UNISINOS

Estruturas Avançadas de Dados I
(Árvores TRIE)

Prof. Gilberto Irajá Müller

Última atualização 11/09/2017

Introdução

- Desenvolvida em 1960 por Edward Fredkin;

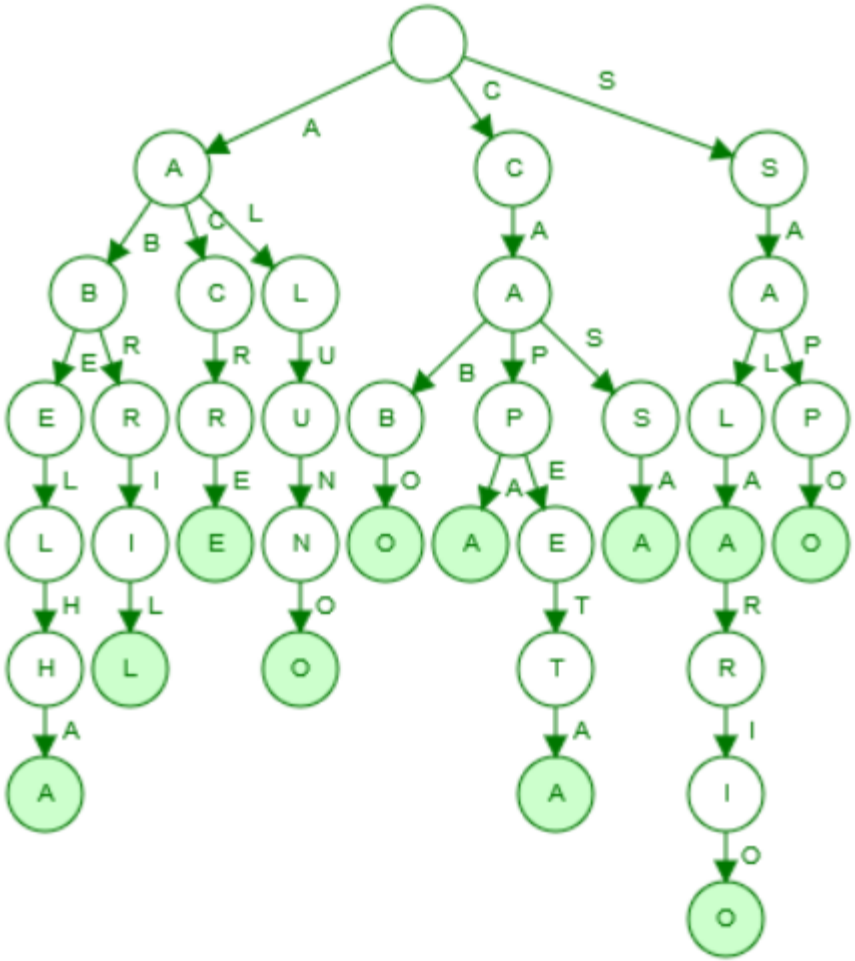
- O nome "TRIE" vem de Re**trie**val (recuperação de dados);

- Pronuncia-se "TRAI" ou "TRI" para distinguir de "Tree";

- **Ideia geral:** usar partes das CHAVES como caminho busca.

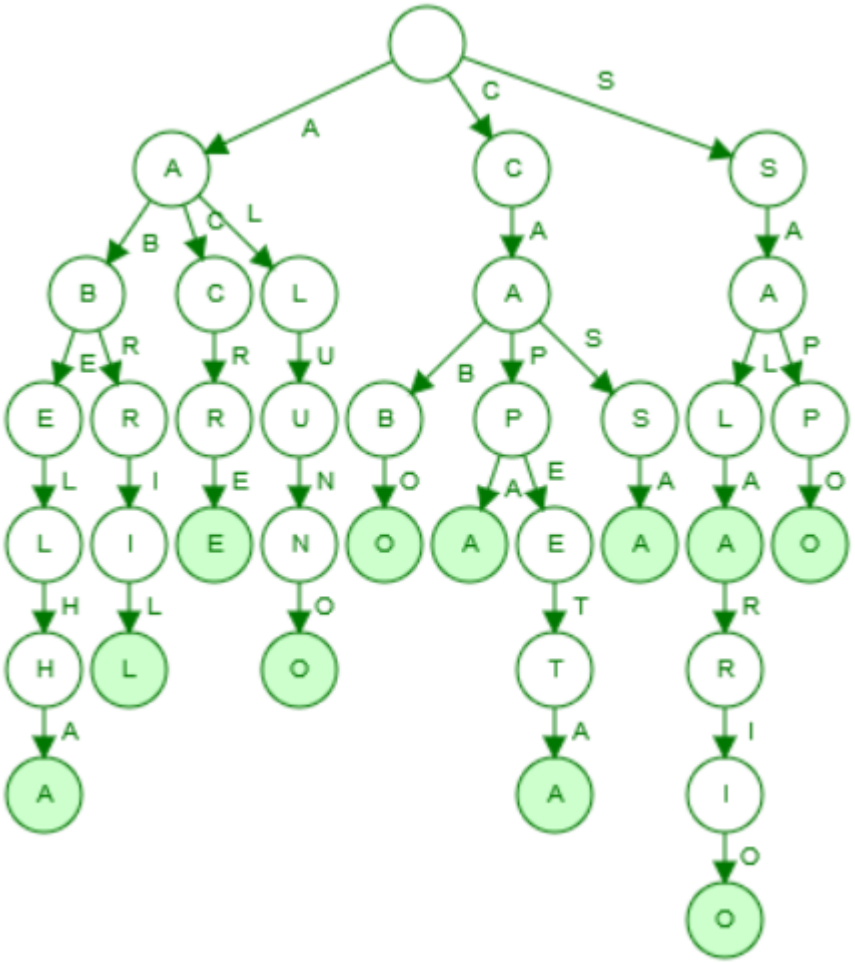
Características

- Árvore é **ordenada** e **n-ária**;



Características

- Chave em **geral caracteres**: {A, B, C, D, ...}, porém, podemos ter toda a tabela ASC.

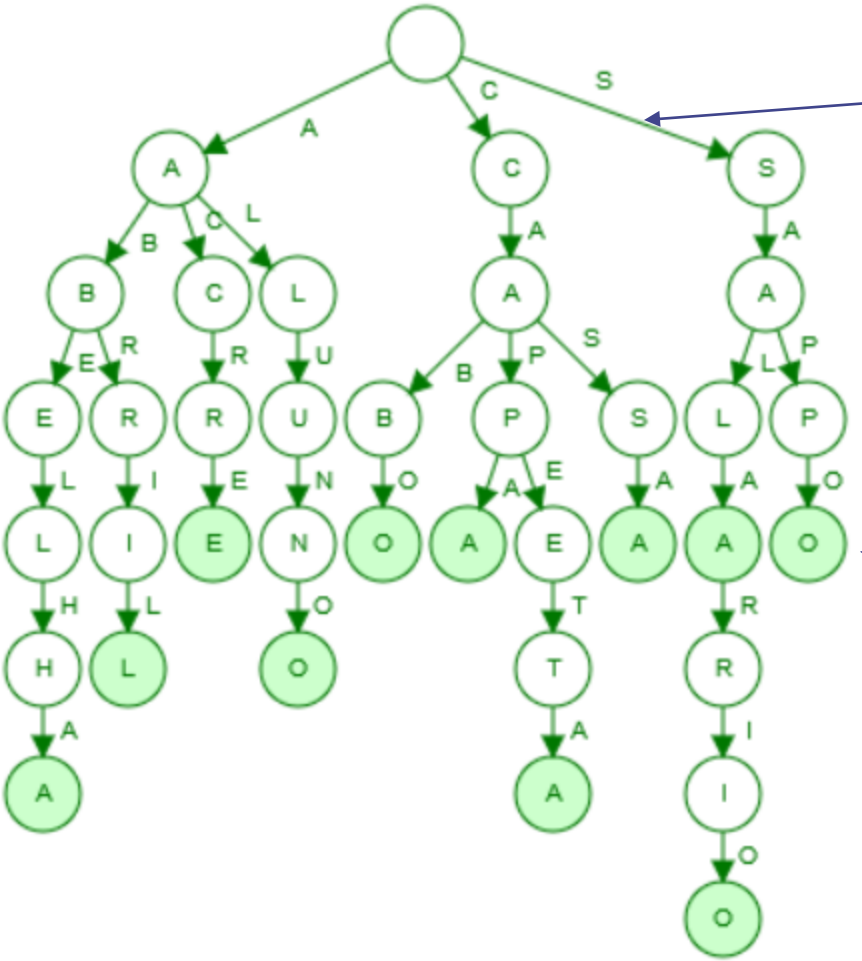


Chaves:

ABELHA
ABRIL
ACRE
ALUNO
CABO
CAPA
CAPETA
CASA
SALA
SALARIO
SAPO

Características

- Os caracteres são **armazenados no nó** (não há uma chave);
- Cada nó tem ***R* filhos**, um para cada possibilidade de caractere.



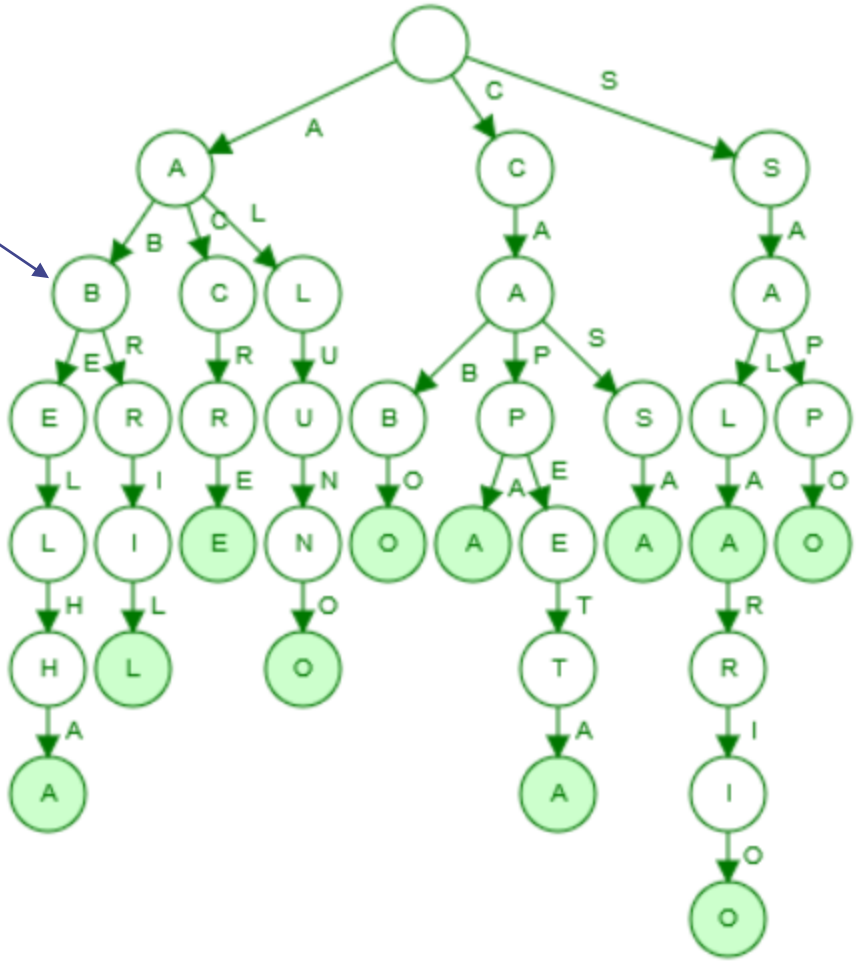
Link com todas as chaves que iniciam com "S".

Uso de uma terminação para indicar que ali tem uma palavra (chave).

Características

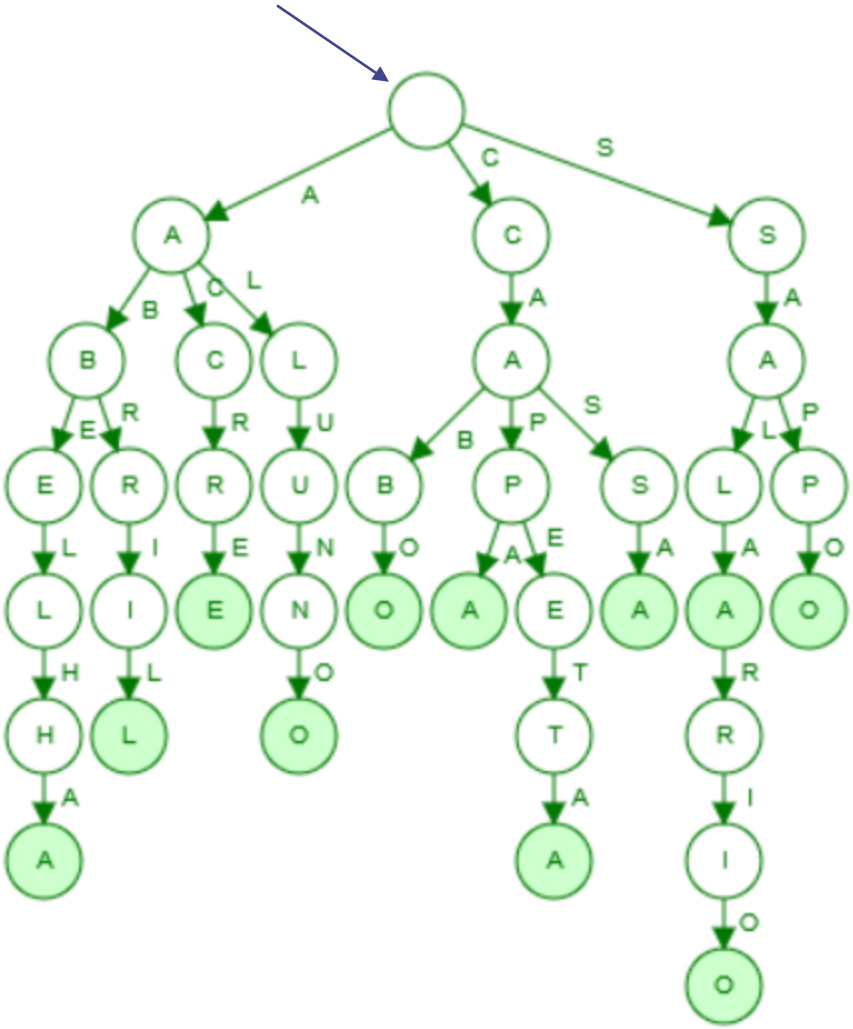
- **Descendentes** do mesmo nó com mesmo **prefixo.**

ABELHA
ABRIL



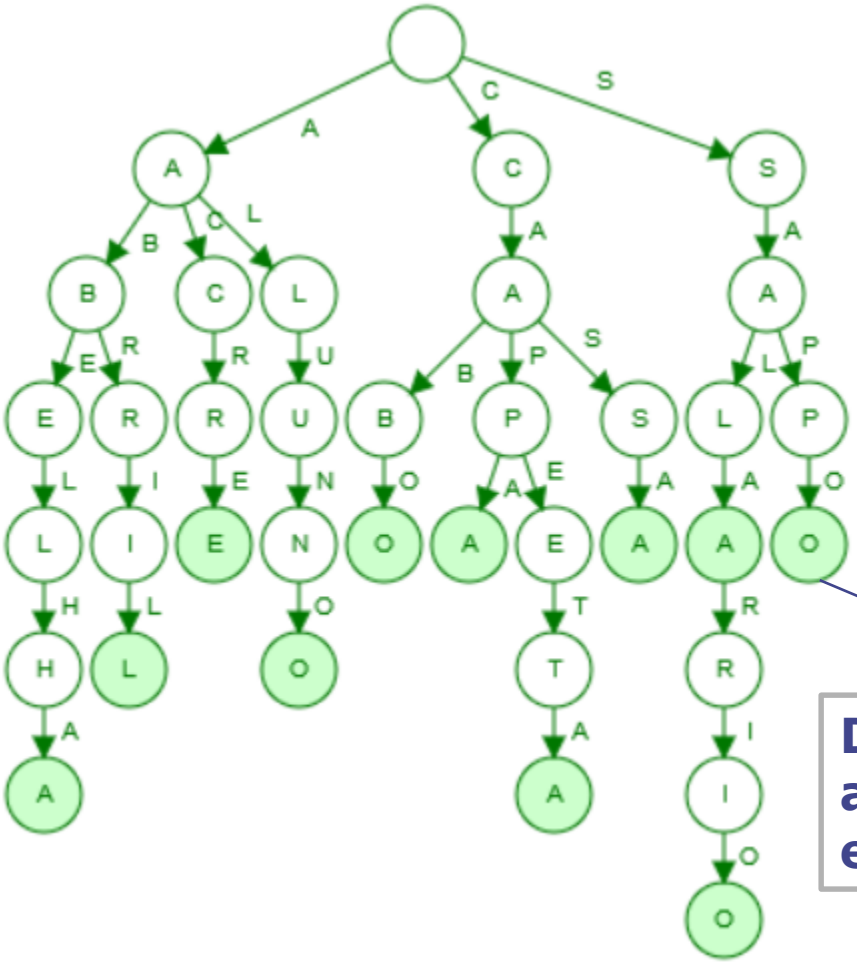
Características

- **Raiz:** cadeia vazia



Características

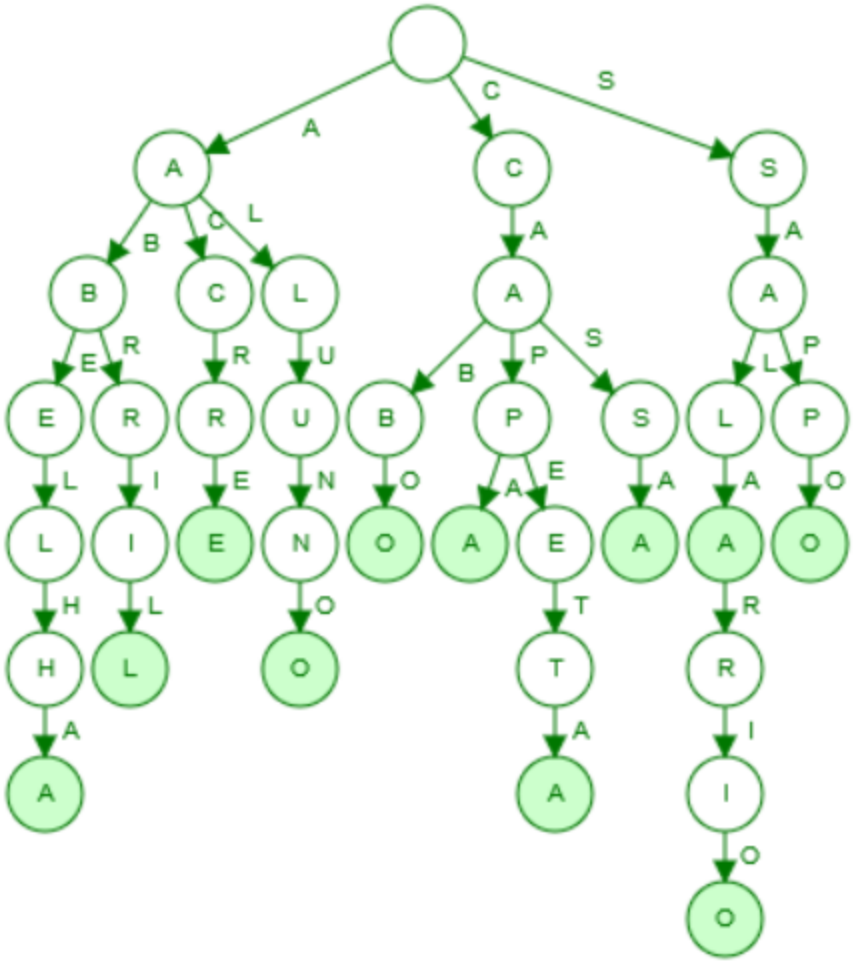
- Valores ou elementos associados a **folhas** ou a alguns **nós internos** de interesse.



Definição: comum aos anfíbios anuros em geral, ...

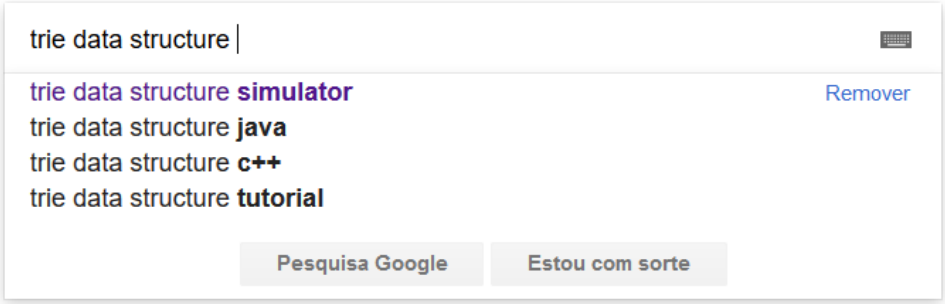
Características

- O **comprimento** corresponde ao tamanho do **alfabeto** e pode ser visto como um autômato finito. Cada **nível** que se desce corresponde a avançar um elemento na chave.



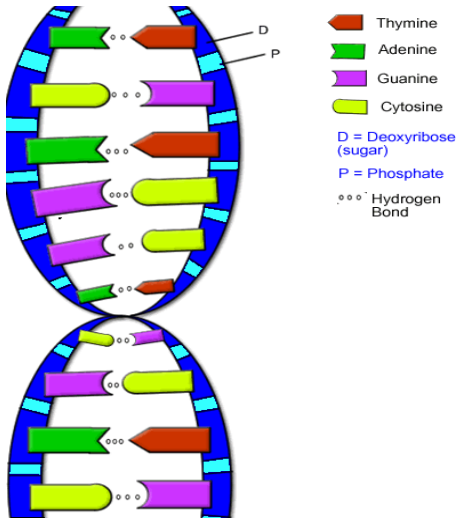
Aplicações

- Dicionários (telefone celular);
- Corretores Ortográficos;
- Autopreenchimento:
  - browsers,
  - e-mail,
  - linguagens de programação.



Aplicações

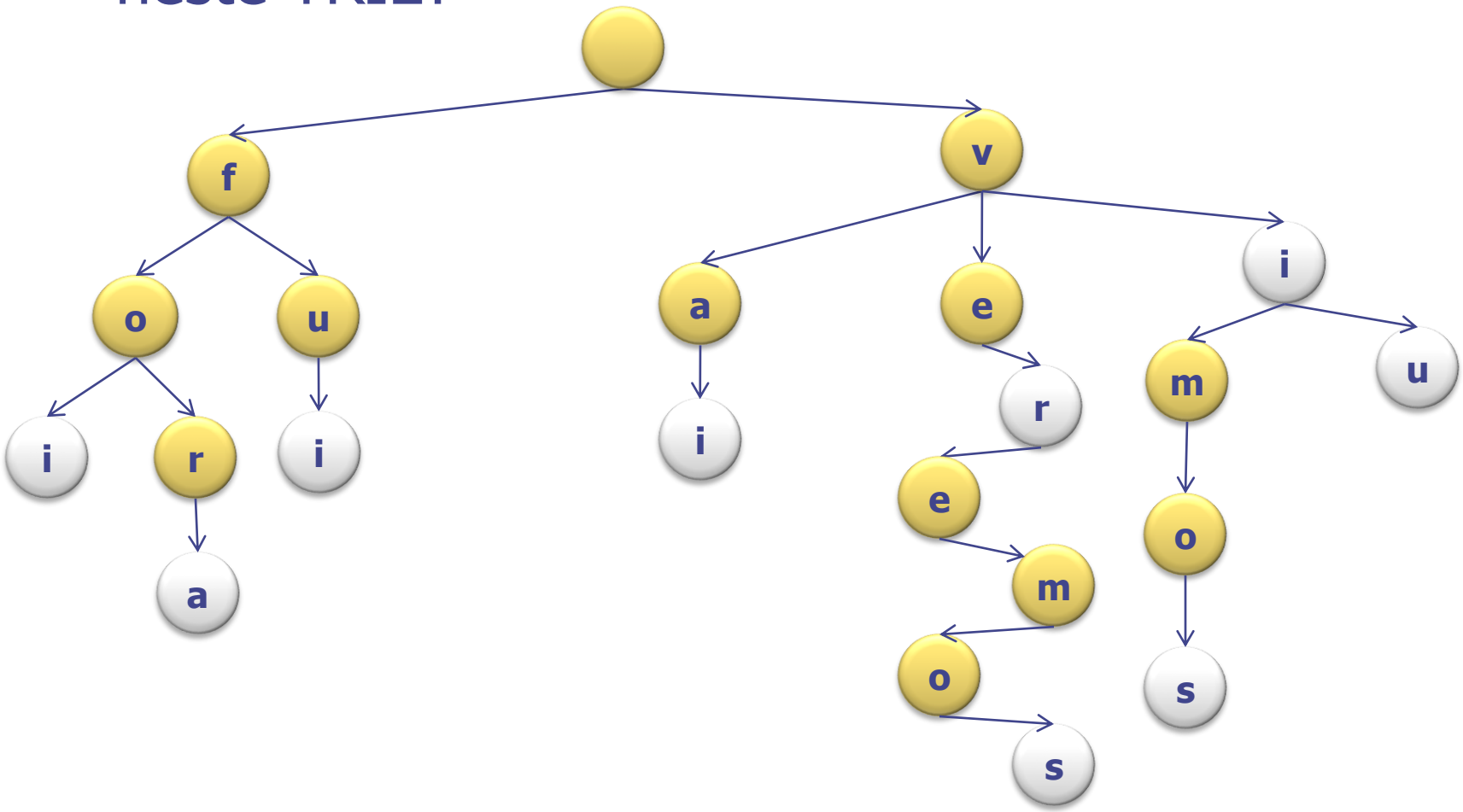
- Compressão de dados;
- Biologia computacional;



- Tabelas de roteamento para endereços IP;
- Armazenar e consultar documentos XML;
- Fundamental para o Burstsort (o método mais rápido de ordenação de strings em memória/cache);
- Tabelas de símbolos em compiladores.

Exercício 1

- Quais chaves/palavras estão representadas neste TRIE?



Interface de um TRIE

```
public interface TrieADT<V> {
    public void clear();
    public boolean isEmpty();
    public V search(String key);
    public void insert(String key, V value);
    public void delete(String key);
    public Iterable<String> keysWithPrefix(String prefix);
}
```

Estrutura de um TRIE (R-Way)

```
public class WayTrie<V> implements TrieADT<V> {
  private static final int R = 256; // extended ASCII

  private Node root;

  private static class Node {
    private Object value;
    private Node[] next = new Node[R];
  }

  @Override
  public void clear() {
    root = null;
  }

  @Override
  public boolean isEmpty() {
    return root == null;
  }

  @Override
  public V search(String key) {
    Node node = search(root, key, 0);
    if (node == null)
      return null;
    return (V) node.value;
  }
// Continua
```

Algumas implementações consideram apenas 26 caracteres (alfabeto). Chama-se 26-Way Tree.

Muitos "nulls" nos nós.

```
private Node search(Node node, String key, int index) {
  if (node == null)
     return null;
  if (index == key.length())
     return node;
  char c = key.charAt(index);
  return search(node.next[c], key, index + 1);
}
```

Operação de Inserção

- Faz-se uma busca pela chave a ser inserida. Se ela já existir no TRIE, então, finaliza-se a operação;

- Caso contrário, é recuperado o nó até onde ocorre a maior substring (igualdade) da palavra a ser inserida;

- O restante dos seus caracteres são adicionados no TRIE a partir daquele nó, inserindo o valor no nó final.

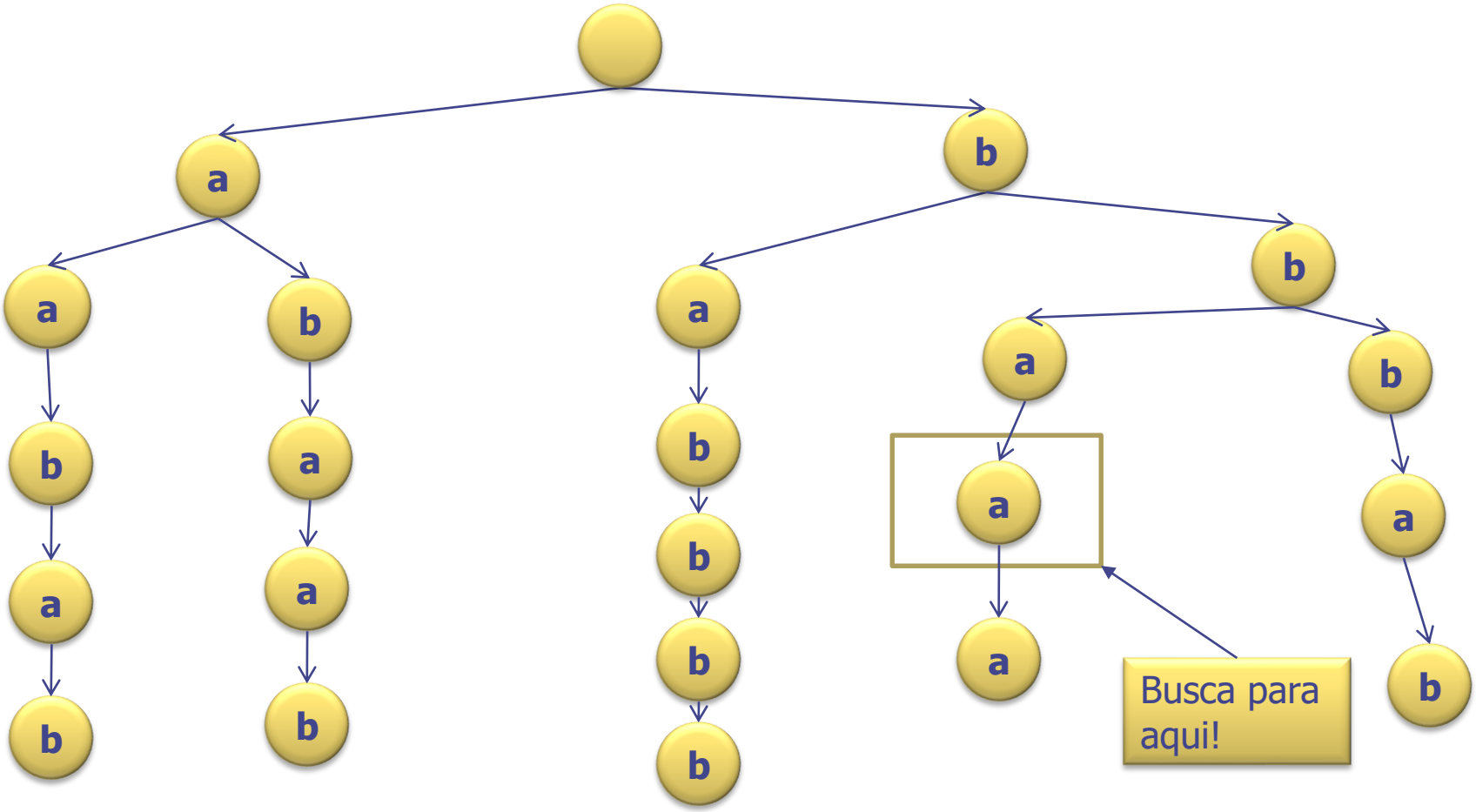
Estrutura de um TRIE (Inserção)

```java
@Override
public void insert(String key, V value) {
  root = insert(root, key, value, 0);
}

private Node insert(Node node, String key, V value, int index) {
  if (node == null)
    node = new Node();
  if (index == key.length()) {
    node.value = value;
    return node;
  }
  char c = key.charAt(index);
  node.next[c] = insert(node.next[c], key, value, index + 1);
  return node;
}
```

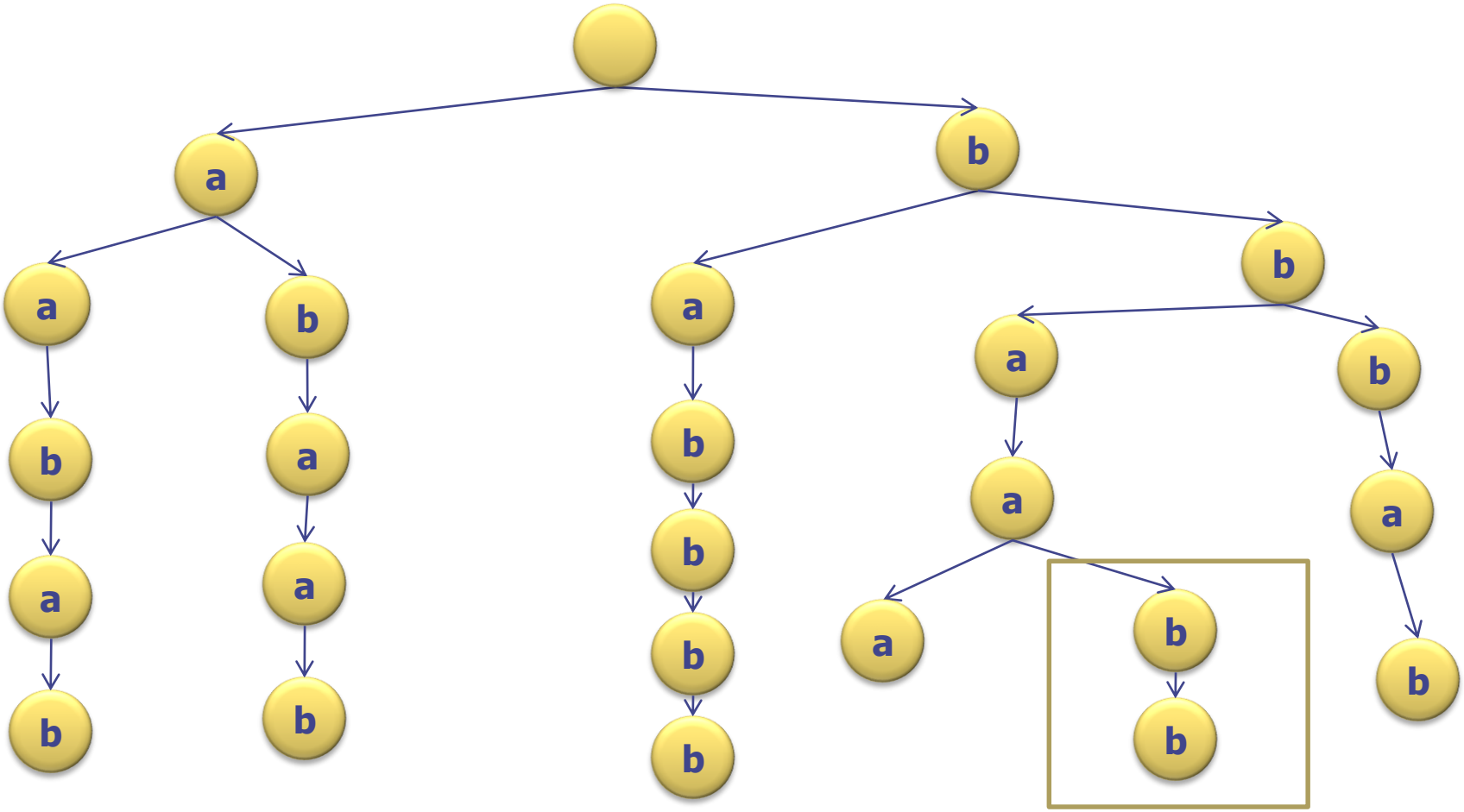
Operação de Inserção (cont.)

- Inserção de bbaabb



Operação de Inserção (cont.)

- Inserção de bbaabb



Exclusão

- Busca-se o **nó** que representa o final da palavra a ser excluída;
- São excluídos os nós que possuem apenas **um filho** pelo caminho ascendente;
- A exclusão é concluída quando se encontra um nó com **mais de um** filho.

Estrutura de um TRIE (Exclusão)

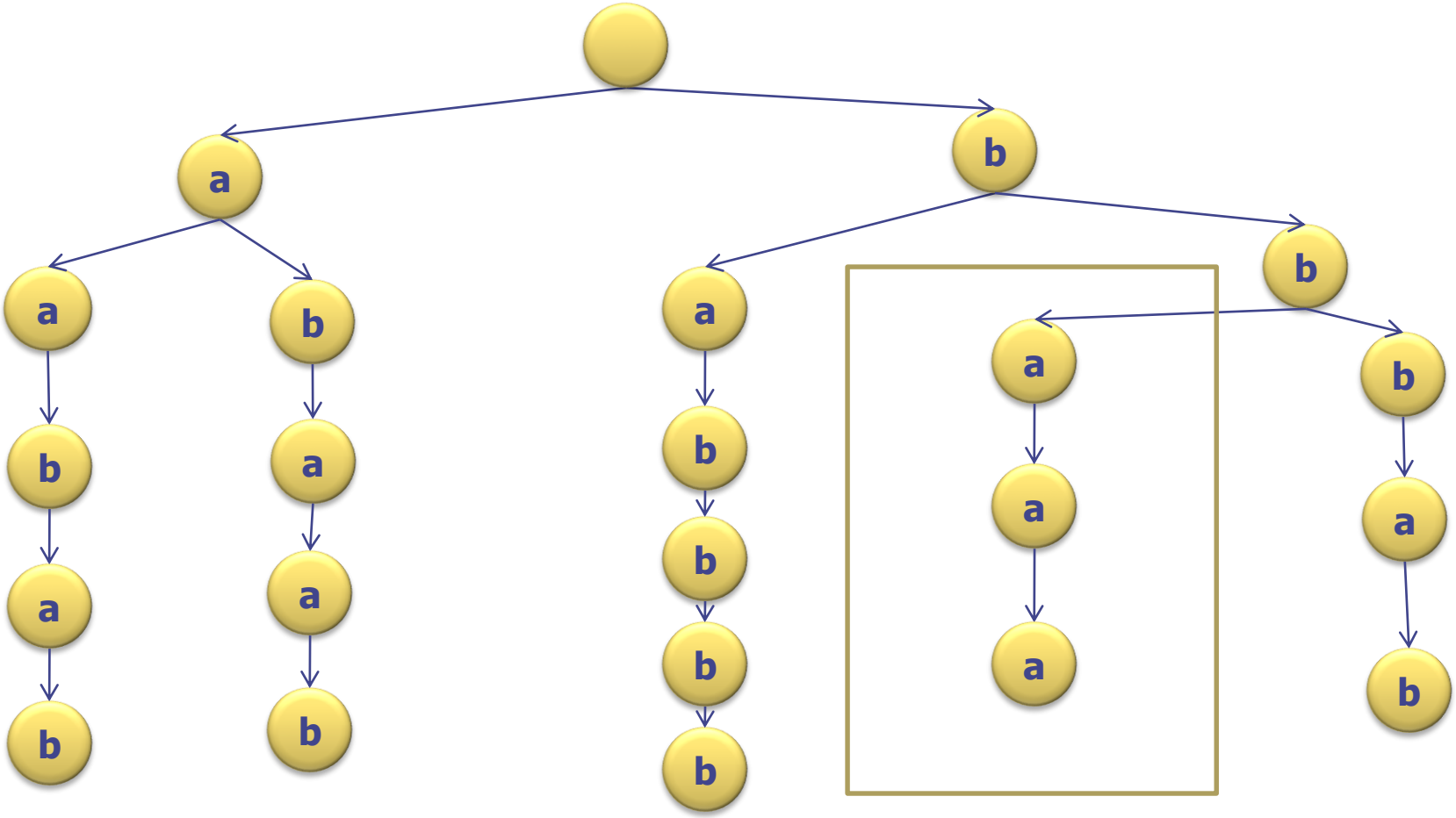
```java
@Override
public void delete(String key) {
  root = delete(root, key, 0);
}

private Node delete(Node node, String key, int index) {
  if (node == null)
     return null;
  if (index == key.length()) {
     node.value = null;
  } else {
     char c = key.charAt(index);
     node.next[c] = delete(node.next[c], key, index + 1);
  }

  if (node.value != null)
    return node;
  for (int c = 0; c < R; c++)
    if (node.next[c] != null)
       return node;
  return null;
}
```

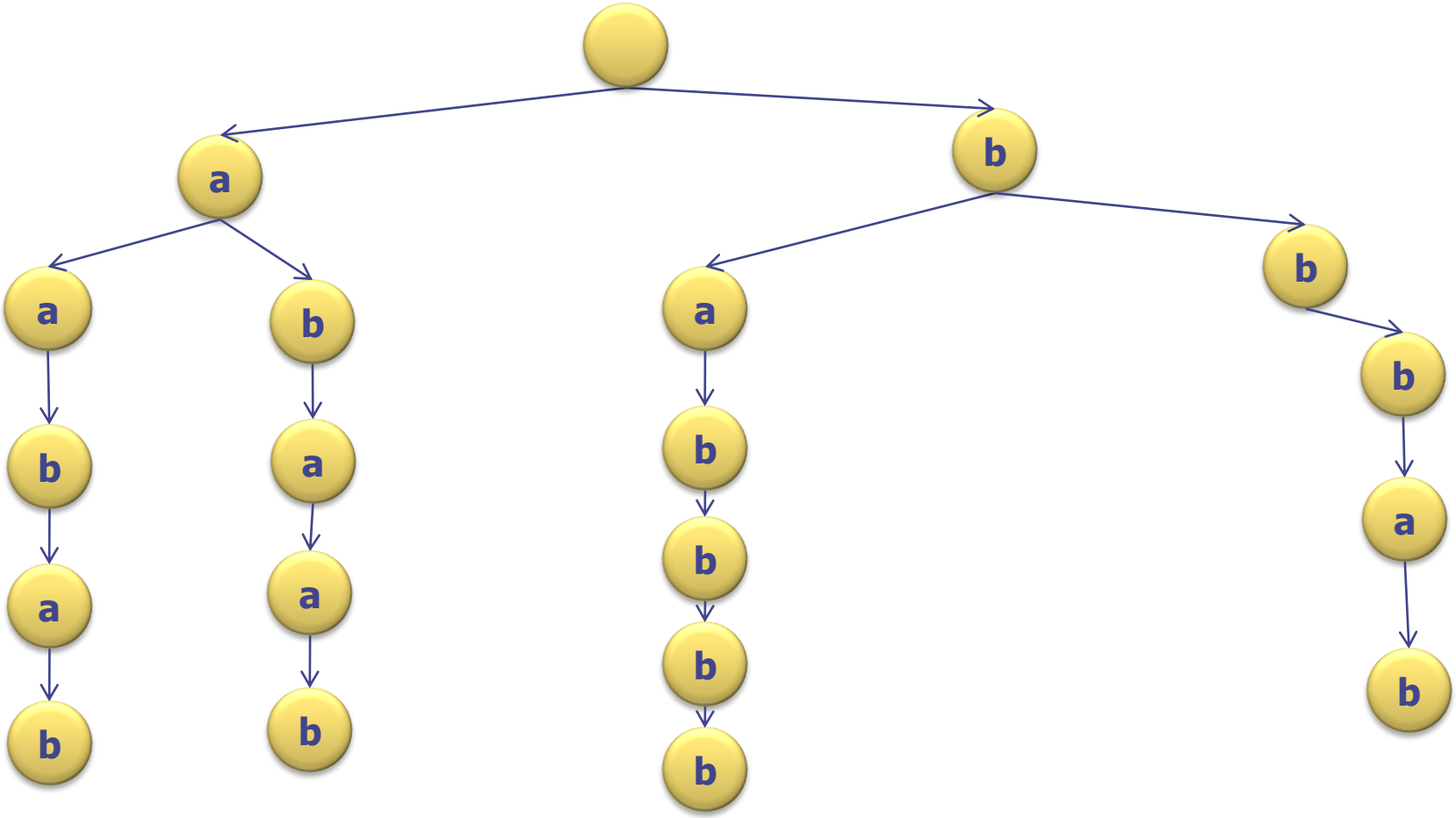
Exclusão (cont.)

- Exclusão de bbaaa



Exclusão (cont.)

- Exclusão de bbaaa



Estrutura de um TRIE (cont.)

```
@Override
public Iterable<String> keysWithPrefix(String prefix) {
  Queue<String> results = new LinkedList<>();
  Node node = search(root, prefix, 0);
  collect(node, new StringBuilder(prefix), results);
  return results;
}

private void collect(Node node, StringBuilder prefix, Queue<String> results) {
  if (node == null)
      return;
  if (node.value != null)
      results.add(prefix.toString());
  for (char c = 0; c < R; c++) {
      prefix.append(c);
      collect(node.next[c], prefix, results);
      prefix.deleteCharAt(prefix.length() - 1);
  }
}
```

Complexidade

- A **altura** da árvore é igual ao **comprimento da chave mais longa**; o tempo de execução das operações não depende do número de elementos da árvore;

- Complexidade no pior caso é O(AK)
  - A = tamanho do alfabeto
  - K = tamanho da chave

- A utilização de um TRIE só compensa se o acesso aos componentes individuais das chaves for bastante rápido. Quanto maior a estrutura mais eficiente o uso do espaço.

UNISINOS

Tipos de TRIES

- Existem muitas variantes e tipos de TRIES
  - R-WAY (exemplo em Java)
  - TST (Ternary Search Trie)
  - DST (Digital Search Tree)
  - Suffix Tree
  - PATRICIA Tree
  - DAWG (Directed Acyclic Word Graph)
  - Entre outros

TRIE como Auto Preenchimento

- Aplicação usual de TRIE é o **auto preenchimento;**

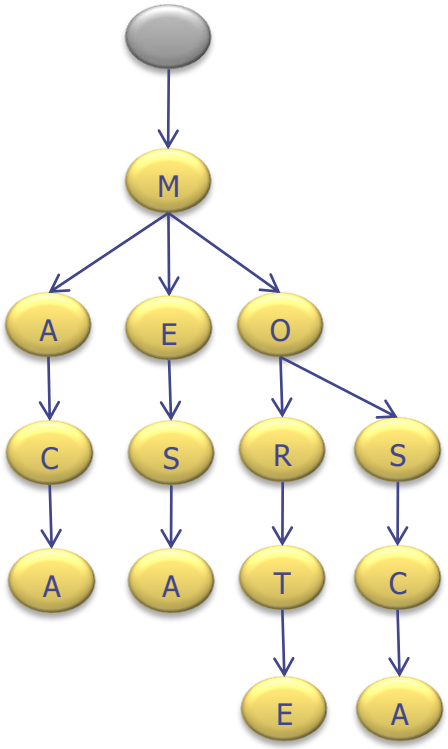
- Nesse tipo de aplicação as palavras digitadas são comparadas com um dicionário armazenado em arquivo e, a cada letra digitada, é sugerido um conjunto de palavras.

TRIE como Auto Preenchimento (cont.)

- Supondo que iremos digitar a palavra "MOSCA".

M O S C A

Sugestões:
MACA
MESA
MORTE
MOSCA

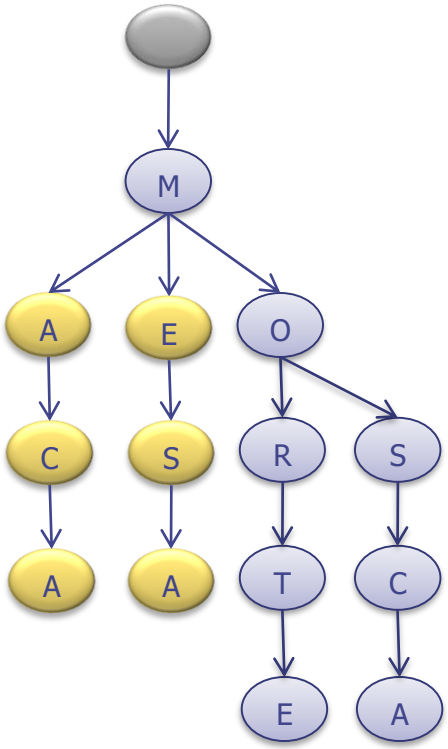


TRIE como Auto Preenchimento (cont.)

- Supondo que iremos digitar a palavra "MOSCA".

M O S C A

Sugestões:
MORTE
MOSCA

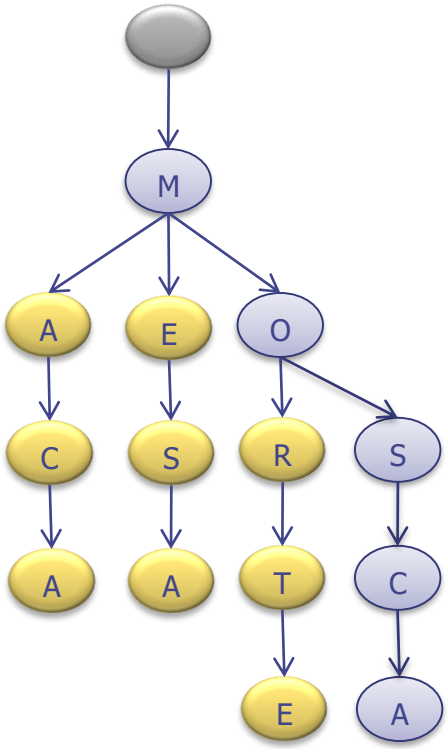


TRIE como Auto Preenchimento (cont.)

- Supondo que iremos digitar a palavra "MOSCA".

M O S C A

Sugestões:
MOSCA



TRIE como Corretor Ortográfico

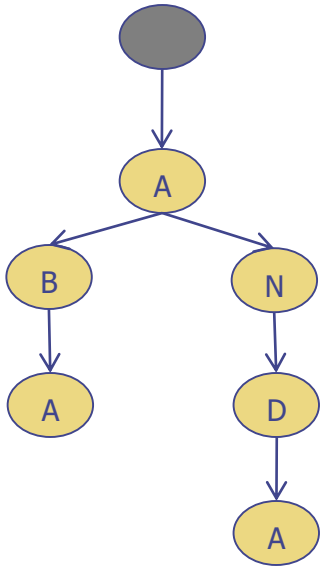
- **Busca:** localizar um dado que corresponde a chave informada;
- **Problema:** Considerando um cadastro de pessoas onde temos nomes com grafias semelhantes (Manuel/Manoel, Elaine/Elayne, Luis/Luiz), podem ocorrer erros na entrada desses dados, ou seja, possível erro de digitação;
- **Solução do problema:** existe um método de busca por **aproximação** de correspondência, onde podemos localizar dados que são semelhantes a uma chave informada. Pela estrutura de representação de caractere a caractere usada nos TRIEs, acaba-se tendo um desempenho muito bom nesse tipo de aplicação.

Técnicas

- **Substituição**: avança um caractere na chave e avança um nível na árvore;
- **Exclusão:** avança um nível na árvore;
- **Inserção**: avança um caractere na chave;
- **Transposição:** avança um nível na árvore testando a posição atual da chave, se coincidir, avança um caractere na chave e retrocede um nível na árvore para confirmar a inversão.
- Obs.: algumas exceções quando é o último caractere da árvore.

Técnica de Substituição

- Ao digitar a palavra ADA, será verificado se existe no dicionário através do TRIE conforme abaixo:



A – A (ok)

B – D (erro)

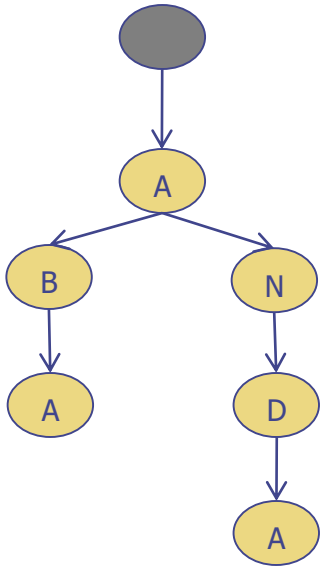
N – D (erro)

A – A (ok). **Ocorre a regra de substituição, pois avança no nível da árvore e na chave.**

Como o último elemento está na folha e, na outra subárvore não coincide, então, pela regra de substituição, será sugerido a palavra **ABA (substituição do D pelo B)**. O algoritmo pode parar aqui ou continuar utilizando outras regras.

Técnica de Exclusão

- Ao digitar a palavra ADA, será verificado se existe no dicionário através do TRIE conforme abaixo:



A – A (ok)

B – D (erro)

N – D (erro)

A – D (erro). Avança somente no nível da árvore.

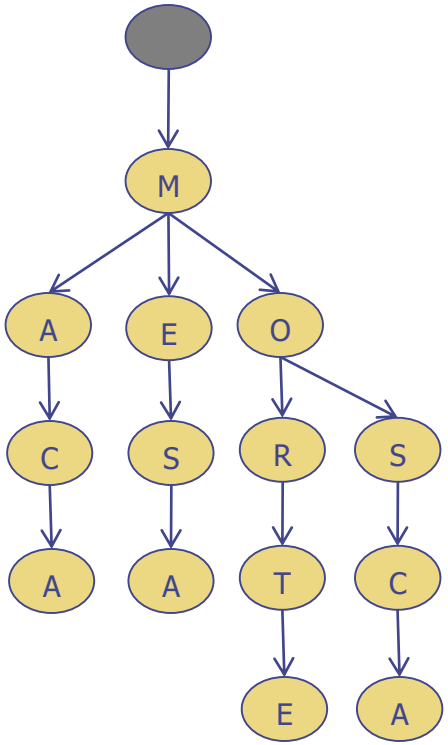
D – D (ok). Avança somente no nível da árvore.

A – A (ok). Avança somente no nível da árvore.

Detectado erro de exclusão, onde a letra "N" foi suprimida da chave. Neste caso, será sugerido a palavra **ANDA**.

Técnica de Inserção

- Ao digitar a palavra MEDSA, será verificado se existe no dicionário através do TRIE conforme abaixo:



M – M (ok)

A – E (erro)

E – E (ok)

S – D (erro). **Detecta a inserção.**

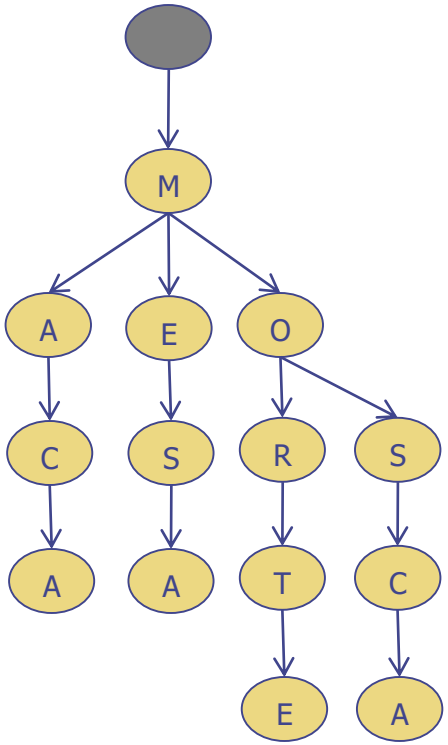
S – S (ok)

A – A (ok)

Detectado erro de inserção (avança um caractere na chave), em virtude da letra "D". Neste caso, será sugerido a palavra **MESA**.

Técnica de Transposição

- Ao digitar a palavra MSEA, será verificado se existe no dicionário através do TRIE conforme abaixo:



M – M (ok)

A – S (erro)

E – S (erro)

O – S (erro)

C – S (erro)

S – S (ok)

E – E (ok). **Retrocede para verificar a transposição.**

A – A (ok). Avança dois níveis para verificar o fim da palavra. Encontrará **MESA**.

Estrutura da Ternary Search TRIE

```
public class TernaryTrie<V> implements TrieADT<V> {
  private Node<V> root;

  private static class Node<V> {
    private char c;
    private Node<V> left, middle, right;
    private V value;
  }

  @Override
  public void clear() {
    root = null;
  }

  @Override
  public boolean isEmpty() {
    return root == null;
  }

  @Override
  public V search(String key) {
    Node<V> node =  search(root, key, 0);
    if (node != null) return node.value;
    return null;
  }
// Continua
```

```
private Node<V> search(Node<V> node, String key, int index) {
  if (node == null) return null;

  char c = key.charAt(index);
  if (c < node.c) return search(node.left, key, index);
  else if (c > node.c) return search(node.right, key, index);
  else if (index < key.length() - 1)
    return search(node.middle, key, index + 1);
  else return node;
}
```

Estrutura da Ternary Search TRIE (cont.)

```
@Override
public void insert(String key, V value) {
   if (search(key) == null) {
     root = insert(root, key, value, 0);
   }
}

private Node<V> insert(Node<V> node, String key, V value, int index) {
  char c = key.charAt(index);
  if (node == null) {
     node = new Node<V>();
     node.c = c;
  }
  if (c < node.c) node.left = insert(node.left, key, value, index);
  else if (c > node.c) node.right = insert(node.right, key, value, index);
  else if (index < key.length() - 1) node.middle = insert(node.middle, key, value, index + 1);
  else node.value = value;

  return node;
}
```

Estrutura da Ternary Search TRIE (cont.)

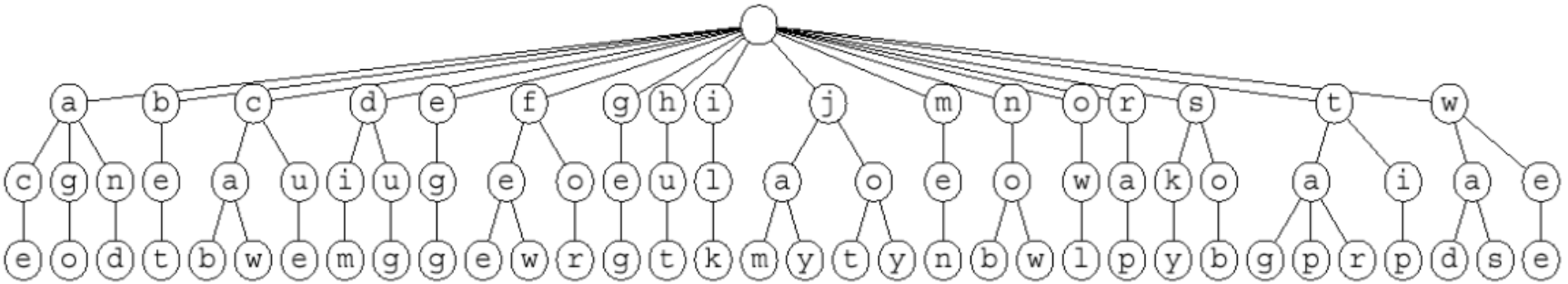
```
@Override
public Iterable<String> keysWithPrefix(String prefix) {
  Queue<String> queue = new LinkedList<>();
  Node<V> node = search(root, prefix, 0);
  if (node == null) return queue;
  if (node.value != null) queue.add(prefix);
  collect(node.middle, new StringBuilder(prefix), queue);
  return queue;
}

private void collect(Node<V> node, StringBuilder prefix, Queue<String> queue) {
  if (node == null) return;
  collect(node.left,  prefix, queue);
  if (node.value != null) queue.add(prefix.toString() + node.c);
  collect(node.middle, prefix.append(node.c), queue);
  prefix.deleteCharAt(prefix.length() - 1);
  collect(node.right, prefix, queue);
}
```

Obs.: O contrato delete não tem implementação!

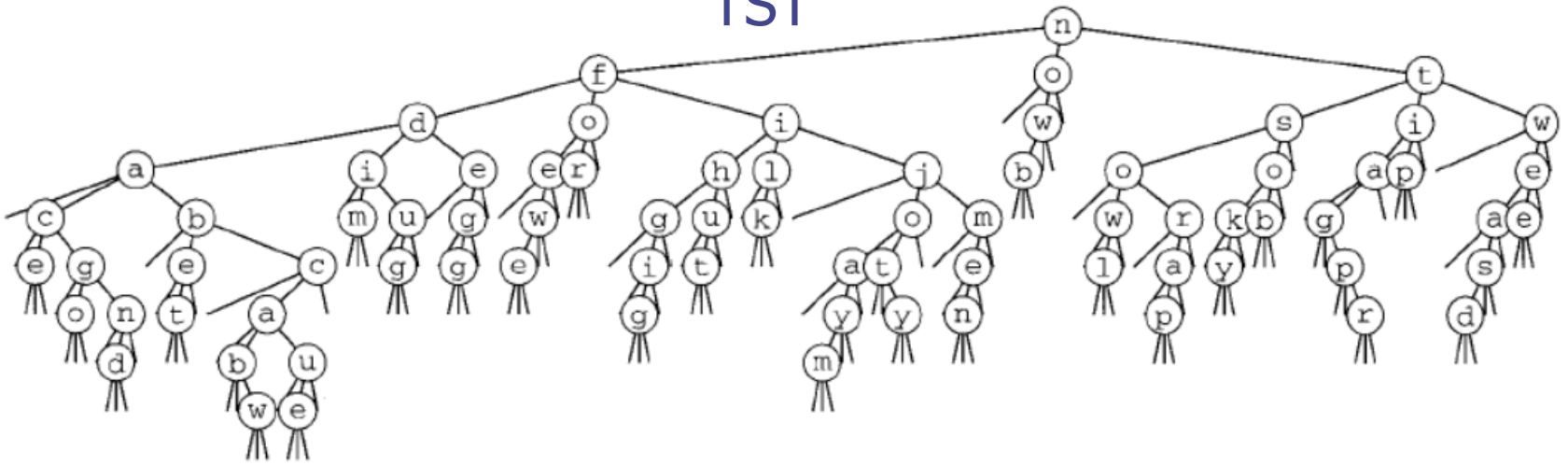
26-Way Trie vs. TST

26-Way Trie



26 links "nulls" em cada folha. 1035 links "nulls" não mostrados.

TST



3 links "nulls" em cada folha. 155 links "nulls".

Exercício Prático

Com base na classe WayTrie, desenvolva os seguintes contratos:

Contrato	Considerações
countKeysWithPrefix(String prefix)	**Parâmetro**: prefixo que filtra as palavras derivadas. **Retorno**: número de palavras encontradas com base no prefixo. Caso não haja palavra com o prefixo, retornar 0.
String longestPrefixOf(String key)	**Parâmetro**: chave a ser verificada. **Retorno**: o maior prefixo correspondente à chave.
Iterable<String> keysByPattern(String pattern)	**Parâmetro**: padrão para filtrar as palavras. Para inserir um caractere coringa, utilize o ponto ('.') dentro de um padrão. Ex.: trie.keysByPattern("v..") retornará todas as palavras que iniciam com "v" e que possuem três caracteres. Poderá usar combinações com o caractere coringa. Ex.: trie.keysByPattern("..."). O padrão define as combinações e o tamanho. **Retorno**: Iterador com as palavras encontradas. **Dica**: use como base o mesmo método keysWithPrefix(String prefix).

O último contrato é opcional. Portanto, somente fazer se o grupo deseja se aprofundar no tema.

Exercícios Teóricos

Exercício 2. Demonstre a árvore TRIE (passo-a-passo) para as seguintes chaves:

MACA

MACHO

MATO

BALA

BANANA

BALELA

BALEIA

Exercícios Teóricos (cont.)

- **Exercício 3.** Como seria um **corretor ortográfico** ao procurar a chave MALA e MATA, referente o TRIE anterior?

- **Exercício 4.** Qual seria o resultado utilizando uma árvore TRIE de **autopreenchimento** baseado no TRIE do exercício 2? A partir das digitações abaixo.

  B

  BA

  BAL

  BALE

  BALEL

Simulador

https://www.cs.usfca.edu/~galles/visualization/Trie.html

Referências Bibliográficas

- CORMEN, Thomas H. et al. **Introduction to algorithms.** 3. ed. Cambridge: MIT, 2009. xix. 1292 p.

- http://algs4.cs.princeton.edu/52trie/. Acessado em 16/11/2016.