

# Relatório APS ED2

Henriko Inácio Alberton

Michel Augusto de Souza

*Universidade Tecnológica Federal do Paraná – UTFPR*

*COCIC – Coordenação do Curso de Bacharelado em Ciência da Computação  
Campo Mourão, Paraná, Brasil*

## 1. Introdução

Neste trabalho implementamos uma Radix Tree com uma aplicação de indexação de texto, no item 2, demonstrar como ficaria a inserção de algumas palavras na Radix Tree e explicar a o funcionamento dela e as utilidades da mesma. No item 3 iremos explicar a aplicação feita, o funcionamento dela e um teste de conformidade. No item 4 iremos explicar o algoritmo de Aho-Corasick e Similaridade utilizando Trigrams. No item 6 temos como a nossa aplicação é executada e um teste de conformidade.

## 2. Radix Tree

Uma Radix Tree é uma Trie melhorada por assim dizer, uma Trie é uma estrutura do tipo árvore que guarda dentro dela uma palavra, sendo que cada nó é uma letra da palavra, e palavras similares tem o mesmo prefixo, como visto na imagem abaixo.

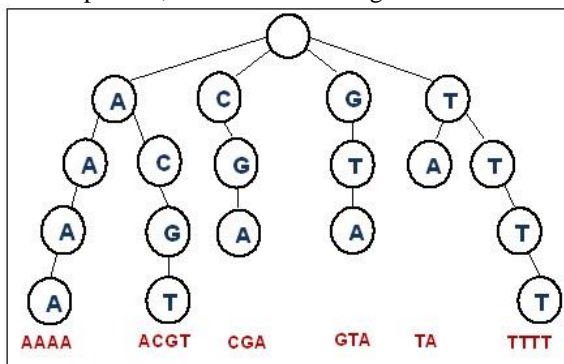


Figura 1 - Visualização de uma Trie

Já uma Radix Tree utiliza um conceito parecido, porém Radix significa base, ou seja, é uma trie com bases diferentes, consequentemente, ela guarda um prefixo da palavra, independente de qual seja o seu tamanho, e o nó subsequente, vai determinar o restante

da palavra(ou mais prefixos), ou caso o prefixo seja uma palavra, a árvore guarda uma chave que diz se o nó é palavra ou não. Assim como mostrado na imagem abaixo, onde “google.com/” é tanto um nó prefixo, quanto um nó palavra.

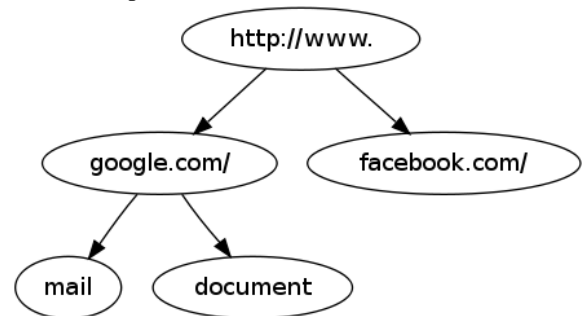


Figura 2 - Visualização de uma Radix Tree

Uma das utilidades da Radix Tree é a indexação de um texto, que foi a nossa aplicação escolhida para implementar. Porém, tal estrutura pode ser utilizada para muitas outras finalidades, como por exemplo sugestão de palavras, busca a partir do prefixo, etc.

## 3. Aplicação

A aplicação escolhida para implementar foi uma não tão fácil, como descrito no documento que o professor disponibilizou.

Primeiramente, o nosso algoritmo lê um arquivo de texto, que por convenção, normalmente nas tries é chamado de dicionário, porém no nosso programa é chamado de entrada, e insere todas as palavras maiores de 2 letras em nossa Radix Tree, ignorando qualquer caractere especial que não seja do alfabeto latino (ou seja, nosso algoritmo inclui palavras acentuada), e também armazena na estrutura a linha em que tal palavra se encontra, essa parte do algoritmo foi implementada utilizando expressões regulares.

A visualização da árvore pode ser vista na imagem abaixo.

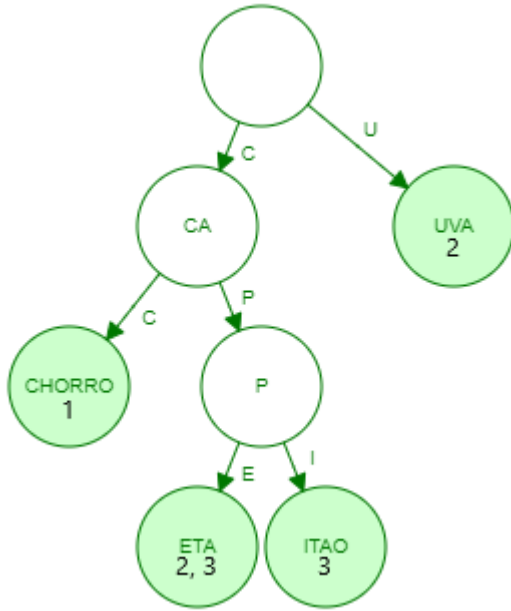


Figura 3 - Visualização da inserção do texto

Logo abaixo, temos o texto de entrada, o número da linha foi colocado no texto apenas para ser mais fácil de visualizar, o algoritmo conta a quantidade de linhas automaticamente.

```
1 - cachorro
2 - uva, capeta 1230851 uh ab
3 - Capita0, ./=lfd CaPeTa
```

Figura 4 - Texto de Inserção

Tendo então agora todas as palavras de entrada dispostas na árvore, nosso algoritmo imprime em um arquivo de texto chamado out.txt todo o conteúdo da árvore, de forma esporádica, ou seja, a sequência não importa muito, porém esse arquivo de texto vai ser utilizado logo em seguida para salvar a entrada em ordem alfabética.

Para disponibilizar os dados em forma alfabética, é utilizado uma árvore rubro-negra (mais especificamente, uma implementação em Java dela que é a TreeMap) que por sua natureza, tende a manter os dados de forma um pouco ordenada, por assim dizer. A impressão dos dados em ordem alfabética ocorre praticamente de forma natural (impressão em ordem), já que a árvore se balanceia. Os dados são gravados em um arquivo chamado output.txt.

Após o arquivo de output ser escrito com os dados de forma ordenada, é chamada uma função simples para imprimir o conteúdo do arquivo na tela.

## 4. Aho-Corasick e Similaridade

O algoritmo de Aho-Corasick é um algoritmo de busca de String em um dicionário de palavras, que retorna todas as entradas iguais a String em uma única passada pela árvore inteira, o algoritmo geralmente é implementado utilizando uma máquina de estados finitos.

Trie for arr[] = {he, she, his, hers}

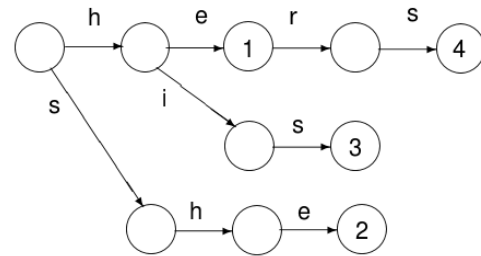
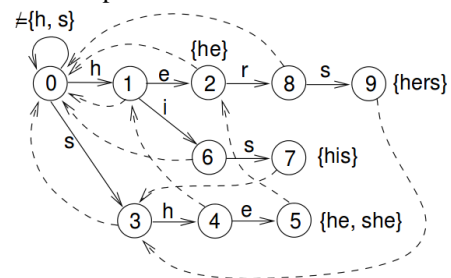


Figura 4 - Trie preenchida

Após a trie ser preenchida, o autômato gerado vai ter transições falhas que voltam de volta à raiz, e transições da trie, que são as palavras inseridas.



Dashed arrows are failed transactions.

Normal arrows are Goto (or Trie) transactions

Figura 5 - Autômato preenchido

As transições falhas ligam buscas falhas e os nós em outras ramificações que compartilham o maior sufixo comum entre eles.

### 4.1. Similaridade

O conceito de similaridade utilizando trigrams depende em dividir a entrada em uma String de 3 caracteres, sejam eles espaços, pontos ou letras, nada é ignorado, e caso a String não tenha 3 caracteres, é adicionado espaços em branco no começo dela.

Como por exemplo a entrada "Chateau blanc" é dividida em 14 trigrams:

```
[ b ] [ c ] [ bl ] [ ch ] [ anc ] [ ate ] [ au ] [ bla ] [ cha ] [ eau ]
[ hat ] [ lan ] [ nc ] [ tea ].
```

E uma segunda entrada "Chateau Cheval Blanc" que contém 19 trigrams:

[ b] [ c] [ bl] [ ch] [anc] [ate] [au ] [bla] [cha] [che]  
[eau] [evl] [hat] [hev] [la ] [lan] [nc ] [tea] [vla].

Depois é pego o número de trigrams iguais e dividido pelo número de trigrams da maior entrada, para assim pegar a similaridade entre eles. Sendo assim, nossos trigrams tem 14 similaridades entre eles, logo,  $14/19 = 73,68\%$

## 5. Utilização

Nossa aplicação pode ser invocada utilizando o seguinte comando na linha de terminal “java -jar APS-ED2.jar teste.txt”

Para essa simples explicação, utilizaremos a mesma entrada descrita no Item 3. Após inserir todas as palavras, a nossa árvore fica assim

```
|
  |-CA
    |--CHORRO CACHORRO [1]
    |--P
      ---ETA CAPETA [2, 3]
      ---ITÃO CAPITÃO [3]
  |-UVA UVA [2]
```

E a nossa saída em output.txt será a seguinte:

```
CACHORRO [1]
CAPETA [2, 3]
CAPITÃO [3]
UVA [2]
```