ACH2024

Aulas 13 e 14

String matching com e sem Tries

Profa. Ariane Machado Lima

Strings

- Sequência de símbolos sobre um alfabeto Σ de tamanho d
- Usada para representar muitas coisas (cada caso com um alfabeto específico):
  - Imagens
  - Sequências biológicas (de nucleotídeos ou aminoácidos)
  - Textos em linguagem natural (ex: páginas web, texto literário, programas, ...)

String Pattern Matching

É comum querermos localizar um certo padrão em um “texto”

→ identificar um substring P (padrão) em um string T (texto)

Problema geral: dado um texto T de tamanho n e um padrão P de tamanho m ($n \geq m$), verificar se P é uma substring de T

Veremos 3 algoritmos para resolver esse problema:

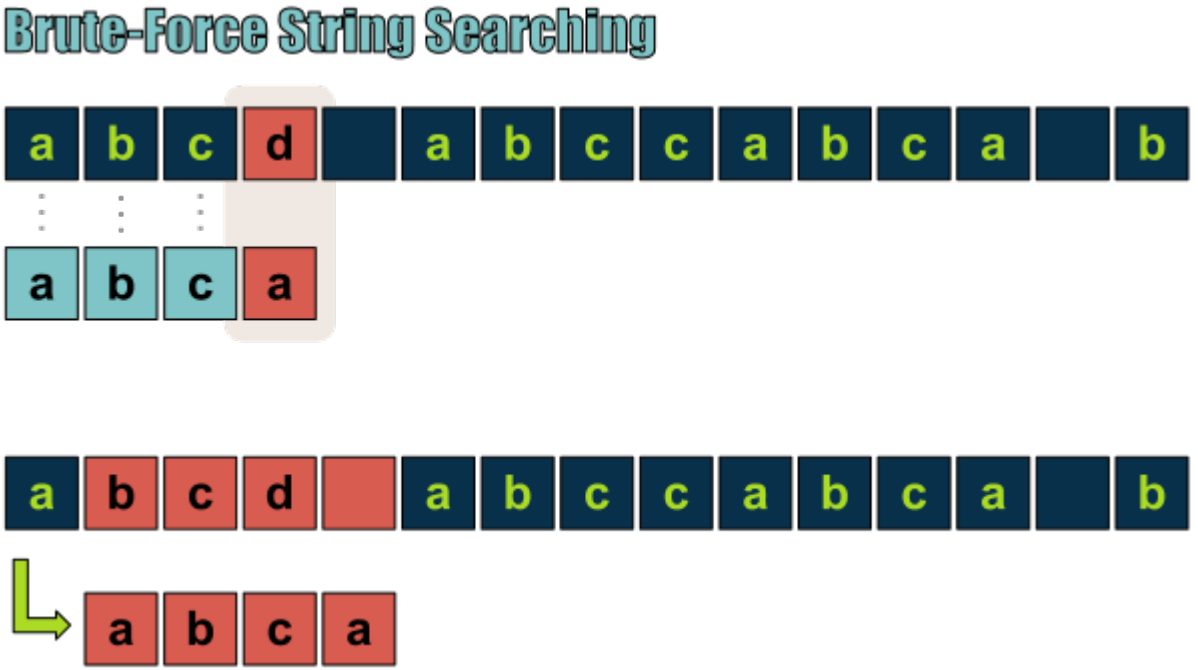
- Força Bruta
- Boyer-Moore
- Knuth-Morris-Pratt (KMP) – a ser implementado como parte do EP 2

Algoritmo Força Bruta

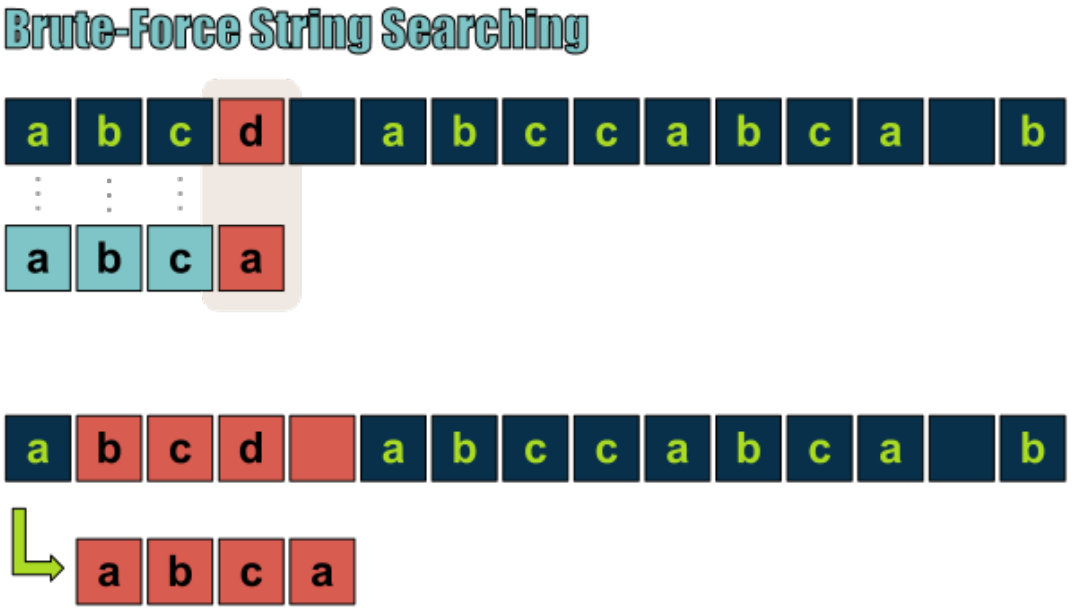
- Como seria um algoritmo “força bruta” ?
- Exemplo: procurar no texto T abaixo o padrão P abaixo:
  T = abcd abccabca b
  P = abca

Algoritmo Força Bruta

- Compara o padrão P com o texto T para cada possível deslocamento de P relativo a T, até que:
  - Um match seja encontrado (de todos os caracteres de P)
  - Todos os substrings de T de tamanho P tenham sido comparados a P



Algoritmo Força Bruta

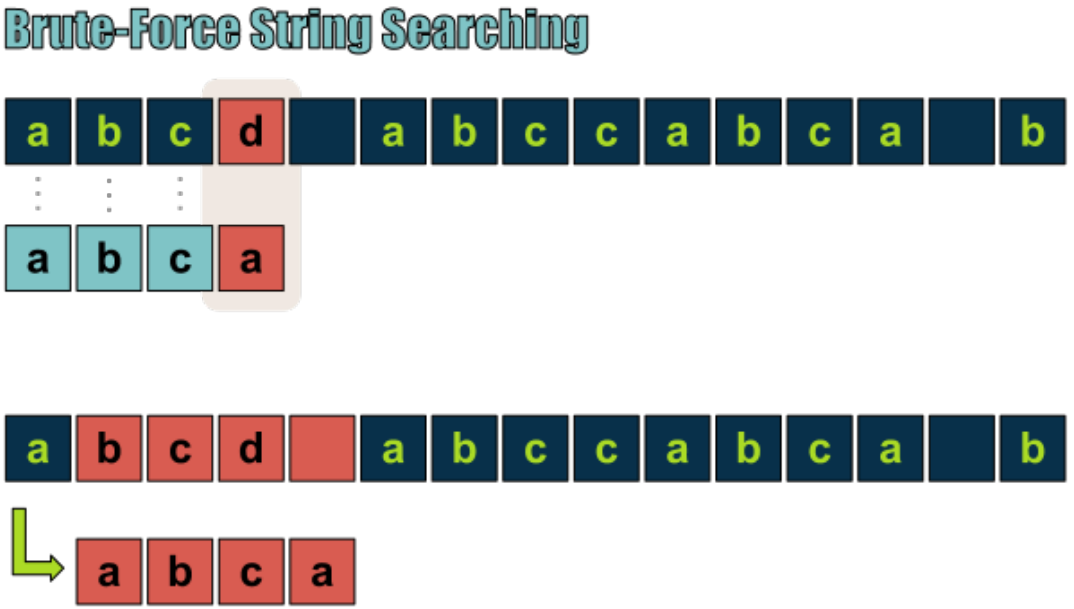


Algorithm *BruteForceMatch*(*T*, *P*)

Input text *T* of size *n* and pattern *P* of size *m*

Output starting index of a substring of *T* equal to *P* or −1 if no such substring exists

Algoritmo Força Bruta




```
Algorithm BruteForceMatch(T, P)
    Input text T of size n and pattern
        P of size m
    Output starting index of a
        substring of T equal to P or −1
        if no such substring exists
    for i ← 0 to n − m
        { test shift i of the pattern }
        j ← 0
        while j < m ∧ T[i + j] = P[j]
            j ← j + 1
        if j = m
            return i {match at i}
    return -1 {no match anywhere}
```

Algoritmo Força Bruta

- Compara o padrão P com o texto T para cada possível deslocamento de P relativo a T, até que:
  - Um match seja encontrado (de todos os caracteres de P)
  - Todos os substrings de T de tamanho P tenham sido comparados a P
- Tempo: ?

Algoritmo Força Bruta

- Compara o padrão P com o texto T para cada possível deslocamento de P relativo a T, até que:
  - Um match seja encontrado (de todos os caracteres de P)
  - Todos os substrings de T de tamanho P tenham sido comparados a P
- Tempo: $O(nm)$

Algoritmo Força Bruta

- Compara o padrão P com o texto T para cada possível deslocamento de P relativo a T, até que:
  - Um match seja encontrado (de todos os caracteres de P)
  - Todos os substrings de T de tamanho P tenham sido comparados a P
- Tempo: $O(nm)$
- Pior caso:

Algoritmo Força Bruta

- Compara o padrão P com o texto T para cada possível deslocamento de P relativo a T, até que:
  - Um match seja encontrado (de todos os caracteres de P)
  - Todos os substrings de T de tamanho P tenham sido comparados a P
- Tempo: $O(nm)$

- Pior caso:
- Exemplo de pior caso quando há um match:

Algoritmo Força Bruta

- Compara o padrão P com o texto T para cada possível deslocamento de P relativo a T, até que:
  - Um match seja encontrado (de todos os caracteres de P)
  - Todos os substrings de T de tamanho P tenham sido comparados a P
- Tempo: $O(nm)$

- Pior caso:
- Exemplo de pior caso quando há um match:

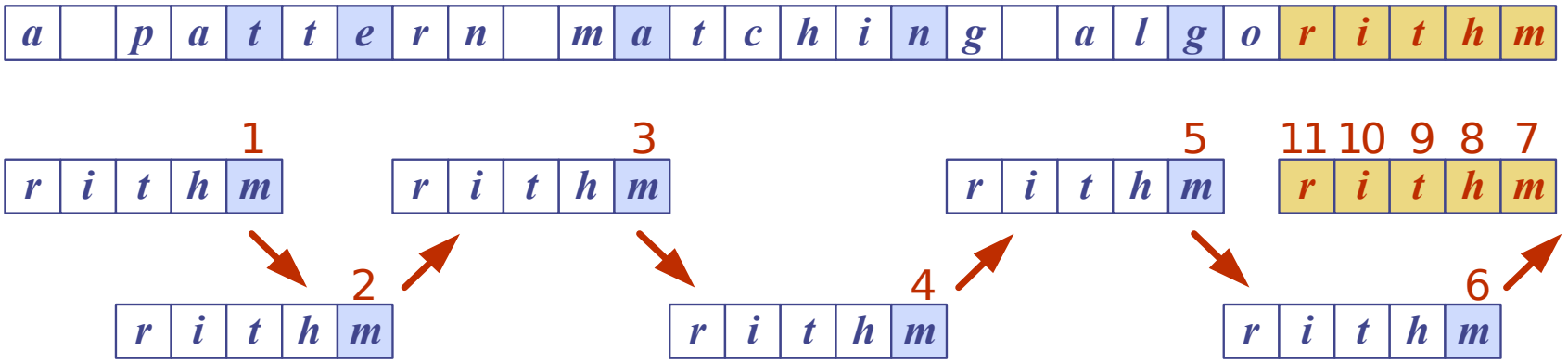
$T = aaa \ldots ah$

$P = aaah$

Pode ocorrer em imagens, sequências biológicas, mas dificilmente em linguagens naturais

Algoritmo de Boyer-Moore

- Baseado em duas heurísticas:
  - Espelho: Comparar P com uma subsequência de T de trás para frente
  - Salto de caracteres: Quando ocorre um mismatch em $T[i] = c$, há 3 situações possíveis:
    - ***CASO 1:*** ***c*** não ocorre em **P**, logo não faz sentido fazer outras comparações de ***T[i]*** com qualquer outro caracter de **P**. Logo, **P** deve ser deslocado para alinhar **P[0]** com **T[i+1]**
    - ***CASO 2:*** ***c*** ocorre em **P**, então desloque ***P*** para alinhar a última ocorrência de ***c*** em ***P*** com ***T[i]*** (a não ser que seja a próxima situação)
    - ***CASO 3:*** ***c*** ocorre em **P** mas a última ocorrência de ***c*** em ***P*** já foi comparada na comparação corrente (ie, está em um índice l > j no qual ocorreu o mismatch T[i] ≠ P[j], ex: T = aaaa, P = baa), então desloque só uma posição



Função Última-Ocorrência

- Necessidade de pré-processar o padrão P , com base no alfabeto Σ , para construir a função L de "última-ocorrência" mapeando Σ inteiros, na qual $L(c)$ é definido como:

  o maior índice i tal que $P[i] = c$ ou

  -1 se não existe tal índice

- Example:

  $\Sigma = \{a, b, c, d\}$

  $P = abacab$

c	a	b	c	d
$L(c)$	4	5	3	-1

- A função de última-ocorrência pode ser representada por um array indexado pelos "códigos" dos caracteres.
- Pode ser computada em tempo $O(m + d)$, no qual m é o comprimento de P e d é o tamanho de Σ

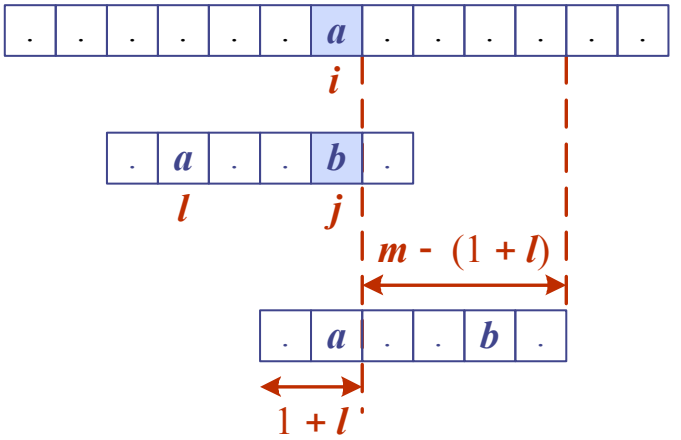
EXERCÍCIO: IMPLEMENTÁ-LA!

Algoritmo de Boyer-Moore

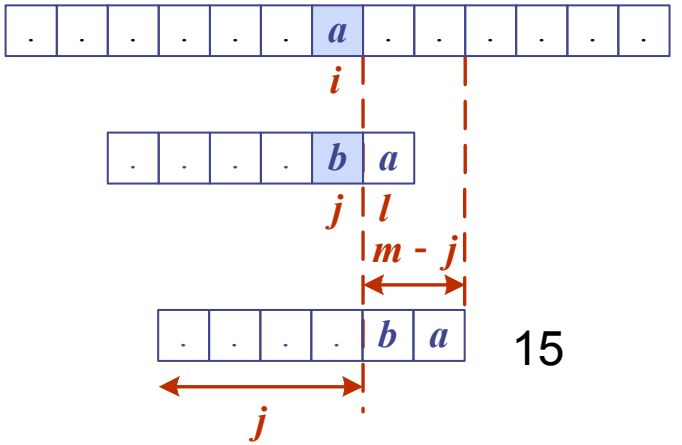
```
Algorithm BoyerMooreMatch(T, P, Σ)
    L ← lastOccurenceFunction(P, Σ )
    i ← m - 1
    j ← m - 1
    repeat
    if T[i] = P[j]
        if j = 0
            return i { match at i }
        else
            i ← i - 1
            j ← j - 1
    else
        { character-jump }
        l ← L[T[i]]
        i ← i + m – min(j, 1 + l)
        j ← m - 1
  until i > n - 1
  return -1 { no match }
```

Caso 1: $l = -1$ Logo 1+l = 0 é o mínimo

Caso 2: $l < j$ Logo 1+l é o mínimo, pelo menos igual

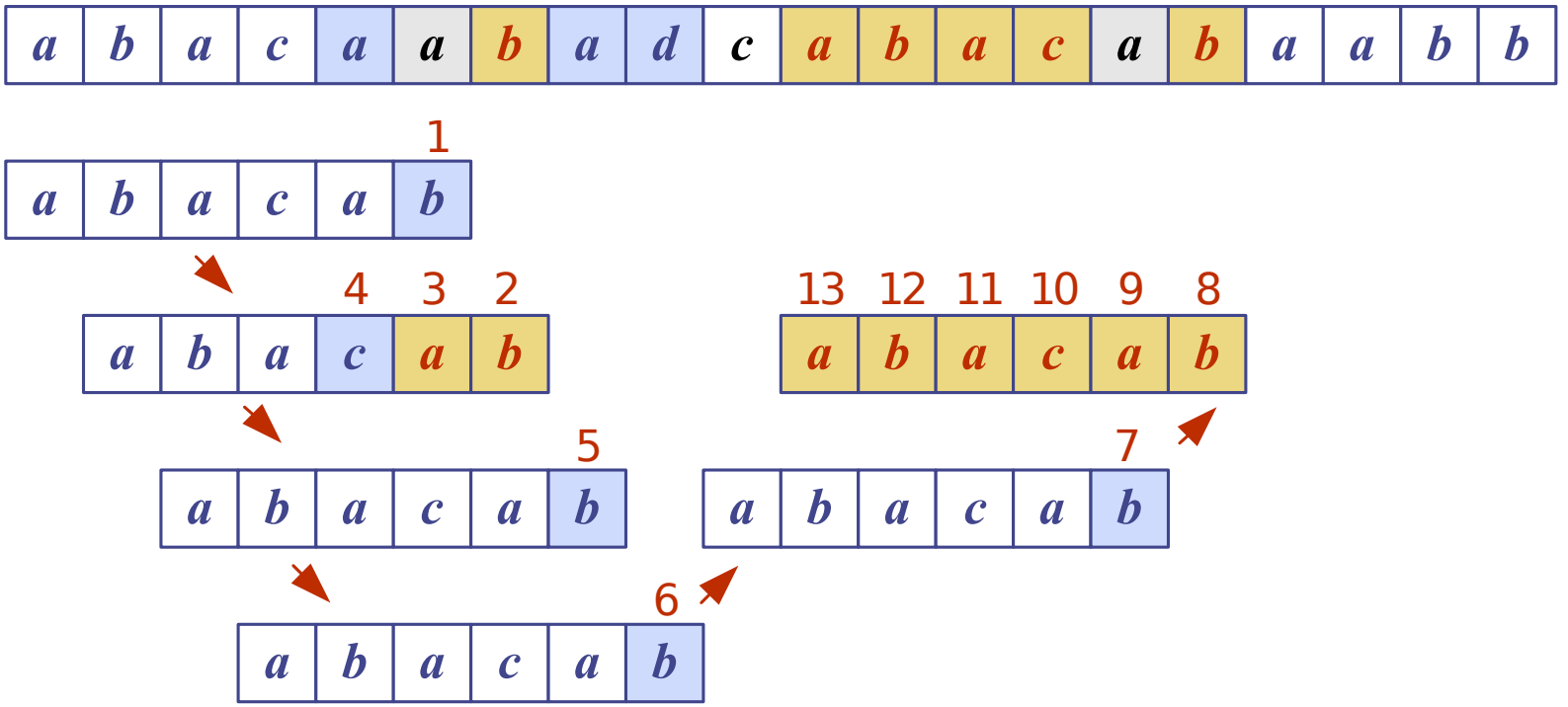


Caso 3: $j < l$ Logo j é o mínimo



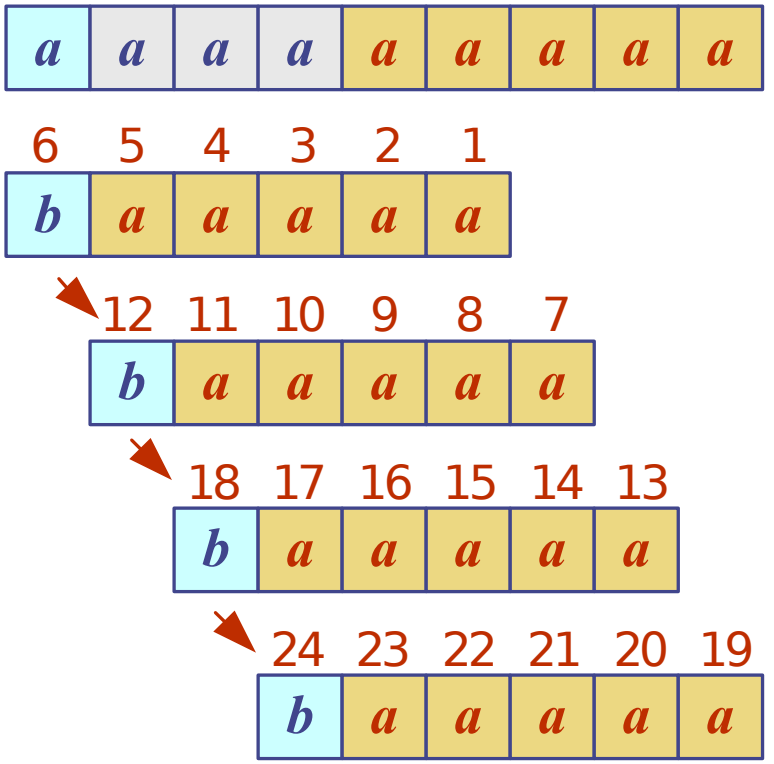
Exemplo

(números em vermelho indicam ordem das comparações realizadas)



Complexidade do Alg. Boyer-Moore

- Qual o pior caso para esse algoritmo?



Complexidade do Alg. Boyer-Moore

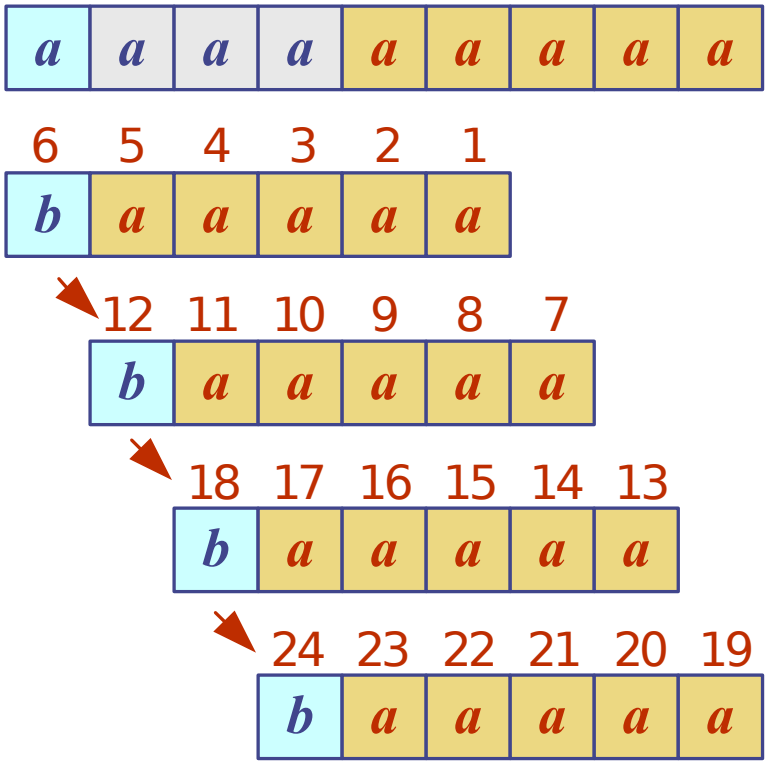
- Qual o pior caso para esse algoritmo?

Exemplo do pior caso:

$T = aaa \ldots a$

$P = baaa$

- Portanto seu tempo de execução é



Complexidade do Alg. Boyer-Moore

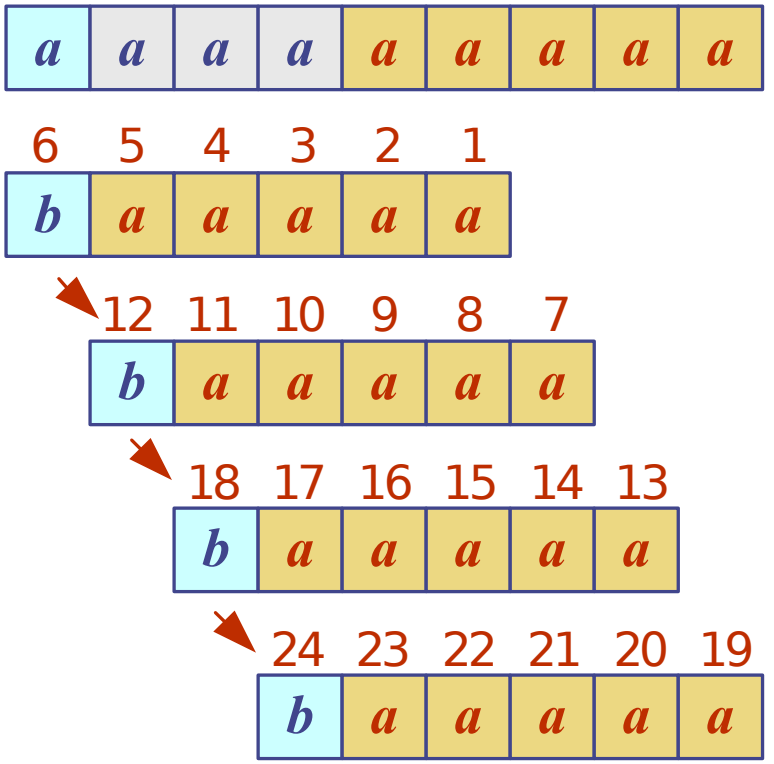
- Qual o pior caso para esse algoritmo?

Exemplo do pior caso:

$T = aaa \ldots a$

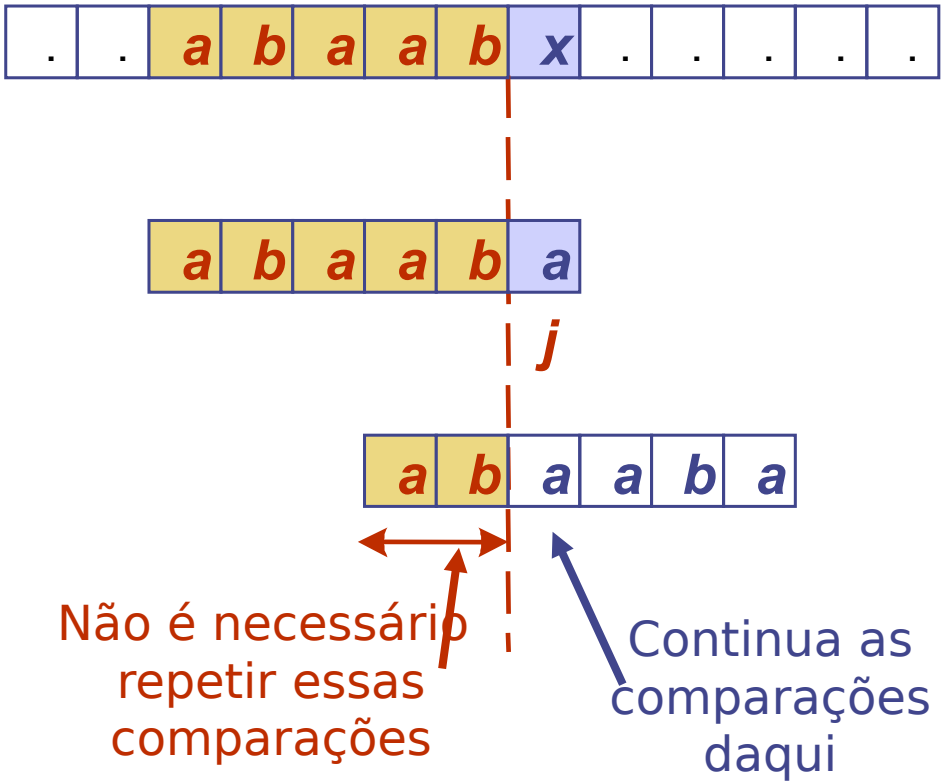
$P = baaa$

- Portanto seu tempo de execução é $O(nm + d)$ (O(m+d) do cálculo da função de última ocorrência + O(nm)para o matching propriamente dito)
- O pior caso pode acontecer em imagens e sequências biológicas, mas é improvável em linguagens naturais
- Na prática significativamente mais rápido que o algoritmo de força bruta em textos de linguagens naturais



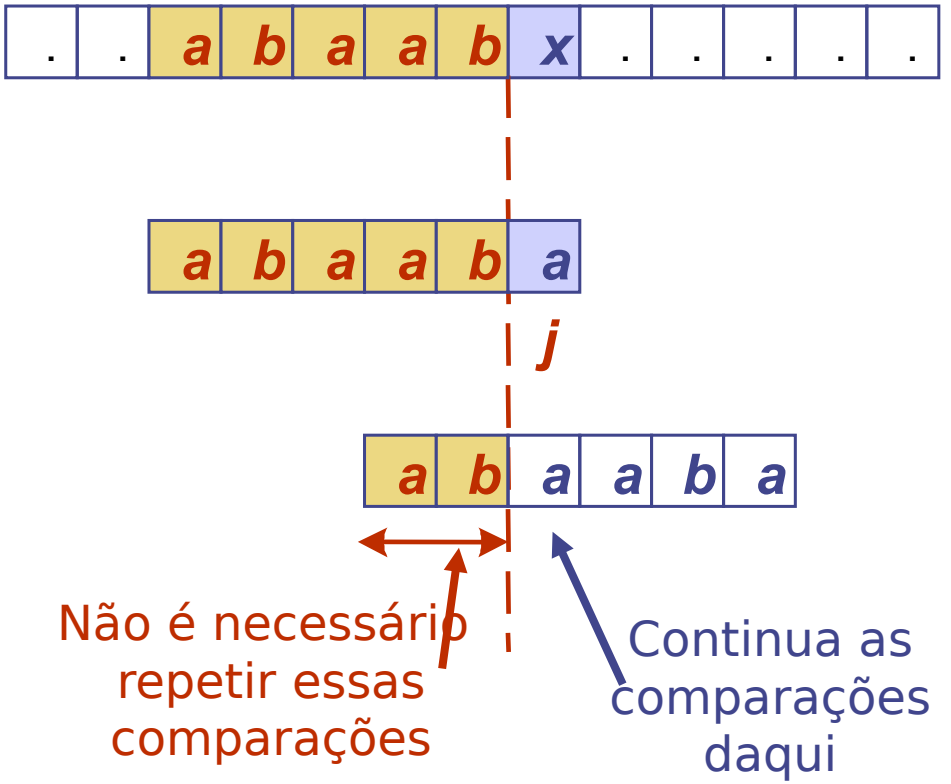
Algoritmo Knuth-Morris-Pratt (KMP)

- Compara o padrão ao texto da esquerda para a direita, mas desloca o padrão de forma mais inteligente que o algoritmo de força bruta
- Quando ocorre um mismatch, qual é o máximo de deslocamento (a partir do início de P) que podemos fazer evitando comparações redundantes?
- Resposta:



Algoritmo Knuth-Morris-Pratt (KMP)

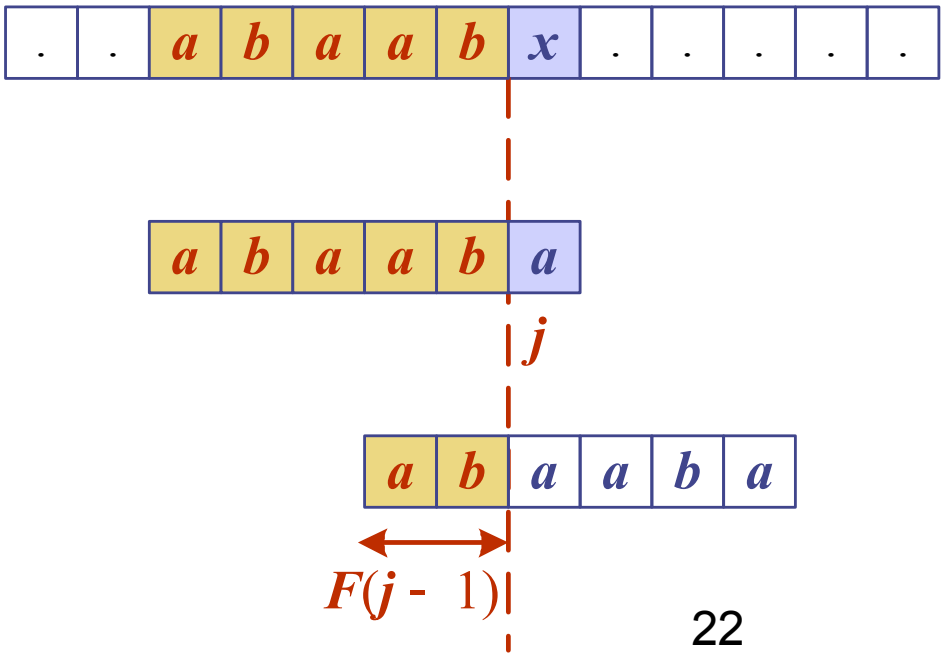
- Compara o padrão ao texto da esquerda para a direita, mas desloca o padrão de forma mais inteligente que o algoritmo de força bruta
- Quando ocorre um mismatch, qual é o máximo de deslocamento (a partir do início de P) que podemos fazer evitando comparações redundantes?
- Resposta: O comprimento do maior prefixo de $P[0..j-1]$ que seja um sufixo de $P[1..j-1]$



KMP - Função de Falha

- Pré-processamento do padrão para encontrar matches de prefixos do padrão com o próprio padrão
- A **função de falha** $F(j)$ é definida como o comprimento do maior prefixo de $P[0..j]$ que seja também um sufixo de $P[1..j]$
- Algoritmo KMP modifica o algoritmo de força-bruta de tal forma que, se um mismatch ocorre em $P[j] \neq T[i]$ faz-se $j \leftarrow F(j-1)$

j	0	1	2	3	4	5
$P[j]$	a	b	a	a	b	a
$F(j)$	0	0	1	1	2	3



O Algoritmo KMP

```
Algorithm KMPMatch(T, P)
    F ← failureFunction(P)
    i ← 0
    j ← 0
    while i < n
    if T[i] = P[j]
        if j = m - 1
            return i - j { match }
        else
            i ← i + 1
            j ← j + 1
    else
        if j > 0
            j ← F[j - 1]
        else
            i ← i + 1
  return -1 { no match }
```

O Algoritmo KMP

```
Algorithm KMPMatch(T, P)
    F ← failureFunction(P)
    i ← 0
    j ← 0
    while i < n
    if T[i] = P[j]
        if j = m - 1
            return i - j { match }
        else
            i ← i + 1
            j ← j + 1
    else
        if j > 0
            j ← F[j - 1]
        else
            i ← i + 1
    return -1 { no match }
```

- A função de falha pode ser representada por um vetor e pode ser computada em tempo $O(m)$

- Em cada iteração do while,

Ou a distância $k = i - j$ é incrementada em pelo menos 1 (observe que $F(j - 1) < j$)

Ou i sofre incremento de 1, sem aumentar k

- Portanto, não há mais de $2n$ iterações do while (no máximo n incrementos do i e n incrementos de k)

- Portanto, KMP roda em tempo

$O(m + n)$

O Algoritmo KMP

```
Algorithm KMPMatch(T, P)
    F ← failureFunction(P)
    i ← 0
    j ← 0
    while i < n
    if T[i] = P[j]
        if j = m - 1
            return i - j { match }
        else
            i ← i + 1
            j ← j + 1
    else
        if j > 0
            j ← F[j - 1]
        else
            i ← i + 1
  return -1 { no match }
```

- A função de falha pode ser representada por um vetor e pode ser computada em tempo $O(m)$

- Em cada iteração do while,

Ou a distância $k = i - j$ é incrementada em pelo menos 1 (observe que $F(j - 1) < j$)

Ou i sofre incremento de 1, sem aumentar k

- Portanto, não há mais de $2n$ iterações do while (no máximo n incrementos do i e n incrementos de k)

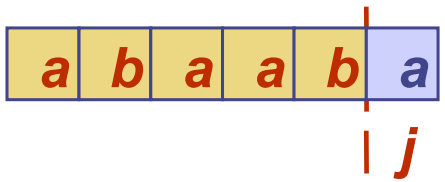
- Portanto, KMP roda em tempo

$O(m + n)$

Computando a Função de Falha

```
Algorithm failureFunction(P)
    F[0] ← 0
    i ← 1  { computando F[i], i = fim do sufixo }
    j ← 0  { posição (do prefixo) sendo comparada }
    while i < m
    if P[i] = P[j]
        {we have matched j + 1 chars}
        F[i] ← j + 1
        i ← i + 1
        j ← j + 1
    else if j > 0 then
        {use failure function to shift P}
        j ← F[j - 1]
    else
        F[i] ← 0 { no match }
        i ← i + 1
```

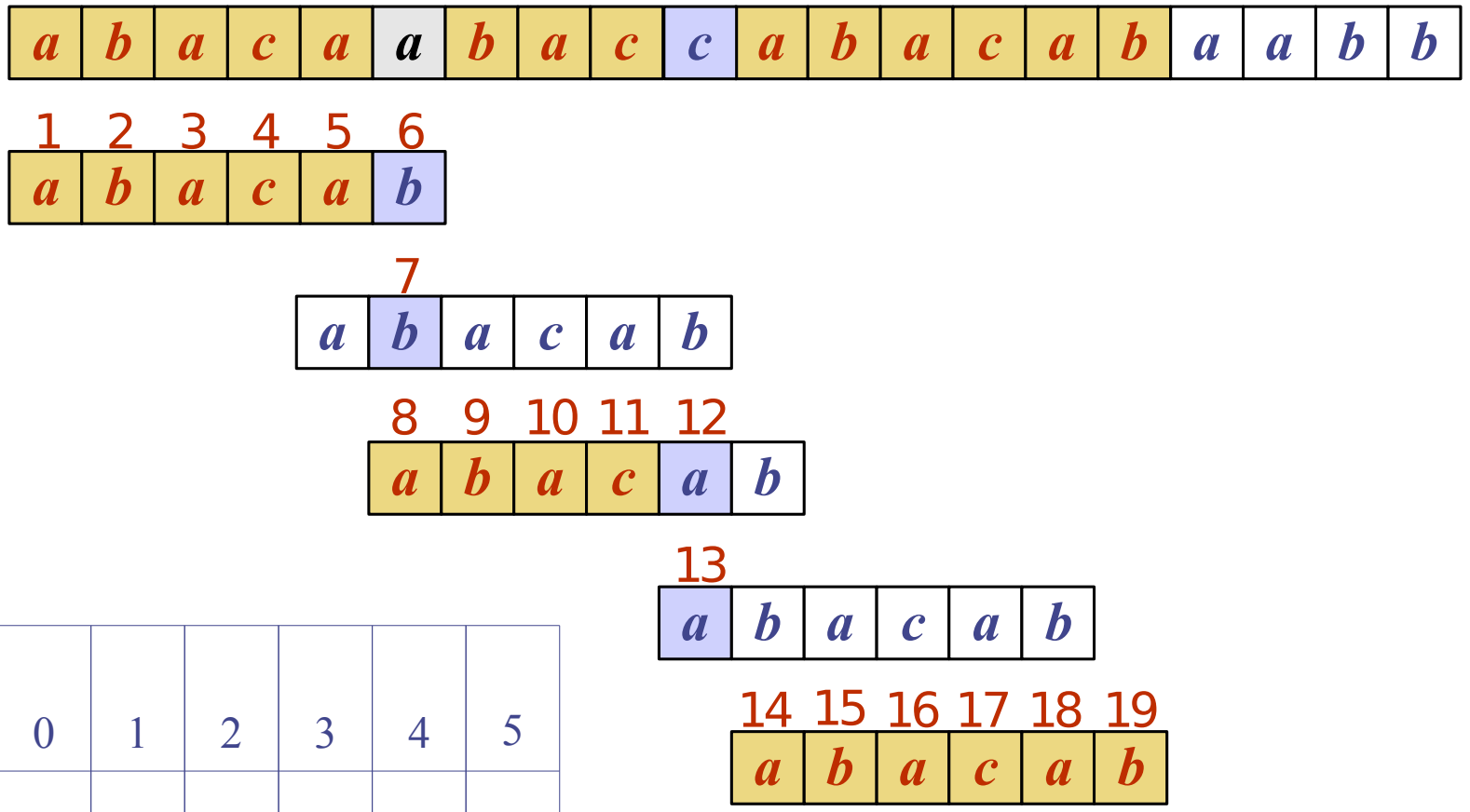
- Pode ser representada por um vetor e pode ser computada em tempo $O(m)$
- Aplicação do próprio algoritmo KMP
- Em cada iteração do while,
  - Ou i sofre incremento de 1,
  - Ou a distância $k = i - j$ é incrementada em 1 (observe que $F(j - 1) < j$)
- Portanto, não há mais de $2m$ iterações do while



j	0	1	2	3	4	5
$P[j]$	a	b	a	a	b	a
$F(j)$	0	0	1	1	2	3

KMP - Exemplo

(números em vermelho indicam ordem das comparações realizadas)



j	0	1	2	3	4	5
$P[j]$	*a*	*b*	*a*	*c*	*a*	*b*
$F(j)$	0	0	1	0	1	2

String Pattern Matching

usando TRIES

String Pattern Matching

- Imagine um website que permita que usuários façam buscas (string matching) na Bíblia (texto com aproximadamente 3,5 milhões de caracteres)
- Na maioria dos casos, o tamanho do padrão buscado é bem menor que o tamanho do texto (m << n)
- Mesmo o algoritmo KMP gastaria sempre um alto tempo para efetuar a busca….
- O que será que poderia ser feito?
  - Daria para fazer algum tipo de pré-processando do texto (tempo gasto antes de colocar o site no ar) de forma que, posteriormente, os usuários possam executar buscas mais rápidas?

String Pattern Matching

- Resposta: SIM!
- Quando há VÁRIAS buscas em um MESMO texto (potencialmente grande) é vantajoso investir tempo e espaço pré-processando o texto de forma a, posteriormente, conseguir realizar buscas mais rápidas.

- O texto será pré-processado e armazenado em uma estrutura de dados chamada TRIE, na qual as buscas serão feitas em tempo proporcional ao comprimento do padrão buscado.
- Tries são estruturas de dados compactas para representar um conjunto de strings, como todas as palavras de um texto

- Trie vem de reTRIEval

String Pattern Matching com TRIES

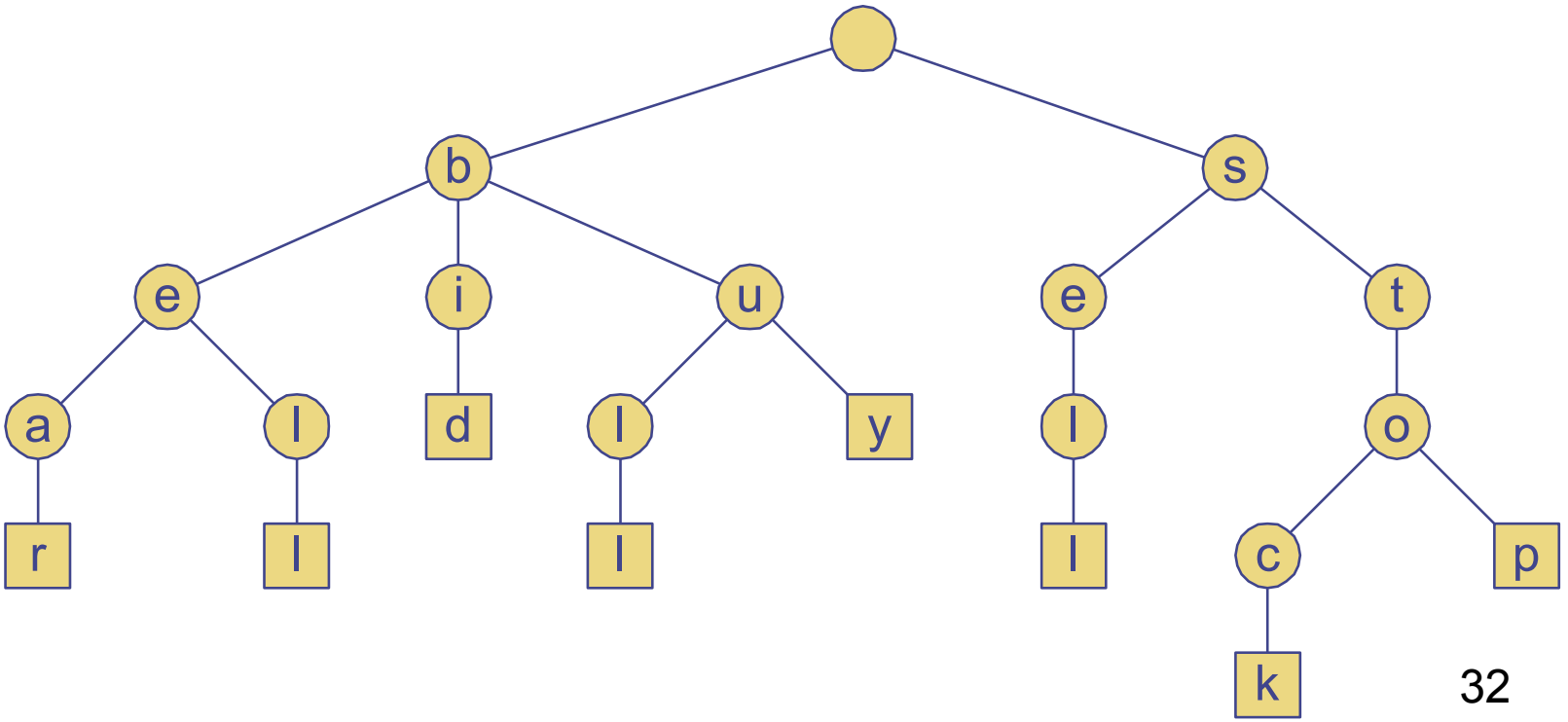
Para fins de simplificação (sem perda de generalidade), vamos considerar o caso em que o padrão é UMA palavra do texto

Trie Padrão

A trie padrão para um conjunto de strings S é uma árvore **ordenada** (ie, a ordem dos filhos importa) tal que:

- Cada nó (a menos da raiz) é rotulado com um caracter
- Os filhos de um nó são ordenados alfabeticamente da esquerda para a direita (no máximo d filhos)
- Cada caminho da raiz até uma folha armazena uma string de S

Exemplo: trie padrão para S = { bear, bell, bid, bull, buy, sell, stock, stop }

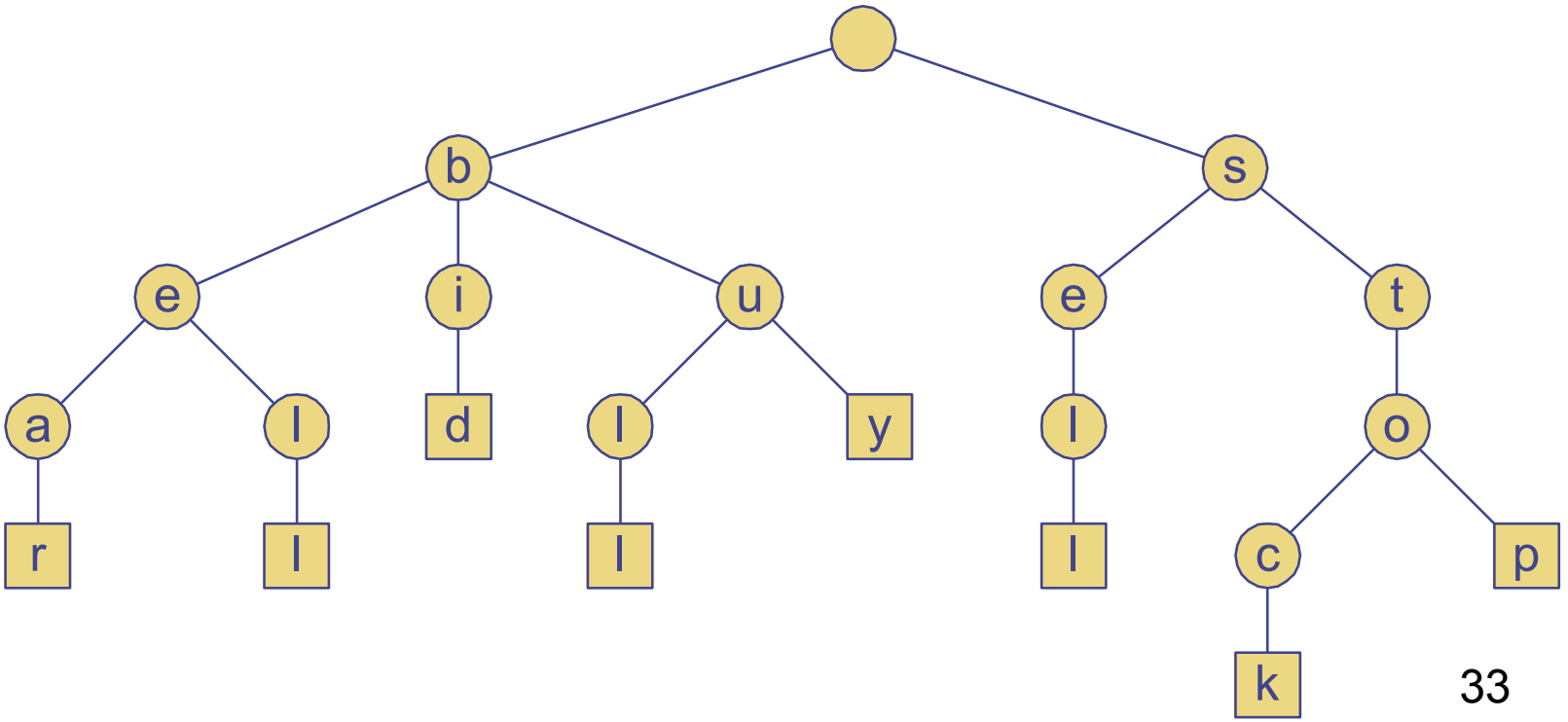


Trie, ou árvore de prefixos

A trie padrão para um conjunto de strings S é uma árvore **ordenada** (ie, a ordem dos filhos importa) tal que:

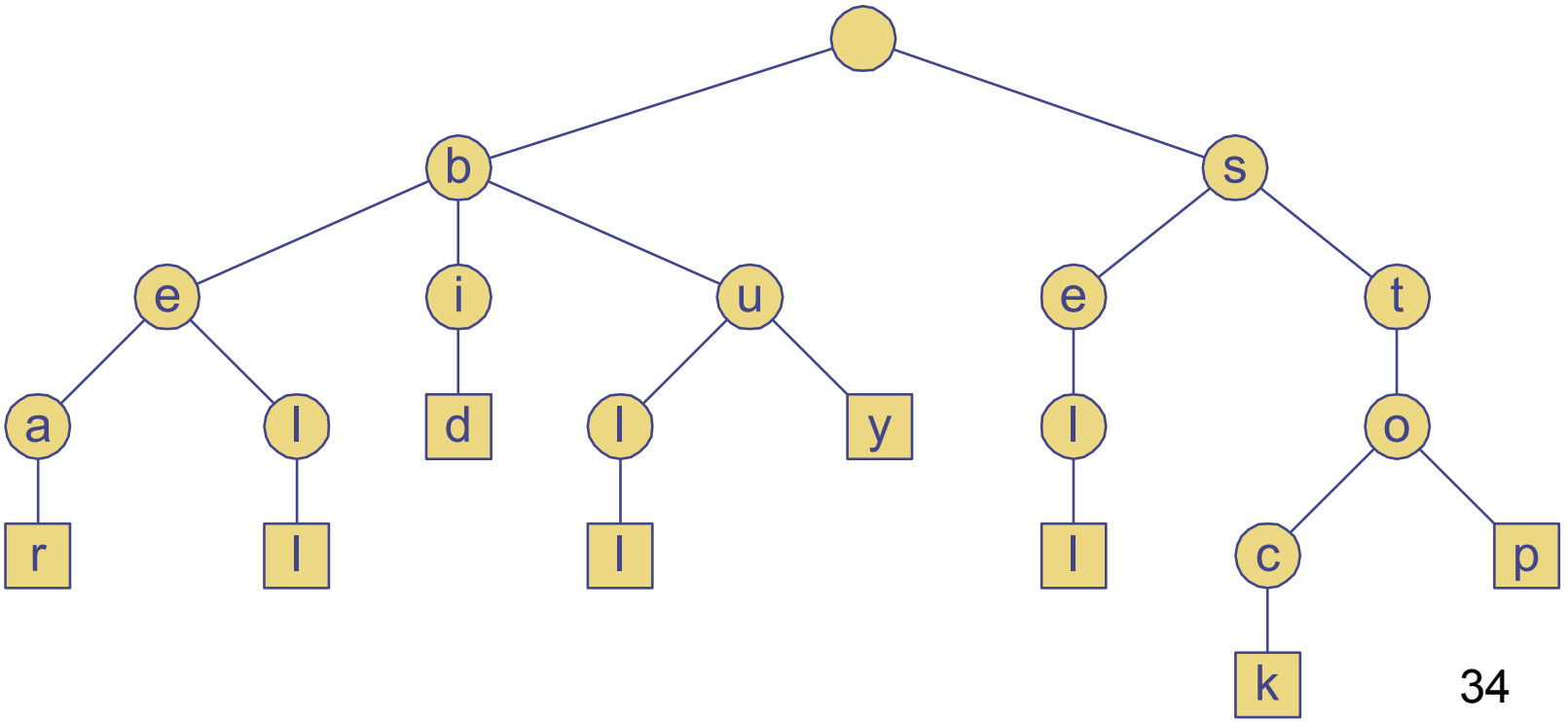
- Cada nó (a menos da raiz) é rotulado com um caracter
- Os filhos de um nó são ordenados alfabeticamente da esquerda para a direita (no máximo d filhos – d-way tries)
- Cada caminho da raiz até uma folha armazena uma string de S

Exemplo: trie padrão para S = { bear, bell, bid, bull, buy, sell, stock, stop }



Tries Padrão - Espaço

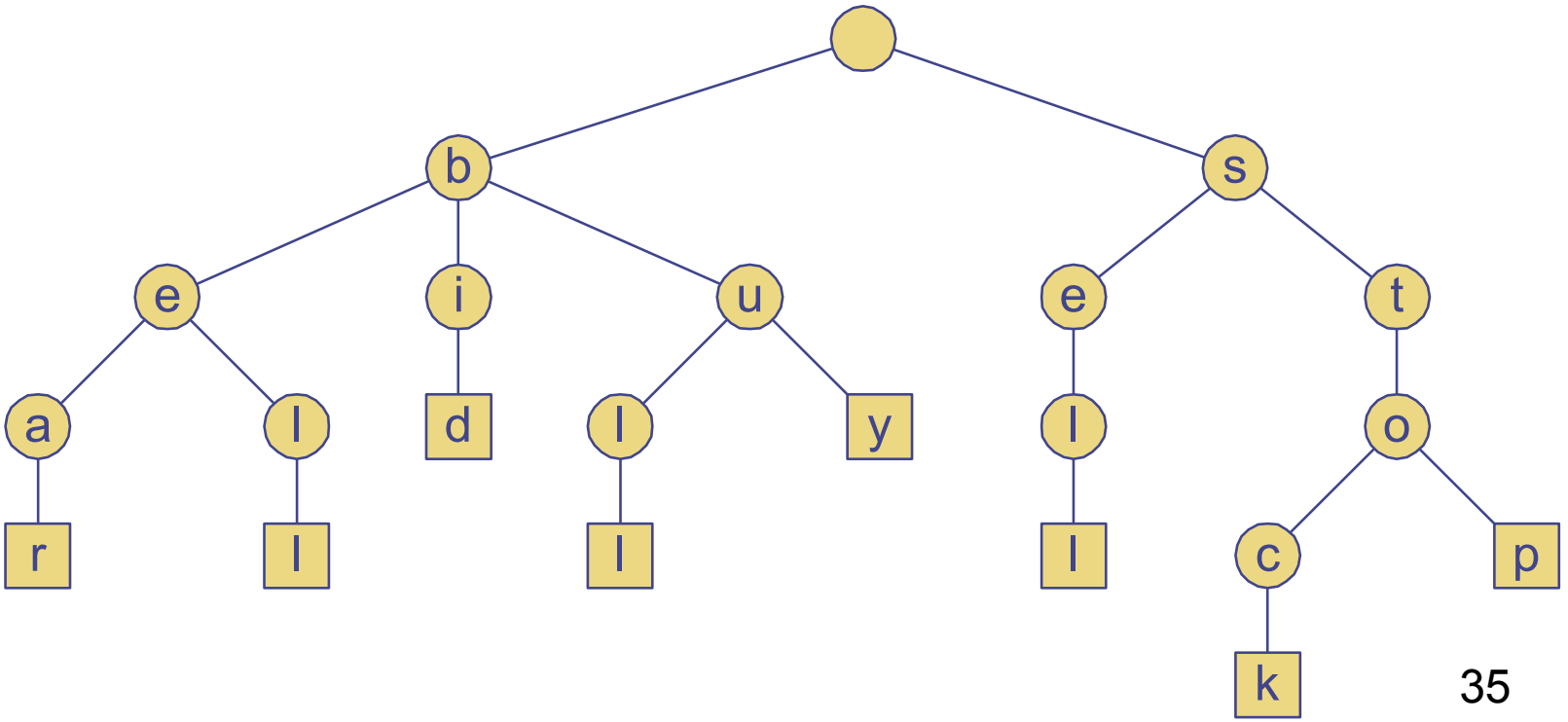
Quanto de espaço ocupa uma trie padrão?



Tries Padrão - Espaço

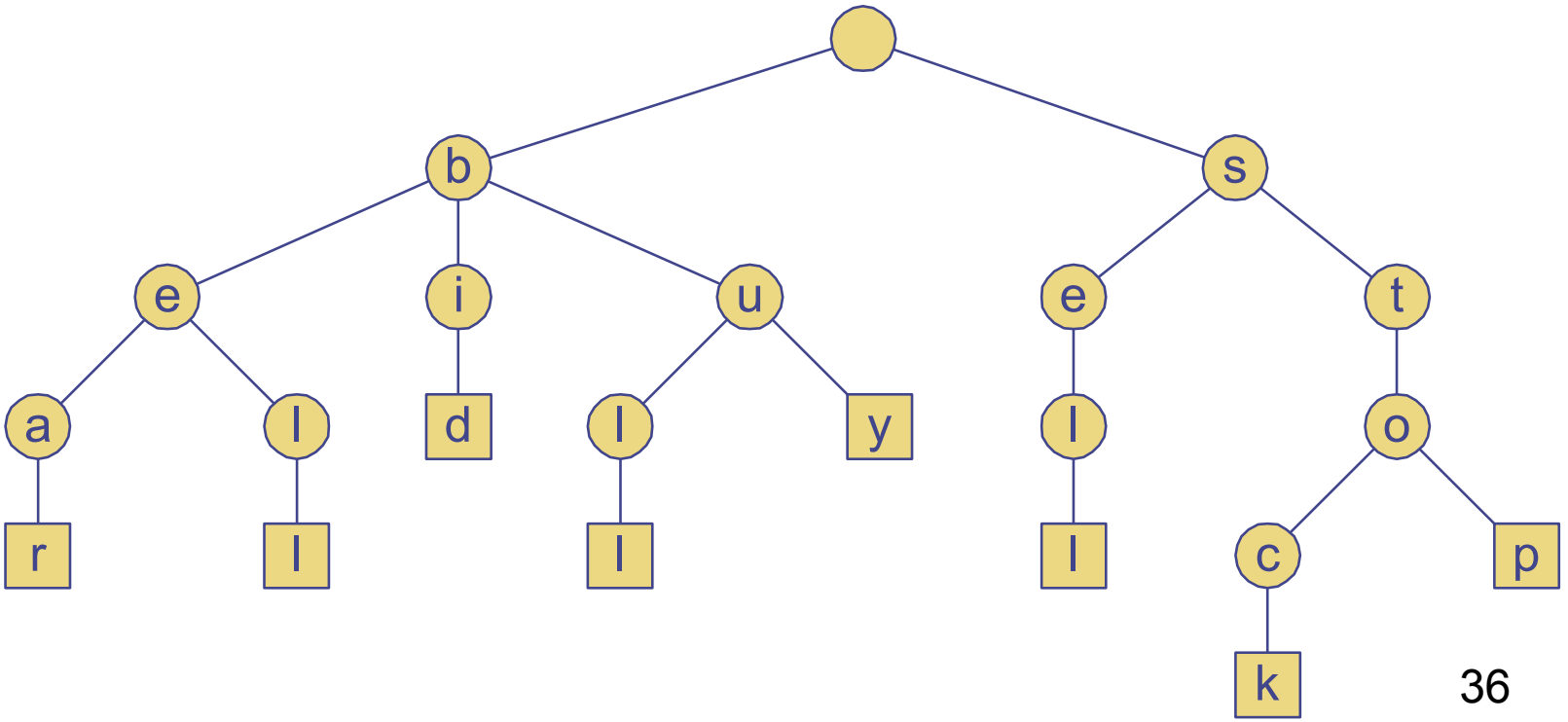
Quanto de espaço ocupa uma trie padrão? $O(n)$, sendo

- n = tamanho total dos strings em S (soma de todos os comprimentos)



Tries Padrão - Inserção

Dada uma nova string $s = s_0, s_1, \ldots, s_{m-1}$



Tries Padrão - Inserção

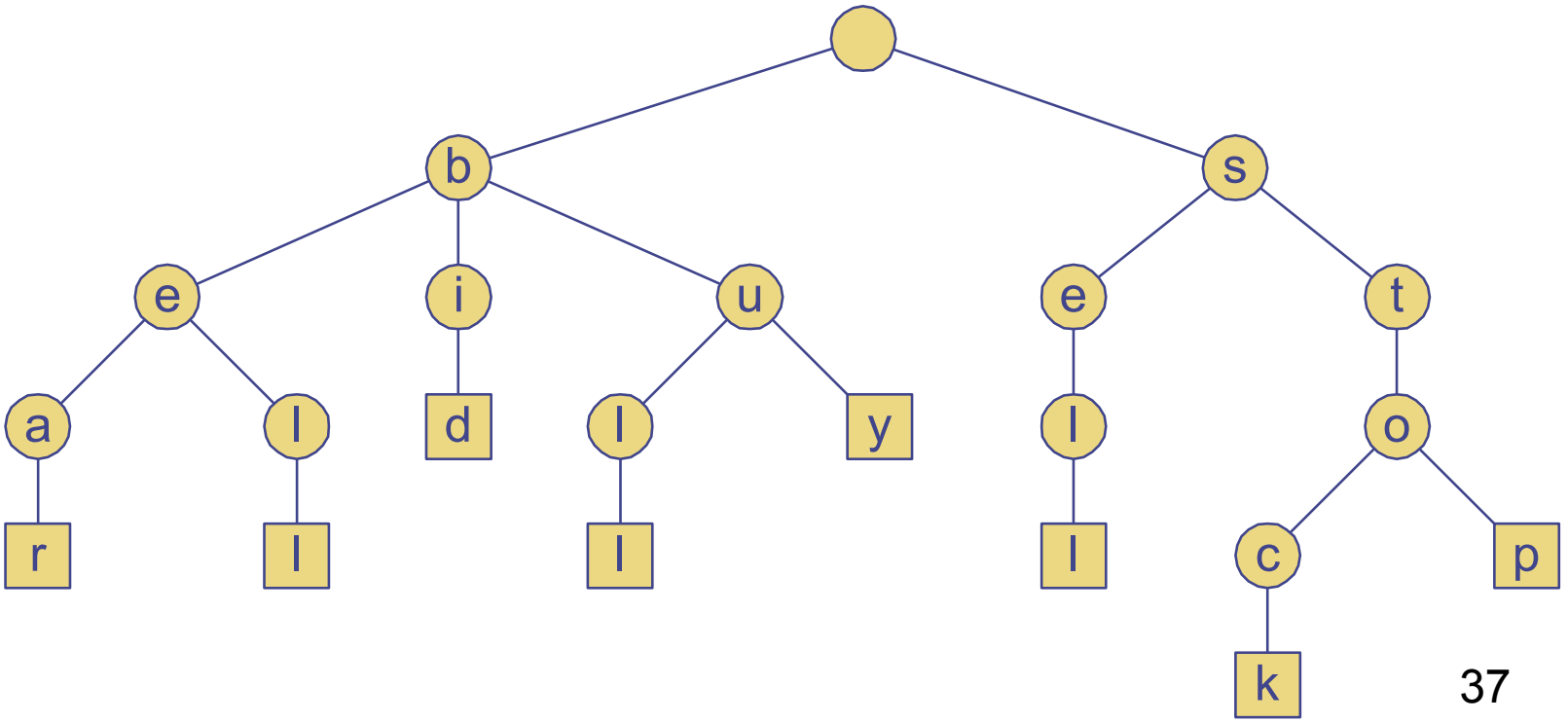
Dada uma nova string $s = s_0, s_1, \ldots, s_{m-1}$

x ← raiz

Para cada i = 0, …, m-1

Procura filho de $x = s_i$ ou se não existe crio um na posição adequada

x ← esse filho ou o novo nó



Tries Padrão - Inserção

Dada uma nova string $s = s_0, s_1, \ldots, s_{m-1}$

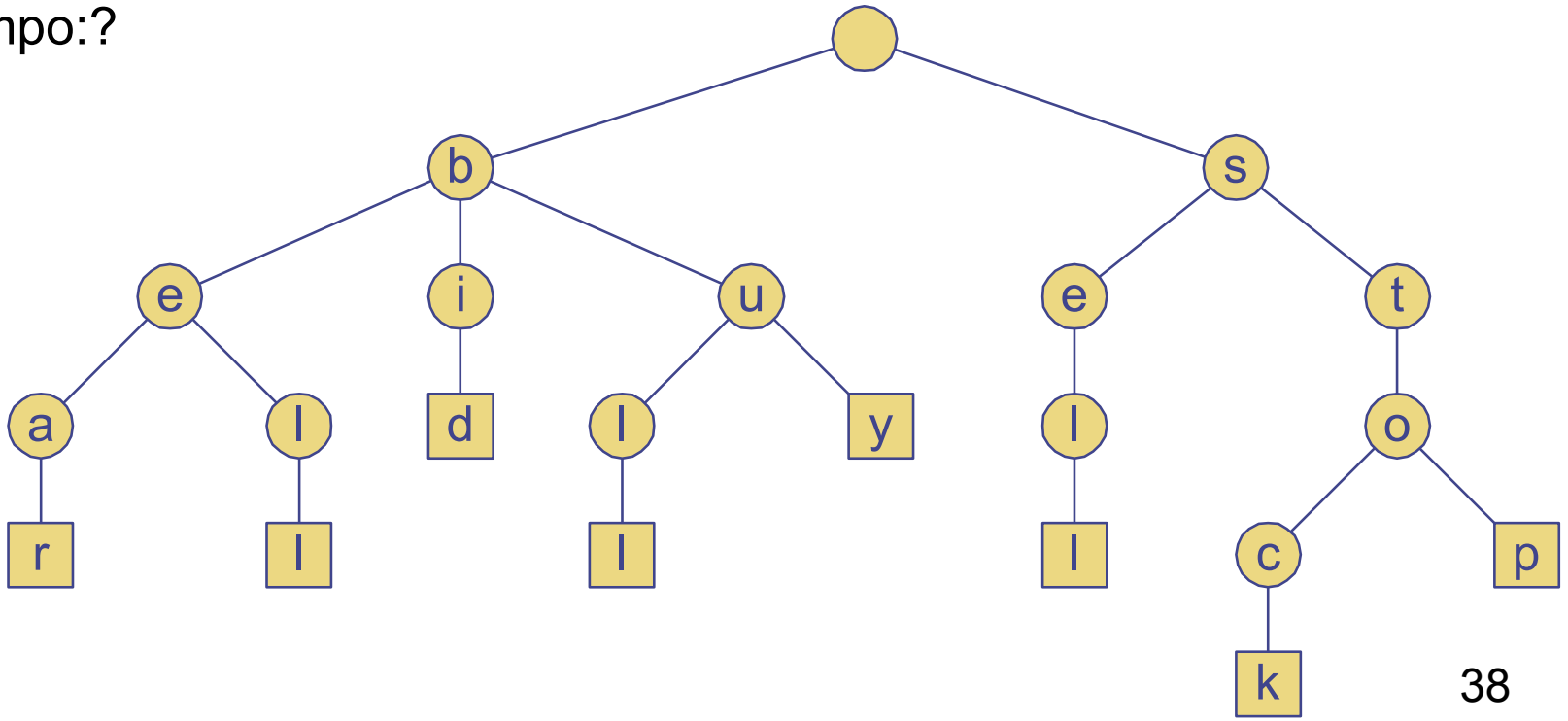
x ← raiz

Para cada i = 0, …, m-1

Procura filho de $x = s_i$ ou se não existe crio um na posição adequada

x ← esse filho ou o novo nó

Tempo:?



Tries Padrão - Inserção

Dada uma nova string $s = s_0, s_1, \ldots, s_{m-1}$

x ← raiz

Para cada i = 0, …, m-1

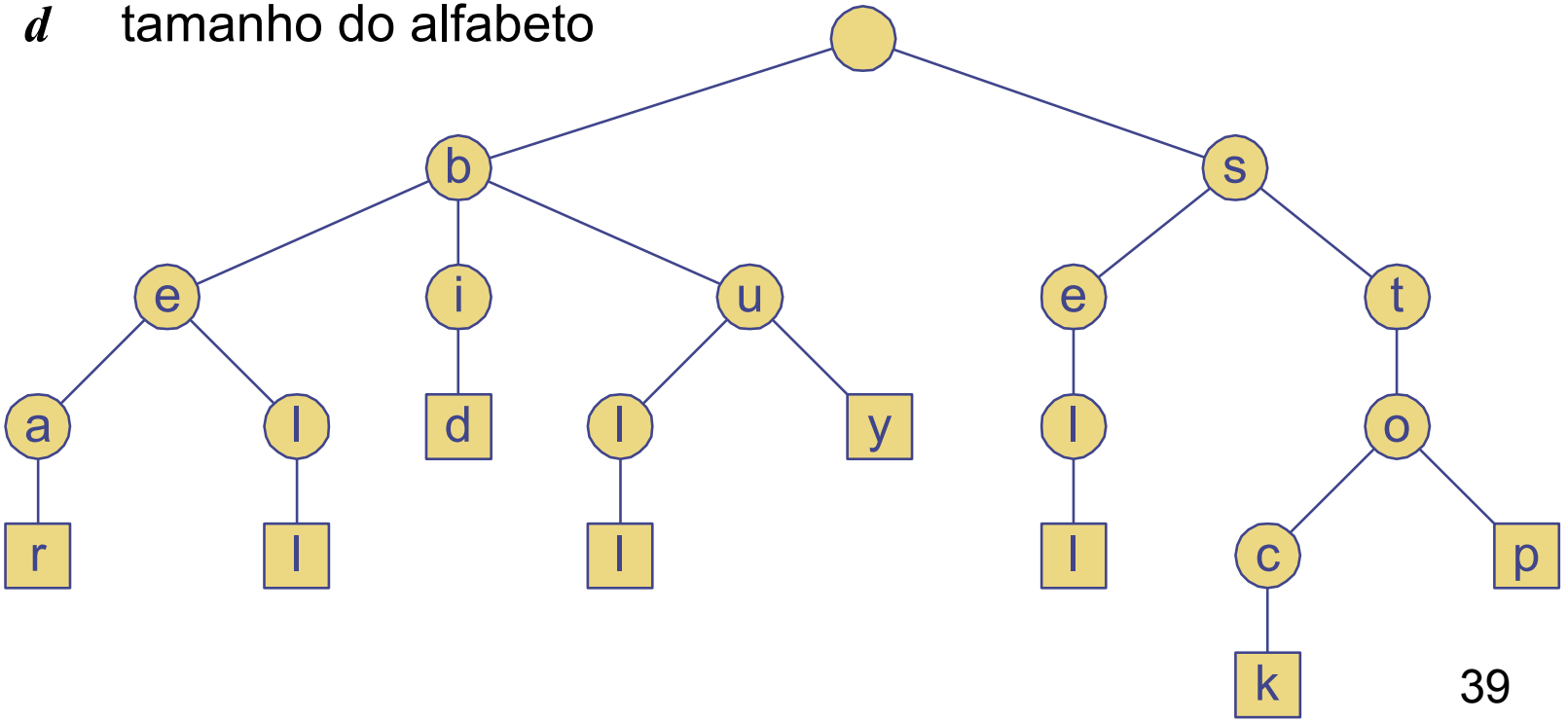
Procura filho de $x = s_i$ ou se não existe crio um na posição adequada

x ← esse filho ou o novo nó

Tempo: **O(dm)**

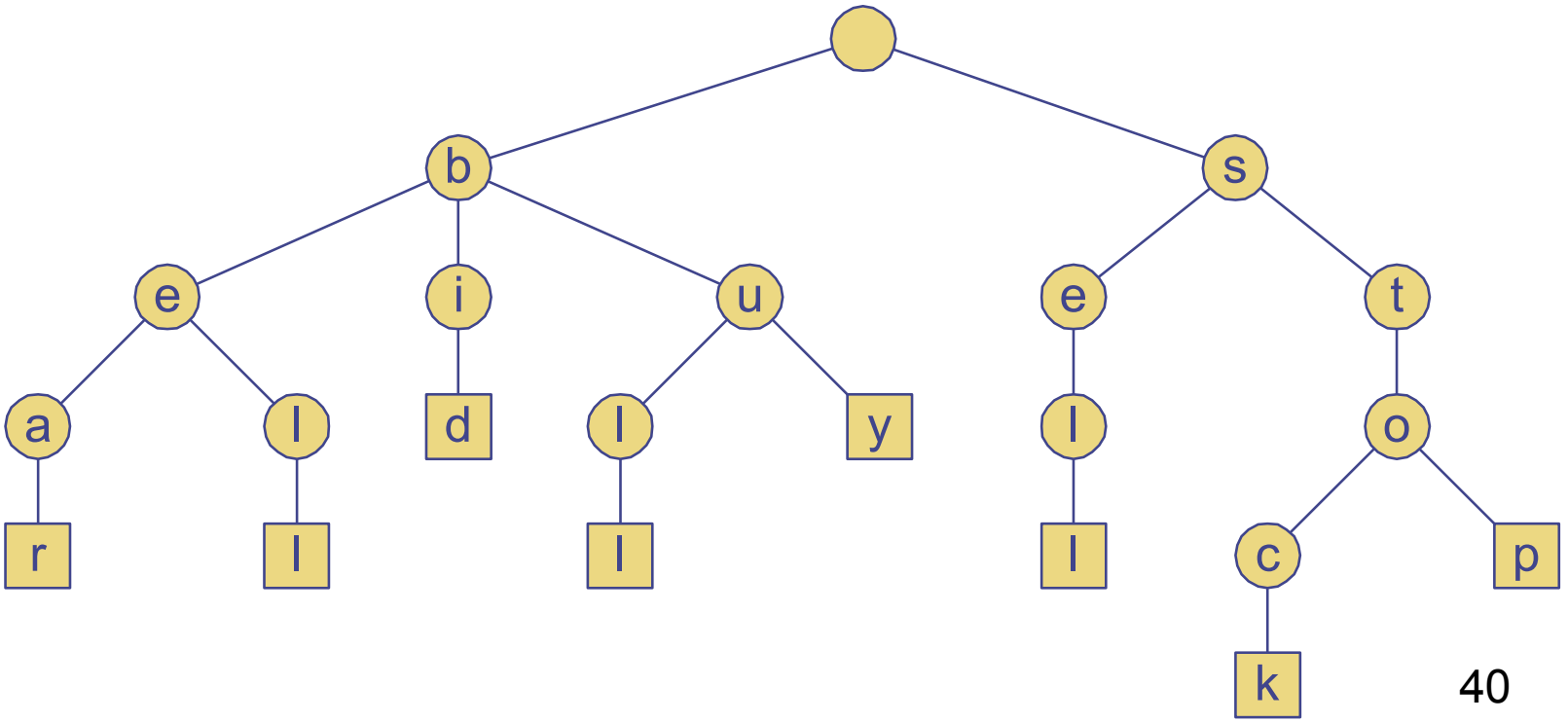
m comprimento da string sendo inserida

d tamanho do alfabeto



Tries Padrão - Remoção

Dada uma nova string $s = s_0, s_1, \ldots, s_{m-1}$



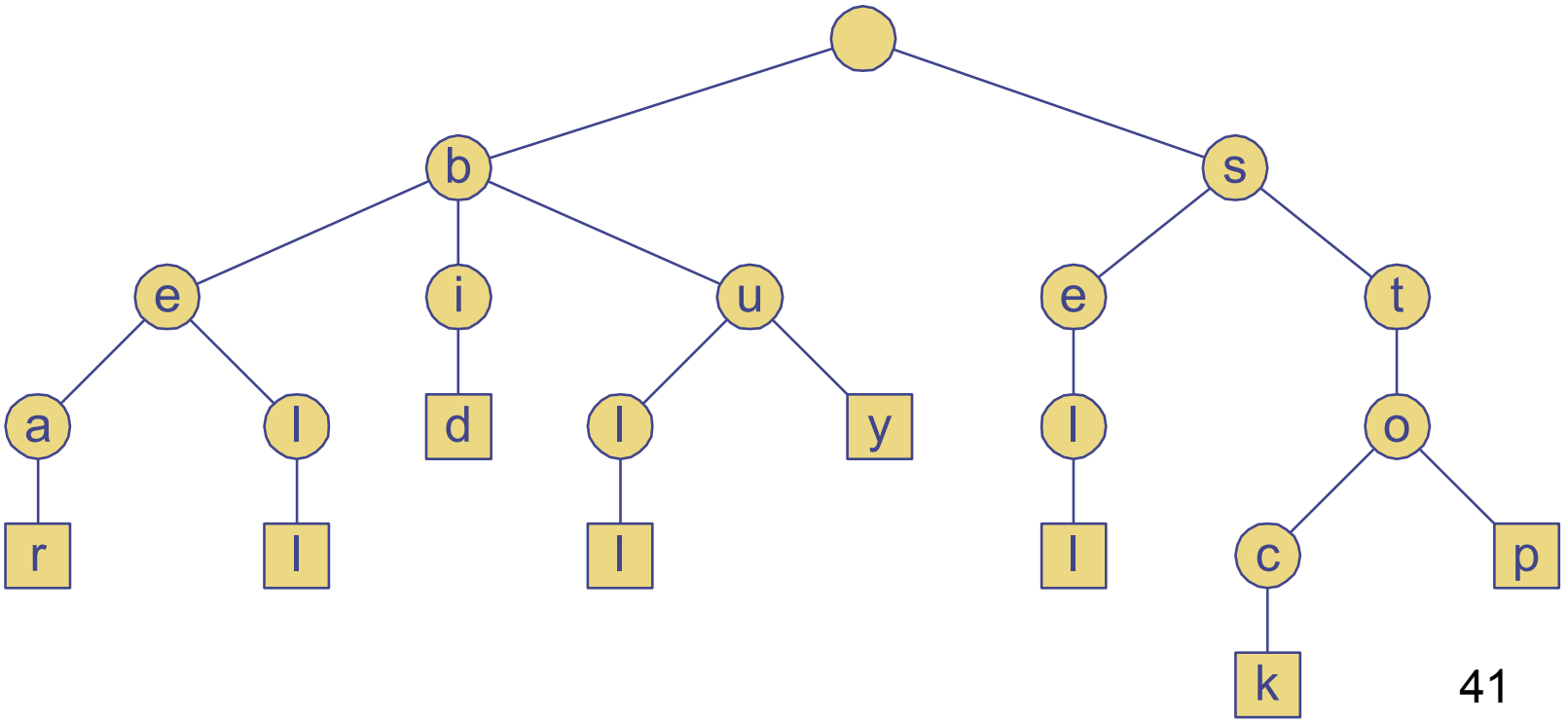
Tries Padrão - Remoção

Dada uma nova string s = s_0, s_1, …, s_{m-1}

Acha X ← folha relativa à s (processo semelhante à inserção, sem criação de novos nós)

Deleta x e sua aresta

Deleta recursivamente o pai se ele ficou sem filhos



Tries Padrão - Remoção

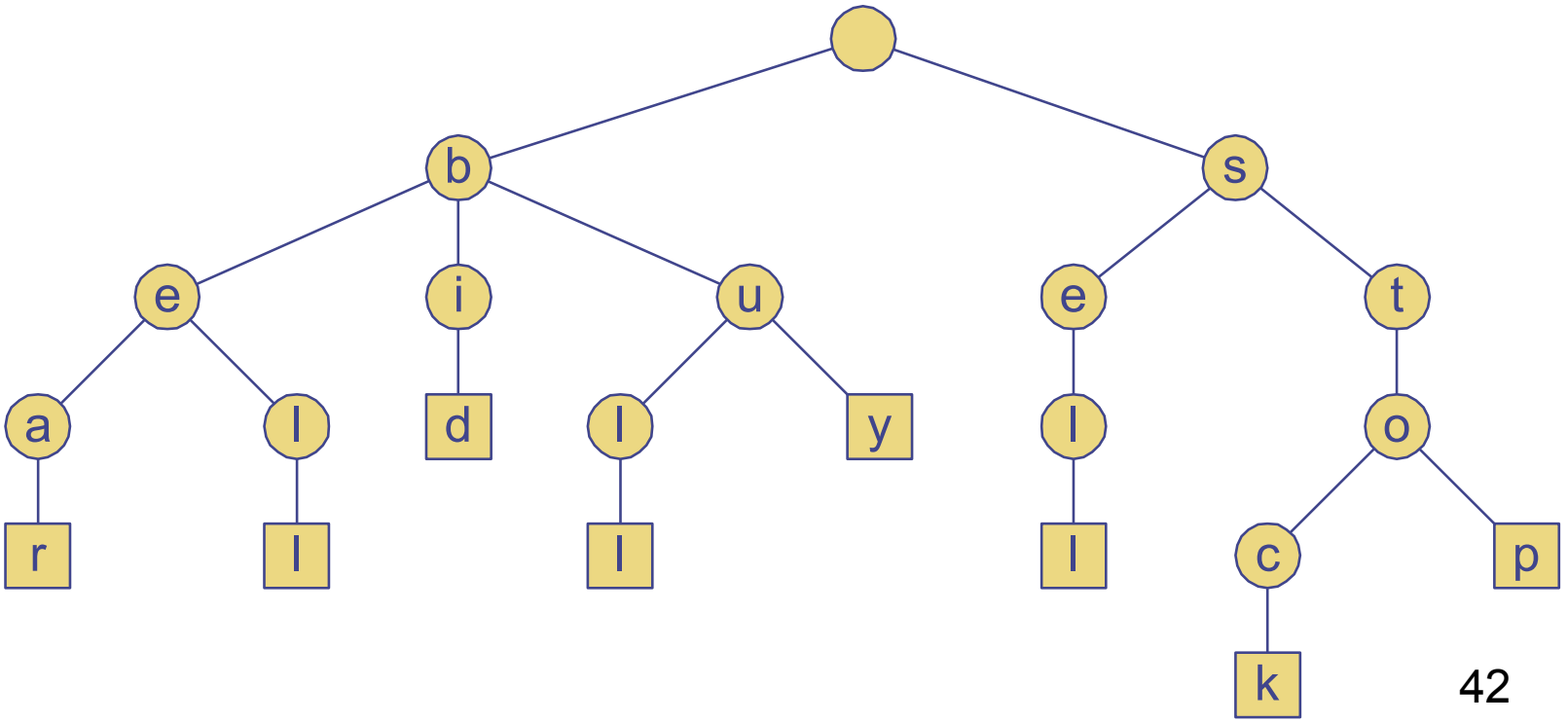
Dada uma nova string s = s_0, s_1, …, s_{m-1}

Acha X ← folha relativa à s (processo semelhante à inserção, sem criação de novos nós)

Deleta x e sua aresta

Deleta recursivamente o pai se ele ficou sem filhos

Tempo: ?



Tries Padrão - Remoção

Dada uma nova string s = s_0, s_1, …, s_{m-1}

Acha X ← folha relativa à s (processo semelhante à inserção, sem criação de novos nós)

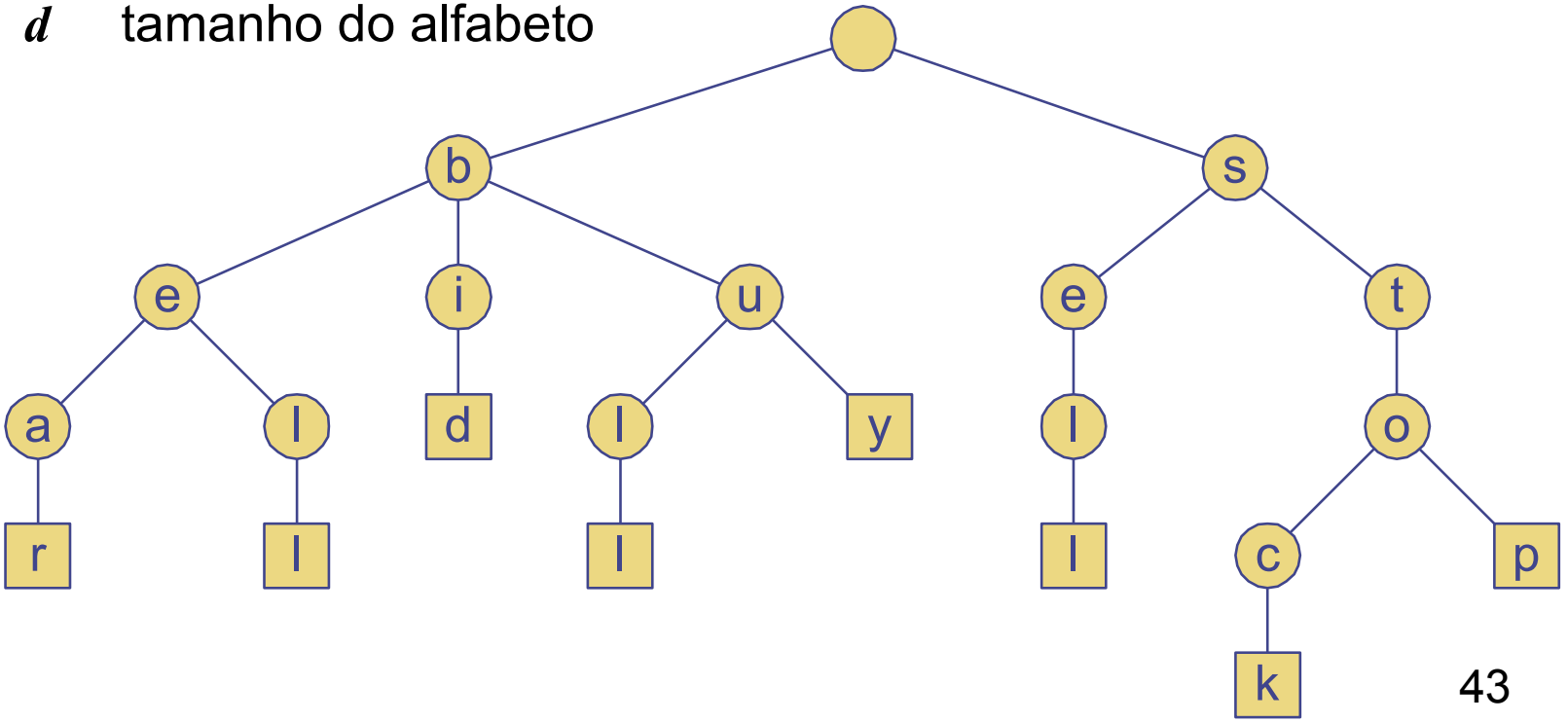
Deleta x e sua aresta

Deleta recursivamente o pai se ele ficou sem filhos

Tempo: **O(dm)**

m comprimento da string sendo removida

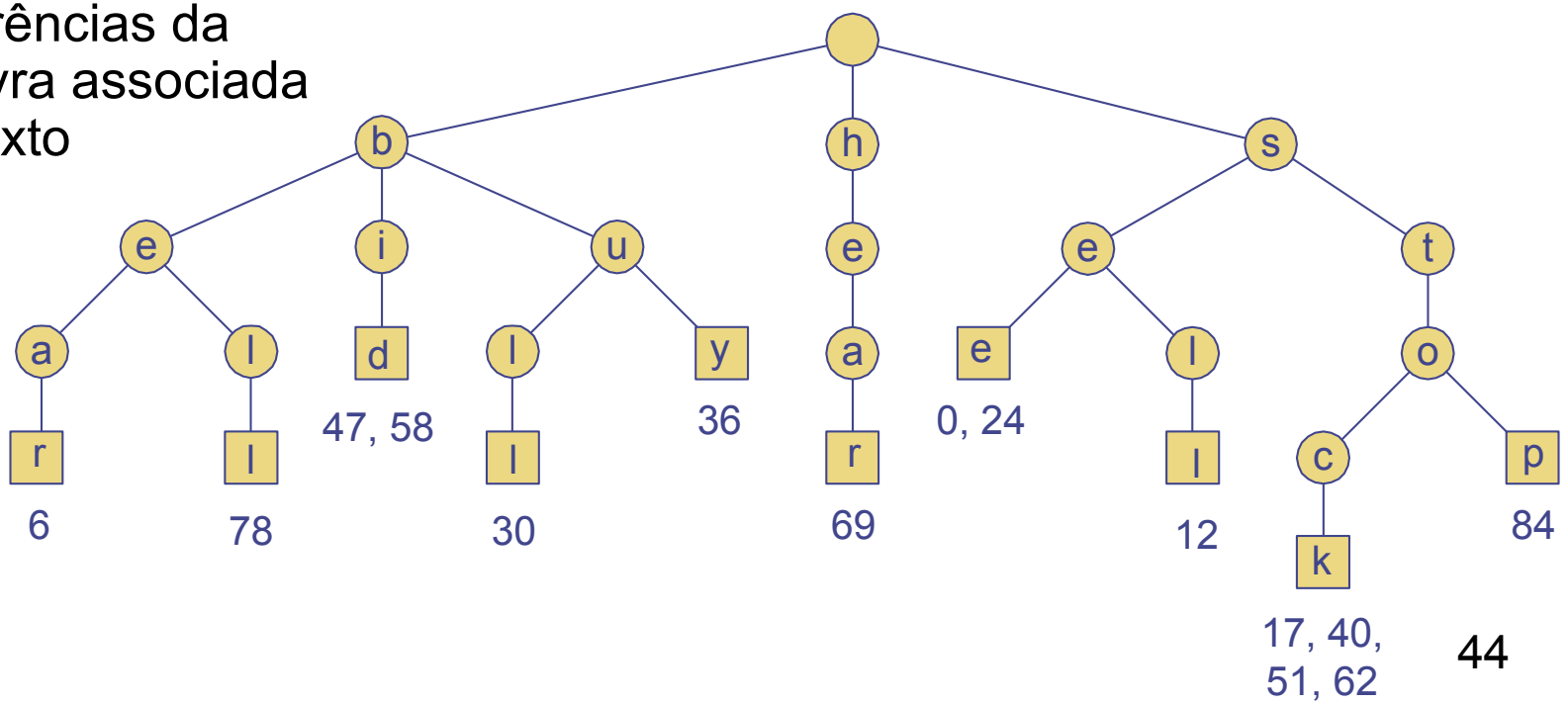
d tamanho do alfabeto



Matching de uma palavra usando uma Trie padrão

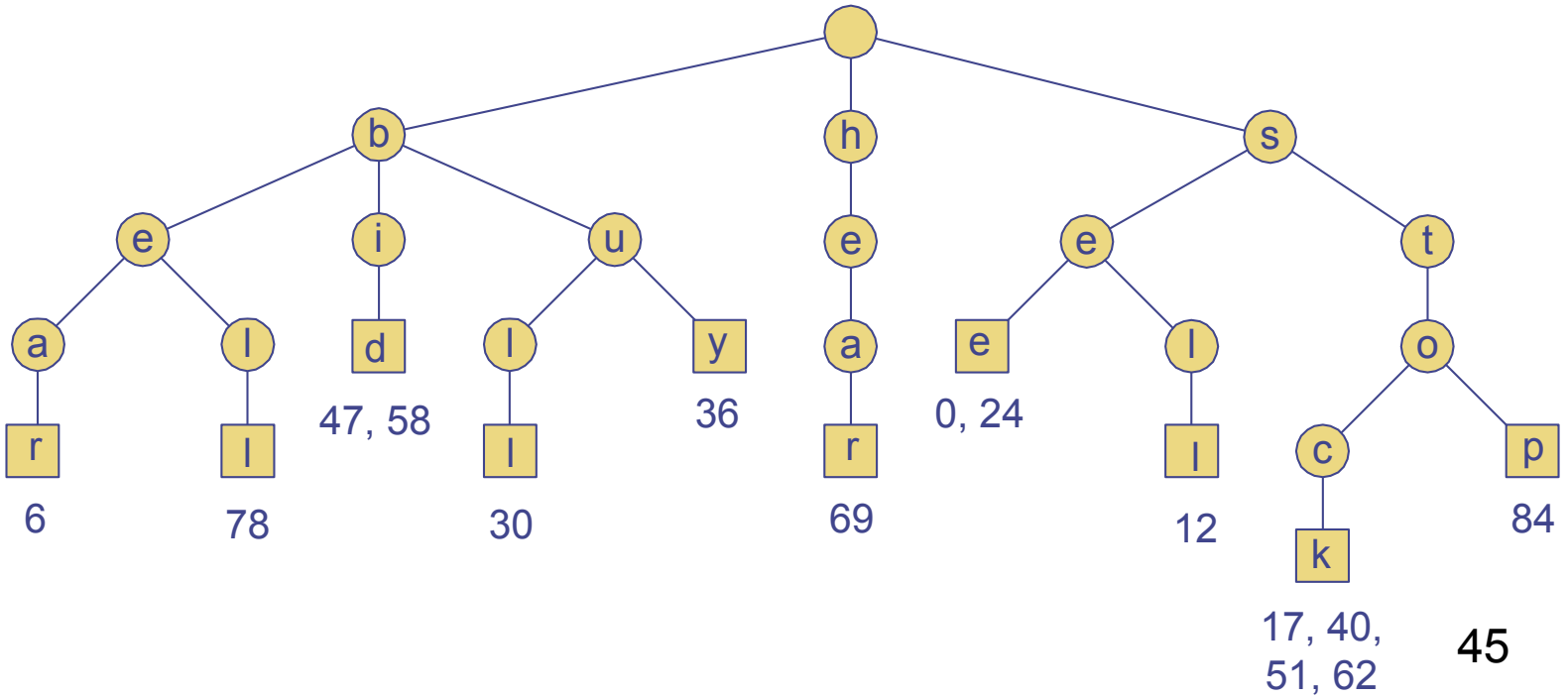
Inserimos as palavras de um texto em uma trie

Cada folha armazena as localizações das ocorrências da palavra associada no texto



Tries Padrão

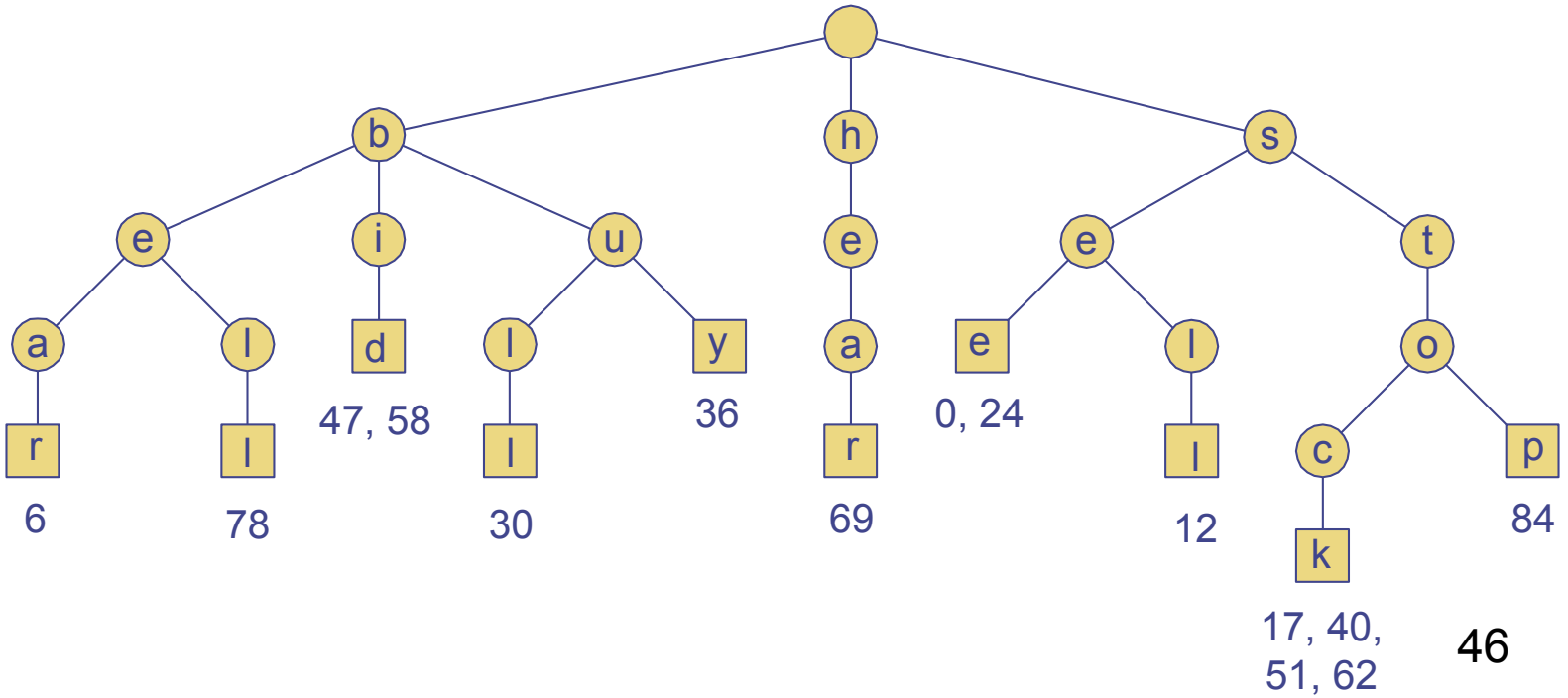
E se o texto tiver palavras que são prefixos de outras palavras?



Tries Padrão

E se o texto tiver palavras que são prefixos de outras palavras?

→ Problema (fim de palavra deveria ser uma folha…)

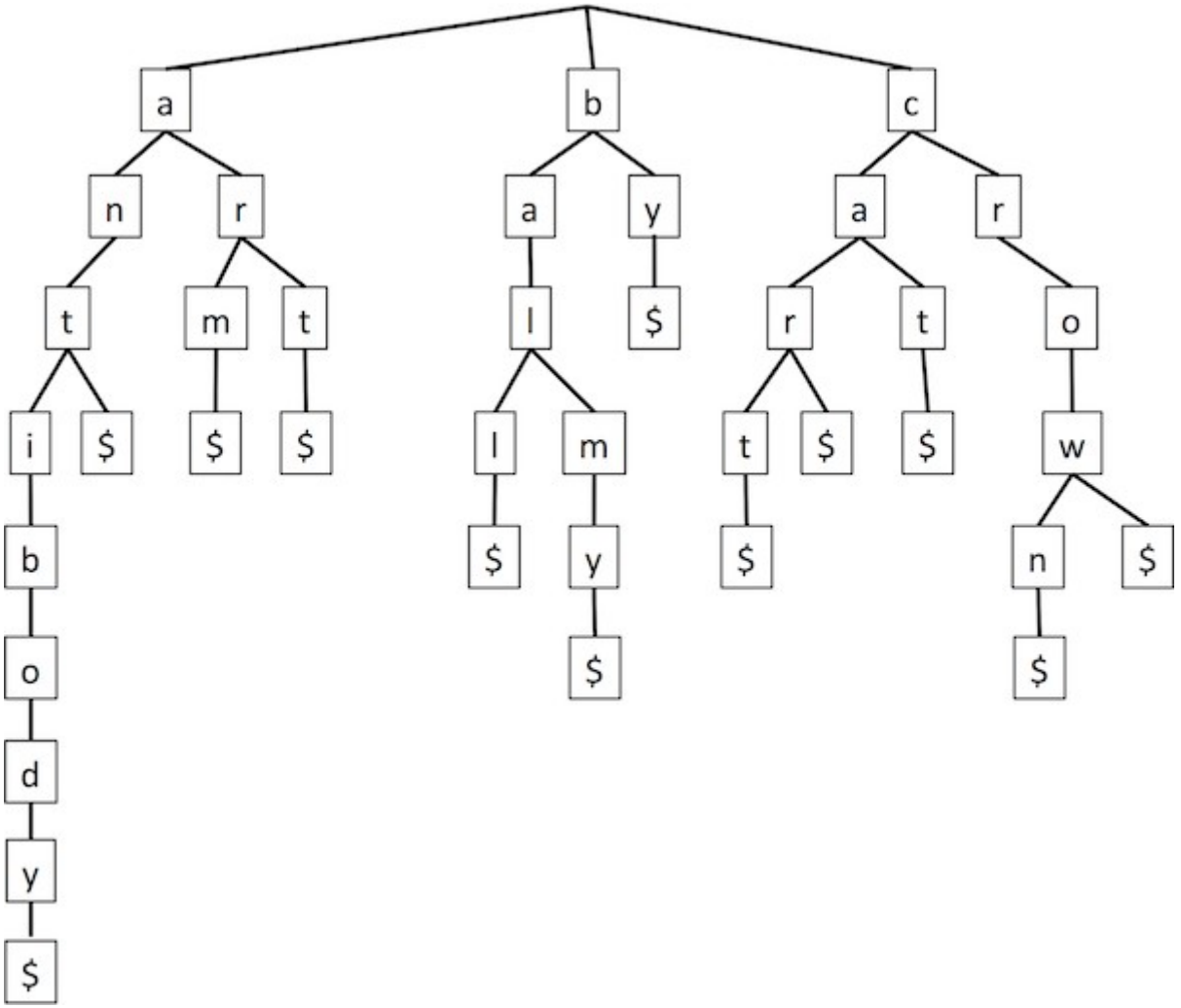


Tries Padrão

E se o texto tiver palavras que são prefixos de outras palavras?
→ Problema (fim de palavra deveria ser uma folha…)

Solução 1: adicionar ao final de cada palavra um símbolo especial de "fim de palavra" (ex: "$")

Esse tipo de trie, com valores ou significados apenas nas folhas, são também chamadas de "tries limpas"

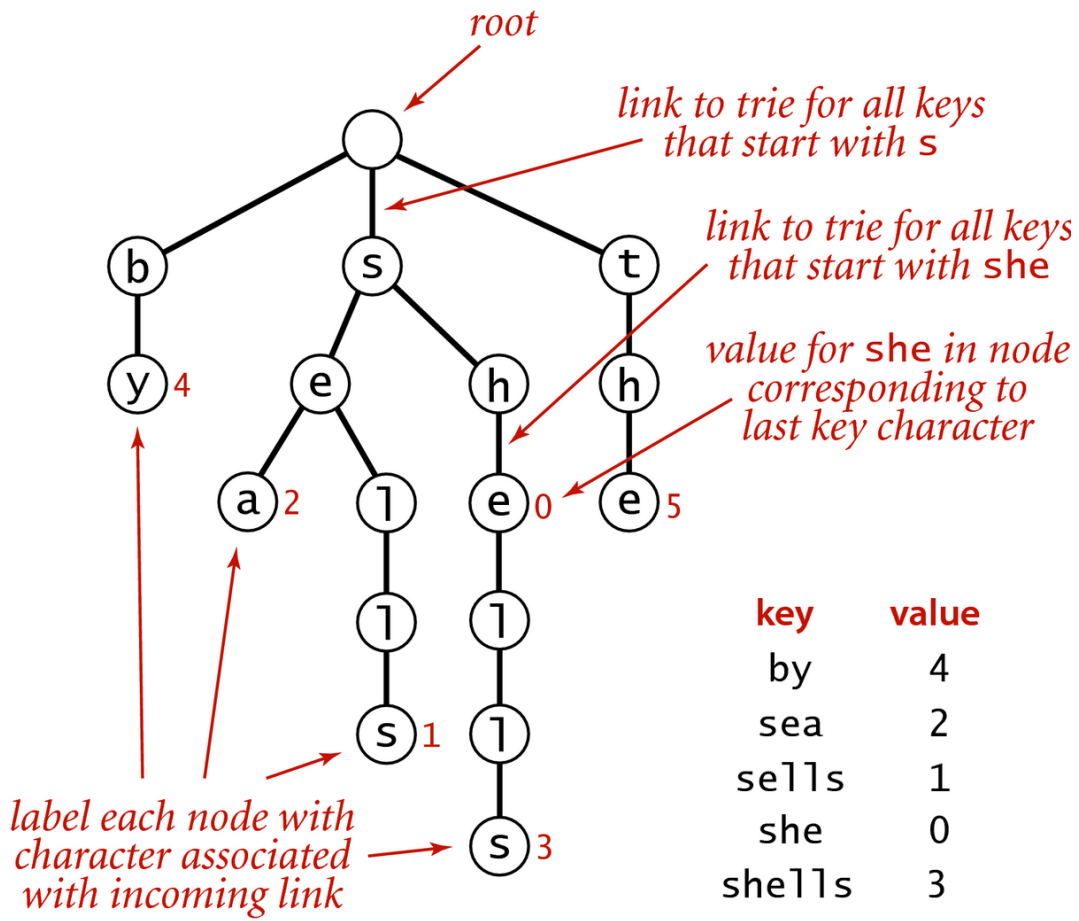


Tries Padrão

E se o texto tiver palavras que são prefixos de outras palavras?
→ Problema (fim de palavra deveria ser uma folha…)

Solução 2: nós internos
podem também possuir valor
(no caso, os índices
onde iniciam as palavras)

Observação: necessidade
de adaptação nas rotinas de
inserção e remoção →
EXERCÍCIO

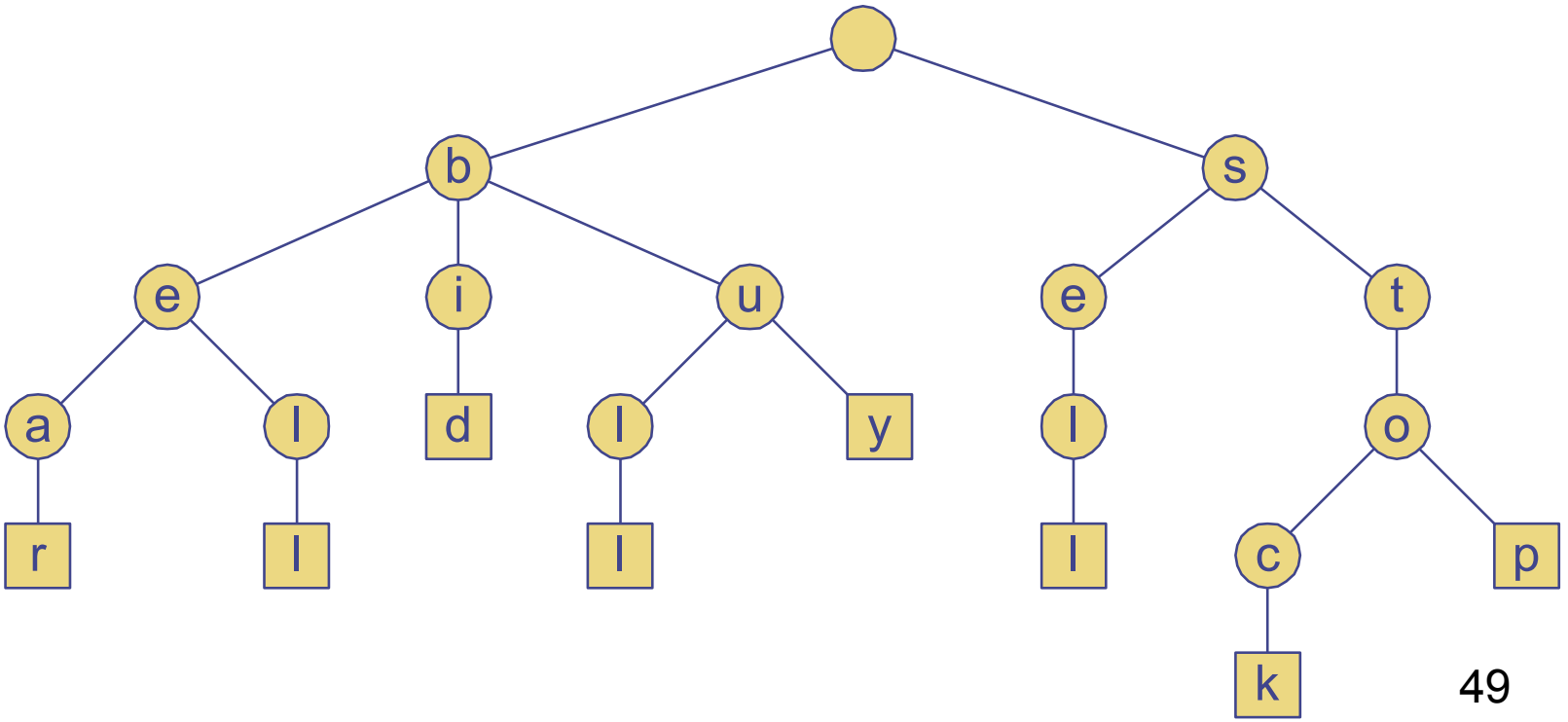


Anatomy of a trie

Tries Padrão: conclusões

Busca, inserção e remoção em tempo proporcional ao tamanho do padrão (não do texto)

Mas o espaço é caro (proporcional ao tamanho do texto, em caracteres)

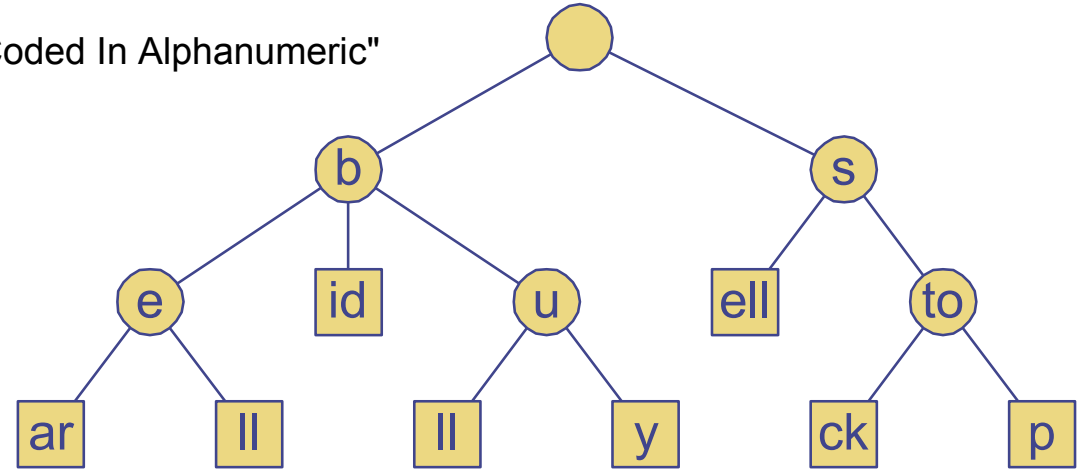


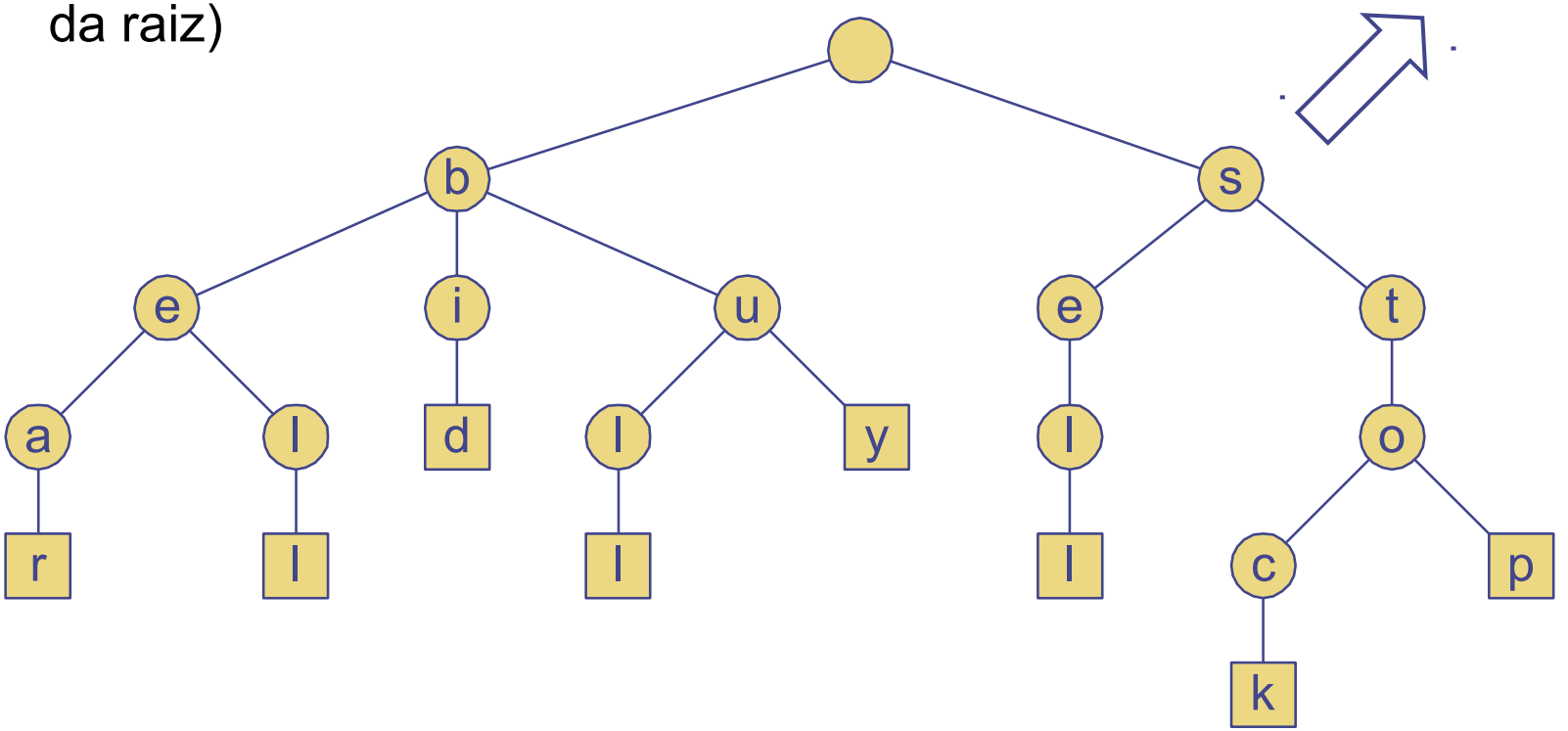
Tries Comprimidas, Radix Trie ou Trie Patricia

"Practical Algorithm To Retrieve Information Coded In Alphanumeric"

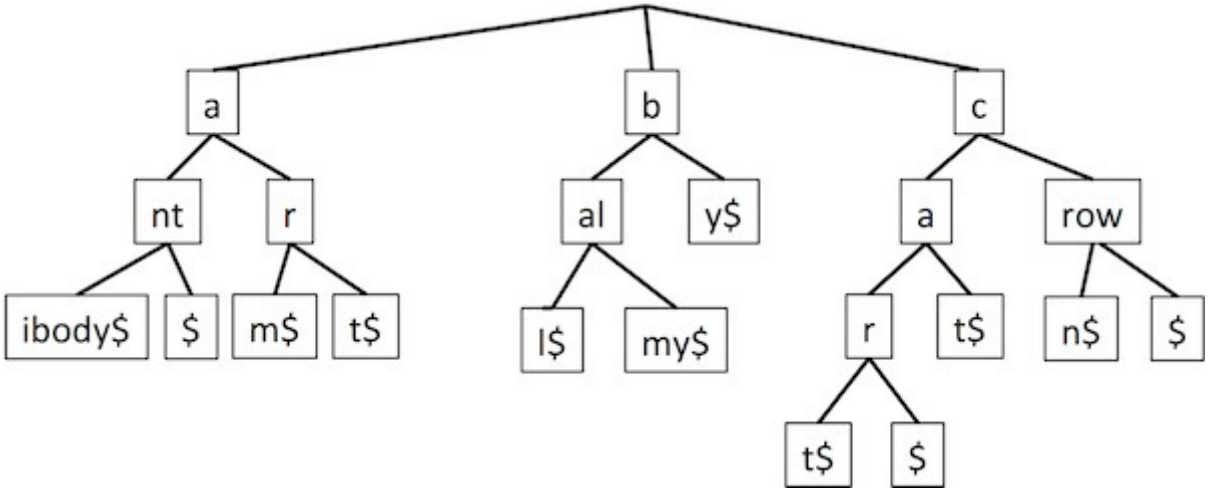
Nós internos possuem pelo menos dois filhos

Obtida a partir da trie padrão comprimindo cadeias de nós com apenas um filho (a menos da raiz)





Tries Comprimidas com caracter especial de fim de palavra



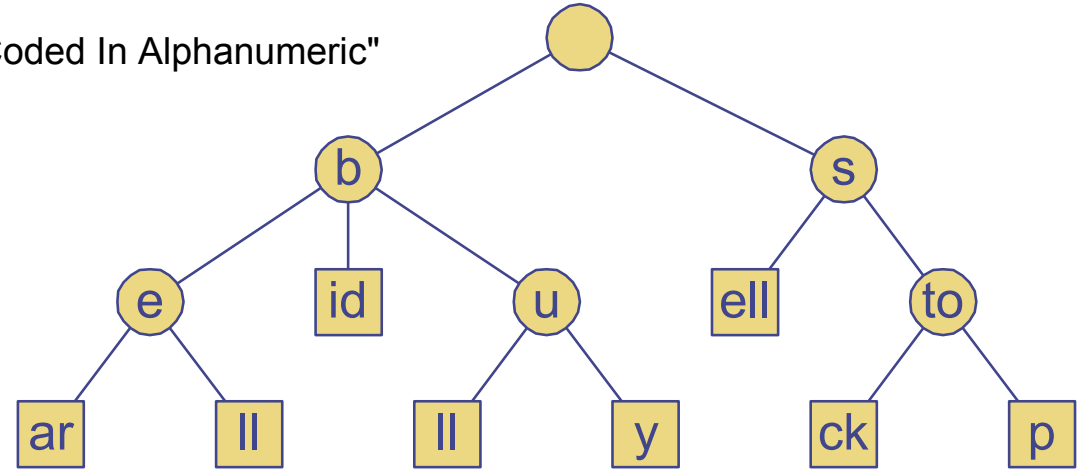
Tries Comprimidas, Radix Trie ou Trie Patricia

"Practical Algorithm To Retrieve Information Coded In Alphanumeric"

Nós internos possuem pelo menos dois filhos

Obtida a partir da trie padrão comprimindo cadeias de nós com apenas um filho (a menos da raiz)

Para um texto com s strings, o número de nós de uma trie comprimida é **O(s)**

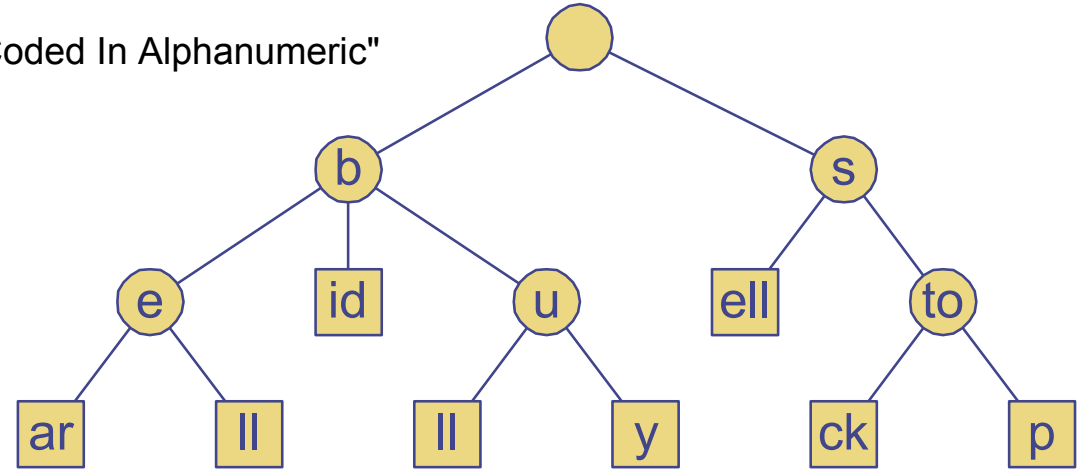


Tries Comprimidas, Radix Trie ou Trie Patricia

"Practical Algorithm To Retrieve Information Coded In Alphanumeric"

Nós internos possuem pelo menos dois filhos

Obtida a partir da trie padrão comprimindo cadeias de nós com apenas um filho (a menos da raiz)



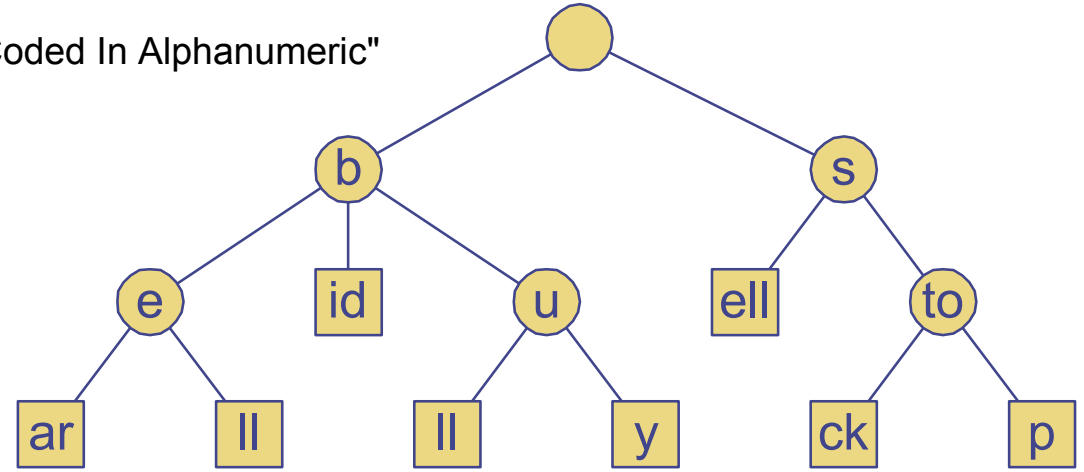
Há realmente ganho de espaço se ainda tenho que colocar todos os caracteres na árvore?

Tries Comprimidas, Radix Trie ou Trie Patricia

"Practical Algorithm To Retrieve Information Coded In Alphanumeric"

Nós internos possuem pelo menos dois filhos

Obtida a partir da trie padrão comprimindo cadeias de nós com apenas um filho (a menos da raiz)



Há realmente ganho de espaço se ainda tenho que colocar todos os caracteres na árvore?

Sim, se eu representar cada nó um pouco diferente…
Não armazenar explicitamente os caracteres, mas uma “referência” a eles...

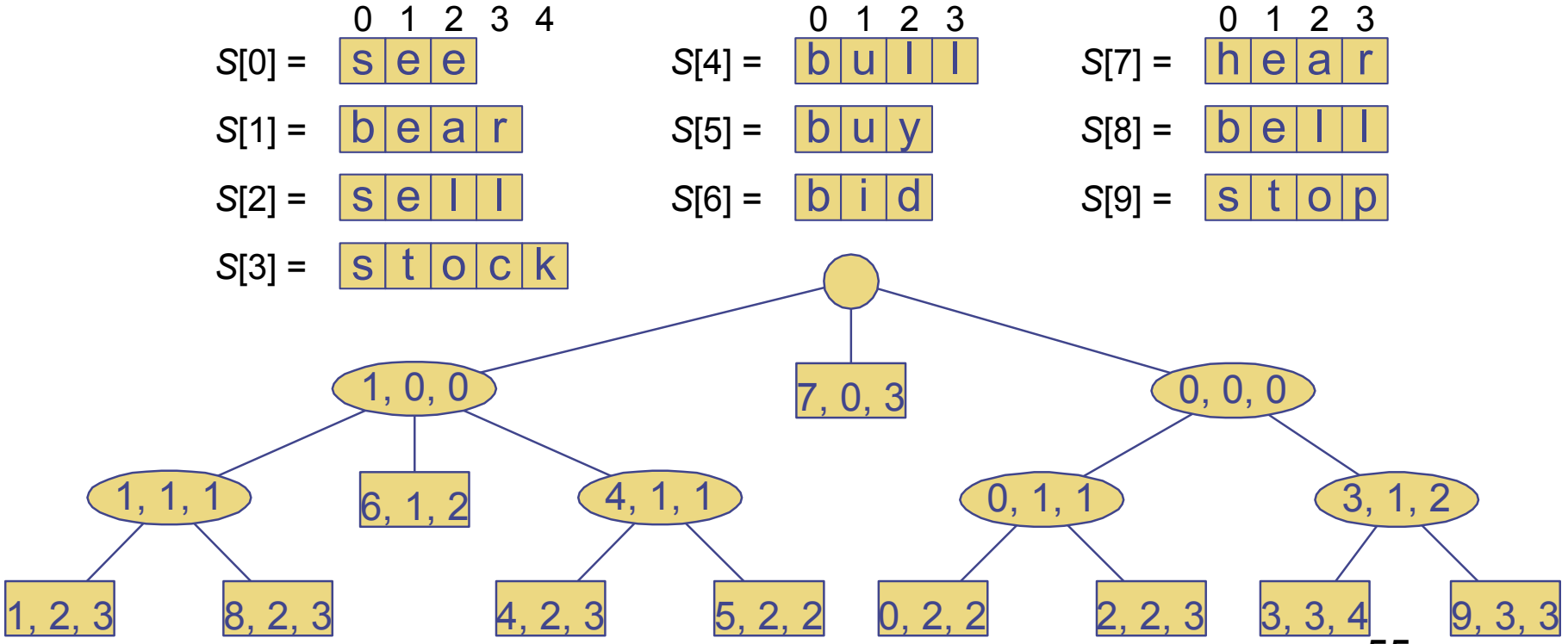
Representação compacta de tries comprimidas

As palavras ficam armazenadas em um vetor S (uma palavra em cada entrada S[i] do vetor)

Cada nó da trie é um trio (i, j, k) representando a substring de S[i] que se inicia na posição j e termina na posição k: S[i][j] S[i][j+1] … S[i][k]

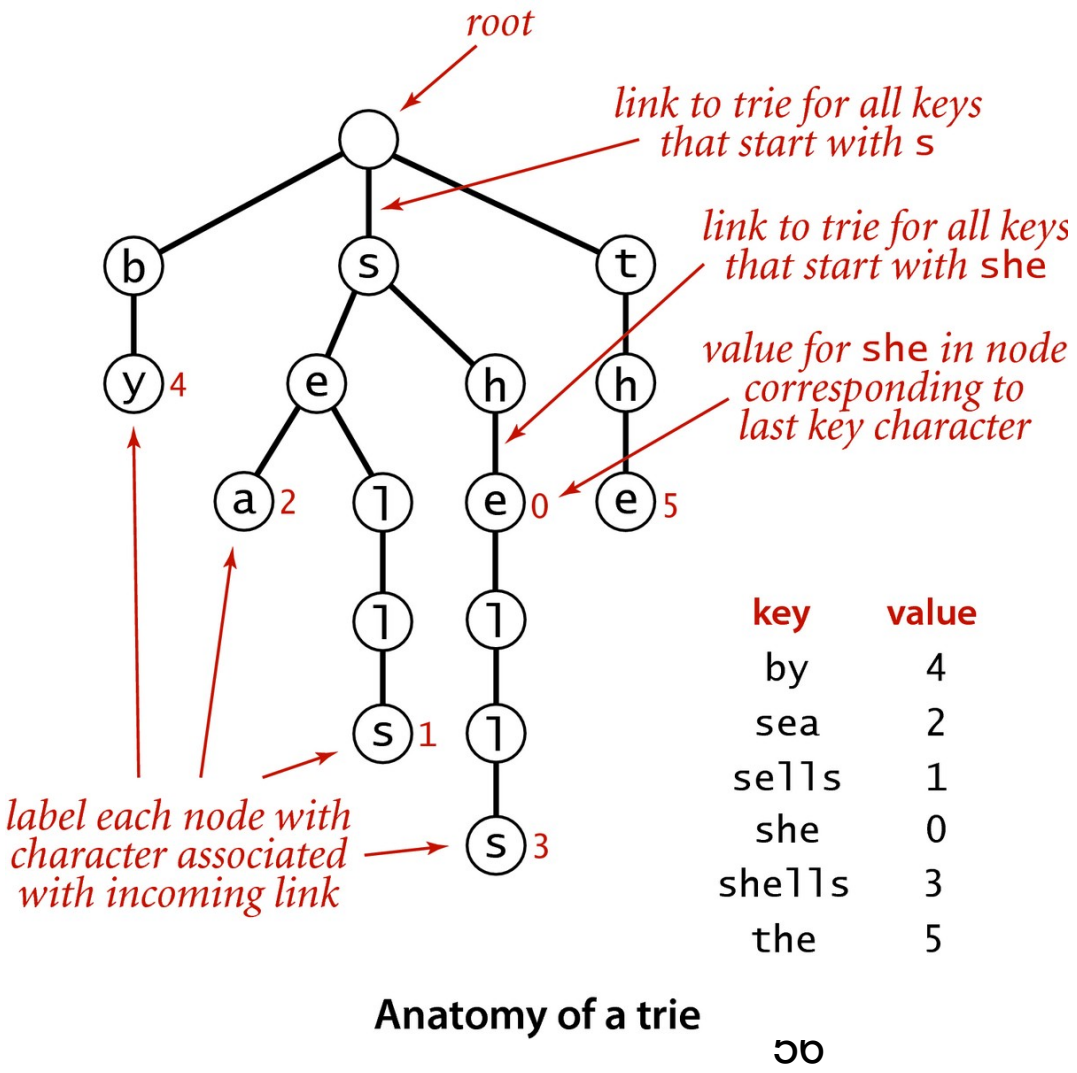
Usa $O(s)$ espaço, sendo s o número de strings no array S

A trie serve como uma estrutura auxiliar de índice (não para armazenamento propriamento dito)



Dicionários usando Tries

- Tries podem ser utilizadas para implementar um dicionário, que mantém pares <chave, valor> (hash faz isso também)
  - Esse valor pode ser uma lista de valores
  - Na verdade, isso é o que foi feito nos slides anteriores para armazenar, para uma dada palavra (chave) o índice de seu início em um texto (valor)



Anatomy of a trie

Arquivos invertidos (ou índices invertidos)

- Um exemplo de aplicação desses dicionários são arquivos invertidos (ou índices invertidos):
  - um dicionário na forma de índice (como índice remissivo), na qual as chaves (termos do índice) são strings e há vários valores associados a cada uma (lista de ocorrências), ie, páginas nas quais ocorre a palavra
  - Também usado (por máquinas de busca na web) para indexar páginas na web nais quais ocorre uma dada palavra

Arquivos invertidos

- Índice invertido pode ser implementado com:
  - Um vetor armazenando as listas de ocorrências dos termos (sem ordenação)
  - Uma trie comprimida para o conjunto de chaves, cujo valor armazena o índice (do vetor) da lista de ocorrência da chave associada ao respectivo nó
- A trie só tem as chaves para poder caber na memória, e as listas de ocorrências ficam armazenadas em disco
- Se o objetivo da busca é um “E” ou “OU” de palavras-chaves (como em uma busca web), recupera-se todas as listas de ocorrências e calcula-se a intersecção ou união, respectivamente
- Isso sem considerar a classificação por relevância, um importante serviço adicional ao usuário.

Referências

Livro/Transparências do livro do Goodrich & Tamassia, cap 12.2 e 12.3.