STRINGS

Problemas relacionados com strings

Exemplos

- Dado um texto e uma string (padrão), encontrar todas as ocorrências exatas da string no texto (problema clássico de pesquisa de uma string num texto)
- Dado um conjunto de strings, ordená-lo alfabeticamente
- Dado um conjunto de strings ordenado alfabeticamente, pesquisar uma dada substring naquele conjunto
- Dado um conjunto de strings, encontrar a string mais longa (de maior tamanho) que ocorre pelo menos k vezes
- Dado duas strings, encontrar a substring mais longa comum às duas strings

Exemplos

- Dado duas strings com o mesmo comprimento, determinar a distância de edição entre elas ou alinhamento de strings (número de posições nas quais elas diferem entre si), com várias operações disponíveis: remoção, adição e substituição de carateres
- Dado um conjunto de strings, determinar a "melhor" árvore que as pode descrever e ligar (árvore filogenética - representa as relações evolutivas entre as strings)
- Dado um conjunto de strings, determinar a menor super-string que contém todas as strings do conjunto (um dos principais problemas do sequenciamento de DNA)
- ...

Ordenação de strings

Introdução

- Os métodos de ordenação comparam chaves
  - abordagem genérica que se aplica a qualquer tipo de chaves: inteiros, reais ou carateres
- Alguns dos algoritmos mais conhecidos:
  - ordenação por troca (borbulhagem - “Bubblesort”)
  - ordenação por seleção
  - ordenação por separação (Quicksort)
  - ordenação por fusão (Mergesort)

- Abordagem alternativa para ordenação é processar as chaves por partes
  - exemplo: processar as primeiras letras do nome, quando se procura o nome num catálogo
  - não é necessário comparar chaves

Ideia

- Partir/separar uma chave em vários pedaços/partes
  - dígitos de um número numa dada base (radix)
    - 312 tem os dígitos 3, 1 e 2 na base 10
    - 312 tem os dígitos 100111000 na base 2
    - ”exemplo” tem 6 carateres (base 256)
- Ordenar de acordo com o primeiro pedaço/parte
  - primeiro vêm os números cujo dígito mais à esquerda é 0,
  - depois vêm os números cujo dígito mais à esquerda é 1,
  - depois vêm os números cujo dígito mais à esquerda é 2,
  - e assim sucessivamente
- Pode-se ordenar repetindo esse processo para todos os pedaços/partes

Algoritmo Radix

- O algoritmo Radix foi desenvolvido para ordenar grandes números inteiros
  - como trata um **inteiro** como uma **string de dígitos**, então
    é um algoritmo para ordenar strings
- Existem dois tipos de ordenação por Radix
  - ordenação radix MSD
    o processo começa pelo início da string (primeiro caráter, digito mais significativo)
  - ordenação radix LSD:
    o processo começa pelo fim da string (último caráter, dígito menos significativo)

Algoritmo Radix LSD – ordenação por dígito (exemplo)

- Ordenar crescentemente a lista de inteiros: 123, 142, 87, 263, 233, 14 e 132
- Considerar todos os elementos com o mesmo tamanho: colocar 0 à esquerda
- Começar por considerar a parte composta pelo dígito mais à direita

0	1	2	3	4	5	6
123	142	087	263	233	014	132

- Contar o número de elementos que existem com cada dígito, na parte considerada

	0	1	2	3	4	5	6	7	8	9
contador	0	0	0 1 2	0 1 2 3	0 1	0	0	0 1	0	0

Algoritmo Radix LSD – ordenação por dígito (exemplo)

- Calcular a posição dos elementos (da esquerda para a direita) na lista

	0	1	2	3	4	5	6	7	8	9
contador	0	0	0 1 2	0 1 2 3	0 1	0	0	0 1	0	0
posição			0 1	2 3 4	5			6		

- Colocar os elementos nas suas posições, seguindo a ordem anterior

123 (2), 142 (0), 087 (6), 263 (3), 233 (4), 014 (5) e 132 (1):

0	1	2	3	4	5	6
123	142	087	263	233	014	132
142	132	123	263	233	014	087

Algoritmo Radix LSD – ordenação por dígito (exemplo)

- Repetir o mesmo processo para o próximo dígito (da direita para a esquerda)

0	1	2	3	4	5	6
123	142	087	263	233	014	132
142	132	123	263	233	014	087

- Contar o número de elementos que existem com cada dígito, na parte considerada:

	0	1	2	3	4	5	6	7	8	9
contador	0	0 1	0 1	0 1 2	0 1	0	0 1	0	0 1	0

Algoritmo Radix LSD – ordenação por dígito (exemplo)

- Calcular a posição dos elementos (da esquerda para a direita) na lista

	0	1	2	3	4	5	6	7	8	9
contador	0	0 1	0 1	0 1 2	0 1	0	0 1	0	0 1	0
posição		0	1	2 3	4		5		6	

- Colocar os elementos nas suas posições, seguindo a ordem anterior

142 (4), 132 (2), 123 (1), 263 (5), 233 (3), 014 (0), 087 (6).

0	1	2	3	4	5	6
123	142	087	263	233	014	132
142	132	123	263	233	014	087
014	123	132	233	142	263	087

Algoritmo Radix LSD – ordenação por dígito (exemplo)

- Repetir o mesmo processo para o próximo dígito (da direita para a esquerda)

0	1	2	3	4	5	6
123	142	087	263	233	014	132
142	132	123	263	233	014	087
014	123	132	233	142	263	087

- Contar o número de elementos que existem com cada dígito na parte considerada

	0	1	2	3	4	5	6	7	8	9
contador	0 1 2	0 1 2 3	0 1 2	0	0	0	0	0	0	0

Algoritmo Radix LSD – ordenação por dígito (exemplo)

- Calcular a posição dos elementos (da esquerda para a direita) na lista

	0	1	2	3	4	5	6	7	8	9
contador	0 1 2	0 1 2 3	0 1 2	0	0	0	0	0	0	0
posição	0 1	2 3 4	5 6							

- Colocar os elementos nas suas posições, seguindo a ordem anterior

014 (0), 123 (2), 132 (3), 233 (5), 142 (4), 263 (6), 087 (1).

0	1	2	3	4	5	6
123	142	087	263	233	014	132
142	132	123	263	233	014	087
014	123	132	233	142	263	087
014	087	123	132	142	233	263

Algoritmo Radix LSD – para números inteiros tratados como strings

- Entrada
  - R = { S_1, S_2, …, S_N } de strings de comprimento **m** sobre o alfabeto { 0, …, base-1 }
  - número de strings do conjunto R, **N**
  - **base** (maior dígito considerado)
  - comprimento das strings (têm todas o mesmo comprimento), **numDigitos**
- Saída
  - R ordenado lexicograficamente por ordem crescente

Algoritmo Radix LSD – para números inteiros tratados como strings

```
algoritmo RadixLSD
  d ← 0
  enquanto (d ≤ numDigitos-1) fazer
     j ← 0
     enquanto (j ≤ base-1) fazer
        cont[j] ← 0
        j ← j + 1
     fim_enquanto
     para k desde 1 até N fazer
        dig ← S_k[d]
        cont[dig+1] ← cont[dig+1] + 1
     fim_para
     para j desde 1 até base-1 fazer
        cont[j] ← cont[j] + cont[j-1]
     fim_para
...
```

Algoritmo Radix LSD – para números inteiros tratados como strings

```
...
      para k desde 0 até base-1 fazer
         dig ← S_k[d]
         posicao ← cont[dig]
         cont[dig] ← cont[dig] + 1
         Aux[posicao] ← R[k]
      fim_para
      para k desde 0 até base-1 fazer
         R[k] ← Aux[k]
      fim_para
      d ← d + 1
   fim_enquanto
fim_algoritmo
```

Algoritmo Radix LSD – vantagens e desvantagens

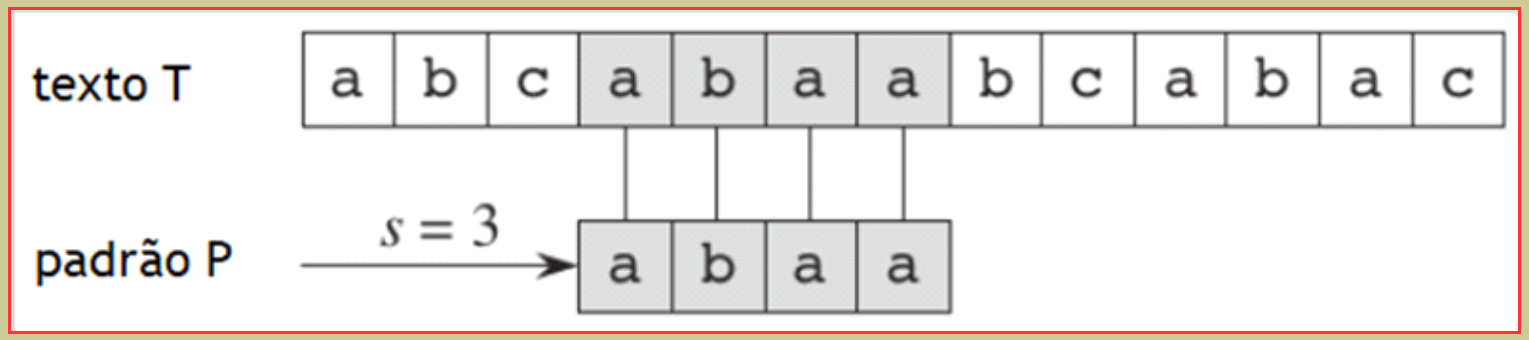
- Vantagens
  - estável
  - não compara as chaves
- Desvantagens
  - nem sempre é fácil otimizar a inspeção de dígitos
  - depende do hardware
  - só é bom se o número de dígitos for pequeno

Pesquisar uma substring numa string (Casamento de Padrões)

Definição

- Consiste em determinar a ocorrência de um determinado padrão num texto
  - dado um texto T de comprimento n (T[1...n], com T[i] $\in \Sigma$)
  - dado um padrão P de comprimento m (P[1...m], com P[i] $\in \Sigma$) e m $\leq$ n
    - Σ é o alfabeto a que pertence os elementos de T e P

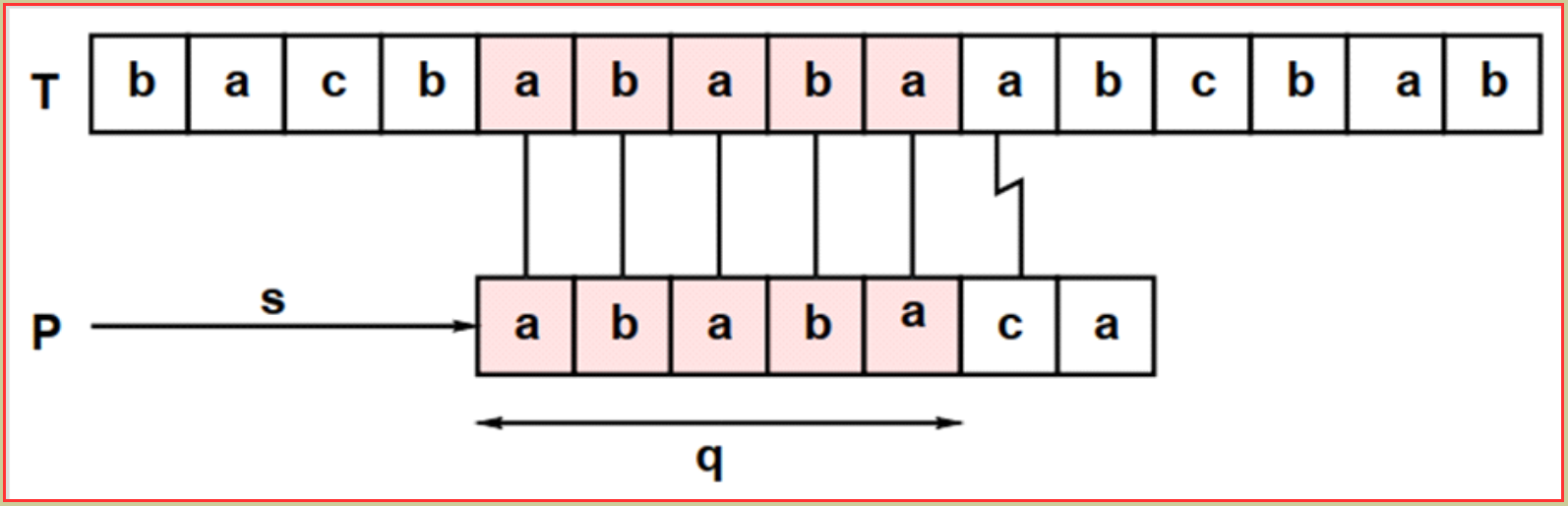
      ex: Σ = { 0, 1 }, Σ = { a, b, …, z }
  - determinar todos os deslocamentos válidos de P em T, em que
    - deslocamento válido s significa que T[s+1...s+m] = P[1...m]



- Uma variante comum deste problema é determinar apenas um deslocamento válido
  - por exemplo: o primeiro

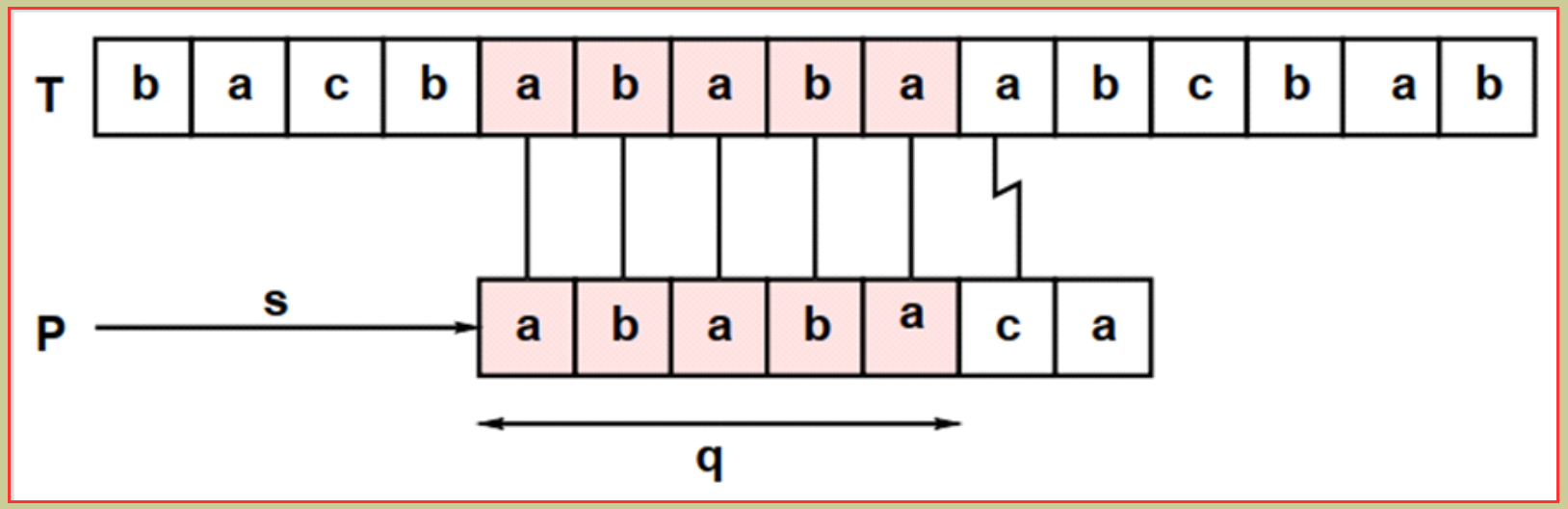
Algoritmo de Knuth-Morris-Pratt (KMP)

- O algoritmo KMP utiliza uma operação auxiliar (Prefixo de um Padrão)
  - que é pré-calculada a partir do padrão P
  - contém informação sobre como o padrão casa consigo mesmo
  - esta informação evita testar deslocamentos inúteis
  - calcula e devolve PP[1...m] (PP = Prefixo de Padrão)
- Prefixo de um Padrão
  - considere-se o padrão P = **ababaca**
  - confronte-se P com um texto T, considerando o deslocamento s



Algoritmo de Knuth-Morris-Pratt (KMP)

- Prefixo de um Padrão



- ao testar a ocorrência de P,
  - verifica-se que a comparação falha ao sexto caráter de **P**
  - logo, 5 carateres (**q = 5**) casaram
- pode-se usar esta informação para determinar que certos deslocamentos são inválidos; por exemplo, pode-se afirmar que,
  - o deslocamento **s+1** é necessariamente inválido
  - o deslocamento **s+2** é potencialmente válido

Algoritmo de Knuth-Morris-Pratt (KMP)

- Prefixo de um Padrão
  - para se determinar qual o próximo deslocamento potencialmente válido:
    - verificar que deslocamento não fere o casamento de **P** consigo mesmo
  - ou seja,
    - se **q** carateres casaram com deslocamento **s**,
    - o próximo deslocamento potencialmente válido é:

      s' = s + (q – PP[q])

    - PP[q] é o maior inteiro menor que **q**, que corresponde ao maior prefixo de **P** que é sufixo próprio de P[1...q]
  - No exemplo:

    PP[5] = 3

    logo, s’ = s + (5 – 3) = s + 2

Algoritmo de Knuth-Morris-Pratt (KMP)

- Prefixo de um Padrão
  - exemplo do cálculo de PP:

k	1	2	3	4	5	6	7	8	9	10
P[k]	a	b	a	b	a	b	a	b	c	a
PP[k]	0	0	1	2	3	4	5	6	0	1

Algoritmo de Knuth-Morris-Pratt (KMP)

```
algoritmo calculaPrefixoPadrao (P)
  M ← |P|
  PP[1] ← 0
  k ← 0
  q ← 2
  enquanto (q ≤ m) fazer
     enquanto (k > 0  e  P[k+1] ≠ P[q]) fazer
        k ← PP[k]
     fim_enquanto
     se (P[k+1] = P[q]) então
        k ← k + 1
     fim_se
     PP[q] ← k
     q ← q + 1
  fim_enquanto
  devolver PP
fim_algoritmo
```

Algoritmo de Knuth-Morris-Pratt (KMP)

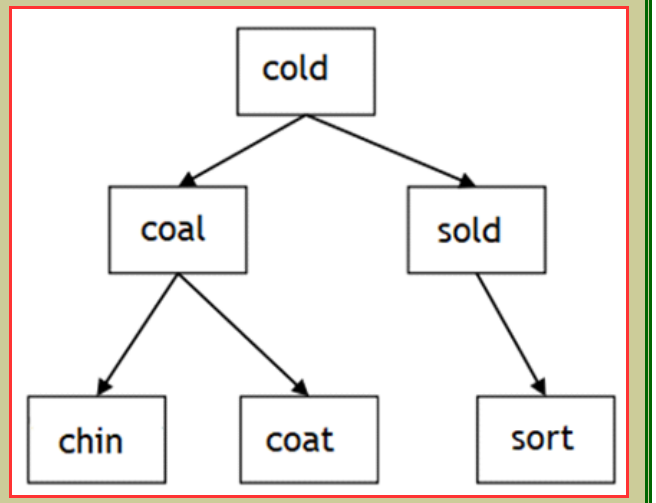
```
algoritmo KMP (T, P)
  n ← |T|
  m ← |P|
  PP ← calculaPrefixoPadrao(P)
  q ← 0
  k ← 1
  enquanto (k ≤ n) fazer
     enquanto (q > 0  e  P[q+1] ≠ T[k]) fazer
        q ← PP[q]
     fim_enquanto
     se (P[q+1] = T[k]) então
        q ← q + 1
     fim_se
     se (q = m) então
        q ← PP[q]
     fim_se
     k ← k + 1
  fim_enquanto
 fim_algoritmo
```

Trie (Prefix Tree)

Introdução

- Pode-se guardar um conjunto de chaves com uma Árvore Binária de Pesquisa
  - custo de pesquisar uma chave na árvore:
    - O(log(n)), com n = número de nodos da árvore
  - custo de comparar uma chave com o elemento do nodo:
    - valores inteiros: O(1)
    - strings: O(k), com k = comprimento da string
  - logo, quando as chaves são strings, o custo total é:

    O(k log(n))



Introdução

- Pode-se guardar um conjunto de chaves com uma Tabela de Hash (Hash Table)

	. . .	10	11	12	. . .
Tabela	. . .	"coal"	"coat"	"cold"	. . .

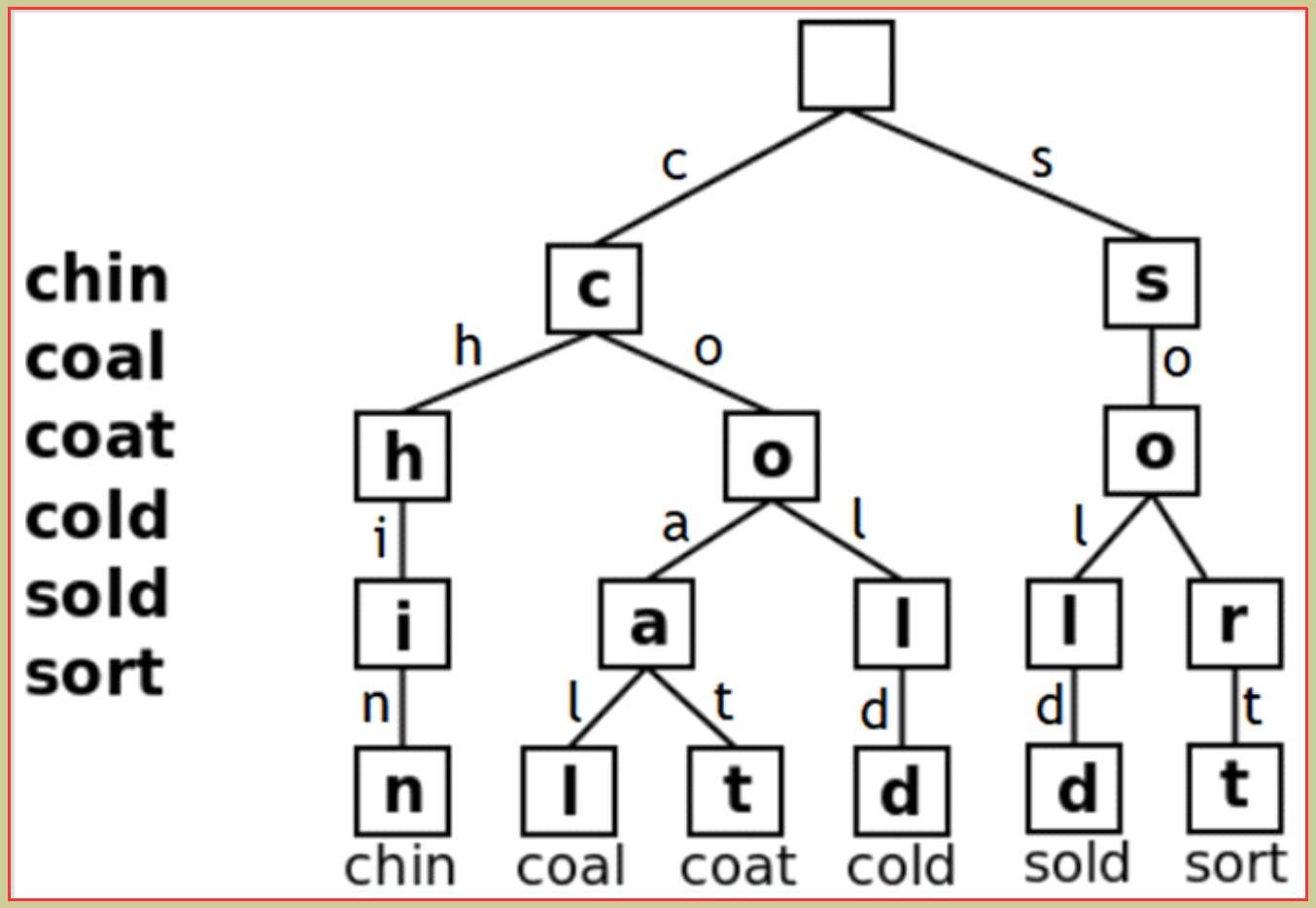
  - hash("coat") = 11
  - custo de pesquisar a posição original de uma chave na Tabela (aplicar hash): O(1)
  - custo de comparar uma chave com o elemento que está naquela posição:
    - a chave é um valor inteiro: O(1)
    - a chave é uma string: O(k), k = comprimento da string
  - portanto, quando as chaves são strings, o custo total é:
    - se não há colisões: O(k)
    - se há colisões: O(q.k), q = número de colisões
  - assumindo que as chaves são únicas e não há colisões, o custo da operação pesquisar pode ser O(k): ou seja, o tempo de comparação de uma chave
    - é independente do número de chaves que estão guardadas (n), e
    - apenas depende do tamanho da chave

Definição

- Uma **Trie** ou **Prefix Tree** ("Árvore de Prefixos") é uma EAD que representa um conjunto de strings/palavras (que podem ter valores associados a si), em que
  - a raiz representa a string/palavra vazia
  - os descendentes partilham o mesmo prefixo
- Um nodo, em vez de guardar uma chave completa, guarda um prefixo da chave
- Numa **Trie** são partilhados os caminhos das chaves que têm um prefixo comum

Definição

- Exemplo:



- Nota: as letras podem ser representadas nos arcos, em vez de nos nós

Estrutura dos nodos de uma Trie

- Árvore **densa** (com maioria dos carateres) ou **tamanho elevado** do alfabeto
  - array de 1 dimensão (vetor) de ponteiros para os filhos
  - um ponteiro para cada possível caráter do alfabeto

```
struct NodoTrie {
    char  Elemento;
    struct NodoTrie  *Filhos[tamAlfabeto];
};
```

Estrutura dos nodos de uma Trie

- Árvore **esparsa**
  - uma lista (ligada) de filhos
  - um ponteiro para o primeiro filho da lista

```
struct NodoTrie {
    char  Elemento;
    struct NodoTrie  *Filho;
    struct NodoTrie *Prox;
}
```

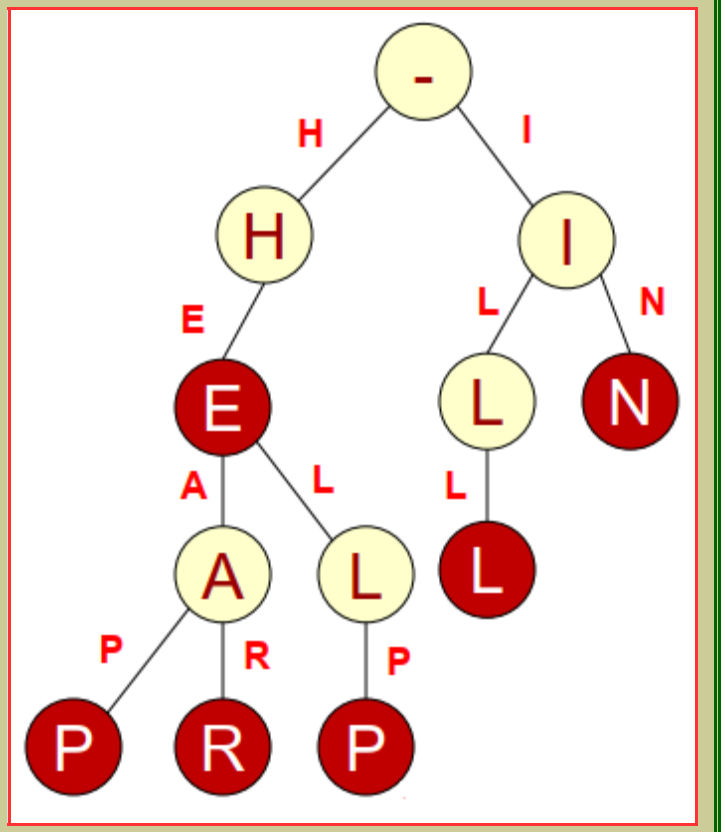
Operações sobre uma Trie

- Pesquisa de uma string numa Trie
  - consume um caráter em cada iteração (comparação)
  - no fim da pesquisa da chave
    - se foram consumidos todos os carateres da chave, então esta chave existe na Trie
  - ordem de complexidade: O(n), em que n = tamanho da string
- Inserir uma string numa Trie
  - pesquisa até a chave ser consumida num caminho da Trie que existe (chave é substring de uma string já inserida na Trie)
  - pesquisa até o caminho da Trie ser NULL (consume parte da string), estende o caminho adicionando nodos com os carateres da string que faltam
  - ordem de complexidade: O(n), em que n = tamanho da string

Operações sobre uma Trie

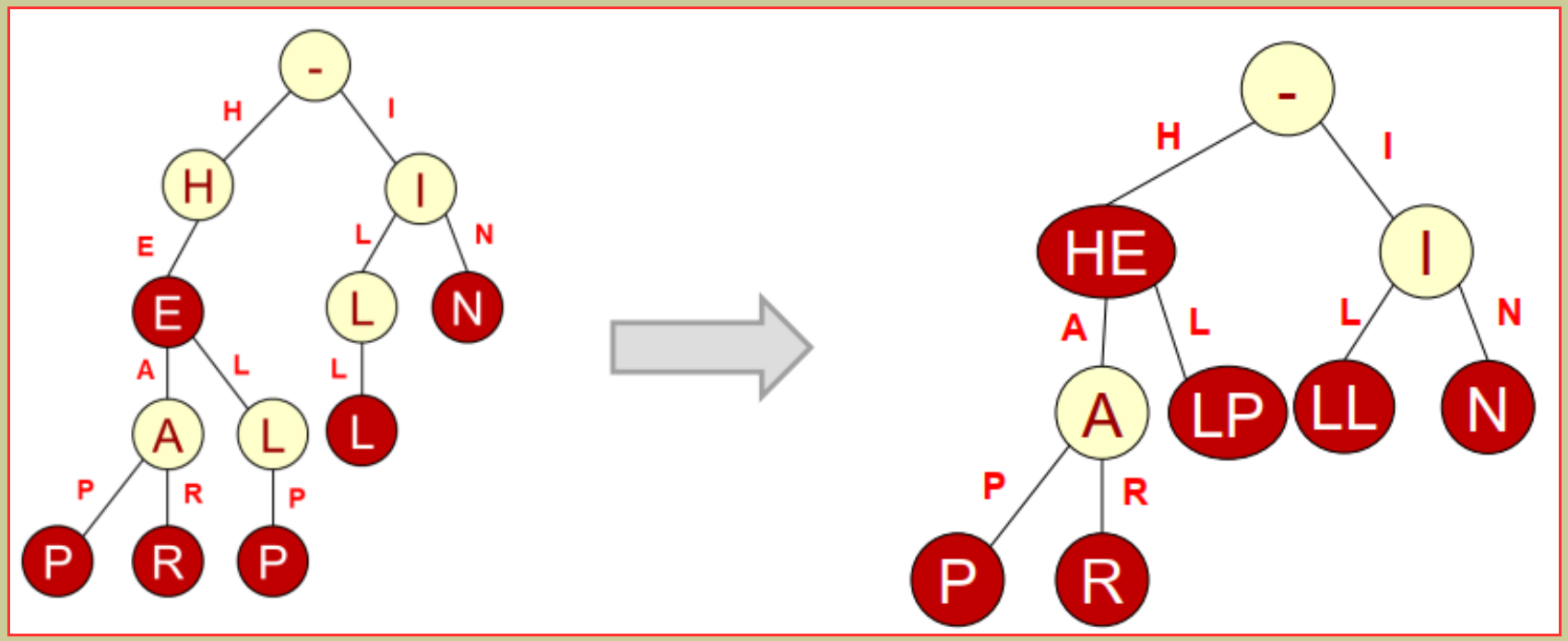
- Remover uma string de uma Trie
  - Como a remoção de uma chave funcionaria numa Trie?
  - Com que casos se deve ter cuidado?
  - A remoção de uma chave significa sempre a remoção de um nó?
  - Se removermos um nó, seria apenas um nó no teste?
  - ordem de complexidade: O(n), em que

    n = tamanho da string



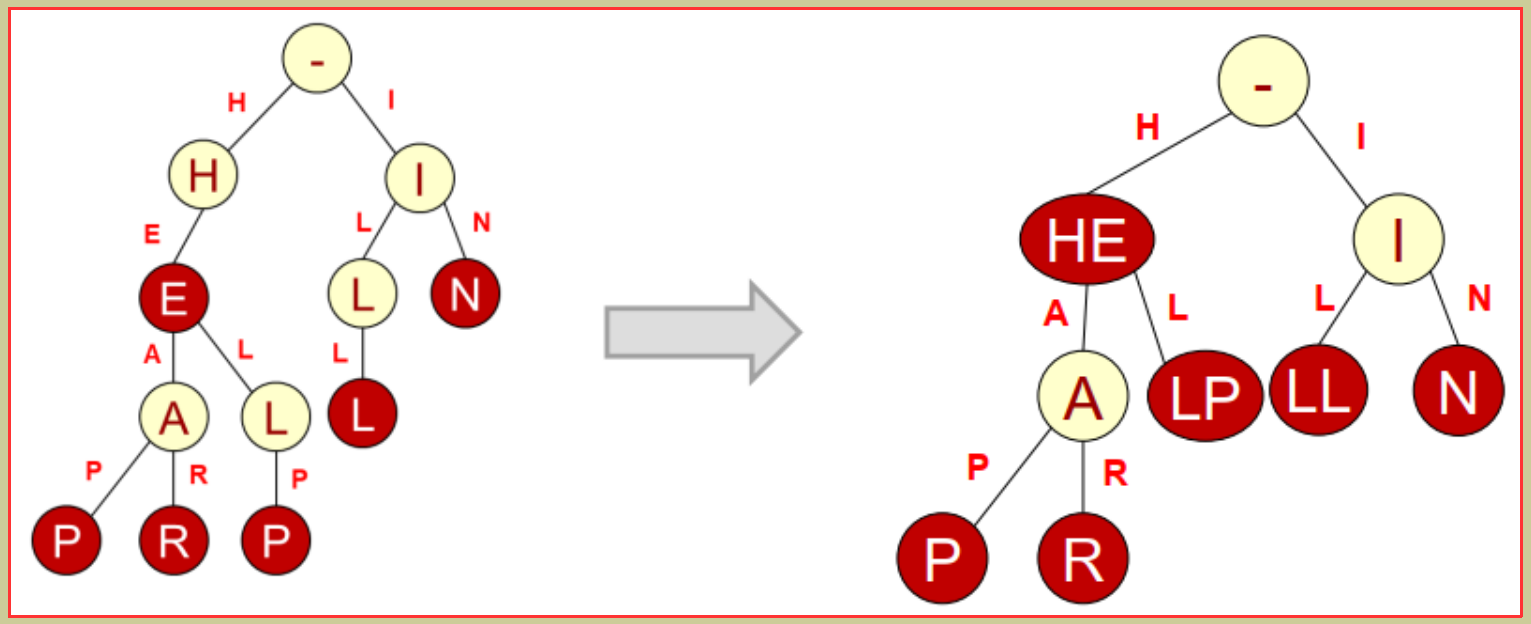
Compressed Prefix Tree

- Pode-se reduzir o número de nodos, guardando substrings em cada nodo
  - se um nodo tem apenas um filho, então combinar com ele



Compressed Prefix Tree

- Percorre a chave (string) com base no comprimento da substring no nodo atual e, de seguida, usa o próximo caráter da chave para escolher o nodo filho
- A chave não está presente
  - se os carateres da string chave se esgotarem antes da substring no nodo ou
  - se nenhum filho lhe corresponde
- Exemplos: 'H', 'HERD'



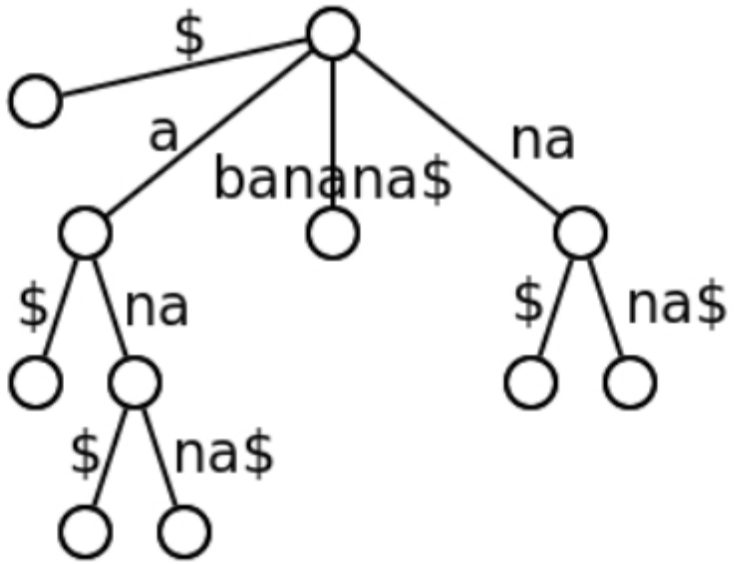
Suffix Tree

Definição

- Como a pesquisa de substrings numa Trie não é eficiente, é necessário uma estrutura de dados diferente
- Uma **Suffix Tree** ("Árvore de Sufixos") é essencialmente uma Compressed Prefix Trie de todos os sufixos de uma dada string

banana

banana$
anana$
nana$
ana$
na$
a$
$



NOTA:

$ é o símbolo usado para marcar o fim da string

Operações sobre uma Suffix Tree

- **Verificar** se uma string de tamanho n é uma substring da Suffix Tree
  - ordem de complexidade: O(n)
- **Criar** uma Suffix Tree de uma string de tamanho n
  - ordem de complexidade: O(n)
- **Inserir** mais do que uma string numa Suffx Tree:
  - uma Suffix Tree Generalizada é precisamente uma Suffix Tree de um conjunto de strings

Exemplos de utilização de Suffix Tree

- Determinar a substring de maior tamanho que se repete numa única string
  - encontrar o nó de maior profundidade por onde dois sufixos diferentes passaram
- Determinar a maior substring comum a duas strings
  - colocar as duas strings numa Suffix Tree e pesquisar o nodo com maior profundidade pelo qual ambas as substrings passaram
- Determinar a substring de tamanho k (k-gram) mais frequente
  - para todos os nodos com profundidade k, determinar qual é que tem mais folhas descendentes
- Determinar a substring única de menor tamanho
  - pesquisar o nodo de menor profundidade com apenas uma folha descendente
- ...

Suffix Arrays

Definição

- O maior problema na utilização de Suffix Tree está no facto de necessitar de muito espaço de memória (para guardar muitos dados)
- Uma alternativa muito mais eficiente, em termos de espaço de memória, para o mesmo tipo de aplicações, é o Suffix Array (“Array de sufixos”):
  - um array ordenado de todos os sufixos

Exemplo

- Considerar S = “banana”

i	1	2	3	4	5	6	7
s[i]	b	a	n	a	n	a	$

Sufixos de S

Sufixo	**i**
banana$	1
anana$	2
nana$	3
ana$	4
na$	5
a$	6
$	7

Sufixos de S ordenados

Sufixo	**i**
$	7
a$	6
ana$	4
anana$	2
banana$	1
na$	5
nana$	3

- O Suffix Array A contém as posições iniciais daqueles sufixos ordenados

i	1	2	3	4	5	6	7
A[i]	7	6	4	2	1	5	3

Operações num Suffix Array

- Como se verifica se uma string P é uma substring de um texto T?
- Pode-se usar a pesquisa binária no Suffix Array de T?
- Sem estruturas de dados auxiliares
  - cada comparação tem um custo de O(|P|) e
  - é necessário fazer-se O(log |T|) comparações,
  - conduzindo a um algoritmo de complexidade de O(|P| x log |T|)