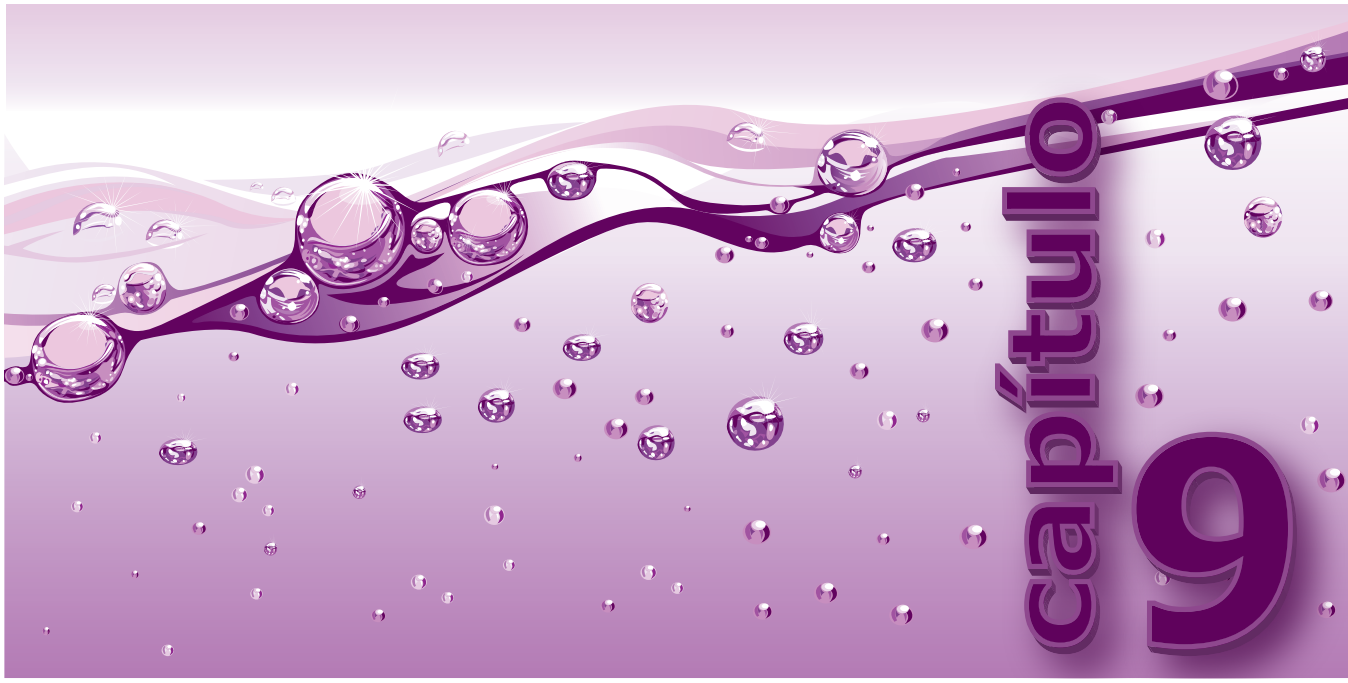


Casamento de Strings e Tries

Após estudar este capítulo, você deverá ser capaz de:

- Definir os seguintes conceitos no contexto de casamento de strings:
  - ❒ Padrão
  - ❒ Texto
  - ❒ Janela de texto
  - ❒ Alfabeto
  - ❒ Substring
  - ❒ Borda
  - ❒ Salto
  - ❒ FB
  - ❒ KMP
  - ❒ BM
  - ❒ BMH
  - ❒ KR
  - ❒ Trie
  - ❒ Léxico
  - ❒ Prefixo e prefixo próprio
  - ❒ Sufixo e sufixo próprio
  - ❒ Algoritmo de força bruta
  - ❒ Retrocesso
  - ❒ Fluxo contínuo
  - ❒ Casamento de palavras
- Analisar, usando notação θ, o pior e o melhor casos de um algoritmo de casamento de strings
- Evitar ocorrência de overflow numa implementação do algoritmo de Karp e Rabin
- Implementar os algoritmos FB, KMP e BMH
- Discutir diferenças e semelhanças entre os algoritmos FB, KMP, BM, BMH e KR
- Diferenciar nó final e nó-folha de uma trie
- Explicar as operações de busca, inserção e remoção em tries
- Implementar uma trie simples
- Justificar os custos temporais de operações com tries
- Discutir vantagens e desvantagens do uso de tries com relação a outras estruturas de dados usadas em implementações de tabelas de busca
- Representar prefixos usando uma trie

objetivos

CASAMENTO DE STRINGS é uma operação fundamental em programação. Cada vez que um usuário efetua uma operação de busca num documento de texto, banco de dados ou mecanismo de busca da internet, essa operação recebe o suporte de um algoritmo de casamento de strings. Outras aplicações comuns de casamento de strings incluem processamento de linguagem natural, bioinformática, checagem de *spam* e varredura antivírus.

Dado um string (**texto**) de comprimento n e outro string de comprimento m, denominado **padrão**, casamento de strings consiste em verificar se o padrão ocorre no texto ou, mais precisamente, se o padrão é um substring do texto. Tipicamente, o valor de n é muito maior do que m.

Existem inúmeros algoritmos de casamento de strings com texto. Neste capítulo, cinco deles serão discutidos:

- Algoritmo de casamento por força bruta — abreviadamente, algoritmo **FB** (v. **Seção 9.2**)
- Algoritmo de Knuth, Morris e Pratt — abreviadamente, algoritmo **KMP** (v. **Seção 9.3**)
- Algoritmo de Boyer e Moore — abreviadamente, algoritmo **BM** (v. **Seção 9.4**)
- Algoritmo de Boyer e Moore e Horspool — abreviadamente, algoritmo **BMH** (v. **Seção 9.5**)
- Algoritmo de Karp e Rabin — abreviadamente, algoritmo **KR** (v. **Seção 9.6**)

O problema de casamento de strings pode ser visto como um problema de busca no qual o padrão é a chave de busca, mas as estruturas de dados e os algoritmos para busca vistos até aqui não são convenientes para lidar com esse tipo específico de problema.

Todos os algoritmos de casamento de strings discutidos aqui dedicam-se a encontrar a primeira ocorrência de um padrão num texto, mas eles podem ser facilmente estendidos para encontrar todas as ocorrências de um determinado padrão num texto.

Neste capítulo, a estrutura de dados **trie**, usada principalmente para representar strings, é examinada na **Seção 9.8**.

9.1 Conceitos

9.1.1 Terminologia

Casamento de strings (ou **de padrões**) consiste em verificar se um string faz parte de outro string; ou, em outras palavras, se um string é **substring** de outro. O string que se tenta verificar se é substring de outro (i.e., o string menor) é denominado **padrão** neste contexto e representado pela letra P, enquanto o string maior é denominado **texto** e representado por T. O comprimento do string menor (o padrão) será sempre representado por m e o comprimento do string maior será sempre n. O termo *texto* é usado neste contexto mesmo que o string assim denominado não constitua um texto no sentido convencional.

Um algoritmo de casamento de strings retorna o índice que indica a posição da primeira ocorrência do padrão no texto quando ocorre casamento (i.e., quando o padrão realmente ocorre no texto). Caso contrário, o algoritmo retorna um valor (normalmente, -1) que indica a ausência do padrão no texto.

No presente contexto, **alfabeto** consiste no conjunto dos símbolos do qual são derivados os componentes dos strings sob discussão. Esses símbolos continuarão sendo chamados **caracteres** (como é usual) e alfabetos serão representados por $\sum$ (letra grega sigma maiúscula). Exemplos de alfabetos são:

- $\sum = \{0, 1\}$ — alfabeto binário
- $\sum = \{A, C, G, T\}$ — alfabeto genético
- $\sum = \{a, b, \ldots, z\}$ — alfabeto romano (ou latino)

O **tamanho de um alfabeto** $\sum$ é representado por $|\sum|$, de modo que, por exemplo, se $\sum = \{0, 1\}$, $|\sum| = 2$. De modo semelhante, o tamanho de um string s será representado por $|s|$.

Um **substring** *s[i..j]* de um string *s* é o string constituído pelos caracteres de *s* que se encontram entre os índices (posições) *i* e *j*, incluindo os caracteres nessas posições. A indexação de strings começa com zero (como é usual em C). Por exemplo, no string *s* `"CCCAAACTGCTTTAATCAAAAAC"`, o string `"GCTTT"` é um substring de *s* que se encontra entre os índices *8* e *12* e, assim, ele pode ser representado por *s[8..12]*.

Um string s_1 é **prefixo** de outro string *s* se ele é um substring de *s* que ocorre entre os índices *0* e *j*, sendo $j < |s|$. Um string s_1 é **prefixo próprio** de outro string *s* se ele é prefixo de *s* e $s_1 \neq s$. Note que, de acordo com essa definição, qualquer string é prefixo de si próprio, mas um string não pode ser prefixo *próprio* de si mesmo.

Um string s_1 é **sufixo** de outro string *s* se ele é um substring de *s* que ocorre entre os índices *i* e $|s| - 1$, sendo $i \geq 0$. Um string s_1 é **sufixo próprio** de outro string *s* se ele é sufixo de *s* e $s_1 \neq s$.

Uma **borda** de um string é um substring que tanto é prefixo próprio quanto sufixo próprio desse string. Por exemplo, o substring `"ab"` é uma borda do string `"abcab"`. Um string pode ter mais de uma borda. Por exemplo, o string `"aaaa"` possui três bordas: `"a"`, `"aa"` e `"aaa"` (mas `"aaaa"` não é uma borda desse string). Os conceitos de prefixos e sufixos próprios e de borda são ilustrados na **Figura 9–1**.

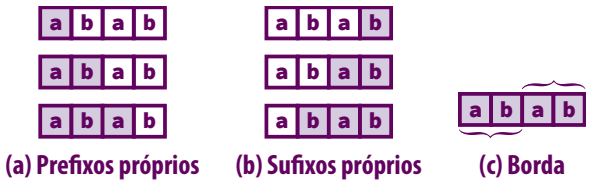


Figura 9–1: Prefixos e Sufixos Próprios e Borda

Convencionalmente, quando um string não possui borda, diz-se que o tamanho de sua borda é zero.

9.1.2 Visualização

Um artifício que facilita o entendimento de algoritmos de casamento de strings consiste em imaginar que enquanto um casamento não é encontrado ou o final do texto não é atingido o padrão **desliza** paralelamente ao texto no sentido do início para o final do texto, como é ilustrado na **Figura 9–2**.

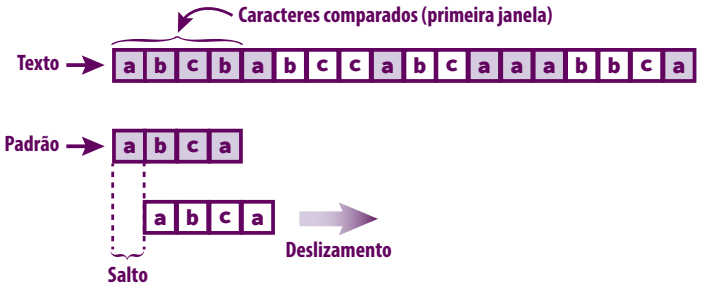


Figura 9–2: Deslizamento de Padrão com Relação a Texto

Cada deslizamento do padrão corresponde a pelo menos um caractere no texto e é denominado **salto**. O número de caracteres saltados antes do início de uma nova comparação depende do algoritmo em questão. Por exemplo, o tamanho do salto nos algoritmos de força bruta (v. **Seção 9.2**) e de Karp e Rabin (v. **Seção 9.6**) é igual a *1*. Por outro lado, os saltos efetuados de acordo com os algoritmos de Knuth, Morris e Pratt (v. **Seção 9.3**) e de Boyer e Moore (v. **Seção 9.4**) podem ser maiores do que *1*.

O substring do texto que se encontra alinhado com o padrão é denominado **janela de texto** (ou apenas **janela**). Durante uma operação de casamento de strings, a janela de texto move-se para a direita. Por exemplo, na **Figura 9–2**, inicialmente, a janela de texto é o substring `"abcb"`; após o deslizamento mostrado nessa figura, a janela de texto passa a ser `"bcba"`. Em qualquer caso, o tamanho da janela de texto é igual ao tamanho do padrão.

Em qualquer algoritmo de casamento, a primeira janela é obtida alinhando-se o primeiro caractere do padrão com o primeiro caractere do texto (v. **Figura 9–2**).

9.1.3 Casamento de Strings com Retrocesso

Um algoritmo de casamento de strings atua com **retrocesso** quando o próximo caractere do texto a ser comparado pode estar numa posição anterior àquela do último caractere do texto que foi comparado. Um exemplo de algoritmo dessa natureza, ilustrado na **Figura 9–3**, é o algoritmo de casamento por força bruta (v. **Seção 9.2**). Por outro lado, num algoritmo de casamento de strings que não apresenta retrocesso, o índice do próximo caractere do texto a ser comparado é sempre maior do que ou igual ao índice do último caractere do texto comparado, como mostra a **Figura 9–4**. O algoritmo de Knuth, Morris e Pratt, apresentado na **Seção 9.3**, é um exemplo de algoritmo sem retrocesso.

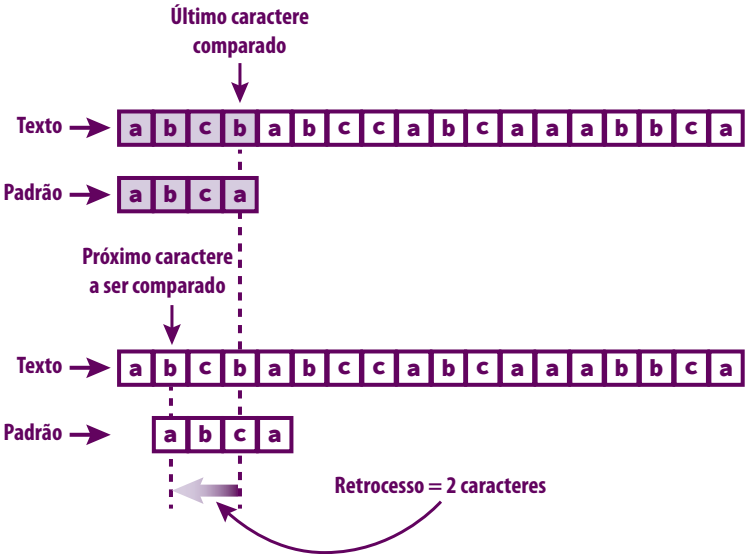


Figura 9–3: Casamento de Strings com Retrocesso

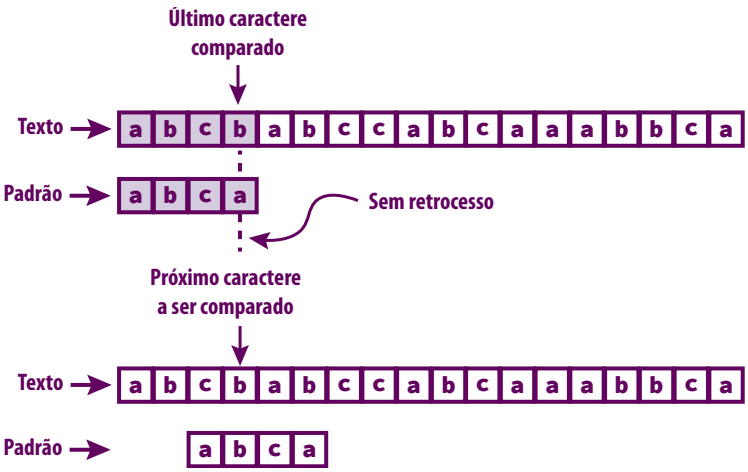


Figura 9–4: Casamento de Strings sem Retrocesso

O fato de um algoritmo de casamento de strings apresentar ou não retrocesso é importante porque, às vezes, não há espaço ou tempo disponível para armazenamento do texto com o qual se procura casar um padrão, conforme será visto na **Seção 9.10.7**.

9.2 Casamento de Strings por Força Bruta (FB)

9.2.1 Visão Geral

O algoritmo mais óbvio de casamento de strings consiste em checar, para cada posição possível no texto, se o padrão se encontra nessa posição. Esse raciocínio dá origem ao **algoritmo de casamento por força bruta** (ou **algoritmo FB**, de modo abreviado), que recebe essa denominação porque, ao contrário dos demais algoritmos de casamento, ele não usa qualquer informação que possa ser obtida sobre o texto ou o padrão. Esse algoritmo é o mais fácil de ser implementado, mas, em compensação, também é o mais ineficiente.

Usando a primeira janela de texto, o algoritmo **FB** compara sequencialmente os respectivos caracteres no texto e no padrão até que (1) ocorra uma discordância entre caracteres ou (2) todos os caracteres tenham sido comparados sem haver discordância. No primeiro caso, passa-se para a próxima janela, que se encontra um caractere adiante, e repete-se o procedimento. Assim o algoritmo encerra em duas situações:

[1] É encontrado um casamento. Isso ocorre quando, durante uma comparação de caracteres numa janela, não aparece nenhuma discordância entre caracteres.

[2] Não há mais possibilidade de sucesso. Ou seja, não há mais janela de texto a ser comparada.

A **Figura 9–5** ilustra um exemplo de casamento utilizando o algoritmo **FB**. Retângulos em coloridos representam caracteres que são comparados. Nesse exemplo ocorrem *24* comparações antes que o padrão seja encontrado no texto.

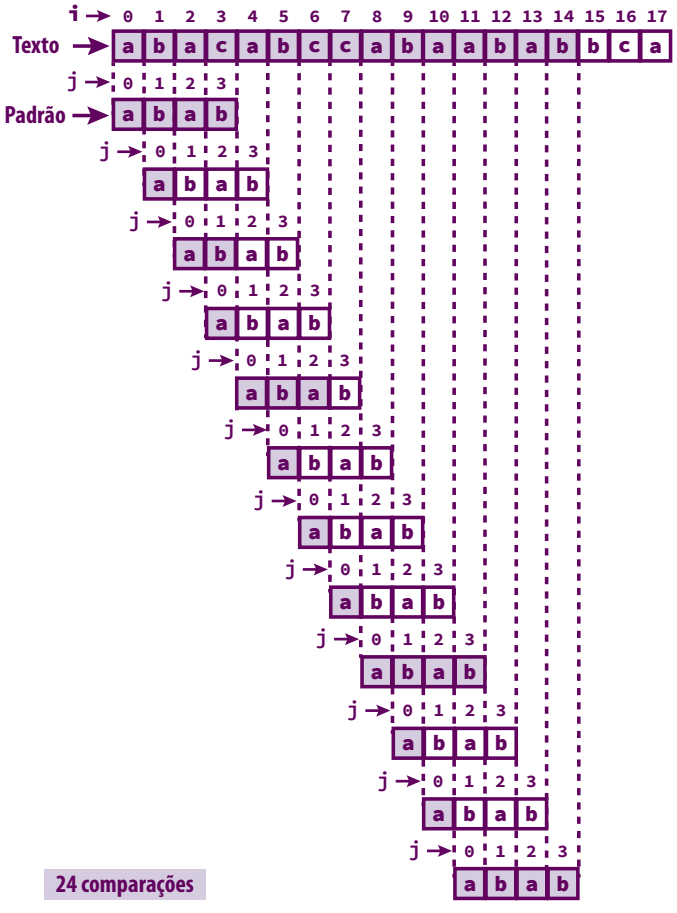


Figura 9–5: Exemplo de Casamento de Strings por Força Bruta

9.2.2 Algoritmo

A **Figura 9–6** apresenta o algoritmo de casamento por força bruta.

ALGORITMO FB

ENTRADA: O texto *t* de comprimento *n* e o padrão *p* a ser procurado de comprimento *m*

SAÍDA: A posição no texto da primeira ocorrência do padrão, se ele for encontrado no texto, ou um valor negativo, em caso contrário

1. Atribua *0* a *i*
2. Atribua *0* a *j*
3. Enquanto $i < n$ e $j < m$, faça:
   - 3.1 Enquanto os caracteres no índice *i* do texto e no índice *j* do padrão forem diferentes, faça:
     - 3.1.1 Atribua a *i* o índice do próximo caractere a ser comparado no texto
     - 3.1.2 Atribua a *j* o índice do primeiro caractere do padrão
4. Se $j = m$, retorne a posição do padrão no texto
5. Caso contrário, retorne um valor negativo

FIGURA 9–6: ALGORITMO DE CASAMENTO POR FORÇA BRUTA

9.2.3 Implementação

A função `CasamentoFB()` a seguir implementa o algoritmo de casamento de strings por força bruta. Essa função usa os seguintes parâmetros:

- `t` (entrada) — string que representa o texto
- `p` (entrada) — string que representa o padrão

A função `CasamentoFB()` retorna a posição no texto da primeira ocorrência do padrão, se ele for encontrado no texto, ou `-1`, em caso contrário.

```
int CasamentoFB(const char *t, const char *p)
{
   int n = strlen(t), /* Comprimento do texto  */
       m = strlen(p), /* Comprimento do padrão */
       i,  /* Índice de um caractere do texto  */
       j;  /* Índice de um caractere do padrão */

      /* Enquanto os índices i e j referem-se a caracteres coincidentes eles  */
      /* são incrementados. Se eles se referirem a caracteres que não casam,  */
      /* enquanto isso ocorrer, j é reiniciado com o índice inicial do padrão */
      /* e i é associado ao próximo caractere do texto a ser comparado.      */
   for (i = 0, j = 0; i < n && j < m; ++i, ++j) {
         /* Enquanto os caracteres ora sendo comparados não casarem, */
         /* associa j ao índice inicial do padrão e i ao índice do   */
         /* próximo caractere a ser comparado no texto               */
      while (t[i] != p[j]) {
         i = i - j + 1;
         j = 0;
      }
   }

      /* Se o índice j for igual ao tamanho do padrão, todos */
      /* os caracteres do padrão casaram com caracteres do   */
      /* texto. Caso contrário, o laço for encerrou porque   */
      /* todos os caracteres do texto foram examinados       */
      /* (i.e., i == n) sem que o padrão fosse encontrado.   */
```

```
   if (j == m)
      return i - m; /* O padrão foi encontrado. Retorna-se sua posição no texto. */
   else
      return -1; /* Padrão não foi encontrado no texto */
}
```

9.2.4 Análise

Lema 9.1: Existem $n - m + 1$ janelas de tamanho m num texto contendo n caracteres.

Prova: A **Figura 9–7** ilustra o posicionamento da última janela de texto. Como mostra essa figura, antes da última janela de texto, há $n - m$ janelas. Portanto, levando-se em conta essa última janela, existem $n - m + 1$ janelas de texto. ■

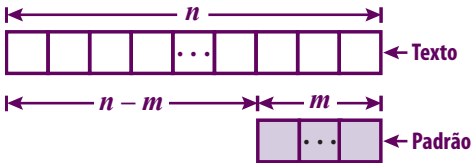


Figura 9–7: Número de Janelas num Texto

Teorema 9.1: No pior caso, o custo temporal do algoritmo **FB** é $\theta(m \cdot (n - m))$, em que n é o tamanho do texto e m é o tamanho do padrão.

Prova: O pior caso de casamento por força bruta ocorre numa das seguintes situações:

1. Quando todos os caracteres do texto e do padrão são iguais a um certo caractere, com exceção do último caractere do padrão, como, por exemplo:

   ```
   texto[] = "aaaaaaaaaaaaaaaaaa"
   padrao[] = "aaaab"
   ```

2. Quando todos os caracteres do texto e do padrão são iguais a um certo caractere x e o último de cada um desses strings é igual a um certo caractere y, como no exemplo:

   ```
   texto[] = "aaaaaaaaaaaaaaaaab"
   padrao[] = "aaaab"
   ```

Nesses dois casos, todas as janelas de texto são usadas e, de acordo com o **Lema 9.1**, o número dessas janelas é $n - m + 1$. No pior caso, em cada janela de texto, é necessário efetuar m comparações de caracteres até descobrir se o padrão casa ou não com a janela de texto. Assim o custo temporal do algoritmo de força bruta é, no pior caso, $\theta(m \cdot (n - m))$. ■

Teorema 9.2: No melhor caso[1], o custo temporal do algoritmo **FB** é $\theta(n - m)$, em que n é o tamanho do texto e m é o tamanho do padrão.

Prova: O melhor caso do algoritmo **FB** ocorre quando o primeiro caractere do padrão não ocorre no texto, como, por exemplo:

```
texto[]  = "aabccaabbaa"
padrao[] = "xaa"
```

Nesse caso, é efetuada apenas uma comparação em cada uma das $n - m + 1$ janelas, de modo que o custo temporal do algoritmo **FB** no melhor caso é $\theta(n - m)$. ■

Embora strings como aqueles que aparecem em casos extremos sejam improváveis numa linguagem natural, eles podem ocorrer com frequência, por exemplo, em linguagem binária ou genética.

[1] O melhor caso de um algoritmo de casamento de strings exclui a situação trivial na qual o padrão encontra-se no início do texto. Em tal situação, o custo temporal de qualquer desses algoritmos é $\theta(m)$.

Em termos de análise espacial, o algoritmo de casamento de strings por força bruta, tem custo $\theta(1)$, pois ele não requer espaço adicional proporcional a qualquer dos strings.

9.3 Algoritmo de Knuth, Morris e Pratt (KMP)

9.3.1 Visão Geral

O **algoritmo de Knuth, Morris e Pratt** (doravante **algoritmo KMP**) foi concebido por Donald Knuth e Vaughan Pratt e, independentemente, por James Morris em 1974. O artigo que descreve esse algoritmo foi publicado conjuntamente pelos três cientistas em 1977 (v. **Bibliografia**).

A ideia central que norteia o algoritmo **KMP** consiste em tomar conhecimento de alguns caracteres no texto sempre que eles casarem com alguns caracteres do padrão antes que ocorra uma discordância entre caracteres. Ou seja, esse algoritmo baseia-se no fato de que, quando ocorre uma tentativa frustrada de casamento, o próprio padrão contém informações suficientes para determinar em que posição do texto um casamento poderá ocorrer. Portanto esse algoritmo efetua um pré-processamento do padrão P para obter informações que permitam efetuar saltos maiores do que aqueles efetuados pelo algoritmo de força bruta. Mais precisamente, o algoritmo **KMP** cria uma tabela contendo informações sobre o padrão que permitem saltar caracteres do texto quando essas informações indicarem que existem comparações que não têm chance de ser bem-sucedidas.

No artigo original de Knuth, Morris e Pratt, essa tabela é denominada **função de falha**, mas ela recebe diversas denominações (p. ex., **tabela de prefixos**) dependendo do livro-texto ou artigo consultado sobre o assunto. Aqui, essa tabela será denominada **tabela de maiores bordas** (abreviadamente, **TMB**), pois parece que essa é a denominação que melhor lhe faz jus, como você verá adiante.

Antes de prosseguir, é importante alertar o leitor que a interpretação que será adotada para o algoritmo **KMP** é um pouco diferente daquela do artigo que lhe deu origem (e de muitos outros textos relacionados ao tema também), embora a ideia original seja preservada.

9.3.2 Tabela de Maiores Bordas

A tabela utilizada pelo algoritmo **KMP** é baseada nos maiores prefixos próprios que também são sufixos de cada substring do padrão. Ora, como foi visto na **Seção 9.1.1**, um prefixo próprio que também é sufixo é denominado *borda*, o que justifica a denominação adotada. A **Tabela 9–1** mostra como se calcula a tabela de maiores bordas (*TMB*) do string `"abab"`, ao passo que a **Figura 9–8** mostra essa tabela já pronta.

String (padrão): "abab"			
j	Substring	TMB[j]	Justificativa
0	"a"	0	O substring não tem borda
1	"ab"	0	O substring não tem borda
2	"aba"	1	A única borda do substring é "a"
3	"abab"	2	A única borda do substring é "ab"

Tabela 9–1: Exemplo de Cálculo de Tabela de Maiores Bordas 1

j	0	1	2	3
p[j]	a	ab	aba	abab
TMB[j]	0	0	1	2

Figura 9–8: Exemplo de Tabela de Maiores Bordas 1

A **Tabela 9–2** e a **Figura 9–9** apresentam outro exemplo de cálculo de tabela de maiores bordas.

String (padrão): "aaaa"			
j	Substring	TMB[j]	Justificativa
0	"a"	0	O substring não tem borda
1	"aa"	1	A única borda do substring é "a"
2	"aaa"	2	O substring tem duas bordas: "a" e "aa" e o tamanho da maior borda é 2
3	"aaaa"	3	O substring tem três bordas: "a", "aa" e "aaa" e o tamanho da maior borda é 3

Tabela 9–2: Exemplo de Cálculo de Tabela de Maiores Bordas 2

j	0	1	2	3
p[j]	a	aa	aaa	aaaa
TMB[j]	0	1	2	3

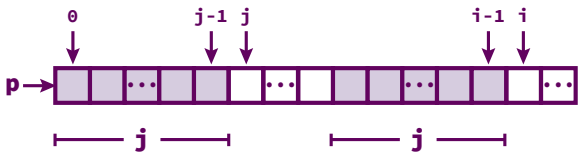
Figura 9–9: Exemplo de Tabela de Maiores Bordas 2

Como se vê nos exemplos acima, para cada posição *j* do padrão, armazena-se na tabela *TMB* o tamanho de sua maior borda até a posição anterior. Esse valor representa o quanto se deve recuar para tentar encontrar um casamento após uma discordância de caracteres na posição *j*.

9.3.3 Algoritmos

Criar uma tabela de maiores bordas para um string manualmente é uma tarefa relativamente simples, conforme foi visto acima. A questão agora é como desenvolver um algoritmo que descreva os passos necessários para criação de uma tal tabela. Essa questão será respondida a seguir.

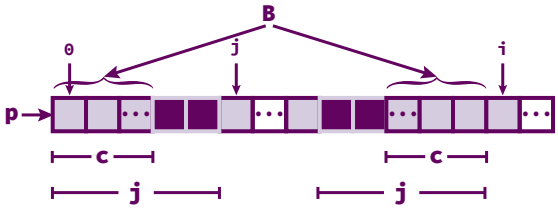
Suponha que os valores de uma *TMB* entre os índices *0* e *i – 1* de um string *p* sejam conhecidos. Então como se calcula *TMB[i]*? Seja *j* o comprimento da maior borda do substring *p[0.. i – 1]*, como mostra **Figura 9–10**. Sob essas considerações, tem-se que o prefixo que começa em *0* e termina em *j – 1* é igual ao sufixo que termina em *i – 1*, de modo que ambos têm comprimento *j*. Assim se *p[j]* for igual a *p[i]*, então *TMB[i]* será igual a *j + 1*. Depois que se calcula o valor de *TMB[i]*, prossegue-se com o cálculo de *TMB* para os próximos valores de *i* e *j*. O trecho do algoritmo correspondente a esse arrazoado é apresentado usando a sintaxe de C na porção direita da **Figura 9–10**.




```
if (p[j] == p[i]) {
   tmb[i] = j + 1;
   ++j;
   ++i;
}
```

Figura 9–10: Implementação de TMB do Algoritmo KMP 1

Agora suponha que, na **Figura 9–10**, *p[j]* seja diferente de *p[i]*. Nesse caso, é preciso encontrar o maior substring *B* que seja uma borda para *p[0..i – 1]* (v. **Figura 9–11**). Como o substring *B* é sufixo do substring que termina em *i – 1*, ele também é sufixo do substring que termina em *j – 1*. Logo *B* é a maior borda do substring *p[0..j – 1]*, de modo que o comprimento *c* de *B* deve ser *TMB[j – 1]*, que, por hipótese, é o tamanho da maior borda do substring *p[0..j – 1]*. Assim, nesse caso, quando *j* é diferente de *0*, ele deve ser atualizado para *TMB[j – 1]*, enquanto, quando *j* é igual a *0*, *TMB[i]* deve ser igual a *0* e o valor de *i* deve ser incrementado, como mostra o código na porção direita da **Figura 9–11**.




```
if (p[j] != p[i])
   if (j != 0)
      j = tmb[j - 1];
   else {
      tmb[i] = 0;
      ++i;
   }
```

Figura 9–11: Implementação de TMB do Algoritmo KMP 2

A **Figura 9–12** apresenta o algoritmo **KMP**.

Algoritmo KMP

Entrada: O texto *t* de comprimento *n* e o padrão *p* a ser procurado de comprimento *m*

Saída: A posição no texto da primeira ocorrência do padrão, se ele for encontrado no texto, ou um valor negativo, em caso contrário

1. Crie a tabela de maiores bordas (*TMB*) usando o algoritmo **CriaTMB**
2. Atribua a *i* o índice do primeiro caractere do texto
3. Atribua a *j* o índice do primeiro caractere do padrão
4. Enquanto $i < n$ e $j < m$, faça:
   - **4.1** Se os caracteres nas posições *i* do texto e *j* do padrão forem iguais:
     - **4.1.1** Se $j = m - 1$, retorne a posição do padrão no texto
     - **4.1.2** Caso contrário, incremente *i* e *j*
   - **4.2** Caso contrário, se $j > 0$, atribua a *j* o valor de *TMB[j – 1]*
   - **4.3** Caso contrário, incremente *i*
5. Retorne um valor negativo

Figura 9–12: Algoritmo KMP

A **Figura 9–13** mostra o algoritmo de criação da tabela de maiores bordas do algoritmo **KMP**.

Algoritmo CriaTMB

Entrada: Um string de comprimento *m*

Saída: A tabela de maiores borda (*TMB*) do string

1. Atribua *1* a *i*
2. Atribua *0* a *j*
3. Atribua *0* ao primeiro elemento da tabela
4. Enquanto $i < m$, faça:
   - **4.1** Se os caracteres nas posições *i* e *j* forem iguais:
     - **4.1.1** Atribua $j + 1$ a *TMB[i]*
     - **4.1.2** Incremente *i*
     - **4.1.3** Incremente *j*
   - **4.2** Caso contrário, se $j \neq 0$, atribua a *j* o valor de *TMB[j – 1]*
   - **4.3** Caso contrário:
     - **4.3.1** Atribua *0* a *TMB[i]*
     - **4.3.1** Incremente *i*

Figura 9–13: Algoritmo de Criação de TMB

Inicialmente, o algoritmo **KMP** funciona como o algoritmo de força bruta buscando no texto o primeiro caractere do padrão. Se ocorrer casamento, o segundo caractere do padrão é comparado com o próximo caractere do texto após o casamento. Essa comparação continua até que todos os caracteres do padrão casem ou até que ocorra uma discordância entre caracteres. Então o algoritmo usa o conhecimento adquirido com a análise prévia do padrão para saltar algumas comparações. Se não existirem bordas no padrão, o algoritmo **KMP** compara o primeiro caractere do padrão com o caractere após o último casamento. Desse modo, o algoritmo **KMP** pode saltar comparações entre caracteres que já casaram. Usando informações sobre bordas, o algoritmo **KMP** garante que não salta caracteres muito adiante. Se existe uma borda e ela casa no início do texto, o algoritmo **KMP** saltará apenas a borda inicial. Ou seja, ele não irá cotejar o primeiro caractere do padrão com o caractere após o último casamento; em vez disso, ele comparará o caractere depois da primeira ocorrência da borda no padrão com o caractere após a discordância no texto. Isso faz com que um possível casamento não seja saltado.

A **Figura 9–14** apresenta um exemplo de casamento de strings utilizando o algoritmo **KMP**. Nessa figura, o padrão e o texto são os mesmos do exemplo ilustrado na **Figura 9–5**. Note que a tabela de maiores bordas é usada para evitar uma das comparações entre um caractere do padrão e um caractere do texto. Observe ainda que o número de comparações neste exemplo é menor do que o número de comparações no exemplo da **Figura 9–5**.

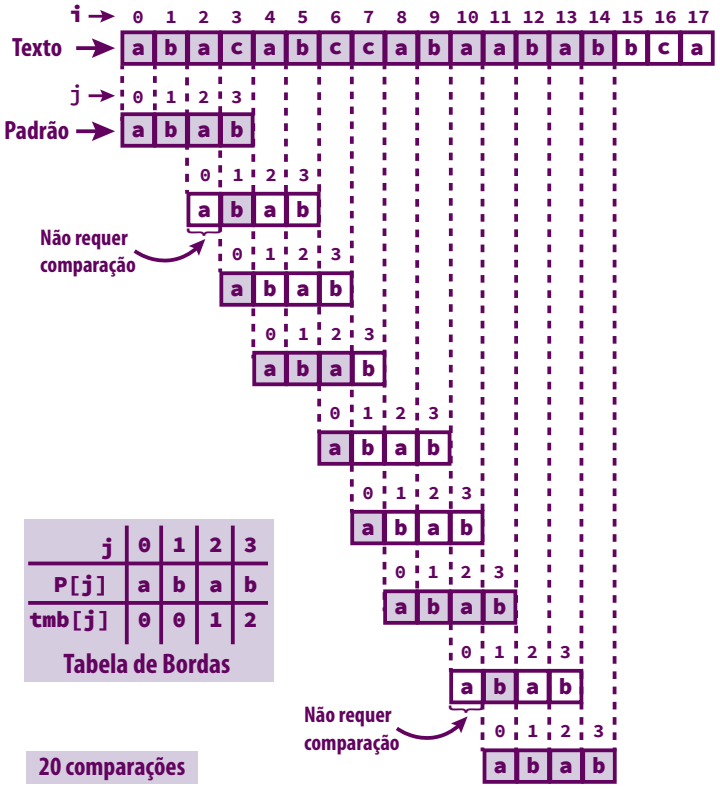


Figura 9–14: Exemplo de Uso do Algoritmo KMP 1

9.3.4 Implementação

A função `CriaTMB()` exibida adiante atribui valores para os elementos da tabela de maiores bordas usada pelo algoritmo **KMP** de acordo com a **Figura 9–13**. Essa função retorna o endereço da tabela e seus parâmetros são:

- `tabela` (saída) — array que armazenará a tabela
- `p` (entrada) — string que representa o padrão
- `m` (entrada) — tamanho do padrão

```
int *CriaTMB(int tmb[], const char *p, int m)
{
   int i = 1 , j = 0;

   tmb[0] = 0; /* O primeiro elemento da tabela é sempre 0 */

   while (i < m)
      if (p[j] == p[i]) {
         tmb[i] = j + 1;
         ++j;
         ++i;
      } else {
         if (j != 0) {
            j = tmb[j - 1];
         } else {
            tmb[i] = 0;
            ++i;
         }
      }

   return tmb;
}
```

A função `CasamentoKMP()` verifica se um padrão (representado pelo parâmetro `p`) ocorre num texto (representado pelo parâmetro `t`) usando o algoritmo **KMP**. Ela retorna a posição (índice) do padrão no texto, se ele for encontrado, ou `-1`, em caso contrário.

```
int CasamentoKMP(const char *t, const char *p)
{
   int n = strlen(t), /* Comprimento do texto  */
       m = strlen(p), /* Comprimento do padrão */
       i,  /* Índice de um caractere do texto  */
       j,  /* Índice de um caractere do padrão */
      *tmb; /* Ponteiro para a tabela de maiores bordas */

      /* Aloca espaço para a tabela de maiores bordas */
   ASSEGURA( tmb = malloc(m*sizeof(int)), "Impossivel aloca tabela de bordas" );

   CriaTMB(tmb, p, m); /* Cria a tabela de maiores bordas */

      /* Efetua a busca pelo padrão no texto */
   for (i = 0, j = 0; i < n; ) {
         /* Verifica se os caracteres ora comparados casam */
      if (t[i] == p[j]) {
            /* Caracteres comparados casaram. Se esse era o último caractere do */
            /* padrão, retorna-se a  posição na qual ocorreu o casamento. Caso  */
            /* contrário, incrementam-se os índices.                            */
         if (j == m - 1) {
            free(tmb); /* A tabela tmb não é mais necessária */
            return i - j; /* Padrão foi encontrado no texto */
         } else {/* Compara os dois próximos caracteres */
            ++i;
            ++j;
         }
      } else if (j > 0) {
            /* Os caracteres não casaram e j é diferente */
            /* de 0. Ajusta-se j, mas não se ajusta i.   */
         j = tmb[j - 1];
      } else {
```

```
            /* Os caracteres não casaram e j é igual a   */
            /* 0 (i.e., o primeiro caractere do padrão). */
         ++i; /* Incrementa-se i, mas não se altera j.     */
      }
   }

   free(tmb); /* A tabela tmb não é mais necessária */

   return -1; /* O padrão não foi encontrado no texto */
}
```

A parte principal da função `CasamentoKMP()` é o laço **for** que compara um caractere em `t[]` com outro caractere em `p[]` a cada iteração. Sempre que ocorre um casamento entre um caractere do texto e outro do padrão, os índices `i` e `j` são incrementados. Se ocorrer uma discordância após alguns caracteres já terem casado, consulta-se a tabela de maiores bordas para determinar o novo índice em `p[]` no qual a comparação irá continuar. Se houver uma discordância e a comparação estiver no início de `p[]`, incrementa-se o índice de `t[]` e mantém-se o índice de `p[]` onde ele estava (i.e., no início de `p[]`). Esse processo continua até que se encontre um casamento ou até que o índice `i` seja igual a `n`, que é o comprimento de `t[]`. As comparações saltadas são desnecessárias, pois a tabela de bordas garante que todas as comparações evitadas são redundantes; i.e., elas envolvem comparar caracteres que já se sabe que casam.

9.3.5 Análise

Lema 9.2: O custo temporal de construção da tabela de maiores bordas de um string com m caracteres é $\theta(m)$.

Prova: O laço (**Passo 4** na **Figura 9–13**) do algoritmo de construção dessa tabela é executado m vezes e, portanto, seu custo é $\theta(m)$. Todos os demais passos têm custo $\theta(1)$. ■

Teorema 9.3: No pior caso, o custo temporal do algoritmo **KMP** é $\theta(m + n)$, em que n é o tamanho do texto e m é o tamanho do padrão.

Prova: Cada vez que o laço do algoritmo **KMP** (**Passo 4** na **Figura 9–12**) é executado, o valor de i é incrementado ou o padrão desliza para a direita por um valor determinado pela tabela de saltos. Ambos os eventos podem ocorrer no máximo n vezes, de modo que o laço pode ser executado, no máximo, $2 \cdot n$ vezes. Como o custo de cada passo do corpo do laço é $\theta(1)$, o custo desse laço é $\theta(n)$. Por outro lado, de acordo com o **Lema 9.2**, o custo temporal do algoritmo de construção da tabela de maiores bordas é $\theta(m)$. Como as duas partes do algoritmo **KMP** têm custos temporais $\theta(m)$ e $\theta(n)$, o custo temporal no pior caso do algoritmo é $\theta(m + n)$. ■

A título de ilustração, suponha que se tenha um padrão com *1000* caracteres, sendo que os primeiros *999* caracteres são iguais a `'a'` enquanto o último caractere é igual a `'b'`. Suponha ainda que se tenha um texto com um milhão de caracteres `'a'` e que o algoritmo **KMP** processe esse padrão e esse texto. Quando esse algoritmo encerrar sua primeira tentativa frustrada de casamento ele incrementará o índice de busca no texto, mas ele sabe que os primeiros *998* caracteres na nova posição já casam. Quer dizer, o algoritmo **KMP** compara *999* caracteres do padrão antes de descobrir que o milésimo caractere não casa. Avançando uma posição no texto, descarta-se o primeiro `'a'`, de modo que o algoritmo **KMP** desvenda que há *998* caracteres `'a'` que casam com o padrão e não os compara novamente. Ou seja, nesse caso, o algoritmo **KMP** atribui *998* à nova posição de comparação no texto. O conhecimento usado pelo algoritmo **KMP** para efetuar esse salto é derivado da tabela de maiores bordas. Em idêntica situação, o algoritmo **FB** efetuaria cerca de um bilhão de comparações. A **Figura 9–15** ilustra esse exemplo.

O custo temporal do algoritmo **KMP** é considerado ótimo para o pior caso, pois, nesse caso, inevitavelmente, qualquer algoritmo de casamento de strings deve comparar todos os caracteres do texto com todos os caracteres

do padrão um número constante de vezes. A eficiência do algoritmo **KMP** é derivada do fato de ele não efetuar comparações que são redundantes.

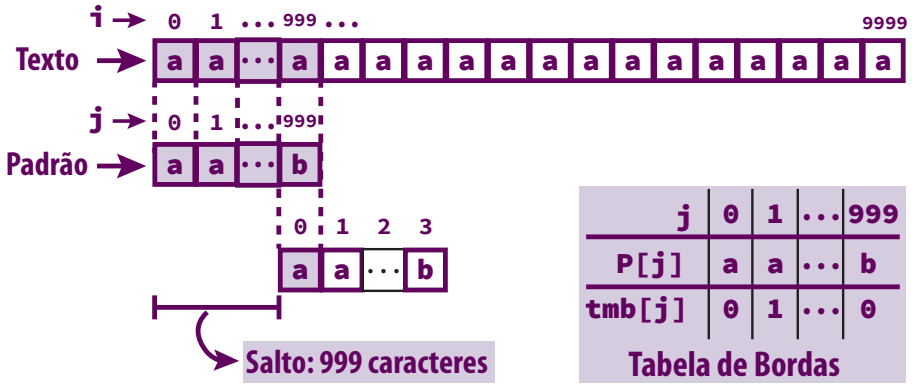


Figura 9–15: Exemplo de Uso do Algoritmo KMP 2

Na prática, o melhor desempenho do algoritmo **KMP** com relação ao algoritmo de força bruta não é tão importante porque poucas aplicações envolvem casamentos de padrões e textos tão repetitivos. Quer dizer, o algoritmo **KMP** é mais conveniente para casamento de strings derivados de pequenos alfabetos, como aquele que contém símbolos de DNA. Além disso, como esse algoritmo não retrocede no texto, ele é conveniente para casamento de padrão com texto que flui continuamente (p.ex., texto introduzido via teclado — v. **Seção 9.10.7**) e que não tem tamanho definido a priori. Algoritmos que requerem retrocesso no texto (p.ex., o algoritmo de força bruta) não são adequados para essa última tarefa. É importante ainda mencionar que o custo do algoritmo **KMP** não depende do tamanho do alfabeto, como ocorre, por exemplo, com o algoritmo **BM** (v. **Seção 9.4**).

9.4 Algoritmo de Boyer e Moore (BM)

9.4.1 Visão Geral

O **algoritmo de Boyer e Moore**, doravante também denominado **algoritmo BM**, foi desenvolvido por Robert Boyer e J Moore em 1977 (v. **Bibliografia**) baseado na ideia básica que norteia o algoritmo **KMP**. Ou seja, assim como ocorre com o algoritmo **KMP**, o algoritmo **BM** usa informações sobre bordas para saltar caracteres no texto e assim acelerar a busca. Agora, as semelhanças entre esses dois algoritmos param por aqui, pois o algoritmo **BM** possui três características que o distingue dos demais algoritmos de casamento de strings:

- O algoritmo **BM** inicia as comparações de caracteres a partir do final do padrão, em vez de a partir do seu início, como fazem os demais algoritmos de casamento de strings discutidos até aqui.

Além disso, ele usa duas regras para saltar caracteres de modo semelhante ao que o algoritmo **KMP** faz. As regras usadas pelo algoritmo **BM** são:

- **Regra do mau caractere**. Quando se compara um caractere do padrão com outro no texto, se ocorre uma discordância, o algoritmo verifica se o caractere discordante no texto aparece no padrão. Se esse for o caso, alinha-se a ocorrência mais à direita desse caractere no padrão com o caractere no texto que provocou a última discordância.
- **Regra do bom sufixo**. Essa regra procura sufixos repetidos no padrão (assim como **KMP** faz com prefixos). Se um caractere no texto e outro no padrão não casam, move-se o padrão para a direita de modo a alinhar qualquer sufixo que tenha casado até então com a ocorrência mais à direita do sufixo no padrão.

Essas regras são denominadas *heurísticas*, pois quando uma delas é usada isoladamente, é possível que não se obtenha o salto almejado. Por exemplo, quando é usada isoladamente, a regra do mau caractere pode resultar num salto negativo (v. **Seção 9.5**).

Assim como ocorre com o algoritmo **KMP**, o algoritmo **BM** também tem uma **fase de pré-processamento**. Durante essa fase, ele usa ambas as regras descritas acima para criar duas tabelas que estipulam, para cada posição do padrão, de quanto será o salto dele com relação ao texto quando ocorrer uma discordância entre caracteres na respectiva posição. Durante a segunda fase do algoritmo **BM**, denominada **fase de casamento**, ele consulta essas duas tabelas para obter o maior salto possível durante o deslizamento do padrão em busca de uma nova tentativa de casamento.

9.4.2 Heurística do Mau Caractere

De acordo com o algoritmo **BM**, um **mau caractere** é um caractere do texto que causa uma discordância quando é comparado com um caractere do padrão. Quando tal caractere aparece em alguma posição no padrão, pode-se deslocar o padrão adiante de modo que essa ocorrência seja alinhada com o mau caractere, como mostra a **Figura 9–16**. Nessa figura, os caracteres `'b'` e `'c'` são discordantes, de maneira que o caractere `'b'` (no texto) se torna um mau caractere. Esse caractere aparece nas posições *0* e *2* do padrão, sendo que a última ocorrência é na posição *2*. Assim o padrão pode ser avançado, de modo que essa última ocorrência seja alinhada com o mau caractere.

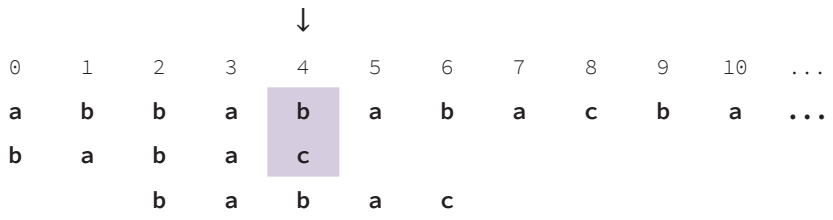


Figura 9–16: Regra do Mau Caractere do Algoritmo BM

9.4.3 Heurística do Bom Sufixo

Algumas vezes, a regra do mau caractere não funciona. Por exemplo, na situação mostrada na **Figura 9–17**, os caracteres `'a'` no texto e `'b'` no padrão são discordantes. Nesse caso, um alinhamento do mau caractere (i.e., `'a'`) do texto com o último respectivo caractere do padrão não é a melhor opção, pois esse deslocamento seria negativo (i.e., o caractere `'a'` na posição *3* do padrão seria alinhado com o caractere `'a'` na posição *2* do texto). É num caso como esse que se usa a regra do bom sufixo. Quer dizer, como mostra a **Figura 9–17**, o sufixo `"ab"` do padrão casou com o substring `"ab"` no texto e esse mesmo sufixo aparece novamente na posição `1` do padrão. Assim o padrão pode ser deslocado de modo que essa ocorrência do sufixo seja alinhada com o referido substring do texto, como mostra as áreas escuras da referida figura.

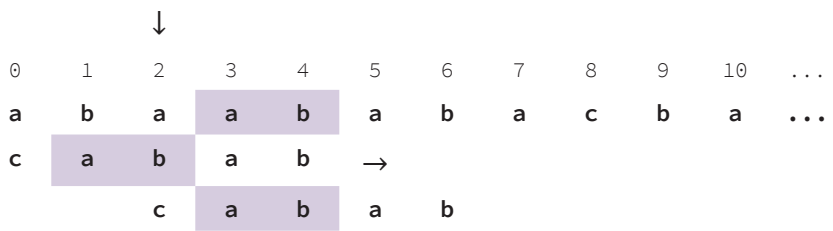


Figura 9–17: Regra do Bom Sufixo versus Regra do Mau Caractere do Algoritmo BM

Na **Figura 9–18**, não há nenhuma outra ocorrência de `"ab"` no padrão, de maneira que ele pode ser deslocado para logo adiante de `"ab"` (i.e., para a posição *5* do texto). Utilizando-se a regra do mau caractere (`'c'`, nesse caso), o salto seria menor porque o caractere `'c'` no padrão seria alinhado com o caractere `'c'` na posição *2* do texto.

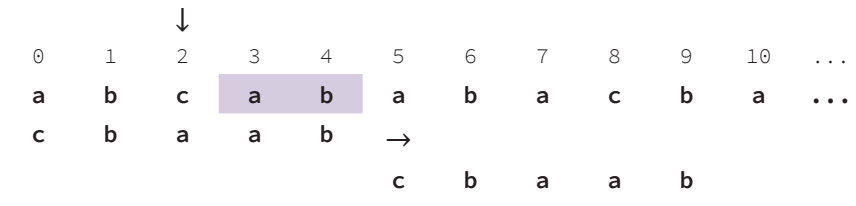


Figura 9–18: Caso 1 da Regra do Bom Sufixo do Algoritmo BM

A aplicação da regra do bom sufixo nem sempre é tão simples quanto parece. Considere, por exemplo, o caso apresentado na **Figura 9–19**. Nessa situação, não há nenhuma outra ocorrência do substring `"bab"` além daquela que já casou com o texto. Mas, nesse caso, o padrão não pode ser deslocado para a posição *5* como no caso anterior. Ou seja, ele pode ser deslocado apenas para a posição *3*, uma vez que o prefixo `"ab"` do padrão casa com o final de `"bab"`.

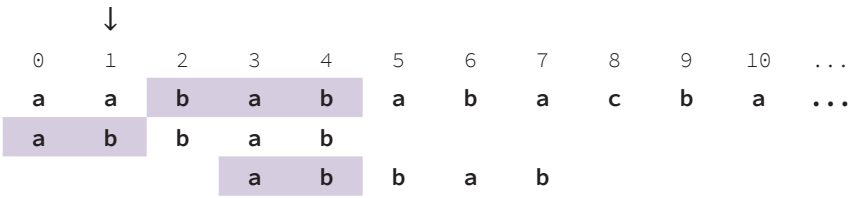


Figura 9–19: Caso 2 da Regra do Bom Sufixo do Algoritmo BM

Na discussão a seguir, a situação ilustrada na **Figura 9–17** será rotulada como **Caso 1**, ao passo que aquela apresentada na **Figura 9–19** será denominada **Caso 2**. A **Figura 9–20** resume esses dois casos.

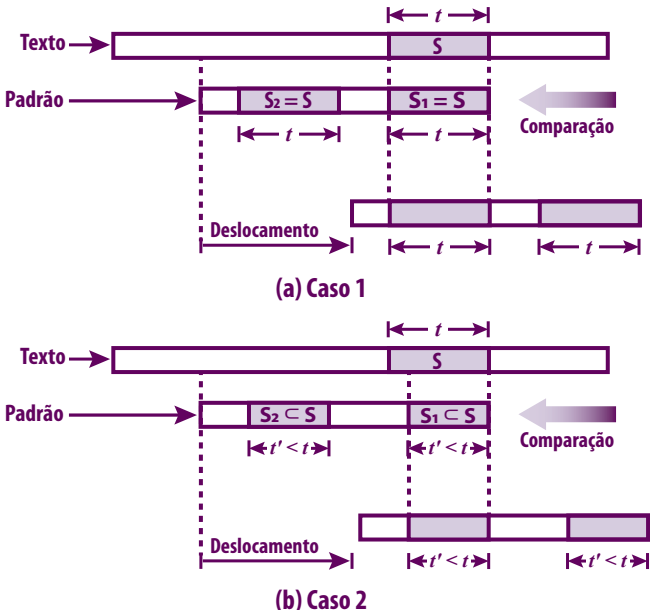


Figura 9–20: Casos de Bom Sufixo do Algoritmo BM

O processamento do **Caso 1** é semelhante à criação da tabela de maiores bordas do algoritmo **KMP**, pois um bom sufixo é uma borda de um sufixo do padrão. Portanto as bordas dos sufixos do padrão devem ser determinadas. Contudo, agora, é necessário um mapeamento inverso entre uma dada borda e o menor sufixo do padrão que contém essa borda. Além disso, é preciso que as bordas em questão não tenham o mesmo caractere como vizinho esquerdo, pois isso poderá causar uma discordância entre esses caracteres após deslocar o padrão.

No **Caso 2**, uma parte do bom sufixo ocorre no início do padrão, o que significa que essa parte é uma borda do padrão. Assim o padrão pode ser deslocado tanto quanto sua borda correspondente permitir [v. **Figura 9–20 (b)**]. No processamento desse caso, para cada sufixo, a maior borda do padrão que está contida nesse sufixo é determinada.

9.4.4 Algoritmos

A **Figura 9–21** apresenta o algoritmo **BM**.

ALGORITMO BM

ENTRADA: O texto *t* de comprimento *n* e o padrão *p* a ser procurado de comprimento *m*

SAÍDA: A posição no texto da primeira ocorrência do padrão, se ele for encontrado no texto, ou um valor negativo, em caso contrário

1. Atribua a *i* o índice do primeiro caractere do texto
2. Armazene no array *tm* a tabela de maus caracteres obtida usando o algoritmo **CRIATBMAUSCARACTERES**
3. Armazene no array *tb* a tabela de bons sufixos obtida usando o algoritmo **CRIATBBONSSUFIXOS**
4. Enquanto $i \leq n - m$, faça:
   - **4.1** Atribua a *j* o índice do último caractere do padrão
   - **4.2** Enquanto $j \geq 0$ e os caracteres na posição *j* do padrão e na posição $i + j$ do texto forem iguais, decremente o valor de *j*
   - **4.3** Se $j < 0$, retorne o valor de *i*
   - **4.4** Atribua a *c* o caractere que se encontra na posição $i + j$ no texto
   - **4.5** Acrescente a *i* o maior valor dentre *tb[j + 1]* e *j – tm[c]*
5. Retorne *–1*

FIGURA 9–21: ALGORITMO BM

O algoritmo **CRIATBMAUSCARACTERES** invocado pelo algoritmo **BM** é exibido na **Figura 9–22**.

ALGORITMO CRIATBMAUSCARACTERES

ENTRADA: Um string *p* de comprimento *m*

SAÍDA: A tabela de maus caracteres criada

1. Crie um array *tm* do tamanho do alfabeto sob consideração
2. Atribua *–1* a cada elemento do array *tm*
3. Para cada caractere *c* do padrão, atribua sua posição no padrão a *tm[c]*

FIGURA 9–22: ALGORITMO DE CRIAÇÃO DA TABELA DE MAUS CARACTERES

A **Figura 9–23** apresenta o algoritmo **CRIATBBONSSUFIXOS** invocado pelo algoritmo **BM**.

ALGORITMO CRIATBBONSSUFIXOS

ENTRADA: Um string *p* de comprimento *m*

SAÍDA: A tabela de bons sufixos criada (*tb*)

1. Atribua *0* a cada elemento de *tb*
2. Atribua *0* a cada elemento de um array auxiliar *aux* com $m + 1$ elementos
3. Atribua *m* a *i*
4. Atribua $m + 1$ a *j*
5. Atribua $m + 1$ ao último elemento do array *aux*

CONTINUA

FIGURA 9–23: ALGORITMO DE CRIAÇÃO DA TABELA DE BONS SUFIXOS

Algoritmo CriaTbBonsSufixos (Continuação)

6. Enquanto $i > 0$, faça:
  6.1 Enquanto $j \leq m$ e os caracteres nas posições $i - 1$ e $j - 1$ de p forem diferentes, faça:
    6.1.1 Se $tb[j] = 0$, atribua $j - 1$ a $tb[j]$
    6.1.2 Atribua $aux[j]$ a j
  6.2 Decremente i
  6.3 Decremente j
  6.4 Atribua j a $aux[i]$
7. Atribua $aux[0]$ a j
8. Atribua 0 a i
9. Enquanto $i \leq m$, faça:
  9.1 Se $tb[i] = 0$, atribua j a $tb[i]$
  9.2 Se $i = j$, atribua $aux[j]$ a j

Figura 9–23 (Cont.): Algoritmo de Criação da Tabela de Bons Sufixos

A **Figura 9–24** mostra um exemplo de uso do algoritmo **BM**. Nessa figura, o texto e o padrão são os mesmos usados nos exemplos da **Figura 9–5** e da **Figura 9–14**. Note que, na **Figura 9–24**, o número de comparações é o menor dentre todos esses exemplos.

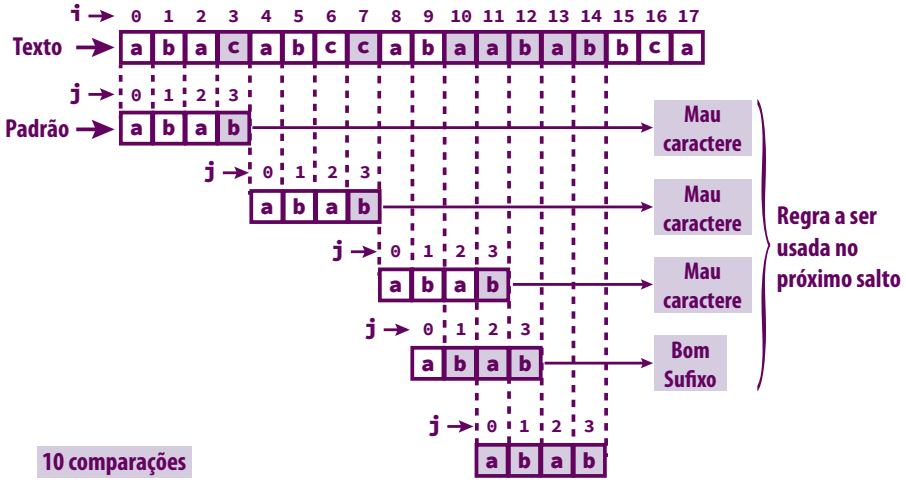


Figura 9–24: Exemplo de Uso do Algoritmo BM

9.4.5 Implementação

A função `TabelaMauCaractereBM()` atribui valores para os elementos da tabela de maus caracteres usada pelo algoritmo **BM**. Seus parâmetros são o padrão `p` e seu tamanho `m`.

```
int *TabelaMauCaractereBM(const char *p, int m)
{
   int *tab,
        caractere,
        i;

      /* O número de elementos do array que representa a tabela de */
      /* maus caracteres é igual ao tamanho do alfabeto utilizado  */
   tab = calloc(TAM_ALFABETO, sizeof(int));
```

```
   ASSEGURA(tab, "Impossivel alocar tabela de maus caracteres");

      /* Inicialmente, supõe-se que nenhum caractere do     */
      /* alfabeto faz parte do padrão e portanto o elemento */
      /* correspondente a cada caractere é iniciado com -1  */
   for (i = 0; i < TAM_ALFABETO; ++i)
      tab[i] = -1;

      /* Atribui valores aos elementos correspondentes */
      /* aos caracteres que fazem parte do padrão      */
   for (i = 0; i < m; ++i) {
         /* Obtém o caractere que está na posição i do padrão */
      caractere = (int) p[i];

         /* O valor da tabela correspondente ao */
         /* caractere é a sua posição no padrão */
      tab[caractere] = i;
   }

   return tab;
}
```

A constante `TAM_ALFABETO` usada pela função `TabelaMauCaractereBM()` especifica o tamanho do alfabeto usado no contexto dos strings que serão comparados. Essa função é relativamente fácil de entender seguindo os comentários que a acompanham. Note que, nessa tabela, cada caractere é associado à sua última posição no padrão. Se um caractere do alfabeto em consideração não aparece no padrão, seu valor é *–1*. Por exemplo, quando a função `TabelaMauCaractereBM()` recebe como parâmetro o padrão `"abacab"`, ela produz a seguinte tabela:

Caractere (c)	tab[c]
'a'	4
'b'	5
'c'	3
Qualquer outro caractere	-1

A função `TabelaSufixosBM()`, apresentada a seguir, cria a tabela de bons sufixos do algoritmo **BM** e seus parâmetros são:

- `p` (entrada) — string que representa o padrão
- `m` (entrada) — tamanho do padrão

```
int *TabelaSufixosBM(const char *p, int m)
{
   int  i = m,
        j = m + 1,
       *bons, /* Tabela de bons sufixos */
       *aux;  /* Tabela auxiliar */

      /* Aloca espaço para os arrays bons[] e aux[] e testa suas alocações */
   bons = calloc(m + 1, sizeof(int));
   ASSEGURA(bons, "Impossivel alocar array bons[]");
   aux = calloc(m + 1, sizeof(int));
   ASSEGURA(aux, "Impossivel alocar array aux[]");

               /* Caso 1 */

      /* O sufixo que inicia na posição m não possui   */
      /* borda. Portanto, aux[m] recebe o valor m + 1. */
```

```
   aux[m] = m + 1;

   while (i > 0) {
      while (j <= m && p[i - 1] != p[j - 1]) {
            /* O salto correspondente à posição j é armazenado em s[j], */
            /* desde que o valor de s[j] seja diferente de 0            */
         if (bons[j] == 0)
            bons[j] = j - i;

         j = aux[j]; /* Passa para a próxima borda do padrão */
      }

      --i; /* Passa para o próximo sufixo */
      --j;

         /* Cada elemento do array aux[] contém a posição inicial da */
         /* maior borda do sufixo do padrão que inicia na posição i  */
      aux[i] = j;
   }

               /* Caso 2 */

      /* Armazena em j a posição inicial da maior    */
      /* borda do padrão, que é armazenada em aux[0] */
   j = aux[0];

      /* Armazena o valor de aux[0] em cada elemento de s[] que é igual a 0.  */
      /* Quando o sufixo do padrão se torna menor do que aux[0], mantém-se a  */
      /* próxima borda mais larga do padrão (i.e., aux[j]).                   */
   for (i = 0; i <= m; i++) {
      if (bons[i] == 0)
         bons[i] = j;

      if (i == j)
         j = aux[j];
   }
   free(aux); /* A tabela auxiliar não é mais necessária */

   return bons;
}
```

A seção intitulada **Caso 1** da função `TabelaSufixosBM()` usa o array auxiliar `aux[]` do qual cada elemento `aux[i]` contém a posição inicial da maior borda do sufixo do padrão que inicia na posição `i`. O sufixo que inicia na posição `m` não possui borda, de modo que `aux[m]` recebe o valor `m + 1`. O tamanho de cada borda é calculado checando-se se uma borda menor que já tenha sido encontrada tem como vizinho esquerdo o mesmo caractere. Entretanto o caso em que uma borda não pode ser estendida para a esquerda também é de interesse, visto que ele pode levar a um possível deslocamento do padrão quando ocorre uma discordância entre caracteres. Portanto o salto correspondente é armazenado no array `s[]`, desde que a respectiva posição do array ainda não esteja ocupada (i.e., desde que o valor desse elemento seja `0`), o que ocorre quando um sufixo menor tem a mesma borda. A **Figura 9–25** apresenta um exemplo dos resultados obtidos com a execução da seção intitulada **Caso 1** da função `TabelaSufixosBM()`.

			↓		↓			
i	0	1	2	3	4	5	6	7
p	a	b	b	a	b	a	b	
aux[i]	5	6	4	5	6	7	7	8
s[i]	0	0	0	0	2	0	4	1

FIGURA 9–25: EXEMPLO DE CASO 1 DA REGRA DO BOM SUFIXO DO ALGORITMO BM

A maior borda do sufixo `"babab"`, que começa na posição *2*, é `"bab"`, que começa na posição *4*. Portanto `aux[2]` é igual a `4`. A maior borda do sufixo `"ab"` que começa na posição *5* é o string vazio, que começa na posição `7`. Portanto `aux[5]` é igual a `7`.

Os valores do array `s[]` são determinados pelas bordas que não podem ser estendidas à esquerda. O sufixo `"babab"` que começa na posição *2* tem borda igual a `"bab"`, que começa na posição *4*. Essa borda não pode ser estendida à esquerda porque $p[1] \neq p[3]$. O resultado da diferença $4 - 2$ é o salto quando ocorre um casamento com `"bab"` seguido por uma discordância entre caracteres. Portanto `s[4]` é igual a `2`.

O mesmo sufixo `"babab"` também tem borda igual a `"b"`, que começa na posição `6`. Essa borda também não pode ser estendida à esquerda. O resultado da diferença $6 - 2$ é o tamanho do salto quando ocorre um casamento com `"b"` seguido por uma discordância entre caracteres. Portanto `s[6]` é igual a `4`.

O sufixo `"b"` que começa na posição *6* tem borda vazia, que começa na posição *7*. Essa borda não pode ser estendida à esquerda e o resultado da diferença $7 - 6$ é o tamanho do salto quando ocorre discordância entre caracteres na primeira comparação (i.e., quando não há bom sufixo). Portanto `s[7]` é igual a `1`.

Na seção intitulada **Caso 2** da função `TabelaSufixosBM()`, a posição inicial da maior borda do padrão é armazenada em `aux[0]`, que, no exemplo da **Figura 9–25**, é *5*, uma vez que a borda `"ab"` começa na posição *5*. Nessa função, o valor `aux[0]` é inicialmente armazenado em cada posição disponível (i.e., cujo elemento vale `0`) do array `s[]`, mas, quando o sufixo do padrão se torna menor do que `aux[0]`, a função mantém a próxima borda mais larga do padrão; ou seja, `aux[j]`. A **Figura 9–26** apresenta um exemplo dos resultados obtidos com a execução da função `TabelaSufixosBM()`. Nessa figura, usa-se o mesmo padrão utilizado no exemplo da **Figura 9–25**.

i	0	1	2	3	4	5	6	7
p	a	b	b	a	b	a	b	
aux[i]	5	6	4	5	6	7	7	8
s[i]	5	5	5	5	2	5	4	1

FIGURA 9–26: EXEMPLO DE CASO 2 DA REGRA DO BOM SUFIXO DO ALGORITMO BM

A função `CasamentoBM()` verifica se um padrão ocorre num texto usando o algoritmo **BM**. Seus parâmetros são `t` e `p`, que são strings que representam, respectivamente, o texto e o padrão.

```
int CasamentoBM(const char *t, const char *p)
{
   int i = 0,
       j,
       caractere,
       n = strlen(t), /* Tamanho do texto  */
       m = strlen(p), /* Tamanho do padrão */
      *maus, /* Tabela de maus caracteres */
      *bons; /* Tabela de bons sufixos    */

   maus = TabelaMauCaractereBM(p, m); /* Obtém a tabela de maus caracteres */

   bons = TabelaSufixosBM(p, m); /* Obtém a tabela de bons sufixos */

      /* Compara caracteres do padrão com caracteres do texto da direita para a */
      /* esquerda. Quando ocorre uma discordância entre caracteres, o padrão é  */
      /* deslocado pelo maior valor dentre aqueles encontrados nos arrays       */
      /* bons[] e maus[].                                                       */
   while (i <= n - m) {
      j = m - 1;
```

```
      while (j >= 0 && p[j] == t[i + j])
         j--;

      if (j < 0) {
         break; /* O padrão foi encontrado no texto */
      } else {
         caractere = (int) t[i + j];
         i += MAX(bons[j + 1], j - maus[caractere]);
      }
   }

   free(bons); /* O array bons[] não é mais necessário */
   free(maus); /* O array maus[] não é mais necessário */

      /* Verifica se o padrão foi encontrado no texto */
   if (i <= n - m)
      return i; /* O padrão foi encontrado */
   else
      return -1; /* O padrão não foi encontrado */
}
```

A função `CasamentoBM()` compara caracteres do padrão com caracteres do texto da direita para a esquerda. Quando ocorre uma discordância entre caracteres, o padrão é deslocado pelo maior valor dentre os valores resultantes das regras do mau caractere e do bom sufixo.

9.4.6 Análise

No pior caso, o custo temporal do algoritmo **BM** é $\theta(m + n)$, quando o padrão encontra-se no texto, e $\theta(m \cdot n)$, quando o padrão não aparece no texto. A prova dessa afirmação é bem complicada e está além do escopo deste livro. O leitor interessado poderá encontrar essa prova no artigo original de Boyer e Moore (1977) e no artigo de Richard Cole (1991) (v. **Bibliografia**).

O melhor caso do algoritmo **BM** acontece quando em cada nova tentativa de casamento o primeiro caractere comparado não ocorre no padrão. Nesse caso, esse algoritmo requer apenas $\theta(n/m)$ comparações. Se o tamanho do alfabeto for grande em comparação com o tamanho do padrão, o algoritmo **BM** tem custo médio $\theta(n/m)$. Isso ocorre porque saltos de m caracteres ocorrem com frequência devido à regra do mau caractere[2].

O processamento do padrão para atender a regra do bom sufixo é um tanto complicado de entender e implementar, de modo que muito autores deixam essa regra de lado com a desculpa de que essa regra não economiza tantas comparações e que a regra do mau caractere é suficiente. Todavia essas afirmações nem sempre correspondem à realidade, notadamente quando se lida com alfabetos pequenos, como o alfabeto binário ou o alfabeto genético.

9.5 Algoritmo (ou Simplificação) de Horspool (BMH)

9.5.1 Visão Geral

O **algoritmo de Horspool** é uma simplificação do algoritmo de Boyer e Moore, mas, apesar disso, ele é bem mais simples de entender e implementar do que o algoritmo no qual ele se baseia. O algoritmo de Horspool, tipicamente denominado **algoritmo de Boyer, Moore e Horspool** (ou **algoritmo BMH**), foi desenvolvido por Nigel Horspool em 1980 (v. **Bibliografia**).

A regra do mau caractere usada pelo algoritmo de Boyer e Moore não é muito eficiente para alfabetos pequenos, mas quando o alfabeto é relativamente grande em comparação ao tamanho do padrão (p.ex., quando o alfabeto

[2] As provas dessas últimas afirmações podem ser encontradas no artigo de Baeza-Yates e Régnier (1992) (v. **Bibliografia**).

sob consideração é usado em linguagem natural), essa regra é bastante útil. Assim uma adaptação dessa regra para uso isolado resulta num algoritmo de casamento bem eficiente na prática.

Na fase de pré-processamento do algoritmo **BMH**, o alfabeto e o padrão são levados em consideração na criação de uma **tabela de saltos**. Essa tabela associa cada caractere do alfabeto a um valor inteiro não negativo que indica o tamanho do salto a ser efetuado quando ocorre uma discordância entre o caractere do padrão e o respectivo caractere do texto ora sendo comparados. Quando um caractere do alfabeto não faz parte do padrão, o tamanho do salto é igual ao tamanho do padrão; caso contrário, o tamanho do salto é igual ao número de caracteres entre o caractere em questão e o último caractere do padrão. Ou seja, formalmente, o valor de cada salto é determinado pela fórmula:

$$tab(c) = \begin{cases} m & se\ c \notin P \\ m - i - 1 & se\ c \in P \end{cases}$$

Nessa fórmula, c é o caractere para o qual o salto será calculado, m é o tamanho do padrão e i é a posição do caractere no padrão. É importante salientar que, embora não esteja explícito nessa fórmula, o último caractere do padrão não é levado em consideração na construção dessa tabela.

Observe que existem duas diferenças visíveis entre uma tabela de maus caracteres do algoritmo **BM** e uma tabela de saltos do algoritmo **BMH**:

[1] A cada caractere que não faz parte do padrão, o algoritmo que calcula tabela de maus caracteres associa -1. Por outro lado, nesse caso, a tabela de saltos do algoritmo **BMH** atribui ao mesmo caractere o tamanho m do padrão.

[2] A cada caractere que se encontra na posição i do padrão, uma tabela de maus caracteres associa o valor de i; por sua vez, uma tabela de saltos associa $m - i - 1$ ao mesmo caractere.

Mas a principal diferença entre os algoritmos **BM** e **BMH** diz respeito a como essas tabelas são interpretadas pelos respectivos algoritmos. A tabela de maus caracteres é consultada usando-se como índice um mau caractere, como foi visto na **Seção 9.4**. Por sua vez, o algoritmo **BMH** consulta sua tabela de saltos usando como índice o caractere mais à direita da janela de texto, como mostra a **Figura 9–27**. Além disso, quando a tabela de maus caracteres é usada isoladamente (i.e., sem a tabela de bons sufixos do algoritmo **BM**), ela pode produzir saltos negativos, como mostra essa mesma figura.

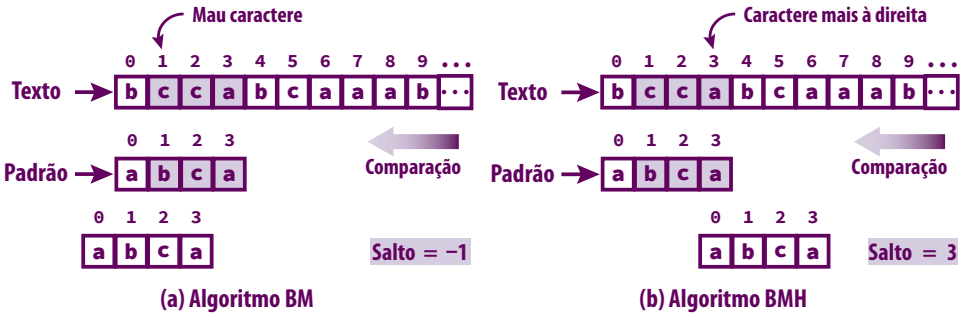


FIGURA 9–27: ALGORITMO BM VS ALGORITMO BMH

A **Figura 9–28** mostra um exemplo de criação de uma tabela de saltos do algoritmo **BMH**. Neste exemplo, o padrão é `"AGCCAGC"` e o alfabeto considerado é o alfabeto genético. Note que, na construção da tabela, levam-se em consideração informações sobre o padrão e sobre o alfabeto, e não apenas sobre o padrão, como ocorre com o algoritmo **KMP**. Note ainda que o tamanho da tabela de saltos é igual ao tamanho do alfabeto (como ocorre com a tabela de maus caracteres do algoritmo **BM**) e que o último caractere do padrão não é levado em consideração (o que não ocorre com uma tabela de maus caracteres).

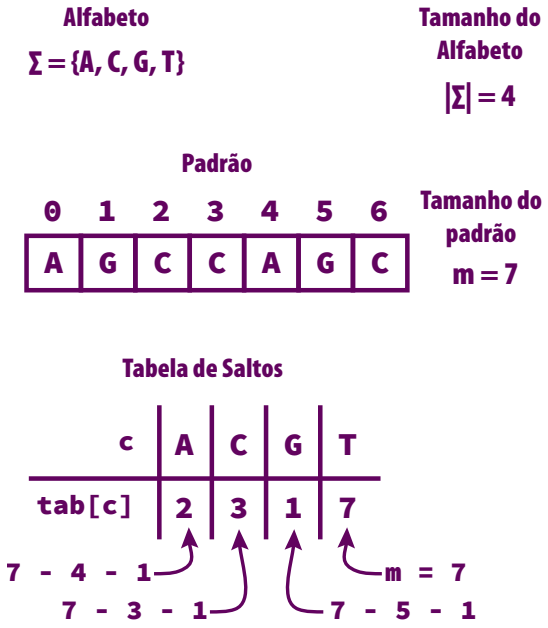


Figura 9–28: Exemplo de Tabela de Saltos do Algoritmo BMH

A **Figura 9–29** mostra um exemplo completo de casamento utilizando o algoritmo **BMH**. Note que o padrão e o texto nesse exemplo são os mesmos do exemplo da **Figura 9–24** que explora o algoritmo **BM**. Enfim observe que o número de comparações é o mesmo nesses dois exemplos.

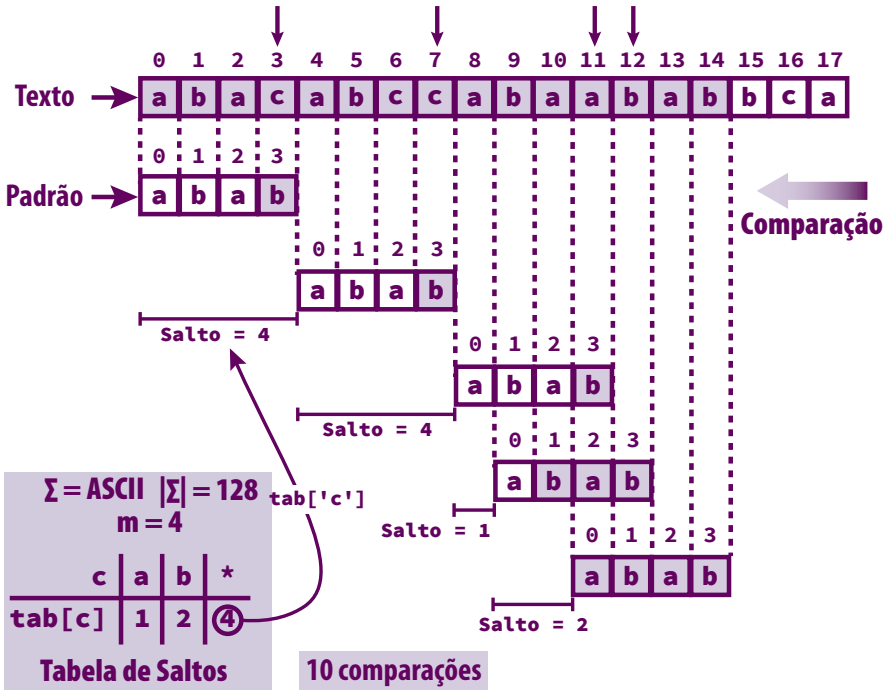


Figura 9–29: Exemplo de Casamento Usando o Algoritmo BMH

9.5.2 Algoritmos

O algoritmo **BMH** é apresentado na **Figura 9–30**.

ALGORITMO BMH

ENTRADA: O texto t de comprimento n, o padrão p a ser procurado de comprimento m

SAÍDA: A posição no texto da primeira ocorrência do padrão, se ele for encontrado no texto, ou um valor negativo, em caso contrário

1. Armazene num array (ts) a tabela de saltos obtida utilizando o algoritmo **CRIATABELADESALTOSBMH**
2. Atribua a i o índice do primeiro caractere do texto
3. Enquanto $i \leq n - m$, faça:
   - **3.1** Atribua a j a posição do último caractere do padrão
   - **3.2** Enquanto $j \geq 0$ e os caracteres na posição j do padrão e $i + j$ do texto são iguais, decremente j
   - **3.3** Se $j < 0$, retorne o valor de i
   - **3.4** Atribua a c o caractere que se encontra na posição $i + m - 1$ do texto
   - **3.5** Acrescente a i o valor de $ts[c]$
4. Retorne -1

FIGURA 9–30: ALGORITMO BMH

O algoritmo **CRIATABELADESALTOSBMH** invocado pelo algoritmo **BMH** é exibido na **Figura 9–31**.

ALGORITMO CRIATABELADESALTOSBMH

ENTRADA: Um string p de comprimento m e o tamanho $|\sum|$ do alfabeto

SAÍDA: A tabela de saltos (ts) criada

1. Crie um array ts de tamanho $|\sum|$
2. Atribua m a cada elemento do array ts
3. Para cada caractere c do padrão, atribua $m - i - 1$ a $ts[c]$

FIGURA 9–31: ALGORITMO DE CRIAÇÃO DE TABELA DE SALTOS

9.5.3 Implementação

A função `TabelaDeSaltosBMH()` exibida abaixo cria uma tabela de saltos usada pelo algoritmo **BMH** de acordo com aquilo que foi discutido acima. Essa função retorna o endereço da tabela criada e seus parâmetros são:

- `p` (entrada) — string que representa o padrão
- `m` (entrada) — tamanho padrão
- `tamAlfabeto` (entrada) — tamanho do alfabeto

```
int *TabelaDeSaltosBMH(const char *p, int m, int tamAlfabeto)
{
   int i, *tab, caractere;

      /* Aloca espaço para a tabela e checa alocação */
   tab = malloc(tamAlfabeto*sizeof(int));
   ASSEGURA(tab, "Impossivel alocar tabela de saltos");

      /* Inicia o valor do salto de cada caractere */
      /* do alfabeto com o tamanho do padrão       */
   for (i = 0; i < tamAlfabeto; ++i)
      tab[i] = m;

      /* Determina o valor do salto de cada caractere  */
      /* do padrão de acordo com sua posição no padrão */
```

```
   for (i = 0; i < m - 1; ++i) {
      caractere = p[i];
      tab[caractere] = m - i - 1;
   }

   return tab;
}
```

A função `CasamentoBMH()` verifica se um padrão (representado pelo parâmetro `p`) ocorre num texto (representado pelo parâmetro `t`) usando o algoritmo de Boyer, Moore e Horspool (**BMH**). O parâmetro `tamAlfabeto` representa o tamanho do alfabeto usado pelo padrão e pelo texto. Essa função retorna a posição (índice) do padrão no texto, se ele for encontrado, ou `-1`, caso contrário.

```
int CasamentoBMH(const char *t, const char *p, int tamAlfabeto)
{
   int i = 0, /* Índice do primeiro caractere do texto alinhado com o padrão */
       j,
      *tabela, /* Tabela de saltos */
       m = strlen(p), /* Tamanho do padrão */
       n = strlen(t), /* Tamanho do texto */
       c; /* Último caractere do texto alinhado com o padrão */

      /* Fase 1: Pré-processamento */
   tabela = TabelaDeSaltosBMH(p, m, tamAlfabeto);

      /* Fase 2: Casamento */
   while (i <= n - m) {
         /* Compara, da direita para a esquerda, os caracteres alinhados do */
         /* padrão e do texto até todos os caracteres do padrão terem sido  */
         /* comparados ou até encontrar uma discordância                    */
      for (j = m - 1; j >= 0 && t[i + j] == p[j]; --j)
         ; /* Vazio */

      if (j < 0) /* Verifica se houve casamento */
         return i; /* Padrão encontrado no texto */

         /* Obtém o último caractere do texto alinhado com o padrão */
      c = t[i + m - 1];

      i += tabela[c]; /* Calcula o tamanho do próximo salto */
   }

   return -1; /* O padrão não foi encontrado no texto */
}
```

9.5.4 Análise

Teorema 9.4: O custo espacial do algoritmo **BMH** é $\theta(|\sum|)$, sendo $|\sum|$ o tamanho do alfabeto.

Prova: O algoritmo **BMH** usa uma tabela (array) de tamanho igual a $|\sum|$. Portanto o custo espacial desse algoritmo é $\theta(|\sum|)$. ■

Teorema 9.5: O custo temporal da fase de pré-processamento do algoritmo **BMH** é $\theta(m + |\sum|)$, sendo $|\sum|$ o tamanho do alfabeto e m o tamanho do padrão.

Prova: O custo temporal do **Passo 2** do algoritmo de criação da tabela de saltos é $\theta(|\sum|)$ e o custo temporal do **Passo 3** desse algoritmo é $\theta(m)$. Logo o custo temporal desse algoritmo é $\theta(m + |\sum|)$. ■

No pior caso, o custo temporal do algoritmo **BMH** é $\theta(m \cdot n)$, sendo m o tamanho do padrão e n o tamanho do texto. No caso médio, o algoritmo **BMH** apresenta custo temporal $\theta(n)$. As provas dessas afirmações podem ser encontradas no artigo de Baeza-Yates e Régnier (1992) (v. **Bibliografia**).

O algoritmo **BMH** tem seu melhor desempenho quando é usado com padrões longos, sempre que ele consistentemente encontra um caractere dissonante no final ou próximo do caractere final da posição corrente no texto e o caractere final do padrão não ocorre em nenhum outro local dentro do padrão. Por exemplo, a tentativa de casamento de um padrão com *32* caracteres terminando com `'z'` com um texto com *255* caracteres no qual não aparece `'z'` custaria *224* comparações de caracteres. O pior caso do algoritmo de Horspool semelhante a esse exemplo é um padrão de um caractere `'a'` seguido por *31* caracteres `'z'` num texto consistindo de *255* caracteres `'z'`. Isso acarretará *31* comparações de caracteres bem-sucedidas, uma comparação de um caractere que falha e então passa adiante um caractere. Esse processo irá se repetir *223* mais vezes (*255* – *32*), de modo que o número total de comparações será *7.168* (*32*×*224*).

Assim o pior caso do algoritmo de Horspool ocorre quando o salto prescrito pela tabela de saltos é consistentemente pequeno e uma grande porção do padrão casa com o texto. Nesse caso, o custo é bem maior do que aquele apresentado pelo algoritmo original de Boyer e Moore.

9.6 Algoritmo de Karp e Rabin (KR)

9.6.1 Visão Geral

O **algoritmo de Karp e Rabin** (abreviadamente, **algoritmo KR**) é um algoritmo de casamento de strings desenvolvido por Richard Karp e Michael Rabin em 1987 (v. **Bibliografia**). Esse algoritmo tem como destaque principal o uso de dispersão, que foi estudada exaustivamente na implementação de tabelas de busca no **Capítulo 7**.

O uso de dispersão no algoritmo **KR** consiste inicialmente em aplicar uma função de dispersão ao padrão que se deseja procurar de modo a obter um valor de dispersão. Então essa função é aplicada a cada janela de texto e o valor obtido é comparado com o valor de dispersão do padrão. Se eles forem iguais, o casamento precisa ser confirmado comparando-se os caracteres, assim como faz o algoritmo **FB**. Essa comparação parece ser redundante, mas ela é realmente necessária, pois existe a possibilidade de ocorrência de um **casamento falso** (v. adiante).

Uma ideia inicial, mas bastante ingênua, para obtenção de um valor de dispersão associado a um string resume-se em simplesmente somar os valores dos caracteres que constituem o string no código de caracteres utilizado. Por exemplo, o valor de dispersão do string `"bola"` seria obtido simplesmente como[3]:

$$f(\text{"bola"}) = \text{'b'} + \text{'o'} + \text{'l'} + \text{'a'}$$

Essa singela ideia, entretanto, não funciona, pois qualquer string obtido a partir da permutação dos caracteres de outro string resultaria num mesmo valor de dispersão devido à comutatividade da operação de adição. Ou seja, dois strings que contêm os mesmos caracteres sempre resultarão em colisão quando essa técnica aditiva é usada.

O algoritmo **KR** é conhecido como algoritmo de **casamento por impressão digital** porque ele usa uma pequena quantidade de informação (i.e., um valor de dispersão, que é tal qual uma *impressão digital*) para representar um padrão relativamente grande. Então ele procura essa impressão digital (valor de dispersão) no texto.

O algoritmo **KR** baseia-se no fato de dois strings iguais apresentarem sempre os mesmos valores de dispersão. Portanto, em princípio, para resolver o problema de casamento, bastaria encontrar um substring no texto que possuísse o mesmo valor de dispersão que o padrão que se tenta encontrar. No entanto, existem dois problemas com essa abordagem básica que precisam ser resolvidos para que essa ideia prospere.

O primeiro problema é que, como foi visto acima, pode ser que dois strings diferentes resultem no mesmo valor de dispersão. Isso ocorre porque, como tipicamente há muitas janelas a ser comparadas, para manter os valores de dispersão pequenos e assim evitar a ocorrência de overflow, colisões são praticamente inevitáveis. Portanto,

[3] Aqui, o valor de um caractere entre plicas resulta no valor inteiro associado a ele no código de caracteres utilizado, assim como ocorre em C. Os valores utilizados neste livro são obtidos na implementação de C utilizada pelo autor. Em outra implementação esses valores podem ser diferentes, mas esse fato não invalida o raciocínio desenvolvido aqui.

quando o valor de dispersão do padrão coincide com o valor de dispersão de alguma janela do texto, deve-se checar se ocorreu de fato um casamento. Essa verificação é efetuada por meio de força bruta (v. **Seção 9.2**). Assim, se ocorrerem muitas colisões, haverá muitas comparações entre o padrão e janelas do texto e, como cada comparação tem custo temporal $\theta(m)$, o algoritmo terá custo temporal igual àquele do algoritmo de força bruta.

A solução para o primeiro problema apontado acima é o uso de uma boa função de dispersão, como aquela conhecida como (**impressão**) **digital de Rabin**. Essa função trata cada substring como um valor inteiro expresso em alguma base numérica e considera o resto da divisão do valor assim obtido por um número primo relativamente grande. Formalmente, o cálculo do valor de dispersão do string s indexado de 0 a $m - 1$ pode ser expresso como:

$$f(s[0..m-1]) = (c_0 b^{m-1} + c_1 b^{m-2} + \ldots + c_{m-1} b^0) \bmod q \qquad \textbf{Fórmula 9–1}$$

Nessa fórmula, c_i $(0 \leq i < m)$ é o valor inteiro do caractere que se encontra no índice i do string, $b > 1$ é a aludida base numérica, q é o número primo mencionado e *mod* denota o resto da divisão (que, em C, é representado pelo operador `%`). Por exemplo, o valor de dispersão do string `"bola"` calculado de acordo com a digital de Rabin usando 2 como base numérica e o número primo 1009 poderia ser[4]:

$$f(\text{"bola"}) = ('b' \cdot 2^3 + 'o' \cdot 2^2 + 'l' \cdot 2^1 + 'a' \cdot 2^0) \bmod 1009 = 532$$

É importante observar que, qualquer que seja a base numérica considerada, a fórmula de cálculo de valores de dispersão usados pela digital de Rabin leva em consideração a posição de cada caractere no string, de modo que strings constituídos pelos mesmos caracteres em posições distintas são associados a valores de dispersão distintos.

Nos exemplos apresentados aqui, a base é arbitrariamente escolhida como sendo igual a 2, mas é aconselhável que a base usada no cálculo de valor de dispersão seja igual a $|\sum|$ (i.e., do tamanho do alfabeto sob consideração). Se o valor de b for menor do que o tamanho do alfabeto, poderão ocorrer muitas colisões triviais. Além disso, os caracteres devem ser mapeados no intervalo $[0..b-1]$. Se você não seguir essas recomendações, poderá encontrar sérias dificuldades na implementação do algoritmo **KR**.

Do ponto de vista matemático, a **Fórmula 9–1** produz resultados rigorosamente corretos, mas a implementação direta dessa fórmula pode resultar em valores de dispersão indesejados, como mostra a função `DigitalRabinErrada()` a seguir. Essa função calcula e retorna, propositalmente de modo incorreto e ineficiente, o valor de dispersão de um string e seus parâmetros são:

- `s` (entrada) — o string
- `b` (entrada) — a base numérica utilizada no cálculo
- `q` (entrada) — o número primo usado no cálculo

```
int DigitalRabinErrada(const char *s, int b, int q)
{
   int i,   /* Índice de um caractere do string */
       grau, /* Grau do polinômio */
       potencia = 1, /* Base elevada a uma potência */
       valor = 0; /* Valor de dispersão a ser retornado */

   grau = strlen(s) - 1; /* Calcula o grau do polinômio */

   for (i = grau; i >= 0; --i) { /* Obtém o valor do polinômio */
      valor += s[i]*potencia;
      potencia *= b;
   }
   return valor%q; /* Retorna o valor de dispersão */
}
```

[4] Esses valores foram escolhidos arbitrariamente com o propósito de facilitar o entendimento e não devem ser imitados na prática.

A função `DigitalRabinErrada()` é uma implementação direta da **Fórmula 9–1** discutida acima e funciona razoavelmente bem com strings curtos, como `"bola"`, por exemplo. Acontece, porém, que essa função não funciona adequadamente quando o comprimento do string que ela recebe como primeiro parâmetro é muito grande, como ocorre, por exemplo, com o string `"anticonstitucionalissimamente"`. De fato, quando essa função é chamada tendo como primeiro parâmetro esse string, ela retorna um valor de dispersão negativo. Mas como esse resultado é possível? Afinal, a função obtém esse valor somando termos que são sempre positivos.

O que ocorre quando a função `DigitalRabinErrada()` recebe um string de comprimento tão grande como parâmetro é aquilo que já era esperado: overflow. Antes de conhecer a solução adequada para o problema apresentado por essa função, é conveniente que se entenda os papéis desempenhados pelo resto da divisão (*mod*) por um número primo q utilizado na **Fórmula 9–1**, que são dois:

[1] Evitar a ocorrência de overflow. Isso independe do fato de q ser primo, mas requer que seu valor seja tal que mantenha os valores calculados abaixo do maior valor do tipo inteiro usado para representá-los. Na função `DigitalRabinErrada()`, esse tipo é **int**, mas substituí-lo por um tipo mais largo (p. ex., **long long**) não resolve definitivamente o problema.

[2] Reduzir o número de casamentos falsos. Em outras palavras, deseja-se distribuir uniformemente os valores de dispersão de modo a reduzir o número de colisões. Uma maneira de obter essa distribuição é usar um número primo relativamente grande.

Em resumo, o valor de q não pode ser tão grande que deixe de evitar overflow nem tão pequeno que permita muitas colisões.

Um problema com a função `DigitalRabinErrada()` é que, quando o operador **%** é aplicado, um eventual overflow já terá ocorrido. Outro problema com essa função é sua ineficiência. Quer dizer, se você observar atentamente a **Fórmula 9–1**, constatará que ela se trata simplesmente da avaliação para um valor igual à base numérica em questão de um polinômio de grau $m - 1$ cujos coeficientes são os valores inteiros atribuídos aos caracteres do string.

Provavelmente, você já deve ter estudado o famoso **método de Horner** que, entre outras coisas, permite calcular o valor numérico de qualquer polinômio utilizando um procedimento simples e eficiente[5]. O método de Horner é eficiente nesse caso porque ele não requer que as potências dos termos do polinômio sejam explicitamente calculadas, de modo que as instruções que envolvem a variável `potencia` na função `DigitalRabinErrada()` podem ser dispensadas.

Utilizando esse conhecimento matemático, a **Fórmula 9–1** pode ser corretamente implementada em C como:

```
int DigitalRabin(const char *s, int b, int q)
{
   int i, valor = 0,
       grau = strlen(s) - 1; /* Grau do polinômio */

   for (i = 0; i <= grau; ++i)
      valor = (valor*b + s[i])%q;
   return valor%q;
}
```

Outro problema que não é aparente na função `DigitalRabin()` diz respeito ao próprio cálculo dos valores de dispersão dos substrings (janelas) que constituem o texto no qual as buscas serão efetuadas. Quer dizer, como esses valores dependem dos valores dos caracteres que constituem esses substrings, o custo temporal da aplicação de uma função de dispersão é $\theta(m)$, em que m é o tamanho do padrão (e de cada janela de texto). Mas, se esse for o caso e se for necessário calcular um valor de dispersão para cada substring (janela) do texto, então

[5] Se você desconhece o método de Horner ou a implementação dele, consulte o **Apêndice B** do **Volume 1**.

o algoritmo até então descrito é praticamente equivalente ao algoritmo de força bruta. Aliás, isso tornaria o algoritmo proposto um pouco pior do que o algoritmo **FB**, visto que calcular os valores de dispersão do padrão e do texto envolve muito mais processamento do que simplesmente comparar os mesmos caracteres. O mesmo método usado para cálculo do valor de dispersão do padrão pode ser usado para cálculo dos valores de dispersão das janelas que serão comparadas, mas o custo temporal corresponderia ao número de operações de multiplicação, soma e resto de divisão para cada janela do texto, o que resultaria num total de $n \cdot m$ operações no pior caso. Novamente, não há nenhuma evolução com relação ao algoritmo **FB**.

A solução do problema exposto no parágrafo precedente constitui o grande insight do algoritmo de Karp e Rabin e consiste em usar uma **função de dispersão rolante**. Tal função permite que se calcule o valor de dispersão de uma janela do texto a partir do valor de dispersão da janela precedente do mesmo texto. Ou seja, para um dado padrão, usando-se uma função de dispersão rolante, apenas o cálculo do valor de dispersão da primeira janela de um texto terá custo temporal $\theta(m)$, visto que ela não possui janela precedente. As demais janelas terão seus valores de dispersão calculados com custo temporal $\theta(1)$.

A impressão digital de Rabin é uma função de dispersão que pode ser efetivamente usada como função de dispersão rolante. Para tal, deve-se subtrair o valor do termo correspondente ao primeiro caractere do substring precedente e acrescentar um termo novo correspondente ao último caractere do substring corrente utilizando a **Fórmula 9–2**:

$$f(s') = (b \cdot (f(s) - s[0] \cdot b^{m-1}) + s'[m-1] \cdot b^0) \bmod q \qquad \textbf{Fórmula 9–2}$$

Nessa fórmula, tem-se que:

- s' e s representam, respectivamente, a janela corrente e a janela precedente.
- $f(s')$ e $f(s)$ são os valores de dispersão das janelas corrente e precedente, respectivamente.

Na **Fórmula 9–2**, o termo:

$$b \cdot (f(s) - s[0] \cdot b^{m-1})$$

corresponde à remoção da contribuição do primeiro caractere da janela precedente para o valor de dispersão dessa janela. Nessa remoção, a multiplicação pela base b aparece porque os caracteres restantes na janela precedente passam a ocupar uma posição anterior na janela corrente. Por exemplo, considerando `"bol"` como a janela precedente s e `"ola"` como a janela corrente s', o caractere `'o'` ocupa a posição *1* na janela s e a posição *0* na janela s'. Assim o expoente desse caractere, que era *1* no cálculo do valor de dispersão da janela s, passa a ser *2* no cálculo do valor de dispersão da janela s', como mostra a **Figura 9–32**.

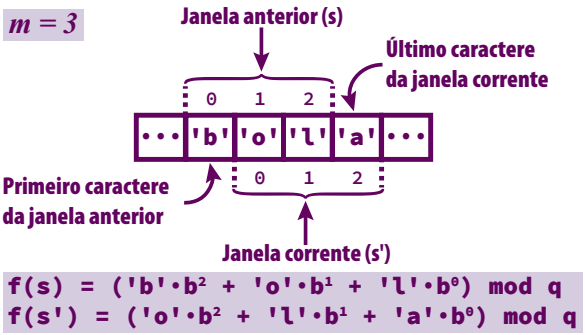


Figura 9–32: Uso de Dispersão Rolante no Algoritmo KR

Na **Fórmula 9–2**, o termo:

$$s'[m-1].b^0$$

representa o acréscimo da contribuição do último caractere da janela corrente no cálculo do valor de dispersão dessa janela.

É muito importante notar que o valor da expressão:

$$b \cdot (f(s) - s[0] \cdot b^{m-1})$$

pode ser negativo, o que pode parecer estranho à primeira vista. Afinal, como essa expressão pode eventualmente ser negativa se ela representa o valor de dispersão de um string menos a contribuição de um caractere desse mesmo string para esse valor? Não seria o caso de se estar afirmando que uma porção é maior do que o todo do qual ela faz parte? De fato, essas questões são intrigantes se você esqueceu que na realidade *f(s)* não representa um *todo*, pois ele foi obtido como um resto de divisão por meio da **Fórmula 9–1**. Portanto o resultado dessa última expressão pode realmente ser negativo e, como consequência, o valor da expressão:

$$b \cdot (f(s) - s[0] \cdot b^{m-1}) + s'[m-1] \cdot b^{0}$$

também pode ser negativo.

Esse último fato não constitui nenhum problema do ponto de vista conceitual, visto que, em aritmética modular (euclidiana), o resultado de uma operação como:

$$a \bmod b$$

em que $b > 0$, resulta sempre num valor r, sendo que $0 \leq r < |b|$.

Acontece, porém, que raramente o operador *mod* da aritmética modular é implementado dessa maneira em linguagens de programação. Por exemplo, em C, o operador é representado pelo operador **%,** que nem sempre resulta no valor esperado com o uso de *mod*. Precisamente, em compiladores que seguem os padrões mais recentes da linguagem C (C99 e C11)[6], o resultado da divisão inteira é obtido como se o quociente da divisão real dos respectivos operandos tivesse sua parte fracionária descartada. Por exemplo, o quociente da divisão real de *–17* por *5* é *–3,4*; desprezando-se a parte fracionária desse resultado, obtém-se *–3*, que é o quociente da divisão inteira *–17/5*. Tendo calculado o resultado de uma divisão inteira, o resto dessa mesma divisão pode ser obtido por meio da fórmula:

$$resto = numerador - quociente \times denominador$$

Assim, utilizando essa última fórmula e o resultado obtido para *–17/5*, tem-se que:

–17%5 é igual a $-17 - -3 \times 5$, que é igual a $-17 + 15$, que é igual a *–2*

Em contraste, o resultado dessa operação obtido com o operador *mod* da aritmética modular euclidiana seria *3*, pois $5 \times (-4) + 17$ é igual a *3*. É fácil mostrar que, em geral, quando o resultado de *x%y* é negativo e *y* é positivo, obtém-se o resultado (euclidiano) desejado somando-se esse resultado com *y*.

Outras propriedades importantes da operação *mod* [que, aliás, já foram empregadas na implementação da função `DigitalRabin()` exibida acima] são as seguintes:

- $(x + y) \bmod z$ é equivalente a $[(x \bmod z) + (y \bmod z)] \bmod z$
- $(x \cdot y) \bmod z$ é equivalente a $[(x \bmod z) \cdot (y \bmod z)] \bmod z$

De acordo com essas propriedades, cada operação de soma ou multiplicação deve ser seguida por uma redução de valor usando *mod* para tornar os resultados intermediários pequenos e eliminar a possibilidade de overflow (desde que o valor de *q* utilizado no cálculo do resto da divisão não seja grande demais).

Para afastar a possibilidade de ocorrência de overflow no cálculo de valores de dispersão, deve-se levar em consideração que cada parcela da **Fórmula 9–2** deve caber numa variável do tipo inteiro utilizado. Ou seja, supondo que *max* seja o maior valor desse tipo, o seguinte raciocínio é empregado para determinar qual deve ser o maior valor do número primo que pode ser utilizado:

1. A maior parcela da **Fórmula 9–2** é $b \cdot s[0] \cdot b^{m-1}$.

[6] Em padrões anteriores, a situação era ainda pior porque o resultado não era portável se o numerador ou o denominador fosse negativo. Para uma discussão completa sobre esse assunto, sugere-se que o leitor consulte *Programando em C: Volume 1 – Fundamentos (2008)* deste autor (v. **Bibliografia**).

2. Como, seguindo as recomendações expostas acima, os valores inteiros associados aos caracteres do alfabeto em questão são restritos ao intervalo $[0..b-1]$, o maior valor de s[0] é $b-1$.
3. Como o valor de b^{m-1}, que é a maior potência do polinômio usado para calcular valores de dispersão, é determinado utilizando-se *mod q*, o maior valor de b^{m-1} é $q-1$.
4. Supondo que *max* seja o maior valor do tipo inteiro utilizado e considerando os argumentos precedentes, tem-se:

$$b \cdot (b-1) \cdot (q-1) \leq max \Rightarrow q \leq max/(b^2 - b) + 1$$

Logo, para evitar uma eventual ocorrência de overflow, o valor de q deve ser o maior número primo que satisfaça essa última relação.

9.6.2 Algoritmo

A **Figura 9–33** apresenta o algoritmo **KR**.

ALGORITMO KR

ENTRADA: O texto *t* de comprimento *n* e o padrão *p* a ser procurado de comprimento *m*

SAÍDA: A posição no texto da primeira ocorrência do padrão, se ele for encontrado no texto, ou um valor negativo, em caso contrário

1. Calcule o valor de dispersão do padrão (*dp*)
2. Calcule o valor de dispersão da primeira janela do texto (*dt*)
3. Atribua a *i* o índice do primeiro caractere do texto
4. Enquanto $i < n$, faça:
   - 4.1 Se $dp = dt$ e *p* casar com a janela de texto que começa em *i*, retorne *i*
   - 4.2 Incremente *i*
   - 4.3 Atribua a *dt* o valor de dispersão da janela de texto que começa em *i*
5. Retorne um valor negativo

FIGURA 9–33: ALGORITMO KR

Os cálculos dos valores de dispersão aos quais o algoritmo da **Figura 9–33** faz referência usa o algoritmo de Horner descrito no **Apêndice B** do **Volume 1**.

9.6.3 Implementação

A função `CasamentoKR()` apresentada abaixo verifica se um padrão ocorre num texto usando o algoritmo **KR**. Ela implementa o arrazoado apresentado acima e seus parâmetros são:

- `t` (entrada) — string que representa o texto
- `p` (entrada) — string que representa o padrão
- `b` (entrada) — a base numérica utilizada
- `q` (entrada) — o número primo utilizado

```
int CasamentoKR(const char *t, const char *p, int b, int q)
{
   int dispersaoPadrao = 0, /* Valor de dispersão do padrão */
       dispersaoTexto = 0, /* Valor de dispersão de cada janela */
       maiorPotencia = 1,   /* Valor da base elevado a m - 1 */
       i, /* Índice da primeira ocorrência do padrão no texto encontrada */
       m = strlen(p), /* Tamanho do padrão */
       n = strlen(t); /* Tamanho do texto  */
```

```
      /* Se o tamanho do texto for menor do que o tamanho */
      /* do padrão, o padrão jamais será encontrado       */
   if (n < m)
      return -1;

      /* Calcula a maior potência do polinômio */
   for (i = 0; i < m - 1; ++i)
      maiorPotencia = maiorPotencia*b%q;

      /* Calcula o valor de dispersão do padrão e da primeira janela do texto */
   for (i = 0; i < m; ++i) {
      dispersaoPadrao = (dispersaoPadrao*b + p[i])%q;
      dispersaoTexto = (dispersaoTexto*b + t[i])%q;
   }

      /* Efetua a busca pelo padrão no texto */
   for (i = 0; i < n; ++i) {
         /* Se os valores de dispersão do padrão e da janela coincidem, */
         /* verifica se os strings correspondentes realmente são iguais */
      if (dispersaoPadrao == dispersaoTexto && !memcmp(p, t+i, m))
         return i; /* Padrão encontrado no texto */

         /* Calcula o valor de dispersão da próxima janela */
      dispersaoTexto = ( b*(dispersaoTexto - t[i]*maiorPotencia) + t[i + m] )%q;

         /* Corrige o valor da dispersão se ele ficou negativo */
      if (dispersaoTexto < 0)
         dispersaoTexto += q;
   }

   return -1; /* Se ainda não houve retorno, o padrão não foi encontrado */
}
```

A função **main**() a seguir pode ser utilizada para testar a implementação do algoritmo **KR**. Nessa função, `BASE` e `PRIMO` são constantes simbólicas previamente definidas que representam, respectivamente, os valores de b e q discutidos acima. Note que essa função testa os valores dessas constantes para verificar se é seguro chama a função `CasamentoKR()` sem que ocorra overflow. A função `MaiorPrimo()` chamada por **main**() retorna o maior número primo menor do que ou igual ao valor recebido como parâmetro e é relativamente simples de implementar.

```
int main(void)
{
   char *texto = "caabaccabacabaabbabacaabaccabacabaabbabaca";
   char *padrao = "abacaabacc";
   int   posicao,
         b = BASE,
         q = PRIMO,
         qMax = INT_MAX/(BASE*BASE - BASE);

      /* Teste preventivo de overflow */
   if (q > qMax) {
      qMax = MaiorPrimo(qMax);
      printf("\n\t\a>>> Ocorrera' overflow!\n"
             "\n\t>>> O maior valor de q deveria ser %d\n", qMax);
      return 1;
   }

   printf("\n>>>> Usando Algoritmo KR <<<\n");
```

```
   posicao = CasamentoKR(texto, padrao, b, q);

   if (posicao >= 0)
      printf( "\n>>> \"%s\" encontrado em \"%s\" na posicao %d\n",
              padrao, texto, posicao );
   else
      printf( "\n>>> \"%s\" nao foi encontrado em \"%s\"\n", padrao, texto );

   return 0;
}
```

9.6.4 Análise

Lema 9.3: O custo temporal da fase de pré-processamento do algoritmo **KR** é $\theta(m)$.

Prova: Em qualquer caso, os dois primeiros laços da função `CasamentoKR()`, que constituem a fase de pré-processamento do algoritmo **KR**, são executados m vezes, de modo que o custo temporal dessa fase do algoritmo **KR** tem custo temporal $\theta(m)$. ■

Teorema 9.6: No melhor caso, o algoritmo **KR** tem custo temporal $\theta(m + n)$.

Prova: No melhor caso, o custo de processamento de cada uma das $n - m + 1$ janelas de texto (v. **Lema 9.1**) é $\theta(1)$. Logo o custo temporal da fase de processamento nesse caso é $\theta(n - m)$. Como o custo temporal de pré-processamento é $\theta(m)$, de acordo com o **Lema 9.3**, o custo temporal do algoritmo **KR** inteiro, no melhor caso, é $\theta(m + n)$. ■

Teorema 9.7: No pior caso, o algoritmo **KR** tem custo temporal $\theta(m \cdot (n - m))$.

Prova: No pior caso, ocorrem colisões nas $n - m + 1$ janelas de texto. O restante da prova é similar àquela do **Teorema 9.1**. ■

Na prática, o pior caso desse algoritmo é muito raro de acontecer (v. abaixo). Esse caso ocorre quando o valor de dispersão do padrão e aquele de cada janela do texto são iguais e o padrão em si não casa com nenhuma dessas janelas. A **Figura 9–34** ilustra essa situação. No caso ilustrado nessa figura, o custo temporal de processamento é $\theta(m \cdot (n - m))$, visto que todas as $n - m + 1$ janelas do texto são comparados [e cada comparação tem custo temporal $\theta(m)$], pois eles resultam no mesmo valor de dispersão. Contudo o programador precisará se esforçar muito para criar uma função de dispersão tão ruim...

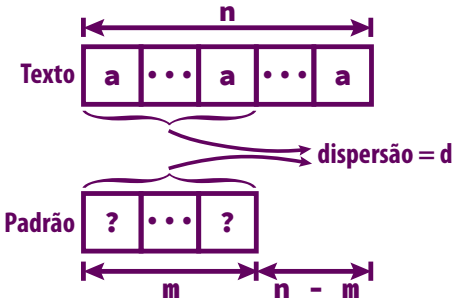


Figura 9–34: Pior Caso do Algoritmo KR

Para casamentos envolvendo um texto e um único padrão, o algoritmo de Karp e Rabin não é bom, em termos de custo temporal, a outros algoritmos tais como **KMP** (v. **Seção 9.3**) e **BM** (v. **Seção 9.4**). Por outro lado, o algoritmo **KR** é imbatível quando a tarefa consiste em verificar possíveis casamentos entre um conjunto de padrões de mesmo tamanho e um texto (v. **Seção 9.10.6**). Nesse caso, cria-se uma tabela contendo os valores de dispersão de todos os substrings do texto com o mesmo tamanho do padrão. O custo de criação dessa tabela é relativamente alto [i.e., $\theta(m \cdot n)$], mas, em compensação, a soma de todas as buscas por casamentos passa a ter custo $\theta(m + k)$, sendo k o número de padrões que serão testados. Isso ocorre porque, usando-se a tabela

de dispersão, pode-se verificar se o valor de dispersão de um dado padrão coincide com o valor de dispersão de um substring do texto com custo $\theta(1)$.

9.7 Comparando Algoritmos de Casamento de Strings

A **Tabela 9–3** apresenta um resumo das avaliações dos algoritmos de casamento de strings apresentados neste livro.

Algoritmo	Pré-processamento	Processamento		
		Melhor Caso	Caso Médio	Pior Caso
FB	*Não há*	$\theta(n - m)$	$\theta(m + n)$	$\theta(m \cdot (n - m))$
KMP	$\theta(m)$	$\theta(n)$	$\theta(n)$	$\theta(n)$
BM	$\theta(m + \lvert\sum\rvert)$	$\theta(n/m)$	$\theta(n)$	$\theta(m \cdot n)$
BMH	$\theta(m + \lvert\sum\rvert)$	$\theta(n/m)$	$\theta(n)$	$\theta(m \cdot n)$
KR	$\theta(m)$	$\theta(n - m)$	$\theta(m + n)$	$\theta(m \cdot (n - m))$

Tabela 9–3: Custos Temporais de Algoritmos de Casamento de Strings

Com exceção do algoritmo de força bruta discutido na **Seção 9.2**, todos os demais algoritmos de casamento de strings apresentados neste capítulo têm uma **fase de pré-processamento** do padrão que precede a **fase de processamento**, que é aquela na qual o algoritmo tenta encontrar um casamento do padrão com o texto. A **Tabela 9–3** mostra, para cada algoritmo discutido neste capítulo, o custo de pior caso de cada uma dessas fases. Obviamente, o custo total de cada algoritmo é a soma dos custos dessas respectivas fases.

É interessante notar que, na prática, esses custos nem sempre refletem a realidade. Por exemplo, no pior caso, algoritmo de Karp e Rabin tem custo igual a $\theta(m \cdot (n - m))$, o que parece indicar que ele não é melhor do que o algoritmo de força bruta, mas de fato o algoritmo **KR**, em média, é excelente em situações práticas.

Cada algoritmo de casamento de strings apresenta características atraentes dependendo do contexto em que são utilizados. O algoritmo **FB** é o mais fácil de implementar e funciona bem em casos comuns. O algoritmo **KMP** tem custo temporal linear garantido e não requer retrocesso. O algoritmo **BM** tem custo temporal sublinear no melhor caso. Esse último algoritmo é bem mais eficiente do que o algoritmo **BMH** para pequenos alfabetos, mas sua regra de bom sufixo não é fácil de implementar e isso faz com que o algoritmo **BMH** seja preferido em detrimento ao algoritmo **BM**.

Os algoritmos **BMH** e **KMP** aceleram a operação de casamento de um padrão com um texto pré-processando o padrão para determinar o tamanho de cada salto após ocorrer um descasamento. Em algumas aplicações, contudo, é interessante adotar uma abordagem complementar, na qual se consideram algoritmos de casamento que pré-processam o texto para dar suporte a consultas múltiplas. Essa abordagem é conveniente para aplicações nas quais uma série de consultas sejam executadas sobre um texto fixo, de modo que o custo inicial de pré-processamento do texto é compensado pela aceleração de cada consulta subsequente. Por exemplo, um site que permita ao usuário buscas sobre a obra de um determinado autor. O algoritmo **KR** permite facilmente a implementação dessa abordagem.

A **Tabela 9–4** resume as principais vantagens e desvantagens dos algoritmos de casamento de string discutidos neste capítulo.

Algoritmo	Prós	Contras
FB	❒ Fácil de implementar ❒ Útil em muitas situações triviais	❒ Muito lento ❒ Tem retrocesso
KMP	❒ Não apresenta retrocesso ❒ Muito rápido	❒ Usa espaço adicional com custo $\theta(m)$
BM	❒ Rápido	❒ Usa espaço adicional com custo $\theta(\|\sum\|)$ ❒ Não serve para casamento usando texto com fluxo contínuo ❒ A regra do bom sufixo não é fácil de implementar
BMH	❒ Rápido	❒ Usa espaço adicional com custo $\theta(\|\sum\|)$ ❒ Não serve para casamento usando texto com fluxo contínuo
KR	❒ Não usa espaço adicional ❒ Adequado para casamento de um texto fixo com vários padrões	❒ Difícil de implementar se o programador não dispõe de conhecimento matemático adequado

Tabela 9–4: Vantagens e Desvantagens de Algoritmos de Casamento de Strings

9.8 Tries

Os algoritmos de casamento vistos na última seção melhoravam suas eficiências pré-processando padrões (para determinar os valores dos saltos nos algoritmos **KMP** e **BMH** e para calcular valores de dispersão no algoritmo **KR**). Nesta seção, serão apresentados algoritmos de casamento de strings que pré-processam o texto, em vez do padrão. Essa abordagem é adequada para aplicações nas quais várias consultas são executadas num texto mantido fixo, de modo que o custo inicial de pré-processamento do texto é compensado por uma maior rapidez em cada busca subsequente. Um exemplo de tal situação seria um site que permite efetuar buscas por palavras na obra de Machado de Assis.

9.8.1 Conceitos

Trie é um tipo especializado de árvore multidirecional utilizado para armazenar pares chave/valor nos quais as chaves podem ser decompostas em partes menores, como, por exemplo, strings que podem ser decompostos em caracteres. Aliás, tries são tipicamente usadas para representar chaves que são strings. Desse modo, os caracteres que constituem uma chave são usados para guiar as buscas, como será visto adiante. Tries foram inventadas por Edward Fredkin em 1960 (v. **Bibliografia**).

Em geral, trie é um tipo de árvore que usa as partes constituintes de uma chave para guiar operações de busca, inserção e remoção[7]. Em particular, tries são usadas para armazenar strings de modo a suportar operações sobre eles de modo bastante eficiente. Nesse último caso, as principais operações que são facilitadas com o uso de tries são casamento de strings e casamento de prefixos. Essa última operação consiste em encontrar todos os strings no conjunto de strings armazenados numa trie que apresentem um determinado prefixo.

[7] A denominação trie é derivada de quatro letras da palavra *retrieval*, que significa recuperação (de informação, neste contexto) em inglês. Portanto *trie* não possui tradução em outra língua. Para evitar confusão com a palavra *tree*, que significa árvore em inglês, nativos dessa língua pronunciam *trai*. Em português, você pode pronunciar trie como bem desejar, pois não haverá confusão alguma..

Se um texto for grande, imutável e forem efetuadas operações frequentes de casamento com diferentes padrões (p. ex., busca por palavras na obra de Machado de Assis), é mais sensato pré-processar o texto em vez de pré-processar cada padrão. Assim, na prática, tries são estruturas de dados usadas para representar um conjunto de palavras de um texto. Desse modo, tries suportam casamento de strings em tempo proporcional ao tamanho do padrão. Tries também são usadas para armazenar grandes dicionários, em programas de verificação de ortografia e em processamento de linguagem natural. Outras aplicações de tries que merecem destaque são: compressão de dados, bioinformática (p.ex., casamento de segmentos de DNA), mecanismos de busca e autocompletação de código (encontrada em bons editores de programas).

Formalmente, uma **trie padrão** que armazena um conjunto S de n strings de comprimento total M de um alfabeto $\sum$ de tamanho $|\sum|$ é uma árvore multidirecional T que apresenta as seguintes propriedades:

- Cada nó de T, com exceção da raiz, é rotulado com um caractere de $\sum$.
- Os filhos de um nó de T são ordenados alfabeticamente; i.e., a ordenação desses filhos é determinada pela ordenação canônica do alfabeto $\sum$.
- T possui n nós finais (v. adiante), cada um dos quais associado a um string de S, de modo que a concatenação dos caracteres associados aos nós no caminho da raiz até um nó final f de T resulta no string de S associado com f.
- Cada nó interno de T pode ter entre 1 e $|\sum|$ filhos.
- A altura de T é igual ao comprimento do string mais longo do conjunto S.
- O número de nós de T é $\theta(M)$. O pior caso para o número de nós de uma trie ocorre quando não existe nenhum par de strings que compartilhe um prefixo comum. Ou seja, qualquer nó interno, com exceção da raiz, tem apenas um filho.
- O formato de uma trie é independente da ordem com que suas chaves são inseridas. Essa propriedade significa que existe apenas um formato de trie para um conjunto de chaves, independentemente da ordem na qual essas chaves aparecem. Como comparação, lembre-se que o formato de árvores binárias de busca depende da ordem na qual as chaves são inseridas (v. **Capítulo 4**).

Considere como exemplo, a trie da **Figura 9–35**, que armazena o seguinte conjunto de strings:

```
S = {"a", "da", "de", "do", "o", "que", "rata", "rato",
     "rei", "roeu", "roma", "rouba", "roupa"}
```

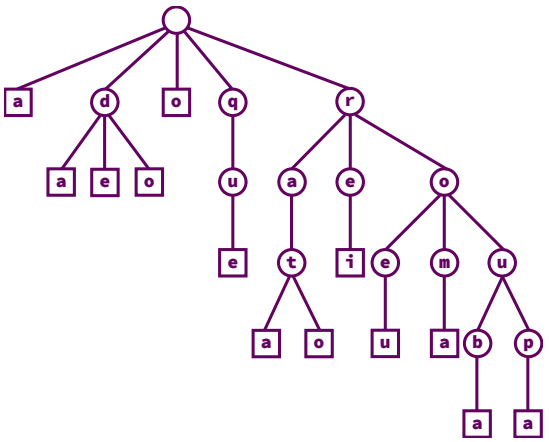


Figura 9–35: Exemplo de Trie Básica

Nesse último exemplo, o alfabeto em consideração é o alfabeto latino, mesmo que apenas algumas letras (símbolos) desse alfabeto sejam utilizadas na composição da árvore.

Se o alfabeto usado na construção de uma trie consistir de apenas dois caracteres, ela será essencialmente uma árvore binária. Por outro lado, se $|\sum| > 2$, então a trie será uma árvore multidirecional de ordem $|\sum|$. Apesar disso, é comum que vários nós internos de uma trie padrão tenham bem menos que $|\sum|$ filhos. Por exemplo, a trie apresentada na **Figura 9–35** possui vários nós internos com apenas um filho.

Note que, na **Figura 9–35**, nenhuma chave é prefixo de outra chave. Mas esse fato não constitui um problema para a forma de representação de tries proposta aqui. Na referida figura, cada nó retangular é denominado **nó final**, pois ele indica o final de uma chave (string). Nós finais não devem ser confundidos com nós folhas, que são nós cujos filhos são todos nulos, como foi visto nos **Capítulos 4** e **6**. De fato, todos os nós finais na **Figura 9–35** também são folhas, mas esse nem sempre é o caso, como mostra a **Figura 9–36**, na qual estão representados os string `"rei"` e `"reino"`, sendo que o primeiro deles é prefixo do segundo. Resumindo, todo nó-folha de uma trie é um nó final, mas nem todo nó final é um nó-folha.

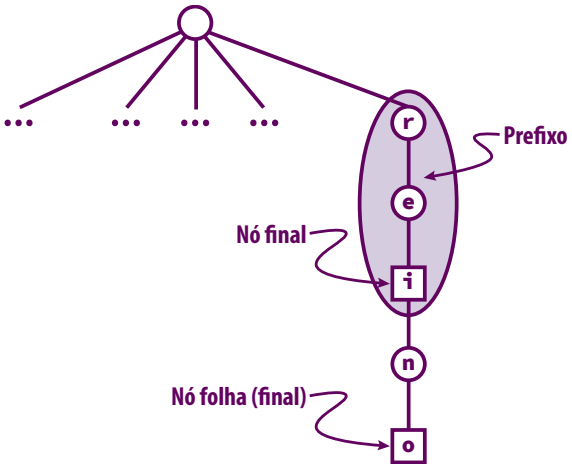


Figura 9–36: Representação de Prefixo em Trie

Busca

Uma busca por um string numa trie começa em sua raiz e segue o caminho indicado pela sequência determinada pelos caracteres do string procurado. Inicialmente, o primeiro caractere da chave é usado como índice para determinar qual é a ramificação a ser seguida. Se essa ramificação for nula, conclui-se que a chave não está na trie. Caso contrário, usa-se o segundo caractere da chave para determinar qual é a próxima ramificação a ser seguida no próximo nível, e assim por diante. A busca encerra quando se encontra um nó final (que pode ser folha ou não) que corresponda ao último caractere da chave de busca (string) ou quando não se encontra uma ramificação correspondente a um caractere do string. Se esse caminho pode ser seguido e termina num nó final, então pode-se concluir que o string procurado se encontra na trie. Por exemplo, o caminho de nós visitados numa busca pelo string `"roupa"` na **Figura 9–35** é determinado pela sequência de caracteres:

$$r \rightarrow o \rightarrow u \rightarrow p \rightarrow a$$

Assim, na trie ilustrada na **Figura 9–35**, seguindo-se esse caminho chega-se a um nó final que representa o caractere `'a'`, que é o último caractere desse string. Por outro lado, se o caminho determinado pelo string procurado não pode ser seguido até o último caractere do string, sendo esse caractere representado por um nó final da trie, então conclui-se que o string não se encontra na trie. Por exemplo, na trie da **Figura 9–35**, o caminho especificado pelo string `"querubim"` é interrompido na letra `'e'`, o que permite concluir que esse string não faz parte da trie. Mais um exemplo: o caminho determinado pelo string `"rom"` pode ser seguido até o final, mas o caminho termina num nó que não é final (i.e., nesse caminho, o último caractere `'m'` não é representado por um nó final).

A **Figura 9–37** mostra a busca bem-sucedida da chave `"roma"` numa trie, enquanto a **Figura 9–38** ilustra a busca malsucedida da chave `"quepe"` na mesma trie.

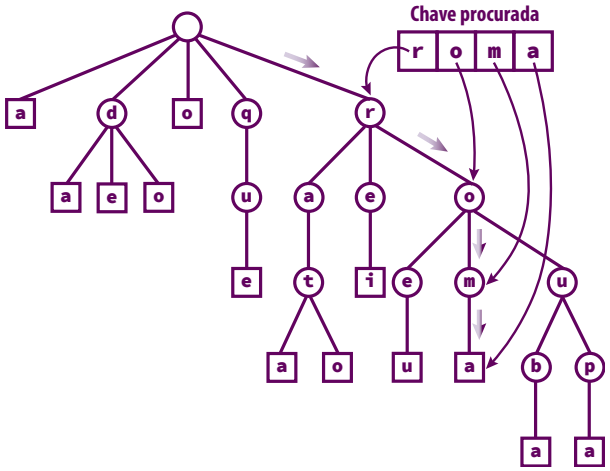


FIGURA 9–37: EXEMPLO DE BUSCA BEM-SUCEDIDA EM TRIE

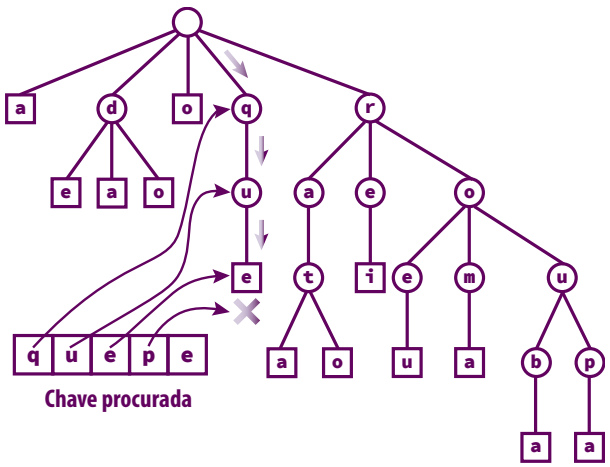


FIGURA 9–38: EXEMPLO DE BUSCA MALSUCEDIDA EM TRIE

A **Figura 9–39** apresenta o algoritmo de busca em trie.

ALGORITMO BUSCAEMTRIE

ENTRADA: Uma trie e uma chave de busca (string)

SAÍDA: *1*, se a chave for encontrada ou *0*, em caso contrário

1. Atribua a uma variável *c* o primeiro caractere da chave de busca
2. Faça um ponteiro *p* apontar para o filho da raiz da trie associado a *c*
3. Enquanto o ponteiro *p* não for nulo e *c* não for o último caractere da chave, faça:
   - 3.1 Atribua a *c* o próximo caractere da chave de busca
   - 3.2 Se o nó para o qual *p* aponta possuir um filho nulo associado a *c*, retorne *0*
   - 3.3 Caso contrário, faça *p* apontar para o filho associado a *c*
4. Se *p* for nulo ou estiver apontando para um nó que não é final, retorne *0*
5. Caso contrário, retorne *1*

FIGURA 9–39: ALGORITMO DE BUSCA EM TRIE

Inserção

Para inserir uma chave numa trie, segue-se o caminho especificado pelos caracteres da chave até que não se consiga mais prosseguir. Se durante esse percurso, for encontrada uma folha na trie, novos nós devem ser criados e acrescentados à trie de modo a completar o caminho correspondente à nova chave. Se a chave for nova ou for uma extensão de uma chave já presente na trie, é necessário acrescentar os nós da chave que ainda não existem e assinalar o nó que representa seu último caractere como um nó final. Se a chave ora acrescentada for prefixo de uma chave já presente na trie, deve-se simplesmente assinalar o nó que representa o último caractere da nova chave como um nó final. A **Figura 9–40** ilustra o processo de inserção do string `"romano"` na trie da **Figura 9–38**.

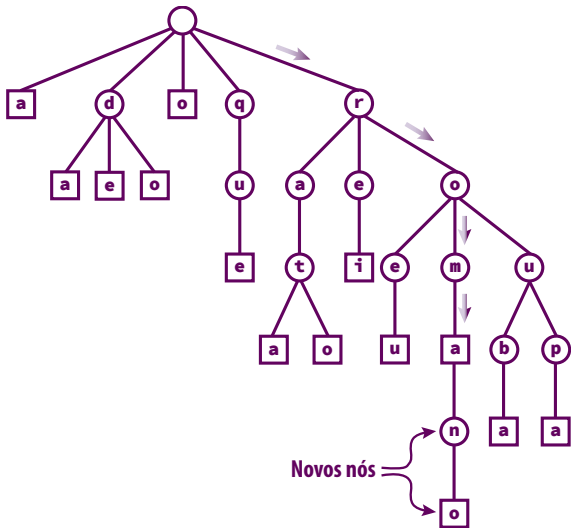


Figura 9–40: Exemplo de Inserção de Chave (String) em Trie

A **Figura 9–41** apresenta o algoritmo de inserção em trie.

Algoritmo InsereEmTrie

Entrada: Uma chave (string)
Entrada/Saída: Uma trie

1. Considere a raiz da trie o nó corrente
2. Atribua a uma variável *c* o primeiro caractere da chave a ser inserida
3. Repita o seguinte:
   - **3.1** Se o nó corrente possuir um filho nulo associado a *c*, crie um novo filho *f* para o nó corrente e associe-o a *c*
   - **3.2** Faça com que *f* seja o novo nó corrente
   - **3.3** Se *c* não for o último caractere da chave, atribua a *c* o próximo caractere da chave de busca
   - **3.4** Caso contrário:
     - **3.4.1** Faça com que o nó corrente seja um nó final
     - **3.4.2** Encerre

Figura 9–41: Algoritmo de Inserção em Trie

Remoção

A remoção de uma chave de uma trie é efetuada recursivamente de modo ascendente, como é usual em remoção em árvores. Mas não se aflija, pois remoção em tries é muito mais fácil do que remoção em árvores binárias

balanceadas (v. **Capítulo 4**) ou mesmo em árvores B (v. **Capítulo 6**). Para remover uma chave de uma trie, percorre-se o caminho determinado pelos caracteres da chave. Então supondo que essa chave tenha sido encontrada, se o nó que representa o último caractere da chave (i.e., o nó final da chave) não possuir nenhum filho, ele é removido da trie, como mostra a **Figura 9–42**. Caso esse nó possua pelo menos um filho, seu conteúdo é alterado, de modo que ele deixa de ser um nó final, como mostra a **Figura 9–43**.

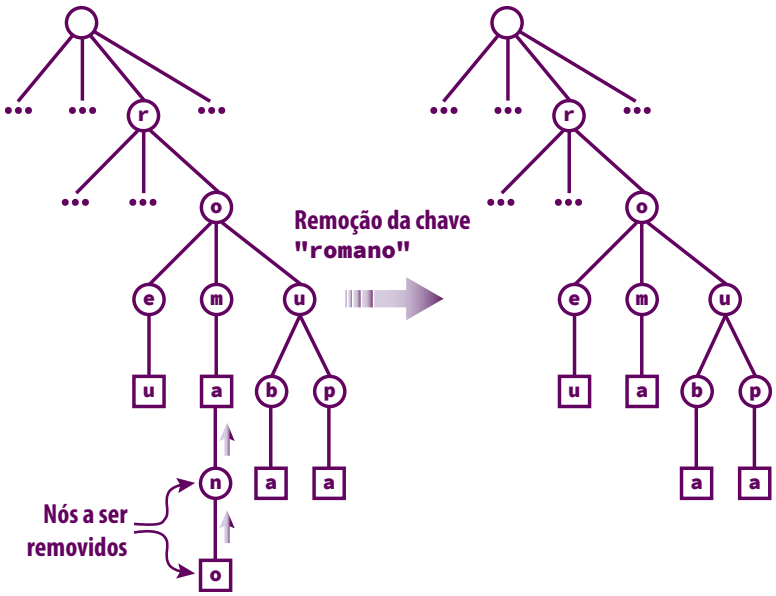


Figura 9–42: Exemplo de Remoção de Chave em Trie com Remoção de Nós

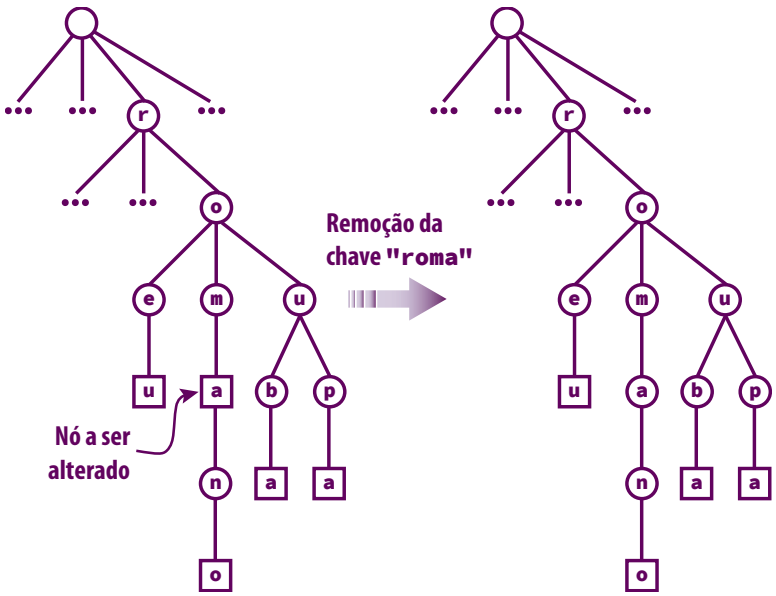


Figura 9–43: Exemplo de Remoção de Chave em Trie sem Remoção de Nó

Após a remoção do nó final de uma chave, visita-se o pai desse nó e verifica-se se ele (o pai) é um nó final (de outra chave, obviamente). Se esse for o caso, o processo de remoção está encerrado. Em caso contrário, verifica-se se esse nó tinha como filho apenas o nó que foi removido e, se for assim, ele também é removido; se não for assim, a remoção é encerrada. Esse processo prossegue enquanto estiver ocorrendo remoção de nós ou até que a raiz da trie seja alcançada. É importante lembrar que a raiz de uma trie nunca é removida, pois,

diferentemente do que ocorre com outros tipos de árvores, uma trie nunca é vazia (no sentido usual de árvore vazia), mesmo que ela não contenha nenhuma chave. Na trie ilustrada na **Figura 9–42**, apenas dois nós são removidos na remoção da chave `"romano"`. Por sua vez, na remoção da chave `"roma"` mostrada na **Figura 9–43**, não ocorre remoção de nenhum nó.

A **Figura 9–44** apresenta o algoritmo de remoção em trie.

ALGORITMO REMOVEEMTRIE

ENTRADA: A chave (*c*) a ser removida

ENTRADA/SAÍDA: Uma trie

RETORNO: Um valor informando o sucesso ou o fracasso da operação

1. Se *c* não estiver presente na trie, encerre informando o fracasso da operação
2. Se *c* não for prefixo de outra chave na trie nem houver nenhuma outra chave que seja prefixo de *c*, faça o seguinte:
   - **2.1** Remova todos os nós no caminho desde a raiz da trie até o nó final de *c*
   - **2.2** Retorne informando o sucesso da operação
3. Se *c* for prefixo de outra chave da trie:
   - **3.1** Faça com que o nó final que representa o último caractere de *c* deixe de ser nó final
   - **3.2** Retorne informando o sucesso da operação
4. Se houver na trie pelo menos uma chave que seja prefixo de *c*:
   - **4.1** Remova os nós no caminho do nó final de *c* até o nó final do maior prefixo de *c*, sem incluir esse nó final do referido prefixo
   - **4.2** Retorne informando o sucesso da operação

FIGURA 9–44: ALGORITMO DE REMOÇÃO EM TRIE

O **Passo 3.1** do algoritmo **REMOVEEMTRIE** é ilustrado na **Figura 9–43**, enquanto o **Passo 3.2** é demonstrado na **Figura 9–42**.

9.8.2 Implementação

Tries podem ser implementadas de diversas maneiras e a implementação que será adotada aqui tem motivação didática. Quer dizer, essa implementação não é eficiente em termos de uso de espaço, mas é bastante eficiente em termos de custo temporal.

A **Figura 9–45** mostra as representações expandida (i.e., fiel à implementação) e simplificada (i.e., conceitual) do string `"que"`. A representação simplificada [v. **Figura 9–45(b)**] é aquela que tem sido utilizada neste livro para ilustrar graficamente o conceito de trie, enquanto a representação expandida [v. **Figura 9–45(a)**] reflete o modo como os nós de uma trie são implementados.

Como ilustra a **Figura 9–45(a)**, na implementação de trie desenvolvida aqui, um nó é representado por uma estrutura com dois campos:

[1] O primeiro campo de um nó é um array de ponteiros de tamanho igual a $|\sum|$, que apontam para os filhos do nó. Desse modo, a associação de cada caractere ao respectivo nó é implícita, pois quando um ponteiro da trie aponta para um filho, esse ponteiro está associado a um caractere e esse caractere é exatamente aquele associado ao filho, como mostra a **Figura 9–45 (a)**. Nessa figura, o ponteiro que emana do elemento do array rotulado com *q* indica que o filho para o qual esse ponteiro aponta está associado a esse caractere. É importante notar que os caracteres que constituem os strings armazenados na trie não são realmente armazenados na trie, como parece indicar a **Figura 9–45 (a)**. Quer dizer, as

letras que aparecem no array de ponteiros de cada nó devem ser interpretadas como os índices desse array correspondentes a essas letras. Ou seja, o que existe de fato é um mapeamento entre os caracteres do alfabeto em consideração e os índices desse array. Nessa implementação específica na qual o alfabeto é constituído por todas as letras minúsculas do alfabeto latino (sem inclusão de diacríticos), esse mapeamento é obtido por meio da definição de macro:

```
#define INDICE_CARACTERE(c) ((c) - 'a')
```

[2] O segundo campo de cada nó da trie é um inteiro, denominado `ehFinal`, que indica se esse nó é um nó final (nesse caso, o valor desse campo é `1`) ou não (nesse caso, o valor desse campo é `0`). Na implementação de um dicionário[8], no sentido mais amplo desse termo, esse campo pode ser substituído por um ponteiro para uma lista encadeada que armazena os valores associados a cada chave (v. exemplo na **Seção 9.10.2**).

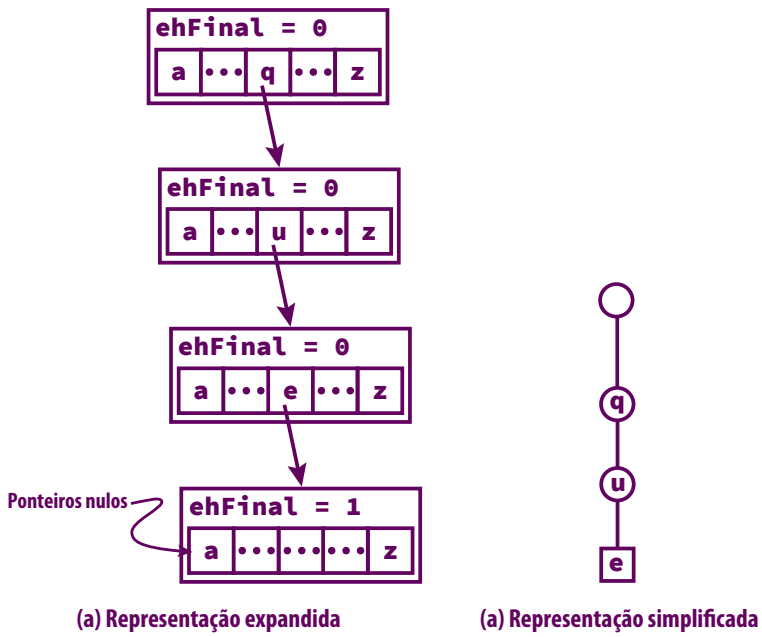


FIGURA 9–45: REPRESENTAÇÕES EXPANDIDA E SIMPLIFICADA DE TRIE

As seguintes definições de tipo são usadas para definir nós e ponteiros para nós da trie:

```
typedef struct rotNoTrie {
          struct rotNoTrie *filhos[TAM_ALFABETO];
          int              ehFinal;
       } tNoTrie, *tTrie;
```

Por simplicidade, serão considerados apenas caracteres que sejam letras minúsculas não acentuadas do alfabeto latino, de modo que a constante `TAM_ALFABETO`, que especifica o tamanho do alfabeto pode ser especificado como[9]:

```
#define TAM_ALFABETO 'z' - 'a' + 1
```

A função `IniciaTrie()` inicia uma trie contendo apenas uma raiz vazia e retorna o endereço da raiz da trie.

[8] No presente contexto, um dicionário é simplesmente uma tabela de busca que permite chaves secundárias.

[9] Essa definição é portável, pois não se conhece qualquer código de caracteres no qual as letras, maiúsculas ou minúsculas, não estejam ordenadas. Entretanto pode haver um desperdício adicional de memória se houver espaço entre letras que não sejam caracteres. Por exemplo, no código EBCDIC, o resultado dessa operação é *41*, o que significa que cinco caracteres que não são letras e que nunca serão usados estão incluídas no alfabeto.

```
tTrie IniciaTrie(void)
{
   tNoTrie *pNo = NovoNoDeTrie();

   ASSEGURA(pNo, "Impossivel alocar no' de trie");

   return pNo;
}
```

A função `NovoNoDeTrie()`, aloca dinamicamente um nó de uma trie e inicia seus campos. Ela retorna o endereço do nó criado, se a alocação foi bem-sucedida, ou **NULL**, em caso contrário.

```
tNoTrie *NovoNoDeTrie(void)
{
   int      i;
   tNoTrie *pNo = malloc(sizeof(*pNo));

   if (pNo) {
      pNo->ehFinal = 0; /* Por enquanto, o nó não é final */

         /* Inicia cada filho do nó com NULL */
      for (i = 0; i < TAM_ALFABETO; i++)
         pNo->filhos[i] = NULL;
   }

   return pNo;
}
```

A função `InsereEmTrie()` insere uma nova chave (string) numa trie. Essa função tem como parâmetros:

- `raiz` (entrada/saída) — endereço da raiz da trie
- `chave` (entrada) — chave que será inserida

É importante notar que, se a chave já existir, ela não será inserida. Além disso, o último nó visitado indica o final de um string, de modo que se a chave era apenas um prefixo de alguma outra chave na trie, ele passará a ser uma chave.

```
void InsereEmTrie(tNoTrie *raiz, const char *chave)
{
   int      indice; /* Índice de um caractere no array de filhos de um nó */
   tNoTrie *p; /* Ponteiro usado para descer na trie */

   for (p = raiz; *chave; ++chave) {
         /* Obtém o índice do caractere corrente da chave */
      indice = INDICE_CARACTERE(*chave);

         /* Certifica-se que o índice obtido é válido */
     ASSEGURA(indice >= 0 && indice < TAM_ALFABETO, "Indice invalido em InsereEmTrie()");

         /* Verifica se o caractere corrente */
         /* existe no presente nó            */
      if (!p->filhos[indice])
            /* O referido caractere não existe e é preciso */
            /* criar mais um filho para o corrente nó      */
         p->filhos[indice] = NovoNoDeTrie();

      p = p->filhos[indice]; /* Passa para o próximo nível da trie */
   }

      /* O último nó visitado deve indicar o final de um string */
   p->ehFinal = 1;
}
```

A função `InsereEmTrie()` utiliza a macro `INDICE_CARACTERE` apresentada acima para obter o índice de um ponteiro que representa um caractere no array de ponteiros de um nó:

A função `BuscaEmTrie()` efetua uma busca numa trie e seus parâmetros são:

- `raiz` (entrada) — endereço da raiz da trie
- `chave` (entrada) — a chave de busca

A função `BuscaEmTrie()` retorna `1`, se a chave for encontrada, ou `0`, em caso contrário.

```
int BuscaEmTrie(tNoTrie *raiz, const char *chave)
{
   int      indice; /* Índice de um caractere no array de filhos de um nó  */
   tNoTrie *p; /* Ponteiro usado para descer na trie */

   for (p = raiz; *chave && p; ++chave) {
         /* Obtém o índice do caractere corrente da chave */
      indice = INDICE_CARACTERE(*chave);

         /* Certifica-se que o índice obtido é válido */
      ASSEGURA( indice >= 0 && indice < TAM_ALFABETO,
                "Indice invalido em BuscaEmTrie()" );

         /* Se o caractere corrente da chave não existe   */
         /* no presente nó, a chave não faz parte da trie */
      if (!p->filhos[indice])
         return 0;

      p = p->filhos[indice]; /* Passa para o próximo nível da trie */
   }

      /* Se o laço encerrou porque p assumiu NULL, a chave não se encontra na trie */
   if (!p)
      return 0;

      /* A chave está presente na trie se o último nó visitado for um nó final */
   return p->ehFinal;
}
```

A função `EhNoFinalDeTrie()` verifica se um nó de uma trie indica o final de um string e seu único parâmetro é o endereço do referido nó. Ela retorna `1`, se o nó indicar o final de um string, ou `0`, em caso contrário.

```
int EhNoFinalDeTrie(const tNoTrie *pNo)
{
   return pNo->ehFinal;
}
```

A função `EhNoVazioDeTrie()` verifica se um nó de uma trie está vazio. O único parâmetro dessa função é o endereço do referido nó e ela retorna `1`, se esse nó estiver vazio, ou `0`, em caso contrário.

```
int EhNoVazioDeTrie(const tNoTrie *pNo)
{
   int i;

      /* Verifica se o nó possui algum filho */
   for(i = 0; i < TAM_ALFABETO; i++)
         /* Se o nó tiver pelo menos um filho, ele não está vazio */
      if(pNo->filhos[i])
         return 0; /* Nó não está vazio */

   return 1; /* Nó não possui nenhum filho */
}
```

A função `RemoveEmTrie()` é chamada para remover uma chave de uma trie e seus parâmetros são:

- `pTrie` (entrada) — raiz da trie
- `chave` (entrada) — chave a ser removida

Na realidade, essa função é apenas uma interface para a função `EhNoVazioDeTrie()` que realmente realiza a tarefa de remoção e que será apresentada mais adiante.

```
void RemoveEmTrie(tTrie pTrie, const char *chave)
{
   int tam = strlen(chave);

      /* Se a chave não for um string vazio, chama */
      /* a função EhNoVazioDeTrie() para removê-la */
   if(tam > 0) {
         /* O nível inicial de remoção é 0, que corresponde à raiz da trie */
      EhNoVazioDeTrie(pTrie, chave, 0, tam);
   }
}
```

A função `RemoveEmTrieAux()` é aquela que de fato efetua remoção e seus parâmetros são:

- `pNo` (entrada/saída) — endereço do nó que poderá ser removido
- `chave` (entrada) — chave a ser removida
- `nivel` (entrada) — nível de remoção (v. adiante)
- `tam` (entrada) — tamanho da chave a ser removida

Note que o parâmetro `nivel` varia de `0`, que corresponde à raiz da trie até `tam - 1`, que corresponde ao último caractere da chave. A função `RemoveEmTrieAux()` retorna `1`, se o nó estiver vazio; ou `0`, em caso contrário.

```
int RemoveEmTrieAux(tNoTrie *pNo, const char *chave, int nivel, int tam)
{
   int indice; /* Índice de um caractere no array de filhos de um nó  */

   if (!pNo)
      return 0; /* pNo não aponta para nenhum nó */

   /* A recursão encerra quando o nível for maior do que o */
   /* nível do último caractere da chave, dado por tam - 1 */

      /* Verifica se o nível corrente é igual ao tamanho da chave  */
   if (nivel == tam) { /* Base da recursão */
         /* Se o nó indicar o final da chave, ele deixará de ser um nó final */
      if (pNo->ehFinal) {
         pNo->ehFinal = 0; /* Nó deixa de ser final de string */

            /* Se o nó estiver vazio, ele precisa ser removido */
         if(EhNoVazioDeTrie(pNo))
            return 1;

         return 0; /* Nó não deve ser removido */
      }
   } else { /* Caso recursivo */
         /* Obtém o índice do caractere corrente da chave */
      indice = INDICE_CARACTERE(chave[nivel]);

         /* Certifica-se que o índice obtido é válido */
      ASSEGURA( indice >= 0 && indice < TAM_ALFABETO,
                "Indice invalido em RemoveEmTrieAux()" );

         /* Efetua a remoção no nó do próximo nível e */
         /* verifica se esse nó precisa ser removido  */
```

```
      if(RemoveEmTrieAux(pNo->filhos[indice], chave, nivel + 1, tam)) {
         FREE(pNo->filhos[indice]); /* Nó precisa ser removido */

            /* Sobe a árvore recursivamente removendo */
            /* os nós que devem ser removidos         */
         return !EhNoFinalDeTrie(pNo) && EhNoVazioDeTrie(pNo);
      }
   }
}
```

A função `RemoveEmTrieAux()` faz uso da seguinte macro, que libera um nó e torna nulo o ponteiro usado para liberá-lo:

```
#define FREE(p) do {free(p); p = NULL;} while(0)
```

9.8.3 Análise

Teorema 9.8: No pior caso, o custo temporal de uma operação de busca numa trie por um string de comprimento m é $\theta(m)$.

Prova: Numa operação de busca, no máximo $m + 1$ nós da trie são visitados e o custo de visita a cada nó é $\theta(1)$, visto que o tempo gasto para verificar se um caractere encontra-se ou não num nó é constante (i.e., basta obter o índice do ponteiro associado ao caractere no array de ponteiros e verificar se esse ponteiro é **NULL** ou não). Portanto, no pior caso, o custo temporal de uma operação de busca numa trie por um string de comprimento m é $\theta(m)$. ■

Teorema 9.9: O custo temporal de remoção de um string de comprimento m de uma trie é $\theta(m)$.

Prova: O raciocínio é o mesmo empregado na prova do **Teorema 9.8**. ■

Teorema 9.10: No pior caso, os custos temporal e espacial de inserção de um string de comprimento m numa trie são $\theta(m \cdot |\sum|)$.

Prova: No pior caso, a inserção de uma chave de comprimento m requer a criação de m nós e a criação de um nó tem custos temporal e espacial $\theta(|\sum|)$. Portanto, no pior caso, o custo temporal e o custo espacial de uma operação de inserção são $\theta(m \cdot |\sum|)$. ■

Do ponto de vista teórico, os resultados exibidos nesses teoremas, representam o melhor que se pode obter como custo temporal para essas operações, pois é impossível obter um custo temporal para uma operação de busca melhor do que o tamanho da chave de busca. Quer dizer, qualquer que seja a estrutura de dados ou algoritmo usado para implementar uma tabela de busca, não é possível saber que uma chave de busca se encontra na tabela se todos os caracteres que a constituem não forem comparados. Como as operações de inserção e remoção também envolvem busca, esse mesmo raciocínio é aplicável. Outra implicação importante dos resultados do último parágrafo é que eles indicam que os custos das operações de busca, inserção e remoção são independentes do número de chaves armazenadas numa trie.

Agora suponha que se esteja procurando por uma chave numa trie e se verifica que o primeiro caractere dessa chave não se encontra nessa trie. Então obviamente, pode-se concluir que a chave não faz parte da trie examinando-se apenas um nó. De fato, esse exemplo não representa um caso isolado, pois, tipicamente, verificar que uma chave não se encontra numa trie requer que apenas alguns nós sejam examinados. Mais precisamente, é possível demonstrar que, quando uma chave não se encontra numa trie construída a partir de n chaves formadas aleatoriamente com caracteres de um alfabeto $\sum$, o número médio de nós examinados numa operação de busca por uma chave é dado aproximadamente por $log_{|\sum|}\ n$. Isso significa que, considerando-se essas suposições, a busca por uma chave que não se encontra numa trie não depende do comprimento da chave. Por exemplo, de acordo com essa proposição, a busca por uma chave que representa uma palavra da língua portuguesa

numa trie contendo um milhão de chaves requer que se examinem apenas cerca de quatro nós (pois, $log_{26}\ 1.000.000 \simeq 4{,}24$), independentemente do tamanho dessa chave[10].

Ocorre, porém, que não é o caso que palavras da língua portuguesa (ou qualquer outra língua natural) sejam obtidas por meio de escolhas ao acaso das letras que constituem o alfabeto utilizado. Contudo, mesmo que a precisão do raciocínio apresentado no último parágrafo seja meramente teórica, essa alegação é capaz de prever com razoável exatidão aquilo que ocorre na prática. Ou seja, com efeito, a busca por chaves inexistentes numa trie requer que apenas alguns poucos nós sejam examinados.

O principal problema apresentado por tries é a quantidade de espaço que elas necessitam e, pior, uma grande quantidade desse espaço é desperdiçado. Ou seja, muitos nós podem ter apenas algumas poucas ramificações que não são nulas, mas, mesmo assim, eles usam espaço proporcional a $|\sum|$. Quanto maior for o tamanho $|\sum|$ do alfabeto e quanto maiores forem os tamanhos das chaves, maior será o gasto de memória.

O custo espacial de uma trie é $\theta(|\sum| \cdot M \cdot n)$, sendo n o número de chaves e M é o tamanho médio das chaves armazenadas na trie. Existem representações eficientes de tries (p.ex., trie PATRICIA — v. **Seção 9.8.4**) que minimizam o uso de espaço de memória, mas essas representações não são tão fáceis de implementar quanto o tipo básico de trie apresentado aqui.

A **Tabela 9–5** apresenta uma comparação entre tries e outros tipos de árvores de busca, notadamente árvores binárias.

Vantagens de Tries	Desvantagens de Tries
Numa trie, busca, inserção e remoção usando uma chave de comprimento m tem custo temporal $\theta(m)$ no pior caso. Numa árvore binária balanceada, essas operações têm custo temporal $\theta(\log n)$, em que n é o número de chaves na árvore. Por uma questão de justiça, nessa comparação, o custo temporal para tries pode ser considerado $\theta(1)$.	**Tries são capazes de apresentar as chaves que elas armazenam de modo ordenado, mas essa ordenação corresponde à ordenação lexicográfica.**
Uma trie pode requerer menos espaço do que uma árvore binária quando ela contém um grande número de chaves curtas, porque elas não são armazenadas explicitamente e existe um compartilhamento de nós entre as chaves.	**Tries não são adequadas para implementação de tabelas de busca residentes em memória secundária.**
A maioria dos tipos de chaves pode ser interpretada como string. Por exemplo, um número inteiro pode ser visto como um string de dígitos da base decimal ou binária.	**Nem sempre é fácil representar chaves de determinado tipo (p.ex., números reais) como strings.**
Tries não requerem balanceamento, como ocorre com árvores binárias.	—
Tries permitem encontrar prefixos de chaves facilmente. Elas também permitem associar um valor a um conjunto de chaves que compartilham um prefixo comum.	—

Tabela 9–5: Tries versus Outros Tipos de Árvores de Busca

Tries são especialmente adequadas quando as chaves apresentam tamanhos variados e quando se espera que um número razoável de buscas não seja bem-sucedido (porque a chave não faz parte da trie). Por outro lado, se as chaves têm tamanho fixo ou se existe a expectativa que a maioria das buscas seja bem-sucedida, então tries compactadas podem ser uma melhor opção (v. **Seção 9.8.4**).

[10] Aqui, consideram-se palavras que usam apenas letras maiúsculas sem acentuação que ainda persistem em muitos sistemas antigos.

A altura de uma trie é determinada pela maior chave armazenada nela, de modo que, se as chaves forem palavras da língua portuguesa, a profundidade de uma trie não deve ser muito grande. De acordo com artigo publicado por Pedro Quaresma e Augusto Pinho em 2007 (v. **Bibliografia**), a média de comprimento das palavras da língua portuguesa é de *4,64*. Portanto esse último valor corresponde ao número médio de comparações de caracteres numa trie contendo um número razoável de palavras dessa língua, independentemente de esse número ser *10.000*, *100.000* ou *1.000.000*. A título de comparação, uma árvore binária de busca balanceada (v. **Seção 4.3**) contendo *100.000* chaves tem uma altura dada por $\lceil log_2\ 100.000 \rceil = 19$. Concluindo, em situações nas quais rapidez de processamento é essencial e uso excessivo de espaço não constitui um problema, tries podem ser a melhor escolha.

9.8.4 Implementações Alternativas de Tries

Conforme já foi afirmado, existem diversas maneiras de implementar o conceito de trie. Como é usual, cada uma dessas alternativas possui vantagens e desvantagens. Algumas dessas implementações alternativas de tries serão brevemente discutidas nesta seção.

A principal razão pela qual o espaço ocupado por uma trie é demasiadamente grande é o fato de chaves longas tenderem a apresentar sufixos longos na trie, com cada nó tendo um único ponteiro para o próximo nó (e, portanto, $|\sum| - 1$ ponteiros nulos). Essa situação não é difícil de corrigir.

A **Figura 9–46** mostra uma implementação alternativa de tries que resolve o problema indicado no último parágrafo. Nessa implementação, em vez de usar um array de ponteiros como na implementação discutida na **Seção 9.8.2**, cada nó, com exceção da raiz da trie, contém apenas dois ponteiros: o primeiro deles aponta para o primeiro filho do nó e o segundo ponteiro aponta para uma lista encadeada que armazena os irmãos desse nó. Esse tipo de representação de trie não é difícil de implementar, mas é um pouco complicado visualizar os strings armazenados usando essa representação. Por exemplo, tente descobrir quais são os strings armazenados na trie da **Figura 9–46** e confira sua resposta na nota de rodapé[11].

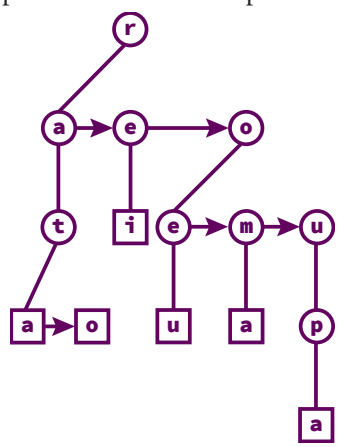


FIGURA 9–46: IMPLEMENTAÇÃO DE TRIE USANDO LISTA ENCADEADA

Outra alternativa de implementação de trie, que é baseada na ideia anterior, consiste em substituir a lista encadeada que une os filhos de um nó por uma árvore binária de busca. Essa nova ideia dá origem a uma estrutura de dados denomina **árvore ternária de busca**. Essa estrutura é explorada em profundidade no livro de seu criador — Robert Sedgewick (v. **Bibliografia**).

Outra técnica de implementação de tries é denominada **redução de alfabeto** e consiste em interpretar os strings originais como strings mais longos que usam um alfabeto de menor tamanho. Por exemplo, um string que usa

[11] Os strings são: `"rata"`, `"rato"`, `"rei"`, `"roeu"`, `"roma"` e `"roupa"`.

originalmente *n* bytes passa a ser considerado um string que usa *2n* metades de um byte. Então uma trie básica que usa um array de *256* (i.e., 2^8) ponteiros em cada nó passa a usar um array com apenas 16 (i.e, 2^4) ponteiros em cada nó. Embora a economia de espaço seja substancial, cada operação básica sobre uma trie implementada de acordo com esse esquema requer o dobro de visitas a nós. Além disso, a implementação dessa ideia requer programação de baixo nível e não é tão trivial quanto a implementação apresentada aqui.

Uma trie pode apresentar longas cadeias de nós, cada um dos quais contendo apenas uma ramificação. Essa situação pode causar enorme desperdício de memória e uma maneira de resolver esse problema é usando uma **trie compactada** (ou **trie PATRICIA**[12]).

Uma trie compactada é uma variante de trie na qual cadeias de nós que possuem apenas um filho são compactados num único nó. Quer dizer, numa trie convencional, cada nó representa um único caractere, enquanto numa trie compactada um nó pode representar um string inteiro. Portanto numa trie compactada, o grau de cada nó é pelo menos igual a dois.

Uma trie compactada pode ser obtida a partir de uma trie padrão por meio da compressão de cadeias de nós redundantes. Um nó é considerado **redundante** se ele possui apenas um filho e não é a raiz da trie. A **Figura 9–47** mostra uma trie compactada obtida a partir da eliminação dos nós redundantes da trie ilustrada na **Figura 9–35**.

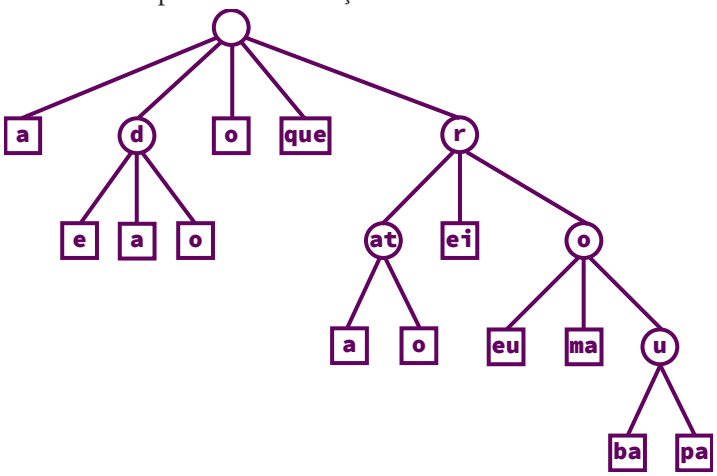


Figura 9–47: Trie Compactada (PATRICIA)

De fato, tries compactadas ocupam bem menos espaço do que tries tradicionais, mas, em compensação, são muito mais difíceis de implementar e são muito mal documentadas na literatura de programação.

9.9 Casamento de Strings vs Casamento de Palavras

Tries facilitam um tipo especial de casamento de strings denominado **casamento de palavras**. Numa operação de casamento de palavras, tenta-se determinar se uma determinada palavra casa exatamente com uma palavra do texto. Assim casamento de palavras difere de casamento ordinário de strings porque agora não se permite a um padrão (no sentido descrito na **Seção 9.1**) casar com um substring qualquer do texto.

Informalmente, casamento ordinário de strings e casamento de palavras são operações que o leitor provavelmente utiliza em seu cotidiano. Quer dizer, a maioria dos editores e processadores de texto oferecem a opção de busca por palavra inteira e, quando essa opção é escolhida, ocorre casamento de palavras; caso contrário, ocorre casamento de strings com texto, que foi discutido na **Seção 9.1**. Por exemplo, a **Figura 9–48** mostra a janela de diálogo apresentada pelo aplicativo Microsoft Word 2010 quando um usuário deseja localizar uma palavra

[12] A denominação *PATRICIA* insinua um nome feminino. No entanto, ela foi cunhada por Donald Morrison em 1969 (v. **Bibliografia**) como acrônimo de *Practical Algorithm To Retrieve Information Coded in Alphanumeric*.

inteira ou parte de uma palavra. No primeiro caso, o usuário está usando casamento de palavras, enquanto no segundo caso, ele usa casamento de strings.

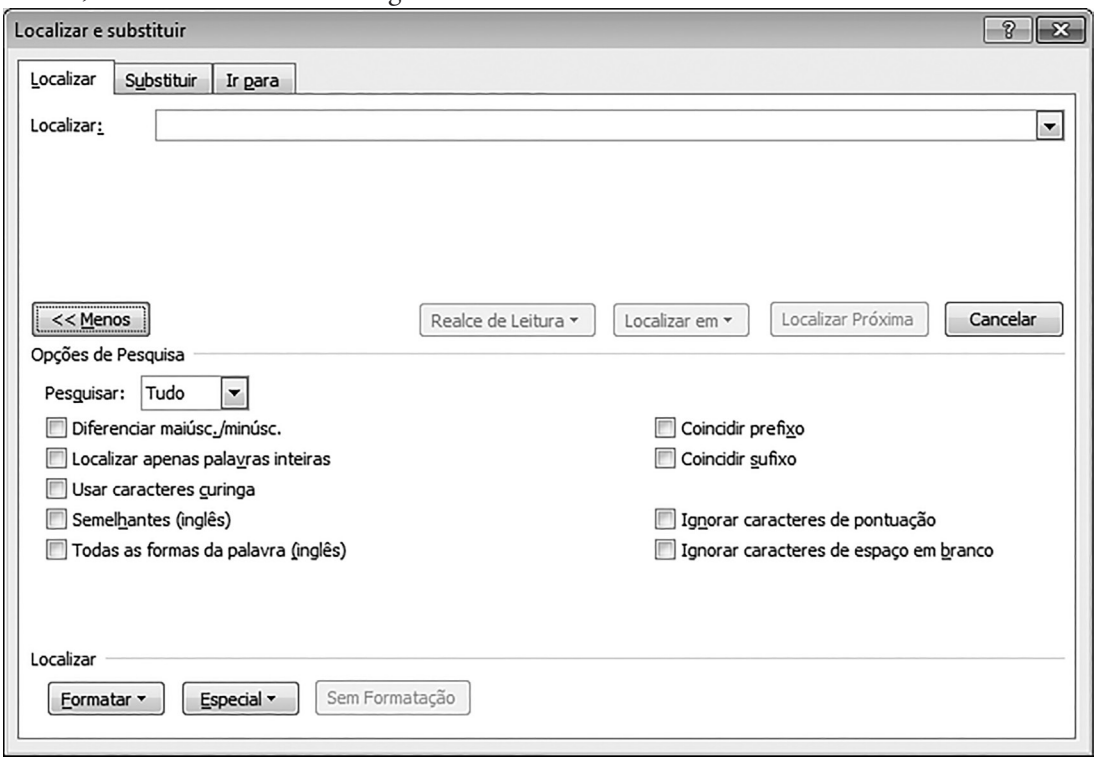


FIGURA 9–48: CASAMENTO DE PALAVRAS E DE STRINGS EM MICROSOFT WORD 2010

Em casamento de palavras, um padrão pode casar apenas com as palavras de um texto. Agora, aquilo que é considerado palavra depende da aplicação e deve ser decidido antes da construção da trie que representa as palavras do texto. Tipicamente, num texto em linguagem natural, palavras são separadas por espaços em branco e símbolos de pontuação (v. exemplo na **Seção 9.10.2**).

A **Figura 9–49** mostra o texto *o rato que roeu a roupa do rei de roma que rouba a roupa do rato roeu a roupa da rata* codificado numa trie. Note, nessa figura, que existem três casamentos do string `"ra"` com esse texto, mas não há nenhum casamento da palavra `"ra"` com o texto em questão.

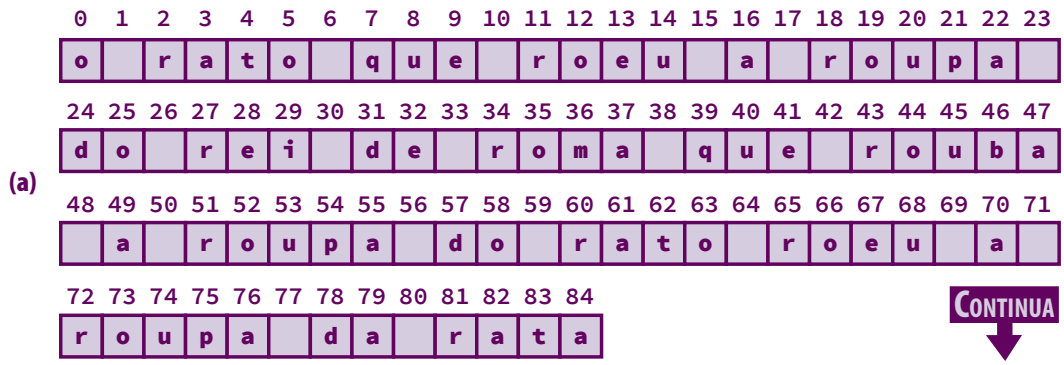


FIGURA 9–49: EXEMPLO DE TRIE REPRESENTANDO PALAVRAS DE UM TEXTO

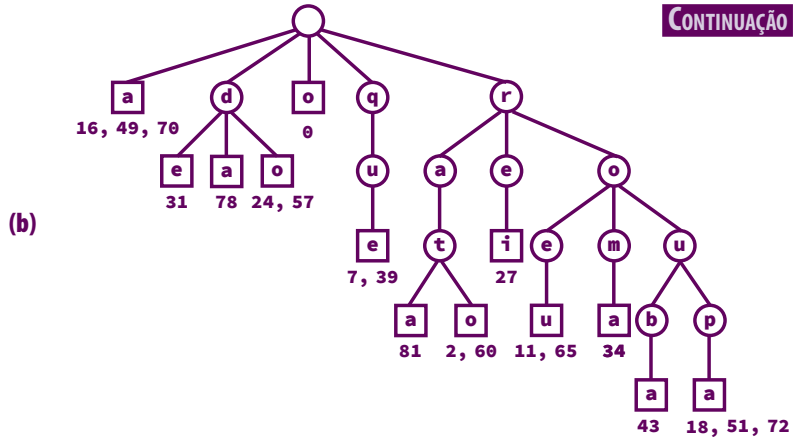


FIGURA 9–49 (CONT.): EXEMPLO DE TRIE REPRESENTANDO PALAVRAS DE UM TEXTO

Na **Seção 9.10.2** você aprenderá como indexar texto de modo que ele permita casamento de palavras.

9.10 Exemplos de Programação

9.10.1 Separando um String em Partes (Tokens)

Preâmbulo: Um **token** é uma sequência de caracteres considerada como uma unidade que possui significado próprio num determinado contexto. Tokens num string são identificados por caracteres separadores que os delimitam. Por exemplo, um comando de um sistema operacional pode ser considerado como um string composto de tokens, como o comando `ls` de sistemas da família Unix a seguir:

```
ls -alt
```

Nesse exemplo, o comando é composto por dois tokens: (1) `ls`, que é o nome do comando, e (2) `-alt`, que são as opções do comando. Nesse caso, o separador de tokens é espaço em branco. O comando completo é considerado um string, mas cada token que o compõe também é considerado um string (ou, melhor, um substring).

A função **strtok()** divide um string em tokens e seu protótipo é:

```
char *strtok(char *str, const char *separadores)
```

Nesse protótipo, os parâmetros têm os seguintes significados:

- `str` é o string a ser dividido em tokens
- `separadores` é um string contendo os caracteres que separam as partes

A primeira chamada de **strtok()** retorna o endereço do primeiro token encontrado no string `str` e um caractere terminal de string é colocado nesse parâmetro ao final do referido token. Chamadas subsequentes dessa função usando **NULL** como primeiro parâmetro retornarão os tokens seguintes até que nenhum deles seja remanescente no string original. Quando nenhum token é encontrado, a função **strtok()** retorna **NULL**. Deve-se chamar atenção para o fato de a função **strtok()** modificar o string passado como primeiro parâmetro. Portanto se for necessário preservar o string original, faça uma cópia dele antes de chamar essa função.

A função **strtok()** apresenta um sério problema quando o string que está sendo separado em partes contém dois ou mais separadores seguidos. Nesse caso, essa função não funciona como se

poderia esperar, pois ela considera esses separadores como se fossem um único separador. Como ilustração desse problema, considere o seguinte programa:

```
#include <stdio.h>
#include <string.h>

int main(void)
{
   char string[] = "um,dois,,quatro,cinco";
   char separadores[] = " ,"; /* Note o espaço em branco */
   char *token;
   int  i = 0;

   printf("\nString a ser separado em tokens: \"%s\"\n", string);
   printf("\nOs tokens sao:\n");

   token = strtok(string, separadores);

   while (token) {
      printf("\tToken %d: %s\n", ++i, *token ? token : "<vazio>");
      token = strtok(NULL, separadores);
   }
   return 0;
}
```

Quando executado, o programa acima escreve o seguinte na tela:

```
String a ser separado em tokens: "um,dois,,quatro,cinco"

Os tokens sao:
        Token 1: um
        Token 2: dois
        Token 3: quatro
        Token 4: cinco
```

O problema ilustrado acima ocorre em muitas situações de natureza prática, como por exemplo, quando se processam arquivos de dados em formato CSV. Um **arquivo CSV** é um arquivo de texto contendo um registro em cada linha, sendo que os campos de cada registro são separados por vírgulas. Num arquivo CSV, quando um campo não possui valor, o espaço que esse valor deveria ocupar é vazio, de modo que aparecem duas vírgulas em sequência. Um programa que processa um arquivo CSV precisa ler cada linha do arquivo e separar seus campos. Assim a maneira mais fácil de fazer isso é armazenar cada linha como string e separar esse string em tokens. É aí que a função **strtok()** se mostra inconveniente.

Problema: Escreva uma função que corrige essa deficiência da função **strtok()**.

Solução: A função `ObtemTokens()` apresentada a seguir corrige a deficiência da função **strtok()**apontada acima. Os parâmetros e os valores retornados por essas duas funções têm as mesmas interpretações.

```
char *ObtemTokens(char *str, char const *sep)
{
   static char *proximoToken; /* Aponta para o próximo token, se ele existir */
   char        *s, /* Apontará para o string no qual a */
                   /* busca pelo token será efetuada   */
               *inicio = NULL; /* Guardará o início do token corrente */

      /* Se 'str' não for um ponteiro nulo, o próximo  */
      /* token será obtido a partir do início de 'str' */
   if (str)
      proximoToken = str;
```

```
      /* Se 'proximoToken' for um ponteiro nulo, o */
      /* string ora explorado não tem mais tokens  */
   if (!proximoToken)
      return NULL; /* Não há mais token nesse string */

      /* Obtém o endereço do primeiro separador encontrado em 'proximoToken' */
   s = strpbrk(proximoToken, sep);
      /* Verifica se foi encontrado algum separador no string ora sendo explorado */
   if (s) {
         /* Termina o token corrente na posição em */
         /* que se encontra o separador encontrado */
      *s = '\0';
      inicio = proximoToken; /* Guarda o início do token corrente */

         /* O próximo token começará no primeiro caractere após o separador */
      proximoToken = ++s;
   } else
      if (*proximoToken) {
            /* Não foi encontrado nenhum separador, mas o string  */
            /* corrente não é vazio e seu endereço será retornado */
         inicio = proximoToken;

         /* Não foi encontrado nenhum separador, de modo que */
         /* não haverá mais nenhum token na próxima chamada  */
      proximoToken = NULL;
   }

   return inicio;
}
```

A função **strpbrk()**, que é chamada por `ObtemTokens()`, procura num string (primeiro parâmetro) a primeira ocorrência de qualquer caractere presente noutro string (segundo parâmetro). A função **strpbrk()** retorna o endereço do primeiro caractere do segundo parâmetro encontrado no primeiro parâmetro ou **NULL**, se tal caractere não for encontrado.

9.10.2 Quantas Vezes Machado Fala em Amor?

Preâmbulo: O escritor brasileiro **Machado de Assis** é considerado pela maioria dos críticos o maior nome da literatura nacional. Sua obra inclui romances, poemas, crônicas, peças de teatro, contos, críticas literárias e muito mais. Machado foi fundador e primeiro presidente da Academia Brasileira de Letras. Nosso orgulho por ele nivela-se ao que o povo inglês tem por Shakespeare e o russo tem por Dostoiévski.

Problema: (a) Escreva um programa que utilize uma trie básica, como aquela apresentada na **Seção 9.8**, para indexar as palavras da obra de Machado de Assis, de tal modo que o programa permita consultas como:

```
>>> Digite a palavra a ser procurada: amor

>>> "amor" foi encontrada 84 vezes.
>>> Deseja exibir as ocorrencias (s/n)? s

Linha: 2627     Posicao: 51
... [Trecho removido]
Linha: 224      Posicao: 37
```

(b) Apresente uma análise do custo temporal da solução em termos de notação ó.

Observação: Obviamente, Machado seria considerado um insensível se falasse em *amor* apenas 84 vezes em sua obra. Ocorre, porém, que, apesar de a obra de Machado ser de domínio público há longo

tempo, o autor não conseguiu encontrar um único arquivo de texto puro contendo toda a obra desse gigante da literatura nacional. Portanto o exemplo apresentado acima refere-se apenas ao romance *Ressurreição*, publicado em 1872 (v. **Bibliografia**), que o autor da presente obra dedicou-se a converter em texto puro (v. **Apêndice A**).

Solução de (a): O conceito básico de trie discutido na **Seção 9.8** continuará sendo válido aqui, de modo que apenas as alterações na implementação vista naquela seção serão apresentadas a seguir.

Em primeiro lugar, o tipo de nó apresentado na **Seção 9.8.2** precisa ser alterado para permitir que se armazenem as posições na quais os strings se encontram no texto. O novo tipo de nó é apresentado a seguir:

```
typedef struct rotNoTrieMachado {
           struct rotNoTrieMachado *filhos[TAM_ALFABETO];
           tListaSE                 valores;
        } tNoTrieMachado, *tTrieMachado;
```

Note que a única alteração com relação ao tipo apresentado na **Seção 9.8.2** é que o segundo campo de cada nó (i.e., `valores`) representa um ponteiro para uma lista simplesmente encadeada do tipo `tListaSE`, que foi discutida no **Capítulo 10** do **Volume 1**. O conteúdo efetivo de cada nó dessa lista armazena o índice da linha do arquivo e a posição nessa linha na qual a palavra se encontra. Assim o tipo desse conteúdo é definido como:

```
typedef struct {
           int linha;
           int pos;
        } tConteudoMachado;
```

A função `NovoNoDeTrie()`, discutida na **Seção 9.8.2**, que cria um nó da trie também precisa ser ligeiramente alterada para refletir a alteração do tipo de nó, como na função `NovoNoTrieMachado()` a seguir.

```
static tNoTrieMachado *NovoNoTrieMachado(void)
{
   tNoTrieMachado *pNo;
   int             i;

   pNo = malloc(sizeof(*pNo));

   if (pNo) {
      IniciaListaSE(&pNo->valores); /* Por enquanto, o nó não é final */

      for (i = 0; i < TAM_ALFABETO; i++) /* Inicia cada filho do nó com NULL */
         pNo->filhos[i] = NULL;
   }

   return pNo;
}
```

A função `IniciaListaSE()` chamada por `NovoNoTrieMachado()` inicia a lista encadeada representada pelo campo `valores` como uma lista vazia.

A função `EhNoFinalTrieMachado()` vista abaixo recebe o endereço de um nó como parâmetro e retorna `1`, se esse nó for final, ou `0`, em caso contrário. Nesta implementação, um nó é final se a lista encadeada apontada pelo campo `valores` não for vazia; se essa lista estiver vazia, o nó não é um nó final.

```
static int EhNoFinalTrieMachado(const tNoTrieMachado *pNo)
{
   return !EstaVaziaLSE(pNo->valores);
}
```

A função `InsereTrieMachado()` apresentada a seguir insere uma nova palavra (string) na trie. Comparada com a função de idêntica denominação apresentada na **Seção 9.8.2**, a função a seguir apresenta um parâmetro a mais (i.e., `valor`). Esse parâmetro será o conteúdo efetivo do nó que será acrescentado à lista encadeada apontada pelo campo `valores` do nó associado ao último caractere de uma palavra (representada pelo parâmetro `chave`).

```
void InsereTrieMachado(tNoTrieMachado *raiz, const char *chave, const tConteudo *valor)
{
   int             indice;/* Índice de um caractere no array de filhos de um nó */
   tNoTrieMachado *p = raiz; /* Ponteiro usado para descer na trie */

   for (; *chave; ++chave) {
         /* Obtém o índice do caractere corrente da chave */
      indice = INDICE_CARACTERE(*chave);

         /* Certifica-se que o índice obtido é válido */
      ASSEGURA( indice >= 0 && indice < TAM_ALFABETO,
                "Indice invalido em InsereTrieMachado()" );

         /* Verifica se o caractere corrente existe no presente nó */
      if (!p->filhos[indice])
            /* O referido caractere não existe e é preciso */
            /* criar mais um filho para o corrente nó      */
         p->filhos[indice] = NovoNoTrieMachado();

      p = p->filhos[indice]; /* Passa para o próximo nível da trie */
   }

      /* O último nó visitado deve indicar o final de uma chave. Neste */
      /* caso, armazena-se o valor  associado a essa chave na lista.   */
   InsereNoLSE(&p->valores, *valor);
}
```

As seguintes observações sobre a função `InsereTrieMachado()` são pertinentes:

- Se a chave já existir, um novo nó será inserido na lista encadeada apontada pelo campo `valores`. Esse nó conterá o índice da linha no texto e o índice da palavra nessa linha.
- O último nó visitado indica o final de um string, de maneira que se a chave era apenas prefixo de alguma outra chave na trie, ele passará a ser uma chave.

A função `BuscaTrieMachado()` a seguir efetua uma busca numa trie do tipo discutido aqui e retorna o endereço da lista encadeada contendo os valores associados à chave, se ela for encontrada, ou **NULL**, em caso contrário.

```
tListaSE BuscaTrieMachado(tNoTrieMachado *raiz, const char *chave)
{
   int             indice; /* Índice de um caractere no array de filhos de um nó */
   tNoTrieMachado *p = raiz; /* Ponteiro usado para descer na trie */

   for (; *chave && p; ++chave) {
         /* Obtém o índice do caractere corrente da chave */
      indice = INDICE_CARACTERE(*chave);

         /* Certifica-se que o índice obtido é válido */
      ASSEGURA(indice >= 0 && indice < TAM_ALFABETO, "Indice invalido em Busca()");

         /* Se o caractere corrente da chave não existe   */
         /* no presente nó, a chave não faz parte da trie */
      if (!p->filhos[indice])
         return NULL;

      p = p->filhos[indice]; /* Passa para o próximo nível da trie */
   }
```

```
      /* Se p aponta para um nó válido, esse nó aponta */
      /* para a lista que contém os valores procurados */
   if (p)
      return p->valores;

   return NULL; /* A chave não foi encontrada */
}
```

A função `RemoveEmTrie()`, que remove chaves e foi discutida na **Seção 9.8.2**, não precisa ser alterada para se adaptar à corrente situação, mas a função `RemoveEmTrieAux` daquela mesma seção precisa ser modificada para remover a lista encadeada apontada por um nó final que é removido ou deixa de ser um nó final. A alteração dessa última função é relativamente simples e não será apresentada aqui, mas o programa completo pode ser obtido no site dedicado ao livro na internet.

A função **main()** do programa que resolve o problema proposto é apresentada a seguir:

```
int main(void)
{
   tListaSE         lista;
   tTrieMachado     raiz;
   char            *umaPalavra;
   tConteudoMachado conteudo = {0, 0};
   int              op,
                    nOcorrencias;

   raiz = ConstroiTrieMachado(NOME_ARQUIVO); /* Cria a trie */

   while (1) {
      ApresentaMenu();

      op = LeOpcao("12345");

      if (op == '5') { /* Encerra o programa */
         printf("\nBye!\n");
         break; /* Saída do laço */
      }

      switch (op) {
         case '1': /* Acrescenta uma palavra na trie */
            printf("\n>>> Digite a palavra a ser acrescentada: ");
            umaPalavra = LeLinhaIlimitada(NULL, stdin);

            if (umaPalavra && ValidaPalavra(umaPalavra) )
               InsereTrieMachado(raiz, umaPalavra, &conteudo);

            free(umaPalavra);
            printf("\n>>> Acrescimo efetuado\n");

            break;

         case '2': /* Remove uma palavra da trie */
            printf("\n>>> Digite a palavra a ser removida: ");
            umaPalavra = LeLinhaIlimitada(NULL, stdin);

            RemoveTrieMachado(&raiz, umaPalavra);

            break;

         case '3': /* Verifica se uma palavra faz parte da trie */
            printf("\n>>> Digite a palavra a ser procurada: ");
            umaPalavra = LeLinhaIlimitada(NULL, stdin);

            lista = BuscaTrieMachado(raiz, umaPalavra);
```

```
            if (!lista) {
               printf("\n>>> \"%s\" nao foi encontrado\n", umaPalavra);
            } else {
               nOcorrencias = ComprimentoListaSE(lista);

               printf( "\n>>> \"%s\" foi encontrada %d vezes. "
                       "\n>>> Deseja exibir as ocorrencias" (s/n)? ",
                       umaPalavra, nOcorrencias );

               op = LeOpcao("sSnN");

               if (op == 's' || op == 'S')
                  ExibeLista(lista);
            }

            free(umaPalavra);

            break;

         case '4': /* Numero de palavras */
            printf( "\n\n>>> Numero de chaves na trie: %d\n",
                    NumeroDeChavesTrieMachado(raiz) );

            break;

         default: /* O programa não deve chegar até aqui */
            printf("\nEste programa contem um erro\n");
            return 1;
      }
   }

   return 0;
}
```

A função `NumeroDeChavesTrieMachado()`, que é chamada quando o usuário escolhe a opção `'4'`, calcula o número de chaves armazenadas na trie e sua implementação é deixada como exercício para o leitor.

A função `ConstroiTrieMachado()`, que é responsável pela construção da trie contendo as palavras da obra de Machado de Assis, é definida como:

```
tTrieMachado ConstroiTrieMachado(const char *nomeArq)
{
   FILE        *stream; /* Stream associado ao arquivo */
                        /* no qual ocorrerá a leitura  */
   char        *linha; /* Apontará para cada linha lida */
   int          nLinha = 0;
   tTrieMachado trie = IniciaTrieMachado();

      /* Abre arquivo em formato de texto para leitura */
   stream = fopen(nomeArq, "r");

      /* Se o arquivo não foi aberto, nada mais pode ser feito */
   ASSEGURA(stream, "Arquivo não foi aberto");

   while (1) {
         /* Lê uma linha no arquivo de entrada */
      linha = LeLinhaIlimitada(NULL, stream);

         /* Verifica se o final do arquivo foi atingido */
      if (feof(stream))
         break;

         /* Verifica se ocorreu erro de leitura */
```

```
      ASSEGURA(!ferror(stream), "Erro de leitura");

      ++nLinha; /* Mais uma linha lida */

         /* Insere as palavras da linha lida na trie */
      InsereLinhaEmTrieMachado(trie, linha, nLinha);

      free(linha);
   }

   fclose(stream); /* Processamento terminado. Fecha o arquivo. */

   free(linha);

   return trie;
}
```

A função `ConstroiTrieMachado()` cria uma trie com o conteúdo de um arquivo de texto e seu único parâmetro especifica o nome desse arquivo. Essa função retorna o endereço da trie criada e usa as seguintes funções auxiliares:

- `LeLinhaIlimitada()` — que foi definida no **Capítulo 9** do **Volume 1**
- `InsereLinhaEmTrieMachado()` — que será definida a seguir

A função `InsereLinhaEmTrieMachado()` insere as palavras que fazem parte de uma linha numa trie e seus parâmetros são:

- `raiz` (entrada) — raiz da trie
- `linha` (entrada/saída) — string que contém as palavras
- `nLinha` (entrada) — índice associado ao string que contém as palavras

```
void InsereLinhaEmTrieMachado(tTrieMachado raiz, char *linha, int nLinha)
{
   tConteudoMachado conteudo;
   int              posPalavra = 0;
   char            *pPalavra;

   pPalavra = strtok(linha, SEPARADORES); /* Obtém o primeiro token */

      /* Verifica se foi obtido algum token não vazio */
   if (!pPalavra || !*pPalavra)
      return; /* Não há palavras */

      /* O valor do campo 'linha' será o mesmo para todas as palavras desta linha */
   conteudo.linha = nLinha;

      /* A função ValidaPalavra() remove símbolos diacríticos e converte */
      /* maiúsculas em minúsculas para simplificar o problema. Se, após  */
      /* tudo isso, não sobrarem apenas letras de 'a' a 'z', o token é   */
      /* rejeitado como palavra.                                         */
   if (ValidaPalavra(pPalavra)) {
         /* Se desejar indexação de palavras a partir de */
         /* zero, troque incremento prefixo por sufixo   */
      conteudo.pos = ++posPalavra;

      InsereTrieMachado(raiz, pPalavra, &conteudo);
   }

      /* Obtém as demais palavras da linha e insere-as na trie */
   while (1) {
       pPalavra = strtok(NULL, SEPARADORES); /* Obtém o próximo token */

         /* Se não houver mais tokens, encerra o laço */
```

```
      if (!pPalavra)
         break;

        /* Insere apenas palavras válidas na trie */
      if (pPalavra && ValidaPalavra(pPalavra)) {
         conteudo.pos = ++posPalavra;

         InsereTrieMachado(raiz, pPalavra, &conteudo);
      }
   }
}
```

A função **`ValidaPalavra()`** chamada por **`InsereLinhaEmTrieMachado()`** remove símbolos diacríticos e converte maiúsculas em minúsculas para simplificar o problema. Se, após tudo isso, não sobrarem apenas letras de **`'a'`** a **`'z'`**, o token é rejeitado como palavra. Essa função é relativamente fácil de implementar e não será discutida em detalhes aqui. Ela não seria necessária na língua nativa de Shakespeare...

Quando esse programa é executado, ele apresenta o seguinte menu de opções:

```
>>>>> Opcoes <<<<<

 (1) Acrescenta palavra
 (2) Remove palavra
 (3) Busca palavra
 (4) Numero de palavras
 (5) Encerra o programa

 >>> Escolha sua opcao: 4
```

Quando o usuário escolhe a opção **`4`**, como mostrado acima, o programa responde:

```
>>> Numero de chaves na trie: 5843
```

Se o usuário escolher a opção **`3`**, poderá ocorrer o seguinte diálogo:

```
>>> Digite a palavra a ser procurada: amor
```

E o usuário obterá como resultado aquilo que aparece no início desta seção.

Solução de (b): Usando uma trie, casamento de palavras (texto) com um padrão de comprimento m tem custo temporal $\theta(m \cdot |\sum|)$, independentemente do tamanho do texto. Se o tamanho do alfabeto não muda (como ocorre com texto em linguagem natural), uma busca por palavra tem custo temporal $\theta(m)$; i.e., esse custo é proporcional ao tamanho do padrão.

9.10.3 Maior Prefixo Comum (MPC) a um Conjunto de Strings

Preâmbulo: Suponha que se tenha o seguinte conjunto de strings:

```
{"rata", "rato", "ratazana", "ratao", "ratinho"}
```

Então a pergunta é: *qual é o maior prefixo comum a todos esses strings?* Essa questão é fácil de responder por mera inspeção dos strings em apreço. Os prefixos comuns a todos strings desse conjunto são: **`"r"`**, **`"ra"`** e **`"rat"`**. Portanto a resposta à questão é **`"rat"`**, que é o maior desses strings.

Agora torna-se bem complicado encontrar o maior prefixo comum a um conjunto de strings quando esse conjunto é bem grande (imagine, por exemplo, um conjunto contendo milhares de strings). Nesse caso, existem diversas maneiras de resolver este problema e uma das mais fáceis e elegantes é usando uma trie para representar o conjunto de strings. A **Figura 9–50** mostra como o problema pode ser resolvido usando trie.

Como mostra a **Figura 9–50**, o prefixo comum a um conjunto de strings é obtido percorrendo-se o caminho que vai da raiz da trie até a primeira bifurcação dos strings na trie que começam com o mesmo caractere.

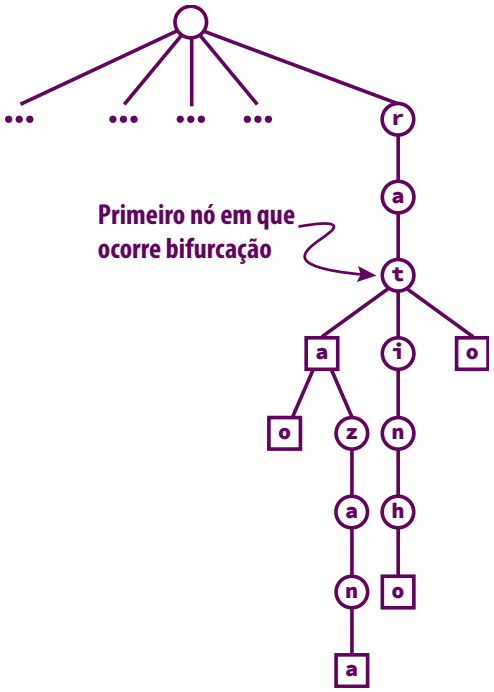


FIGURA 9–50: TRIE USADA PARA DETERMINAR O MPC DE UM CONJUNTO DE STRINGS

Problema: Escreva um programa que utilize uma trie para armazenar um conjunto de strings e determine o maior prefixo comum a esse conjunto de strings.

Solução: A implementação de trie apresentada na **Seção 9.8.2** pode ser utilizada com o acréscimo da função `MPC()` apresentada a seguir. Essa função encontra o maior prefixo comum (MPC) de um conjunto de strings armazenados numa trie e seu único parâmetro é o endereço da raiz da trie.

```
char *MPC(const tNoTrie *raiz)
{
   static char    prefixo[TAM_MAIOR_PALAVRA] = {0};
   const tNoTrie *p = raiz;
   int            indice,
                  tamPrefixo = 0;

      /* Tenta encontrar a primeira bifurcação da trie */
   while (NumeroDeFilhosDeNoDeTrie(p, &indice) == 1 && !p->ehFinal) {
      p = p->filhos[indice];

         /* Certifica-se que não haverá corrupção de memória */
      ASSEGURA(tamPrefixo < TAM_MAIOR_PALAVRA, "A Wikipedia estava errada");

         /* Acrescenta o caractere encontrado ao array que armazena o prefixo */
      prefixo[tamPrefixo] = 'a' + indice;

      ++tamPrefixo; /* O tamanho do prefixo aumentou */
   }

   return prefixo;
}
```

A constante `TAM_MAIOR_PALAVRA` utilizada pela função acima é definida como:

```
#define TAM_MAIOR_PALAVRA 46 + 1
```

Essa constante especifica o tamanho de um string capaz de conter a maior palavra da língua portuguesa (de acordo com a Wikipédia).

A função `MPC()` retorna o endereço do string que armazena o prefixo desejado. É importante notar que, como ela retorna o endereço de um array de duração fixa, a cada chamada dessa função, esse array pode ser sobrescrito. Essa função faz uso da função `NumeroDeFilhos()`, que conta e retorna o número de filhos de um nó de uma trie, e que usa os seguintes parâmetros:

- `pNo` (entrada) — endereço do referido nó
- `indice` (saída) — índice do último filho do nó que não é **NULL**

A função `NumeroDeFilhosDeNoDeTrie()` é definida como:

```
int NumeroDeFilhosDeNoDeTrie(const tNoTrie *pNo, int *indice)
{
   int i,
       nFilhos = 0; /* Armazenará o número de filhos */

   for (i = 0; i < TAM_ALFABETO; ++i)
      if (pNo->filhos[i]) {
         nFilhos++; /* Mais um filho encontrado */

         *indice = i; /* Atualiza o parâmetro indice */
      }

   return nFilhos;
}
```

A função **main()** a seguir complementa o programa solicitado:

```
int main(void)
{
   const char *chaves[] = { "rata", "rato", "ratazana", "ratao", "ratinho" };
   tTrie       raiz = IniciaTrie();
   int         i, nChaves;

   raiz = IniciaTrie(); /* Inicia a trie */

   nChaves = sizeof(chaves)/sizeof(chaves[0]);

      /* Constrói a trie */
   for (i = 0; i < nChaves; ++i)
      InsereEmTrie(raiz, chaves[i]);

      /* Efetua buscas na trie e apresenta os resultados */
   ResultadoBuscaEmTrie(raiz, "rato");
   ResultadoBuscaEmTrie(raiz, "ratazana");
   ResultadoBuscaEmTrie(raiz, "romano");
   ResultadoBuscaEmTrie(raiz, "ratinho");

   printf("\n\n>>> Numero de chaves na trie: %d\n", NumeroDeChavesEmTrie(raiz));

   printf("\n>>> Maior prefixo na trie: %s\n", MPC(raiz));

   RemoveEmTrie(raiz, "rato");
   RemoveEmTrie(raiz, "ratinho");

   printf("\n>>> Removida a chave \"%s\"", "rato");
   printf("\n>>> Removida a chave \"%s\"\n", "ratinho");
```

```
    /* Efetua buscas na trie e apresenta os resultados */
  ResultadoBuscaEmTrie(raiz, "rato");
  ResultadoBuscaEmTrie(raiz, "ratinho");

  printf("\n\n>>> Numero de chaves na trie: %d\n", NumeroDeChavesEmTrie(raiz));

  printf("\n>>> Maior prefixo na trie: %s\n", MPC(raiz));

  return 0;
}
```

Quando executado, o programa acima apresenta o seguinte resultado:

```
>>> A chave "rato" esta' presente na trie
>>> A chave "ratazana" esta' presente na trie
>>> A chave "romano" NAO esta' presente na trie
>>> A chave "ratinho" esta' presente na trie

>>> Numero de chaves na trie: 5

>>> Maior prefixo na trie: rat

>>> Removida a chave "rato"
>>> Removida a chave "ratinho"

>>> A chave "rato" NAO esta' presente na trie
>>> A chave "ratinho" NAO esta' presente na trie

>>> Numero de chaves na trie: 3

>>> Maior prefixo na trie: rata
```

A função `ResultadoBuscaEmTrie()` chamada por **main()** simplesmente efetua uma busca na trie usando a função `BuscaEmTrie()`, definida na **Seção 9.8.2**, e apresenta o resultado.

9.10.4 Maior Subsequência Comum (MSC) a Dois Strings

Preâmbulo: Aplicações em bioinformática precisam com frequência comparar cadeias de DNA de dois ou mais organismos diferentes. Uma **cadeia de DNA** consiste num string constituído por letras de um alfabeto de quatro letras $\sum = \{A, C, G, T\}$ que representam moléculas denominadas **bases**[13]. Assim qualquer cadeia de DNA pode ser representada como um string que usa apenas as letras desse alfabeto. Por exemplo, o DNA de um organismo (fictício) pode ser representado como S_1 = `"ACCGGTCGAGTGCGCGGAAGCCGGCCGAA"` e o DNA de outro organismo pode ser representado como S_2 = "`GTCGTTCGGAATGCCGTTGCTCTGTAAA`".

Uma das razões pelas quais se desejam comparar duas cadeias de DNA é determinar a similaridade entre elas, de modo que se possa decidir qual é a proximidade entre dois organismos. Essa similaridade pode ser definida de diversas maneiras. Por exemplo, pode-se concluir que duas cadeias de DNA são similares se uma delas é substring da outra. Mas outra abordagem pode considerar duas cadeias S_1 e S_2 similares se existe uma terceira cadeia S_3 na qual suas letras aparecem tanto em S_1 quanto em S_2. Nesse caso, as letras de S_3 aparecem na mesma ordem em que elas aparecem em S_1 e em S_2, mas não são necessariamente consecutivas. Ainda nesse caso, quanto maior for o comprimento de S_3, maior será a similaridade entre S_1 e S_2. Por exemplo, se S_1 for `"ACCGGTCGAGTGCGCGGAAGCCGGCCGAA"` e S_2 for "`GTCGTTCGGAATGCCGTTGCTCTGTAAA`", a maior cadeia S_3 será `"GTCGTCGGAAGCCGGCCGAA"`. Essa será a noção de similaridade a ser adotada aqui. Encontrar tal cadeia S_3 é um exemplo de um problema de determinação da **maior subsequência comum** (abreviadamente, **MSC**) a dois strings, que será formalmente descrito a seguir.

[13] Essas bases são **adenina** (representada por **A**), **guanina** (**G**), **citosina** (C) e **timina** (**T**).

Uma subsequência de um dado string é uma sequência obtida removendo-se zero ou mais caracteres desse string. Formalmente, dada uma sequência $X = \{x_1, x_2, ..., x_m\}$, outra sequência $Z = \{z_1, z_2, ..., z_k\}$ é uma subsequência de X se existe uma sequência estritamente crescente $\{i_1, i_2, ..., i_k\}$ de índices de X tal que $\forall j = 1, 2, ..., k$, tem-se que $x_{ij} = z_j$. Por exemplo, $Z = \{B, C, D, B\}$ é uma subsequência de $X = \{A, B, C, B, D, A, B\}$ sendo $\{2, 3, 5, 7\}$ a sequência correspondente de índices. Enfim essa definição significa que uma subsequência de X é uma sequência de caracteres presentes em X que não são necessariamente contíguos em X, mas que aparecem na mesma ordem em X.

Dadas duas sequências X e Y, diz-se que uma sequência Z é uma subsequência comum de X e Y se Z for a subsequência tanto de X quanto de Y. Por exemplo, se $X = \{A, B, C, B, D, A, B\}$ e $Y = \{B, D, C, A, B, A\}$, a sequência $\{B, C, A\}$ é uma subsequência comum de X e Y. No entanto, a sequência $\{B, C, A\}$ não é uma subsequência comum mais longa (MSC) de X e Y, pois ela tem comprimento 3 e a sequência $\{B, C, B, A\}$, que também é comum a X e Y, tem comprimento 4. Assim a sequência $\{B, C, B, A\}$ é uma MSC de X e Y, assim como é o caso da sequência $\{B, D, A, B\}$, uma vez que X e Y não possuem nenhuma subsequência comum de comprimento maior do que 4.

Resolver o problema MSC usando uma abordagem de bruta força requer enumerar todas as subsequências de X e testar cada uma delas para verificar se também é uma subsequência de Y, sempre considerando a maior subsequência que seja encontrada. Como X possui 2^m subsequências (pois cada caractere de X faz parte ou não de uma subsequência de X), essa abordagem tem custo temporal exponencial, o que a torna impraticável para sequências longas.

Problema: (a) Escreva uma função que apresenta a MSC de dois strings. (b) Usando a notação ó, avalie o custo temporal dessa função.

Observações: Este problema não deve ser confundido com os seguintes problemas clássicos, que são semelhantes em termos de enunciado, mas não são semelhantes em termos de complexidade de solução**:**

- Menor prefixo comum a um conjunto de string — esse problema foi resolvido na **Seção 9.10.3**.
- Menor substring a um conjunto de string — esse problema pode ser resolvido utilizando-se árvores de sufixos, que não são discutidas neste livro.

A solução a ser apresentada aqui é baseada no paradigma algorítmico conhecido como **programação** (ou **otimização**) **dinâmica**, mas você não precisa conhecer os princípios que norteiam esse paradigma para entender a solução proposta.

No problema MSC, têm-se dois strings, X e Y, de comprimento n e m, respectivamente, e pede-se para encontrar um string mais longo Z que seja uma subsequência de X e Y. Como X e Y são strings, tem-se um conjunto natural de índices com os quais subproblemas podem ser definidos. Tal subproblema consiste em determinar o comprimento de uma MSC de $X[0..i]$ e $Y[0..j]$ representada por $M[i, j]$.

Essa definição permite escrever $M[i, j]$ em termos de soluções de subproblemas divididos em dois casos:

- **Caso 1**: $X[i] = Y[j]$. Seja $c = X[i] = Y[j]$. Nesse caso, pode-se mostrar[14] que uma MSC de $X[0..i]$ e $Y[0..j]$ termina com c. Portanto pode-se considerar:

  $$M[i, j] = M[i - 1, j - 1] + 1 \text{ se } X[i] = Y[j] \quad \textbf{[1]}$$

[14] O leitor interessado em aspectos teóricos deste problema pode consultar Goodrich (2015) ou Cormen (2009) (v. **Bibliografia**).

A **Figura 9–51** ilustra este caso:

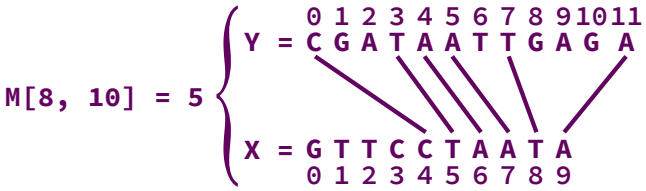


Figura 9–51: Exemplo do Caso 1 do Algoritmo MSC

- **Caso 2**: $X[i] \neq Y[j]$. Neste caso, não se pode ter uma subsequência comum que inclua tanto $X[i]$ quanto $Y[j]$. Quer dizer, uma subsequência comum pode terminar com $X[i]$, com $Y[j]$ ou com nenhum dos dois, mas não com esses dois valores ao mesmo tempo. Portanto considera-se:

$$M[i, j] = max\{M[i-1, j], M[i, j-1]\} \text{ se } X[i] \neq Y[j] \quad \textbf{[2]}$$

A **Figura 9–52** ilustra este caso:

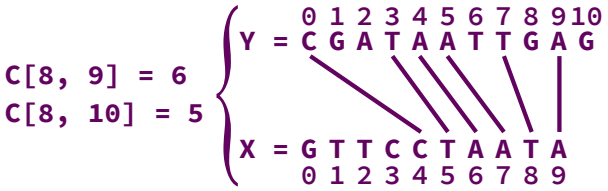


Figura 9–52: Exemplo do Caso 2 do Algoritmo MSC

Resumindo o que foi exposto acima, tem-se que:

$$M[i, j] = \begin{cases} 0 & \text{se } i = 0 \text{ ou } j = 0 \\ M[i-1, j-1] + 1 & \text{se } i, j > 0 \text{ e } x_i = x_j \\ max(M[i, j-1], M[i-1, j]) & \text{se } i, j > 0 \text{ e } x_i \neq x_j \end{cases}$$

Para que as equações **[1]** e **[2]** façam sentido quando $i = 0$ ou $j = 0$, define-se $M[i, -1] = 0$ para $i = -1, 0, 1, \ldots, n-1$ e $C[-1, j] = 0$ para $j = -1, 0, 1, \ldots, m-1$.

Para transformar a definição de $M[i, j]$ num algoritmo, é necessário usar uma matriz $M_{(n+1)\times(m+1)}$ iniciada com os casos de fronteira em que $i = 0$ ou $j = 0$. Ou seja, $M[i, -1] = 0$ para $i = -1, 0, 1, \ldots, n-1$ e $M[-1, j] = 0$ para $j = -1, 0, 1, \ldots, m-1$. Então iterativamente os valores de M são construídos até que se obtenha $M[n-1, m-1]$, que é o comprimento da MSC de X e Y.

Após obter os valores da matriz $M[i, j]$, construir uma MSC é relativamente fácil. Um método consiste em iniciar em $M[n-1, m-1]$ e examinar os elementos da matriz, construindo uma MSC do final para o início. Em qualquer elemento $M[i, j]$, determina-se se $X[i] = Y[j]$. Se esse for o caso, então considera-se $X[i]$ como o próximo caractere da subsequência (lembrando que $X[i]$ deve vir antes do último caractere encontrado, a não ser que não haja nenhum outro). Em seguida, passa-se para $M[i-1, j-1]$. Se $X[i] \neq Y[j]$, então passa-se para o maior valor entre $M[i, j-1]$ e $M[i-1, j]$. Esse procedimento encerra quando se atinge um valor de fronteira (i.e., com $i = -1$ ou $j = -1$).

A **Figura 9–53** ilustra o procedimento descrito acima. Nessa figura, o valor do elemento $M[6, 5]$, que se encontra na última linha e na última coluna da tabela, é o comprimento de uma MSC de X e Y. Para $i, j > 0$, o elemento $M[i, j]$ depende apenas do fato de $x_i = y_j$ e dos valores dos

elementos *M[i – 1, j]*, *M[i, j – 1]* e *M[i – 1, j – 1]* que são calculados antes de *M[i, j]*. Para obter os caracteres que constituem uma MSC, seguem-se as setas a partir do canto inferior direito da matriz. Na mesma figura, a sequência seguida aparece com fundo escurecido. Cada seta ↖ na sequência com fundo escurecido corresponde a um elemento para o qual $x_i = y_j$ de modo que o caractere correspondente faz parte de uma MSC.

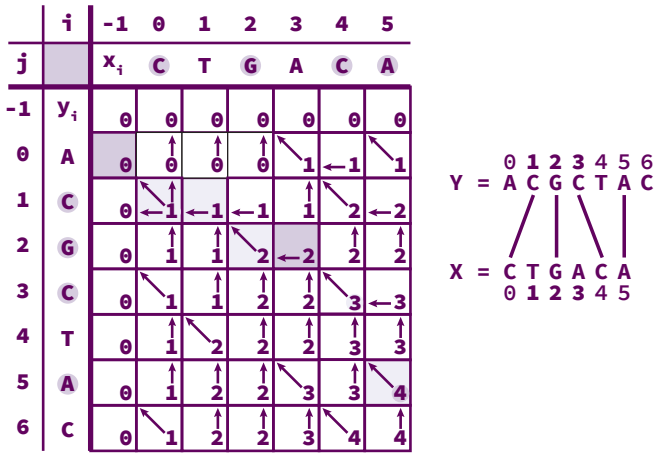


FIGURA 9–53: EXEMPLO DE DETERMINAÇÃO DE MSC

Solução de (a): A função `ExibeMSC()`, apresentada adiante, exibe na tela o comprimento do MSC de dois strings e o primeiro MSC encontrado seguindo o procedimento descrito acima. Os parâmetros dessa função são os strings que representam as sequências e os tamanhos desses strings.

```
void ExibeMSC( const char *X, const char *Y, int m, int n )
{
   int    i, j,
          indice,
          M[m + 1][n + 1]; /* Válido nos padrões C99 e C11 */
   char  *MSC;

   /* O primeiro segmento da função é dedicado ao cálculo do comprimento do MSC */

      /* Constrói a tabela M[m + 1][n + 1] de modo ascendente. M[i][j] */
      /* contém o comprimento de MSC de X[0..i - 1] e Y[0..j - 1]      */
   for (i = 0; i <= m; ++i)
      for (j = 0; j <= n; ++j)
         if (i == 0 || j == 0)
            M[i][j] = 0;
         else if (X[i - 1] == Y[j - 1])
            M[i][j] = M[i - 1][j - 1] + 1;
         else
            M[i][j] = MAX(M[i - 1][j], M[i][j - 1]);

      /* Exibe na tela o comprimento de MSC */
   printf( "\n>>> Comprimento de MSC de %s e %s: %d\n", X, Y, M[m][n] );

   /* O segundo segmento a seguir é dedicado à apresentação do MSC */

      /* Inicia 'indice' com o último valor da tabela */
   indice = M[m][n];

      /* Cria um array de caracteres para armazenar o MSC */
   MSC = calloc(indice + 1, sizeof(char));
```

```
      /* Inicia com o último elemento da matriz e prossegue */
      /* armazenando os caracteres no array MSC[]           */
   i = m;
   j = n;

   while (i > 0 && j > 0) {
         /* Se os caracteres corrente em X e Y são os     */
         /* mesmos, então esse caractere faz parte do MSC */
      if (X[i - 1] == Y[j - 1]) {
            /* Acrescenta o caractere corrente ao resultado */
         MSC[indice - 1] = X[i - 1];

         --i; /* Atualiza os valores de i, j e indice */
         --j;
         --indice;
      } else if (M[i - 1][j] > M[i][j - 1]) {
            /* Os caracteres correntes em X e Y não são os */
            /* mesmos. Prossegue em direção do valor maior */
         --i;
      } else
         --j; /* Idem */
   }

   printf("\n>>> MSC de %s e %s: %s\n", X, Y, MSC); /* Exibe na tela o MSC */

   free(MSC); /* Libera o espaço ocupado pelo array */
}
```

Solução de (b): A função `ExibeMSC()` é dividida em dois segmentos distintos: o primeiro segmento é responsável pelo cálculo do comprimento de uma MSC e o segundo segmento é responsável por encontrar uma MSC. O custo temporal do primeiro segmento é $\theta(n \cdot m)$, enquanto o segundo segmento apresenta um custo temporal adicional $\theta(n + m)$. Portanto o custo temporal dessa função é $\theta(n \cdot m)$.

9.10.5 Distância de Edição

Preâmbulo: Um verificador ortográfico faz sugestões para palavras que ele não encontra em seu repertório por julgá-las incorretas. Considerando uma palavra *s* (string) incorreta, um verificador ortográfico sugere palavras que se encontram em sua lista de palavras e que são próximas de *s*. Uma das formas mais rudimentares de implementar essas sugestões é por meio do conceito de **distância de edição**.

A distância de edição entre dois strings é o número mínimo de caracteres alterados, inseridos ou removidos num dos strings de modo que se obtenha como resultado o outro string. A **Tabela 9–6** apresenta exemplos de distâncias de edição entre dois strings.

Strings	Distância de Edição	Justificativa
`s1 = "abc"` `s2 = "abd"`	1	Basta alterar um caractere de `s1` para obter `s2` (ou vice-versa)
`s1 = "abc"` `s2 = "bd"`	2	O caractere `'a'` de `s1` deve ser removido e o caractere `'c'` deve ser alterado para `'d'` para obter `s2`
`s1 = "abc"` `s2 = "abc"`	0	Não é preciso alterar, inserir ou remover qualquer caractere de `s1` para obter `s2`

Tabela 9–6: Exemplos de Distância de Edição

Observação: Tipicamente, verificadores ortográficos utilizam um algoritmo conhecido como **distância de Levenshtein** ou um algoritmo ainda mais sofisticado com o mesmo propósito. Portanto distância de edição, conforme descrito aqui, serve apenas como introdução a esse tema.

Problema: (a) Escreva uma função que recebe dois strings como parâmetros e retorna a distância de edição entre eles. (b) Avalie o custo temporal dessa função em termos de notação ó.

Solução de (a): A função `DistanciaDeEdicao()` apresentada a seguir calcula e retorna a distância de edição de dois strings recebidos como parâmetros.

```
int DistanciaDeEdicao( char *s1, const char *s2)
{
   int da, /* Distância de edição devido a alteração */
       di, /* Distância de edição devido a inserção  */
       dr; /* Distância de edição devido a remoção   */

   /* Se um dos strings é vazio, a distância de edição é o tamanho do outro string */

   if(*s1 == 0)
      return strlen(s2);

   if(*s2 == 0)
      return strlen(s1);

     /* Se os dois primeiros caracteres correspondentes nos dois */
      /* strings são iguais, passa-se para os caracteres seguintes */
   if(*s1 == *s2)
      da = DistanciaDeEdicao(s1 + 1, s2 + 1);
   else /* Alteração */
      da = 1 + DistanciaDeEdicao(s1 + 1, s2 + 1);

   di = 1 + DistanciaDeEdicao(s1, s2 + 1); /* Inserção */

   dr = 1 + DistanciaDeEdicao(s1 + 1, s2); /* Remoção */

   return MenorDe3(da, di, dr);
}
```

A função `DistanciaDeEdicao()` verifica se algum dos strings recebidos como parâmetro é vazio. Se esse for o caso, ela retorna o comprimento do outro string como distância de edição. Caso contrário, ela verifica se os dois primeiros caracteres dos dois strings são iguais. Se eles forem iguais, a função obtém o valor de `da` chamando a si mesma recursivamente usando como parâmetros `s1 + 1` e `s2 + 1`. Por outro lado, se os dois primeiros caracteres dos dois strings não são iguais, a função supõe que a alteração de um caractere se faz necessária e o valor da variável `da` é obtido acrescentando-se `1` (devido a atualização) à distância de edição entre os strings `s1 + 1` e `s2 + 1`. Em seguida, a função `DistanciaDeEdicao()` é chamada recursivamente duas vezes:

[1] Na primeira dessas chamadas, essa função calcula a distância de edição devido a uma eventual inserção. Ou seja, ela calcula o valor de `di` acrescentando `1` à distância de edição entre os strings `s1` e `s2 + 1` para levar em consideração a remoção de um caractere de `s1` de modo a obter `s2`.

[2] Na segunda dessas chamadas, a função calcula o valor de `dr` acrescentando `1` à distância de edição entre os strings `s1 + 1` e `s2` para levar em consideração a inserção de um caractere em `s1` de modo a obter `s2`.

Ao final, após determinar os valores de `da`, `di` e `dr`, a função `DistanciaDeEdicao()` chama a função `MenorDe3()` para calcular o menor valor dentre os três valores inteiros `d1`, `d2`, `d3` e retorna esse valor. A função `MenorDe3()` é muito fácil de ser implementada e sua apresentação não cabe num livro de estruturas de dados.

Solução de (b): Como, para cada caractere em `s1`, a função `DistanciaDeEdicao()` é chamada recursivamente três vezes, o custo temporal pode ser calculado usando-se a seguinte relação de recorrência (v. **Capítulo 6** e **Apêndice B** do **Volume 1**):

$$T(n) = 3 \cdot T(n - 1)$$

em que $n = min\{|s1|, |s2|\}$ (i.e., n é o menor comprimento entre os strings `s1` e `s2`, que são parâmetros da função). Resolvendo-se essa relação de recorrência, obtém-se que o custo temporal da função é $\theta(3^n)$. Portanto esse custo é exponencial e não pode ser subestimado. Por exemplo, considere um programa contendo a função `DistanciaDeEdicao()` que seja executado como mostrado a seguir:

```
>>> Digite o primeiro string: Otorrinolaringol

>>> Digite o segundo string: Otorrinolaringl

Calculando distancia de edicao...

>>> Distancia de edicao entre
"Otorrinolaringol" (16 caracteres) e
"Otorrinolaringl" (15 caracteres): 1
```

O problema com o programa acima é que ele leva quase 20 minutos para encontrar a distância de edição entre os strings `"Otorrinolaringol"` e `"Otorrinolaringl"`. O computador utilizado na execução desse programa usa um processador Intel de *3,5 GHz* e tem memória de *16 GiB*. Para um programa que se propõe a ser um verificador ortográfico esse tempo corresponde a uma eternidade para o usuário, mas como o leitor foi avisado no início desta seção, esse tipo de distância de edição tem objetivo meramente didático e não deve ser usado na prática.

9.10.6 Casamento Léxico

Preâmbulo: Uma vantagem do algoritmo **KR** (v. **Seção 9.6**) é que ele permite resolver de modo eficiente uma generalização do problema de casamento de strings denominada **casamento léxico**. Nesse tipo de problema, tem-se um conjunto $P = \{p_1, p_2, \ldots, p_k\}$ contendo k padrões diferentes e um texto T. Então o problema consiste em encontrar todas as posições em T nos quais um padrão p_i de P é um substring. Neste contexto, o conjunto P é o **léxico** de strings que se espera verificar se eles se encontram em T. Por exemplo, P poderia consistir de um conjunto de palavras (ou frases) comumente encontradas na obra de um certo autor (p. ex., Machado de Assis) e T poderia ser um artigo que se suspeita que seja um plágio da obra desse mesmo autor. Então o investigador dessa suspeita tentaria encontrar as frases de P em T de modo a confirmar sua suspeita.

Problema: Escreva uma função que mostre com que frequência um léxico ocorre num texto. Para simplificar, suponha que todos os padrões desse léxico sejam do mesmo tamanho m.

Solução: A solução para o problema proposto consiste em duas etapas:

1. Na primeira etapa, o valor de dispersão de cada padrão do léxico é calculado e armazenado numa tabela que associa cada padrão a seu respectivo valor de dispersão.
2. Na segunda etapa, calcula-se o valor de dispersão de cada substring do texto que tenha tamanho m. Se for encontrado um casamento entre um determinado valor de dispersão encontrado no texto e um valor de dispersão calculado na primeira fase, comparam-se os respectivos strings para verificar se eles realmente casam ou se ocorre um falso casamento, como ilustra a **Figura 9–54**.

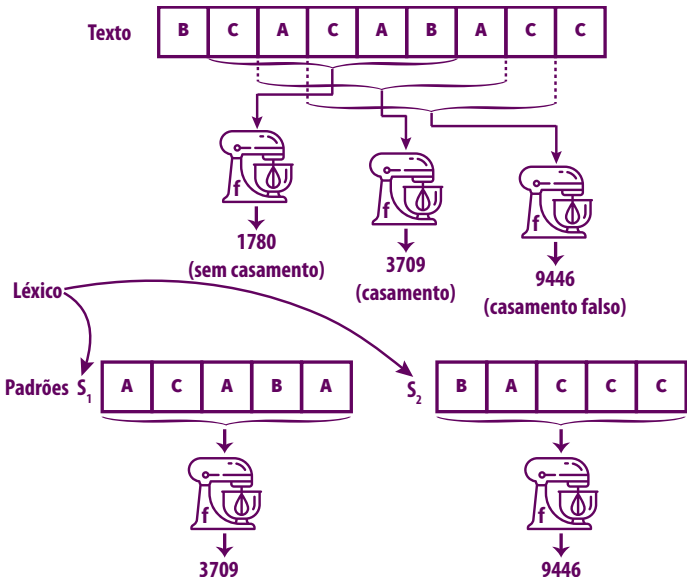


Figura 9–54: Exemplo de Casamento Léxico

Será usada a seguinte definição de tipo para a estrutura que armazena um padrão do léxico em questão:

```
typedef struct {
           char *padrao;      /* O padrão */
           int   dispersao;   /* Seu valor de dispersão */
           int   ocorrencias; /* Número de ocorrências no texto */
        } tPadraoLexico;
```

A função `DispersoesLexico()` calcula os valores de dispersão de um array de padrões (léxico) usando uma digital de Rabin e seus parâmetros são:

- `lexico` (entrada/saída) — array que contém os padrões
- n (entrada) — número de elementos do array
- `m` (entrada) — tamanho de cada string do array
- `base` (entrada) — base que será usada no cálculo dos valores de dispersão
- `primo` (entrada) — número primo que será usado no cálculo dos valores de dispersão

```
void DispersoesLexico(tPadraoLexico lexico[], int n, int m, int b, int q)
{
   int i, j;

      /* Calcula os valores de dispersão dos padrões */
   for (i = 0; i < n; ++i)
      for (j = 0; j < m; ++j)
         lexico[i].dispersao = ( lexico[i].dispersao*b + lexico[i].padrao[j] )%q;
}
```

A função `CasamentoLexico()`, definida adiante, verifica a ocorrência de algum padrão de um conjunto de padrões (léxico) num texto usando o algoritmo **KR**. Os parâmetros dessa função são:

- `nomeArq` (entrada) — nome do arquivo que contém o texto
- `lexico` (entrada) — array que contém os padrões
- `k` (entrada) — número de elementos do array (léxico)
- `m` (entrada) — tamanho de cada padrão no léxico
- `b` (entrada) — a base numérica utilizada em cálculo de valores de dispersão
- `q` (entrada) — o número primo utilizado em cálculo de valores de dispersão

```
void CasamentoLexico( const char *nomeArq, tPadraoLexico lexico[],
                      int k, int m, int b, int q)
{
   int   dispersaoTexto = 0, /* Dispersão de cada janela */
         maiorPotencia = 1, /* Valor da base elevado a m - 1 */
         i, j,
         nLinha; /* Tamanho de uma linha lida no arquivo */
   char *linha; /* Apontará para cada linha lida */
   FILE *stream; /* Stream associado ao arquivo no qual ocorrerá a leitura */

   stream = fopen(nomeArq, "r"); /* Abre arquivo em formato de texto para leitura */

      /* Se o arquivo não foi aberto, nada mais pode ser feito */
   ASSEGURA(stream, "Arquivo não foi aberto");

      /* Calcula o valor da maior potência do polinômio */
   for (i = 0; i < m - 1; ++i)
      maiorPotencia = maiorPotencia*b%q;

      /* Lê cada linha do arquivo de texto e conta o número de ocorrências */
      /* de cada padrão que compõe o léxico nessa linha                    */
   while (1) {
         /* Lê uma linha no arquivo de entrada */
      linha = LeLinhaIlimitada(&nLinha, stream);

         /* Se a linha for NULL, encerra-se o laço */
      if (!linha)
         break;

      dispersaoTexto = 0; /* Inicia o valor de dispersão das janelas da linha */

         /* Calcula o valor de dispersão da primeira */
         /* janela da linha lida recentemente        */
      for (i = 0; i < m; ++i)
         dispersaoTexto = (dispersaoTexto*b + linha[i])%q;

         /* Procura casamentos entre os padrões do léxico e a linha corrente */
      for (i = 0; i <= nLinha - m; ++i ) {
            /* Checa casamentos dos padrões com a janela corrente da linha corrente */
         for (j = 0; j < k; ++j)
            if ( lexico[j].dispersao == dispersaoTexto &&
                 !memcmp(lexico[j].padrao, linha + i, m) )
                  ++lexico[j].ocorrencias;

            /* Calcula o valor de dispersão da próxima janela da linha */
         dispersaoTexto = (b*(dispersaoTexto-linha[i]*maiorPotencia) + linha[i+m])%q;

            /* Corrige o valor de dispersão se ele for negativo */
         if(dispersaoTexto < 0)
            dispersaoTexto += q;
      }

      free(linha); /* Libera o espaço ocupado pela linha */
   }

   fclose(stream); /* Fecha o arquivo */

      /* Verifica se ocorreu erro de leitura */
   ASSEGURA(!ferror(stream), "Erro de leitura");
}
```

A função **main()** apresentada a seguir pode ser usada para completar o programa de casamento léxico:

```
int main(void)
{
   tPadraoLexico lexico[] = { {"casa", 0, 0}, {"olho", 0, 0},
                              {"vida", 0, 0}, {"hora", 0, 0},
                              {"amor", 0, 0}, {"alma", 0, 0},
                              {"nome", 0, 0}, {"modo", 0, 0},
                              {"novo", 0, 0}, {"self", 0, 0}
                            };
   int           m = strlen(lexico->padrao), /* Os padrões têm o mesmo tamanho */
                 k = sizeof(lexico)/sizeof(lexico[0]), /* No. de padrões no léxico */
                 i,
                 b = BASE,
                 q = PRIMO,
                 qMax = INT_MAX/(BASE*BASE - BASE);

      /* Teste preventivo de overflow */
   if (q > qMax) {
      qMax = MaiorPrimo(qMax);
      printf("\n\t\a>>> Ocorrera' overflow!\n"
             "\n\t>>> O maior valor de q deveria ser %d\n", qMax);
      return 1;
   }

      /* Calcula os valores de dispersão dos padrões que constituem o léxico */
   DispersoesLexico(lexico, k, m, b, q);

      /* Efetua o casamento dos padrões com o texto */
   CasamentoLexico(NOME_ARQUIVO, lexico, k, m, b, q);

      /* Apresenta o resultado do casamento léxico */
   for (i = 0; i < k; ++i)
      if (lexico[i].ocorrencias)
         printf("\n\t>>> \"%s\" foi encontrado %d %s no texto", lexico[i].padrao,
                lexico[i].ocorrencias, lexico[i].ocorrencias > 1 ? "vezes" : "vez");
      else
         printf("\n\t>>> \"%s\" nao foi encontrado no texto", lexico[i].padrao);

   putchar('\n'); /* Enfeite */

   return 0;
}
```

Essa função **main()** faz uso das constantes simbólicas `PRIMO` e `BASE`, que foram discutidas na **Seção 9.6**, e da constante `NOME_ARQUIVO`, que representa nome do arquivo de texto utilizado. A função `MaiorPrimo()` chamada por **main()** também foi discutida na referida seção.

Quando um programa constituído pelas funções acima é executado, ele produz como resultado:

```
>>> "casa" foi encontrado 119 vezes no texto
>>> "olho" foi encontrado 114 vezes no texto
>>> "vida" foi encontrado 83 vezes no texto
>>> "hora" foi encontrado 66 vezes no texto
>>> "amor" foi encontrado 128 vezes no texto
>>> "alma" foi encontrado 52 vezes no texto
>>> "nome" foi encontrado 7 vezes no texto
>>> "modo" foi encontrado 30 vezes no texto
>>> "novo" foi encontrado 14 vezes no texto
>>> "self" nao foi encontrado no texto
```

9.10.7 Casamento de Strings em Fluxo Contínuo

Problema: (a) Escreva um programa que procura casar um string (padrão) com um texto cujos caracteres são lidos um a um num meio de entrada. (b) Apresente uma função **main()** que use a função solicitada no item (a) para (1) tentar casar um determinado padrão com um texto introduzido pelo usuário via teclado e (2) tentar casar outro padrão com um texto armazenado em arquivo. **Observação:** Em nenhuma situação, o texto deve ser armazenado pelo programa que o recebe como entrada.

Solução de (a): De todos os algoritmos de casamento de strings examinados neste livro, o algoritmo de Knuth, Morris e Pratt (**KMP**) é o mais adequado para resolver o problema proposto, pois ele não apresenta retrocesso (v. **Seção 9.1.3**).

A função `CasaFluxoContinuo()` a seguir é uma adaptação do algoritmo **KMP** para resolver o problema em questão. Ou seja, ela tenta casar um string (padrão) com um texto cujos caracteres são lidos um a um num meio de entrada. Essa função retorna a posição da primeira ocorrência do padrão no texto, se ele for encontrado, ou `-1`, em caso contrário. Os parâmetros da função `CasaFluxoContinuo()` são:

- `p` (entrada) — string que representa o padrão
- `stream` (entrada) — stream associado ao arquivo que contém o texto
- `num` (saída) — se não for **NULL**, esse parâmetro armazenará o número de caracteres lidos

O parâmetro `num` da função `CasaFluxoContinuo()` é útil na fase de testes e depuração do programa que usa essa função.

```
int CasaFluxoContinuo(const char *p, FILE *stream, int *num)
{
   int c,        /* Armazena cada caractere lido */
       nCar = 0, /* Número de caracteres lidos */
       m = strlen(p), /* Comprimento do padrão */
       i = 0, /* Índice de um caractere do padrão */
      *tab; /* Ponteiro para a tabela de prefixos */

      /* Cria a tabela de prefixos */
   tab = CriaTabelaKMP(malloc(m*sizeof(int)), p);

   c = fgetc(stream); /* Lê o primeiro caractere do texto */

      /* O laço a seguir encerra quando o final do arquivo */
      /* for atingido ou for encontrado um casamento       */
   while (1) {
         /* Verifica se o final de arquivo foi atingido */
         /* ou ocorreu algum erro de leitura            */
      if (c == EOF)
         break;

      ++nCar; /* Foi lido mais um caractere */

         /* Enquanto os caracteres ora sendo comparados não casarem, associa */
         /* i ao índice do padrão indicado pela tabela de prefixos           */
      while (i >= 0 && c != p[i])
          i = tab[i];

      ++i; /* Passa para o próximo caractere do padrão */

         /* Se já ocorreu casamento, encerra o laço */
      if (i >= m)
         break;

      c = fgetc(stream); /* Lê o próximo caractere do texto */
   }
```

```
   /* O array que armazena a tabela de prefixos não é mais necessário */
   free(tab);

   /* Se o terceiro parâmetro não for NULL, armazena */
   /* nele o número de caracteres lidos              */
   if (num)
      *num = nCar;

   /* Se o padrão foi encontrado, retorna-se sua      */
   /* posição no texto; caso contrário, retorna-se -1 */
   return i == m ? nCar - m : -1;
}
```

Note que a função `CasaFluxoContinuo()` chama a função `CriaTMB()`, que é aquela mesma definida na **Seção 9.3**.

É importante ainda observar que a função `CasaFluxoContinuo()` encerra apenas quando é encontrado um casamento ou quando o final do arquivo é atingido, o que pode ser um problema para o usuário desavisado que introduz um texto via teclado e não sabe simular final de arquivo. Ou seja, o usuário precisa ser um tanto *nerd* para saber que `[CTRL] + [D]` e `[CTRL] + [Z]` são usados com esse propósito em sistemas das famílias Unix e Windows/DOS, respectivamente.

Solução de (b): A função **main()** a seguir atende o requisito (b) do enunciado do problema.

```
int main(void)
{
   char *p1 = "amor",/* Padrão usado na primeira parte do programa */
        *p2 = "ATGAGCGGCGCCTGCA"; /* Padrão a ser usado na     */
                                  /* segunda parte do programa */
   int   pos, /* Possível posição de casamento  */
         nCaracteres; /* Número de caracteres lidos */
   FILE *stream; /* Stream associado ao arquivo que contém um */
                 /* banco de dados de DNA em formato FASTA    */

      /*                                                        */
      /* Primeira parte: casamento com texto digitado via teclado */
      /*                                                        */

   printf("Digite um texto falando em \"%s\":\n> ", p1);

      /* Chama a função CasaFluxoContinuo() para ler */
      /* o texto e verificar se ele contém o padrão  */
   pos = CasaFluxoContinuo(p1, stdin, &nCaracteres);

      /* Apresenta o resultado */
   if (pos >= 0)
      printf("\n\t>>> \"%s\" aparece na posicao %d do texto\n", p1, pos);
   else
      printf( "\n\t>>> \"%s\" NAO aparece no texto\n", p1);

      /* Informa quantos caracteres foram lidos */
   printf("\n\t>>> Foram lidos %d caracteres\n", nCaracteres);

      /*** Segunda parte: casamento com texto armazenado em arquivo ***/

      /* Tenta abrir arquivo de dados em modo de texto apenas para leitura */
   stream = fopen(NOME_ARQ, "r");

      /* Se o arquivo não foi aberto, encerra o programa */
   if (!stream) {
```

```
      printf("\nArquivo \"%s\" nao pode ser aberto\n", NOME_ARQ);
      return 1;
   }

      /* Chama a função CasaFluxoContinuo() para ler  */
      /* o arquivo e verificar se ele contém o padrão */
   pos = CasaFluxoContinuo(p2, stream, &nCaracteres);

   fclose(stream); /* Arquivo já pode ser fechado */

      /* Apresenta o resultado */
   if (pos >= 0)
      printf("\n\t>>> \"%s\" aparece na posicao %d do \n", p2, pos);
   else
      printf( "\n\t>>> \"%s\" NAO aparece no \n", p2);

   printf("\t>>> arquivo \"%s\"\n", NOME_ARQ);

      /* Informa quantos caracteres foram lidos */
   printf("\n\t>>> Foram lidos %d caracteres\n", nCaracteres);

   return 0;
}
```

Um exemplo de execução de um programa contendo essas duas últimas funções é apresentado a seguir:

```
Digite um texto falando em "amor":
> E' o amor oooooo

        >>> "amor" aparece na posicao 5 do texto

        >>> Foram lidos 9 caracteres

        >>> "ATGAGCGGCGCCTGCA" aparece na posicao 54161148 do
        >>> arquivo "I:\Dados\DNA.txt"

        >>> Foram lidos 54161164 caracteres
```

O valor da constante `NOME_ARQ` usada pela função **main()** acima é `"DNA.txt"`, que é o nome de um arquivo de texto contendo parte do genoma humano (v. **Apêndice A**).

9.11 Exercícios de Revisão

Conceitos (Seção 9.1)

1. (a) O que é casamento de strings? (b) Por que essa operação é tão importante em programação?
2. No contexto de casamento de strings, defina:
   (a) Padrão
   (b) Texto
   (c) Alfabeto
   (d) Substring
3. Defina os seguintes conceitos no contexto de casamento de strings:
   (a) Prefixo
   (b) Prefixo próprio
   (c) Sufixo
   (d) Sufixo próprio
4. (a) O que é uma borda? (b) Qual deve ser o comprimento mínimo de um string para que ele possa ter uma borda?

5. Quais são as bordas (se alguma existe) de cada um dos seguintes strings:
   (a) `"CGATT"`
   (b) `"AAA"`
6. Qual é o tamanho da maior borda de cada um dos seguintes strings?
   (a) `"A"`
   (b) `"AA"`
   (c) `"AAA"`
   (d) `"ACCA"`
7. Defina o conceito de salto utilizado por um algoritmo de casamento de strings.
8. (a) O que é um algoritmo de casamento de strings com retrocesso? (b) Exiba um exemplo de algoritmo de casamento de strings que apresente retrocesso e de outro que não apresenta retrocesso.
9. Em quais situações o uso de um algoritmo de casamento de strings com retrocesso não é apropriado?
10. Nas análises de algoritmos de casamento de strings, por que não se considera como melhor caso a situação na qual o padrão encontra-se na primeira janela de texto?

Casamento de Strings por Força Bruta (FB) (Seção 9.2)

11. (a) Descreva o algoritmo de casamento de strings por força bruta. (b) Por que esse algoritmo recebe essa denominação?
12. (a) Qual é o melhor caso do algoritmo **FB**? (b) Qual é o custo temporal desse algoritmo nesse caso?
13. (a) Qual é o pior caso do algoritmo **FB**? (b) Qual é o custo temporal desse algoritmo nesse caso?
14. Mostre por meio de diagramas quais são as comparações efetuadas quando se tenta casar o padrão `"10001"` com o texto `"000010001010001"` usando o algoritmo **FB**.

Algoritmo de Knuth, Morris e Pratt (KMP) (Seção 9.3)

15. Quais são as duas principais diferenças entre os algoritmos **FB** e **KMP**?
16. Apresente a tabela de maiores bordas do string `"AGCTAGT"`.
17. Apresente a tabela de maiores bordas do string `"abacab"`.
18. Qual é o papel desempenhado pela tabela de maiores bordas no algoritmo **KMP**?
19. (a) Apresente todos os prefixos próprios do string `"CGTACGTT"`. (b) Apresente todos os sufixos próprios do string `"CGTACGTT"`. (c) Quais prefixos próprios do string `"CGTACGTT"` também são sufixos próprios desse mesmo string?
20. Apresente a tabela de maiores bordas para o padrão `"aaaaaaaa"` de acordo com o algoritmo **KMP**.
21. Quais são as bordas do string `"aaabbaaa"`?
22. Qual é a maior borda do string `"CGTACGTTCGTACG"`?
23. Apresente a tabela de maiores bordas para o padrão `"01101011111011"`.
24. (a) Apresente a tabela de bordas para o padrão `"10001"`. (b) Mostre por meio de diagramas quais são as comparações efetuadas quando se tenta casar o padrão `"10001"` com o texto `"000010001010001"` usando o algoritmo **KMP**.
25. (a) Qual é o melhor caso do algoritmo **KMP**? (b) Qual é o custo temporal desse algoritmo nesse caso?
26. (a) Qual é o pior caso do algoritmo **KMP**? (b) Qual é o custo temporal desse algoritmo nesse caso?

Algoritmo de Boyer e Moore (BM) (Seção 9.4)

27. Em linhas gerais, em que diferem os algoritmos **BM** e **KMP**?
28. Descreva a regra do mau caractere usada pelo algoritmo **BM**.

29. O que é um bom sufixo de acordo com o algoritmo **BM**.

30. (a) Descreva o **Caso 1** da regra do bom sufixo do algoritmo **BM**. (b) Descreva o **Caso 2** da regra do bom sufixo do algoritmo **BM**.

31. Por que as regras usadas pelo algoritmo **BM** são denominadas *heurísticas*?

32. Descreva o algoritmo usado para obtenção da tabela de bons sufixos do algoritmo **BM**.

33. Por que a regra do bom sufixo do algoritmo **BM** não deve ser desprezada?

34. Mostre por meio de diagramas quais são as comparações efetuadas quando se tenta casar o padrão `"10001"` com o texto `"000010001010001"` usando o algoritmo **BM**, supondo o uso do alfabeto `Σ = {0, 1}`.

Algoritmo (ou Simplificação) de Horspool (BMH) (Seção 9.5)

35. Descreva a simplificação de Horspool.

36. Qual é a característica única no algoritmo **BMH** que o torna diferente dos demais algoritmos de casamentos de strings apresentados neste livro?

37. Quais são as diferenças entre a tabela de mau caractere do algoritmo **BM** e a tabela de saltos do algoritmo **BMH**?

38. Mostre por meio de diagramas quais são as comparações efetuadas quando se tenta casar o padrão `"10001"` com o texto `"000010001010001"` usando o algoritmo **BMH**, supondo o uso do alfabeto `Σ = {0, 1}`.

39. Apresente a tabela resultante do processamento do padrão `"CGTACGTTCGTAC"` por cada um dos seguintes algoritmos. Quando for necessário, suponha que o alfabeto em questão é `Σ = {A, C, G, T}`.

(a) Algoritmo **KMP**

(b) Algoritmo **BM** (nesse caso, apenas a tabela de maus caracteres)

(c) Algoritmo **BMH**

40. Apresente a tabela de saltos usada pelo Algoritmo **BMH** para o padrão:

```
"tv faz quengo explodir com whisky jb"
```

supondo que o alfabeto em questão seja (note que o primeiro caractere desse alfabeto é espaço em branco):

```
Σ = { , a, b, c, d, e, f, g, h, i, j, k, l, m,
      n, o, p, q, r, s, t, u, v, w, x, y, z}
```

Algoritmo de Karp e Rabin (KR) (Seção 9.6)

41. Qual é a característica única no algoritmo **KR** que o torna diferente dos demais algoritmos de casamentos de strings apresentados neste capítulo?

42. (a) O que é valor de dispersão? (b) O que é função de dispersão?

43. Por que uma função que soma os valores dos caracteres de um string não deve ser usada como função de dispersão para strings?

44. (a) O que é um casamento falso no algoritmo **KR**? (b) Por existem casamentos falsos no algoritmo **KR**?

45. Descreva o método de Horner. (Essa é uma questão mais apropriada para o ENEM!)

46. Qual é a importância desse método no algoritmo **KR**? (Essa é uma questão de estruturas de dados.)

47. (a) O que é uma função de dispersão rolante? (b) Que vantagens uma função dessa natureza oferece com relação a uma função de dispersão comum no algoritmo **KR**?

48. (a) O que é casamento por impressão digital? (b) Qual é a justificativa para essa denominação?

49. Por que se usa resto de divisão (*mod*) em funções de dispersão no algoritmo **KR**?

50. (a) Qual é o melhor caso do algoritmo **KR**? (b) Qual é o custo temporal desse algoritmo nesse caso?

51. (a) Qual é o pior caso do algoritmo **KR**? (b) Qual é o custo temporal desse algoritmo nesse caso?

52. (a) O que é fase de pré-processamento de um algoritmo de casamento de strings? (b) Qual é o único algoritmo de casamento de strings discutido aqui que não apresenta essa fase?

53. O que ocorre na fase de pré-processamento do algoritmo **KR**?

54. Na utilização do algoritmo **KR**, suponha que a base de conversão seja `10` e que o número primo utilizado seja `31`. Quantas tentativas de casamento o referido algoritmo efetuará antes de encontrar o padrão `"53"` no texto `"4131592653589793"`?

55. Considerando o alfabeto genético `Σ = {A, C, G, T}` e supondo que $q = 13$, determine o número de casamentos falsos encontrados pelo algoritmo **KR** quando ele tenta casar o padrão `"GCGG"` com o texto `"AAGCGATCGGTCCAGCGGCCAB"`.

56. Quais são os problemas apresentados pela função `DigitalRabinErrada()` discutida na **Seção 9.6**?

57. Quais são as diferenças entre o operador euclidiano *mod* e o operador **%** da linguagem C?

58. Explique como são escolhidos os valores de b e q do algoritmo **KR**.

59. Quando a aplicação do operador **%** de C resulta num valor negativo, como se corrige esse resultado para que ele se assemelhe ao resultado do operador euclidiano *mod*?

60. Considerando o alfabeto genético `Σ = {A, C, G, T}`, suponha que se deseja procurar casamentos entre strings constituídos por símbolos desse alfabeto (e. g, entre `"ACCT"` e `"TAAGGACCTCCAAA"`). Se forem usados os valores $b = 4$ como tamanho do alfabeto e $q = 65075243$ como número primo, poderá ocorrer overflow? Explique.

Comparando Algoritmos de Casamento de Strings (Seção 9.7)

61. Considerando o padrão `"AGAGAGAG"` e o texto `"AGAGAGAGAAGAGAGAGAAAAAAAA"`, apresente diagramas semelhantes àqueles apresentados na **Seção 9.1** que mostrem todos os passos seguidos pelos seguintes algoritmos até encontrar o primeiro casamento.

(a) Algoritmo **FB**
(b) Algoritmo **KMP**
(c) Algoritmo **BM**
(d) Algoritmo **BMH**
(e) Algoritmo **KR**

62. Considerando o padrão `"aaaaaaab"` e o texto `"aaaaaaaaaaaaaaaaaaaaaaaab"`, apresente diagramas semelhantes àqueles apresentados na **Seção 9.1** que mostrem todos os passos seguidos pelos seguintes algoritmos até encontrar o primeiro casamento.

(a) Algoritmo **FB**
(b) Algoritmo **KMP**
(c) Algoritmo **BM**
(d) Algoritmo **BMH**
(e) Algoritmo **KR**

63. Qual é o custo espacial de cada um dos seguintes algoritmos:

(a) Algoritmo **FB**
(b) Algoritmo **KMP**
(c) Algoritmo **BM**
(d) Algoritmo **BMH**
(e) Algoritmo **KR**

64. Em que situações o uso de cada um dos seguintes algoritmos de casamento de strings é recomendável?

(a) Algoritmo **FB**
(b) Algoritmo **KMP**
(c) Algoritmo **BM**
(d) Algoritmo **BMH**
(e) Algoritmo **KR**

Tries (Seção 9.8)

65. (a) O que é uma trie? (b) Cite três aplicações de tries.

66. Tries podem ser usadas para implementação de tabelas de busca do mesmo modo que árvores binárias, por exemplo? Explique.

67. Considerando o alfabeto genético `Σ = {A, C, G, T}`, desenhe uma trie para o seguinte conjunto de strings:

```
S = {"ACAC", "CACA", "GGGGG", "CCAAAA", "GAA", "CCAAGG", "GCGG", "GCGA"}
```

68. Construa uma trie que contenha todos os strings resultantes da combinação das letras `a`, `b`, `c`, `d` e que comecem com a letra `a`. O alfabeto considerado deve ser `Σ = {a, b, c, d}`.

69. (a) O que é nó final de uma trie? (b) Qual é a diferença entre nó final e nó-folha de uma trie?

70. Por que tries não são adequadas para implementação de tabelas de buscas residentes em memória secundária?

71. Como prefixos podem ser representados numa trie?

72. Descreva os algoritmos de (a) busca, (b) inserção e (c) remoção numa trie.

73. Como é feito o mapeamento entre os caracteres de um alfabeto e o array de ponteiros de cada nó usado na implementação trie apresentada na **Seção 9.8.2**?

74. (a) Qual é o custo temporal de uma operação de busca numa trie? (b) Qual é o custo temporal de uma operação de inserção numa trie? (c) Qual é o custo temporal de uma operação de remoção numa trie?

75. Qual é o custo espacial da implementação de trie apresentada na **Seção 9.8.2**.

76. Apresente vantagens e desvantagens do uso de tries com relação a outras estruturas de dados hierárquicas usadas na implementação de tabelas de busca.

77. Apresente duas maneiras alternativas de implementação de tries que apresentem custo espacial bem menor do que a implementação de trie apresentada na **Seção 9.8.2**.

Casamento de Strings vs Casamento de Palavras (Seção 9.9)

78. (a) O que é uma palavra no contexto de casamento de strings? (b) O que é casamento de palavras? (c) Qual é a diferença entre casamento de palavras e casamento de strings?

79. (a) Como você efetua buscas por palavras em seu editor de texto favorito? (b) Como você efetua buscas por strings em seu editor de texto favorito?

80. Descreva como tries facilitam casamentos de palavras.

Exemplos de Programação (Seção 9.10)

81. (a) Descreva o funcionamento da função **strtok()**. (b) Qual é um possível problema dessa função quando ela é usada no processamento de arquivos CSV?

82. (a) Por que o primeiro parâmetro de **strtok()** não deve ser um string constante? (b) O que pode acontecer se essa recomendação não for seguida?

83. Em que diferem as funções `ObtemTokens()` e **strtok()**?

84. Por que, quando uma palavra é encontrada, o programa da **Seção 9.10.2** apresenta suas ocorrências em ordem inversa de linhas?

85. Descreva o algoritmo usado para encontrar o maior prefixo comum (MPC) a um conjunto de strings implementado na **Seção 9.10.3**.

86. Encontre um MSC para os strings `"10010101"` e `"010110110"`.

87. Apresente uma situação em Engenharia de Software na qual o problema de determinação da maior subsequência comum (MSC) a dois strings discutido na **Seção 9.10.4** esteja presente.

88. O que é distância de edição e qual é sua importância prática?

89. Por que o algoritmo para cálculo de distância de edição descrito e implementado na **Seção 9.10.5** é impraticável?

90. Utilize seu mecanismo de busca favorito para pesquisar sobre a distância de Levenshtein e descreva o algoritmo utilizado para calcular essa distância.

91. (a) O que é casamento léxico? (b) Quais vantagens o algoritmo **KR** apresenta com relação aos demais algoritmos de casamento de strings discutidos neste capítulo que o faz ser escolhido para resolver o problema casamento léxico?

92. (a) Descreva o problema de casamento de padrão com texto em fluxo contínuo. (b) Explique por que, dentre os algoritmos de casamento de strings estudados neste capítulo, o mais adequado para resolver esse problema é o algoritmo **KMP**.

93. O algoritmo **FB** pode ser utilizado para resolver o problema de casamento de padrão com texto em fluxo contínuo discutido na **Seção 9.10.7**? Explique sua resposta.

94. O algoritmo **KR** pode ser utilizado para resolver o problema de casamento de padrão com texto em fluxo contínuo discutido na **Seção 9.10.7**? Explique sua resposta.

95. Se o algoritmo **BM** (ou **BMH**) não apresenta retrocesso, por que ele não pode ser usado para resolver o problema de casamento de padrão com texto em fluxo contínuo discutido na **Seção 9.10.7**?

96. A função `CasaFluxoContinuo()` apresenta uma pequena inconveniência para o usuário de um programa que a utiliza. (a) Qual é essa inconveniência? (b) Esse problema pode resolvido simplesmente substituindo-se a instrução **if**:

```
if (c == EOF)
   break;
```

por:

```
if (c == '\n' || c == EOF)
   break;
```

Explique por que essa solução não foi adotada.

9.12 Exercícios de Programação

EP9.1 Escreva uma função que retorna o token de ordem `n` de um string, se ele existir; caso contrário, a função deve retornar **NULL**.

EP9.2 Escreva um programa que verifica se um número de CPF introduzido pelo usuário é válido. Um número de CPF tem 11 dígitos divididos em duas partes: a parte principal com 9 dígitos e os dígitos verificadores, que são os dois últimos dígitos. A validação de um número de CPF segue o procedimento descrito abaixo:

- Verificação do primeiro dígito de controle (penúltimo dígito do número):
  - ◊ Multiplique os inteiros representados pelos dígitos da parte principal, do primeiro ao último, respectivamente, por `10`, `9`, `8`, ..., `2` e some os resultados obtidos.
  - ◊ Calcule o resto da divisão do resultado obtido no item anterior por `11`.

- ◊ Se o resto da divisão for `0` ou `1`, o primeiro dígito verificador deverá ser igual a `0`; caso contrário, esse dígito deverá ser igual a `11` menos o referido resto de divisão.

- Verificação do segundo dígito de controle (último dígito do número):
  - ◊ Multiplique os inteiros representados pelos dígitos da parte principal e pelo primeiro dígito de controle, do primeiro ao último, respectivamente, por `11`, `10`, `9`, ..., `2` e some os resultados obtidos.
  - ◊ Calcule o resto da divisão do resultado obtido no item anterior por `11`.
  - ◊ Se o resto da divisão for `0` ou `1`, o segundo dígito verificador deverá ser igual a `0`; caso contrário, esse dígito deverá ser igual a `11` menos o último resto de divisão.

A principal diferença entre essas duas verificações é que a segunda inclui o primeiro dígito de controle.

EP9.3 (a) Escreva uma função, denominada `OcorrenciasStr()`, que conta o número de ocorrências de um string (primeiro parâmetro) em outro string (segundo parâmetro). (b) Escreva um programa para testar a função solicitada no item (a).

EP9.4 Escreva uma função que remove de um string todas as ocorrências de um dado caractere. O protótipo dessa função deve ser:

```
char *RemoveCaractere(char *str, int remover)
```

Nesse protótipo, `str` é o string que será eventualmente modificado, `remover` é o caractere a ser removido e o retorno da função deve ser o endereço inicial do string.

EP9.5 (a) Implemente uma função, cujo protótipo é:

```
char *RemoveCaracteres(char *str, const char *aRemover)
```

que remove do primeiro string recebido como parâmetro todos os caracteres presentes no segundo parâmetro, que também é um string. O retorno dessa função deve ser o endereço do string eventualmente alterado. (b) Escreva uma função **main()** que recebe dois strings como argumentos de linha de comando e remove do primeiro string todos os caracteres presentes no segundo string. [**Sugestão:** Use a função `RemoveCaractere()` solicitada no exercício **EP9.4**.]

EP9.6 Escreva uma função que remove todas as ocorrências de um dado string em outro string e retorna o número de remoções efetuadas.

EP9.7 Escreva uma função que substitui todas as ocorrências de um substring num string por outro substring.

EP9.8 Escreva uma função, denominada `IntEmString()`, que converte um valor do tipo **int** em string.

EP9.9 Escreva uma função que implemente o algoritmo **KMP** e que retorne o número de ocorrências do padrão no texto.

EP9.10 Escreva uma função que implemente o algoritmo **KR** e retorne o endereço de uma lista simplesmente encadeada contendo em cada nó da lista a posição de uma ocorrência do padrão no texto. Se não houver nenhuma ocorrência do padrão no texto, a função deve retornar **NULL**.

EP9.11 Escreva uma função que efetua um caminhamento sobre uma trie do tipo definido na **Seção 9.8.2** e exibe na tela todos os strings armazenados na trie em ordem alfabética.

EP9.12 Escreva uma função que resolve o problema de casamento de padrão com texto em fluxo contínuo discutido na **Seção 9.10.7** utilizando o algoritmo **KR**.

EP9.13 Escreva uma função, denominada `DestroiTrie()`, que libera o espaço ocupado por uma trie.

EP9.14 Escreva a função `NumeroDeChaves()` que calcula e retorna o número de chaves de uma trie.

EP9.15 Na **Seção 9.8.4**, foi descrita uma implementação alternativa de tries que, em vez de usar um array de ponteiros como na implementação discutida em detalhes no texto, cada nó, com exceção da raiz da trie, contém apenas dois ponteiros: o primeiro deles aponta para o primeiro filho do nó e o segundo ponteiro aponta para uma lista encadeada que armazena os irmãos desse nó. Implemente funções que executem as operações de busca, inserção e remoção em tries implementadas desse modo.

EP9.16 Quando a primeira letra não é comum ao conjunto de strings armazenados numa trie a função `MPC()` apresentada na **Seção 9.10.3** retorna **NULL**, de modo que numa aplicação real provavelmente ela sempre retornará esse valor, pois raramente se terá uma trie em que todas as chaves começam pela mesma letra. Escreva uma função denominada `MPC2()` semelhante à função `MPC()` que recebe um parâmetro adicional que representa a letra inicial dos prefixos a ser procurados.

EP9.17 Altere o programa apresentado na **Seção 9.10.6** de modo que, ao final, ele apresente o número de casamentos falsos encontrados.

EP9.18 Altere o programa apresentado na **Seção 9.10.6** de modo que os padrões não tenham o mesmo tamanho.