Pesquisa em Memória Primária*

Última alteração: 10 de Outubro de 2006

*Transparências elaboradas por Fabiano C. Botelho, Leonardo Rocha, Leonardo Mata e Nivio Ziviani

Pesquisa em Memória Primária

- Introdução - Conceitos Básicos
- Pesquisa Seqüencial
- Pesquisa Binária
- Árvores de Pesquisa
  - Árvores Binárias de Pesquisa sem Balanceamento
  - Árvores Binárias de Pesquisa com Balanceamento
    - Árvores SBB
    - Transformações para Manutenção da Propriedade SBB
- Pesquisa Digital
  - Trie
  - Patricia
- Transformação de Chave (*Hashing*)
  - Funções de Transformação
  - Listas Encadeadas
  - Endereçamento Aberto
  - *Hashing* Perfeito

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

Tabelas

- 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.

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 freqüentes.

     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 Seqüencial

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

Pesquisa Seqüencial

```
package cap5;
import cap4.Item; // vide programa do capítulo 4
public class Tabela {
  private Item registros[];
  private int n;

  public Tabela (int maxN) {
    this.registros = new Item[maxN+1];
    this.n = 0;
  }
  public int pesquisa (Item reg) {
    this.registros[0] = reg; // sentinela
    int i = this.n;
    while (this.registros[i].compara (reg) != 0) i--;
    return i;
  }
  public void insere (Item reg) throws Exception {
    if (this.n == (this.registros.length - 1))
      throw new Exception ("Erro: A tabela esta cheia");
    this.registros[++this.n] = reg;
  }
}
```


Pesquisa Seqüencial

- Cada registro contém um campo chave que identifica o registro.
- A Interface $Item$ definida no capítulo 4 foi utilizada por permitir a criação de métodos genéricos.
- Além da chave, podem existir outros componentes em um registro, os quais não têm influência nos algoritmos.
- O método $pesquisa$ retorna o índice do registro que contém a chave passada como parâmetro no registro reg; caso não esteja presente, o valor retornado é zero.
- Essa implementação não suporta mais de um registro com a mesma chave.

Pesquisa Seqüencial

- 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 seqüencial rápida**.

Pesquisa Seqüencial

Análise

- Pesquisa com sucesso:

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

- Pesquisa sem sucesso:

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

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

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.

Exemplo de Pesquisa Binária para a Chave G

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

Algoritmo de Pesquisa binária

- O algoritmo foi feito com um método da classe $Tabela$ apresentada anteriormente.

```
public int binaria (Item chave) {
  if (this.n == 0) return 0;
  int esq = 1, dir = this.n, i;
  do {
    i = (esq + dir) / 2;
    if (chave.compara (this.registros[i]) > 0) esq = i + 1;
    else dir = i − 1;
  } while ((chave.compara (this.registros[i]) != 0)
                                        && (esq <= dir));
  if (chave.compara (this.registros[i]) == 0) return i;
  else return 0;
}
```


Pesquisa Binária

Análise

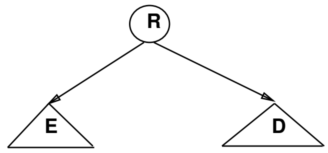
- A cada iteração do algoritmo, o tamanho da tabela é dividido ao meio.
- **Logo:** o número de vezes que o tamanho da tabela é dividido ao meio é cerca de $\log n$.
- **Ressalva:** o custo para manter a tabela ordenada é alto:
  a cada inserção na posição p da tabela implica no deslocamento dos registros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

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 seqüencial eficientes.
  2. Facilidade de inserção e retirada de registros.
  3. Boa taxa de utilização de memória.
  4. Utilização de memória primária e secundária.

Árvores Binárias de Pesquisa sem Balanceamento

- Para qualquer nó que contenha um registro

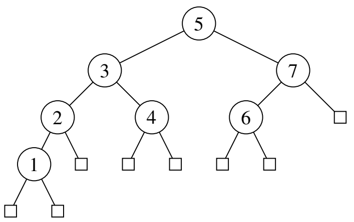


Temos a relação invariante

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

Exemplo



- 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:

- Contém as operações $inicializa$, $pesquisa$, $insere$ e $retira$.
- A operação $inicializa$ é implementada pelo construtor da classe $ArvoreBinaria$.

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

```
package cap5;
import cap4.Item; // vide programa do capítulo 4
public class ArvoreBinaria {
  private static class No {
    Item reg;
    No esq, dir;
  }
  private No raiz;
// Entram aqui os métodos privados das transparências 21, 22 e
26
  public ArvoreBinaria () {
    this.raiz = null;
  }
  public Item pesquisa (Item reg) {
    return this.pesquisa (reg, this.raiz);
  }
  public void insere (Item reg) {
    this.raiz = this.insere (reg, this.raiz);
  }
  public void retira (Item reg) {
    this.raiz = this.retira (reg, this.raiz);
  }
}
```


Método para Pesquisar na Árvore

Para encontrar um registro com uma chave reg:

- Compare-a com a chave que está na $raiz$.
- Se é menor, vá para a subárvore esquerda.
- Se 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 então o registro contendo a chave passada em reg é retornado.

```
private Item pesquisa (Item reg, No p) {
  if (p == null) return null; // Registro não encontrado
  else if (reg.compara (p.reg) < 0)
        return pesquisa (reg, p.esq);
  else if (reg.compara (p.reg) > 0)
        return pesquisa (reg, p.dir);
  else return p.reg;
}
```


Procedimento para Inserir na Árvore

- Atingir uma referência **n**ull em um processo de pesquisa significa uma pesquisa sem sucesso.
- Caso se queira inseri-lo na árvore, a referência **n**ull atingida é justamente o ponto de inserção.

```
private No insere (Item reg, No p) {
  if (p == null) {
    p = new No (); p.reg = reg;
    p.esq = null; p.dir = null;
  }
  else if (reg.compara (p.reg) < 0)
        p.esq = insere (reg, p.esq);
  else if (reg.compara (p.reg) > 0)
        p.dir = insere (reg, p.dir);
  else System.out.println ("Erro: Registro ja existente");
  return p;
}
```

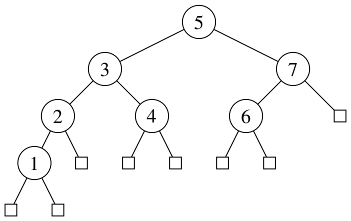

Programa para Criar a Árvore

```
package cap5;
import java.io.*;
import cap4.MeuItem; // vide programa do capítulo 4
public class CriaArvore {
  public static void main (String[] args) throws Exception {
    ArvoreBinaria dicionario = new ArvoreBinaria ();
    BufferedReader in = new BufferedReader (
                        new InputStreamReader (System.in));
    int chave = Integer.parseInt (in.readLine());
    while (chave > 0) {
      MeuItem item = new MeuItem (chave);
      dicionario.insere (item);
      chave = Integer.parseInt (in.readLine());
    }
  }
}
```

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.

Exemplo da Retirada de um Registro da Árvore

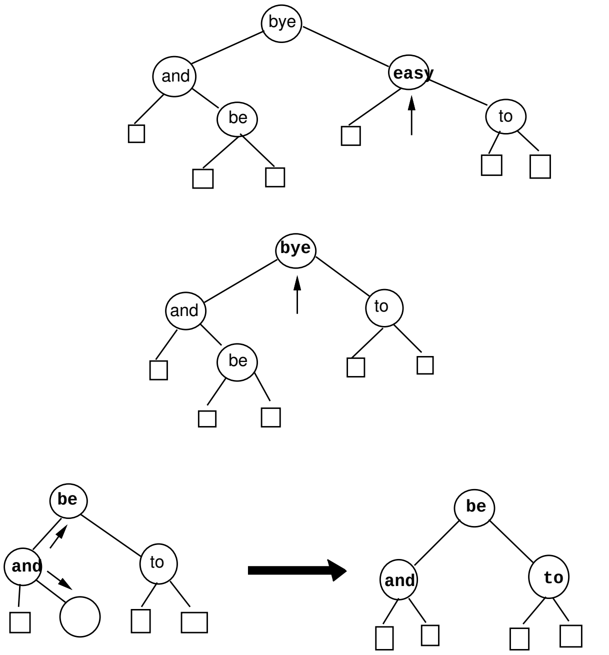


Assim: para retirar o registro com chave 5 na árvore basta trocá-lo pelo registro com chave 4 ou pelo registro com chave 6, e então retirar o nó que recebeu o registro com chave 5.

Método para retirar reg da árvore

```
private No antecessor (No q, No r) {
  if (r.dir != null) r.dir = antecessor (q, r.dir);
  else { q.reg = r.reg; r = r.esq; }
  return r;
}
private No retira (Item reg, No p) {
  if (p == null)
        System.out.println("Erro: Registro nao encontrado");
  else if (reg.compara (p.reg) < 0)
        p.esq = retira (reg, p.esq);
  else if (reg.compara (p.reg) > 0)
        p.dir = retira (reg, p.dir);
  else {
    if (p.dir == null) p = p.esq;
    else if (p.esq == null) p = p.dir;
    else p.esq = antecessor (p, p.esq);
  }
  return p;
}
```


Outro Exemplo de Retirada de Nó

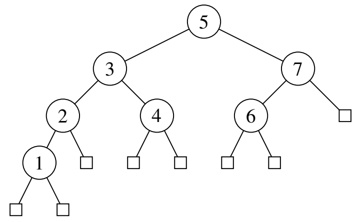


Caminhamento Central

- 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.

Caminhamento Central

- Percorrer a árvore:



  usando caminhamento central recupera as chaves na ordem 1, 2, 3, 4, 5, 6 e 7.

- Caminhamento $central$ e impressão da árvore:

```
public void imprime () { this.central (this.raiz); }

private void central (No p) {
  if (p != null) {
    central (p.esq);
    System.out.println (p.reg.toString());
    central (p.dir);
  }
}
```


Análise

- O número de comparações em uma pesquisa com sucesso:

$$\begin{aligned} &\text{melhor caso} &: C(n) = O(1),\\ &\text{pior caso} &: C(n) = O(n),\\ &\text{caso médio} &: C(n) = O(\log n). \end{aligned}$$

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

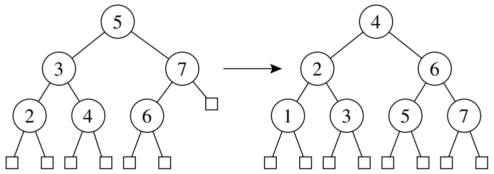
Análise

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.

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 do exemplo à esquerda e obter a árvore à direita do mesmo exemplo é necessário movimentar todos os nós da árvore original.
- **Exemplo:**



Uma Forma de Contornar este Problema

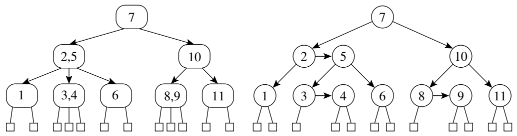
- Procurar solução intermediária que possa manter árvore "quase-balanceada", em vez de tentar manter a árvore completamente balanceada.
- **Objetivo:** Procurar obter bons tempos de pesquisa, próximos do tempo ótimo da árvore completamente balanceada, mas sem pagar muito para inserir ou retirar da árvore.
- **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 da transparência anterior é $8 = (0 + 1 + 1 + 2 + 2 + 2)$.

Árvores SBB

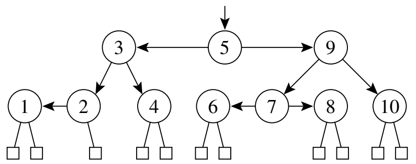
- Árvores B ⇒ estrutura para memória secundária. (Bayer R. e McCreight E.M., 1972)
- **Árvore 2-3** ⇒ caso especial da árvore B.
- Cada nó tem duas ou três subárvores.
- Mais apropriada para memória primária.
- **Exemplo: Uma árvore 2-3 e a árvore B binária correspondente**(Bayer, R. 1971)



Árvores SBB

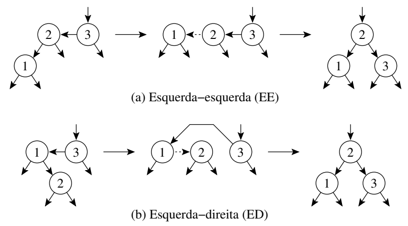
- Árvore 2-3 ⇒ **árvore B binária** (assimetria inerente)
  1. Referências à esquerda apontam para um nó no nível abaixo.
  2. Referências à direita podem ser verticais ou horizontais.

  Eliminação da assimetria nas árvores B binárias ⇒ árvores B binárias simétricas (*Symmetric Binary B-trees* – SBB)
- **Árvore SBB** é uma árvore binária com 2 tipos de referências: verticais e horizontais, tal que:
  1. todos os caminhos da raiz até cada nó externo possuem o mesmo número de referências verticais, e
  2. não podem existir dois referências 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 referência vertical mais baixo na árvore.
- Dependendo da situação anterior à inserção ou retirada, podem aparecer dois referências horizontais sucessivos
- **Neste caso:** é necessário realizar uma transformação.
- **Transformações Propostas por Bayer R. 1972**



(a) Esquerda−esquerda (EE)

(b) Esquerda−direita (ED)

Estrutura e operações do dicionário para árvores SBB

- Diferenças da árvore sem balanceamento:
  - constantes $Horizontal$ e $Vertical$: representam as inclinações das referências às subárvores;
  - campo $propSBB$: utilizado para verificar quando a propriedade SBB deixa de ser satisfeita
  - campos $incE$ e $incD$: indicam o tipo de referência (horizontal ou vertical) que sai do nó.
- A operação inicializa é implementada pelo construtor da classe $ArvoreSBB$.
- As demais operações são implementadas utilizando métodos privados sobrecarregados.

Estrutura e operações do dicionário para árvores SBB

```
package cap5;
import cap4.Item; // vide programa do capítulo 4
public class ArvoreSBB {
  private static class No {
    Item reg; No esq, dir; byte incE, incD;
  }
  private static final byte Horizontal = 0;
  private static final byte Vertical = 1;
  private No raiz; private boolean propSBB;

  // Entram aqui os métodos privados das transparências 21, 40,
41 e 48
  public ArvoreSBB () {
    this.raiz = null; this.propSBB = true;
  }
  public Item pesquisa (Item reg) {
    return this.pesquisa (reg, this.raiz); }
  public void insere (Item reg) {
    this.raiz = insere (reg, null, this.raiz, true); }
  public void retira (Item reg) {
    this.raiz = this.retira (reg, this.raiz); }
  // Entra aqui o método para imprimir a árvore da tranaparên-
cia 29
}
```

Métodos para manutenção da propriedade SBB

```
private No ee (No ap) {
  No ap1 = ap.esq; ap.esq = ap1.dir; ap1.dir = ap;
  ap1.incE = Vertical; ap.incE = Vertical; ap = ap1;
  return ap;
}
private No ed (No ap) {
  No ap1 = ap.esq; No ap2 = ap1.dir; ap1.incD = Vertical;
  ap.incE = Vertical; ap1.dir = ap2.esq; ap2.esq = ap1;
  ap.esq = ap2.dir; ap2.dir = ap; ap = ap2;
  return ap;
}
private No dd (No ap) {
  No ap1 = ap.dir; ap.dir = ap1.esq; ap1.esq = ap;
  ap1.incD = Vertical; ap.incD = Vertical; ap = ap1;
  return ap;
}
private No de (No ap) {
  No ap1 = ap.dir; No ap2 = ap1.esq; ap1.incE = Vertical;
  ap.incD = Vertical; ap1.esq = ap2.dir; ap2.dir = ap1;
  ap.dir = ap2.esq; ap2.esq = ap; ap = ap2;
  return ap;
}
```


Método para inserir na árvore SBB

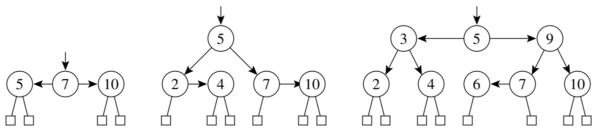
```
private No insere (Item reg, No pai, No filho, boolean filhoEsq) {
  if (filho == null) {
    filho = new No (); filho.reg = reg;
    filho.incE = Vertical; filho.incD = Vertical;
    filho.esq = null; filho.dir = null;
    if (pai != null)
      if (filhoEsq) pai.incE = Horizontal; else pai.incD = Horizontal;
    this.propSBB = false;
  }
  else if (reg.compara (filho.reg) < 0) {
    filho.esq = insere (reg, filho, filho.esq, true);
    if (!this.propSBB)
      if (filho.incE == Horizontal) {
        if (filho.esq.incE == Horizontal) {
          filho = this.ee (filho); // transformação esquerda-esquerda
          if (pai != null)
            if (filhoEsq) pai.incE=Horizontal; else pai.incD=Horizontal;
        }
      else if (filho.esq.incD == Horizontal) {
          filho = this.ed (filho); // transformação esquerda-direita
            if (pai != null)
            if (filhoEsq) pai.incE=Horizontal;
            else pai.incD=Horizontal;
        }
      }
      else this.propSBB = true;
}

      // Continua na próxima transparência
```


Método para inserir na árvore SBB

```
  else if (reg.compara (filho.reg) > 0) {
    filho.dir = insere (reg, filho, filho.dir, false);
    if (!this.propSBB)
      if (filho.incD == Horizontal) {
        if (filho.dir.incD == Horizontal) {
          filho = this.dd (filho); // transformação direita-direita
          if (pai != null)
            if (filhoEsq) pai.incE=Horizontal; else pai.incD=Horizontal;
        }
        else if (filho.dir.incE == Horizontal) {
          filho = this.de (filho); // transformação direita-esquerda
          if (pai != null)
            if (filhoEsq) pai.incE=Horizontal; else pai.incD=Horizontal;
        }
      }
      else this.propSBB = true;
  }
  else {
    System.out.println ("Erro: Registro ja existente");
    this.propSBB = true;
  }
  return filho;
}
```


Exemplo

- Inserção de uma seqüência de chaves em uma árvore SBB inicialmente vazia.
  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.



Procedimento Retira

- Assim como o método $insere$ mostrado anteriormente, o método $retira$ possui uma versão privada, que foi sobrecarregada com uma interface que contém um parâmetro a mais que a sua versão pública.
- O método privado $retira$ utiliza três métodos auxiliares, a saber:
  - $esqCurto$ ($dirCurto$) é chamado quando um nó folha (que é referenciado por uma referência 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 método $antecessor$ localiza o nó antecessor para ser trocado com o nó a ser retirado.

Método auxiliar $esqCurto$ para retirada da árvore SBB

```
// Folha esquerda retirada => árvore curta na altura esquerda
private No esqCurto (No ap) {
  if (ap.incE == Horizontal) {
    ap.incE = Vertical; this.propSBB = true;
  }
  else if (ap.incD == Horizontal) {
    No ap1 = ap.dir; ap.dir = ap1.esq; ap1.esq = ap; ap = ap1;
    if (ap.esq.dir.incE == Horizontal) {
      ap.esq = this.de (ap.esq); ap.incE = Horizontal;
    }
    else if (ap.esq.dir.incD == Horizontal) {
      ap.esq = this.dd (ap.esq); ap.incE = Horizontal;
    }
    this.propSBB = true;
  }
  else {
    ap.incD = Horizontal;
    if (ap.dir.incE == Horizontal) {
      ap = this.de (ap); this.propSBB = true;
    }
    else if (ap.dir.incD == Horizontal) {
      ap = this.dd (ap); this.propSBB = true;
    }
  } return ap;
}
```


Método auxiliar $dirCurto$ para retirada da árvore SBB

```
// Folha direita retirada => árvore curta na altura direita
private No dirCurto (No ap) {
  if (ap.incD == Horizontal) {
    ap.incD = Vertical; this.propSBB = true;
  }
  else if (ap.incE == Horizontal) {
    No ap1 = ap.esq; ap.esq = ap1.dir; ap1.dir = ap; ap = ap1;
    if (ap.dir.esq.incD == Horizontal) {
      ap.dir = this.ed (ap.dir); ap.incD = Horizontal;
    }
    else if (ap.dir.esq.incE == Horizontal) {
      ap.dir = this.ee (ap.dir); ap.incD = Horizontal;
    }
    this.propSBB = true;
  }
  else {
    ap.incE = Horizontal;
    if (ap.esq.incD == Horizontal) {
      ap = this.ed (ap); this.propSBB = true;
    }
    else if (ap.esq.incE == Horizontal) {
      ap = this.ee (ap); this.propSBB = true;
    }
  } return ap;
}
```


Método auxiliar $antecessor$ para retirada da árvore SBB

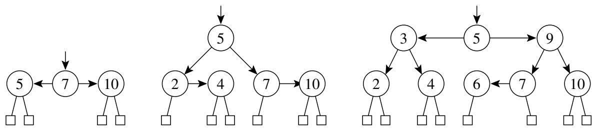
```
private No antecessor (No q, No r) {
  if (r.dir != null) {
    r.dir = antecessor (q, r.dir);
    if (!this.propSBB) r = this.dirCurto (r);
  }
  else {
    q.reg = r.reg;
    r = r.esq;
    if (r != null) this.propSBB = true;
  }
  return r;
}
```

Método *retira* para retirada da árvore SBB

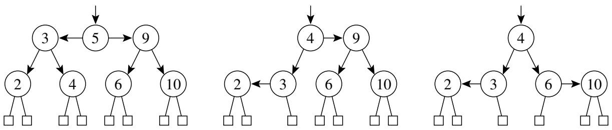
```
private No retira (Item reg, No ap) {
  if (ap == null) {
    System.out.println ("Erro: Registro nao encontrado");
    this.propSBB = true;
  }
  else if (reg.compara (ap.reg) < 0) {
    ap.esq = retira (reg, ap.esq);
    if (!this.propSBB)
      ap = this.esqCurto (ap);
  }
  else if (reg.compara (ap.reg) > 0) {
    ap.dir = retira (reg, ap.dir);
    if (!this.propSBB) ap = this.dirCurto (ap);
  }
  else { // encontrou o registro
    this.propSBB = false;
    if (ap.dir == null) {
      ap = ap.esq;
      if (ap != null) this.propSBB = true;
    }
    else if (ap.esq == null) {
      ap = ap.dir;
      if (ap != null)
        this.propSBB = true;
    }
    else {
      ap.esq = antecessor (ap, ap.esq);
      if (!this.propSBB)
        ap = this.esqCurto (ap);
    }
  }
  return ap;
}
```


Exemplo

- **Dada a Árvore:**

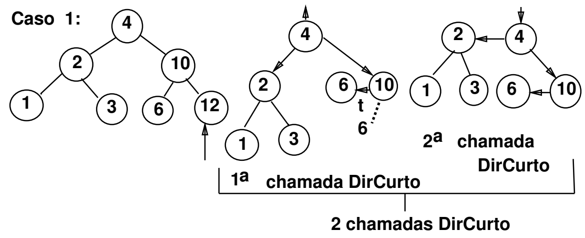


- Resultado obtido quando se retira uma seqüência de chaves da árvore SBB mais à direita acima:
  - A árvore à esquerda é obtida após a retirada da chave 7 da árvore à direita acima.
  - A árvore do meio é obtida após a retirada da chave 5 da árvore anterior.
  - A árvore à direita é obtida após a retirada da chave 9 da árvore anterior.

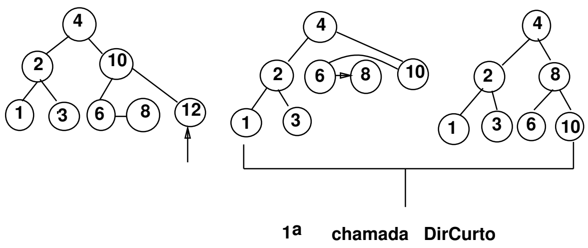


Exemplo: Retirada de Nós de SBB

Caso 1:

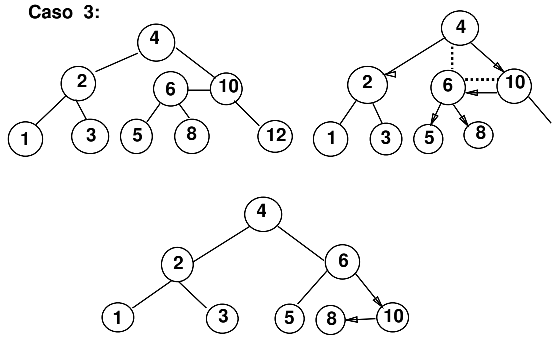


Caso 2:

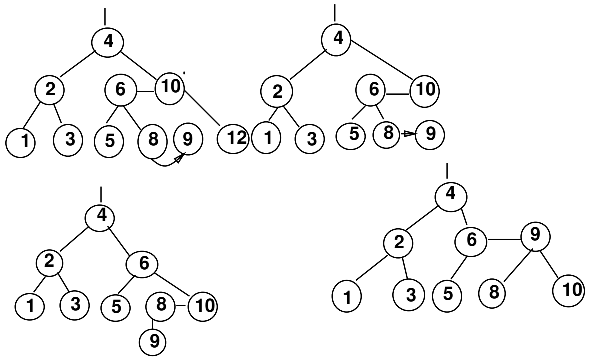


Exemplo: Retirada de Nós de SBB

Caso 3:



Se nodo 8 tem filho:



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 referências 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 referências no maior caminho entre a raiz e um nó externo.
- A altura k é maior que a altura h sempre que existirem referências horizontais na árvore.
- Para uma árvore SBB com n nós internos, temos que

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

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 na árvore SBB é

$$\begin{aligned} &\text{melhor caso} &&: C(n) = O(1), \\ &\text{pior caso} &&: C(n) = O(\log n), \\ &\text{caso médio} &&: C(n) = O(\log n). \end{aligned}$$

- **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 seqüência 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**

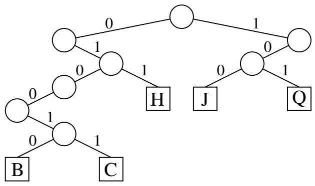
Trie

- 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 seqüência 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 seqüência 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.**

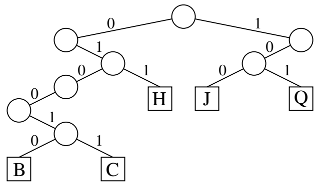
Exemplo

- **Dada as chaves de 6 bits:**

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

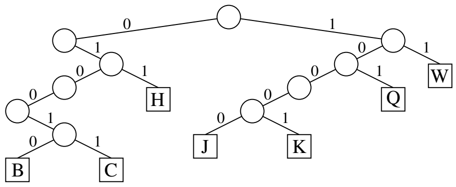


Inserção das Chaves W e K na Trie Binária



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.



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.

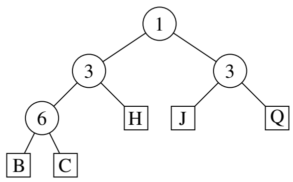
Patricia - Practical Algorithm To Retrieve Information Coded In Alphanumeric

- 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.

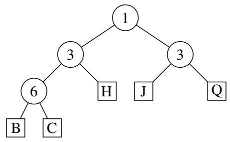
Mais sobre Patricia

- O algoritmo para construção da árvore Patricia é baseado no método de pesquisa digital, mas sem apresentar 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:

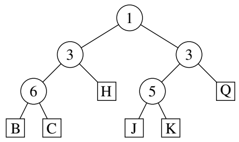
$$\begin{aligned} B &= 010010 \\ C &= 010011 \\ H &= 011000 \\ J &= 100001 \\ Q &= 101000 \end{aligned}$$



Inserção da Chave K

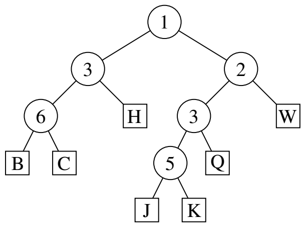


- Para inserir a chave K $=$ 100010 na árvore acima, a pesquisa inicia pela raiz e termina quando se chega ao nó externo contendo J.
- Os índices dos bits nas chaves estão ordenados da esquerda para a direita. Bit de índice 1 de K é 1 $\rightarrow$ a subárvore direita Bit de índice 3 $\rightarrow$ subárvore 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.



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, sendo, neste caso, 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.



Estrutura de dados e operações da árvore Patricia

- Em uma árvore Patricia existem dois tipos de nós diferentes: internos e externos. Para implementar essa característica foi utilizado o mecanismo de herança e polimorfismo da linguagem Java.

```
package cap5;
public class ArvorePatricia {
  private static abstract class PatNo { }
  private static class PatNoInt extends PatNo {
    int index;
    PatNo esq, dir;
  }
  private static class PatNoExt extends PatNo { char chave; }

  private PatNo raiz;
  private int nbitsChave;

  // Entram aqui os métodos privados das transparências 64, 65 e 68
  public ArvorePatricia (int nbitsChave) {
    this.raiz = null; this.nbitsChave = nbitsChave;
  }
  public void pesquisa (char k){ this.pesquisa (k, this.raiz);}
  public void insere (char k){ this.raiz = this.insere (k, this.raiz);}
}
```

Métodos Auxiliares

```
// Retorna o i-ésimo bit da chave k a partir da esquerda
private int bit (int i, char k) {
  if (i == 0) return 0;
  int c = (int)k;
  for (int j = 1; j <= this.nbitsChave − i; j++) c = c/2;
  return c % 2;
}
// Verifica se p é nó externo
private boolean eExterno (PatNo p) {
  Class classe = p.getClass ();
  return classe.getName().equals(PatNoExt.class.getName());
}
```

Método para criar nó interno:

```
private PatNo criaNoInt (int i, PatNo esq, PatNo dir) {
  PatNoInt p = new PatNoInt ();
  p.index = i; p.esq = esq; p.dir = dir;
  return p;
}
```


Métodos Auxiliares

Método para criar nó externo:

```
private PatNo criaNoExt (char k) {
  PatNoExt p = new PatNoExt ();
  p.chave = k;
  return p;
}
```

Método para pesquisa:

```
private void pesquisa (char k, PatNo t) {
  if (this.eExterno (t)) {
    PatNoExt aux = (PatNoExt)t;
    if (aux.chave == k) System.out.println ("Elemento encontrado");
    else System.out.println ("Elemento nao encontrado");
  }
  else {
    PatNoInt aux = (PatNoInt)t;
    if (this.bit (aux.index, k) == 0) pesquisa (k, aux.esq);
    else pesquisa (k, aux.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 (isso 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 seqüência 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. Caso contrário, ou seja, 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 pela referência à 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. Esse é o ponto de inserção e o par de nós é inserido.

Algoritmo de inserção

```
private PatNo insereEntre (char k, PatNo t, int i) {
  PatNoInt aux = null;
  if (!this.eExterno (t)) aux = (PatNoInt)t;
  if (this.eExterno (t) || (i < aux.index)) { // Cria um novo nó
externo
    PatNo p = this.criaNoExt (k);
    if (this.bit (i, k) == 1) return this.criaNoInt (i, t, p);
    else return this.criaNoInt (i, p, t);
  } else {
    if (this.bit (aux.index, k) == 1)
      aux.dir = this.insereEntre (k, aux.dir, i);
    else aux.esq = this.insereEntre (k, aux.esq, i);
    return aux;
  }
}

private PatNo insere (char k, PatNo t) {
  if (t == null) return this.criaNoExt (k);
  else {
    PatNo p = t;
    while (!this.eExterno (p)) {
      PatNoInt aux = (PatNoInt)p;
      if (this.bit (aux.index, k) == 1) p = aux.dir;
      else p = aux.esq;
    }
    PatNoExt aux = (PatNoExt)p;
    int i = 1; // acha o primeiro bit diferente
    while ((i <= this.nbitsChave)&&
           (this.bit (i, k) == this.bit (i, aux.chave))) i++;
    if (i > this.nbitsChave) {
      System.out.println ("Erro: chave ja esta na arvore");
      return t;
    }
    else return this.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.
- Como as transformações sobre as chaves são aritméticas, deve-se transformar as chaves não-numéricas em números.
- Em Java, basta realizar uma conversão de cada caractere da chave não numérica para um número inteiro.

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

- As chaves não numéricas devem ser transformadas em números:

$$K = \sum_{i=0}^{n-1} chave[i] \times p[i],$$

- n é o número de caracteres da chave.
- $chave[i]$ corresponde à representação **A**SCII ou **U**nicode do i-ésimo caractere da chave.
- $p[i]$ é um inteiro de um conjunto de pesos gerados randomicamente para $0 \leq i \leq n-1$.
- Vantagem de se usar pesos: Dois conjuntos diferentes de pesos $p_1[i]$ e $p_2[i]$, $0 \leq i \leq n-1$, levam a duas funções de transformação $h_1(K)$ e $h_2(K)$ diferentes.

Transformação de Chaves Não Numéricas

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

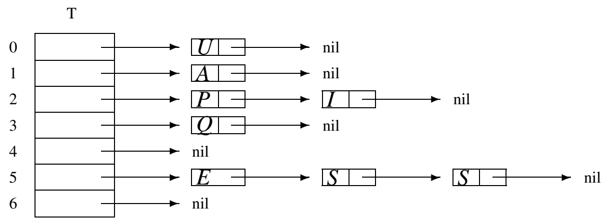
```
private int[] geraPesos (int n) {
  int p[] = new int[n];
  java.util.Random rand = new java.util.Random ();
  for (int i = 0; i < n; i++) p[i] = rand.nextInt(M) + 1;
  return p;
}
```

- **Implementação da função de transformação:**

```
private int h (String chave, int[] pesos) {
  int soma = 0;
  for (int i = 0; i < chave.length(); i++)
    soma = soma + ((int)chave.charAt (i)) * pesos[i];
  return soma % this.M;
}
```


Listas Encadeadas

- Uma das formas de resolver as **colisões** é simplesmente construir uma lista linear encadeada para cada endereço da tabela. Assim, todas as chaves com mesmo endereço são encadeadas em uma lista linear.
- **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:
- Por exemplo, $h(A) = h(1) = 1$, $h(E) = h(5) = 5$, $h(S) = h(19) = 5$, e assim por diante.



Estrutura e operações do dicionário para listas encadeadas

- Em cada entrada da lista devem ser armazenados uma $chave$ e um registro de dados cujo tipo depende da aplicação.
- A classe interna $Celula$ é utilizada para representar uma entrada em uma lista de chaves que são mapeadas em um mesmo endereço i da tabela, sendo $0 \leq i \leq M - 1$.
- O método $equals$ da classe $Celula$ é usado para verificar se duas células são iguais (isto é, possuem a mesma chave).
- A operação inicializa é implementada pelo construtor da classe $TabelaHash$.

Estrutura e operações do dicionário para listas encadeadas

```
package cap5.listaenc;
import cap3.autoreferencia.Lista; // vide Programas do capítulo 3
public class TabelaHash {
  private static class Celula {
    String chave; Object item;
    public Celula (String chave, Object item) {
      this.chave = chave; this.item = item;
    }
    public boolean equals (Object obj) {
      Celula cel = (Celula)obj;
      return chave.equals (cel.chave);
    }
  }
  private int M; // tamanho da tabela
  private Lista tabela[];
  private int pesos[];
  public TabelaHash (int m, int maxTamChave) {
    this.M = m; this.tabela = new Lista[this.M];
    for (int i = 0; i < this.M; i++)
        this.tabela[i] = new Lista ();
    this.pesos = this.geraPesos (maxTamChave);
  }
  // Entram aqui os métodos privados da transparência 76.
        // Continua na próxima transparência
```

Estrutura e operações do dicionário para listas encadeadas

```
public Object pesquisa (String chave) {
  int i = this.h (chave, this.pesos);
  if (this.tabela[i].vazia()) return null; // pesquisa sem sucesso
  else {
    Celula cel=(Celula)this.tabela[i].pesquisa(
                                   new Celula(chave,null));
    if (cel == null) return null; // pesquisa sem sucesso
    else return cel.item;
  }
}
public void insere (String chave, Object item) {
  if (this.pesquisa (chave) == null) {
    int i = this.h (chave, this.pesos);
    this.tabela[i].insere (new Celula (chave, item));
  }
  else System.out.println ("Registro ja esta presente");
}
public void retira (String chave) throws Exception {
  int i = this.h (chave, this.pesos);
  Celula cel = (Celula)this.tabela[i].retira (
                                   new Celula (chave,null));
  if (cel == null)
     System.out.println ("Registro nao esta presente");
} }
```


Análise

- Assumindo que qualquer item do conjunto tem igual probabilidade de ser endereçado para qualquer entrada da tabela, então o comprimento esperado de cada lista encadeada é N/M, em que 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, sendo que 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 torna-se constante, isto é, independente de N.

Endereçamento Aberto

- Quando o número de registros a serem armazenados na tabela puder ser previamente estimado, então não haverá necessidade de usar listas encadeadas 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 listas encadeadas em cada entrada dela.
- 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:

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

Exemplo

- Se a i-ésima letra do alfabeto é representada pelo número i e a função de transformação $h(chave) = chave \bmod M$ é utilizada para $M = 7$,
- então o resultado da inserção das chaves $L\ U\ N\ E\ S$ na tabela, 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*

Estrutura e operações do dicionário usando *endereçamento aberto*

- A tabela agora é constituída por um arranjo de células.
- A classe interna $Celula$ é utilizada para representar uma célula da tabela.
- A operação inicializa é implementada pelo construtor da classe $TabelaHash$.
- As operações utilizam alguns métodos auxiliares durante a execução.

Estrutura e operações do dicionário usando *endereçamento aberto*

```java
package cap5.endaberto;
public class TabelaHash {
  private static class Celula {
    String chave; Object item; boolean retirado;
    public Celula (String chave, Object item) {
      this.chave = chave; this.item = item;
      this.retirado = false;
    }
    public boolean equals (Object obj) {
      Celula cel = (Celula)obj;
      return chave.equals (cel.chave);
    }
  }
  private int M; // tamanho da tabela
  private Celula tabela[];
  private int pesos[];

  // Entram aqui os métodos privados da transparência 76
  public TabelaHash (int m, int maxTamChave) {
    this.M = m; this.tabela = new Celula[this.M];
    for (int i = 0; i < this.M; i++)
        this.tabela[i] = null; // vazio
    this.pesos = this.geraPesos (maxTamChave);
  }
        // Continua na próxima transparência
```


Estrutura e operações do dicionário usando *endereçamento aberto*

```java
public Object pesquisa (String chave) {
  int indice = this.pesquisaIndice (chave);
  if (indice < this.M) return this.tabela[indice].item;
  else return null; // pesquisa sem sucesso
}
public void insere (String chave, Object item) {
  if (this.pesquisa (chave) == null) {
    int inicial = this.h (chave, this.pesos);
    int indice = inicial; int i = 0;
    while (this.tabela[indice] != null &&
           !this.tabela[indice].retirado &&
           i < this.M)
              indice = (inicial + (++i)) % this.M;
    if (i < this.M) this.tabela[indice] =
             new Celula (chave, item);
    else System.out.println ("Tabela cheia");
  } else System.out.println ("Registro ja esta presente");
}
      // Continua na próxima transparência
```


Estrutura e operações do dicionário usando *endereçamento aberto*

```java
  public void retira (String chave) throws Exception {
    int i = this.pesquisaIndice (chave);
    if (i < this.M) {
      this.tabela[i].retirado = true;
      this.tabela[i].chave = null;
    } else System.out.println ("Registro nao esta presente");
  }
  private int pesquisaIndice (String chave) {
   int inicial = this.h (chave, this.pesos);
   int indice = inicial; int i = 0;
   while (this.tabela[indice] != null &&
          !chave.equals (this.tabela[indice].chave) &&
          i < this.M) indice = (inicial + (++i)) % this.M;
   if (this.tabela[indice] != null &&
       chave.equals (this.tabela[indice].chave))
         return indice;
   else return this.M;    // pesquisa sem sucesso
 }
}
```

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).
- Este fenômeno ocorre na medida em que a tabela começa a ficar cheia, pois a inserção de uma nova chave tende a ocupar uma posição na tabela que esteja contígua a outras posições já ocupadas, o que deteriora o tempo necessário para novas pesquisas.
- 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 $O(1)$.

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 $O(N)$.

Hashing Perfeito

- 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

- Não há necessidade de armazenar a chave, pois o registro é localizado sempre a partir do resultado da função de transformação.
- Uma função de transformação perfeita é específica para um conjunto de chaves conhecido.
- A desvantagem no caso é o espaço ocupado para descrever a função de transformação hp.
- Entretanto, é possível obter um método com $M \approx 1,25N$, para valores grandes de N.

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:

$$hp(x) = (g(h_1(x)) + g(h_2(x))) \bmod N,$$

na qual $h_1(x)$ e $h_2(x)$ são duas funções não perfeitas, 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

- Dado um grafo não direcionado $G = (V, A)$, onde $|V| = M$ e $|A| = N$, encontre uma função $g : V \to [0, N-1]$, definida como $hp(a = (u, v) \in A) = (g(u) + g(v)) \bmod N$.
- Em outras palavras, estamos procurando uma atribuição de valores aos vértices de G 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 grafos e hipergrafos randômicos.

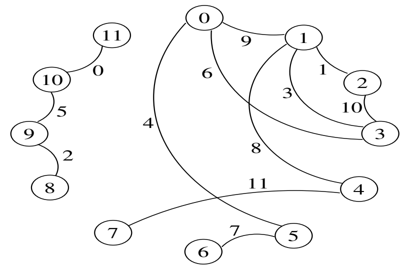
Exemplo

- **Chaves:** 12 meses do ano abreviados para os três primeiros caracteres.
- **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*:

Chave x	$h_1(x)$	$h_2(x)$	$hp(x)$
jan	10	11	0
fev	1	2	1
mar	8	9	2
abr	1	3	3
mai	0	5	4
jun	10	9	5
jul	0	3	6
ago	5	6	7
set	4	1	8
out	0	1	9
nov	3	2	10
dez	4	7	11

Grafo Randômico gerado

- O problema de obter a função g é equivalente a encontrar um grafo não direcionado contendo M vértices e N arestas.

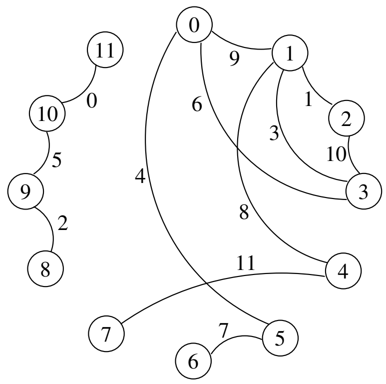


- Os vértices são rotulados com valores no intervalo $0 \ldots M-1$
- As arestas definidas por $(h_1(x), h_2(x))$ para cada uma das N chaves x.
- Cada chave corresponde a uma aresta que é rotulada com o valor desejado para a função hp perfeita.
- Os valores das duas funções $h_1(x)$ e $h_2(x)$ definem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

Obtenção da Função g a Partir do Grafo

- **Passo importante:** conseguir um arranjo g de vértices para inteiros no intervalo $0 \ldots N-1$ tal que, para cada aresta $(h_1(x), h_2(x))$, o valor de $hp(x) = g(h_1(x)) + g(h_2(x))) \bmod N$ seja igual ao rótulo da aresta.
- **Algoritmo:**
  1. Qualquer vértice não processado é escolhido e feito $g[v] = 0$.
  2. As arestas que saem do vértice v são seguidas e o valor $g(u)$ do vértice u destino é rotulado com o valor da diferença entre o valor da aresta (v, u) e $g(v)$, tomado $\bmod N$.
  3. Procura-se o próximo componente conectado ainda não visitado e os mesmos passos descritos acima são repetidos.

Seguindo o Algoritmo para Obter g no Exemplo dos 12 Meses do Ano



	Chave x	$h_1(x)$	$h_2(x)$	$hp(x)$		$v:$	$g(v)$
	jan	10	11	0		0	0
	fev	1	2	1		1	9
	mar	8	9	2		2	4
	abr	1	3	3		3	6
	mai	0	5	4		4	11
(a)	jun	10	9	5	(b)	5	4
	jul	0	3	6		6	3
	ago	5	6	7		7	0
	set	4	1	8		8	0
	out	0	1	9		9	2
	nov	3	2	10		10	3
	dez	4	7	11		11	9

Problema

- **Quando o grafo contém ciclos:** o mapeamento a ser realizado pode rotular de novo um vértice já processado e que tenha recebido outro rótulo com valor diferente.
- Por exemplo, se a aresta $(5, 6)$, que é a aresta de rótulo 7, tivesse sido sorteada para a aresta $(8, 11)$, o algoritmo tentaria atribuir dois valores distintos para o valor de $g[11]$.
- Para enxergar isso, vimos que se $g[8] = 0$, então $g[11]$ deveria ser igual a 7, e não igual ao valor 9 obtido acima.
- Um grafo que permite a atribuição de dois valores de g para um mesmo vértice, não é válido.
- Grafos acíclicos não possuem este problema.
- Um caminho seguro para se ter sucesso é obter antes um grafo acíclico e depois realizar a atribuição de valores para o arranjo g. Czech, Havas e Majewski (1992).

Primeiro Refinamento do Procedimento para Atribuir Valores ao Arranjo g

```
boolean rotuleDe (int v, int c, Grafo G, int g[]) {
  boolean grafoRotulavel = true;
  if (g[v] != Indefinido) if (g[v] != c)
              grafoRotulavel = false;
  else {
    g[v] = c;
    for (u ∈ G.listaAdjacentes (v))
      rotuleDe (u, (G.aresta (v,u) − g[v]) % N, g);
  }
  return grafoRotulavel;
}
boolean atribuig (Grafo G, int g[]) {
  boolean grafoRotulavel = true;
  for (int v = 0; v < M; v++) g[v] = Indefinido;
  for (int v = 0; v < M; v++)
    if (g[v] == Indefinido)
      grafoRotulavel = rotuleDe (v, 0, G, g);
  return grafoRotulavel;
}
```

Algoritmo para Obter a Função de Transformação Perfeita

```
void obtemHashingPerfeito () {
  Ler conjunto de N chaves;
  Escolha um valor para M;
  do {
    Gera os pesos p1[i] e p2[i] para
                    0 ≤ i ≤ maxTamChave − 1;
    Gera o grafo G = (V, A);
    grafoRotulavel = atribuig (G, g);
  } while (!grafoRotulavel);
  Retorna p1 , p2 e g;
}
```


Estruturas de dados e operações para obter a função *hash* perfeita

```java
package cap5.fhpm;
import java.io.*;
import cap7.listaadj.arranjo.Grafo; // vide Programas do capítulo 7
public class FHPM {
  private int p1[], p2[]; // pesos de h1 e h2
  private int g[]; // função g
  private int N;    // número de chaves
  private int M;    // número de vértices
  private int maxTamChave, nGrafosGerados, nGrafosConsiderados;
  private final int Indefinido = −1;
  // Entram aqui os métodos privados das transparências 76, 103 e 104
  public FHPM (int maxTamChave, int n, float c) {
    this.N = n; this.M = (int)(c*this.N);
    this.maxTamChave = maxTamChave; this.g = new int[this.M];
  }
  public void obtemHashingPerfeito (String nomeArqEnt)
                                      throws Exception {
    BufferedReader arqEnt = new BufferedReader (
                        new FileReader (nomeArqEnt));
    String conjChaves[] = new String[this.N];
    this.nGrafosGerados = 0;
    this.nGrafosConsiderados = 0; int i = 0;
// Continua na próxima transparência
```


Estruturas de dados e operações para obter a função *hash* perfeita

```java
    while ((i < this.N) &&
        ((conjChaves[i] = arqEnt.readLine()) != null)) i++;
    if (i != this.N)
      throw new Exception ("Erro: Arquivo de entrada possui"+
                           "menos que "+
                           this.N + " chaves");
    boolean grafoRotulavel = true;
    do {
      Grafo grafo = this.geraGrafo (conjChaves);
      grafoRotulavel = this.atribuig (grafo);
    }while (!grafoRotulavel);
    arqEnt.close ();
  }
  public int hp (String chave) {
    return (g[h (chave, p1)] + g[h (chave, p2)]) % N;
  }
  // Entram aqui os métodos públicos dos Programas 106 e 107
}
```


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

```java
private Grafo geraGrafo (String conjChaves[]) {
  Grafo grafo; boolean grafoValido;
  do {
    grafo = new Grafo (this.M, this.N); grafoValido = true;
    this.p1 = this.geraPesos (this.maxTamChave);
    this.p2 = this.geraPesos (this.maxTamChave);
    for (int i = 0; i < this.N; i++) {
      int v1 = this.h (conjChaves[i], this.p1);
      int v2 = this.h (conjChaves[i], this.p2);
      if ((v1 == v2) || grafo.existeAresta(v1, v2)) {
        grafoValido = false; grafo = null; break;
      } else {
        grafo.insereAresta (v1, v2, i);
        grafo.insereAresta (v2, v1, i);
      }
    }
    this.nGrafosGerados ++;
  } while (!grafoValido);
  return grafo;
}
```

Rotula Grafo e Atribui Valores para O Arranjo g

```
private boolean rotuleDe (int v, int c, Grafo grafo) {
  boolean grafoRotulavel = true;
  if (this.g[v] != Indefinido) {
    if (this.g[v] != c) {
      this.nGrafosConsiderados++; grafoRotulavel = false;
    }
  } else {
    this.g[v] = c;
    if (!grafo.listaAdjVazia (v)) {
      Grafo.Aresta adj = grafo.primeiroListaAdj (v);
      while (adj != null) {
        int u = adj.peso () − this.g[v];
        if (u < 0) u = u + this.N;
        grafoRotulavel = rotuleDe(adj.vertice2(),u,grafo);
        if (!grafoRotulavel) break; // sai do loop
        adj = grafo.proxAdj (v);
      }
    }
  }
  return grafoRotulavel;
}

// Continua na próxima transparência
```


Rotula Grafo e Atribui Valores para O Arranjo g

```
private boolean atribuig (Grafo grafo) {
  boolean grafoRotulavel = true;
  for (int v = 0; v < this.M; v++) this.g[v] = Indefinido;
  for (int v = 0; v < this.M; v++) {
    if (this.g[v] == Indefinido)
      grafoRotulavel = this.rotuleDe (v, 0, grafo);
    if (!grafoRotulavel) break;
  }
  return grafoRotulavel;
}
```


Método para salvar no disco a função de transformação perfeita

```
public void salvar (String nomeArqSaida) throws Exception {
  BufferedWriter arqSaida = new BufferedWriter (
                            new FileWriter (nomeArqSaida));
  arqSaida.write (this.N + " (N)\n");
  arqSaida.write (this.M + " (M)\n");
  arqSaida.write (this.maxTamChave + " (maxTamChave)\n");

  for (int i = 0; i < this.maxTamChave; i++)
    arqSaida.write (this.p1[i] + " ");
  arqSaida.write ("(p1)\n");

  for (int i = 0; i < this.maxTamChave; i++)
    arqSaida.write (this.p2[i] + " ");
  arqSaida.write ("(p2)\n");

  for (int i = 0; i < this.M; i++)
    arqSaida.write (this.g[i] + " ");
  arqSaida.write ("(g)\n");

  arqSaida.write ("No. grafos gerados por geraGrafo:" +
                  this.nGrafosGerados + "\n");
  arqSaida.write ("No. grafos considerados por atribuig:" +
                 (this.nGrafosConsiderados + 1) + "\n");
  arqSaida.close ();
}
```


Método para ler do disco a função de transformação perfeita

```
public void ler (String nomeArqFHPM) throws Exception {
  BufferedReader arqFHPM = new BufferedReader (
                           new FileReader (nomeArqFHPM));
  String temp = arqFHPM.readLine(), valor = temp.substring(0,
                                          temp.indexOf (" "));
  this.N = Integer.parseInt (valor);
  temp = arqFHPM.readLine(); valor = temp.substring(0,
                                   temp.indexOf (" "));
  this.M = Integer.parseInt (valor);
  temp = arqFHPM.readLine(); valor = temp.substring(0,
                                   temp.indexOf (" "));
  this.maxTamChave = Integer.parseInt (valor);
  temp = arqFHPM.readLine(); int inicio = 0;
  this.p1 = new int[this.maxTamChave];
  for (int i = 0; i < this.maxTamChave; i++) {
    int fim = temp.indexOf (' ', inicio);
    valor = temp.substring(inicio, fim);
    inicio = fim + 1; this.p1[i] = Integer.parseInt (valor);
  }
  temp = arqFHPM.readLine(); inicio = 0;
  this.p2 = new int[this.maxTamChave];
  for (int i = 0; i < this.maxTamChave; i++) {
    int fim = temp.indexOf (' ', inicio);
    valor = temp.substring(inicio, fim);
    inicio = fim + 1; this.p2[i] = Integer.parseInt (valor);
  }
  temp = arqFHPM.readLine(); inicio = 0;
  this.g = new int[this.M];
  for (int i = 0; i < this.M; i++) {
    int fim = temp.indexOf (' ', inicio); valor =
    temp.substring(inicio, fim);
    inicio = fim + 1; this.g[i] = Integer.parseInt (valor);
  }
  arqFHPM.close();
}
```

Programa para gerar uma função de transformação perfeita

```
package cap5;
import java.io.*;
import cap5.fhpm.FHPM; // vide transparência 101
public class GeraFHPM {
  public static void main (String[] args) {
    BufferedReader in = new BufferedReader (
                          new InputStreamReader (System.in));
    try {
      System.out.print ("Numero de chaves:");
      int n = Integer.parseInt (in.readLine ());
      System.out.print ("Tamanho da maior chave:");
      int maxTamChave = Integer.parseInt (in.readLine ());
      System.out.print("Nome do arquivo com chaves a serem lidas:");
      String nomeArqEnt = in.readLine ();
      System.out.print ("Nome do arquivo para gravar a FHPM:");
      String nomeArqSaida = in.readLine ();
      FHPM fhpm = new FHPM (maxTamChave, n, 3);
      fhpm.obtemHashingPerfeito (nomeArqEnt);
      fhpm.salvar (nomeArqSaida);
    } catch (Exception e) {System.out.println (e.getMessage ());}
  }
}
```


Programa para testar uma função de transformação perfeita

```
package cap5;
import java.io.*;
import cap5.fhpm.FHPM; // vide transparência 101
public class TestaFHPM {
  public static void main (String[] args) {
    BufferedReader in = new BufferedReader (
                          new InputStreamReader (System.in));
    try {
      System.out.print ("Nome do arquivo com a FHPM:");
      String nomeArqEnt = in.readLine ();
      FHPM fhpm = new FHPM (0, 0, 0);
      fhpm.ler (nomeArqEnt);
      System.out.print ("Chave:");String chave = in.readLine ();
      while (!chave.equals ("aaaaaa")) {
        System.out.println ("Indice: " + fhpm.hp (chave));
        System.out.print ("Chave:"); chave = in.readLine ();
      }
    } catch (Exception e) {System.out.println (e.getMessage ());}
  }
}
```


Análise

- **A questão crucial é:** quantas interações são necessárias para obter um grafo $G = (V, A)$ que seja rotulável?
- Para grafos arbitrários, é difícil achar uma solução para esse problema, isso se existir tal solução.
- Entretanto, para **grafos acíclicos**, a função g existe sempre e pode ser obtida facilmente.
- Assim, a resposta a esta questão depende do valor de M que é escolhido no primeiro passo do algoritmo.
- Quanto maior o valor de M, mais esparso é o grafo e, conseqüentemente, mais provável que ele seja acíclico.

Análise

- Segundo Czech, Havas e Majewski (1992), quando $M \leq 2N$ a probabilidade de gerar aleatoriamente um grafo acíclico tende para zero quando N cresce.
- Isto ocorre porque o grafo se torna denso, e o grande número de arestas pode levar à formação de ciclos.
- Por outro lado, quando $M > 2N$, a probabilidade de que um grafo randômico contendo M vértices e N arestas seja acíclico é aproximadamente

$$\sqrt{\frac{M - 2N}{M}},$$

- E o número esperado de grafos gerados até que o primeiro acíclico seja obtido é:

$$\sqrt{\frac{M}{M - 2N}}.$$

Análise

- Para $M = 3N$ o número esperado de iterações é $\sqrt{3}$, $\Rightarrow$ em média, aproximadamente 1,7 grafos serão testados antes que apareça um grafo acíclico.
- Logo, a complexidade de tempo para gerar a função de transformação é proporcional ao número de chaves a serem inseridas na tabela *hash*, desde que $M > 2N$.
- O grande inconveniente de usar $M = 3N$ é o espaço necessário para armazenar o arranjo g.
- Por outro lado, considerar $M < 2N$ pode implicar na necessidade de gerar muitos gráficos randômicos até que um grafo acíclico seja encontrado.

Outra Alternativa

- Não utilizar grafos tradicionais, mas sim **hipergrafos**, ou r-grafos, nos quais cada aresta conecta um número qualquer r de vértices.
- Para tanto, basta usar uma terceira função h_3 para gerar um trigrafo com arestas conectando três vértices, chamado de 3-grafo.
- Em outras palavras, cada aresta é uma tripla do tipo $(h_1(x), h_2(x), h_3(x))$, e a função de transformação é dada por:

$$h(x) = (g(h_1(x)) + g(h_2(x)) + g(h_3(x))) \bmod N.$$

Outra Alternativa

- Nesse caso, o valor de M pode ser próximo a $1,23N$.
- Logo, o uso de trigrafos reduz o custo de espaço da função de transformação perfeita, mas aumenta o tempo de acesso ao dicionário.
- Além disso, o processo de rotulação não pode ser feito como descrito.
- Ciclos devem ser detectados previamente, utilizando a seguinte propriedade de r-grafos:

  Um r-grafo é **acíclico** se e somente se a remoção repetida de arestas contendo apenas vértices de grau 1 (isto é, vértices sobre os quais incide apenas uma aresta) elimina todas as arestas do grafo.

Experimentos

# Chaves	# Chamadas $geraGrafo$	# Chamadas $atribuig$	Tempo (s)
10	3586	1	0.130
20	20795	16	0.217
30	55482	24	0.390
40	52077	33	0.432
50	47828	19	0.462
60	27556	10	0.313
70	26265	17	0.351
80	161736	92	1.543
90	117014	106	1.228
100	43123	26	0.559