (Tipog *g, int Indice, unsigned char Valor)
{ int i, Pos; i   = Indice / 4;
  Pos = (Indice % 4);
  Pos = Pos * 2;          /* Cada valor ocupa 2 bits */
  g[i] &=  ~(3U << Pos);  /* zera os dois bits a atribuir */
  g[i] |= (Valor << Pos); /* realiza a atribuicao */
} /* AtribuiValor2Bits */

char ObtemValor2Bits (Tipog *g, int Indice)
{ int i, Pos;
  i = Indice / 4;
  Pos = (Indice % 4);
  Pos = Pos * 2; /* Cada valor ocupa 2 bits */
  return (g[i] >> Pos) & 3U;
} /* 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.

Obtem a Função Hash Perfeita

```
TipoIndice hp (TipoChave Chave,
               Tipor r,
               TipoTodosPesos Pesos,
               Tipog g)
{ int i, v = 0; TipoArranjoVertices a;
  for (i = 0; i < r; i++)
  { a[i] = h(Chave, Pesos[i]);
    v += g[a[i]];
  }
  v = v % r;  return a[v];
}
```

- 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 132 para descobrir qual foi o vértice da aresta escolhido para a chave.

Obtem a Função Hash Perfeita

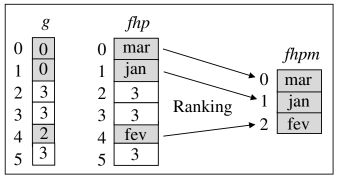
- A partir do arranjo g podemos obter uma função *hash* perfeita para uma tabela com intervalo $[0, M-1]$.
- Para uma chave $k \in S$ a função hp tem a seguinte forma:

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

- 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(“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(“fev”) $= 4$, e assim por diante.

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 $Bits$

```
TipoIndice hp (TipoChave Chave,
               Tipor r,
               TipoTodosPesos Pesos,
               Tipog g)
{ int i, v = 0; TipoArranjoVertices a;
  for (i = 0; i < r; i++)
  { a[i] = h(Chave, Pesos[i]);
    v += g[a[i]];
  }
  v = v % r;   return a[v];
}
```

- Basta substituir no programa do slide 133 o comando v := v + g[a[i]]; pelo comando v := v + ObtemValor2Bits(g, a[i]);

Atribui Valores para g Usando 2 *Bits* por Entrada

```
void Atribuig (TipoGrafo *Grafo, TipoArranjoArestas L, Tipog *g)
{ int i, j, u, Soma; TipoValorVertice v; TipoAresta a;
  unsigned int valorg2bits; unsigned char Visitado[MAXNUMVERTICES];
  if (Grafo->r <= 3) /* valores de 2 bits requerem r <= 3 */
  { for (i = Grafo->NumVertices - 1; i >= 0; i--)
    { AtribuiValor2Bits(g, i, Grafo->r); Visitado[i] = FALSE; }
    for (i = Grafo->NumArestas - 1; i >= 0; i--)
    { a = L[i]; Soma = 0;
      for (v = Grafo->r - 1; v >= 0; v--)
      { if (!Visitado[a.Vertices[v]])
        { Visitado[a.Vertices[v]] = TRUE; u = a.Vertices[v]; j = v; }
        else Soma += ObtemValor2Bits(g, a.Vertices[v]);
      }
      valorg2bits = (j - Soma) % Grafo->r;
      while (valorg2bits > Grafo->r) valorg2bits += Grafo->r;
      AtribuiValor2Bits (g, u, valorg2bits);
    }
  }
} /*--Fim 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 $\text{Valor} = 2$ na posição $\text{Indice} = 4$ de g (no caso, $g[4] = 2$):
  - No primeiro comando o *byte* que vai receber $\text{Valor} = 2 = (10)_2$ 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) $=$ **not** $((00000011)_2$ 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 $(\text{XXXXXX10})_2$, onde X representa 0 ou 1.

Implementação da Tabela T_r

```
void GeraTr (TipoTr Tr)
{ int i, j, v, Soma = 0;
  for (i = 0; i <= MAXTRVALUE; i++)
  { Soma = 0; v = i;
    for (j = 1; j <= 4; j++)
      { if ((v & 3) != NAOATRIBUIDO) Soma = Soma + 1;
        v = v >> 2;
      }
    Tr[i] = Soma;
  }
} /* GeraTr */
```


Implementação da Tabela T_r

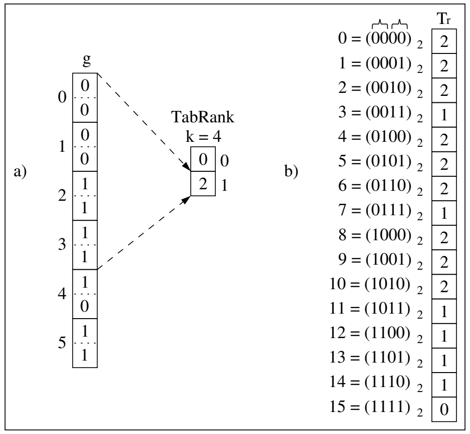
- Para calcular o $rank(u)$ usando as tabelas TabRank e T_r são necessários dois passos:
  - Obter o *rank* do maior índice precomputado $v \le u$ em TabRank.
  - Usar T_r para contar número de vértices atribuídos de v até $u - 1$.
- Na figura do slide 140 T_r possui 16 entradas necessárias para armazenas 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 $r \ge 2$ 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.

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.

```
void GeraTabRank (Tipog *g, TipoValorVertice Tamg,
                  TipoK k, TipoTabRank *TabRank)
{ int i, Soma = 0;
  for (i = 0; i < Tamg; i++)
    { if (i % k == 0) TabRank[i / k] = Soma;
      if (ObtemValor2Bits(g, i) != NAOATRIBUIDO) Soma = Soma + 1;
    }
} /* 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.

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 $b_k = 8$.

Função de Transformação Perfeita Usando 2 *Bits*

```
TipoIndice hpm (TipoChave Chave, Tipor r, TipoTodosPesos Pesos, Tipog * g,
                TipoTr Tr, TipoK k, TipoTabRank *TabRank)
{ TipoIndice i, j, u, Rank, Byteg;
  u = hp (Chave, r, Pesos, g);
  j = u / k;        Rank = TabRank[j];
  i = j * k;        j  = i;
  Byteg = j / 4;  j = j + 4;
  while (j < u)
  { Rank = Rank + Tr[g[Byteg]];
    j = j + 4;  Byteg = Byteg + 1;
  }
  j = j - 4;
  while (j < u)
  { if (ObtemValor2Bits (g,j) != NAOATRIBUIDO) Rank = Rank+1;
    j = j + 1;
  }
  return Rank;
} /* hpm */
```


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 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 132 envolve a avaliação de três funções *hash* universais, com um custo final $O(1)$.
- O tempo necessário para avaliar a função hpm do slide 144 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 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 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 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.