Introdução*

Última alteração: 10 de Outubro de 2006

*Transparências elaboradas por Charles Ornelas, Leonardo Rocha, Leonardo Mata e Nivio Ziviani

Algoritmos, Estruturas de Dados e Programas

- Os algoritmos fazem parte do dia-a-dia das pessoas. Exemplos de algoritmos:
  - instruções para o uso de medicamentos,
  - indicações de como montar um aparelho,
  - uma receita de culinária.
- Seqüência de ações executáveis para a obtenção de uma solução para um determinado tipo de problema.
- Segundo Dijkstra, um algoritmo corresponde a uma descrição de um padrão de comportamento, expresso em termos de um conjunto finito de ações.
  - Executando a operação $a + b$ percebemos um padrão de comportamento, mesmo que a operação seja realizada para valores diferentes de a e b.

Estruturas de dados

- Estruturas de dados e algoritmos estão intimamente ligados:
  - não se pode estudar estruturas de dados sem considerar os algoritmos associados a elas,
  - assim como a escolha dos algoritmos em geral depende da representação e da estrutura dos dados.
- Para resolver um problema é necessário escolher uma abstração da realidade, em geral mediante a definição de um conjunto de dados que representa a situação real.
- A seguir, deve ser escolhida a forma de representar esses dados.

Escolha da Representação dos Dados

- A escolha da representação dos dados é determinada, entre outras, pelas operações a serem realizadas sobre os dados.
- Considere a operação de adição:
  - Para pequenos números, uma boa representação é por meio de barras verticais (caso em que a operação de adição é bastante simples).
  - Já a representação por dígitos decimais requer regras relativamente complicadas, as quais devem ser memorizadas.
  - Entretanto, quando consideramos a adição de grandes números é mais fácil a representação por dígitos decimais (devido ao princípio baseado no peso relativo da posição de cada dígito).

Programas

- Programar é basicamente estruturar dados e construir algoritmos.
- Programas são formulações concretas de algoritmos abstratos, baseados em representações e estruturas específicas de dados.
- Programas representam uma classe especial de algoritmos capazes de serem seguidos por computadores.
- Um computador só é capaz de seguir programas em linguagem de máquina (seqüência de instruções obscuras e desconfortáveis).
- É necessário construir linguagens mais adequadas, que facilitem a tarefa de programar um computador.
- Uma linguagem de programação é uma técnica de notação para programar, com a intenção de servir de veículo tanto para a expressão do raciocínio algorítmico quanto para a execução automática de um algoritmo por um computador.

Tipos de Dados

- Caracteriza o conjunto de valores a que uma constante pertence, ou que podem ser assumidos por uma variável ou expressão, ou que podem ser gerados por uma função.
- Tipos simples de dados são grupos de valores indivisíveis (como os tipos básicos *int, boolean, char* e *float* de Java).
  - Exemplo: uma variável do tipo *boolean* pode assumir o valor verdadeiro ou o valor falso, e nenhum outro valor.
- Os tipos estruturados em geral definem uma coleção de valores simples, ou um agregado de valores de tipos diferentes.

Tipos Abstratos de Dados (TAD's)

- Modelo matemático, acompanhado das operações definidas sobre o modelo.
  - Exemplo: o conjunto dos inteiros acompanhado das operações de adição, subtração e multiplicação.
- TAD's são utilizados extensivamente como base para o projeto de algoritmos.
- A implementação do algoritmo em uma linguagem de programação específica exige a representação do TAD em termos dos tipos de dados e dos operadores suportados.
- A representação do modelo matemático por trás do tipo abstrato de dados é realizada mediante uma estrutura de dados.
- Podemos considerar TAD's como generalizações de tipos primitivos e procedimentos como generalizações de operações primitivas.
- O TAD encapsula tipos de dados. A definição do tipo e todas as operações ficam localizadas numa seção do programa.

Implementação de TAD's

- Considere uma aplicação que utilize uma lista de inteiros. Poderíamos definir TAD Lista, com as seguintes operações:
  1. faça a lista vazia;
  2. obtenha o primeiro elemento da lista; se a lista estiver vazia, então retorne nulo;
  3. insira um elemento na lista.
- Há várias opções de estruturas de dados que permitem uma implementação eficiente para listas (por ex., o tipo estruturado arranjo).
- Cada operação do tipo abstrato de dados é implementada como um procedimento na linguagem de programação escolhida.
- Qualquer alteração na implementação do TAD fica restrita à parte encapsulada, sem causar impactos em outras partes do código.
- Cada conjunto diferente de operações define um TAD diferente, mesmo atuem sob um mesmo modelo matemático.
- A escolha adequada de uma implementação depende fortemente das operações a serem realizadas sobre o modelo.

Medida do Tempo de Execução de um Programa

- O projeto de algoritmos é fortemente influenciado pelo estudo de seus comportamentos.
- Depois que um problema é analisado e decisões de projeto são finalizadas, é necessário estudar as várias opções de algoritmos a serem utilizados, considerando os aspectos de tempo de execução e espaço ocupado.
- Muitos desses algoritmos são encontrados em áreas como pesquisa operacional, otimização, teoria dos grafos, estatística, probabilidades, entre outras.

Tipos de Problemas na Análise de Algoritmos

- **Análise de um algoritmo particular**.
  - Qual é o custo de usar um dado algoritmo para resolver um problema específico?
  - Características que devem ser investigadas:
    - análise do número de vezes que cada parte do algoritmo deve ser executada,
    - estudo da quantidade de memória necessária.
- **Análise de uma classe de algoritmos**.
  - Qual é o algoritmo de menor custo possível para resolver um problema particular?
  - Toda uma família de algoritmos é investigada.
  - Procura-se identificar um que seja o melhor possível.
  - Coloca-se **limites** para a complexidade computacional dos algoritmos pertencentes à classe.

Custo de um Algoritmo

- Determinando o menor custo possível para resolver problemas de uma dada classe, temos a medida da dificuldade inerente para resolver o problema.
- Quando o custo de um algoritmo é igual ao menor custo possível, o algoritmo é **ótimo** para a medida de custo considerada.
- Podem existir vários algoritmos para resolver o mesmo problema.
- Se a mesma medida de custo é aplicada a diferentes algoritmos, então é possível compará-los e escolher o mais adequado.

Medida do Custo pela Execução do Programa

- Tais medidas são bastante inadequadas e os resultados jamais devem ser generalizados:
  - os resultados são dependentes do compilador que pode favorecer algumas construções em detrimento de outras;
  - os resultados dependem do *hardware*;
  - quando grandes quantidades de memória são utilizadas, as medidas de tempo podem depender deste aspecto.
- Apesar disso, há argumentos a favor de se obterem medidas reais de tempo.
  - Ex.: quando há vários algoritmos distintos para resolver um mesmo tipo de problema, todos com um custo de execução dentro de uma mesma ordem de grandeza.
  - Assim, são considerados tanto os custos reais das operações como os custos não aparentes, tais como alocação de memória, indexação, carga, dentre outros.

Medida do Custo por meio de um Modelo Matemático

- Usa um modelo matemático baseado em um computador idealizado.
- Deve ser especificado o conjunto de operações e seus custos de execuções.
- É mais usual ignorar o custo de algumas das operações e considerar apenas as operações mais significativas.
- Ex.: algoritmos de ordenação. Consideramos o número de comparações entre os elementos do conjunto a ser ordenado e ignoramos as operações aritméticas, de atribuição e manipulações de índices, caso existam.

Função de Complexidade

- Para medir o custo de execução de um algoritmo é comum definir uma função de custo ou **função de complexidade** f.
- $f(n)$ é a medida do tempo necessário para executar um algoritmo para um problema de tamanho n.
- Função de **complexidade de tempo**: $f(n)$ mede o tempo necessário para executar um algoritmo em um problema de tamanho n.
- Função de **complexidade de espaço**: $f(n)$ mede a memória necessária para executar um algoritmo em um problema de tamanho n.
- Utilizaremos f para denotar uma função de complexidade de tempo daqui para a frente.
- A complexidade de tempo na realidade não representa tempo diretamente, mas o número de vezes que determinada operação considerada relevante é executada.

Exemplo - Maior Elemento

- Considere o algoritmo para encontrar o maior elemento de um vetor de inteiros $v[0..n-1], n \geq 1$.

```
package cap1;
public class Max {
  public static int max (int v[], int n) {
    int max = v[0];
    for (int i = 1; i < n; i++)
        if (max < v[i]) max = v[i];
    return max;
  }
}
```

- Seja f uma função de complexidade tal que $f(n)$ é o número de comparações entre os elementos de v, se v contiver n elementos.
- Logo $f(n) = n - 1, \text{ para } n > 0$.
- Vamos provar que o algoritmo apresentado no programa acima é **ótimo**.

Exemplo - Maior Elemento

- **Teorema**: Qualquer algoritmo para encontrar o maior elemento de um conjunto com n elementos, $n \geq 1$, faz pelo menos $n - 1$ comparações.
- **Prova**: Deve ser mostrado, por meio de comparações, que cada um dos $n - 1$ elementos é menor do que algum outro elemento.
- Logo $n - 1$ comparações são necessárias. □
- O teorema acima nos diz que, se o número de comparações for utilizado como medida de custo, então o método max da classe Max é ótimo.

Tamanho da Entrada de Dados

- A medida do custo de execução de um algoritmo depende principalmente do tamanho da entrada dos dados.
- É comum considerar o tempo de execução de um programa como uma função do tamanho da entrada.
- Para alguns algoritmos, o custo de execução é uma função da entrada particular dos dados, não apenas do tamanho da entrada.
- No caso do método max do programa do exemplo, o custo é uniforme sobre todos os problemas de tamanho n.
- Já para um algoritmo de ordenação isso não ocorre: se os dados de entrada já estiverem quase ordenados, então o algoritmo pode ter que trabalhar menos.

Melhor Caso, Pior Caso e Caso Médio

- **Melhor caso**: menor tempo de execução sobre todas as entradas de tamanho n.
- **Pior caso**: maior tempo de execução sobre todas as entradas de tamanho n.
- Se f é uma função de complexidade baseada na análise de pior caso, o custo de aplicar o algoritmo nunca é maior do que $f(n)$.
- **Caso médio** (ou caso esperado): média dos tempos de execução de todas as entradas de tamanho n.
- Na análise do caso esperado, supõe-se uma **distribuição de probabilidades** sobre o conjunto de entradas de tamanho n e o custo médio é obtido com base nessa distribuição.
- A análise do caso médio é geralmente muito mais difícil de obter do que as análises do melhor e do pior caso.
- É comum supor uma distribuição de probabilidades em que todas as entradas possíveis são igualmente prováveis.
- Na prática isso nem sempre é verdade.

Exemplo - Registros de um Arquivo

- Considere o problema de acessar os **registros** de um arquivo.
- Cada registro contém uma **chave** única que é utilizada para recuperar registros do arquivo.
- O problema: dada uma chave qualquer, localize o registro que contenha esta chave.
- O algoritmo de pesquisa mais simples é o que faz a **pesquisa seqüencial**.
- Seja f uma função de complexidade tal que $f(n)$ é o número de registros consultados no arquivo (número de vezes que a chave de consulta é comparada com a chave de cada registro).
  - melhor caso: $f(n) = 1$ (registro procurado é o primeiro consultado);
  - pior caso: $f(n) = n$ (registro procurado é o último consultado ou não está presente no arquivo);
  - caso médio: $f(n) = (n + 1)/2$.

Exemplo - Registros de um Arquivo

- No estudo do caso médio, vamos considerar que toda pesquisa recupera um registro.
- Se p_i for a probabilidade de que o i-ésimo registro seja procurado, e considerando que para recuperar o i-ésimo registro são necessárias i comparações, então $f(n) = 1 \times p_1 + 2 \times p_2 + 3 \times p_3 + \cdots + n \times p_n$.
- Para calcular $f(n)$ basta conhecer a distribuição de probabilidades p_i.
- Se cada registro tiver a mesma probabilidade de ser acessado que todos os outros, então $p_i = 1/n, 0 \leq i < n$.
- Neste caso $f(n) = \frac{1}{n}(1+2+3+\cdots+n) = \frac{1}{n}\left(\frac{n(n+1)}{2}\right) = \frac{n+1}{2}$.
- A análise do caso esperado revela que uma pesquisa com sucesso examina aproximadamente metade dos registros.

Exemplo - Maior e Menor Elemento (1)

- Considere o problema de encontrar o maior e o menor elemento de um vetor de inteiros $v[0..n-1], n \geq 1$.
- Um algoritmo simples pode ser derivado do algoritmo apresentado no programa para achar o maior elemento.
- O vetor $maxMin$ definido localmente no método $maxMin1$ é utilizado para retornar nas posições 0 e 1 o maior e o menor elemento do vetor v, respectivamente.

```
package cap1;
public class MaxMin1 {
  public static int [] maxMin1 (int v[], int n) {
    int max = v[0], min = v[0];
    for (int i = 1; i < n; i++) {
      if (v[i] > max) max = v[i];
      if (v[i] < min) min = v[i];
    }
    int maxMin[] = new int[2];
    maxMin[0] = max; maxMin[1] = min;
    return maxMin;
  }
}
```

Exemplo - Maior e Menor Elemento (1)

- Seja $f(n)$ o número de comparações entre os elementos de v, se v contiver n elementos.
- Logo $f(n) = 2(n-1), \text{ para } n > 0$, para o melhor caso, pior caso e caso médio.
- MaxMin1 pode ser facilmente melhorado: a comparação $v[i] < min$ só é necessária quando a comparação $v[i] > max$ é falsa.
- A seguir, apresentamos essa versão melhorada.

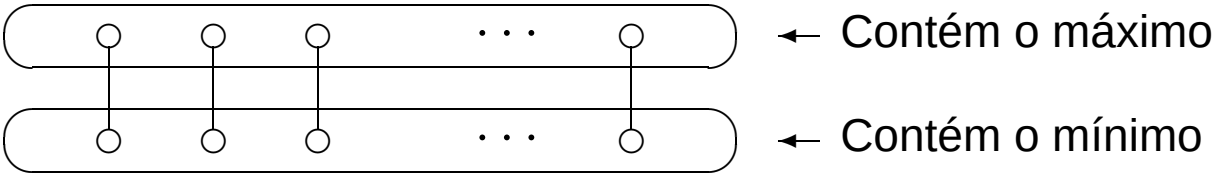
Exemplo - Maior e Menor Elemento (2)

```
package cap1;
public class MaxMin2 {
  public static int [] maxMin2 (int v[], int n) {
    int max = v[0], min = v[0];
    for (int i = 1; i < n; i++) {
      if (v[i] > max) max = v[i];
      else if (v[i] < min) min = v[i];
    }
    int maxMin[] = new int[2];
    maxMin[0] = max; maxMin[1] = min;
    return maxMin;
  }
}
```

- Para a nova implementação temos:
  - melhor caso: $f(n) = n - 1$ (quando os elementos estão em ordem crescente);
  - pior caso: $f(n) = 2(n - 1)$ (quando os elementos estão em ordem decrescente);
  - caso médio: $f(n) = 3n/2 - 3/2$.
- No caso médio, $v[i]$ é maior do que max a metade das vezes.
- Logo $f(n) = n - 1 + \frac{n-1}{2} = \frac{3n}{2} - \frac{3}{2}$, para $n > 0$.

Exemplo - Maior e Menor Elemento (3)

- Considerando o número de comparações realizadas, existe a possibilidade de obter um algoritmo mais eficiente:
  1. Compare os elementos de v aos pares, separando-os em dois subconjuntos (maiores em um e menores em outro), a um custo de $\lceil n/2 \rceil$ comparações.
  2. O máximo é obtido do subconjunto que contém os maiores elementos, a um custo de $\lceil n/2 \rceil - 1$ comparações.
  3. O mínimo é obtido do subconjunto que contém os menores elementos, a um custo de $\lceil n/2 \rceil - 1$ comparações.



Exemplo - Maior e Menor Elemento (3)

```
package cap1;
public class MaxMin3 {
  public static int [] maxMin3 (int v[], int n) {
    int max, min, FimDoAnel;
    if ((n % 2) > 0) { v[n] = v[n−1]; FimDoAnel = n; }
    else FimDoAnel = n−1;
    if (v[0] > v[1]) { max = v[0]; min = v[1]; }
    else { max = v[1]; min = v[0]; }
    int i = 2;
    while (i < FimDoAnel) {
      if (v[i] > v[i+1]) {
        if (v[i] > max) max = v[i];
        if (v[i+1] < min) min = v[i+1];
      }
      else {
        if (v[i] < min) min = v[i];
        if (v[i+1] > max) max = v[i+1];
      }
      i = i + 2;
    }
    int maxMin[] = new int[2];
    maxMin[0] = max; maxMin[1] = min;
    return maxMin;
  }
}
```

Exemplo - Maior e Menor Elemento (3)

- Os elementos de v são comparados dois a dois e os elementos maiores são comparados com max e os elementos menores são comparados com min.
- Quando n é ímpar, o elemento que está na posição $v[n-1]$ é duplicado na posição $v[n]$ para evitar um tratamento de exceção.
- Para esta implementação, $f(n) = \frac{n}{2} + \frac{n-2}{2} + \frac{n-2}{2} = \frac{3n}{2} - 2$, para $n > 0$, para o melhor caso, pior caso e caso médio.

Comparação entre os Algoritmos MaxMin1, MaxMin2 e MaxMin3

- A tabela apresenta uma comparação entre os algoritmos dos programas MaxMin1, MaxMin2 e MaxMin3, considerando o número de comparações como medida de complexidade.
- Os algoritmos MaxMin2 e MaxMin3 são superiores ao algoritmo MaxMin1 de forma geral.
- O algoritmo MaxMin3 é superior ao algoritmo MaxMin2 com relação ao pior caso e bastante próximo quanto ao caso médio.

Os três algoritmos	$f(n)$		
	Melhor caso	Pior caso	Caso médio
MaxMin1	$2(n-1)$	$2(n-1)$	$2(n-1)$
MaxMin2	$n-1$	$2(n-1)$	$3n/2-3/2$
MaxMin3	$3n/2-2$	$3n/2-2$	$3n/2-2$

Limite Inferior - Uso de um Oráculo

- Existe possibilidade de obter um algoritmo MaxMin mais eficiente?
- Para responder temos de conhecer o **limite inferior** para essa classe de algoritmos.
- Técnica muito utilizada: uso de um oráculo.
- Dado um modelo de computação que expresse o comportamento do algoritmo, o oráculo informa o resultado de cada passo possível (no caso, o resultado de cada comparação).
- Para derivar o limite inferior, o oráculo procura sempre fazer com que o algoritmo trabalhe o máximo, escolhendo como resultado da próxima comparação aquele que cause o maior trabalho possível necessário para determinar a resposta final.

Exemplo de Uso de um Oráculo

- **Teorema**: Qualquer algoritmo para encontrar o maior e o menor elementos de um conjunto com n elementos não ordenados, $n \geq 1$, faz pelo menos $3\lceil n/2 \rceil - 2$ comparações.
- **Prova**: A técnica utilizada define um oráculo que descreve o comportamento do algoritmo por meio de um conjunto de n–tuplas, mais um conjunto de regras associadas que mostram as tuplas possíveis (estados) que um algoritmo pode assumir a partir de uma dada tupla e uma única comparação.
- Uma 4–tupla, representada por (a, b, c, d), onde os elementos de:
  - $a \rightarrow$ nunca foram comparados;
  - $b \rightarrow$ foram vencedores e nunca perderam em comparações realizadas;
  - $c \rightarrow$ foram perdedores e nunca venceram em comparações realizadas;
  - $d \rightarrow$ foram vencedores e perdedores em comparações realizadas.

Exemplo de Uso de um Oráculo

- O algoritmo inicia no estado $(n, 0, 0, 0)$ e termina com $(0, 1, 1, n-2)$.
- Após cada comparação a tupla (a, b, c, d) consegue progredir apenas se ela assume um dentre os seis estados possíveis abaixo:
  - $(a-2, b+1, c+1, d)$ se $a \geq 2$ (dois elementos de a são comparados)
  - $(a-1, b+1, c, d)$ ou $(a-1, b, c+1, d)$ ou $(a-1, b, c, d+1)$ se $a \geq 1$ (um elemento de a comparado com um de b ou um de c)
  - $(a, b-1, c, d+1)$ se $b \geq 2$ (dois elementos de b são comparados)
  - $(a, b, c-1, d+1)$ se $c \geq 2$ (dois elementos de c são comparados)
  - O primeiro passo requer necessariamente a manipulação do componente a.
  - O caminho mais rápido para levar a até zero requer $\lceil n/2 \rceil$ mudanças de estado e termina com a tupla $(0, n/2, n/2, 0)$ (por meio de comparação dos elementos de a dois a dois).

Exemplo de Uso de um Oráculo

- A seguir, para reduzir o componente b até um são necessárias $\lceil n/2 \rceil - 1$ mudanças de estado (mínimo de comparações necessárias para obter o maior elemento de b).
- Idem para c, com $\lceil n/2 \rceil - 1$ mudanças de estado.
- Logo, para obter o estado $(0, 1, 1, n-2)$ a partir do estado $(n, 0, 0, 0)$ são necessárias

$$\lceil n/2 \rceil + \lceil n/2 \rceil - 1 + \lceil n/2 \rceil - 1 = \lceil 3n/2 \rceil - 2$$

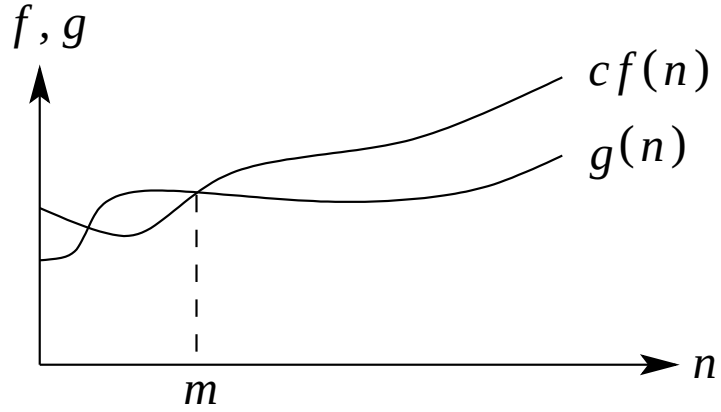
  comparações. □
- O teorema nos diz que se o número de comparações entre os elementos de um vetor for utilizado como medida de custo, então o algoritmo MaxMin3 é **ótimo**.

Comportamento Assintótico de Funções

- O parâmetro n fornece uma medida da dificuldade para se resolver o problema.
- Para valores suficientemente pequenos de n, qualquer algoritmo custa pouco para ser executado, mesmo os ineficientes.
- A **escolha do algoritmo** não é um problema crítico para problemas de tamanho pequeno.
- Logo, a análise de algoritmos é realizada para valores grandes de n.
- Estuda-se o comportamento assintótico das **funções de custo** (comportamento de suas funções de custo para valores grandes de n)
- O comportamento assintótico de $f(n)$ representa o limite do comportamento do custo quando n cresce.

Dominação assintótica

- A análise de um algoritmo geralmente conta com apenas algumas operações elementares.
- A medida de custo ou medida de complexidade relata o crescimento assintótico da operação considerada.
- **Definição**: Uma função $f(n)$ **domina assintoticamente** outra função $g(n)$ se existem duas constantes positivas c e m tais que, para $n \geq m$, temos $|g(n)| \leq c \times |f(n)|$.

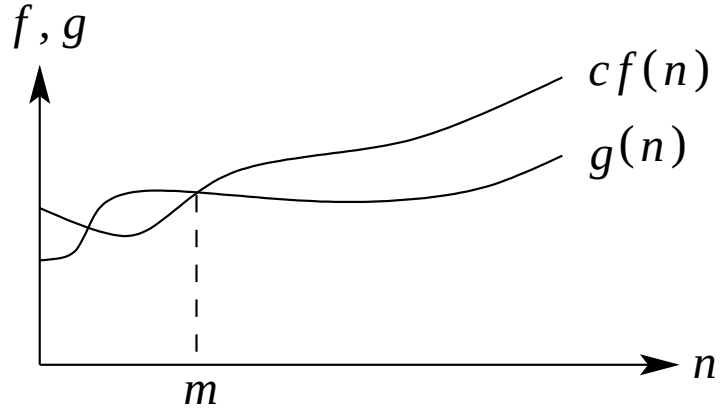


Exemplo:

- Sejam $g(n) = (n+1)^2$ e $f(n) = n^2$.
- As funções $g(n)$ e $f(n)$ dominam assintoticamente uma a outra, desde que $|(n+1)^2| \leq 4|n^2|$ para $n \geq 1$ e $|n^2| \leq |(n+1)^2|$ para $n \geq 0$.

Notação O

- Escrevemos $g(n) = O(f(n))$ para expressar que $f(n)$ domina assintoticamente $g(n)$. Lê-se $g(n)$ é da ordem no máximo $f(n)$.
- Exemplo: quando dizemos que o tempo de execução $T(n)$ de um programa é $O(n^2)$, significa que existem constantes c e m tais que, para valores de $n \geq m$, $T(n) \leq cn^2$.
- Exemplo gráfico de dominação assintótica que ilustra a notação O.



  O valor da constante m mostrado é o menor valor possível, mas qualquer valor maior também é válido.
- **Definição**: Uma função $g(n)$ é $O(f(n))$ se existem duas constantes positivas c e m tais que $g(n) \leq cf(n)$, para todo $n \geq m$.

Exemplos de Notação O

- **Exemplo**: $g(n) = (n+1)^2$.
  - Logo $g(n)$ é $O(n^2)$, quando $m = 1$ e $c = 4$.
  - Isso porque $(n+1)^2 \leq 4n^2$ para $n \geq 1$.
- **Exemplo**: $g(n) = n$ e $f(n) = n^2$.
  - Sabemos que $g(n)$ é $O(n^2)$, pois para $n \geq 0$, $n \leq n^2$.
  - Entretanto $f(n)$ não é $O(n)$.
  - Suponha que existam constantes c e m tais que para todo $n \geq m$, $n^2 \leq cn$.
  - Logo $c \geq n$ para qualquer $n \geq m$, e não existe uma constante c que possa ser maior ou igual a n para todo n.

Exemplos de Notação O

- **Exemplo**: $g(n) = 3n^3 + 2n^2 + n$ é $O(n^3)$.
  - Basta mostrar que $3n^3 + 2n^2 + n \leq 6n^3$, para $n \geq 0$.
  - A função $g(n) = 3n^3 + 2n^2 + n$ é também $O(n^4)$, entretanto esta afirmação é mais fraca do que dizer que $g(n)$ é $O(n^3)$.
- **Exemplo**: $g(n) = \log_5 n$ é $O(\log n)$.
  - O $\log_b n$ difere do $\log_c n$ por uma constante que no caso é $\log_b c$.
  - Como $n = c^{\log_c n}$, tomando o logaritmo base b em ambos os lados da igualdade, temos que
    $\log_b n = \log_b c^{\log_c n} = \log_c n \times \log_b c$.

Operações com a Notação O

$$
\begin{array}{rcll}
f(n) & = & O(f(n)) & \\
c \times O(f(n)) & = & O(f(n)) & c = constante \\
O(f(n)) + O(f(n)) & = & O(f(n)) & \\
O(O(f(n)) & = & O(f(n)) & \\
O(f(n)) + O(g(n)) & = & O(max(f(n), g(n))) & \\
O(f(n))O(g(n)) & = & O(f(n)g(n)) & \\
f(n)O(g(n)) & = & O(f(n)g(n)) &
\end{array}
$$

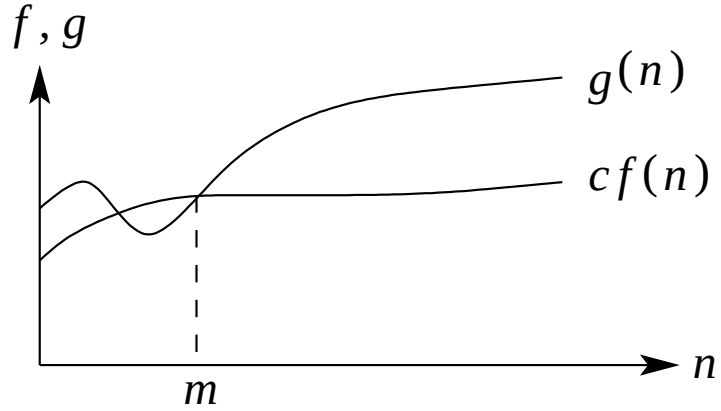
Exemplo: regra da soma $O(f(n)) + O(g(n))$.

- Suponha três trechos cujos tempos de execução são $O(n)$, $O(n^2)$ e $O(n \log n)$.
- O tempo de execução dos dois primeiros trechos é $O(max(n, n^2))$, que é $O(n^2)$.
- O tempo de execução de todos os três trechos é então $O(max(n^2, n \log n))$, que é $O(n^2)$.

Exemplo: O produto de $[\log n + k + O(1/n)]$ por $[n + O(\sqrt{n})]$ é $n \log n + kn + O(\sqrt{n} \log n)$.

Notação Ω

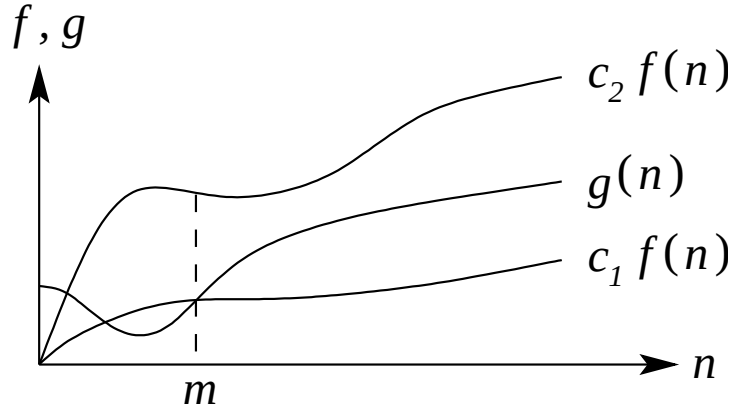
- Especifica um limite inferior para $g(n)$.
- **Definição**: Uma função $g(n)$ é $\Omega(f(n))$ se existirem duas constantes c e m tais que $g(n) \geq cf(n)$, para todo $n \geq m$.
- **Exemplo**: Para mostrar que $g(n) = 3n^3 + 2n^2$ é $\Omega(n^3)$ basta fazer $c = 1$, e então $3n^3 + 2n^2 \geq n^3$ para $n \geq 0$.
- **Exemplo**: Seja $g(n) = n$ para n ímpar $(n \geq 1)$ e $g(n) = n^2/10$ para n par $(n \geq 0)$.
  - Neste caso $g(n)$ é $\Omega(n^2)$, bastando considerar $c = 1/10$ e $n = 0, 2, 4, 6, \ldots$
- Exemplo gráfico para a notação Ω



- Para todos os valores à direita de m, o valor de $g(n)$ está sobre ou acima do valor de $cf(n)$.

Notação Θ

- **Definição**: Uma função $g(n)$ é $\Theta(f(n))$ se existirem constantes positivas c_1, c_2 e m tais que $0 \leq c_1 f(n) \leq g(n) \leq c_2 f(n)$, para todo $n \geq m$.
- Exemplo gráfico para a notação Θ



- Dizemos que $g(n) = \Theta(f(n))$ se existirem constantes c_1, c_2 e m tais que, para todo $n \geq m$, o valor de $g(n)$ está sobre ou acima de $c_1 f(n)$ e sobre ou abaixo de $c_2 f(n)$.
- Isto é, para todo $n \geq m$, a função $g(n)$ é igual a $f(n)$ a menos de uma constante.
- Neste caso, $f(n)$ é um **limite assintótico firme**.

Exemplo de Notação Θ

- Seja $g(n) = n^2/3 - 2n$.
- Vamos mostrar que $g(n) = \Theta(n^2)$.
- Temos de obter constantes c_1, c_2 e m tais que $c_1 n^2 \leq \frac{1}{3}n^2 - 2n \leq c_2 n^2$ para todo $n \geq m$.
- Dividindo por n^2 leva a $c_1 \leq \frac{1}{3} - \frac{2}{n} \leq c_2$.
- O lado direito da desigualdade será sempre válido para qualquer valor de $n \geq 1$ quando escolhemos $c_2 \geq 1/3$.
- Escolhendo $c_1 \leq 1/21$, o lado esquerdo da desigualdade será válido para qualquer valor de $n \geq 7$.
- Logo, escolhendo $c_1 = 1/21$, $c_2 = 1/3$ e $m = 7$, verifica-se que $n^2/3 - 2n = \Theta(n^2)$.
- Outras constantes podem existir, mas o importante é que existe alguma escolha para as três constantes.

Notação o

- Usada para definir um limite superior que não é assintoticamente firme.
- **Definição**: Uma função $g(n)$ é $o(f(n))$ se, para qualquer constante $c > 0$, então $0 \leq g(n) < cf(n)$ para todo $n \geq m$.
- **Exemplo**: $2n = o(n^2)$, mas $2n^2 \neq o(n^2)$.
- Em $g(n) = O(f(n))$, a expressão $0 \leq g(n) \leq cf(n)$ é válida para alguma constante $c > 0$, mas em $g(n) = o(f(n))$, a expressão $0 \leq g(n) < cf(n)$ é válida para todas as constantes $c > 0$.
- Na notação o, a função $g(n)$ tem um crescimento muito menor que $f(n)$ quando n tende para infinito.
- Alguns autores usam $\lim_{n\to\infty} \frac{g(n)}{f(n)} = 0$ para a definição da notação o.

Notação ω

- Por analogia, a notação ω está relacionada com a notação Ω da mesma forma que a notação o está relacionada com a notação O.
- **Definição**: Uma função $g(n)$ é $\omega(f(n))$ se, para qualquer constante $c > 0$, então $0 \leq cf(n) < g(n)$ para todo $n \geq m$.
- **Exemplo**: $\frac{n^2}{2} = \omega(n)$, mas $\frac{n^2}{2} \neq \omega(n^2)$.
- A relação $g(n) = \omega(f(n))$ implica $\lim_{n\to\infty} \frac{g(n)}{f(n)} = \infty$, se o limite existir.

Classes de Comportamento Assintótico

- Se f é uma **função de complexidade** para um algoritmo F, então $O(f)$ é considerada a **complexidade assintótica** ou o comportamento assintótico do algoritmo F.
- A relação de dominação assintótica permite comparar funções de complexidade.
- Entretanto, se as funções f e g dominam assintoticamente uma a outra, então os algoritmos associados são equivalentes.
- Nestes casos, o comportamento assintótico não serve para comparar os algoritmos.
- Por exemplo, considere dois algoritmos F e G aplicados à mesma classe de problemas, sendo que F leva três vezes o tempo de G ao serem executados, isto é, $f(n) = 3g(n)$, sendo que $O(f(n)) = O(g(n))$.
- Logo, o comportamento assintótico não serve para comparar os algoritmos F e G, porque eles diferem apenas por uma constante.

Comparação de Programas

- Podemos avaliar programas comparando as funções de complexidade, negligenciando as constantes de proporcionalidade.
- Um programa com tempo de execução $O(n)$ é melhor que outro com tempo $O(n^2)$.
- Porém, as constantes de proporcionalidade podem alterar esta consideração.
- Exemplo: um programa leva $100n$ unidades de tempo para ser executado e outro leva $2n^2$. Qual dos dois programas é melhor?
  - depende do tamanho do problema.
  - Para $n < 50$, o programa com tempo $2n^2$ é melhor do que o que possúi tempo $100n$.
  - Para problemas com entrada de dados pequena é preferível usar o programa cujo tempo de execução é $O(n^2)$.
  - Entretanto, quando n cresce, o programa com tempo de execução $O(n^2)$ leva muito mais tempo que o programa $O(n)$.

Principais Classes de Problemas

- $f(n) = O(1)$.
  - Algoritmos de complexidade $O(1)$ são ditos de **complexidade constante**.
  - Uso do algoritmo independe de n.
  - As instruções do algoritmo são executadas um número fixo de vezes.
- $f(n) = O(\log n)$.
  - Um algoritmo de complexidade $O(\log n)$ é dito de **complexidade logarítmica**.
  - Típico em algoritmos que transformam um problema em outros menores.
  - Pode-se considerar o tempo de execução como menor que uma constante grande.
  - Quando n é mil, $\log_2 n \approx 10$, quando n é 1 milhão, $\log_2 n \approx 20$.
  - Para dobrar o valor de $\log n$ temos de considerar o quadrado de n.
  - A base do logaritmo muda pouco estes valores: quando n é 1 milhão, o $\log_2 n$ é 20 e o $\log_{10} n$ é 6.

Principais Classes de Problemas

- $f(n) = O(n)$.
  - Um algoritmo de complexidade $O(n)$ é dito de **complexidade linear**.
  - Em geral, um pequeno trabalho é realizado sobre cada elemento de entrada.
  - É a melhor situação possível para um algoritmo que tem de processar/produzir n elementos de entrada/saída.
  - Cada vez que n dobra de tamanho, o tempo de execução também dobra.
- $f(n) = O(n \log n)$.
  - Típico em algoritmos que quebram um problema em outros menores, resolvem cada um deles independentemente e juntando as soluções depois.
  - Quando n é 1 milhão, $n\log_2 n$ é cerca de 20 milhões.
  - Quando n é 2 milhões, $n\log_2 n$ é cerca de 42 milhões, pouco mais do que o dobro.

Principais Classes de Problemas

- $f(n) = O(n^2)$.
  - Um algoritmo de complexidade $O(n^2)$ é dito de **complexidade quadrática**.
  - Ocorrem quando os itens de dados são processados aos pares, muitas vezes em um anel dentro de outro.
  - Quando n é mil, o número de operações é da ordem de 1 milhão.
  - Sempre que n dobra, o tempo de execução é multiplicado por 4.
  - Úteis para resolver problemas de tamanhos relativamente pequenos.
- $f(n) = O(n^3)$.
  - Um algoritmo de complexidade $O(n^3)$ é dito de **complexidade cúbica**.
  - Úteis apenas para resolver pequenos problemas.
  - Quando n é 100, o número de operações é da ordem de 1 milhão.
  - Sempre que n dobra, o tempo de execução fica multiplicado por 8.

Principais Classes de Problemas

- $f(n) = O(2^n)$.
  - Um algoritmo de complexidade $O(2^n)$ é dito de **complexidade exponencial**.
  - Geralmente não são úteis sob o ponto de vista prático.
  - Ocorrem na solução de problemas quando se usa **força bruta** para resolvê-los.
  - Quando n é 20, o tempo de execução é cerca de 1 milhão. Quando n dobra, o tempo fica elevado ao quadrado.
- $f(n) = O(n!)$.
  - Um algoritmo de complexidade $O(n!)$ é dito de complexidade exponencial, apesar de $O(n!)$ ter comportamento muito pior do que $O(2^n)$.
  - Geralmente ocorrem quando se usa **força bruta** para na solução do problema.
  - $n = 20 \rightarrow 20! = 2432902008176640000$, um número com 19 dígitos.
  - $n = 40 \rightarrow$ um número com 48 dígitos.

Comparação de Funções de Complexidade

Função de custo	Tamanho n					
	10	20	30	40	50	60
n	0,00001 s	0,00002 s	0,00003 s	0,00004 s	0,00005 s	0,00006 s
n^2	0,0001 s	0,0004 s	0,0009 s	0,0016 s	0,0.35 s	0,0036 s
n^3	0,001 s	0,008 s	0,027 s	0,64 s	0,125 s	0.316 s
n^5	0,1 s	3,2 s	24,3 s	1,7 min	5,2 min	13 min
2^n	0,001 s	1 s	17,9 min	12,7 dias	35,7 anos	366 séc.
3^n	0,059 s	58 min	6,5 anos	3855 séc.	10^8 séc.	10^{13} séc.

Função de custo de tempo	Computador atual	Computador 100 vezes mais rápido	Computador 1.000 vezes mais rápido
n	t_1	$100\ t_1$	$1000\ t_1$
n^2	t_2	$10\ t_2$	$31,6\ t_2$
n^3	t_3	$4,6\ t_3$	$10\ t_3$
2^n	t_4	$t_4 + 6,6$	$t_4 + 10$

Algoritmos Polinomiais

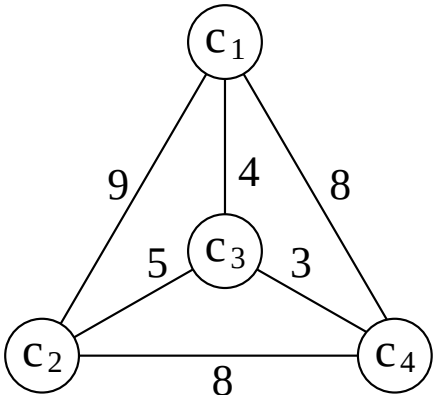
- **Algoritmo exponencial** no tempo de execução tem função de complexidade $O(c^n), c > 1$.
- **Algoritmo polinomial** no tempo de execução tem função de complexidade $O(p(n))$, onde $p(n)$ é um polinômio.
- A distinção entre estes dois tipos de algoritmos torna-se significativa quando o tamanho do problema a ser resolvido cresce.
- Por isso, os algoritmos polinomiais são muito mais úteis na prática do que os exponenciais.
- Algoritmos exponenciais são geralmente simples variações de pesquisa exaustiva.
- Algoritmos polinomiais são geralmente obtidos mediante entendimento mais profundo da estrutura do problema.
- Um problema é considerado:
  - intratável: se não existe um algoritmo polinomial para resolvê-lo.
  - bem resolvido: quando existe um algoritmo polinomial para resolvê-lo.

Algoritmos Polinomiais $\times$ Algoritmos Exponenciais

- A distinção entre algoritmos polinomiais eficientes e algoritmos exponenciais ineficientes possui várias exceções.
- Exemplo: um algoritmo com função de complexidade $f(n) = 2^n$ é mais rápido que um algoritmo $g(n) = n^5$ para valores de n menores ou iguais a 20.
- Também existem algoritmos exponenciais que são muito úteis na prática.
- Exemplo: o algoritmo Simplex para programação linear possui complexidade de tempo exponencial para o pior caso mas executa muito rápido na prática.
- Tais exemplos não ocorrem com freqüência na prática, e muitos algoritmos exponenciais conhecidos não são muito úteis.

Exemplo de Algoritmo Exponencial

- Um **caixeiro viajante** deseja visitar n cidades de tal forma que sua viagem inicie e termine em uma mesma cidade, e cada cidade deve ser visitada uma única vez.
- Supondo que sempre há uma estrada entre duas cidades quaisquer, o problema é encontrar a menor rota para a viagem.
- A figura ilustra o exemplo para quatro cidades c_1, c_2, c_3, c_4, em que os números nos arcos indicam a distância entre duas cidades.



- O percurso $< c_1, c_3, c_4, c_2, c_1 >$ é uma solução para o problema, cujo percurso total tem distância 24.

Exemplo de Algoritmo Exponencial

- Um algoritmo simples seria verificar todas as rotas e escolher a menor delas.
- Há $(n-1)!$ rotas possíveis e a distância total percorrida em cada rota envolve n adições, logo o número total de adições é $n!$.
- No exemplo anterior teríamos 24 adições.
- Suponha agora 50 cidades: o número de adições seria $50! \approx 10^{64}$.
- Em um computador que executa 10^9 adições por segundo, o tempo total para resolver o problema com 50 cidades seria maior do que 10^{45} séculos só para executar as adições.
- O problema do caixeiro viajante aparece com freqüência em problemas relacionados com transporte, mas também aplicações importantes relacionadas com otimização de caminho percorrido.

Técnicas de Análise de Algoritmos

- Determinar o tempo de execução de um programa pode ser um problema matemático complexo;
- Determinar a ordem do tempo de execução, sem preocupação com o valor da constante envolvida, pode ser uma tarefa mais simples.
- A análise utiliza técnicas de matemática discreta, envolvendo contagem ou enumeração dos elementos de um conjunto:
  - manipulação de somas,
  - produtos,
  - permutações,
  - fatoriais,
  - coeficientes binomiais,
  - solução de **equações de recorrência**.

Análise do Tempo de Execução

- Comando de atribuição, de leitura ou de escrita: $O(1)$.
- Seqüência de comandos: determinado pelo maior tempo de execução de qualquer comando da seqüência.
- Comando de decisão: tempo dos comandos dentro do comando condicional, mais tempo para avaliar a condição, que é $O(1)$.
- Anel: soma do tempo de execução do corpo do anel mais o tempo de avaliar a condição para terminação (geralmente $O(1)$), multiplicado pelo número de iterações.
- **Procedimentos não recursivos**: cada um deve ser computado separadamente um a um, iniciando com os que não chamam outros procedimentos. Avalia-se então os que são chamam os já avaliados (utilizando os tempos desses). O processo é repetido até chegar no programa principal.
- **Procedimentos recursivos**: associada uma função de complexidade $f(n)$ desconhecida, onde n mede o tamanho dos argumentos.

Procedimento não Recursivo

Algoritmo para ordenar os n elementos de um conjunto A em ordem ascendente.

```
package cap1;
public class Ordenacao {
  public static void ordena (int v[], int n) {
(1) for (int i = 0; i < n − 1; i++) {
(2)   int min = i;
(3)   for (int j = i + 1; j < n; j++)
(4)     if (v[j] < v[min])
(5)       min = j;
      /* Troca v[min] e v[i] */
(6)   int x = v[min];
(7)   v[min] = v[i];
(8)   v[i] = x;
    }
  }
}
```

- Seleciona o menor elemento do conjunto.
- Troca este com o primeiro elemento $v[0]$.
- Repita as duas operações acima com os $n-1$ elementos restantes, depois com os $n-2$, até que reste apenas um.

Análise do Procedimento não Recursivo

Anel Interno

- Contém um comando de decisão, com um comando apenas de atribuição. Ambos levam tempo constante para serem executados.
- Quanto ao corpo do comando de decisão, devemos considerar o pior caso, assumindo que serSS sempre executado.
- O tempo para incrementar o índice do anel e avaliar sua condição de terminação é $O(1)$.
- O tempo combinado para executar uma vez o anel é $O(max(1,1,1)) = O(1)$, conforme regra da soma para a notação O.
- Como o número de iterações é $n - i$, o tempo gasto no anel é $O((n-i) \times 1) = O(n-i)$, conforme regra do produto para a notação O.

Análise do Procedimento não Recursivo

Anel Externo

- Contém, além do anel interno, quatro comandos de atribuição.
  $O(max(1, (n-i), 1, 1, 1)) = O(n-i)$.
- A linha (1) é executada $n-1$ vezes, e o tempo total para executar o programa está limitado ao produto de uma constante pelo **somatório** de $(n-i)$:
  $\sum_1^{n-1}(n-i) = \frac{n(n-1)}{2} = \frac{n^2}{2} - \frac{n}{2} = O(n^2)$
- Considerarmos o número de comparações como a medida de custo relevante, o programa faz $(n^2)/2 - n/2$ comparações para ordenar n elementos.
- Considerarmos o número de trocas, o programa realiza exatamente $n-1$ trocas.

Procedimento Recursivo

```
    void pesquisa(n) {
(1)   if (n <= 1)
(2)     'inspecione elemento' e termine
      else {
(3)     para cada um dos n elementos 'inspecione elemento';
(4)     pesquisa(n/3);
      }
    }
```

- Para cada procedimento recursivo é associada uma função de complexidade $f(n)$ desconhecida, onde n mede o tamanho dos argumentos para o procedimento.
- Obtemos uma equação de recorrência para $f(n)$.
- **Equação de recorrência**: maneira de definir uma função por uma expressão envolvendo a mesma função.

Análise do Procedimento Recursivo

- Seja $T(n)$ uma função de complexidade que represente o número de inspeções nos n elementos do conjunto.
- O custo de execução das linhas (1) e (2) é $O(1)$ e o da linha (3) é exatamente n.
- Usa-se uma **equação de recorrência** para determinar o nº de chamadas recursivas.
- O termo $T(n)$ é especificado em função dos termos anteriores $T(1)$, $T(2)$, ..., $T(n-1)$.
- $T(n) = n + T(n/3), \;\; T(1) = 1$ (para $n = 1$ fazemos uma inspeção)
- Por exemplo, $T(3) = T(3/3) + 3 = 4$, $T(9) = T(9/3) + 9 = 13$, e assim por diante.
- Para calcular o valor da função seguindo a definição são necessários $k - 1$ passos para computar o valor de $T(3^k)$.

Exemplo de Resolução de Equação de Recorrência

- Sustitui-se os termos $T(k)$, $k < n$, até que todos os termos $T(k)$, $k > 1$, tenham sido substituídos por fórmulas contendo apenas $T(1)$.

$$\begin{aligned} T(n) &= n + T(n/3) \\ T(n/3) &= n/3 + T(n/3/3) \\ T(n/3/3) &= n/3/3 + T(n/3/3/3) \\ \vdots & \quad \vdots \\ T(n/3/3\cdots/3) &= n/3/3\cdots/3 + T(n/3\cdots/3) \end{aligned}$$

- Adicionando lado a lado, temos $T(n) = n + n \cdot (1/3) + n \cdot (1/3^2) + n \cdot (1/3^3) + \cdots + (n/3/3\cdots/3)$ que representa a soma de uma série geométrica de razão $1/3$, multiplicada por n, e adicionada de $T(n/3/3\cdots/3)$, que é menor ou igual a 1.

Exemplo de Resolução de Equação de Recorrência

$T(n) = n + n \cdot (1/3) + n \cdot (1/3^2) + n \cdot (1/3^3) + \cdots + + (n/3/3 \cdots /3)$

- Se desprezarmos o termo $T(n/3/3 \cdots /3)$, quando n tende para infinito, então $T(n) = n \sum_{i=0}^{\infty} (1/3)^i = n \left(\frac{1}{1-\frac{1}{3}} \right) = \frac{3n}{2}$.
- Se considerarmos o termo $T(n/3/3/3 \cdots /3)$ e denominarmos x o número de subdivisões por 3 do tamanho do problema, então $n/3^x = 1$, e $n = 3^x$. Logo $x = \log_3 n$.
- Lembrando que $T(1) = 1$ temos $T(n) = \sum_{i=0}^{x-1} \frac{n}{3^i} + T(\frac{n}{3^x}) = n \sum_{i=0}^{x-1} (1/3)^i + 1 = \frac{n(1-(\frac{1}{3})^x)}{(1-\frac{1}{3})} + 1 = \frac{3n}{2} - \frac{1}{2}$.
- Logo, o programa do exemplo é $O(n)$.

A Linguagem de Programação Java

- **Programação orientada a objetos:** nasceu porque algumas **linguagens procedimentais** se mostraram inadequadas para a construção de programas de grande porte.
- Existem dois tipos de problemas:
  1. **Falta de correspondência entre o programa e o mundo real:** Os procedimentos implementam tarefas e estruturas de dados armazenam informação, mas a maioria dos objetos do mundo real contém as duas coisas.
  2. **Organização interna dos programas:** Não existe uma maneira flexível para dizer que determinados procedimentos poderiam acessar uma variável enquanto outros não.

A Linguagem de Programação Java

- **Programação orientada a objetos:** permite que objetos do mundo real que compartilham propriedades e comportamentos comuns sejam agrupados em classes.
- Estilo de programação diretamente suportado pelo conceito de classe em Java.
- Pode-se também impor restrições de visibilidade aos dados de um programa.
- Classes e objetos são os conceitos fundamentais nas linguagens orientadas a objeto.
- A linguagem Java possui um grau de orientação a objetos maior do que a linguagem C++.
- Java não é totalmente orientada a objetos como a linguagem **S**malltalk.
- Java não é totalmente orientada a objetos porque, por questões de eficiência, foram mantidos alguns tipos primitivos e suas operações.

Principais Componentes de um Programa Java

- Em Java, as funções e os procedimentos são chamados de **métodos.**
- Um **objeto** contém métodos e variáveis que representam seus campos de dados (atributos).
  - Ex: um objeto $painelDeControle$ deveria conter não somente os métodos $ligaForno$ e $desligaForno$, mas também as variáveis $temperaturaCorrente$ e $temperaturaDese\text{-}jada$.
- O conceito de objeto resolve bem os problemas apontados anteriormente.
  - Os métodos $ligaForno$ e $desligaForno$ podem acessar as variáveis $temperaturaCorrente$ e $temperaturaDesejada$, mas elas ficam escondidas de outros métodos que não fazem parte do objeto $painelDeControle$.

Principais Componentes de um Programa Java

- O conceito de classe nasceu da necessidade de se criar diversos objetos de um mesmo tipo.
- Dizemos que um objeto pertence a uma classe ou, mais comumente, que é uma **i**nstância

```
package cap1;
class PainelDeControle {
   private float temperaturaCorrente;
   private float temperaturaDesejada;

   public void ligaForno () {
      // código do método
   }
   public void desligaForno() {
      // código do método
   }
}
```

- A palavra chave **class** introduz a classe $PainelDeControle$.
- A palavra chave **void** é utilizada para indicar que os métodos não retornam nenhum valor.

Principais Componentes de um Programa Java

- Um objeto em Java é criado usando a palavra chave **n**ew
- É necessário armazenar uma referência para ele em uma variável do mesmo tipo da classe, como abaixo:

  PainelDeControle painel1, painel2;

- Posteriormente, cria-se os objetos, como a seguir:

  painel1 = **n**ew PainelDeControle ();
  painel2 = **n**ew PainelDeControle ();

- Outras partes do programa interagem com os métodos dos objetos por meio do operador (**.**), o qual associa um objeto com um de seus métodos, como a seguir:

  painel1.ligaForno ();

Herança e Polimorfismo

- **Herança:** criação de uma classe a partir de uma outra classe.
- A classe é estendida a partir da classe base usando a palavra chave **extends**.
- A classe estendida (**subclasse**) tem todas as características da classe base (**superclasse**) mais alguma característica adicional.
- **Polimorfismo:** tratamento de objetos de classes diferentes de uma mesma forma.
- As classes diferentes devem ser derivadas da mesma classe base.

Herança e Polimorfismo

```
package cap1;
class Empregado {
  protected float salario;
  public float salarioMensal () { return salario; }
  public void imprime () { System.out.println ("Empregado"); }
}
class Secretaria extends Empregado {
  private int velocidadeDeDigitacao;
  public void imprime () { System.out.println ("Secretaria");}
}
class Gerente extends Empregado {
  private float bonus;
  public float salarioMensal () { return salario + bonus; }
  public void imprime () { System.out.println ("Gerente"); }
}
public class Polimorfismo {
  public static void main (String[] args) {
    Empregado empregado = new Empregado ();
    Empregado secretaria = new Secretaria ();
    Empregado gerente = new Gerente ();
    empregado.imprime (); secretaria.imprime ();
    gerente.imprime ();
  }
}
```

Objetos e Tipos Genéricos

- Uma estrutura de dados é genérica quando o tipo dos dados armazenados na estrutura é definido na aplicação que a utiliza (**objetos genéricos**).
- Um **objeto genérico** pode armazenar uma referência para um objeto de qualquer classe (classe **Object** em Java).
- Os mecanismos de **herança** e **polimorfismo** que permitem a implementação de estruturas de dados genéricas.

```
package cap1.objetogenerico;
public class Lista {
  private static class Celula {
    Object item; Celula prox;
  }
  private Celula primeiro, ultimo;
}
```

- O objeto $item$ é definido como um objeto genérico, assim $Lista$ pode ter objetos de classes distintas em cada item

Objetos e Tipos Genéricos

- Para evitar que se declare o tipo de cada objeto a ser inserido ou retirado da lista, a **Versão 5** da linguagem Java introduziu um mecanismo de definição de um tipo genérico.
- **Tipo genérico:** definição de um parâmetro de tipo que deve ser especificado na aplicação que utiliza a estrutura de dados:

```
package cap1.tipogenerico;
public class Lista<T> {
  private static class Celula<T> {
    T item;
    Celula<T> prox;
  }
  private Celula<T> primeiro, ultimo;
}
```

- O objeto $item$ tem de ser uma instância de um tipo genérico T que será fornecido quando um objeto da classe $Lista$ for instanciado.
  - Para instanciar uma lista de inteiros basta declarar o comando “Lista<Integer> lista = **n**ew Lista<Integer>();”.

Sobrecarga

- A **sobrecarga** acontece quando determinado objeto se comporta de diferentes formas.
- É um tipo de **p**olimorfismo *ad hoc*, no qual um identificador representa vários métodos com computações distintas.

```
public float salarioMensal (float desconto) {
  return salario + bonus − desconto;
}
```

- O programa acima apresenta um exemplo de sobrecarga do método $salarioMensal$ da classe $Gerente$ mostrada em um programa anterior, em que um $desconto$ é subtraído de $salario + bonus$.
- Note que o método $salarioMensal$ do programa acima possui uma assinatura diferente da assinatura apresentada no programa anterior.

Sobrescrita

- A ocultação de um método de uma classe mais genérica em uma classe mais específica é chamada de **s**obrescrita
- Por exemplo, o método $imprime$ da classe $Empregado$ apresentada nas parte de Herança e Polimorfismo, foi sobrescrito nas classes $Gerente$ e $Secretaria$.
- Para sobrescrever um método em uma subclasse é preciso que ele tenha a mesma assinatura na superclasse.

Programa Principal

```
package cap1;
class ContaBancaria {
  private double saldo;
  public ContaBancaria (double saldoInicial) {
    saldo = saldoInicial;
  }
  public void deposito (double valor) {
    saldo = saldo + valor;
  }
  public void saque (double valor) {
    saldo = saldo – valor;
  }
  public void imprime () {
    System.out.println ("saldo=" + saldo);
  }
}
public class AplicacaoBancaria {
  public static void main (String[] args) {
    ContaBancaria conta1 = new ContaBancaria (200.00);
    System.out.print ("Antes da movimentacao, ");
    conta1.imprime ();
    conta1.deposito (50.00); conta1.saque (70.00);
    System.out.print ("Depois da movimentacao, ");
    conta1.imprime ();
  }
}
```

Programa Principal

- Programa anterior modela uma conta bancária típica com as operações: cria uma conta com um saldo inicial; imprime o saldo; realiza um depósito; realiza um saque e imprime o novo saldo;
- A classe $Contabancaria$ tem um campo de dados chamado $saldo$ e três métodos chamados $deposito$, $saque$ e $imprime$.
- Para **c**ompilar o Programa acima a partir de uma linha de comando em MS-DOS ou Linux, fazemos:

  javac -d **./** $AplicacaoBancaria.java$

  e para executá-lo, fazemos:

  java $cap1.AplicacaoBancaria$
- A classe $ContaBancaria$ tem um método especial denominado **construtor**, que é chamado automaticamente sempre que um novo objeto é criado com o comando **new** e tem sempre o mesmo nome da classe.

Modificadores de Acesso

- **Modificadores de acesso:** determinam quais outros métodos podem acessar um campo de dados ou um método.
- Um campo de dados ou um método que seja precedido pelo modificador **private** pode ser acessado somente por métodos que fazem parte da mesma classe.
- Um campo de dados ou um método que seja precedido pelo modificador **public** pode ser acessado por métodos de outras classes.
  - Classe modificada com o modificador **public** indica que a classe é visível externamente ao pacote em que ela foi definida (classe $AplicacaoBancaria$, **package** cap1).
  - Em cada arquivo de um programa Java só pode existir uma classe modificada por **p**ublic, e o nome do arquivo deve ser o mesmo dado à classe.
- Os campos de dados de uma classe são geralmente feitos **private** e os métodos são tornados **public**.

Modificadores de Acesso

- Modificador protected: utilizado para permitir que somente subclasses de uma classe mais genérica possam acessar os campos de dados precedidos com **protected**.
- Um campo de dados ou um método de uma classe declarado como **static** pertence à classe e não às suas instâncias, ou seja, somente um campo de dados ou um método será criado pelo compilador para todas as instâncias.
- Os métodos de uma classe que foram declarados **static** operam somente sobre os campos da classe que também foram declarados **static**.
- Se além de **static** o método for declarado **public** será possível acessá-lo com o nome da classe e o operador (**.**).

Modificadores de Acesso

```
package cap1;
class A {
  public static int total;
  public int media;
}
public class B {
  public static void main (String[] args) {
    A a = new A(); a.total = 5; a.media = 5;
    A b = new A(); b.total = 7; b.media = 7;
  }
}
```

- No exemplo acima, o campo de dados $total$ pertence somente à classe A, enquanto o campo de dados $media$ pertence a todas as instâncias da classe A.
- Ao final da execução do método $main$, os valores de $a.total$ e $b.total$ são iguais a 7, enquanto os valores de $a.media$ e $b.media$ são iguais a 5 e 7, respectivamente.

Interfaces

- Uma **i**nterface em Java é uma **classe abstrata** que não pode ser instanciada, cujos os métodos devem ser **p**ublic e somente suas assinaturas são definidas
- Uma **i**nterface é sempre implementada por outras classes.
- Utilizada para prover a especificação de um comportamento que seja comum a um conjunto de objetos.

```
package cap1;
import java.io.*;
public class Max {
  public static Item max (Item v[], int n) {
    Item max = v[0];
    for (int i = 1; i < n; i++)
        if (max.compara (v[i]) < 0) max = v[i];
    return max;
  }
}
```

- O programa acima apresenta uma versão generalizada do programa para obter o máximo de um conjunto de inteiros.

Interfaces

- Para permitir a generalização do tipo de dados da chave é necessário criar a interface $Item$ que apresenta a assinatura do método abstrato $compara$.

```
package cap1;
public interface Item {
  public int compara (Item it);
}
```

- A classe $MeuItem$, o tipo de dados da chave é definido e o método $compara$ é implementado.

```
package cap1;
import java.io.*;
public class MeuItem implements Item {
  public int chave;
  // outros componentes do registro

  public MeuItem (int chave) { this.chave = chave; }
  public int compara (Item it) {
    MeuItem item = (MeuItem) it;
    if (this.chave < item.chave) return -1;
    else if (this.chave > item.chave) return 1;
    return 0;
  }
}
```

Interfaces

```
package cap1;
public class EncontraMax {
   public static void main (String[] args) {
      MeuItem v[] = new MeuItem[2];
      v[0] = new MeuItem (3); v[1] = new MeuItem (10);
      MeuItem max = (MeuItem) Max.max (v, 2);
      System.out.println ("Maior chave: " + max.chave);
   }
}
```

- O programa acima ilustra a utilização do método $compara$ apresentado.
- Note que para atribuir a um objeto da classe $MeuItem$ o valor máximo retornado pelo método max é necessário fazer uma conversão do tipo $Item$ para o tipo $MeuItem$, conforme ilustra a penúltima linha do método $main$.

Pacotes

- A linguagem Java permite agrupar as classes e as interfaces em pacotes(do inglês, **p**ackage.
- Convenientes para organizar e separar as classes de um conjunto de programas de outras bibliotecas de classes, evitando colisões entre nomes de classes desenvolvidas por uma equipe composta por muitos programadores.
- Deve ser realizada sempre na primeira linha do arquivo fonte, da seguinte forma por exemplo:

  package cap1;

- É possível definir subpacotes separados por ".", por exemplo, para definir o subpacote $arranjo$ do pacote $cap3$ fazemos:

  package cap3.arranjo;

- A utilização de uma classe definida em outro pacote é realizada através da palavra chave **import**. O comando abaixo possibilita a utilização de todas as classes de um pacote:

  import cap3.arranjo.*;

Pacotes

- É possível utilizar determinada classe de um pacote sem importá-la, para isso basta prefixar o nome da classe com o nome do pacote durante a declaração de uma variável. Exemplo:

  cap3.arranjo.Lista lista;

- Para que uma classe possa ser importada em um pacote diferente do que ela foi definida é preciso declará-la como pública por meio do modificador **public**.

- Se o comando **package** não é colocado no código fonte, então Java adiciona as classes daquele código fonte no que é chamado de pacote *default*

- Quando o modificador de um campo ou método não é estabelecido, diz-se que o campo ou método possui visibilidade *default*, ou seja, qualquer objeto de uma classe do pacote pode acessar diretamente aquele campo (ou método).

Classes Internas

- Java permite realizar aninhamento de classes como abaixo:

```
package cap1;
public class Lista {
   // Código da classe Lista
   private class Celula {
      // Código da classe Celula
   }
}
```

- Classes internas são muito úteis para evitar conflitos de nomes.
- Os campos e métodos declarados na classe externa podem ser diretamente acessados dentro da classe interna, mesmo os declarados como **protected** ou **private**, mas o contrário não é verdadeiro.
- As classes externas só podem ser declaradas como públicas ou com visibilidade *default*.
- As classes internas podem também ser qualificadas com os modificadores **p**rivate, **p**rotected e **s**tatic e o efeito é mesmo obtido sobre qualquer atributo da classe externa.

O Objeto *this*

- Toda instância de uma classe possui uma variável especial chamada **t**his, que contém uma referência para a própria instância.
- Em algumas situações resolve questões de ambigüidade.

```
package cap1;
public class Conta {
   private double saldo;
   public void alteraSaldo (double saldo) {
      this.saldo = saldo;
   }
}
```

- No exemplo acima, o parâmetro $saldo$ do método $alteraSaldo$ possui o mesmo nome do campo de instância $saldo$ da classe $Conta$.
- Para diferenciá-los é necessário qualificar o campo da instância com o objeto **t**his.

Exceções

- As exceções são erros ou anomalias que podem ocorrer durante a execução de um programa.
- Deve ser obrigatoriamente representada por um objeto de uma subclasse da classe ***Throwable***, que possui duas subclasses diretas: (i) ***Exception*** e (ii) ***Error***
- Uma abordagem simples para tratar uma exceção é exibir uma mensagem relatando o erro ocorrido e retornar para quem chamou ou finalizar o programa, como no exemplo abaixo:

```
int divisao (int a, int b) {
  try {
     if (b == 0) throw new Exception ("Divisao por zero");
     return (a/b);
  }
  catch (Exception objeto) {
     System.out.println ("Erro:" + objeto.getMessage());
     return (0);
  }
}
```

Exceções

- O comando **try** trata uma exceção que tenha sido disparada em seu interior por um comando **throw**
- O comando **throw** instancia o objeto que representa a exceção e o envia para ser capturado pelo trecho de código que vai tratar a exceção.
- O comando **catch** captura a exceção e fornece o tratamento adequado.
- Uma abordagem mais elaborada para tratar uma exceção é separar o local onde a exceção é tratada do local onde ela ocorreu.
- Importante pelo fato de que um trecho de código em um nível mais alto pode possuir mais informação para decidir como melhor tratar a exceção.

Exceções

- No exemplo abaixo a exceção não é tratada no local onde ela ocorreu, e esse fato é explicitamente indicado pelo comando **t**hrows

```
int divisao (int a, int b) throws {
  if (b == 0) throw new Exception ("Divisao por zero");
  return (a/b);
}
```

- Considerando que o método $divisao$ está inserido em uma classe chamada $Divisao$, o trecho de código abaixo ilustra como capturar o objeto exceção que pode ser criado no método:

```
Divisao d = new Divisao ();
try {
  d.divisao (3, 0);
}
catch(Exception objeto) {
  System.out.println("Erro:"+objeto.getMessage());
}
```

Saída de Dados

- Os tipos primitivos e objetos do tipo $String$ podem ser impressos com os comandos

  System.out.print (var);
  System.out.println (var);

- O método $print$ deixa o cursor na mesma linha e o método $println$ move o cursor para a próxima linha de saída.

Entrada de Dados

- Todo programa em Java que tenha leitura de dados tem de incluir o comando no início do programa

```
import java.io.*;
```

- Método para ler do teclado uma cadeia de caracteres terminada com a tecla *Enter*:

```
public static String getString () throws
IOException {
   InputStreamReader inputString = new
InputStreamReader (System.in);
   BufferedReader buffer = new BufferedReader
(inputString);
   String s = buffer.readLine (); return s;
}
```

- Método para realizar a entrada de um caractere a partir do teclado:

```
public static char getChar () throws IOException {
  String s = getString ();
  return s.charAt (0);
}
```

- Se o que está sendo lido é de outro tipo, então é necessário realizar uma conversão.

Diferenças entre Java e C++

- A maior diferença entre Java e C++ é a ausência de apontadores em Java(não utiliza apontadores explicitamente).
- Java trata tipos de dados primitivos, tais como **int**, **double** e **float**, de forma diferente do tramento dado a objetos.
- Em Java, uma referência pode ser vista como um apontador com a sintaxe de uma variável.
- A linguagem C++ tem variáveis referência, mas elas têm de ser especificadas de forma explícita com o símbolo &.
- Outra diferença significativa está relacionada com o operador de atribuição (=):
  - C++: após a execução de um comando com operador (=), passam a existir dois objetos com os mesmos dados estáticos.
  - Java: após a execução de um comando com operador (=), passam a existir duas variáveis que se referem ao mesmo objeto.

Diferenças entre Java e C++

- Em Java e em C++ os objetos são criados utilizando o operador **new**, entretanto, em Java o valor retornado é uma referência ao objeto criado, enquanto em C++ o valor retornado é um apontador para o objeto criado.
- A eliminação de apontadores em Java tem por objetivo tornar o *software* mais seguro, uma vez que não é possível manipular o endereço de $conta1$, evitando que alguém possa acidentalmente corromper o endereço.
- Em C++, a memória alocada pelo operador **new** tem de ser liberada pelo programador quando não é mais necessária, utilizando o operador **d**elete.
- Em Java, a liberação de memória é realizada pelo sistema de forma transparente para o programador (**coleta de lixo**, do inglês *garbage collection*).

Diferenças entre Java e C++

- Em Java, os objetos são passados para métodos como referências aos objetos criados, entretanto, os tipos primitivos de dados em Java são sempre passados por valor
- Em C++ uma passagem por referência deve ser especificada utilizando-se o &, caso contrário, temos uma passagem por valor.
- No caso de tipos primitivos de dados, tanto em Java quanto em C++ o operador de igualdade (==) diz se duas variáveis são iguais.
- No caso de objetos, em C++ o operador diz se dois objetos contêm o mesmo valor e em Java o operador de igualdade diz se duas referências são iguais, isto é, se apontam para o mesmo objeto.
- Em Java, para verificar se dois objetos diferentes contêm o mesmo valor é necessário utilizar o método $equals$ da classe $Object$ (O programador deve realizar a sobrescrita desse método para estabelecer a relação de igualdade).

Diferenças entre Java e C++

- Em C++ é possível redefinir operadores como $+$, $-$, $*$, $=$, de tal forma que eles se comportem de maneira diferente para os objetos de uma classe particular, mas em Java, não existe sobrecarga de operadores.
- Por questões de eficiência foram mantidos diversos tipos primitivos de dados, assim variáveis declaradas como um tipo primitivo em Java permitem acesso direto ao seu valor, exatamente como ocorre em C++.
- Em Java, o tipo **boolean** pode assumir os valores **false** ou **true** enquanto em C++ os valores inteiros 0 e 1
- O tipo **byte**não existe em C++.
- O tipo **char** em Java é sem sinal e usa dois bytes para acomodar a representação
- O tipo **Unicode** de caracteres acomoda caracteres internacionais de linguas tais como chinês e japonês.

Diferenças entre Java e C++

- O tipo **short** tem tratamento parecido em Java e C++.
- Em Java, o tipo **int** tem sempre 32 *bits*, enquanto em C++ de tamanho, dependendo de cada arquitetura do computador onde vai ser executado.
- Em Java, o tipo **float** usa o sufixo F (por exemplo, 2.357F) enquanto o tipo **d**ouble não necessita de sufixo.
- Em Java, o tipo **long** usa o sufixo L (por exemplo, 33L); quaisquer outros tipos inteiros não necessitam de sufixo.

Paradigmas de Projeto de Algoritmos*

Última alteração: 10 de Outubro de 2006

*Transparências elaboradas por Charles Ornelas, Leonardo Rocha, Leonardo Mata e Nivio Ziviani

Paradigmas de Projeto de Algoritmos

- indução,
- recursividade,
- algoritmos tentativa e erro,
- divisão e conquista,
- balanceamento,
- programação dinâmica,
- algoritmos gulosos,
- algoritmos aproximados.

Indução Matemática

- É útil para provar asserções sobre a correção e a eficiência de algoritmos.
- Consiste em inferir uma lei geral a partir de instâncias particulares.
- Seja T um teorema que que tenha como parâmetro um número natural n. Para provar que T é válido para todos os valores de n, basta provarmos que:
  1. T é válido para $n = 1$;
  2. Para todo $n > 1$, se T é válido para $n - 1$, então T é válido para n.
- A condição 1 é chamada de **passo base**.
- Provar a condição 2 é geralmente mais fácil que provar o teorema diretamente, uma vez que podemos usar a asserção de que T é válido para $n - 1$.
- Esta afirmativa é chamada de **hipótese de indução** ou **passo indutivo**
- As condições 1 e 2 implicam T válido para $n = 2$, o que junto com a condição 2 implica T também válido para $n = 3$, e assim por diante.

Exemplo de Indução Matemática

$S(n) = 1 + 2 + \cdots + n = n(n+1)/2$

- Para $n = 1$ a asserção é verdadeira, pois $S(1) = 1 = 1 \times (1+1)/2$ (passo base).
- Assumimos que a soma dos primeiros n números naturais $S(n)$ é $n(n+1)/2$ (hipótese de indução).
- Pela definição de $S(n)$ sabemos que $S(n+1) = S(n) + n + 1$.
- Usando a hipótese de indução, $S(n+1) = n(n+1)/2 + n + 1 = (n+1)(n+2)/2$, que é exatamente o que queremos provar.

Limite Superior de Equações de Recorrência

- A solução de uma equação de recorrência pode ser difícil de ser obtida.
- Nesses casos, pode ser mais fácil tentar adivinhar a solução ou chegar a um limite superior para a ordem de complexidade.
- Adivinhar a solução funciona bem quando estamos interessados apenas em um limite superior, ao invés da solução exata.
- Mostrar que um limite existe é mais fácil do que obter o limite.
- Ex.: $T(2n) \leq 2T(n) + 2n - 1, \;\; T(2) = 1$, definida para valores de n que são potências de 2.
  - O objetivo é encontrar um limite superior na notação O, onde o lado direito da desigualdade representa o pior caso.

Indução Matemática para Resolver Equação de Recorrência

$T(2n) \leq 2T(n) + 2n - 1, \;\; T(2) = 1$, definida para valores de n que são potências de 2.

- Procuramos $f(n)$ tal que $T(n) = O(f(n))$, mas fazendo com que $f(n)$ seja o mais próximo possível da solução real para $T(n)$.
- Vamos considerar o palpite $f(n) = n^2$.
- Queremos provar que $T(n) = O(f(n))$ utilizando indução matemática em n.
- Passo base: $T(2) = 1 \leq f(2) = 4$.
- Passo de indução: provar que $T(n) \leq f(n)$ implica $T(2n) \leq f(2n)$.

$$\begin{aligned} T(2n) &\leq 2T(n) + 2n - 1, && \text{(def. da recorrência)} \\ &\leq 2n^2 + 2n - 1, && \text{(hipótese de indução)} \\ &< (2n)^2, \end{aligned}$$

que é exatamente o que queremos provar. Logo, $T(n) = O(n^2)$.

Indução Matemática para Resolver Equação de Recorrência

- Vamos tentar um palpite menor, $f(n) = cn$, para alguma constante c.
- Queremos provar que $T(n) \leq cn$ implica em $T(2n) \leq c2n$. Assim:

$$\begin{aligned} T(2n) &\leq 2T(n) + 2n - 1, && \text{(def. da recorrência)} \\ &\leq 2cn + 2n - 1, && \text{(hipótese de indução)} \\ &> c2n. \end{aligned}$$

- cn cresce mais lentamente que $T(n)$, pois $c2n = 2cn$ e não existe espaço para o valor $2n - 1$.
- Logo, $T(n)$ está entre cn e n^2.

Indução Matemática para Resolver Equação de Recorrência

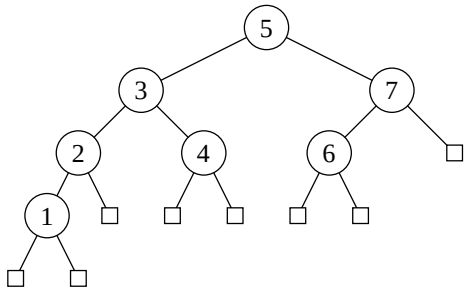
- Vamos então tentar $f(n) = n \log n$.
- Passo base: $T(2) < 2 \log 2$.
- Passo de indução: vamos assumir que $T(n) \leq n \log n$.
- Queremos mostrar que $T(2n) \leq 2n \log 2n$. Assim:

$$\begin{aligned} T(2n) &\leq 2T(n) + 2n - 1, \quad \text{(def. da recorrência)} \\ &\leq 2n \log n + 2n - 1, \ \text{(hipótese de indução)} \\ &< 2n \log 2n, \end{aligned}$$

- A diferença entre as fórmulas agora é de apenas 1.
- De fato, $T(n) = n \log n - n + 1$ é a solução exata de $T(n) = 2T(n/2) + n - 1, \ \ T(1) = 0$, que descreve o comportamento do algoritmo de ordenação *Mergesort*.

Recursividade

- Um método que chama a si mesmo, direta ou indiretamente, é dito **recursivo**.
- Recursividade permite descrever algoritmos de forma mais clara e concisa, especialmente problemas recursivos por natureza ou que utilizam estruturas recursivas.
- Ex.: árvore binária de pesquisa:
  - Todos os registros com chaves menores estão na subárvore esquerda;
  - Todos os registros com chaves maiores estão na subárvore direita.




```
package cap2;
public class ArvoreBinaria {
   private static class No {
      Object reg;
      No esq, dir;
   }
   private No raiz;
}
```

Recursividade

- Algoritmo para percorrer todos os registros em ordem de **caminhamento central**:
  1. caminha na subárvore esquerda na ordem central;
  2. visita a raiz;
  3. caminha na subárvore direita na ordem central.
- No caminhamento central, os nós são visitados em ordem lexicográfica das chaves.

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

Implementação de Recursividade

- Usa-se uma **pilha** para armazenar os dados usados em cada chamada de um procedimento que ainda não terminou.
- Todos os dados não globais vão para a pilha, registrando o estado corrente da computação.
- Quando uma ativação anterior prossegue, os dados da pilha são recuperados.
- No caso do caminhamento central:
  - para cada chamada recursiva, o valor de p e o endereço de retorno da chamada recursiva são armazenados na pilha.
  - Quando encontra p$=$null o procedimento retorna para quem chamou utilizando o endereço de retorno que está no topo da pilha.

Problema de Terminação em Procedimentos Recursivos

- Procedimentos recursivos introduzem a possibilidade de iterações que podem não terminar: existe a necessidade de considerar o problema de **terminação**.
- É fundamental que a chamada recursiva a um procedimento P esteja sujeita a uma condição B, a qual se torna não-satisfeita em algum momento da computação.
- Esquema para procedimentos recursivos: composição $\mathcal{C}$ de comandos S_i e P.
  $P \equiv$ **if** B **then** $\mathcal{C}[S_i, P]$
- Para demonstrar que uma repetição termina, define-se uma função $f(x)$, sendo x o conjunto de variáveis do programa, tal que:
  1. $f(x) \leq 0$ implica na condição de terminação;
  2. $f(x)$ é decrementada a cada iteração.

Problema de Terminação em Procedimentos Recursivos

- Uma forma simples de garantir terminação é associar um parâmetro n para P (no caso **por valor**) e chamar P recursivamente com $n-1$.
- A substituição da condição B por $n > 0$ garante terminação.
  $P \equiv$ **if** $n > 0$ **then** $\mathcal{P}[S_i, P(n-1)]$
- É necessário mostrar que o nível mais profundo de recursão é finito, e também possa ser mantido pequeno, pois cada ativação recursiva usa uma parcela de memória para acomodar as variáveis.

Quando Não Usar Recursividade

- Nem todo problema de natureza recursiva deve ser resolvido com um algoritmo recursivo.
- Estes podem ser caracterizados pelo esquema $P \equiv$ **if** B **then** (S, P)
- Tais programas são facilmente transformáveis em uma versão não recursiva
  $P \equiv (x := x_0;$ **while** B **do** $S)$

Exemplo de Quando Não Usar Recursividade

- Cálculo dos **números de Fibonacci**
  $f_0 = 0, f_1 = 1,$
  $f_n = f_{n-1} + f_{n-2}$para$n \geq 2$
- Solução: $f_n = \frac{1}{\sqrt{5}}[\Phi^n - (-\Phi)^{-n}]$, onde $\Phi = (1 + \sqrt{5})/2 \approx 1,618$ é a **razão de ouro**.
- O procedimento recursivo obtido diretamente da equação é o seguinte:

```
package cap2;
public class Fibonacci {
  public static int fibRec (int n) {
    if (n < 2) return n;
    else return (fibRec (n−1) + fibRec (n−2));
  }
}
```

- O programa é extremamente ineficiente porque recalcula o mesmo valor várias vezes.
- Neste caso, a complexidade de espaço para calcular f_n é $O(\Phi^n)$.
- Considerando que a complexidade de tempo $f(n)$ é o número de adições, $f(n) = O(\Phi^n)$.

Versão iterativa do Cálculo de Fibonacci

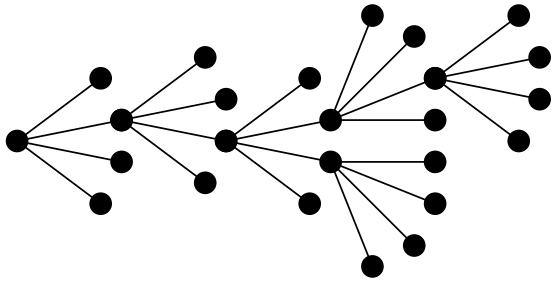
```
package cap2;
public class Fibonacci {
  public static int fibIter (int n) {
    int i = 1, f = 0;
    for (int k = 1; k <= n; k++) {
      f = i + f;
      i = f – i;
    }
    return f;
  }
}
```

- O programa tem complexidade de tempo $O(n)$ e complexidade de espaço $O(1)$.
- Devemos evitar uso de recursividade quando existe uma solução óbvia por iteração.
- Comparação versões recursiva e iterativa:

n	10	20	30	50	100
$fibRec$	8 ms	1 s	2 min	21 dias	10^9 anos
$fibIter$	1/6 ms	1/3 ms	1/2 ms	3/4 ms	1,5 ms

Algoritmos Tentativa e Erro (*Backtracking*)

- **Tentativa e erro**: decompor o processo em um número finito de subtarefas parciais que devem ser exploradas exaustivamente.
- O processo de tentativa gradualmente constrói e percorre uma árvore de subtarefas.



- Algoritmos tentativa e erro não seguem regra fixa de computação:
  - Passos em direção à solução final são tentados e registrados;
  - Caso esses passos tomados não levem à solução final, eles podem ser retirados e apagados do registro.
- Quando a pesquisa na árvore de soluções cresce rapidamente é necessário usar **algoritmos aproximados** ou **heurísticas** que não garantem a solução ótima mas são rápidos.

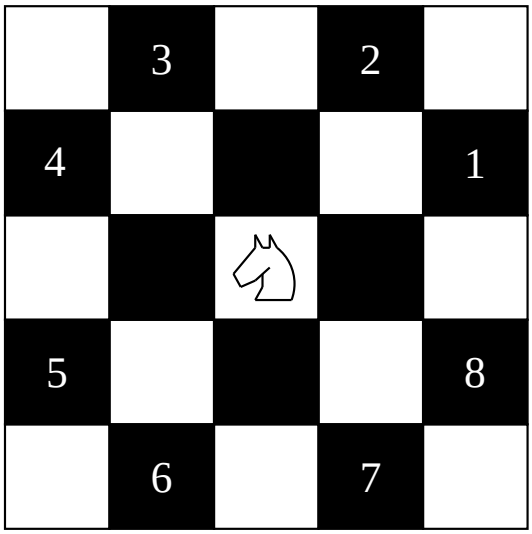
Backtracking: Passeio do Cavalo

- Tabuleiro com $n \times n$ posições: cavalo movimenta-se segundo as regras do xadrez.
- Problema: a partir de (x_0, y_0), encontrar, se existir, um passeio do cavalo que visita todos os pontos do tabuleiro uma única vez.
- Tenta um próximo movimento:

```
void tenta () {
  inicializa seleção de movimentos;
  do {
    seleciona próximo candidato ao movimento;
    if (aceitável) {
        registra movimento;
        if (tabuleiro não está cheio) {
            tenta novo movimento; // Chamada recursiva para tenta
            if (não sucedido) apaga registro anterior;
        }
    }
  } while (movimento não sucedido e não acabaram candidatos a movimento);
}
```

Exemplo de Backtracking - Passeio do Cavalo

- O tabuleiro pode ser representado por uma matriz $n \times n$.
- A situação de cada posição pode ser representada por um inteiro para recordar o histórico das ocupações:
  - t[x,y] = 0, campo $< x, y >$ não visitado,
  - t[x,y] = i, campo $< x, y >$ visitado no i-ésimo movimento, $1 \leq i \leq n^2$.
- Regras do xadrez para os movimentos do cavalo:



Implementação do Passeio do Cavalo

```
package cap2;
public class PasseioCavalo {
   private int n; // Tamanho do lado do tabuleiro
   private int a[], b[], t[][];

   public PasseioCavalo (int n) {
      this.n = n;
      this.t = new int[n][n];   this.a = new int[n];
      this.b = new int[n];
      a[0] = 2;  a[1] = 1;  a[2] =-1; a[3] =-2;
      b[0] = 1;  b[1] = 2;  b[2] = 2; b[3] = 1;
      a[4] = -2; a[5] = -1; a[6] = 1; a[7] = 2;
      b[4] = -1; b[5] = -2; b[6] =-2; b[7] = -1;
      for (int i = 0; i < n; i++)
          for (int j = 0; j < n; j++) t[i][j] = 0;
      t[0][0] = 1; // escolhemos uma casa do tabuleiro
   }
// {-- Entra aqui os métodos tenta, imprimePasseio e main mostrados a seguir --}
   }
```

Implementação do Passeio do Cavalo

```
public boolean tenta (int i, int x, int y) {
  int u, v, k; boolean q;
  k = −1;  // inicializa seleção de movimentos
  do {
    k = k + 1; q = false;
    u = x + a[k]; v = y + b[k];
    /* Teste para verificar se os limites do tabuleiro
       serão respeitados. */
    if ((u >= 0) && (u <= 7) && (v >= 0) && (v <= 7))
      if (t[u][v] == 0) {
        t[u][v] = i;
        if (i < n * n) { // tabuleiro não está cheio
          q = tenta (i+1, u, v); // tenta novo movimento
          if (!q) t[u][v] = 0; // não sucedido apaga reg. anterior
        }
        else q = true;
      }
  } while (!q && (k != 7)); // não há casas a visitar a partir de x,y
  return q;
}
```

Implementação do Passeio do Cavalo

```
public void imprimePasseio () {
    for (int i = 0; i < n; i++) {
      for (int j = 0; j < n; j++)
        System.out.print ("\t" +this.t[i][j]);
      System.out.println ();
    }
  }
  public static void main (String[] args) {
    PasseioCavalo passeioCavalo = new PasseioCavalo (8);
    boolean q = passeioCavalo.tenta (2, 0, 0);
    if (q) passeioCavalo.imprimePasseio();
    else System.out.println ("Sem solucao");
  }
```

Divisão e Conquista

- Consiste em dividir o problema em partes menores, encontrar soluções para as partes, e combiná-las em uma solução global.
- Exemplo: encontrar o maior e o menor elemento de um vetor de inteiros, $v[0..n-1]$, $n \geq 1$,. *Algoritmo na próxima página*.
- Cada chamada de $maxMin4$ atribui à $maxMin[0]$ e $maxMin[1]$ o maior e o menor elemento em $v[linf]$, $v[linf+1], \ldots,$ $v[lsup]$, respectivamente.

Divisão e Conquista

```
package cap2;
public class MaxMin4 {
  public static int [] maxMin4(int v[], int linf, int lsup){
    int maxMin[] = new int[2];
    if (lsup – linf <= 1) {
      if (v[linf] < v[lsup]) {
            maxMin[0] = v[lsup]; maxMin[1] = v[linf];
            }
      else {
            maxMin[0] = v[linf]; maxMin[1] = v[lsup];
      }
    }
    else {
      int meio = (linf + lsup)/2;
      maxMin = maxMin4 (v, linf, meio);
      int max1 = maxMin[0], min1 = maxMin[1];
      maxMin = maxMin4 (v, meio + 1, lsup);
      int max2 = maxMin[0], min2 = maxMin[1];
      if (max1 > max2) maxMin[0] = max1;
      else maxMin[0] = max2;
      if (min1 < min2) maxMin[1] = min1;
      else maxMin[1] = min2;
    }
    return maxMin;
  }
}
```

Divisão e Conquista - Análise do Exemplo

- Seja $T(n)$ uma função de complexidade tal que $T(n)$ é o número de comparações entre os elementos de v, se v contiver n elementos.

$$\begin{aligned} T(n) &= 1, && \text{para } n \leq 2, \\ T(n) &= T(\lfloor n/2 \rfloor) + T(\lceil n/2 \rceil) + 2, && \text{para } n > 2. \end{aligned}$$

- Quando $n = 2^i$ para algum inteiro positivo i:

$$\begin{aligned} T(n) &= 2T(n/2) + 2 \\ 2T(n/2) &= 4T(n/4) + 2 \times 2 \\ 4T(n/4) &= 8T(n/8) + 2 \times 2 \times 2 \\ &\vdots \quad\quad \vdots \\ 2^{i-2}T(n/2^{i-2}) &= 2^{i-1}T(n/2^{i-1}) + 2^{i-1} \end{aligned}$$

- Adicionando lado a lado, obtemos:

$$\begin{aligned} &T(n) = 2^{i-1}T(n/2^{i-1}) + \textstyle\sum_{k=1}^{i-1} 2^k = \\ &= 2^{i-1}T(2) + 2^i - 2 = 2^{i-1} + 2^i - 2 = \tfrac{3n}{2} - 2. \end{aligned}$$

- Logo, $T(n) = 3n/2 - 2$ para o melhor caso, o pior caso e o caso médio.

Divisão e Conquista - Análise do Exemplo

- Conforme o Teorema da página 10 do livro, o algoritmo anterior é **ótimo**.
- Entretanto, ele pode ser pior do que os apresentados no Capítulo 1, pois, a cada chamada do método salva os valores de $linf$, $lsup$, $maxMin[0]$ e $maxMin[1]$, além do endereço de retorno dessa chamada.
- Além disso, uma comparação adicional é necessária a cada chamada recursiva para verificar se $lsup - linf \leq 1$.
- n deve ser menor do que a metade do maior inteiro que pode ser representado pelo compilador, para não provocar *overflow* na operação $linf + lsup$.

Divisão e Conquista - Teorema Mestre

- Teorema Mestre: Sejam $a \geq 1$ e $b > 1$ constantes, $f(n)$ uma função assintoticamente positiva e $T(n)$ uma medida de complexidade definida sobre os inteiros. A solução da equação de recorrência:

$$T(n) = aT(n/b) + f(n),$$

para b uma potência de n é:

1. $T(n) = \Theta(n^{\log_b a})$, se $f(n) = O(n^{\log_b a - \epsilon})$ para alguma constante $\epsilon > 0$,
2. $T(n) = \Theta(n^{\log_b a} \log n)$, se $f(n) = \Theta(n^{\log_b a})$,
3. $T(n) = \Theta(f(n))$, se $f(n) = \Omega(n^{\log_b a + \epsilon})$ para alguma constante $\epsilon > 0$, e se $af(n/b) \leq cf(n)$ para alguma constante $c < 1$ e todo n a partir de um valor suficientemente grande.

- O problema é dividido em a subproblemas de tamanho n/b cada um sendo resolvidos recursivamente em tempo $T(n/b)$ cada.
- A função $f(n)$ descreve o custo de dividir o problema em subproblemas e de combinar os resultados de cada subproblema.

Divisão e Conquista - Teorema Mestre

- A prova desse teorema não precisa ser entendida para ele ser aplicado.
- Em cada um dos três casos a função $f(n)$ é comparada com a função $n^{\log_b a}$ e a solução de $T(n)$ é determinada pela maior dessas duas funções.
  - No caso 1, $f(n)$ tem de ser polinomialmente menor do que $n^{\log_b a}$.
  - No caso 2, se as duas funções são iguais, então $T(n) = \Theta(n^{\log_b a} \log n) = \Theta(f(n) \log n)$.
  - No caso 3, $f(n)$ tem de ser polinomialmente maior do que $n^{\log_b a}$ e, além disso, satisfazer a condição de que $af(n/b) \leq cf(n)$.
- Ele não pode ser aplicado nas aplicações que ficam entre os casos 1 e 2 (quando $f(n)$ é menor do que $n^{\log_b a}$, mas não polinomialmente menor), entre os casos 2 e 3 (quando $f(n)$ é maior do que $n^{\log_b a}$, mas não polinomialmente maior) ou quando a condição $af(n/b) \leq cf(n)$ não é satisfeita.

Divisão e Conquista - Exemplo do Uso do Teorema Mestre

- Considere a equação de recorrência:

$$T(n) = 4T(n/2) + n,$$

onde $a = 4$, $b = 2$, $f(n) = n$ e $n^{\log_b a} = n^{\log_2 4} = \Theta(n^2)$.

- O caso 1 se aplica porque $f(n) = O(n^{\log_b a - \epsilon}) = O(n)$, onde $\epsilon = 1$, e a solução é $T(n) = \Theta(n^2)$.

Balanceamento

- No projeto de algoritmos, é importante procurar sempre manter o **balanceamento** na subdivisão de um problema em partes menores.
- Divisão e conquista não é a única técnica em que balanceamento é útil.

Vamos considerar um exemplo de ordenação

- Seleciona o menor elemento do conjunto $v[0..n-1]$ e então troca este elemento com o primeiro elemento $v[0]$.
- Repete o processo com os $n-1$ elementos, resultando no segundo maior elemento, o qual é trocado com o segundo elemento $v[1]$.
- Repetindo para $n-2$, $n-3$, . . ., 2 ordena a seqüência.

Balanceamento - Análise do Exemplo

- O algoritmo leva à equação de recorrência: $T(n) = T(n-1) + n - 1, \quad T(1) = 0$, para o número de comparações entre elementos.
- Substituindo:

$$\begin{aligned} T(n) &= T(n-1) + n - 1 \\ T(n-1) &= T(n-2) + n - 2 \\ \vdots & \quad \vdots \\ T(2) &= T(1) + 1 \end{aligned}$$

- Adicionando lado a lado, obtemos: $T(n) = T(1) + 1 + 2 + \cdots + n - 1 = \frac{n(n-1)}{2}$.
- Logo, o algorimo é $O(n^2)$.
- Embora o algoritmo possa ser visto como uma aplicação recursiva de divisão e conquista, ele não é eficiente para valores grandes de n.
- Para obter eficiência assintótica é necessário **balanceamento**: dividir em dois subproblemas de tamanhos aproximadamente iguais, ao invés de um de tamanho 1 e o outro de tamanho $n - 1$.

Exemplo de Balanceamento - Mergesort

- **Intercalação**: unir dois arquivos ordenados gerando um terceiro ordenado (*merge*).
- Colocar no terceiro arquivo o menor elemento entre os menores dos dois arquivos iniciais, desconsiderando este mesmo elemento nos passos posteriores.
- Este processo deve ser repetido até que todos os elementos dos arquivos de entrada sejam escolhidos.
- Algoritmo de ordenação (**Mergesort**):
  - dividir recursivamente o vetor a ser ordenado em dois, até obter n vetores de 1 único elemento.
  - Aplicar a intercalação tendo como entrada 2 vetores de um elemento, formando um vetor ordenado de dois elementos.
  - Repetir este processo formando vetores ordenados cada vez maiores até que todo o vetor esteja ordenado.

Exemplo de Balanceamento - Implementação do Mergesort

```
package cap2;
public class Ordenacao {
  public static void mergeSort (int v[], int i, int j) {
    if (i < j) {
      int m = (i + j)/2;
      mergeSort (v, i, m); mergeSort (v, m + 1, j);
      merge (v, i, m, j); // Intercala v[i..m] e v[m+1..j] em
v[i..j]
    }
  }
}
```

- Considere n como sendo uma potência de 2.
- $merge(v, i, m, j)$, recebe duas seqüências ordenadas $v[i..m]$ e $v[m + 1..j]$ e produz uma outra seqüência ordenada dos elementos de $v[i..m]$ e $v[m + 1..j]$.
- Como $v[i..m]$ e $v[m + 1..j]$ estão ordenados, $merge$ requer no máximo $n - 1$ comparações.
- $merge$ seleciona repetidamente o menor dentre os menores elementos restantes em $v[i..m]$ e $v[m + 1..j]$. Caso empate, retira de qualquer uma delas.

Análise do Mergesort

- Na contagem de comparações, o comportamento do Mergesort pode ser representado por:
  $T(n) = 2T(n/2) + n - 1, \quad T(1) = 0$
- No caso da equação acima temos:

$$\begin{aligned} T(n) &= 2T(n/2) + n - 1 \\ 2T(n/2) &= 2^2 T(n/2^2) + 2\frac{n}{2} - 2 \times 1 \\ &\vdots \quad\quad \vdots \\ 2^{i-1}T(n/2^{i-1}) &= 2^i T(n/2^i) + 2^{i-1}\frac{n}{2^{i-1}} - 2^{i-1} \end{aligned}$$

- Adicionando lado a lado:

$$\begin{aligned} T(n) &= 2^i T(n/2^i) + \sum_{k=0}^{i-1} n - \sum_{k=0}^{i-1} 2^k \\ &= in - \frac{2^{i-1+1} - 1}{2 - 1} \\ &= n \log n - n + 1. \end{aligned}$$

- Logo, o algoritmo é $O(n \log n)$.
- Para valores grandes de n, o balanceamento levou a um resultado muito superior, saímos de $O(n^2)$ para $O(n \log n)$.

Programação Dinâmica

- Quando a soma dos tamanhos dos subproblemas é $O(n)$ então é provável que o algoritmo recursivo tenha **complexidade polinomial**.
- Quando a divisão de um problema de tamanho n resulta em n subproblemas de tamanho $n-1$ então é provável que o algoritmo recursivo tenha **complexidade exponencial**.
- Nesse caso, a técnica de programação dinâmica pode levar a um algoritmo mais eficiente.
- A programação dinâmica calcula a solução para todos os subproblemas, partindo dos subproblemas menores para os maiores, armazenando os resultados em uma tabela.
- A vantagem é que uma vez que um subproblema é resolvido, a resposta é armazenada em uma tabela e nunca mais é recalculado.

Programação Dinâmica - Exemplo

Produto de n matrizes

- $M = M_1 \times M_2 \times \cdots \times M_n$, onde cada M_i é uma matriz com d_{i-1} linhas e d_i colunas.
- A ordem da multiplicação pode ter um efeito enorme no número total de operações de adição e multiplicação necessárias para obter M.
- Considere o produto de uma matriz $p \times q$ por outra matriz $q \times r$ cujo algoritmo requer $O(pqr)$ operações.
- Considere o produto $M = M_1[10, 20] \times M_2[20, 50] \times M_3[50, 1] \times M_4[1, 100]$, onde as dimensões de cada matriz está mostrada entre colchetes.
- A avaliação de M na ordem $M = M_1 \times (M_2 \times (M_3 \times M_4))$ requer 125.000 operações, enquanto na ordem $M = (M_1 \times (M_2 \times M_3)) \times M_4$ requer apenas 2.200.

Programação Dinâmica - Exemplo

- Tentar todas as ordens possíveis para minimizar o número de operações $f(n)$ é exponencial em n, onde $f(n) \geq 2^{n-2}$.
- Usando programação dinâmica é possível obter um algoritmo $O(n^3)$.
- Seja m_{ij} menor custo para computar $M_i \times M_{i+1} \times \cdots \times M_j$, para $1 \leq i \leq j \leq n$.
- Neste caso,

$$m_{ij} = \begin{cases} 0, & \text{se } i = j, \\ \text{Min}_{i \leq k < j} \ (m_{ik} + m_{k+1,j} + d_{i-1} d_k d_j), & \text{se } j > i. \end{cases}$$

- m_{ik} representa o custo mínimo para calcular $M' = M_i \times M_{i+1} \times \cdots \times M_k$
- $m_{k+1,j}$ representa o custo mínimo para calcular $M'' = M_{k+1} \times M_{k+2} \times \cdots \times M_j$.
- $d_{i-1} d_k d_j$ representa o custo de multiplicar $M'[d_{i-1}, d_k]$ por $M''[d_k, d_j]$.
- $m_{ij},\ j > i$ representa o custo mínimo de todos os valores possíveis de k entre i e $j - 1$, da soma dos três termos.

Programação Dinâmica - Exemplo

- O enfoque programação dinâmica calcula os valores de m_{ij} na ordem crescente das diferenças nos subscritos.
- O calculo inicia com m_{ii} para todo i, depois $m_{i,i+1}$ para todo i, depois $m_{i,i+2}$, e assim sucessivamente.
- Desta forma, os valores m_{ik} e $m_{k+1,j}$ estarão disponíveis no momento de calcular m_{ij}.
- Isto acontece porque $j - i$ tem que ser estritamente maior do que ambos os valores de $k - i$ e $j - (k + 1)$ se k estiver no intervalo $i \leq k < j$.
- Programa para computar a ordem de multiplicação de n matrizes, $M_1 \times M_2 \times \cdots \times M_n$, de forma a obter o menor número possível de operações.

Programação Dinâmica - Implementação

```
package cap2;
import java.io.*;
public class AvaliaMultMatrizes {
  public static void main(String[] args)throws IOException {
    int n, maxn = Integer.parseInt (args[0]);
    int d[] = new int[maxn + 1];
    int m[][] = new int[maxn][maxn];
    BufferedReader in = new BufferedReader (
                        new InputStreamReader (System.in));
    System.out.print ("Numero de matrizes n:");
    n = Integer.parseInt (in.readLine());
    System.out.println ("Dimensoes das matrizes:");
    for (int i = 0; i <= n; i++) {
      System.out.print (" d["+i+"] = ");
      d[i] = Integer.parseInt (in.readLine());
   }
    // Continua na próxima transparência...
```

Programação Dinâmica - Continuação da Implementação

```
    for (int i = 0; i < n; i++) m[i][i] = 0;
    for (int h = 1; h < n; h++) {
      for (int i = 1; i <= n - h; i++) {
        int j = i + h;
        m[i-1][j-1] = Integer.MAX_VALUE;
        for (int k = i; k < j; k++) {
          int temp = m[i-1][k-1] + m[k][j-1] +
          + d[i-1] * d[k] * d[j];
          if (temp < m[i-1][j-1]) m[i-1][j-1] = temp;
        }
        System.out.print(" m[" +i+"]["+j+"]= "+ m[i-1][j-1]);
      }
      System.out.println();
    }
  }
}
```

Programação Dinâmica - Implementação

- A execução do programa obtém o custo mínimo para multiplicar as n matrizes, assumindo que são necessárias pqr operações para multiplicar uma matriz $p \times q$ por outra matriz $q \times r$.
- A execução do programa para as quatro matrizes onde d_0, d_1, d_2, d_3, d_4 são 10, 20, 50, 1, 100, resulta:

$m_{11} = 0$	$m_{22} = 0$	$m_{33} = 0$	$m_{44} = 0$
$m_{12} = 10.000$	$m_{23} = 1.000$	$m_{34} = 5.000$	
$m_{13} = 1.200$	$m_{24} = 3.000$		
$m_{14} = 2.200$			

Programação Dinâmica - Princípio da Otimalidade

- A ordem de multiplicação pode ser obtida registrando o valor de k para cada entrada da tabela que resultou no mínimo.
- Essa solução eficiente está baseada no **princípio da otimalidade**:
  - em uma seqüência ótima de escolhas ou de decisões cada subseqüência deve também ser ótima.
- Cada subseqüência representa o custo mínimo, assim como m_{ij}, $j > i$.
- Assim, todos os valores da tabela representam escolhas ótimas.
- O princípio da otimalidade não pode ser aplicado indiscriminadamente.
- Quando o princípio não se aplica é provável que não se possa resolver o problema com sucesso por meio de programação dinâmica.

Aplicação do Princípio da Otimalidade

- Por exemplo, quando o problema utiliza recursos limitados, quando o total de recursos usados nas subinstâncias é maior do que os recursos disponíveis.
- Se o caminho mais curto entre Belo Horizonte e Curitiba passa por Campinas:
  - o caminho entre Belo Horizonte e Campinas também é o mais curto possível
  - assim como o caminho entre Campinas e Curitiba.
  - Logo, o princípio da otimalidade se aplica.

Não Aplicação do Princípio da Otimalidade

- No problema de encontrar o caminho mais longo entre duas cidades:
  - Um caminho simples nunca visita uma mesma cidade duas vezes.
  - Se o caminho mais longo entre Belo Horizonte e Curitiba passa por Campinas, isso não significa que o caminho possa ser obtido tomando o caminho simples mais longo entre Belo Horizonte e Campinas e depois o caminho simples mais longo entre Campinas e Curitiba.
  - Quando os dois caminhos simples são ajuntados é pouco provável que o caminho resultante também seja simples.
  - Logo, o princípio da otimalidade não se aplica.

Algoritmos Gulosos

- Resolve problemas de otimização.
- Exemplo: algoritmo para encontrar o caminho mais curto entre dois vértices de um grafo.
  - Escolhe a aresta que parece mais promissora em qualquer instante;
  - Independente do que possa acontecer mais tarde, nunca reconsidera a decisão.
- Não necessita avaliar alternativas, ou usar procedimentos sofisticados para desfazer decisões tomadas previamente.
- Problema geral: dado um conjunto C, determine um subconjunto $S \subseteq C$ tal que:
  - S satisfaz uma dada propriedade P, e
  - S é mínimo (ou máximo) em relação a algum critério α.
- O **algoritmo guloso** para resolver o problema geral consiste em um processo iterativo em que S é construído adicionando-se ao mesmo elementos de C um a um.

Características dos Algoritmos Gulosos

- Para construir a solução ótima existe um conjunto ou lista de candidatos.
- São acumulados um conjunto de candidatos considerados e escolhidos, e o outro de candidatos considerados e rejeitados.
- Existe função que verifica se um conjunto particular de candidatos produz uma *solução* (sem considerar otimalidade no momento).
- Outra função verifica se um conjunto de candidatos é *viável* (também sem preocupar com a otimalidade).
- Uma *função de seleção* indica a qualquer momento quais dos candidatos restantes é o mais promissor.
- Uma *função objetivo* fornece o valor da solução encontrada, como o comprimento do caminho construído (não aparece de forma explicita no algoritmo guloso).

Pseudo Código de Algoritmo Guloso

```
Conjunto guloso (Conjunto C) { /* C: conjunto de candidatos */
  S = ∅; /* S contém conjunto solução */
  while ((C ≠ ∅) && not solução(S)) {
     x = seleciona (C);
     C = C − x;
     if (viável (S + x)) S = S + x;
    }
    if (solução (S))
        return S else return ("Não existe solução");
}
```

- Inicialmente, o conjunto S de candidatos escolhidos está vazio.
- A cada passo, o melhor candidato restante ainda não tentado é considerado. O critério de escolha é ditado pela função de seleção.
- Se o conjunto aumentado de candidatos se torna inviável, o candidato é rejeitado. Senão, o candidato é adicionado ao conjunto S de candidatos escolhidos.
- A cada aumento de S verificamos se S constitui uma solução ótima.

Características da Implementação de Algoritmos Gulosos

- Quando funciona corretamente, a primeira solução encontrada é sempre ótima.
- A função de seleção é geralmente relacionada com a função objetivo.
- Se o objetivo é:
  - maximizar $\Rightarrow$ provavelmente escolherá o candidato restante que proporcione o maior ganho individual.
  - minimizar $\Rightarrow$ então será escolhido o candidato restante de menor custo.
- O algoritmo nunca muda de idéia:
  - Uma vez que um candidato é escolhido e adicionado à solução ele lá permanece para sempre.
  - Uma vez que um candidato é excluído do conjunto solução, ele nunca mais é reconsiderado.

Algoritmos Aproximados

- Problemas que somente possuem algoritmos exponenciais para resolvê-los são considerados “difíceis”.
- Problemas considerados intratáveis ou difíceis são muito comuns.
- Exemplo: **problema do caixeiro viajante** cuja complexidade de tempo é $O(n!)$.
- Diante de um problema difícil é comum remover a exigência de que o algoritmo tenha sempre que obter a solução ótima.
- Neste caso procuramos por algoritmos eficientes que não garantem obter a solução ótima, mas uma que seja a mais próxima possível da solução ótima.

Tipos de Algoritmos Aproximados

- **Heurística**: é um algoritmo que pode produzir um bom resultado, ou até mesmo obter a solução ótima, mas pode também não produzir solução alguma ou uma solução que está distante da solução ótima.
- **Algoritmo aproximado**: é um algoritmo que gera **soluções aproximadas** dentro de um limite para a razão entre a solução ótima e a produzida pelo algoritmo aproximado (comportamento monitorado sob o ponto de vista da qualidade dos resultados).

Estruturas de Dados Básicas*

Última alteração: 10 de Outubro de 2006

*Transparências elaboradas por Charles Ornelas, Leonardo Rocha, Leonardo Mata e Nivio Ziviani

Listas Lineares

- Uma das formas mais simples de interligar os elementos de um conjunto.
- Estrutura em que as operações inserir, retirar e localizar são definidas.
- Podem crescer ou diminuir de tamanho durante a execução de um programa, de acordo com a demanda.
- Itens podem ser acessados, inseridos ou retirados de uma lista.
- Duas listas podem ser concatenadas para formar uma lista única, ou uma pode ser partida em duas ou mais listas.
- Adequadas quando não é possível prever a demanda por memória, permitindo a manipulação de quantidades imprevisíveis de dados, de formato também imprevisível.
- São úteis em aplicações tais como manipulação simbólica, gerência de memória, simulação e compiladores.

Definição de Listas Lineares

- Seqüência de zero ou mais itens $x_1, x_2, \cdots, x_n$, na qual x_i é de um determinado tipo e n representa o tamanho da lista linear.
- Sua principal propriedade estrutural envolve as posições relativas dos itens em uma dimensão.
  - Assumindo $n \geq 1$, x_1 é o primeiro item da lista e x_n é o último item da lista.
  - x_i precede x_{i+1} para $i = 1, 2, \cdots, n-1$
  - x_i sucede x_{i-1} para $i = 2, 3, \cdots, n$
  - o elemento x_i é dito estar na i-ésima posição da lista.

TAD Listas Lineares

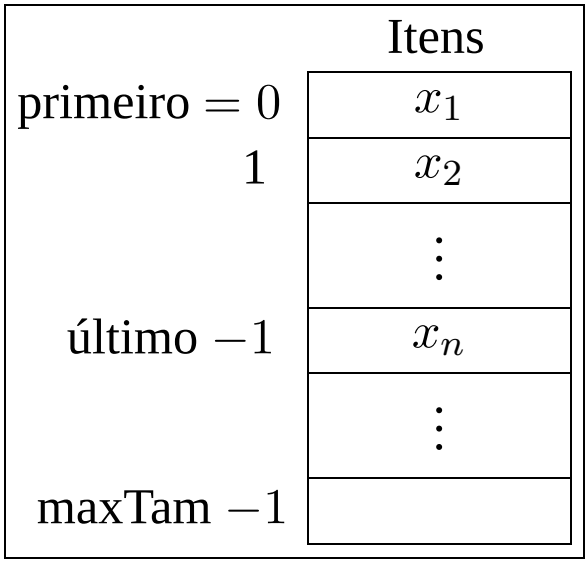
- O conjunto de operações a ser definido depende de cada aplicação.
- Um conjunto de operações necessário a uma maioria de aplicações é:
  1. Criar uma lista linear vazia.
  2. Inserir um novo item imediatamente após o *i*-ésimo item.
  3. Retirar o *i*-ésimo item.
  4. Localizar o *i*-ésimo item para examinar e/ou alterar o conteúdo de seus componentes.
  5. Combinar duas ou mais listas lineares em uma lista única.
  6. Partir uma lista linear em duas ou mais listas.
  7. Fazer uma cópia da lista linear.
  8. Ordenar os itens da lista em ordem ascendente ou descendente, de acordo com alguns de seus componentes.
  9. Pesquisar a ocorrência de um item com um valor particular em algum componente.

Implementações de Listas Lineares

- Várias estruturas de dados podem ser usadas para representar listas lineares, cada uma com vantagens e desvantagens particulares.
- As duas representações mais utilizadas são as implementações por meio de arranjos e de estruturas auto-referenciadas.
- Exemplo de Conjunto de Operações:
  1. Lista(maxTam). Cria uma lista vazia.
  2. insere(x). Insere x após o último item da lista.
  3. retira(x). Retorna o item x que está na posição p da lista, retirando-o da lista e deslocando os itens a partir da posição p+1 para as posições anteriores.
  4. vazia(). Esta função retorna *true* se lista vazia; senão retorna *false*.
  5. imprime(). Imprime os itens da lista na ordem de ocorrência.

Implementação de Listas por meio de Arranjos

- Os itens da lista são armazenados em posições contíguas de memória.
- A lista pode ser percorrida em qualquer direção.
- A inserção de um novo item pode ser realizada após o último item com custo constante.
- A inserção de um novo item no meio da lista requer um deslocamento de todos os itens localizados após o ponto de inserção.
- Retirar um item do início da lista requer um deslocamento de itens para preencher o espaço deixado vazio.



Estrutura da Lista Usando Arranjo

- Os itens são armazenados em um arranjo de tamanho suficiente para armazenar a lista.
- O campo Último referencia para a posição seguinte a do último elemento da lista.
- O *i*-ésimo item da lista está armazenado na *i*-ésima posição do arranjo, $1 \leq i <$Último.
- A constante MaxTam define o tamanho máximo permitido para a lista.

```
package cap3.arranjo;
public class Lista {
  private Object item[];
  private int primeiro, ultimo, pos;
  // Operações
  public Lista (int maxTam) { // Cria uma Lista vazia
    this.item = new Object[maxTam]; this.pos = -1;
    this.primeiro = 0; this.ultimo = this.primeiro;
  }
```

Operações sobre Lista Usando Arranjo

```
public Object pesquisa (Object chave) {
  if (this.vazia () || chave == null) return null;
  for (int p = 0; p < this.ultimo; p++)
    if (this.item[p].equals (chave)) return this.item[p];
  return null;
}
public void insere (Object x) throws Exception {
  if (this.ultimo >= this.item.length)
    throw new Exception ("Erro: A lista esta cheia");
  else { this.item[this.ultimo] = x;
         this.ultimo = this.ultimo + 1; }
}
public Object retira (Object chave) throws Exception {
  if (this.vazia () || chave == null)
    throw new Exception ("Erro : A lista esta vazia");
  int p = 0;
  while(p < this.ultimo && !this.item[p].equals(chave))p++;
  if (p >= this.ultimo) return null; // Chave não en-
contrada
  Object item = this.item[p];
  this.ultimo = this.ultimo – 1;
  for (int aux = p; aux < this.ultimo; aux++)
    this.item[aux] = this.item[aux + 1];
  return item;
}
```

Operações sobre Lista Usando Arranjo

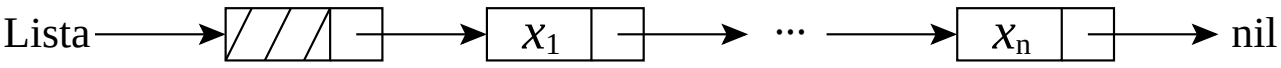
```
  public Object retiraPrimeiro () throws Exception {
    if (this.vazia ()) throw new Exception
            ("Erro : A lista esta vazia");
    Object item = this.item[0];
    this.ultimo = this.ultimo – 1;
    for (int aux = 0; aux < this.ultimo; aux++)
      this.item[aux] = this.item[aux + 1];
    return item;
  }
  public Object primeiro () {
          this.pos = –1; return this.proximo (); }
  public Object proximo () {
    this.pos++;
    if (this.pos >= this.ultimo) return null;
    else return this.item[this.pos];
  }
  public boolean vazia () {
          return (this.primeiro == this.ultimo); }
  public void imprime () {
    for (int aux = this.primeiro; aux < this.ultimo; aux++)
      System.out.println (this.item[aux].toString ());
  }
}
```

Lista Usando Arranjo - Vantagens e Desvantagens

- Vantagem: economia de memória (os apontadores são implícitos nesta estrutura).
- Desvantagens:
  - custo para inserir ou retirar itens da lista, que pode causar um deslocamento de todos os itens, no pior caso;
  - em aplicações em que não existe previsão sobre o crescimento da lista, a utilização de arranjos exigir a realocação de memória.

Implementação de Listas por meio de Estruturas Auto-Referenciadas

- Cada item da lista contém a informação que é necessária para alcançar o próximo item.
- Permite utilizar posições não contíguas de memória.
- É possível inserir e retirar elementos sem necessidade de deslocar os itens seguintes da lista.
- Há uma **célula cabeça** para simplificar as operações sobre a lista.



Implementação de Listas por meio de Estruturas Auto-Referenciadas

- A lista é constituída de células.
- Cada célula contém um item da lista e uma referência para a célula seguinte.
- A classe $Lista$ contém uma referência para a célula cabeça, uma referência para a última célula da lista e uma referência para armazenar a posição corrente na lista.

```
package cap3.autoreferencia;
public class Lista {
  private static class Celula { Object item; Celula prox; }
  private Celula primeiro, ultimo, pos;
  // Operações
  public Lista () { // Cria uma Lista vazia
    this.primeiro = new Celula (); this.pos = this.primeiro;
    this.ultimo = this.primeiro; this.primeiro.prox = null;
  }
```

Lista Usando Estruturas Auto-Referenciadas

```
public Object pesquisa (Object chave) {
    if (this.vazia () || chave == null) return null;
    Celula aux = this.primeiro;
    while (aux.prox != null) {
       if (aux.prox.item.equals (chave)) return aux.prox.item;
      aux = aux.prox;
    } return null;
  }
  public void insere (Object x) {
    this.ultimo.prox = new Celula ();
    this.ultimo = this.ultimo.prox;
    this.ultimo.item = x; this.ultimo.prox = null;
  }
  public Object retira (Object chave) throws Exception {
    if (this.vazia () || (chave == null))
      throw new Exception
      ("Erro: Lista vazia ou chave invalida");
    Celula aux = this.primeiro;
    while (aux.prox!=null && !aux.prox.item.equals(chave))
            aux=aux.prox;
    if (aux.prox == null) return null; // não encontrada
    Celula q = aux.prox;
    Object item = q.item; aux.prox = q.prox;
    if (aux.prox == null) this.ultimo = aux;  return item;
  }
```

Lista Usando Estruturas Auto-Referenciadas

```
public Object retiraPrimeiro () throws Exception {
    if (this.vazia ()) throw new Exception
            ("Erro: Lista vazia");
    Celula aux = this.primeiro; Celula q = aux.prox;
    Object item = q.item; aux.prox = q.prox;
    if (aux.prox == null) this.ultimo = aux;  return item;
  }
  public Object primeiro () {
          this.pos = primeiro; return proximo (); }
  public Object proximo () {
    this.pos = this.pos.prox;
    if (this.pos == null) return null;
    else return this.pos.item;
  }
  public boolean vazia () {
          return (this.primeiro == this.ultimo); }
  public void imprime () {
    Celula aux = this.primeiro.prox;
    while (aux != null) {
      System.out.println (aux.item.toString ());
      aux = aux.prox; }
  }
}
```

Lista Usando Estruturas Auto-Referenciadas - Vantagens e Desvantagens

- Vantagens:
  - Permite inserir ou retirar itens do meio da lista a um custo constante (importante quando a lista tem de ser mantida em ordem).
  - Bom para aplicações em que não existe previsão sobre o crescimento da lista (o tamanho máximo da lista não precisa ser definido *a priori*).
- Desvantagem: utilização de memória extra para armazenar as referências.

Exemplo de Uso Listas - Vestibular

- Num vestibular, cada candidato tem direito a três opções para tentar uma vaga em um dos sete cursos oferecidos.
- Para cada candidato é lido um registro:
  - $chave$: número de inscrição do candidato.
  - $notaFinal$: média das notas do candidato.
  - $opcao$: vetor contendo a primeira, a segunda e a terceira opções de curso do candidato.

```
short chave;      // assume valores de 1 a 999.
byte  notaFinal;  // assume valores de 0 a 10.
byte  opcao[];    // arranjo de 3 posições;
```

- Problema: distribuir os candidatos entre os cursos, segundo a nota final e as opções apresentadas por candidato.
- Em caso de empate, os candidatos serão atendidos na ordem de inscrição para os exames.

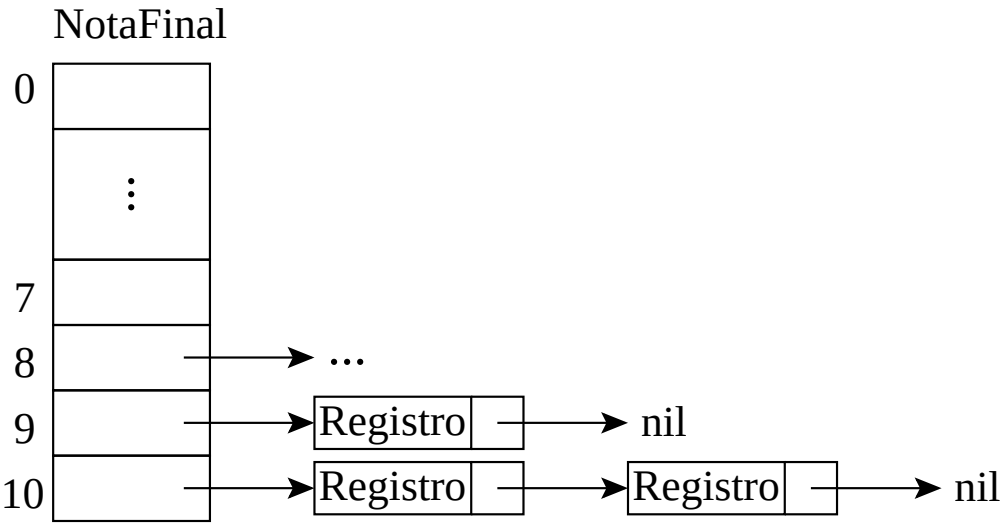
Vestibular - Possível Solução

- ordenar registros pelo campo $notaFinal$, respeitando a ordem de inscrição;
- percorrer cada conjunto de registros com mesma $notaFinal$, começando pelo conjunto de $notaFinal$, 10, seguido pelo de $notaFinal$ 9, e assim por diante.
  - Para um conjunto de mesma $notaFinal$ tenta-se encaixar cada registro desse conjunto em um dos cursos, na primeira das três opções em que houver vaga (se houver).
- Primeiro refinamento:

```
void Vestibular {
  ordena os registros pelo campo notaFinal;
  for (nota = 10; nota >= 0; nota--)
    while (houver registro com mesma nota)
      if (existe vaga em um dos cursos
                de opção do candidato)
        insere registro no conjunto de aprovados
      else insere registro no conjunto de reprovados;
  imprime aprovados por curso;
  imprime reprovados;
}
```

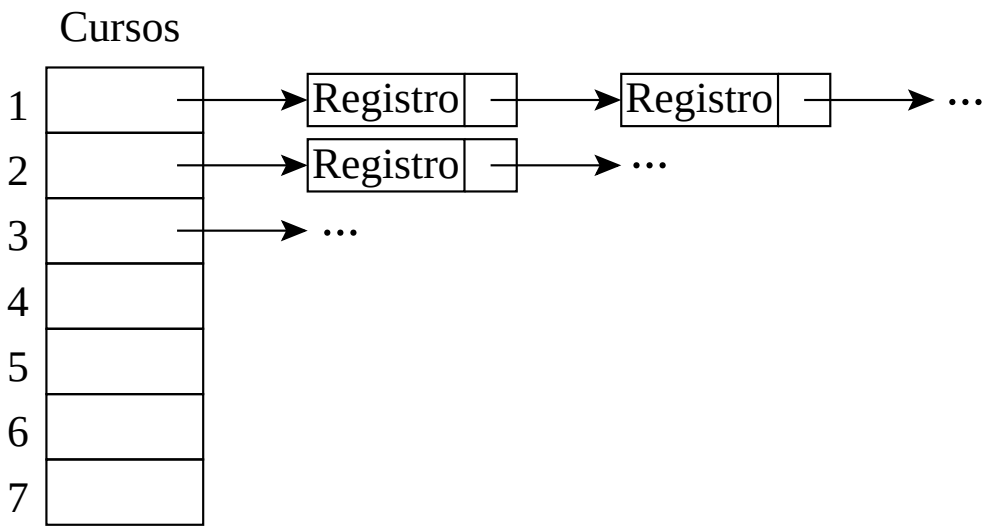
Vestibular - Classificação dos Alunos

- Uma boa maneira de representar um conjunto de registros é com o uso de listas.
- Ao serem lidos, os registros são armazenados em listas para cada nota.
- Após a leitura do último registro os candidatos estão automaticamente ordenados por $notaFinal$.
- Dentro de cada lista, os registros estão ordenados por ordem de inscrição, desde que os registros sejam lidos na ordem de inscrição de cada candidato e inseridos nesta ordem.



Vestibular - Classificação dos Alunos

- As listas de registros são percorridas, iniciando-se pela de $notaFinal$ 10, seguida pela de $notaFinal$ 9, e assim sucessivamente.
- Cada registro é retirado e colocado em uma das listas da abaixo, na primeira das três opções em que houver vaga.



- Se não houver vaga, o registro é colocado em uma lista de reprovados.
- Ao final a estrutura acima conterá a relação de candidatos aprovados em cada curso.

Vestibular - Segundo Refinamento

```
void Vestibular {
   lê número de vagas para cada curso;
   inicializa listas de classificação, de aprovados e de reprovados;
   lê registro; // vide formato do registro na transparência 15
   while (chave ≠ 0) {
      insere registro nas listas de classificação, conforme notaFinal;
      lê registro;
   }
   for (nota = 10; nota >= 0; nota−−)
      while (houver próximo registro com mesma notaFinal) {
         retira registro da lista;
         if (existe vaga em um dos cursos de opção do candidato) {
            insere registro na lista de aprovados;
            decrementa o número de vagas para aquele curso;
         }
         else insere registro na lista de reprovados;
         obtém próximo registro;
      }
   imprime aprovados por curso;
   imprime reprovados;
}
```

Vestibular - Refinamento Final

- Observe que o programa é completamente independente da implementação do tipo abstrato de dados $Lista$.

```
package cap3;
import java.io.*;
import cap3.autoreferencia.Lista; // vide programa da transparência 11
public class Vestibular {
  private class Definicoes {
    public static final int nOpcoes = 3;
    public static final int nCursos = 7;
  }
  private static class Registro {
    short chave;  byte notaFinal;
    byte opcao[] = new byte[Definicoes.nOpcoes];
    public String toString () {
            return new String ("" + this.chave); }
  }
  private static BufferedReader in =
      new BufferedReader(new InputStreamReader(System.in));
```

Vestibular - Refinamento Final

```
static Registro leRegistro () throws IOException {
  // os valores lidos devem estar separados por brancos
  Registro registro = new Registro ();
  String str = in.readLine ();
  registro.chave = Short.parseShort (str.substring (0,
                                     str.indexOf (" ")));
  registro.notaFinal = Byte.parseByte (str.substring (
                                   str.indexOf (" ") + 1));
  for (int i = 0; i < Definicoes.nOpcoes; i++)
    registro.opcao[i] = Byte.parseByte (in.readLine ());
  return registro;
}
public static void main (String[] args) {
  Registro registro = null;
  Lista classificacao[] = new Lista[11];
  Lista aprovados[] = new Lista[Definicoes.nCursos];
  Lista reprovados = new Lista ();
  long vagas[] = new long[Definicoes.nCursos];
  boolean passou;
  int i;
  try {
    for (i = 0; i < Definicoes.nCursos; i++)
      vagas[i] = Long.parseLong (in.readLine ());
    for(i = 0; i < 11; i++)classificacao[i] = new Lista();
  // Continua na próxima transparência
```

Vestibular - Refinamento Final (Cont.)

```
    for (i = 0; i < Definicoes.nCursos; i++)
      aprovados[i] = new Lista ();
    registro = leRegistro ();
    while (registro.chave != 0) {
      classificacao[registro.notaFinal].insere (registro);
      registro = leRegistro ();
    }
    for (int Nota = 10; Nota >= 0; Nota--){
      while (!classificacao[Nota].vazia ()) {
        registro =
         (Registro) classificacao[Nota].retiraPrimeiro();
        i = 0; passou = false;
        while (i < Definicoes.nOpcoes && !passou) {
         if (vagas[registro.opcao[i]-1] > 0) {
          aprovados[registro.opcao[i]-1].insere(registro);
          vagas[registro.opcao[i]-1]--; passou = true;
         }
         i++;
        }
        if (!passou) reprovados.insere (registro);
      }
    }
  } catch (Exception e) {
          System.out.println (e.getMessage ()); }
```

// Continua na próxima transparência

Vestibular - Refinamento Final (Cont.)

```
 for (i = 0; i < Definicoes.nCursos; i++) {
       System.out.println ("Relacao dos aprovados no Curso" +
                                                  (i + 1));
       aprovados[i].imprime ();
     }
     System.out.println ("Relacao dos reprovados");
     reprovados.imprime ();
  }
}
```

- O exemplo mostra a importância de utilizar **tipos abstratos de dados** para escrever programas, em vez de utilizar detalhes particulares de implementação.
- Altera-se a implementação rapidamente. Não é necessário procurar as referências diretas às estruturas de dados por todo o código.
- Este aspecto é particularmente importante em programas de grande porte.

Pilha

- É uma lista linear em que todas as inserções, retiradas e, geralmente, todos os acessos são feitos em apenas um extremo da lista.
- Os itens são colocados um sobre o outro. O item inserido mais recentemente está no topo e o inserido menos recentemente no fundo.
- O modelo intuitivo é o de um monte de pratos em uma prateleira, sendo conveniente retirar ou adicionar pratos na parte superior.
- Esta imagem está freqüentemente associada com a teoria de autômato, na qual o topo de uma pilha é considerado como o receptáculo de uma cabeça de leitura/gravação que pode empilhar e desempilhar itens da pilha.

Propriedade e Aplicações das Pilhas

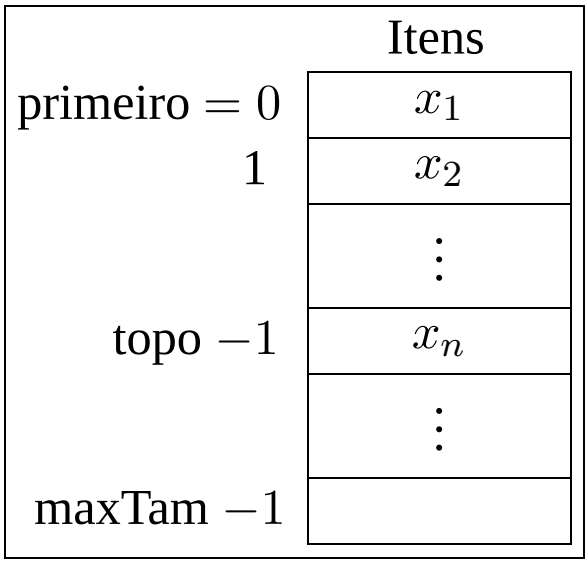
- Propriedade: o último item inserido é o primeiro item que pode ser retirado da lista. São chamadas listas **lifo** (“last-in, first-out”).
- Existe uma ordem linear para pilhas, do “mais recente para o menos recente”.
- É ideal para processamento de estruturas aninhadas de profundidade imprevisível.
- Uma pilha contém uma seqüência de obrigações adiadas. A ordem de remoção garante que as estruturas mais internas serão processadas antes das mais externas.
- Aplicações em estruturas aninhadas:
  - Quando é necessário caminhar em um conjunto de dados e guardar uma lista de coisas a fazer posteriormente.
  - O controle de seqüências de chamadas de subprogramas.
  - A sintaxe de expressões aritméticas.
- As pilhas ocorrem em estruturas de natureza recursiva (como árvores). Elas são utilizadas para implementar a **recursividade**.

TAD Pilhas

- Conjunto de operações:
  1. Cria uma pilha Vazia.
  2. Verifica se a lista está vazia. Retorna *true* se a pilha está vazia; caso contrário, retorna *false*.
  3. Empilhar o item x no topo da pilha.
  4. Desempilhar o item x no topo da pilha, retirando-o da pilha.
  5. Verificar o tamanho atual da pilha.
- Existem várias opções de estruturas de dados que podem ser usadas para representar pilhas.
- As duas representações mais utilizadas são as implementações por meio de *arranjos* e de *estruturas auto-referenciadas*.

Implementação de Pilhas por meio de Arranjos

- Os itens da pilha são armazenados em posições contíguas de memória.
- Como as inserções e as retiradas ocorrem no topo da pilha, um cursor chamado Topo é utilizado para controlar a posição do item no topo da pilha.



Estrutura e Operações sobre Pilhas Usando Arranjos

- Os itens são armazenados em um arranjo de tamanho suficiente para conter a pilha.
- O outro campo do mesmo registro contém uma referência para o item no topo da pilha.
- A constante $maxTam$ define o tamanho máximo permitido para a pilha.

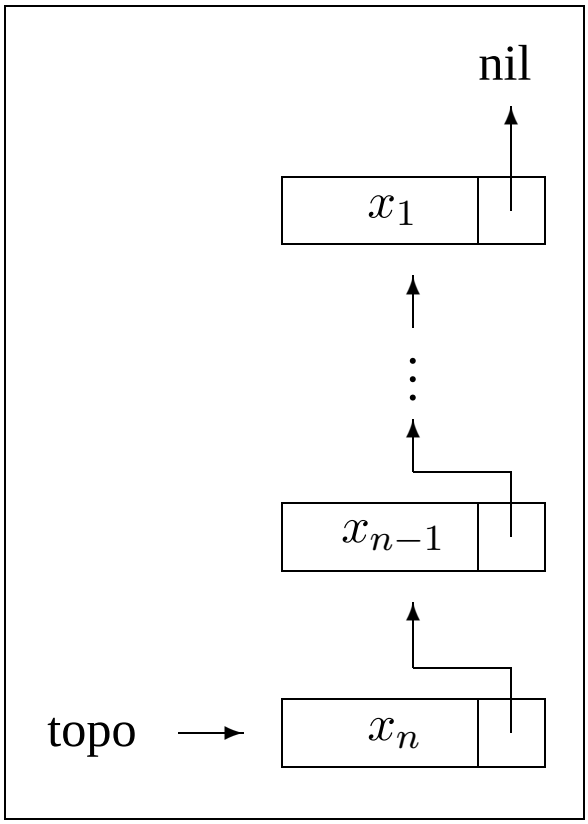
```
package cap3.arranjo;
public class Pilha {
  private Object item[];
  private int    topo;
  // Operações
  public Pilha (int maxTam) { // Cria uma Pilha vazia
    this.item = new Object[maxTam]; this.topo = 0;
  }
  public void empilha (Object x) throws Exception {
    if (this.topo == this.item.length)
      throw new Exception ("Erro: A pilha esta cheia");
    else this.item[this.topo++] = x;
  }
    // Continua na próxima transparência
```

Estrutura e Operações sobre Pilhas Usando Arranjos

```
  public Object desempilha () throws Exception {
    if (this.vazia())
      throw new Exception ("Erro: A pilha esta vazia");
    return this.item[--this.topo];
  }
  public boolean vazia () {
    return (this.topo == 0);
  }
  public int tamanho () {
    return this.topo;
  }
}
```

Implementação de Pilhas por meio de Estruturas Auto-Referenciadas

- Ao contrário da implementação de listas lineares por meio e estruturas auto-referenciadas não há necessidade de manter uma célula cabeça é no topo da pilha.
- Para desempilhar um item, basta desligar a célula que contém x_n e a célula que contém x_{n-1} passa a ser a célula de topo.
- Para empilhar um novo item, basta fazer a operação contrária, criando uma nova célula para receber o novo item.



Estrutura e operações sobre Pilhas Usando Estruturas Auto-Referenciadas

- O campo tam evita a contagem do número de itens no método $tamanho$.
- Cada célula de uma pilha contém um item da pilha e uma referência para outra célula.
- A classe $Pilha$ contém uma referência para o topo da pilha.

```
package cap3.autoreferencia;
public class Pilha {
  private static class Celula {
    Object item;
    Celula prox;
  }
  private Celula topo;
  private int    tam;
  // Operações
  public Pilha () { // Cria uma Pilha vazia
    this.topo = null; this.tam = 0;
  }
  // Continua na próxima transparência
```

Estrutura e operações sobre Pilhas Usando Estruturas Auto-Referenciadas

```
public void empilha (Object x) {
    Celula aux = this.topo;
    this.topo = new Celula ();
    this.topo.item = x;
    this.topo.prox = aux;
    this.tam++;
  }
  public Object desempilha () throws Exception {
    if (this.vazia ())
      throw new Exception ("Erro: A pilha esta vazia");
    Object item = this.topo.item;
    this.topo = this.topo.prox;
    this.tam--;
    return item;
  }
  public boolean vazia () {
    return (this.topo == null);
  }
  public int tamanho () {
    return this.tam;
  }
}
```

Exemplo de Uso Pilhas - Editor de Textos (ET)

- "#": cancelar caractere anterior na linha sendo editada. Ex.: UEM##FMB#G $\rightarrow$ UFMG.
- "\": cancela todos os caracteres anteriores na linha sendo editada.
- "*": salta a linha. Imprime os caracteres que pertencem à linha sendo editada, iniciando uma nova linha de impressão a partir do caractere imediatamente seguinte ao caractere salta-linha. Ex: DCC*UFMG.* $\rightarrow$
  DCC
  UFMG.
- Vamos escrever um Editor de Texto (ET) que aceite os três comandos descritos acima.
- O ET deverá ler um caractere de cada vez do texto de entrada e produzir a impressão linha a linha, cada linha contendo no máximo 70 caracteres de impressão.
- O ET deverá utilizar o **tipo abstrato de dados** $Pilha$ definido anteriormente, implementado por meio de arranjo.

Sugestão de Texto para Testar o ET

Este et# um teste para o ET, o extraterrestre em

JAVA.*Acabamos de testar a capacidade de o ET saltar de linha,

utilizando seus poderes extras (cuidado, pois agora vamos estourar

a capacidade máxima da linha de impressão, que é de 70

caracteres.)*O k#cut#rso dh#e Estruturas de Dados et# h#um

cuu#rsh#o #x# x?*!#?!#+.* Como et# bom

n#nt#ao### r#ess#tt#ar mb#aa#triz#cull#ado nn#x#ele!\ Sera

que este funciona\\\? O sinal? não### deve ficar! ~

ET - Implementação

- Este programa utiliza um tipo abstrato de dados sem conhecer detalhes de sua implementação.
- A implementação do TAD $Pilha$ que utiliza arranjo pode ser substituída pela implementação que utiliza estruturas auto-referenciadas sem causar impacto no programa.

```
package cap3;
import cap3.arranjo.Pilha; // vide programa da transparência 11

public class ET {
  private class Definicoes {
    public static final int  maxTam          = 70;
    public static final char cancelaCarater = '#';
    public static final char cancelaLinha    = '\\';
    public static final char saltaLinha      = '*';
    public static final char marcaEof        = '~';
  }
  private static void imprime(Pilha pilha)throws Exception {
    Pilha pilhaAux = new Pilha (Definicoes.maxTam);
    Character x;
    // Continua na próxima transparência
```

ET - Implementação

```
    while (!pilha.vazia ()) {
      x = (Character) pilha.desempilha ();
      pilhaAux.empilha (x);
    }
    while (!pilhaAux.vazia ()) {
      x = (Character) pilhaAux.desempilha ();
      System.out.print (x);
    }
    System.out.print ('\n');
  }
  public static void main (String[] args) {
    Pilha pilha = new Pilha (Definicoes.maxTam);
    try {
      char c = (char) System.in.read ();
      Character x = new Character (c);
      if (x.charValue () == '\n') x =
                    new Character (' ');
      while (x.charValue () != Definicoes.marcaEof) {
        if (x.charValue () == Definicoes.cancelaCarater) {
          if (!pilha.vazia ()) x =
                    (Character) pilha.desempilha ();
        }
        else if (x.charValue () == Definicoes.cancelaLinha)
          pilha = new Pilha (Definicoes.maxTam);
          // Continua na próxima transparência
```

ET - Implementação

```
            else if (x.charValue () == Definicoes.saltaLinha)
              imprime (pilha);
            else {
              if (pilha.tamanho () == Definicoes.maxTam)
                    imprime (pilha);
              pilha.empilha (x);
            }
            c = (char) System.in.read ();
            x = new Character (c);
            if (x.charValue () == '\n') x = new Character (' ');
          }
          if (!pilha.vazia ()) imprime (pilha);
        } catch (Exception e) {
              System.out.println (e.getMessage ()); }
    }
}
```

Filas

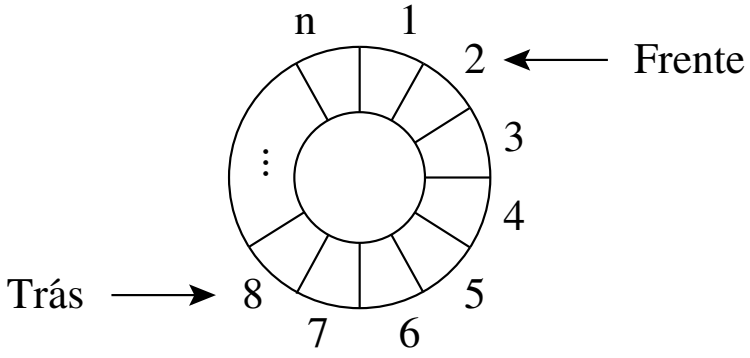
- É uma lista linear em que todas as inserções são realizadas em um extremo da lista, e todas as retiradas e, geralmente, os acessos são realizados no outro extremo da lista.
- O modelo intuitivo de uma fila é o de uma fila de espera em que as pessoas no início da fila são servidas primeiro e as pessoas que chegam entram no fim da fila.
- São chamadas listas **fifo** (“first-in”, “first-out”).
- Existe uma ordem linear para filas que é a “ordem de chegada”.
- São utilizadas quando desejamos processar itens de acordo com a ordem “primeiro-que-chega, primeiro-atendido”.
- Sistemas operacionais utilizam filas para regular a ordem na qual tarefas devem receber processamento e recursos devem ser alocados a processos.

TAD Filas

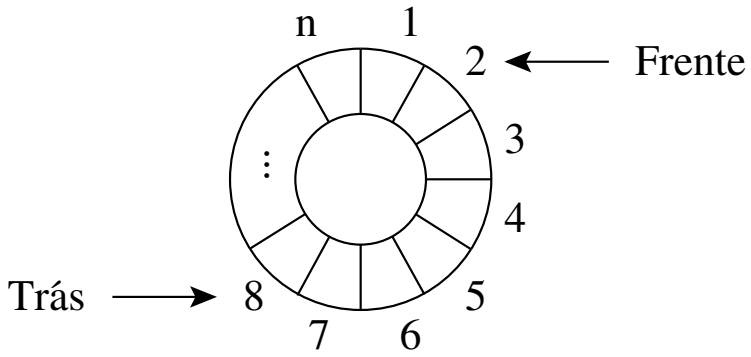
- Conjunto de operações:
  1. Criar uma fila vazia.
  2. Enfileirar o item x no final da fila.
  3. Desenfileirar. Essa função retorna o item x no início da fila e o retira da fila.
  4. Verificar se a fila está vazia. Essa função retorna *true* se a fila está vazia; do contrário, retorna *false*.

Implementação de Filas por meio de Arranjos

- Os itens são armazenados em posições contíguas de memória.
- A operação $enfileira$ faz a parte de trás da fila expandir-se.
- A operação $desenfileira$ faz a parte da frente da fila contrair-se.
- A fila tende a caminhar pela memória do computador, ocupando espaço na parte de trás e descartando espaço na parte da frente.
- Com poucas inserções e retiradas, a fila vai ao encontro do limite do espaço da memória alocado para ela.
- Solução: imaginar o arranjo como um círculo. A primeira posição segue a última.



Implementação de Filas por meio de Arranjos



- A fila se encontra em posições contíguas de memória, em alguma posição do círculo, delimitada pelos apontadores $frente$ e *trás*.
- Para enfileirar, basta mover o apontador *trás* uma posição no sentido horário.
- Para desenfileirar, basta mover o apontador $frente$ uma posição no sentido horário.

Estrutura da Fila Usando Arranjo

- O tamanho máximo do arranjo circular é definido pela constante $maxTam$.
- Os outros campos da classe $Fila$ contêm referências para a parte da frente e de trás da fila.
- Nos casos de fila cheia e fila vazia, os apontadores $frente$ e *trás* apontam para a mesma posição do círculo.
- Uma saída para distinguir as duas situações é deixar uma posição vazia no arranjo.
- Neste caso, a fila está cheia quando *trás*$+1$ for igual a $frente$.
- A implementação utiliza aritmética modular nos procedimentos $enfileira$ e $desenfileira$ (**%** do Java).

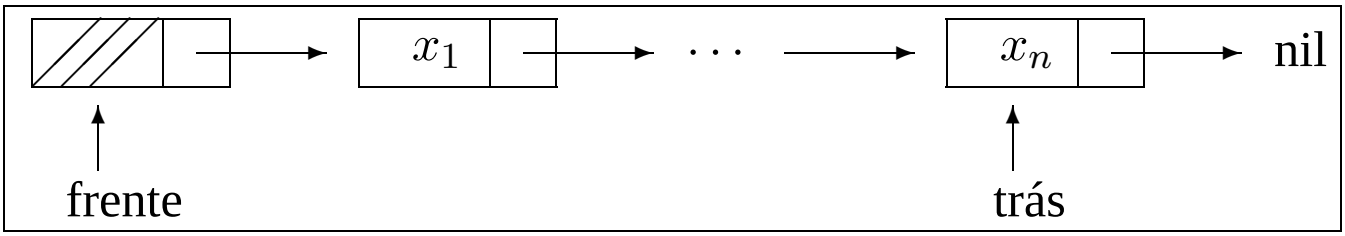
```
package cap3.arranjo;
public class Fila {
  private Object item[];
  private int    frente, tras;
    // Continua na próxima transparência
```

Estrutura e operações sobre Filas Usando arranjo

```
// Operações
  public Fila (int maxTam) { // Cria uma Fila vazia
    this.item = new Object[maxTam];
    this.frente = 0;
    this.tras = this.frente;
  }
  public void enfileira (Object x) throws Exception {
    if ((this.tras + 1) % this.item.length == this.frente)
      throw new Exception ("Erro: A fila esta cheia");
    this.item[this.tras] = x;
    this.tras = (this.tras + 1) % this.item.length;
  }
public Object desenfileira () throws Exception {
    if (this.vazia ())
      throw new Exception ("Erro: A fila esta vazia");
    Object item = this.item[this.frente];
    this.frente = (this.frente + 1) % this.item.length;
    return item;
  }
  public boolean vazia () {
    return (this.frente == this.tras);
  }
}
```

Implementação de Filas por meio de Estruturas Auto-Referenciadas

- Há uma célula cabeça é para facilitar a implementação das operações $enfileira$ e $desenfileira$ quando a fila está vazia.
- Quando a fila está vazia, os apontadores $frente$ e *trás* referenciam para a célula cabeça.
- Para enfileirar um novo item, basta criar uma célula nova, ligá-la após a célula que contém x_n e colocar nela o novo item.
- Para desenfileirar o item x_1, basta desligar a célula cabeça da lista e a célula que contém x_1 passa a ser a célula cabeça.



Estrutura da Fila Usando Estruturas Auto-Referenciadas

- A fila é implementada por meio de células.
- Cada célula contém um item da fila e uma referência para outra célula.
- A classe $Fila$ contém uma referência para a frente da fila (célula cabeça) e uma referência para a parte de trás da fila.

```
package cap3.autoreferencia;
public class Fila {
  private static class Celula { Object item; Celula prox; }
  private Celula frente;
  private Celula tras;
    // Operações
  public Fila () { // Cria uma Fila vazia
    this.frente = new Celula ();
    this.tras = this.frente;
    this.frente.prox = null;
  }
   // Continua na próxima transparência
```

Estrutura Operações da fila usando estruturas auto-referenciadas

```
  public void enfileira (Object x) {
    this.tras.prox = new Celula ();
    this.tras = this.tras.prox;
    this.tras.item = x;
    this.tras.prox = null;
  }
  public Object desenfileira () throws Exception {
    Object item = null;
    if (this.vazia ())
      throw new Exception ("Erro: A fila esta vazia");
    this.frente = this.frente.prox;
    item = this.frente.item;
    return item;
  }

  public boolean vazia () {
    return (this.frente == this.tras);
  }
}
```

Ordenação*

Última alteração: 10 de Outubro de 2006

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

Ordenação

- Introdução - Conceitos Básicos
- Ordenação Interna
  - Ordenação por Seleção
  - Ordenação por Inserção
  - Shellsort
  - Quicksort
  - Heapsort
  - Ordenação Parcial
    - Seleção Parcial
    - Inserção Parcial
    - Heapsort Parcial
    - Quicksort Parcial
- Ordenação Externa
  - Intercalação Balanceada de Vários Caminhos
  - Implementação por meio de Seleção por Substituição
  - Considerações Práticas
  - Intercalação Polifásica
  - Quicksort Externo

Introdução - Conceitos Básicos

- Ordenar: processo de rearranjar um conjunto de objetos em uma ordem ascendente ou descendente.
- A ordenação visa facilitar a recuperação posterior de itens do conjunto ordenado.
  - Dificuldade de se utilizar um catálogo telefônico se os nomes das pessoas não estivessem listados em ordem alfabética.
- A maioria dos métodos de ordenação é baseada em **comparações** das chaves.
- Existem métodos de ordenação que utilizam o princípio da **distribuição**.

Introdução - Conceitos Básicos

- Exemplo de ordenação por distribuição: considere o

  problema de ordenar um baralho com 52 cartas na ordem:

$$\mathsf{A} < 2 < 3 < \cdots < 10 < \mathsf{J} < \mathsf{Q} < \mathsf{K}$$

e

$$\clubsuit < \diamondsuit < \heartsuit < \spadesuit.$$

- Algoritmo:
  1. Distribuir as cartas abertas em treze montes: ases, dois, três, . . ., reis.
  2. Colete os montes na ordem especificada.
  3. Distribua novamente as cartas abertas em quatro montes: paus, ouros, copas e espadas.
  4. Colete os montes na ordem especificada.

Introdução - Conceitos Básicos

- Métodos como o ilustrado são também conhecidos como **ordenação digital**, **radixsort** ou **bucketsort**.
- O método não utiliza comparação entre chaves.
- Uma das dificuldades de implementar este método está relacionada com o problema de lidar com cada monte.
- Se para cada monte nós reservarmos uma área, então a demanda por memória extra pode tornar-se proibitiva.
- O custo para ordenar um arquivo com n elementos é da ordem de $O(n)$.
- Notação utilizada nos algoritmos:
  - Os algoritmos trabalham sobre os registros de um arquivo.
  - Cada registro possui uma **chave** utilizada para controlar a ordenação.
  - Podem existir outros componentes em um registro.

Introdução - Conceitos Básicos

- Qualquer tipo de chave sobre o qual exista uma regra de ordenação bem-definida pode ser utilizado.
- Em Java pode-se criar um tipo de registro genérico chamado **iterface**
- Uma **interface** é uma classe que não pode ser instanciada e deve ser implementada por outra classe.
- A **iterface** para a chave de ordenação chama-se $Item$ inclui os métodos $compara$, $alteraChave$ e $recuperaChave$.

```
package cap4;
public interface Item {
   public int compara (Item it);
   public void alteraChave (Object chave);
   public Object recuperaChave ();
}
```

Introdução - Conceitos Básicos

- A classe $MeuItem$ define o tipo de dados **int** para a chave e implementa os métodos $compara$, $alteraChave$ e $recuperaChave$.
- O método $compara$ retorna um valor menor do que zero se $a < b$, um valor maior do que zero se $a > b$, e um valor igual a zero se $a = b$.

```
package cap4;
import java.io.*;
public class MeuItem implements Item {
  private int chave;
  // outros componentes do registro

  public MeuItem (int chave) { this.chave = chave; }

  public int compara (Item it) {
    MeuItem item = (MeuItem) it;
    if (this.chave < item.chave) return -1;
    else if (this.chave > item.chave) return 1;
    return 0;
  }
// Continua na próxima transparência
```

Introdução - Conceitos Básicos

```
  public void alteraChave (Object chave) {
     Integer ch = (Integer) chave;
     this.chave = ch.intValue ();
  }
  public Object recuperaChave () {
          return new Integer (this.chave); }
}
```

- Deve-se ampliar a interface $Item$ sempre que houver necessidade de manipular a chave de um registro.
- O método $compara$ é **sobrescrito** para determinar como são comparados dois objetos da classe $MeuItem$.
- Os métodos $alteraChave$ e $recuperaChave$ são sobrescritos para determinar como alterar e como recuperar o valor da chave de um objeto da classe $MeuItem$.

Introdução - Conceitos Básicos

- Um método de ordenação é **estável** se a ordem relativa dos itens com chaves iguais não se altera durante a ordenação.
- Alguns dos métodos de ordenação mais eficientes não são estáveis.
- A estabilidade pode ser forçada quando o método é não-estável.
- Sedgewick (1988) sugere agregar um pequeno índice a cada chave antes de ordenar, ou então aumentar a chave de alguma outra forma.

Introdução - Conceitos Básicos

- Classificação dos métodos de ordenação:
  - Ordenação interna: arquivo a ser ordenado cabe todo na memória principal.
  - Ordenação externa: arquivo a ser ordenado não cabe na memória principal.
- Diferenças entre os métodos:
  - Em um método de ordenação interna, qualquer registro pode ser imediatamente acessado.
  - Em um método de ordenação externa, os registros são acessados seqüencialmente ou em grandes blocos.

Ordenação Interna

- Na escolha de um algoritmo de ordenação interna deve ser considerado o tempo gasto pela ordenação.
- Sendo n o número registros no arquivo, as medidas de complexidade relevantes são:
  - Número de comparações $C(n)$ entre chaves.
  - Número de movimentações $M(n)$ de itens do arquivo.
- O uso econômico da memória disponível é um requisito primordial na ordenação interna.
- Métodos de ordenação ***in situ*** são os preferidos.
- Métodos que utilizam listas encadeadas não são muito utilizados.
- Métodos que fazem cópias dos itens a serem ordenados possuem menor importância.

Ordenação Interna

- Classificação dos métodos de ordenação interna:
  - Métodos simples:
    - Adequados para pequenos arquivos.
    - Requerem $O(n^2)$ comparações.
    - Produzem programas pequenos.
  - Métodos eficientes:
    - Adequados para arquivos maiores.
    - Requerem $O(n \log n)$ comparações.
    - Usam menos comparações.
    - As comparações são mais complexas nos detalhes.
    - Métodos simples são mais eficientes para pequenos arquivos.

Ordenação Interna

- A classe mostrada a seguir apresenta os métodos de ordenação interna que serão estudados.
- Utilizaremos um vetor v de registros do tipo $Item$ e uma variável inteira n com o tamanho de v.
- O vetor contém registros nas posições de 1 até n, e a 0 é utilizada para **s**entinelas.

```
package cap4.ordenacaointerna;
import cap4.Item;    // vide transparência 6

public class Ordenacao {
   public static void selecao (Item v[], int n)
   public static void insercao (Item v[], int n)
   public static void shellsort (Item v[], int n)
   public static void quicksort (Item v[], int n)
   public static void heapsort (Item v[], int n)
}
```

Ordenação por Seleção

- Um dos algoritmos mais simples de ordenação.
- Algoritmo:
  - Selecione o menor item do vetor.
  - Troque-o com o item da primeira posição do vetor.
  - Repita essas duas operações com os $n-1$ itens restantes, depois com os $n-2$ itens, até que reste apenas um elemento.
- O método é ilustrado abaixo:

	1	2	3	4	5	6
Chaves iniciais:	*O*	*R*	*D*	*E*	*N*	*A*
i = 1	***A***	*R*	*D*	*E*	*N*	***O***
i = 2	*A*	***D***	***R***	*E*	*N*	*O*
i = 3	*A*	*D*	***E***	***R***	*N*	*O*
i = 4	*A*	*D*	*E*	***N***	***R***	*O*
i = 5	*A*	*D*	*E*	*N*	***O***	***R***

- As chaves em negrito sofreram uma troca entre si.

Ordenação por Seleção

```
public static void selecao (Item v[], int n) {
  for (int i = 1; i <= n - 1; i++) {
    int min = i;
    for (int j = i + 1; j <= n; j++)
      if (v[j].compara (v[min]) < 0) min = j;
    Item x = v[min]; v[min] = v[i]; v[i] = x;
  }
}
```

Análise

- Comparações entre chaves e movimentações de registros:

$$C(n) = \frac{n^2}{2} - \frac{n}{2}$$
$$M(n) = 3(n-1)$$

- A atribuição $min = j$ é executada em média $n \log n$ vezes, Knuth (1973).

Ordenação por Seleção

Vantagens:

- Custo linear no tamanho da entrada para o número de movimentos de registros.
- É o algoritmo a ser utilizado para arquivos com registros muito grandes.
- É muito interessante para arquivos pequenos.

Desvantagens:

- O fato de o arquivo já estar ordenado não ajuda em nada, pois o custo continua quadrático.
- O algoritmo não é **estável**.

Ordenação por Inserção

- Método preferido dos jogadores de **cartas**.
- Algoritmo:
  - Em cada passo a partir de i=2 faça:
    - Selecione o i-ésimo item da seqüência fonte.
    - Coloque-o no lugar apropriado na seqüência destino de acordo com o critério de ordenação.
- O método é ilustrado abaixo:

	1	2	3	4	5	6
Chaves iniciais:	***O***	*R*	*D*	*E*	*N*	*A*
i = 2	***O***	***R***	*D*	*E*	*N*	*A*
i = 3	***D***	***O***	***R***	*E*	*N*	*A*
i = 4	***D***	***E***	***O***	***R***	*N*	*A*
i = 5	***D***	***E***	***N***	***O***	***R***	*A*
i = 6	***A***	***D***	***E***	***N***	***O***	***R***

- As chaves em negrito representam a seqüência destino.

Ordenação por Inserção

```
public static void insercao (Item v[], int n) {
  int j;
  for (int i = 2; i <= n; i++) {
    Item x = v[i];
    j = i – 1;
    v[0] = x; // sentinela
    while (x.compara (v[j]) < 0) {
      v[j + 1] = v[j]; j––;
    }
    v[j + 1] = x;
  }
}
```

- A colocação do item no seu lugar apropriado na seqüência destino é realizada movendo-se itens com chaves maiores para a direita e então inserindo o item na posição deixada vazia.

Ordenação por Inserção

Considerações sobre o algoritmo:

- O processo de ordenação pode ser terminado pelas condições:
  - Um item com chave menor que o item em consideração é encontrado.
  - O final da seqüência destino é atingido à esquerda.
- Solução:
  - Utilizar um registro **sentinela** na posição zero do vetor.

Ordenação por Inserção

Análise

- Seja $C(n)$ a função que conta o número de comparações.
- No anel mais interno, na i-ésima iteração, o valor de C_i é:

$$\begin{aligned}
&\text{melhor caso} && : C_i(n) = 1 \\
&\text{pior caso} && : C_i(n) = i \\
&\text{caso médio} && : C_i(n) = \tfrac{1}{i}(1 + 2 + \cdots + i) = \tfrac{i+1}{2}
\end{aligned}$$

- Assumindo que todas as permutações de n são igualmente prováveis no caso médio, temos:

$$\begin{aligned}
&\text{melhor caso} && : C(n) = (1 + 1 + \cdots + 1) = n - 1 \\
&\text{pior caso} && : C(n) = (2 + 3 + \cdots + n) = \tfrac{n^2}{2} + \\
& && \quad \tfrac{n}{2} - 1 \\
&\text{caso médio} && : C(n) = \tfrac{1}{2}(3 + 4 + \cdots + n + 1) = \\
& && \quad \tfrac{n^2}{4} + \tfrac{3n}{4} - 1
\end{aligned}$$

Ordenação por Inserção

Análise

- Seja $M(n)$ a função que conta o número de movimentações de registros.
- O número de movimentações na i-ésima iteração é:

$$M_i(n) = C_i(n) - 1 + 3 = C_i(n) + 2$$

- Logo, o número de movimentos é:

$$\begin{aligned}
\text{melhor caso} &: M(n) = (3 + 3 + \cdots + 3) = \\
&\quad 3(n-1) \\
\text{pior caso} &: M(n) = (4 + 5 + \cdots + n + 2) = \\
&\quad \frac{n^2}{2} + \frac{5n}{2} - 3 \\
\text{caso médio} &: M(n) = \frac{1}{2}(5 + 6 + \cdots + n + 3) = \\
&\quad \frac{n^2}{4} + \frac{11n}{4} - 3
\end{aligned}$$

Ordenação por Inserção

- O número mínimo de comparações e movimentos ocorre quando os itens estão originalmente em ordem.
- O número máximo ocorre quando os itens estão originalmente na ordem reversa.
- É o método a ser utilizado quando o arquivo está “quase” ordenado.
- É um bom método quando se deseja adicionar uns poucos itens a um arquivo ordenado, pois o custo é linear.
- O algoritmo de ordenação por inserção é **estável**.

Shellsort

- Proposto por Shell em 1959.
- É uma extensão do algoritmo de ordenação por inserção.
- Problema com o algoritmo de ordenação por inserção:
  - Troca itens adjacentes para determinar o ponto de inserção.
  - São efetuadas $n - 1$ comparações e movimentações quando o menor item está na posição mais à direita no vetor.
- O método de Shell contorna este problema permitindo trocas de registros distantes um do outro.

Shellsort

- Os itens separados de h posições são rearranjados.
- Todo h-ésimo item leva a uma seqüência ordenada.
- Tal seqüência é dita estar h-ordenada.
- Exemplo de utilização:

	1	2	3	4	5	6
Chaves iniciais:	*O*	*R*	*D*	*E*	*N*	*A*
h = 4	*N*	*A*	*D*	*E*	*O*	*R*
h = 2	*D*	*A*	*N*	*E*	*O*	*R*
h = 1	*A*	*D*	*E*	*N*	*O*	*R*

- Quando $h = 1$ Shellsort corresponde ao algoritmo de inserção.

Shellsort

- Como escolher o valor de h:
  - Seqüência para h:

$$h(s) = 3h(s-1) + 1, \quad para\ s > 1$$

$$h(s) = 1, \qquad\qquad\qquad para\ s = 1.$$

  - Knuth (1973, p. 95) mostrou experimentalmente que esta seqüência é difícil de ser batida por mais de 20% em eficiência.
  - A seqüência para h corresponde a 1, 4, 13, 40, 121, 364, 1.093, 3.280, . . .

Shellsort

```
public static void shellsort (Item v[], int n) {
  int h = 1;
  do h = h * 3 + 1; while (h < n);
  do {
    h /= 3;
    for (int i = h + 1; i <= n; i++) {
      Item x = v[i]; int j = i;
      while (v[j - h].compara (x) > 0) {
        v[j] = v[j - h]; j -= h;
        if (j <= h) break;
      }
      v[j] = x;
    }
  } while (h != 1);
}
```

- A implementação do Shellsort não utiliza registros **sentinelas**.
- Seriam necessários h registros sentinelas, uma para cada h-ordenação.

Shellsort

Análise

- A razão da eficiência do algoritmo ainda não é conhecida.
- Ninguém ainda foi capaz de analisar o algoritmo.
- A sua análise contém alguns problemas matemáticos muito difíceis.
- A começar pela própria seqüência de incrementos.
- O que se sabe é que cada incremento não deve ser múltiplo do anterior.
- Conjecturas referente ao número de comparações para a seqüência de Knuth:

$$Conjetura\ 1 \quad : C(n) = O(n^{1,25})$$

$$Conjetura\ 2 \quad : C(n) = O(n(\ln n)^2)$$

Shellsort

- Vantagens:
  - $Shellsort$ é uma ótima opção para arquivos de tamanho moderado.
  - Sua implementação é simples e requer uma quantidade de código pequena.
- Desvantagens:
  - O tempo de execução do algoritmo é sensível à ordem inicial do arquivo.
  - O método não é **estável**,

Quicksort

- Proposto por Hoare em 1960 e publiccado em 1962.
- É o algoritmo de ordenação interna mais rápido que se conhece para uma ampla variedade de situações.
- Provavelmente é o mais utilizado.
- A idéia básica é dividir o problema de ordenar um conjunto com n itens em dois problemas menores.
- Os problemas menores são ordenados independentemente.
- Os resultados são combinados para produzir a solução final.

Quicksort

- A parte mais delicada do método é relativa ao método $particao$.
- O vetor $v[esq..dir]$ é rearranjado por meio da escolha arbitrária de um **pivô** x.
- O vetor v é particionado em duas partes:
  - A parte esquerda com chaves menores ou iguais a x.
  - A parte direita com chaves maiores ou iguais a x.

Quicksort

- Algoritmo para o particionamento:
  1. Escolha arbitrariamente um **pivô** x.
  2. Percorra o vetor a partir da esquerda até que $v[i] \geq x$.
  3. Percorra o vetor a partir da direita até que $v[j] \leq x$.
  4. Troque $v[i]$ com $v[j]$.
  5. Continue este processo até os apontadores i e j se cruzarem.
- Ao final, o vetor $v[esq..dir]$ está particionado de tal forma que:
  - Os itens em $v[esq], v[esq+1], \ldots, v[j]$ são menores ou iguais a x.
  - Os itens em $v[i], v[i+1], \ldots, v[dir]$ são maiores ou iguais a x.

Quicksort

- Ilustração do processo de partição:

1	2	3	4	5	6
O	*R*	***D***	*E*	*N*	*A*
A	*R*	***D***	*E*	*N*	*O*
A	***D***	*R*	*E*	*N*	*O*

- O pivô x é escolhido como sendo $v[(i + j) \,/\, 2]$.
- Como inicialmente $i = 1$ e $j = 6$, então $x = v[3] = D$.
- Ao final do processo de partição i e j se cruzam em $i = 3$ e $j = 2$.

Quicksort

Método Partição:

```
private static class LimiteParticoes { int i; int j; }
private static LimiteParticoes particao
                              (Item v[], int esq, int dir) {
  LimiteParticoes p = new LimiteParticoes ();
  p.i = esq;  p.j = dir;
  Item x = v[(p.i + p.j) / 2]; // obtém o pivo x
  do {
    while (x.compara (v[p.i]) > 0) p.i++;
    while (x.compara (v[p.j]) < 0) p.j--;
    if (p.i <= p.j) {
      Item w = v[p.i]; v[p.i] = v[p.j]; v[p.j] = w;
      p.i++; p.j--;
    }
  } while (p.i <= p.j);
  return p;
}
```

- O modificador **private** nessa classe encapsula os métodos para serem utilizados somente dentro da classe $Ordenacao$.
- O anel interno do procedimento Particao é extremamente simples.
- Razão pela qual o algoritmo $Quicksort$ é tão rápido.

Quicksort

Método $ordena$ e algoritmo $Quicksort$:

```
private static void ordena (Item v[], int esq, int dir) {
  LimiteParticoes p = particao (v, esq, dir);
  if (esq < p.j) ordena (v, esq, p.j);
  if (p.i < dir) ordena (v, p.i, dir);
}

public static void quicksort (Item v[], int n) {
        ordena (v, 1, n);
        }
```

Quicksort

- Exemplo do estado do vetor em cada chamada recursiva do procedimento Ordena:

Chaves iniciais:	*O*	*R*	*D*	*E*	*N*	*A*
1	*A*	***D***	*R*	*E*	*N*	*O*
2	***A***	*D*				
3			***E***	*R*	*N*	*O*
4				***N***	*R*	*O*
5					*O*	***R***
	A	*D*	*E*	*N*	*O*	*R*

- O **pivô** é mostrado em negrito.

Quicksort

Análise

- Seja $C(n)$ a função que conta o número de comparações.
- Pior caso:

$$C(n) = O(n^2)$$

- O pior caso ocorre quando, sistematicamente, o pivô é escolhido como sendo um dos extremos de um arquivo já ordenado.
- Isto faz com que o procedimento Ordena seja chamado recursivamente n vezes, eliminando apenas um item em cada chamada.
- O pior caso pode ser evitado empregando pequenas modificações no algoritmo.
- Para isso basta escolher três itens quaisquer do vetor e usar a **mediana dos três** como pivô.

Quicksort

Análise

- Melhor caso:

$$C(n) = 2C(n/2) + n = n \log n - n + 1$$

- Esta situação ocorre quando cada partição divide o arquivo em duas partes iguais.
- Caso médio de acordo com Sedgewick e Flajolet (1996, p. 17):

$$C(n) \approx 1,386n \log n - 0,846n,$$

- Isso significa que em média o tempo de execução do $Quicksort$ é $O(n \log n)$.

Quicksort

- Vantagens:
  - É extremamente eficiente para ordenar arquivos de dados.
  - Necessita de apenas uma pequena pilha como memória auxiliar.
  - Requer cerca de $n \log n$ comparações em média para ordenar n itens.
- Desvantagens:
  - Tem um pior caso $O(n^2)$ comparações.
  - Sua implementação é muito delicada e difícil:
    - Um pequeno engano pode levar a efeitos inesperados para algumas entradas de dados.
  - O método não é **estável**.

Heapsort

- Possui o mesmo princípio de funcionamento da ordenação por seleção.
- Algoritmo:
  1. Selecione o menor item do vetor.
  2. Troque-o com o item da primeira posição do vetor.
  3. Repita estas operações com os $n-1$ itens restantes, depois com os $n-2$ itens, e assim sucessivamente.
- O custo para encontrar o menor (ou o maior) item entre n itens é $n-1$ comparações.
- Isso pode ser reduzido utilizando uma fila de prioridades.

Heapsort

Filas de Prioridades

- É uma estrutura de dados onde a chave de cada item reflete sua habilidade relativa de abandonar o conjunto de itens rapidamente.
- Aplicações:
  - SOs usam filas de prioridades, nas quais as chaves representam o tempo em que eventos devem ocorrer.
  - Métodos numéricos iterativos são baseados na seleção repetida de um item com maior (menor) valor.
  - Sistemas de gerência de memória usam a técnica de substituir a página menos utilizada na memória principal por uma nova página.

Heapsort

Filas de Prioridades - Tipo Abstrato de Dados

- Operações:
  1. Constrói uma fila de prioridades a partir de um conjunto com n itens.
  2. Informa qual é o maior item do conjunto.
  3. Retira o item com maior chave.
  4. Insere um novo item.
  5. Aumenta o valor da chave do item i para um novo valor que é maior que o valor atual da chave.
  6. Substitui o maior item por um novo item, a não ser que o novo item seja maior.
  7. Altera a prioridade de um item.
  8. Remove um item qualquer.
  9. Ajunta duas filas de prioridades em uma única.

Heapsort

Filas de Prioridades - Representação

- Representação através de uma lista linear ordenada:
  - Neste caso, Constrói leva tempo $O(n \log n)$.
  - Insere é $O(n)$.
  - Retira é $O(1)$.
  - Ajunta é $O(n)$.
- Representação é através de uma lista linear não ordenada:
  - Neste caso, Constrói tem custo linear.
  - Insere é $O(1)$.
  - Retira é $O(n)$.
  - Ajunta é $O(1)$ para apontadores e $O(n)$ para arranjos.

Heapsort

Filas de Prioridades - Representação

- A melhor representação é através de uma estruturas de dados chamada *heap*:
  - Neste caso, Constrói é $O(n)$.
  - Insere, Retira, Substitui e Altera são $O(\log n)$.
- **Observação**:
  Para implementar a operação Ajunta de forma eficiente e ainda preservar um custo logarítmico para as operações Insere, Retira, Substitui e Altera é necessário utilizar estruturas de dados mais sofisticadas, tais como árvores binomiais (Vuillemin, 1978).

Heapsort

Filas de Prioridades - Algoritmos de Ordenação

- As operações das filas de prioridades podem ser utilizadas para implementar algoritmos de ordenação.
- Basta utilizar repetidamente a operação Insere para construir a fila de prioridades.
- Em seguida, utilizar repetidamente a operação Retira para receber os itens na ordem reversa.
- O uso de listas lineares não ordenadas corresponde ao método da seleção.
- O uso de listas lineares ordenadas corresponde ao método da inserção.
- O uso de *heaps* corresponde ao método $Heapsort$.

Heapsort

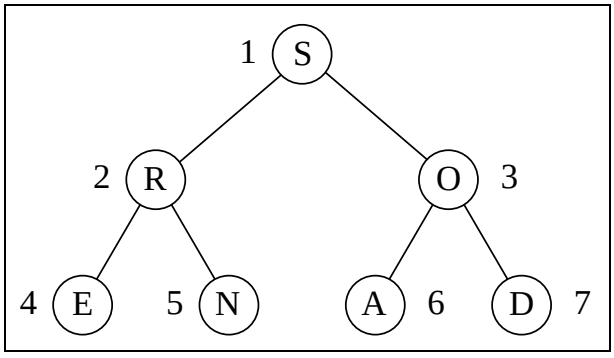
Heaps

- É uma seqüência de itens com chaves $c[1], c[2], \ldots, c[n]$, tal que:

$$c[i] \geq c[2i],$$
$$c[i] \geq c[2i+1],$$

  para todo $i = 1, 2, \ldots, n/2$.

- A definição pode ser facilmente visualizada em uma árvore binária completa:



- **árvore binária completa**:
  - Os nós são numerados de 1 a n.
  - O primeiro nó é chamado raiz.
  - O nó $\lfloor k/2 \rfloor$ é o pai do nó k, para $1 < k \leq n$.
  - Os nós $2k$ e $2k+1$ são os filhos à esquerda e à direita do nó k, para $1 \leq k \leq \lfloor k/2 \rfloor$.

Heapsort

Heaps

- As chaves na árvore satisfazem a condição do *heap*.
- As chaves em cada nó s ao maiores do que as chaves em seus filhos.
- A chave no nó raiz é a maior chave do conjunto.
- Uma árvore binária completa pode ser representada por um arranjo:

1	2	3	4	5	6	7
S	*R*	*O*	*E*	*N*	*A*	*D*

- A representação é extremamente compacta.
- Permite caminhar pelos nós da árvore facilmente.
- Os filhos de um nó i estão nas posições $2i$ e $2i + 1$.
- O pai de um nó i está na posição $i \,/\, 2$.

Heapsort

Heaps

- Na representação do *heap* em um arranjo, a maior chave está sempre na posição 1 do vetor.
- Os algoritmos para implementar as operações sobre o *heap* operam ao longo de um dos caminhos da árvore.
- Um algoritmo elegante para construir o *heap* foi proposto por Floyd em 1964.
- O algoritmo não necessita de nenhuma memória auxiliar.
- Dado um vetor $v[1], v[2], \ldots, v[n]$.
- Os itens $v[n/2+1], v[n/2+2], \ldots, v[n]$ formam um *heap*:
  - Neste intervalo não existem dois índices i e j tais que $j = 2i$ ou $j = 2i+1$.

Heapsort

Estrutura de dados fila de prioridades implementada utilizando um *heap*

```
package cap4.ordenacaointerna;
import cap4.Item; // vide transparência 6
public class FPHeapMax {
   private Item v[];
   private int  n;
   public FPHeapMax (Item v[]) {
      this.v = v; this.n = this.v.length – 1;
   }
   public void refaz (int esq, int dir)
   public void constroi ()
   public Item max ()
   public Item retiraMax () throws Exception
   public void aumentaChave (int i, Object chaveNova)
                        throws Exception
   public void insere (Item x) throws Exception
}
```

Heapsort

Heaps

- Algoritmo:

	1	2	3	4	5	6	7
Chaves iniciais:	*O*	*R*	*D*	*E*	*N*	*A*	*S*
Esq = 3	*O*	*R*	***S***	*E*	*N*	*A*	***D***
Esq = 2	*O*	*R*	*S*	*E*	*N*	*A*	*D*
Esq = 1	***S***	*R*	***O***	*E*	*N*	*A*	*D*

- Os itens de $v[4]$ a $v[7]$ formam um *heap*.
- O *heap* é estendido para a esquerda $(esq = 3)$, englobando o item $v[3]$, pai dos itens $v[6]$ e $v[7]$.
- A condição de *heap* é violada:
  - O *heap* é refeito trocando os itens D e S.
- O item R é incluindo no *heap* $(esq = 2)$, o que não viola a condição de *heap*.
- O item O é incluindo no *heap* $(esq = 1)$.
- A Condição de *heap* violada:
  - O *heap* é refeito trocando os itens O e S, encerrando o processo.

Heapsort

Heaps

- O Programa que implementa a operação que informa o item com maior chave:

```
public Item max () {
        return this.v[1];
        }
```

- Método para refazer o *heap*:

```
public void refaz (int esq, int dir) {
  int j = esq * 2; Item x = this.v[esq];
  while (j <= dir) {
    if ((j < dir) && (this.v[j].compara (this.v[j + 1]) < 0))
          j++;
    if (x.compara (this.v[j]) >= 0) break;
    this.v[esq] = this.v[j];
    esq = j; j = esq * 2;
  }
  this.v[esq] = x;
}
```

Heapsort

Heaps

Método para construir o *heap*:

```
// Usa o método refaz da transparência 49
public void constroi() {
   int esq = n / 2 + 1;
   while (esq > 1) {
      esq--;
      this.refaz(esq, this.n);
   }
}
```

Heapsort

Heaps

Programa que implementa a operação de retirar o item com maior chave:

```
// Usa o método refaz da transparência 49
public Item retiraMax () throws Exception {
  Item maximo;
  if (this.n < 1) throw new Exception ("Erro: heap vazio");
  else {
    maximo = this.v[1];
    this.v[1] = this.v[this.n--];
    refaz (1, this.n);
  }
  return maximo;
}
```

Heapsort

Heaps

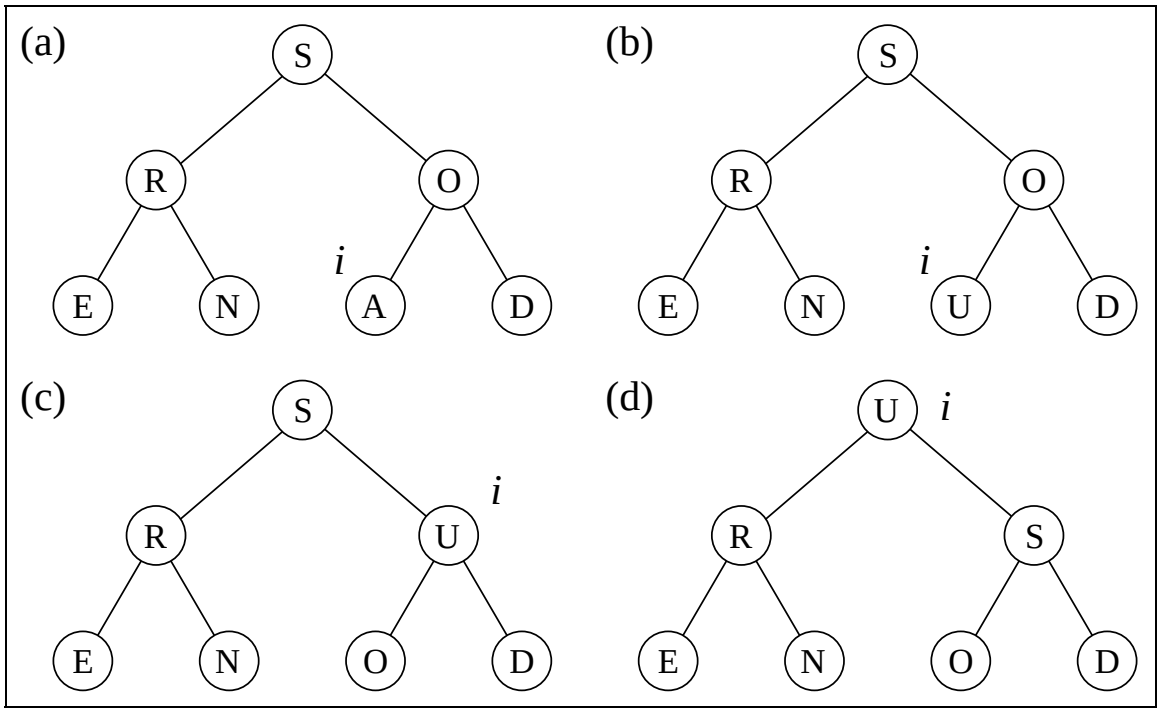
Programa que implementa a operação de aumentar o valor da chave do item i:

```
public void aumentaChave (int i, Object chaveNova)
                                              throws Exception {
  Item x = this.v[i];
  if (chaveNova == null)
    throw new Exception ("Erro: chaveNova com valor null");
  x.alteraChave (chaveNova);
  while ((i > 1) && (x.compara (this.v[i / 2]) >= 0)) {
    this.v[i] = this.v[i / 2]; i /= 2;
  }
  this.v[i] = x;
}
```

Heapsort

Heaps

- Exemplo da operação de aumentar o valor da chave do item na posição i:



- O tempo de execução do procedimento AumentaChave em um item do *heap* é $O(\log n)$.

Heapsort

Heaps

Programa que implementa a operação de inserir um novo item no *heap*:

```
// Usa o método aumentaChave da tranparência 52
public void insere (Item x) throws Exception {
   this.n++;
   if (this.n == this.v.length) throw new Exception
            ("Erro: heap cheio");
   Object chaveNova = x.recuperaChave ();
   this.v[this.n] = x;
   this.v[this.n].alteraChave (new Integer
                  (Integer.MIN_VALUE)); //−∞
   this.aumentaChave (this.n, chaveNova);
}
```

Heapsort

- Algoritmo:
  1. Construir o *heap*.
  2. Troque o item na posição 1 do vetor (raiz do *heap*) com o item da posição n.
  3. Use o procedimento Refaz para reconstituir o *heap* para os itens $v[1]$, $v[2], \ldots, v[n-1]$.
  4. Repita os passos 2 e 3 com os $n-1$ itens restantes, depois com os $n-2$, até que reste apenas um item.

Heapsort

- Exemplo de aplicação do $Heapsort$:

1	2	3	4	5	6	7
S	*R*	*O*	*E*	*N*	*A*	*D*
R	***N***	*O*	*E*	***D***	*A*	***S***
O	*N*	***A***	*E*	*D*	*R*	
N	***E***	*A*	***D***	*O*		
E	***D***	*A*	*N*			
D	***A***	*E*				
A	*D*					

- O caminho seguido pelo procedimento Refaz para reconstituir a condição do *heap* está em negrito.
- Por exemplo, após a troca dos itens S e D na segunda linha da Figura, o item D volta para a posição 5, após passar pelas posições 1 e 2.

Heapsort

Programa que mostra a implementação do $Heapsort$ para um conjunto de n itens implementado como um vetor do tipo $Item$:

```
public static void heapsort (Item v[], int n) {
  // Usa a classe FPHeapMax da transparência 47
  FPHeapMax fpHeap = new FPHeapMax (v);
  int dir = n;
  fpHeap.constroi (); // constroi o heap
  while (dir > 1) { // ordena o vetor
    Item x = v[1];
    v[1] = v[dir];
    v[dir] = x;
    dir--;
    fpHeap.refaz (1, dir);
  }
}
```

Análise

- O procedimento Refaz gasta cerca de $\log n$ operações, no pior caso.
- Logo, $Heapsort$ gasta um tempo de execução proporcional a $n \log n$, no pior caso.

Heapsort

- Vantagens:
  - O comportamento do $Heapsort$ é sempre $O(n \log n)$, qualquer que seja a entrada.
- Desvantagens:
  - O anel interno do algoritmo é bastante complexo se comparado com o do $Quicksort$.
  - O $Heapsort$ não é **estável**.
- Recomendado:
  - Para aplicações que não podem tolerar eventualmente um caso desfavorável.
  - Não é recomendado para arquivos com poucos registros, por causa do tempo necessário para construir o *heap*.

Comparação entre os Métodos

Complexidade:

	Complexidade
Inserção	$O(n^2)$
Seleção	$O(n^2)$
Shellsort	$O(n \log n)$
Quicksort	$O(n \log n)$
Heapsort	$O(n \log n)$

- Apesar de não se conhecer analiticamente o comportamento do $Shellsort$, ele é considerado um método eficiente.

Comparação entre os Métodos

Tempo de execução:

- Oservação: O método que levou menos tempo real para executar recebeu o valor 1 e os outros receberam valores relativos a ele.
- Registros na ordem aleatória:

	5.00	5.000	10.000	30.000
Inserção	11,3	87	161	–
Seleção	16,2	124	228	–
Shellsort	1,2	1,6	1,7	2
Quicksort	1	1	1	1
Heapsort	1,5	1,6	1,6	1,6

- Registros na ordem ascendente:

	500	5.000	10.000	30.000
Inserção	1	1	1	1
Seleção	128	1.524	3.066	–
Shellsort	3,9	6,8	7,3	8,1
Quicksort	4,1	6,3	6,8	7,1
Heapsort	12,2	20,8	22,4	24,6

Comparação entre os Métodos

Tempo de execução:

- Registros na ordem descendente:

	500	5.000	10.000	30.000
Inserção	40,3	305	575	–
Seleção	29,3	221	417	–
Shellsort	1,5	1,5	1,6	1,6
Quicksort	1	1	1	1
Heapsort	2,5	2,7	2,7	2,9

Comparação entre os Métodos

Observações sobre os métodos:

1. $Shellsort$, $Quicksort$ e $Heapsort$ têm a mesma ordem de grandeza.
2. O $Quicksort$ é o mais rápido para todos os tamanhos aleatórios experimentados.
3. A relação $Heapsort/Quicksort$ se mantém constante para todos os tamanhos, sendo o $Heapsort$ mais lento.
4. A relação $Shellsort/Quicksort$ aumenta à medida que o número de elementos aumenta; para arquivos pequenos (500 elementos), o $Shellsort$ é mais rápido que o $Heapsort$; porém, quando o tamanho da entrada cresce, essa relação se inverte.
5. O método da Inserção é o mais rápido para qualquer tamanho se os elementos estão ordenados; Ele é o mais lento para qualquer tamanho se os elementos estão em ordem descendente;
6. Entre os algoritmos de custo $O(n^2)$, o Inserção é melhor para todos os tamanhos aleatórios experimentados.

Comparação entre os Métodos

Influência da ordem inicial do registros:

	Shellsort			Quicksort			Heapsort		
	5.000	10.000	30.000	5.000	10.000	30.000	5.000	10.000	30.000
Asc	1	1	1	1	1	1	1,1	1,1	1,1
Des	1,5	1,6	1,5	1,1	1,1	1,1	1	1	1
Ale	2,9	3,1	3,7	1,9	2,0	2,0	1,1	1	1

1. O $Shellsort$ é bastante sensível à ordenação ascendente ou descendente da entrada;
2. Em arquivos do mesmo tamanho, o $Shellsort$ executa mais rápido para arquivos ordenados.
3. O $Quicksort$ é sensível à ordenação ascendente ou descendente da entrada.
4. Em arquivos do mesmo tamanho, o $Quicksort$ executa mais rápido para arquivos ordenados.
5. O $Quicksort$ é o mais rápido para qualquer tamanho para arquivos na ordem ascendente.
6. O $Heapsort$ praticamente não é sensível à ordenação da entrada.

Comparação entre os Métodos

Método da Inserção:

- É o mais interessante para arquivos com menos do que 20 elementos.
- O método é estável.
- Possui comportamento melhor do que o método da **bolha (**$Bubblesort$**)** que também é estável.
- Sua implementação é tão simples quanto as implementações do $Bubblesort$ e Seleção.
- Para arquivos já ordenados, o método é $O(n)$.
- O custo é linear para adicionar alguns elementos a um arquivo já ordenado.

Comparação entre os Métodos

Método da Seleção:

- É vantajoso quanto ao número de movimentos de registros, que é $O(n)$.
- Deve ser usado para arquivos com registros muito grandes, desde que o tamanho do arquivo não exceda 1.000 elementos.

Comparação entre os Métodos

Shellsort:

- É o método a ser escolhido para a maioria das aplicações por ser muito eficiente para arquivos de tamanho moderado.
- Mesmo para arquivos grandes, o método é cerca de apenas duas vezes mais lento do que o $Quicksort$.
- Sua implementação é simples e geralmente resulta em um programa pequeno.
- Não possui um pior caso ruim e quando encontra um arquivo parcialmente ordenado trabalha menos.

Comparação entre os Métodos

Quicksort:

- É o algoritmo mais eficiente que existe para uma grande variedade de situações.
- É um método bastante frágil no sentido de que qualquer erro de implementação pode ser difícil de ser detectado.
- O algoritmo é recursivo, o que demanda uma pequena quantidade de memória adicional.
- Seu desempenho é da ordem de $O(n^2)$ operações no pior caso.
- O principal cuidado a ser tomado é com relação à escolha do pivô.
- A escolha do elemento do meio do arranjo melhora muito o desempenho quando o arquivo está total ou parcialmente ordenado.
- O pior caso tem uma probabilidade muito remota de ocorrer quando os elementos forem aleatórios.

Comparação entre os Métodos

Quicksort:

- Geralmente se usa a mediana de uma amostra de três elementos para evitar o pior caso.
- Esta solução melhora o caso médio ligeiramente.
- Outra importante melhoria para o desempenho do $Quicksort$ é evitar chamadas recursivas para pequenos subarquivos.
- Para isto, basta chamar um método de ordenação simples nos arquivos pequenos.
- A melhoria no desempenho é significativa, podendo chegar a 20% para a maioria das aplicações (Sedgewick, 1988).

Comparação entre os Métodos

Heapsort:

- É um método de ordenação elegante e eficiente.
- Apesar de ser cerca de duas vezes mais lento do que o $Quicksort$, não necessita de nenhuma memória adicional.
- Executa sempre em tempo proporcional a $n \log n$,
- Aplicações que não podem tolerar eventuais variações no tempo esperado de execução devem usar o $Heapsort$.

Comparação entre os Métodos

Considerações finais:

- Para registros muito grandes é desejável que o método de ordenação realize apenas n movimentos dos registros.
- Com o uso de uma **ordenação indireta** é possível se conseguir isso.
- Suponha que o arquivo A contenha os seguintes registros: $v[1], v[2], \ldots, v[n]$.
- Seja p um arranjo $p[1], p[2], \ldots, p[n]$ de apontadores.
- Os registros somente são acessados para fins de comparações e toda movimentação é realizada sobre os apontadores.
- Ao final, $p[1]$ contém o índice do menor elemento de v, $p[2]$ o índice do segundo menor e assim sucessivamente.
- Essa estratégia pode ser utilizada para qualquer dos métodos de ordenação interna.

Ordenação Parcial

- Consiste em obter os k primeiros elementos de um vetor ordenado com n elementos.
- Quando $k = 1$, o problema se reduz a encontrar o mínimo (ou o máximo) de um conjunto de elementos.
- Quando $k = n$ caímos no problema clássico de ordenação.

Ordenação Parcial

Aplicações:

- Facilitar a busca de informação na Web com as **máquinas de busca**:
  - É comum uma consulta na Web retornar centenas de milhares de documentos relacionados com a consulta.
  - O usuário está interessado apenas nos k documentos mais relevantes.
  - Em geral k é menor do que 200 documentos.
  - Normalmente são consultados apenas os dez primeiros.
  - Assim, são necessários algoritmos eficientes de ordenação parcial.

Ordenação Parcial

Algoritmos considerados:

- Seleção parcial.
- Inserção parcial.
- $Heapsort$ parcial.
- $Quicksort$ parcial.
- A classe $OrdenacaoParcial$ é mostrada a seguir.

```
package cap4.ordenacaointerna;
import cap4.Item;    // vide transparência 5

public class OrdenacaoParcial {
   public static void selecaoParcial(Item v[], int n, int k)
   public static void insercaoParcial(Item v[], int n, int k)
   public static void insercaoParcial2(Item V[], int n, int k)
   public static void quicksortParcial(Item v[], int n, int k)
   public static void heapsortParcial(Item v[], int n, int k)
}
```

Seleção Parcial

- Um dos algoritmos mais simples.
- Princípio de funcionamento:
  - Selecione o menor item do vetor.
  - Troque-o com o item que está na primeira posição do vetor.
  - Repita estas duas operações com os itens $n-1, n-2 \ldots n-k$.

Seleção Parcial

```
public static void selecaoParcial (Item v[], int n, int k){
  for (int i = 1; i <= k; i++) {
    int min = i;
    for (int j = i + 1; j <= n; j++)
      if (v[j].compara (v[min]) < 0) min = j;
    Item x = v[min]; v[min] = v[i]; v[i] = x;
  }
}
```

Análise:

- Comparações entre chaves e movimentações de registros:

$$C(n) = kn - \frac{k^2}{2} - \frac{k}{2}$$

$$M(n) = 3k$$

Seleção Parcial

- É muito simples de ser obtido a partir da implementação do algoritmo de ordenação por seleção.
- Possui um comportamento espetacular quanto ao número de movimentos de registros:
  - Tempo de execução é linear no tamanho de k.

Inserção Parcial

- Pode ser obtido a partir do algoritmo de ordenação por Inserção por meio de uma modificação simples:
  - Tendo sido ordenados os primeiros k itens, o item da k-ésima posição funciona como um pivô.
  - Quando um item entre os restantes é menor do que o pivô, ele é inserido na posição correta entre os k itens de acordo com o algoritmo original.

Inserção Parcial

```
public static void insercaoParcial (Item v[], int n, int k){
  int j;
  for (int i = 2; i <= n; i++) {
    Item x = v[i];
    if (i > k) j = k; else j = i − 1;
    v[0] = x; // sentinela
    while (x.compara (v[j]) < 0) {
      v[j + 1] = v[j]; j−−;
    }
    v[j + 1] = x;
  }
}
```

Obs:

1. A modificação realizada verifica o momento em que i se torna maior do que k e então passa a considerar o valor de j igual a k a partir deste ponto.
2. O algoritmo não preserva o restante do vetor.

Inserção Parcial

Algoritmo de Inserção Parcial que preserva o restante do vetor:

```
public static void insercaoParcial2 (Item v[], int n, int k){
  int j;
  for (int i = 2; i <= n; i++) {
    Item x = v[i];
    if (i > k) {
      j = k;
      if (x.compara (v[k]) < 0) v[i] = v[k];
    }
    else j = i – 1;
    v[0] = x; // sentinela
    while (x.compara (v[j]) < 0) {
      if (j < k) v[j + 1] = v[j];
      j––;
    }
    if (j < k) v[j + 1] = x;
  }
}
```

Inserção Parcial

Análise:

- No anel mais interno, na i-ésima iteração o valor de C_i é:

$$\begin{aligned}
&\text{melhor caso} && : C_i(n) = 1 \\
&\text{pior caso} && : C_i(n) = i \\
&\text{caso médio} && : C_i(n) = \tfrac{1}{i}(1 + 2 + \cdots + i) = \tfrac{i+1}{2}
\end{aligned}$$

- Assumindo que todas as permutações de n são igualmente prováveis, o número de comparações é:

$$\begin{aligned}
&\text{melhor caso} && : C(n) && = (1 + 1 + \cdots + 1) = n - 1 \\
&\text{pior caso} && : C(n) && = (2 + 3 + \cdots + k + (k+1)(n-k)) \\
& && && = kn + n - \tfrac{k^2}{2} - \tfrac{k}{2} - 1 \\
&\text{caso médio} && : C(n) && = \tfrac{1}{2}(3 + 4 + \cdots + k + 1 + (k+1)(n-k)) \\
& && && = \tfrac{kn}{2} + \tfrac{n}{2} - \tfrac{k^2}{4} + \tfrac{k}{4} - 1
\end{aligned}$$

Inserção Parcial

Análise:

- O número de movimentações na i-ésima iteração é:

$$M_i(n) = C_i(n) - 1 + 3 = C_i(n) + 2$$

- Logo, o número de movimentos é:

$$\begin{array}{lll}
\text{melhor caso} & : M(n) & = (3 + 3 + \cdots + 3) = 3(n-1) \\
\text{pior caso} & : M(n) & = (4 + 5 + \cdots + k + 2 + (k+1)(n-k)) \\
 & & = kn + n - \frac{k^2}{2} + \frac{3k}{2} - 3 \\
\text{caso médio} & : M(n) & = \frac{1}{2}(5 + 6 + \cdots + k + 3 + (k+1)(n-k)) \\
 & & = \frac{kn}{2} + \frac{n}{2} - \frac{k^2}{4} + \frac{5k}{4} - 2
\end{array}$$

- O número mínimo de comparações e movimentos ocorre quando os itens estão originalmente em ordem.

- O número máximo ocorre quando os itens estão originalmente na ordem reversa.

Heapsort Parcial

- Utiliza um tipo abstrato de dados *heap* para informar o menor item do conjunto.
- Na primeira iteração, o menor item que está em $v[1]$

  (raiz do *heap*) é trocado com o item que está em $v[n]$.
- Em seguida o *heap* é refeito.
- Novamente, o menor está em A[1], troque-o com A[n-1].
- Repita as duas últimas operações até que o k-ésimo menor seja trocado com $v[n-k]$.
- Ao final, os k menores estão nas k últimas posições do vetor v.

Heapsort Parcial

```
public static void heapsortParcial (Item v[], int n, int k){
// Coloca menor em v[n], segundo em v[n-1],...,k-esimo em v[n-k]
  FPHeapMin fpHeap = new FPHeapMin (v);
  int dir = n, aux = 0;
  fpHeap.constroi (); // constroi o heap
  while (aux < k) { // ordena o vetor
    Item x = v[1]; v[1] = v[dir]; v[dir] = x;
    dir--; aux++;
    fpHeap.refaz (1, dir);
  }
}
```

Heapsort Parcial

Análise:

- O $Heapsort$ Parcial deve construir um *heap* a um custo $O(n)$.
- O método $refaz$ tem custo $O(\log n)$.
- O método $heapsortParcial$ chama o método $refaz\ k$ vezes.
- Logo, o algoritmo apresenta a complexidade:

$$O(n + k\log n) = \begin{cases} O(n) & \text{se } k \leq \frac{n}{\log n} \\ O(k\log n) & \text{se } k > \frac{n}{\log n} \end{cases}$$

Quicksort Parcial

- Assim como o Quicksort, o Quicksort Parcial é o algoritmo de ordenação parcial mais rápido em várias situações.
- A alteração no algoritmo para que ele ordene apenas os k primeiros itens dentre n itens é muito simples.
- Basta abandonar a partição à direita toda vez que a partição à esquerda contiver k ou mais itens.
- Assim, a única alteração necessária no Quicksort é evitar a chamada recursiva $ordena(i, dir)$.

Quicksort Parcial

Chaves iniciais:	*O*	*R*	*D*	*E*	*N*	*A*
1	*A*	***D***	*R*	*E*	*N*	*O*
2	***A***	*D*				
3			***E***	*R*	*N*	*O*
4				***N***	*R*	*O*
5					*O*	***R***
	A	*D*	*E*	*N*	*O*	*R*

- Considere $k = 3$ e D o pivô para gerar as linhas 2 e 3.
- A partição à esquerda contém dois itens e a partição à direita contém quatro itens.
- A partição à esquerda contém menos do que k itens.
- Logo, a partição direita não pode ser abandonada.
- Considere E o pivô na linha 3.
- A partição à esquerda contém três itens e a partição à direita também.
- Assim, a partição à direita pode ser abandonada.

Quicksort Parcial

```
private static void ordena(Item v[],int esq,int dir,int k){
  LimiteParticoes p = particao (v, esq, dir);
  if (p.j – esq >= k – 1) {
    if (esq < p.j) ordena (v, esq, p.j, k);
    return;
  }
  if (esq < p.j) ordena (v, esq, p.j, k);
  if (p.i < dir) ordena (v, p.i, dir, k);
}
public static void quicksortParcial(Item v[],int n,int k){
  ordena (v, 1, n, k);
}
```

Quicksort Parcial

Análise:

- A análise do Quicksort é difícil.
- O comportamento é muito sensível à escolha do pivô.
- Podendo cair no melhor caso $O(k \log k)$.
- Ou em algum valor entre o melhor caso e $O(n \log n)$.

Comparação entre os Métodos de Ordenação Parcial

n, k	Seleção	Quicksort	Inserção	Inserção2	Heapsort
$n : 10^1 \quad k : 10^0$	1	2,5	1	1,2	1,7
$n : 10^1 \quad k : 10^1$	1,2	2,8	1	1,1	2,8
$n : 10^2 \quad k : 10^0$	1	3	1,1	1,4	4,5
$n : 10^2 \quad k : 10^1$	1,9	2,4	1	1,2	3
$n : 10^2 \quad k : 10^2$	3	1,7	1	1,1	2,3
$n : 10^3 \quad k : 10^0$	1	3,7	1,4	1,6	9,1
$n : 10^3 \quad k : 10^1$	4,6	2,9	1	1,2	6,4
$n : 10^3 \quad k : 10^2$	11,2	1,3	1	1,4	1,9
$n : 10^3 \quad k : 10^3$	15,1	1	3,9	4,2	1,6
$n : 10^5 \quad k : 10^0$	1	2,4	1,1	1,1	5,3
$n : 10^5 \quad k : 10^1$	5,9	2,2	1	1	4,9
$n : 10^5 \quad k : 10^2$	67	2,1	1	1,1	4,8
$n : 10^5 \quad k : 10^3$	304	1	1,1	1,3	2,3
$n : 10^5 \quad k : 10^4$	1445	1	33,1	43,3	1,7
$n : 10^5 \quad k : 10^5$	∞	1	∞	∞	1,9
$n : 10^6 \quad k : 10^0$	1	3,9	1,2	1,3	8,1
$n : 10^6 \quad k : 10^1$	6,6	2,7	1	1	7,3
$n : 10^6 \quad k : 10^2$	83,1	3,2	1	1,1	6,6
$n : 10^6 \quad k : 10^3$	690	2,2	1	1,1	5,7
$n : 10^6 \quad k : 10^4$	∞	1	5	6,4	1,9
$n : 10^6 \quad k : 10^5$	∞	1	∞	∞	1,7
$n : 10^6 \quad k : 10^6$	∞	1	∞	∞	1,8
$n : 10^7 \quad k : 10^0$	1	3,4	1,1	1,1	7,4
$n : 10^7 \quad k : 10^1$	8,6	2,6	1	1,1	6,7
$n : 10^7 \quad k : 10^2$	82,1	2,6	1	1,1	6,8
$n : 10^7 \quad k : 10^3$	∞	3,1	1	1,1	6,6
$n : 10^7 \quad k : 10^4$	∞	1,1	1	1,2	2,6
$n : 10^7 \quad k : 10^5$	∞	1	∞	∞	2,2
$n : 10^7 \quad k : 10^6$	∞	1	∞	∞	1,2
$n : 10^7 \quad k : 10^7$	∞	1	∞	∞	1,7

Comparação entre os Métodos de Ordenação Parcial

1. Para valores de k até 1.000, o método da InserçãoParcial é imbatível.
2. O $Quicksort$ Parcial nunca ficar muito longe da InserçãoParcial.
3. Na medida em que o k cresce,o $Quicksort$ Parcial é a melhor opção.
4. Para valores grandes de k, o método da InserçãoParcial se torna ruim.
5. Um método indicado para qualquer situação é o QuicksortParcial.
6. O $Heapsort$ Parcial tem comportamento parecido com o do $Quicksort$ Parcial.
7. No entano, o $Heapsort$ Parcial é mais lento.

Ordenação Externa

- A ordenação externa consiste em ordenar arquivos de tamanho maior que a memória interna disponível.
- Os métodos de ordenação externa são muito diferentes dos de ordenação interna.
- Na ordenação externa os algoritmos devem diminuir o número de acesso as unidades de memória externa.
- Nas memórias externas, os dados são armazenados como um arquivo seqüencial.
- Apenas um registro pode ser acessado em um dado momento.
- Esta é uma restrição forte se comparada com as possibilidades de acesso em um vetor.
- Logo, os métodos de ordenação interna são inadequados para ordenação externa.
- Técnicas de ordenação completamente diferentes devem ser utilizadas.

Ordenação Externa

Fatores que determinam as diferenças das técnicas de ordenação externa:

1. Custo para acessar um item é algumas ordens de grandeza maior.
2. O custo principal na ordenação externa é relacionado a transferência de dados entre a memória interna e externa.
3. Existem restrições severas de acesso aos dados.
4. O desenvolvimento de métodos de ordenação externa é muito dependente do estado atual da tecnologia.
5. A variedade de tipos de unidades de memória externa torna os métodos dependentes de vários parâmetros.
6. Assim, apenas métodos gerais serão apresentados.

Ordenação Externa

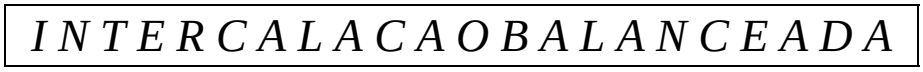
- O método mais importante é o de ordenação por intercalação.
- Intercalar significa combinar dois ou mais blocos ordenados em um único bloco ordenado.
- A intercalação é utilizada como uma operação auxiliar na ordenação.
- Estratégia geral dos métodos de ordenação externa:
  1. Quebre o arquivo em blocos do tamanho da memória interna disponível.
  2. Ordene cada bloco na memória interna.
  3. Intercale os blocos ordenados, fazendo várias passadas sobre o arquivo.
  4. A cada passada são criados blocos ordenados cada vez maiores, até que todo o arquivo esteja ordenado.

Ordenação Externa

- Os algoritmos para ordenação externa devem reduzir o número de passadas sobre o arquivo.
- Uma boa medida de complexidade de um algoritmo de ordenação por intercalação é o número de vezes que um item é lido ou escrito na memória auxiliar.
- Os bons métodos de ordenação geralmente envolvem no total menos do que dez passadas sobre o arquivo.

Intercalação Balanceada de Vários Caminhos

- Considere um arquivo armazenado em uma fita de entrada:



- Objetivo:
  - Ordenar os 22 registros e colocá-los em uma fita de saída.
- Os registros são lidos um após o outro.
- Considere uma memória interna com capacidade para para três registros.
- Considere que esteja disponível seis unidades de fita magnética.

Intercalação Balanceada de Vários Caminhos

- Fase de criação dos blocos ordenados:

fita 1:	*I N T*	*A C O*	*A D E*
fita 2:	*C E R*	*A B L*	*A*
fita 3:	*A A L*	*A C N*	

Intercalação Balanceada de Vários Caminhos

- Fase de intercalação - Primeira passada:
  1. O primeiro registro de cada fita é lido.
  2. Retire o registro contendo a menor chave.
  3. Armazene-o em uma fita de saída.
  4. Leia um novo registro da fita de onde o registro retirado é proveniente.
  5. Ao ler o terceiro registro de um dos blocos, sua fita fica inativa.
  6. A fita é reativada quando o terceiro registro das outras fitas forem lidos.
  7. Neste instante um bloco de nove registros ordenados foi formado na fita de saída.
  8. Repita o processo para os blocos restantes.
- Resultado da primeira passada da segunda etapa:

fita 4:	*A A C E I L N R T*
fita 5:	*A A A B C C L N O*
fita 6:	*A A D E*

Intercalação Balanceada de Vários Caminhos

- Quantas passadas são necessárias para ordenar um arquivo de tamanho arbitrário?
  - Seja n, o número de registros do arquivo.
  - Suponha que cada registro ocupa m palavras na memória interna.
  - A primeira etapa produz n/m blocos ordenados.
  - Seja $P(n)$ o número de passadas para a fase de intercalação.
  - Seja f o número de fitas utilizadas em cada passada.
  - Assim:

    $$P(n) = \log_f \frac{n}{m}.$$

    No exemplo acima, n=22, m=3 e f=3 temos:

    $$P(n) = \log_3 \frac{22}{3} = 2.$$

Intercalação Balanceada de Vários Caminhos

- No exemplo foram utilizadas $2f$ fitas para uma intercalação-de-f-caminhos.
- É possível usar apenas $f + 1$ fitas:
  - Encaminhe todos os blocos para uma única fita.
  - Redistribuia estes blocos entre as fitas de onde eles foram lidos.
  - O custo envolvido é uma passada a mais em cada intercalação.
- No caso do exemplo de 22 registros, apenas quatro fitas seriam suficientes:
  - A intercalação dos blocos a partir das fitas 1, 2 e 3 seria toda dirigida para a fita 4.
  - Ao final, o segundo e o terceiro blocos ordenados de nove registros seriam transferidos de volta para as fitas 1 e 2.

Implementação por meio de Seleção por Substituição

- A implementação do método de intercalação balanceada pode ser realizada utilizando filas de prioridades.
- As duas fases do método podem ser implementadas de forma eficiente e elegante.
- Operações básicas para formar blocos ordenados:
  - Obter o menor dentre os registros presentes na memória interna.
  - Substituí-lo pelo próximo registro da fita de entrada.
- Estrutura ideal para implementar as operações: *heap*.
- Operação de substituição:
  - Retirar o menor item da fila de prioridades.
  - Colocar um novo item no seu lugar.
  - Reconstituir a propriedade do *heap*.

Implementação por meio de Seleção por Substituição

Algoritmo:

1. Inserir m elementos do arquivo na fila de prioridades.
2. Substituir o menor item da fila de prioridades pelo próximo item do arquivo.
3. Se o próximo item é menor do que o que saiu, então:
   - Considere-o membro do próximo bloco.
   - Trate-o como sendo maior do que todos os itens do bloco corrente.
4. Se um item marcado vai para o topo da fila de prioridades então:
   - O bloco corrente é encerrado.
   - Um novo bloco ordenado é iniciado.

Implementação por meio de Seleção por Substituição

Primeira passada sobre o arquivo exemplo:

Entra	1	2	3
E	*I*	*N*	*T*
R	*N*	*E**	*T*
C	*R*	*E**	*T*
A	*T*	*E**	*C**
L	*A**	*E**	*C**
A	*C**	*E**	*L**
C	*E**	*A*	*L**
A	*L**	*A*	*C*
O	*A*	*A*	*C*
B	*A*	*O*	*C*
A	*B*	*O*	*C*
L	*C*	*O*	*A**
A	*L*	*O*	*A**
N	*O*	*A**	*A**
C	*A**	*N**	*A**
E	*A**	*N**	*C**
A	*C**	*N**	*E**
D	*E**	*N**	*A*
A	*N**	*D*	*A*
	A	*D*	*A*
	A	*D*	
	D		

Os asteriscos indicam quais chaves pertencem a blocos diferentes.

Implementação por meio de Seleção por Substituição

- Tamanho dos blocos produzidas para chaves randômicas:
  - Os blocos ordenados são cerca de duas vezes o tamanho dos blocos criados pela ordenação interna.
- Isso pode salvar uma passada na fase de intercalação.
- Se houver alguma ordem nas chaves, os blocos ordenados podem ser ainda maiores.
- Se nenhuma chave possui mais do que m chaves maiores do que ela, o arquivo é ordenado em um passo.
- Exemplo para as chaves RAPAZ:

Entra	1	2	3
A	*A*	*R*	*P*
Z	*A*	*R*	*P*
	P	*R*	*Z*
	R	*Z*	
	Z		

Implementação por meio de Seleção por Substituição

- Fase de intercalação dos blocos ordenados obtidos na primeira fase:
  - Operação básica: obter o menor item dentre os ainda não retirados dos f blocos a serem intercalados.

Algoritmo:

- Monte uma fila de prioridades de tamanho f.
- A partir de cada uma das f entradas:
  - Substitua o item no topo da fila de prioridades pelo próximo item do mesmo bloco do item que está sendo substituído.
  - Imprima em outra fita o elemento substituído.

Implementação por meio de Seleção por Substituição

- Exemplo:

Entra	1	2	3
A	*A*	*C*	*I*
L	*A*	*C*	*I*
E	*C*	*L*	*I*
R	*E*	*L*	*I*
N	*I*	*L*	*R*
	L	*N*	*R*
T	*N*	*R*	
	R	*T*	
	T		

- Para f pequeno não é vantajoso utilizar seleção por substituição para intercalar blocos:
  - Obtém-se o menor item fazendo $f - 1$ comparações.
- Quando f é 8 ou mais, o método é adequado:
  - Obtém-se o menor item fazendo $\log_2 f$ comparações.

Considerações Práticas

- As operações de entrada e saída de dados devem ser implementadas eficientemente.
- Deve-se procurar realizar a leitura, a escrita e o processamento interno dos dados de forma simultânea.
- Os computadores de maior porte possuem uma ou mais unidades independentes para processamento de entrada e saída.
- Assim, pode-se realizar processamento e operações de E/S simultaneamente.

Considerações Práticas

- Técnica para obter superposição de E/S e processamento interno:
  - Utilize $2f$ áreas de entrada e $2f$ de saída.
  - Para cada unidade de entrada ou saída, utiliza-se duas áreas de armazenamento:
    1. Uma para uso do processador central
    2. Outra para uso do processador de entrada ou saída.
  - Para entrada, o processador central usa uma das duas áreas enquanto a unidade de entrada está preenchendo a outra área.
  - Depois a utilização das áreas é invertida entre o processador de entrada e o processador central.
  - Para saída, a mesma técnica é utilizada.

Considerações Práticas

- Problemas com a técnica:
  - Apenas metade da memória disponível é utilizada.
  - Isso pode levar a uma ineficiência se o número de áreas for grande.
    Ex: Intercalação-de-f-caminhos para f grande.
  - Todas as f áreas de entrada em uma intercalação-de-f-caminhos se esvaziando aproximadamente ao mesmo tempo.

Considerações Práticas

- Solução para os problemas:
  - Técnica de previsão:
    - Requer a utilização de uma única área extra de armazenamento durante a intercalação.
    - Superpõe a entrada da próxima área que precisa ser preenchida com a parte de processamento interno do algoritmo.
    - É fácil saber qual área ficará vazia primeiro.
    - Basta olhar para o último registro de cada área.
    - A área cujo último registro é o menor, será a primeira a se esvaziar.

Considerações Práticas

- Escolha da ordem de intercalação f:
  - Para fitas magnéticas:
    - f deve ser igual ao número de unidades de fita disponíveis menos um.
    - A fase de intercalação usa f fitas de entrada e uma fita de saída.
    - O número de fitas de entrada deve ser no mínimo dois.
  - Para discos magnéticos:
    - O mesmo raciocínio acima é válido.
    - O acesso seqüencial é mais eficiente.
  - Sedegwick (1988) sugere considerar f grande o suficiente para completar a ordenação em poucos passos.
  - Porém, a melhor escolha para f depende de vários parâmetros relacionados com o sistema de computação disponível.

Intercalação Polifásica

- Problema com a intercalação balanceada de vários caminhos:
  - Necessita de um grande número de fitas.
  - Faz várias leituras e escritas entre as fitas envolvidas.
  - Para uma intercalação balanceada de f caminhos são necessárias $2f$ fitas.
  - Alternativamente, pode-se copiar o arquivo quase todo de uma única fita de saída para f fitas de entrada.
  - Isso reduz o número de fitas para $f + 1$.
  - Porém, há um custo de uma cópia adicional do arquivo.
- Solução:
  - **Intercalação polifásica**.

Intercalação Polifásica

- Os blocos ordenados são distribuídos de forma desigual entre as fitas disponíveis.
- Uma fita é deixada livre.
- Em seguida, a intercalação de blocos ordenados é executada até que uma das fitas esvazie.
- Neste ponto, uma das fitas de saída troca de papel com a fita de entrada.

Intercalação Polifásica

- Exemplo:
  - Blocos ordenados obtidos por meio de seleção por substituição:

fita 1:	*I N R T*	*A C E L*	*A A B C L O*
fita 2:	*A A C E N*	*A A D*	
fita 3:			

  - Configuração após uma intercalação-de-2-caminhos das fitas 1 e 2 para a fita 3:

fita 1:	*A A B C L O*	
fita 2:		
fita 3:	*A A C E I N N R T*	*A A A C D E L*

Intercalação Polifásica

- Exemplo:
  - Depois da intercalação-de-2-caminhos das fitas 1 e 3 para a fita 2:

fita 1:
fita 2: *A A A A B C C E I L N N O R T*
fita 3: *A A A C D E L*

  - Finalmente:

fita 1: *A A A A A A A B C C C D E E I L L N N O R T*
fita 2:
fita 3:

  - A intercalação é realizada em muitas fases.
  - As fases não envolvem todos os blocos.
  - Nenhuma cópia direta entre fitas é realizada.

Intercalação Polifásica

- A implementação da intercalação polifásica é simples.
- A parte mais delicada está na distribuição inicial dos blocos ordenados entre as fitas.
- Distribuição dos blocos nas diversas etapas do exemplo:

fita 1	fita 2	fita 3	Total
3	2	0	5
1	0	2	3
0	1	1	2
1	0	0	1

Intercalação Polifásica

Análise:

- A análise da intercalação polifásica é complicada.
- O que se sabe é que ela é ligeiramente melhor do que a intercalação balanceada para valores pequenos de f.
- Para valores de $f > 8$, a intercalação balanceada pode ser mais rápida.

Quicksort Externo

- Foi proposto por Monard em 1980.
- Utiliza o paradigma de **divisão e conquista**.
- O algoritmo ordena ***in situ*** um arquivo $A = \{R_1, \ldots, R_n\}$ de n registros.
- Os registros estão armazenados consecutivamente em memória secundária de acesso randômico.
- O algoritmo utiliza somente $O(\log n)$ unidades de memória interna e não é necessária nenhuma memória externa adicional.

Quicksort Externo

- São necessários quatro métodos adicionais para a classe $MeuItem$ (transparência 6).
- O método $toString$ especifica como o objeto é formatado.
- Os métodos $leArq$ e $gravaArq$ são utilizados para ler e gravar um objeto da classe $MeuItem$ em um arquivo de acesso aleatório.
- O método $tamanho$ retorna o tamanho em *bytes* de um objeto da classe $MeuItem$.

```
public String toString () { return "" + this.chave; }

public void gravaArq (RandomAccessFile arq)
                                        throws IOException {
  arq.writeInt (this.chave);
}
public void leArq (RandomAccessFile arq)
                                        throws IOException {
  this.chave = arq.readInt ();
}
public static int tamanho () { return 4; /* 4 bytes */ }
```

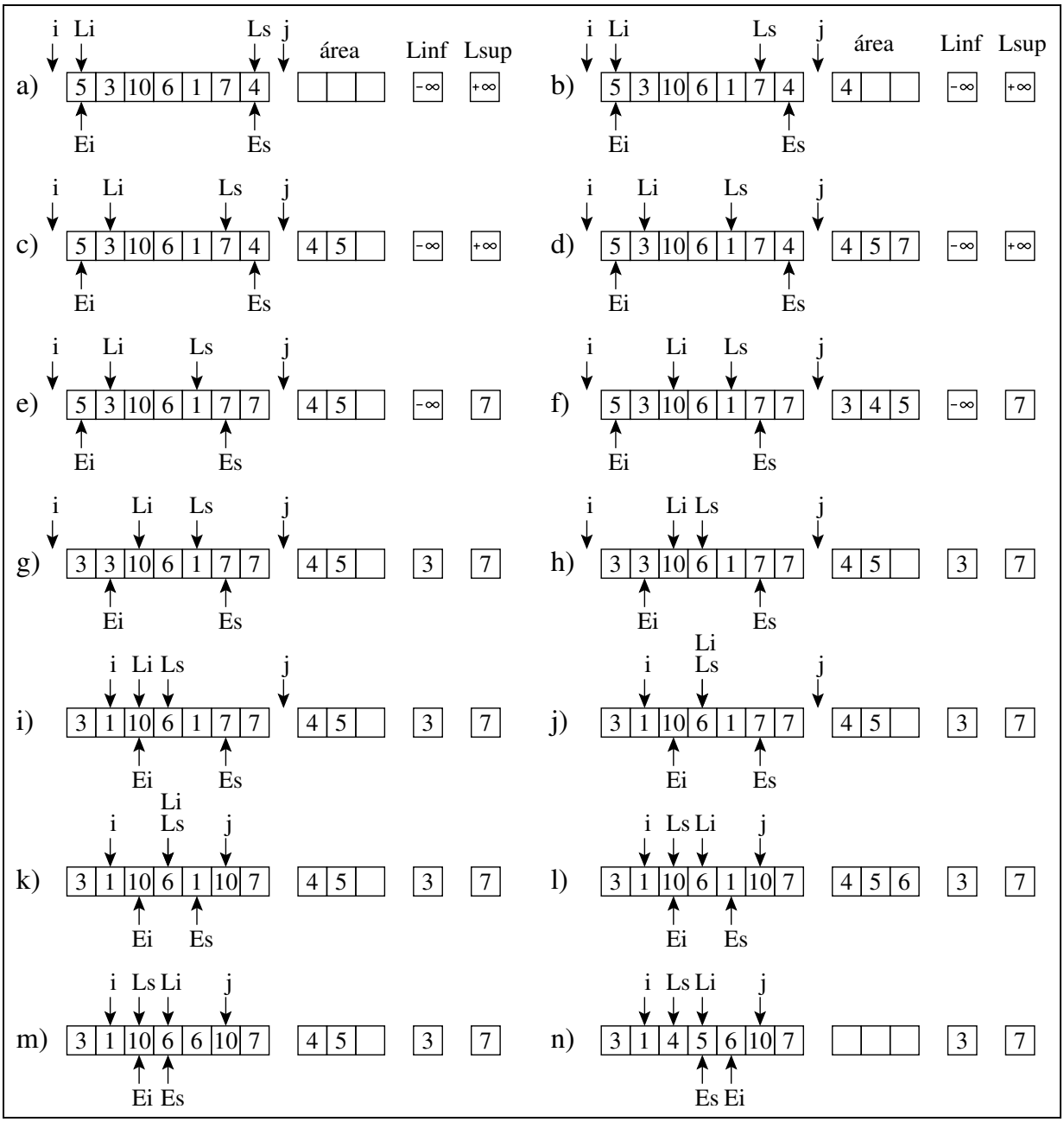
Quicksort Externo

- Seja R_i, $1 \leq i \leq n$, o registro que se encontra na i-ésima posição de A.
- Algoritmo:
  1. Particionar A da seguinte forma: $\{R_1, \ldots, R_i\} \leq R_{i+1} \leq R_{i+2} \leq \ldots \leq R_{j-2} \leq R_{j-1} \leq \{R_j, \ldots, R_n\}$,
  2. chamar recursivamente o algoritmo em cada um dos subarquivos $A_1 = \{R_1, \ldots, R_i\}$ e $A_2 = \{R_j, \ldots, R_n\}$.

Quicksort Externo

- Para o partionamento é utilizanda uma área de armazenamento na memória interna.
- Tamanho da área: $\mathrm{TamArea} = j - i - 1$, com $\mathrm{TamArea} \geq 3$.
- Nas chamadas recusivas deve-se considerar que:
  - Primeiro deve ser ordenado o subarquivo de menor tamanho.
  - Condição para que, na média, $O(\log n)$ subarquivos tenham o processamento adiado.
  - Subarquivos vazios ou com um único registro são ignorados.
  - Caso o arquivo de entrada A possua no máximo $\mathrm{TamArea}$ registros, ele é ordenado em um único passo.

Quicksort Externo



Quicksort Externo

- O programa a seguir apresenta a classe $QuicksortExterno$, na qual são definidos as estruturas de dados e os métodos utilizados pelo algoritmo $Quicksort$ Externo.

```
package cap4.ordenacaoexterna;
import cap3.arranjo.Area;
import cap4.MeuItem;
import java.io.*;
public class QuicksortExterno {
   private static class LimiteParticoes { int i; int j; }
   private RandomAccessFile arqLi;
   private RandomAccessFile arqEi;
   private RandomAccessFile arqLEs;
   private boolean          ondeLer;
   private MeuItem          ultLido;
   private Area             area;
   private int              tamArea;
   // Métodos utilizados pelo método particao do quicksort externo
   private int leSup (int ls) throws IOException
   private int leInf (int li) throws IOException
   private int inserirArea () throws Exception
```

// Continua na próxima transparência

Quicksort Externo

```
  private int escreveMax (int es) throws Exception
  private int escreveMin (int ei) throws IOException
  private int retiraMax () throws Exception
  private int retiraMin () throws Exception
  private LimiteParticoes particao
                    (int esq, int dir) throws Exception
  public QuicksortExterno (String nomeArq, int tamArea)
  throws FileNotFoundException {
    this.arqLi    = new RandomAccessFile (nomeArq, "rws");
    this.arqEi    = new RandomAccessFile (nomeArq, "rws");
    this.arqLEs   = new RandomAccessFile (nomeArq, "rws");
    this.tamArea = tamArea;
  }
  public void quicksortExterno
                    (int esq, int dir) throws Exception
  public void fechaArquivos() throws Exception {
    this.arqEi.close();
    this.arqLi.close();
    this.arqLEs.close();
  }
}
```

Quicksort Externo

```
public void quicksortExterno (int esq, int dir)
                                        throws Exception {
   if (dir – esq < 1) return;
   LimiteParticoes p = particao (esq, dir);
   if (p.i – esq < dir – p.j) { // ordene primeiro o subar-
quivo menor
      quicksortExterno (esq, p.i);
      quicksortExterno (p.j, dir);
   }
   else {
      quicksortExterno (p.j, dir);
      quicksortExterno (esq, p.i);
   }
}
```

Quicksort Externo

Métodos auxiliares utilizados pelo método *particao*:

```
private int leSup (int ls) throws IOException {
  this.ultLido = new Meuitem (0);
  arqLEs.seek ((ls − 1) ∗ Meuitem.tamanho ());
  this.ultLido.leArq (arqLEs); ondeLer = false;
  return −−ls;
}
private int leInf (int li) throws IOException {
  this.ultLido = new Meuitem (0);
  this.ultLido.leArq (arqLi); ondeLer = true;
  return ++li;
}
private int inserirArea () throws Exception {
  area.insereItem (this.ultLido);
  return area.obterNumCelOcupadas ();
}
```

Quicksort Externo

Métodos auxiliares utilizados pelo método $particao$:

```
private int escreveMax (int es) throws Exception {
  arqLEs.seek ((es − 1) ∗ MeuItem.tamanho ());
  this.ultLido.gravaArq (arqLEs);
  return −−es;
}
private int escreveMin (int ei) throws IOException {
  this.ultLido.gravaArq (arqEi);
  return ++ei;
}
private int retiraMax () throws Exception {
  this.ultLido = (MeuItem) area.retiraUltimo ();
  return area.obterNumCelOcupadas ();
}
private int retiraMin () throws Exception {
  this.ultLido = (MeuItem) area.retiraPrimeiro ();
  return area.obterNumCelOcupadas ();
}
```

Quicksort Externo

Método Partição:

```
private LimiteParticoes particao (int esq, int dir)
                                              throws Exception {
  int ls = dir, es = dir, li = esq, ei = esq, nrArea = 0;
  Meuitem linf = new Meuitem (Integer.MIN_VALUE); // −∞
  Meuitem lsup = new Meuitem (Integer.MAX_VALUE); // ∞
  this.ondeLer = true;
  LimiteParticoes p = new LimiteParticoes ();
  this.area = new Area (this.tamArea);
  arqLi.seek ((li − 1) * Meuitem.tamanho ());
  arqEi.seek ((ei − 1) * Meuitem.tamanho ());
  p.i = esq − 1; p.j = dir + 1;
  while (ls >= li) {
    if (nrArea < this.tamArea − 1) {
      if (ondeLer) ls = this.leSup (ls);
      else li = leInf (li);
      nrArea = inserirArea ();
    }
    else {
      if (ls == es) ls = leSup (ls);
      else if (li == ei) li = leInf (li);
      else if (ondeLer) ls = leSup (ls);
      else li = leInf (li);
// Continua na próxima transparência
```

Quicksort Externo

Método Partição:

```
        if (ultLido.compara (lsup) > 0) {
          p.j = es; es = escreveMax (es);
        }
        else if (ultLido.compara (linf) < 0) {
          p.i = ei; ei = escreveMin (ei);
        }
        else {
          nrArea = inserirArea ();
          if (ei – esq < dir – es) {
            nrArea = retiraMin ();
            linf = this.ultLido; ei=escreveMin (ei);
          }
          else {
            nrArea = retiraMax ();
            lsup = this.ultLido; es=escreveMax (es);
          }
        }
      }
    }
    while (ei <= es) {
            nrArea = retiraMin (); ei = escreveMin (ei);
            }
    return p;
}
```

Quicksort Externo

Programa teste:

```
package cap4;
import java.io.RandomAccessFile;
import cap4.ordenacaoexterna.QuicksortExterno; // vide transparência 122
public class TestaQuicksortExterno {
  public static void main (String[] args) {
    try {
      RandomAccessFile arq = new RandomAccessFile ("qe.dat", "rwd");
      MeuItem item = new MeuItem (5); item.gravaArq (arq);
      item = new MeuItem (3); item.gravaArq (arq);
      item = new MeuItem (10); item.gravaArq (arq);
      item = new MeuItem (6); item.gravaArq (arq);
      item = new MeuItem (1); item.gravaArq (arq);
      item = new MeuItem (7); item.gravaArq (arq);
      item = new MeuItem (4); item.gravaArq (arq);
      arq.close ();
      QuicksortExterno quicksortExterno=new QuicksortExterno("qe.dat",3);
      quicksortExterno.quicksortExterno (1, 7);
      quicksortExterno.fechaArquivos ();
      arq = new RandomAccessFile ("qe.dat", "r");
      item.leArq (arq);
      while (arq.getFilePointer () < arq.length ()) {
        System.out.println ("Registro=" + item.toString ());
        item.leArq (arq);
      }
      System.out.println ("Registro=" + item.toString ()); arq.close ();
    } catch (Exception e) { System.out.println (e.getMessage ()); }
  }
}
```

Quicksort Externo

Análise:

- Seja n o número de registros a serem ordenados.
- Seja e b o tamanho do bloco de leitura ou gravação do Sistema operacional.
- Melhor caso: $O(\frac{n}{b})$
  - Por exemplo, ocorre quando o arquivo de entrada já está ordenado.
- Pior caso: $O(\frac{n^2}{\mathrm{TamArea}})$
  - ocorre quando um dos arquivos retornados pelo procedimento Particao tem o maior tamanho possível e o outro é vazio.
  - A medida que n cresce, a probabilidade de ocorrência do pior caso tende a zero.
- Caso Médio: $O(\frac{n}{b}log(\frac{n}{\mathrm{TamArea}}))$
  - É o que tem amaior probabilidade de ocorrer.

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 $\Rightarrow$ 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 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:

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

- Pesquisa sem sucesso:

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

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

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

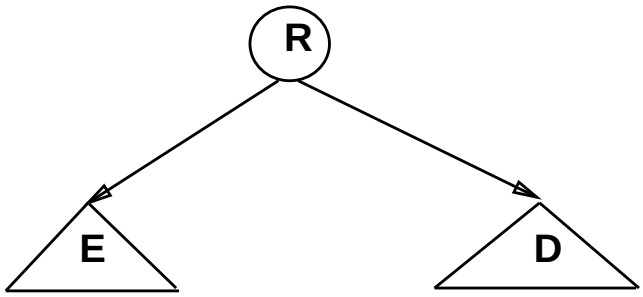
- A cada iteração do algoritmo, o tamanho da tabela é dividido ao meio.
- **Logo:** o número de vezes que o tamanho da tabela é dividido ao meio é cerca de $\log n$.
- **Ressalva:** o custo para manter a tabela ordenada é alto:
  a cada inserção na posição p da tabela implica no deslocamento dos registros 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

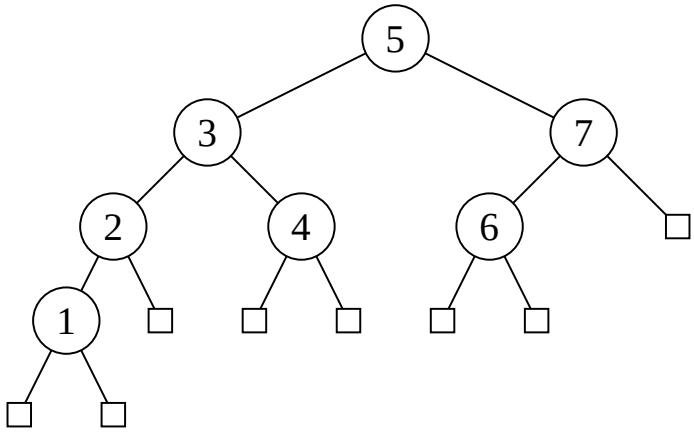


Temos a relação invariante

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;
}
```

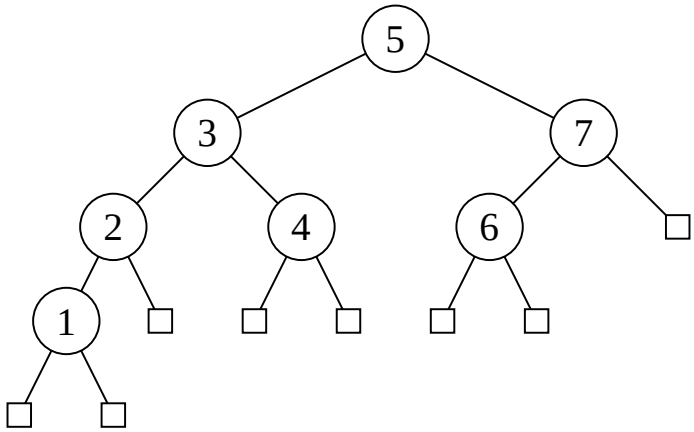
Programa para Criar a Árvore

```
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

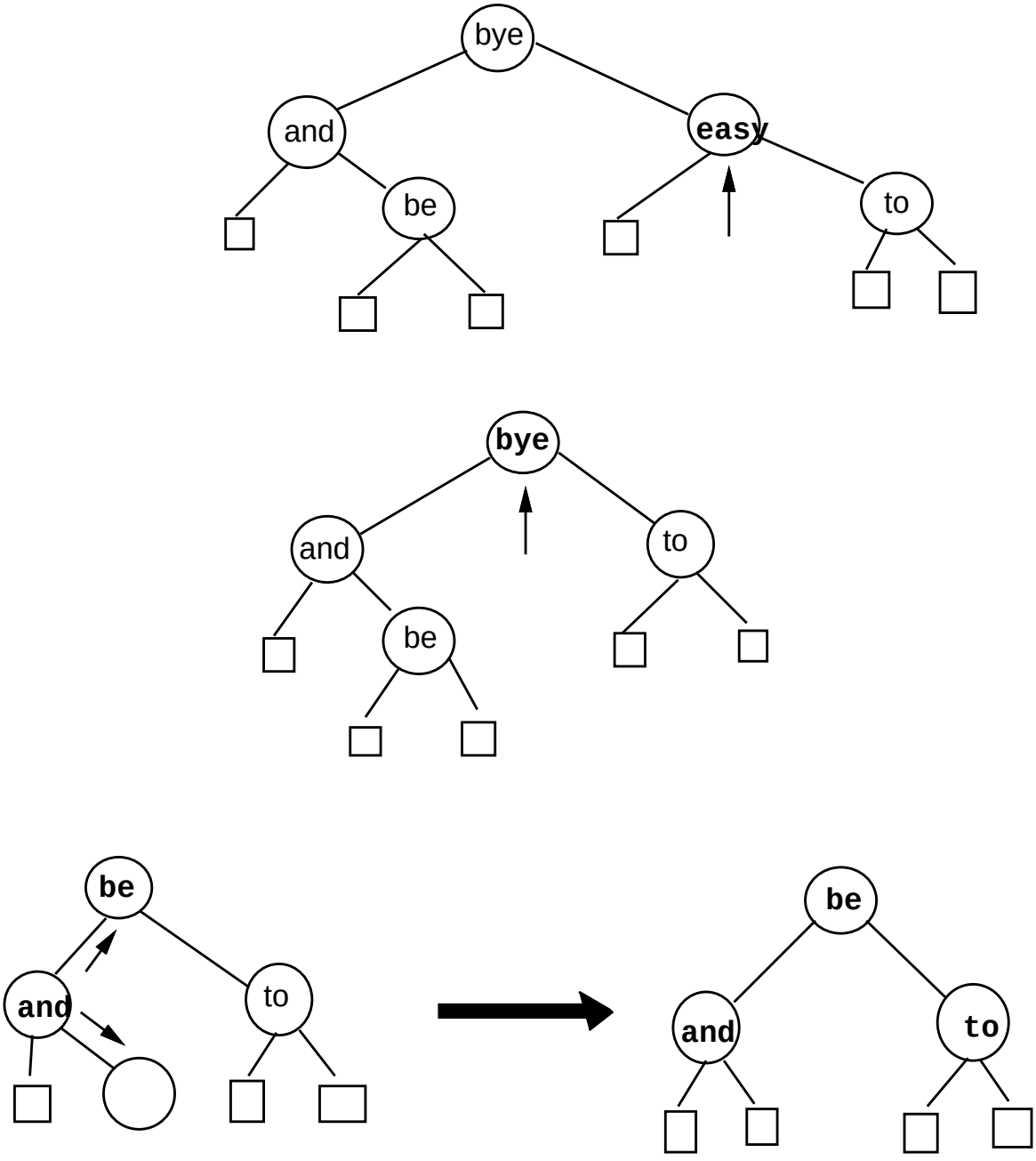


Assim: para retirar o registro com chave 5 na árvore basta trocá-lo pelo registro com chave 4 ou pelo registro com chave 6, e então retirar o nó que recebeu o registro com chave 5.

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

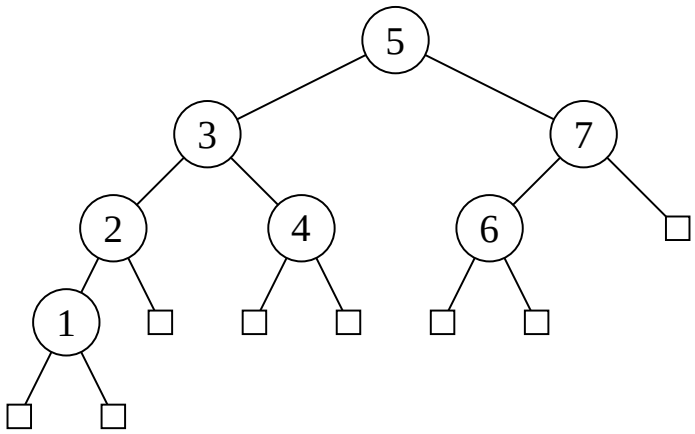


Caminhamento Central

- 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.

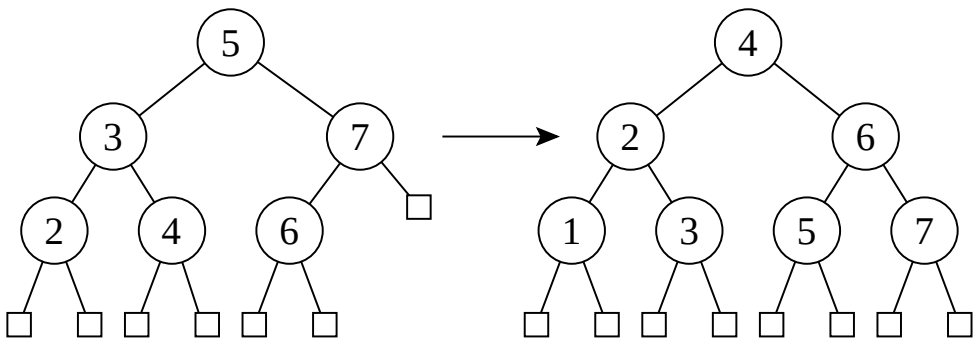
Análise

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 $\Rightarrow$ 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 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.

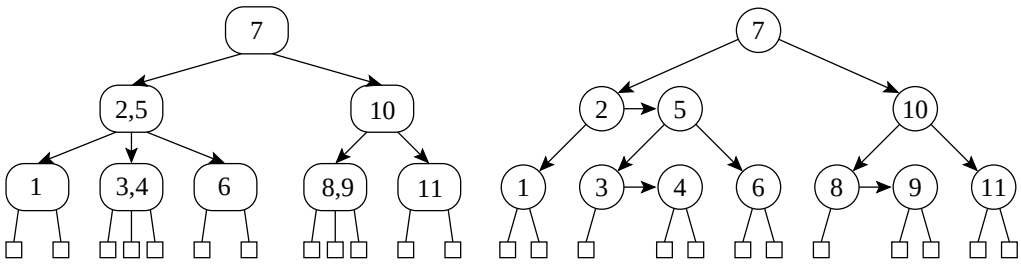
- **Heurísticas:** existem várias heurísticas baseadas no princípio acima.
- Gonnet e Baeza-Yates (1991) apresentam algoritmos que utilizam vários critérios de balanceamento para árvores de pesquisa, tais como restrições impostas:
  - na diferença das alturas de subárvores de cada nó da árvore,
  - na 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

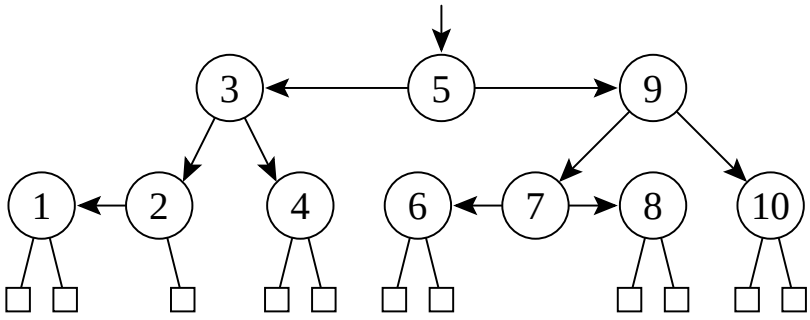
- Árvores B $\Rightarrow$ estrutura para memória secundária. (Bayer R. e McCreight E.M., 1972)
- **Árvore 2-3** $\Rightarrow$ caso especial da árvore B.
- Cada nó tem duas ou três subárvores.
- Mais apropriada para memória primária.
- **Exemplo: Uma árvore 2-3 e a árvore B binária correspondente**(Bayer,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



Árvores SBB

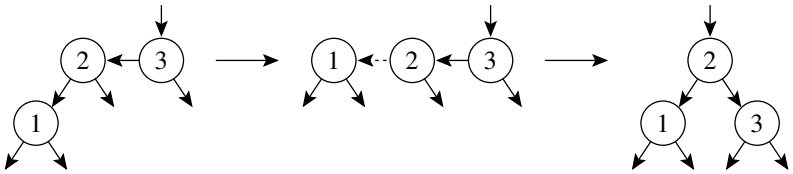
- Árvore 2-3 $\Rightarrow$ **árvore B binária** (assimetria inerente)
  1. Referências à esquerda apontam para um nó no nível abaixo.
  2. Referências à direita podem ser verticais ou horizontais.

  Eliminação da assimetria nas árvores B binárias $\Rightarrow$ 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.

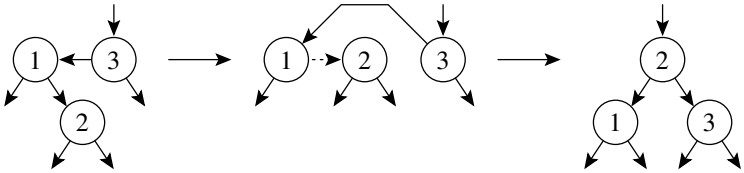


Transformações para Manutenção da Propriedade SBB

- 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

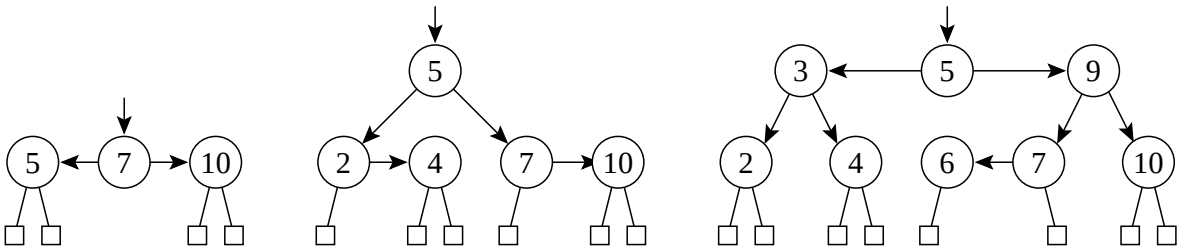
```
private No insere (Item reg, No pai, No filho, boolean filhoEsq) {
  if (filho == null) {
    filho = new No (); filho.reg = reg;
    filho.incE = Vertical; filho.incD = Vertical;
    filho.esq = null; filho.dir = null;
    if (pai != null)
      if (filhoEsq) pai.incE = Horizontal; else pai.incD = 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

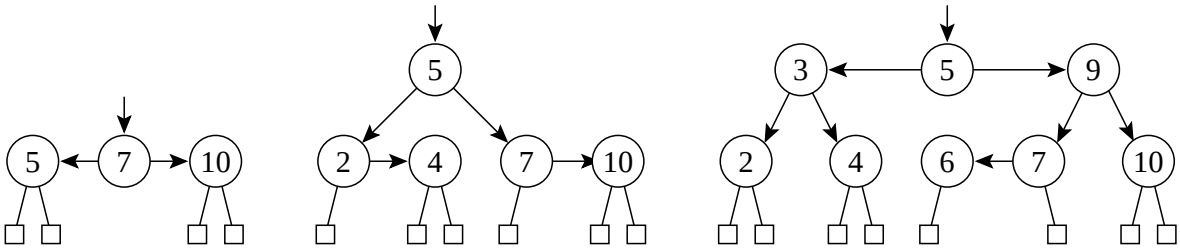
```
private No antecessor (No q, No r) {
  if (r.dir != null) {
    r.dir = antecessor (q, r.dir);
    if (!this.propSBB) r = this.dirCurto (r);
  }
  else {
    q.reg = r.reg;
    r = r.esq;
    if (r != null) this.propSBB = true;
  }
  return r;
}
```

Método *retira* para retirada da árvore SBB

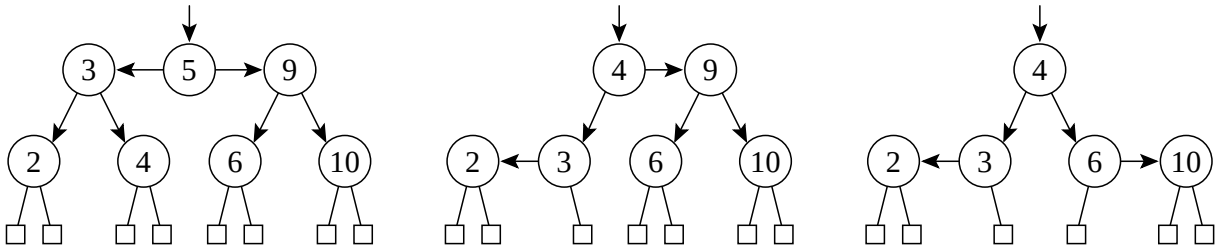
```
private No retira (Item reg, No ap) {
  if (ap == null) {
    System.out.println ("Erro: Registro nao encontrado");
    this.propSBB = true;
  }
  else if (reg.compara (ap.reg) < 0) {
    ap.esq = retira (reg, ap.esq);
    if (!this.propSBB)
      ap = this.esqCurto (ap);
  }
  else if (reg.compara (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

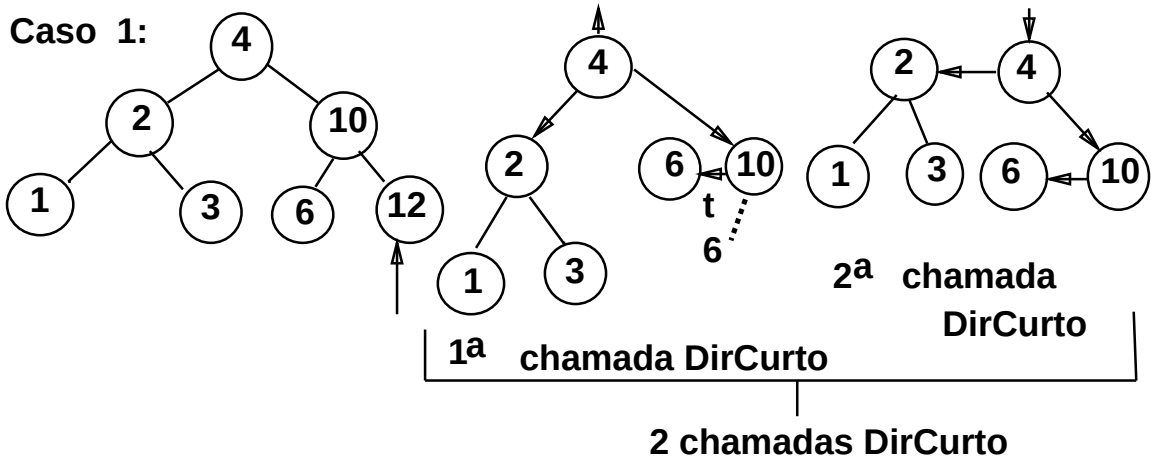
- **Dada a Árvore:**

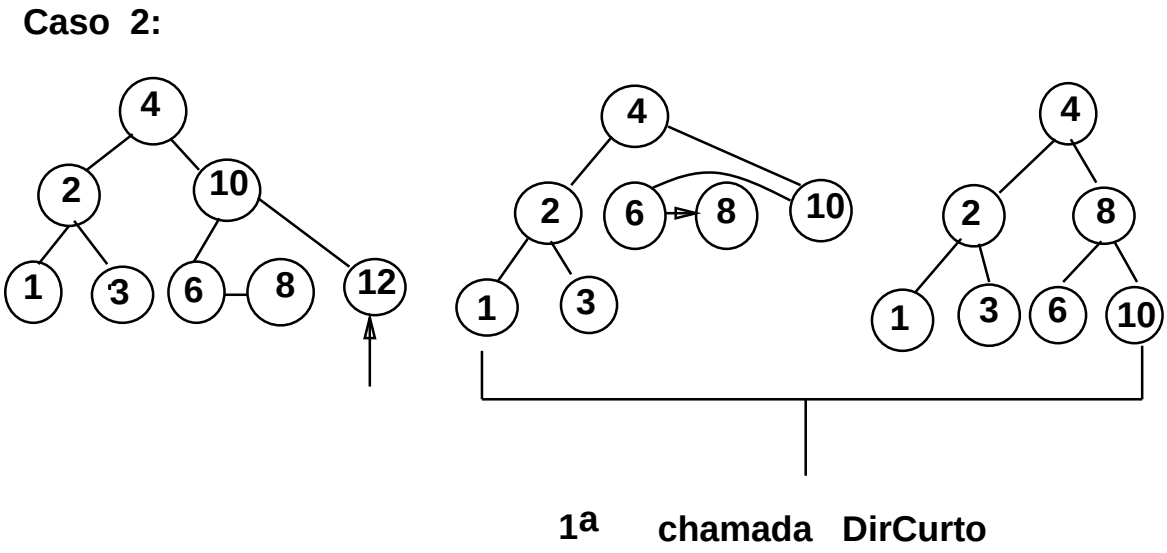


- Resultado obtido quando se retira uma seqüência de chaves da árvore SBB mais à direita acima:
  - A árvore à esquerda é obtida após a retirada da chave 7 da árvore à direita acima.
  - A árvore do meio é obtida após a retirada da chave 5 da árvore anterior.
  - A árvore à direita é obtida após a retirada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



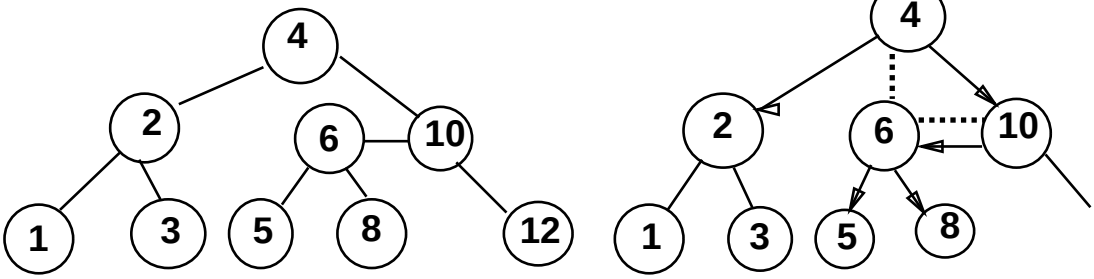
Exemplo: Retirada de Nós de SBB

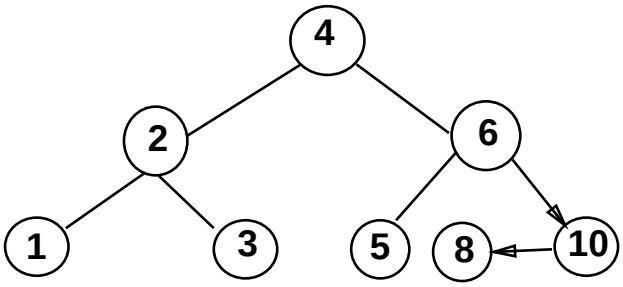




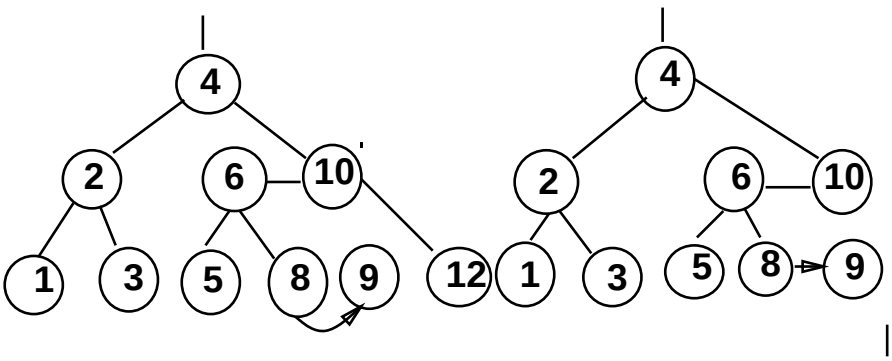
Exemplo: Retirada de Nós de SBB

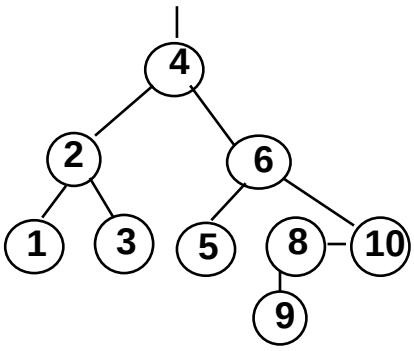
Caso 3:

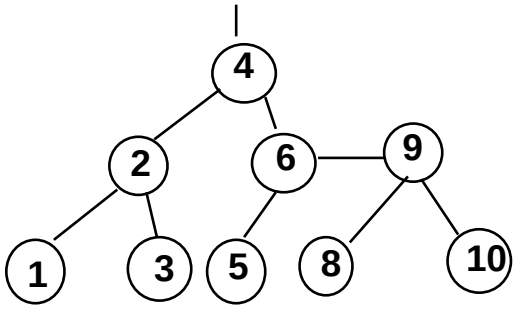




Se nodo 8 tem filho:







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**

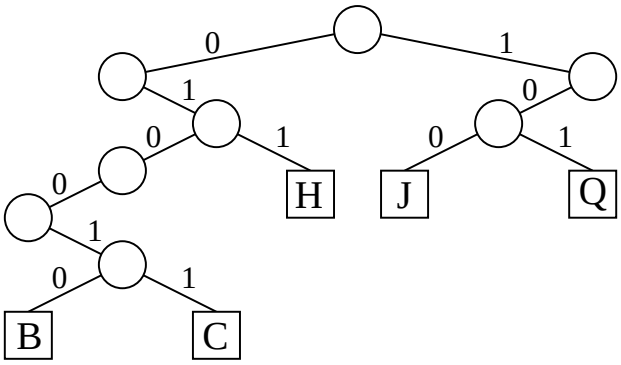
Trie

- 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**.

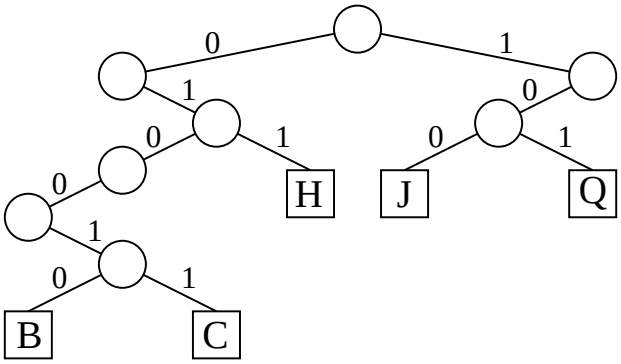
Exemplo

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

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

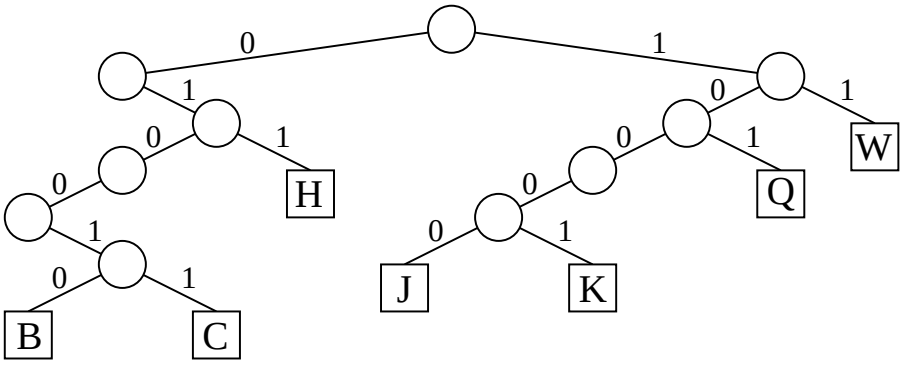


Inserção das Chaves W e K na Trie Binária



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.

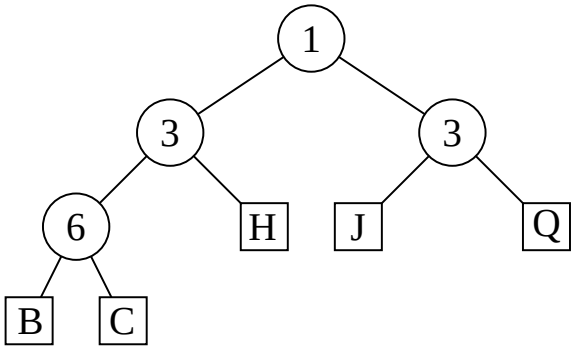
Patricia - Practical Algorithm To Retrieve Information Coded In Alphanumeric

- Criado por Morrison D. R. 1968 para aplicação em recuperação de informação em arquivos de grande porte.
- Knuth D. E. 1973 → 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.

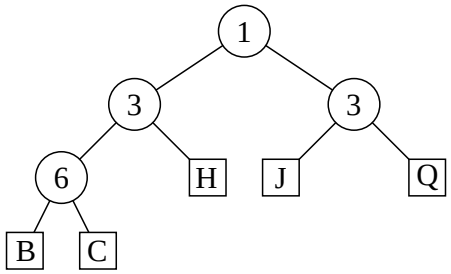
Mais sobre Patricia

- 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:

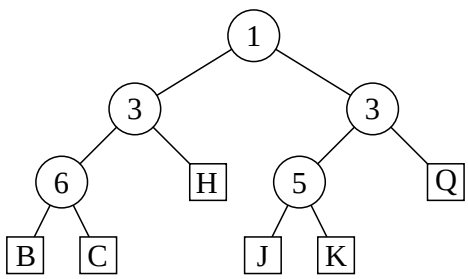
$$\begin{aligned} \mathrm{B} &= 010010 \\ \mathrm{C} &= 010011 \\ \mathrm{H} &= 011000 \\ \mathrm{J} &= 100001 \\ \mathrm{Q} &= 101000 \end{aligned}$$



Inserção da Chave K

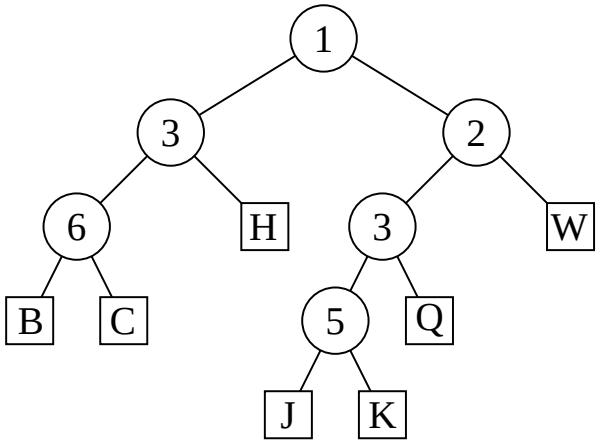


- Para inserir a chave K $=$ 100010 na árvore acima, a pesquisa inicia pela raiz e termina quando se chega ao nó externo contendo J.
- Os índices dos bits nas chaves estão ordenados da esquerda para a direita. Bit de índice 1 de K é 1 $\rightarrow$ a subárvore direita Bit de índice 3 $\rightarrow$ subárvore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

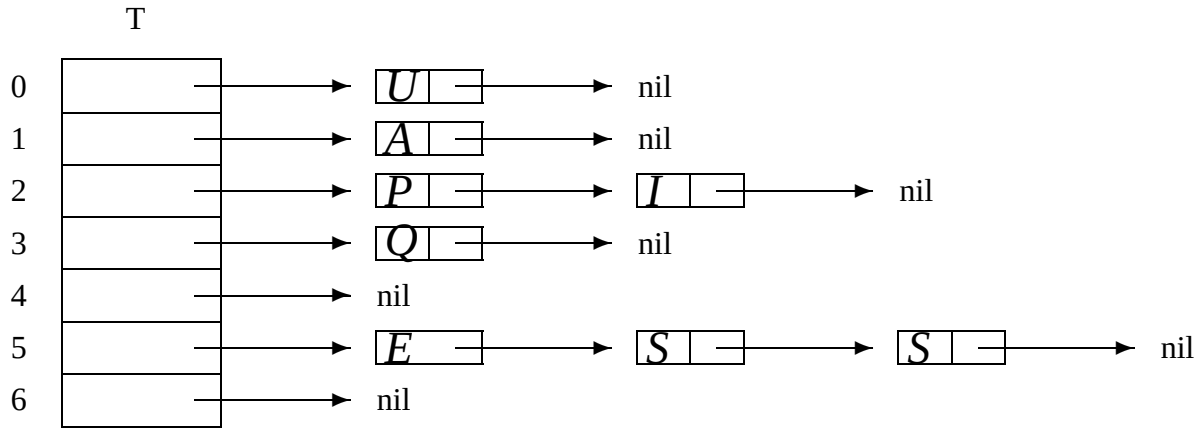
```
private int[] geraPesos (int n) {
  int p[] = new int[n];
  java.util.Random rand = new java.util.Random ();
  for (int i = 0; i < n; i++) p[i] = rand.nextInt(M) + 1;
  return p;
}
```

- **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:

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

Exemplo

- Se a i-ésima letra do alfabeto é representada pelo número i e a função de transformação $h(chave) = chave \bmod M$ é utilizada para $M = 7$,

- então o resultado da inserção das chaves $L\ U\ N\ E\ S$ na tabela, 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*

```
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*

```
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*

```
  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 $g : V \rightarrow [0, N - 1]$, 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.

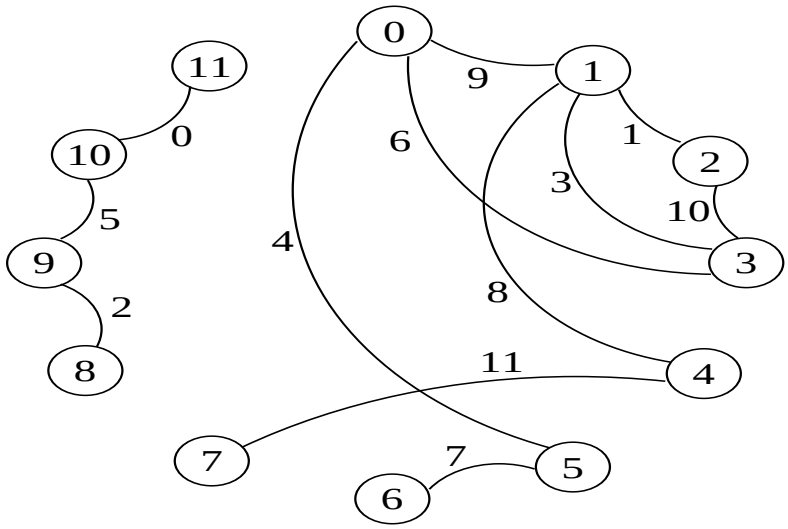
Exemplo

- **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.



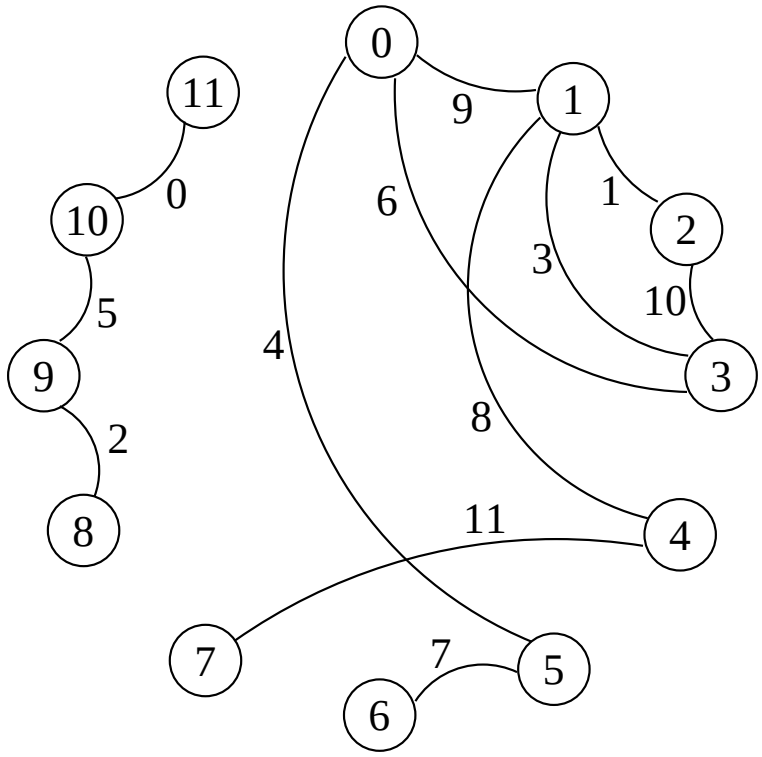
- Os vértices são rotulados com valores no intervalo $0 \ldots M-1$
- As arestas definidas por $(h_1(x), h_2(x))$ para cada uma das N chaves x.
- Cada chave corresponde a uma aresta que é rotulada com o valor desejado para a função hp perfeita.
- Os valores das duas funções $h_1(x)$ e $h_2(x)$ definem 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 $p_1[i]$ e $p_2[i]$ para
      $0 \leq i \leq maxTamChave - 1$;
    Gera o grafo $G = (V, A)$;
    grafoRotulavel = atribuig (G, g);
  } **while** (!grafoRotulavel);
  Retorna p_1 , p_2 e g;
}

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

```
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

```
      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*

```
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

Pesquisa em Memória Secundária*

Última alteração: 10 de Outubro de 2006

*Transparências elaboradas por Wagner Meira Jr, Flávia Peligrinelli Ribeiro, Nívio Ziviani e Charles Ornelas, Leonardo Rocha, Leonardo Mata

Introdução

- **Pesquisa em memória secundária**: arquivos contém mais registros do que a memória interna pode armazenar.
- Custo para acessar um registro é algumas ordens de grandeza maior do que o custo de processamento na memória primária.
- Medida de complexidade: custo de trasferir dados entre a memória principal e secundária (minimizar o número de transferências).
- Memórias secundárias: apenas um registro pode ser acessado em um dado momento (acesso seqüencial).
- Memórias primárias: acesso a qualquer registro de um arquivo a um custo uniforme (acesso direto).
- Em um método eficiente de pesquisa, o aspecto sistema de computação é importante.
- As características da arquitetura e do sistema operacional da máquina tornam os métodos de pesquisa dependentes de parâmetros que afetam seus desempenhos.

Modelo de Computação para Memória Secundária - Memória Virtual

- Normalmente implementado como uma função do sistema operacional.
- Modelo de armazenamento em dois níveis, devido à necessidade de grandes quantidades de memória e o alto custo da memória principal.
- Uso de uma pequena quantidade de memória principal e uma grande quantidade de memória secundária.
- Programador pode endereçar grandes quantidades de dados, deixando para o sistema a responsabilidade de trasferir o dado da memória secundária para a principal.
- Boa estratégia para algoritmos com pequena localidade de referência.
- Organização do fluxo entre a memória principal e secundária é extremamente importante.

Memória Virtual

- Organização de fluxo $\rightarrow$ transformar o endereço usado pelo programador na localização física de memória correspondente.
- *Espaço de Endereçamento* $\rightarrow$ endereços usados pelo programador.
- *Espaço de Memória* $\rightarrow$ localizações de memória no computador.
- O espaço de endereçamento N e o espaço de memória M podem ser vistos como um mapeamento de endereços do tipo: $f : N \rightarrow M$.
- O mapeamento permite ao programador usar um espaço de endereçamento que pode ser maior que o espaço de memória primária disponível.

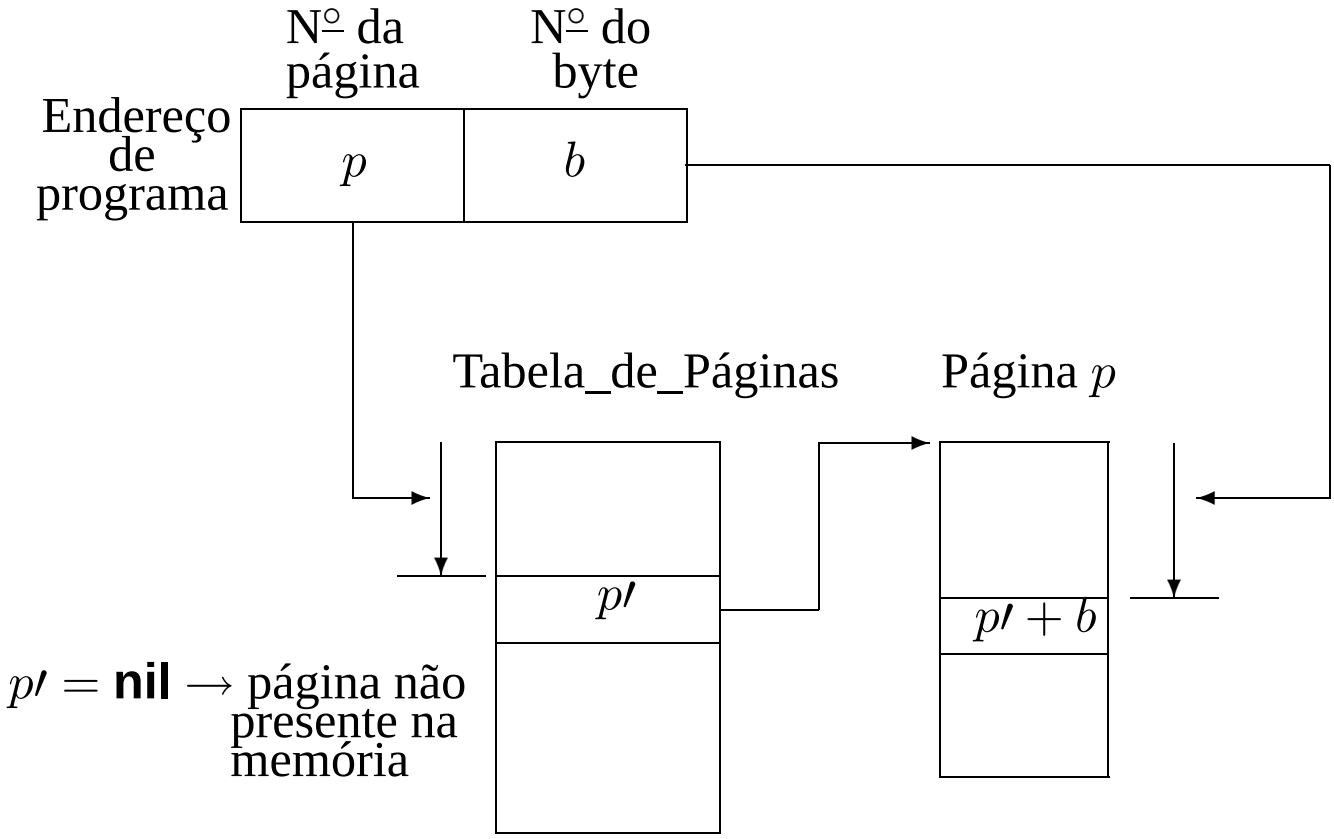
Memória Virtual: Sistema de Paginação

- O espaço de endereçamento é dividido em páginas de tamanho igual, em geral, múltiplos de 512 *bytes*.
- A memória principal é dividida em molduras de páginas de tamanho igual.
- As molduras de páginas contêm algumas páginas ativas enquanto o restante das páginas estão residentes em memória secundária (páginas inativas).
- O mecanismo possui duas funções:
  1. Mapeamento de endereços $\rightarrow$ determinar qual página um programa está endereçando, encontrar a moldura, se existir, que contenha a página.
  2. Transferência de páginas $\rightarrow$ transferir páginas da memória secundária para a memória primária e transferí-las de volta para a memória secundária quando não estão mais sendo utilizadas.

Memória Virtual: Sistema de Paginação

- Endereçamento da página $\rightarrow$ uma parte dos bits é interpretada como um número de página e a outra parte como o número do byte dentro da página (*offset*).
- Mapeamento de endereços $\rightarrow$ realizado através de uma Tabela de Páginas.
  - a p-ésima entrada contém a localização p' da Moldura de Página contendo a página número p desde que esteja na memória principal.
- O mapeamento de endereços é: $f(e) = f(p, b) = p' + b$, onde e é o endereço do programa, p é o número da página e b o número do byte.

Memória Virtual: Mapeamento de Endereços



Memória Virtual: Reposição de Páginas

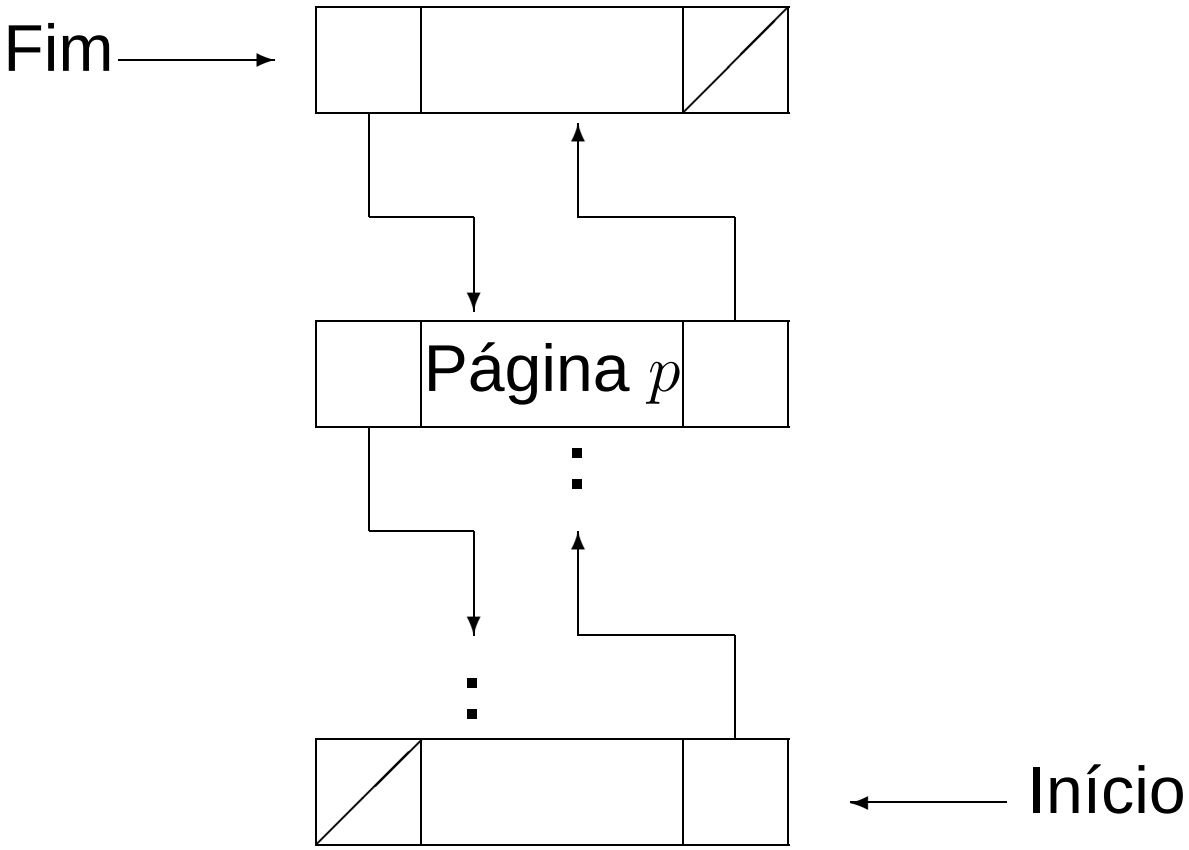
- Se não houver uma moldura de página vazia $\rightarrow$ uma página deverá ser removida da memória principal.
- Ideal $\rightarrow$ remover a página que não será referenciada pelo período de tempo mais longo no futuro.
  - tentamos inferir o futuro a partir do comportamento passado.

Memória Virtual: Políticas de Reposição de Páginas

- **Menos Recentemente Utilizada (LRU):**
  - um dos algoritmos mais utilizados,
  - remove a página menos recentemente utilizada,
  - parte do princípio que o comportamento futuro deve seguir o passado recente.
- **Menos Freqüentemente Utilizada (LFU):**
  - remove a página menos feqüentemente utilizada,
  - inconveniente: uma página recentemente trazida da memória secundária tem um baixo número de acessos registrados e pode ser removida.
- **Ordem de Chegada (FIFO):**
  - remove a página que está residente há mais tempo,
  - algoritmo mais simples e barato de manter,
  - desvantagem: ignora o fato de que a página mais antiga pode ser a mais referenciada.

Memória Virtual: Política LRU

- Toda vez que uma página é utilizada ela é removida para o fim da fila.
- A página que está no início da fila é a página LRU.
- Quando uma nova página é trazida da memória secundária ela deve ser colocada na moldura que contém a página LRU.



Memória Virtual: Estrutura de Dados

```
package cap6.umtipo;
class Registro {
   private short chave;
   // Outros componentes e métodos de um registro
}
class Endereco {
   private short p;
   private byte b; // b ∈ [0, itensPorPagina − 1]
   // Métodos para operar com um endereço
}
class Item {
   private Registro reg;
   private Endereco esq, dir;
   // Métodos para operar com um item
}
public class Pagina {
   private Item pagina[];
   public Pagina (byte itensPorPagina) {
      // itensPorPagina = tamanhoDaPagina/tamanhoDoItem
      this.pagina = new Item[itensPorPagina];
   }
   // Métodos para operar com uma página
}
```

Memória Virtual

- Em casos em que precisamos manipular mais de um arquivo ao mesmo tempo:
  - Deve-se utilizar os mecanismos de **H**erança e **P**olimorfismo de Java que permitem que uma página possa ser definida como vários tipos diferentes.
  - A fila de molduras é única → cada moldura deve ter indicado o arquivo a que se refere aquela página.

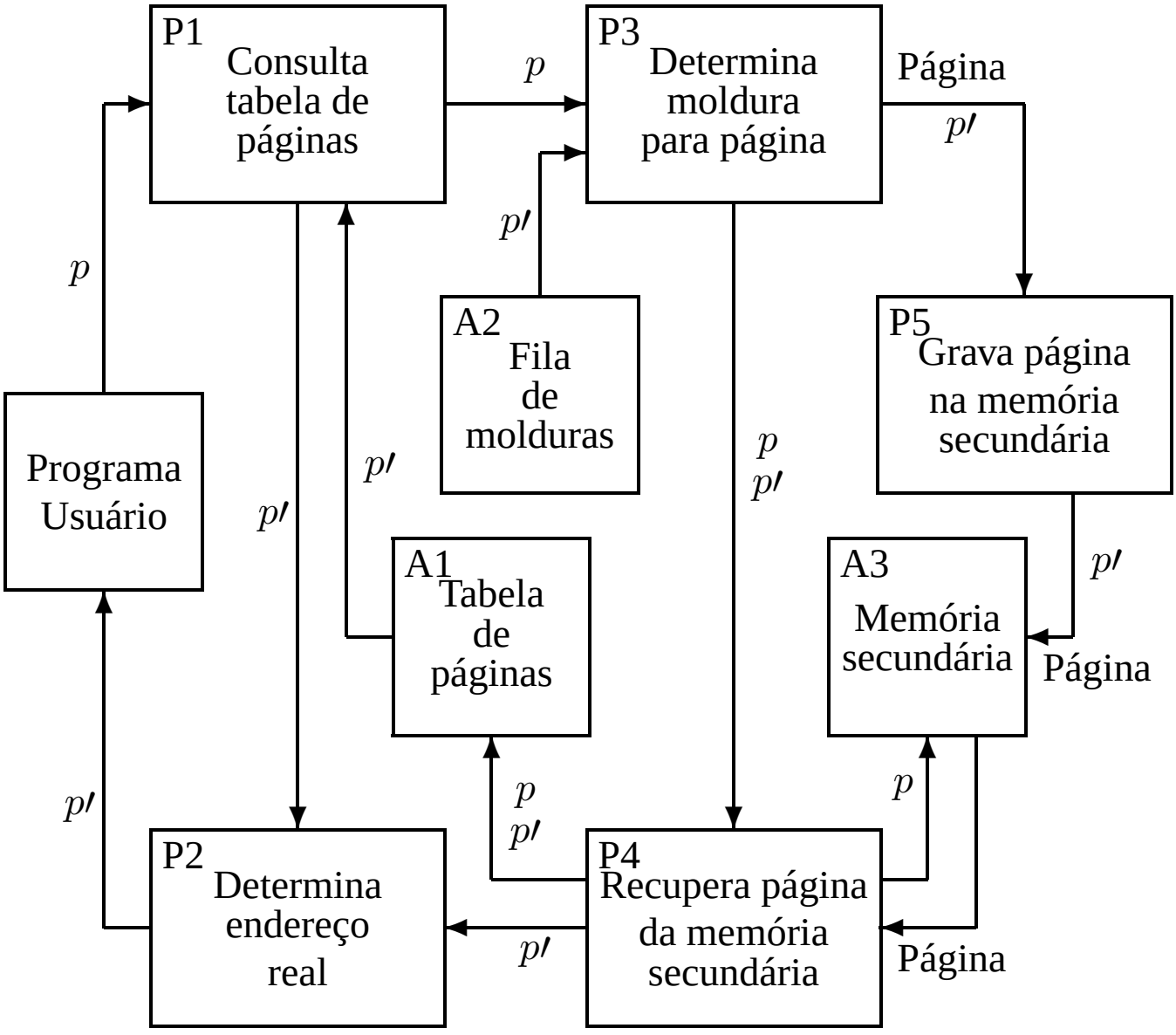
```
package cap6.variostipos;
public abstract class Pagina {
   // Componentes e métodos de uma página
}
class PaginaA extends Pagina {
   // Componentes e métodos de uma página do tipo A
}
class PaginaB extends Pagina {
   // Componentes e métodos de uma página do tipo B
}
class PaginaC extends Pagina {
   // Componentes e métodos de uma página do tipo C
}
```

Memória Virtual

- Procedimentos para comunicação com o sistema de paginação:
  - $obtemRegistro \rightarrow$ torna disponível um registro.
  - $escreveRegistro \rightarrow$ permite criar ou alterar o conteúdo de um registro.
  - $descarregaPaginas \rightarrow$ varre a fila de molduras para atualizar na memória secundária todas as páginas que tenham sido modificadas.

Memória Virtual - Transformação do Endereço Virtual para Real

- Quadrados → resultados de processos ou arquivos.
- Retângulos → processos transformadores de informação.



Acesso Seqüencial Indexado

- Utiliza o princípio da pesquisa seqüencial → cada registro é lido seqüencialmente até encontrar uma chave maior ou igual a chave de pesquisa.
- Providências necessárias para aumentar a eficiência:
  - o arquivo deve ser mantido ordenado pelo campo chave do registro,
  - um arquivo de índices contendo pares de valores $< x, p >$ deve ser criado, onde x representa uma chave e p representa o endereço da página na qual o primeiro registro contém a chave x.
  - Estrutura de um arquivo seqüencial indexado para um conjunto de 15 registros:

3	14	25	41
1	2	3	4

1 | 3 5 7 11 | 2 | 14 17 20 21 | 3 | 25 29 32 36 | 4 | 41 44 48 |

Acesso Seqüencial Indexado: Disco Magnético

- Dividido em círculos concêntricos (trilhas).
- Cilindro → todas as trilhas verticalmente alinhadas e que possuem o mesmo diâmetro.
- Latência rotacional → tempo necessário para que o início do bloco contendo o registro a ser lido passe pela cabeça de leitura/gravação.
- Tempo de busca (*seek time*) → tempo necessário para que o mecanismo de acesso desloque de uma trilha para outra (maior parte do custo para acessar dados).
- Acesso seqüencial indexado = acesso indexado + organização seqüencial,
- Aproveitando características do disco magnético e procurando minimizar o número de deslocamentos do mecanismo de acesso → esquema de índices de cilindros e de páginas.

Acesso Seqüencial Indexado: Disco Magnético

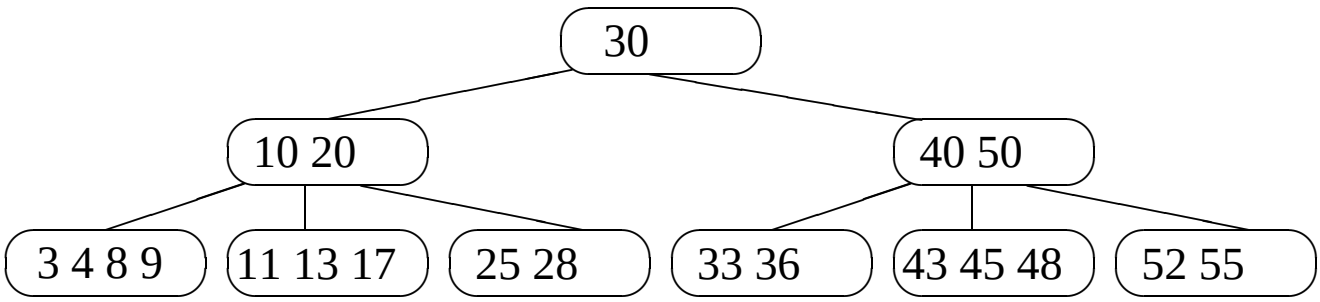
- Para localizar o registro que contenha uma chave de pesquisa são necessários os seguintes passos:
  1. localize o cilindro correspondente à chave de pesquisa no índice de cilindros;
  2. desloque o mecanismo de acesso até o cilindro correspondente;
  3. leia a página que contém o índice de páginas daquele cilindro;
  4. leia a página de dados que contém o registro desejado.

Acesso Seqüencial Indexado: Discos Óticos de Apenas-Leitura (CD-ROM)

- Grande capacidade de armazenamento (600 MB) e baixo custo para o usuário final.
- Informação armazenada é estática.
- A eficiência na recuperação dos dados é afetada pela localização dos dados no disco e pela seqüência com que são acessados.
- Velocidade linear constante $\rightarrow$ trilhas possuem capacidade variável e tempo de latência rotacional varia de trilha para trilha.
- A trilha tem forma de uma espiral contínua.
- Tempo de busca: acesso a trilhas mais distantes demanda mais tempo que no disco magnético. Há necessidade de deslocamento do mecanismo de acesso e mudanças na rotação do disco.
- Varredura estática: acessa conjunto de trilhas vizinhas sem deslocar mecanismo de leitura.
- Estrutura seqüencial implementada mantendo-se um índice de cilindros na memória principal.

Árvores B

- Árvores n-árias: mais de um registro por nodo.
- Em uma árvore B de ordem m:
  - página raiz: 1 e $2m$ registros.
  - demais páginas: no mínimo m registros e $m+1$ descendentes e no máximo $2m$ registros e $2m+1$ descendentes.
  - páginas folhas: aparecem todas no mesmo nível.
- Os registros aparecem em ordem crescente da esquerda para a direita.
- Extensão natural da árvore binária de pesquisa.
- Árvore B de ordem $m = 2$ com três níveis:



Árvores B - Estrutura e operações do dicionário para árvore B

- A estrutura de dados árvore B será utilizada para implementar o tipo abstrato de dados Dicionário e suas operações: $inicializa$, $pesquisa$, $insere$ e $retira$.
- A operação $inicializa$ é implementada pelo construtor da classe $ArvoreB$. As demais operações são descritas a seguir.
- A operação $pesquisa$ é implementada por um método privado sobrecarregado. Este método é semelhante ao método $pesquisa$ para a árvore binária de pesquisa.

Árvores B - Estrutura e operações do dicionário para árvore B

```
package cap6;
import cap4.Item; // vide Programa do capítulo 4
public class ArvoreB {
  private static class Pagina {
    int n; Item r[]; Pagina p[];
    public Pagina (int mm) {
      this.n = 0; this.r = new Item[mm];
      this.p = new Pagina[mm+1];
    }
  }
  private Pagina raiz;
  private int m, mm;
  // Entra aqui o método privado da transparência 21
  public ArvoreB (int m) {
    this.raiz = null; this.m = m; this.mm = 2*m;
  }
  public Item pesquisa (Item reg) {
    return this.pesquisa (reg, this.raiz);
  }
  public void insere (Item reg) { vide transparências 24 e
25 }

  public void retira (Item reg) { vide transparências 30, 31
e 32 }
}
```

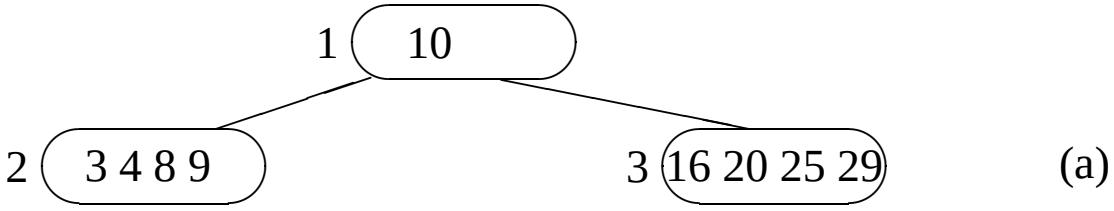
Árvores B - Método para pesquisar na árvore B

```
private Item pesquisa (Item reg, Pagina ap) {
   if (ap == null) return null; // Registro não encontrado
   else {
      int i = 0;
      while ((i < ap.n−1) && (reg.compara (ap.r[i]) > 0)) i++;
      if (reg.compara (ap.r[i]) == 0) return ap.r[i];
      else if (reg.compara (ap.r[i]) < 0)
                return pesquisa (reg, ap.p[i]);
      else return pesquisa (reg, ap.p[i+1]);
   }
}
```

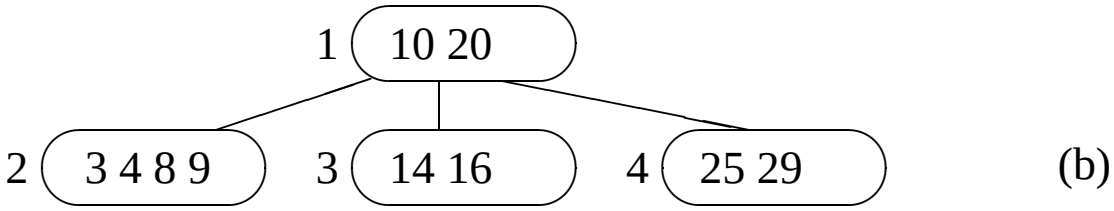
Árvores B - Inserção

1. Localizar a página apropriada aonde o regisro deve ser inserido.
2. Se o registro a ser inserido encontra uma página com menos de $2m$ registros, o processo de inserção fica limitado à página.
3. Se o registro a ser inserido encontra uma página cheia, é criada uma nova página, no caso da página pai estar cheia o processo de divisão se propaga.

Exemplo: Inserindo o registro com chave 14.



(a)

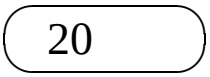


(b)

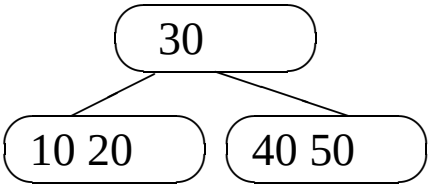
Árvores B - Inserção

Exemplo de inserção das chaves: 20, 10, 40, 50, 30, 55, 3, 11, 4, 28, 36, 33, 52, 17, 25, 13, 45, 9, 43, 8 e 48

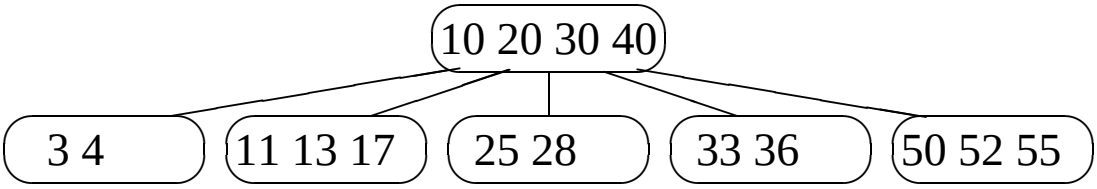
(a)



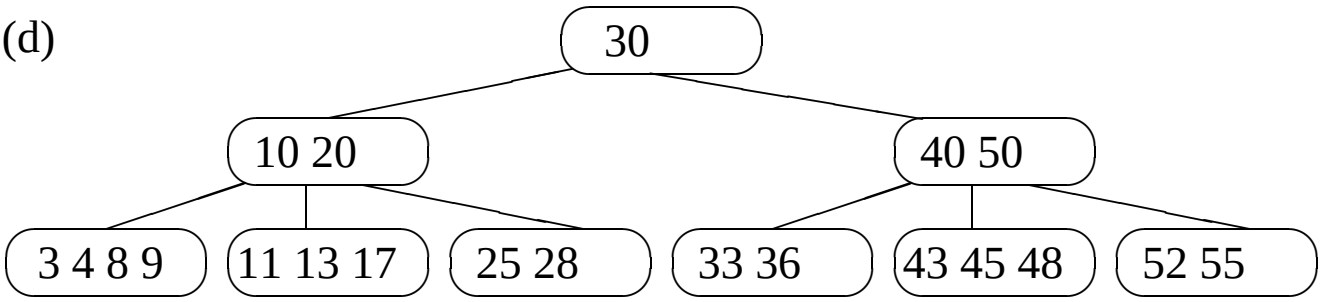
(b)



(c)



(d)



Árvores B - Método $insereNaPagina$

```
private void insereNaPagina
                    (Pagina ap, Item reg, Pagina apDir) {
  int k = ap.n - 1;
  while ((k >= 0) && (reg.compara (ap.r[k]) < 0)) {
    ap.r[k+1] = ap.r[k]; ap.p[k+2] = ap.p[k+1]; k--;
  }
  ap.r[k+1] = reg; ap.p[k+2] = apDir; ap.n++;
}
```

Árvores B - Refinamento final do método *insere*

```
public void insere (Item reg) {
  Item regRetorno[] = new Item[1];
  boolean cresceu[] = new boolean[1];
  Pagina apRetorno = this.insere (reg, this.raiz, regRetorno, cresceu);
  if (cresceu[0]) {
    Pagina apTemp = new Pagina(this.mm);
    apTemp.r[0] = regRetorno[0];
    apTemp.p[0] = this.raiz;
    apTemp.p[1] = apRetorno;
    this.raiz = apTemp; this.raiz.n++;
  } else this.raiz = apRetorno;
}
private Pagina insere (Item reg, Pagina ap, Item[] regRetorno,
                       boolean[] cresceu) {
  Pagina apRetorno = null;
  if (ap == null) { cresceu[0] = true; regRetorno[0] = reg; }
  else {
    int i = 0;
    while ((i < ap.n-1) && (reg.compara (ap.r[i]) > 0)) i++;
    if (reg.compara (ap.r[i]) == 0) {
      System.out.println ("Erro: Registro ja existente");
      cresceu[0] = false;
    }
    else {
      if (reg.compara (ap.r[i]) > 0) i++;
      apRetorno = insere (reg, ap.p[i], regRetorno, cresceu);
      if (cresceu[0])
        if (ap.n < this.mm) { // Página tem espaço
          this.insereNaPagina (ap, regRetorno[0], apRetorno);
          cresceu[0] = false; apRetorno = ap;
        }

        // Continua na próxima transparência
```

Árvores B - Refinamento final do método *insere*

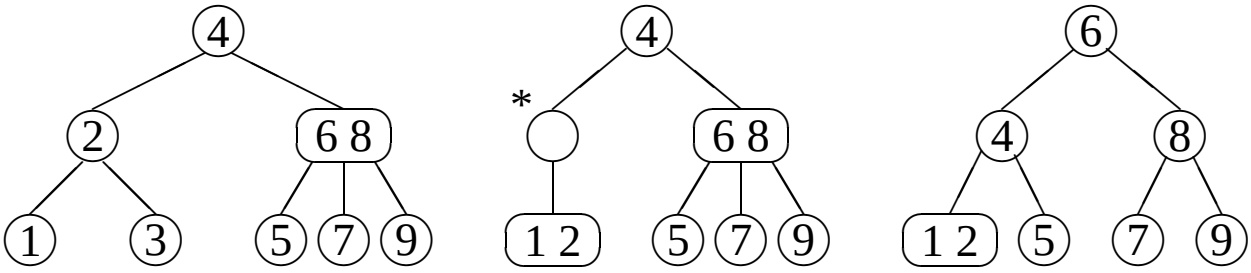
```
        else { // Overflow: Página tem que ser dividida
          Pagina apTemp = new Pagina (this.mm); apTemp.p[0] = null;
          if (i <= this.m) {
            this.insereNaPagina (apTemp,ap.r[this.mm-1],ap.p[this.mm]);
            ap.n--;
            this.insereNaPagina (ap, regRetorno[0], apRetorno);
          } else this.insereNaPagina (apTemp, regRetorno[0], apRetorno);
          for (int j = this.m+1; j < this.mm; j++) {
            this.insereNaPagina (apTemp, ap.r[j], ap.p[j+1]);
            ap.p[j+1] = null; // transfere a posse da memória
          }
          ap.n = this.m; apTemp.p[0] = ap.p[this.m+1];
          regRetorno[0] = ap.r[this.m]; apRetorno = apTemp;
        }
      }
    }
    return (cresceu[0] ? apRetorno : ap);
}
```

Árvores B - Remoção

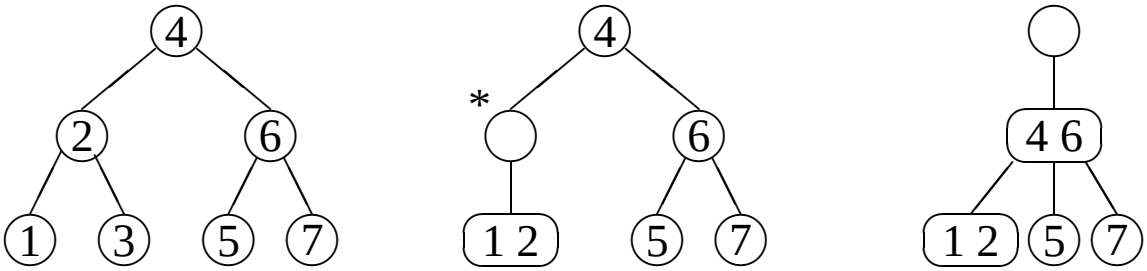
- Página com o registro a ser retirado é folha:
  1. retira-se o registro,
  2. se a página não possui pelo menos de m registros, a propriedade da árvore B é violada. Pega-se um registro emprestado da página vizinha. Se não existir registros suficientes na página vizinha, as duas páginas devem ser fundidas em uma só.
- Pagina com o registro não é folha:
  1. o registro a ser retirado deve ser primeiramente substituído por um registro contendo uma chave adjacente.

Árvores B - Remoção

Exemplo: Retirando a chave 3.



(a) Página vizinha possui mais do que m registros

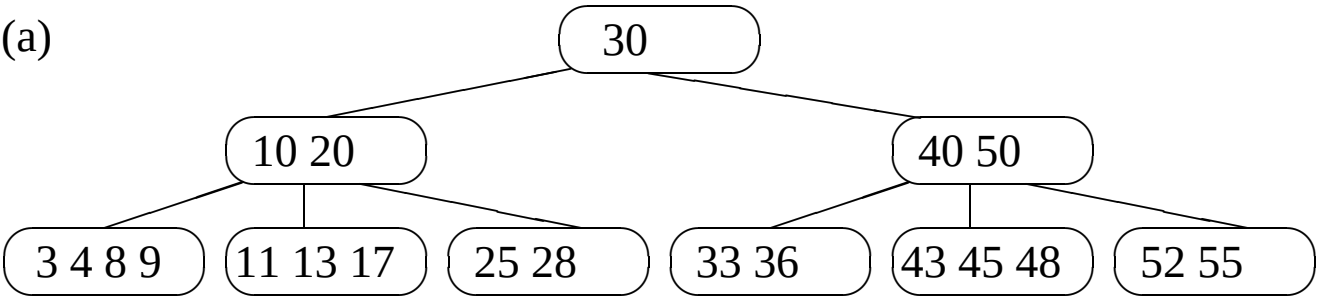


(b) Página vizinha possui exatamente m registros

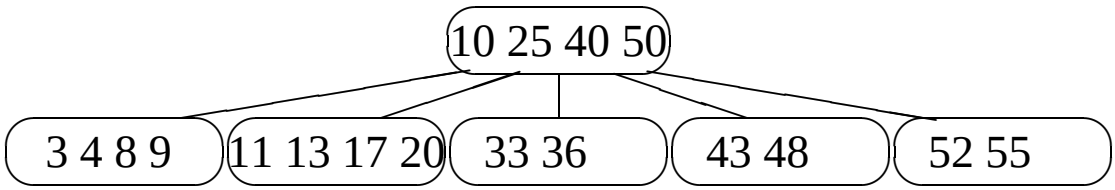
Árvores B - Remoção

Exemplo de remoção das chaves 45 30 28; 50 8 10 4 20 40 55 17 33 11 36; 3 9 52.

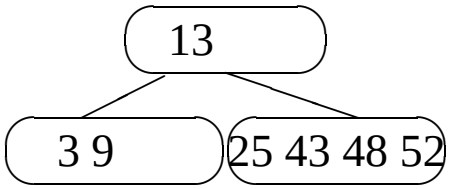
(a)



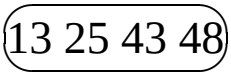
(b)



(c)



(d)



Árvores B - Operação *retira*

```
public void retira (Item reg) {
  boolean diminuiu[] = new boolean[1];
  this.raiz = this.retira (reg, this.raiz, diminuiu);
  if (diminuiu[0] && (this.raiz.n == 0)) { // Árvore diminui na altura
    this.raiz = this.raiz.p[0];
  }
}
private Pagina retira (Item reg, Pagina ap, boolean[] diminuiu) {
  if (ap == null) {
    System.out.println ("Erro: Registro nao encontrado");
    diminuiu[0] = false;
  }
  else {
    int ind = 0;
    while ((ind < ap.n-1) && (reg.compara (ap.r[ind]) > 0)) ind++;
    if (reg.compara (ap.r[ind]) == 0) { // achou
      if (ap.p[ind] == null) { // Página folha
        ap.n--; diminuiu[0] = ap.n < this.m;
        for (int j = ind; j < ap.n; j++) {
          ap.r[j] = ap.r[j+1]; ap.p[j] = ap.p[j+1];
        }
        ap.p[ap.n] = ap.p[ap.n+1];
        ap.p[ap.n+1] = null; // transfere a posse da memória
      }
      else { // Página não é folha: trocar com antecessor
        diminuiu[0] = antecessor (ap, ind, ap.p[ind]);
        if (diminuiu[0]) diminuiu[0] = reconstitui (ap.p[ind], ap, ind);
      }
    }
    else { // não achou
      if (reg.compara (ap.r[ind]) > 0) ind++;
      ap.p[ind] = retira (reg, ap.p[ind], diminuiu);
      if (diminuiu[0]) diminuiu[0] = reconstitui (ap.p[ind], ap, ind);
    }
  }
  return ap;
}
```

Árvores B - Método *antecessor* utilizado no método *retira*

```
private boolean antecessor(Pagina ap, int ind, Pagina apPai) {
  boolean diminuiu = true;
  if (apPai.p[apPai.n] != null) {
    diminuiu = antecessor (ap, ind, apPai.p[apPai.n]);
    if (diminuiu)
      diminuiu=reconstitui (apPai.p[apPai.n],apPai,apPai.n);
  }
  else {
    ap.r[ind] = apPai.r[--apPai.n];
    diminuiu = apPai.n < this.m;
  }
  return diminuiu;
}
```

Árvores B - Método *reconstitui* utilizado no método *retira*

```
private boolean reconstitui (Pagina apPag, Pagina apPai, int posPai) {
  boolean diminuiu = true;
  if (posPai < apPai.n) { // aux = Página à direita de apPag
    Pagina aux = apPai.p[posPai+1];
    int dispAux = (aux.n – this.m + 1)/2;
    apPag.r[apPag.n++] = apPai.r[posPai]; apPag.p[apPag.n] = aux.p[0];
    aux.p[0] = null; // transfere a posse da memória
    if (dispAux > 0) { // Existe folga: transfere de aux para apPag
      for (int j = 0; j < dispAux – 1; j++) {
        this.insereNaPagina (apPag, aux.r[j], aux.p[j+1]);
        aux.p[j+1] = null; // transfere a posse da memória
      }
      apPai.r[posPai] = aux.r[dispAux – 1];
      aux.n = aux.n – dispAux;
      for (int j = 0;  j < aux.n; j++) aux.r[j] = aux.r[j+dispAux];
      for (int j = 0; j <= aux.n; j++) aux.p[j] = aux.p[j+dispAux];
      aux.p[aux.n+dispAux] = null; // transfere a posse da memória
      diminuiu = false;
    }
    else { // Fusão: intercala aux em apPag e libera aux
      for (int j = 0; j < this.m; j++) {
        this.insereNaPagina (apPag, aux.r[j], aux.p[j+1]);
        aux.p[j+1] = null; // transfere a posse da memória
      }
      aux = apPai.p[posPai+1] = null; // libera aux
      for (int j = posPai; j < apPai.n–1; j++) {
        apPai.r[j] = apPai.r[j+1]; apPai.p[j+1] = apPai.p[j+2];
      }
      apPai.p[apPai.n––] = null; // transfere a posse da memória
      diminuiu = apPai.n < this.m;
    }
  }
```

// *Continua na próxima transparência*

Árvores B - Método *reconstitui* utilizado no método *retira*

```
  else { // aux = Página à esquerda de apPag
    Pagina aux = apPai.p[posPai-1];
    int dispAux = (aux.n - this.m + 1)/2;
    for (int j = apPag.n-1; j >= 0; j--) apPag.r[j+1] = apPag.r[j];
    apPag.r[0] = apPai.r[posPai-1];
    for (int j = apPag.n; j >= 0; j--) apPag.p[j+1] = apPag.p[j];
    apPag.n++;
    if (dispAux > 0) { // Existe folga: transfere de aux para apPag
      for (int j = 0; j < dispAux - 1; j++) {
        this.insereNaPagina (apPag, aux.r[aux.n-j-1], aux.p[aux.n-j]);
        aux.p[aux.n-j] = null; // transfere a posse da memória
      }
      apPag.p[0] = aux.p[aux.n - dispAux + 1];
      aux.p[aux.n - dispAux + 1] = null; // transfere a posse da memória
      apPai.r[posPai-1] = aux.r[aux.n - dispAux];
      aux.n = aux.n - dispAux; diminuiu = false;
    }
    else { // Fusão: intercala apPag em aux e libera apPag
      for (int j = 0; j < this.m; j++) {
        this.insereNaPagina (aux, apPag.r[j], apPag.p[j+1]);
        apPag.p[j+1] = null; // transfere a posse da memória
      }
      apPag = null; // libera apPag
      apPai.p[apPai.n--] = null; // transfere a posse da memória
      diminuiu = apPai.n < this.m;
    }
  }
  return diminuiu;
}
```

Árvores B* - Estrutura e operações do dicionário para árvore B*

```
package cap6;
import cap4.Item; // vide Programa do capítulo 4
public class ArvoreBEstrela  {
  private static abstract class Pagina {
    int n; Item chaves[];
  }
  private static class PaginaInt extends Pagina {
    Pagina p[];
    public PaginaInt (int mm) {
      this.n = 0; this.chaves = new Item[mm];
      this.p = new Pagina[mm+1];
    }
  }
  private static class PaginaExt extends Pagina {
    Object registros[];
    public PaginaExt (int mm2) {
      this.n = 0; this.chaves = new Item[mm2];
      this.registros = new Object[mm2];
    }
  }
  private Pagina raiz;
  private int mm, mm2;

  // Entram aqui os métodos privados apresentados na transparência 36
  public ArvoreBEstrela (int mm, int mm2) {
    this.raiz = null; this.mm = mm; this.mm2 = mm2;
  }
  public Object pesquisa (Item chave) {
    return this.pesquisa (chave, this.raiz);
  }
}
```

Árvores B* - Pesquisa

- Semelhante à pesquisa em árvore B,
- A pesquisa sempre leva a uma página folha,
- A pesquisa não pára se a chave procurada for encontrada em uma página índice. O apontador da direita é seguido até que se encontre uma página folha.

Árvores B* - Método para pesquisar na árvore B*

```
private Object pesquisa (Item chave, Pagina ap) {
  if (ap == null) return null; // Registro não encontrado
  else {
    if (this.eInterna (ap)) {
      int i = 0; PaginaInt aux = (PaginaInt)ap;
      while ((i < aux.n-1) && (chave.compara (aux.chaves[i]) > 0)) i++;
      if (chave.compara (aux.chaves[i]) < 0)
        return pesquisa (chave, aux.p[i]);
      else return pesquisa (chave, aux.p[i+1]);
    }
    else {
      int i = 0; PaginaExt aux = (PaginaExt)ap;
      while ((i < aux.n-1) && (chave.compara (aux.chaves[i]) > 0)) i++;
      if (chave.compara (aux.chaves[i]) == 0) return aux.registros[i];
      return null; // Registro não encontrado
    }
  }
}

private boolean eInterna (Pagina ap) {
  Class classe = ap.getClass ();
  return classe.getName().equals(PaginaInt.class.getName());
}
```

Árvores B* - Inserção e Remoção

- Inserção na árvore B*
  - Semelhante à inserção na árvore B,
  - Diferença: quando uma folha é dividida em duas, o algoritmo promove uma cópia da chave que pertence ao registro do meio para a página pai no nível anterior, retendo o registro do meio na página folha da direita.
- Remoção na árvore B*
  - Relativamente mais simples que em uma árvore B,
  - Todos os registros são folhas,
  - Desde que a folha fique com pelo menos metade dos registros, as páginas dos índices não precisam ser modificadas, mesmo se uma cópia da chave que pertence ao registro a ser retirado esteja no índice.

Acesso Concorrente em Árvore B*

- Acesso simultâneo a banco de dados por mais de um usuário.
- Concorrência aumenta a utilização e melhora o tempo de resposta do sistema.
- O uso de árvores B* nesses sistemas deve permitir o processamento simultâneo de várias solicitações diferentes.
- Necessidade de criar mecanismos chamados protocolos para garantir a integridade tanto dos dados quanto da estrutura.
- Página segura: não há possibilidade de modificações na estrutura da árvore como conseqüência de inserção ou remoção.
  - inserção $\rightarrow$ página segura se o número de chaves é igual a $2m$,
  - remoção $\rightarrow$ página segura se o número de chaves é maior que m.
- Os algoritmos para acesso concorrente fazem uso dessa propriedade para aumentar o nível de concorrência.

Acesso Concorrente em Árvore B* - Protocolos de Travamentos

- Quando uma página é lida, a operação de recuperação a trava, assim, outros processos, não podem interferir com a página.
- A pesquisa continua em direção ao nível seguinte e a trava é liberada para que outros processos possam ler a página .
- Processo leitor $\rightarrow$ executa uma operação de recuperação
- Processo modificador $\rightarrow$ executa uma operação de inserção ou retirada.
- Dois tipos de travamento:
  - Travamento para leitura $\rightarrow$ permite um ou mais leitores acessarem os dados, mas não permite inserção ou retirada.
  - Travamento exclusivo $\rightarrow$ nenhum outro processo pode operar na página e permite qualquer tipo de operação na página.

Árvore B - Considerações Práticas

- Simples, fácil manutenção, eficiente e versátil.
- Permite acesso seqüencial eficiente.
- Custo para recuperar, inserir e retirar registros do arquivo é logaritmico.
- Espaço utilizado é, no mínimo 50% do espaço reservado para o arquivo,
- Emprego onde o acesso concorrente ao banco de dados é necessário, é viável e relativamente simples de ser implementado.
- Inserção e retirada de registros sempre deixam a árvore balanceada.
- Uma árvore B de ordem m com N registros contém no máximo cerca de $log_{m+1}N$ páginas.

Árvore B - Considerações Práticas

- Limites para a altura máxima e mínima de uma árvore B de ordem m com N registros: $\log_{2m+1}(N+1) \leq altura \leq 1 + \log_{m+1}\left(\frac{N+1}{2}\right)$
- Custo para processar uma operação de recuperação de um registro cresce com o logaritmo base m do tamanho do arquivo.
- Altura esperada: não é conhecida analiticamente.
- Há uma conjectura proposta a partir do cálculo analítico do número esperado de páginas para os quatro primeiros níveis (das folha em direção à raiz) de uma **árvore 2-3** (árvore B de ordem $m = 1$).
- Conjetura: a altura esperada de uma árvore 2-3 **randômica** com N chaves é $\overline{h}(N) \approx \log_{7/3}(N+1)$.

Árvores B Randômicas - Outras Medidas de Complexidade

- A utilização de memória é cerca de $\ln 2$.
  - Páginas ocupam $\approx$ 69% da área reservada após N inserções randômicas em uma árvore B inicialmente vazia.
- No momento da inserção, a operação mais cara é a partição da página quando ela passa a ter mais do que $2m$ chaves. Envolve:
  - Criação de nova página, rearranjo das chaves e inserção da chave do meio na página pai localizada no nível acima.
  - $Pr\{$j partições$\}$: probabilidade de que j partições ocorram durante a N-ésima inserção randômica.
  - Árvore 2-3: $Pr\{$0 partições$\} = \frac{4}{7}$, $Pr\{$1 ou mais partições$\} = \frac{3}{7}$.
  - Árvore B de ordem m: $Pr\{$0 partições$\} = 1 - \frac{1}{(2\ln 2)m} + O(m^{-2})$, $Pr\{$1 ou + partições$\} = \frac{1}{(2\ln 2)m} + O(m^{-2})$.
  - Árvore B de ordem $m = 70$: 99% das vezes nada acontece em termos de partições durante uma inserção.

Árvores B Randômicas - Acesso Concorrente

- Foi proposta uma técnica de aplicar um travamento na *página segura mais profunda* (Psmp) no caminho de inserção.
- Uma página é **segura** se ela contém menos do que $2m$ chaves.
- Uma página segura é a mais profunda se não existir outra página segura abaixo dela.
- Já que o travamento da página impede o acesso de outros processos, é interessante saber qual é a probabilidade de que a página segura mais profunda esteja no primeiro nível.
- Árvore 2-3: $Pr\{$Psmp esteja no $1^{\underline{o}}$ nível$\} = \frac{4}{7}$, $Pr\{$Psmp esteja acima do $1^{\underline{o}}$ nível$\} = \frac{3}{7}$.
- Árvore B de ordem m:
  $Pr\{$Psmp esteja no $1^{\underline{o}}$ nível$\} =$
  $1 - \frac{1}{(2 \ln 2)m} + O(m^{-2})$,
  $Pr\{$Psmp esteja acima do $1^{\underline{o}}$ nível$\} = \frac{3}{7} =$
  $\frac{1}{(2 \ln 2)m} + O(m^{-2})$.

Árvores B Randômicas - Acesso Concorrente

- Novamente, em árvores B de ordem $m = 70$: 99% das vezes a Psmp está em uma folha. (Permite alto grau de concorrência para processos modificadores.)
- Soluções muito complicadas para permitir concorrência de operações em árvores B não trazem grandes benefícios.
- Na maioria das vezes, o travamento ocorrerá em páginas folha. (Permite alto grau de concorrência mesmo para os protocolos mais simples.)

Árvore B - Técnica de Transbordamento (ou Overflow)

- Assuma que um registro tenha de ser inserido em uma página cheia, com $2m$ registros.
- Em vez de particioná-la, olhamos primeiro para a página irmã à direita.
- Se a página irmã possui menos do que $2m$ registros, um simples rearranjo de chaves torna a partição desnecessária.
- Se a página à direita também estiver cheia ou não existir, olhamos para a página irmã à esquerda.
- Se ambas estiverem cheias, então a partição terá de ser realizada.
- Efeito da modificação: produzir uma árvore com melhor utilização de memória e uma altura esperada menor.
- Produz uma utilização de memória de cerca de 83% para uma árvore B randômica.

Árvore B - Influência do Sistema de Paginação

- O número de níveis de uma árvore B é muito pequeno (três ou quatro) se comparado com o número de molduras de páginas.
- Assim, o sistema de paginação garante que a página raiz esteja sempre na memória principal (se for adotada a política LRU).
- O esquema LRU faz também com que as páginas a serem particionadas em uma inserção estejam automaticamente disponíveis na memória principal.
- A escolha do tamanho adequado da ordem m da árvore B é geralmente feita levando em conta as características de cada computador.
- O tamanho ideal da página da árvore corresponde ao tamanho da página do sistema, e a transferência de dados entre as memórias secundária e principal é realizada pelo sistema operacional.
- Estes tamanhos variam entre 512 *bytes* e 4.096 *bytes*, em múltiplos de 512 *bytes*.

Algoritmos em Grafos*

Última alteração: 10 de Outubro de 2006

*Transparências elaboradas por Charles Ornelas, Leonardo Rocha, Leonardo Mata e Nivio Ziviani

Motivação

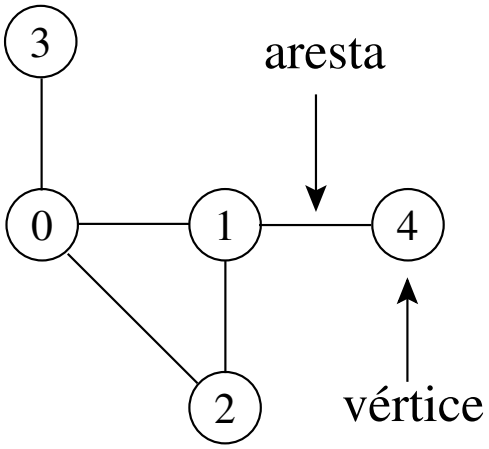
- Muitas aplicações em computação necessitam considerar conjunto de conexões entre pares de objetos:
  - Existe um caminho para ir de um objeto a outro seguindo as conexões?
  - Qual é a menor distância entre um objeto e outro objeto?
  - Quantos outros objetos podem ser alcançados a partir de um determinado objeto?
- Existe um tipo abstrato chamado grafo que é usado para modelar tais situações.

Aplicações

- Alguns exemplos de problemas práticos que podem ser resolvidos através de uma modelagem em grafos:
  - Ajudar máquinas de busca a localizar informação relevante na Web.
  - Descobrir os melhores casamentos entre posições disponíveis em empresas e pessoas que aplicaram para as posições de interesse.
  - Descobrir qual é o roteiro mais curto para visitar as principais cidades de uma região turística.

Conceitos Básicos

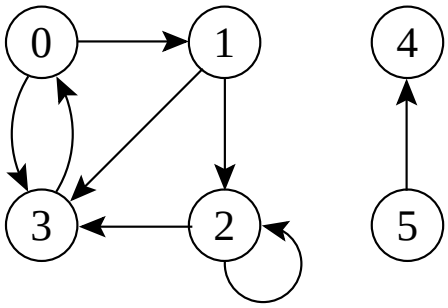
- **Grafo**: conjunto de vértices e arestas.
- **Vértice**: objeto simples que pode ter nome e outros atributos.
- **Aresta**: conexão entre dois vértices.



- Notação: $G = (V, A)$
  - G: grafo
  - V: conjunto de vértices
  - A: conjunto de arestas

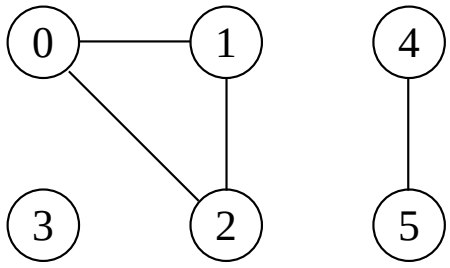
Grafos Direcionados

- Um grafo direcionado G é um par (V, A), onde V é um conjunto finito de vértices e A é uma relação binária em V.
  - Uma aresta (u, v) sai do vértice u e entra no vértice v. O vértice v é **adjacente** ao vértice u.
  - Podem existir arestas de um vértice para ele mesmo, chamadas de *self-loops*.



Grafos Não Direcionados

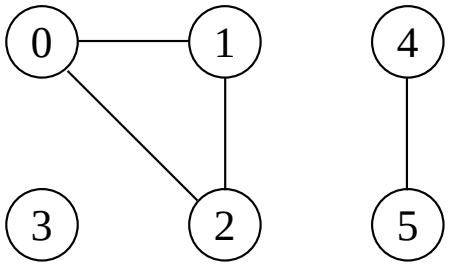
- Um grafo não direcionado G é um par (V, A), onde o conjunto de arestas A é constituído de pares de vértices não ordenados.
  - As arestas (u, v) e (v, u) são consideradas como uma única aresta. A relação de adjacência é simétrica.
  - *Self-loops* não são permitidos.



Grau de um Vértice

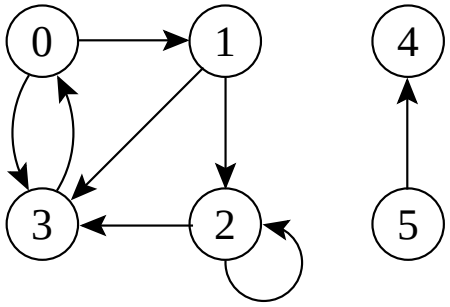
- Em grafos não direcionados:
  - O grau de um vértice é o número de arestas que incidem nele.
  - Um vérice de grau zero é dito **isolado** ou **não conectado**.

Ex.: O vértice 1 tem grau 2 e o vértice 3 é isolado.



- Em grafos direcionados
  - O grau de um vértice é o número de arestas que saem dele (*out-degree*) mais o número de arestas que chegam nele (*in-degree*).

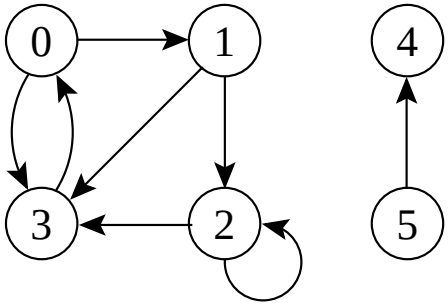
Ex.: O vértice 2 tem *in-degree* 2, *out-degree* 2 e grau 4.



Caminho entre Vértices

- Um caminho de **comprimento** k de um vértice x a um vértice y em um grafo $G = (V, A)$ é uma seqüência de vértices $(v_0, v_1, v_2, \ldots, v_k)$ tal que $x = v_0$ e $y = v_k$, e $(v_{i-1}, v_i) \in A$ para $i = 1, 2, \ldots, k$.
- O comprimento de um caminho é o número de arestas nele, isto é, o caminho contém os vértices $v_0, v_1, v_2, \ldots, v_k$ e as arestas $(v_0, v_1), (v_1, v_2), \ldots, (v_{k-1}, v_k)$.
- Se existir um caminho c de x a y então y é **alcançável** a partir de x via c.
- Um caminho é **simples** se todos os vértices do caminho são distintos.

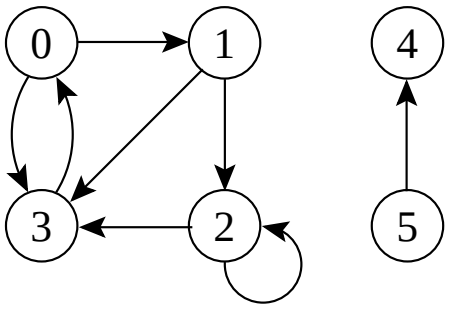
Ex.: O caminho $(0, 1, 2, 3)$ é simples e tem comprimento 3. O caminho $(1, 3, 0, 3)$ não é simples.



Ciclos

- Em um grafo direcionado:
  - Um caminho $(v_0, v_1, \ldots, v_k)$ forma um ciclo se $v_0 = v_k$ e o caminho contém pelo menos uma aresta.
  - O ciclo é simples se os vértices $v_1, v_2, \ldots, v_k$ são distintos.
  - O *self-loop* é um ciclo de tamanho 1.
  - Dois caminhos $(v_0, v_1, \ldots, v_k)$ e $(v'_0, v'_1, \ldots, v'_k)$ formam o mesmo ciclo se existir um inteiro j tal que $v'_i = v_{(i+j) \bmod k}$ para $i = 0, 1, \ldots, k-1$.

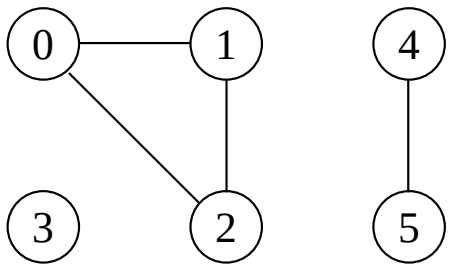
Ex.: O caminho $(0, 1, 2, 3, 0)$ forma um ciclo. O caminho$(0, 1, 3, 0)$ forma o mesmo ciclo que os caminhos $(1, 3, 0, 1)$ e $(3, 0, 1, 3)$.



Ciclos

- Em um grafo não direcionado:
  - Um caminho $(v_0, v_1, \ldots, v_k)$ forma um ciclo se $v_0 = v_k$ e o caminho contém pelo menos três arestas.
  - O ciclo é simples se os vértices $v_1, v_2, \ldots, v_k$ são distintos.

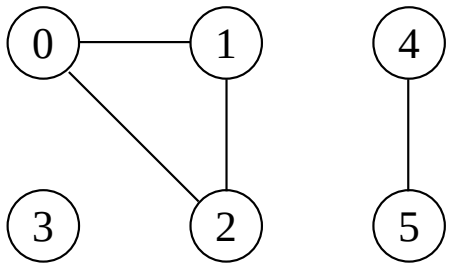
Ex.: O caminho $(0, 1, 2, 0)$ é um ciclo.



Componentes Conectados

- Um grafo não direcionado é conectado se cada par de vértices está conectado por um caminho.
- Os componentes conectados são as porções conectadas de um grafo.
- Um grafo não direcionado é conectado se ele tem exatamente um componente conectado.

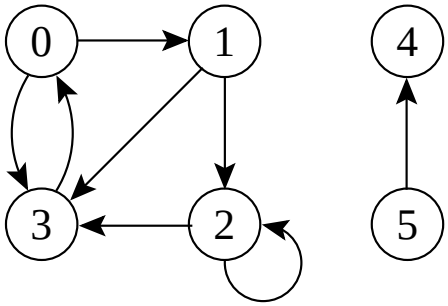
Ex.: Os componentes são: $\{0, 1, 2\}$, $\{4, 5\}$ e $\{3\}$.



Componentes Fortemente Conectados

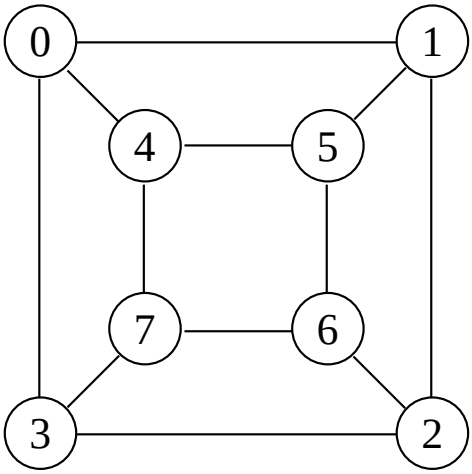
- Um grafo direcionado $G = (V, A)$ é **fortemente conectado** se cada dois vértices quaisquer são alcançáveis a partir um do outro.
- Os **componentes fortemente conectados** de um grafo direcionado são conjuntos de vértices sob a relação “são mutuamente alcançáveis”.
- Um **grafo direcionado fortemente conectado** tem apenas um componente fortemente conectado.

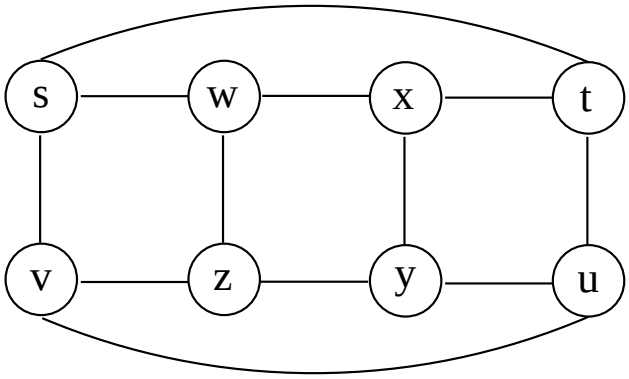
Ex.: $\{0, 1, 2, 3\}$, $\{4\}$ e $\{5\}$ são os componentes fortemente conectados, $\{4, 5\}$ não o é pois o vértice 5 não é alcançável a partir do vértice 4.



Grafos Isomorfos

- $G = (V, A)$ e $G' = (V', A')$ são isomorfos se existir uma bijeção $f : V \rightarrow V'$ tal que $(u, v) \in A$ se e somente se $(f(u), f(v)) \in A'$.

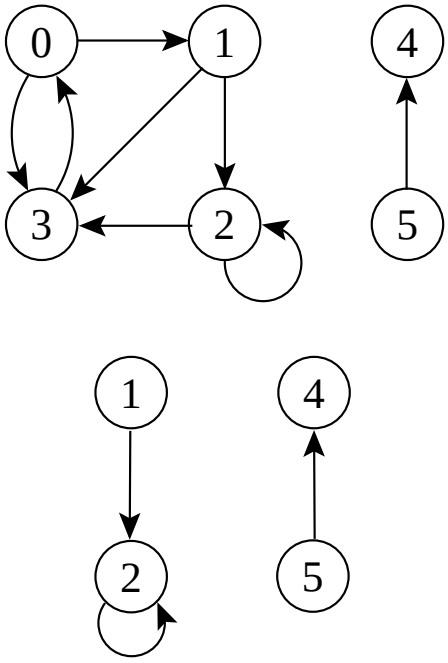




Subgrafos

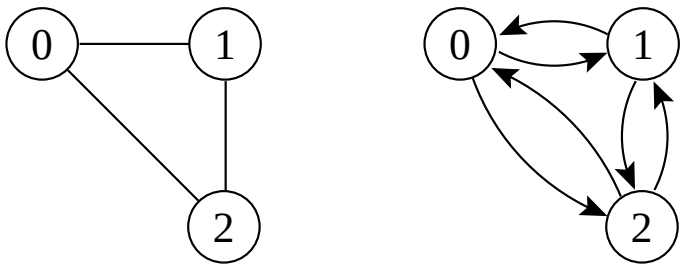
- Um grafo $G' = (V', A')$ é um subgrafo de $G = (V, A)$ se $V' \subseteq V$ e $A' \subseteq A$.
- Dado um conjunto $V' \subseteq V$, o subgrafo induzido por V' é o grafo $G' = (V', A')$, onde $A' = \{(u, v) \in A | u, v \in V'\}$.

Ex.: Subgrafo induzido pelo conjunto de vértices $\{1, 2, 4, 5\}$.



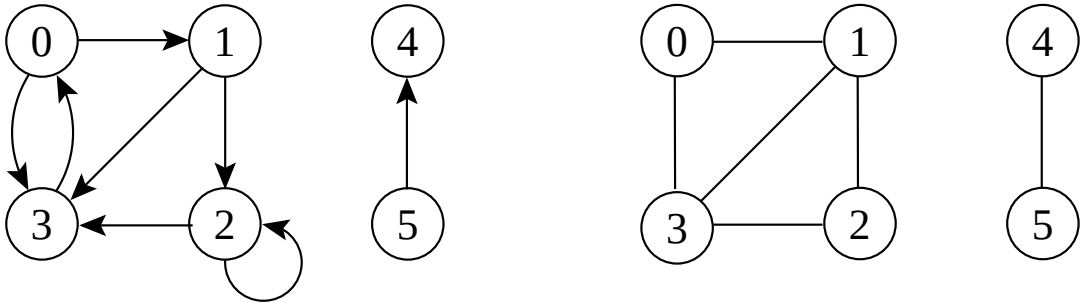
Versão Direcionada de um Grafo Não Direcionado

- A versão direcionada de um grafo não direcionado $G = (V, A)$ é um grafo direcionado $G' = (V', A')$ onde $(u, v) \in A'$ se e somente se $(u, v) \in A$.
- Cada aresta não direcionada (u, v) em G é substituída por duas arestas direcionadas (u, v) e (v, u)
- Em um grafo direcionado, um **vizinho** de um vértice u é qualquer vértice adjacente a u na versão não direcionada de G.



Versão Não Direcionada de um Grafo Direcionado

- A versão não direcionada de um grafo direcionado $G = (V, A)$ é um grafo não direcionado $G' = (V', A')$ onde $(u, v) \in A'$ se e somente se $u \neq v$ e $(u, v) \in A$.
- A versão não direcionada contém as arestas de G sem a direção e sem os *self-loops*.
- Em um grafo não direcionado, u e v são vizinhos se eles são adjacentes.



Outras Classificações de Grafos

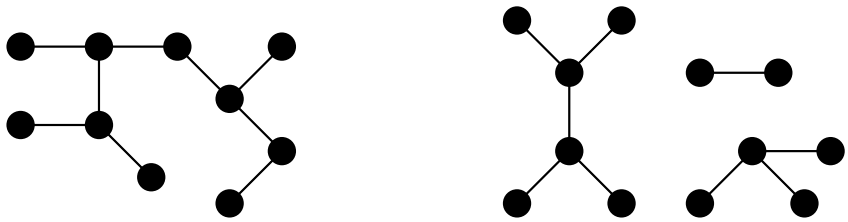
- **Grafo ponderado**: possui pesos associados às arestas.
- **Grafo bipartido**: grafo não direcionado $G = (V, A)$ no qual V pode ser particionado em dois conjuntos V_1 e V_2 tal que $(u, v) \in A$ implica que $u \in V_1$ e $v \in V_2$ ou $u \in V_2$ e $v \in V_1$ (todas as arestas ligam os dois conjuntos V_1 e V_2).
- **Hipergrafo**: grafo não direcionado em que cada aresta conecta um número arbitrário de vértices.

Grafos Completos

- Um grafo completo é um grafo não direcionado no qual todos os pares de vértices são adjacentes.
- Possui $(|V|^2 - |V|)/2 = |V|(|V| - 1)/2$ arestas, pois do total de $|V|^2$ pares possíveis de vértices devemos subtrair $|V|$ *self-loops* e dividir por 2 (cada aresta ligando dois vértices é contada duas vezes).
- O número total de **grafos diferentes** com $|V|$ vértices é $2^{|V|(|V|-1)/2}$ (número de maneiras diferentes de escolher um subconjunto a partir de $|V|(|V| - 1)/2$ possíveis arestas).

Árvores

- **Árvore livre**: grafo não direcionado acíclico e conectado. É comum dizer apenas que o grafo é uma árvore omitindo o “livre”.
- **Floresta**: grafo não direcionado acíclico, podendo ou não ser conectado.
- **Árvore geradora** de um grafo conectado $G = (V, A)$: subgrafo que contém todos os vértices de G e forma uma árvore.
- **Floresta geradora** de um grafo $G = (V, A)$: subgrafo que contém todos os vértices de G e forma uma floresta.



(a) (b)

O Tipo Abstratos de Dados Grafo

- Importante considerar os algoritmos em grafos como **tipos abstratos de dados**.
- Conjunto de operações associado a uma estrutura de dados.
- Independência de implementação para as operações.

Operadores do TAD Grafo

1. Criar um grafo vazio.
2. Inserir uma aresta no grafo.
3. Verificar se existe determinada aresta no grafo.
4. Obter a lista de vértices adjacentes a determinado vértice.
5. Retirar uma aresta do grafo.
6. Imprimir um grafo.
7. Obter o número de vértices do grafo.
8. Obter o transposto de um grafo direcionado.
9. Obter a aresta de menor peso de um grafo.

Operação “Obter Lista de Adjacentes”

1. Verificar se a lista de adjacentes de um vértice v está vazia. Retorna *true* se a lista de adjacentes de v está vazia.
2. Obter o primeiro vértice adjacente a um vértice v, caso exista. Retorna o endereço do primeiro vértice na lista de adjacentes de v.
3. Obter o próximo vértice adjacente a um vértice v, caso exista. Retorna a próxima aresta que o vértice v participa.

Implementação da Operação “Obter Lista de Adjacentes”

- É comum encontrar um pseudo comando do tipo:

  for u $\in$ iista de adjacentes (v) **do** { faz algo com u }

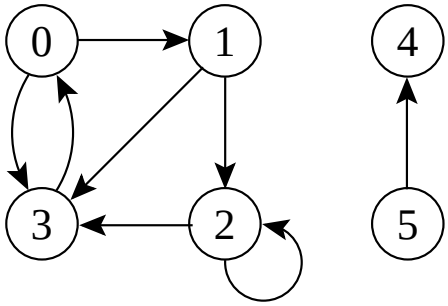
- O trecho de programa abaixo apresenta um possível refinamento do pseudo comando acima.

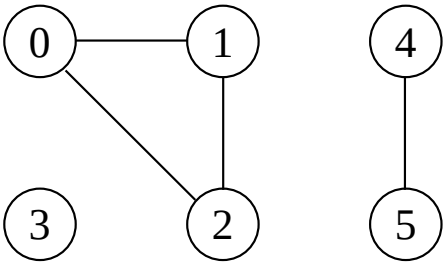
```
if (!grafo.listaAdjVazia (v)) {
  Aresta aux = grafo.primeiroListaAdj (v);
  while (aux != null) {
    int u = aux.vertice2 (); int peso = aux.peso ();
    aux = grafo.proxAdj (v);
  }
}
```

Matriz de Adjacência

- A matriz de adjacência de um grafo $G = (V, A)$ contendo n vértices é uma matriz $n \times n$ de *bits*, onde $A[i, j]$ é 1 (ou verdadeiro) se e somente se existe um arco do vértice i para o vértice j.
- Para grafos ponderados $A[i, j]$ contém o rótulo ou peso associado com a aresta e, neste caso, a matriz não é de *bits*.
- Se não existir uma aresta de i para j então é necessário utilizar um valor que não possa ser usado como rótulo ou peso.

Matriz de Adjacência - Exemplo





	0	1	2	3	4	5
0		1		1		
1			1	1		
2			1	1		
3	1					
4						
5						

(a)

	0	1	2	3	4	5
0		1	1			
1	1		1			
2	1	1				
3						
4						
5						

(b)

Matriz de Adjacência - Análise

- Deve ser utilizada para grafos **densos**, onde $|A|$ é próximo de $|V|^2$.
- O tempo necessário para acessar um elemento é independente de $|V|$ ou $|A|$.
- É muito útil para algoritmos em que necessitamos saber com rapidez se existe uma aresta ligando dois vértices.
- A maior desvantagem é que a matriz necessita $\Omega(|V|^2)$ de espaço. Ler ou examinar a matriz tem complexidade de tempo $O(|V|^2)$.

Matriz de Adjacência - Implementação

- A inserção de um novo vértice ou retirada de um vértice já existente pode ser realizada com custo constante.

```
package cap7.matrizadj;
public class Grafo {
  public static class Aresta {
    private int v1, v2, peso;
    public Aresta (int v1, int v2, int peso) {
      this.v1 = v1; this.v2 = v2; this.peso = peso; }
    public int peso () { return this.peso; }
    public int v1 () { return this.v1; }
    public int v2 () { return this.v2; }
  }
  private int mat[][]; // pesos do tipo inteiro
  private int numVertices;
  private int pos[]; // posição atual ao se percorrer os adjs de um vértice v
  public Grafo (int numVertices) {
    this.mat = new int[numVertices][numVertices];
    this.pos = new int[numVertices]; this.numVertices = numVertices;
    for (int i = 0; i < this.numVertices; i++) {
      for (int j = 0; j < this.numVertices; j++) this.mat[i][j] = 0;
      this.pos[i] = -1;
    }
  }
  public void insereAresta (int v1, int v2, int peso) {
    this.mat[v1][v2] = peso; }
  public boolean existeAresta (int v1, int v2) {
    return (this.mat[v1][v2] > 0);
  }
```

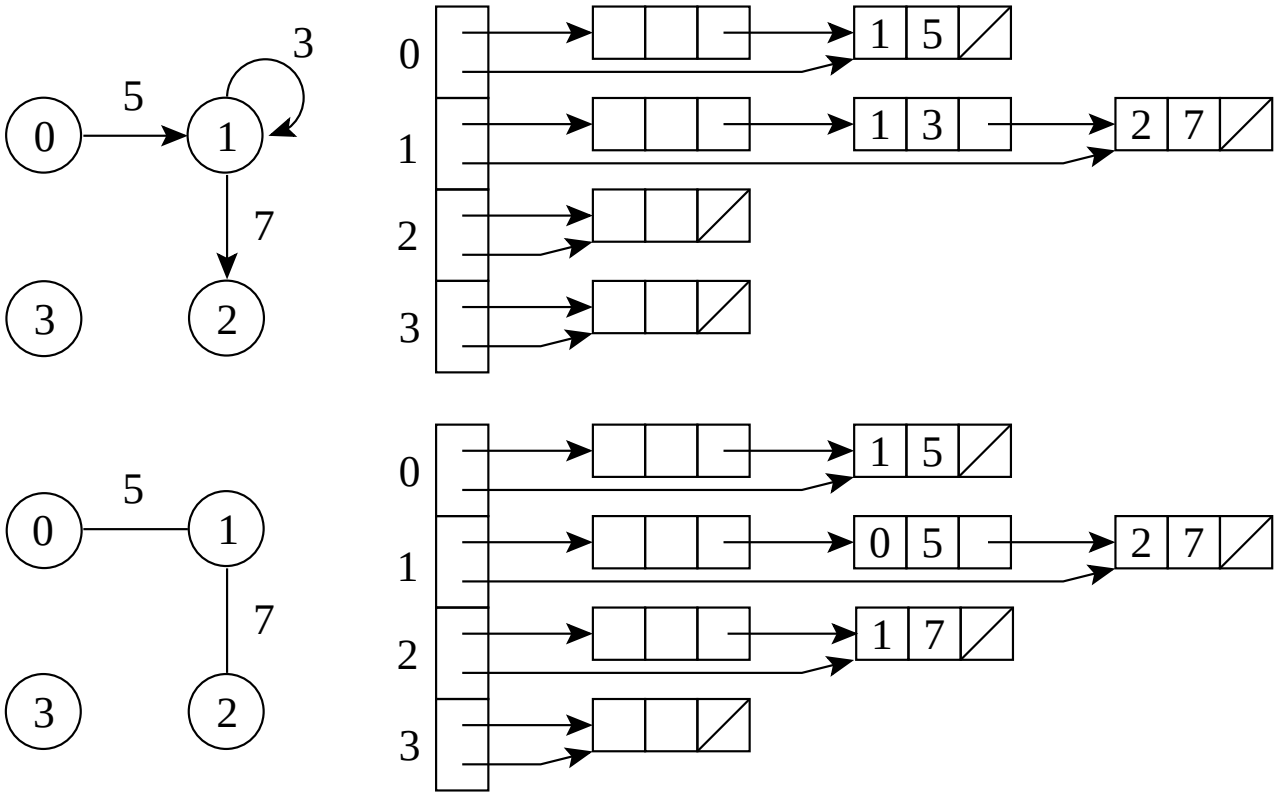
Matriz de Adjacência - Implementação

```
public boolean listaAdjVazia (int v) {
  for (int i =0; i < this.numVertices; i++)
    if (this.mat[v][i] > 0) return false;
  return true;
}
public Aresta primeiroListaAdj (int v) {
  // Retorna a primeira aresta que o vértice v participa ou
  // null se a lista de adjacência de v for vazia
  this.pos[v] = −1; return this.proxAdj (v);
}
public Aresta proxAdj (int v) {
 // Retorna a próxima aresta que o vértice v participa ou
 // null se a lista de adjacência de v estiver no fim
  this.pos[v] ++;
  while ((this.pos[v] < this.numVertices) &&
          (this.mat[v][this.pos[v]] == 0)) this.pos[v]++;
  if (this.pos[v] == this.numVertices) return null;
  else return new Aresta (v, this.pos[v], this.mat[v][this.pos[v]]);
}
public Aresta retiraAresta (int v1, int v2) {
  if (this.mat[v1][v2] == 0) return null; // Aresta não existe
  else {
    Aresta aresta = new Aresta (v1, v2, this.mat[v1][v2]);
    this.mat[v1][v2] = 0; return aresta;
  }
}
```

Matriz de Adjacência - Implementação

```
  public void imprime () {
    System.out.print ("   ");
    for (int i = 0; i < this.numVertices; i++)
      System.out.print (i + "   ");
    System.out.println ();
    for (int i = 0; i < this.numVertices; i++) {
      System.out.print (i + "  ");
      for (int j = 0; j < this.numVertices; j++)
        System.out.print (this.mat[i][j] + "   ");
      System.out.println ();
    }
  }
  public int numVertices () {
    return this.numVertices;
  }
}
```

Listas de Adjacência usando Estruturas Auto-Referenciadas



- Um arranjo adj de $|V|$ listas, uma para cada vértice em V.
- Para cada $u \in V$, $adj[u]$ contém todos os vértices adjacentes a u em G.

Listas de adjacência - Análise

- Os vértices de uma lista de adjacência são em geral armazenados em uma ordem arbitrária.
- Possui uma complexidade de espaço $O(|V| + |A|)$
- Indicada para grafos **esparsos**, onde $|A|$ é muito menor do que $|V|^2$.
- É compacta e usualmente utilizada na maioria das aplicações.
- A principal desvantagem é que ela pode ter tempo $O(|V|)$ para determinar se existe uma aresta entre o vértice i e o vértice j, pois podem existir $O(|V|)$ vértices na lista de adjacentes do vértice i.

Listas de Adjacência usando Estruturas Auto-Referenciadas - Implementação

- A seguir apresentamos a implementação do **tipo abstrato de dados grafo** utilizando listas encadeadas implementadas por meio de estruturas auto-referenciadas para as sete primeiras operações definidas anteriormente.
- A classe $Aresta$ representa as informações de uma aresta para que os usuários da classe $Grafo$ possam acessá-las.
- A classe $Celula$ é utilizada para representar uma entrada na lista de adjacência de um vértice do grafo.
- O método $equals$ é usado para verificar se um vértice qualquer v é adjacente a um outro vértice u ao se percorrer a lista de adjacentes de u.

Listas de Adjacência usando Estruturas Auto-Referenciadas - Implementação

```
package cap7.listaadj.autoreferencia;
import cap3.autoreferencia.Lista;

public class Grafo {
  public static class Aresta {
    private int v1, v2, peso;
    public Aresta (int v1, int v2, int peso) {
      this.v1 = v1; this.v2 = v2; this.peso = peso;
    }
    public int peso () { return this.peso; }
    public int v1 () { return this.v1; }
    public int v2 () { return this.v2; }
  }
  private static class Celula {
    int vertice, peso;
    public Celula (int v, int p) {this.vertice = v; this.peso = p;}
    public boolean equals (Object obj) {
      Celula item = (Celula) obj;
      return (this.vertice == item.vertice);
    }
  }
  private Lista adj[];
  private int numVertices;
  public Grafo (int numVertices) {
    this.adj = new Lista[numVertices]; this.numVertices = numVertices;
    for (int i = 0; i < this.numVertices; i++) this.adj[i] = new Lista();
  }
  public void insereAresta (int v1, int v2, int peso) {
    Celula item = new Celula (v2, peso);
    this.adj[v1].insere (item);
  }
```

Listas de Adjacência usando Estruturas Auto-Referenciadas - Implementação

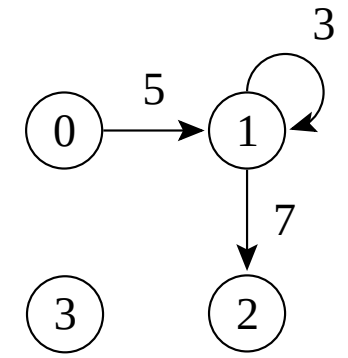
```
public boolean existeAresta (int v1, int v2) {
  Celula item = new Celula (v2, 0);
  return (this.adj[v1].pesquisa (item) != null);
}
public boolean listaAdjVazia (int v) {
  return this.adj[v].vazia ();
}
public Aresta primeiroListaAdj (int v) {
  // Retorna a primeira aresta que o vértice v participa ou
  // null se a lista de adjacência de v for vazia
  Celula item = (Celula) this.adj[v].primeiro ();
  return item != null ? new Aresta (v, item.vertice, item.peso): null;
}
public Aresta proxAdj (int v) {
  // Retorna a próxima aresta que o vértice v participa ou
  // null se a lista de adjacência de v estiver no fim
  Celula item = (Celula) this.adj[v].proximo ();
  return item != null ? new Aresta (v, item.vertice, item.peso): null;
}
public Aresta retiraAresta (int v1, int v2) throws Exception {
  Celula chave = new Celula (v2, 0);
  Celula item = (Celula) this.adj[v1].retira (chave);
  return item != null ? new Aresta (v1, v2, item.peso): null;
}
```

Listas de Adjacência usando Estruturas Auto-Referenciadas - Implementação

```
  public void imprime () {
    for (int i = 0; i < this.numVertices; i++) {
      System.out.println ("Vertice " + i + ":");
      Celula item = (Celula) this.adj[i].primeiro ();
      while (item != null) {
        System.out.println ("  " + item.vertice + " (" +item.peso+ ")");
        item = (Celula) this.adj[i].proximo ();
      }
    }
  }
  public int numVertices () {
    return this.numVertices;
  }
}
```

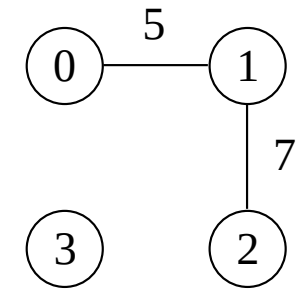
Listas de Adjacência usando Arranjos

(a)



		cab	prox	peso
V	0	4	4	
	1	6	5	
	2	2	0	
	3	3	0	
A	4	1	0	5
	5	1	6	3
	6	2	0	7

(b)



		cab	prox	peso
V	0	4	4	
	1	6	5	
	2	7	7	
	3	3	0	
A	4	1	0	5
	5	0	6	5
	6	2	0	7
	7	1	0	7

- *cab*: endereços do último item da lista de adjacentes de cada vértice (nas $|V|$ primeiras posições) e os vértices propriamente ditos (nas $|A|$ últimas posições)
- *prox*: endereço do próximo item da lista de adjacentes.
- *peso*: valor do peso de cada aresta do grafo (nas últimas $|A|$ posições).

Listas de Adjacência usando Arranjos - Implementação

```
package cap7.listaadj.arranjo;
public class Grafo {
  public static class Aresta {
    private int v1, v2, peso;
    public Aresta (int v1, int v2, int peso) {
      this.v1 = v1; this.v2 = v2; this.peso = peso;
    }
    public int peso () { return this.peso; }
    public int v1 () { return this.v1; }
    public int v2 () { return this.v2; }
  }
  private int cab[], prox[], peso[];
  private int pos[]; // posição atual ao se percorrer os adjs de um vértice v
  private int numVertices, proxDisponivel;
  public Grafo (int numVertices, int numArestas) {
    int tam = numVertices + 2*numArestas;
    this.cab  = new int[tam]; this.prox = new int[tam];
    this.peso = new int[tam]; this.numVertices = numVertices;
    this.pos = new int[this.numVertices];
    for (int i = 0; i < this.numVertices; i++) {
      this.prox[i] = 0;
      this.cab[i] = i;
      this.peso[i] = 0;
      this.pos[i] = i;
    }
    this.proxDisponivel = this.numVertices;
  }
```

Listas de Adjacência usando Arranjos - Implementação

```
public void insereAresta (int v1, int v2, int peso) {
  if (this.proxDisponivel == this.cab.length)
    System.out.println ("Nao ha espaco disponivel para a aresta");
  else {
    int ind = this.proxDisponivel++;
    this.prox[this.cab[v1]] = ind;
    this.cab[ind] = v2; this.cab[v1] = ind;
    this.prox[ind] = 0; this.peso[ind] = peso;
  }
}
public boolean existeAresta (int v1, int v2) {
  for (int i = this.prox[v1]; i != 0; i = this.prox[i])
    if (this.cab[i] == v2) return true;
  return false;
}
public boolean listaAdjVazia (int v) {
  return (this.prox[v] == 0);
}
public Aresta primeiroListaAdj (int v) {
  // Retorna a primeira aresta que o vértice v participa ou
  // null se a lista de adjacência de v for vazia
  this.pos[v] = v;
  return this.proxAdj (v);
}
public Aresta proxAdj (int v) {
  // Retorna a próxima aresta que o vértice v participa ou
  // null se a lista de adjacência de v estiver no fim
  this.pos[v] = this.prox[this.pos[v]];
  if (this.pos[v] == 0) return null;
  else return new Aresta (v,this.cab[pos[v]],this.peso[pos[v]]);
}
```

Listas de Adjacência usando Arranjos - Implementação

```
  public Aresta retiraAresta (int v1, int v2) {
    int i;
    for (i = v1; this.prox[i] != 0; i = this.prox[i])
      if (this.cab[this.prox[i]] == v2) break;
    int ind = this.prox[i];
    if (this.cab[ind] == v2) { // encontrou aresta
      Aresta aresta = new Aresta(v1, v2, this.peso[ind]);
      this.cab[ind] = this.cab.length; // marca como removido
      if (this.prox[ind] == 0) this.cab[v1] = i; // último vértice
      this.prox[i] = this.prox[ind];
      return aresta;
    } else return null;
  }
  public void imprime () {
    for (int i = 0; i < this.numVertices; i++) {
      System.out.println ("Vertice " + i + ":");
      for (int j = this.prox[i]; j != 0; j = this.prox[j])
        System.out.println ("  " + this.cab[j]+" (" +this.peso[j]+ ")");
    }
  }
  public int numVertices () { return this.numVertices; }
}
```

Busca em Profundidade

- A busca em profundidade, do inglês *depth-first search*), é um algoritmo para caminhar no grafo.
- A estratégia é buscar o mais profundo no grafo sempre que possível.
- As arestas são exploradas a partir do vértice v mais recentemente descoberto que ainda possui arestas não exploradas saindo dele.
- Quando todas as arestas adjacentes a v tiverem sido exploradas a busca anda para trás para explorar vértices que saem do vértice do qual v foi descoberto.
- O algoritmo é a base para muitos outros algoritmos importantes, tais como verificação de grafos acíclicos, ordenação topológica e componentes fortemente conectados.

Busca em Profundidade

- Para acompanhar o progresso do algoritmo cada vértice é colorido de branco, cinza ou preto.
- Todos os vértices são inicializados branco.
- Quando um vértice é *descoberto* pela primeira vez ele torna-se cinza, e é tornado preto quando sua lista de adjacentes tenha sido completamente examinada.
- $d[v]$: tempo de descoberta
- $t[v]$: tempo de término do exame da lista de adjacentes de v.
- Estes registros são inteiros entre 1 e $2|V|$ pois existe um evento de descoberta e um evento de término para cada um dos $|V|$ vértices.

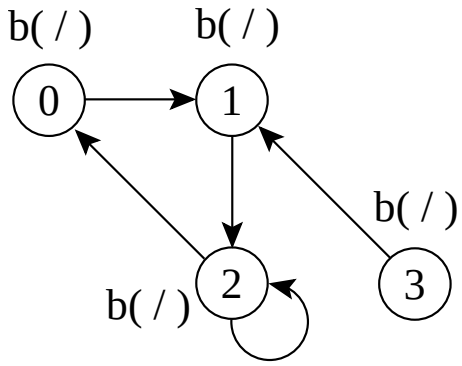
Busca em Profundidade - Implementação

```
package cap7;
import cap7.listaadj.autoreferencia.Grafo;
public class BuscaEmProfundidade {
  public static final byte branco = 0;
  public static byte cinza         = 1;
  public static byte preto         = 2;
  private int d[], t[], antecessor[];
  private Grafo grafo;

  public BuscaEmProfundidade (Grafo grafo) {
    this.grafo = grafo; int n = this.grafo.numVertices();
    d = new int[n]; t = new int[n]; antecessor = new int[n];
  }
  private int visitaDfs (int u, int tempo, int cor[]) {
    cor[u] = cinza; this.d[u] = ++tempo;
    if (!this.grafo.listaAdjVazia (u)) {
      Grafo.Aresta a = this.grafo.primeiroListaAdj (u);
      while (a != null) {
        int v = a.v2 ();
        if (cor[v] == branco) {
          this.antecessor[v] = u;
          tempo = this.visitaDfs (v, tempo, cor);
        }
        a = this.grafo.proxAdj (u);
      }
    }
    cor[u] = preto; this.t[u] = ++tempo;
    return tempo;
  }
```

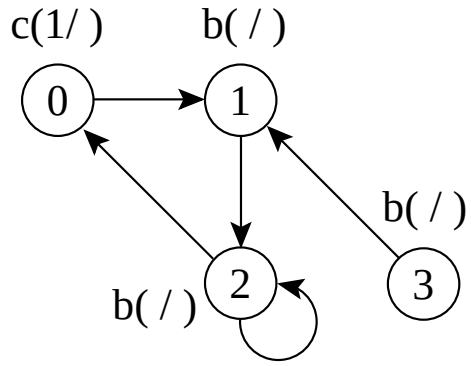
Busca em Profundidade - Implementação

```
  public void buscaEmProfundidade () {
    int tempo = 0; int cor[] = new int[this.grafo.numVertices ()];
    for (int u = 0; u < grafo.numVertices (); u++) {
      cor[u] = branco; this.antecessor[u] = -1;
    }
    for (int u = 0; u < grafo.numVertices (); u++)
      if (cor[u] == branco) tempo = this.visitaDfs (u, tempo, cor);
  }
  public int d (int v) { return this.d[v]; }
  public int t (int v) { return this.t[v]; }
  public int antecessor (int v) { return this.antecessor[v]; }
}
```

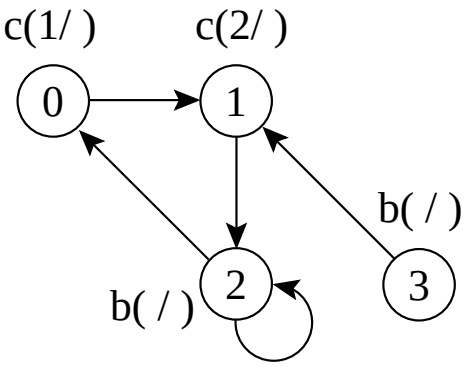
Busca em Profundidade - Exemplo



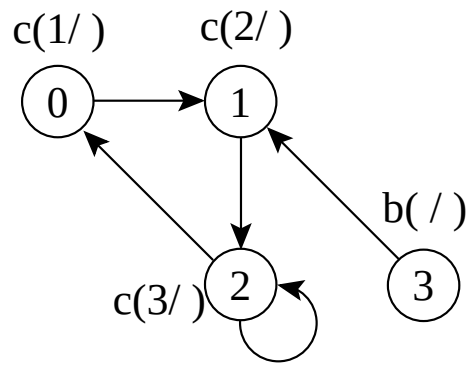
(a)



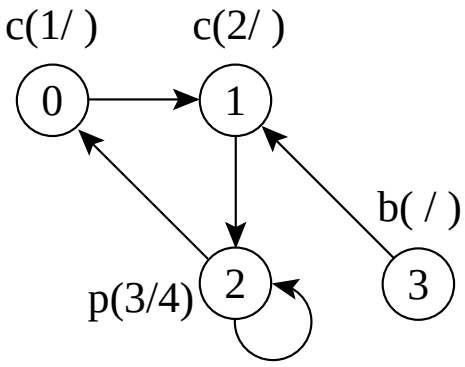
(b)



(c)

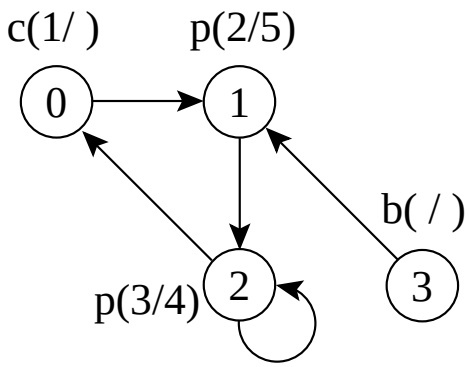


(d)

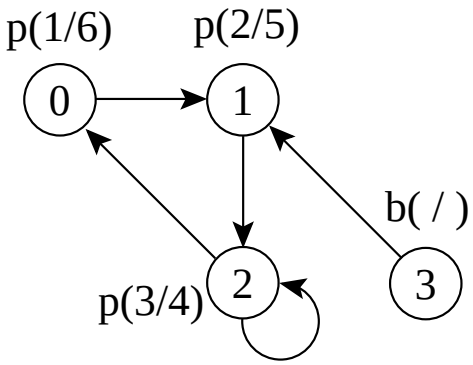


(e)

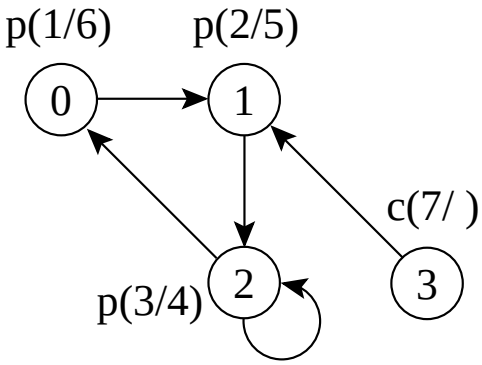
Busca em Profundidade - Exemplo



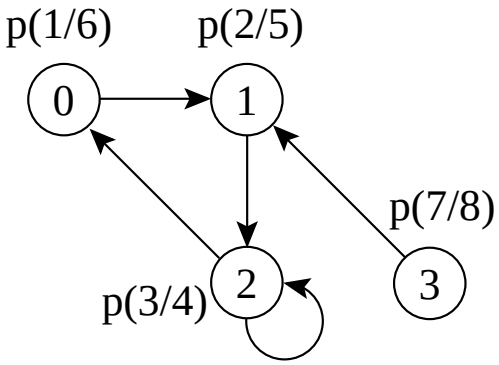
(f)



(g)



(h)



(i)

Busca em Profundidade - Análise

- Os dois anéis do método $buscaEmProfundidade$ têm custo $O(|V|)$ cada um, a menos da chamada do método $visitaDfs(u, tempo, cor)$ no segundo anel.
- O método $visitaDfs$ é chamado exatamente uma vez para cada vértice $u \in V$, desde que $visitaDfs$ seja chamado apenas para vértices brancos, e a primeira ação é pintar o vértice de cinza.
- Durante a execução de $visitaDfs(u, tempo, cor)$, o anel principal é executado $|adj[u]|$ vezes.
- Desde que

$$\sum_{u \in V} |adj[u]| = O(|A|),$$

  o tempo total de execução de $visitaDfs$ é $O(|A|)$.
- Logo, a complexidade total do método $buscaEmProfundidade$ é $O(|V| + |A|)$.

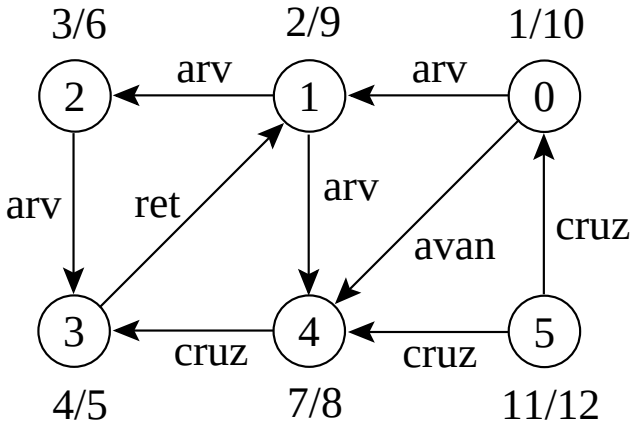
Classificação de Arestas

- Existem:

1. **Arestas de árvore**: são arestas de uma árvore de busca em profundidade. A aresta (u, v) é uma aresta de árvore se v foi descoberto pela primeira vez ao percorrer a aresta (u, v).
2. **Arestas de retorno**: conectam um vértice u com um antecessor v em uma árvore de busca em profundidade (inclui *self-loops*).
3. **Arestas de avanço**: não pertencem à árvore de busca em profundidade mas conectam um vértice a um descendente que pertence à árvore de busca em profundidade.
4. **Arestas de cruzamento**: podem conectar vértices na mesma árvore de busca em profundidade, ou em duas árvores diferentes.

Classificação de Arestas

- Classificação de arestas pode ser útil para derivar outros algoritmos.
- Na busca em profundidade cada aresta pode ser classificada pela cor do vértice que é alcançado pela primeira vez:
  - Branco indica uma aresta de árvore.
  - Cinza indica uma aresta de retorno.
  - Preto indica uma aresta de avanço quando u é descoberto antes de v ou uma aresta de cruzamento caso contrário.



Teste para Verificar se Grafo é Acíclico

- A busca em profundidade pode ser usada para verificar se um grafo é acíclico ou contém um ou mais ciclos.
- Se uma aresta de retorno é encontrada durante a busca em profundidade em G, então o grafo tem ciclo.
- Um grafo direcionado G é acíclico se e somente se a busca em profundidade em G não apresentar arestas de retorno.

Busca em Largura

- Expande a fronteira entre vértices descobertos e não descobertos uniformemente através da largura da fronteira.
- O algoritmo descobre todos os vértices a uma distância k do vértice origem antes de descobrir qualquer vértice a uma distância $k + 1$.
- O grafo $G(V, A)$ pode ser direcionado ou não direcionado.

Busca em Largura

- Cada vértice é colorido de branco, cinza ou preto.
- Todos os vértices são inicializados branco.
- Quando um vértice é descoberto pela primeira vez ele torna-se cinza.
- Vértices cinza e preto já foram descobertos, mas são distinguidos para assegurar que a busca ocorra em largura.
- Se $(u, v) \in A$ e o vértice u é preto, então o vértice v tem que ser cinza ou preto.
- Vértices cinza podem ter alguns vértices adjacentes brancos, e eles representam a fronteira entre vértices descobertos e não descobertos.

Busca em Largura - Implementação

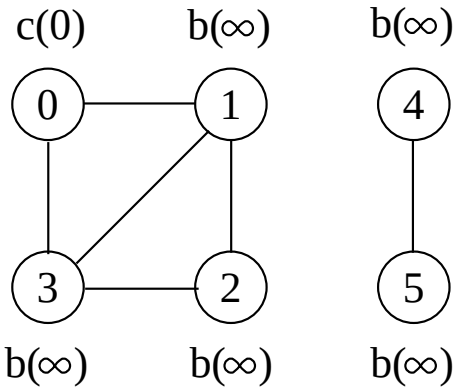
```
package cap7;
import cap3.autoreferencia.Fila;
import cap7.listaadj.autoreferencia.Grafo;
public class BuscaEmLargura {
  public static final byte branco = 0;
  public static byte cinza         = 1;
  public static byte preto         = 2;
  private int d[], antecessor[];
  private Grafo grafo;
  public BuscaEmLargura (Grafo grafo) {
    this.grafo = grafo; int n = this.grafo.numVertices();
    this.d = new int[n]; this.antecessor = new int[n];
  }
  private void visitaBfs (int u, int cor[]) throws Exception {
    cor[u] = cinza; this.d[u] = 0;
    Fila fila = new Fila (); fila.enfileira (new Integer (u));
    while (!fila.vazia ()) {
      Integer aux = (Integer)fila.desenfileira (); u = aux.intValue();
      if (!this.grafo.listaAdjVazia (u)) {
        Grafo.Aresta a = this.grafo.primeiroListaAdj (u);
        while (a != null) {
          int v = a.v2 ();
          if (cor[v] == branco) {
            cor[v] = cinza; this.d[v] = this.d[u] + 1;
            this.antecessor[v] = u; fila.enfileira (new Integer (v));
          }
          a = this.grafo.proxAdj (u);
        }
      }
      cor[u] = preto;
    }
  }
```

Busca em Largura - Implementação

```
  public void buscaEmLargura () throws Exception {
    int cor[] = new int[this.grafo.numVertices ()];
    for (int u = 0; u < grafo.numVertices (); u++) {
      cor[u] = branco; this.d[u] = Integer.MAX_VALUE;
      this.antecessor[u] = -1;
    }
    for (int u = 0; u < grafo.numVertices (); u++)
      if (cor[u] == branco) this.visitaBfs (u, cor);
  }
  public int d (int v) { return this.d[v]; }
  public int antecessor (int v) { return this.antecessor[v]; }
}
```

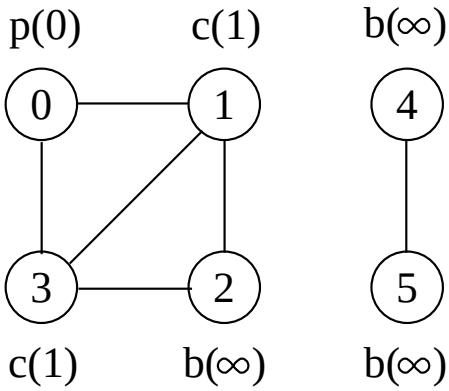
Busca em Largura - Exemplo

(a)



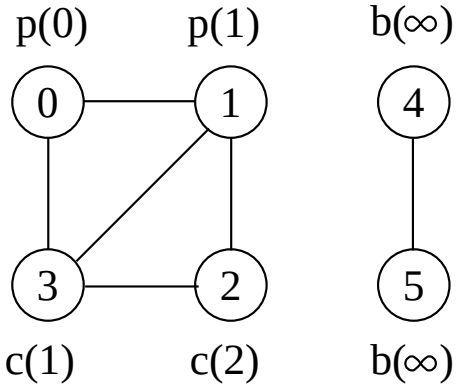
F | 0 |
0

(b)



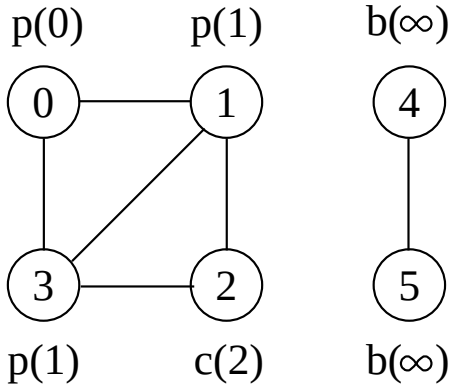
F | 1 | 3 |
1 1

(c)



F | 3 | 2 |
1 2

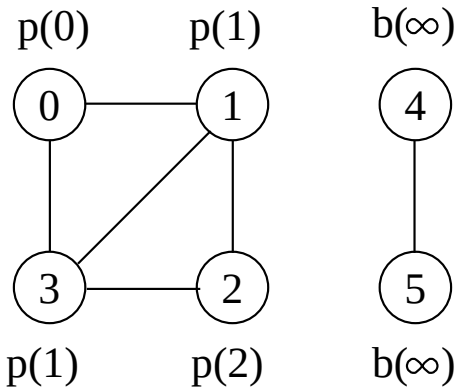
(d)



F | 2 |
2

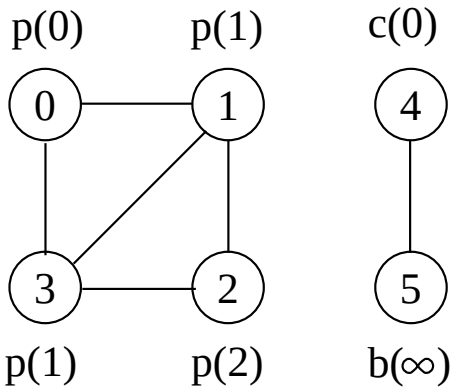
Busca em Largura - Exemplo

(e)



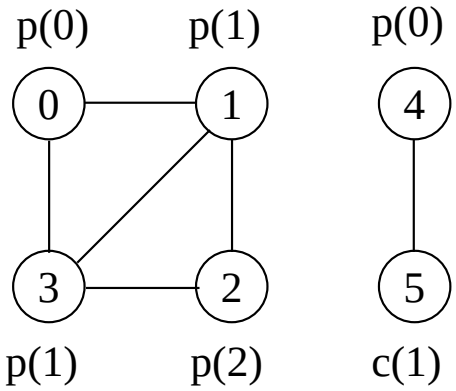
F ∅

(f)



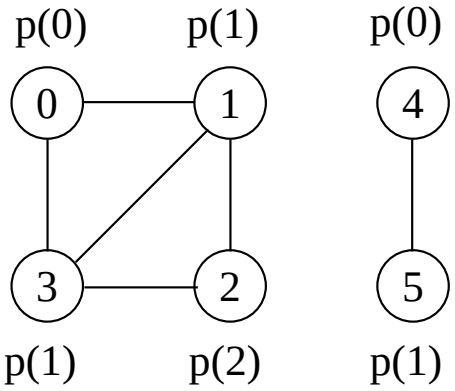
F 4
0

(g)



F 5
1

(h)



F ∅

Busca em Largura - Análise (para listas de adjacência)

- O custo de inicialização do primeiro anel no método $buscaEmLargura$ é $O(|V|)$.
- O custo do segundo anel é também $O(|V|)$.
- Método $visitaBfs$: enfileirar e desenfileirar têm custo $O(1)$, logo, o custo total com a fila é $O(|V|)$.
- Cada lista de adjacentes é percorrida no máximo uma vez, quando o vértice é desenfileirado.
- Desde que a soma de todas as listas de adjacentes é $O(|A|)$, o tempo total gasto com as listas de adjacentes é $O(|A|)$.
- Complexidade total: é $O(|V| + |A|)$.

Caminhos Mais Curtos

- A busca em largura obtém o **caminho mais curto** de u até v.
- O procedimento *VisitaBfs* contrói uma árvore de busca em largura que é armazenada na variável $antecessor$.
- O programa abaixo imprime os vértices do caminho mais curto entre o vértice origem e outro vértice qualquer do grafo, a partir do vetor $antecessor$. obtido na busca em largura.

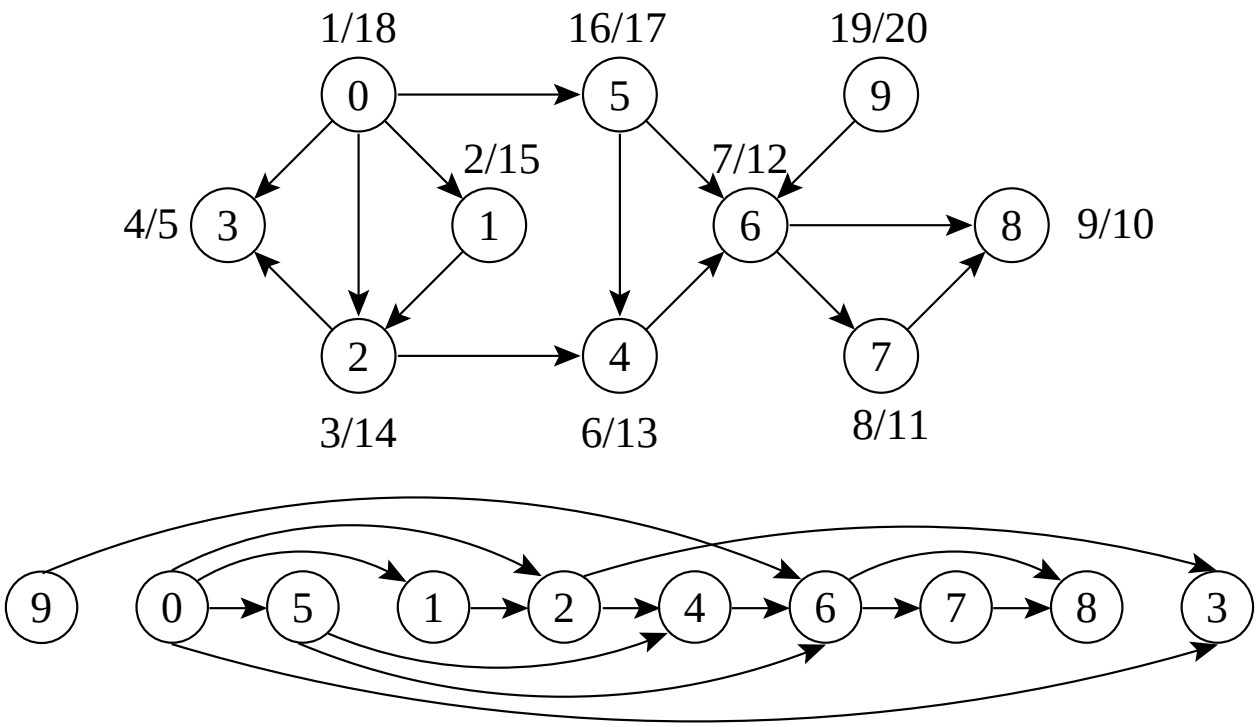
```
public void imprimeCaminho (int origem, int v) {
  if (origem == v) System.out.println (origem);
  else if (this.antecessor[v] == -1)
    System.out.println ("Nao existe caminho de " + origem + " ate " + v);
  else {
    imprimeCaminho (origem, this.antecessor[v]);
    System.out.println (v);
  }
}
```

Ordenação Topológica

- Ordenação linear de todos os vértices, tal que se G contém uma aresta (u, v) então u aparece antes de v.
- Pode ser vista como uma ordenação de seus vértices ao longo de uma linha horizontal de tal forma que todas as arestas estão direcionadas da esquerda para a direita.
- Pode ser feita usando a busca em profundidade.

Ordenação Topológica

- Os grafos direcionados acíclicos são usados para indicar precedências entre eventos.
- Uma aresta direcionada (u, v) indica que a atividade u tem que ser realizada antes da atividade v.



Ordenação Topológica

- Algoritmo para ordenar topologicamente um grafo direcionado acíclico $G = (V, A)$:
  1. Aplicar a busca em profundidade no grafo G para obter os tempos de término $t[u]$ para cada vértice u.
  2. Ao término de cada vértice, insira-o na frente de uma lista linear encadeada.
  3. Retornar a lista encadeada de vértices.
- A Custo $O(|V| + |A|)$, uma vez que a busca em profundidade tem complexidade de tempo $O(|V| + |A|)$ e o custo para inserir cada um dos $|V|$ vértices na frente da lista linear encadeada custa $O(1)$.

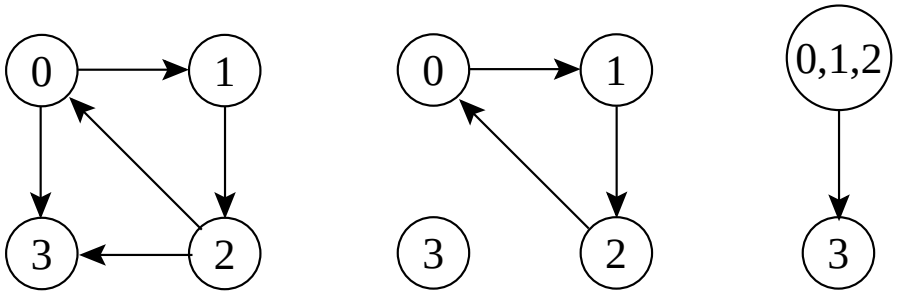
Ordenação Topológica - Implementação

- Basta inserir uma chamada ao método $inserePrimeiro$ no método $buscaDfs$, logo após o momento em que o tempo de término $t[u]$ é obtido e o vértice é pintado de $preto$.
- Ao final, basta retornar a lista obtida.

```
// Insere antes do primeiro item da lista
public void inserePrimeiro (Object item) {
  Celula aux =  this.primeiro.prox;
  this.primeiro.prox = new Celula ();
  this.primeiro.prox.item = item;
  this.primeiro.prox.prox = aux;
}
```

Componentes Fortemente Conectados

- Um componente fortemente conectado de $G = (V, A)$ é um conjunto maximal de vértices $C \subseteq V$ tal que para todo par de vértices u e v em C, u e v são mutuamente alcançáveis
- Podemos particionar V em conjuntos V_i, $1 \leq i \leq r$, tal que vértices u e v são equivalentes se e somente se existe um caminho de u a v e um caminho de v a u.



(a) (b) (c)

Componentes Fortemente Conectados - Algoritmo

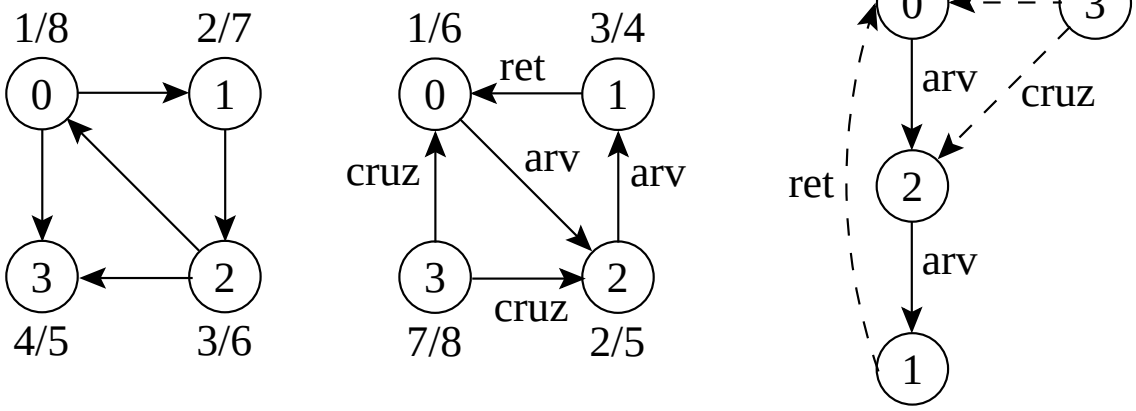
- Usa o **transposto** de G, definido $G^T = (V, A^T)$, onde $A^T = \{(u, v) : (v, u) \in A\}$, isto é, A^T consiste das arestas de G com suas direções invertidas.
- G e G^T possuem os mesmos componentes fortemente conectados, isto é, u e v são mutuamente alcançáveis a partir de cada um em G se e somente se u e v são mutuamente alcançáveis a partir de cada um em G^T.

Componentes Fortemente Conectados - Algoritmo

1. Aplicar a busca em profundidade no grafo G para obter os tempos de término $t[u]$ para cada vértice u.
2. Obter G^T.
3. Aplicar a busca em profundidade no grafo G^T, realizando a busca a partir do vértice de maior $t[u]$ obtido na linha 1. Se a busca em profundidade não alcançar todos os vértices, inicie uma nova busca em profundidade a partir do vértice de maior $t[u]$ dentre os vértices restantes.
4. Retornar os vértices de cada árvore da floresta obtida na busca em profundidade na linha 3 como um componente fortemente conectado separado.

Componentes Fortemente Conectados - Exemplo

- A parte (b) apresenta o resultado da busca em profundidade sobre o grafo transposto obtido, mostrando os tempos de término e a classificação das arestas.
- A busca em profundidade em G^T resulta na floresta de árvores mostrada na parte (c).



(a) (b) (c)

Componentes Fortemente Conectados - Implementação

```
public Grafo grafoTransposto () {
  Grafo grafoT = new Grafo (this.numVertices);
  for (int v = 0; v < this.numVertices; v++)
    if (!this.listaAdjVazia (v)) {
      Aresta adj = this.primeiroListaAdj (v);
      while (adj != null) {
        grafoT.insereAresta (adj.v2 (), adj.v1 (), adj.peso ());
        adj = this.proxAdj (v); }
    }
  return grafoT;
}
```

Componentes Fortemente Conectados - Implementação

```
package cap7;
import cap7.listaadj.autoreferencia.Grafo;
public class Cfc {
  private static class TempoTermino {
    private int numRestantes, t[];
    private boolean restantes[];
    public TempoTermino (int numVertices) {
      t = new int[numVertices];
      restantes = new boolean[numVertices];
      numRestantes = numVertices;
    }
    public int maxTT () {
      int vMax = 0;
      while (!this.restantes[vMax]) vMax++;
      for (int i = 0; i < this.t.length; i ++) {
        if (this.restantes[i]) {
          if (this.t[i] > this.t[vMax]) vMax = i;
        }
      }
      return vMax;
    }
  }
  private Grafo grafo;
  public Cfc (Grafo grafo) {
    this.grafo = grafo;
  }
```

Componentes Fortemente Conectados - Implementação

```
  private void visitaDfs (Grafo grafo, int u, TempoTermino tt) {
    tt.restantes[u] = false; tt.numRestantes--;
    System.out.println ("  Vertice: "+u);
    if (!grafo.listaAdjVazia (u)) {
      Grafo.Aresta a = grafo.primeiroListaAdj (u);
      while (a != null) {
        int v = a.v2 ();
        if (tt.restantes[v]) { this.visitaDfs (grafo, v, tt); }
        a = grafo.proxAdj (u);
      }
    }
  }
  public void obterCfc () {
    BuscaEmProfundidade dfs = new BuscaEmProfundidade (this.grafo);
    dfs.buscaEmProfundidade ();
    TempoTermino tt = new TempoTermino (this.grafo.numVertices ());
    for (int u = 0; u < this.grafo.numVertices (); u++) {
      tt.t[u] = dfs.t (u); tt.restantes[u] = true;
    }
    Grafo grafoT = this.grafo.grafoTransposto ();
    while (tt.numRestantes > 0) {
      int vRaiz = tt.maxTT ();
      System.out.println ("Raiz da proxima arvore: " + vRaiz);
      this.visitaDfs (grafoT, vRaiz, tt);
    }
  }
}
```

Componentes Fortemente Conectados - Análise

- Utiliza o algoritmo para busca em profundidade duas vezes, uma em G e outra em G^T. Logo, a complexidade total é $O(|V| + |A|)$.

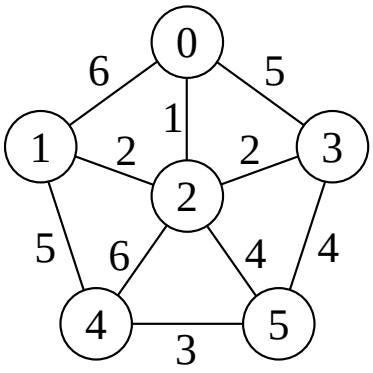
Árvore Geradora Mínima - Aplicação

- Projeto de redes de comunicações conectando n localidades.
- Arranjo de $n-1$ conexões, conectando duas localidades cada.
- Objetivo: dentre as possibilidades de conexões, achar a que usa menor quantidade de cabos.
- Modelagem:
  - $G = (V, A)$: grafo conectado, não direcionado.
  - V: conjunto de cidades.
  - A: conjunto de possíveis conexões
  - $p(u, v)$: peso da aresta $(u, v) \in A$, custo total de cabo para conectar u a v.
- Solução: encontrar um subconjunto $T \subseteq A$ que conecta todos os vértices de G e cujo peso total $p(T) = \sum_{(u,v)\in T} p(u, v)$ é minimizado.

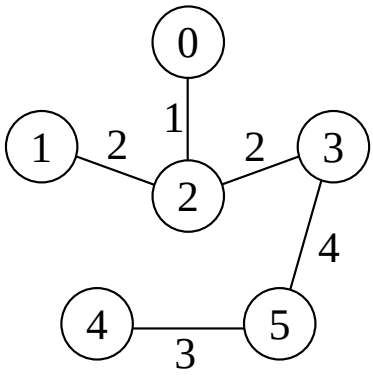
Árvore Geradora Mínima (AGM)

- Como $G' = (V, T)$ é acíclico e conecta todos os vértices, T forma uma árvore chamada **árvore geradora** de G.
- O problema de obter a árvore T é conhecido como **árvore geradora mínima** (AGM).

Ex.: Árvore geradora mínima T cujo peso total é 12. T não é única, pode-se substituir a aresta $(3, 5)$ pela aresta $(2, 5)$ obtendo outra árvore geradora de custo 12.



(a)



(b)

AGM - Algoritmo Genérico

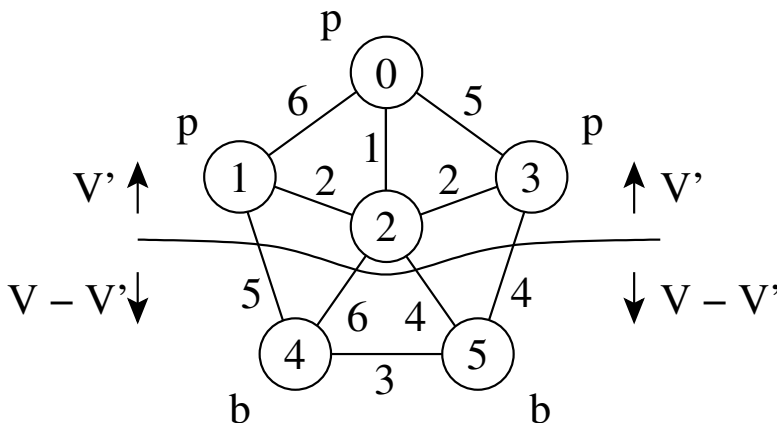
- Uma estratégia **gulosa** permite obter a AGM adicionando uma aresta de cada vez.
- Invariante: Antes de cada iteração, S é um subconjunto de uma árvore geradora mínima.
- A cada passo adicionamos a S uma aresta (u, v) que não viola o invariante. (u, v) é chamada de uma **aresta segura**.

```
    void GenericoAGM
1     S = ∅;
2     while (S não constitui uma árvore geradora mínima)
3       (u, v) = seleciona (A);
4       if (aresta (u, v) é segura para S) S = S + {(u, v)}
5     return S;
```

- Dentro do **while**, S tem que ser um subconjunto próprio da AGM T, e assim tem que existir uma aresta $(u, v) \in T$ tal que $(u, v) \notin S$ e (u, v) é seguro para S.

AGM - Definição de Corte

- Um **corte** $(V', V - V')$ de um grafo não direcionado $G = (V, A)$ é uma partição de V.
- Uma aresta $(u, v) \in A$ *cruza* o corte $(V', V - V')$ se um de seus vértices pertence a V' e o outro vértice pertence a $V - V'$.
- Um corte *respeita* um conjunto S de arestas se não existirem arestas em S que o cruzem.
- Uma aresta cruzando o corte que tenha custo mínimo sobre todas as arestas cruzando o corte é uma *aresta leve*.



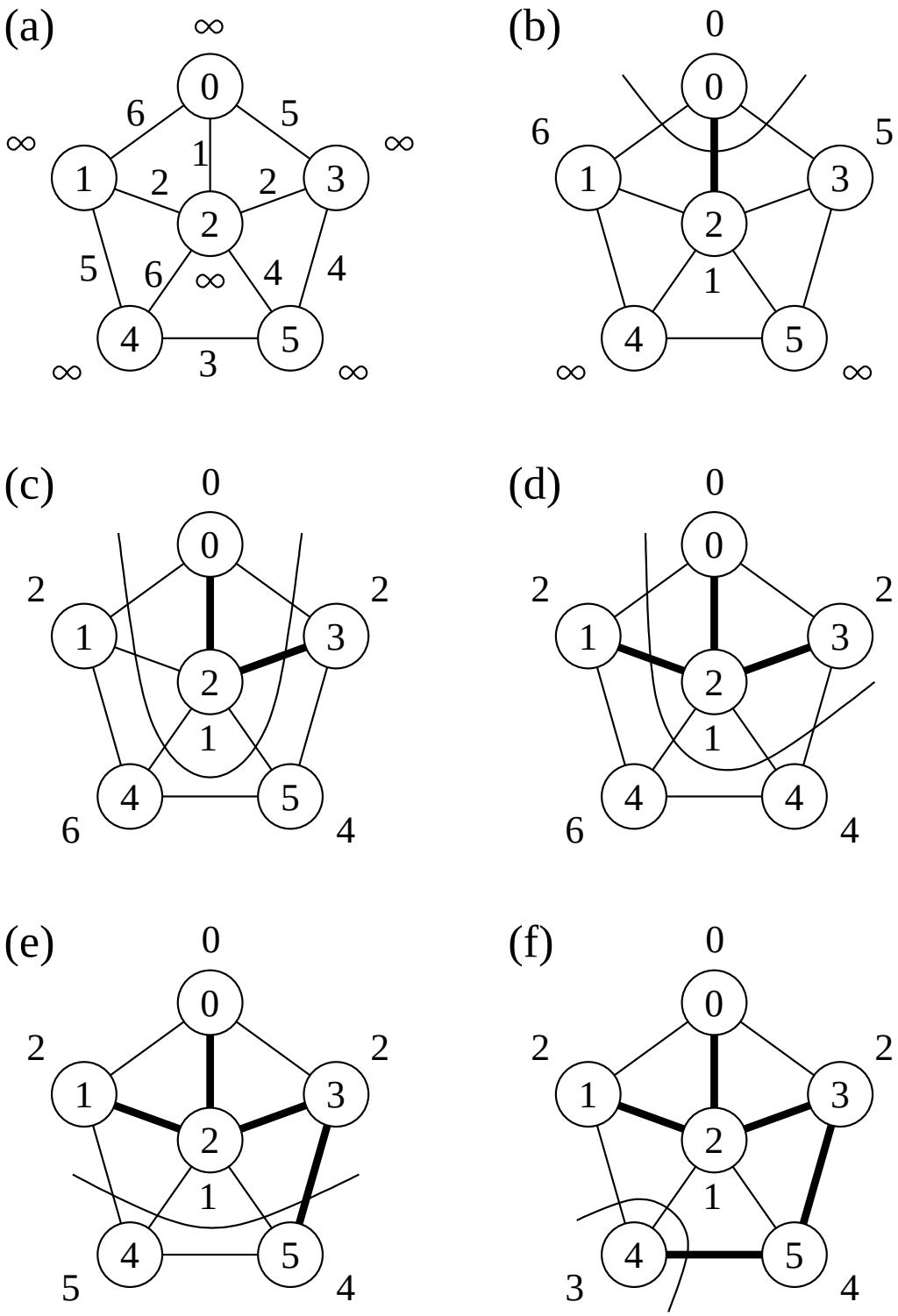
AGM - Teorema para reconhecer arestas seguras

- Seja $G = (V, A)$ um grafo conectado, não direcionado, com pesos p sobre as arestas V.
- seja S um subconjunto de V que está incluído em alguma AGM para G.
- Seja $(V', V - V')$ um corte qualquer que respeita S.
- Seja (u, v) uma aresta leve cruzando $(V', V - V')$.
- Satisfeitas essas condições, a aresta (u, v) é uma aresta segura para S.

AGM - Algoritmo de Prim

- O algoritmo de Prim para obter uma AGM pode ser derivado do algoritmo genérico.
- O subconjunto S forma uma única árvore, e a aresta segura adicionada a S é sempre uma aresta de peso mínimo conectando a árvore a um vértice que não esteja na árvore.
- A árvore começa por um vértice qualquer (no caso 0) e cresce até que “gere” todos os vértices em V.
- A cada passo, uma aresta leve é adicionada à árvore S, conectando S a um vértice de $G_S = (V, S)$.
- De acordo com o teorema anterior, quando o algoritmo termina, as arestas em S formam uma árvore geradora mínima.

Algoritmo de Prim - Exemplo



Algoritmo de Prim - Heap Indireto

```
package cap7;
public class FPHeapMinIndireto {
  private double p[];
  private int n, pos[], fp[];

  public FPHeapMinIndireto (double p[], int v[]) {
    this.p = p; this.fp = v; this.n = this.fp.length-1;
    this.pos = new int[this.n];
    for (int u = 0; u < this.n; u++) this.pos[u] = u+1;
  }
  public void refaz (int esq, int dir) {
    int j = esq * 2; int x = this.fp[esq];
    while (j <= dir) {
      if ((j < dir) && (this.p[fp[j]] > this.p[fp[j + 1]])) j++;
      if (this.p[x] <= this.p[fp[j]]) break;
      this.fp[esq] = this.fp[j]; this.pos[fp[j]] = esq;
      esq = j; j = esq * 2;
    }
    this.fp[esq] = x; this.pos[x] = esq;
  }
  public void constroi () {
    int esq = n / 2 + 1;
    while (esq > 1) { esq--; this.refaz (esq, this.n); }
  }
  public int retiraMin () throws Exception {
    int minimo;
    if (this.n < 1) throw new Exception ("Erro: heap vazio");
    else {
      minimo = this.fp[1]; this.fp[1] = this.fp[this.n];
      this.pos[fp[this.n--]] = 1; this.refaz (1, this.n);
    }
    return minimo;
  }
```

Algoritmo de Prim - Heap Indireto

```
public void diminuiChave (int i, double chaveNova) throws Exception {
   i = this.pos[i]; int x = fp[i];
   if (chaveNova < 0)
      throw new Exception ("Erro: chaveNova com valor incorreto");
   this.p[x] = chaveNova;
   while ((i > 1) && (this.p[x] <= this.p[fp[i / 2]])) {
      this.fp[i] = this.fp[i / 2]; this.pos[fp[i / 2]] = i; i /= 2;
   }
   this.fp[i] = x; this.pos[x] = i;
}
boolean vazio () { return this.n == 0; }
```

- O programa acima apresenta a classe $FPHeapMinIndireto$ com as estruturas de dados e as operações necessárias para operar com um *heap* indireto.
- O arranjo $pos[v]$ fornece a posição do vértice v dentro do *heap fp*, permitindo assim que o vértice v possa ser acessado a um custo $O(1)$.
- O acesso ao vértice v é necessário para a operação $diminuiChave$.

Algoritmo de Prim - Implementação

```
package cap7;
import cap7.listaadj.autoreferencia.Grafo;
public class AgmPrim {
  private int antecessor[];
  private double p[];
  private Grafo grafo;
  public AgmPrim (Grafo grafo) { this.grafo = grafo; }
  public void obterAgm (int raiz) throws Exception {
    int n = this.grafo.numVertices();
    this.p = new double[n]; // peso dos vértices
    int vs[] = new int[n+1]; // vértices
    boolean itensHeap[] = new boolean[n]; this.antecessor = new int[n];
    for (int u = 0; u < n; u ++) {
      this.antecessor[u] = -1;
      p[u] = Double.MAX_VALUE; // ∞
      vs[u+1] = u; // Heap indireto a ser construído
      itensHeap[u] = true;
    }
```

Algoritmo de Prim - Implementação

```
    p[raiz] = 0;
    FPHeapMinIndireto heap = new FPHeapMinIndireto (p, vs);
    heap.constroi ();
    while (!heap.vazio ()) {
      int u = heap.retiraMin (); itensHeap[u] = false;
      if (!this.grafo.listaAdjVazia (u)) {
        Grafo.Aresta adj = grafo.primeiroListaAdj (u);
        while (adj != null) {
          int v = adj.v2 ();
          if (itensHeap[v] && (adj.peso () < this.peso (v))) {
            antecessor[v] = u; heap.diminuiChave (v, adj.peso ());
          }
          adj = grafo.proxAdj (u);
        }
      }
    }

  public int antecessor (int u) { return this.antecessor[u]; }
  public double peso (int u) { return this.p[u]; }

  public void imprime () {
    for (int u = 0; u < this.p.length; u++)
      if (this.antecessor[u] != -1)
        System.out.println ("(" +antecessor[u]+ "," +u+ ") -- p:" +
                            peso (u));
  }
}
```

Algoritmo de Prim - Implementação

- A classe $AgmPrim$ implementa o algoritmo de Prim, cujo grafo de entrada G é fornecido através do construtor da classe $AgmPrim$.
- O método $obterAgm$ recebe o vértice *raiz* como entrada.
- O campo $antecessor[v]$ armazena o antecessor de v na árvore.
- Quando o algoritmo termina, a fila de prioridades *fp* está vazia, e a árvore geradora mínima S para G é:
- $$S = \{(v, antecessor[v]) : v \in V - \{raiz\}\}.$$
- Os métodos públicos $antecessor$, $peso$ e $imprime$ são utilizados para permitir ao usuário da classe $AgmPrim$ obter o antecessor de um certo vértice, obter o peso associado a um vértice e imprimir as arestas da árvore, respectivamente.

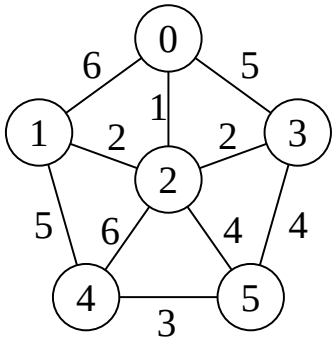
Algoritmo de Prim - Análise

- O corpo do anel **while** é executado $|V|$ vezes.
- O método $refaz$ tem custo $O(\log |V|)$.
- Logo, o tempo total para executar a operação retira o item com menor peso é $O(|V| \log |V|)$.
- O **while** mais interno para percorrer a lista de adjacentes é $O(|A|)$ (soma dos comprimentos de todas as listas de adjacência é $2|A|$).
- O teste para verificar se o vértice v pertence ao *heap* A tem custo $O(1)$.
- Após testar se v pertence ao *heap* e o peso da aresta (u, v) é menor do que $p[v]$, o antecessor de v é armazenado em $antecessor[v]$ e uma operação $diminuiChave$ é realizada sobre o *heap* na posição $pos[v]$, a qual tem custo $O(\log |V|)$.
- Logo, o tempo total para executar o algoritmo de Prim é $O(|V \log |V| + |A| \log |V|) = O(|A| \log |V|)$.

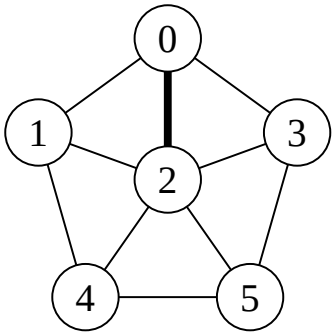
AGM - Algoritmo de Kruskal

- Pode ser derivado do algoritmo genérico.
- S é uma floresta e a aresta segura adicionada a S é sempre uma aresta de menor peso que conecta dois componentes distintos.
- Considera as arestas ordenadas pelo peso.

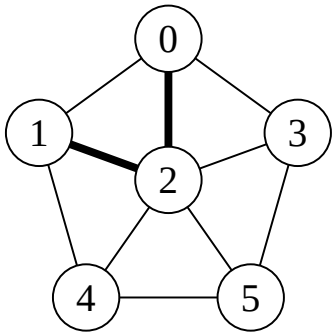
(a)



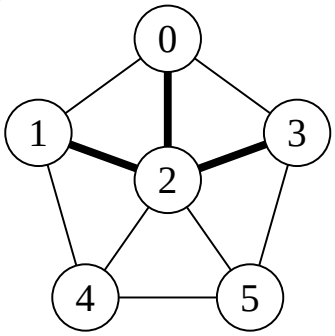
(b)



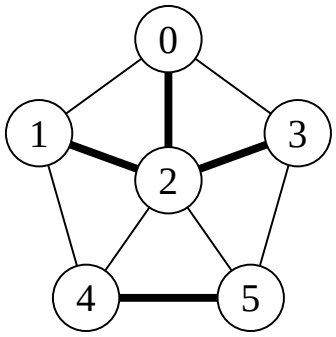
(c)



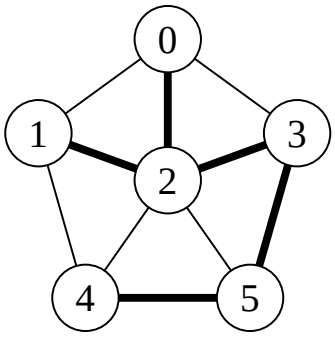
(d)



(e)



(f)



AGM - Algoritmo de Kruskal

- Sejam C_1 e C_2 duas árvores conectadas por (u, v):
  - Como (u, v) tem de ser uma aresta leve conectando C_1 com alguma outra árvore, (u, v) é uma aresta segura para C_1.
- É guloso porque, a cada passo, ele adiciona à floresta uma aresta de menor peso.
- Obtém uma AGM adicionando uma aresta de cada vez à floresta e, a cada passo, usa a aresta de menor peso que não forma ciclo.
- Inicia com uma floresta de $|V|$ árvores de um vértice: em $|V|$ passos, une duas árvores até que exista apenas uma árvore na floresta.

Algoritmo de Kruskal - Implementação

- Usa fila de prioridades para obter arestas em ordem crescente de pesos.
- Testa se uma dada aresta adicionada ao conjunto solução S forma um ciclo.
- Tratar **conjuntos disjuntos**: maneira eficiente de verificar se uma dada aresta forma um ciclo. Utiliza estruturas dinâmicas.
- Os elementos de um conjunto são representados por um objeto. Operações:
  - Criar um novo conjunto cujo único membro é x, o qual passa a ser seu representante.
  - Fazer a união de dois conjuntos dinâmicos cujos representantes são x e y. A operação une os conjuntos dinâmicos que contêm x e y, digamos C_x e C_y, em um novo conjunto que é a união desses dois conjuntos.
  - Encontrar o conjunto de um dado elemento x. Essa operação retorna uma referência ao representante do conjunto (único) contendo x.

Algoritmo de Kruskal - Implementação

- Primeiro refinamento:

```
   void Kruskal (Grafo grafo)
     ConjuntoDisjunto conj = new ConjuntoDisjunto ();
1.   S = ∅;
2.   for (int v=0; v<grafo.numVertices(); v++) conj.criaConjunto(v);
3.   Ordena as arestas de A pelo peso;
4.   for (cada (u, v) de A tomadas em ordem ascendente de peso)
5.     if (conj.encontraConjunto (u) != conj.encontraConjunto (v))
6.       S = S + {(u, v)};
7.       conj.uniao (u, v);
```

- A implementação das operações $uniao$ e $encontraConjunto$ deve ser realizada de forma eficiente.
- Esse problema é conhecido na literatura como **União-EncontraConjunto**.

AGM - Análise do Algoritmo de Kruskal

- A inicialização do conjunto S tem custo $O(1)$.
- Ordenar arestas (linha 3) custa $O(|A| \log |A|)$.
- A linha 2 realiza $|V|$ operações $criaConjunto$.
- O anel (linhas 4-7) realiza $O(|A|)$ operações $encontraConjunto$ e $uniao$, a um custo $O((|V| + |A|)\alpha(|V|))$ onde $\alpha(|V|)$ é uma função que cresce lentamente ($\alpha(|V|) < 4$).
- O limite inferior para construir uma estrutura dinâmica envolvendo m operações $encontraConjunto$ e $uniao$ e n operações $criaConjunto$ é $m\alpha(n)$.
- Como G é conectado temos que $|A| \geq |V| - 1$, e assim as operações sobre conjuntos disjuntos custam $O(|A|\alpha(|V|)$.
- Como $\alpha(|V|) = O(\log |A|) = O(\log |V|)$, o tempo total do algoritmo de Kruskal é $O(|A| \log |A|)$.
- Como $|A| < |V|^2$, então $\log |A| = O(\log |V|)$, e o custo do algoritmo de Kruskal é também $O(|A| \log |V|)$.

Caminhos Mais Curtos - Aplicação

- Um motorista procura o caminho mais curto entre Diamantina e Ouro Preto. Possui mapa com as distâncias entre cada par de interseções adjacentes.
- Modelagem:
  - $G = (V, A)$: grafo direcionado ponderado, mapa rodoviário.
  - V: interseções.
  - A: segmentos de estrada entre interseções
  - $p(u, v)$: peso de cada aresta, distância entre interseções.
- Peso de um caminho: $p(c) = \sum_{i=1}^{k} p(v_{i-1}, v_i)$
- Caminho mais curto:

$$\delta(u, v) = \begin{cases} min\left\{p(c) : u \overset{c}{\rightsquigarrow} v\right\} & \text{se existir caminho de } u \text{ a } v \\ \infty & \text{caso contrário} \end{cases}$$

- **Caminho mais curto** do vértice u ao vértice v: qualquer caminho c com peso $p(c) = \delta(u, v)$.

Caminhos Mais Curtos

- **Caminhos mais curtos a partir de uma origem**: dado um grafo ponderado $G = (V, A)$, desejamos obter o caminho mais curto a partir de um dado vértice origem $s \in V$ até cada $v \in V$.
- Muitos problemas podem ser resolvidos pelo algoritmo para o problema origem única:
  - **Caminhos mais curtos com destino único**: reduzido ao problema origem única invertendo a direção de cada aresta do grafo.
  - **Caminhos mais curtos entre um par de vértices**: o algoritmo para origem única é a melhor opção conhecida.
  - **Caminhos mais curtos entre todos os pares de vértices**: resolvido aplicando o algoritmo origem única $|V|$ vezes, uma vez para cada vértice origem.

Caminhos Mais Curtos

- A representação de caminhos mais curtos em um grafo $G = (V, A)$ pode ser realizada por um vetor chamado $antecessor$.
- Para cada vértice $v \in V$ o $antecessor[v]$ é um outro vértice $u \in V$ ou *null* (-1).
- O algoritmo atribui ao $antecessor$ os rótulos de vértices de uma cadeia de antecessores com origem em v e que anda para trás ao longo de um caminho mais curto até o vértice origem s.
- Dado um vértice v no qual $antecessor[v] \neq null$, o método $imprimeCaminho$ pode imprimir o caminho mais curto de s até v.

Caminhos Mais Curtos

- Os valores em $antecessor[v]$, em um passo intermediário, não indicam necessariamente caminhos mais curtos.
- Entretanto, ao final do processamento, $antecessor$ contém uma árvore de caminhos mais curtos definidos em termos dos pesos de cada aresta de G, ao invés do número de arestas.
- Caminhos mais curtos não são necessariamente únicos.

Árvore de caminhos mais curtos

- Uma árvore de caminhos mais curtos com raiz em $u \in V$ é um subgrafo direcionado $G' = (V', A')$, onde $V' \subseteq V$ e $A' \subseteq A$, tal que:
  1. V' é o conjunto de vértices alcançáveis a partir de $s \in G$,
  2. G' forma uma árvore de raiz s,
  3. para todos os vértices $v \in V'$, o caminho simples de s até v é um caminho mais curto de s até v em G.

Algoritmo de Dijkstra

- Mantém um conjunto S de vértices cujos caminhos mais curtos até um vértice origem já são conhecidos.
- Produz uma árvore de caminhos mais curtos de um vértice origem s para todos os vértices que são alcançáveis a partir de s.
- Utiliza a técnica de **relaxamento**:
  - Para cada vértice $v \in V$ o atributo $p[v]$ é um limite superior do peso de um caminho mais curto do vértice origem s até v.
  - O vetor $p[v]$ contém uma estimativa de um caminho mais curto.
- O primeiro passo do algoritmo é inicializar os antecessores e as estimativas de caminhos mais curtos:
  - $antecessor[v] = null$ para todo vértice $v \in V$,
  - $p[u] = 0$, para o vértice origem s, e
  - $p[v] = \infty$ para $v \in V - \{s\}$.

Relaxamento

- O **relaxamento** de uma aresta (u, v) consiste em verificar se é possível melhorar o melhor caminho até v obtido até o momento se passarmos por u.
- Se isto acontecer, $p[v]$ e $antecessor[v]$ devem ser atualizados.

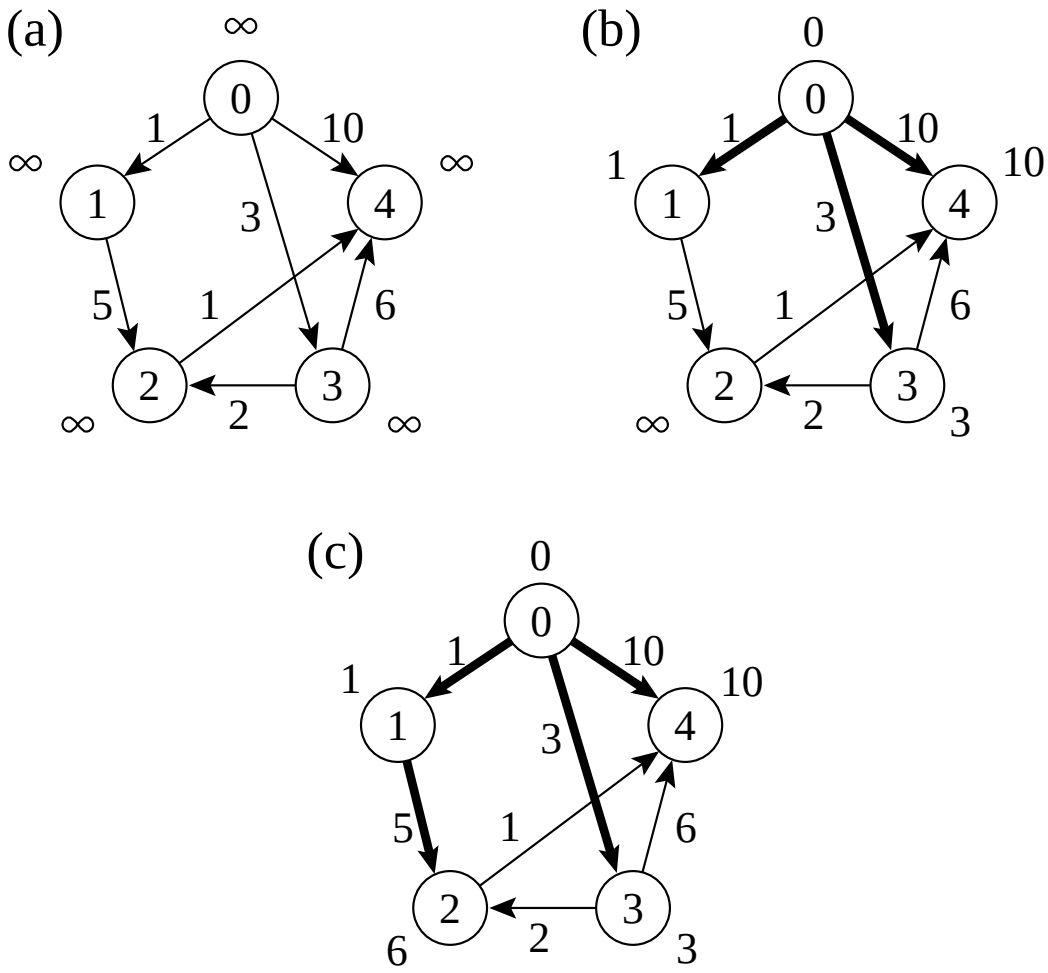
```
if (p[v] > p[u] + peso da aresta (u,v))
  p[v] = p[u] + peso da aresta (u,v);
  antecessor[v] = u;
```

Algoritmo de Dijkstra - 1º Refinamento

```
   dijkstra (Grafo grafo, int raiz)
1.   for (int v = 0; v < grafo.numVertices (); v++)
2.     p[v] = Infinito;
3.     antecessor[v] = −1;
4.   p[raiz] = 0;
5.   Constroi heap sobre vértices do grafo;
6    S = ∅;
7.   while (!heap.vazio ())
8.     u = heap.retiraMin ();
9      S = S + u;
10.    for (v ∈ grafo.listaAdjacentes (u))
11.      if (p[v] > p[u] + peso da aresta (u,v))
12.        p[v] = p[u] + peso da aresta (u,v);
13.        antecessor[v] = u;
```

- Invariante: o número de elementos do *heap* é igual a $V - S$ no início do anel **while**.
- A cada iteração do **while**, um vértice u é extraído do *heap* e adicionado ao conjunto S, mantendo assim o invariante.
- A operação$retiraMin$ obtém o vértice u com o caminho mais curto estimado até o momento e adiciona ao conjunto S.
- No anel da linha 10, a operação de relaxamento é realizada sobre cada aresta (u, v) adjacente ao vértice u.

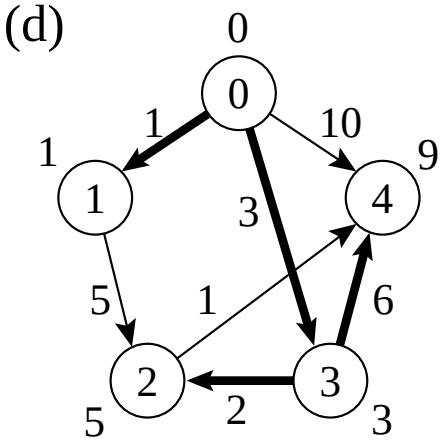
Algoritmo de Dijkstra - Exemplo



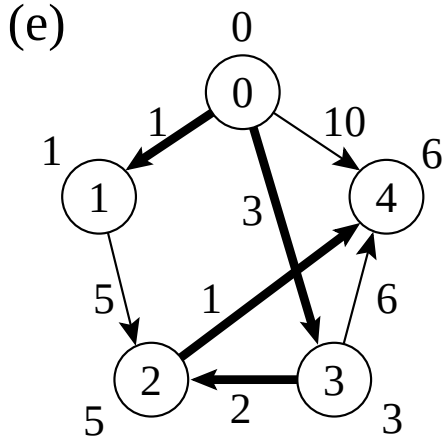
Iteração	S	d[0]	d[1]	d[2]	d[3]	d[4]
(a)	$\emptyset$	∞	∞	∞	∞	∞
(b)	$\{0\}$	0	1	∞	3	10
(c)	$\{0, 1\}$	0	1	6	3	10

Algoritmo de Dijkstra - Exemplo

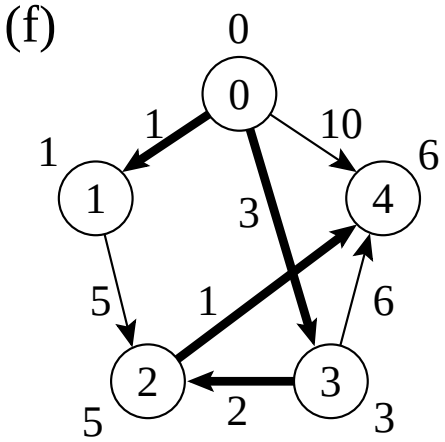
(d)



(e)



(f)



Iteração	S	d[0]	d[1]	d[2]	d[3]	d[4]
(d)	$\{0, 1, 3\}$	0	1	5	3	9
(e)	$\{0, 1, 3, 2\}$	0	1	5	3	6
(f)	$\{0, 1, 3, 2, 4\}$	0	1	5	3	6

Algoritmo de Dijkstra

- Para realizar de forma eficiente a seleção de uma nova aresta, todos os vértices que não estão na árvore de caminhos mais curtos residem no *heap* A baseada no campo *p*.
- Para cada vértice v, $p[v]$ é o caminho mais curto obtido até o momento, de v até o vértice raiz.
- O *heap* mantém os vértices, mas a condição do *heap* é mantida pelo caminho mais curto estimado até o momento através do arranjo $p[v]$, o *heap* é indireto.
- o arranjo $pos[v]$ fornece a posição do vértice v dentro do *heap*, permitindo assim que o vértice v possa ser acessado a um custo $O(1)$ para a operação $diminuiChave$.

Algoritmo de Dijkstra - Implementação

```
package cap7;
import cap7.listaadj.autoreferencia.Grafo;
public class Dijkstra {
  private int antecessor[];
  private double p[];
  private Grafo grafo;

  public Dijkstra (Grafo grafo) { this.grafo = grafo; }
  public void obterArvoreCMC (int raiz) throws Exception {
    int n = this.grafo.numVertices();
    this.p = new double[n]; // peso dos vértices
    int vs[] = new int[n+1]; // vértices
    this.antecessor = new int[n];
    for (int u = 0; u < n; u ++) {
      this.antecessor[u] = -1;
      p[u] = Double.MAX_VALUE; // ∞
      vs[u+1] = u; // Heap indireto a ser construído
    }
    p[raiz] = 0;
```

Algoritmo de Dijkstra - Implementação

```
FPHeapMinIndireto heap = new FPHeapMinIndireto (p, vs);
    heap.constroi ();
    while (!heap.vazio ()) {
      int u = heap.retiraMin ();
      if (!this.grafo.listaAdjVazia (u)) {
        Grafo.Aresta adj = grafo.primeiroListaAdj (u);
        while (adj != null) {
          int v = adj.v2 ();
          if (this.p[v] > (this.p[u] + adj.peso ())) {
            antecessor[v] = u;
            heap.diminuiChave (v, this.p[u] + adj.peso ());
          }
          adj = grafo.proxAdj (u);
        }
      }
    }
  public int antecessor (int u) { return this.antecessor[u]; }
  public double peso (int u) { return this.p[u]; }
  public void imprimeCaminho (int origem, int v) {
    if (origem == v) System.out.println (origem);
    else if (this.antecessor[v] == -1)
      System.out.println ("Nao existe caminho de " +origem+ " ate " +v);
    else {
      imprimeCaminho (origem, this.antecessor[v]);
      System.out.println (v);
    }
  }
}
```

Porque o Algoritmo de Dijkstra Funciona

- O algoritmo usa uma estratégia gulosa: sempre escolher o vértice mais leve (ou o mais perto) em $V - S$ para adicionar ao conjunto solução S,
- O algorimo de Dijkstra sempre obtém os caminhos mais curtos, pois cada vez que um vértice é adicionado ao conjunto S temos que $p[u] = \delta(raiz, u)$.

O Tipo Abstrato de Dados Hipergrafo

- Um **h**ipergrafo ou $r-$grafo é um grafo não direcionado $G = (V, A)$ no qual cada aresta $a \in A$ conecta r vértices, sendo r a ordem do hipergrafo.
- Os grafos estudados até agora são 2-grafos (ou hipergrafos de ordem 2).
- São utilizados para auxiliar na obtenção de funções de transformação perfeitas mínimas.
- A forma mais adequada para representar um hipergrafo é por meio de **l**istas de incidência.
- Em uma representação de um grafo não direcionado usando listas de incidência, para cada vértice v do grafo é mantida uma lista das arestas que incidem sobre o vértice v.
- Essa é uma estrutura orientada a arestas e não a vértices como as representações.
- Isso evita a duplicação das arestas ao se representar um grafo não direcionado pela versão direcionada correspondente.

O Tipo Abstrato de Dados Hipergrafo

- Operações de um tipo abstrato de dados hipergrafo:
  1. Criar um hipergrafo vazio.
  2. Inserir uma aresta no hipergrafo.
  3. Verificar se existe determinada aresta no hipergrafo.
  4. Obter a lista de arestas incidentes em determinado vértice.
  5. Retirar uma aresta do hipergrafo.
  6. Imprimir um hipergrafo.
  7. Obter o número de vértices do hipergrafo.
  8. Obter a aresta de menor peso de um hipergrafo.

O Tipo Abstrato de Dados Hipergrafo

- Uma operação que aparece com freqüência é a de obter a lista de arestas incidentes em determinado vértice.
- Para implementar esse operador precisamos de três operações sobre hipergrafos, a saber:
  1. Verificar se a lista de arestas incidentes em um vértice v está vazia.
  2. Obter a primeira aresta incidente a um vértice v, caso exista.
  3. Obter a próxima aresta incidente a um vértice v, caso exista.

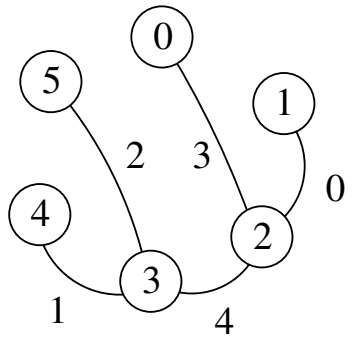
O Tipo Abstrato de Dados Hipergrafo

- A estrutura de dados usada para representar o hipergrafo é orientada a arestas
- As arestas são armazenadas em um arranjo chamado $arestas$.
- Em cada índice a do arranjo $arestas$, são armazenados os r vértices da aresta a e o seu $peso$.
- As listas de arestas incidentes nos vértices são armazenadas em dois arranjos: $prim$ (ponto de entrada para a lista de arestas incidentes) e $prox$ (as arestas subseqüentes).
- Valores armazenados nos arranjos $prim$ e $prox$ são obtidos pela equação $a + i|A|$, sendo $0 \leq i \leq r - 1$ e a um índice de uma aresta.
- Para se ter acesso a uma aresta a armazenada em $arestas[a]$ é preciso tomar os valores armazenados nos arranjos $prim$ e $prox$ módulo $|A|$.
- O valor -1 é utilizado para finalizar a lista.
- $prim$ deve possuir $|V|$ entradas.
- $prox$ deve possuir $r|A|$ entradas.

O Tipo Abstrato de Dados Hipergrafo - Exemplo

- Para descobrir quais são as arestas que contêm determinado vértice v é preciso percorrer a lista de arestas que inicia em $prim[v]$ e termina quando $prox[\ldots prim[v] \ldots] = -1$.
- Exemplo, ao se percorrer a lista das arestas do vértice 2, os valores $\{4, 8, 5\}$ são obtidos, os quais representam as arestas que contêm o vértice 2, ou seja, $\{4 \bmod 5 = 4, 8 \bmod 5 = 3, 5 \bmod 5 = 0\}$.

(a)



(b)

	0	1	2	3	4
arestas	(1,2,0)	(3,4,1)	(3,5,2)	(0,2,3)	(2,3,4)

	0	1	2	3	4	5
prim	3	0	4	9	6	7

	0	1	2	3	4	5	6	7	8	9
prox	−1	−1	1	−1	8	−1	−1	−1	5	2

O Tipo Abstrato de Dados Hipergrafo - Implementação

- A variável r é utilizada para armazenar a ordem do hipergrafo.
- $numVertices$ contém o número de vértices do hipergrafo.
- $proxDisponivel$ contém a próxima posição disponível para inserção de uma nova aresta.
- pos é utilizado para reter a posição atual na lista de incidência de um vértice v.

```
package cap7.listincidencia;
public class HiperGrafo {
  public static class Aresta {
    private int vertices[];
    private int peso;
    public Aresta (int vertices[], int peso) {
      this.vertices = vertices;
      this.peso = peso;
    }
    public int peso () { return this.peso; }
    public int vertice (int i) { return this.vertices[i]; }
    public int[] vertices () { return this.vertices; }
    public boolean equals (Object aresta) {
      Aresta a = (Aresta)aresta;
      if (a.vertices.length != this.vertices.length) return false;
      for (int i = 0; i < this.vertices.length; i++)
        if (this.vertices[i] != a.vertices[i]) return false;
      return true;
    }
```

O Tipo Abstrato de Dados Hipergrafo - Implementação

```
  public String toString () {
    String res = "{"; int i = 0;
    for (i = 0; i < this.vertices.length-1; i++)
      res += this.vertices[i] + ", ";
    res += this.vertices[i] + "} (" + this.peso + ")";
    return res;
  }
}
private int numVertices, proxDisponivel, r;
private Aresta arestas[];
private int prim[], prox[];
private int pos[];

public HiperGrafo (int numVertices, int numArestas, int r) {
  this.arestas = new Aresta[numArestas];
  this.prim = new int[numVertices];
  for (int i = 0; i < numVertices; i++) this.prim[i] = -1;
  this.prox = new int[r*numArestas];
  this.numVertices = numVertices;
  this.proxDisponivel = 0;
  this.r = r;
  this.pos = new int[numVertices];
}
```

O Tipo Abstrato de Dados Hipergrafo - Implementação

```
public void insereAresta (int vertices[], int peso) {
  if (this.proxDisponivel == this.arestas.length)
    System.out.println ("Nao ha espaco disponivel para a aresta");
  else {
    int a = this.proxDisponivel++; int n = this.arestas.length;
    this.arestas[a] = new Aresta (vertices, peso);
    for (int i = 0; i < this.r; i++) {
      int ind = a + i*n;
      this.prox[ind] = this.prim[this.arestas[a].vertices[i]];
      this.prim[this.arestas[a].vertices[i]] = ind;
    }
  }
}
public boolean existeAresta (int vertices[]) {
  for (int v = 0; v < this.r; v++)
    for (int i = this.prim[vertices[v]]; i != -1; i = this.prox[i]) {
      int a = i % this.arestas.length;
      if (this.arestas[a].equals (new Aresta (vertices, 0)))
        return true;
    }
  return false;
}
```

O Tipo Abstrato de Dados Hipergrafo - Implementação

```
public boolean listaIncVazia (int v) { return (this.prim[v] == -1); }
public Aresta primeiraListaInc (int v) {
  // Retorna a primeira aresta incidente no vértice v ou
  // null se a lista de arestas incidentes em v for vazia
  this.pos[v] = this.prim[v];
  int a = this.pos[v] % this.arestas.length;
  if (a >= 0) return this.arestas[a]; else return null;
}
public Aresta proxInc (int v) {
  // Retorna a próxima aresta incidente no vértice v ou null
  // se a lista de arestas incidentes em v estiver no fim
  this.pos[v] = this.prox[this.pos[v]];
  int a = this.pos[v] % this.arestas.length;
  if (a >= 0) return this.arestas[a]; else return null;
}
public Aresta retiraAresta (int vertices[]) {
  int n = this.arestas.length, a = 0; Aresta aresta = null;
  for (int i = 0; i < this.r; i++) {
    int prev = -1, aux = this.prim[vertices[i]];
    a = aux % n; aresta = new Aresta (vertices, 0);
    while ((aux >= 0) && (!this.arestas[a].equals (aresta))) {
      prev = aux; aux = this.prox[aux]; a = aux % n; }
    if (aux >= 0) { // achou
      if (prev == -1) this.prim[vertices[i]] = this.prox[aux];
      else this.prox[prev] = this.prox[aux];
      aresta = this.arestas[a];
    } else return null; // não achou
  }
  this.arestas[a] = null; // Marca como removido
  return aresta;
}
```

O Tipo Abstrato de Dados Hipergrafo - Implementação

```
  public void imprime () {
    for (int i = 0; i < this.numVertices; i++) {
      System.out.println ("Vertice " + i + ":");
      for (int j = this.prim[i]; j != -1; j = this.prox[j]) {
        int a = j % this.arestas.length;
        System.out.println ("  a: " + this.arestas[a]); }
    }
  }
  public int numVertices () { return this.numVertices; }
}
```

Processamento de Cadeias de Caracteres*

Última alteração: 10 de Outubro de 2006

*Transparências elaboradas por Charles Ornelas, Fabiano Botelho, Leonardo Rocha, Leonardo Mata e Nivio Ziviani

Definição e Motivação

- **Cadeia de caracteres**: seqüência de elementos denominados caracteres.
- Os caracteres são escolhidos de um conjunto denominado **alfabeto**.

  Ex.: em uma cadeia de *bits* o alfabeto é $\{0, 1\}$.
- **Casamento de cadeias de caracteres** ou **casamento de padrão**: encontrar todas as ocorrências de um padrão em um texto.
- Exemplos de aplicação:
  - edição de texto;
  - recuperação de informação;
  - estudo de seqüências de DNA em biologia computacional.

Notação

- Texto: arranjo $T[0..n-1]$ de tamanho n;
- Padrão: arranjo $P[0..m-1]$ de tamanho $m \leq n$.
- Os elementos de P e T são escolhidos de um alfabeto finito Σ de tamanho c.

  Ex.: $\Sigma = \{0, 1\}$ ou $\Sigma = \{a, b, \ldots, z\}$.
- **Casamento de cadeias** ou **casamento de padrão**: dados duas cadeias P (padrão) de comprimento $|P| = m$ e T (texto) de comprimento $|T| = n$, onde $n \gg m$, deseja-se saber as ocorrências de P em T.

Categorias de Algoritmos

- P e T não são pré-processados:
  - algoritmo seqüencial, on-line e de tempo-real;
  - padrão e texto não são conhecidos *a priori*.
  - complexidade de tempo $O(mn)$ e de espaço $O(1)$, para pior caso.
- P pré-processado:
  - algoritmo seqüencial;
  - padrão conhecido anteriormente permitindo seu pré-processamento.
  - complexidade de tempo $O(n)$ e de espaço $O(m + c)$, no pior caso.
  - ex.: programas para edição de textos.

Categorias de Algoritmos

- P e T são pré-processados:
  - algoritmo constrói índice.
  - complexidade de tempo $O(\log n)$ e de espaço é $O(n)$.
  - tempo para obter o índice é grande, $O(n)$ ou $O(n \log n)$.
  - compensado por muitas operações de pesquisa no texto.
  - Tipos de índices mais conhecidos:
    * Arquivos invertidos
    * Árvores *trie* e árvores Patricia
    * Arranjos de sufixos

Exemplos: P e T são pré-processados

- Diversos tipos de índices: arquivos invertidos, árvores *trie* e Patricia, e arranjos de sufixos.
- Um **arquivo invertido** possui duas partes: **vocabulário** e **ocorrências**.
- O vocabulário é o conjunto de todas as palavras distintas no texto.
- Para cada palavra distinta, uma lista de posições onde ela ocorre no texto é armazenada.
- O conjunto das listas é chamado de ocorrências.
- As posições podem referir-se a palavras ou caracteres.

Exemplo de Arquivo Invertido

0 6 15 21 25 35 44 52

Texto exemplo. Texto tem palavras. Palavras exercem fascínio.

exemplo	6
exercem	44
fascínio	52
palavras	25 35
tem	21
texto	0 15

Arquivo Invertido - Tamanho

- O vocabulário ocupa pouco espaço.
- A previsão sobre o crescimento do tamanho do vocabulário é dada pela lei de Heaps.
- **Lei de Heaps:** o vocabulário de um texto em linguagem natural contendo n palavras tem tamanho $V = Kn^{\beta} = O(n^{\beta})$, em que K e β dependem das características de cada texto.
- K geralmente assume valores entre 10 e 100, e β é uma constante entre 0 e 1, na prática ficando entre 0,4 e 0,6.
- O vocabulário cresce sublinearmente com o tamanho do texto, em uma proporção perto de sua raiz quadrada.
- As ocorrências ocupam muito mais espaço.
- Como cada palavra é referenciada uma vez na lista de ocorrências, o espaço necessário é $O(n)$.
- Na prática, o espaço para a lista de ocorrências fica entre 30% e 40% do tamanho do texto.

Arquivo Invertido - Pesquisa

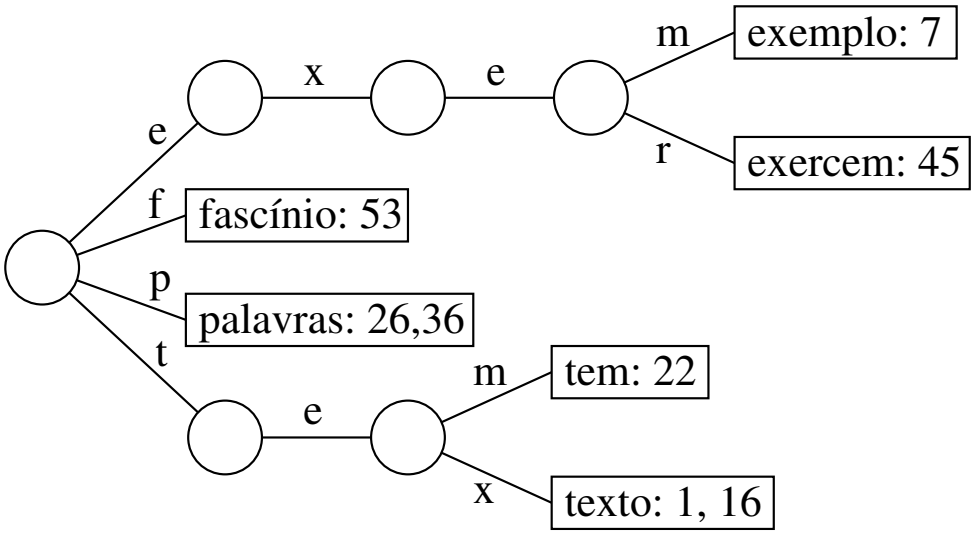
- A pesquisa tem geralmente três passos:
  - *Pesquisa no vocabulário*: palavras e padrões da consulta são isoladas e pesquisadas no vocabulário.
  - *Recuperação das ocorrências*: as listas de ocorrências das palavras encontradas no vocabulário são recuperadas.
  - *Manipulação das ocorrências*: as listas de ocorrências são processadas para tratar frases, proximidade, ou operações booleanas.
- Como a pesquisa em um arquivo invertido sempre começa pelo vocabulário, é interessante mantê-lo em um arquivo separado.
- Na maioria das vezes, esse arquivo cabe na memória principal.

Arquivo Invertido - Pesquisa

- A pesquisa de palavras simples pode ser realizada usando qualquer estrutura de dados que torne a busca eficiente, como *hashing*, árvore *trie* ou árvore B.
- As duas primeiras têm custo $O(m)$, onde m é o tamanho da consulta (independentemente do tamanho do texto).
- Guardar as palavras na ordem lexicográfica é barato em termos de espaço e competitivo em desempenho, já que a pesquisa binária pode ser empregada com custo $O(\log n)$, sendo n o número de palavras.
- A pesquisa por frases usando índices é mais difícil de resolver.
- Cada elemento da frase tem de ser pesquisado separadamente e suas listas de ocorrências recuperadas.
- A seguir, as listas têm de ser percorridas de forma sicronizada para encontrar as posições nas quais todas as palavras aparecem em seqüência.

Arquivo Invertido Usando Trie

- Arquivo invertido usando uma **árvore *trie*** para o texto: `Texto exemplo. Texto tem palavras. Palavras exercem fascínio.`



- O vocabulário lido até o momento é colocado em uma árvore *trie*, armazenando uma lista de ocorrências para cada palavra.
- Cada nova palavra lida é pesquisada na *trie*:
  - Se a pesquisa é sem sucesso, então a palavra é inserida na árvore e uma lista de ocorrências é inicializada com a posição da nova palavra no texto.
  - Senão, uma vez que a palavra já se encontra na árvore, a nova posição é inserida ao final da lista de ocorrências.

Casamento Exato

- Consiste em obter todas as ocorrências **exatas** do padrão no texto.

  Ex.: ocorrência exata do padrão `teste`.

```
   teste
os testes testam estes alunos . . .
```

- Dois enfoques:

1. leitura dos caracteres do texto um a um: algoritmos força bruta, Knuth-Morris-Pratt e Shift-And.
2. pesquisa de P em uma janela que desliza ao longo de T, pesquisando por um sufixo da janela que casa com um sufixo de P, por comparações da direita para a esquerda: algoritmos Boyer-Moore-Horspool e Boyer-Moore.

Casamento Exato - Métodos Considerados

- A classe $CasamentoExato$ apresenta a assinatura dos métodos de casamento exato implementados a seguir.
- $maxChar$ é utilizada para representar o tamanho do alfabeto considerado (caracteres ASCII).
- As cadeias de caracteres T e P são representadas por meio da classe ***String***.

```
package cap8;
public class CasamentoExato {
   private static final int maxChar = 256;
   // Assinatura dos métodos para casamento exato considerados
   public static void forcaBruta (String T, int n, String P, int m)
   public static void shiftAndExato (String T, int n, String P, int m)
   public static void bmh (String T, int n, String P, int m)
   public static void bmhs (String T, int n, String P, int m)
}
```

Força Bruta - Implementação

- É o algoritmo mais simples para casamento de cadeias.
- A idéia é tentar casar qualquer subcadeia no texto de comprimento m com o padrão.

```
public static void forcaBruta (String T, int n, String P, int m) {
  // Pesquisa P[0..m-1] em T[0..n-1]
  for (int i = 0; i < (n – m + 1); i ++) {
    int k = i; int j = 0;
    while ((j < m) && (T.charAt (k) == P.charAt (j))) { j++; k++; }
    if (j == m) System.out.println ("Casamento na posicao: " + i);
  }
}
```

Força Bruta - Análise

- Pior caso: $C_n = m \times n$.
- O pior caso ocorre, por exemplo, quando $P =$ aab e $T =$aaaaaaaaaa.
- Caso esperado:
  $\overline{C_n} = \frac{c}{c-1}\left(1 - \frac{1}{c^m}\right)(n - m + 1) + O(1)$
- O caso esperado é muito melhor do que o pior caso.
- Em experimento com texto randômico e alfabeto de tamanho $c = 4$, o número esperado de comparações é aproximadamente igual a 1,3.

Autômatos

- Um autômato é um modelo de computação muito simples.
- Um **autômato finito** é definido por uma tupla $(Q, I, F, \Sigma, \mathcal{T})$, onde Q é um conjunto finito de estados, entre os quais existe um estado inicial $I \in Q$, e alguns são estados finais ou estados de término $F \subseteq Q$.
- Transições entre estados são rotuladas por elementos de $\Sigma \cup \{\epsilon\}$, em que Σ é o alfabeto finito de entrada e ϵ é a transição vazia.
- As transições são formalmente definidas por uma função de transição $\mathcal{T}$.
- $\mathcal{T}$ associa a cada estado $q \in Q$ um conjunto $\{q_1, q_2, \ldots, q_k\}$ de estados de Q para cada $\alpha \in \Sigma \cup \{\epsilon\}$.

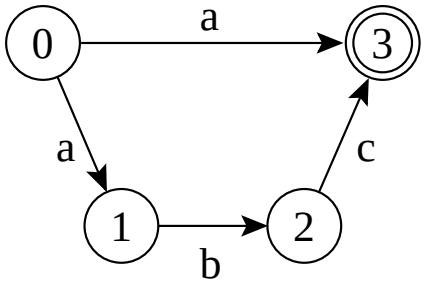
Tipos de Autômatos

- **Autômato finito não-determinista**:
  - Quando $\mathcal{T}$ é tal que existe um estado q associado a um dado caractere α para mais de um estado, digamos $\mathcal{T}(q, \alpha) = \{q_1, q_2, \ldots, q_k\}$, $k > 1$, ou existe alguma transição rotulada por ϵ.
  - Neste caso, a função de transição $\mathcal{T}$ é definida pelo conjunto de triplas $\Delta = \{(q, \alpha, q')$, onde $q \in Q$, $\alpha \in \Sigma \cup \{\epsilon\}$, e $q' \in \mathcal{T}(q, \alpha)$.
- **Autômato finito determinista**:
  - Quando a função de transição $\mathcal{T}$ é definida pela função $\delta = Q \times \Sigma \cup \epsilon \rightarrow Q$.
  - Neste caso, se $\mathcal{T}(q, \alpha) = \{q'\}$, então $\delta(q, \alpha) = q'$.

Exemplo de Autômatos

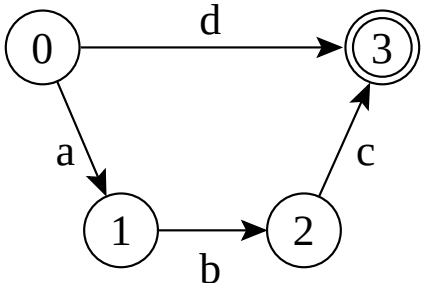
- Autômato finito não-determinista.

  A partir do estado 0, através do caractere de transição a é possível atingir os estados 2 e 3.



- Autômato finito determinista.

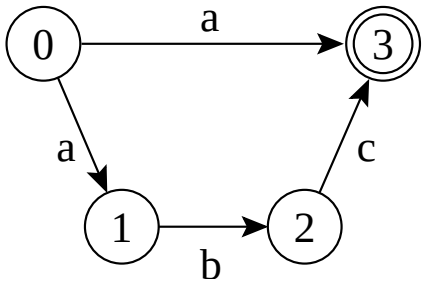
  Para cada caractere de transição todos os estados levam a um único estado.



Reconhecimento por Autômato

- Uma cadeia é **reconhecida** por $(Q, I, F, \Sigma, \Delta)$ ou $(Q, I, F, \Sigma, \delta)$ se qualquer um dos autômatos rotula um caminho que vai de um estado inicial até um estado final.
- A **linguagem reconhecida** por um autômato é o conjunto de cadeias que o autômato é capaz de reconher.

Ex.: a linguagem reconhecida pelo autômato abaixo é o conjunto de cadeias $\{a\}$ e $\{abc\}$ no estado 3.



Transições Vazias

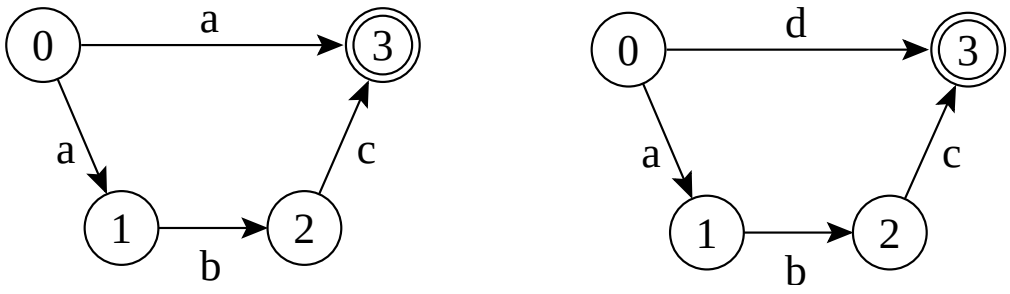
- São transições rotulada com uma cadeia vazia ϵ, também chamadas de **transições-ϵ**, em autômatos não-deterministas
- Não há necessidade de se ler um caractere para caminhar através de uma transição vazia.
- Simplificam a construção do autômato.
- Sempre existe um autômato equivalente que reconhece a mesma linguagem sem transições-ϵ.

Estados Ativos

- Se uma cadeia x rotula um caminho de I até um estado q então o estado q é considerado ativo depois de ler x.
- Um autômato finito determinista tem no máximo um estado ativo em um determinado instante.
- Um autômato finito não-determinista pode ter vários estados ativos.
- Casamento aproximado de cadeias pode ser resolvido por meio de autômatos finitos não-deterministas.

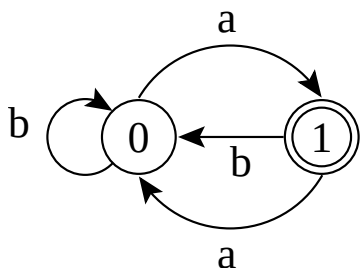
Ciclos em Autômatos

- Os autômatos abaixo são **acíclicos** pois as transições não formam ciclos.



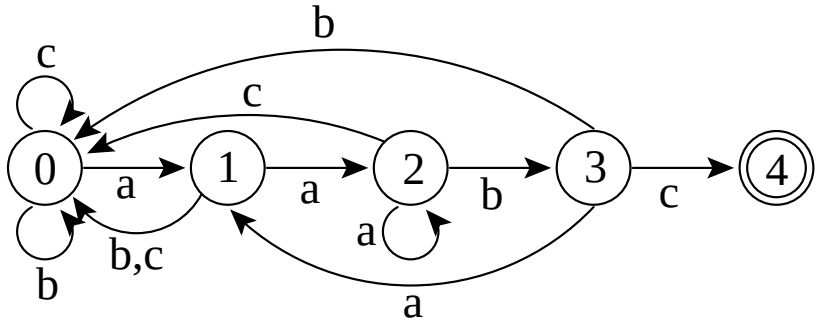
- **Autômatos finitos cíclicos**, deterministas ou não-deterministas, são úteis para **casamento de expressões regulares**
- A linguagem reconhecida por um autômato cíclico pode ser infinita.

Ex: o autômato abaixo reconhece ba, bba, bbba, bbbba, e assim por diante.



Exemplo de Uso de Autômato

- O autômato abaixo reconhece $P =$ {aabc}.



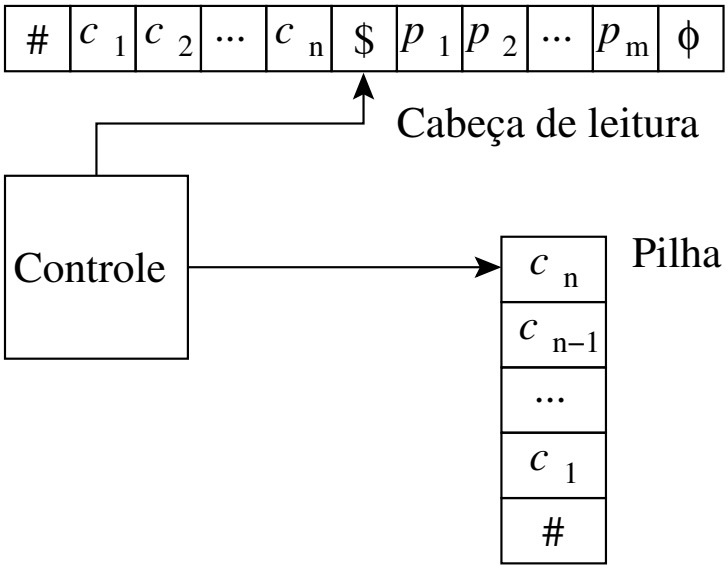
- A pesquisa de P sobre um texto T com alfabeto $\Sigma =$ {a, b, c} pode ser vista como a simulação do autômato na pesquisa de P sobre T.
- No início, o estado inicial ativa o estado 1.
- Para cada caractere lido do texto, a aresta correspondente é seguida, ativando o estado destino.
- Se o estado 4 estiver ativo e um caractere c é lido o estado final se torna ativo, resultando em um casamento de aabc com o texto.
- Como cada caractere do texto é lido uma vez, a complexidade de tempo é $O(n)$, e de espaço é $m + 2$ para vértices e $|\Sigma| \times m$ para arestas.

Knuth-Morris-Pratt (KMP)

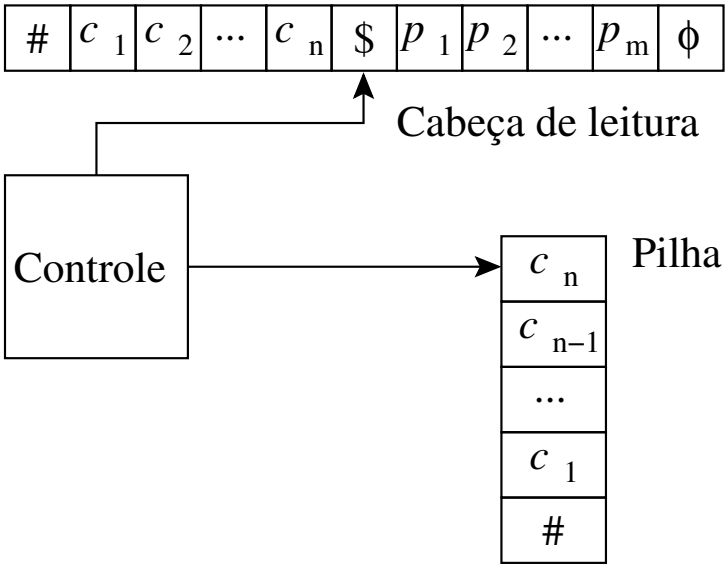
- O KMP é o primeiro algoritmo (1977) cujo pior caso tem complexidade de tempo linear no tamanho do texto, $O(n)$.
- É um dos algoritmos mais famosos para resolver o problema de casamento de cadeias.
- Tem implementação complicada e na prática perde em eficiência para o Shift-And e o Boyer-Moore-Horspool.
- Até 1971, o limite inferior conhecido para busca exata de padrões era $O(mn)$.

KMP - 2DPDA

- Em 1971, Cook provou que qualquer problema que puder ser resolvido por um autômato determinista de dois caminhos com memória de pilha (*Two-way Deterministic Pushdown Store Automaton*, 2DPDA) pode ser resolvido em tempo linear por uma máquina RAM.
- O 2DPDA é constituído de:
  - uma fita apenas para leitura;
  - uma pilha de dados (memória temporária);
  - um controle de estado que permite mover a fita para esquerda ou direita, empilhar ou desempilhar símbolos, e mudar de estado.



KMP - Casamento de cadeias no 2DPDA



- No autômato da acima, a entrada é constituída da cadeia $\#c_1c_2 \cdots c_n\$p_1p_2 \cdots p_m\phi$.
- A partir de $\#$ todos os caracteres são empilhados até encontrar o caractere $\$$.
- A leitura cotinua até encontrar o caractere ϕ.
- A seguir a leitura é realizada no sentido contrário, iniciando por p_n, comparado-o com o último caractere empilhado, no caso c_n.
- Esta operação é repetida para os caracteres seguintes, e se o caractere $\$$ for atingido então as duas cadeias são iguais.

KMP - Algoritmo

- Primeira versão do KMP é uma simulação linear do 2DPDA
- O algoritmo computa o sufixo mais longo no texto que é também o prefixo de P.
- Quando o comprimento do sufixo no texto é igual a $|P|$ ocorre um casamento.
- O pré-processamento de P permite que nenhum caractere seja reexaminado.
- O apontador para o texto nunca é decrementado.
- O pré-processamento de P pode ser visto como a construção econômica de um autômato determinista que depois é usado para pesquisar pelo padrão no texto.

Shift-And

- O Shift-And é vezes mais rápido e muito mais simples do que o KMP.
- Pode ser estendido para permitir casamento aproximado de cadeias de caracteres.
- Usa o conceito de **paralelismo de bit**:
  - técnica que tira proveito do paralelismo intrínseco das operações sobre *bits* dentro de uma palavra de computador.
  - É possível empacotar muitos valores em uma única palavra e atualizar todos eles em uma única operação.
- Tirando proveito do paralelismo de *bit*, o número de operações que um algoritmo realiza pode ser reduzido por um fator de até w, onde w é o número de *bits* da palavra do computador.

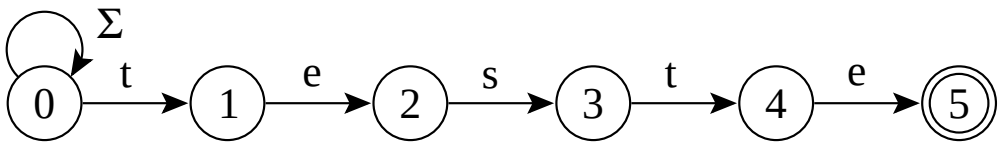
Shift-And - Notação para Operações Usando Paralelismo de *bit*

- Para denotar **repetição de *bit*** é usado exponenciação: $01^3 = 0111$.
- Uma seqüência de *bits* $b_0 \ldots b_{c-1}$ é chamada de **máscara de bits** de comprimento c, e é armazenada em alguma posição de uma palavra w do computador.
- Operações sobre os *bits* da palavra do computador:
  - "$|$": operação *or*;
  - "$\&$": operação *and*;
  - "$\sim$": complementa todos os *bits*;
  - "$>>$": move os *bits* para a direita e entra com zeros à esquerda (por exemplo, $b_0, b_1, \ldots, b_{c-2}, b_{c-1} >> 2 = 00b_0, b_1, \ldots, b_{c-3}$).

Shift-And - Princípio de Funcionamento

- Mantém um conjunto de todos os prefixos de P que casam com o texto já lido.
- Utiliza o paralelismo de *bit* para atualizar o conjunto a cada caractere lido do texto.
- Este conjunto é representado por uma máscara de *bits* $R = (b_0, b_1, \ldots, b_{m-1})$.
- O algoritmo Shift-And pode ser visto como a simulação de um autômato que pesquisa pelo padrão no texto (não-determinista para simular o paralelismo de *bit*).

Ex.: Autômato não-determinista que reconhece todos os prefixos de $P =$`{teste}`



Shift-And - Algoritmo

- O valor 1 é colocado na j-ésima posição de $R = (b_0, b_1, \ldots, b_{m-1})$ se e somente se $p_0 \ldots p_j$ é um sufixo de $t_0 \ldots t_i$, onde i corresponde à posição corrente no texto.
- A j-ésima posição de R é dita estar *ativa*.
- b_{m-1} ativo significa um casamento.
- R', o novo valor do conjunto R, é calculado na leitura do próximo caractere t_{i+1}.
- A posição $j + 1$ no conjunto R' ficará ativa se e somente se a posição j estiver ativa em R e t_{i+1} casa com p_{i+1} ($p_0 \ldots p_j$ era um sufixo de $t_0 \ldots t_i$ e t_{i+1} casa com p_{j+1}).
- Com o uso de paralelismo de *bit* é possível computar o novo conjunto com custo $O(1)$.

Shift-And - Pré-processamento

- O primeiro passo é a construção de uma tabela M para armazenar uma máscara de *bits* $b_0 \ldots, b_{m-1}$ para cada caractere.

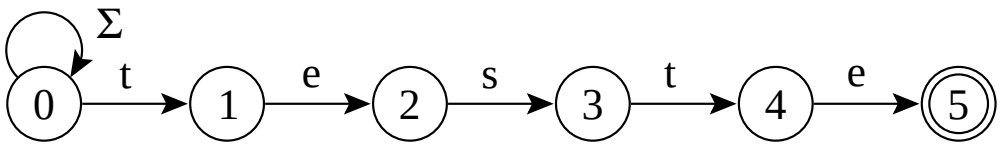
  Ex.: máscaras de *bits* para os caracteres presentes em $P =$`teste`.

	0	1	2	3	4
M[t]	1	0	0	1	0
M[e]	0	1	0	0	1
M[s]	0	0	1	0	0

- A máscara em $M[$`t`$]$ é 10010, pois o caractere `t` aparece nas posições 0 e 3 de P.

Shift-And - Pesquisa

- O valor do conjunto é inicializado como $R = 0^m$ (0^m significa 0 repetido m vezes).
- Para cada novo caractere t_{i+1} lido do texto o valor do conjunto R' é atualizado: $R' = ((R >> 1) \mid 10^{m-1}) \ \& \ M[T[i]]$.
- A operação "$>>$" desloca todas as posições para a direita no passo $i + 1$ para marcar quais posições de P eram sufixos no passo i.
- A cadeia vazia ϵ também é marcada como um sufixo, permitindo um casamento na posição corrente do texto (*self-loop* no início do autômato).



- Do conjunto obtido até o momento, são mantidas apenas as posições que t_{i+1} casa com p_{j+1}, obtido com a operação *and* desse conjunto de posições com o conjunto $M[t_{i+1}]$ de posições de t_{i+1} em P.

Exemplo de funcionamento do Shif-And

Pesquisa do padrão $P =$`{teste}` no texto $T =$`{os testes ...}`.

Texto	$(R >> 1)\|10^{m-1}$					R'				
o	1	0	0	0	0	0	0	0	0	0
s	1	0	0	0	0	0	0	0	0	0
	1	0	0	0	0	0	0	0	0	0
t	1	0	0	0	0	1	0	0	0	0
e	1	1	0	0	0	0	1	0	0	0
s	1	0	1	0	0	0	0	1	0	0
t	1	0	0	1	0	1	0	0	1	0
e	1	1	0	0	1	0	1	0	0	***1***
s	1	0	1	0	0	0	0	1	0	0
	1	0	0	1	0	0	0	0	0	0

Shift-And - Implementação

```
void Shift-And (P = p_0 p_1 ... p_{m-1}, T = t_0 t_1 ... t_{n-1})
  // Pré-processamento
  for (c ∈ Σ) M[c] = 0^m;
  for (j = 0; j < m; j++) M[p_j] = M[p_j] | 0^j 10^{m-j-1};
  // Pesquisa
  R = 0^m;
  for (i = 0; i < n; i++)
    R = ((R >> 1) | 10^{m-1}) & M[T[i]];
    if (R & 0^{m-1}1 ≠ 0^m) "Casamento na posicao i - m + 1";
```

- **Análise**: O custo do algoritmo Shift-And é $O(n)$, desde que as operações possam ser realizadas em $O(1)$ e o padrão caiba em umas poucas palavras do computador.

Boyer-Moore-Horspool (BMH)

- Em 1977, foi publicado o algoritmo Boyer-Moore (BM).
- A idéia é pesquisar no padrão no sentido da direita para a esquerda, o que torna o algoritmo muito rápido.
- Em 1980, Horspool apresentou uma simplificação no algoritmo original, tão eficiente quanto o algoritmo original, ficando conhecida como Boyer-Moore-Horspool (BMH).
- Pela extrema simplicidade de implementação e comprovada eficiência, o BMH deve ser escolhido em aplicações de uso geral que necessitam realizar casamento exato de cadeias.

Funcionamento do BM e BMH

- O BM e BMH pesquisa o padrão P em uma janela que desliza ao longo do texto T.
- Para cada posição desta janela, o algoritmo pesquisa por um sufixo da janela que casa com um sufixo de P, com comparações realizadas no sentido da direita para a esquerda.
- Se não ocorrer uma desigualdade, então uma ocorrência de P em T ocorreu.
- Senão, o algoritmo calcula um deslocamento que o padrão deve ser deslizado para a direita antes que uma nova tentativa de casamento se inicie.
- O BM original propõe duas heurísticas para calcular o deslocamento: ocorrência e casamento.

BM - Heurística Ocorrência

- Alinha a posição no texto que causou a colisão com o primeiro caractere no padrão que casa com ele;

Ex.: P ={cacbac}, T ={aabcaccacbac}.

```
0 1 2 3 4 5 6 7 8 9 0 1
c a c b a c
a a b c a c c a c b a c
  c a c b a c
    c a c b a c
      c a c b a c
        c a c b a c
```

- A partir da posição 5, da direita para a esquerda, existe uma colisão na posição 3 de T, entre b do padrão e c do texto.
- Logo, o padrão deve ser deslocado para a direita até o primeiro caractere no padrão que casa com c.
- O processo é repetido até encontrar um casamento a partir da posição 6 de T.

BM - Heurística Casamento

- Ao mover o padrão para a direita, faça-o casar com o pedaço do texto anteriormente casado.

Ex.: P ={cacbac} no texto T ={aabcaccacbac}.

0 1 2 3 4 5 6 7 8 9 0 1

c a c b a c

a a b c a c c a c b a c

c a c b a c

c a c b a c

- Novamente, a partir da posição 5, da direita para a esquerda, existe uma colisão na posição 3 de T, entre o b do padrão e o c do texto.
- Neste caso, o padrão deve ser deslocado para a direita até casar com o pedaço do texto anteriormente casado, no caso ac, deslocando o padrão 3 posições à direita.
- O processo é repetido mais uma vez e o casamento entre P e T ocorre.

Escolha da Heurística

- O algoritmo BM escolhe a heurística que provoca o maior deslocamento do padrão.
- Esta escolha implica em realizar uma comparação entre dois inteiros para cada caractere lido do texto, penalizando o desempenho do algoritmo com relação a tempo de processamento.
- Várias propostas de simplificação ocorreram ao longo dos anos.
- As que produzem os melhores resultados são as que consideram apenas a heurística ocorrência.

Algoritmo Boyer-Moore-Horspool (BMH)

- A simplificação mais importante é devida a Horspool em 1980.
- Executa mais rápido do que o algoritmo BM original.
- Parte da observação de que qualquer caractere já lido do texto a partir do último deslocamento pode ser usado para endereçar a tabela de deslocamentos.
- Endereça a tabela com o caractere no texto correspondente ao último caractere do padrão.

BMH - Tabela de Deslocamentos

- Para pré-computar o padrão o valor inicial de todas as entradas na tabela de deslocamentos é feito igual a m.
- A seguir, apenas para os $m-1$ primeiros caracteres do padrão são usados para obter os outros valores da tabela.
- Formalmente, $d[x] = min\{j \text{ tal que } j = m \mid (1 \leq j < m \ \& \ P[m-j-1] = x)\}$.

Ex.: Para o padrão $P =$`{teste}`, os valores de d são $d[\texttt{t}] = 1$, $d[\texttt{e}] = 3$, $d[\texttt{s}] = 2$, e todos os outros valores são iguais ao valor de $|P|$, nesse caso $m = 5$.

BMH - Implementação

```
public static void bmh (String T, int n, String P, int m) {
  // Pré-processamento do padrão
  int d[] = new int[maxChar];
  for (int j = 0; j < maxChar; j++) d[j] = m;
  for (int j = 0; j < (m-1); j++) d[(int)P.charAt (j)] = m - j - 1;
  int i = m - 1;
  while (i < n) { // Pesquisa
    int k = i; int j = m - 1;
    while ((j >= 0) && (T.charAt (k) == P.charAt (j))) { j--; k--; }
    if (j < 0)
      System.out.println ("Casamento na posicao: " + (k + 1));
    i = i + d[(int)T.charAt (i)];
  }
}
```

- $d[(int)T.charAt(i)]$ equivale ao endereço na tabela d do caractere que está na i-ésima posição no texto, a qual corresponde à posição do último caractere de P.

Algoritmo BMHS - Boyer-Moore-Horspool-Sunday

- Sunday (1990) apresentou outra simplificação importante para o algoritmo BM, ficando conhecida como BMHS.
- Variante do BMH: endereçar a tabela com o caractere no texto correspondente ao próximo caractere após o último caractere do padrão, em vez de deslocar o padrão usando o último caractere como no algoritmo BMH.
- Para pré-computar o padrão, o valor inicial de todas as entradas na tabela de deslocamentos é feito igual a $m + 1$.
- A seguir, os m primeiros caracteres do padrão são usados para obter os outros valores da tabela.
- Formalmente $d[x] = min\{j \text{ tal que } j = m \mid (1 \leq j \leq m \ \& \ P[m - j] = x)\}$.
- Para o padrão $P = \texttt{teste}$, os valores de d são $d[\texttt{t}] = 2$, $d[\texttt{e}] = 1$, $d[\texttt{s}] = 3$, e todos os outros valores são iguais ao valor de $|P| + 1$.

BMHS - Implementação

- O pré-processamento do padrão ocorre nas duas primeiras linhas do código.
- A fase de pesquisa é constituída por um anel em que i varia de $m-1$ até $n-1$, com incrementos $d[(int)T.charAt(i+1)]$, o que equivale ao endereço na tabela d do caractere que está na $i+1$-ésima posição no texto, a qual corresponde à posição do último caractere de P.

```
public static void bmhs (String T, int n, String P, int m) {
  // Pré-processamento do padrão
  int d[] = new int[maxChar];
  for (int j = 0; j < maxChar; j++) d[j] = m + 1;
  for (int j = 0; j < m; j++) d[(int)P.charAt (j)] = m - j;
  int i = m - 1;
  while (i < n) { // Pesquisa
    int k = i; int j = m - 1;
    while ((j >= 0) && (T.charAt (k) == P.charAt (j))) { j--; k--; }
    if (j < 0)
      System.out.println ("Casamento na posicao: " + (k + 1));
    i = i + d[(int)T.charAt (i+1)];
  }
}
```

BH - Análise

- Os dois tipos de deslocamento (ocorrência e casamento) podem ser pré-computados com base apenas no padrão e no alfabeto.
- Assim, a complexidade de tempo e de espaço para esta fase é $O(m + c)$.
- O pior caso do algoritmo é $O(nm)$.
- O melhor caso e o caso médio para o algoritmo é $O(n/m)$, um resultado excelente pois executa em tempo sublinear.

BMH - Análise

- O deslocamento ocorrência também pode ser pré-computado com base apenas no padrão e no alfabeto.
- A complexidade de tempo e de espaço para essa fase é $O(m + c)$.
- Para a fase de pesquisa, o pior caso do algoritmo é $O(nm)$, o melhor caso é $O(n/m)$ e o caso esperado é $O(n/m)$, se c não é pequeno e m não é muito grande.

BMHS - Análise

- Na variante BMHS, seu comportamento assintótico é igual ao do algoritmo BMH.
- Entretanto, os deslocamentos são mais longos (podendo ser iguais a $m + 1$), levando a saltos relativamente maiores para padrões curtos.
- Por exemplo, para um padrão de tamanho $m = 1$, o deslocamento é igual a $2m$ quando não há casamento.

Casamento Aproximado

- O casamento aproximado de cadeias permite operações de inserção, substituição e retirada de caracteres do padrão.

Ex.: Três ocorrências do padrão `teste` em que os casos de inserção, substituição, retirada de caracteres no padrão acontecem:

1. um espaço é inserido entre o terceiro e quarto caracteres do padrão;
2. o último caractere do padrão é substituído pelo caractere a;
3. o primeiro caractere do padrão é retirado.

```
      tes te
          testa
                 este
os testes testam estes alunos . . .
```

Distância de Edição

- É número k de operações de inserção, substituição e retirada de caracteres necessário para transformar uma cadeia x em outra cadeia y.
- $ed(P, P')$: distância de edição entre duas cadeias P e P'; é o menor número de operações necessárias para converter P em P', ou vice versa.

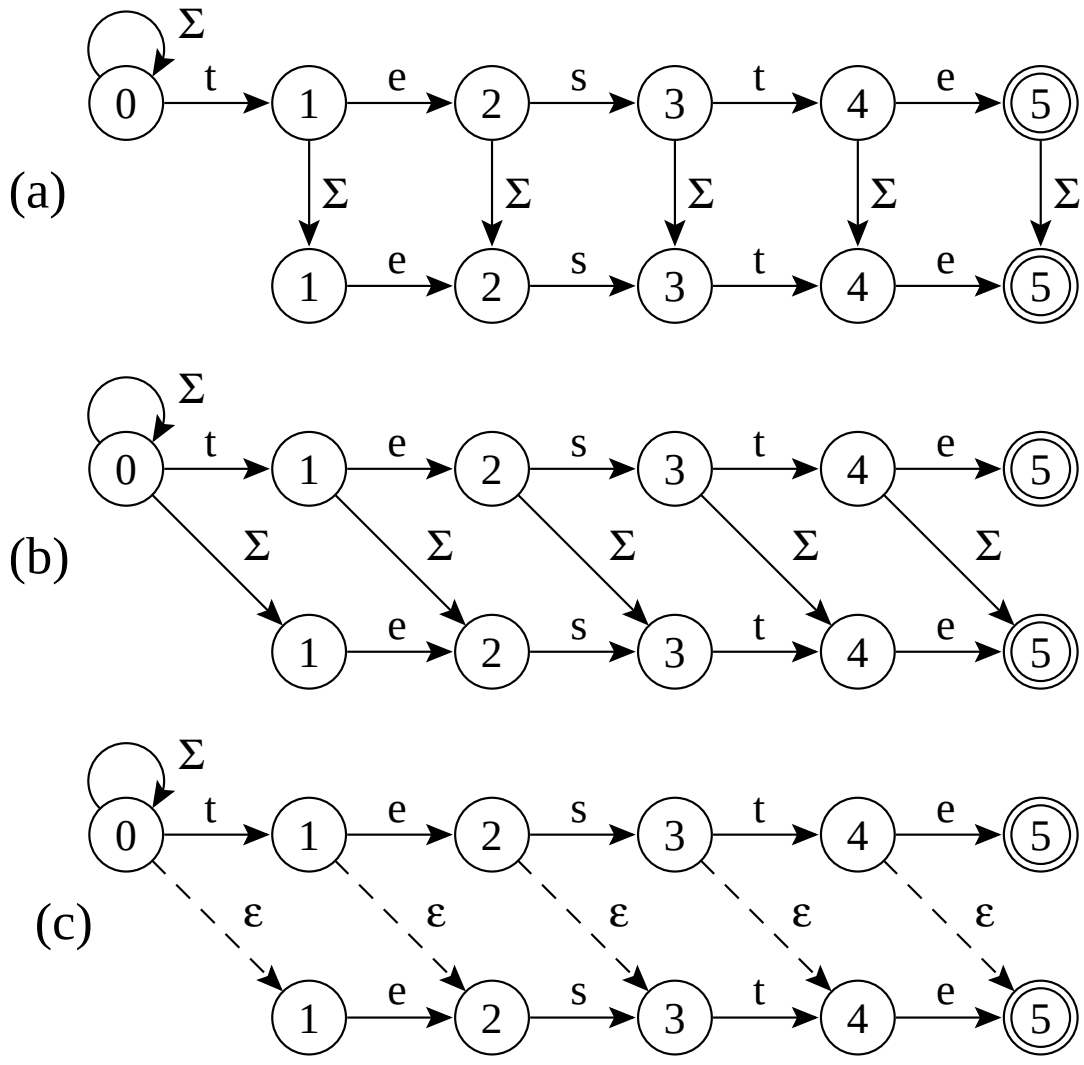
  Ex.: $ed(\texttt{t}este, \texttt{e}stende) = 4$, valor obtido por meio de uma retirada do primeiro t de P e a inserção dos 3 caracteres nde ao final de P.
- O problema do casamento aproximado de cadeias é o de encontrar todas as ocorrências em T de cada P' que satisfaz $ed(P, P') \leq k$.

Casamento Aproximado

- A busca aproximada só faz sentido para $0 < k < m$, pois para $k = m$ toda subcadeia de comprimento m pode ser convertida em P por meio da substituição de m caracteres.
- O caso em que $k = 0$ corresponde ao casamento exato de cadeias.
- O nível de erro $\alpha = k/m$, fornece uma medida da fração do padrão que pode ser alterado.
- Em geral $\alpha < 1/2$ para a maioria dos casos de interesse.
- **Casamento aproximado de cadeias**, ou **casamento de cadeias permitindo erros**: um número limitado k de operações (erros) de inserção, de substituição e de retirada é permitido entre P e suas ocorrências em T.
- A pesquisa com casamento aproximado é modelado por autômato não-determinista.
- O algoritmo de casamento aproximado de cadeias usa o **paralelismo de bit**.

Exemplo de Autômato para Casamento Aproximado

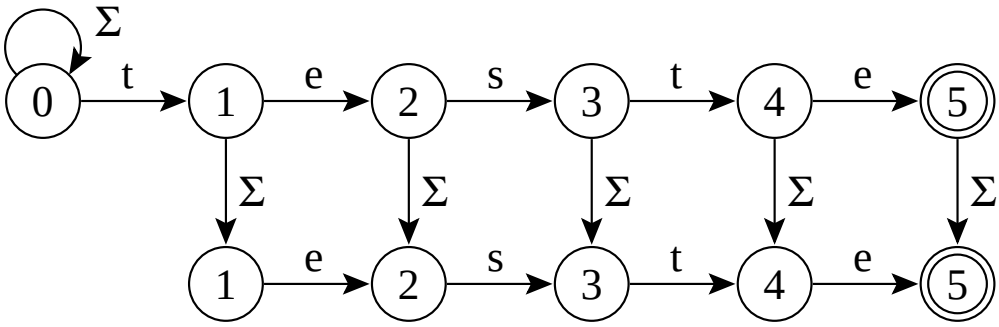
- $P =$`{teste}` e $k = 1$: (a)inserção; (b)substituição e (c)retirada.



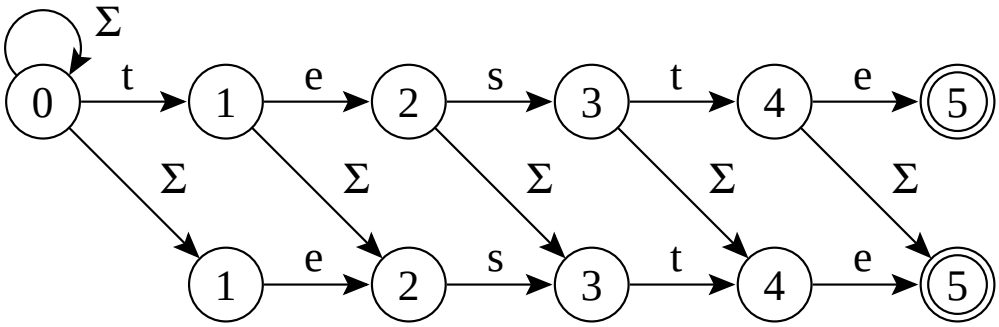
- Casamento de caractere é representado por uma aresta horizontal. Avançamos em P e T.
- O *self-loop* permite que uma ocorrência se inicie em qualquer posição em T.

Exemplo de Autômato para Casamento Aproximado

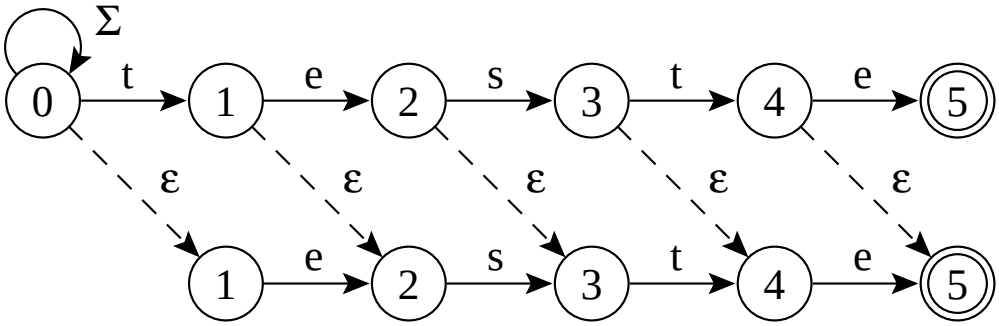
- Uma aresta vertical insere um caractere no padrão. Avançamos em T mas não em P.



- Uma aresta diagonal sólida substitui um caractere. Avançamos em T e P.

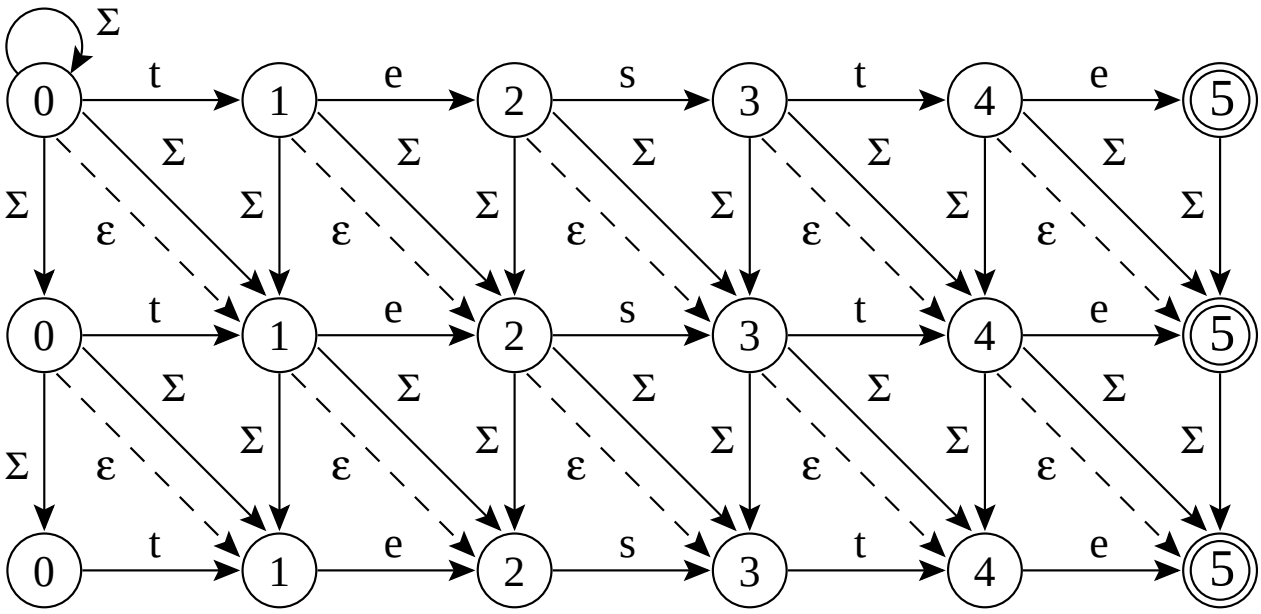


- Uma aresta diagonal tracejada retira um caractere. Avançamos em P mas não em T (transição-ϵ)



Exemplo de Autômato para Casamento Aproximado

- $P =$ {`teste`} e $K = 2$.
- As três operações de distância de edição estão juntas em um único autômato:
  - Linha 1: casamento exato ($k = 0$);
  - Linha 2: casamento aproximado permitindo um erro ($k = 1$);
  - Linha 3: casamento aproximado permitindo dois erros ($k = 2$).



- Uma vez que um estado no autômato está ativo, todos os estados nas linhas seguintes na mesma coluna também estão ativos.

Shift-And para Casamento Aproximado

- Utiliza **paralelismo de bit**.
- Simula um autômato não-determinista.
- Empacota cada linha j ($0 < j \leq k$) do autômato não-determinista em uma palavra R_j diferente do computador.
- Para cada novo caractere lido do texto todas as transições do autômato são simuladas usando operações entre as $k + 1$ máscaras de *bits*.
- Todas as $k + 1$ máscaras de *bits* têm a mesma estrutura e assim o mesmo *bit* é alinhado com a mesma posição no texto.

Shift-And para Casamento Aproximado

- Na posição i do texto, os novos valores R'_j, $0 < j \leq k$, são obtidos a partir dos valores correntes R_j:
  - $R'_0 = ((R_0 >> 1) \mid 10^{m-1}) \ \& \ M[T[i]]$
  - $R'_j = ((R_j >> 1) \ \& \ M[T[i]]) \mid R_{j-1} \mid (R_{j-1} >> 1) \mid (R'_{j-1} >> 1) \mid 10^{m-1}$, onde M é a tabela do algoritmo Shift-And para casamento exato.
- A pesquisa inicia com $R_j = 1^j 0_{m-j}$.
- R_0 equivale ao algoritmo Shift-And para casamento exato.
- As outras linhas R_j recebem 1s (estados ativos) também de linhas anteriores.
- Considerando um automato para casamento aproximado, a fórmula para R' expressa:
  - arestas horizontais indicando casamento;
  - verticais indicando inserção;
  - diagonais cheias indicando substituição;
  - diagonais tracejadas indicando retirada.

Shif-And para Casamento Aproximado - Implementação

void Shift-And-Aproximado ($P = p_0 p_1 \ldots p_{m-1},\ T = t_0 t_1 \ldots t_{n-1}, k$)

```
// Pré-processamento
for (c ∈ Σ) M[c] = 0^m;
for (j = 0; j < m; j++) M[p_j] = M[p_j] | 0^j 10^{m-j-1};
// Pesquisa
for (j = 0; j <= k; j++) R_j = 1^j 0^{m-j};
for (i = 0; i < n; i++)
  Rant = R_0;
  Rnovo = ((Rant >> 1) | 10^{m-1}) & M[T[i]];
  R_0 = Rnovo;
  for (j = 1; j <= k; j++)
    Rnovo = ((R_j >> 1 & M[T[i]]) | Rant | ((Rant | Rnovo) >> 1);
    Rant = R_j;
    R_j = Rnovo | 10^{m-1};
  if (Rnovo & 0^{m-1}1 ≠ 0^m) "Casamento na posicao i";
```

Shif-And p/ Casam. Aprox. - Exemplo

- Padrão: `teste`. Texto: `os testes testam`. Permitindo um erro ($k = 1$) de inserção).
- $R'_0 = (R_0 >> 1)|10^{m-1}\&M[T[i]]$ e
  $R'_1 = (R_1 >> 1)\&M[T[i]]|R_0|(10^{m-1})$
- Uma ocorrência exata na posição 8 (“e”) e duas, permitindo uma inserção, nas posições 8 e 11 (“s” e “e”, respectivamente).

Texto	$(R_0 >> 1)\|10^{m-1}$	R'_0	$R_1 >> 1$	R'_1
o	1 0 0 0 0	0 0 0 0 0	0 1 0 0 0	1 0 0 0 0
s	1 0 0 0 0	0 0 0 0 0	0 1 0 0 0	1 0 0 0 0
	1 0 0 0 0	0 0 0 0 0	0 1 0 0 0	1 0 0 0 0
t	1 0 0 0 0	1 0 0 0 0	0 1 0 0 0	1 0 0 0 0
e	1 1 0 0 0	0 1 0 0 0	0 1 0 0 0	1 1 0 0 0
s	1 0 1 0 0	0 0 1 0 0	0 1 1 0 0	1 1 1 0 0
t	1 0 0 1 0	1 0 0 1 0	0 1 1 1 0	1 0 1 1 0
e	1 1 0 0 1	0 1 0 0 ***1***	0 1 0 1 1	1 1 0 1 ***1***
s	1 0 1 0 0	0 0 1 0 0	0 1 1 0 1	1 1 1 0 ***1***
	1 0 0 0 0	0 0 0 0 0	0 1 1 1 0	1 0 1 0 0
t	1 0 0 0 0	1 0 0 0 0	0 1 0 1 0	1 0 0 1 0
e	1 1 0 0 0	0 1 0 0 0	0 1 0 0 1	1 1 0 0 ***1***
s	1 0 1 0 0	0 0 1 0 0	0 1 1 0 0	1 1 1 0 0
t	1 0 0 1 0	1 0 0 1 0	0 1 1 1 0	1 0 1 1 0
a	1 1 0 0 1	0 0 0 0 0	0 1 0 1 1	1 0 0 1 0
m	1 0 0 0 0	0 0 0 0 0	0 1 0 0 1	1 0 0 0 0

Shif-And p/ Casam. Aprox. - Exemplo

- Padrão: `teste`. Texto: `os testes testam`. Permitindo um erro de inserção, um de retirada e um de substituição.
- $R'_0 = (R_0 >> 1)|10^{m-1}\&M[T[i]]$ e $R'_1 = (R_1 >> 1)\&M[T[i]]|R_0|(R'_0 >> 1)|(R_0 >> 1)|(10^{m-1})$.
- Uma ocorrência exata na posição 8 (“e”) e cinco, permitindo um erro, nas posições 6, 8, 11, 13 e 14 (“`t`”, “s”, “e”, “`t`” e “a”, respec.).

Texto	$(R_0 >> 1)\|10^{m-1}$	R'_0	$R_1 >> 1$	R'_1
o	1 0 0 0 0	0 0 0 0 0	0 1 0 0 0	1 0 0 0 0
s	1 0 0 0 0	0 0 0 0 0	0 1 0 0 0	1 0 0 0 0
	1 0 0 0 0	0 0 0 0 0	0 1 0 0 0	1 0 0 0 0
t	1 0 0 0 0	1 0 0 0 0	0 1 0 0 0	1 1 0 0 0
e	1 1 0 0 0	0 1 0 0 0	0 1 1 0 0	1 1 1 0 0
s	1 0 1 0 0	0 0 1 0 0	0 1 1 1 0	1 1 1 1 0
t	1 0 0 1 0	1 0 0 1 0	0 1 1 1 1	1 1 1 1 ***1***
e	1 1 0 0 1	0 1 0 0 ***1***	0 1 1 1 1	1 1 1 1 ***1***
s	1 0 1 0 0	0 0 1 0 0	0 1 1 1 1	1 1 1 1 ***1***
	1 0 0 0 0	0 0 0 0 0	0 1 1 1 1	1 0 1 1 0
t	1 0 0 0 0	1 0 0 0 0	0 1 0 1 1	1 1 0 1 0
e	1 1 0 0 0	0 1 0 0 0	0 1 1 0 1	1 1 1 0 ***1***
s	1 0 1 0 0	0 0 1 0 0	0 1 1 1 0	1 1 1 1 0
t	1 0 0 1 0	1 0 0 1 0	0 1 1 1 1	1 1 1 1 ***1***
a	1 1 0 0 1	0 0 0 0 0	0 1 1 1 1	1 1 0 1 ***1***
m	1 0 0 0 0	0 0 0 0 0	0 1 1 0 1	1 0 0 0 0

Compressão - Motivação

- Explosão de informação textual disponível *on-line*:
  - Bibliotecas digitais.
  - Sistemas de automação de escritórios.
  - Bancos de dados de documentos.
  - World-Wide Web.
- Somente a Web tem hoje bilhões de páginas estáticas disponíveis.
- Cada bilhão de páginas ocupando aproximadamente 10 *terabytes* de texto corrido.
- Em setembro de 2003, a máquina de busca Google (www.google.com.br) dizia ter mais de 3,5 bilhões de páginas estáticas em seu banco de dados.

Características necessárias para sistemas de recuperação de informação

- Métodos recentes de compressão têm permitido:
  1. Pesquisar diretamente o texto comprimido mais rapidamente do que o texto original.
  2. Obter maior compressão em relação a métodos tradicionais, gerando maior economia de espaço.
  3. Acessar diretamente qualquer parte do texto comprimido sem necessidade de descomprimir todo o texto desde o início (Moura, Navarro, Ziviani e Baeza-Yates, 2000; Ziviani, Moura, Navarro e Baeza-Yates, 2000).
- Compromisso espaço X tempo:
  - vencer-vencer.

Porque Usar Compressão

- **Compressão de texto -** maneiras de representar o texto original em menos espaço:
  - Substituir os símbolos do texto por outros que possam ser representados usando um número menor de *bits* ou *bytes*.
- **Ganho obtido:** o texto comprimido ocupa menos espaço de armazenamento $\Rightarrow$ menos tempo para ser lido do disco ou ser transmitido por um canal de comunicação e para ser pesquisado.
- **Preço a pagar:** custo computacional para codificar e decodificar o texto.
- **Avanço da tecnologia:** De acordo com Patterson e Hennessy (1995), em 20 anos, o tempo de acesso a discos magnéticos tem se mantido praticamente constante, enquanto a velocidade de processamento aumentou aproximadamente 2 mil vezes $\Rightarrow$ melhor investir mais poder de computação em compressão em troca de menos espaço em disco ou menor tempo de transmissão.

Razão de Compressão

- Definida pela porcentagem que o arquivo comprimido representa em relação ao tamanho do arquivo não comprimido.
- **Exemplo:** se o arquivo não comprimido possui 100 *bytes* e o arquivo comprimido resultante possui 30 *bytes*, então a razão de compressão é de 30%.
- Utilizada para medir O ganho em espaço obtido por um método de compressão.

Outros Importantes Aspectos a Considerar

Além da economia de espaço, deve-se considerar:

- Velocidade de compressão e de descompressão.
- Possibilidade de realizar **casamento de cadeias** diretamente no texto comprimido.
- Permitir acesso direto a qualquer parte do texto comprimido e iniciar a descompressão a partir da parte acessada:

  Um sistema de recuperação de informação para grandes coleções de documentos que estejam comprimidos necessitam acesso direto a qualquer ponto do texto comprimido.

Compressão de Textos em Linguagem Natural

- Um dos métodos de codificação mais conhecidos é o de **Huffman** (1952):
  - A idéia do método é atribuir códigos mais curtos a símbolos com freqüências altas.
  - Um código único, de tamanho variável, é atribuído a cada símbolo diferente do texto.
  - As implementações tradicionais do método de Huffman consideram caracteres como símbolos.
- Para aliar as necessidades dos algoritmos de compressão às necessidades dos sistemas de recuperação de informação apontadas acima, deve-se considerar palavras como símbolos a serem codificados.
- Métodos de Huffman baseados em caracteres comprimem o texto para aproximadamente 60%.
- Métodos de Huffman baseados em palavras comprimem o texto para valores pouco acima de 25%.

Vantagens dos Métodos de Huffman Baseados em Palavras

- Permitem acesso randômico a palavras dentro do texto comprimido.
- Considerar palavras como símbolos significa que a tabela de símbolos do codificador é exatamente o vocabulário do texto.
- Isso permite uma integração natural entre o método de compressão e o arquivo invertido.
- Permitem acessar diretamente qualquer parte do texto comprimido sem necessidade de descomprimir todo o texto desde o início.

Família de Métodos de Compressão Ziv-Lempel

- Substitui uma seqüência de símbolos por um apontador para uma ocorrência anterior daquela seqüência.
- A compressão é obtida porque os apontadores ocupam menos espaço do que a seqüência de símbolos que eles substituem.
- Os métodos Ziv-Lempel são populares pela sua velocidade, economia de memória e generalidade.
- Já o método de Huffman baseado em palavras é muito bom quando a cadeia de caracteres constitui texto em linguagem natural.

Desvantagens dos Métodos de Ziv-Lempel para Ambiente de Recuperação de Informação

- É necessário iniciar a decodificação desde o início do arquivo comprimido $\Rightarrow$ Acesso randômico muito caro.
- É muito difícil pesquisar no arquivo comprimido sem descomprimir.
- Uma possível vantagem do método Ziv-Lempel é o fato de não ser necesário armazenar a tabela de símbolos da maneira com que o método de Huffman precisa.
- No entanto, isso tem pouca importância em um ambiente de recuperação de informação, já que se necessita o vocabulário do texto para criar o índice e permitir a pesquisa eficiente.

Compressão de Huffman Usando Palavras

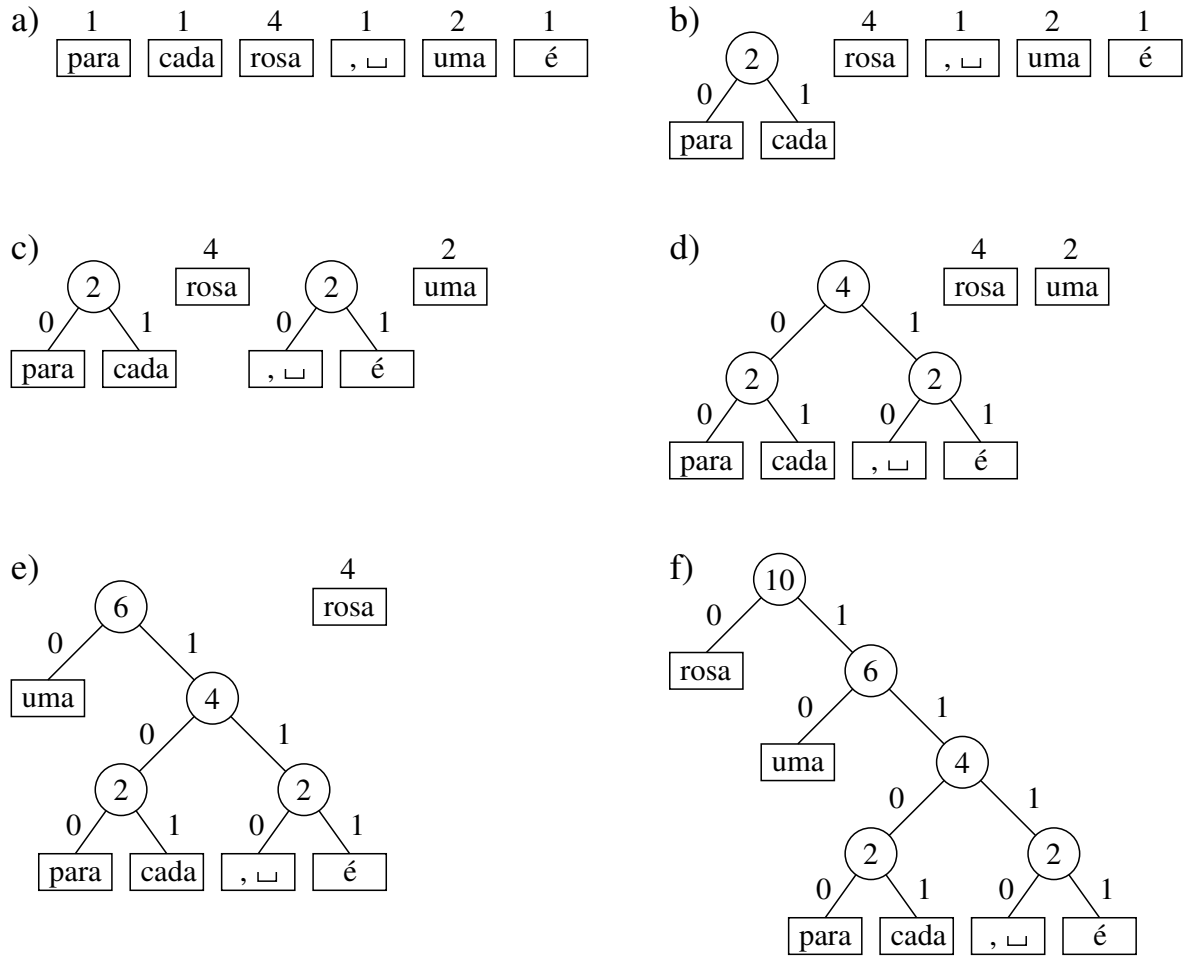
- Técnica de compressão mais eficaz para textos em linguagem natural.
- O método considera cada palavra diferente do texto como um símbolo.
- Conta suas freqüências e gera um código de Huffman para as palavras.
- A seguir, comprime o texto substituindo cada palavra pelo seu código.
- Assim, a compressão é realizada em duas passadas sobre o texto:
  1. Obtenção da freqüência de cada palavra diferente.
  2. Realização da compressão.

Forma Eficiente de Lidar com Palavras e Separadores

- Um texto em linguagem natural é constituído de palavras e de separadores.
- Separadores são caracteres que aparecem entre palavras: espaço, vírgula, ponto, ponto e vírgula, interrogação, e assim por diante.
- Uma forma eficiente de lidar com palavras e separadores é representar o espaço simples de forma implícita no texto comprimido.
- Nesse modelo, se uma palavra é seguida de um espaço, então, somente a palavra é codificada.
- Senão, a palavra e o separador são codificados separadamente.
- No momento da decodificação, supõe-se que um espaço simples segue cada palavra, a não ser que o próximo símbolo corresponda a um separador.

Compressão usando codificação de Huffman

Exemplo: "`para cada rosa rosa, uma rosa é uma rosa`"



OBS: O algoritmo de Huffman é uma abordagem **gulosa**.

Árvore de Huffman

- O método de Huffman produz a árvore de codificação que minimiza o comprimento do arquivo comprimido.
- Existem diversas árvores que produzem a mesma compressão.
- Por exemplo, trocar o filho à esquerda de um nó por um filho à direita leva a uma árvore de codificação alternativa com a mesma razão de compressão.
- Entretanto, a escolha preferencial para a maioria das aplicações é a **árvore canônica**.
- Uma árvore de Huffman é canônica quando a altura da subárvore à direita de qualquer nó nunca é menor que a altura da subárvore à esquerda.

Árvore de Huffman

- A representação do código na forma de árvore facilita a visualização.
- Sugere métodos de codificação e decodificação triviais:
  - **Codificação:** a árvore é percorrida emitindo *bits* ao longo de suas arestas.
  - **Decodificação:** os *bits* de entrada são usados para selecionar as arestas.
- Essa abordagem é ineficiente tanto em termos de espaço quanto em termos de tempo.

Algoritmo Baseado na Codificação Canônica com Comportamento Linear em Tempo e Espaço

- O algoritmo é atribuído a Moffat e Katajainen (1995).
- Calcula os comprimentos dos códigos em lugar dos códigos propriamente ditos.
- A compressão atingida é a mesma, independentemente dos códigos utilizados.
- Após o cálculo dos comprimentos, há uma forma elegante e eficiente para a codificação e a decodificação.

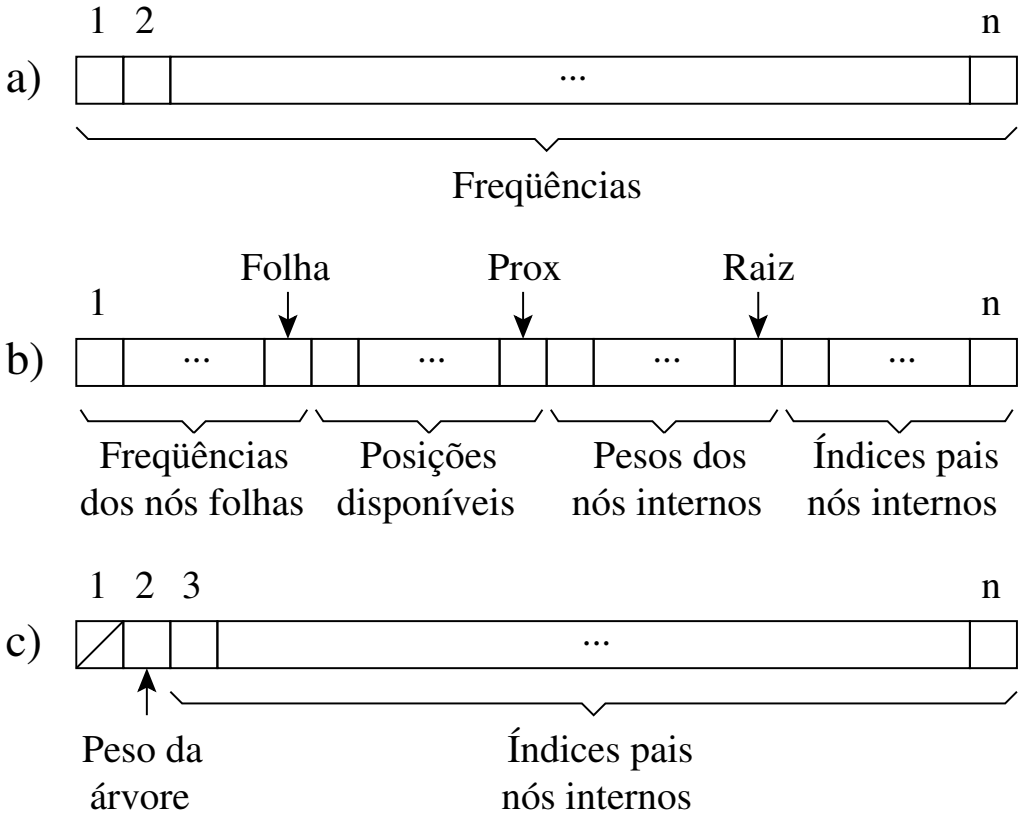
O Algoritmo

- A entrada do algoritmo é um vetor A contendo as freqüências das palavras em ordem não-crescente.
- Freqüências relativas à frase exemplo: “`para cada rosa rosa, uma rosa é uma rosa`”

4	2	1	1	1	1

- Durante sua execução, são utilizados diversos vetores logicamente distintos, mas capazes de coexistirem no mesmo vetor das freqüências.
- O algoritmo divide-se em três fases:
  1. Combinação dos nós.
  2. Conversão do vetor no conjunto das profundidades dos nós internos.
  3. Calculo das profundidades dos nós folhas.

Primeira Fase - Combinação dos nós



- A primeira fase é baseada em duas observações:
  1. A freqüência de um nó só precisa ser mantida até que ele seja processado.
  2. Não é preciso manter apontadores para os pais dos nós folhas, pois eles podem ser inferidos.

     Exemplo: nós internos nas profundidades $[0, 1, 2, 3, 3]$ teriam nós folhas nas profundidades $[1, 2, 4, 4, 4, 4]$.

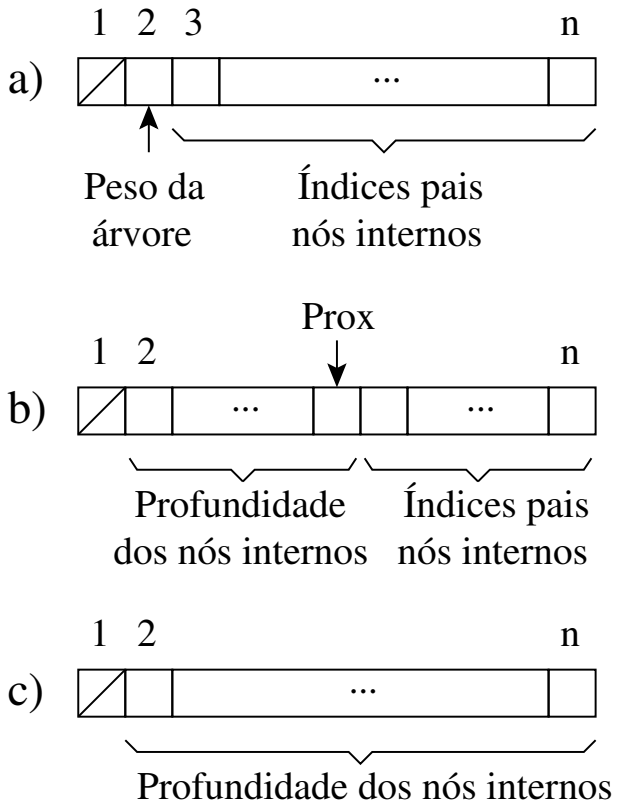
Pseudocódigo para a Primeira Fase

```
void primeiraFase (A, n) {
  raiz = n; folha = n;
  for (prox = n; prox >= 2; prox--){
    // Procura Posição
    if ((não existe folha) || ((raiz > prox) && (A[raiz] <= A[folha]))) {
      // Nó interno
      A[prox] = A[raiz]; A[raiz] = prox; raiz--;
    }
    else { // Nó folha
        A[prox] = A[folha]; folha--;
    }
    // Atualiza Freqüências
    if ((não existe folha) || ((raiz > prox) && (A[raiz] <= A[folha]))) {
      // Nó interno
      A[prox] = A[prox] + A[raiz]; A[raiz] = prox; raiz--;
    }
    else { // Nó folha
        A[prox] = A[prox] + A[folha]; folha--;
    }
  }
}
```

Exemplo de processamento da primeira fase

	1	2	3	4	5	6	Prox	Raiz	Folha
a)	4	2	1	1	1	1	6	6	6
b)	4	2	1	1	1	1	6	6	5
c)	4	2	1	1	1	2	5	6	4
d)	4	2	1	1	1	2	5	6	3
e)	4	2	1	1	2	2	4	6	2
f)	4	2	1	2	2	4	4	5	2
g)	4	2	1	4	4	4	3	4	2
h)	4	2	2	4	4	4	3	4	1
i)	4	2	6	3	4	4	2	3	1
j)	4	4	6	3	4	4	2	3	0
k)	⁄	10	2	3	4	4	1	2	0

Segunda Fase - Conversão do vetor no conjunto das profundidades dos nós internos



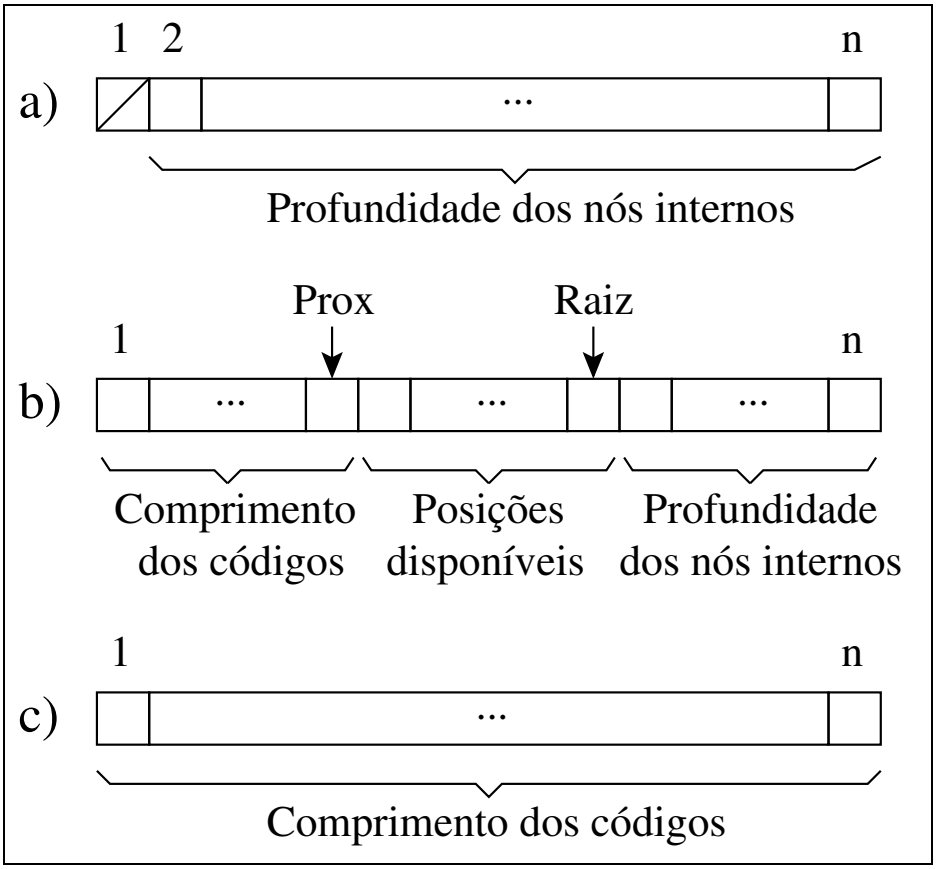
Pseudocódigo para a Segunda Fase

```
void segundaFase (A, n) {
  A[2] = 0;
  for (prox = 3; prox <= n; prox++) A[prox] = A[A[prox]] + 1;
}
```

Profundidades dos nós internos obtida com a segunda fase tendo como entrada o vetor exibido na letra k) da transparência 65:

	0	1	2	3	3

Terceira Fase - Calculo das profundidades dos nós folhas



Pseudocódigo para a Terceira Fase

```
void terceiraFase (A, n) {
  disp = 1; u = 0; h = 0; raiz = 2; prox = 1;
  while (disp > 0) {
    while ((raiz <= n) && (A[raiz] == h)) { u++; raiz++; }
    while (disp > u) { A[prox] = h; prox++; disp--;}
    disp = 2 * u; h++; u = 0;
  }
}
```

- Aplicando-se a Terceira Fase sobre:

	0	1	2	3	3

Os comprimentos dos códigos em número de *bits* são obtidos:

1	2	4	4	4	4

Cálculo do comprimento dos códigos a partir de um vertor de freqüências

```
void calculaCompCodigo (A, n) {
   primeiraFase (A, n);
   segundaFase (A, n);
   terceiraFase (A, n);
}
```

Código Canônico

- Propriedades:
  1. Os comprimentos dos códigos obedecem ao algoritmo de Huffman.
  2. Códigos de mesmo comprimento são inteiros consecutivos.
- A partir dos comprimentos obtidos, o cálculo dos códigos propriamente dito é trivial: o primeiro código é composto apenas por zeros e, para os demais, adiciona-se 1 ao código anterior e faz-se um deslocamento à esquerda para obter-se o comprimento adequado quando necessário.
- **Codificação Canônica Obtida:**

i	Símbolo	Código Canônico
1	`rosa`	`0`
2	`uma`	`10`
3	`para`	`1100`
4	`cada`	`1101`
5	`,␣`	`1110`
6	`é`	`1111`

Elaboração de Algoritmos Eficientes para a Codificação e para a Decodificação

- Os algoritmos são baseados na seguinte observação:
  - Códigos de mesmo comprimento são inteiros consecutivos.
- Os algoritmos são baseados no uso de dois vetores com $maxCompCod$ elementos,sendo $maxCompCod$ o comprimento do maior código.

Vetores $base$ e $offset$

- **Vetor $base$:** indica, para um dado comprimento c, o valor inteiro do primeiro código com esse comprimento.
- O vetor Base é calculado pela relação:

$$\text{base}[c] = \begin{cases} 0 & \text{se } c = 1, \\ 2 \times (\text{base}[c-1] + w_{c-1}) & \text{caso contrário,} \end{cases}$$

  sendo w_c o número de códigos com comprimento c.

- $offset$ indica o índice no vocabulário da primeira palavra de cada comprimento de código c.
- Vetores $base$ e $offset$ para a tabela da transparência 71:

c	$base[c]$	$offset[c]$
1	0	1
2	2	2
3	6	2
4	12	3

Pseudocódigo para Codificação

```
Codigo codifica (i, maxCompCod) {
  c = 1;
  while ((c + 1 <= maxCompCod) && (i >= offset[c + 1])) c++;
  codigo = i − offset[c] + base[c];
  return (codigo, c);
}
```

Obtenção do código:

- O método de codificação recebe como parâmetros o índice i do símbolo a ser codificado e o comprimento $maxCompCod$ dos vetores $base$ e $offset$.
- No anel **while** é feito o cálculo do comprimento c de código a ser utilizado.
- A seguir, basta saber qual a ordem do código para o comprimento c ($i - offset[c]$) e somar esse valor à $base[c]$.

Exemplo de Codificação

- Para a palavra $i = 4$ (“`cada`”):
  1. Verifica-se que é um código de comprimento 4.
  2. Verifica-se também que é o segundo código com esse comprimento.
  3. Assim, seu código é 13 ($4 - \mathit{offset}[4] + \mathit{base}[4]$), o que corresponde a “`1101`” em binário.

Pseudocódigo para Decodificação

```
int decodifica (maxCompCod) {
  c = 1; codigo = leBit (arqComp);
  while ((codigo << 1 ) >= base[c + 1] && ( c + 1 <= maxCompCod )) {
    codigo = (codigo << 1) | leBit (arqComp); c++;
  }
  i = codigo – base[c] + offset[c];
  return i;
}
```

- O programa recebe como parâmetro o comprimento $maxCompCod$ dos vetores $base$ e $offset$.
- Na decodificação, o arquivo de entrada é lido *bit*-a-*bit*, adicionando-se os *bits* lidos ao código e comparando-o com o vetor Base.
- O anel **while** mostra como identificar o código a partir de uma posição do arquivo comprimido.

Exemplo de Decodificação

- Decodificação da seqüência de *bits* “1101”:

c	LeBit	Codigo	Codigo $<< 1$	Base[$c+1$]
1	1	1	-	-
2	1	10 **or** 1 $=$ 11	10	10
3	0	110 **or** 0 $=$ 110	110	110
4	1	1100 **or** 1 $=$ 1101	1100	1100

- A primeira linha da tabela representa o estado inicial do anel **while** quando já foi lido o primeiro *bit* da seqüência, o qual foi atribuído à variável $codigo$.
- A linha dois e seguintes representam a situação do anel **while** após cada respectiva iteração.
- No caso da linha dois da tabela, o segundo *bit* da seqüência foi lido (*bit* “1”) e a variável $codigo$ recebe o código anterior deslocado à esquerda de um *bit* seguido da operação *or* com o *bit* lido.
- De posse do código, $base$ e $offset$ são usados para identificar qual o índice i da palavra no vocabulário, sendo
  $i = codigo - base[c] + offset[c]$.

Pseudocódigo para realizar a compressão

```
void compressao (nomeArqTxt, nomeArqComp) {
  arqComp = new RandomAccessFile (nomeArqComp, "rws");
  arqTxt = new BufferedReader (new FileReader(nomeArqTxt));
  // Primeira etapa
  String palavra = null; TabelaHash vocabulario;
  while (existirem palavras) {
    palavra = proximaPalavra (arqTxt);
    itemVoc = vocabulario.pesquisa (palavra);
    if (itemVoc != null) itemVoc.freq = itemVoc.freq + 1;
    else vocabulario.insere (palavra);
  }
  // Segunda etapa
  A[] = ordenaPorFrequencia (vocabulario); calculaCompCodigo (A, n);
  maxCompCod = constroiVetores (A, n); gravaVocabulario (A, arqComp);
  // Terceira etapa
  while (existirem palavras) {
    palavra = proximaPalavra (arqTxt);
    itemVoc = vocabulario.pesquisa (palavra);
    codigo = codifica(itemVoc.ordem, maxCompCod);
    escreve (codigo, maxCompCod);
  }
}
```

Pseudocódigo para realizar a descompressão

```
void descompressao (nomeArqTxt, nomeArqComp) {
  arqComp = new RandomAccessFile (nomeArqComp, "rws");
  arqTxt = new BufferedWriter (new FileWriter (nomeArqTxt));
  int maxCompCod = leVetores ();
  String vocabulario[] = leVocabulario ();
  while ((i = decodifica (maxCompCod)) >= 0) {
    if ((palavra anterior não é delimitador) && (vocabulario[i] não é delimitador))
      arqTxt.write (" ");
    arqTxt.write (vocabulario[i]);
  }
}
```

Codificação de Huffman Usando Palavras - Análise

- A representação do código de Huffman na forma de uma árvore é ineficiente em termos de espaço e de tempo (não é usado na prática).
- **Codificação canônica**: forma mais eficiente baseada nos comprimentos dos códigos em vez dos códigos propriamente ditos (Moffat e Katajainen - 1995).
- Feito *in situ* a partir de um vetor A contendo as freqüências das palavras em ordem não crescente a um custo $O(n)$ em tempo e em espaço.
- O algoritmo requer apenas os dois vetores $base$ e $offset$ de tamanho $maxCompCod$, sendo $maxCompCod$ o comprimento do maior código.
- A decodificação é também muito eficiente pois apenas os vetores $base$ e $offset$ são consultados.
- Não há necessidade de realizar a decodificação *bit* a *bit*, como na árvore de Huffman.

Codificação de Huffman Usando Bytes

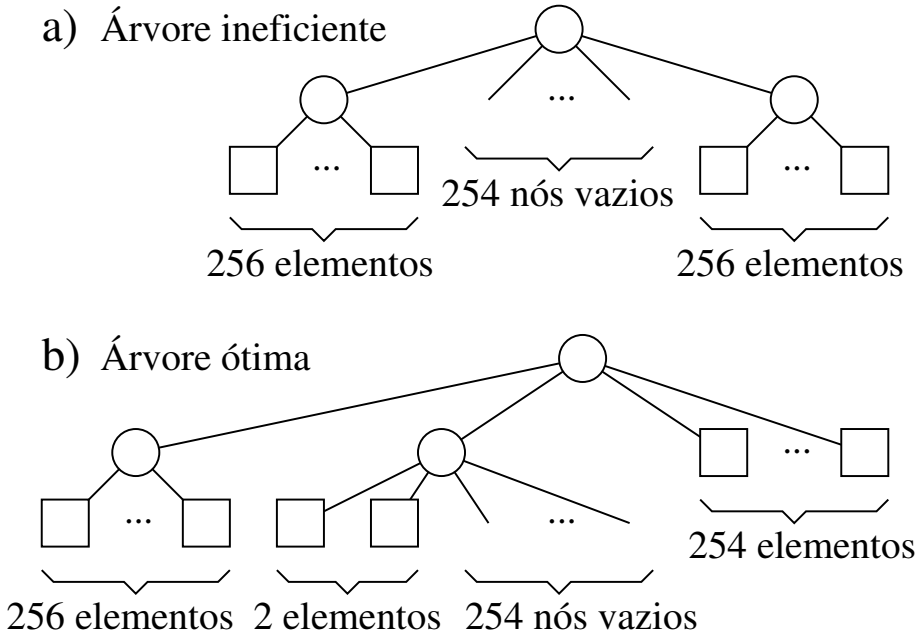
- O método original proposto por Huffman (1952) tem sido usado como um código binário.
- Moura, Navarro, Ziviani e Baeza-Yates (2000) modificaram a atribuição de códigos de tal forma que uma seqüência de *bytes* é associada a cada palavra do texto.
- Conseqüentemente, o grau de cada nó passa de 2 para 256. Essa versão é chamada de *código de Huffman pleno*.
- Outra possibilidade é utilizar apenas 7 dos 8 *bits* de cada *byte* para a codificação, e a árvore passa então a ter grau 128.
- Nesse caso, o oitavo *bit* é usado para marcar o primeiro *byte* do código da palavra, sendo chamado de *código de Huffman com marcação*.

Exemplo de Códigos Plenos e com Marcação

- O código de Huffman com marcação ajuda na pesquisa sobre o texto comprimido.
- **Exemplo:**
  - Código pleno para a palavra “uma” com 3 *bytes* “47 81 8”.
  - Código com marcação para a palavra “uma” com 3 *bytes* “175 81 8”
  - Note que o primeiro *byte* é $175 = 47 + 128$.
- Assim, no código com marcação o oitavo *bit* é 1 quando o *byte* é o primeiro do código, senão ele é 0.

Árvore de Huffman orientada a *bytes*

- A construção da árvore de Huffman orientada a *bytes* pode ocasionar o aparecimento de nós internos não totalmente preenchidos quando a árvore não é binária:



- Na Figura, o alfabeto possui 512 símbolos (nós folhas), todos com a mesma freqüência de ocorrência.
- O segundo nível tem 254 espaços vazios que poderiam ser ocupados com símbolos, mudando o comprimento de seus códigos de 2 para 1 *byte*.

Movendo Nós Vazios para Níveis mais Profundos

- Um meio de assegurar que nós vazios sempre ocupem o nível mais baixo da árvore é combiná-los com os nós de menores freqüências.
- O objetivo é movê-los para o nível mais profundo da árvore.
- Para isso, devemos selecionar o número de símbolos que serão combinados com os nós vazios.
- Essa seleção é dada pela equação $1 + ((n - \text{baseNum}) \mod (\text{baseNum} - 1))$.
- No caso da Figura da transparência anterior é igual a $1 + ((512 - 256) \mod 255) = 2$.

Classe *HuffmanByte*

```
package cap8;
import java.io.*;
import cap5.endaberto.TabelaHash;
import cap4.ordenacaointerna.Ordenacao;
public class HuffmanByte {
  private int baseNum;
  private int base[], offset[];
  private RandomAccessFile arqComp; // Arquivo comprimido
  private String nomeArqTxt; // Nome do arquivo texto a ser comprimido
  private String nomeArqDelim; // Nome do arquivo que contém os delimitadores
  private TabelaHash vocabulario;
  private static class Codigo {
    int codigo;  int c; // Comprimento do código
  }
  public HuffmanByte (String nomeArqDelim, int baseNum, int m,
                      int maxTamChave) throws Exception {
    this.baseNum = baseNum; this.base = null; this.offset = null;
    this.nomeArqTxt = null; this.nomeArqDelim = nomeArqDelim;
    this.vocabulario = new TabelaHash (m, maxTamChave);
  }
  public void compressao (String nomeArqTxt,
                          String nomeArqComp) throws Exception {
    public void descompressao (String nomeArqTxt,
                               String nomeArqComp) throws Exception {
  }
```

Generalização do Cálculo dos Comprimentos dos Códigos

```
private void calculaCompCodigo (ItemVoc[] A, int n) {
  int resto = 0;
  if (n > (this.baseNum – 1)) {
    resto = 1 + ((n – this.baseNum) % (this.baseNum – 1));
    if (resto < 2) resto = this.baseNum;
  }
  else resto = n – 1;
  // noInt: Número de nós internos
  int noInt = 1 + ((n – resto) / (this.baseNum – 1));
  int freqn = ((Integer)A[n].recuperaChave ()).intValue ();
  for (int x = (n – 1); x >= (n – resto + 1); x––){
    int freqx = ((Integer)A[x].recuperaChave ()).intValue ();
    freqn = freqn + freqx;
  }
  A[n].alteraChave (new Integer (freqn));
  // Primeira Fase
  int raiz = n; int folha = n – resto; int prox;
  for (prox = n – 1; prox >= (n – noInt + 1); prox––){
    // Procura Posição
    int freqraiz = ((Integer)A[raiz].recuperaChave ()).intValue ();
    if ((folha < 1) || ((raiz > prox) &&
        (freqraiz <= ((Integer)A[folha].recuperaChave()).intValue()))) {
      // Nó interno
      A[prox].alteraChave (new Integer (freqraiz));
      A[raiz].alteraChave (new Integer (prox)); raiz––;
    }
    else { // Nó folha
      int freqfolha = ((Integer)A[folha].recuperaChave ()).intValue ();
      A[prox].alteraChave (new Integer (freqfolha)); folha––;
    }
```

Generalização do Cálculo dos Comprimentos dos Códigos

```
    // Atualiza Freqüências
    for (int x = 1; x <= (this.baseNum – 1); x++) {
      freqraiz = ((Integer)A[raiz].recuperaChave ()).intValue ();
      int freqprox = ((Integer)A[prox].recuperaChave ()).intValue ();
      if ((folha < 1) || ((raiz > prox) &&
          (freqraiz<=((Integer)A[folha].recuperaChave()).intValue()))) {
        // Nó interno
        A[prox].alteraChave (new Integer (freqprox + freqraiz));
        A[raiz].alteraChave (new Integer (prox)); raiz—;
      }
      else { // Nó folha
        int freqfolha = ((Integer)A[folha].recuperaChave ()).intValue ();
        A[prox].alteraChave(new Integer(freqprox + freqfolha)); folha—;
      }
    }
  }
  // Segunda Fase
  A[raiz].alteraChave (new Integer (0));
  for (prox = raiz + 1; prox <= n; prox++) {
    int pai = ((Integer)A[prox].recuperaChave ()).intValue ();
    int profundidadepai = ((Integer)A[pai].recuperaChave ()).intValue ();
    A[prox].alteraChave (new Integer (profundidadepai + 1));
  }
```

Generalização do Cálculo dos Comprimentos dos Códigos

```
    // Terceira Fase
    int disp = 1; int u = 0; int h = 0; prox = 1;
    while (disp > 0) {
      while ((raiz <= n) &&
        (((Integer)A[raiz].recuperaChave()).intValue() == h)) {u++; raiz++;}
      while (disp > u) {
        A[prox].alteraChave (new Integer (h)); prox++; disp--;
        if (prox > n) { u = 0; break; }
      }
      disp = this.baseNum * u; h = h + 1; u = 0;
    }
}
```

OBS: *baseNum* pode ser usada para trabalharmos com quaisquer bases numéricas menores ou iguais a um *byte*. Por exemplo, para a codificação plena o valor é 256 e para a codificação com marcação o valor é 128.

Mudanças em Relação ao Pseudocódigo Apresentado

- A mudança maior está no código inserido antes da primeira fase para eliminar o problema causado por nós internos da árvore não totalmente preenchidos.
- Na primeira fase, as *baseNum* árvores de menor custo são combinadas a cada passo, em vez de duas como no caso da codificação binária:
  - Isso é feito pelo anel **for** introduzido na parte que atualiza freqüências na primeira fase.
- A segunda fase não sofre alterações.
- A terceira fase recebe a variável $disp$ para indicar quantos nós estão disponíveis em cada nível.

Codificação orientada a *bytes*

```
private Codigo codifica (int ordem, int maxCompCod) {
  Codigo cod = new Codigo (); cod.c = 1;
  while ((cod.c + 1 <= maxCompCod) && (ordem >= this.offset[cod.c + 1]))
    cod.c++;
  cod.codigo = ordem – this.offset[cod.c] + this.base[cod.c];
  return cod;
}
```

OBS: a codificação orientada a *bytes* não requer nenhuma alteração em relação à codificação usando *bits*

Decodificação orientada a *bytes*

```
private int decodifica (int maxCompCod) throws Exception {
  int logBase2 = (int)(Math.log(this.baseNum)/Math.log(2));
  int c = 1; int codigo = this.arqComp.read ();
  if (codigo < 0) return codigo; // Fim de arquivo
  if (logBase2 == 7) codigo = codigo – 128; // Remove o bit de marca-
cao
  while (((c + 1) <= maxCompCod) &&
         ((codigo << logBase2) >= this.base[c+1])) {
    int codigoTmp = this.arqComp.read ();
    codigo = (codigo << logBase2) | codigoTmp; c++; }
  return (codigo – this.base[c] + this.offset[c]);
}
```

Alterações:

1. Permitir a leitura *byte* a *byte* do arquivo comprimido, em vez de *bit* a *bit*. em relação ao número de *bits* que devem ser deslocados à esquerda para se encontrar o comprimento c do código, o qual indexa os vetores $base$ e $offset$.

2. O número de *bits* que devem ser deslocados à esquerda para se encontrar o comprimento c, o qualindexa os vetores $base$ e $offset$, é dado por: $\log_2 \mathrm{BaseNum}$

Cálculo dos Vetores $base$ e $offset$

- O cálculo do vetor $offset$ não requer alteração alguma.
- Para generalizar o cálculo do vetor $base$, basta substituir o fator 2 por $baseNum$, como abaixo:

$$base[c] = \begin{cases} 0 & \text{se } c = 1, \\ baseNum \times (base[c-1] + w_{c-1}) & \text{caso contrário.} \end{cases}$$

Construção dos Vetores $base$ e $offset$

```
private int constroiVetores (ItemVoc A[], int n) throws Exception {
  int maxCompCod = ((Integer)A[n].recuperaChave()).intValue();
  int wcs[] = new int[maxCompCod + 1]; // Ignora a posição 0
  this.offset = new int[maxCompCod + 1]; // Ignora a posição 0
  this.base = new int[maxCompCod + 1]; // Ignora a posição 0
  for (int i = 1; i <= maxCompCod; i++) wcs[i] = 0;
  for (int i = 1; i <= n; i++) {
    int freq = ((Integer)A[i].recuperaChave()).intValue();
    wcs[freq]++; this.offset[freq] = i - wcs[freq] + 1;
  }
  this.base[1] = 0;
  for (int i = 2; i <= maxCompCod; i++) {
    this.base[i] = this.baseNum * (this.base[i-1] + wcs[i-1]);
    if (this.offset[i] == 0) this.offset[i] = this.offset[i-1];
  }
  // Salvando as tabelas em disco
  this.arqComp.writeInt (maxCompCod);
  for (int i = 1; i <= maxCompCod; i++) {
    this.arqComp.writeInt (this.base[i]);
    this.arqComp.writeInt (this.offset[i]); }
  return maxCompCod;
}
```

Extração do próximo símbolo a ser codificado

```
package cap8;
import java.util.StringTokenizer;
import java.io.*;
public class ExtraiPalavra {
  private BufferedReader arqDelim, arqTxt;
  private StringTokenizer palavras;
  private String delimitadores, palavraAnt, palavra;
  private boolean eDelimitador (char ch) {
    return (this.delimitadores.indexOf (ch) >= 0);
  }
  public ExtraiPalavra (String nomeArqDelim, String nomeArqTxt)
                              throws Exception {
    this.arqDelim = new BufferedReader (new FileReader (nomeArqDelim));
    this.arqTxt = new BufferedReader (new FileReader (nomeArqTxt));
    // Os delimitadores devem estar juntos em uma única linha do arquivo
    this.delimitadores = arqDelim.readLine() + "\r\n";
    this.palavras = null; this.palavra = null; this.palavraAnt = " ";
  }
  public String proximaPalavra () throws Exception{
    String palavraTemp = ""; String resultado = "";
    if (this.palavra != null) {
      palavraTemp = palavra; palavra = null;
      palavraAnt = palavraTemp; return palavraTemp;
    }
    if (palavras == null || !palavras.hasMoreTokens ()) {
      String linha = arqTxt.readLine();
      if (linha == null) return null;
      linha += "\n";
      this.palavras=new StringTokenizer (linha,this.delimitadores,true);
    }
```

Extração do próximo símbolo a ser codificado

```
            String aux = this.palavras.nextToken();
        while (eDelimitador (aux.charAt (0)) && palavras.hasMoreTokens ()) {
          palavraTemp += aux; aux = this.palavras.nextToken();
        }
        if (palavraTemp.length () == 0) resultado = aux;
        else {
          this.palavra = aux;
          if (palavraTemp.length () == 1 && palavraTemp.equals(" ") &&
              palavraAnt.length () > 0 && palavra.length () > 0 &&
              !eDelimitador (palavraAnt.charAt (0)) &&
              !eDelimitador (palavra.charAt (0)))
            palavraTemp = palavraTemp.trim ();
          resultado = palavraTemp;
        } this.palavraAnt = resultado; return resultado;
      }
      public void fecharArquivos () throws Exception {
        this.arqDelim.close(); this.arqTxt.close();
      }
}
```

Classe para representar as informações de uma entrada do vocabulário

```
package cap8;
import cap4.Item; // vide Programa ??
public class ItemVoc implements Item {
  private String palavra;
  private int freq, ordem;
  // outros componentes do registro
  public ItemVoc (String palavra, int freq, int ordem) {
    this.palavra = palavra;
    this.freq = freq; this.ordem = ordem;
  }
  public int compara (Item it) {
    ItemVoc item = (ItemVoc) it;
    if (this.freq < item.freq) return 1;
    else if (this.freq > item.freq) return -1;
    return 0;
  }
  public void alteraChave (Object freq) {
    Integer ch = (Integer) freq; this.freq = ch.intValue ();
  }
  public Object recuperaChave () { return new Integer (this.freq); }
  public void alteraOrdem (int ordem) { this.ordem = ordem; }
  public int recuperaOrdem () { return this.ordem; }
  public String palavra () { return this.palavra; }
}
```

Código para Fazer a Compressão

- O Código para fazer a compressão é dividio em três etapas:
  1. Na primeira, as palavras são extraídas do texto a ser comprimido e suas respectivas freqüências são contabilizadas.
  2. Na segunda, são gerados os vetores $base$ e $offset$, os quais são gravados no arquivo comprimido seguidamente do vocabulário. Para delimitar os símbolos do vocabulário no disco, cada um deles é separado pelo caractere zero.
  3. Na terceira, o arquivo texto é percorrido pela segunda vez, sendo seus símbolos novamente extraídos, codificados e gravados no arquivo comprimido.

Código para Fazer a Compressão

```
public void compressao (String nomeArqTxt,
                        String nomeArqComp) throws Exception {
  this.nomeArqTxt = nomeArqTxt;
  this.arqComp = new RandomAccessFile (nomeArqComp, "rws");
  this.primeiraEtapa ();
  int maxCompCod = this.segundaEtapa ();
  this.terceiraEtapa (maxCompCod);
  this.arqComp.close ();
}
```

Primeira etapa da compressão

```
private void primeiraEtapa ( ) throws Exception {
  ExtraiPalavra palavras = new ExtraiPalavra (nomeArqDelim, nomeArqTxt);
  String palavra = null;
  while ((palavra = palavras.proximaPalavra()) != null) {
    // O primeiro espaço depois da palavra não é codificado
    if (palavra.equals ("")) continue;
    ItemVoc itemVoc = (ItemVoc) this.vocabulario.pesquisa (palavra);
    if ( itemVoc != null) { // Incrementa freqüência
      int freq = ((Integer)itemVoc.recuperaChave ()).intValue ();
      itemVoc.alteraChave (new Integer (freq + 1));
    } else { // Insere palavra com freqüência 1
      itemVoc = new ItemVoc (palavra, 1, 0);
      this.vocabulario.insere (palavra, itemVoc);
    }
  }
  palavras.fecharArquivos();
}
```

Segunda etapa da compressão

```
private int segundaEtapa () throws Exception {
  ItemVoc A[] = this.ordenaPorFrequencia ();
  int n = A.length – 1;
  this.calculaCompCodigo (A, n);
  int maxCompCod = this.constroiVetores (A, n);
  // Grava Vocabulário
  this.arqComp.writeInt (n);
  for (int i = 1; i <= n; i++) {
    this.arqComp.writeChars (A[i].palavra ());
    this.arqComp.writeChar ('\0');
    A[i].alteraOrdem (i);
  }
  return maxCompCod;
}
```

Método para ordenar o vocabulário por freqüência

- O objetivo desse método é recuperar as entradas do vocabulário, armazená-las contigüamente em um vetor e ordenar o vetor obtido na ordem não crescente pela freqüência das palavras no texto.
- Para isso, foi criado o operador $recuperaItens$, o qual retorna nas posições de 0 a $n-1$ do vetor $itens$ as n referências às entradas do vocabulário, as quais são objetos do tipo $ItemVoc$.
- Recuperados os itens, o método $ordenaPorFrequencia$ copia as referências aos objetos do tipo $ItemVoc$ que representam as entradas do vocabulário do vetor aux para as posições de 1 a n do vetor A.
- Por fim, na classe $ItemVoc$ o vetor A é ordenado de forma não crescente por suas respectivas freqüências de ocorrência ($Quicksort$).
- O método $ordenaPorFrequencia$ retorna o vetor ordenado.

Método para ordenar o vocabulário por freqüência

```
private ItemVoc[] ordenaPorFrequencia () {
  Object aux[] = this.vocabulario.recuperaItens ();
  ItemVoc A[] = new ItemVoc[aux.length+1]; // Ignora a posição 0
  for (int i = 0; i < aux.length; i++) A[i+1] = (ItemVoc)aux[i];
  Ordenacao.quicksort (A, aux.length);
  return A;
}
```

Operador para recuperar os objetos contidos em uma tabela *hash*

```
public Object[] recuperaItens () {
  int n = 0;
  for (int i = 0; i < this.M; i++)
    if (this.tabela[i] != null && !this.tabela[i].retirado) n++;
  Object itens[] = new Object[n]; n = 0;
  for (int i = 0; i < this.M; i++)
    if (this.tabela[i] != null && !this.tabela[i].retirado)
      itens[n++] = this.tabela[i].item;
  return itens;
}
```

Terceira etapa da compressão

```
private void terceiraEtapa (int maxCompCod) throws Exception {
  ExtraiPalavra palavras = new ExtraiPalavra (nomeArqDelim, nomeArqTxt);
  String palavra = null;
  while ((palavra = palavras.proximaPalavra()) != null) {
    // O primeiro espaço depois da palavra não é codificado
    if (palavra.equals ("")) continue;
    ItemVoc itemVoc = (ItemVoc) this.vocabulario.pesquisa (palavra);
    int ordem = itemVoc.recuperaOrdem ();
    Codigo cod = this.codifica (ordem, maxCompCod);
    this.escreve (cod, maxCompCod);
  }
  palavras.fecharArquivos();
}
```

Método $escreve$

- O método $escreve$ recebe o código e seu comprimento c em um objeto do tipo $Codigo$.
- O código é representado por um inteiro, o que limita seu comprimento a, no máximo, 4 *bytes* em um compilador que usa 4 *bytes* para representar inteiros.
- Primeiramente, o método $escreve$ extrai o primeiro *byte* e, caso o código de Huffman utilizado seja o de marcação (baseNum = 128), coloca a marcação no oitavo *bit*, fazendo uma operação *or* do *byte* com a constante 128.
- Esse *byte* é então colocado na primeira posição do vetor $saida$.
- No anel **w**hile, caso o comprimento c do código seja maior do que um, os demais *bytes* são extraídos e armazenados em $saida[i]$, em que $2 \leq i \leq c$.
- Por fim, o vetor de *bytes* $saida$ é gravado em disco no anel **f**or.

Implementação do Método *escreve*

```
private void escreve (Codigo cod, int maxCompCod) throws Exception {
  int saida[] = new int[maxCompCod + 1]; // Ignora a posição 0
  int logBase2 = (int)(Math.log(this.baseNum)/Math.log(2));
  int mask = (int)Math.pow (2, logBase2) – 1;
  int i = 1; int cTmp = cod.c;
  saida[i] = cod.codigo >> (logBase2*(cod.c – 1));
  if (logBase2 == 7) saida[i] = saida[i] | 128; // Marcação
  i++; cod.c––;
  while (cod.c > 0) {
    saida[i] = (cod.codigo >> (logBase2*(cod.c – 1))) & mask;
    i++; cod.c––;
  }
  for (i = 1; i <= cTmp; i++) this.arqComp.writeByte (saida[i]);
}
```

Descrição do Código para Fazer a Descompressão

- O primeiro passo é recuperar o modelo usado na compressão. Para isso, lê o alfabeto, o vetor $base$, o vetor $offset$ e o vetor $vocabulario$.
- Em seguida, inicia a decodificação, tomando o cuidado de adicionar um espaço em branco entre dois símbolos que sejam palavras.
- O processo de decodificação termina quando o arquivo comprimido é totalmente percorrido.

Código para Fazer a Descompressão

```
public void descompressao (String nomeArqTxt,
                           String nomeArqComp) throws Exception {
  this.nomeArqTxt = nomeArqTxt;
  this.arqComp = new RandomAccessFile (nomeArqComp, "rws");
  BufferedReader arqDelim = new BufferedReader (
                            new FileReader (this.nomeArqDelim));
  BufferedWriter arqTxt = new BufferedWriter (
                          new FileWriter (this.nomeArqTxt));
  String delim = arqDelim.readLine() + "\r\n";
  int maxCompCod = this.leVetores ();
  String vocabulario[] = this.leVocabulario ();
  int ind = 0; String palavraAnt = " ";
  while ((ind = this.decodifica (maxCompCod)) >= 0) {
    if (!eDelimitador (delim, palavraAnt.charAt(0)) &&
        !eDelimitador (delim, vocabulario[ind].charAt(0)))
      arqTxt.write (" ");
    arqTxt.write (vocabulario[ind]);
    palavraAnt = vocabulario [ind];
  }
  arqTxt.close ();
}
```

OBS: Observe que na descompressão, o vocabulário é representado por um vetor de símbolos do tipo $String$.

Métodos auxiliares da descompressão

```
private int leVetores () throws Exception {
  int maxCompCod = this.arqComp.readInt ();
  this.offset = new int[maxCompCod + 1]; // Ignora a posição 0
  this.base = new int[maxCompCod + 1]; // Ignora a posição 0
  for (int i = 1; i <= maxCompCod; i++) {
    this.base[i] = this.arqComp.readInt ();
    this.offset[i] = this.arqComp.readInt ();
  }
  return maxCompCod;
}
private String[] leVocabulario () throws Exception{
  int n = this.arqComp.readInt ();
  String vocabulario[] = new String[n+1]; // Ignora a posição 0
  for (int i = 1; i <= n; i++) {
    vocabulario[i] = ""; char ch;
    while ((ch = this.arqComp.readChar ()) != '\0') {
      vocabulario[i] += ch;
    }
  }
  return vocabulario;
}
private boolean eDelimitador (String delim, char ch) {
  return (delim.indexOf (ch) >= 0);
}
```

Resultados Experimentais

- Mostram que não existe grande degradação na razão de compressão na utilização de *bytes* em vez de *bits* na codificação das palavras de um vocabulário.
- Por outro lado, tanto a descompressão quanto a pesquisa são muito mais rápidas com uma codificação de Huffman usando *bytes* do que uma codificação de Huffman usando *bits*, isso porque deslocamentos de *bits* e operações usando máscaras não são necessárias.
- Os experimentos foram realizados em uma máquina PC Pentium de 200 MHz com 128 *megabytes* de *RAM*.

Resultados Experimentais - Comparação das técnicas de compressão sobre o arquivo WSJ

Dados sobre a coleção usada nos experimentos:

Texto		Vocabulário		Vocab./Texto	
Tam (bytes)	#Palavras	Tam (bytes)	#Palavras	Tamanho	#Palavras
262.757.554	42.710.250	1.549.131	208.005	0,59%	0,48%

Método	Razão de Compressão	Tempo (min) de Compressão	Tempo (min) de Descompressão
Huffman binário	27,13	8,77	3,08
Huffman pleno	30,60	8,67	1,95
Huffman com marcação	33,70	8,90	2,02
Gzip	37,53	25,43	2,68
Compress	42,94	7,60	6,78

Pesquisa em Texto Comprimido

- Uma das propriedades mais atraentes do método de Huffman usando *bytes* em vez de *bits* é que o texto comprimido pode ser pesquisado exatamente como qualquer texto não comprimido.
- Basta comprimir o padrão e realizar uma pesquisa diretamente no arquivo comprimido.
- Isso é possível porque o código de Huffman usa *bytes* em vez de *bits*; de outra maneira, o método seria complicado ou mesmo impossível de ser implementado.

Casamento Exato

Algoritmo:

- Buscar a palavra no vocabulário, podendo usar busca binária nesta fase:
  - Se a palavra for localizada no vocabulário, então o código de Huffman com marcação é obtido.
  - Senão a palavra não existe no texto comprimido.
- A seguir, o código é pesquisado no texto comprimido usando qualquer algoritmo para casamento exato de padrão.
- Para pesquisar um padrão contendo mais de uma palavra, o primeiro passo é verificar a existência de cada palavra do padrão no vocabulário e obter o seu código:
  - Se qualquer das palavras do padrão não existir no vocabulário, então o padrão não existirá no texto comprimido.
  - Senão basta coletar todos os códigos obtidos e realizar a pesquisa no texto comprimido.

Método para realizar busca no arquivo comprimido

```
public void busca (String nomeArqComp) throws Exception {
  BufferedReader in = new BufferedReader (
                      new InputStreamReader (System.in));
  this.arqComp = new RandomAccessFile (nomeArqComp, "rws");
  int maxCompCod = this.leVetores ();
  String vocabulario[] = this.leVocabulario ();
  int codigo; String T = ""; String P = "";
  while ((codigo = this.arqComp.read ()) >= 0) T += (char)codigo;
  while (true) {
    System.out.print ("Padrao (ou s para sair):"); P = in.readLine();
    if (P.equals ("s")) break; int ord = 1;
    for (ord = 1; ord < vocabulario.length; ord++)
      if (vocabulario[ord].equals (P)) break;
    if (ord == vocabulario.length) {
      System.out.println("Padrao:" + P + " nao encontrado"); continue;
    }
    Codigo cod = this.codifica (ord, maxCompCod);
    String Padrao = this.atribui (cod);
    CasamentoExato.bmh (T, T.length (), Padrao, Padrao.length ());
  }
}
```

Método para atribuir o código ao padrão

```
private String atribui (Codigo cod) {
  String P = "";
  P += (char)((cod.codigo >> (7*(cod.c – 1))) | 128);
  cod.c––;
  while (cod.c > 0) {
    P += (char)((cod.codigo >> (7*(cod.c – 1))) & 127);
    cod.c––;
  }
  return P;
}
```

Programa para teste dos algoritmos de compressão, descompressão e busca exata em texto comprimido

```
package cap8;
import java.io.*;
public class Huffman {
  private static BufferedReader in = new BufferedReader (
                                     new InputStreamReader (System.in));
  private static final int baseNum = 128;
  private static final int m = 1001;
  private static final int maxTamPalavra = 15;
  private static void imprime (String msg) {
    System.out.print (msg);
  }
  public static void main (String[] args) throws Exception {
    imprime ("Arquivo com os delimitadores em uma linha:");
    String nomeArqDelim = in.readLine ();
    String opcao = "";
    do {
      imprime ("*************************************\n");
      imprime ("*                Opcoes               *\n");
      imprime ("*-------------------------------------*\n");
      imprime ("* (c) Compressao                      *\n");
      imprime ("* (d) Descompressao                   *\n");
      imprime ("* (p) Pesquisa no texto comprimido    *\n");
      imprime ("* (f) Termina                         *\n");
      imprime ("*************************************\n");
      imprime ("* Opcao:"); opcao = in.readLine();
```

Programa para teste dos algoritmos de compressão, descompressão e busca exata em texto comprimido

```
    if (opcao.toLowerCase().equals ("c")) {
        imprime ("Arquivo texto a ser comprimido:");
        String nomeArqTxt = in.readLine ();
        imprime ("Arquivo comprimido a ser gerado:");
        String nomeArqComp = in.readLine ();
        HuffmanByte huff = new HuffmanByte (nomeArqDelim, baseNum,
                                            m, maxTamPalavra);
        huff.compressao (nomeArqTxt, nomeArqComp);
      }
      else if (opcao.toLowerCase().equals ("d")) {
        imprime ("Arquivo comprimido a ser descomprimido:");
        String nomeArqComp = in.readLine ();
        imprime ("Arquivo texto a ser gerado:");
        String nomeArqTxt = in.readLine ();
        HuffmanByte huff = new HuffmanByte (nomeArqDelim, baseNum,
                                            m, maxTamPalavra);
        huff.descompressao (nomeArqTxt, nomeArqComp);
      }
      else if (opcao.toLowerCase().equals ("p")) {
        imprime ("Arquivo comprimido para ser pesquisado:");
        String nomeArqComp = in.readLine ();
        HuffmanByte huff = new HuffmanByte (null, baseNum,
                                            m, maxTamPalavra);
        huff.busca (nomeArqComp);
      }
    } while (!opcao.toLowerCase().equals ("f"));
  }
}
```

Casamento Aproximado

Algoritmo:

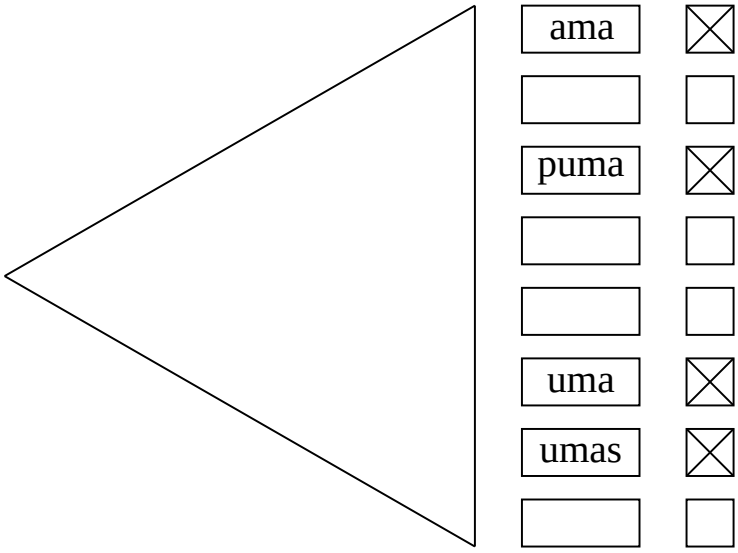
- Pesquisar o padrão no vocabulário. Neste caso, podemos ter:
  - Casamento exato, o qual pode ser uma **pesquisa binária** no vocabulário, e uma vez que a palavra tenha sido encontrada a folha correspondente na árvore de Huffman é marcada.
  - Casamento aproximado, o qual pode ser realizado por meio de pesquisa seqüencial no vocabulário, usando o algoritmo Shift-And.
  - Neste caso, várias palavras do vocabulário podem ser encontradas e a folha correspondente a cada uma na árvore de Huffman é marcada.

Casamento Aproximado

Algoritmo (Continuação):

- A seguir, o arquivo comprimido é lido *byte* a *byte*, ao mesmo tempo que a árvore de decodificação de Huffman é percorrida sincronizadamente.
- Ao atingir uma folha da árvore:
  - se ela estiver marcada, então existe casamento com a palavra do padrão.
- Seja uma folha marcada ou não, o caminhamento na árvore volta à raiz ao mesmo tempo que a leitura do texto comprimido continua.

Esquema geral de pesquisa para a palavra "*uma*" permitindo 1 erro



Casamento Aproximado Usando uma Frase como Padrão

- **Frase:** seqüência de padrões (palavras), em que cada padrão pode ser desde uma palavra simples até uma expressão regular complexa permitindo erros.

Pré-Processamento:

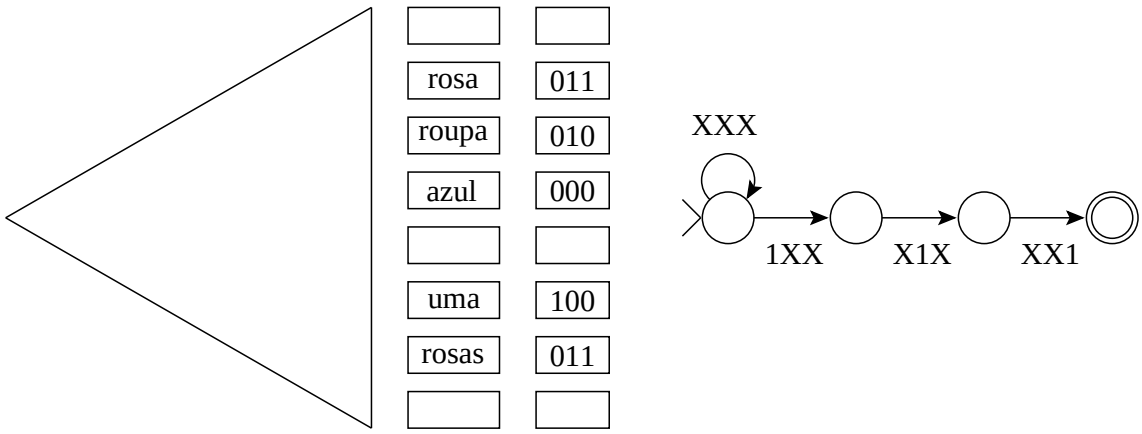
- Se uma frase tem j palavras, então uma máscara de j *bits* é colocada junto a cada palavra do vocabulário (folha da árvore de Huffman).
- Para uma palavra x da frase, o i-ésimo *bit* da máscara é feito igual a 1 se x é a i-ésima palavra da frase.
- Assim, cada palavra i da frase é pesquisada no vocabulário e a i-ésima posição da máscara é marcada quando a palavra é encontrada no vocabulário.

Casamento Aproximado Usando uma Frase como Padrão

Leitura do Texto Comprimido:

- O estado da pesquisa é controlado por um **autômato finito não-determinista** de $j + 1$ estados.
- O autômato permite mover do estado i para o estado $i + 1$ sempre que a i-ésima palavra da frase é reconhecida.
- O estado zero está sempre ativo e uma ocorrência é relatada quando o estado j é ativado.
- Os *bytes* do texto comprimido são lidos e a árvore de Huffman é percorrida como antes.
- Cada vez que uma folha da árvore é atingida, sua máscara de *bits* é enviada para o autômato.
- Um estado ativo $i - 1$ irá ativar o estado i apenas se o i-ésimo *bit* da máscara estiver ativo.
- Conseqüentemente, o autômato realiza uma transição para cada palavra do texto.

Esquema geral de pesquisa para a frase "`uma ro* rosa`"



- O autômato pode ser implementado eficientemente por meio do algoritmo Shift-And
- Separadores podem ser ignorados na pesquisa de frases.
- Da mesma maneira, os artigos, preposições etc., também podem ser ignorados se for conveniente.
- Neste caso, basta ignorar as folhas correspondentes na árvore de Huffman quando a pesquisa chega a elas.
- É raro encontrar esta possibilidade em sistemas de pesquisa *on-line*.

Tempos de pesquisa (em segundos) para o arquivo WSJ, com intervalo de confiança de 99%

Algoritmo	$k = 0$	$k = 1$	$k = 2$	$k = 3$
Agrep	23,8 $\pm$ 0,38	117,9 $\pm$ 0,14	146,1 $\pm$ 0,13	174,6 $\pm$ 0,16
Pesquisa direta	14,1 $\pm$ 0,18	15,0 $\pm$ 0,33	17,0 $\pm$ 0,71	22,7 $\pm$ 2,23
Pesquisa com autômato	22,1 $\pm$ 0,09	23,1 $\pm$ 0,14	24,7 $\pm$ 0,21	25,0 $\pm$ 0,49

Problemas $\mathcal{NP}$-Completo e Algoritmos Aproximados*

Última alteração: 10 de Outubro de 2006

*Transparências elaboradas por Charles Ornelas, Leonardo Rocha, Leonardo Mata, Elisa Tuler e Nivio Ziviani

Introdução

- Problemas intratáveis ou difíceis são comuns na natureza e nas áreas do conhecimento.
- Problemas “fáceis”: resolvidos por algoritmos polinomiais.
- Problemas “difíceis”: somente possuem algoritmos exponenciais para resolvê-los.
- A complexidade de tempo da maioria dos problemas é polinomial ou exponencial.
- **Polinomial**: função de complexidade é $O(p(n))$, em que $p(n)$ é um polinômio.
  - Ex.: algoritmos com pesquisa binária ($O(\log n)$), pesquisa sequencial ($O(n)$), ordenação por inserção ($O(n^2)$), e multiplicação de matrizes ($O(n^3)$).
- **Exponencial**: função de complexidade é $O(c^n)$, $c > 1$.
  - Ex.: **problema do caixeiro-viajante** (PCV) ($O(n!)$).
  - Mesmo problemas de tamanho pequeno a moderado não podem ser resolvidos por algoritmos não-polinomiais.

Problemas $\mathcal{NP}$-Completo

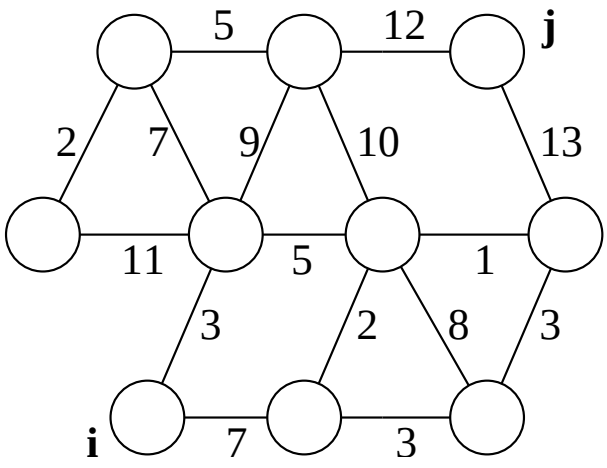
- A teoria de complexidade a ser apresentada não mostra como obter algoritmos polinomiais para problemas que demandam algoritmos exponenciais, nem afirma que não existem.
- É possível mostrar que os problemas para os quais não há algoritmo polinomial conhecido são computacionalmente relacionados.
- Formam a classe conhecida como $\mathcal{NP}$.
- Propriedade: um problema da classe $\mathcal{NP}$ poderá ser resolvido em tempo polinomial se e somente se todos os outros problemas em $\mathcal{NP}$ também puderem.
- Este fato é um indício forte de que dificilmente alguém será capaz de encontrar um algoritmo eficiente para um problema da classe $\mathcal{NP}$.

Classe $\mathcal{NP}$ - Problemas “Sim/Não”

- Para o estudo teórico da complexidade de algoritmos considera-se problemas cujo resultado da computação seja “sim” ou “não”.
- Versão do Problema do Caixeiro-Viajante (PCV) cujo resultado é do tipo “sim/não”:
  - Dados: uma constante k, um conjunto de cidades $C = \{c_1, c_2, \cdots, c_n\}$ e uma distância $d(c_i, c_j)$ para cada par de cidades $c_i, c_j \in C$.
  - Questão: Existe um “roteiro” para todas as cidades em C cujo comprimento total seja menor ou igual a k?
- Característica da classe $\mathcal{NP}$: problemas “sim/não” para os quais uma dada solução pode ser verificada facilmente.
- A solução pode ser muito difícil ou impossível de ser obtida, mas uma vez conhecida ela pode ser verificada em tempo polinomial.

Caminho em um Grafo

- Considere um grafo com peso nas arestas, dois vértices i, j e um inteiro $k > 0$.



- *Fácil:* Existe um caminho de i até j com peso $\leq k$?.
  - Há um algoritmo eficiente com complexidade de tempo $O(A \log V)$, sendo A o número de arestas e V o número de vértices (algoritmo de Dijkstra).
- *Difícil:* Existe um caminho de i até j com peso $\geq k$?
  - Não existe algoritmo eficiente. É equivalente ao PCV em termos de complexidade.

Coloração de um Grafo

- Em um grafo $G = (V, A)$, mapear $C : V \leftarrow S$, sendo S um conjunto finito de cores tal que se $\overline{vw} \in A$ então $c(v) \neq c(w)$ (vértices adjacentes possuem cores distintas).
- O número cromático $X(G)$ de G é o menor número de cores necessário para colorir G, isto é, o menor k para o qual existe uma coloração C para G e $|C(V)| = k$.
- O problema é produzir uma coloração ótima, que é a que usa apenas $X(G)$ cores.
- Formulação do tipo “sim/não”: dados G e um inteiro positivo k, existe uma coloração de G usando k cores?
  - *Fácil*: $k = 2$.
  - *Difícil*: $k > 2$.
- Aplicação: modelar problemas de agrupamento (*clustering*) e de horário (*scheduling*).

Coloração de um Grafo - Otimização de Compiladores

- Escalonar o uso de um número finito de registradores (idealmente com o número mínimo).
- No trecho de programa a ser otimizado, cada variável tem intervalos de tempo em que seu valor tem de permanecer inalterado, como depois de inicializada e antes do uso final.
- Variáveis com interseção nos tempos de vida útil não podem ocupar o mesmo registrador.
- Modelagem por grafo: vértices representam variáveis e cada aresta liga duas variáveis que possuem interseção nos tempos de vida.
- Coloração dos vértices: atibui cada variável a um agrupamento (ou classe). Duas variáveis com a mesma cor não colidem, podendo assim ser atribuídas ao mesmo registrador.

Coloração de um Grafo - Otimização de Compiladores

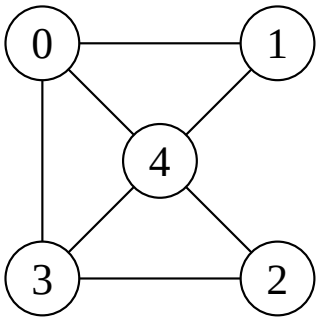
- Evidentemente, não existe conflito se cada vértice for colorido com uma cor distinta.
- O objetivo porém é encontrar uma coloração usando o mínimo de cores (computadores têm um número limitado de registradores).
- **Número cromático**: menor número de cores suficientes para colorir um grafo.

Coloração de um Grafo - Problema de Horário

- Suponha que os exames finais de um curso tenham de ser realizados em uma única semana.
- Disciplinas com alunos de cursos diferentes devem ter seus exames marcados em horários diferentes.
- Dadas uma lista de todos os cursos e outra lista de todas as disciplinas cujos exames não podem ser marcados no mesmo horário, o problema em questão pode ser modelado como um problema de coloração de grafos.

Ciclo de Hamilton

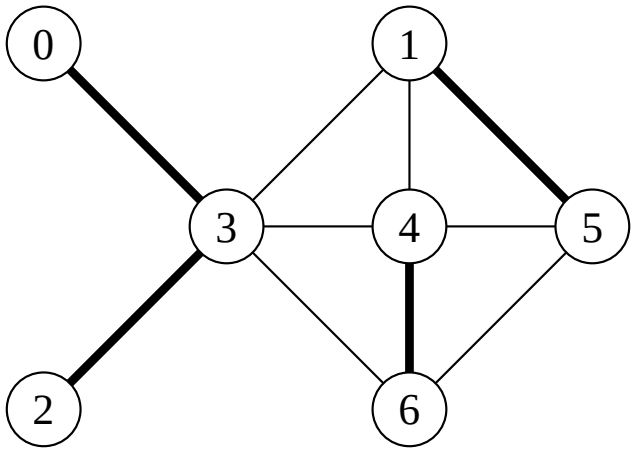
- **Ciclo de Hamilton**: **ciclo simples** (passa por todos os vértices uma única vez).
- **Caminho de Hamilton**: **caminho simples** (passa por todos os vértices uma única vez).
- Exemplo de ciclo de Hamilton: 0 1 4 2 3 0.
  Exemplo de caminho de Hamilton: 0 1 4 2 3.



- Existe um ciclo de Hamilton no grafo G?
  - *Fácil*: Grafos com grau máximo = 2 (vértices com no máximo duas arestas incidentes).
  - *Difícil*: Grafos com grau $>$ 2.
- É um caso especial do PCV. Pares de vértices com uma aresta entre eles tem distância 1 e pares de vértices sem aresta entre eles têm distância infinita.

Cobertura de Arestas

- Uma **cobertura de arestas** de um grafo $G = (V, A)$ é um subconjunto $A' \subset A$ de k arestas tal que todo $v \in V$ é parte de pelo menos uma aresta de A'.
- O conjunto resposta para $k = 4$ é $A' = \{(0, 3), (2, 3), (4, 6), (1, 5)\}$.



- Uma **cobertura de vértices** é um subconjunto $V' \subset V$ tal que se $(u, v) \in A$ então $u \in V'$ ou $v \in V'$, isto é, cada aresta do grafo é incidente em um dos vértices de V'.
- Na figura, o conjunto resposta é $V' = \{3, 4, 5\}$, para $k = 3$.
- Dados um grafo e um inteiro $k > 0$
  - *Fácil*: há uma cobertura de arestas $\leq k$?.
  - *Difícil*: há uma cobertura de vértices $\leq k$?

Algoritmos Não-Deterministas

- **Algoritmos deterministas**: o resultado de cada operação é definido de forma única.
- Em um arcabouço teórico, é possível remover essa restrição.
- Apesar de parecer irreal, este é um conceito importante e geralmente utilizado para definir a classe $\mathcal{NP}$.
- Neste caso, os algoritmos podem conter operações cujo resultado não é definido de forma única.
- **Algorimo não-determinista**: capaz de escolher uma dentre as várias alternativas possíveis a cada passo.
- Algoritmos não-deterministas contêm operações cujo resultado não é unicamente definido, ainda que limitado a um conjunto especificado de possibilidades.

Função *escolhe(C)*

- Algoritmos não-deterministas utilizam uma função *escolhe(C)*, que escolhe um dos elementos do conjunto C de forma arbitrária.
- O comando de atribuição X $\leftarrow$ *escolhe* (1:n) pode resultar na atribuição a X de qualquer dos inteiros no intervalo $[1, n]$.
- A complexidade de tempo para cada chamada da função *escolhe* é $O(1)$.
- Neste caso, não existe nenhuma regra especificando como a escolha é realizada.
- Se um conjunto de possibilidades levam a uma resposta, este conjunto é escolhido sempre e o algoritmo terminará com sucesso.
- Em contrapartida, um algoritmo não-determinista termina sem sucesso se e somente se não há um conjunto de escolhas que indique sucesso.

Comandos *sucesso* e *insucesso*

- Algoritmos não-deterministas utilizam também dois comandos, a saber:
  - *insucesso*: indica término sem sucesso.
  - *sucesso*: indica término com sucesso.
- Os comandos *insucesso* e *sucesso* são usados para definir uma execução do algoritmo.
- Esses comandos são equivalentes a um comando de parada de um algoritmo determinista.
- Os comandos *insucesso* e *sucesso* também têm complexidade de tempo $O(1)$.

Máquina Não-Determinista

- Uma máquina capaz de executar a função *escolhe* admite a capacidade de **computação não-determinista**.
- Uma máquina não-determinista é capaz de produzir cópias de si mesma quando diante de duas ou mais alternativas, e continuar a computação independentemente para cada alternativa.
- A máquina não-determinista que acabamos de definir não existe na prática, mas ainda assim fornece fortes evidências de que certos problemas não podem ser resolvidos por algoritmos deterministas em tempo polinomial, conforme mostrado na definição da classe $\mathcal{NP}$-completo à frente.

Pesquisa Não-Determinista

- Pesquisar o elemento x em um conjunto de elementos $A[1:n]$, $n \geq 1$.

```
void pesquisaND (x, A, 1, n) {
   j ← escolhe (A, 1, n);
   if (A[j] == x) sucesso; else insucesso;
}
```

- Determina um índice j tal que $A[j] = x$ para um término com sucesso ou então insucesso quando x não está presente em A.
- O algoritmo tem complexidade não-determinista $O(1)$.
- Para um algoritmo determinista a complexidade é $O(n)$.

Ordenação Não-Determinista

- Ordenar um conjunto $A[1:n]$ contendo n inteiros positivos, $n \geq 1$.

```
void ordenaND (A, 1, n) {
  for (int i = 1; i <= n; i++) B[i] = 0;
  for (int i = 1; i <= n; i++) {
    j ← escolhe (A, 1, n);
    if (B[j] == 0) B[j] = A[i]; else insucesso;
  }
}
```

- Um vetor auxiliar B[1:n] é utilizado. Ao final, B contém o conjunto em ordem crescente.
- A posição correta em B de cada inteiro de A é obtida de forma não-determinista pela função escolhe.
- Em seguida, o comando de decisão verifica se a posição B[j] ainda não foi utilizada.
- A complexidade é $O(n)$. (Para um algoritmo determinista a complexidade é $O(n \log n)$)

Problema da Satisfabilidade

- Considere um conjunto de **variáveis booleanas** $x_1, x_2, \cdots, x_n$, que podem assumir valores lógicos *verdadeiro* ou *falso*.
- A negação de x_i é representada por $\overline{x_i}$.
- Expressão booleana: variáveis booleanas e operações **ou** ($\vee$) e **e** ($\wedge$). (também chamadas respectivamente de adição e multiplicação).
- Uma expressão booleana E contendo um produto de adições de variáveis booleanas é dita estar na **forma normal conjuntiva**.
- Dada E na forma normal conjuntiva, com variáveis $x_i, 1 \leq i \leq n$, existe uma atribuição de valores verdadeiro ou falso às variáveis que torne E verdadeira ("satisfaça")?
- $E_1 = (x_1 \vee x_2) \wedge (x_1 \vee \overline{x_3} \vee x_2) \wedge (x_3)$ é *satisfatível* ($x_1 = F$, $x_2 = V$, $x_3 = V$).
- A expressão $E_2 = x_1 \wedge \overline{x_1}$ não é *satisfatível*.

Problema da Satisfabilidade

- O algoritmo $avalND(E, n)$ verifica se uma expressão E na forma normal conjuntiva, com variáveis $x_i, 1 \leq i \leq n$, é *satisfatível*.

```
void avalND (E, n) {
  for (int i = 1; i <= n; i++) {
    x_i ← escolhe (true, false);
    if (E(x_1, x_2, ···, x_n) == true) sucesso; else insucesso;
  }
}
```

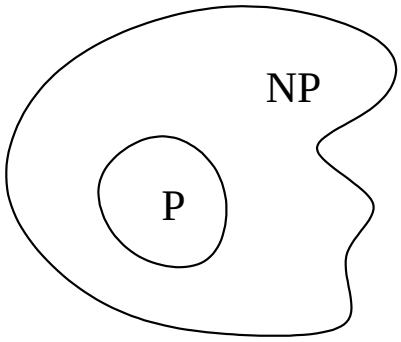
- O algoritmo obtém uma das 2^n atribuições possíveis de forma não-determinista em $O(n)$.
- Melhor algoritmo determinista: $O(2^n)$.
- Aplicação: definição de circuitos elétricos combinatórios que produzam valores lógicos como saída e sejam constituídos de portas lógicas **e**, **ou** e **não**.
- Neste caso, o mapeamento é direto, pois o circuito pode ser descrito por uma expressão lógica na forma normal conjuntiva.

Caracterização das Classes $\mathcal{P}$ e $\mathcal{NP}$

- $\mathcal{P}$: conjunto de todos os problemas que podem ser resolvidos por *algoritmos deterministas* em tempo *polinomial*.
- $\mathcal{NP}$: conjunto de todos os problemas que podem ser resolvidos por *algoritmos não-deterministas* em tempo *polinomial*.
- Para mostrar que um determinado problema está em $\mathcal{NP}$, basta apresentar um algoritmo não-determinista que execute em tempo polinomial para resolver o problema.
- Outra maneira é encontrar um algoritmo determinista polinomial para verificar que uma dada solução é válida.

Existe Diferença entre $\mathcal{P}$ e $\mathcal{NP}$?

- $\mathcal{P} \subseteq \mathcal{NP}$, pois algoritmos deterministas são um caso especial dos não-deterministas.
- A questão é se $\mathcal{P} = \mathcal{NP}$ ou $\mathcal{P} \neq \mathcal{NP}$.
- Esse é o problema não resolvido mais famoso que existe na área de ciência da computação.
- Se existem algoritmos polinomiais deterministas para todos os problemas em $\mathcal{NP}$, então $\mathcal{P} = \mathcal{NP}$.
- Em contrapartida, a prova de que $\mathcal{P} \neq \mathcal{NP}$ parece exigir técnicas ainda desconhecidas.
- Descrição tentativa do mundo $\mathcal{NP}$:



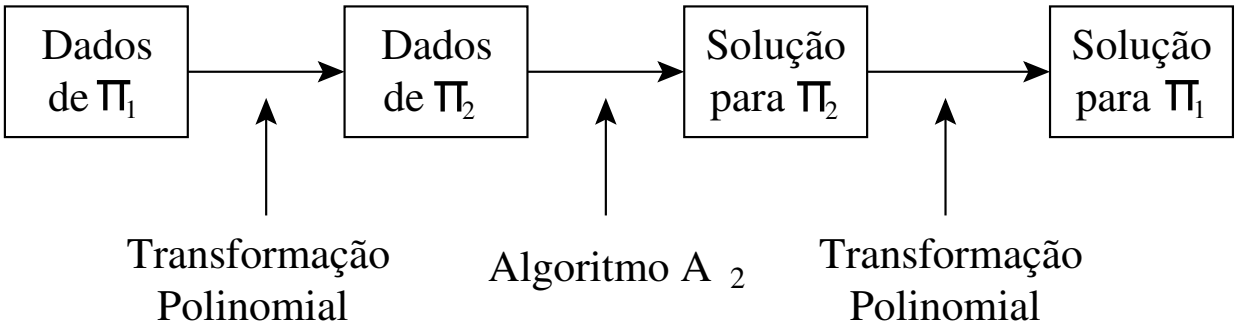
- Acredita-se que $\mathcal{NP} \gg \mathcal{P}$, pois para muitos problemas em $\mathcal{NP}$, não existem algoritmos polinomiais conhecidos, nem um **limite inferior não-polinomial** provado.

$\mathcal{NP} \supset \mathcal{P}$ ou $\mathcal{NP} = \mathcal{P}$? - Conseqüências

- Muitos problemas práticos em $\mathcal{NP}$ podem ou não pertencer a $\mathcal{P}$ (não conhecemos nenhum algoritmo determinista eficiente para eles).
- Se conseguirmos provar que um problema não pertence a $\mathcal{P}$, então temos um indício de que esse problema pertence a $\mathcal{NP}$ e que esse problema é tão difícil de ser resolvido quanto outros problemas $\mathcal{NP}$.
- Como não existe tal prova, sempre há esperança de que alguém descubra um algoritmo eficiente.
- Quase ninguém acredita que $\mathcal{NP} = \mathcal{P}$.
- Existe um esforço considerável para provar o contrário, mas a questão continua em aberto!

Transformação Polinomial

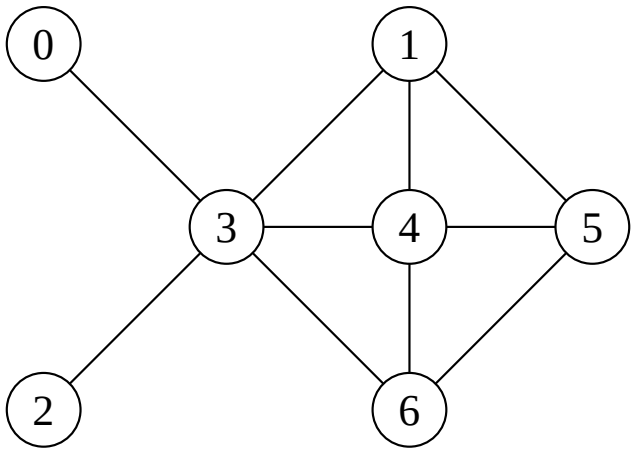
- Sejam Π_1 e Π_2 dois problemas “sim/não”.
- Suponha que um algoritmo A_2 resolva Π_2.
- Se for possível transformar Π_1 em Π_2 e a solução de Π_2 em solução de Π_1, então A_2 pode ser utilizado para resolver Π_1.
- Se pudermos realizar as transformações nos dois sentidos em tempo polinomial, então Π_1 é *polinomialmente transformável* em Π_2.



- Esse conceito é importante para definir a classe $\mathcal{NP}$-completo.
- Para apresentar um exemplo de transformação polinomial vamos necessitar das definições de conjunto independente de vértices e clique de um grafo.

Conjunto Independente de Vértices de um Grafo

- O conjunto independente de vértices de um grafo $G = (V, A)$ é constituído do subconjunto $V' \subseteq V$, tal que $v, w \in V' \Rightarrow (v, w) \notin A$.
- Todo par de vértices de V' é não adjacente (V' é um subgrafo totalmente desconectado).
- Exemplo de cardinalidade 4: $V' = \{0, 2, 1, 6\}$.

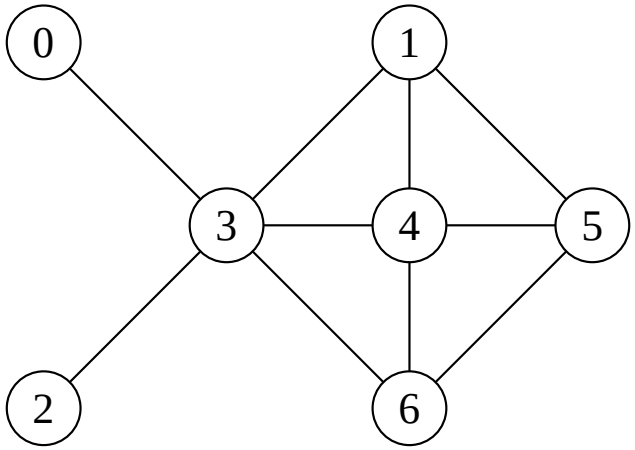


Conjunto Independente de Vértices - Aplicação

- Em problemas de dispersão é necessário encontrar grandes conjuntos independentes de vértices. Procura-se um conjunto de pontos mutuamente separados.
- Exemplo: identificar localizações para instalação de franquias.
- Duas localizações não podem estar perto o suficiente para competir entre si.
- Solução: construir um grafo em que possíveis localizações são representadas por vértices, e arestas são criadas entre duas localizações que estão próximas o suficiente para interferir.
- O maior conjunto independente fornece o maior número de franquias que podem ser concedidas sem prejudicar as vendas.
- Em geral, cunjuntos independentes evitam conflitos entre elementos.

Clique de um grafo

- **Clique** de um grafo $G = (V, A)$ é constituído do subconjunto $V' \subseteq V$, tal que $v, w \in V' \Rightarrow (v, w) \in A$.
- Todo par de vértices de V' é adjacente (V' é um subgrafo completo).
- Exemplo de cardinalidade 3: $V' = \{3, 1, 4\}$.



Clique de um grafo - Aplicação

- O problema de identificar agrupamentos de objetos relacionados freqüentemente se reduz a encontrar grandes cliques em grafos.
- Exemplo: empresa de fabricação de peças por meio de injeção plástica que fornece para diversas outras empresas montadoras.
- Para reduzir o custo relativo ao tempo de preparação das máquinas injetoras, pode-se aumentar o tamanho dos lotes produzidos para cada peça encomendada.
- É preciso identificar os clientes que adquirem os mesmos produtos, para negociar prazos de entrega comuns e assim aumentar o tamanho dos lotes produzidos.
- Solução: construir um grafo com cada vértice representando um cliente e ligar com uma aresta os que adquirem os mesmos produtos.
- Um clique no grafo representa o conjunto de clientes que adquirem os mesmos produtos.

Transformação Polinomial

- Considere Π_1 o problema clique e Π_2 o problema conjunto independente de vértices.
- A instância I de clique consiste de um grafo $G = (V, A)$ e um inteiro $k > 0$.
- A instância $f(I)$ de conjunto independente pode ser obtida considerando-se o grafo complementar $\overline{G}$ de G e o mesmo inteiro k.
- $f(I)$ é uma transformação polinomial:
  1. $\overline{G}$ pode ser obtido a partir de G em tempo polinomial.
  2. G possui clique de tamanho $\geq k$ se e somente se $\overline{G}$ possui conjunto independente de vértices de tamanho $\geq k$.

Transformação Polinomial

- Se existe um algoritmo que resolve o conjunto independente em tempo polinomial, ele pode ser utilizado para resolver clique também em tempo polinomial.
- Diz-se que clique $\propto$ conjunto independente.
- Denota-se $\Pi_1 \propto \Pi_2$ para indicar que Π_1 é polinomialmente transformável em Π_2.
- A relação $\propto$ é transitiva ($\Pi_1 \propto \Pi_2$ e $\Pi_2 \propto \Pi_3 \Rightarrow \Pi_1 \propto \Pi_3$).

Problemas $\mathcal{NP}$-Completo e $\mathcal{NP}$-Difícil

- Dois problemas Π_1 e Π_2 são **polinomialmente equivalentes** se e somente se $\Pi_1 \propto \Pi_2$ e $\Pi_2 \propto \Pi_1$.
- Exemplo: **problema da *satisfabilidade***. Se $SAT \propto \Pi_1$ e $\Pi_1 \propto \Pi_2$, então $SAT \propto \Pi_2$.
- Um problema Π é $\mathcal{NP}$**-difícil** se e somente se $SAT \propto \Pi$ (*satisfabilidade* é redutível a Π).
- Um problema de decisão Π é denominado $\mathcal{NP}$**-completo** quando:
  1. $\Pi \in NP$;.
  2. Todo problema de decisão $\Pi' \in \mathcal{NP}$-completo satisfaz $\Pi' \propto \Pi$.

Problemas $\mathcal{NP}$-Completo e $\mathcal{NP}$-Difícil

- Um problema de decisão Π que seja $\mathcal{NP}$-difícil pode ser mostrado ser $\mathcal{NP}$-completo exibindo um algoritmo não-determinista polinomial para Π.
- Apenas problemas de decisão (“sim/não”) podem ser $\mathcal{NP}$-completo.
- Problemas de otimização podem ser $\mathcal{NP}$-difícil, mas geralmente, se Π_1 é um problema de decisão e Π_2 um problema de otimização, é bem possível que $\Pi_1 \propto \Pi_2$.
- A dificuldade de um problema $\mathcal{NP}$-difícil não é menor do que a dificuldade de um problema $\mathcal{NP}$-completo.

Exemplo - Problema da Parada

- É um exemplo de problema $\mathcal{NP}$-difícil que não é $\mathcal{NP}$-completo.
- Consiste em determinar, para um algoritmo determinista qualquer A com entrada de dados E, se o algoritmo A termina (ou entra em um *loop* infinito).
- É um problema **indecidível**. Não há algoritmo de qualquer complexidade para resolvê-lo.
- Mostrando que $SAT \propto$ problema da parada:
  - Considere o algoritmo A cuja entrada é uma expressão booleana na forma normal conjuntiva com n variáveis.
  - Basta tentar 2^n possibilidades e verificar se E é *satisfatível*.
  - Se for, A pára; senão, entra em *loop*.
  - Logo, o problema da parada é $\mathcal{NP}$-difícil, mas não é $\mathcal{NP}$-completo.

Teorema de Cook

- Existe algum problema em $\mathcal{NP}$ tal que se ele for mostrado estar em $\mathcal{P}$, implicaria $\mathcal{P} = \mathcal{NP}$?
- **Teorema de Cook:** Satisfabilidade (SAT) está em $\mathcal{P}$ se e somente se $\mathcal{P} = \mathcal{NP}$.
- Ou seja, se existisse um algoritmo polinomial determinista para *satisfabilidade*, então todos os problemas em $\mathcal{NP}$ poderiam ser resolvidos em tempo polinomial.
- A prova considera os dois sentidos:
  1. SAT está em $\mathcal{NP}$ (basta apresentar um algoritmo não-determinista que execute em tempo polinomial). Logo, se $\mathcal{P} = \mathcal{NP}$, então SAT está em $\mathcal{P}$.
  2. Se SAT está em $\mathcal{P}$, então $\mathcal{P} = \mathcal{NP}$. A prova descreve como obter de qualquer algoritmo polinomial não determinista de decisão A, com entrada E, uma fórmula $Q(A, E)$ de modo que Q é *satisfatível* se e somente se A termina com sucesso para E. O tempo necessário para construir Q é $O(p^3(n)\log(n))$, em que n é o tamanho de E e $p(n)$ é a complexidade de A.

Prova do Teorema de Cook

- A prova, bastante longa, mostra como construir Q a partir de A e E.
- A expressão booleana Q é longa, mas pode ser construída em tempo polinomial no tamanho de E.
- Prova usa definição matemática da **Máquina de Turing não-determinista** (MTND), capaz de resolver qualquer problema em $\mathcal{NP}$.
  - incluindo uma descrição da máquina e de como instruções são executadas em termos de fórmulas booleanas.
- Estabelece uma correspondência entre todo problema em $\mathcal{NP}$ (expresso por um programa na MTnd) e alguma instância de SAT.
- Uma instância de SAT corresponde à tradução do programa em uma fórmula booleana.
- A solução de SAT corresponde à simulação da máquina executando o programa em cima da fórmula obtida, o que produz uma solução para uma instância do problema inicial dado.

Prova de que um Problema é $\mathcal{NP}$-Completo

- São necessários os seguintes passos:
  1. Mostre que o problema está em $\mathcal{NP}$.
  2. Mostre que um problema $\mathcal{NP}$-completo conhecido pode ser polinomialmente transformado para ele.
- É possível porque Cook apresentou uma prova direta de que SAT é $\mathcal{NP}$-completo, além do fato de a redução polinomial ser transitiva ($SAT \propto \Pi_1 \ \& \ \Pi_1 \propto \Pi_2 \Rightarrow SAT \propto \Pi_2$).
- Para ilustrar como um problema Π pode ser provado ser $\mathcal{NP}$-completo, basta considerar um problema já provado ser $\mathcal{NP}$-completo e apresentar uma redução polinomial desse problema para Π.

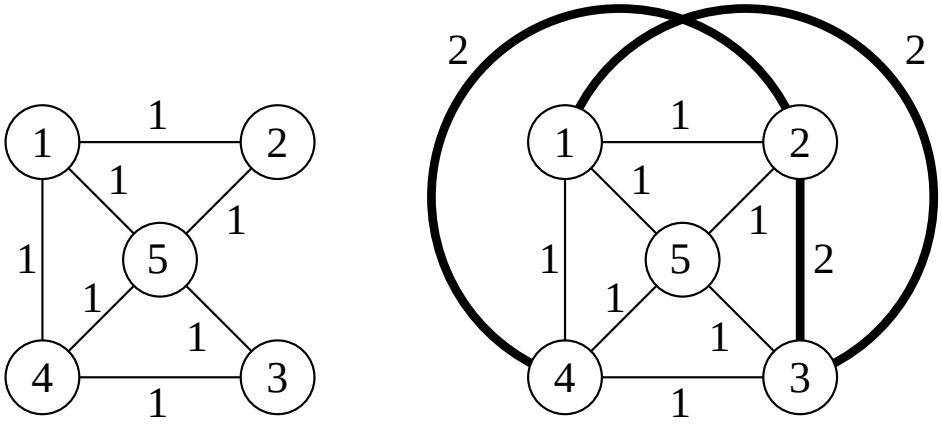
PCV é $\mathcal{NP}$-completo - Parte 1 da Prova

- Mostrar que o Problema do Caixeiro-Viajante (PCV) está em $\mathcal{NP}$.
- Prova a partir do problema **ciclo de Hamilton**, um dos primeiros que se provou ser $\mathcal{NP}$-completo.
- Isso pode ser feito:
  - apresentando (como abaixo) um algoritmo não-determinista polinomial para o PCV ou
  - mostrando que, a partir de uma dada solução para o PCV, esta pode ser verificada em tempo polinomial.

```
void PCVND() {
  i = 1;
  for (int t = 1; t <= v; t++) {
    j ← escolhe(i, lista_adj(i));
    antecessor[j] = i;
  }
}
```

PCV é $\mathcal{NP}$-completo - Parte 2 da Prova

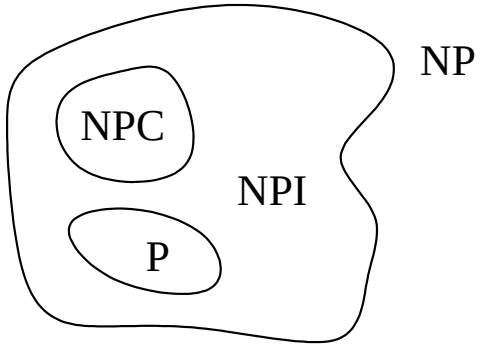
- Apresentar uma redução polinomial do ciclo de Hamilton para o PCV.
- Pode ser feita conforme o exemplo abaixo.



- Dado um grafo representando uma instância do ciclo de Hamilton, construa uma instância do PCV como se segue:
  1. Para cidades use os vértices.
  2. Para distâncias use 1 se existir um arco no grafo original e 2 se não existir.
- A seguir, use o PCV para achar um roteiro menor ou igual a V.
- O roteiro é o ciclo de Hamilton.

Classe $\mathcal{NP}$-Intermediária

- Segunda descrição tentativa do mundo $\mathcal{NP}$, assumindo $\mathcal{P} \neq \mathcal{NP}$.



- Existe uma classe intermediária entre $\mathcal{P}$ e $\mathcal{NP}$ chamada $\mathcal{NPI}$.
- $\mathcal{NPI}$ seria constituída por problemas nos quais ninguém conseguiu uma redução polinomial de um problema $\mathcal{NP}$-completo para eles, onde $\mathcal{NPI} = \mathcal{NP}$ - ($\mathcal{P} \cup \mathcal{NP}$-completo).

Membros Potenciais de $\mathcal{NPI}$

- **Isomorfismo de grafos:** Dados $G = (V, E)$ e $G' = (V, E')$, existe uma função $f : V \to V$, tal que $(u, v) \in E \Leftrightarrow (f(u), f(v)) \in E'$?
  - Isomorfismo é o problema de testar se dois grafos são o mesmo.
  - Suponha que seja dado um conjunto de grafos e que alguma operação tenha de ser realizada sobre cada grafo.
  - Se pudermos identificar quais grafos são duplicatas, eles poderiam ser descartados para evitar trabalho redundante.
- **Números compostos:** Dado um inteiro positivo k, existem inteiros $m, n > 1$ tais que $k = mn$?
  - Princípio da criptografia RSA: é fácil encontrar números primos grandes, mas difícil fatorar o produto de dois deles.

Classe $\mathcal{NP}$-Completo - Resumo

- Problemas que pertencem a $\mathcal{NP}$, mas que podem ou não pertencer a $\mathcal{P}$.
- Propriedade: se qualquer problema $\mathcal{NP}$-completo puder ser resolvido em tempo polinomial por uma máquina determinista, então todos os problemas da classe podem, isto é, $\mathcal{P} = \mathcal{NP}$.
- A falha coletiva de todos os pesquisadores para encontrar algoritmos eficientes para estes problemas pode ser vista como uma dificuldade para provar que $\mathcal{P} = \mathcal{NP}$.
- Contribuição prática da teoria: fornece um mecanismo que permite descobrir se um novo problema é “fácil” ou “difícil”.
- Se encontrarmos um algoritmo eficiente para o problema, então não há dificuldade. Senão, uma prova de que o problema é $\mathcal{NP}$-completo nos diz que o problema é tão “difícil” quanto todos os outros problemas “difíceis” da classe $\mathcal{NP}$-completo.

Problemas Exponenciais

- É desejável resolver instâncias grandes de problemas de otimização em tempo razoável.
- Os melhores algoritmos para problemas $\mathcal{NP}$-completo têm comportamento de pior caso exponencial no tamanho da entrada.
- Para um algoritmo que execute em tempo proporcional a 2^N, não é garantido obter resposta para todos os problemas de tamanho $N \geq 100$.
- Independente da velocidade do computador, ninguém poderia esperar por um algoritmo que leva 2^{100} passos para terminar sua tarefa.
- Um supercomputador poderia resolver um problema de tamanho $N = 50$ em 1 hora, ou $N = 51$ em 2 horas, ou $N = 59$ em um ano.
- Mesmo um computador paralelo contendo um milhão de processadores, (sendo cada processador um milhão de vezes mais rápido que o melhor processador que possa existir) não seria suficiente para chegar a $N = 100$.

O Que Fazer para Resolver Problemas Exponenciais?

- Usar algoritmos exponenciais “eficientes” aplicando técnicas de tentativa e erro.
- Usar algoritmos aproximados. Acham uma resposta que pode não ser a solução ótima, mas é garantido ser próxima dela.
- Concentrar no caso médio. Buscar algoritmos melhores que outros neste quesito e que funcionem bem para as entradas de dados que ocorrem usualmente na prática.
  - Existem poucos algoritmos exponenciais que são muito úteis na prática.
  - Exemplo: Simplex (programação linear). Complexidade de tempo exponencial no pior caso, mas muito rápido na prática.
  - Tais exemplos são raros. A grande maioria dos algoritmos exponenciais conhecidos não é muito útil.

Ciclo de Hamilton - Tentativa e Erro

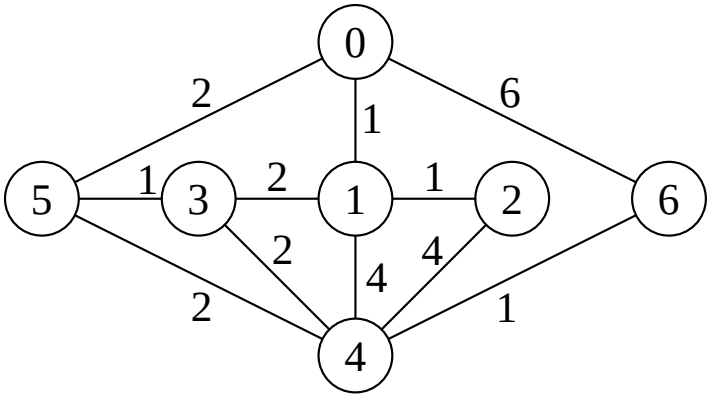
- Ex.: encontrar um **ciclo de Hamilton** em um grafo.
- Obter algoritmo tentativa e erro a partir de algoritmo para caminhamento em um grafo.
- O algoritmo para caminhamento em um grafo faz uma busca em profundidade no grafo em tempo $O(|V| + |A|)$.

Ciclo de Hamilton - Tentativa e Erro

```
package cap9;
import cap7.matrizadj.Grafo;
public class BuscaEmProfundidade {
  private int d[];
  private Grafo grafo;
  public BuscaEmProfundidade (Grafo grafo) {
    this.grafo = grafo; int n = this.grafo.numVertices();
    d = new int[n]; }
  private int visita (int u, int tempo) {
    this.d[u] = ++tempo;
    if (!this.grafo.listaAdjVazia (u)) {
      Grafo.Aresta a = this.grafo.primeiroListaAdj (u);
      while (a != null) {
        int v = a.v2 ();
        if (this.d[v] == 0) tempo = this.visita (v, tempo);
        a = this.grafo.proxAdj (u);}
    }
    return tempo;
  }
  public void buscaEmProfundidade () {
    int tempo = 0;
    for (int u = 0; u < grafo.numVertices (); u++)
      this.d[u] = 0;
    this.visita (0, tempo);
  }
}
```

Ciclo de Hamilton - Tentativa e Erro

- O método $visita$, quando aplicado ao grafo abaixo a partir do vértice 0, obtém o caminho 0 1 2 4 3 5 6, o qual não é um ciclo simples.



	1	2	3	4	5	6
0	1				2	6
1		1	2	4		
2				4		
3				2	1	
4					2	1
5						

- Para encontrar um ciclo de Hamilton, caso exista, devemos visitar os vértices do grafo de outras maneiras.
- A rigor, o melhor algoritmo conhecido resolve o problema tentando todos os caminhos possíveis.

Ciclo de Hamilton - Tentando Todas as Possibilidades

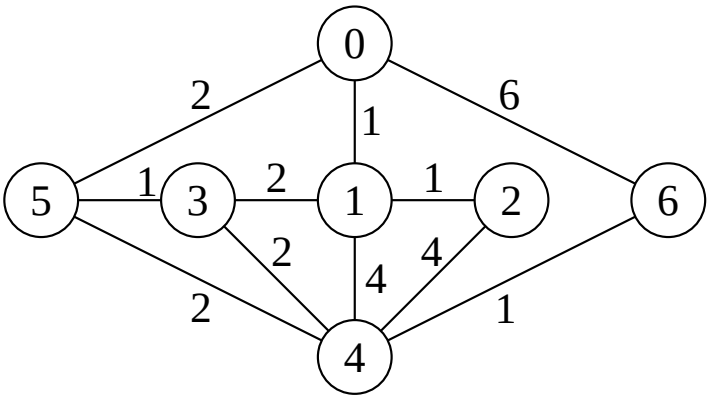
- Para tentar todas as possibilidades, vamos alterar o método Visita.
- Desmarca o vértice já visitado no caminho anterior e permite que seja visitado novamente em outra tentativa.

```
private int visita (int u, int tempo) {
  this.d[u] = ++tempo;
  if (!this.grafo.listaAdjVazia (u)) {
    Grafo.Aresta a = this.grafo.primeiroListaAdj (u);
    while (a != null) {
      int v = a.v2 ();
      if (this.d[v] == 0) tempo = this.visita (v, tempo);
      a = this.grafo.proxAdj (u);}
  }
  tempo--; this.d[u] = 0;
  return tempo;
}
```

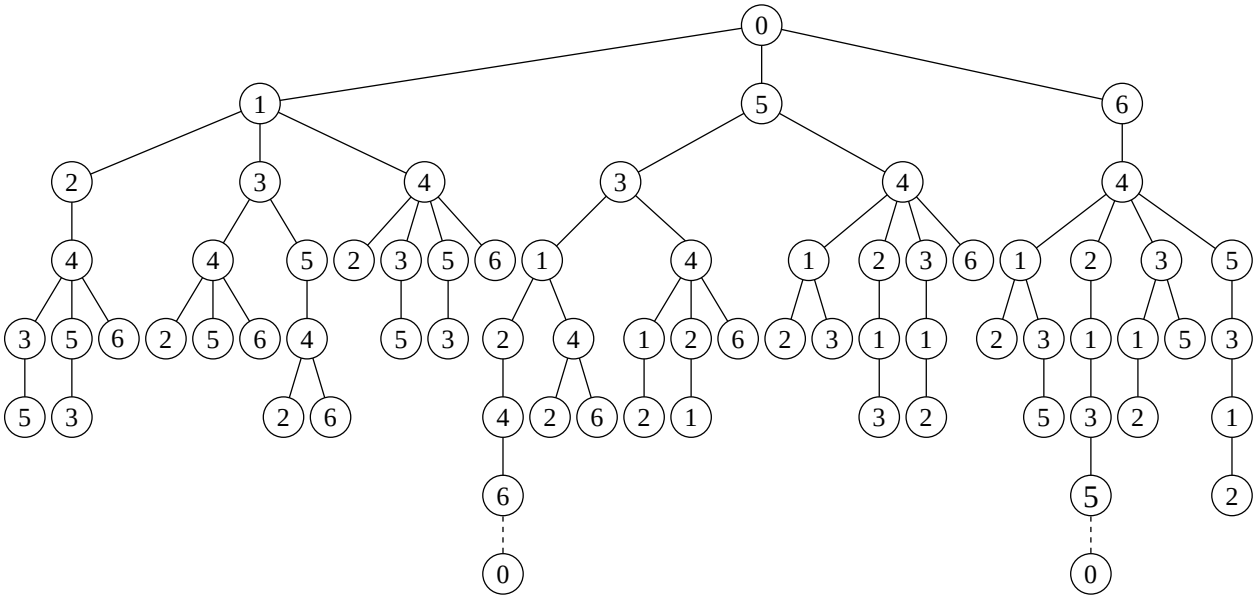
- O custo é proporcional ao número de chamadas para o método Visita.
- Para um grafo completo, (arestas ligando todos os pares de nós) existem $N!$ ciclos simples. Custo é proibitivo.

Ciclo de Hamilton - Tentando Todas as Possibilidades

- Para o grafo



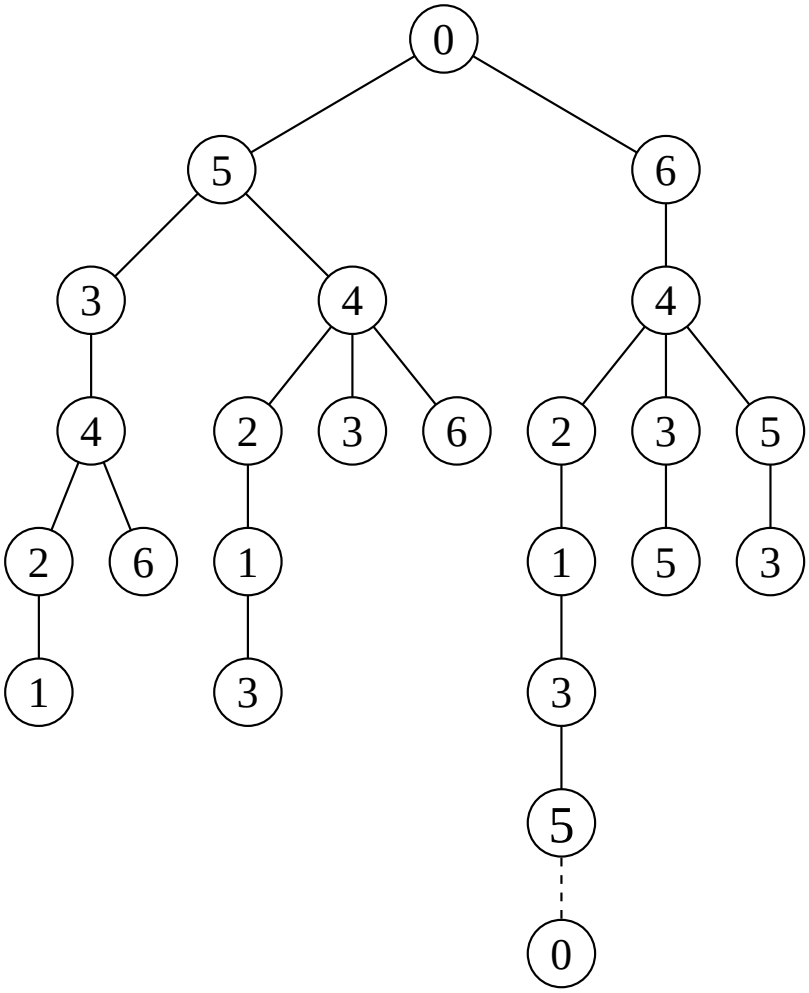
A árvore de caminhamento é:



- Existem duas respostas: 0 5 3 1 2 4 6 0 e 0 6 4 2 1 3 5 0.

Ciclo de Hamilton - Tentativa e Erro com Poda

- Diminuir número de chamadas a Visita fazendo "**poda**" na árvore de caminhamento.
- No exemplo anterior, cada ciclo é obtido duas vezes, caminhando em ambas as direções.
- Insistindo que o nó 2 apareça antes do 0 e do 1, não precisamos chamar Visita para o nó 1 a não ser que o nó 2 já esteja no caminho.
- Árvore de caminhamento obtida:



Ciclo de Hamilton - Tentativa e Erro com Poda

- Entretanto, esta técnica não é sempre possível de ser aplicada.
- Suponha que se queira um caminho de custo mínimo que não seja um ciclo e passe por todos os vértices: 0 6 4 5 3 1 2 é solução.
- Neste caso, a técnica de eliminar simetrias não funciona porque não sabemos *a priori* se um caminho leva a um ciclo ou não.

Ciclo de Hamilton - *Branch-and-Bound*

- Outra saída para tentar diminuir o número de chamadas a Visita é por meio da técnica de ***branch-and-bound***.
- A ideia é cortar a pesquisa tão logo se saiba que não levará a uma solução.
- Corta chamadas a Visita tão logo se chegue a um custo para qualquer caminho que seja maior que um caminho solução já obtido.
- Exemplo: encontrando 0 5 3 1 2 4 6, de custo 11, não faz sentido continuar no caminho 0 6 4 1, de custo 11 também.
- Neste caso, podemos evitar chamadas a Visita se o custo do caminho corrente for maior ou igual ao melhor caminho obtido até o momento.

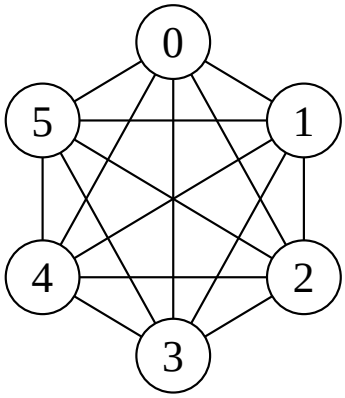
Heurísticas para Problemas $\mathcal{NP}$-Completo

- **Heurística**: algoritmo que pode produzir um bom resultado (ou até a solução ótima), mas pode também não obter solução ou obter uma distante da ótima.
- Uma heurística pode ser determinista ou probabilística.
- Pode haver instâncias em que uma heurística (probabilística ou não) nunca vai encontrar uma solução.
- A principal diferença entre uma heurística probabilística e um **algoritmo Monte Carlo** é que o algoritmo Monte Carlo tem que encontrar uma solução correta com uma certa probabilidade (de preferência alta) para qualquer instância do problema.

Heurística para o PCV

- Algoritmo do vizinho mais próximo, heurística gulosa simples:
  1. Inicie com um vértice arbitrário.
  2. Procure o vértice mais próximo do último vértice adicionado que não esteja no caminho e adicione ao caminho a aresta que liga esses dois vértices.
  3. Quando todos os vértices estiverem no caminho, adicione uma aresta conectando o vértice inicial e o último vértice adicionado.
- Complexidade: $O(n^2)$, sendo n o número de cidades, ou $O(d)$, sendo d o conjunto de distâncias entre cidades.
- Aspecto negativo: embora todas as arestas escolhidas sejam localmente mínimas, a aresta final pode ser bastante longa.

Heurística para o PCV



	1	2	3	4	5
0	3	10	11	7	25
1		8	12	9	26
2			9	4	20
3				5	15
4					18

- Caminho ótimo para esta instância: 0 1 2 5 3 4 0 (comprimento 58).
- Para a heurística do vizinho mais próximo, se iniciarmos pelo vértice 0, o vértice mais próximo é o 1 com distância 3.
- A partir do 1, o mais próximo é o 2, a partir do 2 o mais próximo é o 4, a partir do 4 o mais próximo é o 3, a partir do 3 restam o 5 e o 0.
- O comprimento do caminho 0 1 2 4 3 5 0 é 60.

Heurística para o PCV

- Embora o algoritmo do vizinho mais próximo não encontre a solução ótima, a obtida está bem próxima do ótimo.
- Entretanto, é possível encontrar instâncias em que a solução obtida pode ser muito ruim.
- Pode mesmo ser arbitrariamente ruim, uma vez que a aresta final pode ser muito longa.
- É possível achar um algoritmo que garanta encontrar uma solução que seja razoavelmente boa no pior caso, desde que a classe de instâncias consideradas seja restrita.

Algoritmos Aproximados para Problemas $\mathcal{NP}$-Completo

- Para projetar algoritmos polinomiais para "resolver" um problema de otimização $\mathcal{NP}$-completo é necessário "relaxar" o significado de resolver.
- Removemos a exigência de que o algoritmo tenha sempre de obter a solução ótima.
- Procuramos algoritmos eficientes que não garantem obter a solução ótima, mas sempre obtêm uma próxima da ótima.
- Tal solução, com valor próximo da ótima, é chamada de solução aproximada.
- Um **algoritmo aproximado** para um problema Π é um algoritmo que gera **soluções aproximadas** para Π.
- Para ser útil, é importante obter um limite para a razão entre a solução ótima e a produzida pelo algoritmo aproximado.

Medindo a Qualidade da Aproximação

- O comportamento de algoritmos aproximados quanto Ă qualidade dos resultados (não o tempo para obtê-los) tem de ser monitorado.
- Seja I uma instância de um problema Π e seja $S^*(I)$ o valor da solução ótima para I.
- Um algoritmo aproximado gera uma solução possível para I cujo valor $S(I)$ é maior (pior) do que o valor ótimo $S^*(I)$.
- Dependendo do problema, a solução a ser obtida pode minimizar ou maximizar $S(I)$.
- Para o PCV, podemos estar interessados em um algoritmo aproximado que minimize $S(I)$: obtém o valor mais próximo possível de $S^*(I)$.
- No caso de o algoritmo aproximado obter a solução ótima, então $S(I) = S^*(I)$.

Algoritmos Aproximados - Definição

- Um algoritmo aproximado para um problema Π é um algoritmo polinomial que produz uma solução $S(I)$ para uma instância I de Π.
- O comportamento do algoritmo A é descrito pela **razão de aproximação**

$$R_A(I) = \frac{S(I)}{S^*(I)},$$

  que representa um problema de minimização
- No caso de um problema de maximização, a razão é invertida.
- Em ambos os casos, $R_A(I) \geq 1$.

Algoritmos Aproximados para o PCV

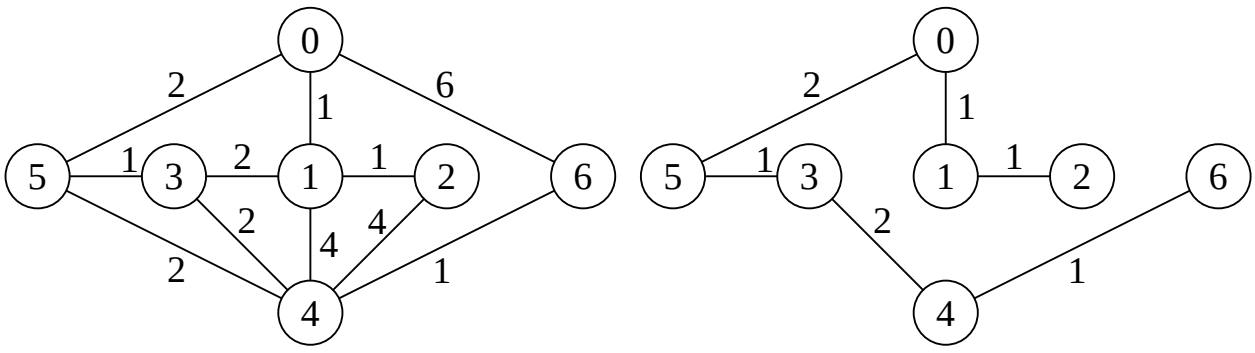
- Seja $G = (V, A)$ um grafo não direcionado, completo, especificado por um par (N, d).
- N é o conjunto de vértices do grafo (cidades), e d é uma função distância que mapeia as arestas em números reais, em que d satisfaz:
  1. $d(i, j) = d(j, i)\ \forall i, j \in N$,
  2. $d(i, j) > 0\ \forall i, j \in N$,
  3. $d(i, j) + d(j, k) \geq d(i, k)\ \forall i, j, k \in N$
- 1$^{\underline{a}}$ propriedade: a distância da cidade i até outra adjacente j é igual à de j até i.
- Quando isso não acontece, temos o problema conhecido como **PCV Assimétrico**
- 2$^{\underline{a}}$ propriedade: apenas distâncias positivas.
- 3$^{\underline{a}}$ propriedade: **desigualdade triangular**. A distância de i até j somada com a de j até k deve ser maior do que a distância de i até k.

Algoritmos Aproximados para o PCV

- Quando o problema exige distâncias não restritas à desigualdade triangular, basta adicionar uma constante k a cada distância.
- Exemplo: as três distâncias envolvidas são 2, 3 e 10, que não obedecem à desigualdade triangular pois $2 + 3 < 10$. Adicionando $k = 10$ às três distâncias obtendo 12, 13 e 20, que agora satisfazem a desigualdade triangular.
- O problema alterado terá a mesma solução ótima que o problema anterior, apenas com o comprimento da rota ótima diferindo de $n \times k$.
- Cabe observar que o PCV equivale a encontrar no grafo $G = (V, A)$ um **ciclo de Hamilton** de custo mínimo.

Árvore Geradora Mínima (AGM)

- Considere um grafo $G = (V, A)$, sendo V as n cidades e A as distâncias entre cidades.
- Uma árvore geradora é uma coleção de $n - 1$ arestas que ligam todas as cidades por meio de um subgrafo conectado único.
- A **árvore geradora mínima** é a árvore geradora de custo mínimo.
- Existem algoritmos polinomiais de custo $O(|A| \log |V|)$ para obter a árvore geradora mínima quando o grafo de entrada é dado na forma de uma matriz de adjacência.
- Grafo e árvore geradora mínima correspondente:



Limite Inferior para a Solução do PCV a Partir da AGM

- A partir da AGM, podemos derivar o limite inferior para o PCV.
- Considere uma aresta (x_1, x_2) do caminho ótimo do PCV. Remova a aresta e ache um caminho iniciando em x_1 e terminando em x_2.
- Ao retirar uma aresta do caminho ótimo, temos uma árvore geradora que consiste de um caminho que visita todas as cidades.
- Logo, o caminho ótimo para o PCV é necessariamente maior do que o comprimento da AGM.
- O **limite inferior** para o custo deste caminho é a AGM.
- Logo, $Otimo_{PCV} > AGM$.

Limite Superior de Aproximação para o PCV

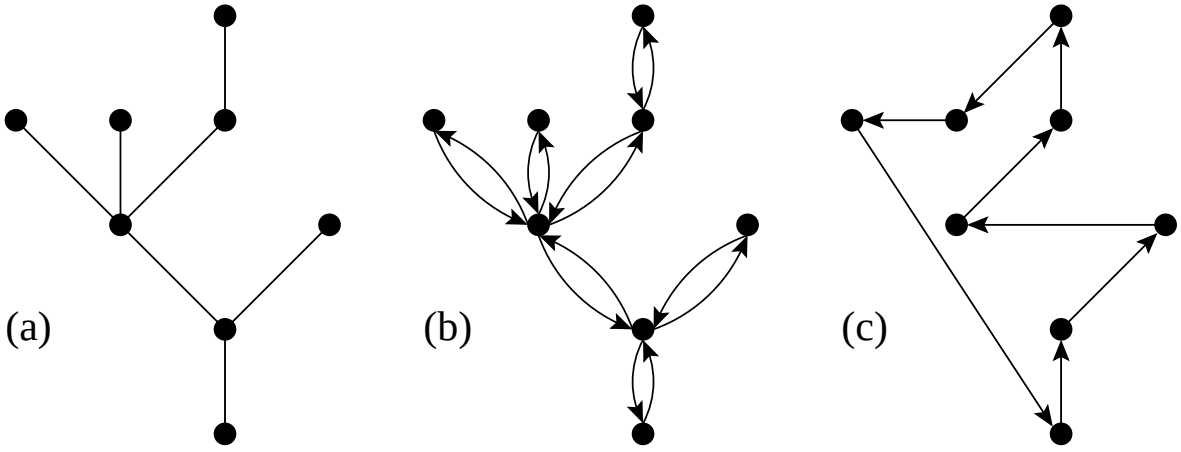
- A desigualdade triangular permite utilizar a AGM para obter um **limite superior** para a razão de aproximação com relação ao comprimento do caminho ótimo.
- Vamos considerar um algoritmo que visita todas as cidades, mas pode usar somente as arestas da AGM.
- Uma possibilidade é iniciar em um vértice folha e usar a seguinte estratégia:
  - Se houver aresta ainda não visitada saindo do vértice corrente, siga essa aresta para um novo vértice.
  - Se todas as arestas a partir do vértice corrente tiverem sido visitadas, volte para o vértice adjacente pela aresta pela qual o vértice corrente foi inicialmente alcançado.
  - Termine quando retornar ao vértice inicial.

Limite Superior de Aproximação para o PCV - Busca em Profundidade

- O algoritmo descrito anteriormente é a Busca em Profundidade aplicada à AGM.
- Verifica-se que:
  - o algoritmo visita todos os vértices.
  - nenhuma aresta é visitada mais do que duas vezes.
- Obtém um caminho que visita todas as cidades cujo custo é menor ou igual a duas vezes o custo da árvore geradora mínima.
- Como o caminho ótimo é maior do que o custo da AGM, então o caminho obtido é no máximo duas vezes o custo do caminho ótimo. $Caminho_{PCV} < 2Otimo_{PCV}$.
- Restrição: algumas cidades são visitadas mais de uma vez.
- Para contornar o problema, usamos a desigualdade triangular.

Limite Superior de Aproximação para o PCV - Desigualdade Triangular

- Introduzimos curto-circuitos que nunca aumentam o comprimento total do caminho.
- Inicie em uma folha da AGM, mas sempre que a busca em profundidade for voltar para uma cidade já visitada, salte para a próxima ainda não visitada.
- A rota direta não é maior do que a anterior indireta, em razão da desigualdade triangular.
- Se todas as cidades tiverem sido visitadas, volte para o ponto de partida.



- O algoritmo constrói um caminho solução para o PCV porque cada cidade é visitada apenas uma vez, exceto a cidade de partida.

Limite Superior de Aproximação para o PCV - Desigualdade Triangular

- O caminho obtido não é maior que o caminho obtido em uma busca em profundidade, cujo comprimento é no máximo duas vezes o do caminho ótimo.
- Os principais passos do algoritmo são:
  1. Obtenha a árvore geradora mínima para o conjunto de n cidades, com custo $O(n^2)$.
  2. Aplique a busca em profundidade na AGM obtida com custo $O(n)$, a saber:
     - Inicie em uma folha (grau 1).
     - Siga uma aresta não utilizada.
     - Se for retornar para uma cidade já visitada, salte para a próxima ainda não visitada (rota direta menor que a indireta pela desigualdade triangular).
     - Se todas as cidades tiverem sido visitadas, volte à cidade de origem.
- Assim, obtivemos um algoritmo polinomial de custo $O(n^2)$, com uma razão de aproximação garantida para o pior caso de $R_A \leq 2$.

Como Melhorar o Limite Superior a Partir da AGM

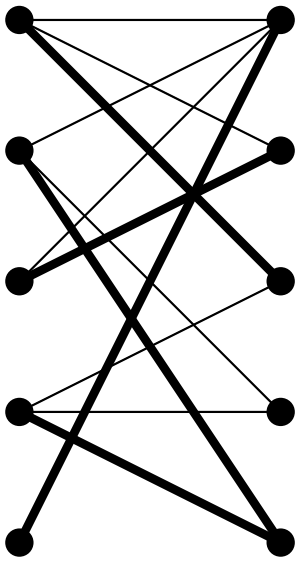
- No algoritmo anterior um caminho para o caixeiro-viajante pode ser obtido dobrando os arcos da AGM, o que leva a um pior caso para a razão de aproximação no máximo igual a 2.
- Melhora-se a garantia de um fator 2 para o pior caso, utilizando o conceito de grafo Euleriano.
- Um **grafo Euleriano** é um grafo conectado no qual todo vértice tem grau par.
- Um grafo Euleriano possui um **caminho Euleriano**, um ciclo que passa por todas as arestas exatamente uma vez.
- O caminho Euleriano em um grafo Euleriano, pode ser obtido em tempo $O(n)$, usando a busca em profundidade.
- Podemos obter um caminho para o PCV a partir de uma AGM, usando o caminho Euleriano e a técnica de curto-circuito.

Como Melhorar o Limite Superior a Partir da AGM

- Passos do algoritmo:
  - Suponha uma AGM que tenha cidades do PCV como vértices.
  - Dobre suas arestas para obter um grafo Euleriano.
  - Encontre um caminho Euleriano para esse grafo.
  - Converta-o em um caminho do caixeiro-viajante usando curto-circuitos.
- Pela desigualdade triangular, o caminho do caixeiro-viajante não pode ser mais longo do que o caminho Euleriano e, conseqüentemente, de comprimento no máximo duas vezes o comprimento da AGM.

Casamento Mínimo com Pesos

- Christophides propôs uma melhoria no algoritmo anterior utilizando o conceito de **casamento mínimo com pesos** em grafos.
- Dado um conjunto contendo um número par de cidades, um casamento é uma coleção de arestas M tal que cada cidade é a extremidade de exatamente um arco em M.
- Um casamento mínimo é aquele para o qual o comprimento total das arestas é mínimo.

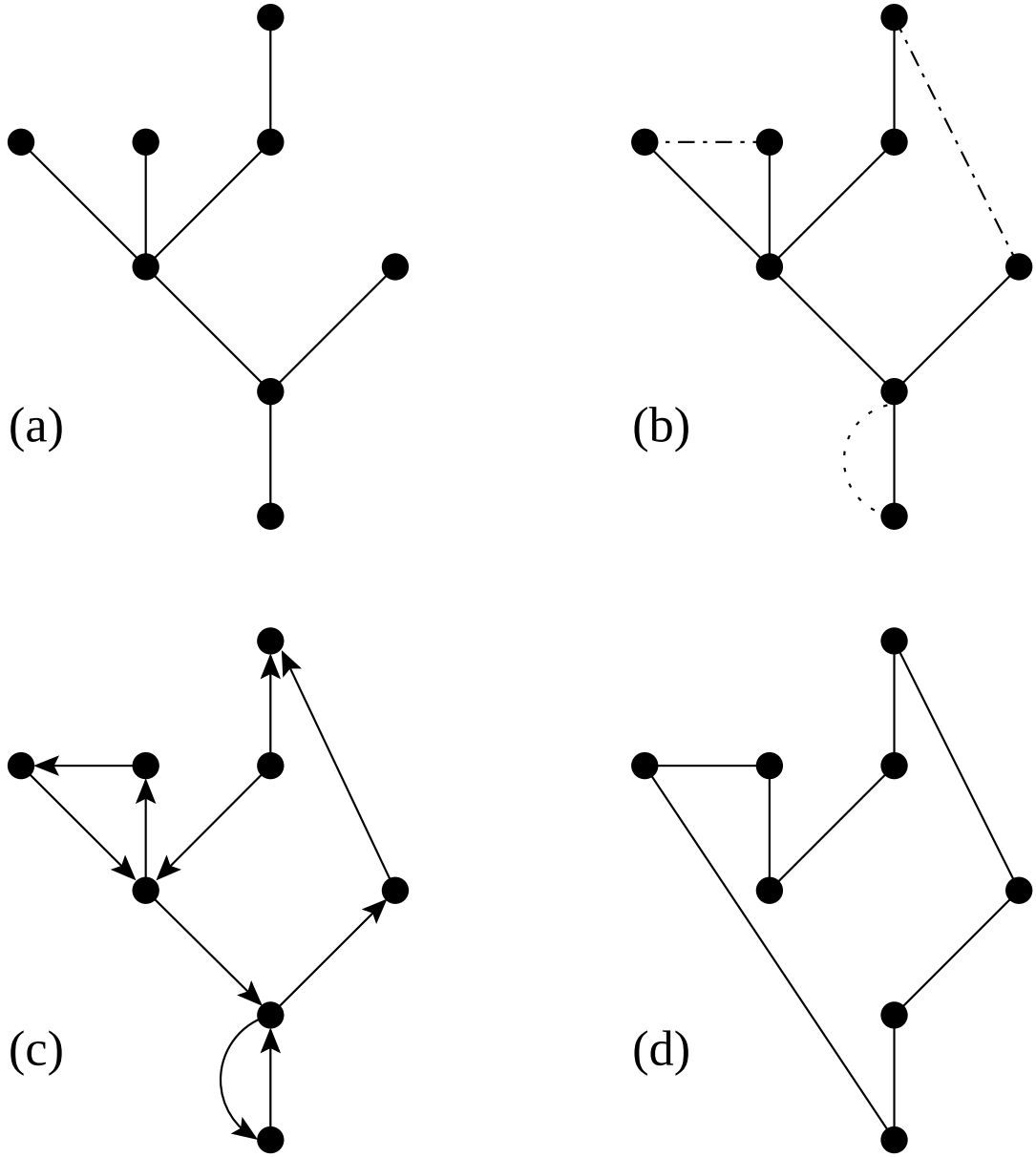


- Todo vértice é parte de exatamente uma aresta do conjunto M.
- Pode ser encontrado com custo $O(n^3)$.

Casamento Mínimo com Pesos

- Considere a AGM T de um grafo.
- Alguns vértices em T já possuem grau par, assim não precisariam receber mais arestas se quisermos transformar a árvore em um grafo Euleriano.
- Os únicos vértices com que temos de nos preocupar são os vértices de grau ímpar.
- Existe sempre um número par de vértices de grau ímpar, desde que a soma dos graus de todos os vértices tenha de ser par porque cada aresta é contada exatamente uma vez.
- Uma maneira de construir um grafo Euleriano que inclua T é simplesmente obter um casamento para os vértices de grau ímpar.
- Isto aumenta de um o grau de cada vértice de grau ímpar. Os de de grau par não mudam.
- Se adicionamos em T um casamento mínimo para os vértices de grau ímpar, obtemos um grafo Euleriano que tem comprimento mínimo dentre aqueles que contêm T.

Casamento Mínimo com Pesos - Exemplo



a. Uma árvore geradora mínima T.

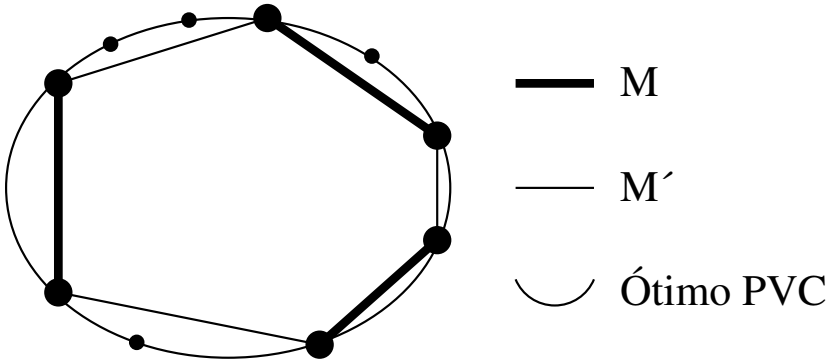
b. T mais um casamento mínimo dos vértices de grau ímpar.

c. Caminho de Euler em (b).

d. Busca em profundidade com curto-circuito.

Casamento Mínimo com Pesos

- Basta agora determinar o comprimento do grafo de Euler.
- Caminho do caixeiro-viajante em que podem ser vistas seis cidades correspondentes aos vértices de grau ímpar enfatizadas.



- O caminho determina os casamentos M e M'.

Casamento Mínimo com Pesos

- Seja I uma instância do PCV, e $Comp(T)$, $Comp(M)$ e $Comp(M')$, respectivamente, a soma dos comprimentos de T, M e M'.
- Pela desigualdade triangular devemos ter que: $Comp(M) + Comp(M') \leq Otimo(I)$,
- Assim, ou M ou M' têm de ter comprimento menor ou igual a $Otimo(I)/2$.
- Logo, o comprimento de um casamento mínimo para os vértices de grau ímpar de T tem também de ter comprimento no máximo $Otimo(I)/2$.
- Desde que o comprimento de M é menor do que o caminho do caixeiro-viajante ótimo, podemos concluir que o comprimento do grafo Euleriano construído é:

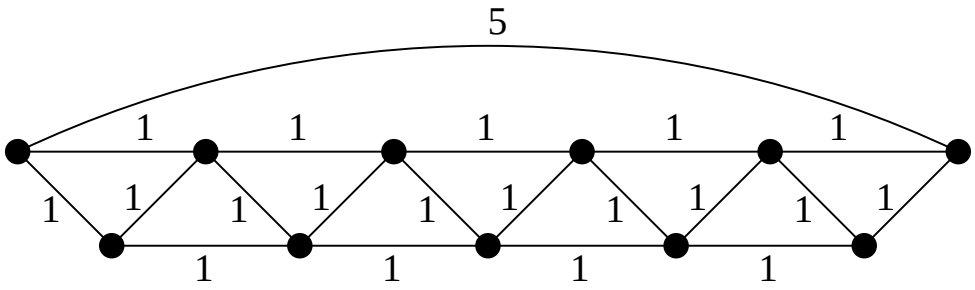
$$Comp(I) < \frac{3}{2} Otimo(I).$$

Casamento Mínimo com Pesos - Algoritmo de Christophides

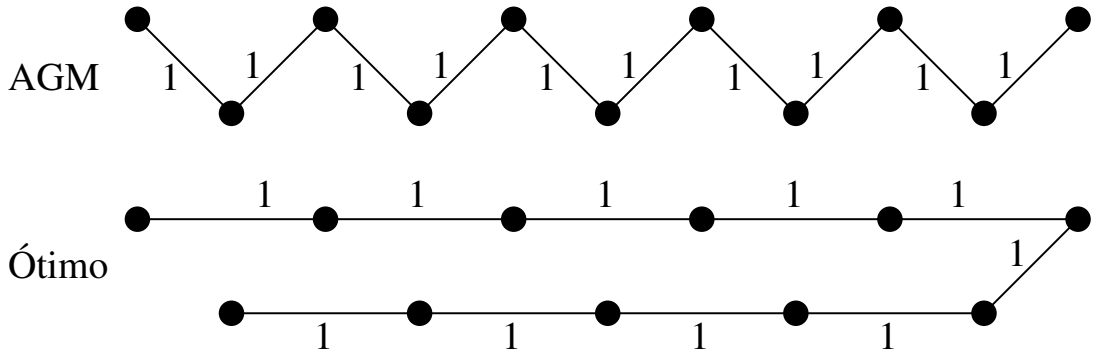
- Os principais passos do algoritmo de Christophides são:
  1. Obtenha a AGM T para o conjunto de n cidades, com custo $O(n^2)$.
  2. Construa um casamento mínimo M para o conjunto de vértices de grau ímpar em T, com custo $O(n^3)$.
  3. Encontre um caminho de Euler para o grafo Euleriano obtido com a união de T e M, e converta o caminho de Euler em um caminho do caixeiro-viajante usando curto-circuitos, com um custo de $O(N)$.
- Assim obtivemos um algoritmo polinomial de custo $O(n^3)$, com uma razão de aproximação garantida para o pior caso de $R_A < 3/2$.

Algoritmo de Christophides - Pior Caso

- Exemplo de pior caso do algoritmo de Christofides:



- A AGM e o caminho ótimo são:



- Neste caso, para uma instância I:

$$C(I) = \frac{3}{2}[Otimo(I) - 1],$$

em que o $Otimo(I) = 11$, $C(I) = 15$, e $AGM = 10$.